Concurrent and Distributed Systems

8 lectures on Concurrent Systems, Michaelmas term 2014
Prof Jean Bacon (standing in for Dr Robert N. M. Watson)
Colour copies of slides are on the course materials page.
Also, Dr Watson's slides from last year
Dr Watson will give the Case Study lecture *Weds Oct 22nd TBC*

- 8 lectures on Distributed Systems (Lent term 2015)
  Dr Robert Watson

- Builds on Part 1A Operating Systems concepts

(coloured notes are expensive – bring some coloured pens?)

Concurrent Systems

1. Introduction and overview
   Concurrency in and supported by OS. Thread models.

2. Shared memory – low level concurrency control

3. Shared memory – high-level language concurrency control
3a. Lock-free programming, if time allows (not to be examined)

4. Inter-process communication with no shared memory

5. Liveness properties – Deadlock

* ←
6. Transactions: composite operations on persistent objects *(Thurs. Oct 23rd)*

7. Concurrency control and recovery for transaction systems

* (8). FreeBSD case study
will be given *Weds Oct 22nd (TBC)* by Dr Watson

Concurrent and Distributed Systems Introduction

- "8 lectures on concurrency control in centralised systems" (with FJava)
  - concurrent execution of software components in main memory
  - concurrent executions involving main memory and persistent storage (concurrency control and crashes)
- 8 lectures on distributed systems (Lent term 2015, after some comms.)

Let's look at the total system picture first

How do distributed systems differ fundamentally from centralised systems?

Fundamental characteristics of *distributed* systems

1. Concurrent execution of components on different nodes
2. Independent failure modes of nodes and connections
3. Network delay between nodes
4. No global time – each node has its own clock

Developers of distributed software have to live with these properties

Some implications: to be studied in lectures 9 – 16

1 Nodes and connections may fail or may just be congested or slow
  - how to program for this and tell the difference?

1, 3 Inconsistent views of state/data when it's copied and distributed
  - can't wait for "no activity" to resolve inconsistencies

4 Timestamps generated by different nodes can't be ordered

What are the fundamental problems for a single node?

single node characteristics cf. distributed systems

1. Concurrent execution of components within a single node YES
2. Failure modes - all components crash together,
   but disc failure modes are independent – *similar concept to DS*
3. Network delay is not a concern, but:
   - data structures on disc are copied to main memory and updated
   - uncertainty about what updates have reached disc are
   like concerns about distributed/replicated data.
4. A node has a single clock
   – event ordering is not a problem in a single node

single node characteristics 1: concurrent execution

Some old systems, e.g. original UNIX, assumed ***uniprocessor*** operation.

Concurrent execution of components is achieved on uniprocessors by ***interleaving*** OS level and user-level components.

Multiprocessors are now the norm – we shall assume that **real concurrent execution** of processes/threads is possible.

Recall **interrupt-driven scheduling** of components: **non-preemptive** or **preemptive**. The latter creates most potential flexibility and most difficulties.

Multi-core instruction sets are being examined in detail and instruction ordering is being found to be problematic sometimes (sequential consistency).

This course will assume sequential ordering of atomic machine-level instructions.

single node characteristics 2: failure modes

Failure modes: all in-memory components crash together when main memory is lost but disc is not affected by a main memory failure

e.g. on main-memory failure during an interaction to ***write*** to disc:

write (fileID) is a composite operation:

* find free block(s) from free-block list
* update file metadata (inode in Unix) to record new block(s)
* transfer data

Updates are made to copies of data structures in main memory.

Even when they are written to disc, they may stay in an in-memory buffer for some time, and writes be re-ordered.

The system restart procedure needs to check/restore consistency on disc.

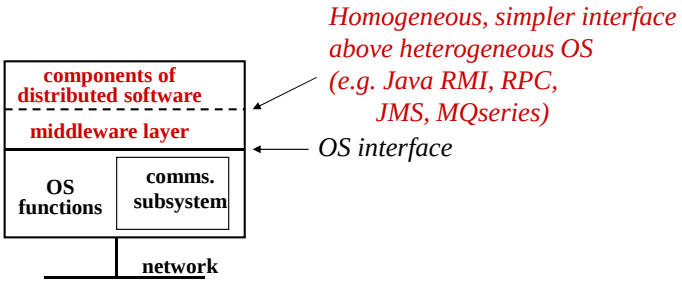
In **lectures 6,7** we consider programs that operate on persistent data on disc.
We define transactions: composite operations
in the presence of both concurrent execution and crashes.

Need for concurrency control in Operating Systems

Concurrency control was first studied for OS and later for programming languages.
Let's see where concurrency occurs in OS and how problems might arise.

CDS: single node as DS component (for lectures 9-16)

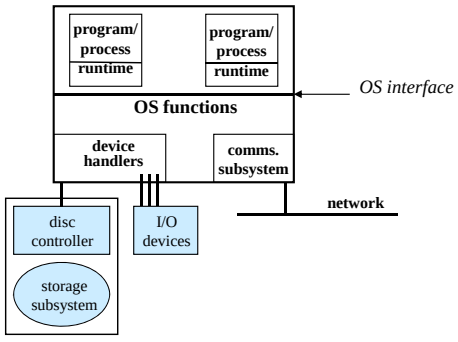
Support for distributed software components is by a software layer (middleware) above potentially heterogeneous OS



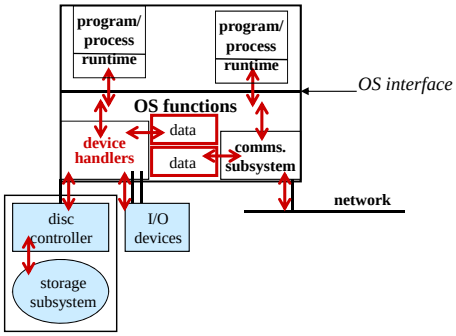
We first consider a single node's software structure and dynamic execution

single node: some software components

- Software structure
- Support for persistent storage
- *Dynamic concurrent execution ?* – see next slide

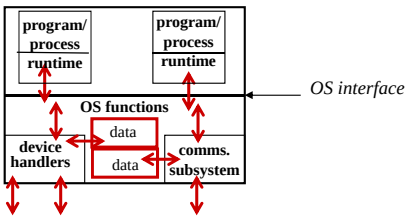


single node – concurrent execution



- note shared data areas in OS: data buffers between devices and process-level data structures on process status etc.
- also, at application level, programs may share data
- also, "threads" (see later) in a concurrent program may share data

Interaction of concurrent execution and process scheduling

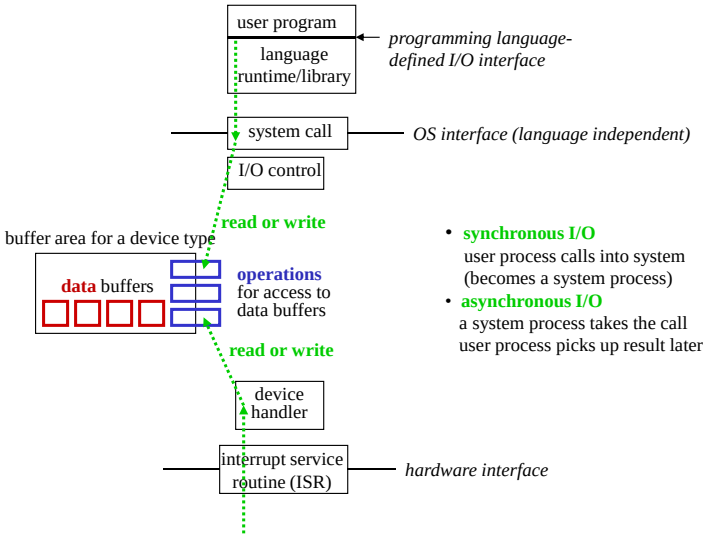


RECALL from part 1A OS:

- assume preemptive scheduling
- interrupt-driven execution – devices, timers, system calls generate interrupts
- OS processes have static priority, page fault > disc > …. > system call handling
- OS process priority higher than application process priority, including system calls

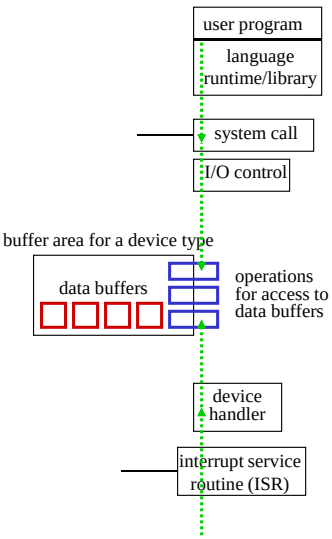
PROBLEM: process preemption while reading/writing shared data

Examples: some OS components and processes - 1



- **synchronous I/O**
  user process calls into system
  (becomes a system process)
- **asynchronous I/O**
  a system process takes the call
  user process picks up result later

Examples: some OS components and processes - 2



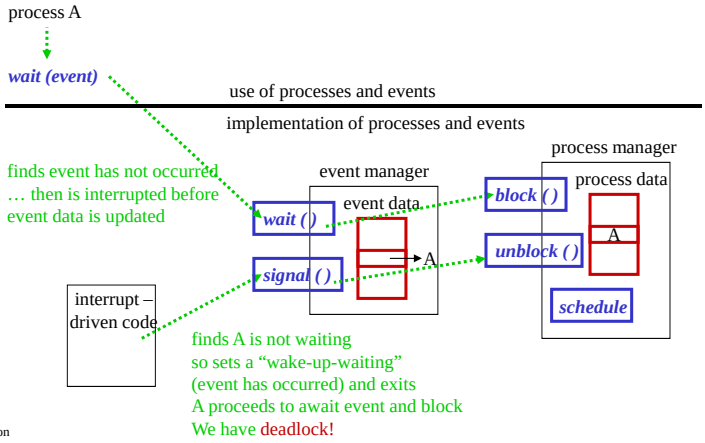
- note priority of device handlers > priority of user calls
- interrupts are independent of process execution
- top-down access could be preempted
  by interrupt- driven, bottom-up access,
  resulting in deadlock or incorrect data.

- so buffers must be accessed under mutual exclusion
  (how? – see later) *but this is not enough* -

- condition synchronisation is also needed:
  process gets mutex access to buffer (how? - see later)
  process finds buffer full on write or empty on read
  process must BLOCK until space or data available
  process ***must not block while holding mutex*** access
  (else we have deadlock!)

Examples: some OS components and processes - 3

- interrupts are independent of process execution
- interrupt-driven code may preempt calls to ***wait*** (Unix ***sleep***) and ***signal*** (Unix ***wakeup***)
- animation first shows intended operation (A recorded) then race condition and deadlock
- so ***wait*** and ***signal*** must be atomic operations



We now look at OS support for multi-threaded processes.

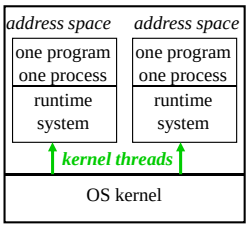
terminology

user threads: defined in a concurrent program

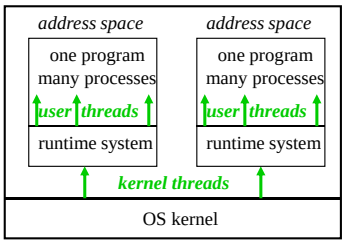
kernel threads: supported and scheduled by the OS

Processes and threads

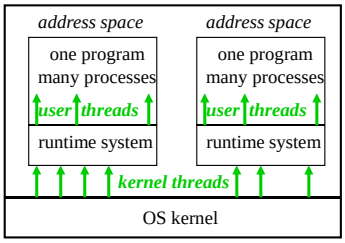
a) Sequential programming languages



b) Concurrent programming language, no OS support (user threads only)

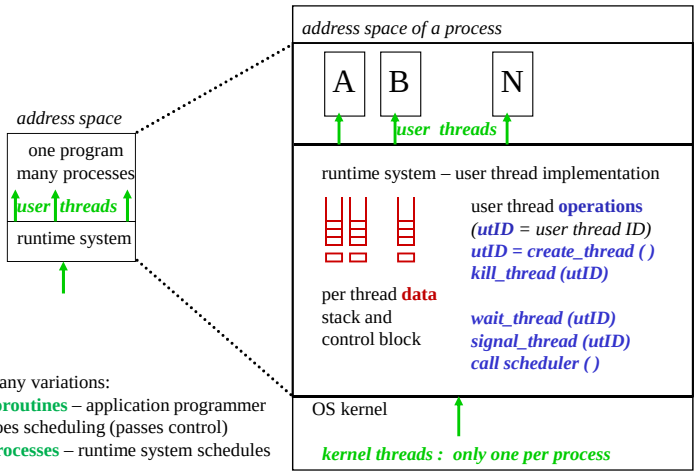


c) Concurrent programming language,
OS kernel threads for user threads



Runtime system - user threads only

17 b) Concurrent programming language, no OS support (user threads only)



see later
wait and *signal*
could be on a mutex,
not on a thread

many variations:
coroutines – application programmer does scheduling (passes control)
processes – runtime system schedules

The OS does not schedule these threads,
but schedules only one process for this program

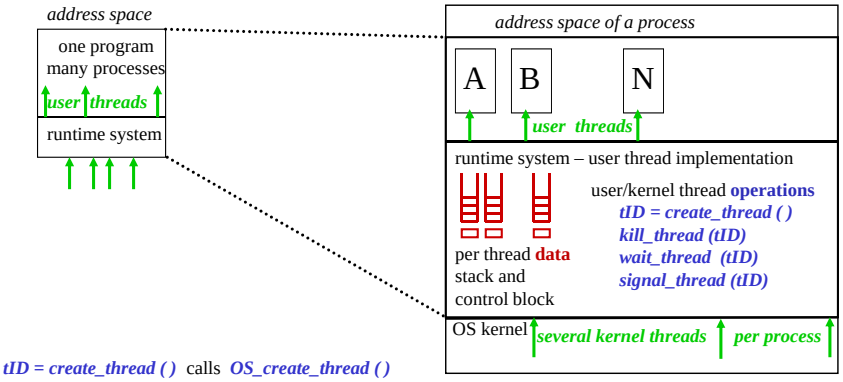
user threads only - implications

1. the application can't respond to OS events by switching user-threads
2. can't use for real-time applications – delay is unbounded
3. the whole process is blocked if any thread makes a system call and blocks
4. applications can't exploit a multiprocessor. The OS knows, and can schedule, only one kernel thread
5. BUT handling shared data in the concurrent program is simple. There is no user-thread preemption i.e. threads are ONLY switched on calls to the runtime system.

   After an interrupt, control returns to the point in the program execution at which the interrupt occurred.

Runtime system - kernel threads

17 c) Concurrent programming language, OS can have several kernel threads per process



tID = create_thread ( ) calls *OS_create_thread ( )*
kill_thread (tID) calls *OS_kill_thread (tID)*
wait_thread (tID) may call *OS_block_thread (tID)*
signal_thread (tID) may call *OS_unblock_thread (tID)*

The OS schedules threads made known to it by the program's runtime system.
The *create_thread* call may be able to indicate a priority for the thread.

kernel threads and user threads

1. thread scheduling is via the OS scheduling algorithm
2. Applications can respond to OS events (e.g. interrupts) by switching threads, but only if OS scheduling is ***preemptive*** and ***priority-based***.
   Real-time response is therefore OS-dependent.
3. user threads can make blocking system calls without blocking the whole process – other threads can run
4. Applications can exploit a multiprocessor (threads can run in parallel)
5. Managing shared writeable data becomes complex
6. There are different ***thread packages*** e.g. posix ***pthreads, FreeBSD …***
7. The runtime need not create exactly one kernel thread per user thread.
   Modern applications may create large numbers of threads (1000s)
   The kernel may allow a maximum number of threads per process related to the number of physical processors.

Lecture 1 summary

Issues of distribution and concurrency

Examples of the need for concurrency control from OS implementations
- showed the need for mutual exclusion and condition synchronisation

Thread support in programming languages and OSs

Next:

How to implement concurrency control for concurrent processes with access to shared writeable data in main memory.