FIT9134
Computer architecture and operating systems

Week 10

Operating Systems V:
Process Management – Deadlocks and IPC

MONASH University
Information Technology

Deadlocks

- where processes are given exclusive access to devices, files, records etc, there is a potential for a condition known as "deadlock" to occur.

- a ***Deadlock*** is a situation where 2 or more processes ***wait indefinitely*** because the resources they need to complete are being held (& never released) by other processes.

Deadlock Conditions

4 conditions must exist simultaneously for deadlocks to occur:

i) Mutual Exclusion – a process is allowed to hold on to an un-shareable resource exclusively

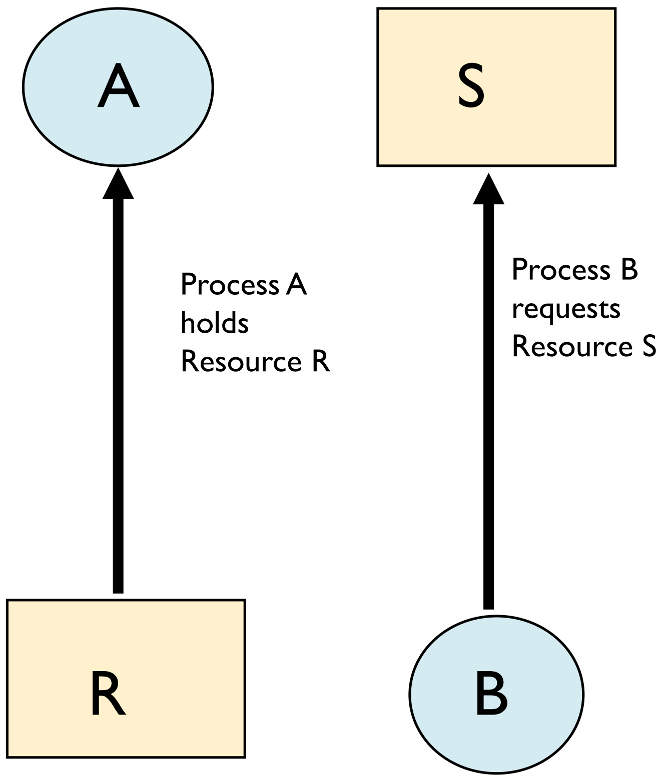
ii) Hold & Wait – a process is allowed to request new resources over time without releasing those it already holds

iii) No Pre-emption – a resource can only be released voluntarily by the process which holds it

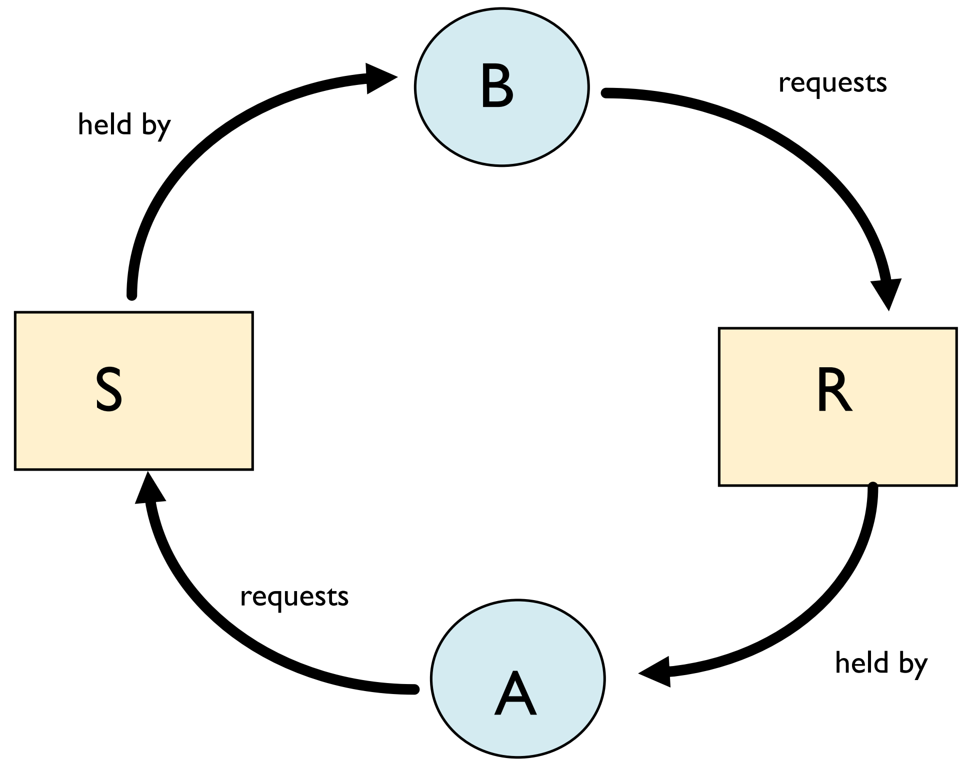
iv) Circular Wait – 2 or more processes are waiting for resources held by each other

Deadlock Conditions

- A ***resource allocation graph*** is often used to model the above conditions.



No Deadlock



Deadlock (Cycle)

Deadlock Management

- 4 possible strategies for dealing with deadlocks:

i. Ostrich Algorithm
ii. Detection and Recovery
iii. Deadlock Prevention
iv. Deadlock Avoidance

Deadlock Management Strategies

i) Ostrich Algorithm

- simply ***ignore the problem***, as the overhead of dealing with all possible deadlock situations may be too high. This can sometimes be the simplest and most effective solution!

ii) Detection and Recovery

- resource graph checked (and updated) for ***cycles*** on each request/release of resources. If cycle is about to be caused by a resource allocation, then one process is suspended, rolled back or killed.

Deadlock Management Strategies

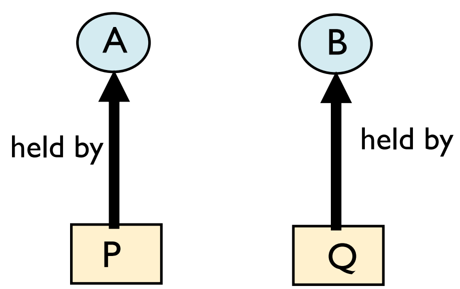
iii) Deadlock Prevention - this requires that one of four conditions as discussed before be eliminated:

- 'mutual exclusion' - cannot be entirely removed, since some resources are un-shareable by nature.
- 'hold & wait' – make all processes request resources in advance => may not be possible, and often causes non-optimal use of resources.
- 'no pre-emption' – forcibly removes all resources held by a process if it requests any resources which cannot be immediately allocated. Restart the process when all resource requests can be satisfied.

Deadlock Management Strategies

- ‘circular wait’ – this condition is preventable if all resources are ordered/numbered, and processes are only allowed to request resources in ascending numerical order => cycles eliminated

Eg.



Deadlock will occur if A requests Q ***AND*** B requests P. But if P has been assigned a number higher than Q, then A will not be allowed to request Q. Similarly, if Q > P, then B is not allowed to request P. So deadlock by circular wait will not occur.

Process Management Strategies

iv) Deadlock Avoidance - relies on ***monitoring*** the use of resources and ***anticipate*** requests for use.

- e.g 'Bankers Algorithm' (refer to example in ***Tanenbaum*** *textbook, Chapter 6)*, based on 'safe states'. State is safe if not deadlocked and there is a way to satisfy all future/pending requests

- Main problem with most deadlock avoidance strategies is resource requirements must be known in advance and stay static for duration of current processes. Considerable overhead involved.

Process Coordination

- A mechanism called "***Semaphore***" may be used to provide Mutual Exclusion and Synchronization between processes

  - e.g. O/S processes placing jobs in print queue and the print process removing them. The 2 processes need to be synchronized.
    - Only one process is allowed to manipulate the queue at any time; the code to do this is sometimes called a "critical code" (code which can only be executed by ***one process at any one time***). We achieve the mutual exclusion and synchronization by placing the "critical code" inside a special "***Wait***" and "***Signal***" code section (using *semaphores*).

Semaphores

- A "*semaphore*" can be thought of as a *non-negative integer* (S) which may only be acted upon by 2 operations:
  - ***signal(S)*** means :

    S = S + 1
  - ***wait(S)*** means :

    while **S is 0** do nothing (ie. "***waits***")
    otherwise ***S = S - 1***

- ***Signal(S)*** and ***Wait(S)*** are ***non-divisible*** operations, & may only be used by one process at a time (ie. if a process wants to perform *wait(S)*, it cannot proceed until another process has performed a *signal(S)* to make the value of *S* +'ve).

Critical Section

- the "critical code to be performed" (called the '*critical section*') of the process may only be executed by one process at a time.

- non-shareable resources e.g. peripherals, files, data tables can be protected from simultaneous access by processes by making those parts of code that access the resource a "***Critical Section***". Only one process can be in a critical section at any one time.

Critical Section example

add2pque is a ***BINARY*** semaphore

start

createSemaphore(add2pque), initial value = 1

....

print manager activated

verify file exists

allocate printer

add to print queue

Critical Section

wait(add2pque)

// critical code (eg. printing) to be performed...

signal(add2pque)

....

end

add2pque becomes 0 (if no other process is in this ***critical section***) and this process goes into the critical section. Otherwise this process **waits**.

add2pque becomes 1, **signaling** that the current process is finished with the ***critical section***, other processes may now enter the ***critical section***.

Semaphore Implementation

- ***wait(S)*** and ***signal(S)*** or similar inter-process communication mechanisms may be implemented as part of the instruction set of the CPU, and are used in inner kernel of the O/S.
  - this allows user programs to implement critical sections in their code
- indivisibility of semaphore operation is crucial; this can be ensured by disabling interrupts on entering semaphore.

Inter-process Communication

- As well as semaphores, Unix also supports a variety of mechanisms that processes can use to communicate with each other.

- These are referred to as **IPC (Inter-process Communication)**:
  - ***Pipes***
  - ***Message queues***
  - ***Shared memory***

Inter-process Communication

- These IPC (pipes, message queues & shared memory) facilities were introduced in System V Unix.
  - now used in most Unix implementations (including Linux)

- These facilities are generally accessed through system calls (calls to routines in the kernel code).

Pipes

- A fundamental IPC facility in Unix.
- A cornerstone of the Unix philosophy of modularity.
- Pipes are used on the command line, eg:

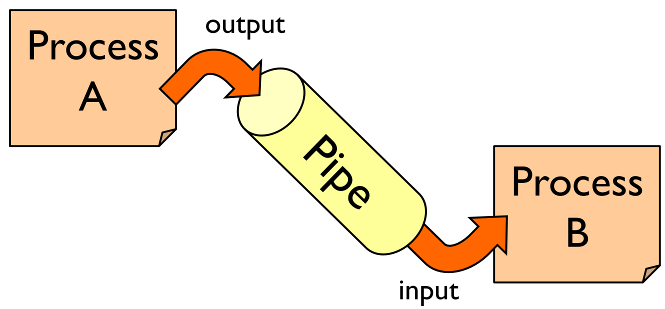
```
$ ls -l | grep '^d' | awk '{print $4}' | sort
```

- Programs can also make use of the pipe facility using appropriate system calls.

These "Pipes" connect the ***output*** of one program to the ***input*** of another program

Pipes

- Pipes are intended for one-way communication between processes.



- Operate on a *first-in first-out* (FIFO) basis. Data flow is one-directional.
- Flow control within the pipe is handled by the kernel on behalf of processes. For example, when a pipe is full, process A will block; when the pipe is empty, process B will block.

Message Queues

- A message is simply a sequence of characters.
- A message queue is a linked list of messages, each of a fixed maximum size.
- New messages are always added to the end of the queue. In this sense, the order of message sending is preserved.

Message Queues

- Once a message queue has been established, processes can place messages on the queue and remove them, through system calls.

- Message queues are in a sense similar to pipes, however message queues are more versatile:
  - messages are distinct, whereas pipes just pass unformatted streams of data.
  - messages can be assigned a *type* which can be used to allow classes messages to be processed in a particular way by a single process or distribute messages to multiple processes.
  - we do not always have to read the first message in the queue first.
  - message queues are persistent

Shared memory

- Often considered the most efficient IPC mechanism.
- Allows multiple processes (with the right permissions) to share the same memory segment.
- The shared memory segment must be created first, then attached to a process (this "attaches" the shared memory segment to the process address space). System calls are provided for control operations (eg. reads & writes) on the shared memory segment.
- Processes can "lock" the shared memory area if they have appropriate permissions.

Some Useful Texts/Resources

- **Lister A.M. & Eager R.D.** *Fundamentals of Operating Systems, Macmillan*
- **H.M. Deitel, P. J. Deitel, D.R. Choffnes** *Operating Systems, Prentice Hall*
- **A.S. Tanenbaum** *Modern Operating Systems, Prentice Hall*
- **A.S. Tanenbaum, A.S Woodhull** *Operating Systems Design and Implementation, Prentice Hall*