CS2106 Cheatsheet

for finals, by Hanming

Processes

Process is a abstraction for a running program. Identified by a **PID**.

Process State

5-Stage Process Model

1. New
2. Ready
3. Running
4. Blocked
5. Terminated

Transitions

1. Create: NIL → New
2. Admit: New → Ready
3. Switch: Ready → Running
4. Switch: Running → Ready
5. Event Wait: Running → Blocked
6. Event Occurs: Blocked → Ready

Process Table

Each process has a Process Control Block (**PCB**) that stores the entire execution context for a process. These blocks are stored in a **Process Table** by the kernel.

OS Interaction with Process

A C/C++ program can call the library version of system calls. A **function wrapper** has same name and parameters as the syscall. A **function adapter** is user-friendlier.

Syscall Mechanism

1. Program invokes library call
2. Library call places syscall number in register
3. Library call executes TRAP to switch to kernel mode
4. Syscall handler is determined using number by dispatcher
5. Syscall handler is executed and completes
6. OS switches to user mode and returns control to library call
7. Library call returns to user program using function return

Exceptions are synchronous and causes an exception handler to execute. **Interrupts** are asynchronous, suspends program and executes an interrupt handler.

Handler Routine

1. Save register/CPU state
2. Perform handler routine
3. Restore register/CPU state
4. Return from interrupt

UNIX Context

A UNIX process abstraction has information on PID, state, parent PID, cumulative CPU time, etc.

`init`
Root of all processes in UNIX. Created in kernel during boot-up, has PID 1.

`fork()`
Creates a child process with copy of parent's executable image. Data is not shared. Function returns `0` for child and `> 0` for parent.

`exec()`
Replaces current executing process image with a new one. Only replaces code, PID and other information remains.

`exit()`
Takes in a status, terminates process and returns status to parent process.

Upon process termination, some resources are not released: PID, status, CPU time, etc. Generally PCB remains.

`wait()`
Waits for a child (blocks), if a child exists, else it returns `-1`. Cleans up on child process, e.g. removes PCB of child.

- **Zombie**: When child finishes but parent does not call `wait()`.
- **Orphan**: Subset of Zombie. When parent process terminates before child process. `init` becomes the parent and does the clean-up.

Process Scheduling

Concurrent execution ensures that multiple processes progress in execution. This can be **psuedoparallelism** or **physical parallelism**.

Common Criteria

1. **Fairness**: Get fair share of CPU time per process or per user. No starvation.
2. **Balance**: All parts of the computing system should be utilised.

Types of Scheduling Algorithms

1. **Non-preemptive/Cooperative**: Process runs until it blocks or gives up CPU voluntarily.
2. **Preemptive**: Process given a fixed amount of time. Possible to block or give up early.

Batch Processing Algorithms

No user interaction, and non-preemptive scheduling is predominant.

Batch Processing Criteria

1. **Turnaround Time**: Finish time minus arrival time.
2. **Throughput**: Number of tasks finished per unit time
3. **CPU utilisation**: Percentage of time when CPU is working on a task

First-Come First-Serve

FIFO queue based on arrival time.

- **No starvation**: Number of tasks in front of task X is always decreasing, assuming all tasks complete.
- **Convoy effect**: One long long task at the start can greatly increase turnaround time for short tasks waiting behind.

Shortest Job First

Select task with smallest total CPU time.

- **Minimises average waiting time**: Math
- **Possible starvation**: Long jobs may not get a chance
- **Difficult to predict**: Need to predict CPU time needed. Can be calculated using $\text{Predicted}_{n+1} = \alpha\text{Actual}_n + (1-\alpha)\text{Predicted}_n$.

Shortest Remaining Time

Select job with shortest remaining time.

- **Preemptive**: New short job can preempt long running job.

Interactive Algorithms

Preemptive scheduling is done to ensure good response times. Thus, the scheduler needs to run periodically.

Interactive Criteria

1. **Response Time**: Response time minus request time.
2. **Predictability**: Less variation means more predictable

Timer Mechanism

- **Timer Interrupt**: Goes off periodically and cannot be intercepted by other programs. The timer interrupt handler invokes the scheduler.
- **Interval of Timer Interrupt (ITI)**: OS scheduler is triggered every interrupt. 1ms to 10ms.
- **Time Quantum**: Execution duration given to a process, can be constant or variable. Must be multiples of ITI. 5ms to 100ms.

Round Robin

Deqeue and enqueue from a queue, giving each process a fixed time quantum. Blocked tasks are queued elsewhere until ready, then it will be queued back.

- **Response time guarantee**: The n-th task will run by $q(n-1)$
- **Choice of time quantum**: If big, better CPU utilisation but longer waiting time. If small, more overhead from context switching but shorter waiting time.

Priority Scheduling

Assign each task a priority value, then select task with highest priority value. If **preemptive**, higher priority tasks can preempt running ones with lower priority. Else, it will wait for next round of scheduling.

- **Handles priority**: Some tasks are more important, prioritising them is good
- **Starvation**: Low priority process can starve, especially if preemptive. Two ways to fix:
  1. Decrease priority of current running process every time quantum
  2. Stop running process after time quantum and exclude it from next round
- **Priority inversion**: A **lower** priority task preempts a higher priority one

Multi-Level Feedback Queue

Follows the following rules:

1. If Priority(A) > Priority(B), then A runs
2. If Priority(A) = Priority(B), then A and B runs in round robin
3. A new job gets max priority
4. If a job uses its time quantum, priority reduced
5. If a job gives up or block before the time quantum is up, priority maintained

- **Adaptive**: Adjusts based on actual behaviour, doesn't need prior knowledge
- **Minimises**: Response time for I/O bound processes and turnaround time for CPU bound processes
- **Possible abuse**: If a process keeps giving up right before time quantum is up

Lottery Scheduling

Gives out "lottery tickets" to processes, and a random one is chosen when scheduling is done.

- **Responsive**: New process can participate in next lottery
- **Level of control**: Such as:
  1. A process can distribute tickets to child processes.
  2. An important process can be given more tickets.
  3. We can give different types of tickets, e.g. tickets for I/O devices, tickets for CPU, depending on the usage of each resource per task.
- **Simple to implement**: Yeah.

Inter-Process Communication

Processes need to share information, but their memory spaces are independent.

Shared Memory

Process 1 creates a shared memory region. Process 2 attached the region to its own memory space. They can now write to that region and data can be seen by both parties.

Advantages

1. **Efficient**: Only creating + attaching require OS
2. **Easy to use**: Shared memory region behaves like a regular one. Can easily write info of any size or type.

Disadvantages

1. **Synchronisation**: Need to sync shared resource
2. **Hard to implement**: Shag

Shared Memory (UNIX)

`shmget()`
Creates a shared memory region, and returns the id. Generally done by master program.

`shmat()`
Attaches shared memory region using the id, and returns a pointer to the region. Done by both master and slave program.

`shmdt()`
Detaches shared memory region using the pointer from attaching. Done by both master and slave program.

`shmctl()`
Destroys the shared memory region using the id. Generally done by master program.

Message Passing

Process 1 prepares a message and sends it to Process 2. The message needs to be stored in kernel memory space. Process 2 receives the message. Both send and receive are system calls and need to go through the OS.

Naming

1. **Direct Communication**: Sender and receiver explicitly name the other party. Needs one link per pair and processes to know the identity of the other part.
2. **Indirect Communication**: Send and receive from a port. One port can be shared among a number of processes.

Synchronisation

1. **Blocking Primitives (Synchronous)**
   - Send: Sender is blocked until message is received
   - Receive: Receiver is blocked until message arrives
2. **Non-Blocking Primitives (Asynchronous)**
   - Send: Sender resumes operation immediately
   - Receive: Receiver gets message if available, else some indication that there is no message yet

Advantages

1. **Portable**: Can be easily implemented across different processing environments
2. **Easy to sync**: Using blocking primitives force sender and receiver to be synchronous

Disadvantages

1. **Inefficient**: Needs OS intervention
2. **Hard to use**: Messages are limited in size and format

Pipe (UNIX)

In UNIX, a process has 3 default comm channels, `stdin`, `stdout` and `stderr`. The `|` in shell directs one process' output to another's input.

More generally, a pipe is a FIFO circular bounded byte buffer with implicit synchronization - writers wait when buffer is full, readers wait when buffer is empty.

Variants

- Multiple readers and writers
- Half-duplex / unidirectional vs full-duplex / bidirectional

`pipe()`
Takes in an integer array of size 2, e.g. `fd[2]`. The resultant `fd[0]` is the reading end and `fd[1]` is the writing end.

`close()`
Takes in a file descriptor and closes it. The file descriptor is now unused. `0` is `stdin`, `1` is `stdout`, `2` is `stderr`.

`dup()`
Takes in a file descriptor (fd) and finds the lowest unused fd and duplicates it there, i.e. the lowest unused fd is now an alias for the argument fd.

`dup2()`
Like `dup()` except that you can specify the new file descriptor to duplicate into.

Signal (UNIX)

An asynchronous notification regarding an event sent to a process or thread. The recipient must either use the default set of handlers or a user supplier handler to handle the signal.

`signal()`
Takes in a signal and a handler that takes in argument of type `int` and returns `void`, and replaces the default handler with that.

Not all signals can have the handler replaced, e.g. `SIGKILL` cannot.

Threads

Process creation using `fork()` and context switching are expensive. Independent memory spaces means harder communication as well. We thus want to add threads so that multiple parts of the same program are executing concurrently.

Thread Model

Threads in the same process share

1. Memory context: Text, Data, Heap
2. OS context: process ID, resources like files, etc.

They have unique

1. Identification (thread ID)
2. Registers (general purpose, special)
3. Stack

Context Switching

Thus, **thread context switching** only involves hardware context, where registers, and FP and SP pointers are changed.

Benefits

- **Economy**: Much less resources needed
- **Resource sharing**: Threads share most of the resources, no need to pass information around
- **Responsiveness**: Multithreaded programs seem more responsive
- **Scalability**: Multithreaded programs can make use of multiple CPUs

Problems

- **System call concurrency**: Parallel system calls are possible, so we need to ensure correctness
- **Process behaviour**: Some parts are questionable
  1. When `fork()` is called, only one thread in new process is created (Linux)
  2. If a single thread calls `exit()`, all threads exit (Linux)
  3. If a single thread calls `exec()`, all threads exit and new executable

User Thread

The thread is implemented as a user library, i.e. a runtime system in the process will handle thread-related operations. The kernel is not aware of the threads in the process.

- **Works on any OS**: All can have multithreaded programs
- **Just library calls**: Thread operations are just lib calls
- **Configurable and flexible**: E.g. can have customised thread scheduling policy
- **OS is not aware**: Scheduling is performed at process level, i.e. if one thread blocks, process gets blocked, and all threads block. Also cannot exploit multiple CPUs

Kernel Thread

The thread is implemented in the OS, and operation is handled as syscalls. Thread-level scheduling by kernel is possible. In fact, the kernel may make use of threads for its own execution.

- **Thread table**: Maintain thread table in kernel, besides process table
- **Thread-level scheduling**: Multiple threads can run simultaneously on multiple CPUs
- **Syscalls are slow**: Expensive as well
- **Less flexible**: Used by all multithreading processes. If too many features, overkill and expensive for simple programs, but if too few features, it won't be flexible enough for some programs

Hybrid Model

We can have both models. OS schedule on kernel threads only, and user threads can bind to a kernel thread. This offers great flexibility.

Simultaneous Multi-Threading

When the hardware supports parallel execution of threads on the same core by providing multiple sets of registers. Example would be hyperthreading on Intel processors.

UNIX Commands

`pthread`
The POSIX thread API is defined by the IEEE and supported on most UNIX variants. As implementation is not specified, it can be implemented as both a user or kernel thread. `pthread_t` is the data type for a pthread id, and `pthread_attr` is a data type for thread attributes.

`pthread_create()`
Takes in 4 parameters and returns an integer, 0 for success.

1. Pointer to pthread id, which it will update
2. Pointer to pthread attributes to control the behaviour of the thread
3. A pointer to the function (start routine) to be executed by the thread. The function should take in a void pointer for arguments, and return a void pointer as well
4. Arguments for the start routine, as a void pointer

`pthread_exit()`
Takes in a void pointer exit value and terminates. If not specified, the return value of the start routine function is captured as the exit value.

`pthread_join()`
Takes in a pthread id and a void pointer pointer for status, and returns 0 if success + updates status, which is the exit value returned by target pthread.

Synchronisation

We face synchronisation problems when we have concurrent processes execute in an interleaving fashion and share some modifiable resource.

Race Condition

Race conditions refer to situations where execution outcomes depend on the order in which the shared resources are accessed or modified.

The solution is to designate the segment with race conditions as a **critical section**.

Critical Section

In a critical section, at any point there can only be one process executing.

- **Mutual exclusion**: If process P is executing in a critical section, all other processes should be not able to enter
- **Progress**: If no process is in the critical section, then one process should be granted access
- **Bounded wait**: If process P asks to enter the critical section, there should be an upper bound on the number of times other processes can enter before P does
- **Independence**: Process not executing in the critical section should not block other processes

Symptoms of Incorrect Synchronisation

- **Deadlock**: All processes blocked, no progress. It needs the following conditions to be all met:
  1. **Mutual exclusion condition**: Each resource is either assigned to exactly one process or to none
  2. **Hold-and-wait condition**: Processes holding resources can ask for more resources
  3. **No-preemption condition**: Cannot forcibly take away resources previously given
  4. **Circular wait condition**: Must have a circular list of two or more processes, each waiting for a resource held by the next
- **Livelock**: In an attempt to avoid deadlock, processes keep changing state and make no other progress
- **Starvation**: Some processes are perpetually denied needed resources and cannot make progress

Implementations

Test And Set

`TestAndSet Register, MemoryLocation`: Assembly level implementation. We load the value at memory location into the register AND change the value at memory location to 1. This implementation works and meets all 4 criteria.

- **Atomic operation**: Single instruction and cannot be interrupted.
- **Busy waiting**: Keep checking until safe to enter, wasteful use of processing power
- Variants exist on most processors, e.g. Compare and Exchange, Atomic Swap, Load Link / Store Conditional

Peterson's Algorithm

The following code segments work on the assumption that writing to `Turn` is an atomic operation. We basically create two boolean conditions that prevent a deadlock.

- **Busy waiting**: Same as before
- **Low level**: Error prone and is not very simple to implement
- **Hard to generalise**: Limited to two processes

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

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

Semaphores and Mutexes

A generalised synchronisation mechanism that specify behaviour and not implementation. It provides a way to block a number of processes, which will then be known as **sleeping processes**.

We can see a semaphore S as a protected integer, with a non-negative initial value. A general semaphore or counting semaphore can have values $S \geq 0$. A binary semaphore or mutex has values $S = 0$ or 1.

`wait()`
Takes in a semaphore. If the semaphore value is ≤ 0, it blocks the current process and decrements the value. This is an atomic operation.

`signal()`
Takes in a semaphore, wakes up one sleeping process (if any) and increments the semaphore value. This is an atomic operation and never blocks.

Invariant

$S_{\text{current}} = S_{\text{initial}} + \#\text{signal}(S) - \#\text{wait}(S)$, where:
$S_{\text{initial}} \geq 0$,
$\#\text{signal}(S)$ is the number of `signal()` executed,
$\#\text{wait}(S)$ is the number of `wait()` operations completed.

Conditional Variable

Similar to semaphore, but more of allowing a process to wait for some event to happen. Once that event happens, a broadcast is made to wake up all waiting tasks.

Classical Problems

We will discuss the problems here. For solutions, which can be lengthy, refer to the **Little Book of Semaphores**.

Producer Consumer

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

How can we synchronise the two?

Readers Writers

Processes share a data structure D, where readers can access and read information from D together, while writers must have exclusive access to D to write information.

How can we synchronise the two? And how do we prevent the writer from starving, i.e. readers keep reading?

Dining Philosophers

Five philosophers are seated around a table, and there are five single chopsticks placed between each pair of philosophers. When any philosopher wants to eat, he/she will have to acquire both chopsticks from his/her left and right.

How can we have a deadlock-free and starvation-free way to allow the philosophers to eat freely?

Contiguous Memory Allocation

The OS needs to perform the following tasks:
- Allocate memory space to new process
- Manage memory space for process
- Protect memory space between processes
- Provide memory-related syscalls
- Manage memory space for internal use

No Memory Abstraction

- **Pros**: Straightforward, fast, addresses fixed during compile time
- **Cons**: Both processes assume they start at 0, resulting in conflicts. It is also hard to protect memory.

Solution 1: Relocate Addresses

Recalculate memory references when process is loaded into memory, e.g. if process B is located at address 8000, add 8000 to all memory addresses.

However, the loading time is slow and it is not easy to distinguish a memory address from any arbitrary integer.

Solution 2: Base + Limit Registers

Use a special **Base Register** that stores the starting address of the process memory space. All memory references will be offset by this register value. We then use a **Limit Register** to indicate the range, i.e. cannot access past the limit.

However, there is a lot of overhead since we need to perform an addition and comparison per access. This is later generalised in segmentation.

Physical and Logical Addresses

Generally, embedding physical addresses in programs is a bad idea. We thus let each process have a self-contained, independent logical memory space that they will reference using logical addresses, then the OS will do the mapping.

We then need some ways to do partitioning of the memory, so that we can switch between processes using the different memory partitions. When the physical memory is full, we can either remove partitions used by terminated processes or swap the blocked process to secondary storage.

The following algorithms work on the following assumptions:
- Each process uses a contiguous memory region
- The physical memory is large enough to contain one or more processes with complete memory space

Fixed Size Allocation Algorithms

Physical memory is split into a fixed number of partitions, and a process will occupy one of them.
- **Internal fragmentation**: When a process does not occupy the entire partition
- **Easy to manage and fast to allocate**: Just give any free partition, it's all the same
- **Need to fit largest process**: This also means all smaller processes will waste space

Variable Size Allocation Algorithms

Also known as dynamic partitioning. We allocate just the right amount of space needed, forming **holes** between such partitions.
- **No internal fragmentation**: All processes get the exact space required
- **External fragmentation**: When holes between processes become unusable, wasting space. Can be fixed via compaction, but it's slow. Still better than internal, since internal is unreachable **Need to maintain more info**: In the OS **Slower to allocate**: Need to find appropriate region

First-Fit Allocation

Take the first hole that is large enough. Split it into two, give the process the required space, and the remaining space is a new hole.

Best-Fit Allocation

Take the smallest hole that is large enough. Rest is same as above.

Worst-Fit Allocation

Take the largest hole. Rest is same as above.

Dellocation and Compaction

Try to merge freed partition with adjacent holes. We can also do compaction i.e. move partitions around, but this is expensive and should not be done frequently.

Partition Info

Partition info can be maintained as either a linked list or a bitmap. For linked list, each node contains:
1. Status: True = Occupied, False = Free
2. Start Address
3. Length of partition/hole
4. Pointer to next node

Buddy System

To be completed...

Disjoint Memory Allocation

We now remove one assumption - that process memory space is contiguous. They can now be in disjoint physical memory locations. This is achieved via paging.

Paging

We split physical memory into **physical frames** and logical memory into **logical pages** of the same size. Then logical memory can remain contiguous, while each page can be mapped into disjoint frames!

Page Table

To support translation, we will use a page table, which is an array where the index is the page number and value is the frame number. Two tricks are employed in calculation:
1. Keep frame/page size as a power of 2
2. Keep frame and page size equal

Then, for page/frame size of 2^n and address of m bits, to translate, we copy the last n bits, the offset, the over, then for the rightmost $m - n$ bits (page number), we index it into the page table and replace it with the value (frame number).

Analysis of Paging

- **No external fragmentation**: No leftover physical memory region
- **Has internal fragmentation**: When a logical memory space is not a multiple of page size
- **Clear separation of logical and physical**: Flexibility and simple translation

Implementation

Each process has its own page table, stored in its PCB. This PCB is in RAM. However, this means we require **two RAM accesses** for every memory reference.
1. Read the indexed page table entry to get frame number
2. Access to actual memory item

Translation Look-Aside Buffer

The TLB is on the chip, and provides hardware support for paging. It caches 4KB of page table entries, and can cache multiple at once, like associative cache.
- **Fast**: Takes 1ns vs 50ns for RAM
- **TLB-Hit vs TLB-Miss**: RAM is only accessed upon the latter, and TLB is updated after that
- **Context switching**: In CS2106, can assume TLB is fully flushed, as it is part of a process' hardware context

Protection

We can extend the paging entries to include bits to support memory protection.
1. **Access right bits**: On whether the page itself is writable, readable or executable, e.g. you cannot write over the text of the process, but you can execute
2. **Valid bit**: Some pages may be out of range for certain processes for certain reasons. OS will set these valid bits when running. If out-of-range access is done, OS will catch

Page Sharing

We can naturally now have multiple pages pointing to the same physical frame. For example, some library code are shared between processes. We can use a **shared bit** in the page table entry to track whether the page is shared.
- **Shared code page**: As mentioned, when there's library code or system calls, etc.
- **Copy-On-Write**: When a parent forks, the parent and child share the same pages, i.e. page table is copied but frames remain. Using a shared bit, when the child needs to update a shared page, then the frame is duplicated and page "unshared"

Segmentation

The memory space actually contains multiple regions, all with different usages in a process. Some remain constant in size, e.g. data and text, while some will shrink and grow during execution time, e.g. stack, heap and library code regions.

It is thus difficult to put all the regions together in a contiguous logical memory space and allow them to shrink/grow freely. Also hard to check if a memory access is in-range.

Segmentation Scheme

- **Memory segments**: Split the regions into their own segments
  - Name: For reference, usually translated to an index, e.g. Text = 0, Data = 1, Heap = 2, etc.
  - Base: The physical base address
  - Limit: Indicate range of segment

Segment Table

We keep a table of [Base, Limit], indexed by the name indices / segment ids.

1. With [SegId, Offset], we use SegId to get the [Base, Limit]
2. We check if Offset < Limit, if so, segmentation fault
3. Else we access Base + Offset

We store this segment table inside registers since its size is fixed and small, and it's fast.

Analysis of Segmentation
- **Independent segments**: Each segment is contiguous and independent, and can shrink and grow independently
- **External fragmentation**: Variable size contiguous memory regions result in the same problem as always
- **Not the same as paging**: Solving different problems

Segmentation with Paging
Each segment is now composed of pages and has a page table of its own. The segment table now points to the page table address instead of the base address. Page limit remains unchanged.

Virtual Memory

We shall remove our last assumption that our physical memory is large enough to hold one or more processes' logical memory space completely. A popular solution is to keep some pages in the physical memory, while others on secondary storage or disks. Bringing the pages in and out of memory is called **swapping**.
- **Logical not restricted by physical**: Logical memory address size is no longer restricted by physical
- **More processes in memory**: Improves CPU utilisation since more processes to choose to run

Extended Paging Scheme
We add a "Is Memory Resident" bit in page table entries. When CPU tries to access non-memory resident pages, a page fault occurs.

Page Fault
This is the updated page accessing process, now with page faults.
1. (Hardware) Check page table. If memory resident, done, else continue
2. (Hardware) **Page fault**: `TRAP` to OS
3. (OS) Locate page in secondary storage
4. (OS) Load copy of page in physical memory frame
5. (OS) Update page table
6. (OS) Go back to step 1 to retry

Locality
There's always the chance of **thrashing**, which is when page faults repeatedly occur. But this is unlikely due to locality.
- **Temporal**: Memory address used now is likely to be used again. Cost of loading is amortized.
- **Spatial**: Nearby memory addresses are likely to be used next. They are included in one page.

Demand Paging
What we have described thus far is demand paging, i.e. OS only copies a page into memory if a page fault occurs, as compared to anticipatory paging. This makes it such that unneeded pages will almost never be loaded, resulting in more efficient use of physical memory.

Page Table Structure
Although the large number of pages are on disk, the page tables themselves still take up a lot of space in memory. This results in high overhead and fragmentation, since the table itself needs to occupy several pages

Direct Paging
We keep all entries in a single page table, and we allocate each process this huge page table. To compute the size of this table:
1. Let's say there are 2^p pages, we will need p bits to specify one unique page
2. If virtual address has 32 bits, and each page/frame is 4KB = 2^{12}B, then $p = 32 - 12 = 20$ bits.
3. This also means 2^{20} pages and page table entries.
4. If each page table entry is 2 bytes, then page table size is $2^{20} \times 2\text{B} = 2\text{MB}$.

But not all processes use the full virtual memory space, so much of this table will be unused.

2-Level Paging
We further split the page table into regions, and only allocate these regions when needed. We then keep a directory of these regions or split page table. Each table entry points to the base address of the next page table. This can be further extended to multilevel paging.
- **Size of page table**: Generally we want each table to be the same size as a page, to reduce fragmentation
- **Page directory base register**: We just need one register to store the base address of this process' page directory, and update this during context switching
- **Overhead calculation**
  1. Let's say all tables/pages are 4KB and each entry is 2B, and virtual address is 32 bits.
  2. Since each frame/page is 4KB, the offset is 2^{12}B, i.e. we will only look the first 20 bits of the address.
  3. Since each page table (the ones that are split) is 4KB big, and each entry is 2B, then each table has $2^{12} \div 2 = 2^{11}$ entries
  4. That means the directory has $2^{20} \div 2^{11} = 2^9$ entries. This means it will take up $2^9 \times 2\text{B} = 2^{10}\text{B} = 1\text{KB}$
  5. Although technically this directory itself would also take 1 page, we can consider it to only have an overhead of 1KB. Let's say we only allocated and are using 3 page tables. Then total overhead = 1KB + 12KB = 13KB.

Inverted Page Table
We want to keep a page table that tells us which process is using which frame. We can thus keep an array with the same size as number of frames. In each entry, we store the **PID** and **page number**.
- **Fast lookup by frame**: Often to support RAM management operations
- **Ease of use**: Frame management is a lot easier + one table for all processes
- **Slow translation**: From page number to frame number, since we need to search through the entire table. Luckily normally we use page directories and tables for that

Page Replacement Algorithms
When there are no free pages during a page fault, we will need to replace an existing page. Based on a **dirty bit** for the page entry, if the page has been modified, it will need to be written back to disk.

Memory Reference Strings
The offset does not matter when talking about page replacement, only page numbers. A sequence of page numbers is called a memory reference string.

Memory Access Time
$T_{\text{access}} = (1-p) \times T_{\text{mem}} + p \times T_{\text{page fault}}$

We want to reduce p since $T_{\text{page fault}} >> T_{\text{mem}}$.

Optimal Page Replacement
Replace the page that will not be used again for the longest period of time.
- **Minimum page faults**: Guaranteed, and often used as a benchmark
- **Need to know the future**: Simply not feasible

FIFO Page Replacement
Evict oldest memory page.
- **Queue**: Maintain a queue of resident page numbers, and update it during page fault
- **Simple to implement**: No need hardware support
- **Bélády's anomaly**: Generally, if the number of frames in RAM increases, then page faults should occur less frequently. However, for FIFO, we may actually see the opposite.
- **Does not exploit temporal locality**: This is the reason for above

Least Recently Used Page Replacement
Replace the page that has not been used in the longest time.
- **Temporal locality**: Makes use of it
- **Close to optimal algorithm**: Good approximation
- **No Bélády's anomaly**: Does not suffer
- **Difficult implementation**: Need to keep track of last access time, thus need substantial hardware support
  1. **Time counter**: A logical time counter that increments for every stored reference and is stored along with it. However, deletion is $O(n)$ since we need to find the page with the lowest counter. We may also have overflow issues with the counter
  2. **Stack**: Maintain a stack of page numbers. When a page is referenced, we pop it out from the stack (if it's inside), and push it to the top. For replacement, we remove the bottom most page. Hard to implement in hardware, and it's not exactly a stack, since we can pop from anywhere.

Second Chance Page Replacement (CLOCK)
It's the same as FIFO, but we maintain a separate **reference bit** for each page entry. When all page entries have the same reference bit, it effectively becomes a FIFO algorithm.
1. We maintain a circular queue of page numbers, and a pointer to the "oldest", or victim page
2. When we load a page entry, we set the reference bit to 0 (for CS2106)
3. Upon accessing the page entry, we will set the reference bit to 1
4. When a page replacement is required, we check the current victim page
   (a) If the reference bit is 0, we replace it
   (b) Else if the reference bit is 1, we set it to 0 and move the pointer to the next page
5. Repeat step 4 until a victim page with reference bit 0 is found

Frame Allocation
If there are N physical frames and M processes competing for frames, we need to distribute these frames.
1. **Equal allocation**: Each process gets $\frac{N}{M}$ frames
2. **Proportional allocation**: Each process gets $\frac{\text{size}_p}{\text{size}_{\text{total}}} \times N$ frames

Local Replacement
Victim page is selected among pages of the process that caused the page fault.
1. **Constant frame allocation**: Number of frames is the same, performance is stable between runs
2. **Insufficient allocation**: Will result in hindering of the process
3. **Thrashing**: Limited to one process, but that single process can hog the I/O and degrade the performance of other processes

Global Replacement
Victim page is selected among pages of all processes.
1. **Self-adjustment**: Processes that need more frames can get them from other processes
2. **Malicious processes**: Can affect others
3. **Inconsistent performance**: Between runs
4. **Cascading thrashing**: One process that thrashes will steal pages from others, resulting in other processes also thrashing

Working Set Model
Generally, the set of pages referenced by a process is quite constant in a period of time due to locality. The number of page faults is minimal until the process transmits to a new locality, e.g. new function call, etc.
- Define a working set window Δ, which is an interval of time
- $W(t, \Delta)$ is the set of active pages in the interval at time t
- We thus want to allocate enough frames for pages in $W(t, \Delta)$ to reduce page fault
- Accuracy of model depends on choice of Δ
  1. Too small: miss pages in current locality
  2. Too large: contains pages from different localities

All Page Entry Bits Thus Far
1. Access Right Bits
2. Valid Bit
3. Is-Shared Bit
4. Is-Memory Resident Bit
5. Dirty Bit
6. Reference Bit (for CLOCK)

File Systems

Consists of files and directories, and provides an abstraction for accessing these.

General Criteria
- **Self-Contained**: Info on media should describe the entire organisation. Hence, can plug-and-play on another system.
- **Persistent**: Data persists beyond processes and OS
- **Efficient**: Good management of free and used space + minimal overhead for bookkeeping

File
A logical unit of information created by a process.

File Metadata
Attributes associated with the file.

Name
A human readable reference to the file. Different FS have different rules. Rules include length, case-sensitivity, allowed special symbols and file extension. Some FS use extension to indicate file type.

File Type
Each file type has an associated set of operations and possibly a specific program to process it.
- **Regular files**: contains user info
  - ASCII files: text files, source codes, etc. Can be printed as is.
  - Binary files: executables, `mp3`, `pdf`, etc. Have a predefined internal structure that needs a specific program to process
- **Directories**: system files for FS structure
- **Special files**: character/block oriented
- **Windows**: Uses file extension as file type
- **UNIX**: Uses **magic number** embedded at beginning of file

File Protection
Go and revise CS2107 notes on Access Control List and permission bits.

File Data
Different Structures
- **Array of bytes**: Just raw bytes having a certain offset from the start of file
- **Fixed length records**: Each record has a fixed size. This array of records can grow or shrink, and you can jump to any record using offset.
- **Variable length records**: Flexible but hard to locate a record

Access Methods
- **Sequential access**: Read in order from beginning. Cannot skip but can rewind.
- **Random access**: Read in any order, either via a `read(offset)` or `seek(offset)`. UNIX and Windows both use the latter.
- **Direct access**: Basically random access but with records instead of bytes

File Operations
Generally these operations are system calls, as it provides protection, and concurrent and efficient access.

Metadata Operations
1. Rename
2. Change attributes, e.g. file access permissions, dates, ownerships, etc.
3. Read attribute, e.g. get file creation time

Data Operations
1. **Create** a new file with no data
2. **Open** a file to prepare the necessary information for file operations later
3. **Read** data from file, usually from current position
4. **Write** data to file, usually from current position
5. **Reposition** (seek) to new location
6. **Truncate** removes data between specified position to end of file

Representation of Open File
1. **File pointer**: Current location in file
2. **Disk location**: Actual file location on disk
3. **Open count**: Useful to determine when to remove the entry

Two Table Approach
1. **System-wide open-file table**
   (a) Keeps track of open files in the system
   (b) If one process opens the same file twice, or two processes open the same file, there will be two separate entries
   (c) If one process opens a file then forks, only one entry here
2. **Per-process open-file table**
   (a) Keeps track of open files for a process, also known as **file descriptors**
   (b) Each entry points to a system-wide table entry
   (c) If one process opens a file then forks, there will be two fds pointing to the same system-wide table entry. They will thus share the same offset.

Directory
Helps user group files, and helps system keep track of files.

Single-Level
All files are contained under the root directory.

Tree-Structured
Since directories can contain directories, they are like trees or subtrees. Files are like leaves.
- **Absolute pathname**: Can refer to file by path from root to file.
- **Relative pathname**: Can refer to file by path from **current working directory**.

DAG
To get a DAG from a tree, we must be able to "skip" levels, i.e. share a file/directory such that it appears in multiple directories but refers to the same copy of actual content.
- **Hard link**: Directories A and B have separate pointers to file F. Only works with files.
  - Pros: Low overhead, only adds pointers
  - Cons: Deletion problems - what if one directory deletes F?
  - `ln` command in UNIX
- **Symbolic link**: Directory B creates a special link file G that contains path name of F. When G is accessed, we access F instead.
  - Pros: Simple deletion. B deletes G, not F. A deletes F, and G remains but does not work.
  - Cons: Larger overhead - G takes up actual disk space
  - `ln -s` command in UNIX

General Graph
Created when the tree has a cycle. Though possible in UNIX, it is not desirable.
- **Hard to traverse**: Need to prevent infinite loops.
- **Hard to determine when to remove a file/directory**

I/O
Disk Structure
- **Track**: One ring around the disk. One disk can have many tracks of different radii
- **Sector**: Much like the sector of a circle, this is a sector of a track
- **Disk head**: The stick thingy that move above the disk and transform the disk's magnetic field into electrical current or vice versa
- **Rotation**: By rotating, we can change sector
- **Seek**: By shifting the disk head towards and away from the centre of the disk, we can change track

Disk Scheduling Algorithms
Due to rotational and seeking latency, we need to schedule disk I/O requests to reduce **overall waiting time**. Generally it's easier to reduce seeking time.
- **First-Come First-Serve**
- **Shortest Seek First**
- **SCAN**: Bidirectional, go from innermost to outermost, then back to innermost. Much like lift algorithms.
- **C-SCAN**: Unidirectional, go from outermost to innermost, then reset.
- **Deadline**: 3 queues for requests
  1. Sorted
  2. Read FIFO (sorted chronologically)
  3. Write FIFO (sorted chronologically)
- **noop**: No sorting involved. Actually same as FCFS
- **cfq**: Completely fair queuing, time sliced and per-process sorted queues
- **bfq**: Budget fair queuing or multiqueue, fair sharing based on number of sectors requested

UNIX Context
`open()`
Takes in a path and flags, and returns a file descriptor (fd) integer.

`read()`
Takes in a fd, a buffer and an integer `n` and reads up to `n` bytes into the buffer. It is sequential - starts at current offset and increments offset by bytes read.

`write()`
Takes in a fd, a buffer and an integer `n` and write up to `n` bytes from buffer into the file. It is sequential - starts at current offset and increments offset by bytes written. Appends new data if file size goes beyond EOF. Throws error if file size limit, quota, disk space, etc. are exceeded.

`lseek()`
Takes in a fd, an `offset` and a `whence` and moves current position in file. If `offset` is positive, move forward, else move backward. The value of `whence` can be:
- `SEEK_SET`: set absolute offset (from start of file)
- `SEEK_CUR`: set relative offset from current position
- `SEEK_END`: set relative offset from end of file

`close()`
Closes an opened fd, and kernel can remove associated data structures. Process termination automatically closes all open files.

FS Implementation

Earlier, we saw that a disk has various sectors. To the OS, what it sees is a 1-D array of **logical blocks**, usually 512B to 4KB big. The mapping between logical blocks and sectors is hardware dependent.

Disk Organisation
- **Master boot record**: Found at sector 0. Contains:
  - Simple boot code
  - Partition table
- **Partitions**: Each partition is an independent file system. Contains:
  1. OS boot block (information for boot-up)
  2. Partition details, e.g. total number of blocks, number and location of free disk blocks
  3. Directory structure
  4. File information
  5. Actual file data

File Information and data
In the process of allocating file data, we need to keep track of logical blocks, allow efficient access and utilise disk space efficiently. There will be **internal fragmentation** as file size may not be a multiple of logical block size.

Contiguous Allocation
Allocate consecutive disk blocks to a file. File information will just store start + length for each file.
- **Easy to keep track**: Each file just needs starting block + length
- **Fast access**: Just need to seek first block
- **External fragmentation**: After a lot of creation/deletion, disk can have many small "gaps"
- **File size**: Needs to be specified in advance

Linked List
Maintain disk blocks as linked list. Each disk block stores the next disk block number besides the file data. File information will store first and last disk number for each file.
- **No external fragmentation**: Solves the previous problem
- **Slow access**: O(n) access
- **Less usable space**: Need to use space for pointer
- **Less reliability**: Fails if one pointer is incorrect

Linked List V2
Same as linked list but now we move all pointers into a **file allocation table** (FAT) in memory. This is used by MS-DOS.
- **Faster access**: Linked list traversal is now in memory
- **Takes up space**: FAT tracks all disk blocks. This number can be huge when disk is large, consuming valuable memory.

Indexed Allocation
We use one block per file to store an array of indices of all other blocks. For example, index of 0-th block will be value of IndexBlock[0].
- **Less memory overhead**: Only index block of opened files need to be in memory
- **Fast direct access**: No need for traversal.
- **Limited max file size**: Max number of blocks is the max array size of index block.
- **Index block overhead**: I guess it consumes space + need to keep it updated?

Free Space Management
We want to know which disk blocks are free or not free, so that we can easily allocate or free blocks.

Bitmap
Each block is 1 bit, if it's 1 means it's free, else it's taken.
- **Easy to manipulate**: Use bit operations to find first free block or n-consecutive free blocks

- **Need to keep it memory**: For efficiency reasons.

Linked List

We basically have a linked list of blocks used to contain indices of free blocks.
- **Easy to locate free block**: Just grab any index
- **Just need pointer to first block in memory**: Though we can cache the blocks
- **High overhead**: We can mitigate this by using free blocks to store this list
- **Runs alternative**: If the free space tends to be in contiguous runs, we can instead store first index + length. But may not work as well once fragmentation worsens

Directory Structure

The goal of a directory structure is to provide some mapping between file name and file information. A sub-directory is usually stored as file entry with special type in a directory.

Linear List

Directory contains a list of entries, one entry per file. The entry contains the file name (and possibly other metadata) and file information or pointer to file information.
- **Inefficient searching**: Especially for large directories or deep tree traversal
- **Solution**: Cache the last few searches

Hash Table

Directory contains a size N hash table. We hash the file name from 0 to $N-1$, and use chained collision resolution.
- **Fast lookup**
- **Limited size for hash table**
- **Depends on a good hash function**

File Information

Two approaches to store file information in a directory entry:
1. **Store everything in directory entry**: Likely have some fixed size entry
2. **Store only file name and pointer**: Pointer to some other data structure for more information

MS-DOS FAT

As mentioned before, it works with linked list block allocation and the FAT is cached in RAM for traversal. The index of the block is the index of its FAT entry. A block is also called a **cluster**.

FAT FS Layout

Each partition has:
1. OS boot block
2. FAT
3. FAT duplicate (optional)
4. Root directory
5. Data blocks

FAT Entry Values
1. FREE code, block is unused
2. Block number of next block
3. EOF code i.e. NULL
4. BAD block, e.g. unusable, disk error, etc.

Directory
- Special type of file
- Root directory stored in special location, rest are in data blocks.
- Each file or subdirectory is represented by a directory entry

Directory Entry

Each entry has a fixed size of 32 bytes.
- Name + Extension (11 bytes)
  - Limited to 8 + 3 characters
  - The first byte of name may have meaning, e.g. deleted, end of directory entries, parent directory, etc.
- Attributes e.g. read-only, is-directory/file flag, hidden flag, etc. (1 byte total)
- Reserved (10 bytes)
- Creation Date (2 bytes) + Time (2 bytes)
  - Year range: 1980 - 2107
  - Accuracy of second is ± 2 seconds
- First disk block (2 bytes)
  - Disk block index is 12, 16 and 32 bits for FAT12, FAT16 and FAT32 respectively
  - For FAT32 lower half is stored (2 bytes)
- File size in bytes (4 bytes)

Tracing Process
1. Get the required directory table from disk blocks. Likely cache at this point.
2. Get required directory entry and find first disk block number
3. Use FAT to find the remaining disk block numbers, until EOF code is encountered
4. Use the numbers to do disk access on file data blocks. If subdirectory, repeat from step 1.

Linux Ext2

The layout of the file system is now slightly different. Blocks are grouped into **block groups** and each file/directory is described by a single special structure called **I-Node** (Index Node), containing file metadata and data block addresses.
- **Master boot record**: Same as before
- **Partitions**: Similar to before
  1. OS boot block
  2. Multiple block groups, each containing
     (a) **Superblock**: Describes the whole file system, e.g. total I-Node number, I-Nodes per group, total disk blocks, disk blocks per group, etc. Duplicated in each block group for redundancy.
     (b) **Group descriptors**: Describe each of the block group, e.g. number of free disk blocks and I-Nodes per group, location of bitmap etc. Duplicated in each block group for redundancy.
     (c) **Block bitmap**: Keeps track of usage status of blocks in this block group (1 = Occupied, 0 = Free)
     (d) **I-Node bitmap**: Same but for I-Nodes
     (e) **I-Node table**: Array of I-Nodes in this block group
     (f) Data blocks

I-Node Structure

Each I-Node is 128 bytes, consisting:
- **Mode**: File type (regular, directory, special) + file permissions (2 bytes)
- **Owner info**: User Id (2 bytes) + Group Id (2 bytes)
- **File size**: Size in bytes (4 bytes for smaller files, 8 bytes for larger files)
- **Timestamps**: Creation, Modification & Deletion timestamps (3 x 4 bytes)
- **Data block pointers**: Indices of data blocks, with 12 x direct, 1 x indirect, 1 x double indirect, 1 x triple indirect (15 x 4 bytes)
- **Reference count**: Number of times this I-Node is referenced by directory entry (2 bytes)
- Other fields

Multilevel Data Blocks

Allows for fast access to small files and flexibility to handle huge files. The first 12 data block pointers directly point to the disk block. The next pointer points to a **single indirect block**, which contains direct pointers. The next pointer points to a **double indirect block**, which contains pointers to single indirect blocks. Finally, we have one **triple indirect block**.

Assuming each disk block address is 4B, each disk block is 1KB, then in total, we have $12 \times 1\text{KB} + 256 \times 1\text{KB} + 256^2 \times 1\text{KB} + 256^3 \times 1\text{KB} = 16843020\text{KB} = 16\text{GB}$.

Directory Structure

A directory is just another file. Within this file, the data blocks form a "linked list" of directory entries for files and subdirectories.
1. I-Node number for this file or subdirectory. 0 is used to indicate unused entry
2. Size of this directory entry, so we can traverse to the next entry
3. Length of the name
4. Type: file, subdirectory or special
5. Name of file/subdirectory (up to 255) characters

Tracing Process
1. Get root directory I-Node. Generally it's a fixed number e.g. 2. Read it.
2. If next part in pathname is a subdirectory
   (a) Locate directory entry in current directory
   (b) Get I-Node number, then read the I-Node, which will contain metadata on the subdirectory data
   (c) Current directory is now this subdirectory. Repeat step 2.
3. Else if it's a file
   (a) Locate directory entry in current directory
   (b) Get I-Node number, then read the I-Node, which will contain metadata on the file data

Higher Level Operations
- **Open**
  1. Go through tracing process. If F is not found, terminate with error.
  2. Load file information into a new entry E in the system-wide table (i.e. store the I-Node pointer)
  3. Create an entry (file descriptor) in process P's table to point to E
  4. Return fd of this entry
- **Delete**
  1. Remove its directory entry by pointing the previous entry to the next entry. If its the first entry, we need to make it a blank record.
  2. Update I-Node bitmap and mark the corresponding I-Node as free
  3. Update block bitmap and mark the corresponding blocks as free
- **Hard Link**
  1. Create a new directory entry in B with same I-Node number as X in A
  2. Update X I-Node's reference count
  3. For deletion, decrement the reference count, then when it goes to 0, perform actual deletion
- **Symbolic Link**
  1. Create a new file Y in directory B, i.e. new Y I-Node + new directory entry for Y
  2. Store pathname of file X in file Y, i.e. that is the file content
  3. As only pathname is stored, the link can be easily invalidated, e.g. via name changes, deletion, etc.

Extra Knowledge

FS Consistency Check

Performed when sudden power loss or crash happens. `CHKDSK` for Windows, `fsck` for Linux.

Defragmentation

When fragmentation on disk gets too severe, I/O performance may suffer. On Windows, official and 3rd party software are used to solve this problem. On Linux, the file allocation algorithm allocates files further apart, and free blocks near existing data blocks are used if possible, keeping fragmentation low when disk occupancy is $< 90\%$.

Journaling

It's kind of like saving state. Information or actual data is written into a separate log file before an operation is performed, so that it can recover to an earlier stable state or re-perform the interrupted file operation.

Interesting FS
- **Virtual FS**: Provides another layer of abstraction on top of existing file systems, allowing applications to access files on different file systems
- **Network FS**: Allows files to reside on different machines, and file operations are now network operations
- **New Technology FS**: Used in Windows XP onwards, provides file encryption, compression, versioning, and hard/symbolic link
- **Ext3/Ext4**: Variant of Ext2 with journaling and expanded max file and file system sizes
- **Hierarchical FS Plus (HFS+)**: Used in Mac OS X, provides compression, encryption, large FS/files/number of files support, metadata journaling