Classical concurrency control (Low-Level): Overview

Controlling access by concurrent processes to shared writeable data has been studied *as part of OS design* since the earliest OSs (1960s onwards).

Concurrent programming languages brought the same problems
to *application programming*.
For example, web servers have to handle large numbers of concurrent requests.

Our starting point – how to think about the problem:

critical regions (CRs): regions of code in concurrent processes
that read or write shared data
and have to be executed sequentially, under mutual exclusion

How to implement CRs (using flags or semaphores)
- without blocking: processes "spin-lock" or "busy-wait" on a flag for their turn to execute the CR
- with blocking: processes that must then wait for their turn, on semaphores

Classical, shared memory, concurrency control: Overview 2

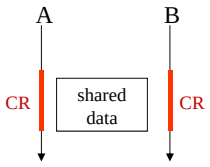
We then look at how semaphores can be used:
1. a single semaphore used to achieve mutual exclusion
2. a single semaphore used to achieve condition synchronisation
3. a single semaphore used for N-resource allocation

Then, how semaphores are implemented.
This may be within the OS, or at application level, in the runtime system of a concurrent programming language

Programming with semaphores, using several semaphores to achieve both mutual exclusion and condition synchronisation
1. single producer, single consumer processes communicating via a shared buffer
2. many producers and consumers
3. readers and writers: multiple readers, single writer concurrency control

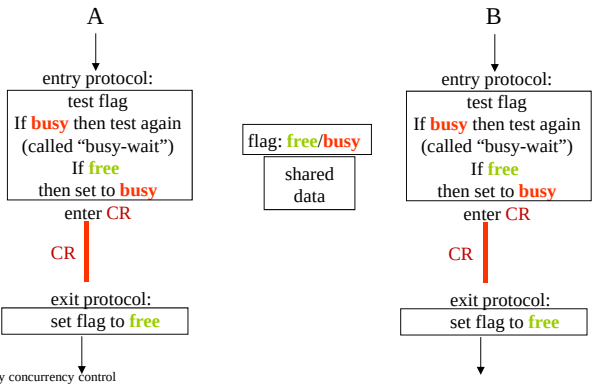
Discussion of semaphore programming: problems/difficulties

Critical regions



Processes A and B contain critical regions (CRs)
(code that reads or writes this shared data)
CRs are needed only if the data is writeable
A CR is associated with some specific shared data

How can CRs be implemented? – first attempt: associate a boolean flag with this shared data



Indivisible test-and-set

The entry protocol is correct only if test and set of flag are atomic/indivisible – HOW?

- forbid interrupts? – NO – this would only work on a uniprocessor, and even then would be inappropriate for general use.
- machine instruction? - YES
- program only – no hardware exclusion - ?NO

CISC machines had many ***read-memory, test result, store-to-memory*** types of instruction
RISC (load/store) architectures can only use a single memory access per instruction

read-and-clear will work:

flag=0 //shared data is busy
flag=1 //shared data is free (initial value)

entry protocol:
read-and-clear, register flag
// if value in register is 0, shared data was busy so retry
// if value in register is 1, shared data was free and you claimed it

We can assume modern computers have such instructions

Mutual exclusion without hardware support

This was a hot topic in the 1970s and 80s.
Examples for N-process mutual exclusion are:

Eisenberg M. A. and McGuire M. R.,
Further comments on Dijkstra's concurrent programming control problem
CACM 15(11), 1972

Lamport L
A new solution to Dijkstra's concurrent programming problem
CACM, 17(8), 1974
(his N-process bakery algorithm)

For uniprocessors and multiprocessors these algorithms impose huge overhead.
In practice, OSs built mutual exclusion on atomic hardware instructions.

It is not proven that these programs are correct for multicore

Dijkstra THE 1968

The entry protocols above involve busy-waiting (retry in a loop if flag is busy), wasting CPU time
It is usually better to block a waiting process

Define a new type of variable – semaphore
Operations for the type are:

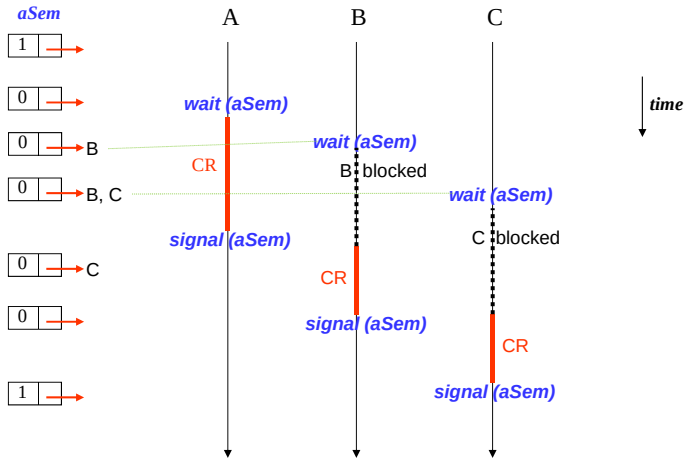
wait (aSem)
if aSem > 0 then aSem = aSem – 1
else suspend the executing process, waiting on aSem

signal (aSem)
if there are no processes waiting on aSem
then aSem = aSem + 1
else free one waiting process – continues after its wait instruction

Implementation: an integer and a queue

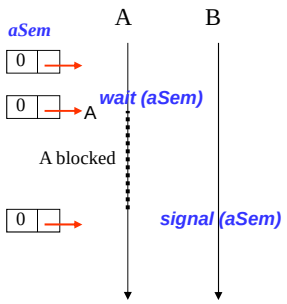
Mutual exclusion using a semaphore

concurrent processes: **serialisation** of critical regions

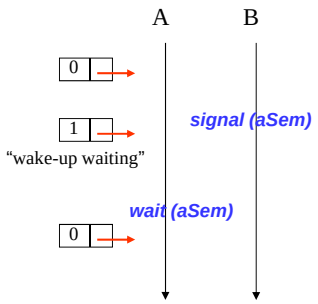


Two-process synchronisation

wait before *signal*



signal before *wait*

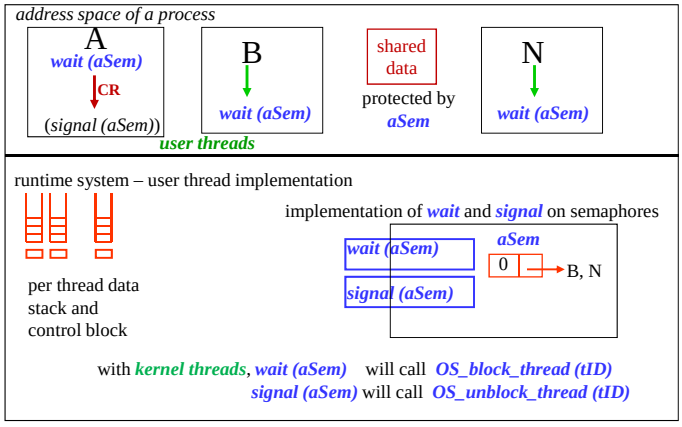


N-resource allocation using a semaphore

Suppose there are N instances of a resource.
Control its allocation using a semaphore ***resSem*** initialised to N.
Each time a process executes ***wait (resSem)*** the semaphore's value is decremented.
When the value is 0 (after N ***wait***s without any ***signal***s),
all subsequent processes executing ***wait (resSem)*** are queued on ***resSem***
until freed by a current user of the resource executing ***signal (resSem).***

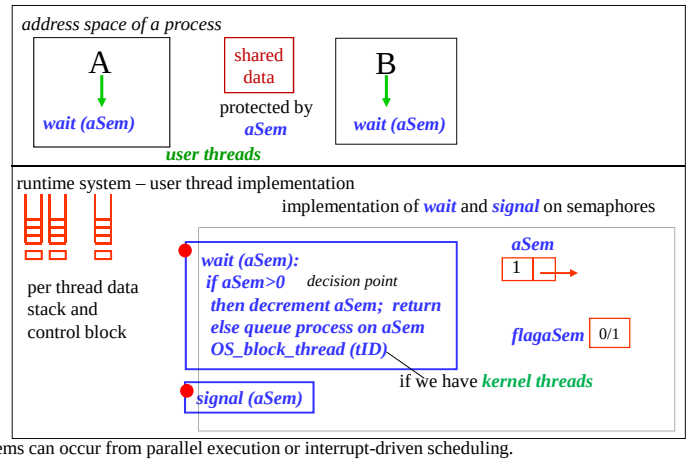
Implementation of semaphores - 1

Here we show a snapshot where A has claimed exclusive access to the shared data via ***wait(aSem)***
and B and N have executed ***wait(aSem)*** and have been blocked (queued on ***aSem***).
But note that ***aSem*** is shared writeable data and ***wait*** is a composite operation
(check value of ***aSem***, decrement and return or queue process on ***aSem***).
Errors could occur!



Implementation of semaphores - 2

Suppose A executes ***wait(aSem)*** then immediately (in parallel) B executes ***wait(aSem).***
Note that ***aSem*** is shared writeable data and ***wait*** is a composite operation. Race conditions are possible……



…. problems can occur from parallel execution or interrupt-driven scheduling.
In the system implementation, ***wait*** and ***signal*** are short and trusted
so protect them by a ***spin-lock*** (busy wait) on a flag, implemented by a hardware instruction (see slide 4) .

Implementation of semaphores -3

For user-threads only (OS sees a single-threaded process) the runtime system does all semaphore and user thread management – no problem from concurrency of ***wait*** and ***signal***,
We have single-threaded execution of the runtime.

When user threads are mapped to kernel threads, ***wait*** and ***signal*** must themselves be made ***atomic operations.*** This is clearly the case for a multiprocessor (parallel execution), and also for a uniprocessor with preemptive scheduling**.**

Solution: Associate a flag (boolean) with each semaphore, and use an atomic hardware instruction such as ***read-and-clear*****,** see slide 5.
The flag must be claimed before a ***wait*** or ***signal*** can be executed for that semaphore.

This also applies to kernel threads executing the OS and using OS- managed semaphores for mutual exclusion and condition synchronisation for shared OS data .

The need for concurrency control first came from OS design. We now have concurrent programming languages for OSs and applications and OSs supporting multi-threaded processes.

Semaphore programming

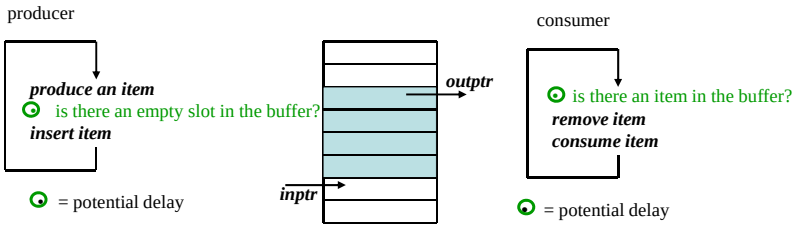
We now develop some concurrent programs that use a number of semaphores for mutual exclusion and condition synchronisation.

1. One producer, one consumer: Two processes communicate through an N-slot cyclic buffer.
   One process inserts records of fixed size, the other removes them.
   Condition synchronisation is needed for when the buffer is full and empty.
   They can run in parallel, accessing different parts of the buffer
   (no need for mutually exclusive access to the buffer).

2. *Any number of producer and consumer processes* communicating
   via the buffer. We now need to ensure mutually exclusive access to the buffer.

3. Readers and writers:
   We note that processes that only read shared data can read simultaneously,
   whereas a process that writes must have exclusive access to the data.
   We develop a solution that gives priority to writers over readers,
   on the assumption that writers are keeping the data up-to-date.

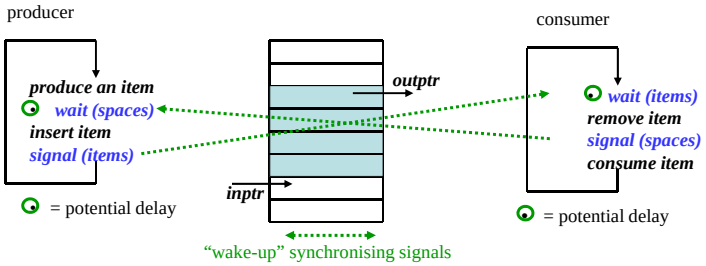
N-slot cyclic buffer, single producer and consumer - 1



two semaphores are needed
- one for the producer to block on ***wait***, when the buffer is full
- one for the consumer to block on ***wait***, when the buffer is empty
- note: blocked processes must be unblocked via ***signal*** on semaphores

N-slot cyclic buffer, single producer and consumer - 2

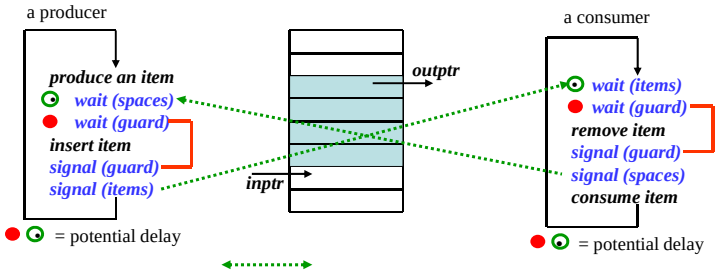
two semaphores are needed:
spaces = N // initially N spaces in buffer, - for the producer to block on when the buffer is full.
items = 0 // initially no items in buffer - for the consumer to block on when the buffer is empty.



programming details are not shown – we focus on condition synchronisation

programming note: ***insert item*** must increment ***inptr*** to point to the next empty slot
remove item must increment ***outptr*** to point to the next item (full slot)
when ***inptr = outptr*** the buffer is either full or empty. You may also keep
an ***integer count*** of the number of items in the buffer: ***count = 0*** (empty) ***count = N*** (full).

N-slot cyclic buffer, many producers and consumers



three semaphores are used:
spaces = N // initially N spaces in buffer - for the producer to block on when the buffer is full
items = 0 // initially no items in buffer - for the consumer to block on when the buffer is empty
guard = 1 // initially the buffer is free - to ensure mutually exclusive access to the buffer
programming notes – as in previous slide

variation: – allow one producer and one consumer to access the buffer in parallel – left as an exercise

Multiple readers, single writer concurrency control -1

Many readers may read simultaneously, a writer must have exclusive access
Assume writers have priority – to keep the data up-to-date.

counts:

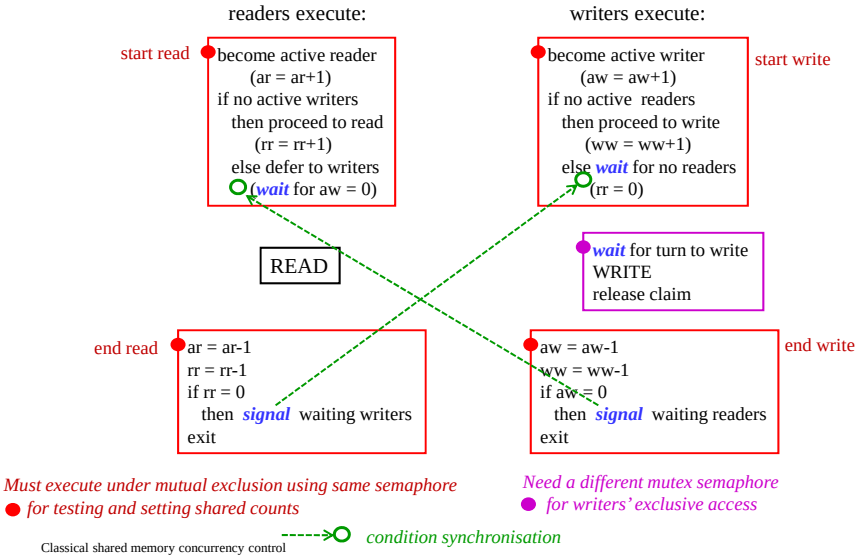
- ar = active readers
- rr = reading readers (active readers who have proceeded to read)
- aw = active writers
- ww = writing writers (active writers who have proceeded to write)
  but they must wait to write one-at-a-time

Semaphores are needed:

- for mutual exclusion ●
  1. to test and update the above counts under exclusion
  2. to ensure writers write under exclusion
- for condition synchronisation ○
  1. readers must wait for aw = 0 and must be woken up after blocking
  2. writers must wait for rr = 0 and must be woken up after blocking

Multiple readers, single writer concurrency control -2

Outline of components of a possible solution (several have been published – see textbooks)



MRSW-3: Beware a naïve implementation:

become active reader
(ar = ar+1)
if no active writers (aw=0)
then proceed to read
(rr = rr+1)
else defer to writers
(*wait* for aw = 0)

wait (CountGuardSem)

signal (CountGuardSem)

Suppose the critical regions that control access to the counts are implemented using a semaphore ***CountGuardSem*** initialised to 1

wait (CountGuardSem)
ar = ar+1
if aw=0 then rr = rr+1
else ***wait (ReadersSem)***
signal (CountGuardSem)

within a critical region the counts may indicate that the process must block until some condition becomes true
deadlock! blocking while holding a semaphore

So *the programmer* has to program to avoid deadlock.
A process that must delay must exit the region before blocking on the condition.
In this case, ***wait (ReadersSem)*** must be executed after ***signal (CountGuardSem)***
but beware concurrency problems between releasing ***CountGuardSem*** and blocking on ***ReadersSem***.
Race conditions could occur

Multiple readers, single writer concurrency control - 4

Complete the program as an exercise. Solutions are in textbooks.

Note that a ***signal*** unblocks only one blocked process. The values of the counts indicate how many signals to send. The last writing writer must unblock all blocked readers. The last reading reader must unblock all waiting writers.

Take care not to ***wait*** while holding the semaphore that protects the shared counts - see previous slide.
That would cause deadlock; no other process could ever access the counts, so could never make the awaited condition true and wake any waiting processes. The deadlocked system would exhibit queues of processes waiting on the various semaphores.

Semaphores - discussion

Semaphores are a widely used mechanism underlying concurrency control in operating systems and concurrent programs

Difficult for programmers to use correctly – programs are complex
- can forget to ***wait*** and corrupt data
- can forget to ***signal*** and cause deadlock

Unconditional commitment to block
- but can ***fork*** a new thread before ***wait***ing to achieve potential concurrent work.

Unbounded delay on wait.

Priority inversion and convoy effect
- a low priority process with a lock can hold up higher priority processes
- a long lock-hold can hold up a lot of potentially short ones

Some of these problems have been addressed by variations in semaphore implementations
e.g. semaphore queue could be ordered by process priority rather than FCFS
e.g. priority inheritance (lock-holder executes at priority of "head waiter")

Event Counts and **Sequencers** are used in some systems at this level of concurrency control.

We now consider hiding the details and problems of semaphore programming by giving the high-level-language programmer easier-to-use concurrency control primitives.

Summary

Studied problem of protecting writeable shared data when accessible by concurrent processes
Critical regions of code access shared data
Associate a different flag or semaphore with each shared data item
Problem if test and set are not indivisible
– need a hardware instruction to test and set (or similar)
Defined semaphores to incorporate process blocking as well as indivisible test-and-set
Looked at problem of implementing semaphore operations correctly under concurrent execution
Attempted semaphore programming
Discussed difficulties

NEXT
Making concurrent programming easier in high-level languages