COMP3231

COURSE SUMMARY

Contents

1. Introduction to Operating Systems

Learning Outcomes

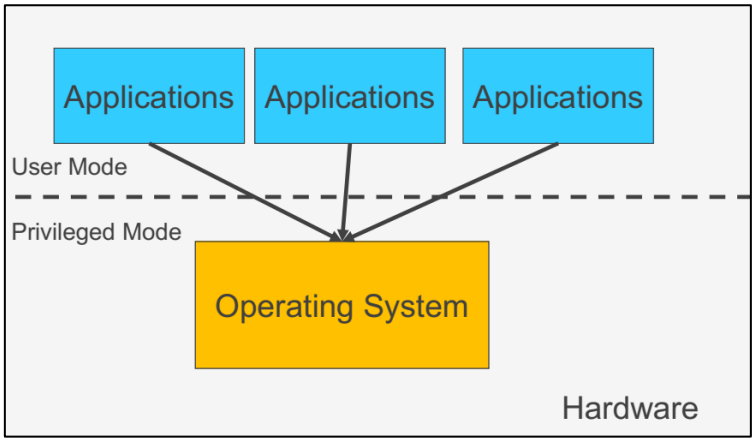
- A high-level understanding of what an operating system is and the role it plays
- A high-level understanding of the structure of operating systems, applications, and the relationship between them
- Some knowledge of the services provided by operating systems
- Exposure to some details of major OS concepts

Roles of an Operating System

- Abstract machine
  - Extends the basic hardware with added functionality
  - Provides high-level abstractions
    - More programmer friendly
    - Hides the details of the hardware, making application code portable
- Resource manager
  - Responsible for allocating resources to users and processes
  - Must ensure:
    - No starvation
    - Progress
    - Allocation is according to some desired policy (first come first served, fair share, weighted fair share, etc.)
  - Overall, that the system is efficiently used

User Mode vs. Privileged Mode

- User mode gives applications access to a restricted subset of the machine in order to protect the OS from applications. An application can only access memory that the OS has made available to it and should not be able to uncontrollably interfere with or bypass the OS directly.
- Privileged mode, or kernel mode, provides unrestricted access to all resources in the machine. This is the mode in which the kernel runs.



- Kernel:
  - Portion of the operating system that is running in privileged mode
  - Contains fundamental functionality – whatever is required to implement other services or provide security

- Assume two simple operations on processes (or threads):
  - **`sleep(P)`** suspends the execution of a running process **`P`**
  - **`wakeup(P)`** resumes the execution of a blocked process **`P`**
- Then, semaphore operations can be defined as:

```
void wait(semaphore S) {
    S.count--;
    if (S.count < 0) {
        // add this process to the queue of sleeping/blocked processes
        enqueue(curproc, S.queue);
        sleep(curproc);
    }
}

void signal(semaphore S) {
    S.count++;
    if (S.count <= 0) {
        // there is a process blocked on the semaphore
        P = dequeue(S.queue);
        wakeup(P);
    }
}
```

- **`wait()`** and **`signal()`** are executed atomically (interrupts are disabled for each)

Semaphore as a General Synchronisation Tool

- The value of **`count`** is the number of processes/threads that can go past the **`wait()`** before a **`signal()`** is required, i.e. how many are allowed in between **`wait()`** and **`signal()`** at a time
- Initialise **`count`** to 0 to execute B in P_j only after A has executed in P_i

P_i	P_j
⋮	⋮
A	$wait(flag)$
$signal(flag)$	B

Semaphore Implementation of a Mutex

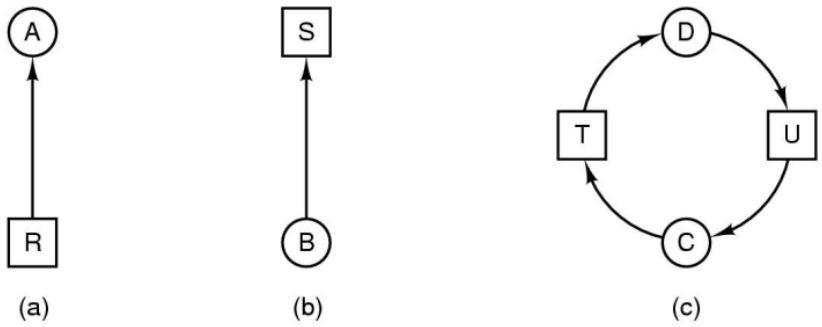
- Mutex = mutual exclusion (i.e. a lock)
- Initialise **`count`** to 1

```
semaphore mutex;
mutex.count = 1;
wait(mutex);      // enter the critical region
do_something();
signal(mutex);    // exit the critical region
```

- Programming with semaphores can be error-prone – e.g. with mutexes, must match a **`signal()`** for every **`wait()`**, otherwise a race condition may occur

Deadlock Modelling

- Deadlocks can be modelled with directed graphs



(a) Resource R assigned to process A
(b) Process B is requesting/waiting for resource S
(c) Process C and D are in deadlock over resources T and U

Dealing with Deadlock

Strategy	Description
Ostrich Algorithm	• Pretend there is no problem • Reasonable if deadlocks occur very rarely and the cost of prevention is high (e.g. it would mean only one process runs at a time)
Prevention	• Preventing one of the four conditions required for deadlock from occurring via resource allocation rules • Attacking the mutual exclusion condition is generally not feasible, as some resources are intrinsically not shareable • Attacking the hold and wait condition requires processes to request resources before it starts running, so a process never has to wait for what it needs – Not always possible – may not know required resources at start of run – Also ties up resources that other processes could be using • Attacking the no pre-emption condition means taking resources away – Not a viable option – Consider a process given access to the printer, when the printer is forcibly taken away halfway through its job • Attacking the circular wait condition: resource ordering – Gets rid of the circular dependency that causes deadlock
Detection and Recovery	• Detection: – Look for cycles in a directed dependency graph – Use an allocation/request matrix with the invariant: sum of current resource allocation + resources available = resources that exist • Assuming deadlocked is detected, restore progress to the system using pre-emption/rollback/killing processes – see below section
Dynamic Avoidance	• Allocating resources to processes in a way that deadlock is avoided, by only allowing progression into safe states (delay servicing some of the requests on purpose because otherwise deadlock might occur) • Can only be done if enough information is available in advance – maximum number of each resource required

- Going down the hierarchy:
  - Decreasing cost per bit
  - Increasing capacity
  - Increasing access time
  - Decreasing frequency of access to the memory by the processor
- Principle of locality: if we access certain data repeatedly, it gets loaded into the cache, so the next access will also likely come from the cache instead of main memory (RAM) instead of having to move it all the way up the hierarchy every time
- Implications:
  - Code that accesses things only once will be low performant
  - Code that accesses things repeatedly will be more performant (e.g. using the same function to do the same thing rather than writing 10 different functions to do the same thing), because it's more likely that the code or data being referenced will be higher up the hierarchy

Caching as a General Technique

- Caching is a general technique for speeding up access to data using intermediate storage – it doesn't necessarily refer to the hardware CPU cache
- For example, we can use main memory as a cache for the hard drive, or use the hard drive as a cache for static content loaded from the internet

Improving Performance

- If the compiler generates an instruction that accesses a variable, we need to load that variable from where it's stored. The instruction will issue an address, which goes into the cache, and the cache will either:
  - Miss, in which case the hardware then fetches that variable from main memory, or…
  - Hit, in which case the cache itself supplies the contents of that memory location at a much lower latency compared to main memory
- We can determine the Effective Access Time by computing the proportion of accesses that are resolved by the cache (fast memory) compared to the proportion of accesses that are resolved by slower memory

Effective Access Time of Memory Subsystem (Formula)

$$T_{\text{eff}} = H \times T_1 + (1 - H) \times T_2$$

$T_1 =$ access time of memory 1

$T_2 =$ access time of memory 2

$H =$ hit rate in memory 1

Example:

- Cache memory access time 1ns
- Main memory access time 10ns
- Hit rate of 95%

$$\begin{aligned} T_{\text{eff}} &= 0.95 \times 10^{-9} + (1 - 0.95) \times (10 \times 10^{-9}) \\ &= 1.5 \times 10^{-9}\ s \end{aligned}$$

File Access Types

- Sequential access
  - Read all bytes/records from the beginning
  - Cannot jump around to a specified location, but can rewind
- Random access
  - Read bytes/records in any order
- The OS can take advantage of sequential access to perform optimisations and do work in advance to make applications run faster
- Disks support random access but perform better when sequentially accessed

File Access Permissions

Permission	Description
None	• User may not know of the existence of the file • User is not allowed to read the directory that includes the file
Knowledge	• User can only determine that the file exists and who its owner is
Execution	• User can load and execute a program but cannot copy it
Reading	• User can read the file for any purpose, including copying and execution
Appending	• User can add data to the file but cannot modify or delete any of the file's contents
Updating	• User can modify, deleted, and add to the file's data • This includes creating the file, rewriting it, and removing all or part of the data
Changing Protection	• User can change access rights granted to other users
Deletion	• User can delete the file
Owner	• Has all rights previously listed • May grant rights to others using the following classes of users – Specific user – User groups – All for public files

UNIX Access Permissions

drwxrwxrwx

- First letter: file type (d for directories, - for regular files)
- Three user categories: user, group, and other
- Three access rights per category: read, write and execute
- The execute permission on a directory controls whether that directory is accessible
- The umask is a set of digits that correspond to each 'triple' of access permissions (rwx)
- Each file can only be associated with one group and there can only be one set of permissions for that group
- Shortcoming: the three user categories are rather coarse, e.g. can't easily assign different permissions on an individual basis to people not part of the same group (would need to use access control lists)

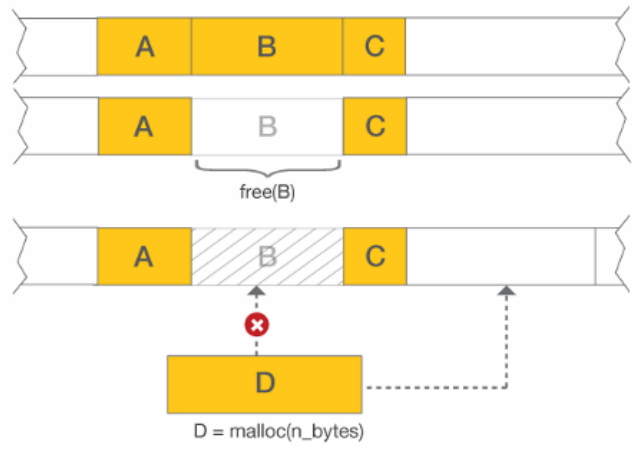
Simultaneous Access

- Most operating systems provide mechanisms for users to manage concurrent access to files
- Examples: `flock()`, `lockf()`, system calls
- Mutual exclusion and deadlock are issues for shared access

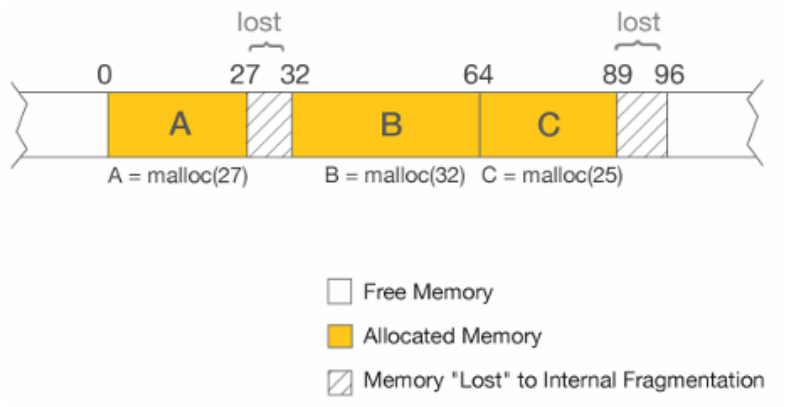
Method	Advantages	Disadvantages
Contiguous allocation	• Easy bookkeeping (only need to keep track of the starting block and length of the file) • Increases performance for sequential operations	• Need the maximum size for the file at the time of creation • As files are deleted, free space becomes divided into many small chunks (external fragmentation)
Dynamic allocation	• No external fragmentation • Does not require pre-allocating disk space	• Partially filled blocks (internal fragmentation) • File blocks are scattered across the disk • Complex metadata management (must maintain the list of blocks for each file)

External and Internal Fragmentation

- External fragmentation: memory space wasted external to allocated regions
  - Memory space exists to satisfy a request but it is unusable as it is not contiguous
  - For example, if blocks of memory are freed up because a file has been deleted, but they are not contiguous, a new file that would otherwise fit within those blocks cannot be written there as it does not fit in any single block



- Internal fragmentation: memory space wasted internal to allocated regions
  - Allocated memory may be slightly larger than requested memory
  - If the minimum allocation size is bigger than what is needed to store a file, the rest of the block will be wasted
  - We can't allocate smaller than the block size if we want to deal with memory in terms of equal-sized chunks rather than arbitrary-sized chunks



Buffering

- The idea of grouping together the information being received at a particular time into a larger entity (e.g. a packet instead of a byte at a time over a network)
- If the overhead is largely the same, it is much more efficient
- The more buffering you do in a fast network, the lower the throughput

Types of Buffering

Type of Buffering	Description
No buffering	• Process must read from/write to device a byte/word at a time • Each individual system call adds significant overhead • Process must wait until each I/O is complete
User-level buffering	• Process specifies a memory buffer that incoming data is placed in until it fills • Filling can be done by interrupt service routine • Only a single system call, and block/wakeup per data buffer (much more efficient)
Single buffer	• Operating system assigns a buffer in kernel's memory for an I/O request • User process can process one block of data while next block is read in • Swapping can occur since input is taking place in system memory, not user memory • Operating system keeps track of assignment of system buffers to user processes • If kernel buffer is full, start to lose characters or drop network packets
Double buffer	• Use two system buffers instead of one • A process can transfer data to or from one buffer while the operating system empties or fills the other buffer • May be insufficient for really bursty traffic
Circular buffer	• More than two buffers are used • Each individual buffer is one unit in a circular buffer • Used when I/O operation must keep up with process

These all attempt to address the *bounded-buffer producer-consumer problem*.

12. Multiprocessors

Learning Outcomes

- An understanding of the structure and limits of multiprocessor hardware
- An appreciation of approaches to operating system support for multiprocessor machines
- An understanding of issues surrounding and approaches to construction of multiprocessor synchronisation primitives

Multiprocessor Systems

- Multiprocessor systems use more than one CPU to improve performance
- Assumes:
  - Workload can be parallelised
  - Workload is not I/O-bound or memory-bound
- Shared-memory multiprocessors: more than one processor sharing the same memory