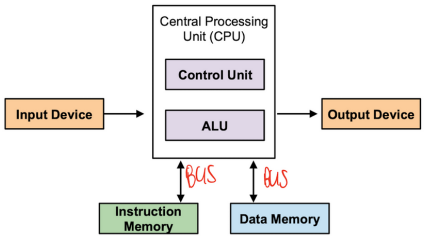
CS2106 Midterm cheatsheet

By: Khoo Jing Hong, Derrick

Introduction to OS:

Defn: Program that acts as intermediary betwn computer user and computer hardware.

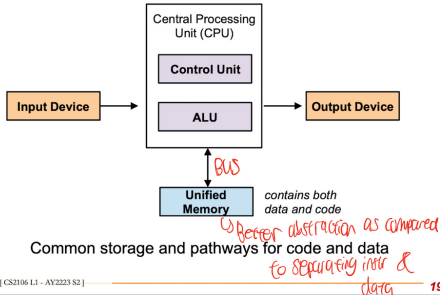
Harvard architecture



Separate storage pathways for code and data

*Inspired by **Harvard Mark I's separation of data and code (punch tapes)**

The **von Neumann** Architecture



Batch OS execute user program(job) one at a time. **Inefficient** because CPU idle when performing I/O operations.

Time-Sharing OS (server alike) User job scheduling involved, leading to **illusion of concurrency**. Users interact with machine using terminals (teletypes). Provides sharing of **CPU time, memory, storage.** Virtualisation of hardware also present as each program executes **as if** it has all the resources to itself.

OS is an abstraction. Provides **ease of programming, portability, efficiency.**
OS is a resource allocator. Manages all resources and arbitrate potentially conflicting requests for efficiency and fairness.
OS is a control program. It controls execution of programs to prevent errors and improper usage, and also provides security and protection.

Operating System Structures:

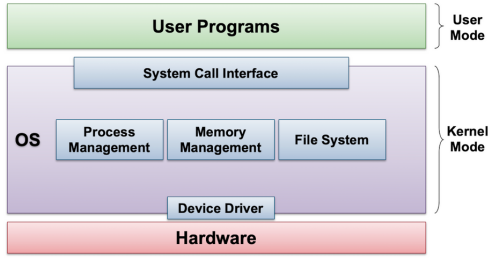
Factors

- Flexibility
- Robustness
- Maintainability
- Performance, scalability

OS as program deals with hardware issues, provides syscall interface and special code for interrupt handlers, device drivers.

Monolithic

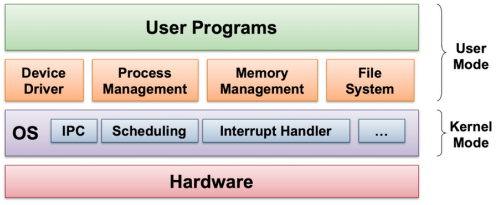
Kernel is 1 big special program
Adv: Performance
Disadv: Complicated, highly coupled components leading to a decrease in reliability (when 1 fails, all fails).



Generic Architecture of Monolithic OS Components

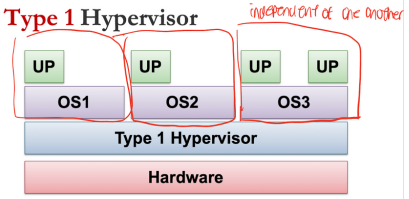
Microkernel

Kernel is very small and clean and provides inter-process communication(IPC)
Adv: Kernel more modular and robust, more extendible and maintanable, better isolation and protection betwn kernel and higher-level services (increase in reliability)
Disadv: Lower performance

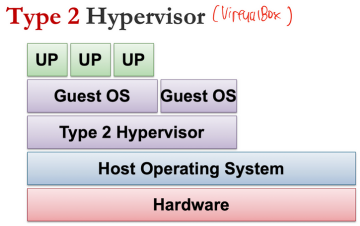


Generic Architecture of Microkernel OS Components

Virtual Machine is a software emulation of hardware. Provides virtualisation of underlying hardware. Can be used to observe the working of OS.
Virtual Machine Monitors Hypervisors.
Type 1 Hypervisor provides individual VMs to guest OSes. (eg: Oracle VM server)



Type 2 Hypervisor runs in host OS, and guest OS runs inside VM.



Process Abstraction:

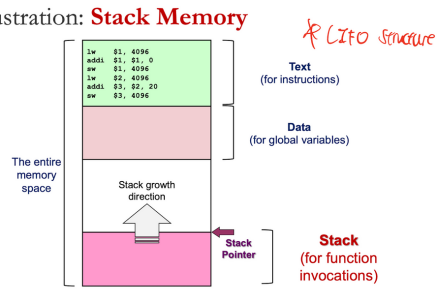
Memory is the storage for instr and data, and managed by OS, accessed through LOAD/STORE instrs.
Cache is fast and invisible to software. Duplicate part of memory for faster access and is split into instr cache and data cache.
Fetch Unit loads instr from memory through Program Counter(PC)
General Purpose Registers(GPR) are registers accessible by user program.
Special Registers are registers such to store PC, Stack Pointer(SP), Frame Pointer(FP).

Function calls:

When f() calls g() -> f() is caller, g() is callee
Stack Frame is created during a function invocation. Top of stack is indicated by **Stack Pointer(SP).**

Stack Memory:
Stack either grows towards higher or lower addresses -> Platform-dependent (Hardware dependent)

Illustration: **Stack Memory**

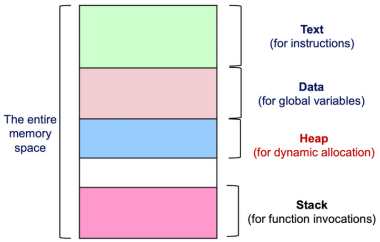


Frame Pointer(FP) is used to traverse and access items in a stack. It is for convenience when there are conditional branches.

GPRs are limited, eg MIPS has 32 while x86 has 16. When GPRs are exhausted, use memory to temporarily hold GPR value, then that GPR can be reused for other purpose and then the GPR value is restored afterwards (**register spilling**).

Stack items include:
- return PC
- saved FP (optional, depends if FP avail)
- saved SP
- saved GPR registers from caller
- Parameters
- Local variables

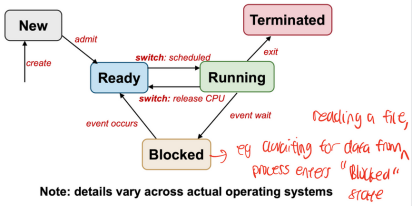
Heap memory(Dynamically allocated memory):
Separate in memory, in heap memory region.



** Process giving up CPU voluntarily -> Running to Ready **ONLY** (eg time quantum used up)

Process State

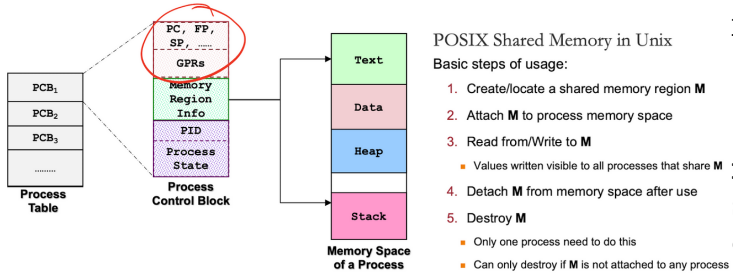
Generic 5-State Process Model



Global view:
Given n processes,
- With 1 CPU core
  - <= 1 process in running state
  - conceptually 1 transition at a time
- With m CPU cores
  - <= m process in running state
  - parallel transitions possible

Next process to run is determined by OS scheduling algorithm.
Process Control Block(PCB) maintained by Kernel in OS. **Process Table** keeps track of all PCBs.

Illustration: **Process Table**



Interrupts. OS does not interfere with hardware operations when an interrupt occurs.
Status registers are registers that saves current state of CPU and enables/disables interrupt

Process Creation in UNIX
fork(): returns PID child process = 0; parent = pid of child process.
Child process is a **duplicate** of current executable image (same code, same address space etc) but data in child is a **COPY**(local variable)
exec(): Execute another existing program in child process instead. However, **parent-child relationship not destroyed.**

Master process init process. Created in kernel at bootup time and has PID = 1.
Process Termination using exit(value). value == 0 means successful execution while value != 0 means problematic execution.
Parent-Child synchronisation:
Parent process needs to block until all child terminates, if not there will be **zombie** processes

Zombie processes Child processes that have terminated but resources are still present in the system because parent did not call wait.
Orphan processes Parent process terminated before child process. This causes child process to send signal to init which utilises wait() to cleanup.

Memory Copy Optimisation:
Copy on Write: Only duplicate a "memory location" when it is written to, otherwise parent and child share same memory location.

Inter-Process Communication:
2 IPC Mechanisms: **Shared-Memory** and **Message Passing**
Shared Memory OS is involved only in creation and attaching of shared memory
Adv: Efficient because OS needed only to setup shared regions.
Ease of use because there is just simple reads and writes to arbitrary data types
Disadv: Limited to a single machine, and requires synchronisation (need to synchronise accesses if not data racing occurs -> incorrect behaviour)
Race conditions due to possible interleaving of read and write operations.

Example creation of Shared Memory

Example: Master program (1/2)

creator of shared memory region

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    int shmid, i, *shm;

    shmid = shmget( IPC_PRIVATE, 40, IPC_CREAT | 0600);

    if (shmid == -1){
        printf("Cannot create shared memory!\n");
        exit(1);
    } else
        printf("Shared Memory Id = %d\n", shmid);

    shm = (int*) shmat( shmid, NULL, 0 );
    if (shm == (int*) -1){
        printf("Cannot attach shared memory!\n");
        exit(1);
    }
```

The master program create the shared memory region and wait for the "worker" program to produce values before proceeding.

Step 1. Create Shared Memory region.

Step 2. Attach Shared Memory region.

Example: Master program (2/2)

first element is a flag

```c
    shm[0] = 0;

    while(shm[0] == 0){
        sleep(3);
    }

    for (i = 0; i < 3; i++){
        printf("Read %d from shared memory.\n", shm[i+1]);
    }

    shmdt( (char*) shm);
    shmctl( shmid, IPC_RMID, 0);

    return 0;
}
```

The first element in the shared memory region is used as "control" value in this example (0: values not ready, 1: values ready).

The next 3 elements are values produced by the worker program.

master waits for worker to change shm[0] flag

Step 4+5. Detach and destroy Shared Memory region.

Example: Worker program

```c
//similar header files
int main()
{   int shmid, i, input, *shm;

    printf("Shared memory id for attachment: ");
    scanf("%d", &shmid);

    shm = (int*)shmat( shmid, NULL, 0);
    if (shm == (int*)-1) {
        printf("Error: Cannot attach!\n");
        exit(1);
    }

    for (i = 0; i < 3; i++){
        scanf("%d", &input);
        shm[i+1] = input;
    }
    shm[0] = 1;

    shmdt( (char*)shm );
    return 0;
}
```

Step 1. By using the shared memory region id directly, we skip ***shmget***() in this case.

get id from console

Step 2. Attach to shared memory region.

Write 3 values into shm[1 to 3]

Let master program know we are done!

Step 4. Detach Shared Memory region.

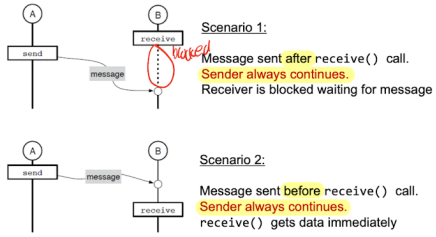
Message Passing involves OS in every message to and fro (syscalls)
Message stored in **kernel memory space**
Direct Communication Sender/Receiver explicitly names other party(eg by PID)
Characteristics:
One link per pair of communicating processes
Need to know identity of other party

Indirect Communication Message are sent to/received from mailbox
Characteristics: OS handles who to recv msg
One mailbox can be shared among processes

Blocking Primitives for CS2106

Receiver:
Blocking receive Receiver must wait for a message if it is not already available
Sender:
Either blocking send or non-blocking send(async message passing)
Problem Finite buffer size means system is truly not async (sender will wait when buffer is full or returns with error)

Async Message Passing example



Scenario 1:
Message sent after receive() call.
Sender always continues.
Receiver is blocked waiting for message

Scenario 2:
Message sent before receive() call.
Sender always continues.
receive() gets data immediately

Buffer creation(mailbox creation)
Large buffer decouples sender and receiver, making them less sensitive to variations in execution; Do not wait for each other unncessarily

- User may need to declare in adv capacity of mailbox

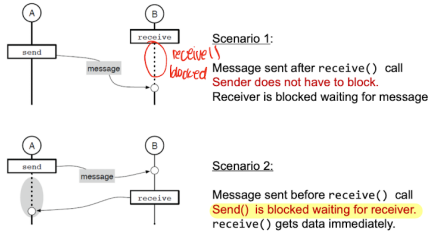
Synchronous Message Passing

- AKA rendezvous

Sender is blocked until receiver performs matching receive(), sender is forced to wait till receiver is ready

- No intermediate buffering required

Synchronous Message Passing Example



Scenario 1:
Message sent after receive() call
Sender does not have to block.
Receiver is blocked waiting for message

Scenario 2:
Message sent before receive() call
Send() is blocked waiting for receiver.
receive() gets data immediately.

Advantages:

- Applicable beyond single machine
- Portable
- Easier synchronisation

Disadvantages:

- Inefficient (requires OS intervention upon every send/receive)
- Harder to use (requires packing / unpacking data into a supported msg format)

Unix Pipes for IPC

In order passing of data from A->B given A|B
UNIX shell pipe normally has 1 writer and 1 reader

Redirection of Output in UNIX

fd values: stdin = 0, stdout = 1, stderr = 2
Use dup2(int oldFileDescriptor, int newFileDescriptor)
to align oldFd to newFd(output in newFd will show in oldFd)

Process Scheduling

Realise that context switching causes OS to incur overhead in switching betwn processes.
Refer to time when process uses CPU as **CPU burst.**
Refer to time when process uses I/O as **I/O burst.**
Compute-Bound processes: Majority of time spent using CPU
I/O Bound processes: Majority of time spent using I/O

Criteria for all processing environments:

- Fairness (also means no starvation)
- Utilization (all parts of computing system should be utilized)

Scheduling policies:

- Non-preemptive(Cooperative)
  - Process stays in running state until it blocks or gives up CPU **voluntarily**
- Preemptive
  - Process gets forcefully removed once time quantum is reached

Batch Processing Algorithms

- No user interaction (Don't care about response time)
- Non-preemptive scheduling is predominant

Criterias for batch processing:

1. Turnaround time (Finish time - arrival time) or total time taken
2. Waiting time (Time spent waiting for CPU)
3. Throughput (Number of tasks finished per unit time)
4. Makespan (Total time from start to finish to process all tasks)
5. CPU Utilization (Percentage of time when CPU is working on a task)

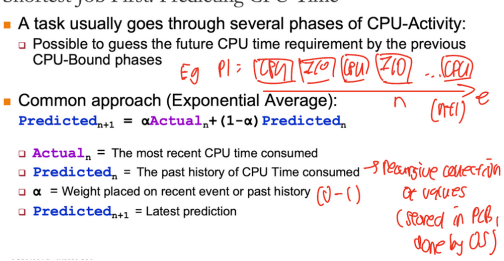
Batch Process Scheduling Algorithms (No time quantum discussion)

1) **First-Come First Served(FCFS)**
Tasks are placed in **FIFO ready queue**, based on arrival time

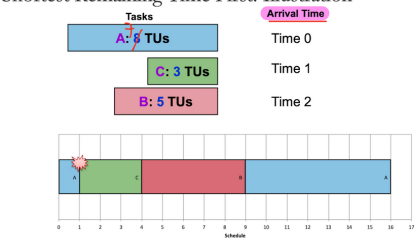
- Task will run until it is done or is blocked. When ready again, rejoin queue from back

Guaranteed to have **no starvation**
However, reordering tasks may **reduce average waiting time. (Shorter jobs at front of queue)**
Problem
Convoy effect eg first task is CPU-bound then remaining tasks are I/O bound. Then CPU is idle when first task blocks on I/O and remaining tasks run on I/O. (Poor utilization of resources)

2) **Shortest Job First (SJF)** -> Minimize average waiting time
Select task with **smallest total CPU time**
Problem
Starvation due to bias towards shorter jobs
Prediction of CPU time needed.



3) **Shortest Remaining Time (SRT)** -> Variation of SJF that is preemptive instead
New job that arrives with shorter remaining time can preempt current running job



Interactive Systems Scheduling Algorithms (Time quantum discussion)

Criterias for interactive systems:

- Response time (**minimized through shorter time quantum**)
- Predictability (Variation in response time, lesser variation == more predictable)

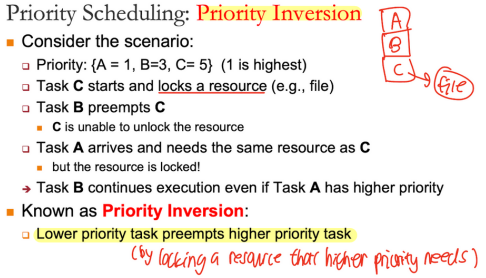
Preemptive scheduling is predominant here to ensure good response time

Time Quantum is the execution duration given to a process. **Short time quantum** = Minimise response time but bigger overhead in context switching. **Long time quantum** = Reduce overhead in context switching for OS, but better CPU utilization.

1) **Round Robin (RR)**
Tasks are stored in FIFO queue. Tasks are picked from front of queue and runs until fixed time slice(quantum) elapsed || task gives up CPU voluntarily || task blocks

- Preemptive version of FCFS
- Response time guarantee
  - Given n tasks and quantum q
  - Time before a task gets CPU is bounded by (n-1)q

2) **Priority Scheduling (No time quantum)**
2 variants: **Preemptive** where higher priority process can preempt current running process with lower priority or **Non-preemptive** where higher priority process has to wait for next round of scheduling.
Problem
Starvation of lower priority processes if high priority process keep hogging CPU



Interactive Systems Scheduling Algorithms (Time quantum discussion)

3) **Multi-Level Feedback Queue(MLFQ)**

MLFQ **minimises both response time** for I/O bound processes(interactive systems) and **turnaround time** for CPU bound processes (batch scheduling)

MLFQ: Rules

- Basic rules:
  1. If Priority(A) > Priority(B) ➔ A runs
  2. If Priority(A) == Priority(B) ➔ A and B runs in RR
- Priority Setting/Changing rules:
  1. New job ➔ Highest priority
  2. If a job fully utilized its time quantum ➔ priority reduced
  3. If a job gives up / blocks before finishes its time quantum ➔ priority retained

Advantage:

Since I/O jobs are usually short jobs, by having an appropriate time quantum, I/O bound processes are always high priority -> **Minimize response time**

Problem

Able to abuse MLFQ by blocking before time quantum.

4) **Lottery Scheduling**

MLFQ: Rules

- Basic rules:
  1. If Priority(A) > Priority(B) ➔ A runs
  2. If Priority(A) == Priority(B) ➔ A and B runs in RR
- Priority Setting/Changing rules:
  1. New job ➔ Highest priority
  2. If a job fully utilized its time quantum ➔ priority reduced
  3. If a job gives up / blocks before finishes its time quantum ➔ priority retained

Lottery Scheduling: Properties

- Responsive:
  - A newly created process can participate in the next lottery
- Provides good level of control:
  - A process can be given Y lottery tickets
    - It can then distribute to its child process
  - An important process can be given more lottery tickets
    - Can control the proportion of usage
  - Each resource can have its own set of tickets
    - Different proportion of usage per resource per task
- Simple Implementation

Synchronisation primitives

Synchronisation is needed to control the interleaving of accesses to a shared resource among processes

Solution Critical Section(CS) for segment of code to write or read a resource.

Overhead Performance due to enforcing of CS.

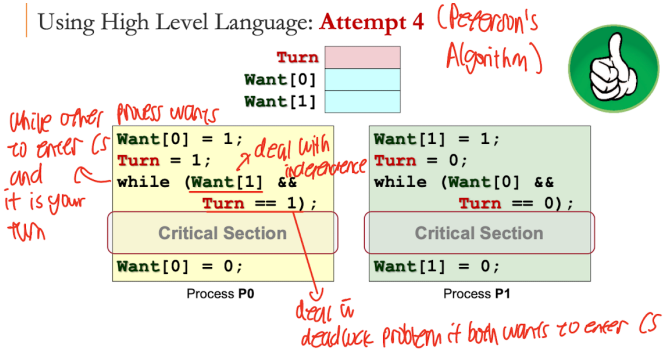
Properties of Correct CS:

- Mutual Exclusion
- Progress (If no process in CS, one of the waiting processes to enter CS should enter)
- Bounded Wait (there exists an upper bound on the number of times other processes can enter CS before Pi, after Pi requests to enter CS)
- Independence (Process not executing in CS should not block other process from entering CS)

Symptoms of wrong implementation of CS:

- **Deadlock** (all processes blocked)
- **Livelock** (when processes keep switching state to avoid deadlock but make no progress)
  - Usually a result of deadlock avoidance mechanisms
- **Starvation**
  - Some processes are blocked forever

Peterson's Algorithm(No semaphore, code level)



Note that switching places of Turn and Want will not work due to **violation of mutual exclusivity.**

Synchronisation primitives

Disadvantages of Peterson:

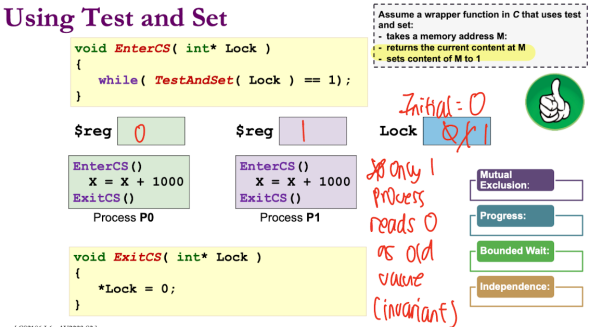
- Busy waiting -> Unwanted usage of CPU
- Too low-level -> Error prone
- Not general -> Hard to solve all synchronisation problems for n processes

Test-And-Set (Assembly level sync)

- Machine instruction to aid synchronization
  - Commonly found in modern processors

```
TestAndSet Register, MemoryLocation
```

- **Behavior:** (reg [lock])
  1. Load the current content at `MemoryLocation` into `Register`
  2. Stores a **1** into `MemoryLocation`
- Important: The above is performed as a **single atomic machine operation**
  - Even in multi-core systems!



Problem

Busy Waiting -> Wasteful CPU usage

No guarantee on bounded-wait -> Unless scheduling algorithm is fair

Using Semaphores

- A semaphore **S** contains an integer value (Clock)
  - Can be initialized to any non-negative values initially
- Two atomic operations:

`Wait( S )`	`Signal( S )`
If `S` <= 0, **blocks** (go to sleep)	**Increments** `S`
Decrement `S`	Wakes up one sleeping process if any
Also known as `P()` or `Down()`	This operation **never** blocks
	Also known as `V()` or `Up()`

Synchronisation primitives

Semaphores

Semaphores: Properties

- Given:
  - $S_{Initial} \geq 0$
- Then, the following **invariant** must be true:

$$S_{current} = S_{Initial} + \#signal(S) - \#wait(S)$$

successfully incremented semaphore value

successfully decrementation semaphore value

`#signal(S)` :
- number of signals() operations executed

`#wait(S)` :
- number of wait() operations **completed**

General Semaphores(Counting semaphores)

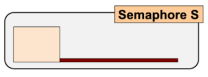
Used in Safe-Distancing problem (allow up to n number of processes to execute)

- Initialise semaphore with n

Binary Semaphores (Mutex semaphore)

- Binary semaphore `S = 1`

```
Wait( s );
Critical Section
Signal( s );
```



- In this case, **S can only be 0** or **1**
  - From the semaphore invariant

Summary: Most Common Uses Of Semaphores

N number of running processes allowed

Critical Section	Safe-distancing problem	General synchronization

```
BinarySemapore mutex = 1;

void use_restroom(){
   wait(mutex);
   critical_section();
   signal(mutex);
}
```

```
Semaphore sem = N;

void cient(){
    wait(sem);
    safe_eating();
    signal(sem);
}
```

```
Semaphore sem = 0;

Process P1:
    produce(X);
    signal(sem);
```

wait() in P1 produce() is called

```
process P2:
    wait(sem);
    consume(X);
```