Process Control using Cintpos Tasks and Coroutines

by

Martin Richards

mr@cl.cam.ac.uk

http://www.cl.cam.ac.uk/users/mr10/

Computer Laboratory
University of Cambridge

Revision date: Mon Dec 21 14:17:19 GMT 2015

This document is still a DRAFT and so feel free to scribble all over it pointing out typos, errors, etc and any comments and advice you may have.

Thank you.

Abstract

This document describes in detail the BCPL implementation of a benchmark program that aims to demonstrate that Cintpos tasks and BCPL style coroutines can be used effectively to implement process control style applications. The benchmark will be translated into other languages such as C and java for comparison.

Keywords

Benchmark, Process Control, BCPL, Cintpos, Coroutines

Contents

Chapter 1

Introduction

This is one of the benchmark programs freely available via my Tcobench distribution (`tcobench.tgz` or `tcobench.zip`) from my home page (`www.cl.cam.ac.uk/users/mr10`). The program is also available as `cintpos/com/tcobench.b` in the Cintpos distribution. The BCPL and Cintpos manual is available as `bcplman.pdf`, also available from my home page.

This benchmark is designed to demonstrate that Cintpos tasks using coroutines can be used effectively to implement process control style applications. It is a complete rewrite of an earlier version written in 2004.

It is designed to be language independent and will be translated into other languages, such as C or Java, for comparison. One such translation is in `java-colib/Tcobench.java` in this Tcobench distribution, but this is still in the early stages of development.

The program creates several Cintpos tasks (equivalent to threads) that run under the control of the Cintpos scheduler. The tasks share the same address space but have their own run time stacks and global vectors. The program performs a long sequence of operations involving task to task and coroutine to coroutine communication and occasional real time delays.

There are several write client tasks that send data to write server tasks. The server tasks have several worker coroutines to process these these requests. The worker coroutines pass the data on to multiplexor tasks that each control of a number of channels used to pass the data to the reading half of the benchmark program. Each channel has a buffer holding data waiting to be transferred. When the buffer is full, a write request to that channel fails and the multiplexor reports this to its server which, in turn, reports it to the write client. After a short delay, the write client tries sending data via a typically different server-multiplexor-channel route.

The reading half of the benchmark contains several read client tasks that send requests to read server tasks. Each read server has several worker coroutines to process these requests, and they send requests to specified multiplexor tasks to read data from specified channels. If the channel buffer is empty, the request fails and the multiplexor reports this to its server which, in turn, reports it to the read client. After a short delay, the read client tries sending another read request via a typically different server-multiplexor-channel route.

Each read and write client repeatedly creates and performs schedules of work items invoking each possible server-multiplex-channel path. If there are s servers, m multiplexors and c channels, there will be $s \times m \times c$ items in each schedule. One of the items is marked to cause a real time delay after the client has successfully processed the request, another item is marked to cause a similar delay in a worker coroutine of its server after successfully processing the request, and a third item causes a delay after data has been successfully transferred to or from the specified channel buffer. The items in the schedules are processed by clients in random order.

The clients are synchronised so that none can create and process its next schedule until all the others have completed their previous schedules.

Each server has worker coroutines to process requests and one logger coroutine. When a server receives a request it passes it to a worker for processing, but there is the constraint that the busiest worker must not process more than 5 requests more than the least busy worker. This constraint is included to demonstrate the use of condition variables and, in particular, the functions `condwait`, `notify` and `notifyAll`. After a worker coroutine has successfully processed a request, it sends two random numbers to the logger coroutine. These numbers are sent separately and so mutex style locking and unlocking is necessary, but since the locking occurs between coroutines within the same task, its implementation is simple and efficient. To check that the locking mechanism works, the logger returns the sum of two numbers back to the originating worker for checking. The communication between worker coroutines and the logger is by means of coroutine based Occum style channels using the functions `cowrite` and `coread`.

The program creates one low priority printer task, and each logger occasionally sends a message to the printer task. It also occasionally delays for a short period. This allows other tasks within the benchmark to gain control, helping to demonstate that the locking mechanism works properly.

Condition variables are also used by worker coroutines when waiting for request packets to process.

Each multiplexor task has one read and one write coroutine per channel to process multiplexor requests. If a write request is received when its channel buffer is full, it is immediately returned to the server with an indication of failure. Otherwise, the data is copied into the channel's buffer and after a possible delay the request packet is returned to the server. If a read request finds that the specified channel buffer is empty, the packet is immediately returned to its server with an indication of failure. The server then returns the packet to is client which after a short delay sends another request normally involving a different server-multiplexor-channel route.

If a read or write request specifies a multiplexor delay, only the appropriate channel coroutine is held up. Requests involving other channels are not affected.

The `stats` task performs various jobs. One is to receive and accumulate data about the frequency of various operations such as calls of `qpkt`, `taskwait`, `callco` and `cowait`. It also performs synchronisation operations involving `sync`, `rddone` and `wrdone` packets. Thirdly it continually measures the CPU utilisation by counting how often it can bounce a packet off the bounce task in each 100 msec period. Since the bounce task has the lowest priority, it can only return a packet when all the other tcobench tasks are suspended.

At the end of the run, the `stats` task outputs a summary of the statistics it has gathered. Typical output is the following.

```
0.000 1> tcobench

Thread and Coroutine Benchmark

loopmax    =    2 (k)
climax     =   20 (n)
srvmax     =   15 (s)
workmax    =   14 (w)
mpxmax     =   10 (m)
chnmax     =   10 (c)
chnbufsize =   35 (b)
delaymsecs =  500 (d)

Requests per schedule = 1500
maxcountdiff          =    5



Start  time:  05-Oct-2015 09:36:00

Finish time:  05-Oct-2015 09:36:35

All clients have finished their work

Number of calls of qpkt:            7077796
Number of calls of taskwait:        7073234
Number of calls of callco:         34771561
Number of calls of cowait:         34771561
Number of calls of resumeco:         172311
Number of calls of condwait(..):     268106
Number of calls of notify(..):       124024
Number of calls of notifyAll(..):      8524
Number of increments:                120000
```

```
   increment had to wait:            27268
Number of calls of lock(..):        120000
   lock had to wait:                 33111
Number of  500 msec delays:            240
Print task counter:                   2400
Calls to logger:                    120000
Send fail count:                      3185
Read fail count:                       839
Bounce task counter:               3276007
Read checksum:                      811018
Write checksum:                     811018
Read count:                          60000
Write count:                         60000

     Approximate CPU utilisation over 339 periods of 100 msecs

  0-10% 10-20% 20-30% 30-40% 40-50% 50-60% 60-70% 70-80% 80-90% 90-100%
   82    204     51      0      1      1      0      0      0      0

Tcobench completed
35.440 1>
```

Note that this output indicates that there were 240 delays each of 500 msecs amounting to a total of 120 seconds. But since the total running time was about 36 seconds, many of these delays must have been done in parallel.

The benchmark program is in the file `tcobench.b` which starts as follows.

```
SECTION "TCOBENCH"

GET "libhdr"

MANIFEST {
  // Packet types
  act_startstats=100  // Controller to stats task
  act_startrdclient   // Controller to client to start it reading
  act_startwrclient   // Controller to client to start it writing
  act_startrdserver   // Controller to server
  act_startwrserver   // Controller to server
  act_startmpx        // Controller to mpx
  act_startbounce     // Controller to bounce task
  act_startprinter    // Controller to printer task
  act_clock           // Stats task to clock device
  act_bounce          // Stats task to bounce task
  act_print           // Server to printer task
  act_read            // A read request from a client or server
  act_write           // A write request from a client or server
  act_sync            // Client to stats task to synchronise with other clients
  act_rddone          // Read client to stats task to synchronise end of run.
  act_wrdone          // Write client to stats task to synchronise end of run.
  act_die             // Controller to most tasks to cause them
                      // to return to DEAD state ready for deletion
  act_calibrate       // Controller to the stats task
  act_run             // Controller to stats task to release
                      // the clients. It is returned when all
                      // clients have completed all their schedules.
  act_addstats        // Send statistics counts to the stats task
  act_prstats         // Output the accumulated statistics

  // Positions of the counters in the statistics vector.
  // These are in the same order as those in the global vector.
  p_qpkt=0
  p_taskwait
  p_callco
  p_resumeco
  p_cowait
  p_condwait
  p_notify
  p_notifyAll
  p_inc
  p_incwait
  p_lock
  p_lockw
  p_delaylong
  p_bounce
  p_print
  p_logger
  p_readfail
  p_sendfail

  p_rdchecksum
  p_wrchecksum
  p_rdcount
  p_wrcount

  statvupb   // Actually one larger than necessary
}
```

The manifest constants with names starting act_ are types (or operators) used in Cintpos packets sent from one task to another, and those constants with names starting p_ are offsets in the vector containing statistics counters. The program continues as follows.

```
STATIC {
  // These static quantites are shared by all tasks in the benchmark,
  // once initialised these variables remain constant.

  rdclientv        = 0   // For read client task ids
  wrclientv        = 0   // For write client task ids
  rdserverv        = 0   // For read server task ids
  wrserverv        = 0   // For write server task ids
  mpxv             = 0   // For multiplexor task ids
  controllertaskid = 0   // To hold the controller task id
  statstaskid      = 0   // To hold the stats task id
  bouncetaskid     = 0   // To hold the bounce task id
  printertaskid    = 0   // To hold the printer task id

  // Default settings
  loopmax
  climax       // Upb of rdclintv and wrclientv
  srvmax       // Upb of rdserverv and wrserverv
  workmax      // Number of work coroutines per server
  mpxmax       // Number of mpx tasks
  chnmax       // Number of buffers per mpx task
  chnbufsize   // The size of each channel buffer
  delaymsecs   // Delay time for read and write delay requests
  maxcountdiff // maximum allowable value of wrkcount-minwrkcount

  requestvupb  // Number of requests per schedule

  tracing
}
```

These static variables are accessible by all tasks in the benchmark and once initialised they remain constant and so can be safely accessed without the need of synchronisation primitives. The program continues with some global declarations as follows.

```
GLOBAL {

  // Each task has its own set of global variables that other tasks
  // cannot access. Note that the counter c_qpkt of how many times
  // this task call qpkt can be incremented efficiently. Such counters
  // are sent to the stats task at the end of the run where they are
  // accumulated and output.
```

```
c_qpkt:ug     // These are the statistics counters and they must
c_taskwait    // be in the same order as the p_ declarations.
c_callco
c_resumeco
c_cowait
c_condwait
c_notify
c_notifyAll
c_inc
c_incwait
c_lock
c_lockw
c_delaylong
c_bounce
c_print
c_logger
c_readfail
c_sendfail
c_rdchecksum
c_wrchecksum
c_rdcount
c_wrcount

modech        // Mode of a client or server, 'R' or 'W'.

pktlist
gomultievent
multi_done    // Used to cause gomultievent to return
              // to single event mode
mainco_ready  // If this is FALSE gomultievent puts new request packets
              // in a queue rather than giving them to the main coroutine.

startuppkt    // The startup packet received by servers and mpx tasks
              // from the controller task. Not returned until the server
              // or mpx task is fully initialised and ready to process
              // requests.
diepkt        // The die packet received by servers and mpx tasks to
              // cause them to return to DEAD state.

condwait      // function to wait on a condition variable
              // eg condwait(@countcondvar)
notify        // wakeup first coroutine waiting on a condition variable
              // eg notify(@pktcondvar)
notifyAll     // wakeup all coroutines waiting on a condition variable
              // eg notifyAll(@countwaitlist)
```

```
lock          // eg lock(@loggerlock)
unlock        // eg unlock(@loggerlock)

startstats
startbounce
startprinter
startclient
startserver
startmpx

// Client task globals
clino         // Number of this client

// Server task globals
serno         // Number of this server
wkq           // Queue of packet waiting to be processed by
              // worker coroutines. Servers place packets in this
              // list when all worker coroutines are busy.
busycount     // Number of server worker coroutines currently
              // processing requests.
wrkcov        // Vector of this server's worker coroutines
countv        // Vector of this server's worker workcounts
              // ie the number of requests successfully processed
              // by each worker.

minwrkcount   // The work count of the this server's least busy worker

countcondvar  // The condition variable for condition:
              //   wrkcount-minwrkcount <= maxcountdiff | diepkt

pktcondvar    // condition variable used by work coroutines waiting
              // for a request packet.

loggerco      // The logger coroutine for this server task
loggerlock    // List of coroutines waiting for the logger

loggerin      // Occam style channel for logger input
loggerout     // Occam style channel for logger output

// Multiplexor task globals
mpxno         // Number of this mpx task
mpxrdcov      // Vector of multiplexor channel read coroutines
mpxwrcov      // Vector of multiplexor channel write coroutines
rdbusyv       // rdbusyv!chnno is TRUE if the read coroutine
```

```
                // for channel chnno is busy.
  wrbusyv       // wrbusyv!chnno is TRUE if the write coroutine
                // for channel chnno is busy.
  rdwkqv        // rdwkqv!chnno is a list of pending mpx read packets
                // for channel chnno.
  wrwkqv        // wrwkqv!chnno is a list of pending mpx write packets
                // for channel chnno.
  bufv          // bufv!chnno is the mpx circular buffer for its channel.
                // These buffers each have climax*srvmax+1 elements.
  bufpv         // bufpv!chnno is the subscript into bufv!chnno where
                // the next write data will be written.
  bufqv         // bufqv!i is the subscript into bufv!i where the next
                // data value will be read from.
                // If bufpv!i=bufqv!i, there is no pending write data
                // within the mpx task for client i.
  mpxbusycount  // Count of the number of channel read and write
                // coroutines that are busy.

  // Language independent random number generator

  seed          // Seed for the machine/language independent
  nextrnd       // random number generator.
  setseed
}
```

These global variables are private to each task and are not normally accessible by other tasks. Again, this means that they can be accessed without the need of synchronisation primitives. The globals whose names start with `c_` are used to hold counters of how often functions such as `qpkt` and `taskwait` are called by the task that owns the globals. Clearly incrementing these counters can be done efficiently. At the end of the run these counters are sent to the `stats` task where they are accumulated with the counters from the other tasks and finally output.

The function `start` is the main function of the benchmark program and is called with an argument of zero when `tcobench` it is being executed by the Command Language Interpreter (CLI). However, it is also the main function of every other task of the benchmark and, as will be seen, such tasks are started by sending them startup packets telling them their role and often provide extra information depending on what the task is meant to do. Startup packets appear as the argument (`pkt`) of `start`.

A tasks in Cintpos is a process that can be pre-empted by an interrupt from a device such as the clock which may cause the task to become suspended and control given to another task. All Cintpos tasks share the same address space, and so, using modern terminology, they would be called threads. Every task has a distinct priority and the Cintpos scheduler ensures that the highest priority task that is not suspended has control. Every tasks has a numerical identifier held in the variable `taskid` in its global vector. Communication between tasks is done by sending a receiving packets using `qpkt` and `taskwait`. A packet is typically of the form:

```
[link,taskid,type,r1,r2,a1,a2,a3,a4,a5,a6]
```

where `link` is used to chain packets together, `taskid` is the identifier of the task or device to which the packet will be sent. The `type` field specifies the purpose of the packet and is often specified using a manifest constant starting with `act_`, such as `act_startprinter` or `act_sync`. When a packet is returned to the task that originally sent it, the `r1` and `r2` fields normally hold result information. The fields `a1` to `a6` are arguments that may be needed by the destination task. Some functions such as `sendpkt` assume that there are always six arguments, but in general any number of arguments are allowed. At the lowest level, a packet is sent to a destination task using `qpkt` as in `qpkt(pkt)` where `pkt` points to the link field of the packet to be sent. This causes the packet to be appended to work queue of the destination task. To simplify returning a packet to its sender, the `taskid` field is automatically replaced by the identifier of the sending task.

A task can extract a packet from its work queue by calling `taskwait()`. If the work queue is empty, the task becomes suspended waiting for a packet to arrive. But note that, if a task is suspended in `taskwait` and a new packet arrives, it will only gain control when all higher priority tasks are suspended.

Chapter 2

The Controller

The benchmark has many tasks. One is called the controller and its task number is that of the CLI executing the `tcobench` command. The controller creates all the other tasks and causes them to run. When the benchmark completes its work, it deletes all the `tcobench` tasks and workspace and returns control the the CLI.

We are now ready to see the definition of `start` which is as follows.

```
LET start(pkt) = VALOF
{ // If pkt=0, this code is being run as the Cintpos command that
  // creates, runs and finally deletes all the tasks of the benchmark.
  // If pkt is non zero it is the startup packet for one of the
  // benchmark tasks.

  initstats()  // Initialise this task's statistics counters.
               // Bear in mind this might be any one of the
               // tcobench tasks.

  UNLESS pkt DO
  { // start is being called from the CLI.
    tcobenchcom()  // Execute the tcobench CLI command.
                   // This will be the controller task.
    RESULTIS 0
  }

  // start is also the main function of every tcobench task,
  // and pkt is the startup packet.
  SWITCHON pkt_type!pkt INTO
  { DEFAULT:
      sawritef("T%z2 Controller:*
               *   System Error: Unexpected packet from %n type %n*n",
                taskid, pkt!pkt_id, pkt!pkt_type)
      abort(999)
      c_qpkt := c_qpkt+1
      qpkt(pkt)
      RESULTIS 0

    CASE act_startstats:    startstats(pkt);   ENDCASE
    CASE act_startbounce:   startbounce(pkt);  ENDCASE
    CASE act_startprinter:  startprinter(pkt); ENDCASE
    CASE act_startrdclient: startclient(pkt);  ENDCASE
    CASE act_startwrclient: startclient(pkt);  ENDCASE
    CASE act_startrdserver: startserver(pkt);  ENDCASE
    CASE act_startwrserver: startserver(pkt);  ENDCASE
    CASE act_startmpx:      startmpx(pkt);     ENDCASE
  }

  RESULTIS 0
}
```

The function `initstats`, defined below, initialises this task's statistics counters held in the global vector. These counters occupy consecutive locations of the global vector so can all be accessed via the pointer `@c_qpkt`, as used in the definition of `sendstats` defined below. At the end of the run that `stats` task is told to output the accumulated statistics by the call `prstats()`, defined below.

```
AND initstats() BE
{ // Initialize this tasks statistics counters
  c_qpkt        := 0
  c_taskwait    := 0
  c_callco      := 0
  c_resumeco    := 0
  c_cowait      := 0
  c_condwait    := 0
  c_notify      := 0
  c_notifyAll   := 0
  c_inc         := 0
  c_incwait     := 0
  c_lock        := 0
  c_lockw       := 0
  c_delaylong   := 0
  c_bounce      := 0
  c_print       := 0
  c_logger      := 0
  c_readfail    := 0
  c_sendfail    := 0
  c_rdchecksum  := 0
  c_wrchecksum  := 0
  c_rdcount     := 0
  c_wrcount     := 0
}

AND sendstats() BE
{ // Send this tasks statistics to the stats task.
  LET sv = @c_qpkt
  sendpkt(notinuse, statstaskid, act_addstats, 0,0, sv)
}

AND prstats() BE
{ // Cause the stats task to print the accumulated statistics.
  sendpkt(notinuse, statstaskid, act_prstats)
}
```

The main program of the controller is called `tcobenchcom` and is defined below. As it is quite long it will be described in sections.

```
AND tcobenchcom() BE
{ // This performs the tcobench CLI command.
  // It is the body of the controller task.

  // Build a segment list containing KLIB, BLIB and this module.
  LET upb,             klib,              blib,    module =
        3, rootnode!rtn_klib, rootnode!rtn_blib, cli_module
  LET segv = @upb
  // segv is used in all the calls of createtask below.

  LET argv = VEC 100

  UNLESS rdargs("-k/N,-n/N,-s/N,-w/N,-m/N,-c/N,-b/N,-d/N,*
              *-t/S,-x/S,-y/S,-z/S",
              argv, 100) DO
  { sawritef("Bad arguments for TCOBENCH*n")
    RETURN
  }

  // The default parameters
  loopmax    :=  2
  climax     := 20
  srvmax     := 15
  workmax    := 14
  mpxmax     := 10
  chnmax     := 10
  chnbufsize :=  0  // Will be set later
```

```
delaymsecs := 500

tracing := argv!8                 // -t/S

IF argv!9 DO                      // -x/S
{ // Alternate setting of the parameters
  loopmax    :=   1
  climax     :=   2
  srvmax     :=   2
  workmax    :=   3
  mpxmax     :=   2
  chnmax     :=   3
}

IF argv!10 DO                     // -y/S
{ // Alternate setting of the parameters
  loopmax    :=   2
  climax     :=   5
  srvmax     :=   3
  workmax    :=   3
  mpxmax     :=   2
  chnmax     :=   3
}

IF argv!11 DO                     // -z/S
{ // Alternate setting of the parameters
  loopmax    :=   3
  climax     :=  10
  srvmax     :=   4
  workmax    :=   7
  mpxmax     :=   3
  chnmax     :=   4
}

// Conditionally override the default or alternate settings
IF argv!0 DO loopmax    := !argv!0 // -k/N
IF argv!1 DO climax     := !argv!1 // -n/N
IF argv!2 DO srvmax     := !argv!2 // -s/N
IF argv!3 DO workmax    := !argv!3 // -w/N
IF argv!4 DO mpxmax     := !argv!4 // -m/N
IF argv!5 DO chnmax     := !argv!5 // -c/N
IF argv!6 DO chnbufsize := !argv!6 // -b/N
IF argv!7 DO delaymsecs := !argv!7 // -d/N

requestvupb := srvmax*mpxmax*chnmax
```

```
IF requestvupb<3 DO
{ sawritef("The request schedule must have at least three elements*n")
  abort(999)
}

UNLESS chnbufsize DO
{ // Set the channel buffer size to about one tenth of the
  // number of values sent to each channel on each iteration.
  chnbufsize := (climax*srvmax)/10 + 5
}

sawritef("*nThread and Coroutine Benchmark*n*n")
sawritef("loopmax    = %i4 (k)*n", loopmax)
sawritef("climax     = %i4 (n)*n", climax)
sawritef("srvmax     = %i4 (s)*n", srvmax)
sawritef("workmax    = %i4 (w)*n", workmax)
sawritef("mpxmax     = %i4 (m)*n", mpxmax)
sawritef("chnmax     = %i4 (c)*n", chnmax)
sawritef("chnbufsize = %i4 (b)*n", chnbufsize)
sawritef("delaymsecs = %i4 (d)*n", delaymsecs)
sawrch('*n')
sawritef("Requests per schedule = %i4*n",   requestvupb)

sawritef("maxcountdiff          = %i4*n",   maxcountdiff)

sawrch('*n')
```

This function starts by creating a segment list of code modules composed of four consecutive local variables `upb`, `klib`, `blib` and `module` and setting `segv` to point to it. Note that `module` points to the loaded `tcobench` program itself. Every `tcobench` task shares the same segment list. The function goes on to read the command arguments using `rdargs` allowing the user to specify the benchmark parameters. There is a default set of parameters, but alternative setting are available using the arguments `-x`, `-y` and `-z`. These can be further modified by the arguments `-k`, `-n`, `-s`, `-w`, `-m`, `-c`, `-b` and `-d`. The argument `-t` sets `tracing` to `TRUE` causing the program to generate a detailed trace of the actions it performs. After setting the parameters they are output using the standalone functions `sawritef` and `sawrch`.

The program contines as follows.

```
rdclientv := getvec(climax)
wrclientv := getvec(climax)
rdserverv := getvec(srvmax)
wrserverv := getvec(srvmax)
```

```
mpxv      := getvec(mpxmax)

IF rdclientv FOR i = 0 TO climax DO rdclientv!i := 0
IF wrclientv FOR i = 0 TO climax DO wrclientv!i := 0
IF rdserverv FOR i = 0 TO srvmax DO rdserverv!i := 0
IF wrserverv FOR i = 0 TO srvmax DO wrserverv!i := 0
IF mpxv FOR i = 0 TO mpxmax DO mpxv!i := 0

bouncetaskid     := 0
printertaskid    := 0
statstaskid      := 0

// This test is delayed until it is safe to jump to fin.
UNLESS rdclientv & wrclientv &
       rdserverv & wrserverv &
       mpxv DO
{ sawritef("More memory needed*n")
  GOTO fin
}

// The main cli task is the controller task for the benchmark
controllertaskid := taskid

// We now create all the tcobench tasks.
c_taskwait := c_taskwait+1
statstaskid := createtask(segv, 1000, 9000)   // The highest priority
UNLESS statstaskid DO
{ sawritef("Unable to create the stats task*n")
  abort(999)
}
IF tracing DO sawritef("T%z2 Stats:   Task created*n", taskid)

c_taskwait := c_taskwait+1
bouncetaskid := createtask(segv, 1000, 10)   // Very low priority
UNLESS bouncetaskid DO
{ sawritef("Unable to create the bounce task*n")
  abort(999)
}
IF tracing DO sawritef("T%z2 Bounce:  Task created*n", taskid)

c_taskwait := c_taskwait+1
printertaskid := createtask(segv, 1000, 11) // Priority just greater than
UNLESS printertaskid DO                     // that of the bounce task
{ sawritef("Unable to create the printer task*n")
  abort(999)
```

```
}
IF tracing DO sawritef("T%z2 Printer: Task created*n", taskid)

FOR clino = 1 TO climax DO
{ LET id = createtask(segv, 1000, 4000+clino)
  c_taskwait := c_taskwait+1
  UNLESS id DO
  { sawritef("Unable to create RC%z2*n", clino)
    abort(999)
  }
  rdclientv!clino := id
  IF tracing DO sawritef("T%z2 RC%z2:    Task created*n", taskid, clino)

  id := createtask(segv, 1000, 5000+clino)
  c_taskwait := c_taskwait+1
  UNLESS id DO
  { sawritef("Unable to create WC%z2*n", clino)
    abort(999)
  }
  wrclientv!clino := id
  IF tracing DO sawritef("T%z2 WC%z2:    Task created*n", taskid, clino)
}

FOR serno = 1 TO srvmax DO
{ LET id = createtask(segv, 1000, 6000+serno) // Priorities higher than the
                                              // read and write clients
  c_taskwait := c_taskwait+1
  UNLESS id DO
  { sawritef("Unable to create RS%z2*n", serno)
    abort(999)
  }
  rdserverv!serno := id
  IF tracing DO sawritef("T%z2 RS%z2:    Task created*n", taskid, serno)
}

FOR serno = 1 TO srvmax DO
{ LET id = createtask(segv, 1000, 7000+serno) // Priority higher than
                                              // read servers
  c_taskwait := c_taskwait+1
  UNLESS id DO
  { sawritef("Unable to create WS%z2*n", serno)
    abort(999)
  }
  wrserverv!serno := id
  IF tracing DO sawritef("T%z2 WS%x2:    Task created*n", taskid, serno)
```

```
}

FOR mpxno = 1 TO mpxmax DO
{ LET id = createtask(segv, 1000, 8000+mpxno) // Even higher priority
  c_taskwait := c_taskwait+1
  UNLESS id DO
  { sawritef("Unable to create M%z2*n", mpxno)
    abort(999)
  }
  mpxv!mpxno := id
  IF tracing DO
    sawritef("T%z2 M%z2:     Task created*n", taskid, mpxno)
}

// All tcobench tasks have been created but left in DEAD state

sawrch('*n')
```

Vectors are allocated to hold the task identifiers of all the read and write client tasks, the read and write servers and the multiplexor tasks. The tasks are then created using `createtask` giving them all the same segment list and a runtime stack size of 1000 words, but they are all given distinct priorities. The bounce, printer and stats tasks have task numbers held in `bouncetaskid`, `printertaskid` and `statstaskid`. At this stage all the `tcobench` tasks except the controller have been created but left in DEAD state, waiting to be started.

The program goes on as follows.

```
// Send startup packets to all the tcobench tasks.

sendpkt(notinuse, statstaskid,   act_startstats)
sendpkt(notinuse, bouncetaskid,  act_startbounce)
sendpkt(notinuse, printertaskid, act_startprinter)

// Send startup packets to all the multiplexor tasks
FOR mpxno = 1 TO mpxmax DO
  sendpkt(notinuse, mpxv!mpxno, act_startmpx, 0, 0, mpxno)

// Send startup packets to all the read server tasks
FOR serno = 1 TO srvmax DO
  sendpkt(notinuse, rdserverv!serno, act_startrdserver, 0, 0, serno)

// Send startup packets to all the write server tasks
FOR serno = 1 TO srvmax DO
  sendpkt(notinuse, wrserverv!serno, act_startwrserver, 0, 0, serno)
```

```
// Send startup packets to all the read client tasks
FOR clino = 1 TO climax DO
  sendpkt(notinuse, rdclientv!clino, act_startrdclient, 0, 0, clino)

// Send startup packets to all the write client tasks
FOR clino = 1 TO climax DO
  sendpkt(notinuse, wrclientv!clino, act_startwrclient, 0, 0, clino)
```

At this stage all the tcobench task have received their startup packets which have told them their roles, but the main work of the benchmark has not yet started since all the clients are waiting for permission to start their schedules of work. The servers and multiplexors will be waiting for their first request packets.

The program continues as follows.

```
// Perform calibration before the clients are started
IF tracing DO
  sawritef("T%z2 Controller:*
           * Sending a calibrate packet to the stats task*n",
           taskid)
sendpkt(notinuse, statstaskid, act_calibrate)

// Finally the controller sends a run packet to the stats task to
// release all the clients. This packet is only returned to the
// controller when all clients have finished all their schedules.
// It is important that calibration is done before sending the run
// packet to the stats task.

IF tracing DO
  sawritef("T%z2 Controller:*
           * Sending run packet to the stats task*n",
           taskid)

wrtime("*nStart  time: ")
sendpkt(notinuse, statstaskid, act_run)
wrtime("*nFinish time: ")

sawritef("*nAll clients have finished their work*n")

// Send die packets to all clients, servers and multiplexor tasks,
// and the bounce and printer tasks.

// Send die packets to all the write client tasks
FOR clino = 1 TO climax DO
  sendpkt(notinuse, wrclientv!clino, act_die)
```

```
// Send die packets to all the read client tasks
FOR clino = 1 TO climax DO
  sendpkt(notinuse, rdclientv!clino, act_die)

// Send die packets to all the read server tasks
FOR serno = 1 TO srvmax DO
  sendpkt(notinuse, rdserverv!serno, act_die)

// Send die packets to all the write server tasks
FOR serno = 1 TO srvmax DO
  sendpkt(notinuse, wrserverv!serno, act_die)

// Send startup packets to all the mpx tasks
FOR mpxno = 1 TO mpxmax DO
  sendpkt(notinuse, mpxv!mpxno, act_die)

sendpkt(notinuse, printertaskid, act_die)
sendpkt(notinuse, bouncetaskid,  act_die)

// All tcobench tasks except the stats task should be returning to
// DEAD state.

IF tracing DO
  sawritef("T%z2 Controller: *
           *All tcobench tasks except the stats task *
           *should be returning to DEAD state*n",
           taskid)
```

This code sends a `run` packet to the `stats` task to release all the clients, but this only happens when all the read and write clients have sent `sync` packets to the `stats` task. The `stats` task returns the `run` packet to the controller when all the read and write clients have finished all their schedules of work. This causes the controller to send `die` packets to all the `tcobench` tasks except the `stats` task, causing them to send their own statistics counters to the `stats` task before returning to DEAD status, ready for deletion.

The program continues as follows.

```
// Arrive here when the benchmark has completed all its work.

// Send the controller's statistics counters to the stats task.
sendstats()

// Cause the accumulated statistics counters to be output.
prstats()
```

```
  // Cause the stats task to return to DEAD state.
  sendpkt(notinuse, statstaskid, act_die)

fin:
  // Delete all the tasks created by the controller.

  IF tracing DO
    sawritef("T%z2: Attempting to delete all tcobench tasks*n", taskid)

  IF mpxv FOR mpxno = 1 TO mpxmax WHILE mpxv!mpxno DO
  { TEST deletetask(mpxv!mpxno)
    THEN { IF tracing DO
             sawritef("T%z2 Controller: M%z2 deleted*n", taskid, mpxno)
           mpxv!mpxno := 0
         }
    ELSE { IF tracing DO
             sawritef("T%z2 Controller: M%z2 NOT deleted*n", taskid, mpxno)
           delay(100)
         }
  }

  IF rdserverv FOR serno = 1 TO srvmax WHILE rdserverv!serno DO
  { TEST deletetask(rdserverv!serno)
    THEN { IF tracing DO
             sawritef("T%z2 Controller: RS%z2 deleted*n", taskid, serno)
           rdserverv!serno := 0
         }
    ELSE { IF tracing DO
             sawritef("T%z2 Controller: RS%z2 NOT deleted*n", taskid, serno)
           delay(100)
         }
  }

  IF wrserverv FOR serno = 1 TO srvmax WHILE wrserverv!serno DO
  { TEST deletetask(wrserverv!serno)
    THEN { IF tracing DO
             sawritef("T%z2 Controller: WS%z2 deleted*n", taskid, serno)
           wrserverv!serno := 0
         }
    ELSE { IF tracing DO
             sawritef("T%z2 Controller: WS%z2 NOT deleted*n", taskid, serno)
           delay(100)
         }
  }
```

```
IF rdclientv FOR clino = 1 TO climax WHILE rdclientv!clino DO
{ TEST deletetask(rdclientv!clino)
  THEN { IF tracing DO
           sawritef("T%z2 Controller: RC%z2 deleted*n", taskid, clino)
         rdclientv!clino := 0
       }
  ELSE { IF tracing DO
           sawritef("T%z2 Controller: RC%z2 NOT deleted*n", taskid, clino)
         delay(100)
       }
}

IF wrclientv FOR clino = 1 TO climax WHILE wrclientv!clino DO
{ TEST deletetask(wrclientv!clino)
  THEN { IF tracing DO
           sawritef("T%z2 Controller: WC%z2 deleted*n", taskid, clino)
         wrclientv!clino := 0
       }
  ELSE { IF tracing DO
           sawritef("T%z2 Controller: WC%z2 NOT deleted*n", taskid, clino)
         delay(100)
       }
}

WHILE bouncetaskid DO
{ TEST deletetask(bouncetaskid)
  THEN { IF tracing DO
           sawritef("T%z2 Controller: bounce deleted*n", taskid)
         bouncetaskid := 0
       }
  ELSE { IF tracing DO
           sawritef("T%z2 Controller: bounce NOT deleted*n", taskid)
         delay(100)
       }
}

WHILE printertaskid DO
{ TEST deletetask(printertaskid)
  THEN { IF tracing DO
           sawritef("T%z2 Controller: printer deleted*n", taskid)
         printertaskid := 0
       }
  ELSE { IF tracing DO
           sawritef("T%z2 Controller: printer NOT deleted*n", taskid)
         delay(100)
```

```
        }
  }

  WHILE statstaskid DO
  { TEST deletetask(statstaskid)
    THEN { IF tracing DO
             sawritef("T%z2 Controller: stats task deleted*n", taskid)
           statstaskid := 0
         }
    ELSE { IF tracing DO
             sawritef("T%z2 Controller: stats task NOT deleted*n", taskid)
           delay(100)
         }
  }

  // All tcobench tasks have now been deleted.

  IF rdclientv DO freevec(rdclientv)
  IF wrclientv DO freevec(wrclientv)
  IF rdserverv DO freevec(rdserverv)
  IF wrserverv DO freevec(wrserverv)
  IF mpxv      DO freevec(mpxv)

  // The test is now complete
  sawritef("*nTcobench completed*n")
} // End of tcobenchcom
```

This code causes the accumulated statistcs counters to be output before finally deleting all the tcobench tasks and workspace.

```
AND wrtime(mess) BE
{ LET v = VEC 14 // Write mess followed by the current date and time.
  datstring(v)
  sawritef("%s %s %s*n", mess, v, v+5)
}
```

This function is used to output the time of day at the start and end of the run.

Chapter 3

The stats Task

The main function of the stats task starts as follows.

```
AND startstats(pkt) BE
{ LET synclist,   synclistlen = 0, 0
  LET donelist,   donelistlen = 0, 0
  LET calibratepkt = 0
  LET runpkt, alldone = 0, FALSE
  LET bounces    = 0   // Number of bounces, zeroed at the start of each period
  LET bouncesmax = 0   // The maximum number of bounces in one clock period
  LET bouncing = FALSE // TRUE if the bounce pkt has been sent to the bounce task.
  LET clocking = FALSE // TRUE if the clock pkt has been sent to the clock
  LET clockpkt  = TABLE notinuse,
                        -1,              // The clock device number
                        act_clock, 0, 0,
                        100              // Return after 100 msec

  LET bouncepkt = TABLE notinuse,  ?, act_bounce, 0, 0
  LET utilisationv = VEC 9
  LET cycle = 1

  diepkt := 0

  bouncepkt!pkt_id := bouncetaskid
  FOR i = 0 TO 9 DO utilisationv!i := 0

  set_process_name("Stats")

  IF tracing DO
    sawritef("T%z2 Stats:   Task ready*n", taskid)

  c_qpkt := c_qpkt+1
  qpkt(pkt) // Return the startup packet
```

The stats task starts by declaring several variables. These are as follows. The variable synclist is used to hold the list of sync packets sent by the read and write clients asking for permission to create and process their next schedules. The length of the list is held in synclistlen. When this length is climax*2, all the clients are ready create and process their next schedules, and the stats task releases them by returning all the sync packets to their clients. Similarly, donelist holds the list of rddone and wrdone packets that clients send when they have finished processing their final schedules. When the length of this list is climax*2, alldone is set to TRUE indicating that it is time to close down the benchmark.

The calibratepkt variable is set when the calibrate packet is received. This indicates that the stats task should calibrate the maximum bounce rate as soon as it is safe to do so. That is when synclistlen is equal to climax*2. Once calibration is completed, the calibrate packet is returned to the controller which then sends a run packet to cause the stats task to let the clients start their first schedules. The run packet is held in runpkt, ready to be returned to the controller when alldone becomes TRUE.

The variable bounces holds the count of how many times the bounce packet has been returned from the bounce task during each 100 msec time period, and bouncesmax holds the maximum number of bounces in any time period. The variable bouncing is only TRUE when the bounce packet has been sent to the bounce task but not yet returned. Similarly, clocking is only TRUE while it is with the clock. The variables clockpkt and bouncepkt hold the clock and bounce packets.

The vector utilisationv is used to hold the CPU utilisation histogram that is computed during the run of the benchmark.

Finally the startup packet is returned to the controller, before entering the main loop of the stats task which deals with packets of the following types: calibrate, run, sync, rddone, wrdone, addstats, prstats and die. The loop starts as follows.

```
{ // Start of stats task event loop.

  c_taskwait := c_taskwait+1
  pkt := taskwait()

  SWITCHON pkt!pkt_type INTO
  { DEFAULT:
      sawritef("T%z2 Stats: Unexpected packet from %n type %n*n",
                taskid, pkt!pkt_id, pkt!pkt_type)
      c_qpkt := c_qpkt + 1
      qpkt(pkt) // Return this unexpected packet
      ENDCASE
```

```
CASE act_calibrate:
  // Calibrate the maximum bounce rate.
  calibratepkt := pkt
  UNLESS synclistlen = climax+climax LOOP
  // Only perform calibration when all read and write clients
  // are ready to run their first schedules.

calibrate:
  // We may arrive here from CASE act_sync:
  IF tracing DO
    sawritef("*nT%z2 Stats: Calibrating the bounce counter*n", taskid)

  { LET cycles = 10 // The number of 100msec time periods to use
                    // when calibrating tha bounce rate.
    bounces := 0

    clocking := TRUE      // Start clocking
    c_qpkt := c_qpkt+1
    qpkt(clockpkt)

    bouncing := TRUE      // Start bouncing
    c_qpkt := c_qpkt+1
    qpkt(bouncepkt) // This returns as soon as the bounce task can return it.


    WHILE clocking | bouncing DO
    { LET pkt = ?
      c_taskwait := c_taskwait+1
      //Wait for a packet from either the clock or bounce task.
      pkt := taskwait()

      SWITCHON pkt!pkt_type INTO
      { DEFAULT:
         sawritef("T%z2 Stats:*
                  * System Error: Unexpected packet from %n type %n*n",
                  taskid, pkt!pkt_id, pkt!pkt_type)
         abort(999)

        CASE act_clock:
          clocking := FALSE
          IF bouncesmax=0 DO bounces := 1 // Ignore first time period
          IF bouncesmax < bounces DO bouncesmax := bounces
          bounces := 0
          cycles := cycles - 1
          IF cycles DO
          { // Start another 100 msecs time period.
            c_qpkt := c_qpkt+1
            qpkt(clockpkt)
            clocking := TRUE
          }
          LOOP

       CASE act_bounce:
          bounces := bounces+1
          bouncing := FALSE
          UNLESS cycles BREAK // Both clk and bounce pkts are back
          c_qpkt := c_qpkt+1
          qpkt(bouncepkt)
          bouncing := TRUE
          LOOP
      }
    }
  }

  // Calibration is complete and both the clock and bounce
  // packets have been returned.

  IF tracing DO
    sawritef("*nT%z2 Stats: bouncesmax = %n*n*n", taskid, bouncesmax)

  pkt := calibratepkt
  calibratepkt := 0
  c_qpkt := c_qpkt+1
  qpkt(pkt) // Give the calibrate packet back to the controller.
  LOOP
```

This loop waits for the next packet by calling `taskwait` and the switches on its type. If the packet had type `calibrate` it is a request from the controller to calibrate the maximum rate a which packets can be bounced off the bounce task when no other tasks as active. The code to do this is at label `CASE act_calibrate:` where the packet is placed in the variable `calibratepkt`. Notice that there is a check to see whether all the client tasks have requested permission to create and process their first schedules. If this is not the case, calibration cannot yet take place and will be activated by a jump to label `calibrate` when the final `sync` packet arrives. During calibration all the other tasks will be suspended.

Calibration runs for 10 time periods of 100 msecs discovering the maximum number of bounces in any one of these periods. This is placed in `bouncesmax`. The variable `cycles` which is initially 10 is decremented every time the `clock` packet is received and when zero the next bounce packet causes the calibration loop to exit. At this point the calibration packet is returned to the controller and `calibratepkt` set to zero.

The program continues as follows.

```
    CASE act_run:
      runpkt := pkt

      // Start the clock
      c_qpkt := c_qpkt+1
      qpkt(clockpkt)
      clocking := TRUE

      // Start bouncing
      c_qpkt := c_qpkt+1
      qpkt(bouncepkt)
      bouncing := TRUE
      // Fall through in CASE act_sync:

    CASE act_sync:
      IF pkt!pkt_type=act_sync DO
      { // Request from a client to start the next schedule.
        // It is returned when all clients have sent sync packets.
        pkt!0 := synclist
        synclist := pkt
        synclistlen := synclistlen+1
        IF tracing DO
          sawritef("T%z2 Stats:*
                   *  Sync received from T%z2*n",
                   taskid, pkt!pkt_id, synclistlen)
      }

      UNLESS synclistlen = climax+climax LOOP

      IF calibratepkt GOTO calibrate

      // Note that controller only sends the run packet after
      // calibration has been completed.

      IF runpkt DO
      { // Run packet and all sync packets received
        // so return the sync packets to their clients.
        LET p = synclist
        IF tracing DO
          sawritef("T%z2 Stats:  %i2 Releasing all clients*n",
                   taskid, cycle)
        cycle := cycle + 1
        synclist, synclistlen := 0, 0
        WHILE p DO
        { pkt := p
          p := p!0
          pkt!0 := notinuse
          c_qpkt := c_qpkt+1
          qpkt(pkt)
        }
      }
      LOOP
```

Since the code for run and sync packets have much in common, they appear together. When the run packet arrives it is placed in runpkt and if synclistlen equals climax*2 all the clients can be release by returning their sync packets to them. But if some clients have not yet asked permission to create and run their next schedule, the other clients must remain suspended until the last sync packet has been received. As stated earlier the controller does not send the run packet until the calibrate packet has been returned. If the last sync packet is received when calibratepkt is non zero execution jumps to the calibration code.

The next two cases deal with the clock and bounce packets. Their code is as follows.

```
  CASE act_clock:
    // Keep clocking until alldone=TRUE
    UNLESS alldone DO
    { LET u = ?

      c_qpkt := c_qpkt+1
      qpkt(pkt)  // Immediately send the packet back to the clock

      // Even after calibration bouncesmax might still increase.
      IF bouncesmax < bounces DO bouncesmax := bounces

      u := 999 * (bouncesmax-bounces) / bouncesmax / 100
      // 0 <= u <= 9
      IF tracing DO
        sawritef("T%z2 Stats: clock packet received, bounces=%n u=%n*n",
                 taskid, bounces, u)

      UNLESS 0<=u<=9 DO
      { sawritef("T%z2 Controller:*
                 * System error -- Utilisation u=%n out of range*n",
                 taskid, u)
        abort(999)
      }
      utilisationv!u := utilisationv!u + 1
      //IF tracing DO
      //  sawritef("T%z2 Controller:*
      //           * Utilisation = %i3  bounces = %i5 bouncesmax = %i5*n",
      //            taskid, u, bounces, bouncesmax)
      bounces := 0
      LOOP
    }

    // alldone=TRUE so we have reached the end of the final
    // time period.
    clocking := FALSE
    // Do not send the packet to the clock device.
    IF tracing DO
      sawritef("T%z2 Stats:*
               * clocking set to FALSE*n",
                taskid)
    LOOP

  CASE act_bounce:
    bounces := bounces+1

    IF clocking DO
    { // Still in a time period so bounce again
      c_qpkt := c_qpkt+1
      qpkt(pkt)
      LOOP
    }

    // clocking=FALSE so the end of the final time period
    // has been reached. This only happens when a clock packet
    // is received when alldone=TRUE.

    // We have thus just received the first bounce packet
    // after the end of the final time period. It is therefore
    // time to return the run packet to the controller.

    bouncing := FALSE
    IF tracing DO
    { sawritef("T%z2 Stats:*
               *   bouncing set to FALSE*n",
               taskid)
    }
    c_qpkt := c_qpkt+1
    qpkt(runpkt)

    // We still need to process addstats packets from dying tasks,
    // the prstats packet from the controller and the final die
    // packet which will cause the stats task to return to DEAD
    // state.
    LOOP
```

The `clock` packet has an argument of 100 causing it to stay with the clock device for 100 msecs before returning. So receiving the `clock` packet indicates that a 100 msec time period has just completed. At this moment the value in `bounces` gives an indication of the CPU utilisation in the period just finished. This is converted to a percentage value and the appropriate entry in `utilisationv` incremented. When `alldone` becomes `TRUE` the `stats` task stops sending the `clock` packet to the clock device and sets `clocking` to `FALSE`. In this state the next `bounce` packet to arrive leaves both `clocking` and `bouncing` set to `FALSE` and so no more `bounce` or `clock` packets will be received and it is time to return the `run` packet to the controller. Note that the `stats` task must continue its event loop waiting of `addstats` packets followed by a `prstats` packet and finally the `die` packet.

The next two cases deal with `rddone` and `wrdone` packets received from clients. These indicate that a client has completed processing its last schedule and so has finished work. The code is as follows.

```
  CASE act_rddone:
  CASE act_wrdone:
  { // Client has finished all its schedules.

    TEST pkt!pkt_type=act_rddone
    THEN { IF tracing DO
             sawritef("T%z2 Stats:*
                      *   rddone packet received from T%z2*n",
                      taskid, pkt!pkt_id)
         }
    ELSE { IF tracing DO
             sawritef("T%z2 Stats:*
                      *   wrdone packet received from T%z2*n",
                      taskid, pkt!pkt_id)
         }

    pkt!0 := donelist
    donelist := pkt
    donelistlen := donelistlen+1
    IF donelistlen = climax+climax DO
    { // All the rddone and wrdone packets have been received
      // so return them to their clients.
      LET p = donelist
      donelist, donelistlen := 0, 0
      WHILE p DO
      { pkt := p
        p := p!0
        pkt!0 := notinuse
        c_qpkt := c_qpkt+1
        qpkt(pkt)
      }

      alldone := TRUE
      IF tracing DO
        sawritef("T%z2 Stats:*
                 *   alldone set to TRUE*n",
                 taskid)
    }
    LOOP
  }
```

As can be seen it stores the `rddone` and `wrdone` packets in the list `donelist`, maintaining its length in `donelistlen`. When this length reaches `climax*2` all clients have completed all their schedules and so `alldone` is set to `TRUE`. However, the `run` packet in `runpkt` in not returned to the controller until the first bounce packet is received after the final time period. This ensures that both `clocking` and `bouncing` are both `FALSE`.

The next two cases deal with `addstats` and `prstats` packets. A task's statistics counters are held in consecutive global locations starting at `c_qpkt` and are passed to the `stats` task as a pointer to `c_qpkt` in the `a1` field of the `addstats` packet. The code for these two cases are as follows and require no further explanation.

```
  CASE act_addstats:
  { // Add the stats from another task.
    LET sv = pkt!pkt_a1
    c_qpkt := c_qpkt + 1 // Because of qpkt below

    // Add a task's statistics counters to the stats task's
    // own counters.
    c_qpkt      := sv!p_qpkt      + c_qpkt
    c_taskwait  := sv!p_taskwait  + c_taskwait
    c_callco    := sv!p_callco    + c_callco
    c_resumeco  := sv!p_resumeco  + c_resumeco
    c_cowait    := sv!p_cowait    + c_cowait
    c_condwait  := sv!p_condwait  + c_condwait
    c_notify    := sv!p_notify    + c_notify
    c_notifyAll := sv!p_notifyAll + c_notifyAll
    c_inc       := sv!p_inc       + c_inc
    c_incwait   := sv!p_incwait   + c_incwait
    c_lock      := sv!p_lock      + c_lock
    c_lockw     := sv!p_lockw     + c_lockw
    c_delaylong := sv!p_delaylong + c_delaylong
    c_bounce    := sv!p_bounce    + c_bounce
    c_print     := sv!p_print     + c_print
    c_logger    := sv!p_logger    + c_logger
    c_readfail  := sv!p_readfail  + c_readfail
    c_sendfail  := sv!p_sendfail  + c_sendfail

    c_rdchecksum := (sv!p_rdchecksum + c_rdchecksum) MOD 1_000_000
    c_wrchecksum := (sv!p_wrchecksum + c_wrchecksum) MOD 1_000_000
    c_rdcount    := sv!p_rdcount     + c_rdcount
    c_wrcount    := sv!p_wrcount     + c_wrcount

    c_qpkt := c_qpkt+1
    qpkt(pkt)    // Return the addstats pkt to its task
    ENDCASE
  }

  CASE act_prstats:
    // Output the statistics counters that have been accumulated in
    // the stats task.

    c_qpkt := c_qpkt + 1 // Because of qpkt below

    sawritef("*n")
    sawritef("Number of calls of qpkt:          %i9*n", c_qpkt)
    sawritef("Number of calls of taskwait:      %i9*n", c_taskwait)
    sawritef("Number of calls of callco:        %i9*n", c_callco)
    sawritef("Number of calls of cowait:        %i9*n", c_cowait)
    sawritef("Number of calls of resumeco:      %i9*n", c_resumeco)
    sawritef("Number of calls of condwait(..):  %i9*n", c_condwait)
    sawritef("Number of calls of notify(..):    %i9*n", c_notify)
    sawritef("Number of calls of notifyAll(..): %i9*n", c_notifyAll)
    sawritef("Number of increments:             %i9*n", c_inc)
    sawritef("   increment had to wait:         %i9*n", c_incwait)
    sawritef("Number of calls of lock(..):      %i9*n", c_lock)
    sawritef("   lock had to wait:              %i9*n", c_lockw)
    sawritef("Number of %i4 msec delays:       %i9*n", delaymsecs, c_delaylong)
```

```
      sawritef("Print task counter:                 %i9*n", c_print)
      sawritef("Calls to logger:                    %i9*n", c_logger)
      sawritef("Send fail count:                    %i9*n", c_sendfail)
      sawritef("Read fail count:                    %i9*n", c_readfail)
      sawritef("Bounce task counter:                %i9*n", c_bounce)

      sawritef("Read checksum:                      %i9*n", c_rdchecksum)
      sawritef("Write checksum:                     %i9*n", c_wrchecksum)
      sawritef("Read count:                         %i9*n", c_rdcount)
      sawritef("Write count:                        %i9*n", c_wrcount)

      sawritef("*n")

    { LET total = 0
      FOR i = 0 TO 9 DO total := total + utilisationv!i

      UNLESS total DO total := 1

      sawritef("     Approximate CPU utilisation over %n periods of 100 msecs*n*n",
               total)
      sawritef("  0-10%% 10-20%% 20-30%% 30-40%% 40-50%% 50-60%%*
               * 60-70%% 70-80%% 80-90%% 90-100%%*n")
      FOR i = 0 TO 9 DO sawritef("%i5  ", utilisationv!i)
      sawritef("*n")
      c_qpkt := c_qpkt+1
      qpkt(pkt)
      ENDCASE
    }
```

The stats task body ends as follows.

```
    CASE act_die:
      IF tracing DO
        sawritef("T%z2 Stats:*
                 * Die packet received*n",
                 taskid)
      diepkt := pkt
      BREAK
  }
} REPEAT

// This point is only reached when when the stats task should
// return to DEAD state, ready for deletion.
```

```
  IF tracing DO
    sawritef("T%z2 Stats:   Returning the die packet to controller*n", taskid)

  c_qpkt := c_qpkt+1
  qpkt(diepkt)

  IF tracing DO
    sawritef("T%z2 Stats:   Returning to DEAD state*n", taskid)

  RETURN // Return the stats task to DEAD state
         // ready for deletion by the controller.
}
```

The controller sends a `die` packet to the `stats` task when all its work is done telling it to return to DEAD state ready for deletion. The `die` packet is the only one that causes an exit from the task's event loop, which otherwise repeatedly waits for packets to process.

Chapter 4

The bounce and printer Tasks

The bounce task is created with the lowest priority of any of the tcobench tasks and so only gain control when all the other tasks are suspended. Its main purpose is to return bounce packets to the stats task allowing it to estimate the CPU utilisation. Its code is as follows.

```
AND startbounce(pkt) BE
{ LET echoco = createco(echofn, 300)
  set_process_name("Bounce")

  IF tracing DO
    sawritef("T%z2 Bounce:  Task ready*n", taskid)

  c_qpkt := c_qpkt+1
  qpkt(pkt) // Return the startup packet

  { c_taskwait := c_taskwait+1
    pkt := taskwait()
    SWITCHON pkt!pkt_type INTO
    { DEFAULT:
        sawritef("T%z2 Bounce: Unexpected packet from %n type %n*n",
                  taskid, pkt!pkt_id, pkt!pkt_type)

      CASE act_bounce:
        c_bounce := c_bounce+1

        // The following loop is to keep the ratio of
        // callcos to qpkts high even when the bounce
        // count is high.
        FOR i = 1 TO 10 DO
        { c_callco := c_callco+1
          callco(echoco, i)
        }
        c_qpkt := c_qpkt+1
        qpkt(pkt) // Return the packet
        LOOP

      CASE act_die:
        IF tracing DO
          sawritef("T%z2 Bounce: die packet received*n", taskid)

        // Delete the echo coroutine
        IF echoco DO { deleteco(echoco); echoco := 0 }

        // Send the bounce statistics to the stats task
        sendstats()

        IF tracing DO
          sawritef("T%z2 Bounce: Returning to DEAD state*n", taskid)

        c_qpkt := c_qpkt+1
        qpkt(pkt) // Return the die packet to the controller.

        RETURN // Return to DEAD state ready for deletion.
    }
  } REPEAT
}
```

The bounce task creates an echo coroutine that it repeated calls to keep the ratio of calls of callco to qpkt roughly the same as if the bounce task was not being used. The body of the echo coroutine is as follows, and as can be seen is trivial.

```
AND echofn(x) BE // The body of echoco
{ c_cowait := c_cowait+1
  x := cowait(x)
} REPEAT
```

The printer task has a priority just greater than the bounce task. It is occasionally sent packets by server tasks logger coroutines. This usually lets other tasks gain control while a logger is processing its request.

```
AND startprinter(pkt) BE
{ LET delaypkt = TABLE notinuse,
                       -1,       // The clock device
                       act_clock,
                       0,0,      // Result fields
```

```
                   10       // Delay for 10 msecs
LET q = 0         // List of pending print packets while delaying.
LET printpkt = 0  // Current print packet, if any.

set_process_name("Printer")

IF tracing DO
  sawritef("T%z2 Printer: Task ready*n", taskid)

c_qpkt := c_qpkt+1
qpkt(pkt) // Return the startup packet

{ // Start of the printer event loop.
  TEST q
  THEN { // Dequeue a pending print packet.
         pkt  := q
         q    := !q
         !pkt := notinuse

         IF tracing DO
           sawritef("T%z2 Printer: Pending packet %n type %n dequeued*n",
                    taskid, pkt, pkt!pkt_type)

       }
  ELSE { // Wait for a print or die packet
         c_taskwait := c_taskwait+1
         pkt := taskwait()
         IF tracing DO
           sawritef("T%z2 Printer: Packet %n type %n received*n",
                    taskid, pkt, pkt!pkt_type)
       }

  SWITCHON pkt!pkt_type INTO
  { DEFAULT:
      sawritef("T%z2 Printer: Unexpected packet from task %n*n",
               taskid, pkt!pkt_id)
      abort(999)

    CASE act_print:  // arg1: modech  arg2: serno
    { LET modech = pkt!pkt_arg1
      LET serno  = pkt!pkt_arg2
      c_print := c_print+1
      IF tracing DO
        sawritef("T%z2 Printer: act_print received from %cS%z2*n",
                 taskid, modech, serno)
        printpkt := pkt // Save the print packet to be returned after
                        // 10 msecs.
        IF tracing DO
          sawritef("T%z2 Printer: sending a delay packet to the clock*n",
                   taskid)

        c_qpkt := c_qpkt+1
        qpkt(delaypkt)

        { // Wait for the delay packet to return, queueing any
          // other packets that arrive in the mean time.
          pkt := taskwait()
          IF pkt=delaypkt BREAK

          // Insert the packet in q.
          !pkt := q
          q := pkt
        } REPEAT

        // The 10 msecs delay has now ended, so return the print
        // packet and process another (possibly pending) request.

        c_qpkt := c_qpkt+1
        qpkt(printpkt) // Return the print packet
        printpkt := 0
        LOOP
      }

      CASE act_die:
        IF tracing DO
          sawritef("T%z2 Printer: die packet received*n", taskid)

        // Send the printer statistics to the stats task
        sendstats()

        c_qpkt := c_qpkt+1
        qpkt(pkt) // Send die packet back to the controller

        IF tracing DO
          sawritef("T%z2 Printer: Returning to DEAD state*n", taskid)

        RETURN  // Return printer task to DEAD state.
    }
  } REPEAT
}
```

Chapter 5

The Read and Write Clients

The read and write client tasks have much in common and so share the same main function. A read client is started by being given a `startrdclient` startup packet and a write client is given a `startwrclient` packet. These clients repeatedly create and perform schedules of requests. Each request specifies a server, a multiplexor and a channel number, and, for write clients, what data is to be sent. Requests also contain a flag indicating whether there should be a real time delay after data has been successfully read from or written to its channel buffer. The flag could be one of the following letters: `c`, `s`, `m`, indicating a delay in a client, a server, a multiplexor, respectively. If the flag letter was `n` then no delay is specified.

The body of read and write clients start as follows.

```
AND startclient(pkt) BE
{ // Body of a client task which runs in single-event mode.
  // pkt is the startup packet or one of the following

  //      act_startrdclient  a1:clino
  // or   act_startwrclient  a1:clino
  // or   act_die

  LET serverv = 0 // Will hold the vector of read or write servers
                  // for this client.
  LET requestv = getvec(requestvupb)
  LET datav    = getvec(requestvupb)

  // Distinquish between read and write clients.
  TEST pkt!pkt_type=act_startrdclient
  THEN modech, serverv := 'R', rdserverv
  ELSE modech, serverv := 'W', wrserverv

  clino := pkt!pkt_arg1 // From the startup packet
```

```
UNLESS requestv & datav DO
{ sawritef("T%z2 %cC%z2: Unable to allocate requestv and datav*n",
            taskid, modech, clino)
  GOTO fin
}

set_process_name("%cC%z2", modech, clino)

setseed(clino+modech*100)

//IF tracing DO
//  sawritef("T%z2 %cC%z2: Started*n", taskid, modech, clino)

c_qpkt := c_qpkt+1
qpkt(pkt)  // Return the start up packet to the controller
```

The argument `pkt` is the startup packet for the client and its type field will be `act_startrdclient` or `act_startwrclient` depending on whether a read or write client is being started. The first argument in field `arg1` of the packet holds the client number which will be in the range 1 to `climax`. For convenience and efficiency the local variable `serverv` is set to point to the vector of read or write servers, as appropriate. The variable `modech` is set to `'R'` or `'W'` depending on whether this task is a read or write client.

The schedule is represented by two vectors `requestv` and `datav`. Both have elements ranging from 1 to `requestvupb`. Request `i` of the schedule has it flag, server number, multiplexor number and channel number packed in `requestv!i`, and `datav!i` holds the data to send if the schedule belongs to a write client. For read clients this field is zero. Clients process the items in their schedules in random order using random numbers generated by the call `nextrnd(requestvupb)`. The call `setseed(clino+modech*100)` ensures that every client starts with a different random number seed.

After all this initialisation the startup packet is returned to the controller allowing it to start another task. The client is now ready to repeatedly create and perform its schedules. The program to do this starts as follows.

```
FOR count = 1 TO loopmax DO
{ // Create the schedule of requests for this client in requestv
  // and datav (for write client data).
  // Each request a specifies different server-multiplexor-channel
  // combination and so requestv will contain servmax*mpxmax*chnmax
  // items. One will be randomly chosen to have a delay in this
  // client, a different one will be randomly chosen to cause
  // a delay in server, and a third different one will delay in
  // a multiplexor.
  LET pos = 0
  LET rqstvupb = requestvupb
  LET clidelayitem = 0  // This will be the subscript of requestv of
                        // the request to delay in this client.
  LET srvdelayitem = 0  // This will be the subscript of requestv of
                        // the request to delay in its server.
  LET mpxdelayitem = 0  // This will be the subscript of requestv of
                        // the request to delay in its multiplexor.

  // This client will not create its next schedule until all other
  // read and write clients are ready to do the same. This is achieved
  // by sending a sync packet to the stats task which is only returned
  // when all client sync packets have been received.

  sendpkt(notinuse, statstaskid, act_sync, 0,0)
```

It will perform the iteration `loopmax` times, but first it must ask permission to create the next schedule. This is done by sending a `sync` packet to the `stats` task. The `sync` packet is only returned when all other clients are similarly ready. The program continues as follows.

```
  // This point is reached when all read and write clients have sent
  // sync packets to the stats task.

  // This client can now start, create and run its next schedule.

  IF tracing DO
  { sawritef("T%z2 %cC%z2:    Starting iteration %n out of %n*n",
              taskid, modech, clino, count, loopmax)
    //abort(1000)
  }

  // Create a list of requests to be processed.

  // Choose three distinct requests to cause real time delays.

  clidelayitem := nextrnd(rqstvupb) // In range 1..requestvupb
  srvdelayitem := nextrnd(rqstvupb) // In range 1..requestvupb
  REPEATWHILE srvdelayitem=clidelayitem
  mpxdelayitem := nextrnd(rqstvupb) // In range 1..requestvupb
  REPEATWHILE mpxdelayitem=clidelayitem |
              mpxdelayitem=srvdelayitem
```

```
FOR serno = 1 TO srvmax DO
  FOR mpxno = 1 TO mpxmax DO
    FOR chnno = 1 TO chnmax DO
    { LET flag = 'n'
      pos := pos+1
      IF pos=clidelayitem DO flag := 'c' // Client delay
      IF pos=srvdelayitem DO flag := 's' // Server delay
      IF pos=mpxdelayitem DO flag := 'm' // Multiplexor delay
      requestv!pos := flag<<24 | serno<<16 | mpxno<<8 | chnno
      datav!pos    := modech='R' -> 0,
                      nextrnd(9999) // Range 1 to 9999
    }

IF tracing DO
{ sawritef("*nT%z2 %cC%z2:    Schedule of work*n",
           taskid, modech, clino)
  FOR i = 1 TO requestvupb DO
  { // Output the schedule as a debugging aid.
    LET req   = requestv!i
    LET flag  = req>>24 & 255
    LET serno = req>>16 & 255
    LET mpxno = req>>8  & 255
    LET chnno = req     & 255
    LET data  = datav!i

    sawritef(" %c%cS%z2M%z2C%z2:%z4",
             flag, modech, serno, mpxno, chnno, data)
    IF i MOD 5 = 0 DO sawritef("*n")
  }
  UNLESS requestvupb MOD 9 = 0 DO sawritef("*n")
  sawritef("*n")
}
```

It chooses three distinct random positions in the schedule for the client, server and multiplexor delay requests. It then creates requests for every server, multiplexor, channel triplet, and creates suitable data values. Note that, although these requests are in a systematic order, they willactually be processed in random order.

It `tracing` is set, the schedule will be output as a debugging aid. The program continues as follows.

```
// Now process the requests in the schedule in random order.
WHILE rqstvupb DO
{ LET i = nextrnd(rqstvupb) // Choose a random request from
  LET req   = requestv!i    // the schedule.
  LET data  = datav!i       // Data for a write client
  LET flag  = req>>24 & 255 // ='c' delay in this client,
                            // ='s' delay in a server
                            // ='m' delay in a multiplexor
                            // ='n' no delay
  LET serno = req>>16 & 255
  LET mpxno = req>>8  & 255
  LET chnno = req     & 255
```

The variable `rqstvupb` was inititially set to `requestvupb`, the number of requests in the schedule, but it is decremented every time a request has been successfully processed. When it reaches zero all the items in the schedule have been successfully processed and so it is time to leave the loop.

The random subscript position of the next request to process is chosen by the call `nextrnd(rqstvupb)`, and its fields placed in local variables. What happens next depends on whether a read or write client is running. The program continues as follows.

```
TEST modech='R'
THEN { // Code for a read client
       IF tracing DO
         sawritef("T%z2 RC%z2:*
                   *    %c%cC%z2S%z2M%z2C%z2:%z4*
                   * Sending read request to server*n",
                   taskid, clino, flag,
                   modech, clino, serno, mpxno, chnno, data)

       data := sendpkt(notinuse, serverv!serno, act_read,
                       0, 0, flag, clino, serno, mpxno, chnno, data)

       IF tracing DO
         sawritef("T%z2 RC%z2:*
                   *    %c%cC%z2S%z2M%z2C%z2:%z4*
                   * Packet returned from server*n",
                   taskid, clino, flag,
                   modech, clino, serno, mpxno, chnno, data)

       IF data=0 DO
       { // Read was not successful since its channel buffer
         // was empty.
         IF tracing DO
           sawritef("T%z2 RC%z2:*
                    *    %c%cC%z2S%z2M%z2C%z2:%z4 read failed*n*n",
                    taskid, clino, flag,
                    modech, clino, serno, mpxno, chnno, data)
```

```
          c_readfail := c_readfail+1

          // Delay for 200 msecs to give other tasks a chance to run,
          // hopefully allowing write clients to put more data in
          // the channels buffers.
          delay(200)

          LOOP // Try sending another random request from the schedule.
        }

        // The read request was successful.

        IF tracing DO
          sawritef("T%z2 RC%z2:*
                   *    %c%cC%z2S%z2M%z2C%z2:%z4 read successful*n",
                   taskid, clino, flag,
                   modech, clino, serno, mpxno, chnno, data)
        c_rdchecksum := (c_rdchecksum + data) MOD 1_000_000
        c_rdcount    := c_rdcount+1
      }
```

This is the code for a read client. as can be seen it uses `sendpkt` to send a packet to its server. The packet has six arguments completely specifying the request. The type field of the packet is `act_read` since we are running as a read client. When this packet returns its result in the `res1` field is placed in `data`.

If `data` was zero, the `read` request had failed because the specified channel buffer was empty, and the client delays for 200 msecs giving write clients a chance to run hopefully putting more data values in the channel buffers. After this delay the read client tries again selecting another random request to process.

If `data` was non zero, it is the data that was successfully extracted from the specified channel buffer. The read checksum is updated appropriately and `c_rdcount` is incremented.

The code for a write client is as follows.

```
    ELSE { // Code for a write client.
        LET rc = ?                          // Return code
        IF tracing DO
          sawritef("T%z2 WC%z2:    %c%cC%z2S%z2M%z2C%z2:%z4*
                   * Sending write request to server*n",
                   taskid, clino, flag,
                   modech, clino, serno, mpxno, chnno, data)
        rc := sendpkt(notinuse, serverv!serno, act_write,
                      0, 0, flag, clino, serno, mpxno, chnno, data)
        IF rc=0 DO
```

```
          { // Write was not successful since its channel buffer
            // was full.
            IF tracing DO
              sawritef("T%z2 WC%z2:*
                       *    %c%cC%z2S%z2M%z2C%z2:%z4 send failed*n*n",
                       taskid, clino, flag,
                       modech, clino, serno, mpxno, chnno, data)
            c_sendfail := c_sendfail+1

            // Delay for 20 msecs to give other tasks a chance to run.
            delay(20)

            LOOP // Try sending another random request from the schedule.
          }

          // The write request was successful in that the data was
          // written by the specified multiplexor in the specified
          // channel buffer. The data may not have been read yet by
          // a read client.

          IF tracing DO
            sawritef("T%z2 WC%z2:*
                     *    %c%cC%z2S%z2M%z2C%z2:%z4 Data sent*n",
                     taskid, clino, flag,
                     modech, clino, serno, mpxno, chnno, data)

          c_wrchecksum := (c_wrchecksum + data) MOD 1_000_000
          c_wrcount := c_wrcount+1
        }
```

Just as for a read client, a packet is sent to its server containing all the information about the request. But this time its is being sent to a write server with a type field of `act_write`. The return code is place in `rc` and, if it is zero, the request failed because the specified channel buffer was full. In which case the write server delays for a short time (20 msecs) before trying again with another random request. If the write request was successful, the write checksum is updated and `c_wrcount` incremented.

Both read and write clients then go on to obey the following.

```
    // The read or write request has been successfully processed.

    // Delay for delaymsecs if a client delay is specified.
    IF flag='c' DO
    { IF tracing DO
        sawritef("T%z2 WC%z2:*
                 *    %c%cC%z2S%z2M%z2C%z2:%z4 Client delay*n",
                  taskid, clino, flag,
                  modech, clino, serno, mpxno, chnno, data)
      c_delaylong := c_delaylong+1
      delay(delaymsecs)
      IF tracing DO
        sawritef("T%z2 WC%z2:*
                 *    %c%cC%z2S%z2M%z2C%z2:%z4 Client delay ended*n",
                  taskid, clino, flag,
                  modech, clino, serno, mpxno, chnno, data)
    }

    // Remove the read or write request from the schedule.
    requestv!i := requestv!rqstvupb
    datav!i    := datav!rqstvupb
    rqstvupb   := rqstvupb-1

    // Process another request of the schedule, if any.
  }
  // Create and perform the next schedule.
}


IF tracing DO
  sawritef("T%z2 %cC%z2:*
           *    All iterations completed*n",
           taskid, modech, clino)
```

If the `flag` field was set to `'c'` then a client delay (typically of 500 msecs) is performed by calling `delay(delaymsecs)`. Before starting on the next randomly selected request from the schedule, the current request is overwritten by the request at the end of the schedule and `rqstvupb` decremented.

Execution eventually falls out of both nested loops, indicating that the client has completed all its schedules. The following code then closes down the client.

```
fin:
  IF requestv DO freevec(requestv)
  IF datav    DO freevec(datav)

  // Tell the stats task that this client has finished its
  // final schedule.
  TEST modech='R'
  THEN { IF tracing DO
           sawritef("T%z2 %cC%z2:*
                 *    Sending act_rddone to the stats task*n",
                 taskid, modech, clino)
      sendpkt(notinuse, statstaskid, act_rddone)
    }
ELSE { IF tracing DO
         sawritef("T%z2 %cC%z2:*
                 *    Sending act_wrdone to the stats task*n",
                 taskid, modech, clino)
      sendpkt(notinuse, statstaskid, act_wrdone)
    }

// Wait for the act_die packet from the controller
IF tracing DO
  sawritef("T%z2 %cC%z2:*
           *    waiting for die packet from the controller*n",
           taskid, modech, clino)

c_taskwait := c_taskwait+1
pkt := taskwait()

UNLESS pkt!pkt_type=act_die DO
{ sawritef("T%z2 %cC%z2:*
           * System error -- act_die packet expected*n", taskid)
  abort(999)
}

IF tracing DO
  sawritef("T%z2 %cC%z2:*
           * die packet received*n",
            taskid, modech, clino)

// Send this client's statistics to the stats task
sendstats()

IF tracing DO
  sawritef("T%z2 %cC%z2:*
           *    Returning the die packet to the controller*n",
           taskid, modech, clino)

c_qpkt := c_qpkt+1 // For qpkt below
qpkt(pkt)

IF tracing DO
```

```
    sawritef("T%z2 %cC%z2:   Returning to DEAD state*n",
              taskid, modech, clino)

  RETURN  // Return this Client to DEAD state
}
```

This code returns the vectors `requestv` and `datav` to free store, then sends a `act_rddone` or `act_wrdone`, as appropriate, to the `stats` task. These are only returned from the `stats` task when all other clients have reached the same state. The client then waits in `taskwait` for the `die` packet from the controller. This causes the client to send its statistics counts to the `stats` task before returning the `die` packet to the controller, and finally returning from the task's main function causing the task to return to DEAD state, ready for deletion by the controller.

Chapter 6

The Read and Write Servers

The read and and write servers receive requests from read and write clients and pass them on to multiplexor tasks. They run in multi-event mode (using `gomultievent`) allowing them to service several requests at the same time. In this mode requests are processed by worker coroutines. If a worker is held up for any reason other workers are able to gain control. Worker coroutines have access to the global vector of the task they belong to. Only one worker can be executing at any moment and it has control over when it hands over control to another coroutine. This greatly reduces the need to use synchronisation primitives to control access to shared data. On some occasions synchronisation primitive are needed and these are provided by the functions `lock`, `unlock`, `condwait`, `notify`, `notifyAll`, `coread` and `cowrite`, described below. Each server has a logger coroutine that workers communicate with after a request has been successfully processed, and loggers occasional send requests to the `printer` task.

Read and write servers have much in common and so share the same main function which starts as follows.

```
AND startserver(pkt) BE
{ // This is the body of a read or write server task most of which runs
  // in multi-event mode using gomultievent.

  // pkt is the startup packet of the form:
  //      act_startrdserver   a1:serno
  // or   act_startwrserver   a1:serno

  // This initialises the task as either a read or write server and
  // sets its own identity in id.

  startuppkt := pkt      // Used by servermaincofn

  serno  := pkt!pkt_arg1 // The read or write server number
  modech := pkt!pkt_type=act_startrdserver -> 'R', 'W'
  set_process_name("%cS%z2", modech, serno)

  // Set random number seed for random values to send
  // by worker coroutines to the logger.
  setseed(serno+modech*100)

  gomultievent(servermaincofn, 1000)
```

The argument `pkt`) is the startup packet and its type field will be either `act_startrdserver` or `act_startwrserver` depending on the kind of server being started. In either case the `arg1` field holds the server number to be copied into the global `srvno`. The variable `modech`, the task's name and the random number seed are then set before calling `gomultievent` to enter multi-event mode. This function is described in detail in section 8.1 below. It enters multi-event mode and creates a coroutine who main function is `servermaincofn` which is then given control. This coroutine allocated some work space and creates and initialises the logger coroutine and the workers, finally suspending itself waiting for a packet typically from a client task.

The server will only leave its event loop in `gomultievent` when it receives a `die` packet from the `controller` task after the client tasks have finished all they have to do. When this happens execution falls into the following code.

```
fin:
  IF tracing DO
    sawritef("T%z2 %cS%z2:    dying*n", taskid, modech, serno)

  // Send this server's statistics to the stats task.
  IF tracing DO
    sawritef("T%z2 %cS%z2:*
              *    Sending server statistics to stats task*n",
              taskid, modech, serno)

  sendstats()

  // Return the die packet to the controller.
  c_qpkt := c_qpkt+1
  qpkt(diepkt)
  IF tracing DO
    sawritef("T%z2 %cS%z2:   returning to DEAD state*n",
               taskid, modech, serno)

  RETURN  // Return server to DEAD state
}
```

The event loop has been exited from as a result of receiving a `die` packet which is placed in `diepkt` for later use. The server's statistics counters are then sent to the `stats` task and the `die` packet returned to the `controller` task by the call `qpkt(diepkt)`. Control then returns from `startserver` causing the task to return to DEAD state ready for deletion by the controller.

Before describing of `servermanicofn` it is worth looking at the functions used by logger coroutine. The main function is as follows.

```
AND loggercofn(arg) BE
{ LET i = 0
  IF tracing DO
    sawritef("T%z2 %cS%z2Log: Ready*n", taskid, modech, serno)

  { // Start of logger loop.
    LET a, b = ?, ?
    a := coread(@loggerin)
    //IF tracing DO
    //  sawritef("T%z2 %cS%z2log: Received a=%n*n",
    //            taskid, modech, serno, a)
    c_logger := c_logger + 1
    i := i+1

    // Occasionally the logger sends a message to the printer task
    IF i MOD 50 = 0 DO
    { //IF tracing DO
      //  sawritef("T%z2 %cS%z2log:*
      //           * logger sending pkt to printer*n",
      //            taskid, modech, serno)
      sendpkt(notinuse, printertaskid, act_print, 0, 0, modech, serno)
      //IF tracing DO
      //  sawritef("T%z2 %cS%z2log:*
      //           * pkt returned from printer*n",
      //           taskid, modech, serno)
    }

    // Occasionally the logger delays briefly
    IF i MOD 7 = 0 DO delay(2)

    b := coread(@loggerin)
    //IF tracing DO
    // sawritef("T%z2 %cS%z2log: Received b=%n*n",
    //           taskid, modech, serno, b)

    { LET x = a+b
      //IF tracing DO
      //  sawritef("T%z2 %cS%z2log: Replying %n*n",
      //            taskid, modech, serno, x, x)
      wrpn(x)
      cowrite(@loggerout, -1)
    }
  } REPEAT
}
```

A logger has a main loop that processes communication with worker coroutines. This involves receiving two numbers from a worker and sending back their sum. Communication is done using the Occam style channel functions `coread` and `cowrite`, described in Section 8.2 below. There is, however, the complication that between reading the first and second numbers, the logger may communicate with the `printer` task or executes a brief real time delay. Worker coroutines will thus need to uses the synchronisation primitives `lock` and `unlock` described in Section 8.3 below.

Each server has global variables `loggerin` and `loggerout` representing Occam style channels, and uses calls such as `coread(@loggerin)` and `cowrite(@loggerin,x)` to transmit a value through a channel. The logger places the first value in the local variable `a` then possibly sends a request to the `printer` task using `sendpkt` followed by possibly delaying for 2 msecs before reading the second values into `b`. It then sends the sum `a+b` to the worker in binary as a sequence of 0s and 1s terminated by a `-1`.

This conversion to binary is performed by the function `wrpn` whose definition is as follows.

```
AND wrpn(x) BE IF x DO
{ wrpn(x>>1)
  cowrite(@loggerout, x&1)
}
```

The conversion from binary back to a number is done by the function `getloggerval`, defined below. This funtion is used by worker coroutines.

```
AND getloggerval() = VALOF
{ LET res = 0

  { LET dig = coread(@loggerout)
    IF dig<0 RESULTIS res
    res := res+res+dig
  } REPEAT
}
```

We are now ready to look at the definition of `servermaincofn` which starts as follows.

```
AND servermaincofn() BE
{ // This is a server's main coroutine running in multi-event mode under
  // the control of gomultievent.
  LET pkt, type = ?, ?

  // mainco_ready is initially set to FALSE by gomultievent.

  diepkt := 0   // Only non zero when the die packet has been received.

  // Condition variables are held in this server's globals and shared
  // by this server's worker coroutines.

  // When a work coroutine wishes to process a request it must increment
  // its count but may have to wait, using condwait(@countwaitlist),
  // if wrkcount-minwrkcount has grown too large. When it increments its
  // count and it is possible that minwrkcount must also be incremented.
  // If this happens all coroutine waiting on condcondvar should be
  // released. This is done by the call: notifyAll(@countwaitlist).

  countcondvar := 0 // Used by workers wishing to increment wrkcount.
  pktcondvar   := 0 // Used by workers waiting for packets to service.

  wkq := 0          // List of packets waiting to be served by worker
                    // coroutines.

  wrkcov := getvec(workmax)    // Vector of worker coroutines.
  countv := getvec(workmax)    // Vector of workcounts.

  UNLESS wrkcov & countv DO
  { sawritef("T%z2 %cS%z2: Unable to allocate vectors*n",
             taskid, modech, serno)
    GOTO fin
  }

  // There are currently no worker coroutines and all counts are zero.
  FOR i = 0 TO workmax DO wrkcov!i, countv!i := 0, 0

  minwrkcount := 0

  busycount  := 0    // Count of worker coroutines that are busy
                     // A worker coroutine is busy when it is processing
                     // a request packet. It is not busy when waiting on
                     // countcondvar or pktcondvar.

  loggerlock := 0    // The logger mutex
  loggerin  := 0     // Occam style channel for logger input
  loggerout := 0     // Occam style channel for logger output

  // Create the logger coroutine for this server.
  loggerco := initco(loggercofn, 1000)
  UNLESS loggerco DO
  { sawritef("T%z2 %cS%z2:*
             * Unable to create logger coroutine*n",
              taskid, modech, serno)
    GOTO fin
  }

  c_callco := c_callco+1 // Because of callco in initco above

  // Create the server work coroutines that will process read or
  // write requests
  FOR wkno = 1 TO workmax DO
  { LET co = initco(workcofn, 1000, wkno)
    UNLESS co DO
    { sawritef("T%z2 %cS%z2W%z2: Unable to create worker coroutine*n",
                taskid, modech, serno, wkno)
      GOTO fin
    }
    c_callco := c_callco+1 // Because of callco in initco above
    wrkcov!wkno := co
    IF tracing DO sawritef("T%z2 %cS%z2W%z2: Worker coroutine ready*n",
                            taskid, modech, serno, wkno)
  }

  //IF tracing DO
  //  sawritef("T%z2 %cS%z2: Ready*n", taskid, modech, serno)

  c_qpkt := c_qpkt+1
  qpkt(startuppkt)
```

This coroutine initialises the variables used by this server. The two condition variables `countcondvar` and `pktcondvar` are both set to zero. The variable `wkq` holds the list of read or write requests received that have not yet been given to worker coroutines because, at the time they were received, all workers were busy with other requests. The vectors `wrkcov` and `countv` range from `1` to `workmax` and hold the worker coroutine pointers and their counts.

The variables `minwrkcount` and `busycount` hold the count value of the least busy worker and the count of how many workers are currently busy processing requests. They are thus both initialised to zero.

Worker coroutines communicate with the logger coroutine using the Occam style channels `loggerin` and `loggerout`. To access these channels, workers must obtain the `loggerlock` lock. This ensures that a worker can complete all its interactions with the logger without other workers interfering. The logger coroutine is held in `loggerco` and the workers and their counts are held in successive elements of `wrkcov` and `countv`.

After initialisation has been completed, the startup packet is returned to the controller and the coroutine enters its event loop whose code is follows.

```
{ // Start of this server's event loop
  LET chnno, data = ?, ?

  // Get a new request from gomultievent.
  // It should be a read, write or die packet.
  // Bounce or clock packets belonging to worker
  // coroutines will be delivered automatically
  // to their coroutines by gomultievent.

  IF tracing DO
    sawritef("T%z2 %cS%z2:*
             *    Waiting for a request packet from a client*n*n",
              taskid, modech, serno)
  mainco_ready := TRUE
  c_cowait := c_cowait+1
  pkt := cowait()
  mainco_ready := FALSE

  SWITCHON pkt!pkt_type INTO
  { DEFAULT:
      sawritef("T%z2 %cS%z2:*
               * Unexpected packet received from %cC%z2*n",
                taskid, modech, serno, modech, clino)
      abort(999)

    CASE act_read:
    CASE act_write:
      type   := pkt!pkt_type
      clino  := pkt!pkt_arg1
      //This work coroutine already knows its serno (and modech).
      mpxno  := pkt!pkt_arg3
      chnno  := pkt!pkt_arg4
      data   := pkt!pkt_arg5  // If it is a write request

    { // Append the packet onto the end of the server's wkq
      // and release the first coroutine, if any, waiting on
```

```
        // pktcondvar
        LET p = @wkq
        WHILE !p DO p := !p
        !p, !pkt := pkt, 0
        //IF tracing DO
        //  sawritef("T%z2 %cS%z2:*
        //           * Calling notify(%n) for condition pktcondvar*n",
        //            taskid, modech, serno, @pktcondvar)

        // Wake up a worker, if any, waiting for a request.
        notify(@pktcondvar)
        LOOP
      }

       CASE act_die:  // From the controller task.
         // Cause this server to return to DEAD state.
         IF tracing DO
           sawritef("T%z2 %cS%z2: Die packet received*n",
                    taskid, modech, serno)
         diepkt := pkt
         // Release all workers on countcondvar.
         // They will all notice that diepkt is non zero.
         notifyAll(@countcondvar)
         BREAK  // Ie go to fin
    }
  } REPEAT

fin:
  // Delete all worker coroutines and the logger.
  IF tracing DO
    sawritef("T%z2 %cS%z2: Deleting worker coroutines*n",
              taskid, modech, serno)

  FOR wkno = 1 TO workmax IF wrkcov!wkno DO deleteco(wrkcov!wkno)
  IF tracing DO
    sawritef("T%z2 %cS%z2: Deleting logger coroutine*n",
              taskid, modech, serno)
  IF loggerco DO deleteco(loggerco)

  IF tracing DO
    sawritef("T%z2 %cS%z2: Freeing the work space*n",
              taskid, modech, serno)
  IF wrkcov   DO freevec(wrkcov)
  IF countv   DO freevec(countv)
  IF tracing DO
    sawritef("T%z2 %cS%z2:*
             * Cause gomultievent to return to single event mode*n",
             taskid, modech, serno)

  multi_done := TRUE
  c_cowait := c_cowait+1
  cowait()  // Return to the controller.

  // This point should not be reached.
  sawritef("T%z2 %cS%z2: servermaincofn: System error*n",
            taskid, modech, serno)
  abort(999)
}
```

The server receives packets using the taskwait call in gomultievent and those that do not belong to workers are passed to the main coroutine using callco. The main coroutine thus uses cowait to receive packets. When the main coroutine is ready to process a packet it sets mainco_ready to TRUE and as soon as it receives one it sets it back to FALSE.

A packet received by gomultievent that does not belong to multi-event coroutines is normally given to mainco, but, if mainco_ready is FALSE, gomultievent appends it to the end of queue. If mainco_ready is TRUE such a packet is given to mainco.

The event loop in mainco only expects read, write and die packets and these are processed by cases in the SWITCHON command.

A read or write request is appended it to wkq by mainco which then calls notify to wake up a worker, if any, that is ready to process the request. Note that it is the worker not servermaincofn that dequeues the packet from wkq.

When a die packet is received, it is placed in diepkt and all workers waiting on pktcondvar and countcondvar are woken up by the call of notifyAll. After this call all workers will have closed down. The die packet causes mainco to leave its event loop, where it deletes the logger and worker coroutines and frees the wrkcov and countv vectors, before setting multi_done to TRUE and returning control to the controller.

Worker coroutines process the requests received by servers. There are several ways in which a worker might be delayed while processing a request. For instance, if the worker has already successfully processed five more requests than the least busy worker, it will suspend itself until minwrkcount is incremented. It may also be delayed when communicating with the logger and if the request specified that there should be a delay in the server it will suspend itself in delay(delaymsecs). So that other workers can run while one is susspended, the server runs in multi-event mode using gomultievent.

When a server receives a request from a client it passes it on to the server's main coroutine which, in turn, passes to a worker to do the work. Since read and write workers have much in common they share the same main function. It is called `workcofn` and its definition starts as follows.

```
AND workcofn(args) BE
{ // This is the body of a server work coroutine.

  LET wkno = args!0 // As supplied by initco in servermaincofn

  // The server number is already set in serno and
  // modech is 'R' if the work coroutine belongs to a read server.
  // If modech is 'W' the work coroutine belongs to a write server.
  // Once initialised, the work coroutine is only given read, write
  // or die packets or replies from previous calls it made using
  // sendpkt

  { // Start of this worker's main loop.

    // First check that this worker's work count is small enough to
    // allow a request to be service.
    LET wrkcount = countv!wkno

    // workcount is the number of read or write packets this worker
    // has successfully processed.  It may never exceed the work count
    // of the least busy work coroutine plus maxcountdiff
    // (typically=5).

    UNLESS wrkcount <= minwrkcount+maxcountdiff DO
      c_incwait := c_incwait+1


    // Suspend this worker if necessary.
    UNTIL wrkcount <= minwrkcount+maxcountdiff | diepkt DO
      condwait(@countcondvar)
```

The worker already knows the number (`srvno`) of its server and whether it is a read or write worker (`modech`). But it must set its own worker number `wkno` obtained from `args`. It then enters the the worker's event loop where it first checks to see if its `wrkcount` is small enough for its to proceed by repeatedly calling `condwait(@countcondvar)}` until `minwrkcount` is large enough or a `die` packet has been received. The program continues as follows.

```
    IF diepkt DO
    { c_cowait := c_cowait + 1
    cowait()                 // Return control to servermaincofn
    abort(9999)              // We should never reach this point
  }

  // Wait for a read or write request

  UNTIL wkq DO
  { c_condwait := c_condwait+1
    condwait(@pktcondvar)
    // wkq now contains at least one request packet so this
    // worker can start processing it.
    busycount := busycount + 1
  }
```

If the server receives a die packet, it is placed in `diepkt` and all workers waiting on `countcondvar` are woken up including this one. The controller only sends a `die` packet when all read and write clients have completed their work, and all workers are no longer busy. At which time `wkq` and `countcondvar` will be both be zero. If a `die` packet has been received the worker immediately returns control to the server's main coroutine.

If a `die` packet has not been received the worker waits for a request packet to be placed in `wkq` calling `condwait(@pktcondvar)`, if necessary.

The following code is obeyed when `wkq` is known to hold a request packet.

```
  { // Extract the first packet from wkq
    LET pkt = VALOF
    { LET p = wkq
      wkq := !p
      !p:= notinuse
      RESULTIS p
    }
    LET type   = pkt!pkt_type
    LET flag   = pkt!pkt_arg1
    LET clino  = pkt!pkt_arg2
    //This work coroutine already knows its serno (and modech).
    LET mpxno  = pkt!pkt_arg4
    LET chnno  = pkt!pkt_arg5
    LET data   = pkt!pkt_arg6  // Non zero, if it is a write request

    IF tracing DO
      sawritef("T%z2 %cS%z2W%z2:*
              * Processing %c%cC%z2S%z2M%z2C%z2:%z4*n",
               taskid, modech, serno, wkno,
               flag, modech, clino, serno, mpxno, chnno, data)
```

This code dequeues the first packet from `wkq` and extracts it parameters into the variables `type`, `flag`, `clino`, `mpxno`, `chnno` and `data`. The program then attempts to process the request using the following code.

```
TEST modech='R'
THEN { // Read request.

       // Send the request to a multiplexor
       IF tracing DO
         sawritef("T%z2 RS%z2W%z2: %cRC%z2S%z2M%z2C%z2:%z4*
                  *  Sending read request to multiplexor*n",
                  taskid, serno, wkno,
                  flag, clino, serno, mpxno, clino, 0)
       data := sendpkt(notinuse, mpxv!mpxno, type,
                       0, 0,     flag, clino, serno, mpxno, chnno, 0)

       // data=0 if the specified channel buffer was empty

       UNLESS data DO
       { // The read failed so immediately return the read
         // request to the client.
         IF tracing DO
           sawritef("T%z2 RS%z2W%z2: %cRC%z2S%z2M%z2C%z2:%z4*
                    * Returning failed read request to client*n",
                     taskid, serno, wkno,
                     flag, clino, serno, mpxno, clino, data)

         pkt!pkt_r1 := 0  // Indicate failure.
         c_qpkt := c_qpkt+1
         qpkt(pkt)        // Return the read packet to the read client.
         busycount := busycount - 1 // This worker is no longer busy.

         LOOP
       }

       IF tracing DO
         sawritef("T%z2 RS%z2W%z2: %cRC%z2S%z2M%z2C%z2:%z4*
                  *  Read request returned from multiplexor with data*n",
                  taskid, serno, wkno,
                  flag, clino, serno, mpxno, clino, data)
     }
ELSE { // Write request
       LET rc = ?

       // Then send the request to the specified multiplexor
```

```
      IF tracing DO
        sawritef("T%z2 WS%z2W%z2: %cWC%z2S%z2M%z2C%z2:%z4*
                 * Sending write request to multiplexor %z2*n",
                  taskid, serno, wkno,
                  flag, clino, serno, mpxno, clino, data, mpxno)
      rc := sendpkt(notinuse, mpxv!mpxno, type,
                    0, 0,
                    flag, clino, serno, mpxno, chnno, data)
      // rc = TRUE  if successful write
      // rc = FALSE if the channel was full

      UNLESS rc DO
      { // The write failed so immediately return the write
        // request to the client.
        IF tracing DO
          sawritef("T%z2 WS%z2W%z2: %cWC%z2S%z2M%z2C%z2:%z4*
                   * Returning failed write request to client*n",
                    taskid, serno, wkno,
                    flag, clino, serno, mpxno, clino, data)

        pkt!pkt_r1 := FALSE // Indicate failure.
        c_qpkt := c_qpkt+1
        qpkt(pkt)           // Return the write packet.
        busycount := busycount - 1 // This worker is no longer busy.

        LOOP
      }
    }
```

Read and write requests are deal with separately by testing the value of `modech`.

A read request causes the worker to send a packet to the specified multiplexor task and wait for the reply which it then places in `data`. This is normally a non zero data value extracted from the specified channel buffer. But if the buffer was empty, zero was returned indicating failure. In which case the request packet is returned to the client with an indication of failure and the worker jumps back to the start of its main loop to process another request. Otherwise execution falls through to code deals with server actions after a after a successful request.

A write request causes the worker to send a packet to the specified multiplexor task and wait for the return code which it places in `rc`. This is normally a non zero indicating that data was successfully inserted into the specified channel buffer. But if the buffer was full, `rc` will be zero indicating failure. In which case the request packet is returned to the client with an indication of failure and the worker jumps back to the start of its main loop to process another request. Otherwise execution falls through to code deals with server actions after a successful request.

The program continues as follows.

```
   // The request was successful.

   IF flag='s' DO
   { // This request specified a server delay, so do it.

     IF tracing DO
       sawritef("T%z2 %cS%z2W%z2: %c%cC%z2S%z2M%z2C%z2:%z4*
                * Start server delay*n",
                 taskid, modech, serno, wkno,
                 flag, modech, clino, serno, mpxno, chnno, data)
     c_delaylong := c_delaylong+1
     delay(delaymsecs)
     IF tracing DO
       sawritef("T%z2 %cS%z2W%z2: %c%cC%z2S%z2M%z2C%z2:%z4*
                * Server delay done*n",
                 taskid, modech, serno, wkno,
                 flag, modech, clino, serno, mpxno, chnno, data)
   }
```

This code performs a delay of typically 500 msecs if the request requires a delay in the server. It is triggered by the `flag` field of the request being set to `s`. The `delay` function sends an appropriate packet to the clock device using `sendpkt` which in multi-event mode only returns when the packet comes back from the clock. See the definition of `gomultievent` and `sndpkt` for more details.

The program continues as follows.

```
   // A request has been successfully processed, so send a message
   // to the server's logger coroutine and check its reply.

   lock(@loggerlock)

   // Send two random numbers in succession to the logger.
   { LET x = nextrnd(99)
     LET y = nextrnd(99)

     IF tracing DO
       sawritef("T%z2 %cS%z2W%z2: Sending %z2 and %z2 to logger*n",
                 taskid, modech, serno, wkno, x, y)
     cowrite(@loggerin, x)

     // Send a pkt to the bounce task to give other tasks a chance
     // to run.
     sendpkt(notinuse, bouncetaskid, act_bounce, 0,0)
```

```
    IF tracing DO
      sawritef("T%z2 %cS%z2W%z2: Sending %z2 to logger, waiting for reply*n",
                taskid, modech, serno, wkno, y)
    cowrite(@loggerin, y)

    // Check the logger's reply is x+y.
    { LET reply = getloggerval()

      TEST reply=x+y
      THEN { IF tracing DO
               sawritef("T%z2 %cS%z2W%z2:*
                        * Reply from logger = %n -- Correct*n",
                         taskid, modech, serno, wkno, reply)
           }
      ELSE { sawritef("T%z2 %cS%z2W%z2:*
                      * Reply from logger = %n -- Bad, should be %n*n",
                      taskid, modech, serno, wkno, reply, x+y)
             abort(999)
           }
    }
  }

  unlock(@loggerlock)
```

This code is only obeyed if a read or write request was successfully performed. It sends two random numbers separately to the `logger` coroutine then waits for the `logger`'s reply. It is necessary to call `lock` before starting this communication and `unlock` at the end. See the definition below of these function for more details. The communication is done using the Occam style channels `loggerin` and `loggerout` implemented by the functions `cowrite` and `coread`. See their definitions below for more details. The value returned by the `logger` is the sum of the two random numbers, but this is sent in binary as a sequence of `0`s and `1`s terminated by `-1`.

The program continues as follows.

```
  // Now increment wrkcount, incrementing minwrkcount if necessary.

  countv!wkno := wrkcount + 1
  c_inc := c_inc + 1

  IF wrkcount=minwrkcount DO
  { // minwrkcount must be incremented if this worker is the
    // only worker with wrkcount=mincount
    minwrkcount := minwrkcount+1
    FOR wrkn = 1 TO workmax IF countv!wrkn < minwrkcount DO
    { // Restore previous minwrkcount
      minwrkcount := minwrkcount-1
      BREAK
    }
    wrkcount := wrkcount+1

    IF tracing DO
    { sawritef("T%z2 %cS%z2W%z2: Wrkcount=%n minwrkcount=%n*n",
                taskid, modech, serno, wkno, wrkcount, minwrkcount)
      prcounts()
    }

    IF wrkcount = minwrkcount DO
    { // minwrkcount has just been incremented so
      // wake up all coroutines waiting on countcondvar
      IF tracing DO
        sawritef("T%z2 %cS%z2W%z2:*
                 * minwrkcount incremented so calling notifyAll %n*n*n",
                  taskid, modech, serno, wkno, @loggerlock)
      notifyAll(@countcondvar)
    }
  }
```

At this stage the worker coroutine must increment it count of how many requests it has successfully processed. This count is held in the local variable `wrkcount` and `countv!wkno`. The only complication is that incrementing `wrkcount` may require `minwrkcount` to be incremented. This only happens if `wrkcount` and `minwrkcount` are equal and all other worker counts are greater than `minwrkcount`. If `minwrkcount` is incremented, all workers waiting for this to happen are woken up by the call `notifyAll(@countcondvar)`. This code is thus another demonstration of condition variables.

The program continues as follows.

```
  // The request has been successfully processed, so
  // return the request packet to the client.
  IF tracing DO
    sawritef("T%z2 WS%z2W%z2: %cWC%z2S%z2M%z2C%z2:%z4*
             * Returning successful request to client*n",
              taskid, serno, wkno,
              flag, clino, serno, mpxno, clino, data)

  TEST modech='R'
  THEN pkt!pkt_res1 := data  // Return the data value, or
  ELSE pkt!pkt_res1 := TRUE  // indicate a successful write.
```

```
      c_qpkt := c_qpkt+1
      qpkt(pkt)                  // Return the request packet to the client.

      busycount := busycount - 1 // This worker is no longer busy.
    }
  } REPEAT
}
```

This code just returns the request packet to its client after filling in a suitable result in its `res1` field. It then goes back to the start of the event loop ready to process another request. Notice that it decrements `busycount` since this worker is no longer processing a request. Notice also that this is done without the need synchronisation primitives since it is only coroutines in this task that accesses `busycount` and only one of them can run at any time and they give up control voluntarily.

```
AND prcounts() BE IF tracing DO
{ sawritef("T%z2 %cS%z2:    Wrkcounts  ",
            taskid, modech, serno)
  FOR wkno = 1 TO workmax DO
    sawritef(" W%z2/%n", wkno, countv!wkno)
  sawritef("   minwrkcount=%n*n", minwrkcount)
}
```

This function is used as a debugging aid to output the worker's counts allowing them to be checked than none exceed `minwrkcount` by more than 5.

Chapter 7

The Multiplexor Tasks

Unlike servers which always process either read or write requests, multiplexors process both read and write requests and expect them in random order. A multiplexor controls several channels which each has an associated buffer to hold data that has been written but not yet read. Each channel is controlled by a read and a write worker coroutine, both of which may perform a real time delays. So that other workers are not held up, the task runs in multi-event mode under the control of `gomultievent`.

```
AND startmpx(pkt) BE
{ // pkt is the startup packet giving the mpx number in its arg1 field.
  // Its type field should be {\tt act\_startmpx}.

  UNLESS pkt!pkt_type=act_startmpx DO
  { sawritef("T%z2: ERROR: startmpx packet expected*n")
    abort(999)
  }

  mpxno := pkt!pkt_arg1  // The number of this mpx task.

  mpxbusycount := 0      // The number of channel corroutines
                         // that are currently busy (ie not
                         // waiting for requests).

  startuppkt := pkt      // This global variable allows
                         // mpxmaincofn to return the startup
                         // packet to the controller when
                         // initialisation is complete.

  diepkt := 0            // This will contain the die packet
                         // when it is received.
  set_process_name("M%z2", mpxno)
  //IF tracing DO
  //  sawritef("T%z2 M%z2: Calling gomultievent*n", taskid, mpxno)

  gomultievent(mpxmaincofn, 1000)
```

This code processes the multiplexor startup packet. It sets the multiplexor number obtained from the `arg1` field and initialises `mpxbusycount` and `diepkt` to zero. The startup packet is placed in to global `startuppkt` for use by `mpxmaincofn`. It then enters multi-event mode by calling `gomultievent`. This creates and enters the main coroutine whose body is `mpxmaincofn`. It only returns from `gomultievent` after receiving a `die` packet indicating that all data transfers have been completed. At this moment it falls into the following code.

```
fin:
  IF tracing DO
    sawritef("T%z2 M%z2:    Sending stats data to the stats task*n",
              taskid, mpxno)

  // Send this multiplexor's statistics to the stats task
  sendstats()

  IF tracing DO
    sawritef("T%z2 M%z2:*
             * Returning the die packet to the controller*n",
             taskid, mpxno)
  c_qpkt := c_qpkt+1
  qpkt(diepkt)

  IF tracing DO
    sawritef("T%z2 M%z2:   Returning to DEAD state*n",
              taskid, mpxno)

  RETURN  // Return this multiplexor to DEAD state
}
```

This code just sends the multiplexors statistics counts to the `stats` task before returning the `die` packet to the controller. It then returns from `startmpx` causing the task to return to DEAD state ready for deletion. The real work of the multiplexor is controlled by its main coroutine whose body starts as follows.

```
AND mpxmaincofn() BE
{ // Allocate all the multiplexor work space.

  mpxrdcov   := getvec(chnmax)
```

```
mpxwrcov   := getvec(chnmax)
rdbusyv    := getvec(chnmax)
wrbusyv    := getvec(chnmax)
rdwkqv     := getvec(chnmax)
wrwkqv     := getvec(chnmax)
bufv       := getvec(chnmax)
bufpv      := getvec(chnmax)
bufqv      := getvec(chnmax)
                             // waiting fo data.

UNLESS mpxrdcov & mpxwrcov &
       rdbusyv  & wrbusyv &
       rdwkqv   & wrwkqv &
       bufpv    & bufqv &
       bufv     DO
{ sawritef("T%z2 M%z2: getvec failure*n", taskid, mpxno)
  GOTO fin
}

FOR chnno = 1 TO chnmax DO
{ mpxrdcov!chnno, mpxwrcov!chnno := 0, 0
  rdbusyv!chnno, wrbusyv!chnno := FALSE, FALSE
  rdwkqv!chnno, wrwkqv!chnno := 0, 0
  bufv!chnno := 0
  bufpv!chnno, bufqv!chnno, bufv!chnno := 0, 0, 0
}

// Allocate all the channel buffers
FOR chnno = 1 TO chnmax DO
{ LET buf = getvec(chnbufsize)
  UNLESS buf DO
  { sawritef("T%z2 M%z2C%z2: Unable to allocate a buffer*n",
             taskid, mpxno, chnno)
    GOTO fin
  }
  bufv!chnno := buf
  IF tracing DO
    sawritef("T%z2 M%z2C%z2:  Allocated buffer size=%n*n",
             taskid, mpxno, chnno, chnbufsize)
}

// Create and initialise all the channel read coroutines.

FOR chnno = 1 TO chnmax DO
{ // The following call will create a channel read coroutine
```

```
    // leaving it waiting for a read request.
    LET co = initco(mpxrdcofn, 1000, mpxno, chnno)
    UNLESS co DO
    { sawritef("T%z2 M%z2RC%z2: Unable to create read coroutine*n",
                taskid, mpxno, chnno)
      GOTO fin
    }
    c_callco := c_callco+1 // Because of callco in initco above
    mpxrdcov!chnno := co
    IF tracing DO
      sawritef("T%z2 M%z2RC%z2:*
               * Created and waiting for a read packet*n",
               taskid, mpxno, chnno)
  }

  // Create and initialise all the channel write coroutines.

  FOR chnno = 1 TO chnmax DO
  { // This creates a channel write coroutine leaving it
    // waiting in cowait for a multiplexor write packet delivered
    // by gomultievent.
    LET co = initco(mpxwrcofn, 1000, mpxno, chnno)
    UNLESS co DO
    { sawritef("T%z2 M%z2WC%z2: Unable to create write coroutine*n",
                taskid, mpxno, chnno)
      GOTO fin
    }
    c_callco := c_callco+1 // Because of callco in initco above
    mpxwrcov!chnno := co
    IF tracing DO
      sawritef("T%z2 M%z2WC%z2:*
               * Created and waiting for a write packet*n",
               taskid, mpxno, chnno)
  }

  IF tracing DO
    sawritef("T%z2 M%z2:*
             * All read and write coroutines created and ready*n*n",
             taskid, mpxno)

  c_qpkt := c_qpkt+1
  qpkt(startuppkt)
```

It starts by allocating several vectors. They all have elements that range from 1 to chnmax corresponding to one element per channel. The read and write coroutines for each channel are held in `mpxrdcov` and `mpxwrcov`. The vectors `rdbusyv` and `wrbusyv` indicate which read and write coroutines are busy processing requests, and `rdwkqv` and `wrwkqv` hold lists of requests for channel coroutines that cannot yet be processed because, at the time they were received, their coroutines were busy. Each channel has a buffer held in `bufv` and two pointers into each buffer held in `bufpv` and `bufqv`. The buffers are circular and capable of holding `chnbufsize` values.

`bufpv!chnno` is the next available write position in the specified buffer, and `bufqv!chnno` is the position in the specified buffer of the next value to be read. If `bufpv!chnno=bufqv!chnno` the circular buffer for channel `chnno` is empty.

After all these vectors have been initialised the startup packet is returned to the controller, and execution enters the multiplexor's event loop which starts as follows. It deals with packets of type `act_read`, `act_write` and `act_die`.

```
{ // Start of this multiplexor's event loop

  // Get next packet from a server via gomultievent
  LET pkt  = ?
  LET type = ?

  // Get a new request from gomultievent
  mainco_ready := TRUE

  // mainco_ready is TRUE when this multiplexor is ready to
  // accept a read, write or die packets.

  c_cowait := c_cowait+1
  pkt  := cowait() // Wait for a read or write from a server
                   // via gomultievent(...),
                   // or a die packet from the controller.
  type := pkt!pkt_type

  mainco_ready := FALSE

  SWITCHON type INTO
  { DEFAULT:
      sawritef("T%z2 %cS%z2:*
               * Unexpected packet received from %cC%z2*n",
                taskid, modech, serno, modech, clino)
      abort(999)

    CASE act_read:
    // arguments: a1:flag a2:clino a3:servno a4:mpxno a5:chnno.

    { LET flag   = pkt!pkt_arg1
      LET clino  = pkt!pkt_arg2
      LET serno  = pkt!pkt_arg3
      LET mpxno  = pkt!pkt_arg4
      LET chnno  = pkt!pkt_arg5
      LET data   = pkt!pkt_arg6  // =0 for read requests

      IF tracing DO
        sawritef("T%z2 M%z2:*
                 *     %cRC%z2S%z2M%z2C%z2:%z4 Read request received*n",
                  taskid, mpxno,
                  flag, clino, serno, mpxno, chnno, data)

      // Insert this packet at head of the queue of
      // read work queue for this channel.

      pkt!pkt_link := rdwkqv!chnno
      rdwkqv!chnno := pkt

      IF tracing DO
        sawritef("T%z2 M%z2RC%z2:*
                 * Inserting read request at head of its rdwkq*n",
                 taskid, mpxno, chnno)

      UNLESS rdbusyv!chnno DO
      { c_callco := c_callco+1
        callco(mpxrdcov!chnno) // Wake up the channel read coroutine
                               // if it was not busy.
      }
      // Process next read, write or die request.
      LOOP
    }
```

If the read coroutine for this channel is currently busy, this request is inserted in its `rdwkq` queue, otherwise it is given to the read coroutine for this channel. A read coroutine is busy when it is processing a read packet for its channel, and is only not busy when it is waiting for another request.

```
  CASE act_write:
  // Arguments: a1:flag a2:clino a3:serno a4:mpxno a5:chnno a6:data.

  { LET flag   = pkt!pkt_arg1
    LET clino  = pkt!pkt_arg2
    LET serno  = pkt!pkt_arg3
    LET mpxno  = pkt!pkt_arg4
    LET chnno  = pkt!pkt_arg5
```

```
      LET data   = pkt!pkt_arg6  // =0 for read requests

      IF tracing DO
        sawritef("T%z2 M%z2:*
                 *     %cWC%z2S%z2M%z2C%z2:%z4 Write request received*n",
                  taskid, mpxno,
                  flag, clino, serno, mpxno, chnno, data)

      // Insert this packet at head of the queue of
      // write work queue for this channel.

      pkt!pkt_link := wrwkqv!chnno
      wrwkqv!chnno := pkt

      IF tracing DO
        sawritef("T%z2 M%z2WC%z2:*
                 * Inserting a write request at head of its wrwkq*n",
                 taskid, mpxno, chnno)

      UNLESS wrbusyv!chnno DO
      { c_callco := c_callco+1
        callco(mpxwrcov!chnno) // Wake up the channel write coroutine
                               // if it was not busy.
      }
      // Process next read, write or die request.
      LOOP
    }
```

If the write coroutine for this channel is currently busy, this request is inserted in its `wrwkq` queue, otherwise it is passed to its write coroutine. If the buffer for the specified channel is not full, the write coroutine transfers the data into the buffer. If the buffer was empty, it wakes up a waiting read coroutine, if any. If a delay is specified it performs the delay. Finally, it returns the request packet to its server and waits for the next request. If the buffer is full, the write packet is immediately returned to its write server with an indication of failure in its `res1` field. A write coroutine is only busy while it is processing a request, and only not busy when it is waiting for another request.

```
    CASE act_die:  // From the controller task
    // This packet has no arguments.

      IF tracing DO
        sawritef("T%z2 M%z2:*
                 * Die packet received*n",
                  taskid, mpxno)
        diepkt := pkt
        BREAK
    }
  } REPEAT

fin:

  // For safety check that none of the read or write
  // coroutines are busy and that their work queues are
  // empty, and that all channel buffers are empty.

  FOR chnno = 1 TO chnmax DO
  { IF bufpv & bufqv UNLESS bufpv!chnno=bufqv!chnno DO
    { sawritef("T%z2 M%z2C%z2: Channel buffer is not empty*n",
                taskid, mpxno, chnno)
      prbufs()
      abort(999)
    }
    IF rdwkqv & rdwkqv!chnno DO
    { sawritef("T%z2 M%z2C%z2: Read work queue is not empty*n",
                taskid, mpxno, chnno)
      abort(999)
    }
    IF wrwkqv & wrwkqv!chnno DO
    { sawritef("T%z2 M%z2C%z2: Write work queue is not empty*n",
                taskid, mpxno, chnno)
      abort(999)
    }
    IF rdbusyv & rdbusyv!chnno DO
    { sawritef("T%z2 M%z2C%z2: A read coroutine is still busy*n",
                taskid, mpxno, chnno)
      abort(999)
    }
    IF wrbusyv & wrbusyv!chnno DO
    { sawritef("T%z2 M%z2C%z2: A write coroutine is still busy*n",
                taskid, mpxno, chnno)
      abort(999)
    }
  }

  // Delete all the read and write coroutines

  FOR chnno = 1 TO chnmax DO
  { IF mpxrdcov & mpxrdcov!chnno DO deleteco(mpxrdcov!chnno)
    IF mpxwrcov & mpxwrcov!chnno DO deleteco(mpxwrcov!chnno)
  }

  // Free all the channel buffers
  IF bufv FOR chnno = 1 TO chnmax IF bufv!chnno DO
    freevec(bufv!chnno)

  // Free all the vectors
  IF mpxrdcov   DO freevec(mpxrdcov)
  IF mpxwrcov   DO freevec(mpxwrcov)
  IF rdbusyv    DO freevec(rdbusyv)
  IF wrbusyv    DO freevec(wrbusyv)
  IF rdwkqv     DO freevec(rdwkqv)
  IF wrwkqv     DO freevec(wrwkqv)
  IF bufv       DO freevec(bufv)
  IF bufpv      DO freevec(bufpv)
  IF bufqv      DO freevec(bufqv)

  IF tracing DO
    sawritef("T%z2 M%z2:*
             * Cause gomultievent to return to single event mode*n",
             taskid, mpxno)

  multi_done := TRUE // Leave multi event mode
  c_cowait := c_cowait+1
  cowait()  // Return to mpxserver in single event mode.

  // This point should not be reached.
  sawritef("T%z2 M%z2: mpxmaincofn: System error*n", taskid, mpxno)
  abort(999)
}
```

A `die` packet only arrives when all client tasks have completed all iterations of their schedules. It causes the multiplexor to leave its event loop and fall into the code that sends its accumulated statistics to the `stats` task, deletes all the multiplexor coroutines and work space and returns to DEAD state, after returning the `die` packet.

Channel read and write coroutines have main functions `mpxrdcofn` and `mpxwrcofn`. The read function starts as follows.

```
AND mpxrdcofn(arg) BE
{ // This is the main function of a read coroutine for channel chnno.
  // It runs in multi-event mode and is woken up by mpxmaincofn when
  // there is a request to process and it is not busy.

  LET mpxno, chnno = arg!0, arg!1
  LET buf   = bufv!chnno
  LET p, q  = ?, ?
  LET flag  = ?
  LET clino = ?
  LET serno = ?
  LET data  = ?
  LET pkt   = ?

  rdbusyv!chnno := TRUE // Only =FALSE when waiting to be woken up.
  mpxbusycount := mpxbusycount+1
```

It first initialises `mpxno` and `chnno` to the multiplexor number and the number of that channel it is controlling from fields in its startup argument. It also initialises `buf` to point to the buffer for the specified channel. It then sets `rdbusyv!chnno` to `TRUE` indicating that this coroutine is currently busy, and also increments `mpxbusycount` which hold the count of how many read and write coroutine of this multiplexor are busy. Execution then falls into the coroutine's main loop which starts as follows.

```
{ // Start of the main loop for the read coroutine for this channel.

  // Get the next read packet, waiting if necessary.

  UNTIL rdwkqv!chnno DO
  { rdbusyv!chnno := FALSE
    mpxbusycount := mpxbusycount-1
    IF tracing DO
    { sawritef("T%z2 M%z2RC%z2:*
               * Waiting for a read request, busycount=%n*n",
               taskid, mpxno, chnno, mpxbusycount)
      //abort(1000)
    }
    c_cowait := c_cowait+1
    cowait()
    rdbusyv!chnno := TRUE
    mpxbusycount := mpxbusycount+1
  }
```

This channel has a list of pending read packets held in `rdwkqv!chnno`. If the list is empty, the coroutine must wait until woken up by a call of `callco` when a suitable read packet becomes available. When this happens the following code is executed.

```
  // Dequeue the next read packet
```

```
pkt := rdwkqv!chnno
rdwkqv!chnno := pkt!pkt_link
pkt!pkt_link := notinuse

// Extract its parameters
flag  := pkt!pkt_a1
clino := pkt!pkt_a2
serno := pkt!pkt_a3
data  := pkt!pkt_a6

p, q := bufpv!chnno, bufqv!chnno

IF p=q DO
{ // The channel buffer is empty, so return the packet
  // to the read server with and indication of failure.
  IF tracing DO
  { prbufs()
    sawritef("T%z2 M%z2RC%z2: %cRC%z2S%z2M%z2C%z2:%z4*
             * Returning failed read packet to its server*n",
              taskid, mpxno, chnno, flag,
              clino, serno, mpxno, chnno, data)
  }

  pkt!pkt_r1 := 0      // Indicate failure
  c_qpkt := c_qpkt+1
  qpkt(pkt)
  LOOP
}
```

This extracts the parameters of the read packet and then tests to see if the channel buffer is empty. If it is, it fills in the `res1` field of the packet to indicate failure and returns it to its server using `qpkt` before returning to the start of the main loop. If the buffer was not empty, execution falls into the following code.

```
pkt!pkt_r1 := buf!q
bufqv!chnno := (q+1) MOD chnbufsize

IF tracing DO
{ sawritef("T%z2 M%z2RC%z2:*
           * Data %z4 extracted from buffer position %n*n",
           taskid, mpxno, chnno, buf!q, q)
  prbufs()
}
```

```
    // Conditionally perform a multiplexor delay
    IF flag='m' DO
    { IF tracing DO
        sawritef("T%z2 M%z2RC%z2: %cRC%z2S%z2M%z2C%z2:%z4*
                 * Channel delay*n",
                  taskid, mpxno, chnno, flag,
                  clino, serno, mpxno, chnno, data)
      c_delaylong := c_delaylong+1
      delay(delaymsecs)
      IF tracing DO
        sawritef("T%z2 M%z2RC%z2: %cRC%z2S%z2M%z2C%z2:%z4*
                 * Channel delay done*n",
                  taskid, mpxno, chnno, flag,
                  clino, serno, mpxno, chnno, data)
    }

    // Return the successful read request packet to its server.
    IF tracing DO
      sawritef("T%z2 M%z2RC%z2:*
               * %cRC%z2S%z2M%z2C%z2:%z4*
               * Returning successful read packet to its server*n",
                taskid, mpxno, chnno, flag,
                clino, serno, mpxno, chnno, data)

    c_qpkt := c_qpkt+1
    qpkt(pkt) // Return the read request packet to its read server.

    // Process the next read packet, if any.
  } REPEAT
}
```

Here it extracts the next data value from the buffer placing in the `res1` field of the packet before incrementing the buffer's read pointer. Remember that the buffer is a circular and of size `chnbufsize`. If `flag` is set to `m`, a multiplexor delay must be performed. This involves a call of `delay` which in multi-event mode does not return until the specified delay period has completed. Finally, the read packet is returned to its server using `qpkt`.

The main function of write coroutine is similar to `mpxrdcofn` but is sufficiently different to deserve its own function which starts as follows.

```
AND mpxwrcofn(arg) BE
{ LET mpxno, chnno = arg!0, arg!1
  LET buf   = bufv!chnno
  LET p, q  = ?, ?
  LET r     = ?
  LET flag  = ?
  LET clino = ?
  LET serno = ?
  LET data  = ?
  LET pkt   = ?

  wrbusyv!chnno := TRUE // Only =FALSE when waiting to be woken up.
  mpxbusycount := mpxbusycount+1
```

As in `mpxrdcofn`, it intialises `mpxno`, `chnno` and `buf`, `mpxbusycount` and the appropriate element of `wrbusyv`. It then enters its main loop which is as follows.

```
{ // Start of the main loop for the write coroutine for this channel.

  // Get the next write packet, waiting if necessary.

  UNTIL wrwkqv!chnno DO
  { wrbusyv!chnno := FALSE
    mpxbusycount := mpxbusycount-1
    IF tracing DO
    { sawritef("T%z2 M%z2WC%z2:*
               * Waiting for a write request*n",
               taskid, mpxno, chnno)
      //abort(1000)
    }
    c_cowait := c_cowait+1
    cowait()
    wrbusyv!chnno := TRUE
    mpxbusycount := mpxbusycount+1
  }

  // Dequeue the next write packet
  pkt := wrwkqv!chnno
  wrwkqv!chnno := pkt!pkt_link
  pkt!pkt_link := notinuse

  // Extract its parameters
  flag  := pkt!pkt_a1
  clino := pkt!pkt_a2
  serno := pkt!pkt_a3
  data  := pkt!pkt_a6

  IF tracing DO
    sawritef("T%z2 M%z2WC%z2: %cWC%z2S%z2M%z2C%z2:%z4*
             * Processing a write request*n",
              taskid, mpxno, chnno, flag,
              clino, serno, mpxno, chnno, data)
  p, q := bufpv!chnno, bufqv!chnno
  r := (p + 1) MOD chnbufsize

  IF r=q DO
  { // The channel buffer was full, so return the packet
    // to the write server with and indication of failure.
    IF tracing DO
    { prbufs()
      sawritef("T%z2 M%z2WC%z2: %cWC%z2S%z2M%z2C%z2:%z4*
               * Returning failed write packet to its server*n",
                taskid, mpxno, chnno, flag,
                clino, serno, mpxno, chnno, data)
    }

    pkt!pkt_res1 := FALSE  // Indicate failure
    c_qpkt := c_qpkt+1
    qpkt(pkt)
    LOOP
  }
```

It extracts the parameters of the write packet and then tests to see if the channel buffer is full. If it is, it places 0 in the res1 field to indicate failure, and returns the packet to its server before jumping back to the start of the main loop.

If the buffer was not full, it falls into the following code.

```
// Push the data value into the channel buffer

pkt!pkt_res1 := TRUE  // Indicate successful write
buf!p := data
bufpv!chnno := r

IF tracing DO
{ sawritef("T%z2 M%z2WC%z2: %cWC%z2S%z2M%z2C%z2:%z4*
           * Data pushed into channel buffer*n",
            taskid, mpxno, chnno, flag,
            clino, serno, mpxno, chnno, data)
  prbufs()
}

// Conditionally perform a channel delay
IF flag='m' DO
{ IF tracing DO
    sawritef("T%z2 M%z2WC%z2: %cWC%z2S%z2M%z2C%z2:%z4*
                 * Multiplexor delay*n",
                  taskid, mpxno, chnno, flag,
                  clino, serno, mpxno, chnno, data)
      c_delaylong := c_delaylong+1
      delay(delaymsecs)
      IF tracing DO
        sawritef("T%z2 M%z2WC%z2: %cWC%z2S%z2M%z2C%z2:%z4*
                 * Multiplexor delay done*n",
                  taskid, mpxno, chnno, flag,
                  clino, serno, mpxno, chnno, data)
    }

    // Return the successful write request packet to its server.
    IF tracing DO
    { sawritef("T%z2 M%z2WC%z2:*
               * %cWC%z2S%z2M%z2C%z2:%z4*
               * Returning successful write packet to its server*n",
                taskid, mpxno, chnno, flag,
                clino, serno, mpxno, chnno, data)
    }

    c_qpkt := c_qpkt+1
    qpkt(pkt)

    // Process the next write packet, if any.
  } REPEAT
}
```

Here it pushes its data value into the buffer and sets the `res1` to `TRUE` to indicate success. If `flag` is set to `m`, a multiplexor delay is required and this is performed by a call of `delay`. Finally the write packet is returned to its server before execution jumps back to the start of the main loop.

As a debugging aid the channel buffers can be output using `prbufs` whose definition is as follows.

```
AND prbufs() BE //IF tracing DO
{ FOR chnno = 1 TO chnmax DO
  { LET p = bufpv!chnno
    LET q = bufqv!chnno
    LET empty = p=q
    LET buf = bufv!chnno
    sawritef("    M%z2C%z2:  Buffer: ", mpxno, chnno)
    UNTIL p=q DO
    { LET val = buf!q
      sawritef("%z4 ", val) // data
      q := (q+1) MOD chnbufsize
    }
    sawritef("*n")

    IF rdwkqv!chnno DO
    { LET p = rdwkqv!chnno
      sawritef("   M%z2C%z2: %s:", mpxno, chnno, empty->"rdrq", "wrrq")
      WHILE p DO
      { LET type = p!pkt_type
        LET ch = type=act_read  -> 'R',
                 type=act_write -> 'W',
                 '?'
        sawritef(" %cS%z2", ch, p!pkt_arg3) // serno
        UNLESS empty DO sawritef("/%z4", p!pkt_arg6)
        p := !p
      }
      sawritef("*n")
    }
  }
}
```

Chapter 8

Miscellaneous Functions

This chapter covers various functions used by the benchmark program. These include `findpkt`, `sndpkt`, `sendpkt` and `gomultievent` used to implement multi-event tasks. Occam style channels are implemented using `cowrite` and `coread`. Locks are implemented by the functions `lock` and `unlock`. Condition variables are implemented using `condwait`, `notifyAll` and `notify`. Finally, a random number system is implemented by `nextrnd` and `setseed`.

8.1 Multi-event Tasks

Tasks in Cintpos often receive packets asking them to perform some operation before returning the packet to the client. These requests are usually performed one at a time, finishing each request before starting work on the next. Such tasks are said to run in single-event mode, but if servicing a request requires interaction with other tasks or devices, particularly if this many involve real time delays, it may be better to allow the task to service more than one request at the same time. This can be done by passing requests to worker coroutines. If one worker coroutine is held up for any reason, control can be passed to another worker to continue servicing a different request. A task organised in this way is said to run in multi-event mode. One way to setup a multi-event task is to call the function `gomultievent` which is described later. But first we will describe some of the functions it uses.

Multi-event tasks have worker coroutines to service requests. If a worker wishes to communicate with a device or another task it should call `sendpkt`. The arguments of this function hold the fields of the packet to be sent. The normal version of `sendpkt` send the packet using `qpkt` then waits for the packet to be returned. Its definition is as follows.

```
AND sendpkt(link,id,type,res1,res2,a1,a2,a3,a4,a5,a6) = VALOF
{ c_qpkt := c_qpkt+1
  UNLESS qpkt(@link) DO
  { sawritef("T%z2: System error in sendpkt, id=%n type=%n*n", taskid, id, type)
    abort(181)
    result2 := 0
    RESULTIS 0
  }

  c_taskwait := c_taskwait+1
  IF taskwait() = @link DO
  { result2 := res2
    RESULTIS res1
  }

  sawritef("T%z2: System error in sendpkt: Bad packet received*n", taskid)
  abort(182)
  result2 := 0
  RESULTIS 0
}
```

This version of sendpkt overrides the standard library version, the only difference being that it has to gather statistics about the usage of qpkt and taskwait. It fails with abort 181 if the call of qpkt fails or if the wrong packet is returned.

In multi-event mode, the packet sent by sendpkt need not be the next packet received by the task. It may, for instance, be a packet a different worker coroutine was expecting or it may be a new request packet. So a different version of sendpkt is required. This multi-event version is called sndpkt and its definition is as follows.

```
AND sndpkt(link, id, act, res1, res2, a1, a2, a3, a4, a5, a6) = VALOF
{ LET next, pkt, co = pktlist, @link, currco

  // Safety check -- sndpkt cannot be called from a root coroutine
  IF currco!co_parent<=0 DO
  { sawritef("T%z2 sndpkt currco=%n:*
             * Can't sndpkt %n(id=%n,type=%n) from root coroutine*n",
             taskid, currco, pkt, id, act)
    abort(999)
  }

  pktlist := @next    // Insert the node [next, pkt, co] at the
                      // head of pktlist.

  c_qpkt := c_qpkt+1
  UNLESS qpkt(pkt) DO // Send the packet.
  { sawritef("T%z2 sndpkt currco=%n:*
             * qpkt failure*n",
             taskid, currco)
    abort(181)
  }

  { LET p = ?
    c_cowait := c_cowait+1
    p := cowait()      // Wait for gomultievent to wake up this
                       // coroutine with the expected packet.
    IF p=pkt BREAK     // Check it was the right one.
    sawritef("T%z2 sndpkt currco=%n:*
             * Wrong packet %n received*n",
             taskid, currco, pkt)
    abort(182)
  } REPEAT

  result2 := res2
  RESULTIS res1
}
```

The arguments of sndpkt are the same as those of sendpkt being the laid out fields of a packet. The pointer to this packet (pkt) is therefore set to @link.

There is a list (pktlist) of packet-coroutine pairs that allows gomultievent to determine whether newly received packet belongs to a waiting a worker coroutine. Note that each node in the pktlist contains three fields [next,pkt,co] formed out of consecutive local variables. Having inserted this node at the head of pktlist, sndpkt sends the packet using qpkt and waits for its return by calling cowait. It only resumes control when gomultievent receives its expected packet.

The returned result is the res1 field of the packet and result2 is set to the res2 field.

The function findpkt, defined below, searches the list of packets belonging to worker coroutines. If a matching item is found, it is dequeued from pktlist returning the appropriate coroutine pointer. If no matching item is found, it returns zero. This function is only used by gomultievent.

```
AND findpkt(pkt) = VALOF
{ LET a = @pktlist

  { LET p = !a
    UNLESS p RESULTIS 0 // Matching item not found

    IF p!1 = pkt DO
    { !a := !p          // Remove the matching item from pktlist.
      RESULTIS p!2      // Return the coroutine pointer.
    }
    a := p
  } REPEAT
}
```

Note that the length of pktlist is limited to the number of worker coroutines controlled by gomultievent since each worker can only contribute at most one item in the list. If the length is large, pktlist could easily be replaced by a hash table, requiring only small modifications to findpkt, sndpkt and gomultievent. For this benchmark, a simple list is fine. If the average length of pktlist is substantially less than the number of worker coroutines, it may be worth allocating fewer workers.

We can now look at the definition of gomultievent which starts as follows.

```
AND gomultievent(maincofn, stacksize) BE
{ LET mainco = createco(maincofn, stacksize)
  LET oldsendpkt = sendpkt
  LET queue  = 0   // Queue of packets waiting to be processed by mainco.

  UNLESS mainco RETURN

  sendpkt, pktlist := sndpkt, 0 // Enter multi-event mode.

  multi_done   := FALSE // =TRUE when it is time to return to single
                        // event mode.
  mainco_ready := FALSE // =TRUE when mainco is ready to process
                        // packets. If FALSE, packets are appended
                        // to queue by gomultievent and given to
                        // mainco later.

  c_callco := c_callco+1
  callco(mainco, 0)     // Tell mainco to initialise itself
                        // by allocating workspace and creating
                        // coroutines as necessary.
```

This function takes two arguments maincofn, and stacksize that it uses to create the coroutine mainco which is then given control using callco allowing it to allocate work space and create worker coroutines. When this initialisation is complete, gomultievent enters its multi-event loop whose code is as follows.

```
  UNTIL multi_done DO      // Multi-event loop
  { LET pkt  = taskwait()
    LET cptr = findpkt(pkt)
    c_taskwait := c_taskwait+1
    TEST cptr
    THEN { // cprt is the multi-event coroutine waiting for this packet.
           c_callco := c_callco+1
           callco(cptr, pkt) // Give pkt to this coroutine.
         }
    ELSE { UNLESS mainco_ready DO
           { // This packet does not belong to a multi-event
             // coroutine and mainco is busy, so append it
             // onto the end of queue and wait for another
             // packet.
             // Note that this code is only obeyed if mainco
             // is suspended waiting for a particular packet,
             // as in a call of delay. In tcobench this never
             // happens.
             LET p = @queue
             WHILE !p DO p := !p
             !pkt, !p := 0, pkt
             LOOP
            }
            // This packet does not belong to a multi-event
            // coroutine and mainco is not busy, so give it
            // to mainco.
            c_callco := c_callco+1
            callco(mainco, pkt) // Give the packet to mainco.
          }

    // Try to deal with the packets in queue.
    WHILE queue & mainco_ready DO
    { pkt := queue        // De-queue a pending request packet.
      queue := !pkt
      !pkt := notinuse
      c_callco := c_callco+1
      callco(mainco, pkt) // Give it to mainco.
    }
  } // End of UNTIL multi_done loop.
```

Packets can be sent to a task using `qpkt` and these are normally placed in the `wkq` field of the task's control block. They can then be extracted from the queue one at a time by calls of `taskwait`. In a multi-event task, the call of `taskwait` occurs at the beginning of its event loop in `gomultievent`, placing the packet in `pkt`. The packet is then looked up using `findpkt` to see if it belongs to a worker coroutine. If it did, `findpkt` would have removed its entry in `pktlist` and returned a pointer to its coroutine which is placed in `cptr`. `gomultievent` then makes the call `callco(cptr,pkt)` to give the packet to its waiting worker.

If the packet does not belong to a waiting worker, it will be a request packet to be processed by mainco, but if mainco is currently busy (mainco_ready=FALSE) it will be appended to the end of the list queue. When mainco finishes processing a packet, it sets mainco_ready to TRUE and returns to gomultievent which immediately gives it the next request packet from queue, if there is one.

When multi_done becomes TRUE control will leave the event loop and fall into the following code.

```
  // All coroutines and work space created by mainco must have
  // been deleted.

  sendpkt := oldsendpkt // Return to single-event mode.

  deleteco(mainco)      // Delete the coroutine created
                        // by gomultievent.
}
```

This causes the task to return to single event mode before returning from gomultievent.

In process control applications, worker coroutines typically use little CPU time before suspending themselves. On the rare occasions where a worker requires substantial CPU time, it should repeated suspend itself to give other workers a chance to proceed. This can easily be done by sending a dummy packet to itself using sendpkt. Since qpkt appends its packet to the end of the task's work queue, earlier packets will be processed before this suspended worker resumes execution.

8.2 Occam Channels

Within the servers, communication between worker coroutines and the logger is done using Occam style channels. This is implemented by the functions coread and cowrite and these provide a good demonstration of the functions callco, cowait and resumeco. The definitions of coread and cowrite are as follows.

```
AND cowrite(ptr, data) BE
{ LET cptr = !ptr

  TEST cptr
  THEN { !ptr := 0
       }
  ELSE { !ptr := currco
         // Wait for the pointer to the reading coroutine
         c_cowait := c_cowait+1
         cptr := cowait()
```

```
        }

  // cptr points to the reading coroutine which is
  // guaranteed to be inactive.
  c_callco := c_callco+1
  callco(cptr, data) // Send the data to coread
}

AND coread(ptr) = VALOF
{ LET cptr = !ptr

  IF cptr DO
  { !ptr := 0         // Clear the channel word
    c_resumeco := c_resumeco+1
    RESULTIS resumeco(cptr, currco)
  }

  !ptr := currco      // Set channel word to this coroutine
  c_cowait := c_cowait+1
  RESULTIS cowait()   // Wait for value from cowrite
}
```

As can be seen, both definitions are short especially if we remove the code to increment the counters `s_callco`, `c_cowait` and `c_resumeco`, which are only present to allow `tcobench` to gather statistics. These functions are somewhat subtle and so they will be described in detail.

We will name the coroutine that makes the `coread` call: coroutine R, and name its parent: coroutine P. So coroutine R will have gained control by a call of `callco` within coroutine P, and if coroutine R now executes `cowait`, control will return to coroutine P. We will name the coroutine that makes the `cowrite` call: coroutine W. The first argument (`ptr`) of both `cowrite` and `coread` is a pointer to the channel word being used to synchronize the communication, and the second argument of `cowrite` holds the data to be transmitted to `coread`. When neither R nor W are ready to communicate, the channel word will be zero. The first coroutine wishing to communicate will find that `!ptr` is zero and will replace it by a pointer to its own coroutine. This is done by the assignment `!ptr:=currco`. It then calls `cowait` to wait for the other coroutine to reach the communication point. When the other coroutine is ready, it will discover that the channel word is non zero and so must point to the coroutine wishing to communicate with it. It places this other coroutine pointer in the variable `cptr`.

If this was done within `cowrite`, it will know that `cptr` points to coroutine R allowing it to immediately transfer the data by the call `callco(cptr,data)` causing `coread` in coroutine R to return with result `data`. The next time coroutine R suspends itself (possibly executing `cowait` in `coread`), control will return to coroutine W causing it to return from its call of `cowrite`.

If it was `coread` within coroutine R that discovered the channel word was non zero, it must give coroutine W the pointer to coroutine R. Unfortunately, if this was done by the call `callco(cptr,currco)`, it would leave coroutine R still active and `cowrite` would be unable to transfer the data it it. However, `coread` can, instead, use `resumeco(cptr,currco)`. This call is similar to `callco(cptr,currco)` but sets the parent field of coroutine W to P, instead of R, and, more crucially, it clears the parent field of coroutine R leaving it inactive, thus allowing `cowrite` to transfer the data by the call `callco(cptr,data)`.

Since `callco`, `cowait` and `resumeco` are all implemented efficiently, this implementation of Occam channels is efficient.

8.3 Locks

As we have seen, a worker coroutine within a `tcobench` server communicates with the logger using Occam style channels. Since the logger occasionally communicates with the printer task or perform real time delays, other coroutines within the server may gain control. Locks are thus necessary to ensure that one coroutine can complete its interractions with the logger without interference from others.

A lock is implemented by the functions `lock` and `unlock` defined below using a simple variable to represent the lock, such as `loggerlock`, normally held in the task's global vector. When unlocked, its value will be zero. If it is locked by just one coroutine and no other coroutines are waiting for it, its value will be `-1`, but if a coroutine has obtained the lock and others are waiting for it, the lock variable will the hold a list of waiting coroutines. Nodes in this list are of the form `[link, cptr]` where `link` is either zero or points to the next node, and `cptr` identifies a coroutine waiting for the lock. Note that these nodes are formed out of consecutive local variables.

The arguments to both `lock` and `unlock` are both pointers to lock variables, for example `lock(@loggerlock)` and `unlock(@loggerlock)`. The definition of `lock` is as follows.

```
AND lock(ptr) BE
{ c_lock := c_lock+1   // Gather locking statistics

  TEST !ptr
  THEN { // The lock was locked
         LET link, cptr = 0, currco // Create a lock node [link, cptr]
         TEST !ptr=-1
         THEN { !ptr := @link       // It was locked by just one
              }                     // coroutine, so make a unit list.
         ELSE { LET p = !ptr        // Othewise append the lock node
                WHILE !p DO p := !p // to the end of the list.
                !p := @link
              }
         c_lockw := c_lockw+1       // Gather locking statistics
         c_cowait := c_cowait+1
         cowait() // Suspend until released by unlock(..).
         // We now own the lock and !ptr will be non zero
       }
  ELSE { !ptr := -1 // Mark as locked by just one coroutine
       }
}
```

The definition of `unlock` is just as simple and is as follows.

```
AND unlock(ptr) BE
{ LET node = !ptr
  UNLESS node DO
  { sawritef("%z2 co %n:*
             * Attempt to free a lock that is not locked*n",
             taskid, currco)
    abort(999)
    RETURN
  }
  TEST node=-1
  THEN { !ptr := 0 // The lock was owned by only one
       }           // coroutine, so free it.
  ELSE { // Release a coroutine waiting on this lock
         LET next, cptr = node!0, node!1
         !ptr := next -> next, -1
         c_callco := c_callco+1
         callco(cptr)
       }
}
```

8.4 Condition Variables

Sometimes a program must wait until a potential complicated condition is satified. This facility is made available by the functions `condwait`, `notify` and `notifyAll`. A typical example is the following.

```
    UNTIL wrkcount <= minwrkcount+maxcountdiff | diepkt DO
      condwait(@countcondvar)
```

Every time the value of a variable changes that could cause the condition to be satified, either the call `notify(@countcondvar)` or `notifyAll(@countcondvar)` could be called to release one or more of the coroutines waiting on this condition. The `UNTIL` loop will immediately call `condwait` again if the condition is still not satisfied.

A condition variable, such as `countcondvar`, is implemented as a list of coroutines waiting on the condition. Each node in the list is of the form: `[next,cptr]` where `next` points to the next node in the list, and `cptr` points to a coroutine. The end of the list is marked by a null pointer. These nodes are formed out of consecutive local variables in `condwait`. The definition of `condwait`, below, is particularly simple, especially if we remove the code to implements the counters `c_condwait` and `c_cowait`.

```
AND condwait(condvarptr) BE
{ LET next, ptr = !condvarptr, currco // Form a node [next, cptr]
  c_condwait := c_condwait+1  // Gather statistics
  !condvarptr := @next
  c_cowait := c_cowait+1
  cowait()                    // Suspend until woken up by norify
                              // or notifyAll indicating that the
                              // waiting condition may now be
}                             // satisfied.
```

To wakeup all coroutines waiting on a specified condition we call `notifyAll`, defined below. Again this function can be even simpler if we remove the code to increment statistics counts.

```
AND notifyAll(condvarptr) BE
{ LET p = !condvarptr

  c_notifyAll := c_notifyAll+1

  !condvarptr := 0 // Clear the wait list since some of released
                   // coroutines may immediately wait on the
                   // same condition.

  WHILE p DO { // Give control, int turn, to every coroutine that
               // is waiting on this condition.
               LET cptr = p!1
               p := !p
               c_callco := c_callco+1
               callco(cptr)
             }
}
```

Sometimes we only wish to release one coroutine waiting on a condition and this is done using `notify`, defined as follows.

```
AND notify(condvarptr) BE
{ LET p = !condvarptr

  c_notify := c_notify+1

  UNLESS p RETURN // No coroutines waiting on this condition

  // There is at least one coroutine waiting on this condition.
  !condvarptr := !p  // Dequeue the first one.

  // Release the first waiting coroutine.
  c_callco := c_callco+1
  callco(p!1)
}
```

Note that using consecutive local variables to represent wait list nodes has the advantage that they are freed automatically at the appropriate moment.

8.5 Random Numbers

This benchmark program uses a language and machine independent random number generator based on a feedback shift register and modulo arithmetic. For each task, the current state is held in the global `seed` which can be set by calling `setseed`. The random numbers themselves are generated by calls of `nextrnd`. These two functions are defined as follows.

```
// Return a machine independent random number between 1 and max
AND nextrnd(max) = VALOF
{ MANIFEST {
    //feedback = #xD008    // 16 15 13 4  => period 2^16-1
    feedback = #x80200003  // 32 22  2 1  => period 2^32-1
    // #x80200003 = 1000 0000 0010 0000 0000 0000 0000 0011
    //             |          |                          ||
    //             |          |                          |1
    //             32         22                         2
  }
  TEST (seed&1)=0
  THEN seed := seed>>1
  ELSE seed := seed>>1 XOR feedback
  RESULTIS (seed>>1) MOD max + 1
}
```

```
AND setseed(a) BE
{ seed := a | 1 // Ensure that the seed is non zero.
  FOR i = 1 TO nextrnd(50) + 10 DO nextrnd(1000)
}
```

This random number generator is based on a 32-bit linear feedback shift register. Every time `nextrnd` is called, `seed` is shifted to the right by one position, and if `seed` was previously odd some of the bits of the new value of `seed` are complemented by the application of `XOR feedback`. Provided `feedback` is well chosen, `seed` will cycle through all $2^{32} - 1$ possible non zero values in seemingly random order. Of course, if `seed` starts at zero it remains at zero for ever. The following code near the beginning of the function `start` can be used to check how many calls of `nextrnd` are necessary to cause `seed` to return to one when started at one.

```
  IF FALSE DO // Comment out this line to test the nextrnd function.
  { LET i = 0
    // Find the period of the random number feedback shift register.
    // It should be 2^32-1 = #xFFFFFFFF.
    seed := 1
    { i := i+1
      nextrnd()
      IF seed=1 DO
      { sawritef("Random number cycle length = %n %x8*n", i, i)
        RESULTIS 0
      }
      IF (i & #xFFFFFF) = 0 DO
        sawritef("i = %x8*n", i)
    } REPEAT
  }
```

If you wish to learn more about linear feedback shift registers do a web search using the keys `linear feedback shift register`.

Appendix A

Java Translation

This is an early and incomplete draft of a Java translation of `tcobench.b`. I have included it mainly to allow me to look at it on my iPad.

```
//############# UNDER DEVELOPMENT ##################################
//############# NOT YET WORKING   ##################################

/*
This will be a Java translation from BCPL of the new version of the
Tcobench Benchmark.  See the file tcobench.pdf available via my home
page (www.cl.cam.ac.uk/users/mr10) for details.

Implemented  by Martin Richards (c) December 2015

Change History

01/10/15 Implementation started.


This Java version of the tcobench benchmark program attempts to follow
the logic of the BCPL version as closely as possible. It uses
coroutines implemented by callco, resumeco and cowait, defined as
methods in the abstract class Task. The functions qpkt and taskwait
are also defined in this class. Sub-classes of Task are use different
constructors to model both Tripos tasks and BCPL coroutines.

The class Globals holds environment values, such as climax and
tracing.  The variable g is present in every task and coroutine
allowing them to use expressions such as g.climax and g.tracing. Once
initialised these values remain constant and so can be accessed
without using synchronisation features. Declaring all variables in
Globals as volatile may be safer but less efficient.
```

Tasks and coroutines are not created using createtask or creatco as in the BCPL, but use new and start typically as follows.

```
  g.clock = new Clock("Clock", g, 10);
  g.clock.start();
```

or

```
  loggerco = new Loggerco("Logger", t);
  loggerco.parent = this;
  loggerco.value = null;
  t.currco = echoco;
  loggerco.start();
```

These both create and start Java threads. The run method defined in Task causes new tasks to be left waiting in taskwait for a startup packet, and non root coroutines to be left in the call of cowait in the coroutine main loop.

```
  while (!dying) c = fn(cowait(c));
```

For coroutines, t is the owning task. This is assigned to the instance variable t allowing convenient access to fields of the owning task such as t.wkq or t.currco. Such values are analogous to BCPL global variables of the owning task. Note that each task has its own work queue and current coroutine and these must be accessible to the methods qpkt, taskwait, callco, resumeco and cowait that are all method of class Task.

When a new coroutine is created t.currco is the creating coroutine and so is the initial parent of the newly created coroutine. Coroutine threads have the same priority as the owning task. The class representing a non-root coroutine should be a sub-class of class Task giving direct access to methods such as qpkt, taskwait, callco and cowait. They should not be sub-classes of the owning task's class.

The clock device is implemented as a high priority task that models the Tripos clock device as closely as possible. It uses a priority queue based on the heap structure used in heapsort.

The Java classes that model the Tripos tasks are: Controller, Clock, Stats, Bounce, Printer, Client, Server and Multiplexor. The classes modelling coroutines are: Workerco, Loggerco, Mpxrdco, Mpxwrco and Echoco. Coroutines belonging to a task can only be created by coroutines belonging to the same task. The root coroutine is created

```
when the task was created, and is initially suspended in taskwait()
waiting for its startup packet. All other coroutines belonging to a
task are suspended in callco, resumeco or cowait except for the
current coroutine whose thread is held in t.currco. The variable t
always points to the instance variables of the owning task.
*/

//*****************************************************************
//************************* Tcobench *******************************
//*****************************************************************

public class Tcobench implements Manifests {
    // This is the main class of the tcobench benchmark program.
    // Its static method main is the first to be entered.

    static String coName = "Tcobench";

    public static void main(String[] args)
    {
// Global variables accessible to all tasks and coroutines
// in the benchmark.
Globals g = new Globals();

g.startmsecs = System.currentTimeMillis(); // Needed by nowmsecs()

System.out.println("\n###### THIS PROGRAM IS UNDER DEVEOPMENT ######\n");

System.out.println("Thread and Coroutine Benchmark");
System.out.println("Implemented in Java using coroutines\n");

// Read the Tcobench parameters and place them in the instance of
// class Globals accessible via g.
g.rdargs(args);

// Create the Controller task
g.controller = new Controller("Controller", g, Thread.MAX_PRIORITY-1);
//if (g.tracing) System.out.println(coName+": Creating task "+g.controller.coName);

        //System.out.println(coName+": Giving the controller it startup pkt");

// We would like to perform:
//    qpkt(new Pkt(null, g.controller, act_startcontroller));
// but this cannot be done from a static context, so we do it
// by ad hoc means.
g.controller.wkq = new Pkt(null, g.controller, act_startcontroller);
g.controller.currco = null; // No valid current coroutine yet.
                                    // This will be the parent of the
                                    // Controller coroutine.
        //System.out.println(coName+": Starting task "+g.controller.coName);
g.controller.start();

// Note that unwait can be used to resume the execution of taskwait
// as well as suspended coroutines since it is just a synchronized
// method that calls notifyAll().

// The controller will find it has its startup packet in wkq
// and so will immediately return with it from its initial call of
// taskwait(). The startup packet will then be given to the
// controller's fn method. Returning from fn will cause run to
// return and this caused the controller to die.

//System.out.println(coName+": Wait for the Controller to finish\n");

        // Wait for the controller to finish
try { g.controller.join(); }
catch(Exception e) {}

System.out.println(coName+": Controller has finished");

System.out.println("Tcobench has finished");

        return;
    }

    public Object fn(Object x) {
System.out.println(coName+": fn called");
return null;
    }
}

//*****************************************************************
//*************************** Cortn *******************************
//*****************************************************************


abstract class Cortn extends Thread implements Manifests {
    // All sub-classes of Cortn must define its main function fn.

    // This is a simple class that just defines a few variables
    // that belong to coroutines.

    // All coroutines have names held in coName
    String coName;
    public Globals g;      // The global parameters, accessible to all
                           // coroutines.

    Task t;   // The task that owns this coroutine. It used
              // by methods such as callco, and cowait defined
              // in this class.
              // Note the r points to the same set of instance
              // variables but is declared in each coroutine class
              // with the appropriated type.

    // Each coroutine has a parent field which may be null.
    public Cortn parent; // This coroutine's parent.
    Object value;        // The value passed to this coroutine.
    boolean isroot;      // =true for root coroutines
    boolean dying;       // =true if this coroutine is dying.

    // Classes defining BCPL style tasks are sub-classes of Task
    // which is itself a sub-class of Cortn. The constructor
    // for such a class creates and instance with a given name,
    // a pointer to its globals, and its thread priority.
    // The resulting thread runs as the root coroutine of the task.

    // Non-root coroutines are implemented as sub-classes of Cortn
    // and have constructors that specified the coroutine name and
    // the pointer to the instance variables of the owning BCPL
    // style task.

    public Cortn(String name, Globals g, int pri) {
// This is used by the constructor in class Task when
// creating a Tripos style task.
        isroot = true;          // This is a non-root coroutine
coName = name;
setPriority(pri);       // Same priority as its owning task.
this.g = g;

// When the thread is started it enters run() defined in
// class Task, not the version of run() defined in this
// class.
    }

    public Cortn(String name, Task t) {
```

```
// This is used by the constructors in non-root coroutines.
// Task r is the owning task. The code that starts this
// coroutine sets the parent field and changes t.currco.
// It then calls start, immediately followed by currcowait().
// This should leave the coroutine suspended in cowait() of
// the coroutine's main loop (defined in run).

        isroot = false;         // This is a non-root coroutine
coName = name;
setPriority(t.pri);     // Same priority as its owning task.
this.g = t.g;
        this.t = t;             // Task t owns this coroutine. It is
                                // used by callco, cowait etc.

dying = false;
//System.out.println(coName+": Coroutine created t="+t);

// parent and t.currco are set by the calling coroutine.
    }

    public void run() {
System.out.println(coName+": Coroutine thread entered for the first time");
Object c = null; // Dummy value.

System.out.println(coName+": parent="+parent.coName);

// Enter the standard coroutine loop.
while (!dying) {
    //System.out.println(coName+": c="+((Integer)c).toString());
    System.out.println(coName+": Calling c=cowait(c)");
    c = cowait(c);
    System.out.println(coName+": Calling c=fn(c)");
    c = fn(c);
    //System.out.println(coName+": fn returned "+c);
}

System.out.println(coName+": Dying");
return;
    }

    abstract public Object fn(Object x);

    // Conventional coroutine API

    // Their implementation is subtle so some explanation is given here.
    // BCPL style coroutines each have their own runtime stack and so
    // in Java they must be threads. Every coroutine has a corresponding
    // a class. Those that define BCPL style tasks are sub-classes
    // of the class Task which in turn is a sub-class of Cortn. When such
    // a task starts it runs are the root coroutine and has direct access
    // all the Tripos style variables such as wkq, currco and parent,
    // and can also call qpkt, taskwait, callco, resumeco and cowait
    // directly. Classes for non-root coroutines are not sub-classes of
    // Task but are just sub-classes of Cortn. They can access the
    // variables of the owning task indirectly using t of type Task.
    // For specific Tcobench coroutines such as workerco there is another
    // pointer r declared with an appropriate type in its own coroutine
    // class. For instance, in class Workerco there is the declaration
    // Server r. This is initialised to point to the same set of
    // instance variables as t but has a type that allows access to
    // variables of the BCPL style task that owns the coroutine. So, for
    // example, a worker can access its server number using r.srvno.
    // Within a non-root coroutine functions such as qpkt and callco can
    // be accessed using t, as in t.qpkt. The same function can also be
    // accessed by r.qpkt, but this is not recommended since qpkt is
    // a method defined in Task.

    // For convenience, every coroutine has a variable g giving access
    // to all the Tcobench parameters such as g.climax and g.tracing.
    // Pointers to all the Tcobench tasks are held in the Globals class.
    // So, for instances, the instance variables of the Stats task are
    // accessible to any coroutine using g.stats.

    // Only one of the coroutines of a Tripos task can be executing at
    // any one time. All the others will be in the body of the
    // synchronized method currcoWait(), probably suspended in the
    // call wait() waiting for currco==this to become true. Note that
    // currcowait and unwait are declared in class Cortn.

    // When a coroutine transfers control to another coroutine
    // target.parent is set appropriately and t.currco is set to the
    // target coroutine. The value to be passed to the target is set
    // in target.value. These assignments are performed by callco,
    // as defined below. It then releases the target thread by the call
    // of notifyAll() in target.unwait(). Since unwait is a synchronized
    // method the variables parent, currco and value will have been
    // flushed to their associated memory locations, ready to be read
    // by the target coroutine.
```

```
    public Object callco(Cortn target, Object arg) {
// This will simultaneously wakeup the target coroutine
// and suspend the current coroutine.
// Note that there is no need for this method to be
// synchronized, since no other coroutine belonging to
// the current task will be running, so parent, value and
// currco will not be changed by other threads.
target.parent = this;
target.value = arg;
//System.out.println(coName+" callco t="+t);
t.currco = target;

//System.out.println(coName+
//    ": callco -- target "+target.coName+
//    " target.parent="+target.parent.coName);

// Transfer control to the target coroutine.
// unwait is synchronized so parent, value and currco wiil
// be written out to main memory.
target.unwait();

// Wait until we are again the current coroutine,
// ie until t.currco==this.
currcoWait();

//System.out.println(coName+
//    ": callco -- Resumed execution with value "+value);
return value;
    }

    public Object resumeco(Cortn target, Object arg) {
Cortn p = parent;   // Ensure that resumeco can resume itself
parent = null;     // ie when target==currco
target.parent = p;
target.value = arg;
t.currco = target;

//System.out.println(coName+": resumeco called => "+target.coName);

// Transfer control to the target coroutine.
target.unwait();

// Wait until we are again the current coroutine,
// ie until t.currco==this.
currcoWait();
```

```
return value;
    }

    public Object cowait(Object arg) {
if ( t.currco.parent==null)
    System.out.println(coName+
       ": ERROR: current parent is null ################\n");

Cortn from = t.currco;
Cortn to = from.parent;

//System.out.println(coName+": from="+from.coName);
//System.out.println(coName+": Setting to="+to.coName);
//System.out.println(coName+": Setting from.parent=null");
//System.out.println(coName+": Setting r="+to.coName);
//System.out.println(coName+": Setting to.value="+arg);
//System.out.println(coName+": Setting t.currco="+to.coName);
//System.out.println(coName+": Calling to.unwait()");

from.parent = null;
to.value = arg;
t.currco = to;

//System.out.println(coName+": cowait giving control to "+target.coName);

// Transfer control to the target coroutine.
to.unwait();

//System.out.println(coName+": cowait calling from.currcoWait()");

// The from coroutine will now wait until it becomes the
// current coroutine.
from.currcoWait();

return value;
    }

    // Internal support functions, currcowait and unwait, which must be
    // in this class since they synchronize on the coroutine thread.

    public synchronized void currcoWait() {
try {
    while (t.currco!=this) {
//System.out.println(coName+
//    ": currcoWait -- r.currco!=this, so call wait()");
wait();
    }
    //System.out.println(coName+
    //        ": currcoWait -- r.currco==this, so return");
}
catch(Exception e) {}
    }

    public synchronized void unwait() {
// Release all this object's threads.
// Note that there may be threads other than the waiting
// coroutine, so notify() alone is not sufficient.
// If in doubt, try it.
//System.out.println(coName+
//    ": unwait -- calling notifyAll");
notifyAll();
    }

    public void delayco(int msecs) {
t.sndpkt(new Pkt(null, g.clock, act_clock, 0, 0, msecs));
    }

    public static String strz(int n, int d) {
String res = "";
        if(n<0) {
    d--;
            n = -n;
    res = res + "-";
}
        if (n<10000000 && d>=8) res = res + "0";
        if (n<1000000 && d>=7) res = res + "0";
        if (n<100000 && d>=6) res = res + "0";
        if (n<10000 && d>=5) res = res + "0";
        if (n<1000 && d>=4) res = res + "0";
        if (n<100 && d>=3) res = res + "0";
        if (n<10 && d>=2) res = res + "0";
return res + n;
    }

    public static String strn(int n, int d) {
String res = "";
        if(n<0) d--;
        if (n<10000000 && d>=8) res = res + " ";
        if (n<1000000 && d>=7) res = res + " ";
        if (n<100000 && d>=6) res = res + " ";
        if (n<10000 && d>=5) res = res + " ";
        if (n<1000 && d>=4) res = res + " ";
        if (n<100 && d>=3) res = res + " ";
        if (n<10 && d>=2) res = res + " ";
return res + n;
    }

    public static void wrz(int n, int d) {
        if(n<0) {
    d--;
            n = -n;
            System.out.print("-");
}
        if (n<10000000 && d>=8) System.out.print("0");
        if (n<1000000 && d>=7) System.out.print("0");
        if (n<100000 && d>=6) System.out.print("0");
        if (n<10000 && d>=5) System.out.print("0");
        if (n<1000 && d>=4) System.out.print("0");
        if (n<100 && d>=3) System.out.print("0");
        if (n<10 && d>=2) System.out.print("0");
        System.out.print(""+n);
    }

    public static void wrn(int n, int d) {
        if(n<0) d--;
        if (n<10000000 && d>=8) System.out.print(" ");
        if (n<1000000 && d>=7) System.out.print(" ");
        if (n<100000 && d>=6) System.out.print(" ");
        if (n<10000 && d>=5) System.out.print(" ");
        if (n<1000 && d>=4) System.out.print(" ");
        if (n<100 && d>=3) System.out.print(" ");
        if (n<10 && d>=2) System.out.print(" ");
        System.out.print(""+n);
    }

    public static String reqstr(char flag, char modech,
 int srvno, int mpxno, int chnno, int data) {
// Return a string such as " nRS02M04C01: 7845"
return " "+flag+modech+
    "S"+strz(srvno,2)+"M"+strz(mpxno,2)+
    "C"+strz(chnno,2)+":"+strz(data,4);
    }

    public static String reqstr(char flag, char modech,
int clino, int srvno, int mpxno, int chnno, int data) {
// Return a string such as "RC02 nRS02M04C01: 7845"
return ""+modech+"C"+strz(clino,2)+
               " "+flag+modech+
       "S"+strz(srvno,2)+"M"+strz(mpxno,2)+
       "C"+strz(chnno,2)+":"+strz(data,4);
    }


    public int nowmsecs() {
// Return msecs since start of the current run.
return (int)(System.currentTimeMillis()-g.startmsecs);
    }

    public void wrtime(String mess) {
java.util.Date date = new java.util.Date();
System.out.println(mess+"  "+date);
    }

    public void realdelay(int msecs) {
try {
    sleep(msecs);
} catch(Exception e) {}
    }
}

//*****************************************************************
//*************************** Task ********************************
//*****************************************************************


abstract class Task extends Cortn {
    // All sub-classes of Task must define the task's or
    // coroutine's main function fn.

    // This class provides all the usual functionality of Tripos,
    // including qpkt, taskwait and the coroutine functions callco,
    // resumeco and cowait, plus many others such as wrx and wrn.

    // Its sub-classes are either tasks or coroutines depending on
    // which constructor is used. Each task has a root coroutine and
    // possibly many non-root coroutines that all run with the same
    // priority as the root.

    // A non-root coroutine can access the variables of the task
    // to which it belongs using expressions such as t.currco.

    // The following variables are directly accessible to all Tcobench tasks
    // and are accessible via t by all coroutines.

    // Variables, such as clinno, that particular to individual Tcobench
    // tasks are declared in their own classes and are accessible via r,
    // declared in each Tcobench class that needs it.

    public int pri;        // The priority of the task and all its coroutines
    public int result2=0;

    // Variables used by gomultievent.
    // They are declared in Task since they are shared by the coroutines
    // belonging to a task. For convenience, gomultievent is declared in
    // class Cortn so is directly accessible to all coroutines.
    public boolean mainco_ready;
    public boolean multi_done;
    Pktlist pktlist = null;   // List of packet-coroutine pairs.


    // There is one work queue shared by all coroutines belonging to
    // each Tripos style class.
    public Pkt wkq=null;   // Used if this thread models a task


    // There is one current coroutine per Tripos style class. It is
    // shared by all coroutines belonging to the task. For convenience
    // callco, resumeco and cowait are declared in class Cortn so can
    // be accessed directly from anywhere.
    public Cortn currco;        // The current coroutine of this task,
                                // normally accessed by t.currco.
                                // Each task has a current coroutine
                                // so it is declared in class Task
                                // and not class Cortn.
                                // Within a task only one coroutine
                                // (the current coroutine) can have
                                // control. All the others will be
                                // suspended in callco, resumeco or
                                // cowait.
                                // Note that parent is declared in
                                // class Cortn.

    // The root coroutine of a task is created when the task is created.
    // All other coroutines belonging to a task are created by coroutines
```

```
    // belonging to that task.

    public Task(String name, Globals g, int pri) {
// This contructor is called by a subclass of Task to
// create a new task thread and its root coroutine.
// When this thread is started it will suspend itself
// in taskwait() waiting for the task's first packet
// which it will then give to fn. On return from fn
// the task will die.
super(name, g, pri);
this.pri = pri;   // Remember the priority for use by
                          // coroutines belonging to this task.
// This is a Tripos style task and its root coroutine.
// We must set t to point to itself since it is needed
// by callco, etc which are defined in class Cortn.
t = this;

// parent and currco are set by the coroutine that starts this
// thread.

//System.out.println(coName+": Task and root coroutine created");
//System.out.println(coName+": but not yet started");
    }

    public abstract Object fn(Object arg);
    // fn is the body of either the root coroutine of a task or
    // the body of a non root coroutine belonging to a task.

    // Normally run will be defined in all the sub-classes of Task.

    public void run() {
        // This is called when the thread first starts. Returning
        // from run causes the thread to die.

//System.out.println(coName+": Thread started");
//if (parent!=null)
//    System.out.println(coName+": parent="+parent.coName);

//System.out.println(coName+": Root coroutine thread started");
// This thread models the root coroutine of a Tripos task.

// Wait for the startup packet then give it to fn.

//System.out.println(coName+": Calling taskwait() for its startup packet");
Pkt startuppkt = taskwait();
```

```
//System.out.print(coName+": Received startup pkt from "+startuppkt.task.coName);
//System.out.println(coName+": New task calling fn(startuppkt)");
fn(startuppkt);
// Returned from fn causes the thread to die.
//System.out.println(coName+": This task returned from fn and is now dying");
    }

    // Locking on a Tripos style task allows qpkt and taskwait
    // exclusive access the task’s work queue wkq. A task should
    // only modify the fields of a packet when it owns the packet.
    // It does not own the packet after calling qpkt until it is
    // returned as the result of taskwait. qpkt and taskwait are
    // not synchronized and are declared in class Cortn, but putpkt
    // is synchronized on its task object so is delared in class
    // Task.

    public synchronized boolean putpkt(Pkt pkt, Task from) {
//System.out.println("\n"+coName+": Got lock on this thread");

// This runs in the thread corresponding to the current task,
// but has a lock on the destination task’s thread and so
// can modify the detination task’s wkq without interference
// from other tasks.

// This is the normal version of putpkt, but for the Clock task
// it is overridden by a special one that processes packets in
// a different way.

// Note that wkq is the work queue of the destination task.
// If wkq was previously null, a call of notifyAll is required
// since the task may be suspended in wait() in the body of
// taskwait.

String fromName = from==null ? "null"
                                    : from.coName; // Name of the task

//System.out.println(coName+": putpkt type="+pkt.type+" arg1="+pkt.arg1);
//System.out.println(coName+
//    ": putpkt appending pkt from "+fromName+" this task’s wkq");

pkt.task = from; // Fill in the return task field.
pkt.link = null;

// Append the packet onto the end of the destination task’s wkq.
```

```
if (wkq==null) {
    //System.out.println(coName+": putpkt wkq was null");
    wkq = pkt; // Make a unit list.
    //System.out.println(coName+": putpkt calling notifyAll()"+
    //    " since wkq was previously null");
    notifyAll();
} else {
    //System.out.println(coName+
    //        ": appending pkt to end of non null wkq");
    Pkt p = wkq;
    // Find the end of the work queue.
    while(p.link!=null) {
    // System.out.println(coName+
    //    ": Skip over wkq pkt from "+p.task.coName);
p = p.link;
    }
    p.link = pkt; // Place the packet at the end.
}
//System.out.println(coName+": Releasing the lock on this thread\n");
return true;
    }

    public boolean qpkt(Pkt pkt) {
//System.out.println(coName+
        //                   ": qpkt to "+pkt.task.coName+
        //                   " type "+pkt.type);
// putpkt is a synchronized method that gains the lock on
// the target thread.
return pkt.task.putpkt(pkt, this);
    }

    public synchronized Pkt taskwait() {
// This must be synchronized on the BCPL style task object since
// it contains wait() and also manipulates wkq.

//System.out.println("\n"+coName+": Got lock on this thread");
// No other threads will be currently manipulating this task's wkq.

try {
    while (wkq==null) {
// We need the while loop since spurious wakeups might happen.
           //System.out.println(coName+
//    ": taskwait: calling wait() since wkq is null");
wait();
           //System.out.println(coName+
//    ": taskwait: woken up");
    }
    //System.out.println(coName+
    //          ": taskwait: wkq non empty");
}
catch(Exception e) {
    System.out.println(coName+
       ": taskwait: exception e="+e.toString());
}

// There is as least one packet in wkq.
Pkt pkt = wkq;
wkq = wkq.link;
//if (pkt.task!=null)
    //System.out.println(coName+
    //        ": taskwait() received pkt from "+pkt.task.coName+
    //        " type="+pkt.type);

//System.out.println(coName+": Released lock on this thread\n");
return pkt;
    }

    public int sendpkt(Pkt pkt) {
//System.out.println(coName+": sendpkt calling qpkt\n");
//System.out.println(coName+": dest="+pkt.task.coName);
//System.out.println(coName+": currco="+t.currco);
//System.out.println(coName+": currco="+t.currco.coName);
qpkt(pkt);
Pkt p = taskwait();
if (p!=pkt) {
    System.out.println(coName+": System error in sendpkt");
    result2 = 0;
    return 0;
}
result2 = pkt.res2;
return pkt.res1;
    }

    public int sndpkt(Pkt pkt) {

if (isroot) {
    System.out.println(coName+": Can't sndpkt from a root coroutine");

}
if (!qpkt(pkt)) {
    System.out.println(coName+": sndpkt  -- qpkt failure");

}
while (true) {
    Pkt p = (Pkt)cowait(null);
    if (p==pkt) {
System.out.println(coName+": sndpkt received the wrong pkt");
continue;
    }
    break;
}
result2 = pkt.res2;
return pkt.res1;
    }

    public Cortn findpkt(Pkt pkt) {
Pktlist p = pktlist;

if (p==null) return null; // pktlist was empty

if (p.pkt==pkt) {
    // The first item in pktlist matches.
    Cortn cptr = p.cptr;
    // Dequeue the first element of pktlist
    pktlist = p.link;
    return cptr;
}

while (true) {
    Pktlist q = p.link;

    if (q==null) return null; // No matching item found.

    // Test an item q in pktlist other than the first.
    if (q.pkt == pkt) {
Cortn cptr = q.cptr;
// Remove item q from pktlist
p.link = q.link;
return cptr;
    }
    p = q;
}
    }

    public void gomultievent(Cortn mainco) {
```

```
Pkt queue = null;

//System.out.println(coName+": gomultievent entered");

// Create an empty pktlist.
pktlist = null;
multi_done = false;   // =true when it is time to return
                              // to single event mode.

mainco_ready = true;  // =true when mainco is ready to process
                              // packets. If false, packets are appended
                              // to queue by gomultievent and given to
                              // mainco later.

//System.out.println(coName+": gomultievent calling callco(mainco,null)");
callco(mainco, null); // Startup servermainco
//System.out.println(coName+": returned from callco(mainco,null)");

// mainco has completed its initialisation.
//System.out.println(coName+": gomultievent entering its multi-event loop");

while( !multi_done) {
    // Start of the multi-event loop.
    System.out.println(coName+": gomultievent calling taskwait()");
    Pkt pkt = taskwait();
    System.out.println(coName+": gomultievent got pkt from "+
            pkt.task.coName);
    Cortn cptr = findpkt(pkt);

    if (cptr!=null) {
// cptr is the multi-event coroutine waiting for this
// packet.
System.out.println(coName+": gomultievent giving pkt to "+cptr.coName);
callco(cptr, pkt); // Give pkt to this coroutine.
    } else {
if (!mainco_ready) {
    // This packet does not belong to a multi-event
    // coroutine and mainco is busy, so append it
    // onto the end of queue and wait for another
    // packet.
    // Note that this code is only obeyed is mainco
    // is suspended waiting for a particular packet.
    // as in a call of delayco(). In Tcobench this
    // never happens.
    pkt.link = null;
    if (queue==null) {
queue = pkt;
    } else {
Pkt p = queue;
while (p.link!=null) p = p.link;
p.link = pkt;
    }
    System.out.println(coName+": gomultievent put pkt in queue");
    continue;
}

// This packet does not belong to a multi-event
// coroutine and mainco is not busy, so give it
// to mainco.
System.out.println(coName+": gomultievent calling callco(mainco, pkt)");
callco(mainco, pkt);
System.out.println(coName+": gomultievent mainco processed pkt");
    }

    // try to deal with the packets in queue.
    while (queue!=null && mainco_ready) {
Pkt p = queue;
queue = queue.link;
p.link = null;
System.out.println(coName+": gomultievent dequeued pkt from queue");
System.out.println(coName+":     then calling callco(mainco, p)");
callco(mainco, p); // Give it to mainco.
    }
} // End of while (!multi_done) loop.
    }

    public void delaytask(int msecs) {
sendpkt(new Pkt(null, g.clock, act_clock, 0, 0, msecs));
    }
}


//*******************************************************************
//************************* Manifests *******************************
//*******************************************************************

interface Manifests {
    // Packet types
    public final static int act_startcontroller = 100;
    public final static int act_startclock      = 101;
    public final static int act_startstats      = 102;
    public final static int act_startbounce     = 103;
    public final static int act_startrdclient   = 104;
    public final static int act_startwrclient   = 105;
    public final static int act_startrdserver   = 106;
    public final static int act_startwrserver   = 107;
    public final static int act_startmpx        = 108;
    public final static int act_startprinter    = 110;
    public final static int act_clock           = 111;
    public final static int act_bounce          = 112;
    public final static int act_print           = 113;
    public final static int act_read            = 114;
    public final static int act_write           = 115;
    public final static int act_sync            = 116;
    public final static int act_rddone          = 117;
    public final static int act_wrdone          = 118;
    public final static int act_die             = 119;

    public final static int act_calibrate       = 120;
    public final static int act_run             = 121;

    public final static int act_addstats        = 122;
    public final static int act_prstats         = 123;

    public final static int act_quickbounce     = 124;
}

//*********************************************************************
//*************************** Globals *********************************
//*********************************************************************

class Globals {
    // Global parameters and constants accessible to every coroutine of
    // every task.

    // There is one instance of this class and every task can refer to it
    // using g. So the global (shared) variables are normally accessed by
    // g.<name> as in g.loopmax

    // Global variables that change dynamically must be updated using
    // synchronised methods.

    public int     loopmax;
    public int     climax;
    public int     srvmax;
    public int     workmax;
    public int     mpxmax;
    public int     chnmax;
    public int     chnbufsize;
    public int     delaymsecs;
    public int     delayticks;
    public int     maxcountdiff;

    public int     requestvupb;

    public boolean tracing = true;

    long startmsecs = 0;

    // Global tasks
    public Controller controller;
    public Clock clock;
    public Stats stats;
    public Bounce bounce;
    public Printer printer;

    public Client rdclientv[];
    public Client wrclientv[];
    public Server rdserverv[];
    public Server wrserverv[];
    public Multiplexor mpxv[];


    public Globals() {};

    public void rdargs(String[] args) {
try {
    // Default settings
    loopmax    = 2;
    climax     = 20;
    srvmax     = 15;
    workmax    = 14;
    mpxmax     = 10;
    chnmax     = 10;
    delaymsecs = 500;

            maxcountdiff = 5;
    tracing    = false;

    for (int i = 0; i < args.length; i++) {
        //System.out.println("arg["+i+"] = " + args[i]);

if (args[i].equals("-t")) {
    tracing = true;
    continue;
}

if (args[i].equals("-x")) {
                    // Alternate setting of the parameters
    loopmax    =   1;
    climax     =   2;
    srvmax     =   2;
    workmax    =   3;
    mpxmax     =   2;
    chnmax     =   3;

    loopmax    =   1;
    climax     =   1;
    srvmax     =   1;
    workmax    =   3;
    mpxmax     =   1;
    chnmax     =   1;
    continue;
}

if (args[i].equals("-y")) {
                    // Alternate setting of the parameters
    loopmax    =   2;
    climax     =   5;
    srvmax     =   3;
    workmax    =   3;
    mpxmax     =   2;
    chnmax     =   3;
    continue;
}

if (args[i].equals("-z")) {
                    // Alternate setting of the parameters
    loopmax    =   3;
    climax     =  10;
    srvmax     =   4;
    workmax    =   7;
    mpxmax     =   3;
    chnmax     =   4;
    continue;
}

if (args[i].equals("-k")) {
    loopmax = Integer.parseInt(args[++i]);
    continue;
}

if (args[i].equals("-n")) {
    climax = Integer.parseInt(args[++i]);
    continue;
}

if (args[i].equals("-s")) {
    srvmax = Integer.parseInt(args[++i]);
    continue;
}

if (args[i].equals("-w")) {
    workmax = Integer.parseInt(args[++i]);
    continue;
}

if (args[i].equals("-m")) {
    mpxmax = Integer.parseInt(args[++i]);
    continue;
}

if (args[i].equals("-c")) {
    chnmax = Integer.parseInt(args[++i]);
    continue;
}

if (args[i].equals("-b")) {
    chnbufsize = Integer.parseInt(args[++i]);
    continue;
}

if (args[i].equals("-d")) {
    delaymsecs = Integer.parseInt(args[++i]);
    continue;
}
```

```
    }
}
catch (NumberFormatException e) {
    System.err.println("Integer argument expected");
}

requestvupb = srvmax*mpxmax*chnmax;

        if(chnbufsize==0) {
    // Set the channel buffer size to about one tenth of the
            // number of values sent to each channel on each iteration
            chnbufsize = (climax*srvmax)/10 + 5;
        }

System.out.println("loopmax    = "+Task.strn(loopmax,4)+" (k)");
System.out.println("climax     = "+Task.strn(climax,4)+" (n)");
System.out.println("srvmax     = "+Task.strn(srvmax,4)+" (s)");
System.out.println("workmax    = "+Task.strn(workmax,4)+" (w)");
System.out.println("mpxmax     = "+Task.strn(mpxmax,4)+" (m)");
System.out.println("chnmax     = "+Task.strn(chnmax,4)+" (c)");
System.out.println("chnbufsize = "+Task.strn(chnbufsize,4)+" (b)");
System.out.println("delaymsecs = "+Task.strn(delaymsecs,4)+" (d)\n");

System.out.println("Requests per schedule = "+Task.strn(requestvupb,4));
System.out.println("maxcountdiff          = "+Task.strn(maxcountdiff,4)+"\n");
    }
}

//*******************************************************************
//************************** Controller *****************************
//*******************************************************************

class Controller extends Task {
    ///task
    // This models the Controller task

    Controller r;

    public Controller(String name, Globals g, int pri) {
// This constructor is used to create the Controller task
// and its root coroutine. The Controller task has no
// other coroutines.
super(name, g, pri);
r = this;
        System.out.println(coName+": Task created");
    }

    public Object fn(Object arg) {
// This is the main function of the root coroutine of
        // the Controller.
// arg is the startup packet and it will not be returned to
// the main program.

Pkt startuppkt = (Pkt)arg;
//System.out.println(coName+": Startup pkt received,");

// Create and start all the tcobench tasks but do not send them
        // startup packets yet.

// Allocate the vectors
g.rdclientv = new Client[g.climax+1];
g.wrclientv = new Client[g.climax+1];
g.rdserverv = new Server[g.srvmax+1];
g.wrserverv = new Server[g.srvmax+1];
g.mpxv      = new Multiplexor[g.mpxmax+1];

//System.out.println(coName+": Creating Clock task");
g.clock = new Clock("Clock", g, 10);
//System.out.println(coName+": Starting task "+g.clock.coName);
        g.clock.start();

//realdelay(2000);

// Send a startup packet to the Clock task that models the
// Tripos Clock device. It should now be suspended in taskwait()
// waiting for its startup packet.

//System.out.println(coName+": Sending startup pkt to task "+g.clock.coName);

sendpkt(new Pkt(null, g.clock, act_startclock));

//System.out.println(coName+": The clock startup packet has been returned\n");

if (true) { // Change to if (true) to test the clock task.
    // Test the clock device
    Rnd r = new Rnd(500);

    System.out.println("\n"+coName+": Testing the Clock task\n");

    for(int i = 1; i<=20; i++) {
```

```
int k1 = 7;
int k2 = 20-k1+1;
if (i<=k2) {
    int delay = 200 + r.next(1000);
    qpkt(new Pkt(null, g.clock, act_clock,
 0, 0,
 delay, i));
    wrn(nowmsecs(), 5);
    System.out.print(" "); wrn(i, 2);
    System.out.print(": delay for "); wrn(delay, 5);
    System.out.print(" until "); wrn(delay+nowmsecs(), 5);
    System.out.print(" sent\n");
}
if (i>=k1) {
    Pkt p = taskwait();
    int delay = p.arg1;
    wrn(nowmsecs(), 5);
    System.out.print(" "); wrn(p.arg2, 2);
    System.out.print(": delay="); wrn(delay,5);
    System.out.print(" done\n");
}
    }

    System.out.println("\nEnd of Clock test\n");

    //System.out.println("Sleeping for 1 second\n");
    //realdelay(1000);

} // End of while(true) loop

//System.out.println(coName+": Creating the Stats task");
g.stats = new Stats("Stats", g, 9);
        g.stats.start();

//System.out.println("\n"+coName+": Creating the other Tcobench tasks\n");

//System.out.println(coName+": Creating and starting the Bounce task");
g.bounce = new Bounce("Bounce", g, 1);
        g.bounce.start();

//System.out.println(coName+": Creating and starting the Printer task");
g.printer = new Printer("Printer", g, 2);
        g.printer.start();

for (int i=1; i<=g.climax; i++) {
    g.rdclientv[i] = new Client("RC"+strz(i,2), g, 3);
    g.rdclientv[i].start();
}
for (int i=1; i<=g.climax; i++) {
    g.wrclientv[i] = new Client("WC"+strz(i,2), g, 3);
    g.wrclientv[i].start();
}

for (int i=1; i<=g.srvmax; i++) {
    g.rdserverv[i] = new Server("RS"+strz(i,2), g, 4);
    g.rdserverv[i].start();
}
for (int i=1; i<=g.srvmax; i++) {
    g.wrserverv[i] = new Server("WS"+strz(i,2), g, 4);
    g.wrserverv[i].start();
}

for (int i=1; i<=g.mpxmax; i++) {
    g.mpxv[i] = new Multiplexor("M"+strz(i,2), g, 5);
    g.mpxv[i].start();
}

// Send startup packets to all the tcobench tasks.

// Send a startup packet to the Stats task.
// This must not happen until the Clock and Bounce tasks have been
// created but not yet given their startup packets.

//System.out.println("\n"+coName+": Sending startup pkt to the Stats task");
sendpkt(new Pkt(null, g.stats, act_startstats));
//System.out.println(coName+": The Stats startup packet has been returned\n");

//System.out.println(coName+": Calling g.stats.incallco()");
g.stats.inccallco();

// Send a startup packet to the Bounce task.

System.out.println("\n"+coName+": All Tcobench tasks have been created\n");

System.out.println(coName+
   ": Sending startup packets to all remaining Tcobench tasks\n");

//System.out.println(coName+": Sending startup pkt to the Bounce task");
sendpkt(new Pkt(null, g.bounce, act_startbounce));
//System.out.println(coName+": The Bounce startup packet has been returned\n");
```

```
//System.out.println(coName+": Sending startup pkt to the Printer task");
sendpkt(new Pkt(null, g.printer, act_startprinter));
//System.out.println(coName+": The Printer startup packet has been returned\n");

for (int mpxno=1; mpxno<=g.mpxmax; mpxno++) {
    sendpkt(new Pkt(null, g.mpxv[mpxno], act_startmpx, 0, 0, mpxno));
}

for (int svrno=1; svrno<=g.srvmax; svrno++) {
    sendpkt(new Pkt(null, g.rdserverv[svrno], act_startrdserver, 0, 0, svrno));
}
for (int svrno=1; svrno<=g.srvmax; svrno++) {
    sendpkt(new Pkt(null, g.wrserverv[svrno], act_startwrserver, 0, 0, svrno));
}

for (int clino=1; clino<=g.climax; clino++) {
    sendpkt(new Pkt(null, g.rdclientv[clino], act_startrdclient, 0, 0, clino));
}
for (int clino=1; clino<=g.climax; clino++) {
    sendpkt(new Pkt(null, g.wrclientv[clino], act_startwrclient, 0, 0, clino));
}

//System.out.println("Sleeping for 1 second\n");
//realdelay(1000);

if (true) {
    Pkt bpkt = new Pkt(null, g.bounce, act_quickbounce);
    int startmsecs = nowmsecs();
    int cycles = 10000;
    //System.out.println(coName+": Test qpkt-taskwait bounce rate, cycles="+cycles+"\n
    for (int i=1; i<=cycles; i++) {
//System.out.println(i+" Sending a bounce packet");
qpkt(bpkt);
//System.out.println(i+" Wait for reply");
taskwait();
    }
    int diff = nowmsecs() - startmsecs;
    System.out.println("\n"+coName+": "+
       (cycles*1000/diff)+" qpkt-taskwait bounces per second\n");
}

//System.out.println("\n"+coName+
//    ": Sending a calibrate packet to the Stats task\n");
sendpkt(new Pkt(null, g.stats, act_calibrate));
```

```
//System.out.println(coName+": Calibrate packet returned from Stats task\n");

System.out.println(coName+": Start  time = "+nowmsecs()+" msecs\n");


System.out.println("\n"+coName+": Sending run packet to the Stats task\n");
sendpkt(new Pkt(null,g.stats,act_run));

// Only gets here when all the clients have complete all their schedules.

System.out.println(coName+": Finish time = "+nowmsecs()+" msecs\n");

//System.out.println(coName+": Sleeping for 1 second");
//realdelay(1000);

g.stats.incqpkt();
g.stats.prstats();

//System.out.println("\n"+coName+": Sending die packets to all tasks\n");


for (int mpxno=1; mpxno<=g.mpxmax; mpxno++) {
    sendpkt(new Pkt(null, g.mpxv[mpxno], act_die));
    System.out.println(coName+
       ": Wait for a Multiplexor task "+g.mpxv[mpxno]+" to finish"
       );
    try { g.mpxv[mpxno].join(); }
    catch(Exception e) {}
    System.out.println(coName+": Task "+g.mpxv[mpxno]+" task has finished");
}

for (int srvno=1; srvno<=g.srvmax; srvno++) {
    sendpkt(new Pkt(null, g.rdserverv[srvno], act_die));
    //System.out.println(coName+": Wait for a server task to finish");
    // Wait for the clock task to finish
    try { g.rdserverv[srvno].join(); }
    catch(Exception e) {}
    System.out.println(coName+": The RD"+strz(srvno,2)+" task has finished");
}
for (int srvno=1; srvno<=g.srvmax; srvno++) {
    sendpkt(new Pkt(null, g.wrserverv[srvno], act_die));
    //System.out.println(coName+": Wait for a server task to finish");
    // Wait for the clock task to finish
    try { g.wrserverv[srvno].join(); }
    catch(Exception e) {}
    System.out.println(coName+": The WS"+strz(srvno,2)+" task has finished");
}

for (int clino=1; clino<=g.climax; clino++) {
    sendpkt(new Pkt(null, g.rdclientv[clino], act_die));
    //System.out.println(coName+": Wait for a client task to finish");
    try { g.rdclientv[clino].join(); }
    catch(Exception e) {}
    System.out.println(coName+": The RC"+strz(clino,2)+" task has finished");
}
for (int clino=1; clino<=g.climax; clino++) {
    sendpkt(new Pkt(null, g.wrclientv[clino], act_die));
    //System.out.println(coName+": Wait for a client task to finish");
    try { g.wrclientv[clino].join(); }
    catch(Exception e) {}
    System.out.println(coName+": The WC"+strz(clino,2)+" task has finished");
}

System.out.println("\n"+coName+": Sending die pkt to Printer task");
qpkt(new Pkt(null, g.printer,
     act_die));
taskwait();
//System.out.println(coName+": Die pkt returned from Printer task");

//System.out.println(coName+": Wait for the Printer task to finish");
        // Wait for the printer task to finish
try { g.printer.join(); }
catch(Exception e) {}
System.out.println(coName+": The Printer task has finished");


System.out.println("\n"+coName+": Sending die pkt to Bounce task");
qpkt(new Pkt(null, g.bounce,
     act_die));
taskwait();
//System.out.println(coName+": Die pkt returned from Bounce task");

//System.out.println(coName+": Wait for the Bounce task to finish");
        // Wait for the bounce task to finish
try { g.bounce.join(); }
catch(Exception e) {}
System.out.println(coName+": The Bounce task has finished");
System.out.println("\n"+coName+": Sending die pkt to Stats task");
qpkt(new Pkt(null, g.stats,
     act_die));
taskwait();
//System.out.println(coName+": Die pkt returned from Stats task");

//System.out.println(coName+": Wait for the Stats task to finish");
        // Wait for the stats task to finish
try { g.stats.join(); }
catch(Exception e) {}
System.out.println(coName+": The Stats task has finished");


System.out.println("\n"+coName+": Sending die pkt to Clock task");
qpkt(new Pkt(null, g.clock,
     act_die));
taskwait();
//System.out.println(coName+": Die pkt returned from clock task");

//System.out.println(coName+": Wait for the Clock task to finish");
        // Wait for the clock task to finish
try { g.clock.join(); }
catch(Exception e) {}
System.out.println(coName+": The Clock task has finished");



        return null;
    }
}

//*******************************************************************
//************************** Stats **********************************
//*******************************************************************

class Stats extends Task {

    // This class holds the statistics counters and the method (prstats)
    // to output them. It controls the synchronisation of the clients
    // involving sync, rddone and wrdone packets. It also estimates the
    // CPU usage by repeatedly bouncing packets off the Bounce task.

    Stats r;

    // Statistics counters
```

```
public int c_qpkt=0;
public int c_taskwait=0;
public int c_callco=0;
public int c_resumeco=0;
public int c_cowait=0;
public int c_condwait=0;
public int c_notify=0;
public int c_notifyAll=0;
public int c_inc=0;
public int c_incwait=0;
public int c_lock=0;
public int c_lockw=0;
public int c_delaylong=0;
public int c_bounce=0;
public int c_print=0;
public int c_logger=0;
public int c_readfail=0;
public int c_sendfail=0;
public int c_rdchecksum=0;
public int c_wrchecksum=0;

public int c_rdcount=0;
public int c_wrcount=0;

Pkt synclist = null;
int synclistlen = 0;
Pkt donelist = null;
int donelistlen = 0;

Pkt calibratepkt = null;
Pkt runpkt = null;
Pkt diepkt = null;
Boolean alldone = false;
int bounces = 0;
int bouncesmax = 1;         // To avaiod division by zero
Boolean bouncing = false;
Boolean clocking = false;
Pkt clockpkt = null;
Pkt bouncepkt = null;
public int utilisationv[];
int cycle = 0;

// This task has no non root coroutines, so only has a
// task constructor.
```

```
    public Stats(String name, Globals g, int pri) {
super(name, g, pri);
r = this;
        System.out.println(coName+": Task created");
    }

    public Object fn(Object arg) {
// This is the main function of the root coroutine of
        // the stats task. It must not recive its startup
// packet until the Clock and Bounce tasks have been
// created (but not given their statup packets.
Pkt startuppkt = (Pkt)arg;

//System.out.println(coName+": Startup packet received");

        clockpkt = new Pkt(null, g.clock, act_clock,
                             0,0,          // res1 res2
                 100);         // 100 msecs delay

        bouncepkt = new Pkt(null, g.bounce, act_bounce);

utilisationv = new int[10];
        for (int i=0; i<=9; i++) utilisationv[i] = 0;
        cycle = 1;

//System.out.println(coName+": Returning the startup packet to task "
        //                        +startuppkt.task.coName);
//c_qpkt++;
        qpkt(startuppkt); // Return the startup packet.

//System.out.println(coName+": Entering its main loop");

        while (true) {
    Pkt pkt = taskwait();

    switch (pkt.type) {
    default:
System.out.println(coName+
   ": Unexpected packet received from "+pkt.task.coName);
qpkt(pkt); // Return this unexpected packet.
continue;

    case act_calibrate:
calibratepkt = pkt;
//System.out.println(coName+
//    ": calibrate packet received from "+pkt.task.coName);
//System.out.println(coName+
//    ": synclistlen="+synclistlen+
//    " 2*climax="+ 2*g.climax);

// Only perform calibration when all read and write clients
// are ready to perform their first schedules.
if (synclistlen != g.climax+g.climax)
    continue;

// All clients are ready to start their first schedules.
calibrate();
//System.out.println(coName+
//    ": Return the calibrate packet to task "+
//    calibratepkt.task.coName);
qpkt(calibratepkt); // Return the calibrate packet to the controller.
calibratepkt = null;
continue;

    case act_run:
// act_run will not be received until after calibration has
// been completed, and that will only happen when all clients
// are ready to start their schedules.
//System.out.println(coName+
//    ": act_run packet received from "+
//    pkt.task.coName);
runpkt = pkt; // Remember the run packet so that it can be
                              // returned at the end.

// clocking and bouncing should both be false.

if (clocking)
    System.out.println(coName+
       ": ERROR: clocking should be false");

if (bouncing)
    System.out.println(coName+
       ": ERROR: bouncing should be false");

// Start the clock
qpkt(clockpkt);
clocking = true;

// Start bouncing
qpkt(bouncepkt);
```

```
bouncing = true;
// Fall through to allow case act_sync to release
// all the clients.

    case act_sync:
if (pkt.type==act_sync) {
    // Request from a client to start the next schedule.
    // It is returned when all clients have sent sync packets.
    pkt.link = synclist;
    synclist = pkt;
    synclistlen++;
    //if (g.tracing) {
    // System.out.println(coName+
    //    ": sync packet received from "+
    //    pkt.task.coName);
    //}
}

//System.out.println(coName+
//    ": synclistlen="+synclistlen+
                //                  "     climax*2="+2*g.climax);

if (synclistlen != g.climax+g.climax) continue;

System.out.println(coName+
   ": All sync packet received");

if (calibratepkt!=null) {
    //System.out.println(coName+
    //        ": and calibrate packet received, so call calibrate");

    calibrate();
    //System.out.println(coName+
    //        ": Return the calibrate packet to task "+
    //        calibratepkt.task.coName);
    qpkt(calibratepkt);
    calibratepkt = null;
    // The act_run packet has not yet been received.
    continue;
}

// The controller only sends the run packet after
// calibration has been completed.

if (runpkt!=null) {
    // The run packet and all sync packets have been received
    // so return the sync packets to theirmclients.
    Pkt p = synclist;

    //if (g.tracing) {
    System.out.println(coName+
       ": Releasing all client");
    cycle++;
    //}
    synclist = null;
    synclistlen = 0;
    while (p!=null) {
pkt = p;
p = p.link;
pkt.link = null;
qpkt(pkt);
    }
}

// We must still be waiting for more sync packets or
// a run packet.
continue;

    case act_clock:
// Keep clocking until alldone is true
if (!alldone) {
    int u;
    qpkt(pkt);

    // Even after calibration bouncesmax may still increase.
    if (bouncesmax < bounces) bouncesmax = bounces;

    u = 999 * (bouncesmax-bounces) / bouncesmax / 100;
    // 0 <= u <= 9

    utilisationv[u]++;
    bounces = 0;
    continue;
}

// alldone is true
clocking = false;
// Do not send the packet to the clock
continue;
```

```
    case act_bounce:
//System.out.println(coName+": bounce pkt received");
if (clocking) {
    // Still in a time perios so bounce again
    qpkt(pkt);
    continue;
}

// clocking is false so the end of the final period
// has been reached. This only happens when a clock packet
// has been received when alldon is true.

// We have just received the first bounce packet
// after the end of the final time period. It is therefore
// time to return the run packet to the controller.

bouncing = false;
qpkt(runpkt);
continue;

    case act_rddone:
    case act_wrdone:
// A client has finished all its schedules.

if (pkt.type==act_rddone) {
    if (g.tracing) {
System.out.println(coName+
   ": rddone packet received from "+
   pkt.task.coName);
    }
} else {
    if (g.tracing) {
System.out.println(coName+
   ": wrdone packet received from "+
   pkt.task.coName);

    }
}

pkt.link = donelist;
donelist = pkt;
donelistlen++;
if( donelistlen == g.climax+g.climax) {
    // All the rddone and wrdone packets have been received
    // so return them to their clients.
    Pkt p = donelist;
    donelist = null;
    donelistlen = 0;

    while (p!=null) {
pkt = p;
p = p.link;
pkt.link = null;
qpkt(pkt);
    }

    alldone = true;
    if (g.tracing) {
System.out.println(coName+
   ": alldone set to true");
    }
}
continue;

    case act_die:
//System.out.println(coName+": die pkt received");
g.stats.incqpkt();
qpkt(pkt);
return null;     // Cause the thread to die
    }
}
    }

    public void calibrate() {
int cycles = 10;
System.out.println(coName+": Calibrating the bounce counter");

bounces = 0;        // Start clocking
clocking = true;
qpkt(clockpkt);

bouncing = true;    // Start bouncing
qpkt(bouncepkt);

while (clocking || bouncing) {
    Pkt pkt= taskwait();

    switch (pkt.type) {
    default:
System.out.println(coName+": Unexpected packet from "+
   pkt.task.coName);
continue;

    case act_clock:
clocking = false;
if (bouncesmax==0) bounces = 0; // ignore first time period
if (bouncesmax<bounces) bouncesmax = bounces;
//System.out.println(coName+
                //                    ": cycles="+cycles+
                //                    " Clock pkt received, bounces="+bounces+
                //                    " bouncesmax="+bouncesmax);
bounces = 0;
cycles--;
if (cycles>0) {
    // Start another 100 msecs time period.
    qpkt(clockpkt);
}
continue;

    case act_bounce:
//System.out.println(coName+": Bounce pkt received "+bounces);
bounces++;
bouncing = false;
if (cycles==0) break; // Both clock and bounce packets are back
qpkt(bouncepkt);
bouncing = true;
continue;
    }

}

//
System.out.println("\n"+coName+": bouncesmax = "+bouncesmax+"\n");
    }

    public synchronized void incqpkt()       { c_qpkt++; }
    public synchronized void inctaskwait()   { c_taskwait++; }
    public synchronized void inccallco()     { c_callco++; }
    public synchronized void incresumeco()   { c_resumeco++; }
    public synchronized void inccowait()     { c_cowait++; }
    public synchronized void inccondwait()   { c_condwait++; }
    public synchronized void incnotify()     { c_notify++; }
    public synchronized void incnotifyAll()  { c_notifyAll++; }
    public synchronized void incinc()        { c_inc++; }
    public synchronized void incincwait()    { c_incwait++; }
    public synchronized void inclock()       { c_lock++; }
    public synchronized void inclockw()      { c_lockw++; }
    public synchronized void incdelaylong()  { c_delaylong++; }
    public synchronized void incbounce()     { c_bounce++; }
    public synchronized void incprint()      { c_print++; }
    public synchronized void inclogger()     { c_logger++; }
    public synchronized void increadfail()   { c_readfail++; }
    public synchronized void incsendfail()   { c_sendfail++; }
    public synchronized void incrdchecksum() { c_rdchecksum++; }
    public synchronized void incwrchecksum() { c_wrchecksum++; }
    public synchronized void incrdcount()    { c_rdcount++; }
    public synchronized void incwrcount()    { c_wrcount++; }

    public synchronized void prstats() {
System.out.println("\n");

System.out.println("Number of calls of qpkt:      "+Task.strn(c_callco,9));
System.out.println("Number of calls of taskwait:  "+Task.strn(c_taskwait,9));
System.out.println("Number of calls of callco:    "+Task.strn(c_callco,9));
System.out.println("Number of calls of resumeco:  "+Task.strn(c_resumeco,9));
System.out.println("Number of calls of condwait:  "+Task.strn(c_condwait,9));
System.out.println("Number of calls of notify:    "+Task.strn(c_notify,9));
System.out.println("Number of calls of notifyAll: "+Task.strn(c_notifyAll,9));
System.out.println("Number of increments:         "+Task.strn(c_inc,9));
System.out.println("    increment had to wait:    "+Task.strn(c_incwait,9));
System.out.println("Number of calls of lock:      "+Task.strn(c_lock,9));
System.out.println("    lock had to wait:         "+Task.strn(c_lockw,9));
System.out.println("Number of "+Task.strn(g.delaymsecs,4)+
   " msec delays:   "+Task.strn(c_delaylong,9));
System.out.println("Print task counter:           "+Task.strn(c_print,9));
System.out.println("Calls to logger:              "+Task.strn(c_print,9));
System.out.println("Send fail count:              "+Task.strn(c_sendfail,9));
System.out.println("Read fail count:              "+Task.strn(c_readfail,9));
System.out.println("Bounce task counter:          "+Task.strn(c_bounce,9));
System.out.println("Read checksum:                "+Task.strn(c_rdchecksum,9));
System.out.println("Write checksum:               "+Task.strn(c_wrchecksum,9));
System.out.println("Read count:                   "+Task.strn(c_rdcount,9));
System.out.println("Write count:                  "+Task.strn(c_wrcount,9));
System.out.println("");

int total = 0;
for(int i = 0; i<=9; i++) total += utilisationv[i];
if (total==0) total++;

        System.out.println(
          "     Approximate CPU utilisation over "+total+" periods of 100 msecs"
        );
        System.out.println("   0-10% 10-20% 20-30% 30-40% 40-50% 50-60%"+
                           " 60-70% 70-80% 80-90% >90%");
for(int i = 0; i<=9; i++) {
    Task.wrn(utilisationv[i], 5);
            System.out.print("  ");
}
System.out.println("\n");
    }
}

//************************************************************************
//************************** Pktlist *********************************
//************************************************************************

class Pktlist {
    // This hold nodes in a list that maps packets to coroutines
    // used in gomultievent.
    public Pktlist link;
    public Pkt pkt;
    public Cortn cptr;

    public Pktlist(Pktlist link, Pkt pkt, Cortn cptr) {
this.link = link;
this.pkt = pkt;
this.cptr = cptr;
    }
}

//************************************************************************
//************************** Tasklist ********************************
//************************************************************************

class Tasklist {
    // This holds a list of tasks (typically coroutine).

    public Tasklist next;
    public Cortn cptr;

    public Tasklist(Tasklist next, Cortn cptr) {
this.next = next;
this.cptr = cptr;
    }
}
```

```
//*******************************************************************
//*************************** Pkt ***********************************
//*******************************************************************

class Pkt {
    // Used in Cintpos-style task to task communication
    public Pkt  link;
    public Task task;
    public int  type;
    public int  res1;
    public int  res2;
    public int  arg1;
    public int  arg2;
    public int  arg3;
    public int  arg4;
    public int  arg5;
    public int  arg6;

    public Pkt(Pkt link, Task task, int type) {
this.link = link;
this.task = task;
this.type = type;
    }

    public Pkt(Pkt link, Task task, int type, int res1, int res2,
               int arg1) {
this.link = link;
this.task = task;
this.type = type;
this.res1 = res1;
this.res2 = res2;
this.arg1 = arg1;
    }

    public Pkt(Pkt link, Task task, int type, int res1, int res2,
               int arg1, int arg2) {
this.link = link;
this.task = task;
this.type = type;
this.res1 = res1;
this.res2 = res2;
this.arg1 = arg1;
this.arg2 = arg2;
    }
    public Pkt(Pkt link, Task task, int type, int res1, int res2,
               int arg1, int arg2, int arg3) {
this.link = link;
this.task = task;
this.type = type;
this.res1 = res1;
this.res2 = res2;
this.arg1 = arg1;
this.arg2 = arg2;
this.arg3 = arg3;
    }

    public Pkt(Pkt link, Task task, int type, int res1, int res2,
               int arg1, int arg2, int arg3, int arg4) {
this.link = link;
this.task = task;
this.type = type;
this.res1 = res1;
this.res2 = res2;
this.arg1 = arg1;
this.arg2 = arg2;
this.arg3 = arg3;
this.arg4 = arg4;
    }

    public Pkt(Pkt link, Task task, int type, int res1, int res2,
               int arg1, int arg2, int arg3, int arg4, int arg5) {
this.link = link;
this.task = task;
this.type = type;
this.res1 = res1;
this.res2 = res2;
this.arg1 = arg1;
this.arg2 = arg2;
this.arg3 = arg3;
this.arg4 = arg4;
this.arg5 = arg5;
    }

    public Pkt(Pkt link, Task task, int type, int res1, int res2,
               int arg1, int arg2, int arg3, int arg4, int arg5, int arg6) {
this.link = link;
this.task = task;
this.type = type;
this.res1 = res1;
this.res2 = res2;
this.arg1 = arg1;
this.arg2 = arg2;
this.arg3 = arg3;
this.arg4 = arg4;
this.arg5 = arg5;
this.arg6 = arg6;
    }
}

//*****************************************************************
//*************************** OccamChannel ***************************
//*****************************************************************

class OccamChannel {
    // This class models an Occam channel. The values that are passed
    // through the channel are always integers.

    String chnName;
    Task t;              // To allow access to t.currco etc

    public OccamChannel(String name, Task t) {
chnName = name;  // Typical name: RS03loggerin
this.t = t;
    }

    Cortn cptr; // The channel word.

    // Remember that when only one of the coroutines belonging to
    // a task can have control. Control is passed from one
    // conroutine to another using synchronized methods so if
    // one coroutine writes to cptr it will be written out to
    // memory before another coroutine tries to read it. There
    // is thus no need to declare cptr as volatile. There is
    // also no need to make coread and cowrite synchronized.

    int coread() {
if (cptr!=null) {
    return ((Integer)(t.resumeco(cptr, t.currco))).intValue();
}

cptr = t.currco;
return ((Integer)(t.cowait(null))).intValue();
    }
    public void cowrite(int x) {
Cortn reader = cptr;

if (reader!=null) {
    // reader points to a reader coroutine that is ready
    // to read data from this channel, so clear the
    // channel word.
    cptr = null;
    // and fall throug to send the data to the reader.
} else {
    // The reader coroutine is not yet ready to read.
    // So tell it the identity of the writer
    // coroutine.
    cptr = (Cortn)t;
    // and wait for the reader coroutine to send us
    // its identity
    reader = (Cortn)(t.cowait(null));
    // and then fall through to send the data.
}
t.callco(reader, new Integer(x));
    }

    public void wrpn(int x) {
if (x!=0) {
    wrpn(x>>>1);
    cowrite(x&1);
}
    }

    public int getloggerval() {
int res = 0;

while (true) {
    int dig = coread();
    if (dig<0) return res;
    res = 2*res+dig;
}
    }

    public synchronized void pr() {
if (cptr==null) {
    System.out.println(chnName+": channel is idle");
} else {
    System.out.println(chnName+": channel points to "+cptr.coName);
```

```
}
    }
}

//*******************************************************************
//*************************** Lock **********************************
//*******************************************************************

class Lock {
    Tasklist colist;
    boolean locked = false;

    public Lock() {
colist = null;
    }

    public synchronized void lock(Task cptr) {
// Usage: eg loggerlock.lock(this);
if (locked) {
    Tasklist node = new Tasklist(null, cptr);

    if (colist==null) {
// It was locked by just one coroutine,
// so make a unit list.
colist = node;
    } else {
// Append the lock node to the end of the list.
Tasklist p = colist;
while (p.next!=null) p = p.next;
p.next = node;
    }
    cptr.cowait(null); // Suspend until released by unlock().
    // We now own the lock.
    return;
}
// We can obtain the lock.
locked = true;
return;
    }

    public synchronized void unlock(Task t) {
// Usage: eg loggerlock.unlock(this);

if (!locked) {
    System.out.println(t.coName+
       ": Attempt to free a lock that was not locked");
    return;
}
if (colist!=null) {
    Cortn cptr = colist.cptr;
    colist = colist.next;
    t.callco(cptr, null);
    return;
}
locked = false;
return;
    }
}


//*********************************************************************
//*************************** Clock *********************************
//*********************************************************************

class Clock extends Task {
    ///task
    // An instance of this class models the Tripos clock device.

    Clock r;

    // Once started it receives act_clock packets and returns them to
    // the sender about arg1 msecs later. It will commit suicide when
    // given an act_die packet.

    // The public variable waiting is only true when the task is waiting
    // in a call of wait. It may be waiting because the priority queue
    // is empty, or it may be waiting for the time to release the next
    // packet from the priority queue. In either case qpkt should call
    // notifyAll to wakeup the wait() call when another packet is sent
    // to the clock task.

    // The arg1 field of a clock packet holds the delay time in msecs,
    // and the res1 field is set to the real wakeup time. These packets
    // are held in a priority queue implemented using the heap structure
    // typically used in heap sort. Each packet in the queue is held in
    // an element of heapv. The number of packets in the queue is heapn
    // and there is the constraint that the res1 field of the packet
    // in heapv[i]  is less than or equal to the res1 fields belonging to
    // packets heapv[2*i] and heapv[2*i+1], if they exist. This guarantees
    // that the packet in heapv[1] is the earliest to be released.
    // Normally the clock task is suspended in a call of wait waiting
    // for a clock packet to arrive. When one comes, it sets the res1
    // field and increments heapn. It it then inserts the packet at
    // position heapn and successively promotes it using upheap until
    // the heap constraint is satisfied, before re-entering the call
    // of wait. The argument of wait is the number of msecs until the
    // earliest packet should be released and if this time period
    // expires before another clock packet arrives, the packet in
    // heapv[1] is released and replaced by the packet in heapv[heapn],
    // if any. The packet in heapv[1] is then successively demoted using
    // downheap until its res1 less than or equal to the res1 fields
    // belonging to packets heapv[2*i] and heapv[2*i+1], if they exist,
    // where i is the current position of the packet. This mechanism
    // assures that both upheap and downheap are reasonably efficient.

    public boolean waiting;
    public Pkt heapv[];
    public Pkt diepkt = null;
    public int heapn;    // Number of packets in the heap
    public int heapnmax; // Current size of the heap

    public Clock(String name, Globals g, int pri) {
super(name, g, pri);
r = this;
        System.out.println(coName+": Task created");
    }

    public Object fn(Object arg) {
int delaymsecs;
Pkt startuppkt = (Pkt) arg;
Pkt clockpkt = null;
diepkt = null;
        //System.out.println(coName+" : fn entered, pkt type="+strn(startuppkt.type,3)

        heapnmax = 10;  // Initial heap size.
heapv = new Pkt[heapnmax+1];
heapn = 0;     // Nothing in the heap yet.
waiting = false;

//synchronized (g.stats) { g.stats.c_qpkt++; }

        //System.out.println(coName+": Returning the startup packet to the "+startuppk
        //System.out.println("Clock: Initialised");
```

```
qpkt(startuppkt); // Return the startup packet to the controller.

        while(true) {
    // This is running in the clock thread.

    // Wait until the earliest packet is the priority queue
    // can be released or until a die packet is received
    // from the controller.

    int now = nowmsecs();

    //System.out.print("Clock: heapn="+strn(heapn,3));
    //System.out.print(" now="+strn(now,6));
    //if (heapn>0) {
    //    System.out.print(" res1="+strn(heapv[1].res1,6));
    //}
    //System.out.println("");
    while (heapn>0 && now>=heapv[1].res1) {
// Release the packet in heapv[1]
qpkt(heapv[1]);
if (heapn>1) heapv[1] = heapv[heapn];
heapn--;
downheap(); // Demote the new packet in heapv[1]
// See if there are more packets to release.
    }

    if (diepkt!=null) {
// Release all clock packets now, return the die packet
// to the controller and then die.
//System.out.println("Clock: Clock is dying");
if (heapn!=0)
    System.out.println("Clock: ERROR Clock priority queue is not empty");

//System.out.println(coName+": Returning the die packet to the controller");
qpkt(diepkt);

//System.out.println(coName+": Returning from fn to die");
return null;
    }

    try {
waiting = true;
if (heapn>0) {
    int dt = heapv[1].res1 - now;
    // Wait for timeout or notify
```

```
        //System.out.println(coName+": Calling wait("+dt+")");
        wait(dt);
    } else {
        //System.out.println(coName+": Calling wait() for the next clock pkt");
        // Wait for notify
        wait();
    }
    waiting = false;
        } catch(Exception e) {}
        // Go back to the start of the while(true) loop
    }
        }

        public synchronized boolean putpkt(Pkt pkt, Task from) {
    //System.out.println("\n"+coName+": Got lock on this thread");

    // This clock version overrides putpkt in class Task.

    // It is running in a client thread (not the clock thread).

    pkt.task = from;        // Set the return task field.
    pkt.link = null;

    // This is called in the destination task (the clock),
    // It puts the packet into the priority queue then calls
    // notifyAll to wakeup the clock thread.

    if (pkt.type==act_die) {
        //System.out.println("Clock: Die packet received");
        diepkt = pkt;
        if (waiting) notifyAll();
        //System.out.println(coName+": Releasing lock on this thread\n");
        return true;
    }

    if (pkt.type==act_startclock) {
        //System.out.println(coName+": Startup packet received");

        // Append the packet onto the end of this task's wkq.
        // It is always the first packet so wkq=null.

        wkq = pkt; // Make a unit list.
        //System.out.println(coName+": putpkt calling notifyAll()"+
        //    " since wkq was previously null");
        notifyAll();
        //System.out.println(coName+": Releasing lock on thread\n");
        return true;
    }

    if (pkt.type!=act_clock) {
        System.out.println(coName+": Unexpected packet, type = "+pkt.type);
        qpkt(pkt);
        System.out.println(coName+": Releasing lock on this thread\n");
        return true;
    }

    pkt.res1 = pkt.arg1 + nowmsecs(); // Time to release.

    if ( heapn==heapnmax) {
        //System.out.println(coName+": About to increase heap size from "+heapnmax);
        //prheap();
        // The heap is full, so we must enlarge it.
        int newnmax = heapnmax*3/2 + 1;
        Pkt newheapv[] = new Pkt[newnmax+1];
        for(int i = 1; i<=heapn; i++) newheapv[i] = heapv[i];
        heapnmax = newnmax;
        heapv = newheapv;
        //System.out.println(coName+": New heap size "+heapnmax);
        //prheap();
    }

    heapn++;
    heapv[heapn] = pkt;
    upheap();

    if (waiting) notifyAll();
    //System.out.println(coName+": Releasing lock on this thread\n");
    return true;
        }

        private void upheap() {
    int p = heapn;
    Pkt pkt = heapv[p];
    int msecs = pkt.res1;

    while (p>1) {
        int q = p/2; // The parent of p.
        if (heapv[q].res1 <= msecs) break;
        // Promote the packet
        heapv[p] = heapv[q];
        p = q;
    }
    heapv[p] = pkt;
    //prheap();
        }

        private void downheap() {
    int p = 1;
    Pkt pkt = heapv[1];
    // p points to a vacant position in the heap,
    // initially location 1.
    int msecs = pkt.res1; // Release time of pkt.

    while (true) {
        int q = p+p; // Position of left child
        if (q > heapn) break; // No children
        // There is at least one child.
                if (q < heapn) {
    // There are two children, choose the earlier.
    if( heapv[q].res1 > heapv[q+1].res1) q++;
        }
        // q is the position of the earlier child.
        if (msecs <= heapv[q].res1) break; // Do not promote.

        // Promote the earlier child.
        heapv[p] = heapv[q];
        p = q;
    }
    heapv[p] = pkt;
    //prheap();
        }

        public synchronized void prheap() {
    if (heapn==0) {
        System.out.println("\nThe priority queue is empty");
    } else {
        System.out.println("\nThe priority queue:");
    }
    for(int i=1; i<=heapn; i++) {
        Pkt pkt = heapv[i];
        String pname = pkt.task==null ? "<null>" : pkt.task.coName;
        System.out.println(strn(1,2)+
           "  "+strn(pkt.arg2,2)+
           "   res1: "+strn(pkt.res1,6)+
           " arg1: "+strn(pkt.arg1,6)+
           " from: "+pname);
    }
    System.out.println("");
        }
    }

    //*****************************************************************
    //*************************** Bounce *******************************
    //*****************************************************************

    class Bounce extends Task {
        ///task

        Bounce r;

        public Bounce(String name, Globals g, int pri) {
    // This constructor is used to create the Bounce task and its root coroutine.
    super(name, g, pri);
    r = this;
            System.out.println(coName+": Task created");
        }

        public Object fn(Object arg) {
            //System.out.println(coName+": fn entered");
            Pkt startuppkt = (Pkt) arg;
    int count = 0;

    // Create and start the echo coroutine.
    //System.out.println(coName+": calling new(Echoco, this)");
    Echoco echoco = new Echoco("Echoco", this);
    // Transfer control to echoco leaving it suspended in its main loop:

    echoco.parent = this;
    echoco.value = null;
    currco = echoco;

    //System.out.println(coName+": calling echoco.start()");
    echoco.start();

    //System.out.println(coName+": Coroutine "+echoco.coName+" created and thread started");

    // Wait until this coroutine is the current coroutine.
    currcoWait();
```

```
if (true) { // =true to test callco and cowait
    int cycles = 10000;
    System.out.println("\n"+coName+": Calling callco and cowait "+cycles+" times");
    int start_time = nowmsecs();

    for (int i=1; i<=cycles; i++) {
//System.out.println(coName+
//    ": "+strn(i,5)+" Calling callco(echoco,"+i+")");
callco(echoco, (Object)(new Integer(i)));
    }

    int diff = nowmsecs() - start_time;
    if (diff==0) diff=1; // To avoid overflow.
    System.out.println("\n"+coName+
       ": Time taken "+diff+ " msecs"+
       " giving "+((cycles*1000)/diff)+
       " callco-cowait bounces per second\n");
}

//System.out.println(coName+
//    ": Returning the startup packet to "+
//    startuppkt.task.coName+"\n");
        qpkt(startuppkt); // Return the startup packet

System.out.println(coName+": Ready");

        while(true) {
    Pkt pkt = taskwait();
    switch (pkt.type) {
    default:
System.out.println(coName+
   ": unexpected packet received from "+
                           pkt.task.coName+
                           " type "+pkt.type);
continue;

    case act_bounce:
count++;
//System.out.println(coName+": bounce pkt received");
                for(int i=1; i<=10; i++) callco(echoco, null);
qpkt(pkt);
continue;

    case act_quickbounce:
// For calibrating the qpkt-taskwait bounce rate.
//System.out.println(coName+": quick bounce pkt received");
qpkt(pkt);
continue;

    case act_die:
//System.out.println(coName+": die pkt received");
//System.out.println(coName+": Set echoco.dying to true");
echoco.dying = true;
//System.out.println(coName+": Calling callco(echoco, null)");
callco(echoco, null);
//System.out.println(coName+": Return the die packet to the Controller");
pkt.res1 = count;
qpkt(pkt);

return null;     // Cause this thread to die
    }
}
    }
}

//*******************************************************************
//************************** Echoco *********************************
//*******************************************************************

class Echoco extends Cortn {
    ///coroutine
    // This is the echo coroutine only used in by the Bounce task. Its
    // purpose is to maintain a reasonably high coroutine change to
    // thread change ratio.

    Globals g;
    Bounce r;            // Pointer to the owning task's variables.
    boolean dying=false; // When =true this coroutine will die.

    public Echoco(String name, Bounce r) {
// Create a non-root coroutine belonging to the Bounce
// task r. By convention r is given the type of the
// owning task. This allows indirect access to all the
// Bounce task's instance variables such as t.currco,
// and methods such as r.qpkt and r.callco that are
// declared in class Task which is the superclass
// of Bounce.
super(name, r);
coName = name;      // Set the coroutine's name.
this.r = r;
```

```
        System.out.println(coName+": Coroutine created");

// When this coroutine is started, it will enter run defined
// class Cortn which contains the standard non-root
// coroutine main loop. The parent variable will already
// be set appropriately and t.currco will point to its
// this coroutine’s instance variables. This coroutine
// will thus immediately suspend itself in cowait(c) of
// the coroutine main loop. When callco is called the
// value will be passed as the argument of fn. A return from
// fn will cause cowait(c) to be called again.
    }

    public Object fn(Object x) {
//System.out.println(coName+": received value "+((Integer)x).intValue());

while (!r.dying) x = cowait(x);

return x;
    }
}


//*********************************************************************
//************************** Printer **********************************
//*********************************************************************

class Printer extends Task {
    ///task

    Printer r;

    public Printer(String name, Globals g, int pri) {
super(name, g, pri);
r = this;
        System.out.println(coName+": Task created");
    }

    public Object fn(Object c) {
        //System.out.println(coName+": fn entered");
        Pkt startuppkt = (Pkt) c;
//int count = 0;
Pkt pkt = null;

Pkt delaypkt = new Pkt(null, g.clock, act_clock,
       0,0,  // res1 res2
       10);  // 10 msecs delay

Pkt q = null;  // List of pending print packets while delaying
Pkt printpkt = null;  // Current print packet, if any.

        //System.out.println(coName+
//    ": Returning the startup packet to "+
//    startuppkt.task.coName);
        qpkt(startuppkt); // Return the startup packet

        //System.out.println(coName+": Entering main loop");

while (true) {
    // Start of printer event loop
    if (q!=null) {
// Dequeue a pending print packet.
pkt = q;
q = q.link;
pkt.link = null;
if (g.tracing)
    System.out.println(coName+": Packet extracted from q");
    } else {
// Wait for a print or die packet.
pkt = taskwait();
if (g.tracing)
    System.out.println(coName+": taskwait returned a packet");
    }

    switch (pkt.type) {
    default:
System.out.println(coName+
   ": Unexpected packet received from "+
   pkt.task.coName);
continue;

    case act_print:
{   char modech = (char) pkt.arg1;
    int serno = pkt.arg2;

    if (g.tracing) {
System.out.println(coName+
   ": act_print packet received from "+
   pkt.task.coName);
    }
    printpkt = pkt;
    if (g.tracing) {
System.out.println(coName+
   ": Sending delay packet to Clock");
    }

    qpkt(delaypkt);

    while (true) {
// Wait for the delay pkt to return.
pkt = taskwait();
if (pkt==delaypkt) break;

// Insert the non delaypkt in q
pkt.link = q;
q = pkt;
    }

    // The delay has now ended, so return the print
    // packet and process another (possibly pending)
    // request.

    qpkt(printpkt);
    printpkt = null;
    continue;
}

    case act_die:
if (g.tracing) {
    System.out.println(coName+": Die packet received");
    System.out.println(coName+": Return the die packet to task "+
       pkt.task.coName);
    System.out.println(coName+": This task is now dying");
}
break;     // Cause the thread to die

    }
}
    }
}


//*************************************************************************
//*************************** Client ***********************************
```

```
//*****************************************************************

class Client extends Task {
    ///task

    Client r;

    public Client(String name, Globals g, int pri) {
super(name, g, pri);
r = this;
        System.out.println(coName+": Task created");
    }

    public Object fn(Object arg) {
        //System.out.println(coName+": fn entered");
        Pkt startuppkt = (Pkt) arg;
Server serverv[]; // This will hold the vector of read or write servers
                  // for this client.
Request requestv[] = new Request[g.requestvupb+1];
char modech;
int clino = startuppkt.arg1;

if (startuppkt.type==act_startrdclient) {
    modech = 'R';
    serverv = g.rdserverv;
} else {
    modech = 'W';
    serverv = g.wrserverv;
}

Rnd rnd = new Rnd(clino+100*modech);

if (false) { // Test the random number generator.
    for (int i=1; i<=50; i++) {
int x = rnd.next(9999);
System.out.print(" "+strn(x,5));
if (i %10 == 0) System.out.println("");
    }
    realdelay(1000);
}

System.out.println(coName+": Initialised");
        qpkt(startuppkt); // Return the startup packet

for (int count = 1; count<=g.loopmax; count++) {
    int pos = 0;
    int rqstvupb = g.requestvupb;
    int clidelayitem = 0; // This will be the subscript of requestv of
                              // the request to delay in the client.
    int srvdelayitem = 0; // This will be the subscript of requestv of
                              // the request to delay in the its server.
    int mpxdelayitem = 0; // This will be the subscript of requestv of
                              // the request to delay in its multiplexor.

    // This client will not create its next schedule until all other
    // read and write clients are ready to do the same. This is achieved
    // by sending a sync packet to the Stats task which is only returned
    // when all client sync packets have been received.

    sendpkt(new Pkt(null, g.stats, act_sync));

    System.out.println("\n"+coName+": Creating and processing schedule "+count);


// Create a list of requests.

// Choose three distinct requests to cause real time delays.

    if (g.requestvupb>=3) {
while (true) {
    clidelayitem = rnd.next(g.requestvupb);
    srvdelayitem = rnd.next(g.requestvupb);
    if (clidelayitem!=srvdelayitem) break;
}
while (true) {
    mpxdelayitem = rnd.next(g.requestvupb);
    if (mpxdelayitem!=clidelayitem ||
mpxdelayitem!=srvdelayitem) break;
}
    }

for (int srvno=1; srvno<=g.srvmax; srvno++)
    for (int mpxno=1; mpxno<=g.mpxmax; mpxno++)
for (int chnno=1; chnno<=g.chnmax; chnno++) {
    int data = 0;
    if (modech=='W') data = rnd.next(9999); // Range 1 to 9999
    char flag = 'n';
    pos++;
    if (pos==clidelayitem) flag = 'c'; // Client delay
    if (pos==srvdelayitem) flag = 's'; // Server delay
    if (pos==mpxdelayitem) flag = 'm'; // Multiplexor delay
    requestv[pos] = new Request(flag, srvno, mpxno,
chnno, data);
}

if (g.tracing || true) {
    System.out.println("\n"+coName+": Schedule of requests");
    for (int i=1; i<=g.requestvupb;i++) {
Request req = requestv[i];
char flag = req.flag;
int srvno = req.srvno;
int mpxno = req.mpxno;
int chnno = req.chnno;
int data = req.data;
System.out.print(" "+flag+modech+
 "S"+strz(srvno,2)+
 "M"+strz(mpxno,2)+
 "C"+strz(chnno,2)+
 ":"+strz(data,4));


if (i%5==0) System.out.println("");
    }
    System.out.println("");
}

//System.out.println("\n"+coName+": Starting next schedule ");

while (rqstvupb>0) {
    int i = rnd.next(rqstvupb);
    Request req = requestv[i];
    int data = req.data;
    char flag = req.flag;
    int srvno = req.srvno;
    int mpxno = req.mpxno;
    int chnno = req.chnno;

    if (modech=='R') {
// Code for a read client
if (g.tracing) {
    System.out.println("RC"+strz(clino,2)+
       "  "+flag+modech+
       "S"+strz(srvno,2)+
       "M"+strz(mpxno,2)+
       "C"+strz(chnno,2)+
       ":"+strz(data,4)+
       " Sending read request to server");
}
data = sendpkt(new Pkt(null,serverv[srvno],act_read,
       flag,clino,srvno,mpxno,chnno,data));
if (g.tracing) {
    System.out.println("RC"+strz(clino,2)+
       "  "+flag+modech+
       "S"+strz(srvno,2)+
       "M"+strz(mpxno,2)+
       "C"+strz(chnno,2)+
       ":"+strz(data,4)+
       " Packet returned from server");
}

if (data==0) {
    // Read was not successful since its channel buffer
    // was empty.

    if (g.tracing) {
System.out.print("RC"+strz(clino,2)+
 "  "+flag+modech+
 "S"+strz(srvno,2)+
 "M"+strz(mpxno,2)+
 "C"+strz(chnno,2)+
 ":"+strz(data,4)+
 " Read failed");
    }

    //c_readfail++;

    // Delay for 200 msecs to give other tasks a chance to run,
    // hopefully allowing write clients to put more data in
    // the channel buffers.
    delaytask(200);

    continue; // try sending another random item from the schedule.
}

// The read request was successful.

if (g.tracing) {
    System.out.println("RC"+strz(clino,2)+
       "  "+flag+modech+
       "S"+strz(srvno,2)+
       "M"+strz(mpxno,2)+
       "C"+strz(chnno,2)+
       ":"+strz(data,4)+
       " Read was successful");
    System.out.println(coName+"Realdelay(2000)\n");
    realdelay(2000);

    //g.c_rdchecksum = (g.rdchecksum+data) % 1000000;
    //g.c_rdcount++;
}
    } else {
// Code for a write client
if (g.tracing) {
    System.out.println("WC"+strz(clino,2)+
       "  "+flag+modech+
       "S"+strz(srvno,2)+
       "M"+strz(mpxno,2)+
       "C"+strz(chnno,2)+
       ":"+strz(data,4)+
       " Sending write request to server");
}
int rc = sendpkt(new Pkt(null,serverv[srvno],act_write,
 flag,clino,srvno,mpxno,chnno,data));
if (g.tracing) {
    System.out.println("WC"+strz(clino,2)+
       "  "+flag+modech+
       "S"+strz(srvno,2)+
       "M"+strz(mpxno,2)+
       "C"+strz(chnno,2)+
       ":"+strz(data,4)+
       " Packet returned from server");
}

if (rc==0) {
    // Write was not successful since its channel buffer
    // was full.

    if (g.tracing) {
System.out.println("WC"+strz(clino,2)+
   "  "+flag+modech+
   "S"+strz(srvno,2)+
   "M"+strz(mpxno,2)+
   "C"+strz(chnno,2)+
   ":"+strz(data,4)+
   " Send failed");
```

```
    }

    //g.c_sendfail++;

    // Delay for 20 msecs to give other tasks a chance to run,
    // hopefully allowing read clients to remove data from
    // the channel buffers.
    delaytask(20);

    continue; // try sending another random request from the schedule.
}

// The write request was successful in that the data was
// written by the specified multiplexor in the specified
// channel buffer. The data may not have been read yet by
// a read client.

if (g.tracing) {
    System.out.println("WC"+strz(clino,2)+
       "  "+flag+modech+
       "S"+strz(srvno,2)+
       "M"+strz(mpxno,2)+
       "C"+strz(chnno,2)+
       ":"+strz(data,4)+
       " Data sent");
    System.out.println(coName+"Realdelay(2000)\n");
    realdelay(2000);

    //g.c_wrchecksum = (g.c_wrchecksum+data) % 1000000;
    //g.c_wrcount++;
}
    }

    // The read or write request has been successfully processed.

    // Delay for delaymsecs if a client delay is specified.

    if (flag=='c') {
if (g.tracing) {
    System.out.println(coName+" "+
       modech+"C"+strz(clino,2)+
       "  "+flag+modech+
       "S"+strz(srvno,2)+
       "M"+strz(mpxno,2)+
       "C"+strz(chnno,2)+
       ":"+strz(data,4)+
       " Client delay");
}
//c_delaylong++;
delaytask(g.delaymsecs);
if (g.tracing) {
    System.out.println(coName+" "+
       modech+"C"+strz(clino,2)+
       "  "+flag+modech+
       "S"+strz(srvno,2)+
       "M"+strz(mpxno,2)+
       "C"+strz(chnno,2)+
       ":"+strz(data,4)+
       " Client delay ended");
}

// remove the read or write request from the schedule.
requestv[i] = requestv[rqstvupb];
rqstvupb--;

//Process another request from the schedule.
    }
    // Create and perform the next schedule.
}
}

// tell the Stats task that this client has finished its
// final schedule.

if (modech=='R') {
    if (g.tracing) {
System.out.println(coName+" "+
   modech+"C"+strz(clino,2)+
   " Sending act_rddone to the Stats task");
sendpkt(new Pkt(null, g.stats, act_rddone));
    }
} else {
    if (g.tracing) {
System.out.println(coName+" "+
   modech+"C"+strz(clino,2)+
   " Sending act_wrdone to the Stats task");
sendpkt(new Pkt(null, g.stats, act_wrdone));
    }
}
```

```
// Wait for the act_die packet from the controller.
if (g.tracing) {
    System.out.println(coName+" "+
       modech+"C"+strz(clino,2)+
       " Waiting for die packet from the controller");
}

//c_taskwait++;
Pkt pkt = taskwait();
if (pkt.type!=act_die) {
    System.out.println(coName+" "+
       modech+"C"+strz(clino,2)+
       " System error -- act_die packet expected");
} else {
    if (g.tracing) {
System.out.println(coName+" "+
   modech+"C"+strz(clino,2)+
   " Die packet received");
    }
}

//c_qpkt++;
qpkt(pkt);

if (g.tracing) {
    System.out.println(coName+" "+
       modech+"C"+strz(clino,2)+
       " Client dying");
}

return null;
    }
}

class Request {
    char flag;    // n=no delay c=client delay, s=server delay, m=multiplexor delay
    int srvno;
    int mpxno;
    int chnno;
    int data;    // Non zero if a write request

    public Request(char flag, int srvno, int mpxno, int chnno, int data) {
this.flag = flag;
this.srvno = srvno;
this.mpxno = mpxno;
this.chnno = chnno;
this.data = data;
    }
}

//*****************************************************************
//************************** Server *******************************
//*****************************************************************

class Server extends Task {
    ///task


    Server r;

    // This class is used to model read and write servers which
    // run in multievent mode using gomultievent defined in Task.

    public Pkt startuppkt = null;
    public Pkt diepkt = null;   // Set non null when a die pkt is received.
    public Pkt serverq = null;
    public int srvno;
    public char modech;
    public OccamChannel loggerin = null;
    public OccamChannel loggerout = null;
    public Loggerlock loggerlock = null;
    public Loggerco loggerco = null;

    public Condvar countcondvar = null;
    public Condvar pktcondvar = null;

    public Workerco wrkcov[] = null; // Vector of worker coroutines.
    public int countv[] = null; // Vector of work counts.

    public int minwrkcount = 0; // Count of the least busy worker.

    public int busycount = 0;  // Count of worker coroutines that are busy.
                        // A worker is busy when it is processing a
                // request packet. It is not busy when waiting
                        // countcondvar or pktcondvar.


    public Server(String name, Globals g, int pri) {
// This constructor is used to create a server tasks and their
// root coroutine.
```

```
super(name, g, pri);
r = this;
        System.out.println(coName+": Task created");
    }

    public Object fn(Object arg) {
// This is the main function of a Server’s root coroutine.

// Method fn is overridden by classes representing non-root
// server coroutines.

        //System.out.println(coName+": started");

// arg is the startup packet which has a type field holding
// either act_startrdserver or act_startwrserver.
// In ether cas the arg1 field holds the server number.
int count = 0;

        startuppkt = (Pkt) arg; // For use by servermainco

srvno = startuppkt.arg1;

modech = startuppkt.type==act_startrdserver ? ’R’ : ’W’;

//if (g.tracing) {
//    System.out.println(coName+": Server started, srvno="+srvno);
//}

Rnd rnd = new Rnd(srvno+100*modech);

Servermainco servermainco =
    new Servermainco(coName+"mainco",    // This server’s main coroutine
     this);  // Task
// Start servermainco leaving it suspended in main loop:
// while(!dying) c = fn(cowait());
//if (g.tracing) {
//    System.out.println(coName+": Calling servermainco.start()");
//    System.out.println(coName+":   to leave it in cowait() in the loop");
//    System.out.println(coName+":   while (!dying)c=fn(cowait(c));");
//    System.out.println(coName+":   fn will be given the argument null.;");
//}

// Give initial control to servermainco. It should immediately
// return control to this task, leaving it suspended in cowait(c).
servermainco.parent = this;
```

```
currco = servermainco;
servermainco.start();

//if (g.tracing) {
//  System.out.println(coName+
//        ": Calling currcoWait(), currco="+currco.coName);
//}
// Wait for servermainco to enter its while(!dying) c = fn(cowait(c)) loop.
currcoWait();

//System.out.println(coName+": Calling gomultievent(servermainco)");
gomultievent(servermainco);
//System.out.println(coName+": Returned from gomultievent(servermainco)");

// On return from gomultievent, the diepkt will be set.

//qpkt(diepkt);

if (g.tracing)
    System.out.println(coName+": Returning to DEAD state");

return null;
    }
}

//*********************************************************************
//************************** Servermainco ****************************
//*********************************************************************

class Servermainco extends Cortn {
    ///coroutine
    // This is the main coroutine of a read or write server.
    // It creates the worker coroutines and the logger, then
    // processes requests from client before being closed down
    // by a die packet from the Controller.

    Server r; // This supercedes Task r delared in class Cortn.

    public Servermainco(String name, Server r) {
// This constructor create a non-root coroutine belonging to
// a server task. The only possible coroutines are workers or
// loggers.
super(name, r);
this.r = r;
System.out.println(coName+": Coroutine created");
```

```
    }

    public Object fn(Object x) {
// This is the main function of servermainco given control
// by gomultievent. It is the main coroutine of a Server task.
// x  will be null and fn will not return.

//System.out.println(coName+": servermainco Coroutine entered, srvno="+r.srvno);

r.diepkt = null;

r.wrkcov = new Workerco[g.workmax+1]; // Vector of worker coroutines.
r.countv = new int[g.workmax+1];      // Vector of work counts.

for (int i=1; i<=g.workmax; i++) {
    r.wrkcov[i] = null;
    r.countv[i] = 0;
}

r.minwrkcount = 0;

r.busycount = 0;  // Count of worker coroutines that are busy.
                          // A worker is busy when it is processing a
                  // request packet. It is not busy when waiting
                          // countcondvar or pktcondvar.

r.loggerlock = new Loggerlock();

r.loggerin = new OccamChannel(r.coName+"loggerin", t);
r.loggerout = new OccamChannel(r.coName+"loggerout", t);

r.loggerco = new Loggerco(r.coName+"log", r);
// Start loggerco leaving it susprnded in main loop:
// while(!dying) c = fn(cowait());

r.loggerco.parent = this;
r.loggerco.value = null;
t.currco = r.loggerco;

r.loggerco.start();

// Wait for it to enter its while(!dying) c = fn(cowait()) loop.
currcoWait();

if (g.tracing) {
    System.out.println(coName+": Coroutine "+r.loggerco.coName+" created");
}


r.countcondvar = new Condvar(r.coName+"countcondvar", t);
if (g.tracing) {
    System.out.println(coName+": "+r.countcondvar.name+" created");
}
r.pktcondvar = new Condvar(r.coName+"pktcondvar", t);
if (g.tracing) {
    System.out.println(coName+": "+r.pktcondvar.name+" created");
}

for (int wrkno = 1; wrkno<=g.workmax; wrkno++) {
    Workerco cptr = new Workerco(""+r.modech+"S"+strz(r.srvno,2)+"W"+strz(wrkno,2), r);
    r.wrkcov[wrkno] = cptr;

    // Give control to the newly created worker coroutine,
    // telling it its worker number.
    cptr.parent = this;
    // The first value given to a worker coroutine is its worker number.
    cptr.value = new Integer(wrkno);
    r.currco = cptr;

            cptr.start();

    // Wait for it to enter its while(!dying) c = fn(cowait()) loop.
    currcoWait();
    if (g.tracing) {
        System.out.println(cptr.coName+": Worker coroutine ready");
    }
}

if (g.tracing) {
  System.out.println(coName+
       ": Returning startup packet to the "+
       r.startuppkt.task.coName);
}

r.qpkt(r.startuppkt);

if (g.tracing) {
    System.out.println(coName+": Initialised");
}
```

```
while (true) {
    // Start of this server’s event loop.

    // Get a client request packet via gomultievent.
    // It should be a read, write or die packet.
    // Bounce and Clock packets belong to worker
    // coroutines and will delivered automatically
    // to their coroutines by gomultievent.

    if (g.tracing) {
     System.out.println(coName+
        ": Waiting for a request from a client via gomultievent");
    }

    t.mainco_ready = true;

    //while (r.serverq==null | !dying) {
    //if (g.tracing) {
    //     System.out.println(coName+
    //        ": Waiting for a serverq to contain a packet");
    // }
    // r.pktcondvar.cocondwait();
    //}
    //if (g.tracing) {
    // System.out.println(coName+
    //    ": Dequeueing a packet from serverq");
    //};
    //Pkt pkt = r.serverq;
    //r.serverq = r.serverq.link;
    //pkt.link = null;

    System.out.println(coName+
            ": Calling cowait(null)");
    Pkt pkt = (Pkt) cowait(null);
    if (pkt==null) {
     System.out.println(coName+
        ": Received null pkt");
     continue;
    }

    t.mainco_ready = false;

    if (g.tracing) {
     System.out.println(coName+
        ": Packet received from gomultievent, type="+strn(pkt.type,3));
    }

    switch (pkt.type) {
    default:
System.out.println(coName+
   ": Unexpected packet received from "+
   pkt.task.coName);
t.qpkt(pkt);
continue;

    case act_read:
    case act_write:
{   Pkt p = r.serverq;
    if (g.tracing) {
System.out.println(coName+
   ": Append pkt to end of this server's serverq");
    }
    pkt.link = null;
    if (r.serverq==null) {
// Make a unit list
r.serverq = pkt;
    } else {
// Append pkt to the end of this server's queue
while (p.link!=null) {
    p = p.link;
}
p.link = pkt;
    }
    realdelay(1000);
    if (g.tracing) {
System.out.println(coName+
   ": Calling notifyAll for condition pktcondvar");
r.pktcondvar.conotifyAll();
    }
    continue;
}

    case act_die: // From the Controller task.
// Cause this server to return to DEAD state.

if (g.tracing) {
    System.out.println(coName+
       ": Die packet received");
}
```

```
r.diepkt = pkt;
// Release all workers on countcondvar.

r.countcondvar.conotifyAll();

// Delete all worker coroutines and the logger.

//for ( int i=1; i<=g.workmax; i++) {
//    killco(wrkcov[i]);
//}
//killco(loggerco);

if (g.tracing) {
    System.out.println(coName+
       ": Entering single event mode");
}

t.multi_done = true;
cowait(null); // Return to the controller.
return null;
    }
}
    }

    public synchronized Pkt waitforserverpkt() {
while (r.serverq==null | !dying) {
    if (g.tracing) {
System.out.println(coName+
   ": Waiting for a serverq to contain a packet");
    }
    r.pktcondvar.cocondwait();
}
if (g.tracing) {
    System.out.println(coName+
       ": Dequeueing a packet from serverq");
    };
Pkt pkt = r.serverq;
r.serverq = r.serverq.link;
pkt.link = null;
return pkt;
    }
}

//*************************************************************************
//************************** Loggermainco ****************************
```

```
//*****************************************************************

class Loggerco extends Cortn {
    ///coroutine
    // This the logger coroutine of a read or write server.
    // It runs in multi-event mode under the control of
    // gomultievent.
    Server r;
    Pkt diepkt = null;

    public Loggerco(String name, Server r) {
super(name, r);
this.r = r; // Allow access to the Server's variables.
    }

    public Object fn(Object x) {
System.out.println(coName+": Ready #################################");

while (true) {
    // Start of the logger loop.
    int i = 0; // Count of values received.

    int a = r.loggerin.coread();
    if (g.tracing) {
System.out.println(coName+
   ": Received a="+a);
    }
    i++;

    // occasionally send a message to the printer task.
    if ((i % 50) == 0) {
if (g.tracing) {
    System.out.println(coName+
       ": Sending pkt to the Printer task");
}
r.sndpkt(new Pkt(null, g.printer, act_print,
       0,0,
       r.modech, r.srvno));
if (g.tracing) {
    System.out.println(coName+
       ": pkt returned from the Printer");
}
    }

    // Occasionally the logger delays briefly.
    if ((i % 7)==0) {
delayco(2); // 2 msecs
    }

    int b = r.loggerin.coread();
    if (g.tracing) {
System.out.println(coName+
   ": Received b="+b);
    }

    { int sum = a+b;
      if (g.tracing) {
  System.out.println(coName+
   ": Replying "+sum);
      }
      r.loggerout.wrpn(sum);
      r.loggerout.cowrite(-1);
    }
}
    }
}

//*******************************************************************
//************************** Loggerlock ******************************
//*******************************************************************

class Loggerlock {
    // pkt is the list of packets waiting for the lock.
    Pkt pkt = null;
}

//*******************************************************************
//************************** Colist ********************************
//*******************************************************************

class Colist {
    // This holds a list coroutines used in Condvar.
    public Colist link;
    public Cortn cptr;

    public Colist(Colist link, Cortn cptr) {
this.link = link;
this.cptr = cptr;
    }
}
```

```
//*******************************************************************
//************************** Condvar *********************************
//*******************************************************************

class Condvar {
    Colist list = null;
    Task t;
    String name;

    public Condvar(String name, Task t) {
// Create a new condition variable belonging to Server r.
this.name = name;
this.t = t;
    }

    public void cocondwait() {
// Typical usage: pktcondvar.condwait();
// Insert a node at the head of list.
list = new Colist(list, t.currco);
System.out.println(name+" Inserted coroutine "+t.currco.coName+" into Condvar list");
System.out.println(name+" Waiting in cocondwait()");
t.cowait(null);
    }

    public void conotifyAll() {
// Typical usage: pkt=pktcondvar.conotifyAll();

// No need to synchronize since only one coroutine of this
// task can run at a time. Note that local data is flushed
// to main memory when control passed from one coroutine
// to another since this involves the calling of the
// synchronized methods unwait and currcowait.

System.out.println(name+": conotifyAll called");

// Release all the coroutines waiting on this condition.
Colist p = list;
if (p==null) {
    System.out.println(name+": conotifyAll p==null");
}
// Clear the list since some may immediately wait on the same
// condition.
list = null;
while (p!=null) {
    System.out.println(name+": conotifyAll calling callco("+p.cptr.coName+",null)");
    t.callco(p.cptr, null);
    p = p.link;
}
    }
}

//***********************************************************************
//*************************** Workerco **********************************
//***********************************************************************

class Workerco extends Cortn {
    ///coroutine
    // This is the main coroutine of a read or write server worker
    // coroutine.

    Server r;
    int wrkno;
    Pkt diepkt = null;
    Pkt pkt = null;

    public Workerco(String name, Server r) {
super(name, r);
this.r = r;
    }

    public Object fn(Object x) {
System.out.println(coName+": fn entered ##########################");
// This is the main function of a worker coroutine belonging
// to a server.

// The first value given to a worker coroutine is its worker number,
// so x is the worker number.
wrkno = ((Integer)x).intValue();

System.out.println(coName+": Initialised and ready, ##########################wrkno="+

while (true) {

    while (r.serverq==null | dying) {
System.out.println(coName+": Worker waiting for next request pkt");
r.pktcondvar.cocondwait();
    }
    if (dying) {
System.out.println(coName+": dying is true");
return null;
    }

    System.out.println(coName+": Extracting pkt from r.serverq");
    Pkt pkt = r.serverq;
    r.serverq = r.serverq.link;
    pkt.link = null;

    // Temp fiddle
    System.out.println(coName+": Temp fiddle #############################");
    System.out.println(coName+": Got pkt from "+pkt.task.coName);
    System.out.println(coName+": Returning to pkt to "+pkt.task.coName);
    pkt.res1 = 1234; // Indicate success.
    r.qpkt(pkt);
    // Wait for another request.
}
    }
}

//*****************************************************************
//*************************** Multiplexor ***************************
//*****************************************************************

class Multiplexor extends Task {
    ///task

    Multiplexor r;         // This is used by the multiplexor
                           // coroutines to access the multiplexor
                           // variables such as mpxrdcov or bufv.

    Pkt startuppkt = null; // This task variable hold the startup
                           // packet. It is returned to the
                           // controller by mpxmainco when
                           // initialisation is complete.

    int mpxno;             // The number of this multiplexor

    int mpxbusycount = 0;  // The number of channel coroutines
                           // that are currently busy (ie not
                           // waiting for requests.

    Pkt diepkt = null;     // This will contain the die packet
                           // when received from the controller.
    Mpxrdco mpxrdcov[] = null; // To hold the channel read coroutines.
    Mpxwrco mpxwrcov[] = null; // To hold the channel write coroutines.
    boolean rdbusyv[]  = null;
    boolean wrbusyv[]  = null;
    Pkt rdwkqv[]       = null;
    Pkt wrwkqv[]       = null;
    Channelbuf bufv[]  = null;

    public Multiplexor(String name, Globals g, int pri) {
// This constructor is used to create a multiplexor task and
// its root coroutine.
super(name, g, pri);
r = this;
        System.out.println(coName+": Task created");
    }

    public Object fn(Object arg) {
// This is the main function of a multiplexor root coroutine.

        startuppkt = (Pkt) arg; // Used by mpxmainco
mpxno = startuppkt.arg1;
//int count = 0;

        //System.out.println(coName+": startup packet received from "+startuppkt.task.

// Create the multiplexor mainco for gomultievent.

Mpxmainco mainco = new Mpxmainco(coName+"mainco", r);

mainco.parent = this;
mainco.value = null;
currco = mainco;

//System.out.println(coName+": Starting coroutine "+mainco.coName);
mainco.start();

// Wait for it to enter its while(!dying) c = fn(cowait()) loop.
currcoWait();

//System.out.println(coName+": Coroutine "+mainco.coName+" created");

gomultievent(mainco);

return null;
```

```
    }
}

//*********************************************************************
//*************************** Mpxmainco ****************************
//*********************************************************************

class Mpxmainco extends Cortn {
    ///coroutine

    Multiplexor r;
    Pkt diepkt;

    public Mpxmainco(String name, Multiplexor r) {
// This constructor creates a non-root coroutine belonging to
// a multiplexor task. The only possible coroutines are channel
// read and write coroutine.
super(name, r);
this.r = r;
    }

    public Object fn(Object arg) {
// This is the main function of a multiplexor mainco
// called from gomultievent.

        //System.out.println(coName+": Multiplexor's mainco entered");

// Allocate all the multiplexor vectors.

r.mpxrdcov = new Mpxrdco[g.chnmax+1];
r.mpxwrcov = new Mpxwrco[g.chnmax+1];
r.rdbusyv  = new boolean[g.chnmax+1];
r.wrbusyv  = new boolean[g.chnmax+1];
r.rdwkqv   = new Pkt[g.chnmax+1];
r.wrwkqv   = new Pkt[g.chnmax+1];
r.bufv     = new Channelbuf[g.chnmax+1];

for (int chnno=1; chnno<=g.chnmax; chnno++) {
    r.mpxrdcov[chnno] = null;
    r.mpxwrcov[chnno] = null;
    r.rdbusyv[chnno]  = false;
    r.wrbusyv[chnno]  = false;
    r.bufv[chnno]     = null;
}
```

```
// Allocate the channel buffers
for (int chnno=1; chnno<=g.chnmax; chnno++) {
    Channelbuf buf = new Channelbuf(g.chnbufsize);
    r.bufv[chnno] = buf;
    if (g.tracing) {
System.out.println(coName+
   ": Created channel buffer "+strz(chnno,2)+
   ", size="+g.chnbufsize);
    }
}

// Create the multiplexor read channel coroutines.
for (int chnno=1; chnno<=g.chnmax; chnno++) {
    Mpxrdco mpxrdco = new Mpxrdco("M"+strz(r.mpxno,2)+"RC"+strz(chnno,2), r);
    // Start mpxrdco leaving it susprnded in main loop:
    // while(!dying) c = fn(cowait());

    mpxrdco.parent = this;
    mpxrdco.value = null;
    t.currco = mpxrdco;

    mpxrdco.start();
    // Wait for it to enter its while(!dying) c = fn(cowait()) loop.
    currcoWait();

    // Tell mpxrdco what its channel number is.
    callco(mpxrdco, new Integer(chnno));

    // Wait for it to respond.
    //if (g.tracing) {
    //    System.out.println(mpxrdco.coName+": Read coroutine started for channel "+ch
    //}
}

// Create the multiplexor write channel coroutines.
for (int chnno=1; chnno<=g.chnmax; chnno++) {
    Mpxwrco mpxwrco = new Mpxwrco("M"+strz(r.mpxno,2)+"WC"+strz(chnno,2), r);
    // Start mpxwrco leaving it susprnded in main loop:
    // while(!dying) c = fn(cowait());

    mpxwrco.parent = this;
    mpxwrco.value = (Object)(new Integer(123));
    t.currco = mpxwrco;
    mpxwrco.start();
    // Wait for it to enter its while(!dying) c = fn(cowait()) loop.
    currcoWait();

    // Tell mpxwrco what its channel number is.
    callco(mpxwrco, new Integer(chnno));
    //if (g.tracing) {
    //    System.out.println(mpxwrco.coName+": Write channel coroutine started");
    //}
}

//System.out.println(coName+
//    ": Returning the startup packet to task "+
//    r.startuppkt.task.coName);
        r.qpkt(r.startuppkt); // Return the startup packet

if (g.tracing) {
    //System.out.println(coName+": All read and write channel coroutines ready");
    System.out.println(r.coName+": Initialised");

}

        while(true) {
    // Get the next request from gomultievent.

    t.mainco_ready = true;
    // mainco_ready is only true when mainco is waiting for
    // a read, write or die request.
    Pkt pkt = (Pkt) cowait(null);

    t.mainco_ready = false;

    switch (pkt.type) {
    default:
System.out.println(coName+
   ": Unexpected packet received from "+pkt.task.coName);
continue;

    case act_read:
    {   // Packet arguments:
// a1:flag ar:clino a3:serno a4:mpxno a5:chnno
char flag = (char) pkt.arg1;
int clino = pkt.arg2;
int serno = pkt.arg3;
int mpxno = pkt.arg4;
int chnno = pkt.arg5;
int data  = pkt.arg6;  // =0 for read requests

if (g.tracing) {
    System.out.println(coName+
                              ": Read request received");
}

// Insert this packet at the head of the read wkq
// for this channel.

pkt.link = r.rdwkqv[chnno];
r.rdwkqv[chnno] = pkt;

if (g.tracing) {
    System.out.println(coName+
                              ": Read request inserted at head of its wkq");
}

if (!r.rdbusyv[chnno]) {
    // Wake up the channel read coroutine
    // if it is not busy.
    callco(r.mpxrdcov[chnno], null);
}

// Process another read, write or die request
continue;
    }

    case act_write:
    { // Packet arguments:
// a1:flag ar:clino a3:serno a4:mpxno a5:chnno
char flag = (char) pkt.arg1;
int clino = pkt.arg2;
int serno = pkt.arg3;
int mpxno = pkt.arg4;
int chnno = pkt.arg5;
int data  = pkt.arg6;

if (g.tracing) {
    System.out.println(coName+
                              ": Write request received");
}

// Insert this packet at the head of the write wkq
// for this channel.

pkt.link = r.wrwkqv[chnno];
r.wrwkqv[chnno] = pkt;

if (g.tracing) {
    System.out.println(coName+
                               ": write request inserted at head of its wkq");
}

if (!r.wrbusyv[chnno]) {
    // Wake up the channel write coroutine
    // if it is not busy.
    callco(r.mpxwrcov[chnno], null);
}

// Process another read, write or die request
continue;
    }

    case act_die:
System.out.println(coName+": die pkt received");
diepkt = pkt;
break;     // Cause this thread to die
    }

    // Reach here if a die packet was received
    break; // Break out of the while(true) loop
}

// Cause this multiplexor to die by returning control
// to gomultievent with multi_done set to true.

r.multi_done = true;
return null;
    }
}

//**********************************************************************
//************************** Mpxrdco ********************************
//**********************************************************************

class Mpxrdco extends Cortn {
    ///coroutine
    // This models a multiplexor read channel coroutine.
    Channelbuf buf;
    Multiplexor r;

    char flag;
    int clino;
    int serno;
    int mpxno;
    int chnno;
    int data;

    Pkt pkt;

    public Mpxrdco(String name, Multiplexor r) {
// This constructor creates a non-root read channel coroutine
// belonging to a multiplexor task.
super(name, r);
this.r = r;          // Multiplexor t owns this coroutine and
                             // provides access to variables
                             // such as mpxrcv and bufv.

//System.out.println(coName+": Multiplexor "+r.coName+" owns this coroutine");
    }

    public Object fn(Object x) {
// This is the main function of a channel read coroutine

mpxno = r.mpxno;
chnno = ((Integer)x).intValue();
buf = r.bufv[chnno];
//System.out.println(coName+": Channel read coroutine ready for channel "+chnno);

r.rdbusyv[chnno] = true; // ONLY = false when waiting to be woken up.

while (true) {
    // Start of the main loop for a read coroutine for this channel.

    // Get the next read packet, waiting if necessary.

    if (r.rdwkqv[chnno]==null) {
// There are no packets in the queue for this channel
// so wait for one
r.rdbusyv[chnno] = false;
r.mpxbusycount--;
if (g.tracing) {
    System.out.println(coName+": Ready");
}

pkt = (Pkt)cowait(null);
r.rdbusyv[chnno] = true;
r.mpxbusycount++;
    } else {
if (g.tracing) {
    System.out.println(coName+
       ": Dequeuing a pkt from "+r.rdwkqv[chnno]);
};

// Dequeue the next read packet
pkt = r.rdwkqv[chnno];
r.rdwkqv[chnno] = pkt.link;
pkt.link = null;
    }

    // Extract the parameters
    flag  = (char) pkt.arg1;
    clino = pkt.arg2;
    serno = pkt.arg3;
    data  = pkt.arg6;

    if (buf.isempty()) {
// The channel buffer is empty, so return the packet
// to the read server with an indication of failure.
if (g.tracing) {
    System.out.println(coName+
       ": Returning failed read request to its server");
}
pkt.res1 = 0;  // Indicate failure.
t.qpkt(pkt);
continue;
    }

    data = buf.get();
    if (g.tracing) {
System.out.println(coName+
   ": data "+strz(data,4)+
   " extracted from the channel buffer");
    }

    // Conditionally perform a multiplexor delay.
    if (flag=='m') {
if (g.tracing) {
    System.out.println(coName+
       ": Channel delay*n");
};

delayco(g.delaymsecs);

if (g.tracing) {
    System.out.println(coName+
       ": Channel delay done*n");
}
    }

    // Return the successful read request packet to its server.
    if (g.tracing) {
System.out.println(coName+
   ": returning successful read packet to its server");
    }
    t.qpkt(pkt); // Return the successful read packet to its server.

    // Process the next read packet, if any.
}
    }
}

//************************************************************************
//************************** Mpxwrco *********************************
//************************************************************************

class Mpxwrco extends Cortn {
    ///coroutine
    // This is the main coroutine of a read server

    Multiplexor r;
    Channelbuf buf; //

    char flag;
    int clino;
    int serno;
    int mpxno;
    int chnno;
    int data;

    Pkt pkt;
    public Mpxwrco(String name, Multiplexor r) {
// This constructor creates a non-root write channel coroutine
// belonging to a multiplexor task.
super(name, r);
this.r = r;
    }

    public Object fn(Object x) {
// This is the main function of a channel write coroutine

mpxno = r.mpxno;
chnno = ((Integer)x).intValue();
buf = r.bufv[chnno];
//System.out.println(coName+": Channel write coroutine ready for channel "+chnno);

r.wrbusyv[chnno] = true; // ONLY = false when waiting to be woken up.

while (true) {
    // Start of the main loop for a write coroutine for this channel.

    // Get the next write packet, waiting if necessary.

    if (r.wrwkqv[chnno]==null) {
// There are no packets in the queue for this channel
// so wait for one
r.wrbusyv[chnno] = false;
r.mpxbusycount--;

if (g.tracing) {
    System.out.println(coName+": Ready");
}

pkt = (Pkt) cowait(null);
r.wrbusyv[chnno] = true;
r.mpxbusycount++;
    } else {

// Dequeue the next write packet
pkt = r.wrwkqv[chnno];
r.wrwkqv[chnno] = pkt.link;
pkt.link = null;
    }

    // Extract the parameters
    flag  = (char) pkt.arg1;
    clino = pkt.arg2;
    serno = pkt.arg3;
    data  = pkt.arg6;

    if (buf.isfull()) {
// The channel buffer is full, so return the packet
// to the write server with an indication of failure.
if (g.tracing) {
    System.out.println(coName+
       ": Returning failed write request to its server");
}
pkt.res1 = 0;  // Indicate failure.
t.qpkt(pkt);
continue;
    }

    buf.put(data);
    if (g.tracing) {
System.out.println(coName+
   ": data "+strz(data,4)+
   " put into the channel buffer");
    }

    // Conditionally perform a multiplexor delay.
    if (flag=='m') {
if (g.tracing) {
    System.out.println(coName+
       ": Channel delay*n");
};

delayco(g.delaymsecs);

if (g.tracing) {
    System.out.println(coName+
       ": Channel delay done*n");
}
    }

    // Return the successful read request packet to its server.
    if (g.tracing) {
System.out.println(coName+
   ": returning successful write packet to its server");
    }
    t.qpkt(pkt); // Return the successful read packet to its server.

    // Process the next read packet, if any.
}
    }
}

//*****************************************************************
//************************* Channelbuf ****************************
//*****************************************************************

class Channelbuf {
    // This class is used to represent channel buffers.
    // It methods are: isempty, isfull, get and put.
    // These metods are synchonized since different threads may try
    // to access the buffer at the same time.

    int chnbufsize;
    int buf[];
    int p;
    int q;

     public Channelbuf(int size) {
 // This will construct a channel buffer of given size.
 this.chnbufsize = size;
 this.buf = new int[chnbufsize];
 p=0; // Initially empty.
 q=0;
    }

    public synchronized boolean isempty() {
return p==q;
    }

    public synchronized boolean isfull() {
return (p+1)%chnbufsize == q;
    }

    public synchronized int get() {
int data = buf[p];
        p = (p+1) % chnbufsize;
return data;
    }

    public synchronized void put(int data) {
        q = (q+1) % chnbufsize;
buf[q] = data;
    }

}


//*******************************************************************
//************************** Rnd ************************************
//*******************************************************************

class Rnd {
     private int seed;

     public Rnd(int a) {
 seed = a & 0xFFFFFFF | 1; // Ensure seed is not zero
         int k = next(50) + 10;
         for(int i = 1; i<=k; i++) next(1000);
    }

  //public final static int feedback = 0xD008;     // 16 15 13 4  => period 2^16-1
    public final static int feedback = 0x80200003; // 32 22  2 1  => period 2^32-1

    // 1000 0000 0010 0000 0000 0000 0000 0011
    // |           |                        ||
    // |           |                        |1
    // 32          22                       2

    public int next(int max) {
if ((seed&1)==0) {
    seed = seed>>>1;
        } else {
            seed = (seed>>>1) ^ feedback;
        }
        return (seed>>>1) % max + 1;
    }
}
```