CS2106 Cheatsheet (Midterm)
by Yiyang, AY21/22

Operating System Introduction

OS Overview

Terminology

Multi-Programming - The ability of the OS to run multiple programs, of same of different users, simultaneously.
Time-Sharing - The ability of the OS to support multiple users to interact with machine using terminals. Need Multi-programming.

Common OS Structures

Monolithic - OS is a big, special program. The traditional approach taken by most Unix variants.
Pros: 1) Ability to follow good SE principles, 2) Better Performance
Cons: Highly coupled components and complicated internal structures

Microkernel - OS consists of many minimal kernels each providing basic and essential facilities, and microkernels are linked via IPC.
Pros: 1) Simplicity, 2) More robust and extensible, 3) Better isolation & protection **Cons**: Communication across components take time, which penalises performance

Other structures: **Layered Systems**, **Client-Server Model**.

Hypervisor

Hypervisors ~ monitors ("containers") for running virtual machines, which allows other OS to run and debug on it. There are 2 types of Hypervisors

- **Type 1 ~** - an OS working directly on hardware. Typically faster but more complicated. Expensive to acquire.
- **Type 2 ~** - a software (but not an OS) running in host OS. Simpler and can be used to emulate hardware you do not have.

Process Management

Process

A **Process** is an abstraction to describe a running program. A **Process Control Block** (PCB) (aka. **Process Table Entry**) is a unit entity storing necessary information (**Execution Context**) for a process. The **Process Table** stores all current PCBs.
Context for a **Context Switch** includes: 1) Registers (GPR & Special), 2) Stack Pointer, and 3) PC.

Generic 5-State Process Model

The Generic 5 States are: **New**, 2) **Ready**, 3) **Terminated**, 4) **Running**, and 5) **Blocked**.
Note: Some systems (including Unix) do not have New - whenever a process is created, it is in Ready.

In addition, in Unix, there are **Orphan Process**, which is a child process with parent terminated before it, so the orphan is "adopted" by `init` and waited properly when done, and 2) **Zombie Process**, which is a child that has exited but the parent is executing but does not call wait on the child.

Memory Regions

The Memory consists of 1) **Text** for instructions, 2) **Data** for global variables, **Heap** for dynamic allocation, e.g. via `malloc()` in C, and 4) **Stack** for function invocations.

Stack & Function Calls

Terminology

In a function call, **Caller** function invokes a function call of **Callee**.
A **Stack Frame** stores information about one callee invocation, and there are two associated pointers. **Stack Pointer** (SP) points to the top of the Stack while **Frame Pointer** (FP) points to a fixed location in a Stack Frame.
Note: FP facilitates easy access of Stack Frame items. It is platform dependent and not compulsory. Every platform has SP.
Register Spilling is the practice of using memory to temporarily store GPRs' values so that GPRs can used reused for other purposes and restored afterwards. It is used during callee invocation.
In a function call, **Stack Frame Setup** is the preparation to make a function call, and **Stack Frame Teardown** is the returning from function call.

Stack Frame Setup and Teardown Example

In **Frame Setup**, Caller: 1) Pass parameters with registers and/or stack, and 2) Save Return PC on stack; Callee: 1) Save old SP, 2) Allocate space for local variables on stack, and 3) Adjust SP to point to new stack top.

In **Frame Teardown**, Callee: 1) Place return result on stack (if applicable), and 2) Restore saved SP; Caller: 1) Utilise return result (if applicable), and 2) Continue execution in caller following return PC.

System Calls

System Call (aka. **Syscall**) ~ API to OS, and it changes from User Mode to Kernel Mode (and changes back to User Mode after the syscall has completed). There are 3 ways of using Syscalls.

1. Use the library version of the syscall. The library version serves as a **function wrapper** and typically has the same name. Applicable to most syscalls in modern languages.
2. Use the user-friendly library version of the syscall. This version has related handles and serves as a **function adapter**. Few syscalls have such.
3. Use the language's syscall method directly. E.g. in C, `long syscall(long number, ...)`.

Exceptions, Traps & Interrupts

Exception - Executing a machine level instruction can cause exceptions, e.g. arithmetic errors or memory accessing errors, and exception handlers are executed. Exceptions are **Synchronous**.

Trap - Exceptions intentionally set-up. Trap is an easy-to-use mechanism to enter kernel mode. Aka. "software interrupts".

Interrupt - External events can cause interrupts during the execution of a program, as a form of notification, and interrupt handlers are executed. Interrupts are usually hardware-related and **Asynchronous**.

Process Scheduling

Scheduling Overview

Scheduling requires **Scheduler**, the CPU component for process scheduling and **Scheduling Algorithm**, that determines which process gets scheduled first.
Depending on whether a process spends majority of time on CPU computation or I/O activity, it can be categorised as **CPU Bound** or **I/O Bound** activity.

Types of Environment

- **Batch Processing** - No user interaction or need to be responsive. Typically use non-preemptive scheduler since easier to implement.
- **Interactive** - Involve user interaction and need to be responsive. Typically preemptive.
- **Real-time** - All tasks have strict deadlines to meet and tasks are usually periodic.

Types of Scheduling Policies

- **Non-preemptive** - aka cooperative. A process stays scheduled until it blocks or gives up CPU voluntarily.
- **Preemptive** - CPU can be taken from running process at any time.

Scheduling Criteria

All scheduling policies consider 1a) Fairness, whether it is biases against / towards any process, 1b) whether all processes can run without **Starvation**, 2) Balanced utilisation of system resources, 3) **Throughout**, number of tasks finished per unit time, 4) **Turnaround Time**, total clock time taken (which is related to total/average waiting time), and 5) **CPU Utilisation**, % of time when CPU is working.
In addition, Interactive System scheduling policies consider: 1) **Response Time**, time between request and response by system, and 2) **Predictability**, as it aims for less variance in response time.

Batch Processing Scheduling Policies

First-Come First-Serve(FCFS)

Maintain a Queue and add to Queue when new processes come. Run a task until it finishes or blocks, then choose next in Queue to run.
Analysis: Non-preemptive. No starvation. Sub-optimal avg waiting time.

Shortest Job First (SJF)

Maintain a Priority Queue of tasks to be scheduled, sorted in ascending order by the **estimated** CPU time needed for each task.
Analysis: Non-preemptive. Starvation possible. CPU time can be estimated from the processes' past running time.

Shortest Remaining Time (SRT)

Preemptive version of SJF, where a new task can preempt currently running task if the estimated time for the former is less than remaining time needed for the latter.
Analysis: Preemptive. Starvation possible. Provide good service for short jobs.

Interactive System Scheduling

Overview

Time interrupt is an interrupt that goes off periodically based on hardware clock and its handler invokes the scheduler to preempt a task. **Interval of Time Interrupt** (ITI) is the timer period, i.e. smallest unit time in hardware clock for the timer. **Time Quantum** is the execution duration given to a process, which can be constant or variable. It must be multiples of ITI.

Round Robin (RR)

Preemptive version of FCFS, where it runs the current task either until it blocks or gives up CPU, or when the time slice (**quantum**) elapses. In the latter situation, put back the task to the rear of Queue.
Analysis: Preemptive, with a timer interrupt required. Response time guarantee, no starvation. Choice of quantum length affects performance.

Priority Scheduling

Maintain a Priority Queue and assign a priority value to every task to be scheduled. Choose tasks with highest priority every time. It can be preemptive or non-preemptive depending on whether higher priority tasks can preempt currently running tasks.
Analysis: Low priority tasks can starvation and preemptiveness does not help.

Multi-Level Feedback Queue(MLFQ)

Maintain multiple FIFO Queues of different priorities. Always pick from Queues of higher priority first, and run RR within each Queue. When a new task comes, put it in the highest priority Queue. Every time the tasks fully utilises its time slice, reduce its priority, otherwise retain its priority.
Analysis: Adaptive, seeking to minimise both response and turnaround time. Starvation possible. High priority Queues tend to have shorter time slices. Yet, possible to exploit the policy.

Lottery Scheduling

Scheduling is done rounds, where in each round, tasks are assigned "lottery tickets", the number of which depends on tasks' priority / estimate time. When picking a process to schedule, pick a ticket randomly from the pool then remove the task's tickets from the pool and pick from remaining tasks next time, until all tasks scheduled once for this round. New processes are not allowed to participate halfway during a round.
Analysis: Response time guarantee and no starvation. Good level of control. Simple.

Inter-Process Communication

Common Mechanisms

Shared Memory

Create a shared memory region once, and attach processes to the shared memory. All processes attached can read and write in shared memory just like to normal variables, achieving **implicit communication**. When done, detach all processes from the region and destroy it.
Pros: 1) Efficient, as no OS involvement during read and write to shared region, and 2) Ease of Use, as the implementation is simple and can be used for data of any type and size.
Cons: 1) Limited to a single machine, and 2) Synchronisation required as race conditions for shared variables can happen.

Message Passing

Explicit communication through exchange of messages. Process 1 prepares a message and send it to OS and Process 2 can receive it later. Both sending and receiving are syscalls.
Pros: 1) Safe, as guarded by OS, 2) Applicable beyond a single machines, 3) No need to synchronise.
Cons: 1) Inefficient, as all steps involve OS, 2) Hard to use, as data need to be packed/unpacked to supported message format.

There are different variants of Message Passing, depending on its Naming Scheme and Synchronisation Behaviours.
Naming Scheme ~ How the sender/receiver identify one another

- **Direct Communication** - Send/Receiver explicitly names the other party. A form of 1-to-1 linkage. Pros: Simple. Cons: Impractical, especially since it requires the Receiver to know who sends it a message.
- **Indirect Communication** - Messages are sent to / read from message storage (aka. **Mailbox**), which can be shared among multiple processes. Pros: Ability to have multiple senders and receivers.

Synchronisation Behaviour ~ whether sending/receiving message is blocked until it is received/has arrived. For this module, assume **all receiving is blocked**.

- **Asynchronous MP** - Non-blocking Send. Pros: Better flow for sender. Cons: 1) Too much freedom for programmer, and 2) Not true asynchronous as **buffer size is finite**, so sending will block when buffer full.
- **Synchronous MP** - Blocking Send, aka Rendezvous.

Synchronous MP can also be used as a form of synchronisation of two processes by using dummy message.

Unix-Specific Mechanisms

Pipes

In Unix, a process has 3 default communication channels, 1) `stdin` for std input, 2) `stdout` for std output, and 3) `stderr` for std error output.

Unix pipes behave like anonymous files, FIFO for data accessing order. Writer wait when pipe buffer is full, and reader waits when buffer empty. This enables Pipe to have **asynchronous** and **implicit** synchronisation.
In addition there are **half-duplex** (unidirectional) and **full-duplex** (bidirectional) pipes.

Signals

Signals are **asynchronous** (meaning can be sent anytime) notifications regarding an event, sent to a process/thread. The recipient must handle the signal by 1) a default set of handlers, or 2) user supplied handlers. Note: Not all signal handlers can be override by user.
Signals are sent from OS to user programs or from one user program to another. Signal handlers run in User Mode.
With user supplied signal handlers, signals can achieve IPC.

Threading

Threading Overview

One process can have multiple threads, thereby achieving **Data Parallelism** (different threads same task different data) and **Task Parallelism** (different threads different tasks).
Pros: 1) Economy, as multi-threading easier than multi-processing, 2) Resource sharing, 3) Responsiveness, of multi-threaded programs, and 4) Scalability, as multi-threaded programs can utilise multiple cores / CPUs.
Cons: 1) Synchronisation getting more complicated, 2) Difficulty with **System Call Concurrency**, and 3) Concerns regarding process behaviours (e.g. fork a multi-threaded process).

Certain information are shared across threads of the same process and some not shared.
Shared: 1) Text (code), 2) Data, 3) Heap, 4) File, and 5) Process ID.
Not Shared: 1) Thread Id, 2) Stack, 3) Registers (GPR & Special), 4) PC

Thread Implementation

User Thread

User Thread ~ Implemented as a user library. Not "recognised" or scheduled by the OS.
Pros: 1) Multi-threading can be done on any systems, 2) Context switch of threads easier, and 3) more configurable and flexible.
Cons: Sub-optimal performance with OS scheduling only processes.

Kernel Thread

Kernel Thread ~ Implemented in OS, with thread operations handled as syscalls. Kernel schedules threads not processes.
Pros: Kernel can schedule threads.
Cons: 1) Slower and more resource-intensive with thread operations as syscalls, and 2) Generally less flexible.

Hybrid Thread

Hybrid Thread ~ OS schedules only kernel threads while user threads can bind to kernel threads. More flexible.

Synchronisation

Critical Section

Race Condition - Interleaving of accesses to a shared modifiable resource. Race condition may lead to incorrect program behaviour so it is handled using **Critical Section** (CS), which is designated code segment with race conditions. At any point of time, at most one process can be in the critical section.

Critical Section has 4 properties:

1. **Mutual Exclusion** - If Process P_1 is in CS, all other processes are prevented from entering CS.
2. **Progress** - If no process is in a CS, one of the waiting processes should be granted access.
3. **Bounded Wait** - After Process P_1 requests to enter CS, there exists an upper bound on number of times other processes can enter CS before P_1.
4. **Independence** - Processes not executing in CS should never block (those waiting to enter CS).

Note: #2 Progress requires #4 Independence first, but it is possible to have #4 Independence without #2 Progress.
Note: #2 Progress ensures no **Deadlock**, a situation where all processes are blocked, and no **Livelock**, a situation where all tasks are running but they do not "make any progress".

HLL Implementation

High-level Language (HLL) Implementations for Synchronisation are those using only normal programming constructs. (Use your brain power LOL)

Failed Attempts

Attempt 1 - Lock Variable: Use a shared boolean, `lock`, to indicate whether any process in CS currently. Busy-wait if `lock == 1`. Enter and update it if `lock == 0`.
Analysis: **NOT WORK** as there can be race condition on `lock` as well, violating # 1 Mutual Exclusion.

Attempt 2 - Disable Interrupt: As a fix to Attempt 1, disable interrupt (and hence no context switching) before a process waits or enters CS, and re-enable interrupts after it leaves CS.
Analysis: **NOT RECOMMENDED** as it only works on single-core systems and there can be race conditions with processes running on other cores in multi-core systems. In addition, disabling interrupts requires permission and is dangerous.

Attempt 3 - Turn Variable: Use a shared variable, `turn`, to indicate which process' turn it is currently. A process checks turn variable and waits until it is its own turn to enter CS.
Analysis: **NOT WORK** as it may violate #4 Independence (`turn = 0` and P_1 busy-waits but P_0 never enters CS).

Attempt 4 - Want Variable: Use a shared boolean array, `want[]`, where each element `want[i]` indicates if Process P_i wants to enter. (In case of 2 processes,) enter if the other process does not want to enter, otherwise wait.
Analysis: **NOT WORK** as it can have a deadlock when `want[0] == 1` and `want[1] == 1`, thereby violating #2 Progress.

Synchronisation Algorithms

Peterson's Algorithm - Combine Attempt 3 and 4. (In case of 2 processes,) wait if the other is waiting and it is the other's turn. If it is my turn, I will enter regardless.
Analysis: **WORK**, but 1) difficult to generalise for ≥ 3 processes, and 2) Busy waiting is used and it is not desired.

Assembly Level Implementation

Test-Set-Lock (TSL) ~ Similar to HLL Attempt 1 Using Lock Variable, but TSL does not have race condition because it is an **Atomic**, i.e. reading and writiing of the variable can only be done by one process at any time, operation ensured by the hardware.
Cons: Busy waiting.

HLL Abstraction

Semaphore

Semaphore - A generalised synchronisation mechanism with functional behaviour of wait and signal.

Implementation

- An integer `S` ~ number of "net wake-ups". Can be initialised to any non-negative integer.
- `wait(S)` - decrement `S`, and if $S \leq 0$, (process calling the semaphore) blocks. Aka. `P()` or `down()`.
- `signal(S)` - increment `S`, and wait up one sleeping process if any. Aka. `V()` or `up()`.

Note: Assume no negative value for semaphore in this module.

Functionality

A **Binary Semaphore** that **initialises to 0** and **ensures only value 0 or 1** is a **Mutex**, i.e. Mutual Exclusion, for preventing race conditions.
Note: Deadlock can be avoided but is possible with improper implementation of two or more mutexes.

A **General Semaphore** serves as a General Synchronisation Tool (not CS in this case) for multiple processes (e.g. ensuring Processes P_1, P_2, P_3 operate in a fixed order. Note: Semaphore do not use busy waiting.

Others

Conditional Variable - An implementation that allows a task to wait for certain event to happen. It can **broadcast**, which wakes up all waiting tasks.

Intermediate Results & Summary

From Tutorials

(Tut1 Qn7) In `int *a = (int *) malloc(sizeof(int));`, the data, `*a`, is created in the Heap while the pointer to this data, `a`, can be in Stack or Data.

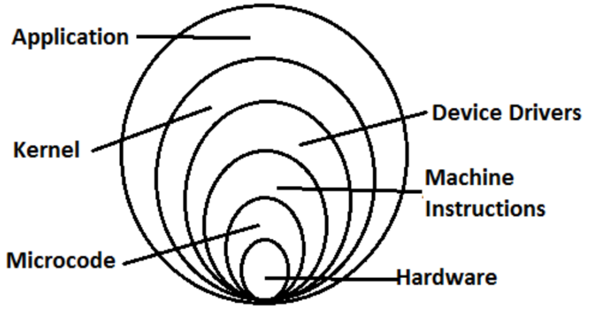
From Past Questions

(AY15/16Sem1 Qn1) Library calls are language dependent, while syscalls depend only on the OS.

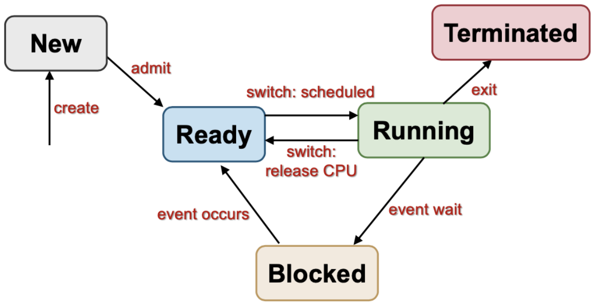
(AY17/18Sem1 Qn8) In scheduling, **Short Response Time** is not the same as **Bounded Waiting Time**. RR ensures latter but not former.

Diagrams

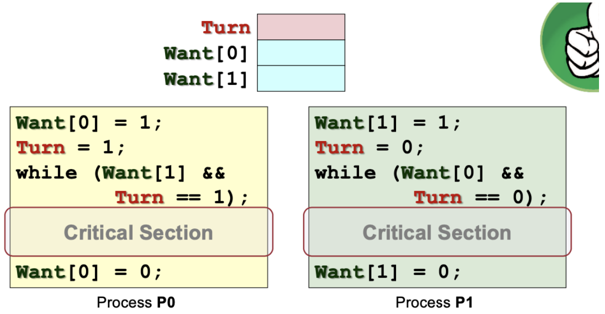
Onion Model of an Operating System:



Generic 5 States of a Process:



Pseudocode for Peterson's Algorithm (note the order of updating variables before busy-wait):



```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
        Turn == 1);
```

```
Want[0] = 0;
```

Process P0

```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
        Turn == 0);
```

```
Want[1] = 0;
```

Process P1

Unix System Calls

Common Process Management Syscalls

`fork()` *Syscall*

`int fork()` creates a duplicate of current process as a child process. It clones everything except PID and PPID. It returns the PID of child process forked in parent process and `0` in child.
Need header `<unistd.h>`.

`exec*()` *Family*

The family of syscalls, `int exec*(...)`, replaces current executing process with a new one. It takes in path to the executable and arguments. It will not return when executing successfully and return `-1` when fails.
There are many variants, such as `execv`, `execl`, `execle`, `execlv`, or `execv`, each with a slight difference in parameter specifications and behaviours.
The typical usage is to create a new child process using `fork()` then run the program using `exec()` in the child.
Need header `<unistd.h>`.

`exit()` *Syscall*

`void exit(int status)` terminates the current process and the value of `status` will return to the parent. The convention is `0` for normal termination and non-zero for error.
`exit()` is the friendly wrapper of `_exit()` as the former does some necessary clean-up before invoking the syscall.
When a child calls `exit()`, it sends `SIGCHLD` signal to the parent.
Need header `<unistd.h>`

`wait()` *Family*

`int wait(int *status)` makes the process block and wait for any one child process to terminate, and the exit status value is stored in the parent's variable referrenced by `*status`. It cleans up remainder of terminated child's system resources.
There are other variants such as `waitpid()` that waits for a specific child.
Need header `<sys/types.h>` and `<sys/wait.h>`.

`getpid()` *Family*

`int getpid()` returns the PID of the current process.
There are similar syscalls such as `getppid()`.
Need header `<unistd.h>`.

IPC-Related Syscalls

`pipe()` *Syscall*

`int pipe(int fd[2])` creates a unidirectional pipe channel with file descriptors `fd[0]` for reading end and `fd[1]` for writing end. It returns `0` for successful creation and non-zero values for error.
Need header `<unistd.h>`.

`dup2()` *Syscall*

`int dup2(int oldfd, int newfd)` replaces the file descriptor of `newfd` with the one specified by `oldfd`.
The syscall is typically used to redirect standard channels of a process to a pipe or file.
Need header `<unistd.h>`.

Shared Memory Syscalls

`shmget()`, `shmat()`, `shmdt()`, and `shmctl()` are syscalls to 1) create a shared memory region, 2) attach the shared memory to the current process, 3) detach the shared memory from the current process, and 4) destroy the shared memory region respective.
Note: `shmget()` returns the ID of the shared memory created, and the ID is the identifier for the shared memory. A memory address is generated only when you attach a process to the shared memory using `shmat()`.
Need header `<sys/shm.h>`.

Semaphore-Related Syscalls

`sem_t` is the type for Semaphore. `sem_init()`, `sem_destroy()`, `sem_wait()`, and `sem_post()` are syscalls to 1) initialise a semaphore, 2) destroy the semaphore, 3) perform wait on the semaphore, and 4) perform signal on the semaphore respectively.
Need header `<semaphore>` and compile with option `-lpthread`.