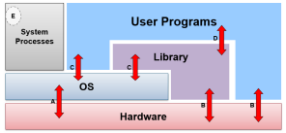
CS2106 AY23/24 Midterms CheatSheet

Chapter 1: Operating Systems



OS as a Program

- OS is also known as the kernel
  - A program with some special features
    - ➢ Deals with hardware issues
    - ➢ Provides **system call interface**
    - ➢ Special code for interrupt handlers, device drivers
- Kernel code has to be different than normal programs:
  - **NO** use of system call in kernel code
  - **CANNOT** use normal libraries
  - **NO** normal I/O

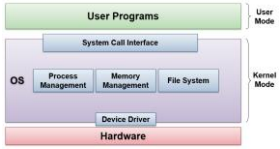
System calls can be both synch/ async, depending on the nature of call

Hardware	Physical components of a computer system
OS	Acts as an intermediary between the hardware and application software.
User Programs	Application softwares, normal programs that run on top of the OS.

OS Structures

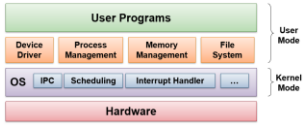
Monolithic

- Kernel → one BIG Program
- ✓ Well Understood
- ✓ Good Performance
- × Highly Coupled Components
- × Complicated Internal Structures (For protection etc)



Microkernel

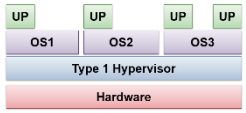
- Kernel → Small & Clean w basic, essential features.
- Uses IPC to communicate
- Built on top of basic facilities
- ✓ Kernel is more robust generally
- ✓ More extendible (scalable)
- ✓ Better isolation & protection between kernel and high level services (More abstraction)
- × Lower Performance



Virtual Machines

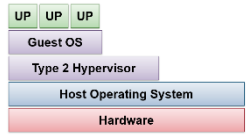
- A software EMULATION of hardware
- Virtualization of underlying hardware
- Normal (primitive) OS can then run on top of the VM
- AKA Hypervisor: Type 1 & Type 2

Type 1 Hypervisor



- Type 1 hypervisor OS:
  - Provides individual ***virtual machines*** to guest OSes
- Type 2 → Saves 1 level of abstraction → saves time and latency
- VM provides a way to observe and analyze (debug, test, scale) the behavior of OS in a controlled env. → Can use debugging tools as well. (No such thing for OS debug).
- VM enables us to run several OSes on the same hardware ATST.

Type 2 Hypervisor

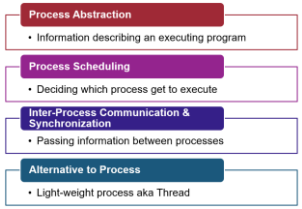


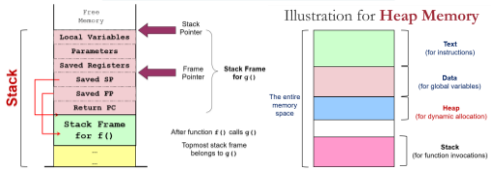
- Type 2 hypervisor OS
  - Runs in host OS
  - Guest OS runs inside Virtual Machine

Chapter 2: Process Abstraction

- The OS should provide **efficient use of the hardware recourses**:
  - By managing the programs and processes executing on the hardware
- The Hardware can only execute 1 instruction at a time. (Either access programs, or read the I/O)
  - CPU is IDLE when I/O is being read
  - I/O is IDLE when CPU is running
- This can be achieved via Multiprogramming, Time-sharing.
  - **Multiple programs** share the hardware

- To be able to run and switch between different programs, additional information from the program(s) needs to be stored and saved somewhere. → Context Switching

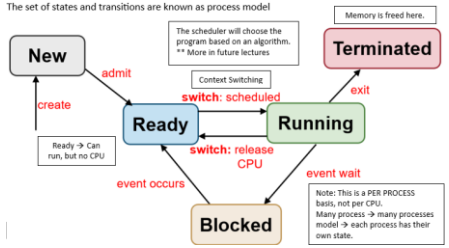




** Implementation Dependent**

Process ID & Process State

- PID → used to identify different processes. PID changes when process dies (i.e it is not static). PID is unique among processes.



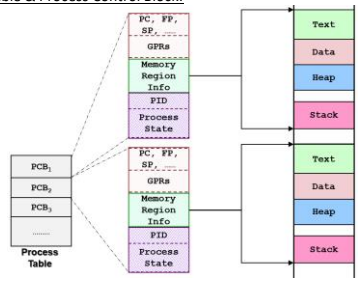
5-Stage Model

New	New process created. May still be initializing (not ready)
Ready	Process is waiting to run
Running	Process is being executed on CPU
Blocked	Process waiting (sleeping/IDLE). Can't execute currently
Terminated	Process finished execution, may need OS Cleanup

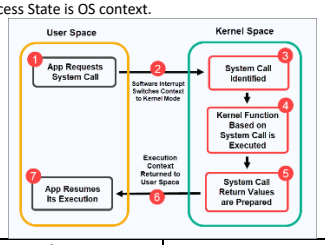
Process State Transitions

Create (nil → New)	• New process created • CPU sets up env required for the process
Admit (New → Ready)	Process is ready to be scheduled for running
Switch (Ready → Running)	Process is selected to run
Switch (Running → Ready)	Process gives up CPU voluntarily, or pre-empted by scheduler (context switch)
Event Wait (Running → Blocked)	Process requests for event/resource/svc which is not available currently.
Event Occurs (Blocked → Ready)	Event occurs → Process can continue

Process Table & Process Control Block:



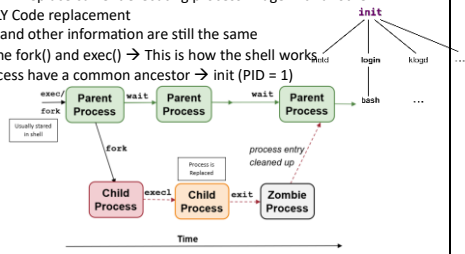
- The kernel maintains the PCB for all processes.
- PID & Process State is OS context.



Exception	Interrupt
• Executing a machine level instr can cause exception • Exception is synchronous - Occurs due to prog exec • Exception handler auto exec - "Forced" function call	• External events can interrupt exec of a program • Interrupt is Asynchronous - occurs indep of prog exec • Prog exec is suspended. • Interrupt handler auto exec

Parent fork() Child and Zombies, exec()

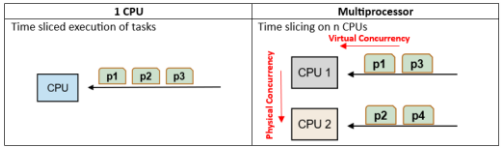
- The child only differs in: PID, Parent PID, fork() return value.
- The child process is a duplicate of the parent. Have same code, address space etc (data in the child is a COPY, NOT SHARED Memory)
  - Parent and Child have independent memory spaces.
  - Note: when fork() called, file descriptors same → Same pipe used.
- execX() → replace current executing process image with another
  - ONLY Code replacement
  - PID and other information are still the same
- Combine fork() and exec() → This is how the shell works
- All process have a common ancestor → init (PID = 1)



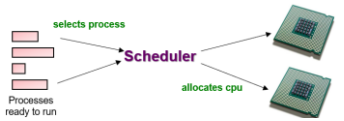
- RMB TO call wait(), to cleanse the zombies 😊
- Zombie processes are "dead" on the PCB, cant kill() it... use wait()
  - If parent terminates before waiting, init adopts child and wait()
  - Don't let Zombies hog up CPU space & memory
- Synchronization (Parent ←→Child) via wait()

Chapter 3: Process Scheduling

- Concurrency → Switch between process super fast (1 core)
- Multiple cores → N cores, do N instructions max at one time
- Interleaving of instructions is possible due to context switching



Scheduler in OS (Scheduler, Scheduling Algorithm)



Scheduling a Process: Step-by-Step

1. Scheduler is triggered (**OS takes over**)
2. **If Context switch is needed:**
   - Context of current running process is saved and placed on blocked queue / ready queue
3. Pick a suitable process **P** to run base on scheduling algorithm
4. Setup the context for **P**
5. Let process **P** run

Non-Pre-empting (Cooperative)

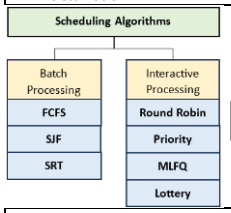
- A process stayed scheduled until:
  1. It blocks OR
  2. Give up the CPU voluntarily.
- A process stayed scheduled (in running state) until it blocks or give up the CPU voluntarily

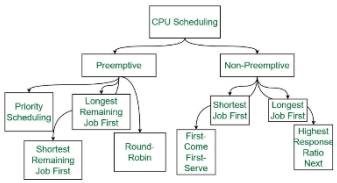
Pre-emptive

- A process is given a **fixed time quota** to run.
  - Possible to block or give up early.
  - At the end of the **time quota**:
    - The current process is suspended.
    - Another process gets picked if available

Criteria for Scheduling Algorithm

Fairness	Balance
• Each process get fair share of CPU Time. 1. Processes get same % CPU 2. Users get same % CPU regardless of #processes. • No starvation	• All parts of the computing system should be utilized. - Computing resources fully utilized - A program should not cause another to crash etc.

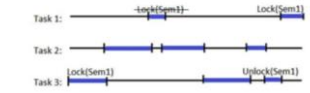




Batch Processing Scheduling Algorithm	
FCFS	• Pick 1st task in queue to run until task done/blocked • Removes block task from queue • Guaranteed no starvation • Reordering can reduce average waiting time
SJF	• Pick task with smallest total CPU time • Minimizes average waiting time (Need to know time req) • CPU guess future time based on previous phases: $Predicted_{n+1} = \alpha\, Actual_n + (1-\alpha) Predicted_n$
SRT	• Pick task with shortest REMAINING TIME (similar to SJF) • Preemptive: A new job with shorter remaining time priority • Possible starvation if there are lots of short jobs. Long 😳
Interactive Processing Scheduling Algorithm	
RR	• Preemptive version of FCFS (FCFS with time limit) • Pick 1st task from queue to run until done/blocked/Timesup • Given n tasks and quantum q. Upper bound wait: (n-1)q • Timer interrupt needed to check on time quantum left
Prio	• Assigns priority value to each tasks, pick highest prio task. • Preemptive version: (High Priority always goes first) - Higher priority process can preempts running process with lower priority

Types of System Calls:Process Control, File Management, Device Management, Information Maintenance, Communication
E.g execl(), fork(), exit() *** System calls can only happen synchronously.
System calls and interrupts all require mode switches (user <-> kernel). A system call does not generally require a context switch to another process

- SJF is the most optimal among non-preempt scheduling algorithms for minimizing average waiting time & average turnaround time
- If pre-emption allowed, SJF can be outperformed by SRT when jobs arrive at different times (**CS diminished benefit of preempt)
- Not all pre-emptive algo have time quantums (e.g SRT), just checks #time needed for process to finish, no time quantum needed

	• Non-preemptive version: (Higher Priority need to queue) - Late coming high priority process has to wait for next round of scheduling • Low Priority processes can STARVE. (Wore in pre-empt var) • Priority Inversion Problem: ◦ A high priority task can become blocked by a lower priority task indefinitely ▪ If lower priority task locks access to resources shared by both tasks ◦ Since higher priority tasks are usually more important and have stricter deadlines, this can lead to system instability and further scheduling problems 
MLFQ	• Multi-Level Feedback Queue → Adaptive • Minimizes response time for IO bound processes • Minimizes turnaround time for CPU bound processes 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 takes longer than time quantum → ↓priority 3. If a job finishes/blocks before time quantum → = prio - Fast jobs retain prio, slow jobs ↓ prio
Lottery	• Give lottery tickets to all processes • Random ticket chosen, winner gets CPU time • The higher the prio → more lottery tickets (higher %%) • A process holding X% of tickets have X% chance of winning → Use CPU X% of the time

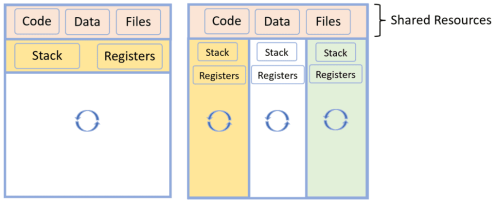
- Time Quantum MUST be multiples of Interval of Time Interrupt (ITI)
- The longer the time quantum, the lesser the #Context Switches
  - Bigger Quantum → Better CPU utilization but longer waiting time
  - Small Quantum → Bigger overhead, but shorter waiting time

Chapter 4: Threads

- Threads are used to make communication between processes easier and cheaper (lesser context switching + memory), since processes are exp.
- With more threads(of control), multiple parts of the program can be executing at the same time → multiple instructions execute FAST.

Process & Thread

- A single process can have multiple threads → aka multithreaded process
  - Every thread will have their own stack → OS implementation driven
  - Every thread will have their own hardware context e.g GPRs, SR etc



Shared Information	Unique/Personal information
• Threads in same process share - Mem context: Text, Heap, Data - OS Context: PID, files etc - Global Var can be shared via threads as well	• Unique Info needed for each thread: - Thread ID (TID) - Registers (GPR, SR) - "Stack"

Note: Hardware context of processes are **NOT duplicated**, they are **shared**

- For multithreads, we just need to maintain the **stack and frame pointer** values for each thread. Thereafter, we just **point to the right values** to access the corresponding stack and frame pointer.

Process Context Switch vs Thread Switch

Process Context Switch	Thread Switch
• OS Context	• Hardware Context
• Hardware Context	- Registers
• Memory Context	- "Stack" (FP and SP reg)

** Threads are lightweight processes

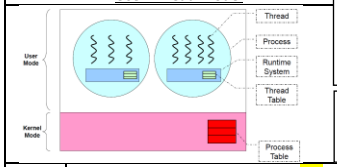
Thread	Process
• Uses Native Threads • Belongs to a Process • Lightweight, fast to start • Shared Memory • Subject to the GIL • Suited to IO-bound Tasks • Create 10s to 1000s Workers	• Uses native Processes • Has Threads and Child Processes • Heavyweight, slow to start • Inter-Process Communication • Not Subject to the GIL • Suited to CPU-bound Tasks • Create 10s of Workers

Pros of Threads	Cons of Threads
• **Economy** - Multiple threads in the same process require less resources to manage as compared to multiple processes. • **Resource Sharing** - Threads share most of the resources of a process - There is no need for additional mechanism to pass information around. - Reduced no. of switches • **Responsiveness** - Multithreaded programs can appear much more responsive. (Context switching faster, less info to be swapped/switched) • **Scalability** - Multithreaded program can take advantage of multiple CPUs. → Better load balancing as well.	• **System Call Concurrency**: - Parallel execution possible - Issue of Re-entrancy: ➢ A function can be called again before completed ➢ Multiple instances of the function called simultaneously ➢ Race Conditions. Data corrupted/ Wrong 😩 • **Process Behavior:** - Threads are implementation dependent. - Different OS manages threads differently.

- Sharing memory is a double-edged sword
- **Good**: Processes can communicate with each other without using costing context switching.
- **Bad**: Adversaries may steal data/ Data may leak out of the program.

Thread Models

User Thread Model

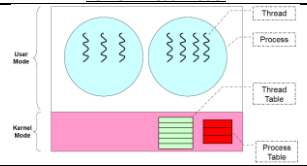


We can customize the library calls to specific process's use case → Save runtime time, more space efficient

All threads run on the same CPU → only can run 1 thread at a time 😩

Pros	• Can have multithreaded program on ANY OS • Thread operations are just **library calls** → no switching costs (savetime) • Generally more configurable and flexible - e.g. Customized thread scheduling policy
Cons	• Kernel is not aware of threads, scheduling is performed at process level - One thread blocked →Process blocked → All threads blocked - **Cannot exploit multiple CPUs!**

Kernel Thread Model



Pros	• Kernel can schedule on thread levels (Thread-Scheduling) flexibility • Parallel execution: More than 1 thread in the same process can run simultaneously on multiple CPUs →i.e can manage each thread individually.
Cons	• Thread operations is now a system call! - Slower and more resource intensive →requires switching kernel <-> user (since thread is implemented in the OS) • Generally less flexible: - Used by all multithreaded programs

Too few features	Too many features
Not flexible enough for some complex programs	Expensive, overkill for simple programs

Hybrid Thread Model

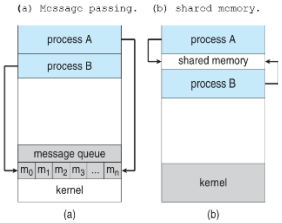
- Have both kernel and user threads
  - OS schedule on KERNEL threads ONLY
  - User thread can BIND to a kernel thread for execution
- Flexible → Limit the concurrency of any process/user by limiting the #user threads that can be active and bound to kernel threads → prevent excess concurrency
- Supports multiple threads, but CANNOT do context switching.

Generally, user threads are much faster than kernel threads, due to the switching costs

Chapter 5: Inter Process Communication (IPC)

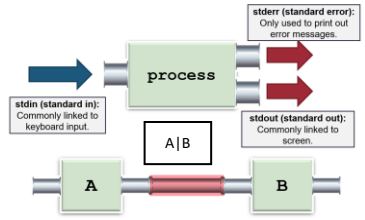
- IPC is used to facilitate processes to exchange information/communicate with one another, regardless of the processes' independent memories.
  - This enables process to navigate critical sections, and prevent race conditions (via Semaphores, Mutexes etc)
- IPC has 2 common mechanisms:

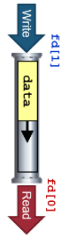
Shared Memory ** Can be done via Master-Slave Process	Shared memory allows multiple processes to access the same region of memory • P_1 creates a shared memory region M • P_2 attaches mem region M to its own mem space • P_1 and P_2 can now communicate using M - M behaves very similar to normal mem region - Any writes to the region can be seen by all other parties ✓ Efficient: Only initial steps (Create & Attach) involves the OS → few switches needed → saves clock cycles ✓ Ease of Use: Behaves similar to normal memory, any info can be conveniently shared. × Synchronization: Need to sync to prevent race cond. × Harder to implement: Memory segments have to be handled with care (esp attachment, deletion, creation)
Message Passing	Message passing involves sending and receiving messages between processes. (Direct & Indirect) • P_1 prepares a message M and send it to P_2 • P_2 receives the message M • Message sending and receiving are usually provided as system calls **Direct**: Sender/Receiver of message explicitly name the other party **Indirect**: Message are sent to / received from message storage (mailbox, mailbox ID) 2 Sync Behaviors: **Blocking Primitives (Synchronous):** • Send() → sender blocked until the msg is received • Receive() → Receiver blocked until a msg has arrived. **Non-Blocking Primitives (Asynchronous)** *need promises • Send() → Sender resume operation immediately • Receiver() → Receiver either receive the message if available/some indication that message not ready yet Pros: • **Portable** - Can easily be implemented on different processing environment • **Easier Synchronization** - E.g When synchronous primitive is used, sender & receiver are implicitly synchronized. Cons: • **Inefficient** - Usually require OS intervention • **Harder to use** - Msg are usually limited in size and/or format - IPC requires OS calls as compared to just pointer access (in shared memory)



- There are also 2 Unix-Specific IPC Mechanisms:

Pipe	Pipes allow communication between two processes through a unidirectional data stream
Signal	Signals are used to notify processes of asynchronous events or to communicate simple messages between processes.





- 2 processes can be connected together via pipes.
  - Pipes are just file descriptors that point where the signal goes:
  - Pipes enables 2 processes to share information with each other
    - ➢ fd[0] → Output signal (Reading end) [Simple, 0 look like O 😊]
    - ➢ fd[1] → Input signal (Writing end)
- The output of A (instead of going to the screen) directly does into B as input (as if it come from keyboard)
  - Its like a redirection of signals 😊
  - We need to control the flow of signals properly, else deadlock 😩

UNIX Pipes

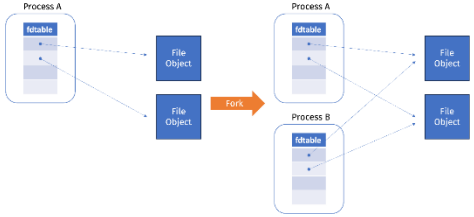
- A pipe can be shared between two processes
- A form of **Producer-Consumer** relationship
  - P produces (writes) n bytes, Q consumes (reads) m bytes

Behaviour	• Similar to an anonymous file • FIFO → Must access data in order of arrival
Semantics	• Pipe functions as **circular bounded byte buffer** with **implicit synchronization** - Writers WAIT when buffer is FULL - Readers WAIT when buffer is EMPTY
Variants	• Can have multiple readers/writers - Normally → only have 1 writer, 1 reader - This can be modified via changing file desc Half-Duplex: 1 write end, 1 read end (Unidir) Full-Duplex: Any end can read/write • Named Pipes → private pipes for processes, Pipe ID

- dup() → Takes a file descriptor & assigns it to another file descriptor
- dup2() → Similar to dup1(), but we can specify where to copy to
- dup3() → Similar to dup2(), but with flag implementation (rwx flags)

Note:

- When a process opens a file – using pipe, in this question – a file object and a file descriptor are allocated on the kernel side.
- Then the file object's address is stored in the array at the position corresponding to the file descriptor.
- Upon forking a process, this array is duplicated, but the file objects themselves are not cloned.



- When fork() is called, both parent and child share the same file object

UNIX Signal

- UNIX Signal is a form of limited IPC
  - An asynchronous notification regarding an event.
  - Sent to a process/thread
- The recipient of the signal must handle the signal by:
  - A default set of handlers OR User supplied handler (for some signals)
- Common signals in Unix: Kill, Stop, Continue, Memory error, etc…
- We can attach the same handler to multiple signals