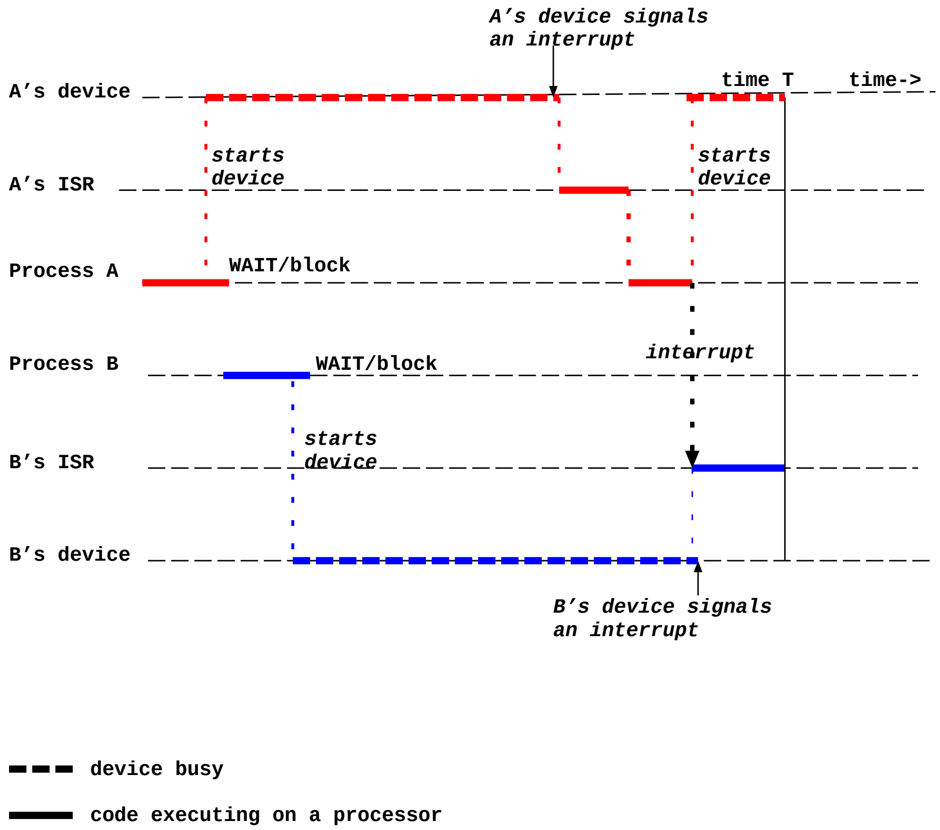
Part 3: Concurrency Control

- motivation
  - hardware-software synchronisation in the OS
  - process-process synchronisation and mutual exclusion in the OS
- multi-threaded OS and processes
- shared-memory (within address-space) IPC
- IPC and system structure
- no-shared-memory (cross-address-space) IPC

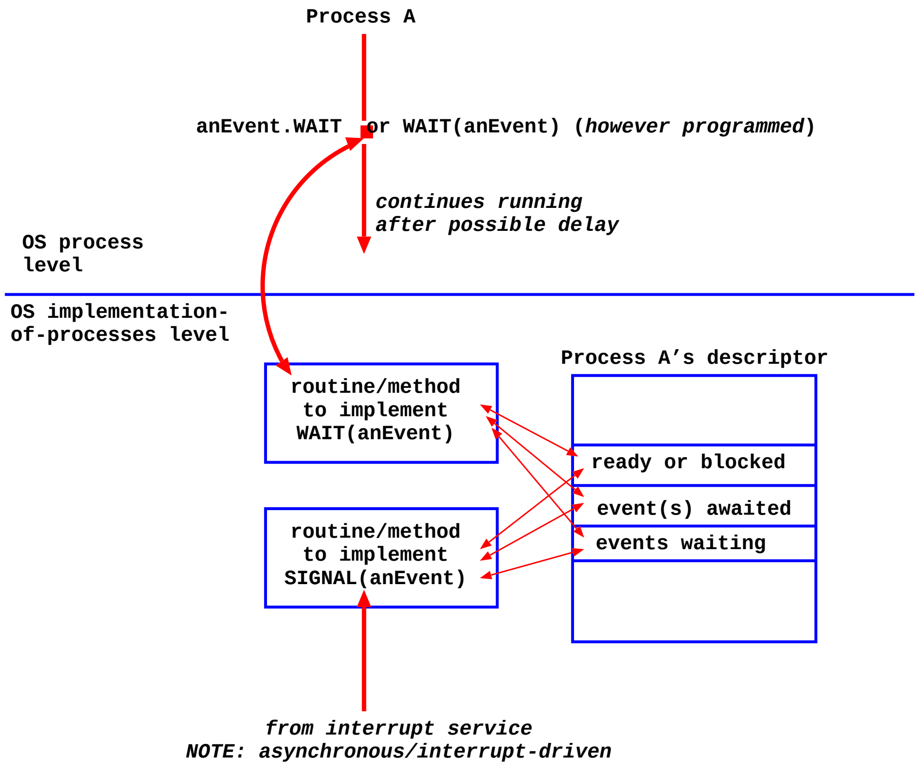
Concurrent device handlers 1



Concurrent device handlers 2

- at time T both process A and process B are ready (runnable)
  - process A was interrupted after starting its device but before WAITing
  - process B was made ready by its ISR
- non-preemptive scheduling - A MUST run again
- preemptive scheduling - OS can choose
- preemptive scheduling - A's device might interrupt while A is still ready but not yet scheduled. The system should store a "wake-up waiting" (device event) for A to find when it blocks waiting for its device's event.

Hardware-software synchronisation 1



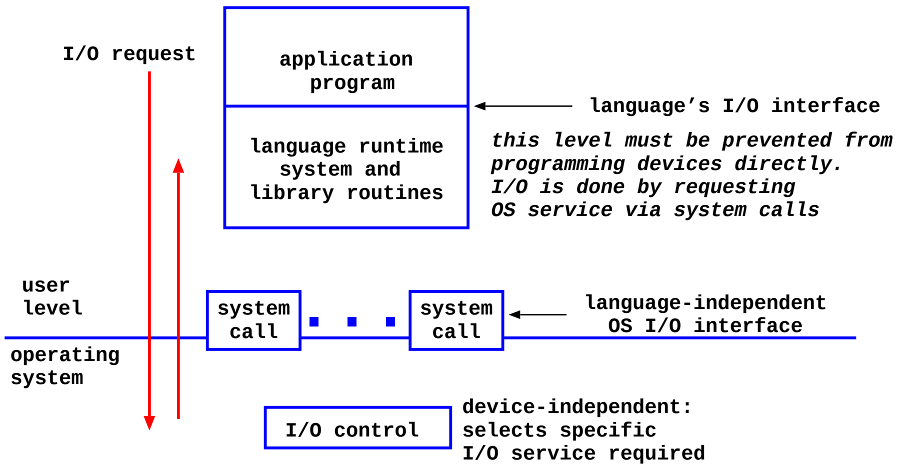
Hardware-software synchronisation 2

- suppose process A's call:
  - enters the WAIT routine
  - reads "events waiting" and finds none
  - ...then is interrupted by a call to SIGNAL
- the SIGNAL routine finds process A ready and its event not in "events awaited" and therefore:
  - sets the event in "events waiting"
  - exits
- the WAIT routine resumes after the interrupt
  - process A sets the event of interest in "events awaited"
  - and its status to blocked (awaiting event)
  - and exits
- we have DEADLOCK
- (see book Concurrent Systems for more O-O view of event management and process/thread management)

Hardware-software synchronisation 3

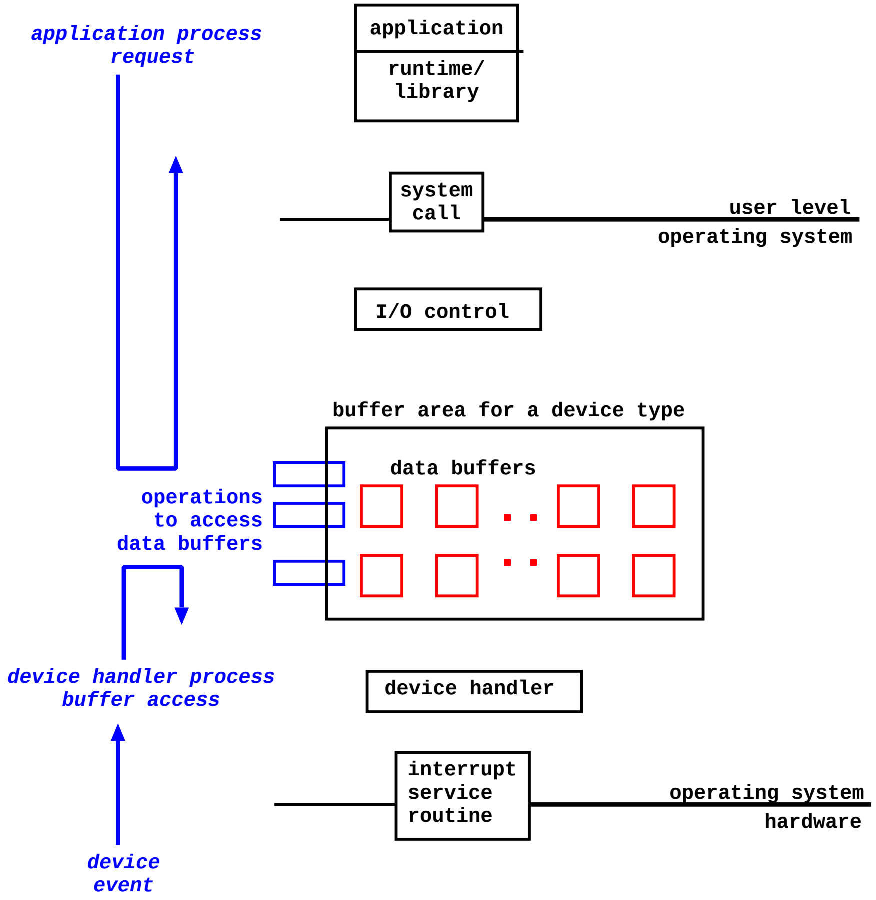
- this is an example of a RACE CONDITION
- the data is shared - read and written by processes/threads executing the WAIT and SIGNAL routines
- we are seeing one possible result of the arbitrary interleaving of the instructions of WAIT and SIGNAL because of interrupt-driven transfer of control
- WAIT and SIGNAL must be made ATOMIC (not-interruptable). It is ESSENTIAL for system correctness that we can do this. - HOW?
  - forbid interrupts? - only works on a uniprocessor
  - composite instructions, semaphores ... see later
- how to achieve correct access to SHARED DATA by CONCURRENT PROCESSES is a general problem:
  - in operating systems - because they are multi-threaded
  - in concurrent applications running at user level

I/O interfaces



- privileged OS programs devices
- unprivileged application makes requests to OS for I/O via system calls

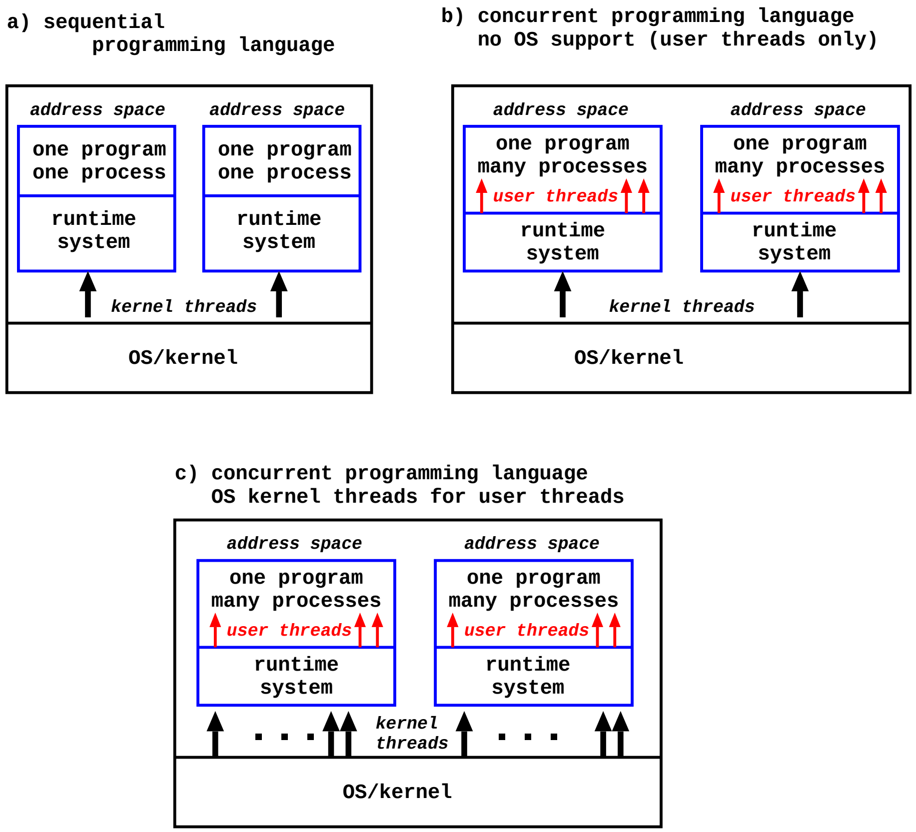
I/O Concurrent OS Execution



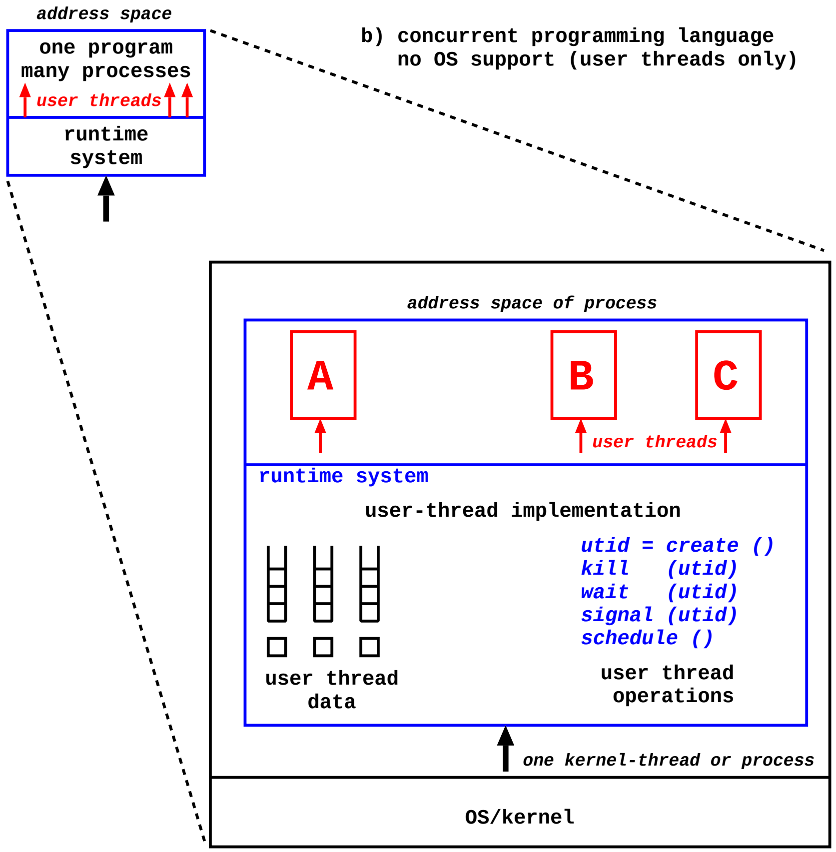
Concurrent OS Execution

- top down, application-driven, OS invocation accesses data buffers
- bottom-up, device-driven, OS execution also accesses data buffers
- need **concurrency control**, support for:
  - **mutual exclusion** from shared data.
    Only one process/thread at once may access writeable data.
    Must be able to wait while another finishes (and signals).
  - **condition synchronisation** over state (e.g. presence/absence) of data.
    Must be able to wait until data is signalled as useable - BUT - must not stop others from accessing it while waiting (or it may never be able to reach the awaited state).

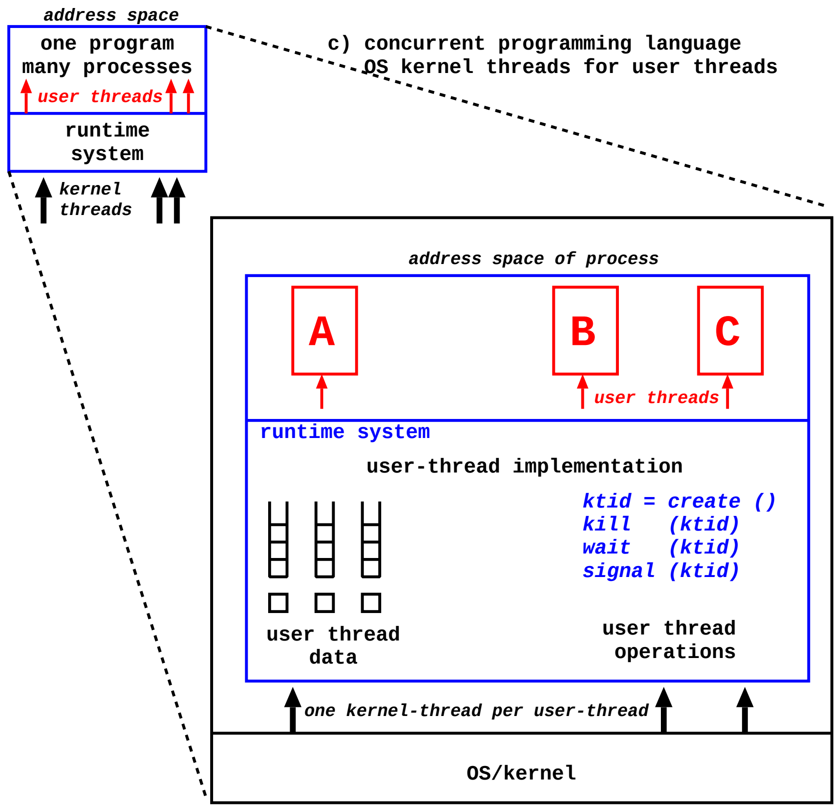
Processes and Threads



Runtime system - user threads



Runtime system - user and kernel threads




```
ktid = create ( )      calls OS:  create-thread   ( )
kill   (ktid)          calls OS:  kill-thread     (ktid)
wait   (ktid)      may call OS:  block-thread    (ktid)
signal (ktid)      may call OS:  unblock-thread  (ktid)
      the OS carries out thread scheduling
```

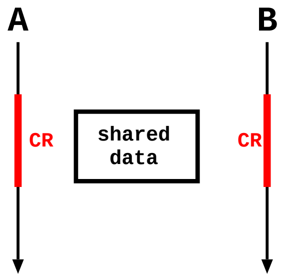
User threads only

- can't respond to OS events by switching user-threads
  can't use for real-time applications - delay is unbounded
- the whole process is blocked if any thread makes a blocking system call
- application can't exploit a multiprocessor
  OS knows only one process/kernel-thread
- BUT handling shared data is simple
  - no user-thread preemption

Kernel threads and user threads

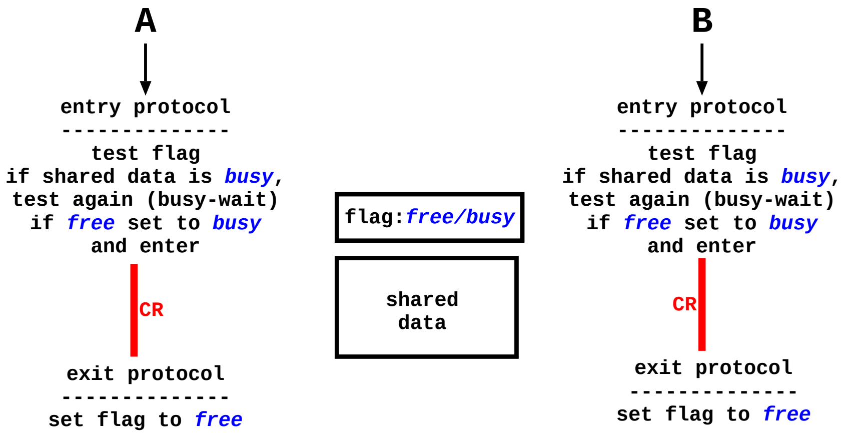
- kernel threads (therefore user threads) are scheduled using the OS's scheduling algorithm
- the application can respond to OS events by switching user-threads - but only if OS scheduling is preemptive and priority-based
  real-time response is therefore OS-dependent.
- user-threads can make blocking OS calls without affecting other user threads which can continue to run
- can exploit a multiprocessor
- there are different thread packages
  needn't have one kernel-thread per user-thread
  - see book Concurrent Systems

Critical regions



processes **A** and **B** contain critical regions (CRs)
- code which reads or writes this shared data

note:
1. CRs needed only if some process writes
2. every CR is associated with some specific shared data



Indivisible test-and-set

- in the entry protocol test and set must be atomic/indivisible - HOW?
  - forbid interrupts? - NO - uniprocessor only and inappropriate for general use.
  - machine instruction - YES
    read-modify-write (test-and-set) (CISC)
    read-and-clear (RISC)

e.g. read-and-clear
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 not zero shared data was free and you claimed it.

- can also be used for condition synchronisation

Semaphores: Dijkstra THE 1968

- entry protocols above involve busy-waiting
- better to block waiting processes

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

wait(aSem) or aSem.semWait (Dijkstra's P(aSem)):
  if aSem>0 then aSem = aSem-1
    else suspend the process waiting on aSem

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

Implementation: an integer and a queue

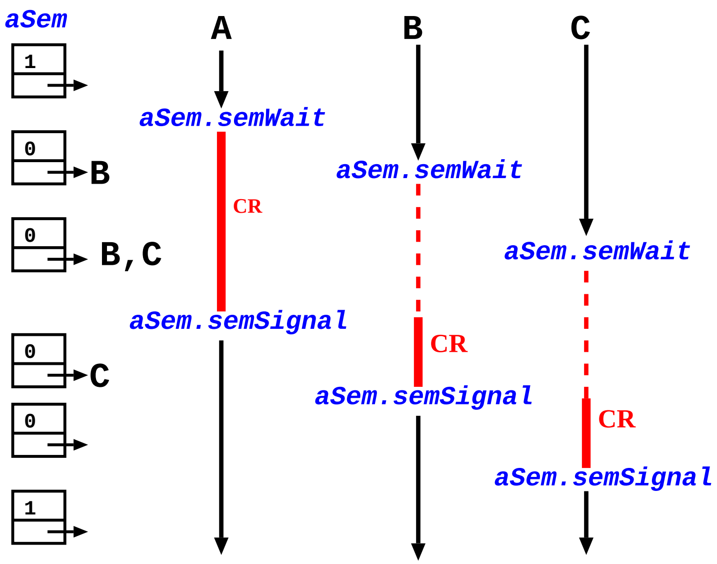
Uses of semaphores

- mutual exclusion - guard a data structure
  values 1 (free) and 0 (busy)
- condition synchronisation
  values 0 (must wait) 1,2,3...N (can proceed)

  e.g. semaphore resource3 is initialised to 3
  process A: *resource3.semWait()* value = 2
  process B: *resource3.semWait()* value = 1
  process C: *resource3.semWait()* value = 0
  process D: *resource3.semWait()* value = 0
  and process D is blocked on resource3

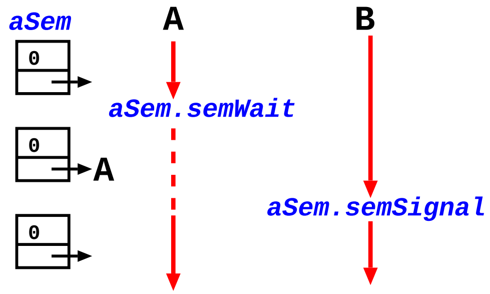
Mutual exclusion

concurrent processes:
serialisation of critical regions

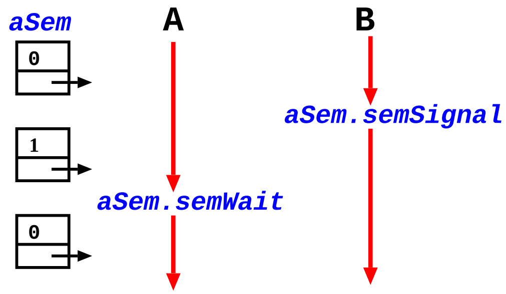


Two-process synchronisation

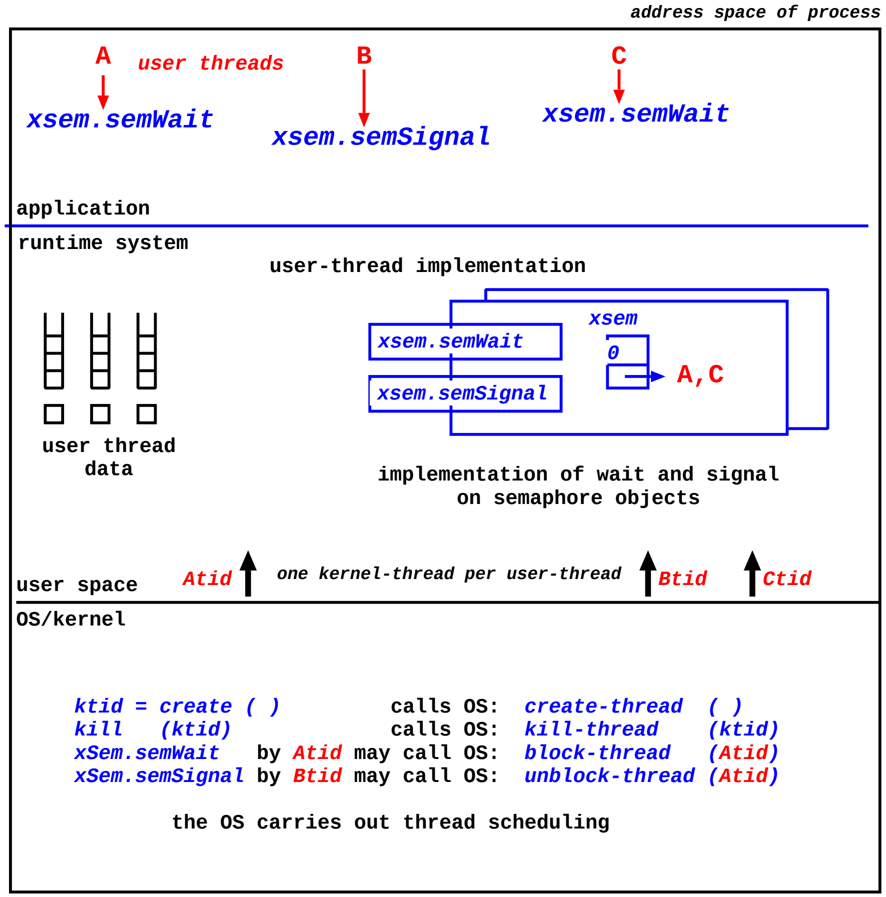
`wait before signal`



`signal before wait`



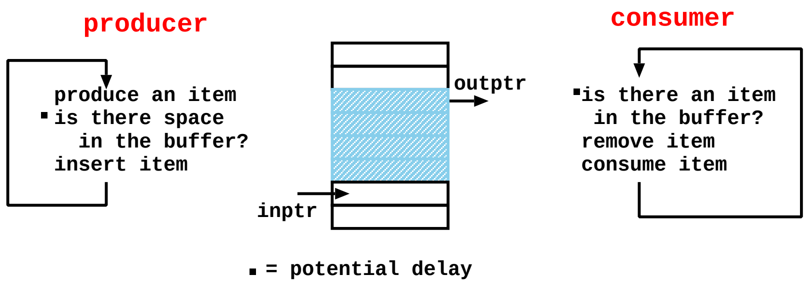
Implementation of semaphores 1



Implementation of semaphores 2

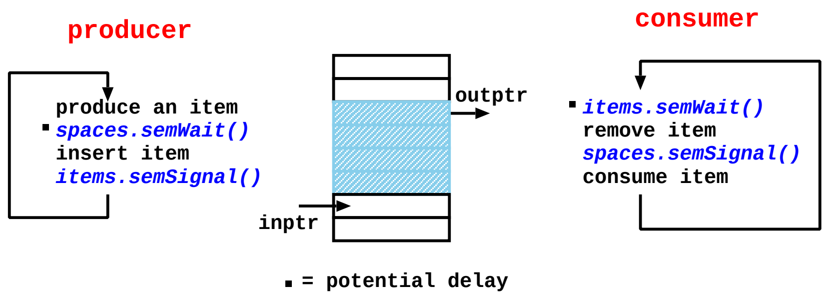
- for user threads only (OS sees only one process) the runtime system does all semaphore and user thread management
- when user threads are mapped to kernel threads semWait and semSignal must themselves be **atomic operations** (multiprocessor or preemptive scheduling) associate a flag with each semaphore object and use a composite instruction such as *read-and-clear*
- crucial and difficult area of system design
- this also applies to kernel threads executing the OS and using OS semaphores for mutual exclusion and condition synchronisation
- the need for concurrency control first arose within OS - now we have concurrent programming languages and OSs support multi-threaded application processes

shared N-slot buffer 1



first assume one producer and one consumer

semaphores: items, initially 0 (items in buffer)
spaces initially N (spaces in buffer)



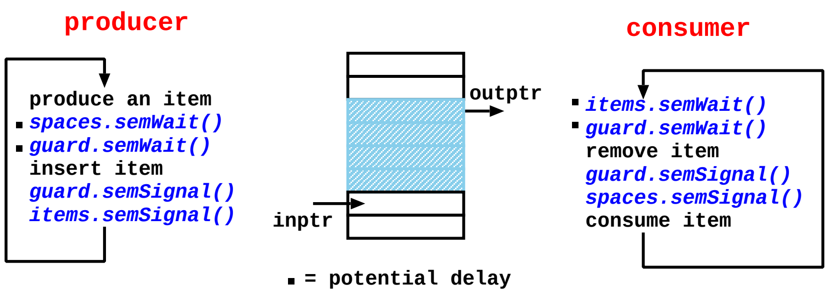
this is programming condition synchronisation

shared N-slot buffer 2

now assume many producers and many consumers

for simplicity, use one semaphore to ensure mutually exclusive access to the buffer by all processes (one of the producers and one of the consumers could access together - we do not show this)

use another semaphore:
guard, initially 1 (buffer free)

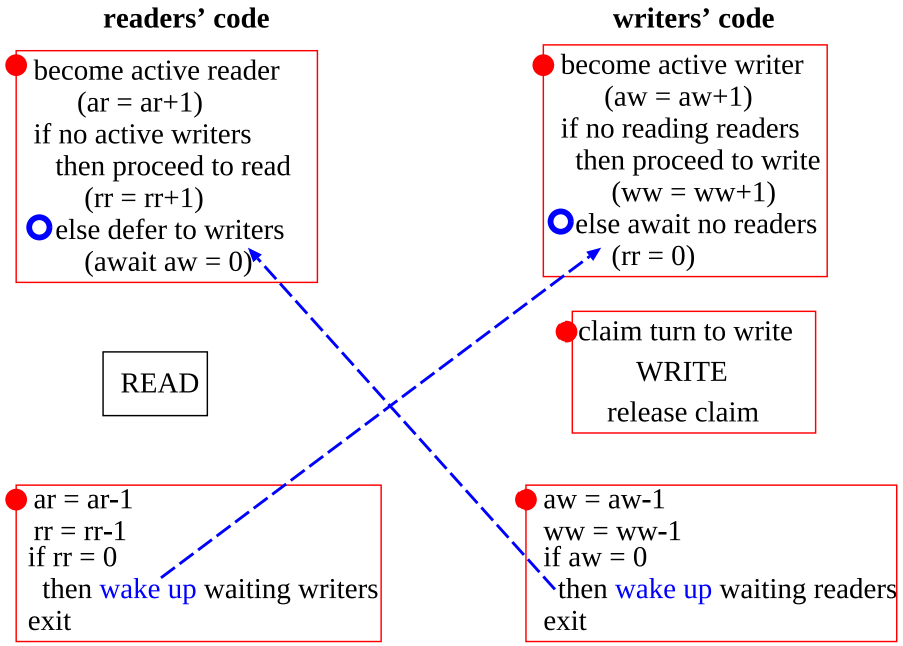


this is programming: mutual exclusion
condition synchronisation

Multiple readers, single writer concurrency control

Many readers may read simultaneously,
a writer must have exclusive access
Assume writers have priority

ar is the count of active readers
rr is the count of reading readers
aw is the count of active writers
ww is the count of writing writers
(but they must take turns to write one-at-a-time



mutual exclusion (to access shared counts)

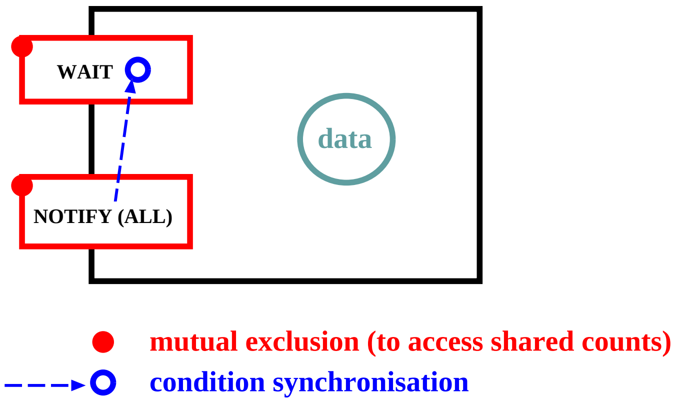
condition synchronisation

Semaphores - discussion

- the mechanism underlying concurrency control in operating systems and concurrent programs
- difficult for programmers to use correctly
  - can forget to wait (corrupt data)
  - can forget to signal (deadlock system)
  - programs are complex - e.g. readers and writers
- unconditional commitment to block
  (but can fork new threads for concurrent activity)
- unbounded delay
- concurrent programming languages provide higher-level constructs but implement them using semaphores:
  conditional critical regions, monitors, synchronised methods and condition variables, active objects, Ada rendezvous and guarded commands

Java concurrency control constructs

SYNCHRONISED methods of an object execute under MUTUAL EXCLUSION with respect to all synchronised methods of the object

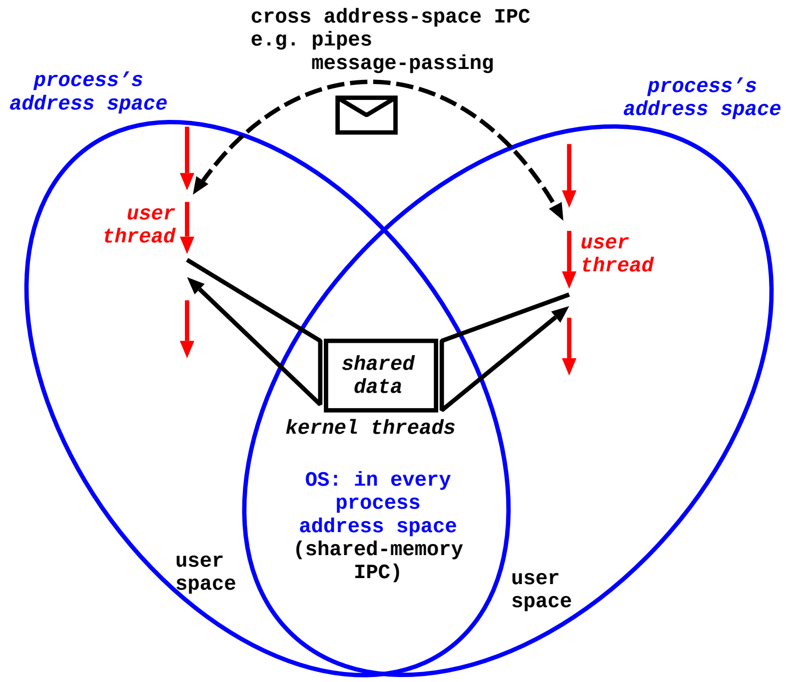


implementation: a semaphore per object

CONDITION SYNCHRONISATION
(not very good)

WAIT (block, release exclusion on object)
NOTIFY (implementation frees an arbitrary thread)
NOTIFY ALL (implementation frees all threads first to be scheduled gets in they MUST retest the WAIT condition)

IPC and system structure 1

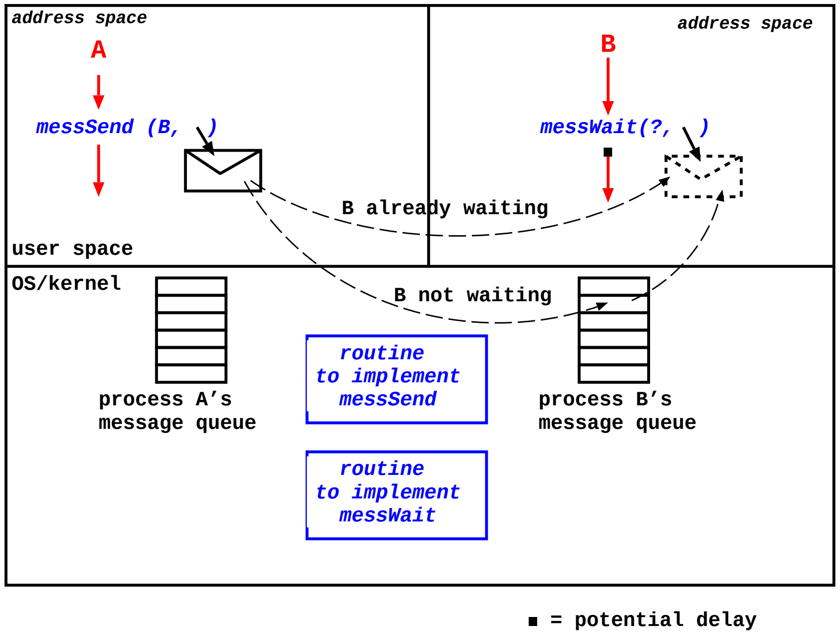


IPC and system structure 2

- user-space processes need to interact
  e.g. clients and servers
- OS must support cross-address space IPC
- shared memory IPC is used:
  - at user level within a multi-threaded program (server or application)
  - within the OS which is inherently multi-threaded:
    user threads call for service and execute the OS as kernel threads

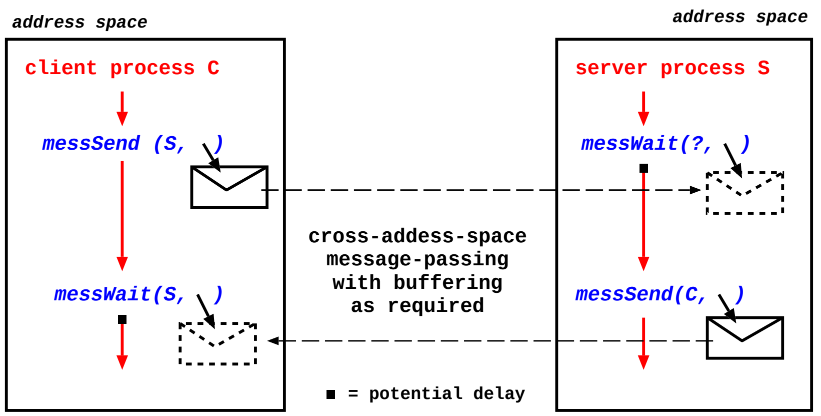
asynchronous message-passing 1

A message has a system-specified header containing the sender and receiver IDs.
The message body is most often unstructured bytes (to the IPC service). The application must interpret.
But typed messages (cf. arguments to procedure calls) are becoming more common in distributed systems.



asynchronous message passing 2

- delay on messWait if no matching messages in queue
- no delay on messSend: message is buffered by system if receiver not waiting
- need to be able to wait for message from "anyone"
- note that message passing implementation needs shared memory concurrency control
- client-server interaction: example as below

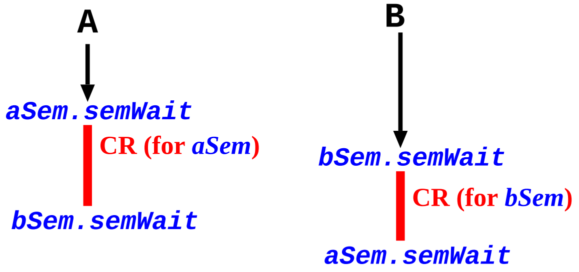


Crashes and composite operations

- For many system failures main memory is lost and persistent store is not.
- Some operations are multi-stage and require changes to data structures in both main memory and persistent memory
  e.g. delete file ⇒ transfer file's disk blocks to free-block list and remove file's entry in superior directory.
  e.g. most database operations.
- if a crash occurs in the middle of a composite operation the persistent store is left in an *inconsistent state*.
- The concept of *atomic operation* or *transaction* is defined: if a transaction completes successfully its results persist; if it does not then the system state must be restored as though it had never started.

Concurrency and composite operations

- We have seen how concurrency control can be implemented for a single operation in main memory (e.g. using a semaphore).
- Now consider composite operations:
  - in main memory
  - involving both main memory and persistent memory
- Composite operations are subject to *deadlock*
  e.g. in main memory:



- Any systems that allocate resources dynamically are subject to deadlock.

The ACID properties of transactions

- Atomicity: the "all or nothing" property defined above.
- Consistency: if the system is in a consistent state a transaction moves it to another consisten state.
- Isolation: The effects of an incomplete transaction are not visible to other (concurrent) transaction.
- Durability: Once a transaction completes (commits) its effects will persist, even if there are subsequent system failures.

If a system supports transactions it must ensure the ACID properties hold under *concurrency* and *crashes*.

Summary of Part 3

You should now understand:

- why OSs must support concurrency control
  - for internal OS execution
  - to support multi-threaded processes
- mutual exclusion and condition synchronisation
- semaphores: uses and implementation
- at least one concurrent program involving several semaphores (if producers and consumers is easy, try readers and writers)
- interprocess communication (IPC) within and across address spaces

and at least have heard about:

- transactions
- deadlocks