CS2106 Cheatsheet
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 writing 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 1** 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.

Classical Synchronisation Problems

Producer-Consumer Problem

Specification: Processes share a bounded buffer of size K. Producers produce items to insert in buffer when buffer not full. Consumers remove items from buffer when buffer is not empty.

Solution 1 - Busy Waiting: Use two shared booleans `canProduce` and `canConsume` for indication. Busy wait on the two booleans. Use another semaphore mutex to ensure no race condition where one of production / consumption can take place at a single time (as both will affect count). Analysis: Solve the problem but uses Busy Waiting, which is not desired.

Solution 2 - Blocking: Use semaphores `notFull` and `notEmpty` to force producers and consumers that are “inappropriate” at the current time to go to sleep, and to wake them up when “appropriate”.
Analysis: Solve the problem efficiently.

Solution 3 - Message passing: Use built-in message passing with desired mechanisms - block on read when empty, block on write when full.

Reader-Writer Problem

Specification: Processes share a data structure D where Readers can retrieve information from D and Writers can modify information in D. Write must have exclusive access to D, while Reader can access with other Readers. i.e. Must write along but can read together.

Solution: Refer to the diagram for solution. Intuitively, it means when there is an existing writer, the first reader waits at the semaphore while subsequent ones wait at the mutex. Analysis: Works, but starvation may arise (e.g. when one write / many readers “hogging” the room).

Dining Philosopher Problem

Specification: (This is a toy problem.) 5 Philosophers seat around a table with 5 single chopstick placed between every pair of philosophers. When any philosopher wants to eat, he/she will take both chopsticks from left and right. Devise a deadlock-free and starve-free algorithm for philosophers to dine.

Solution 1 - Limited Eater ~”Social Distancing”, allow at most 4 to try to eat.

Solution 2 - Tanenbaum Solution ~“Turn & Want” for Mutex Implementation.

Memory Abstraction

Contiguous Memory Allocation

Contiguous Memory Management ~Process must be in memory, as one piece, during whole execution. So each process owns one contiguous memory region.

Memory Partition

Fixed-Size Partition - Physical memory is split into fixed number of partitions of equal size. A process will occupy one. This partition scheme has **Internal Fragmentation** - If a process does not occupy the whole partition, the leftover space is wasted.
Pros: 1) Easy to manage, 2) Fast to allocate (identical free partitions, no need choosing).
Cons: Partition size need to be large enough to contain largest possible process but larger partition size worse internal fragmentation.

Variable-Size Partition - Partition is created based on the actual size of process. OS keep track of the occupied and free memory regions. This partition scheme has **External Fragmentation** ~free memory space (”holes”) created with process operations.
Pros: Flexible and ensure no internal fragmentation.

Cons: 1) Need to maintain information about each partition, 2) Extra time to locate appropriate free region for allocation.
Note: External fragmentation can be handled via **Merging**, merging adjacent free holes, and **Compaction**, moving occupied partitions around to create bigger consolidated holes, but they should not be done frequently.

Dynamic Partitioning Algorithms

Allocation Methods

There are 3 allocations methods: 1) **First-Fit** finding first large-enough hole, 2) **Best-Fit** finding smallest large-enough hole, and 3) **Worst-Fit** finding largest hole.

Allocation Implementation

Linked-List Method - OS maintains a linked list for partition information, where each nodes represents one partition, storing info. about free/occupied, starting address and length, and pointer to next element.

Multiple-Free-List Method - OS keeps separate sets of lists, one set of lists for free holes and the other for occupied partitions. In a set, there are multiple lists of different hole sizes (powers of two).
Analysis: Fast allocation (by getting holes of desired size)

Buddy System - (A more efficient version of Multiple-Free-List) Free block is split into half repeatedly to meet request size. The two halves form a pair of sibling blocks ("buddies"). When both buddy blocks are free, they are merged to former a larger block and checked with its own buddy.
Note: [1]Two adjacent free blocks of same size cannot be merged if they are not buddy. [2] When there is no occupied partition, the physical memory is one single free block.

Disjoint Memory Allocation

Disjoint Memory Allocation ~Processes can have disjoint chunks of memory space allocated.

Paging

Paging ~The physical memory is split into regions of fixed size, known as **Physical Frames** (with size determined by hardware). Logical memory **of a process** is also split into regions of the same size, known as **Logical Pages**. Note: Logical pages are continuous while physical frames are disjoint.
Page Table, a lookup take for translation between pages and frames **per process**, is used for bookkeeping. The physical address can be calculated:

$$\text{Physical Addr} = \text{Frame ID} \times \text{size(Frame)} + \text{Offset}$$

, where Frame ID is obtained from Page Table for each page and Offset is displacement from beginning of page.
Analysis: 1) Paging has no external fragmentation. 2) Paging has internal fragmentation but always less than 1 page per process.

Implementation

Paging scheme needs to keep track of page tables of processes efficiently. Below are two implementations.

Software Solution - OS stores Page Table in PCB, and considers it part of memory context of a process.
Analysis: Slow, as each memory ref now requires 2 memory accesses (1 to read page table entry for Frame ID, 1 to access actual memory)

Hardware Solution - OS has a special component, **Translation Look-Aside Buffer** (TLB), to support paging. It is a cache for page table entries, one TLB per core. `TLB-Hit` and `TLB-Miss` refer to entry found / not found in TLB. Only when not found in TLB, retrieve from Page Table in memory and update TLB.

Application

Protection - Paging can be easily extended to provide memory protection for each entry in a Page Table, using **Access-Right Bits** (`wrx` in Linux) and **Valid Bits**.

Sharing - Page table can allow several processes to share the same physical memory frame by using same frame number. Example: share code (text) pages; implement Copy-On-Write between parent & child.

Segmentation

Segmentation ~Memory is managed at the level of memory segments (Text, Data, Heap, Stack), each of which is mapped (disjointly) to a contiguous physical partition.
Each memory segment has 1) a name (or a `SegID`) and 2) a `Limit`, indicating the exact range of the segment. A segment table stores segment entries for each process. Each logical address is represented `<SegID, Offset>` and need to make sure Offset < Limit all the time.
Pros: Each segment is an independent contiguous memory space, so 1) Follow programmer's view of memory, easy to understand, 2) More Efficient bookkeeping, and 3) Segments can grow / shrink and be protected / shared independently.
Cons: Segmentation uses variable-size contiguous memory regions, so prone to external fragmentation.

Segmentation with Paging

Segmentation with Paging ~Each segment is now composed of several pages instead of a contiguous memory region. Each segment has a page table.
Analysis: 1) Perform no worse than Segmentation or Paging separately. 2) Able to handle different functionalities, as some metadata better managed at segment level and some page level.

Virtual Memory Allocation

Virtual Memory ~A continuation of the idea of Paging Scheme, while some pages are in physical memory, and more "pages" in secondary storage. A `resident bit` in page table entry indicates whether a page is in physical memory.
CPU can only access memory resident pages. A **Page Fault** occurs when CPU tries to access a non-memory resident page. **Thrashing** is when memory access results in page fault most of the time.

Remedies / Attempted Solution for Thrashing:

- **Locality Principle**, both Temporal and Spatial
- Skewed Distribution of Object / Access Popularity
- **Demand Paging**, where process starts with no memory-resident pages, and only allocate a page when there is a page fault.

Analysis of Demand Paging: Pros: 1) Fast startup time for new process, and 2) Small "memory footprint". Cons: 1) Process appears slow at the start due to multiple page faults, and 2) Page faults may have cascading effects on other processes.

Page Table Structures

Direct Paging - Keep all entries in a single (page) table. The implementation used above. Requires the entire structure to be contiguous and fits in one page.
Analysis: It requires huge page table size for large (secondary) memory. Unrealistic.

2-Level Paging ~Page the page table into smaller page tables, each with a "page table number". The first level "page table" is known as the **Page Directory**, and it is stored in the Memory Context of each process in PCB.
Pros: 1) Bypass the requirement for entire structure to be contiguous in memory. 2) No need to allocate memory for empty blocks.
Cons: 1) Require 2 memory access to get the frame number. (Solution: Use **MMU Caches**, "TLB for directory entries" to speed up page-table walk)

Hierarchical Page Table ~Radix-tree structure that extends the idea of 2-Level Page Table to n-level. All tables are setup and wired by the OS in software, but traversed by hardware. Used in most modern computers.

Inverted Page Table - Keep a single mapping of physical frame to `<pid, page#>` that uniquely identifies a memory page. Note: It stores pages belonging to all processes.
Pros: 1) Huge saving, one table for all processes. 2) Fast translation for answering "which process and page does this frame belong to?", an auxiliary structure. Cons: Slow translation from a process' page to physical frame.

Page Replacement Algorithms

Page Replacement

Eviction is the freeing of a memory page to store information for a new memory page (e.g. bring in a non-memory resident page). When a **Clean Page**, page not modified, is evicted, no need to write back. When a **Dirty Page**, page modified, is evicted, need to write back.
We want to find optimal page replacement algorithms so as to minimise page faults and access / updating of secondary memory.

Algorithms

Optimal Page Replacement (OPT) - The theoretical optimal algorithm, not achievable but serves as a benchmark for comparison: If we have future knowledge of memory reference, replace the page that will not be needed again for the longest upcoming period, and this guarantees minimised page faults.

FIFO Page Replacement - Evict memory pages based on their loading time, i.e. evict the oldest memory page.
Implementation: OS maintains a queue for resident page numbers. Remove 1st in queue if replacement needed. Update queue during page fault trap.
Pros: Simple to implement, no need hardware.
Cons: 1) FIFO does not exploit temporal locality (, oldest memory $\neq$ not used recently). 2) Suffer from **Belady's Anomaly**, where the number of physical frames increases, number of page faults should decrease intuitively, but in reality, more page faults occur for FIFO.

Least Recently Used (LRU) - Use the notion of temporal locality, and replace page not used in longest time. Aim to approximate OPT (predict future by mirroring the past).
Pros: Does not suffer from Belady's Anomaly.
Cons: Hard to implement and require additional hardware support.

Second-Chance Page Replacement (CLOCK) - Modified FIFO to give a second chance to pages that are accessed.
Implementation: Use a `Reference Bit` for each entry where `1` for accessed since last reset and `0` otherwise. For replacement, pop if ref bit `0`, or reset the bit if `1`. When access / bring in a page, update ref bit to `1`. Use a Circular List.

Frame Allocation

Local Replacement - Victim pages are selected among pages of the process with page fault. Pros: Number of frames allocated to a process unchanged, so performance stable between multiple runs. Cons: If frames allocated not enough, the process' performance affected.

Global Replacement - Victim pages can be chosen among all physical frames during a page fault. Pros: Allow self-adjustment between processes. Cons: 1) Badly behaved processes can affect others. 2) Number of frames allocated to a process different, so performance vary between runs.

Working Set Model helps determine number of frames allocated to each process. **Working Set**, $W(t, \Delta)$ is the set of active pages in the time interval $(t - \Delta, t)$. It gives info. about frames needed.
Analysis: Accuracy of the model is affected by choice of Δ: If too small, then may miss pages in current working set; If too big: then may contain pages in other working set.

File System Abstraction

File System Introduction

Criteria for a File System: 1) **Self-Contained**, able to "plug-and-play" on another system, 2) **Persistent**, beyond the lifetime of OS and processes, and 3) **Efficient**, which provides good management of free and used space, with minimal overhead.

File

A **File** is a logic unit of info created by process. It contains **Data** and **Metadata**.
Different FS have different naming rules, but they usually follow `Name.Extension` format.

File Type

An OS commonly supports a number of file types, each with an associated set of operations and/or a specific program for processing it. Common file types:

- **Regular Files** - Files containing user info, and can be further categorised as **ASCII Files**, can be displayed as it is, and **Binary Files**, with a predefined internal structure to be processed by specific program.
- **Directories** - System files for FS structure.
- **Special Files** ~character / block oriented.

To distinguish file types: 1) Use file extension as indication, or 2) Use embedded information in the file (e.g. `Magic Number` at the beginning of a file in Unix)

File Protection

To control access to info stored in a file, restrict access based on user identity.

- **Access Control List** - Store a list of user identities and allowed access type. Pros: Very customisable. Cons: Additional info associated with file needed.
- **Permission Bits** - Classify user into 3 classes (Owner, Group, Universe) and give `wrx` bit for each class.

File Data Structure & Access Methods

Structures how a file stores data

- Array of Bytes: Each byte has a unique offset from file start. Hard to jump to any specific "entries".
- Fixed-Length Records: Can jump to any records.
- Variable-Length Records: Flexible but harder to locate a record.

Access Methods:

- Sequential Access: Read data in order from file start. Cannot skip but can rewound.
- Ranom Access: Read data in any order. `Read(Offset)` states the position to start access explicitly, or `Seek()` to move to a new location in file. Both Unix and Windows FS use latter.
- Direct Access: i.e. random access to any records directly, used in files of fixed-length records.

Directories

Directories provide a logical grouping of files (User view) and keep tracks of files (system usage).

Links in Linux

Hard Link - Both files have separate pointers to the actual file in disk. Limited to only files.
Pros: Low overhead (only pointers added).
Cons: Deletion problem.

Symbolic Link - For B being a symbol link of A pointing to the actual file F, B is a file G with path name of F, so accessing G means to find where F is and access it. It acts as an "intermediate jump". Applicable to both files and directories.
Pros: Simple deletion: If B deletes, G gets deleted but F unaffected. If A deletes, F is gone and G remains but not working.
Cons: Large overhead, with special link file taking up actual disk space.

Directory Structures

Tree Structure ~Recursively embed directories in other directories, which forms a tree. To refer a file, use either absolute path-name from root directory, or relative path-name followed from current working directory.

DAG Structure ~Based on the Tree structure, but a file can be shared in different directories. Only one copy of actual content exists but it appears in multiple directories of different path-names.
In Linux, created with Hard Links.

General Graph Structure ~Hard to traverse and maintain (determine when to remove a file / directory, prevention of infinite loops). Not desirable.
In Linux, created with Symbolic Links.

OS and File System

OS provides syscalls for all file operations, and it also maintains information about FS.

Runtime Information for Files

Information kept for an opened file: 1) **File pointer**, current location in file, 2) **Disk Location**, actual file location on disk, and 3) **Open Count**, number of processes that have this file opened.
The OS also maintains a **System-wide Open-File Table**, with one entry for each unique file opened, and **Per-Process Open-File Table**, which has an entry for each file opened by the process, and the entry points to the one in the system-wide table. There is also **Buffer** for information read from / written to disk.
Implications: Two processes can open the same file using same (e.g. parent & child) or different file descriptors. Same process can also open one file twice simultaneously.

File Operations

To `Create` a file: 1) Use full pathname to locate the parent directory. 2) Use free space list to find free disk blocks. 3) Add an entry to parent directory, with relevant file info & metadata.

To `Open` a file F in process P: 1) Search system-wide table for existing entry E: If found, create an entry in P's table pointing to E; Otherwise, continue. 2) Use full pathname to locate file F: When F is located, load its information into a new entry E in system-wide table, and create an entry in P's table pointing to E. 3) Return pointer to entry.

File System Implementation

Hardware

A **Storage** is an 1D array of **Logical Blocks**, smallest accessible units. Logical Blocks are mapped into disk sectors.
Within one disk, there is a **Master Boot Record** at Sector 0, which contains boot code and partition table. It is followed by several **Partitions**, each containing an independent file system.
Within one partition, there are 1) OS Boot-Up Information Block, 2) Block for Partition details (e.g. total number of blocks, number and location of free disk blocks), 3) Directory Structure, 4) Files' info., and 5) Actual File Data.

File Implementations

A file is implemented as a collection of logical blocks. There are different implementations, with different ways of 1) keeping track of the blocks, 2) allowing efficient access, and 3) utilising disk space effectively.

Contiguous Allocation - Allocate consecutive disk blocks to a file.
Pros: 1) Simple to keep track - Each file only needs starting block number & length, 2) Fast access - only need to seek to first block.
Cons: 1) External fragmentation, 2) File size need to be specified in advance.

Linked List Allocation - Keep a linked list of disk blocks for each file. Each block stores 1) actual file data and 2) next disk block's number. File info stores first and last disk block number.
Pros: No external fragmentation.
Cons: 1) Random access in a file is very slow. 2) Overhead for "pointers". 3) Less reliable (e.g. a pointer incorrect or a block corrupted.
Variant: Move all block pointers (of all files in the disk) into a single table, known as **File Allocation Table** (FAT) and store FAT in memory during runtime. Pros: Faster random access (memory traversal now). Cons: FAT can be huge and consume valuable memory space.

Indexed Allocation - Each file has an index block, an array that keeps track of (its) disk block addresses.
Pros: 1) Lesser memory overhead (than Linked List). 2) Fast direct access to certain location/offset of a file.
Cons: 1) Max file size limited by number of block entries in the index block. 2) Index block incurs overhead.
Variant: 1) **Linked Scheme** - keeps a linked list of index blocks, can grow on demand. 2) **Multi-Level Index Scheme** - Each first level index block points to several second level index blocks, with the last level being blocks for file data, example: Unix I-Node.

Free Space Management

Bitmap Implementation - Each disk block is represented by 1 bit (`1` for free, `0` for occupied). Pros: Provide a good set of manipulations. Cons: Need to keep the bitmap in memory for efficiency reason.

Linked List Implementation - Use a linked list of disk blocks, where each disk block contains a number of free disk block numbers, and a pointer to next free space disk block. Pros: 1) Easy to locate free blocks. 2) Only first pointer is needed in memory (though others can be cached). Cons: High overhead.

Directory Implementation

A **Directory Structure** keeps track of files in the directory and map file names to file information. Sub-directories are usually stored as file entries of special type. Several implementations for a Directory exists.

- Linear List Implementation - A directory is a list, where each entry represents a file. Each entry stores 1) filename and metadata, and 2) file info / pointer to file info. Use linear search to locate a file. Analysis: Linear search can be slow, so cache can be used to explore Locality as an improvement.
- Hash Table Implementation - Each directory has a hash table `<FileName, File>`, and it uses Chained Collision Resolution. Analysis: Fast lookup, but limited size for hash table and good hashing function is required.

File Operation

File operations are syscalls provided by the OS.

Disk I/O Scheduling

Time taken for one disk I/O is the sum of 1) **Seek Time**, time for the disk head to move to a position over proper track, 2) **Rotational Latency**, time for the desired sector to rotate under the I/O head, and 3) **Transfer Time**, time to transfer data from disk sector to memory which depends on transfer size and transfer rate.
Time #3 is negligible as compared to the other two (an order of 3 to 5). **Average time** of #1 can be reduced with scheduling algorithms while Time of #2 can be reduced with larger block size (at the compromise of #3, but negligible).
Scheduling Algorithms

- FCFS - Simple but bad
- **Shortest Seek First** (SSF) ~SJF for disk context
- **SCAN / Elevator Algorithm** ~1) bi-directional **SCAN** that alternates between innermost and outermost, and 2) one-directional **C-SCAN** that move from outermost to innermost (i.e. largest to smallest Track number)

File System Case Studies

MS FAT

Linux Ext2

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.
(Tut10 Qn3a) When Process A tries to open a file written by Process B, OS uses Open File Table to check for existing opened files and reject A's request since the file is currently being written.
(Tut10 Qn3b) When Process A tries to use a bogus file descriptor for file-related syscalls, the OS checks whether the fd entry exists and is valid in A's opened file table, and reject the syscall since invalid.
(Tut10 Qn3d) Two processes A and B can read from the same file independently, as they have their own fds, which refers to two distinct locations in the open file table. Each entry of the open file table keep track of the current location separately.

From Past Questions

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

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

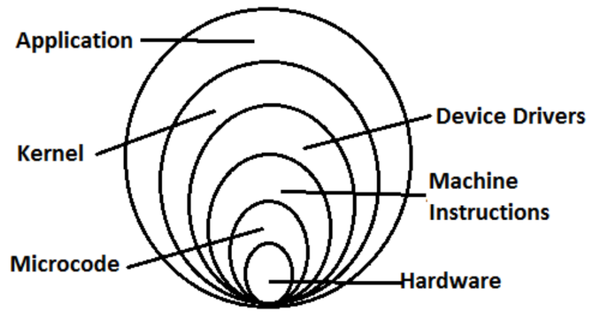
(AY18/19Sem1 Final Qn8) In Ext File System, data blocks for a file come from the same block group. One of the reasons is to reduce fragmentation.

(AY18/19Sem1 Final Qn19) In FAT File System, each Directory Entry is also stored in a logical block. So for file operations that modifies directory entry for the file, the logical block where the directory is in is also modified.

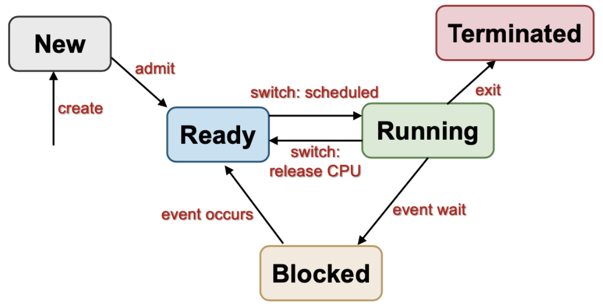
(AY18/19Sem1 Final Qn25) In Ext File System, **Disk Access Number** is the sum of accessing INodes and Contents starting from `"/"`'s INode till the desired block in the file's content. (i.e. 2 accesses needed going down each directory level)

Diagrams

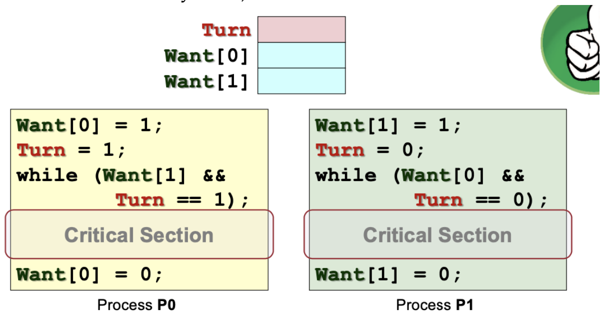
Onion Model of an Operating System:



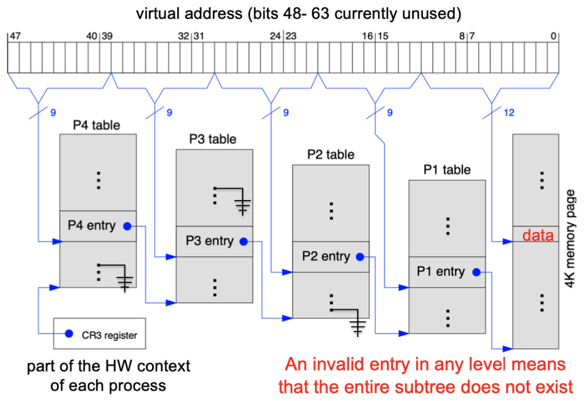
Generic 5 States of a Process:

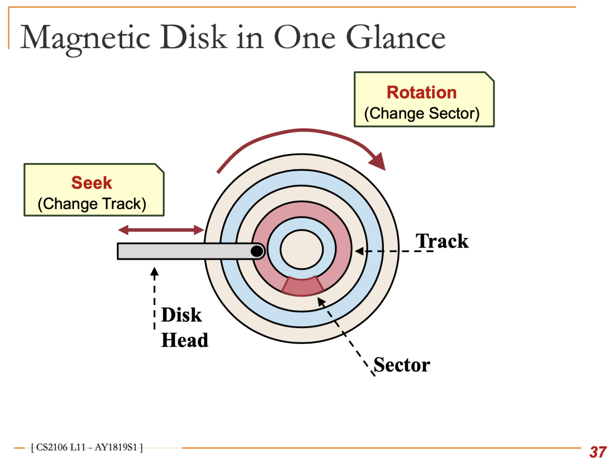


Pseudocode for Peterson's Algorithm (note the order of updating variables before busy-wait):

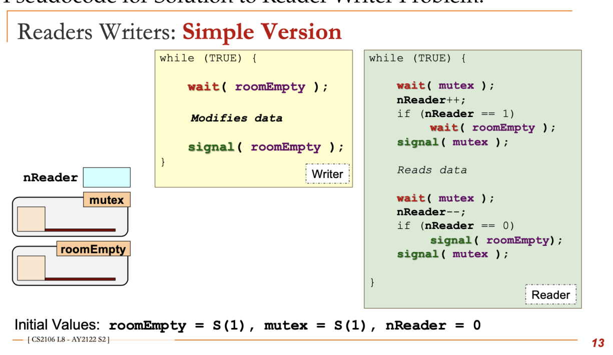

```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
        Turn == 1);
   Critical Section
Want[0] = 0;
```
Process P0

```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
        Turn == 0);
   Critical Section
Want[1] = 0;
```
Process P1


virtual address (bits 48- 63 currently unused)
47
40 39
32 31
24 23
16 15
8 7
0
9
9
9
9
12
P4 table
P3 table
P2 table
P1 table
P4 entry
P3 entry
P2 entry
P1 entry
data
4K memory page
CR3 register
part of the HW context of each process
An invalid entry in any level means that the entire subtree does not exist


Magnetic Disk in One Glance

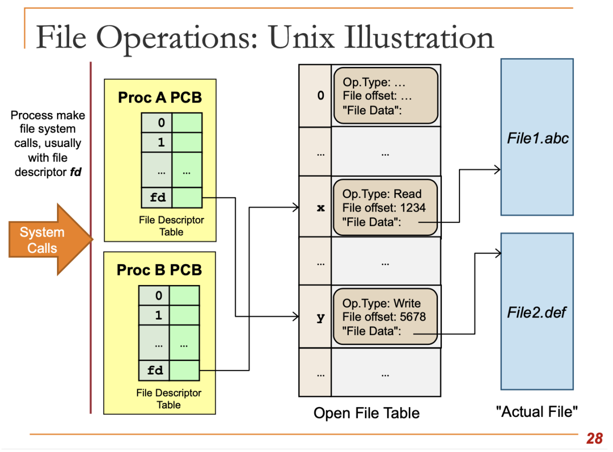
Pseudocode for Solution to Reader Writer Problem:


Readers Writers: Simple Version
```
while (TRUE) {

    wait( roomEmpty );

     Modifies data

    signal( roomEmpty );
}
```
Writer

```
while (TRUE) {

    wait( mutex );
    nReader++;
    if (nReader == 1)
        wait( roomEmpty );
    signal( mutex );

    Reads data

    wait( mutex );
    nReader--;
    if (nReader == 0)
        signal( roomEmpty);
    signal( mutex );

}
```
Reader

Initial Values: `roomEmpty = S(1), mutex = S(1), nReader = 0`

Illustration for File Opening:


File Operations: Unix Illustration

(Tut09 Qns3) Algorithms for Virtual Memory Operations

Algorithm for accessing virtual address VA:

1. [HW] VA is decomposed into <Page#, Offset>
2. [HW] Search TLB for <Page#>:
    a. If TLB miss: Trap to OS { *TLB Fault* }
3. [HW] Is <Page#> memory resident?
    a. Non-memory resident: Trap to OS { *Page Fault* }
4. [HW] Use <Frame#><Offset> to access physical memory.

Algorithm for TLB Fault Handler, given <Page #>:

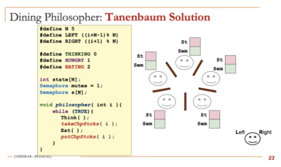
1. **[OS] Access full table of the process (e.g. in the PCB of the process), <Page#> is the index**
2. **[OS] Check whether valid bit is set, if not segmentation fault.**

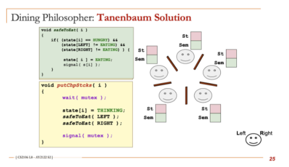
3. **[OS] Load the relevant PTE into TLB.**
4. **[OS] Return from trap.**

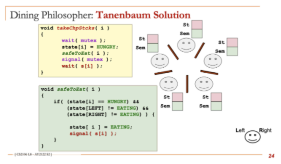
Algorithm for Page Fault Handler, given <Page #>:

1. **[OS] Global/Local replacement, Replacement algorithm applicable here**
2. **[OS] Write out the page to be replaced if needed.**
3. **[OS] Locate the page in secondary swap pages.**
4. **[OS] Load the page into a physical frame.**
5. **[OS] Update page table entries.**
6. **[OS] Update TLB**
7. **[OS] Return from Trap**

Pseudocode for Tanenbaum Algorithm:


Dining Philosopher: Tanenbaum Solution
```
#define N 5
#define LEFT ((i+N-1)% N)
#define RIGHT ((i+1) % N)

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
       Think( );
       takeChpStcks( i );
       Eat( );
       putChpStcks( i );
    }
}
```


Dining Philosopher: Tanenbaum Solution
```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

        state[ i ] = EATING;
        signal( s[i] );
    }
}
```
```
void putChpStcks( i )
{
        wait( mutex );

        state[i] = THINKING;
        safeToEat( LEFT );
        safeToEat( RIGHT );

        signal( mutex );
}
```


Dining Philosopher: Tanenbaum Solution
```
void takeChpStcks( i )
{
        wait( mutex );
        state[i] = HUNGRY;
        safeToEat( i );
        signal( mutex );
        wait( s[i] );
}
```
```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

         state[ i ] = EATING;
         signal( s[i] );
    }
}
```

Idea: Similar to "Turn & Want" for Mutex Implementation. Complicated.

Common File Operations:

Create:	New file is created with no data
Open:	Performed before further operations To prepare the necessary information for file operations later
Read:	Read data from file, usually starting from current position
Write:	Write data to file, usually starting from current position
Repositioning:	Also known as seek Move the current position to a new location No actual Read/Write is performed
Truncate:	Removes data between specified position to end of file

Illustration for Hiearchical Level Paging:

Illustration for One Slice of Magnetic Disk (note that multiple slices of disks stack over one another to form the hard disk):

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`.