The University of Melbourne
School of Computing and Information Systems

COMP30023
Computer Systems

Semester 1, 2017

Synchronization

Multiprocessors and parallelism

Multiprocessors, computers with more than one CPU, have been around since the 1960s, but they have become common only recently. Most recent PCs have two CPUs -called *cores*- on a single chip, and the number of cores per chip is widely expected to double about every 18 months.

A multiprocessor with N CPUs can execute N threads at the same time.

It is trivial for an OS to exploit the availability of this parallelism by running *multiple programs* at the same time. This requires no changes in the programs themselves.

However, if a *single program* wants to make use of more than one CPU, either to model some parallel activity in the real world, or just for more performance, that program must be divided into more than one thread of execution.

Multiprocessors and parallelism

In conventional imperative languages such as C, C++ or Java, the division of the program into threads must be done by the programmer. In some research implementations of declarative programming languages such as Haskell, this task can be done by the compiler.

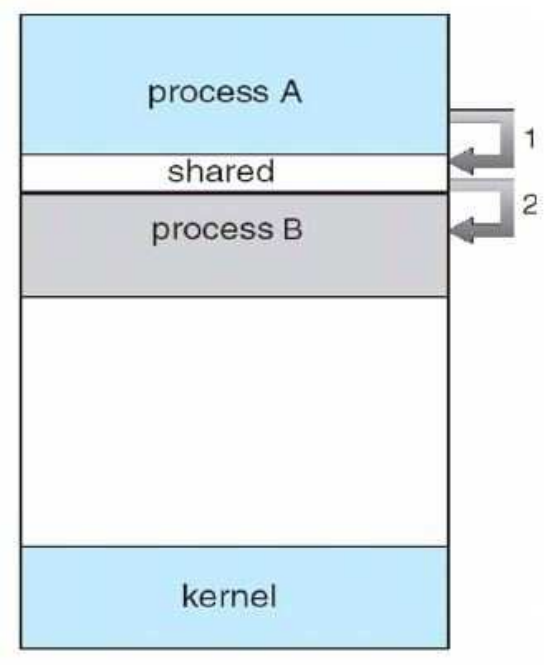
Structuring parallel programs

There are two main ways to structure parallel programs:

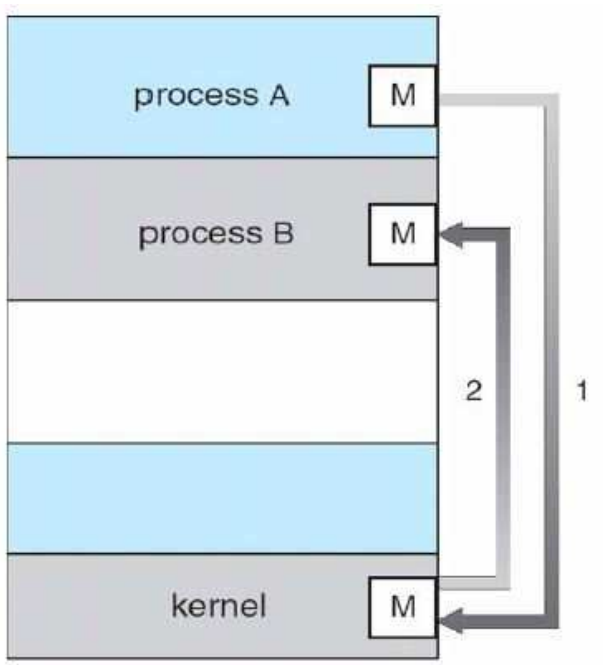
- The threads of the program communicate via **shared data structures**, usually in main memory, but occasionally somewhere else, e.g. in the filesystem.
  This is the usual approach on multiprocessor computers.
- The threads **send messages** directly to each other.
  This is the only feasible approach in distributed systems, though programs on multiprocessor computers can also use it.

Both program design approaches may also be useful on uniprocessors, e.g. to separate client's code from server's code.

Interprocess cooperation



Shared memory



Message passing

Interprocess cooperation via shared memory

Virtual memory systems can arrange to map the same physical memory pages into the virtual address spaces of two or more processes.

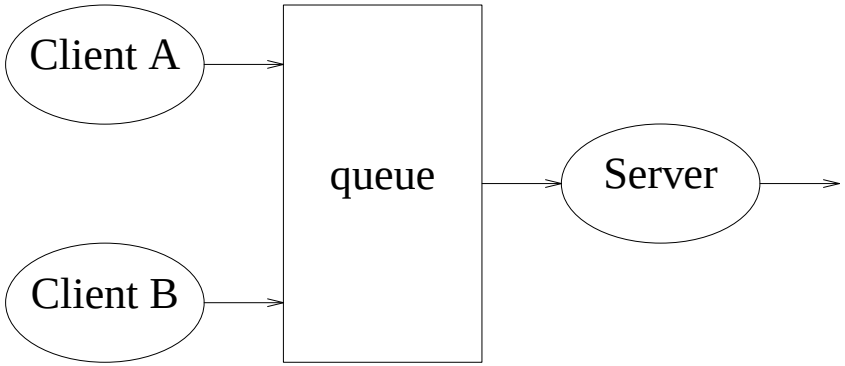
A set of pages mapped into the virtual memories of several processes is usually called a *shared memory segment*.

Modern OSs have system calls both to create new shared memory segments and to attach such a segment to the current process.

A process that has attached a shared memory segment to itself can refer to that segment with the usual instructions for accessing memory.

This makes interprocess communication quite fast, which is why this is the preferred approach when it is available.

Example of shared data



The queue is an array of slots, one slot per item to be processed by the server, and a variable giving the number of the next free slot.

The queue is *shared data*, so both the array and the variable must be in shared memory.

The processes that access the queue must cooperate to ensure its integrity. This is because allowing each process to access shared data *without* cooperation can easily corrupt that data.

Race condition

One possible sequence of events is a *race* between the clients:

- client A on CPU 1 reads freeslot = 7 into a register
- client B on CPU 2 reads freeslot = 7 into a register
- client B on CPU 2 puts its item in slot 7
- client A on CPU 1 puts its item in slot 7
- client A on CPU 1 adds 1 to its register
- client B on CPU 2 adds 1 to its register
- client A on CPU 1 writes freeslot = 8 from its register
- client B on CPU 2 writes freeslot = 8 from its register

Both clients will try put their items into slot 7. If, as shown here, client B puts its item into slot 7 first, that item will overwritten by client A, so B's item is *lost*: it will never be processed by the server process.

Critical sections

The solution is to ensure that only one process is updating freeslot at any one time.

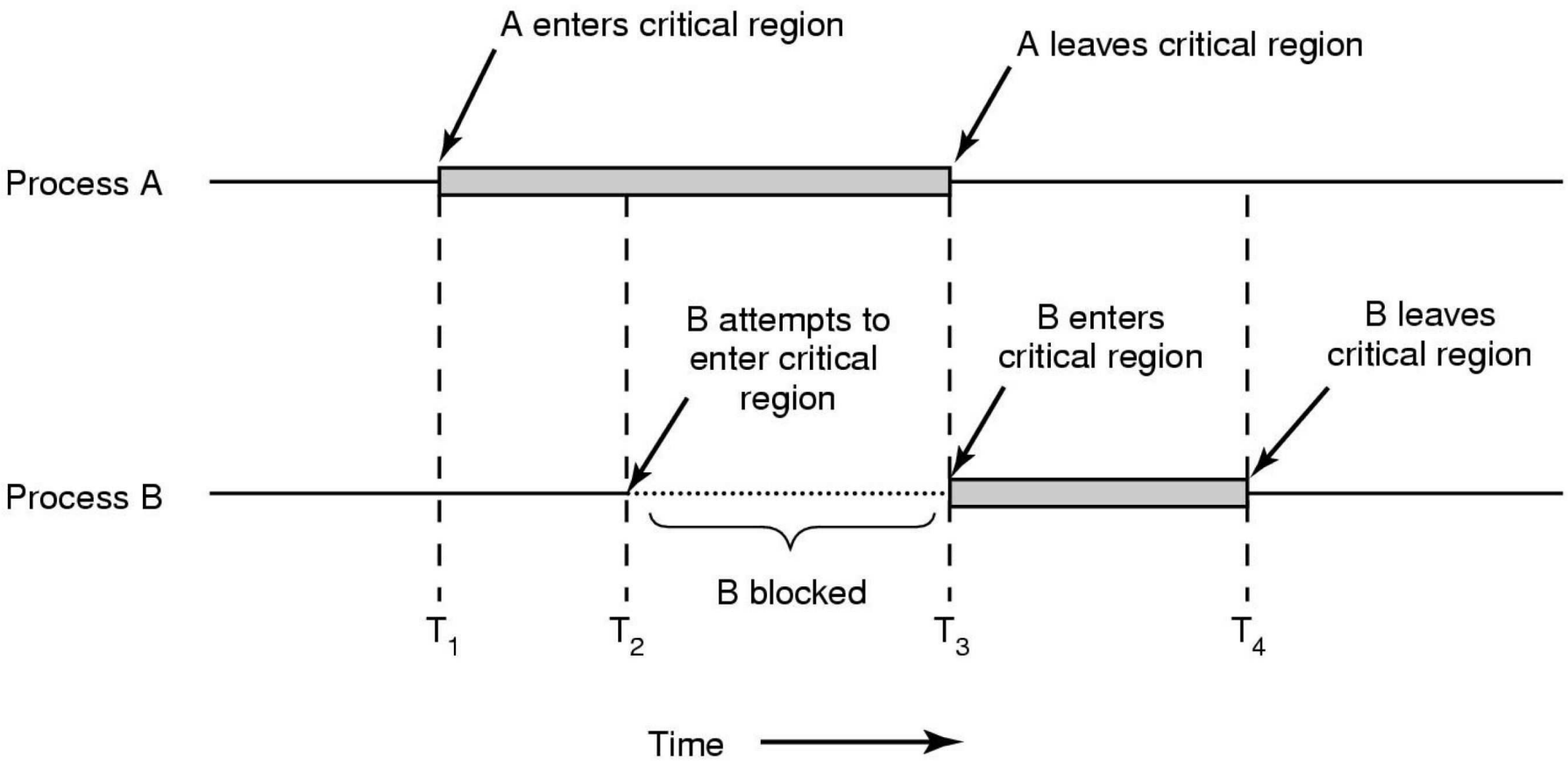
The notion of *critical sections* generalizes this solution: **only one process may be executing code in a given critical section at any one time**. The critical sections *mutually exclude* each other.

A guarantee of mutual exclusion must not require *any* assumptions about the number of processes or their relative speeds.

In this case, the critical section would be

load freeslot into register
copy item to the slot given by that register
increment register
store register to freeslot

Mutual exclusion



Guaranteeing mutual exclusion

The initial values of want1 and want2 are FALSE.

Process P1:	Process P2:
want1 = TRUE;	want2 = TRUE;
while (want2)	while (want1)
;	;
critical section	*critical section*
want1 = FALSE;	want2 = FALSE;

This algorithm guarantees mutual exclusion if read and write operations are atomic (on some modern machines they are *not* atomic).

However, the algorithm can *deadlock* if P1 and P2 proceed in lockstep.

Dekker's algorithm

Process P1:

```
want1 = TRUE;
turn = 1;
while (want2 && turn == 1)
    ;
critical section
want1 = FALSE;
```

Process P2:

```
want2 = TRUE;
turn = 2;
while (want1 && turn == 2)
    ;
critical section
want2 = FALSE;
```

This resolves the deadlock problem: the process that sets turn last has to wait until the other has finished.

Unfortunately, this algorithm is hard to generalize to three or more processes.

Synchronization

Hardware support may be provided for critical section code.

Uniprocessors could disable interrupts. In this situation, currently running code would execute without preemption. However, this is generally too inefficient on multiprocessor systems. (Is this a good idea?)

Modern machines provide special atomic (or non-interruptable) hardware instructions, which always runs to completion or not at all.

On most machines, memory references and assignments (i.e. loads and stores) of words are atomic.

Mutual exclusion via interrupts

Raising the CPU's interrupt priority level to a value that blocks all interrupts, and then restoring the old IPL, will ensure mutual exclusion in a uniprocessor without wasting any time in busy-waiting.

intr_off(); *critical section*; intr_on();

This can be done only in kernel mode. However, the OS cannot execute the user's code. So it supplies services that the user programs can use to get mutual exclusion.

This service cannot be "turn off interrupts".

Test-and-set

The **test-and-set** assembly instruction can test and set a value in a single atomic operation.

The hardware guarantees this operation is atomic. That is, it prevents all accesses to the word/value from other CPUs between the read and the write.

```
boolean test-and-set(boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Test-and-set

The key point of this instruction is that if a thread executes this instruction, the actions – getting the current value of the memory location and setting the new value to TRUE – happen atomically.

We can then use the operation:

```
while (test-and-set(&lock)) ;
critical section
lock = FALSE;
remainder section
```

Not all instruction sets have a test-and-set instruction. Some supply a "swap" instruction instead, which exchanges the contents of a register and a memory word atomically. Still others provide "compare and swap", which does the swap only if a comparison of the two values has a specific result.

Use of test-and-set (additional material)

The 680x0 implementation of routines to enter and leave critical sections:

```
enter_cs:   move.l  #lock, a0
loop:       tas     a0@
            bne     loop
            rts

leave_cs:   move.b  #0, lock
            rts

lock:       dc.b    0
```

This use of test-and-set works for arbitrary numbers of processes.

Spinlocks

This is a generic name for busy-waiting synchronization methods. Dekker's algorithm is an example of using spinlocks.

Spinlocks can waste CPU time, so one must ensure that the time in spinlock is *bounded* and *short*.

Spinlocks can cause deadlock due to *priority inversion*: a high-priority process may busy-wait forever for a lock held by a low-priority process. The high-priority process has the CPU, the low-priority process has the lock, and neither will let go until the other does.

POSIX Threads

When using the Pthread library, we can enforce mutual exclusion using `mutex_`

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex):
pthread_mutex_unlock(&mutex);
```

Think carefully about the positioning of the `mutex` in your thread code.

Semaphores

A semaphore is an integer variable which can be accessed by the primitives wait and signal.

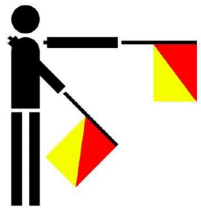
signal(s) Increment s in an atomic (indivisible) operation.

wait(s) Decrement s when $s > 0$ in an atomic (indivisible) operation. If $s = 0$ at the start, the process invoking wait must be delayed until some other process executes a signal on s.

The OS can provide system calls to create and destroy semaphores, and to perform wait and signal operations on them.

Using semaphores

This technique can ensure mutual exclusion for any number of processes. Spinlocks using test-and-set can do this too, however, semaphores are easier to use.



The initial value of s is one.

Process P1:	Process P2:
wait(s);	wait(s);
critical section	*critical section*
signal(s);	signal(s);

Using semaphores

Semaphores can also be used for purposes other than mutual exclusion. The semaphore can encapsulate a non-binary variable.

```
typedef struct
{
    unsigned c;
    Queue    q;
} Sem;
```

c is the value of the semaphore. q is a queue of processes suspended on the semaphore when $c = 0$.

The wait and signal system calls manipulate this structure.

Using semaphores

```
wait(Sem *s)
{
    intr_off();
    if (s->c > 0) {
        s->c -= 1;
    } else {
        suspend(s->q);
    }
    intr_on();
}
```

Suspend enters the id of the calling process in the given queue; it then blocks the process. When wait exits, the scheduler will select the next process to run.

Using semaphores

```
signal(Sem *s)
{
    intr_off();
    if (nonempty(s->q)) {
        first = remove_first_from_queue(s->q);
        activate(first);
    } else {
        s->c += 1;
    }
    intr_on();
}
```

Activate removes a process id from the given queue; it then changes this process to ready. When signal exits, the scheduler may or may not schedule the newly ready process.

POSIX semaphores

Include `semaphore.h`

Create a semaphore variable: `sem_t sem`

Initialize the semaphore variable: `sem_init(sem, 0, init);`

- 2nd argument: amount of sharing (0 only threads in current process)
- 3rd argument: initial value of semaphore counter (0 if using the semaphore for signalling)

Wait function: `sem_wait(sem);`

Signal function: `sem_post(sem);`

Producer-consumer problem (also known as "bounded buffer")

Two sets of processes share a single buffer of fixed size. The *producers* put information into the buffer; the *consumers* remove information from the buffer.

Producers can only deposit items in the pool when space is available for them.

Consumers can only extract items from the pool when they are available.

One class of examples is streaming audio/video programs. For a video player, the items would be frames. The download thread puts frames into the buffer at a variable rate (as the load on the net allows), while the player thread removes frames from the buffer at an even rate.

The solution

mutex = 1;
space = N;
item = 0;

Producer	Consumer
wait(space);	wait(item);
wait(mutex);	wait(mutex);
deposit item	*extract item*
signal(mutex);	signal(mutex);
signal(item);	signal(space);

Each type of process produces what the other needs: items and space respectively.

A summary

Consumer must wait for producer to fill buffers, if none full (scheduling constraint).

Producer must wait for consumer to empty buffers, if all full (scheduling constraint).

Only one thread can manipulate buffer queue at a time (mutual exclusion).

General rule of thumb:

- semaphore fullBuffers – consumers constraint
- semaphore emptyBuffers – producers constraint
- semaphore mutex – mutual exclusion

Deadlock-prone solution

Producer	Consumer
wait(mutex);	wait(mutex);
wait(space);	wait(item);
deposit item	*extract item*
signal(item);	signal(space);
signal(mutex);	signal(mutex);

A producer who wants to deposit an item when the buffer is full will (by holding the mutex) prevent consumers from creating the space it needs to proceed.

Similarly, a consumer who wants to extract an item when the buffer is empty will prevent producers from depositing the item it needs to proceed.

Readers and writers

Two sets of processes share a single piece of data. Multiple *readers* can access this data concurrently; the *writers* need exclusive access.

We need two kinds of synchronization: between readers and writers, and among readers.

The latter is based on the principle of "Will the last person to leave please turn off the lights?".

Readers and writers solution

readers = 0; r = 1; data = 1;

Reader	Writer
wait(r);	
readers += 1;	
if (readers == 1) wait (data);	
signal(r);	wait(data);
read data	*write data*
wait(r);	signal(data);
readers -= 1;	
if (readers == 0) signal(data);	
signal(r);	

Lock granularity

When you have several related data structures (which may or may not be of the same type) that you want to protect with mutual exclusion, you can make one semaphore (or spinlock) to protect them all, or you can make a separate one for each data structure.

The first approach requires less memory, and has less overhead if you access two data structures of the same type. The second permits more parallelism, as several threads can operate on several data structures at the same time.

The amount of data protected by a lock is often referred to as the *grain size*.

Kernels for multiprocessors usually have *large grain* locks on infrequently used data structures and *small grain* locks on frequently used data structures.

Lock granularity

- System-wide lock: Create a single global lock for all critical sections.
- Object-level lock: Create a lock for each object.
- Field-level lock: Create a lock for each field (or array element) within an object.
- What are the effects of using locks at a finer granularity:
  - more opportunities for concurrent operations.
  - more lock and unlock operations.
  - more difficult to implement correctly

Priority inversion

Priority inversion can occur even without busy waiting, like this:

- A low priority thread has grabbed a mutex.
- A high priority thread is blocked, waiting for that mutex.
- A medium priority thread has the CPU. Since it prevents the running of the low priority task, it (through the mutex) also blocks the high priority thread.

The usual fix is *priority inheritance*. This says that the scheduling priority of a thread should be the maximum of

- its own inherent priority, and
- the maximum priority of the threads waiting for any locks it holds.

Java (an alternative example)

A Java program can create a new thread by constructing an object which is an instance of the class Thread.

Every instance of class Object and its subclasses contains a lock (effectively a mutex semaphore).

You can declare that a given region of code is a critical section protected by the mutex semaphore of a given object (e.g. "obj") by surrounding that code region with `synchronized(obj) { ... }`. Note that the fields of `obj` can still be accessed by other code not within a similar synchronized region, but such accesses are not synchronized and are therefore likely to be buggy.

Every instance of class Object and its subclasses also contains a set of waiting threads, and can be used as a primitive form of monitor (without separate condition variables).

Timeouts

It is bad form for a program that interacts with a user to "freeze".

However, this means that interactive programs that need a lock on some resource have a problem: they can't control how long they must wait to get that lock.

Depending on what other thread has the lock and what it is doing, the wait can go on for a long time, or even forever.

The solution is to allow users to specify a timeout on each wait() operation or its equivalent. If the wait() cannot get the lock in that time, it returns anyway with an error indication.

Java's wait monitor operation has a timeout parameter, but has no error indication.

Resources

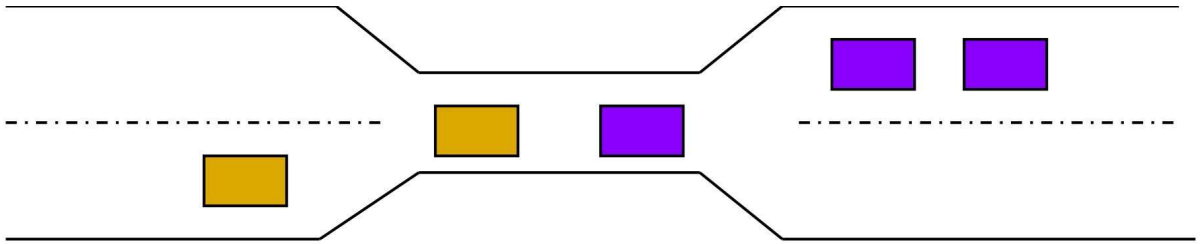
A resource is a commodity required by a process to execute.

Resources can be of several types:

- Serially Reusable Resources: eg CPU cycles, memory space, I/O devices, files (acquire - use - release)
- Consumable Resources: produced / required by a process. e.g. messages, buffers of information, interrupts (create - acquire - use)

How to avoid a deadlock situation ...

Deadlock "bridge" example



Assume traffic in one direction.

Each section of the bridge is viewed as a resource. If a deadlock occurs, it can be resolved only if one car backs up (preempt resources and rollback). Several cars may have to be backed up if a deadlock occurs. Starvation is possible.

Deadlocks

Process A	Process B
request R1	request R2
...	...
use R1	*use R2*
...	...
request R2	request R1
...	...
use R1 & R2	*use R2 & R1*
...	...
release R2	release R1
release R1	release R2

Modeling of deadlocks

A set of processes (or threads) is deadlocked if every process in the set is waiting for an event that can be caused only by another process in the set.

This event is usually releasing a resource, which may be a dedicated I/O device, a block of memory, a lock, etc.

Four conditions must be satisfied before a deadlock can occur.

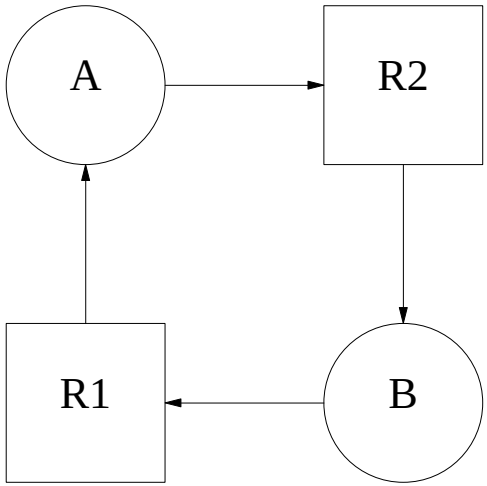
1. Processes must have exclusive access to resources.
2. A resource cannot be forcibly taken away from a process until that process explicitly releases that resource.
3. A process currently holding resources can request new resources.
4. There must be a circular chain of processes such that each process in the chain is waiting for a resource that is held by the next member of the chain.

Resource graphs

These graphs show resources as rectangles and processes as circles.

Resource to process arcs: that resource has been allocated to that process.

Process to resource arcs: that process has requested that resource.



Deadlocks show up as cycles.

Strategies for dealing with deadlocks

- Ignore the problem (the ostrich strategy).
- Detect that a deadlock has occurred, and recover.
- Prevent deadlocks statically by permanently negating one of the four necessary conditions.
- Avoid deadlocks dynamically by not allocating a resource to a process if that allocation can be part of a deadlock chain.

Ostrich strategy

Works extremely well in many environments.

On some systems, crashes for other reasons are (or were) much more frequent than deadlocks.

For example, early versions of Windows did not do sanity or security checks of system call arguments, and so bugs in application programs often caused a crash, signalled to the user by the dreaded Blue Screen Of Death (BSOD).

Some installations of Windows still crash reasonably regularly due to bugs in device drivers, but this is becoming rarer and rarer.

Detection and recovery

The operating system can keep track of the resource graph, and check for the existence of cycles (say once every minute).

If cycles exist, the operating system should pick one process in each cycle and kill it, releasing its resources. It should then restart the killed processes, if this is possible.

This is usually acceptable in transaction processing environments.

For example, if a travel agent's reservation program can't get a seat on a particular flight because someone else's program has a lock on that flight, waiting a second or two and trying again is perfectly acceptable.