1 Introduction to OS

1.1 What is OS?

- A program that acts as an **intermediary** between a computer user and the computer hardware (allows user programs to interact with hardware in a safe way)
- Manages hardware and software resources and provides essential services to other programs

1.2 Brief History of OS

OS evolves with:

- Computer Hardware (e.g. integration of GPU, Multicore CPU)
- User application and usage patterns (e.g. phones have different UI than computers)

1.2.1 OS for the first computers

1. **No OS**
   - Programs directly interacted with hardware
   - **Advantage**: Minimal Overhead
   - **Disadvantages**: Not portable, inflexible and does not use computer resources efficiently
2. **Batch OS**
   - Allows for batch processing but programs by storing programs as digital format (e.g. tapes)
   - As a user, you still interact directly with the hardware, with additional information for the OS (resources required, job specification)
   - **Advantage**: Batch processing (write multiple programs and leave it to run)
   - **Disadvantages**: Inefficient (CPU idle when performing I/O)
3. **Time-sharing OS**
   Time sharing: Multiple users running multiple programs, **Multiprogramming**: 1 user running multiple programs (To have time sharing, we must have multiprogramming)
   - Allows multiple users to interact with machines using terminals
   - supports Job scheduling → concurrency of programs
   - **Advantage**: More efficient usage of CPU by allowing sharing of CPU time, memory and storage

1.3 Motivation for OS

Manage resources and coordination to allow programs to run simultaneously (process sync, resource sharing), Simplify programming (abstraction of hardware, convenient services), Enforce usage policies, Security and protection, User program portability: across different hardware, Efficiency: Sophisticated implementations optimised for particular usage and hardware.

1.4 OS Structures

1.4.1 Monolithic

- **Kernel is one BIG special program**, various services and components are integral part (e.g. Most Unix variants, Windows XP)
- Good SE principles possible with modularisation, separation of interfaces and implementation
- **Advantages**: Well understood, Good performance (Components of OS are not split up → communications between components take minimal time)
- **Disadvantages**: Highly coupled components (entire system breaks if a single component breaks), Usually devolved into very complicated internal structure

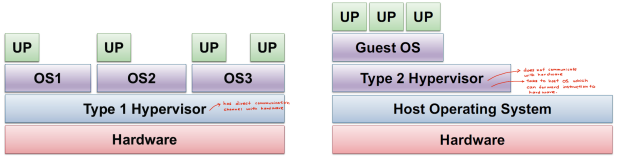
1.4.2 Microkernel

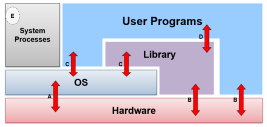
- **Kernel is very small** & clean, only provides basic and essential facilities: IPC, address space & thread management, etc.
- **Higher level services** built on top of the basic facilities, run as server process outside of the OS, using IPC to communicate
- **Advantages**: Kernel is generally more robust & extensible, better isolation & protection between kernel & high level services (if one part breaks, rest of kernel still works)
- **Disadvantages**: Lower performance

1.5 Virtual Machine also known as Hypervisor

A **software emulation of hardware – virtualisation** of underlying hardware (illusion of complete hardware). Allows us to run multiple OS on the same hardware at the same time and can also be used to debug OS.

- **Type 1 Hypervisor**:
  Provides individual VMs to guest OS's and runs directly on hardware (e.g. IBM VM/370)
- **Type 2 Hypervisor**:
  Runs in host OS, guest OS runs inside VM (e.g. VMware)

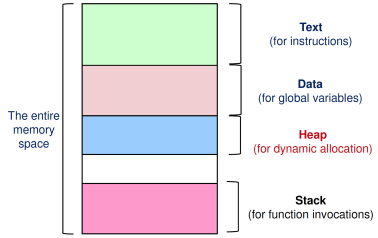




2 Process Abstraction

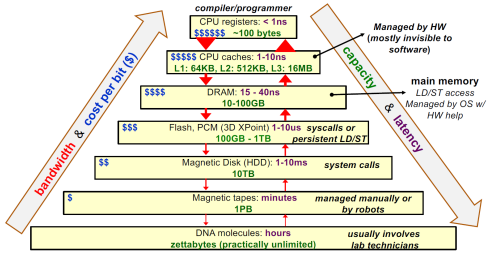
2.1 Process Abstraction

- **Process**: a dynamic abstraction for executing program
- Includes all information required to describe a running program (Memory context, hardware context, OS context)
- Information is used to **context switch** between running programs (to switch from program A to B, we save all information regarding program A then load information of program B)
- An executable binary consists of two major components: instructions and data
- During execution, more information:
  - **Memory context**: text, data, stack, heap
  - **Hardware context**: General Purpose Registers, Program Counter, Stack Pointer, Stack FP, ...
  - **OS context**: PID, Process state, Resources used, File Descriptor



Note: stack grow upwards in this case

2.2 Memory Hierarchy



2.2.1 Caches

Caches duplicate part of the memory for faster access (hardware optimization), are fast and invisible to software, usually split into instruction and data cache (optimized differently)

2.3 Function calls

2.3.1 Challenges faced when calling functions

- How to differentiate between global variables and local variables?
- Once we return from a function where do we return to? There is no static point of return from a function.

2.3.2 Control Flow and Data

Important steps (and their corresponding issues):

1. Setup the parameters (Setup "per-function data")
2. Transfer control to callee (Need some way to jump to function)
3. Setup local variables (How do we store this so there is no conflict?)
4. Store result if applicable (Need some way to capture return result)
5. Return to caller (How do we know where to return to?)

2.4 Stack Memory (LIFO data structure)

- New memory region to store information of a function invocation
- Described by a **stack frame**, containing: Return address of the caller (PC, old SP), Arguments for the function, Storage for local variables, Frame Pointer, Saved Registers
- **Stack Pointer** = The top of stack region (first unused location)
- **Frame Pointer** = points to a **fixed location in a stack frame** (platform dependent)
- **Saved Registers** = memory to temporarily hold General Purpose Registers (GPR) value during **register spilling*** since number of registers are very limited

* Register spilling occurs when GPRs are exhausted and have to be temporarily be stored on memory so they can be reused

2.4.1 Function Call Convention

Setup and teardown of the stack frame is usually done by the programmer or the compiler (we usually don't want the OS to touch the stack)

Stack Frame Setup:

- **Caller**: Pass parameters with registers and/or stack, Save Return PC on stack
- **Callee**: Save the old FP, SP, and registers used by callee, Allocate space for local vars on stack, adjust SP to point to new stack top

Stack Frame Teardown:

- **Callee**: Place return result on stack, Restore saved registers, FP, SP
- **Caller**: Utilize return result, Continues execution

2.5 Dynamically Allocated Memory

Using a separate **heap memory region** that allows for memory space to be acquired during execution time. **Grows in opposite direction from stack**: to allow for more flexibility since there are times we need more stack memory (e.g. recursion) and sometimes we need more heap memory

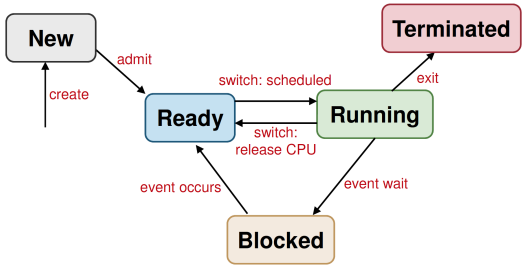
2.5.1 Behaviour of heap

1. Allocated only at runtime: size not known during compilation time → cannot be placed in Data region
2. No definite deallocation timing: can be explicitly freed by programmer or implicitly freed by garbage collector → cannot be placed on stack
3. Tends to create "holes" or "fragments" in memory since any portion of the heap can be deallocated at any time → need to run defragment programs to compact the heap

2.6 Process Identification & Process State

- Using process ID **(PID)**, a unique number among the processes.
- OS dependent issues: Are PID's reused? Are there reserved PID's? Does it limit max number of processes?

5 State Process Model:



1. **New**: process created, may still be initialising, not yet ready
2. **Ready**: process is admitted by scheduler waiting to run
3. **Running**: process being executed on CPU
4. **Blocked**: process waiting, can't execute till event is available
5. **Terminated**: process finished execution, may require OS cleanup

Transitions:

- nil -> New (Create)
- New -> Ready (Admit): Process ready to be scheduled
- Ready -> Running (Switch): Process selected to run
- Running -> Ready (Switch): Process gives up CPU voluntarily or preempted by scheduler
- Running -> Blocked (Event wait): e.g. syscall, waiting for I/O, ...
- Blocked -> Ready (Event occurs)

With 1 CPU:

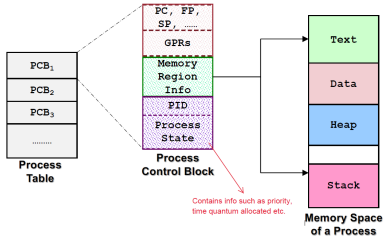
- ≤ 1 process in running state
- conceptually 1 transition at a time

With m CPUs:

- ≤ m processes in running state
- possibly parallel transitions
- each process may be in different states

2.7 Process Table & Process Control Block

- **PCB/Process Table Entry** = entire execution context for a process
- **Process Table** = kernel maintains PCB for all processes, stored as one table
- Stores a bunch of pointers that points to the correct places so that processes can be restored after context switch
- Issues: Scalability, Efficiency



2.8 System Calls

API to OS – provides a way of calling services in kernel, different from normal function call in that have to change from user mode -> kernel mode, more expensive due to OS overhead

2.8.1 APIs in different OS

- Unix variants mostly follows POSIX (portable operating interface) standards and typically have a small number of system calls: 100
- Windows uses Win API which typically have a huge number of system calls: 1000, and new system calls comes with each new version

2.8.2 C/C++ Sys calls

A C/C++ program can call the library version of system calls. A function wrapper has same name and parameters as the syscall. Alternatively, there is also a function adapter which is user-friendlier (less params, more flexible params value).

2.8.3 Sys Call Mechanism

1. User program invokes the library call (normal function call mechanism)
2. Library call places the system call number in a register
3. Library call executes TRAP to switch from user mode to kernel mode
4. In kernel mode, syscall handler is determined using the syscall number as index by **dispatcher**
5. System call handler is executed
6. System call handler ended, control return to library call, switch kernel -> user mode
7. Library call return to user program via normal function return mechanism

* syscall handlers are ptr to functions and are stored in a table to provide efficient lookup

2.9 Exception & Interrupt

Exception:

- **Synchronous**, occurring due to program execution
- Effect: have to execute an **exception handler**, similar to a forced function call
- Can explicitly call TRAP to purposely throw exception → typically used for debugging
- e.g. Overflow, underflow, Divide by 0, Illegal mem address

Interrupt:

- External events interrupting execution, usually hardware-related (timer, mouse moved, keyboard pressed)
- **Asynchronous**, occurring independent of program execution
- Effect: execution is suspended, have to execute **interrupt handler**
- Note that if a program is in the middle of executing an instruction, it would complete that instruction first before passing control to interrupt
- Interrupts can be stacked (timer interrupt can take over a mouse interrupt)

2.9.1 Handler Routine

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

* Exception uses software for handler routine whereas interrupts uses hardware
* Handler uses a different hardware context than user programs

3 Process Abstraction: Unix

3.1 Process in UNIX

Contains information on PID, Process state (running, sleeping, stopped, zombie), Parent PID (PPID), Cumulative CPU time etc.

3.2 Process Creation: Fork

Fork creates a new process known as **child process** which is a duplicate of the current executable image (same code, same address space, same PC etc.). Data is a copy of the parent (i.e. not shared). Only differs in PID, PPID and `fork()` return value (0 for child and > 0 for parent).

3.3 Common UNIX syscalls

- `int execl(char *path, char *arg0, ..., char *argN, NULL);` replaces current executing process image, does not return unless error. Will not exit on error. Only replaces Code and does not replace PID and other information
- `void exit(int status);` Status is returned to parent process and has value 0 to represent normal termination, else problematic. Does not return.
- `int wait(int *status);` returns the PID of terminated child (does not wait for grandchildren), status stores exit status. Blocking (does not block when process has no children). Cleans up remainder of child system resources. Note that `wait()` waits for child even if child calls `execl()`

3.4 Master Process

The common ancestor of all processes is `init` which is created in kernel at boot up time and has a PID = 1

3.5 Process Termination

Most system resources used by process are released on exit. Only PID, status and CPU time are not released (to facilitate clean up by parent when it calls `wait`). Generally, PCB is also not removed. On program termination, open files are flushed automatically.

Zombie process: child process terminates but parent did not call `wait` – child becomes zombie, can fill up processs table, created so that `wait()` system call can work properly, user command running in the background is considered a zombie

Orphaned Process: parent terminates before child – `init` becomes pseudo-parent, who will call `wait` on children

3.6 Implementing `fork()`

Simplified implementation:

1. Create address space of child process
2. Allocate p' = `new PID`
3. Create entry in Process Table
4. Copy kernel environment of parent process (e.g. priority for process scheduling)
5. Initialize child process context: `PID` = p', `PPID` = parent ID, zero CPU time
6. Copy memory regions, program, stack, data (further optimized by using copy-on-write)
7. Acquires shared resources (files, current working directory)
8. Initialize hardware context (Copy registers from parent)
9. Add child process to scheduler queue

* **Copy-on-write** only duplicates a memory location when it is written to (either by parent or child). When data is read only, memory is shared between parent and child

4 Process Scheduling

3 categories of **processing environment**: (1) **Batch Processing**: no user, no interaction, no need to be responsive, high system utilization is critical, (2) **Interactive**: with active user interacting, need to be responsive, consistent in response time, (3) **Real-time Processing**: deadline to meet, usually periodic process

4.1 Criteria for Scheduling Algorithms

- **Fairness**: fair share of CPU time, no starvation (prevalent in priorty based scheduling)
- **Balanceed**: all parts of the computing system should be utilised (shouldn't have a scenario where all processes are waiting on CPU or I/O)

4.2 Types of scheduling policies

- **Non-preemptive (cooperative)** – a process stays scheduled until it blocks/gives up the CPU voluntarily
- **Preemptive**: CPU can be taken from running process at ANY time. A process is given a fixed time quota to run (possible to block or yield early), at the end of the time quota, the running process is suspended and another process gets picked.

4.3 Scheduling a process

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

4.4 Scheduling for Batch Processing

Criteria:

- **Turnaround time**: Total time taken (time when it end - time when it arrived), related to waiting time, waiting time = Turnaround time - Time spent doing work (CPU + I/O)
- **Throughput**: Rate of task completion
- **CPU Utilisation**: % of time when CPU is working on a task, higher the better

4.4.1 First-Come First-Served (FCFS)

- Tasks are stored on a FIFO queue based on arrival time. Pick the head of queue to run until task is done OR task is blocked. Blocked task removed from queue, when it is ready again, placed at back of queue like a newly arrived task.
- **Guaranteed** to have no starvation: # of tasks in front of task X in FIFO is always decreasing -> task X will get its chance eventually.
- Shortcoming: **Convoy Effect** – due to non-preemptiveness, one slow process (CPU intensive) slows down the performance of the entire set of processes.

4.4.2 Shortest Job First (SJF)

- Select the task that needs the shortest amount of CPU time, thus **guarantees** smallest average waiting time (in turn decreasing average turnaround time).
- **Shortcomings**: Need to know total CPU time for a task in advance (have to guess if not available), **starvation is possible** (biased towards short jobs, long jobs may never run)
- Predicting CPU Time, common approach (**Exponential Average**): $\text{Predicted}_{n+1} = \alpha\text{Actual}_n + (1-\alpha)\text{Predicted}_n$, where α represents the significance of immediate past values, and $(1-\alpha)$ represents the significance of past history (typical α value is 1/2)
- Higher α cause predicted value to fluctuate closer to the last value, while lower α allow a predicted value that is closer to the historical value

4.4.3 Shortest Remaining Time (SRT)

- Select job with shortest remaining (or expected) time.
- Variation of SJF that is **preemptive** and uses remaining time.
- New job with shorter remaining time can preempt currently running job
- Provide good service for short jobs even when they arrive late

4.5 Scheduling for Interactive Systems

- Criteria:
  - **Response time**: Time between request and response by system (time when it first start running - time when it first reach)
  - **Predictability**: Lesser variation in response time → better predictability
- **Preemptive** scheduling algorithms are used to ensure good response time, thus scheduler needs to run periodically.
- **Timer interrupt** = interrupt that goes off periodically based on hardware clock, OS ensures timer interrupt is not intercepted by other program/interrupt
- Timer interrupt handler **invokes OS scheduler**.
- **Interval of Timer Interrupt (ITI)** typically 1-10ms, scheduler is triggered every ITI but may not actually run
- **Time Quantum** = execution duration given to a process, can be constant/variable, must be multiple of ITI (commonly 5-100ms)

4.5.1 Round Robin (RR)

- Tasks stored in a FIFO queue, pick task from head of queue until time quantum elapsed OR task gives up CPU voluntarily OR task blocks
- Basically a **preemptive version of FCFS**
- **Response time guarantee**: given n tasks and quantum q, time before a task get CPU is bounded by $(n-1)q$, note that bounded response time ≠ responsive
- Choice of time quantum: big = better CPU utilization, longer waiting time; small = bigger overhead (worse CPU utilization) but shorter waiting time

4.5.2 Priority Scheduling

- Assign a **priority** value to all tasks, select task with highest priority value.
- **Preemptive**: higher priority process can preempt running process with lower priority
- **Non-preemptive**: late coming high priority process has to wait for next round of scheduling
- **Shortcomings**: Low priority process **can starve**, worse in preemptive variant
- **Possible solutions**: Decrease the prioty of currently running process after every time quantum, Give each process a minimum time quantum
- Generally hard to guarantee/control exact amount of CPU time given to a process
- **Priority Inversion**: 3 processes, priorities Hi, Mi, Lo. L locks resource, M pre-empts L, A arrives and tries to lock same resource as L. Then M continues executing although H has higher priority.

4.5.3 Multi-level Feedback Queue (MLFQ)

- Designed to solve the issue of scheduling without perfect knowledge (e.g. no need to guess and estimate running time)
- Adaptive, minimising both response time for IO-bound and turnaround time for CPU-bound
- Rules:
  - Priority(A) > Priority(B) -> A runs (after B time quantum is up)
  - Priority(A) == Priority(B) -> A and B runs in RR
  - New job -> highest priority
  - If a job fully utilised its time slice -> priority reduced
  - If a job gives up/blocks before it finishes the time slice -> priority retained
- **Shortcomings**: (1) **Starvation** – if there are too many interactive jobs, long-running jobs will starve, (2) gaming the scheduler by running for 99% of time quantum, then relinquish the CPU (**hogging CPU**), (3) a program may change its behaviour CPU-bound -> interactive (**not given priority when it is needed**)
- Possible solution:
  - **Priority boost**: Periodically move all processes to highest priority. guaranteeing no starvation as highest priority -> RR, also allows processes with different behaviour phases to be treated correctly even after it reach lowest priority
  - **Cumulative Time Quantum**: Once a job uses up its time allotment at a given level, its priority is reduced

4.5.4 Lottery Scheduling

- Give out "lottery tickets" to processes. When a scheduling decision is needed, a ticket is chosen randomly among eligible tickets.
- In the long run, a process holding X% of tickets can win X% of the lottery held and use the resource X% of the time.
- **Responsive**: newly created process can participate in next lottery, every process gets to run in every round (waiting time is bounded by $2 \times N \times TQ$)
- **Good level of control**: A process can be given lottery tickets to be distributed to its child process, an important process can be given more lottery tickets, each resource can have its own set of tickets
- Simple implementation

5 Inter-Process Communication

- 2 common IPC mechanisms: **Shared-Memory** & **Message Passing**
- 2 Unix-specific IPC mechanisms: **Pipe** and **Signal**

5.1 Shared-Memory

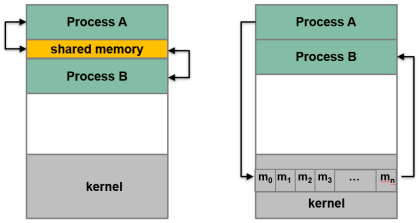
- Implicit communication through reads/writes to shared variables
- General idea: Process p_1 creates a shared memory region M in **user space**, process p_2 attaches m to its own memory space. p_1 and p_2 can now communicate using memory region M. Values written to M is visible to all processes that share M.
- Shared memory region is identified by a number, memory address is only generated when a process attaches to the identified region
- Shared memory region can be accessed by any process (set permission bits to 0666)
- OS involved only in creating and attaching shared memory region
- **Advantages**: Efficient (only initial steps involves OS), Ease of use (information of any type or size can be written easily)
- **Disadvantages**: Synchronisation (shared resource -> need to synchronise access), Implementation is usually harder, Limited to a single machine (software abstractions can be supported in distributed systems but less efficient)
- In Unix: (1) create/locate shared memory region M, (2) Attach M to process memory space, (3) Read/Write M, (4) Detach M from memory space after use, (5) Destroy M (only 1 process does this, can only destroy if M is not attached)

5.2 Message Passing

- Explicit communication through exchange of messages
- General idea: process p_1 prepares a message M and send it to process p_2, p_2 receives the message M. Sending and receiving messages are generally provided using syscalls (OS mediates it → easier to implement, safer but slower since involves OS overhead)
- Message has to be stored in **kernel memory space**, every send/receive operation is a syscall
- **Advantages**: Portable (can be easily implemented on different processing environment), Easier synchronisation (using synchronous primitive)
- **Disadvantages**: Inefficient (usually requiring OS intervention), Harder to use (message usually limited in size and/or format)

5.2.1 Naming (how to identify the other party in the comm):

- **Direct Communication**
  - Sender/receiver **explicitly** name the other party (using PID)
  - Characteristics: 1 link/pair of communicating processes, need to know the identity of the other party
- **Indirect Communication**
  - Message are sent to/received (copied) from message storage (aka mailbox or port)
  - Characteristic: 1 mailbox can be shared among a number of processes
    - Introduces a problem of who receives the message? There are many variations used: one-to-one, one-to-many, many-to-many, many-to-one
    - Since mailbox is in kernel space, OS takes care of synchronisation of message reading and writing



Shared memory Message passing

5.2.2 Synchronisation (behaviour of the sending/receiving ops)

- **Blocking primitives** (synchronous): sender/receiver is blocked until message is received/has arrived
- **Non-blocking Primitive** (asynchronous): sender resume operation immediately, receiver either receive message if available or proceeds empty handed but doesn't block
- Typically `receive()` is blocking → we look at the behaviour of sender to determine whether asynchronous or synchronous

5.2.3 Asynchronous Message Passing

- Assumes that `receive()` is blocking and sender is non-blocking
- System buffers the message (Async message passing **MUST** have a buffer)
- Drawbacks:
  - Too much freedom for programmer, program becomes complex
  - Finite buffer size means system is not truly asynchronous → sender waits when buffer is full
- Buffer: Under OS control → Synchronization burden placed on OS; Decouples sender and receiver → less sensitive to variations in execution → don't have to wait on each other unnecessarily.

5.2.4 Synchronous Message Passing

- Assumes sender is blocking and receiver is also blocking
- **No intermediate buffers needed**
- **Pros**:
  - Applicable **beyond single machine**
  - **Portable**: Easily implemented on many platforms; Parties on different platforms can communicate
  - **Easier synchronization**: Implicit, defined by blocking semantics; No shared memory region (message is copied); Achieves both communication and synchronization simultaneously
- **Cons**:
  - **Inefficient**: Requires OS intervention on every send/receive
  - **Harder to use**: Requires packing/unpacking data into supported format

5.3 Unix Pipes

- A communication channel with 2 ends, for reading and writing.
- A pipe can be shared between 2 processes (producer-consumer)
- Behaviour: like an anonymous file, **FIFO** (in-order access)
- Pipe functions as **circular bounded byte buffer with implicit synchronisation**: writers wait when buffer full, readers wait when buffer empty (**Async** sending)
- Variants: Multiple readers/writers, half-duplex (unidirectional) or full-duplex (bidirectional)
- `int pipe(int fd[]);` returns 0 = success. `fd[0]` reading end, `fd[1]` writing end

5.4 Unix Signal

- An **async** notification regarding an event sent to a process/thread (OS → user program OR user program → user program)
- Recipient of signal handle by a default set of handlers OR user-supplied handler (only applicable to some signals)
- Common signals in Unix: SIGKILL, SIGSTOP, SIGCONT, etc.

6 Process Alternative – Threads

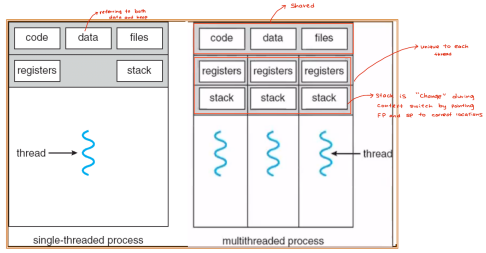
- Motivation:
  - Process is expensive: under `fork()` model – duplicate memory space and process context, context switch requires saving/restoration of process information
  - Hard for independent processes to communicate with each other: independent memory space – no easy way to pass information, requires Inter-Process Communication (IPC)
- A **traditional process has a single thread of control** – only one instruction of the whole program is executing at any one time. Instead, we add more threads of control such that multiple parts of the program are executing simultaneously conceptually.
- Essential with multi-core architectures
  - **Data parallelism**: different threads doing same task on different data (typically GPU does this)
  - **Task parallelism**: different threads do different tasks on same/different data

6.1 Process and Thread

- A single process can have multiple threads
- Threads in the same process shares:
  - **Memory Context**: Text, data, heap
  - **OS Context** (PID, other resources like files, etc.)
- Threads have unique:
  - Identification (Thread id)
  - Registers (general purpose & special, including PC)
  - Stack

6.1.1 Context Switching

- **Process** context switch involves: OS Context, Hardware Context, Memory Context
- **Thread** switch within the same process involves: Hardware context (registers, "stack" – actually just changing FP and SP)
- Can think of threads as a lighter weight process



6.2 Benefits

- **Economy**: requires much less resources
- **Resource sharing**: Threads shares most resources, no need for additional information passing mechanism
- **Responsiveness**: multithreaded programs can appear much more responsive (one thread can deal with I/O while another is processing data)
- **Scalability**: Multithreaded program can take advantage of multiple CPUs

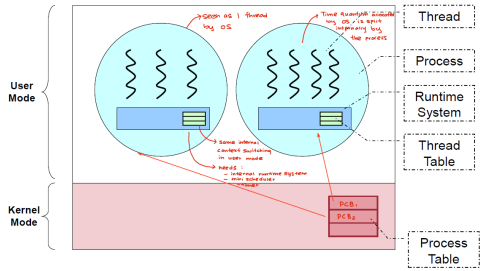
* Note that using threads on a SINGLE CORE does not actually speed up execution time and might instead make it run slower due to overhead

6.3 Problems

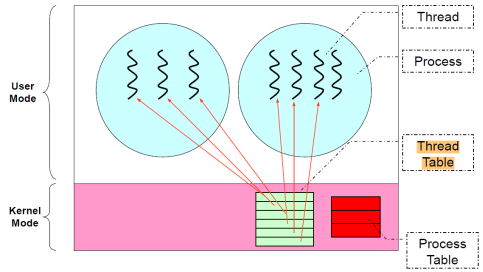
- **Synchronization around shared memory is difficult to handle** – All memory shared except the stack
- **System call concurrency** – have to guarantee correctness and determine the correct behaviour of threads ran in parallel
- **Process behaviour** – impact on process operations, e.g. does `fork()` duplicate threads? If single thread executes `exit()`, how abut the whole process? If a single thread calls `exec()`, how about other threads etc.

6.4 Thread Models

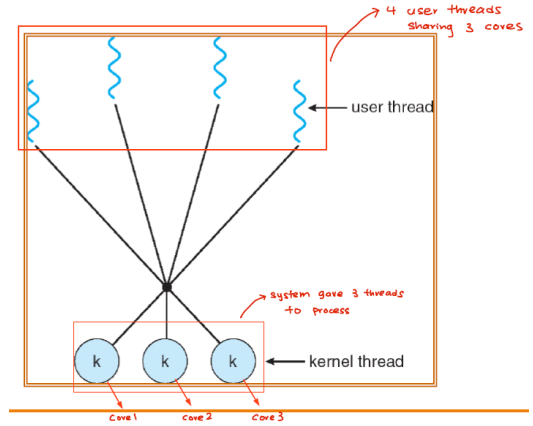
- **User Thread**
  - Implemented as a **user library**, a runtime system in the process handles thread operations (user or library has to handle multiplexing of threads)
  - Kernel is not aware of threads in the process.
  - **Advantages**: Multithreaded program on ANY OS (ensures portability and backward compatibility), thread operations are just library calls, more configurable and flexible (such as customised thread scheduling policy)
  - **Disadvantages**: OS is not aware of threads, scheduling is performed at process level. One thread blocked → process blocked → all threads blocked, cannot exploit multiple CPUs (1 solution is to do Async I/O operation)



- **Kernel Thread**
  - Implemented in the **OS**, thread operation handled as **system calls**.
  - Allows for Thread-level scheduling, basically treat kernel threads as its own process
  - Kernel may make use of threads for its own execution
  - **Advantages**: Kernel can schedule on thread level (threads on same process can run simultaneously on multiple CPUs → 1 thread can be blocked and the rest can still run)
  - **Disadvantages**: Thread operation is a syscall (slower and more resource intensive), generally less flexible (used by all multithreaded programs – many features: expensive, overkill for simple program, few features: not flexible enough for some)



- **Hybrid Thread Model**:
  - Have **both kernel and user threads**, OS schedule on kernel threads only, user thread can bind to a kernel thread.
  - Great flexibility (can limit concurrency of any process/user)



6.5 Threads on Modern Processor (Intel Hyperthreading)

- Threads started off as software mechanism: Userspace lib -> OS aware mechanism
- Hardware support on modern processors, supplying multiple sets of registers to **allow threads to run natively and parallelly on the same core**: Simultaneous Multi-Threading (SMT)

6.6 POSIX Threads: `pthread`

- Standard by IEEE, defining API and behaviour.
- Implementation is not specified → pthread can be implemented as user thread (windows) or kernel thread (Linux)
- `int pthread_create(pthread_t* tidCreated, const pthread_attr_t* threadAttributes, void* (*startRoutine) (void*), void* argForStartRoutine);`
  - `tidCreated`: Thread ID for the created thread
  - `threadAttributes`: Control the behaviour of new thread
  - `startRoutine`: Function pointer to the function to be executed by thread
  - `argForStartRoutine`: Arguments for the startRoutine function
- `int pthread_exit(void* exitValue)`
  - if `pthread_exit()` is not called, a pthread will terminate automatically at the end of startRoutine
  - if `"return XYZ"` is used, the "XYZ" is captured as exit value
- `int pthread_join(pthread_t threadID, void **status);`
  - threadID: TID of the pthread to wait for
  - status: Exit value returned by the pthread
- except for `pthread_exit`, return 0 = success

7 Synchronization

7.1 Race Condition

- When 2/more processes **execute concurrently** in interleaving fashion AND **share a modifiable resource** resulting in non-deterministic execution.
- **Solution**: designate code segment with race condition as **critical section** where at any point in time only 1 process can execute. Ideally, process in critical section should be done atomically (you either do it or you don't, cannot be done "halfway")

7.2 Critical Section

Properties of correct implementation:

- **Mutual Exclusion**: if a process is executing in critical section, all other processes are prevented from entering it (**most critical property**, if this fails we disregard the rest)
- **Progress**: If no process is in critical section, one of the waiting processes should be granted access
- **Bounded Wait**: After a process p_i requests to enter the critical section, $\exists$ an upper-bound of number of times other processes can enter the critical section before p_i
- **Independence**: process not executing in critical section should never block other processes

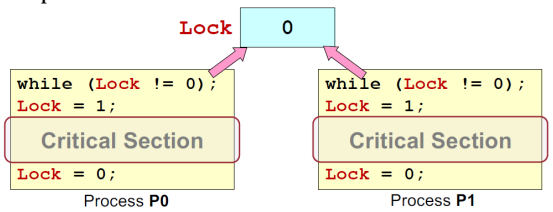
* Note that **Progress and Independence are related**. Not possible to have progress but no independence but possible to have independence but no progress (Attempt 3)

7.2.1 Symptoms of incorrect synchronisation

- usually due to lack of mutual exclusion
- **Deadlock**: all processes blocked → no progress (e.g. 2 processes using message passing, both processes are using blocking receive and waiting to receive msg from the other). Fulfils the following conditions:
  1. **Mutual Exlusion**: Each resource is either assigned to exactly one process or to none
  2. **Hold-and-wait**: Processes holding resources can ask for more resources
  3. **No-preemption**: Cannot forcibly take away resources previously given
  4. **Circular wait**: Must have a circular list of two or more processes, each waiting for a resource held by the next
- **Livelock**: processes keep changing state to avoid deadlock and make no other progress, typically processes are not blocked
- **Starvation**: some processes are blocked forever

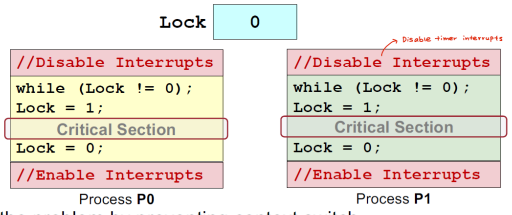
7.3 Implementations of Critical Section

7.3.1 Incorrect Attempt 1



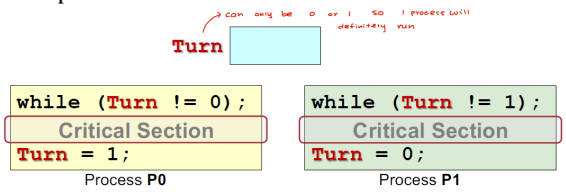
- Makes intuitive sense ☺
  - But it doesn't work properly ☹
- It violates the "Mutual Exclusion" requirement!
  - How? if context switch happens before "Lock = 1", both processes are able to enter the critical section

7.3.2 Attempt 1 Incorrect Fix



- Solves the problem by preventing context switch
- However:
  - Buggy critical section may stall the WHOLE system → interrupts cannot stop the process if anything happens
  - Busy waiting
  - Requires permission to disable/enable interrupts → not readily available in high level programming languages

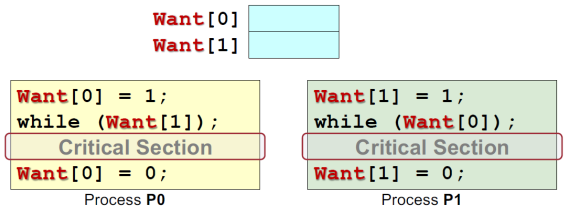
7.3.3 Incorrect Attempt 2



- **Assumption:**
  - P0 and P1 executes the above in loop
  - Take turn to enter **critical section**
- **Problems:**
  - Starvation:
    - E.g. If P0 never enters CS, P1 starve
  - Violates the **independence** property!

→ also violates progress since P0 wants but cannot get

7.3.4 Incorrect Attempt 3



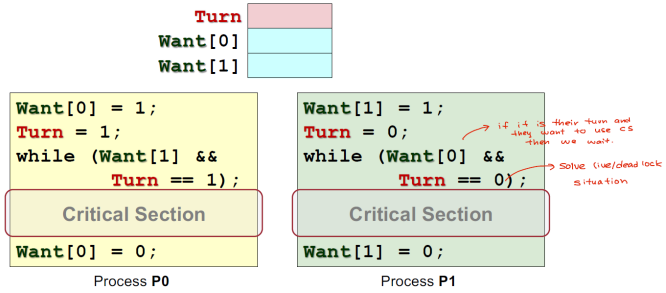
Solves the independence problem

- If P0 or P1 is not around, another process can still enter CS

Problem: → can also be livelock since they are both "doing something" technically

- Deadlock! Try identify the execution sequence that causes deadlock → both wants CS and waiting on each other to backup

7.3.5 Peterson's Algorithms



if it is their turn and they want to use CS then we wait.

Solve (live/dead lock situation

* Note that the order of `Want[X] = 1` followed by `Turn = X` is important!

7.3.6 Peterson's Algorithm Disadvantages

- **Busy Waiting**: wasteful use of processing power due to having while-loop instead of going into blocked state
- **Low level**: higher-level programming construct desirable to simplify mutex, make it less error prone and simpler to implement
- **Not general**: general synchronisation mechanism is desirable, not just mutex (limited to 2 processes)

7.3.7 Test-and-set: an atomic instruction

- Uses machine instructions to aid synchronization.
- Syntax: `TestAndSet Register, MemoryLocation`
- Load the current content at `MemoryLocation` into `Register`, Stores a 1 into `MemoryLocation`
- TestAndSet is performed as a single atomic machine operation and is **atomic even in multi-core system**
- Disadvantage: busy waiting (spin lock) – wasteful use of processing power; Does not guarantee bounded-wait (unless scheduling is fair)

7.3.8 Semaphore

A generalised synchronisation mechanism, providing a way to block a number of processes and a way to unblock one/more sleeping process(es)

- `wait(S)`: if $S > 0$, decrement. If $S \leq 0$, go to sleep
- `signal(S)`: increment S, wakes up one sleeping process (usually the first process that was blocked on semaphore (FIFO)) (if any), operation never blocks
- done with the help of OS to ensure atomicity
- no busy waiting (processes are blocked while waiting)
- solves all known synchronization problems

Properties

- Given $S_{initial} \geq 0$, where #signal(S) = number of `signal()` executed, #wait(S) = number of `wait()` completed (successfully decremented S)
- **Invariant**: $S_{current} = S_{Initial} +$ #signal(S) – #wait(S)

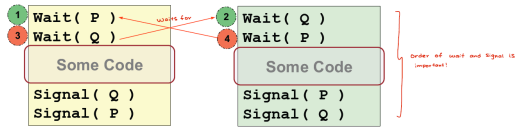
7.3.9 Types of Semaphores

- General Semaphores (Counting Semaphores): $S \geq 0$, useful for safe-distancing problem (limit number of processes running)
- Binary Semaphore (mutex): $S = 0$ or 1, used to create mutual exclusion zone (critical section), S is upper bounded to 1 (if `signal()` called when value is 1 → undefined behaviour)
- General Semaphores can be mimicked by binary semaphores

7.3.10 Incorrect use of semaphore

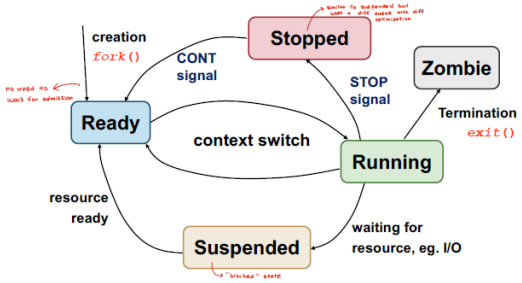
- Deadlock is still possible with incorrect use of semaphore
- Example: Semaphore **P = 1**, **Q = 1** initially

→ 2 semaphores



8 Miscellaneous

Process State Diagram:



Priority Inversion

- Consider the scenario:
  - Priority: {H = 1, M=3, L= 5} (1 is highest)
  - L starts and locks a resource (e.g., a file)
  - M arrives and preempts L (higher priority)
    - L is unable to unlock the file
  - H arrives and need the same resource as L
    - but the resource is locked! And L can't run because of M
  - → M continues executes even if Task H has higher priority
  - → M can starve H for as long as it wants!
- Known as **Priority Inversion**:
  - Lower priority task effectively preempts higher priority task

* Situation requires at least 3 process with mutex, using the system

Priority Inversion: Solution

Priority inheritance

- Temporarily increase priority of L to H
  - Until it unlocks the lock
- Low-priority job *inherits* the priority of the higher priority process
  - Happens when the high-priority process requests a lock held by low-priority process
  - Priority restored upon successful unlock()

9 Office hours (Pre-Midterms)

9.1 OS vs Kernel

Kernel:
- Essence of OS, manages and protects CPU, memory and other key resources
- doesn't use SYSCALLs, normal libraries and normal I/O

OS:
- Just a program with special features: deals with hardware issues, provides SYSCALLs, handles interrupts and device drivers
- User-friendly packaging of kernel: Includes lots of non-essential functionalities (GUI), superset of kernel

9.2 Kernel mode

Kernel mode is the **most privileged mode of execution** in hardware which is mainly ran by the kernel. Other parts of OS use less privileged mode of execution.

9.2.1 Kernel mode vs Super user

- Kernel mode is a mode of execution in hardware; when in this mode, you can execute any instruction with its fullest semantics. These privileges are reserved for the critical part of the kernel.
- Super-user privileges allow a certain user to manage users, files, and processes in an unrestricted manner. This removes certain OS-level restrictions, but not the hardware-level restrictions.
- Super user (e.g. sudo) ≠ kernel mode

9.3 Return value

How does the return value gets accessed after the stackframe is torn down?
- Teardown does NOT destroy anything! Values from the function that returned still stays on stack frame until it is overwritten by other stack frames
- Teardown just changes the value of SP

9.4 Stacks and Frames Exercise

System X has 2 CPUs, each CPU has 4 cores. System X currently runs 100 processes. Assume each process calls a recursive function `factoriel(n)` with n=20.

- How many stacks are there in the system? 100 (one per process)
- How many stack pointers are there in the system? 100 (one per stack)
- How many stack pointer registers are in the system? 8 (one per core)
- How many stack frames are there in the system? Up to 100x20 = 2000
- How many frame pointers are there in the system? 100 (one per stack)
- How many frame pointers registers are there in the system? 8 (one per core)

9.5 Heap

Why dynamically allocated memory cannot use "Data" region?
- Size of data region is determined at compile time → fixed at runtime, dynamically allocated memory can be of any arbitrary size and is unknown at compile time
- Variables in data region lives for the duration of the program whereas dynamically allocated variables can live for as much time as the user wants (until user calls `free()`)

9.6 Interrupt Handling Steps

1. PUSH(status register) -- **hardware**
2. PUSH (PC) -- **hardware**
3. Disable interrupts in the status register -- **hardware**
4. Read Interrupt Vector Table entry (ISR1) -- **hardware**
5. PC ← ISR1 – **hardware**
6. ISR1 execution -- **software**

ISR 1 execution

1. PUSH(registers to be modified, if any) -- software
2. Optionally enable interrupts (all or some) -- software
3. Do the necessary stuff
   - Allowed to modify the pushed registers
4. POP (modified registers, if any)
5. IRET – specialized interrupt return instruction. Semantics:
   - POP(status register) – enables interrupts -- hardware
   - POP(PC) – returns to the main code (next instruction) -- hardware
   - Step 1 of the next instruction after load (PC=27115)

9.7 Traps

Traps is an exception that is intentionally set-up (software-interrupt) and is used primarily as a means to enter kernel mode (e.g. for debugging purpose)

9.7.1 Traps vs Exceptions vs Traps

They are all related and have very similar mechanics of execution:
- Must push PC on the stack
- Divert control to a routine
  - Interrupts: hardware directly loads the address of the routine into PC from IVT
  - Exceptions: the address of the dispatcher is loaded, rather than the final routine; dipatche finds the right routine in software
  - Traps are exceptions that are intentionally set up
- get return PC from the stack and continue

9.8 How does fork bomb works

If a user runs: `while (1) { fork(); }`, it doesn't use any stack space so why would it cause problem?
- It is correct that this code does not use any stack space but since it is creating a new process and duplicating memory, there is some memory overhead (PCB, text, data, stack, heap etc.) which causes the system as a whole to run out of memory

9.9 Preemptive vs Non-preemptive

- Preemptive schedulers are needed to guarantee the responsiveness of the system
- Non-preemptive schedulers are used in simpler systems that do not care about responsiveness → overhead is lower

9.10 Who schedule the scheduler?

- Scheduler doesn't have to be scheduled
- In non-preemptive systems, when a process blocks/exits by calling a syscall, the scheduler is executed as well.
- For preemptive systems, the scheduler is called regularly by a timer interrupt.

9.11 Scheduling Summary

Algo	Pre-emptive?	Starvation?	Responsive?	Comments
FCFS	✗	✗	✗	Simple, Easy to underutilize CPU
SJF	✗	✓	✗	Theocratically optimal but hard to predict how long a task will take + short jobs that arrive later starves
SRT	✓	✓	✗	SJF but fix short job arrive late issue
RR	✓	✗	✗	Pre-emptive FCFS
Priority	✓/✗	✓	✓	Better allocation, but lower priority processes starves, inversion problem
MLFQ	✓	✓	✓	Balanced solution that profiles whether a process is CPU/IO bound
Lottery	✓	✗	✓	Simple theocratical idea, not common

9.12 Are exceptions signals?

Are exceptions a type of signal?
- No, these are separate concepts, but…
- When an exception occurs, that information may be delivered to a process via a signal.
- Example
  - Your program contains an instruction that divides an integer by 0.
  - This instruction is executed by the CPU, which generates an **exception**.
  - The OS receives this exception and knows which process is executing.
  - The OS sends a **SIGFPE signal** to the offending process.
  - The process has to handle this signal or will likely be terminated.

9.13 SRT Timer Interrupts?

Does Shortest Remaining Time scheduling need timer interrupts? Why?
- No, and a good question!
- SRTF only takes actions when a process arrives / leaves / blocks / releases the CPU, not at regular time intervals.
- All of the above actions are done through system calls, so SRTF can just run when such a syscall is executed.

9.14 What makes lottery free of starvation?

What makes the lottery scheduling free of starvation is:
- The concept of rounds; every process gets to participate in lottery in the next round
- Drawing without replacement in each round: this ensures that in every round every ticket is given a time quantum for execution

9.15 Indirect Message Passing

When do we use it?
- often used when it doesn't matter who received which message
- e.g. multiple clients send requests (messages) to a mailbox → mailbox buffers all the messages → multiple servers receive request from same mailbox
- No race conditions due to no shared data (data is copied from sender to buffer and buffer to reader) and also because OS takes care of integrity of its own message buffer
- Mailbox are cleaned up by the OS if process that created it crashes

9.16 Kernel vs User space

- Shared memory could use kernel space, but it's limited
- That's why most shared memory regions are in the user space of the participating processes/threads
- The processes asked for shared memory, so let them do what they want
- In message passing, the OS is responsible for delivering messages and must keep them in its memory

9.17 Kernel threads vs Actual processes

In the kernel threading model, all scheduling is done on threads instead of process? What is the point of the *process* table in this case?
- Excellent question.
- Processes and threads are different entities and therefore have different metadata to store.
  - For process, e.g., process's status, signal, and size information, as well as per-process data that is shared by the kernel threads.
  - For threads, e.g., scheduling, priority, state, CPU usage information of a kernel thread.

9.18 User thread stack

- If OS not aware of user threads, where is the stack and register info stored?
- Program that manages the user thread must manage that information (likely to use heap memory to store)
- On top of memory management, these programs have to handle context switching, register spilling, reloading etc. on their own too

9.19 Critical section properties

Regarding the 4 properties, what's the difference between Progress and Independence? They seem quite similar. Attempt 3 shows that it is possible to have independence but not progress. Is it possible to have progress but not independence?
- Excellent question and observations.
- Progress and independence are related (the *OS Concepts* book treats them as one)
- It is not possible to have progress but no independence (vice versa can)
  - because an independent process may block others and prevent their progress

Progress:
- If no process is in a critical section, one of the waiting processes should be granted access.

Independence:
- Process **not** executing in critical section should never block other process.

9.20 Uses of semaphores

1. **Safe distancing problem**: allow only X number of processes to be running at the same time (General semaphore)
2. **Mutex**: acts as a lock, no starvation under fair scheduling, no deadlock if used correctly, guarantees mutual exclusion (Binary semaphore)
3. **Synchronization tool**: Execute B in P_1 only after A executed in P_0 (semaphore init to 0)
4. **Barrier**: Only continue execution after all processes have reached barrier

10 Classical Synchronization Problems

10.1 Revision: Use of Semaphores

Critical Section

```
BinarySemapore mutex = 1;

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

Safe-distancing problem

```
Semaphore sem = N;

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

General synchronization

```
Semaphore sem = 0;

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

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

10.2 Producer-Consumer Problem

- Processes share a bounded buffer of size K
- Producers produce items to insert in buffer when buffer is not full (< K items)
- Consumers remove items from buffer when buffer is not empty (> 0 items)

10.2.1 Busy waiting solution

```
while (TRUE) {
    Produce Item;
    while(!canProduce);
    wait( mutex );
    if (count < K) {
        buffer[in] = item;
        in = (in+1) % K;
        count++;
        canConsume = TRUE;
    } else
        canProduce = FALSE;
    signal( mutex );
}
```

Producer Process

```
while (TRUE) {
    while (!canConsume);
    wait( mutex );
    if (count > 0) {
        item = buffer[out];
        out = (out+1) % K;
        count--;
        canProduce = TRUE;
    } else
        canConsume = FALSE;
    signal( mutex );
    Consume Item;
}
```

Consumer Process

in: points to beginning of buffer, init to 0
out: points to end of buffer, init to 0
count: # of items in buffer
canProduce: init to 1
canConsume: init to 0
mutex: Semaphore init to 1

Evaluation:
Correctly solves the problem but busy-waiting is used! e.g. 10 consumer, 1 producer, consumer will take up a lot of CPU power doing nothing and negatively impact producer

10.2.2 Blocking Version

```
while (TRUE) {
    Produce Item;

    wait( notFull );
    wait( mutex );
    buffer[in] = item;
    in = (in+1) % K;
    signal( mutex );
    signal( notEmpty );
}
```

Producer Process

```
while (TRUE) {

    wait( notEmpty );
    wait( mutex );
    item = buffer[out];
    out = (out+1) % K;
    signal( mutex );
    signal( notFull );

    Consume Item;
}
```

Consumer Process

`wait(notFull)`: Forces producer to sleep
`wait(notEmpty)`: Forces consumers to sleep
`signal(notFull)`: consumer wakes up producer
`signal(notEmpty)`: producer wakes up consumer
* `count` no longer needed as counting semaphores `notFull` and `notEmpty` acts as count

Initial Values:
in=out=0
mutex=$S(1)$, notFull=$S(k)$, notEmpty=$S(0)$

Evaluation:
Correct solves the problem and "unwanted" producer/consumer will go to sleep on respective semaphore.

10.2.3 Message Passing

Producer

```
while (TRUE) {
    Produce Item;
    mQueue.send(Item);
}
```

Consumer

```
while (TRUE) {
    mQueue.receive(Item)
    Consume Item;
}
```

Elegant way of solving producer/consumer problem using OS:
- `MessageQueue` acts as buffer with K as max elements in buffer
- OS blocks consumers when there are no messages
- OS blocks producer when buffer is full
- OS handles all synchronisation problems
- Asynchronous unless buffer full → Concurrency

10.3 Readers/Writers Problem

- Processes shares a data structure **D**:
  - Reader: Retrieves information from **D**
  - Writer: Modifies information in **D**
- Writer must have exclusive access to **D**, i.e. Writer cannot be with another write nor can it be with readers
- Reader can access with other readers
- Many solutions to this problem!

10.3.1 Simple Version

```
while (TRUE) {
    wait( roomEmpty );

    Modifies data

    signal( roomEmpty );
}
```

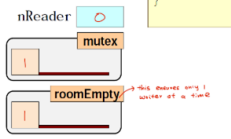
Writer

```
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`

Special cases:
- 1st reader will have to wait on `roomEmpty` to ensure that there are no more writers
- Last reader will have to signal to `roomEmpty` to let writers know that it is safe to write
- Subsequent readers after 1st reader will wait on mutex since 1st reader is holding mutex

Evaluation:
Works but might leader to starvation for the writer! e.g. After the writer leaves, there is many readers that want to read → `roomEmpty` will never be 1 and writer can't write any more

10.4 Dining Philosophers

- 5 philosophers seated around a table
  - 5 single chopsticks places between each pair of philosophers
  - Need to acquire both chopsticks before he can eat
- Want a deadlock-free and starve-free solution

10.4.1 Attempt 1

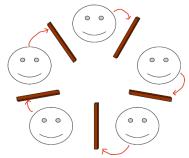
```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)
Semaphore chopsticks[5]={1,1,1,1,1};

//For philosopher i
while (TRUE){

    Think( );
    //hungry, need food!
    wait(chopsticks[LEFT] );
    wait(chopsticks[RIGHT] );

    Eat( );

    signal(chopsticks[LEFT] );
    signal(chopsticks[RIGHT] );
}
```



Can you figure out the problem?

Still causes deadlock when all philosophers simultaneously take the left chopstick → all philosophers cannot proceed since they all wait for right chopsticks

Fix Attempt:
Make each philosopher put down left chopstick if right chopstick cannot be acquired
- No deadlock but livelock!
- All philosophers take their left chopstick, then put it down, then take it up, put it down, ...

10.4.2 Attempt 2

```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)

//For philosopher i
while (TRUE){
    Think( );

    wait( mutex );

    takeChpStck( LEFT );
    takeChpStck( RIGHT );

    Eat( );

    putChpStck( LEFT );
    putChpStck( RIGHT );

    signal( mutex );
}
```

- Works but only 1 philosopher can eat at a time.
- Too trivial solution, kills concurrency

10.4.3 Attempt 3

```
#define N 5
#define LEFT i
#define RIGHT ((i+1) % N)

//For philosopher i
while (TRUE){
    Think( );

    wait( mutex );
    takeChpStck( LEFT );
    takeChpStck( RIGHT );
    signal( mutex );

    Eat( );

    wait( mutex );
    putChpStck( LEFT );
    putChpStck( RIGHT );
    signal( mutex );
}
```

- Deadlock! Imagine the scenario where a philosopher picked up chopsticks then before he put it down another philosopher comes and tries to pick up chopstick and since he cannot pick up he will hold onto mutex lock and not allow the philosopher to put down his chopsticks

10.4.4 Limited Eater

```
Semaphore seats = S(4);
//initialization

void philosopher( int i ){

    while (TRUE){
        Think( );
        wait( seats );
        takeChpStck( LEFT );
        takeChpStck( RIGHT );
        Eat( );
        putChpStck( LEFT );
        putChpStck( RIGHT );
        signal( seats );
    }
}
```

- allows only up to 4 philosophers to attempt to eat → breaks the loop → deadlock impossible!

10.4.5 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 );
    }
}
```

```
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] );
    }
}
```

```
void putChpStcks( i )
{
    wait( mutex );

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

    signal( mutex );
}
```

Why it works?
- Each philosopher holds on to his own semaphore
- Prevents the situation where one philosopher waits on another philosopher semaphore which guarantees no deadlock (starvation is possible)
- Once philosopher done eating then it signals its neighbours that it is safe to eat

11 Memory Abstraction

11.1 Memory Hardware

- Physical memory storage:
  - RAM (Random Access Memory)
    - Access latency (approximately) constant regardless of the address
- Can be treated as 1D array of AUs (addressable unit), typically 1 byte
- Each AU has a unique index (byte addressable)
- 4 segments: Text and Data (fixed size after compilation), Heap and stack (size can change during runtime)

11.2 Problems with unrestricted access to RAM

1. memory hog
2. overriding part of memory already filled
3. allows malicious programs to access sensitive data

11.3 Calculations of Memory Address

- After code is compiled, executable will typically contain compiled code (Text region) and Global Data.
- Stack and Heap are runtime structures which have unknown sizes during compile time and requires OS intervention to ensure that we can allocate memory dynamically without problems.

11.4 Memory Usage: Summary

Generally, 2 types of data in a process:

- **Transient Data**: Valid only for limited duration e.g. function parameters, local variables live only for the duration of a function call
- **Persistent Data**: Valid for duration of the program or until explicitly deallocated e.g. global variables, dynamically allocated memory
- Both types of data sections can **grow/shrink during runtime**

11.5 OS: Managing Memory

1. Allocate memory space to new process and load process into memory
2. Manage memory space of process
3. Protect memory space of process from other processes
4. Provide memory related syscalls e.g. `malloc`, `shm`
5. Manage kernel memory for internal OS use (OS cannot call malloc to allocate new memory for itself!)

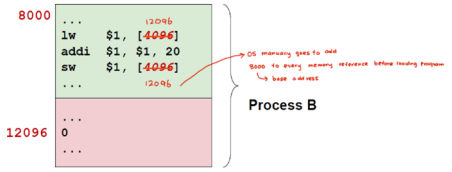
11.6 Memory Abstraction

11.6.1 Without Memory Abstractions

- Processes **directly uses physical addresses**
- Pros:
  - Memory access is straightforward (fastest way of accessing memory)
  - Address in program == Physical address
  - No conversion/mapping required
  - Address fixed at compile time
- Cons:
  - Hard to protect memory space
  - 2 processes occupy same physical location → both processes assume their memory start at 0

11.6.2 Fix Attempt: Memory Relocation

Idea: Recalculate memory references during loading

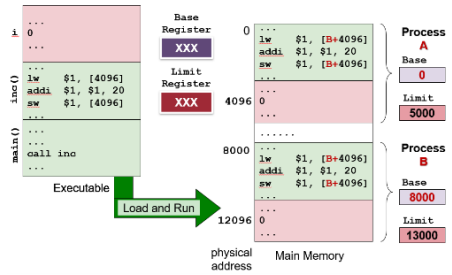


Problems:

- Slow loading time (relies on OS to recalculate address which significantly increases load time)
- Not easy to distinguish pointers from normal integer constant (lots of edge cases)

11.6.3 Fix Attempt 2: Base + Limit Registers

- Use a Base register (part of hardware context) as the base of all memory reference
  - During compilation time, all memory references are compiled as offset from Base e.g. Base + 72
  - At loading time, base register is initialized to starting address of memory space by hardware
- Have another Limit register to indicate the range of memory space of current process
  - Every memory access is checked against the limit to protect memory space integrity (cannot go beyond base + limit)
  - Limit is just an **estimation** as we do not know the actual memory needed during runtime
  - Limit is variable at runtime



Problems:

- To access address Adr:
  1. Actual = Base + Adr (where Adr is the address where the program think the data is)
  2. Check Actual < Limit for validity
- Each memory access incurs addition and comparison which happens in the critical path of execution → slows down execution time

Pros:

- Provides a crude memory abstraction → Address X in 2 different processes no longer share same physical location

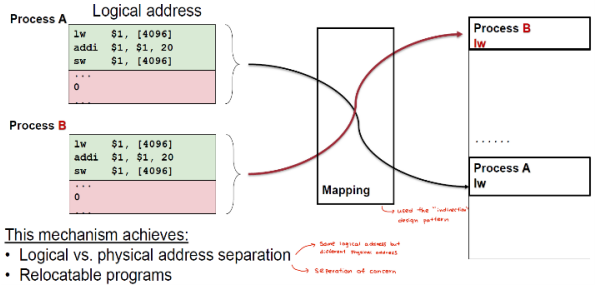
11.6.4 Logical Addresses

From the previous attempts, we can see that embedding actual physical address in program is a bad idea:

- does not allow address to be changed during run time as addresses are determined during compile time

Solution: Logical Address

- Logical Address == how the process views its memory space
- Logical Address ≠ Physical address in general
  - Mapping from logical to physical address needed (mapping maintained by OS)
- Allows each process to be a self-contained, independent logical memory space



11.7 Contiguous Memory Management

Assumptions (only for contiguous memory):

- Process must be in memory (physical memory is large enough to contain ≥ 1 process with complete memory space), as one piece (occupies contiguous memory region), during whole execution

11.7.1 Multitasking, Context Switching & Swapping

- To support **multitasking**: allow multiple processes in the physical memory simultaneously
- When memory is full: Free up memory by removing terminated processes, swapping blocked processes to secondary storage

11.8 Memory Partitioning

Memory Partition: Contiguous memory region allocated to a single process

Allocation Scheme:

1. Fixed-Size partitions
   - Physical memory split into fixed number of partitions of equal size (set to fit maximum expected size of any process)
   - Each process will occupy any ONE of the partitions (1 process → 1 partition)
2. Variable-Size partitions
   - Partition created based on the actual size of process
   - OS keep track of occupied and free memory region
     - Performs splitting and merging when necessary (more complex, less fragmentation, better resource efficiency)

11.8.1 Fixed Partitioning

If a process does not occupy whole partition:

- Leftover space will be wasted (not allocatable to other processes), known as Internal Fragmentation

Pros:

- Easy to manage
- Fast to allocate (every partition is the same, just randomly pick a free partition)

Cons:

- Partition size need to be large enough to contain largest of processes
  - Smaller process will have a lot of internal fragmentation
  - If partition size is too small, some processes cannot run

11.8.2 Dynamic Partitioning

In dynamic allocation, with process creation/termination/swapping, there will be many free memory spaces known as holes. These holes are also called external fragmentation (does not belong to any partitions).

Typically implemented using LinkedList which stores whether a **partition is occupied, start address, length of partition and a pointer to the next node**

Pros:

- Flexible
- Removes internal fragmentation (size of partition is exactly what process needs)

Cons:

- Need to maintain more information in OS (how many partitions, how much memory is allocated to each partition etc.)
- Takes more time to locate appropriate region
- External Fragmentation

11.8.3 Dynamic Partitioning: Allocation Algorithms

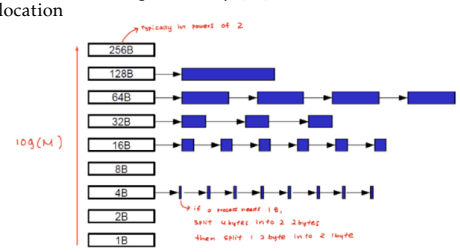
- Assume OS maintains a list of partitions and holes
- 3 algorithms to locate hole of size M ≥ N:
  - **First-Fit**
    - Take first hole that is large enough
  - **Best-Fit** (Minimize the wastage)
    - Find smallest hole that is large enough
  - **Worst-Fit**
    - Find largest hole
  - Split chosen hole into N and M – N
    - N will be new partition
    - M – N will be left over space i.e. a new hole

11.8.4 Merging and Compaction

- **Merging:** When occupied partition is free, try to merge with adjacent free partition to create a bigger hole
- **Compaction:** Move occupied partitions around to create bigger consolidated holes. **Might not be able to do it if process has ptrs** since it might point to wrong address if shifted around. Done only when system cannot allocate partition for new process
- VERY TIME CONSUMING

11.8.5 Multiple Free Lists

1. Separate list of free holes from list of occupied partitions
2. Keep multiple lists of different size holes
   - Take hole from a list that most closely matches request size
   - Partition size increases exponentially (2^x)
   - Faster allocation



11.8.6 Buddy System

- Popular implementation of Multiple Free Lists
- Buddy memory allocation provides efficient
  - Partition splitting
  - Locating good match for hole
  - Partition de-allocation and coalescing
- Idea:
  - Free block split into half repeatedly
    - 2 halves form a buddy block
  - When both buddy blocks free, merge into bigger block

Buddy System Implementation:

- Keep array A[0…k], where 2^k is the largest allocatable block size
  - Each array element, A[j] is a linked list which keeps track of free block(s) of size 2^j and each free block is indicated by their starting address

Algorithm:

- To allocate block of size N:
  1. Find smallest S s.t $2^S \geq N$
  2. Access A[S] for free block
     (a) if free block exists:
       - remove block from free block list
       - allocate block

     (b) Else
       - Find smallest R from S+1 to K s.t A[R] has free block B
       - For (R-1 to S)
         - Repeatedly split B → A[S…R-1] has new free block
       - Goto step 2
- To free a block B:
  1. Check in A[S], where 2^S == size of B
  2. If buddy of B is also free
     - Remove B and C from list
     - Merge B and C to get larger block
     - Repeat step 1 again with larger block
  3. Else (buddy not free)
     - Insert B to list in A[S]
- 2 blocks are buddy of size 2^S if lowest S bits of B and C are identical and bit S of B and C are different e.g. (left = 0 bit, right = 1 bit)

$$\text{buddy} = \begin{cases} 110000 \\ 111000 \end{cases}$$

* Note: Buddy system **suffers from both external and internal fragmentation** but tries to minimize both

12 Disjoint Memory Schemes

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

12.1 Paging

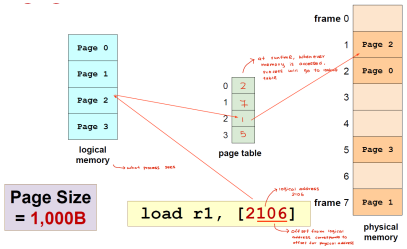
Physical memory is split into **physical frames** and logical memory is split into **logical pages** so the same size (determined by hardware). At runtime, pages of process are loaded into any available memory frame → logical memory can remain contiguous but physical memory can be disjoint!

12.1.1 Tricks to simplify translation

1. Keep frame size as a **power of** 2 (hard to deal with page size of say 97 bytes)
2. Physical frame size == Logical frame size (allows for same offset)
   - For page size of 2^n and address of m bits, to translate, we copy the last n bits (offset) then for the leftmost $m - n$ bits (page number) we just have to map logical address to physical frame

12.1.2 Lookup mechanism

To support translation from logical to physical page, we will need a lookup table (**page table**) which lives in OS memory on the RAM.



Physical add. = Frame# × `sizeof`(frame) + offset

12.1.3 Analysis of paging

- **No external fragmentation:** No leftover physical memory region, every single frame can be used
- **Has internal fragmentation (insignificant):** when logical memory space ≠ multiple of page size. e.g. A process can require 1000 frames → will use up first 999 frames but last 1 may not be fully utilised
- **Clear separation of logical and physical address space:** allows for flexibility and simple translation

12.1.4 Implementation: Pure software solution

OS stores page table information (using pointers) in PCB (kernel space of RAM), specifically in the memory context. However, this means that we will need 2 memory access (just for data) for memory every reference:

1. read indexed page table entry to get frame number
2. access actual memory item

* Possibly ≥ 2 memory access needed since code and data stored differently in modern von Neumann architecture

12.1.5 Paging with Hardware Support (TLB)

Processors have TLB (Translation Look-up Table) which is a specialized on-chip component to support paging. Typically, will have **1 TLB per core or per hardware thread.**

- Acts as **cache** (most frequently used cache) for page table entries
- **Very small**: 10s of entries only
- **Very fast:** ≤ 1 clock cycle (**1ns**), must be fast since TLB accessed multiple times per cycle
- **TLB Hit:** Immediately can get physical address, skip looking up in page table
- **TLB Miss:** access page table in RAM (**50ns**) and update TLB
- Hit ratio is typically 99% for it to be effective

e.g. Assuming TLB hit rate = 90%

Average memory access time = P(hit) x latency(hit)+ P(miss) x latency(miss)
= 0.9 x (1ns + 50ns) + 0.1 x (1ns + 50ns + 50ns)

- **Context switching:**
  - upon context switch, pointer to TLB is saved
  - TLB is flushed (unless PID is saved) so that new process will not get incorrect translation (if 2 process use same logical address) → ensures correctness, security, and safety issues
  - TLB miss rate is high when original process resumes running

12.1.6 TLB Protection

Extend paging entries to include **access-right bit** and **valid bit**

- **Access-right bits**: each PTE will have `writeable, readable, executable` bits. e.g. page containing code should be executable and read only, page containing data should be readable and writeable
  - every memory access is checked against access-right bits in hardware
- **Valid bit:** indicates whether page is valid to access by process
  - OS sets valid bits during runtime
  - every memory access checked against valid bit in hardware, if out of range → caught by OS

12.1.7 Page Sharing

Page table allows several processes to share the same physical memory frame by **pointing to same physical frame** in the PTE.

Usage:

1. **Shared code page:** e.g. standard lib used by 2 processes
2. **Implementing Copy-on-Write:** parent and child "share" pages until someone modifies the memory

12.2 Segmentation

- So far, memory space of process is treated as a **single entity**. However, each process has multiple logical memory region with different usage, permissions, lifetime, and scope.
- Some regions **grow/shrink at runtime** (e.g. stack and heap) which makes it hard to achieve if whole process is 1 piece. We need to provision gaps in advance but makes it hard to check whether memory access is in-range

12.2.1 Segmentation Scheme

Manage memory at level of memory segments:

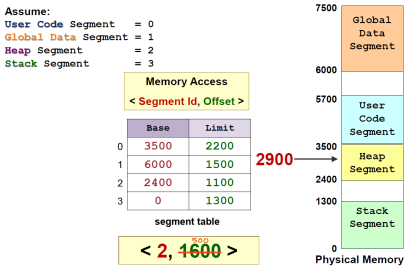
- process now have a collection of segments which are **contiguous partitions** of the same size
- each memory segment has:
  - **name:** for easy reference, usually translated to an index e.g. Text = 0, Data = 1 etc.
  - **limit:** indicate exact range of segment
- memory access becomes **Segment name + Offset** e.g. "Heap" + 245

* Within segment: contiguous. But individual segment can be anywhere in memory

12.2.2 Segment Table

To translate logical address to physical address, we maintain a **segment table** of [Base, Limit] indexed by SegID.

Physical Address = Base + Offset, where offset < limit for valid access



12.2.3 Hardware support

Since segment table is small, we can **store it directly onto the CPU register** to speed up access time

12.2.4 Segmentation Analysis

Pros:

- Each segment is an independent contiguous memory space which matches programmer's view of memory
- Efficient bookkeeping since we can easily control each segment's permission
- Segments can grow/shrink and be protected/shared independently

Cons:

- Segmentation requires variable-size contiguous memory regions which could lead to external fragmentation
- Copy-on-write copies entire segment

12.3 Segmentation + Paging

Each **segment is now composed of pages** and has a page table of its own. Segments can grow/shrink by allocating/deallocating new page then add/remove to/from its page table. Checking of permissions happens in segment level instead of page level. **Copy-on-write will copy the entire segment.**

13 Virtual Memory

We remove another assumption that our physical memory is large enough to hold processes logical memory space completely. We also observe that secondary storage capacity (HDD, SSD) » RAM capacity and some pages are accessed much more often than others.

Idea: Extend on paging scheme and keep some pages on RAM and some pages on SSD/HDD

- **Logical address no longer restricted by physical memory**
- **More processes in memory:** improve CPU usage since more processes can be chosen to run on memory
- **More efficient use of physical memory:** RAM sort of becomes cache of frequently used pages

13.1 Extended Paging Scheme

2 page types:

1. Memory resident: pages in physical memory
2. Non-memory resident: pages in secondary storage

Now requires an additional "**resident bit**" in PTE to indicate whether page is resident in memory. If CPU access non-resident page → page fault → OS bring page from storage to memory (**takes very long, ms**)

13.1.1 Page Access using Extended Paging

1. (Hardware) Check page table. If memory resident, done, else continue
2. (Hardware) **Page fault:** TRAP, pass control to OS
3. (OS) Locate page in secondary storage
4. (OS) Load copy of page in physical memory frame
5. Done using DMA controller → CPU is free
6. (OS) Update page table (set resident bit to 1)
7. (OS) Go back to step 1 to re-execute **SAME** instruction

* Steps 3 and 4 are I/O operation, process will get **blocked** and other processes can run

13.2 Issues with virtual memory

- Secondary storage access time » physical memory access time
  - **ms** vs **ns**, 5 orders of magnitude difference
- If memory access results in page fault most of the time: have to constantly load non-resident pages into memory, known as thrashing

13.2.1 Locality Principles

Thrashing is unlikely to occur in reality due to Temporal and Spatial locality. Most code spend a lot of time on relatively small part of code and only access small part of data in a given time period

- **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.
- It is however still possible for badly designed code/malicious code to behave badly

13.3 How to decide who belongs in memory?

Considerations:

- Large start-up cost if large number of pages to allocate and initialize
- Need to reduce footprint of processes in RAM so that more processes can use physical memory

13.3.1 Demand Paging

Start with no memory resident page and only **allocate page when there is a page fault**.

Pros:

- Fast start-up time for new process (can just start without any page in RAM)
- small memory footprint

Cons:

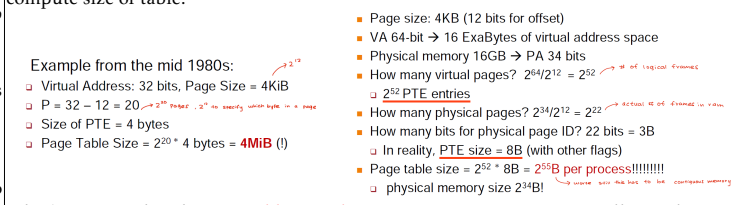
- Process may appear **sluggish at the start** due to page faults
- page faults may have cascading effects on other processes (**thrashing** occurs!)

13.4 Page Table Structure

Page Table information takes up space on physical memory space. Given modern context where logical memory is huge (can be ≥ 8TiB), page table will be as huge which cause a **high overhead** and **fragmentation** since it can occupy several frames.

13.4.1 Direct Paging

Keep all entries in a single table and allocate this huge page table to each process. To compute size of table:

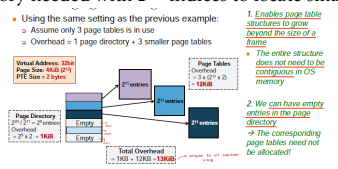


What's worse is that the page table must be contiguous in OS memory to allow indexing to work (access pth entry in one memory access). Since OS memory is small, storing a table of 2^{55} bytes is impossible.

13.5 2-Level Paging

Further split the page table into regions and only allocate these regions when needed. We then keep a **directory** of these regions. Each table entry **points to the base address** of the next page table

- Split page table into smaller page tables (each mini page table has a number associated with it)
- If original page table has 2^p entries:
  - assuming with have 2^M mini page tables, M bits needed to uniquely identify 1 page table
  - Each smaller page table contains $2^{(P-M)}$ entries
- To keep track of smaller page table
  - single page directory needed with 2^M indices to locate smaller page table



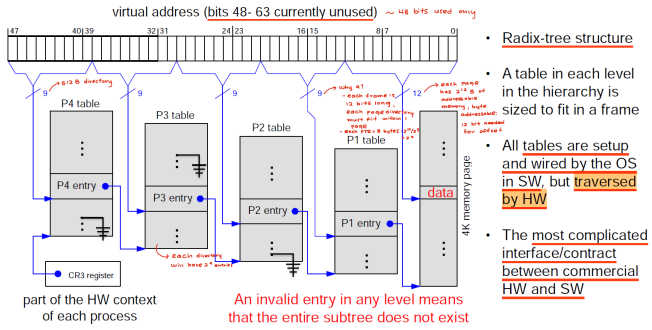
13.5.1 2-Level Paging Problems

Requires **2 serialized memory access** (slow!) to get frame number: 1 to access directory, 1 to access page-table

- Solutions:
  - **TLB**: **eliminates page-table access**, but when there is TLB miss will experience longer page-table walks (traversal of page-table in hardware)
  - **MMU (memory management unit)**: tiny cache in MMU caches frequent page directory entries to speed up page-table walks upon TLB miss. **Eliminates access to page directory.** TLB for directory entries not PTEs. Typically 1 MMU/page directory.

13.5.2 Hierarchical Page Table

Hierarchical Page Tables uses **indirection** design principle to reduce size of page tables.



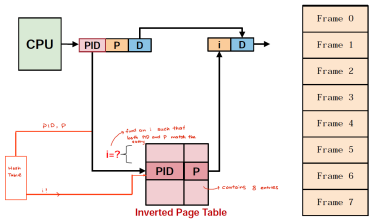
13.6 Inverted Page Table

We want a page table that tells us **which process is using which frame**. Often used as auxiliary structure

- Keep a single mapping (only 1 per system) of physical frame to <pid, page#>
- page# may not be unique among processes
- pid + page# can uniquely identify a memory page

Provides:

- **Fast lookup by frame**: answers questions like who are all the sharers of physical frame X?
- **Huge savings**: one table for all processes
- **Slow translations**: need to search the whole table to lookup page X
  - Not the point of inverted page table



13.6.1 Page Replacement Algorithms

When there are no free pages during page fault, we need to evict a memory page. When page is evicted:

- **Clean page**: not modified → no need to write back
- **Dirty page**: modified → need to write back

13.6.2 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.

13.6.3 Memory Access Time

$T_{access} = (1-p) \times T_{mem} + p \times T_{page\_fault}$, we want to reduce p since $T_{page\_fault} >> T_{mem}$

13.6.4 Optimal Page Replacement

Assuming we have knowledge of the future, we will replace the page that will not be used again for the longest period of time.

- Guarantees minimum page faults (used as benchmark)

13.6.5 FIFO Page Replacement

Pages evicted based on loading time, **evict oldest memory page.**

- **OS maintains queue of resident page numbers**: remove first page in queue if replacement needed, update queue during page fault
- **Simple to implement**: No hardware support needed
- **Belady's Anomaly**: When we increase the number of frames, it leads to more page faults in FIFO algorithm.
  - Due to FIFO not utilising temporal locality
  - Bad performance in practice

13.6.6 LRU (Least recently used)

Replace the page that has not been used in the longest time.

- Honours Temporal Locality
- **Close to OPT algorithm**: predict future by mirroring the past
- **No Belady's anomaly**
- **Difficult to implement**: need to keep track of last access time, 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.

13.6.7 Second-chance Page Replacement (Clock)

Modified FIFO. Maintain an additional reference bit for each entry: 1 means accessed since last reset, 0 means not accessed. Degenerates into regular FIFO if all referenced bit == 1.

1. 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
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

13.7 Frame Allocation

If we have N physical frame and M processes competing for these frames, how do we allocate?

1. **Equal allocation**: each process gets N / M frames
   - possible to have a scenario whereby process A needs 1 page and process B needs 1,000,000 pages but both are allocated 10 pages → process B thrashes
2. **Proportional allocation:** each process gets $\frac{size_p}{size_{total}} \times N$ frames

13.7.1 Local Replacement

Victim page is selected among pages of the process that cause page fault

- **Constant frame allocation**: Number of frames allocated to all processes remains the same → performance is stable between runs
- **Insufficient Allocation**: hinders the progress of process
- **Thrashing**: Limited to one process, but that single process can use up I/O bandwidth and degrade performance of other process

13.7.2 Global Replacement

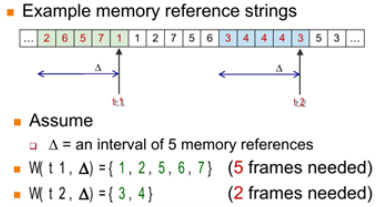
Victim page selected among pages of all processes. Possible for Process P to page fault and evict Process Q frames.

- **Self-adjustment**: Processes that need more frames can get them from other processes that need less
- **Inconsistent performance**: Number of frames allocated to process can differ between runs
  - unfair allocation
- **Cascading thrashing**: one process that thrashes will "steal" page from others, resulting in other processes also thrashing
- **Malicious process**: if another process does I/O and malicious process keeps thrashing, OS busy swapping frames and might not be able to schedule I/O operation, hogs up both I/O and memory bandwidth

13.8 Working Sets

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 working set window Δ which is an interval of time
- W(t, Δ) is the set of active pages in the interval at time t
- We want to allocate enough frames for pages in W(t, Δ) to reduce page fault
- Accuracy of model depends on Δ
  - **Too small**: miss page in locality, lots of page fault (high page fault rate, low CPU utilization)
  - **Too big**: contains page from different locality, CPU utilization will be low (low page fault rate but low CPU utilization too)



13.9 All Page Entry Bits Thus Far

Access right bits, Valid Bit, Is-Memory Resident Bit, Dirty Bit, Reference Bit (for clock algo)

14 File Systems

14.1 Motivation

- Physical memory is **volatile**: Once you shutdown your computer, RAM is cleared
- Have to use external storage to store persistent information: even after restart, data still remains
- Direct access to storage media is **not portable**: Dependent on hardware specification and organization
- File system provides:
  - **abstraction** on top of physical media
  - high level **resource management** scheme: buses to read data and how we allocate memory
  - **Protection** between processes and users: ability to isolate founds among users
  - **Sharing** between processes and users

14.1.1 General Criteria

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

14.2 Memory vs File Management

	Memory Management	File System Management
Underlying Storage	RAM	Disk (HDD/SSD)
Access Speed	Constant (ns - 10ns for first access, can be faster for subsequent ones)	Variable disk I/O time (depends on "seek time")
Unit of Addressing	Physical memory address	Disk sector
Usage	Address space for process (come from program counter or instruction) **Implicit** when process runs	Non-volatile data **Explicit** access
Organization	**Paging/Segmentation:** determined by HW & OS	Many different FS: ext* (Linux), FAT* (Windows), HFS* (Mac OS)etc. (how files are organized is different)

14.3 File System Abstraction

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

14.3.1 Files

- logical unit of information created by process
- essentially an abstract data type with a set of common operations
- Contains **data** (information structured in some way) and **metadata** (additional information associated with file)

14.3.2 File Metadata

Name:	A human readable reference to the file
Identifier:	A unique id for the file used internally by FS (e.g. inode number; so that we know which file to open if they have the same name)
Type:	Indicate different type of files E.g. executable, text file, object file, directory etc
Size:	Current size of file (in bytes, words or blocks) (does not contain size of metadata)
Protection:	Access permissions, can be classified as reading, writing and execution rights
Time, date and owner information:	Creation, last modification time, owner id etc
Table of content:	Information for the FS to determine how to access the file (usually not stored together with the file; stored in some auxiliary data structure)

File Name

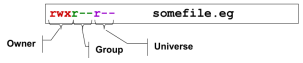
- Different FS has different naming rule (e.g. FAT32 follows 8.3 format → xxxxxxxx.(ext))
- Rules: length, case sensitivity, allowed special symbols and file extension (indicates file type in some FS)

File Type

- OS commonly supports multiple file types
- Each file type has associated set of operations, possibly a specific program for processing (e.g. PowerPoint for .pptx)
- Common file types:
  - **Regular files**: contains user information
    - ASCII files: text files, source codes, etc. Can be printed as is.
    - Binary files: executables `mp3`, `pdf` etc. Have a predefine internal structure that needs a specific program to access
  - **Directories**: system files for FS structure
  - **Special files**: character/block oriented (e.g. `/dev/stdin`)
- **Windows**: Use file extension as file type
- **UNIX**: use magic number embedded at start of file

File Protection

- Controlled access to information stored in file
- Types of access:
  - Read: Retrieve information from file
  - Write: Write/Rewrite the file
  - Execute: Load file into memory and execute it
  - Append: Add new information to the end of file
  - Delete: Remove the file from FS
  - List: Read metadata of a file
- Most common approach: Restrict access base on user identity
  1. Access Control List
     - list of user identity and the allowed access types
     - **Pros**: Very customizable
     - **Cons**: Additional information associated with file (overhead due to more meta-data stored)
  2. Permission Bits:
     - Classified the users into three classes: Owner (user who created file), Group (set of users who need similar access), Universe (all other users)
     - Note that directory permission and file permission are different (allows us to modify a file under a "read only" directory if the file has write permission)



Operations on Metadata

- **Rename:**
  - Change filename
- **Change attributes:**
  - File access permissions
  - Dates
  - Ownership
  - etc
- **Read attribute:**
  - Get file creation time

14.3.3 File Data

Access Methods

- **Sequential Access** (e.g. Tape Drives): **data read in order**, starting from beginning. Cannot skip but can rewind
- **Random Access**: data can be **read in any order**. Either use Read(Offset) or Seek(offset). Basically a special case of direct access where 1 record == 1 byte
- **Direct Access**: used for files containing fixed-length records, allows random access to any record directly. Often used in databases where there is a large# of records

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

14.4 Representation of Open File

OS provides file operations as syscalls to provide protection and allow for concurrent and efficient access. Some information will hence need to be kept by the OS:

1. **File pointer**: Current location in file
2. **Disk location**: Actual file location on disk
3. **Open count**: How many process has this file open. Useful to determine when to remove the entry

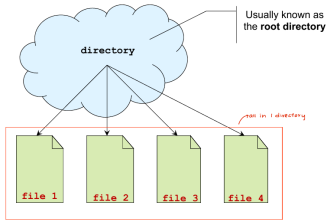
14.4.1 2 Table Approach

1. **System-wide open-file table** (Kept on OS Memory on RAM)
   (a) Keeps track of open files in the system
   (b) If same process opens the same file twice, or two processes open the same file, there will be two separate entries in the open file table (doing different things to file at different offsets → need different fd)
   (c) If one process opens a file then **forks**, only one entry in open file table
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
   (d) If the same process open the same file multiple times, there will be multiple entries in the per-process file table
   (e) **Default file descriptors: (1) STDIN (2) STDOUT (3) STDERR**

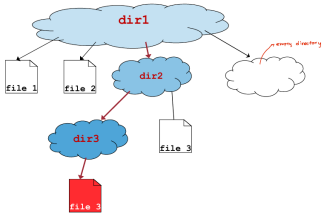
14.5 Directory

Helps to provide a logical grouping of files and keep track of files.

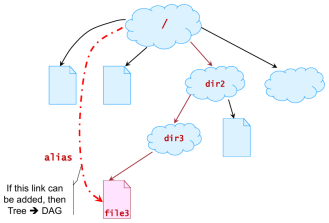
Single Level



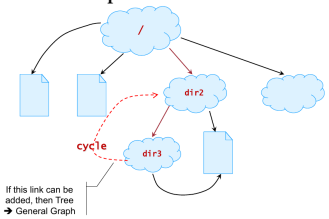
Tree Structure



DAG



General Graph



14.5.1 Tree Structure

- Directories can contain directories: Forms a tree structure where files are leaves
- **Absolute Pathname**: Refer to file by path from Root
- **Relative Pathname**: Refer to file from **current working directory**

14.5.2 DAG

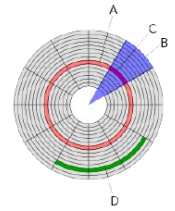
- To get a DAG, we must be able to "skip" level, i.e. share a file/directory such that it appears in multiple directories but refer to same copy of actual content
- **Hard links**: Directories *A* and *B* have separate pointers to file *F*. Only works with files
  - Pros: Low overhead, only pointers are added in directory (only take up directory space)
  - Cons: Deletion problems - if only one of *A* or *B* deletes the file, the data remain. To completely delete data, we will need to delete all reference to the file.
  - `ln` command in UNIX
- **Symbolic Links**: Directory *B* creates a special link file *G* that contains path name of *F*. When *G* is access we access *F* instead.
  - Pros: Simple deletion. When *B* is deleted, delete *G* but not *F*. If *A* is deleted, *F* is gone, *G* remains but does not work
  - Cons: Larger overhead - link file take up actual disk space
  - `ln -s` command in UNIX

14.6 General Graph

Created when tree has a cycle. Possible in UNIX but not desirable.

- Hard to traverse: Need to prevent infinite loops
- Hard to determine when to remove file/directory

14.7 I/O: Disk Structure



A = Track
B = Geometric Sector
C = Track Sector
D = Cluster

- **Track**: One ring around disk. One disk can have many tracks of different radii
- **Geometric Sector**: Sector of a track
- **Disk Head**: Stick thingy that moves above the disk. Transforms disk's magnetic field into electrical currents
- **Rotation**: typically 3600/5400/7200 RPM
- **Seek**: Shifting of the head to change track
- **Time taken to access data on disk**: Seek time + Rotational Delay + Transfer delay

15 File System Implementation

File systems are stored on storage media (e.g. HDD, CD/DVD, SRAM/DRAM). To the OS, disk sectors are treated as a 1-D array of logical blocks, usually 512 bytes to 4KiB. The mapping between the logical block and sectors is **hardware dependent**

15.1 Disk Organization

- **Master boot record (MBR)**: Found at sector 0. Contains:
  - Simple Boot Code
  - Partition Table (tells where partitions start and end)
- **Partitions**: Each partition can contain an independent file system. Contains:
  1. OS Boot-up information
  2. Partition details: total number of blocks, number and location of free disk blocks
  3. Directory Structure
  4. Files Information
  5. Actual file data

15.2 File Implementation

In the process of allocating file data, we need to keep track of logical blocks, allow efficient access and utilize disk space efficiently. There will be internal fragmentation when file size ≠ multiple of logical blocks

15.2.1 Contiguous Allocation

Allocate consecutive disk blocks to a file

Pros:

- Simple to keep track - each file only need starting block# + length
- Fast access: just need to seek first block
- Good for WORM (write once read many) drives e.g. CD-ROM

Cons:

- External Fragmentation: Think of each file as variable size partition. After a lot of creation/deletion, disk can have many small "holes".
- File size need to be specified in advance

15.2.2 Linked List

Maintain disk blocks as linked list. Each block stores ptr to next block and actual file data. File information stores first and last disk block (to support fast append operations)

Pros: Solve fragmentation problem

Cons:

- Random access is very slow
- Less usable space: part of disk block used to store pointer
- Less reliable: fails if 1 pointer is incorrect
- possible to create circular linked list!

15.2.3 Linked List V2

Same as Linked List but we move all ptrs into a file allocation table (FAT) in memory.

- Faster access: Traversal now happens in memory instead of disk
- Takes up space: Fat keeps track of all disk blocks in partition. Huge when disk is large, consuming memory space

15.2.4 Indexed Allocation

Each file has an index block which stores an array of disk block addresses/ IndexBlock[N] == N^{th}+1 block address. Each block now can either store data or the array.

Pros:

- **Lesser memory overhead**: only index block of opened file needs to be in memory
- **Fast direct access**: No need traverse

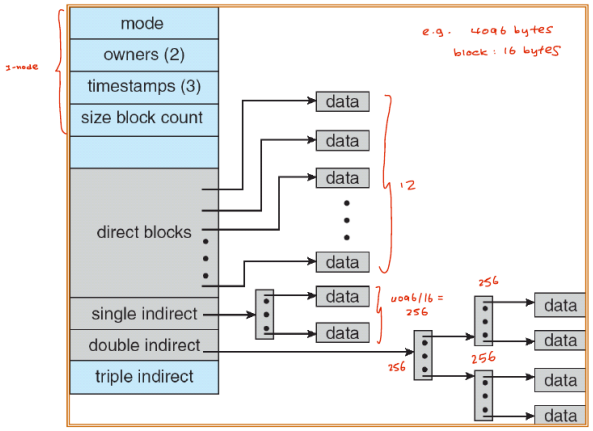
Cons:

- **Limited Max File Size**: Max number of blocks == Number of index block entries
- **Index block overhead**

Variations

- Exists other schemes to allow larger file size
- **Linked scheme**: Keep linked list of index blocks
  - Each index block contains pointer to next index block
- **Multilevel index**: similar idea to multi-level paging
  - N level index blocks points to a number of N + 1 level index blocks
  - UNIX I-Node uses this idea: 12 direct pointers, 1 single indirect block, 1 double indirect block, 1 triple indirect block

I-Node Structure



15.3 Free Space Management

To perform file allocation, we need to know which disk blocks are free/not free

15.3.1 Bitmap

Each block is represented by 1 bit. 1 == free, 0 == occupied

Pros: Easy to manipulate: use bit operations to find first free block or n-consecutive free blocks

Cons: Need to keep in memory for efficiency reason

15.4 Linked List

Use a linked list of disk blocks containing number of free disk block numbers and a pointer to next free space disk block

Pros: Easy to locate free block, only first pointer needs to be in memory (memory efficient) though we can cache other blocks for efficiency

Cons: High overhead as we are storing free block numbers in data block. Can be mitigated by storing free blocks list in free blocks instead.

15.5 Directory Structure

Main task of directory structure is to provide some **mapping between file name and file information** and to keep track of files in a directory. To use a file, have to use the syscall open which locates file information using pathname + filename. Sub-directory is usually store as file entry with a special type in a directory

15.5.1 Linear List

- Directory consists of a list:
  - Each entry represents a file:
    - Store file name (minimum) and possibly other metadata
    - Store file information or pointer to file information
- Locate a file using list:
  - Requires a linear search
    - Inefficient for large directories and/or deep tree traversal
  - Common solution: (make use of temporal locality)
    - Use cache to remember the latest few searches
      - User usually move up/down a path

15.5.2 Hashtable

- Each directory contains a
  - Hash table of size `N`
- To locate a file by file name:
  - File name is hashed into index `K` from 0 to `N-1`
  - `HashTable[K]` is inspected to match file name
    - Usually chained collision resolution is used
    - i.e. file names with same hash value is chained together
      - to form a linked list with list head at `HashTable[ K ]`
- **Pros:**
  - Fast lookup
- **Cons:** (kind of solved using linked list)
  - Hash table has limited size
  - Depends on good hash function

15.5.3 File Information

- File information consists of:
  - File name and other metadata
  - Disk blocks information
    - As discussed in the file allocation schemes earlier
- Two common approaches: (MS-DOS uses this (FAT))
  1. Store everything in directory entry (metadata also stored)
     - A simple scheme is to have a fixed size entry
       - All files have the same amount of space for information
  2. Store only file name and points to some data structure for other info (used by Unix)
     - (directory entry points to inode to get metadata) (metadata)

15.5.4 File System in Action

File Creation

- Let us relook at the file operation
  - With the newly covered details
- To create a file `/.../.../parent/F`:
  - Use full pathname to locate the **parent** directory
    - Search for filename `F` to avoid duplicates
    - If found, file creation terminates with error
    - Search could be on the cached directory structure
  - Use free space list to find free disk block(s)
    - Depends on allocation scheme
  - Add an entry to **parent** directory
    - With relevant file information
    - File name, disk block information etc

File Opening

- Process `P` open file `/.../.../.../F`:
  - Search system-wide table for existing entry `E`
    - If found:
      - Creates an entry in `P`'s table to point to `E`
      - Return a pointer to this entry
    - If not found, continue to next step
  - Use full pathname to locate file `F`
    - If not found, open operation terminates with error
    - When `F` is located, its file information is loaded into a new entry `E` in system-wide table
    - Creates an entry in `P`'s table to point to `E`
    - Return a pointer to this entry
- The returned pointer is used for further read/write operation

15.6 Disk I/O Scheduling

Time taken to perform read/write operation

= **[Seek Time]** + **[Rotational Delay]** + **[Transfer Time]**

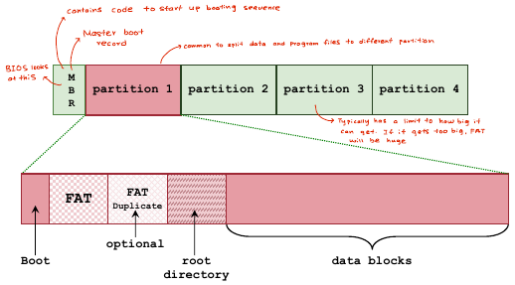
- **Seek time**: time taken to position disk head over correct track (typically 2ms to 10ms)
- **Rotational Latency**: time taken to wait for desired sector to rotate under disk head (4ms to 12.5ms per rotation, average is $\frac{1}{2}$ of 1 full rotation time)
  - Worst case: Sector just passed head, wait 1 full rotation
  - Best case: Sector is just before head. 0 rational delay
- **Transfer delay**: time taken to transfer sector, function of Transfer size / Transfer rate, typically takes μs which is 3 orders of magnitude faster than previous 2 delays
- Possible to suffer from same problems as normal scheduling, e.g. starvation when running SSF

15.6.1 Scheduling Algorithms

- **FCFS**: cannot change sequence by changing arrival time
- **Shortest Seek First (SSF)**
- **SCAN**: Bi-directional, go from innermost to outermost then back to innermost. Much like a lift.
- **C-SCAN**: Unidirectional, Go from outermost to innermost then reset

16 File System Case Studies

16.1 MS FAT File System



16.1.1 File Data and FAT

File Data is allocated to a number of data blocks/clusters and allocation info is kept as a Linked List with pointers kept in FAT. **FAT is entirely cached in RAM** to facilitate linked list traversal and there is **1 entry in FAT table per data block/cluster** (may not be same size as physical block)

FAT Entry Values

1. FREE code: block is unused
2. Block number of next block
3. EOF code: NULL ptr
4. BAD block: block is unusable e.g. disk error due to scratches on disk. Found out using a checksum

Directory

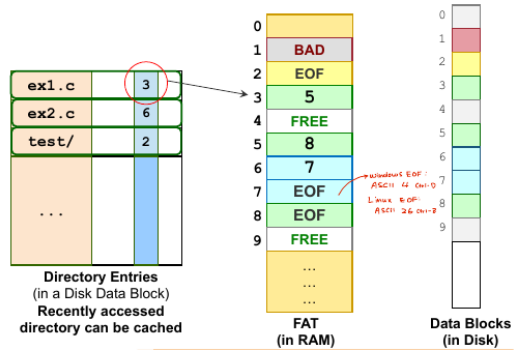
- Special type of file
- Root directory stored in a special location, other directories stored in data blocks
- Each file or subdirectory within the folder is represented by a **directory entry**

Directory Entry

Each entry has fixed-sized of 32 bytes

Contains:

- Name + Extension (11 bytes)
  - Limited to 8 + 3 characters (8 char for file name, 3 characters for file ext.)
  - First byte may have meaning (e.g. deleted, end of directory entries, parent directory)
- Attributes e.g. read-only, is-directory/file flag, hidden-flag (1 byte)
- Reserved: for future use (10 bytes)
- Creation date (2 bytes) + Time (2 bytes)
  - Year range: 1980 – 2107
  - Accurate to ± 2 seconds
- First Disk Block Index (2 bytes)
  - Disk block index is 12, 16 and 32 bits for FAT12, FAT16 and FAT32 respectively
  - For FAT32 lower half is stored
- File Size in bytes (4 bytes)



16.1.2 Tracing Process

1. Use first disk block number stored in directory entry to find the starting point of the linked disk blocks
2. Use FAT to find out the subsequent disk blocks number, terminated by EOF
3. Use disk block number to perform actual disk access on the data blocks

16.1.3 Common Tasks

- File Deletion:
  - "Delete" the directory entry:
    - Set first letter in filename to a special value **0xE5**
  - Free data blocks: (marks data blocks of the file as free, when you need to reuse the blocks just overwrite and overwrite)
    - ~~Set FAT entries in link list to FREE~~
  - Actual data blocks **remain intact**
    - Can attempt to undelete (just set 0xE5 to anything else)
      - (possible to have corrupted undeleted file if some parts of the deleted file has been recycled.)
- Free space management:
  - Do not keep track of free space information
  - Must be calculated by going through the FAT

16.1.4 Variants: FAT12, FAT16, FAT32

As storage gets larger, we want to be able to support larger hard disk as a single partition

- **Disk Cluster**:
  - Instead of using a single disk block as the smallest allocation unit, we use a number of contiguous disk block as smallest allocation unit
- **FAT Size**:
  - Bigger FAT → More disk block/cluster → More bits to represent each disk block/-cluster
  - Existing variants: FAT12, FAT16, FAT23
- Generally cluster size + FAT size determines largest usable partition

- Example (4KiB Cluster):

FAT12	FAT16	FAT32
2^{12} Clusters	2^{16} Clusters	2^{28} Clusters (using 28 bits)
Largest partition: 4KiB * 2^{12} = 16MiB	Largest partition: 4KiB * 2^{16} = 256MiB	Largest partition: 4KiB * 2^{28} = 1TiB

* Actual size is actually a little lesser (due to special values like EOF, FREE etc.) which reduces total number of valid block indices

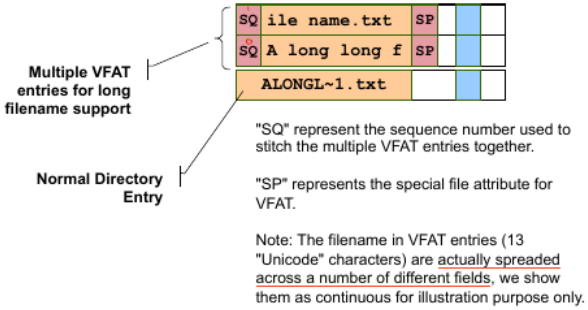
16.1.5 Cluster Size Tradeoff

- Larger cluster size → larger usable partition
- However, larger cluster size also leads to larger internal fragmentation
- On FAT32 there are also further limitation: 32-bit sector count (limited partition size), but only 28 bit is used in the disk block/cluster index

16.2 Long File Name Support

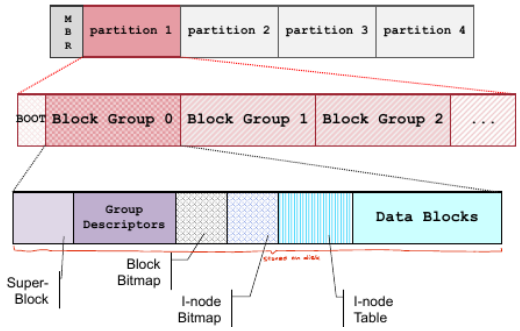
In original FAT, we can only have up to 8 characters as file name.

- Virtual FAT (VFAT) allows for filenames up to **255 characters**
- To support for additional characters, we use **multiple directory entries**.
  - Use previously invalid **file attribute** so non-VFAT applications can ignore additional entries
  - Use first byte to indicate sequence
- Keep 8+3 short version for backward compatibility

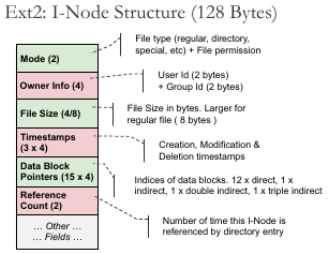


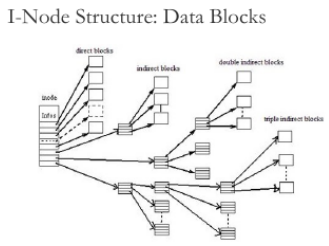
16.3 Linux Ext2

The layout of the file system is now slightly different. Blocks (correspond to one or more disk sectors) are grouped into block groups and each file/directory is describes by a single special structure called **I-Node** (Index Node), containing file metadata and data block addresses



- **Superblock**: Describes the whole file system, e.g. Total I-Nodes number, I-Nodes per group, Total disk blocks, Disk Blocks per group etc. Duplicated in each block group for redundancy
- **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.
- **Block Bitmap:** Keeps track of usage status of blocks in this block group (1 = Occupied, 0 = Free)
- **I-Node Bitmap:** Same as Block bitmap but for I-Nodes
- **I-Node table:** Array of I-Nodes in this block group





Note that file size **does not include file's metadata**, only includes the size of file content only

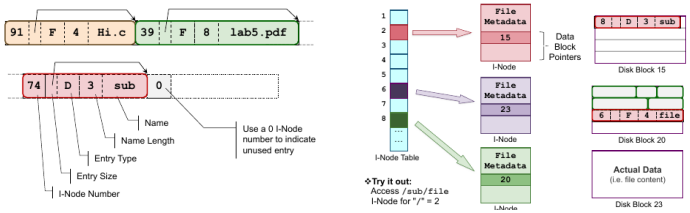
16.3.1 I-Node Data Block Example

- The design of I-Node allows:
  - Fast access to small file
    - The first 12 disk blocks is directly accessible
  - Flexibility in handling huge file
- Example:
  - Each disk block address is 4 bytes
  - Each disk block is 1KiB
    - So, indirect block can store 1KiB/4 = 256 addresses
  - Maximum File Size:
    = Direct blocks + single indirect + double indirect + triple indirect
    = 12 x 1 KiB + 256 x 1KiB + 256^2 x 1KiB + 256^3 x 1KiB
    = 16843020 KiB (16 GiB)

16.3.2 Directory Structure

A directory is just another file. Within this file, the data blocks form a "linked list" (allows for variable length → name can be variable length) of directory entries for files and subdirectories within the directory.

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



16.3.3 Tracing Process

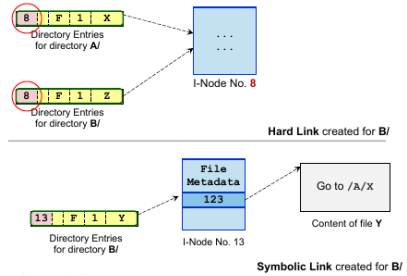
- Give a pathname, e.g. "**/sub/file**"
1. Let **CurDir** = "/"
   - Root directory usually has a fixed I-Node number (e.g. 2)
   - Read the actual I-Node
2. Look at the next part in pathname:
   - If it is a directory, e.g. "**sub/**"
     - Locate the directory entry in **CurDir**
     - Retrieve I-Node number, then read the actual I-Node
     - **CurDir** = next part in pathname
     - Goto Step 2.
   - Else //it is a file
     - Locate the directory entry in CurDir
     - Retrieve I-Node number, then read the actual I-Node

16.3.4 Common Tasks

- Deleting a file:
  - Remove its directory entry from the parent directory:
    - Point the previous entry to the next entry / end
    - To remove first entry: Blank record is needed
  - Update I-node bitmap:
    - Mark the corresponding I-node as free
  - Update Block bitmap:
    - Mark the corresponding block(s) as free
- Question:
  - Is it possible to "undelete" a file under ext2?
  - What if the system crashes between the steps?

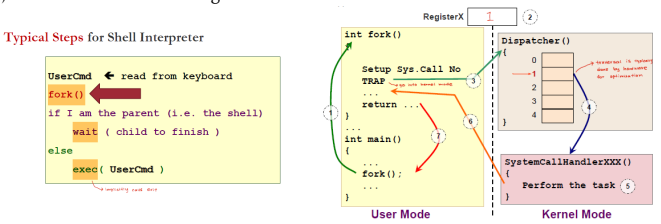
16.3.5 Symbolic Link vs Hard Link

- **Hard Link:** (can have different file names)
  1. Create 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.



17 Process involved in calling `ls`

1. At first, the shell/terminal starts off in the **blocked** state since it is just waiting for the user input and has nothing else to do
2. Once the user presses "l", an interrupt is fired which calls the appropriate **interrupt handler** (e.g. 3) using the **interrupt vector table** (IVT)
3. This in turns calls a **keyboard handler** code which is a piece of code pre-compiled in memory
   - Process of executing interrupt handler, loading in IVT and running handler code is **fully done in hardware**
4. Since it is likely a process X was interrupted, we would **need to save X's context** (e.g. registers, memory context etc.) into the hardware context in the PCB of x
5. This process repeats for each character entered into the command line
6. Once user completes command and presses "enter", the interpreter could be in 1 of 3 states:
   (a) **Ready**: Waiting to be scheduled
   (b) **Running**: Child spawned and running command
   (c) **Blocked**: Parent waiting for child to finish execution



7. Effect of fork?
   - Save CPU register which contains ptr to root of page table
   - No need to save: TLB, caches, process data, process page table, frames of pages
8. PCB_{child} of the spawned child would contain a copy of the parent's PCB. The copying can be done in 2 ways:
   (a) **"Hard way"**: Actually allocate new frames for the copied process upon process creation
   (b) **"Copy-on-write"**: Employs laziness and only allocate new frames when any process writes to data
9. Memory context of the processes will contain both Page Table (incur internal fragmentation due to paging, where page size == physical frame size) and TLB (makes use of temporal and spatial locality)
10. Child now enters the scheduling queue where it could either be in:
    (a) **New**: Freshly forked child
    (b) **Ready**: in the queue waiting to be scheduled
11. While waiting for the child, the parent is blocked which causes its hardware context to be saved so that other processes can run
    - Registers are actually pushed onto the stack due to efficiency (faster to push all register to stack in a single instruction vs copying register value one by one into PCB)
12. We now have to replace the memory content of shell process with ls process by bringing executable of ls into memory
13. After child is done executing, it informs parents through: (1) library call, (2) syscall (3) signal

18 QnA

18.1 OS Abstractions and Protections

Process (CPU Time)

- **Abstraction**: illusion that process executes on CPU all the time
- **Protection**: Execution context of each process is isolated from each other

Memory Management

- **Abstraction**: illusion that process owns the entire memory space
- **Protection**: Memory space of each process are mapped to different physical address, isolating them from each other

File Management

- **Abstraction**: Files is a single contiguous logical entity
- **Protection**: Files can only be opened through system call, OS can prevent files from being opened for incompatible operations

18.2 Swap Files

Should swap files be considered as normal files?

- No. They require contiguous chunk of memory to speed up paging. Normal files may have their data stored at different parts of the secondary storage.

Relationship between Page size and Cluster Size

- Page size should be multiple of cluster size to allow efficient swapping of pages.

Should swap file be system-wide or per-process?

- System wide. Allows OS to allocate a contiguous chunk of secondary storage for it at the start. Also it is hard to predict user usage if we use per-process swap which makes allocation of memory in storage difficult.

18.3 Virtual Memory

Can threads from same process share same page table?

- Yes. Use the same virtual memory space as the process.

Is the memory address returned by `shmat()` likely to be at boundary of memory page?

- Yes. We need to set the translation (i.e. frame number) to the shared region different from other memory locations.

PTE size dependent on?

- RAM! e.g. 16GiB of RAM will require 16GiB / Page Size pages which affects the number of bits required to store all the pages
- Not affected by virtual memory

18.4 iTLB and dTLB

In modern Intel/AMD/ARM cores, there is often a separate TLB for data (dTLB) and for code(iTLB). The reason why they are separate are as follows:

- Instructions and data access **exploit different temporal and spatial locality patterns**. e.g. Instruction access are more sequential. As such separating them will allow them to be more specialized to serve its purpose with a more suitable replacement policy.
- 2 smaller TLB faster than 1 big TLB.
- dTLB > iTLB in size. Separating them allows for better sizing of each TLB.
- Mixing data and code will allow data PTEs to kick out instruction PTEs which will decrease efficiency of TLB,

18.5 Page size tradeoff

Many system uses 4KiB page size but some system allow larger page size of 2MiB or even 1GiB. What are the tradeoffs for that?

- Decreased TLB pressure (generally less PTEs for larger page size)
- Increased internal fragmentation
- Decrease space overhead of page tables
- Decreased latency of page-table walks

18.6 Why do general semaphores exist?

Since general semaphores allows multiple processes in the critical section it wouldn't solve the race conditions problem

- Yes. General semaphores do not solve the issue of data races. However, it is also not designed to solve data races problems
- Solves problems like safe distance problem, Dining philosophers (limited eater), Reader-s/Writers etc.

18.7 Base registers

How exactly does base registers help to speed up relocation of address?

- Without base/limit registers, OS has to fix every program before loading by adding offsets which takes time (**increases loading time**)
- Using registers eliminates the need for OS inspection of code which speeds up loading
- Hardware then hides latency of addition in the pipeline

18.8 Contiguous Allocation

In most cases, the memory size of a process is not known at compile time due to dynamic allocation. How does contiguous allocation deal with this?

1. If there are enough free space outside of its boundary then we can just use that free space to allocate dynamically allocated memory
2. In the worst case, we will have to shift the partitions around to consolidate bigger holes to fit the partition (compaction)

18.9 Compaction vs Defragmentation

- **Compaction**: only done when there isn't a single contiguous chunk of memory of a size that a process needs. Only option left.
- **Defragmentation**: Done to improve performance of file system. Good to do to speed up system but is non-essential.

18.10 Pages

Why are we splitting memory into pages? To answer this question we will consider 2 extremes:

1. Memory is split into logical bytes
   - Maximum flexibility but page table will be huge (super large overhead)
2. Memory has only 1 partition: used on a all-or-nothing basis
   - No flexibility but minimum overhead

Paging sits nicely between the 2 approach which 1) minimizes both internal and external fragmentation and also 2) balances between flexibility and practicality of management

18.11 TLB

Is OS aware of TLBs?

- OS needs to be aware of TLBs in case it needs to invalidate their content (e.g. upon context switch). Not needed if we keep info on PID in PTE.
- When 1 thread does malloc (which changes the page table) all the cores that run a thread of the same process must have their TLB invalidated.

18.12 Memory Allocation

What happens when there is no free frame in memory and a new page needs to be brought into memory?

- Multiple behaviors are possible (and user configurable)
  - However, an existing frame should never be *immediately* overwritten unless it is first saved to swap memory.
  - Overwriting would break the correctness of the program previously using that frame when it runs again!
- The kernel could be configured to do any of these (and more)
  - Panic and shut down the whole system (usually not what we want)
  - Immediately kill the process that made the request
  - Invoke the "out-of-memory (OOM) killer" to kill the 'best possible' processes
  - Perform page replacement to free up some frames

18.13 Inverted Page Tables

Is inverted table mapping from virtual to physical address? And the normal one is from physical to virtual?

The opposite is true:

- Direct ("normal") table: logical (or virtual)→ physical
- Inverted table: physical→ logical + process ID

Also does it store the frame number or the page number?

- It **stores** the **logical page number** (+PID);
- It is looked up by frame number!

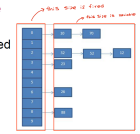
Can the table be indexed by (PID, page)?

- It cannot. We search it until we find the frame that contains the (PID, page).
- A hash-table can be built to allow indexing by (PID, page)

18.13.1 Fixed Inverted Page Table Size?

In that case why is inverted table size fixed since each entry can have many processes?

- The main table (what must be pre-allocated) is fixed
  - Linked list of (logical page, PID) pairs can be appended, e.g., through *separate chaining*



18.13.2 Inverted Page Table vs Hierarchical Page Table Memory Usage

Does inverted page table use more space than hierarchical page table?

- Depends:
  - In an empty system without any processes, it consumes more space
  - In a system with a lot of processes which allocate lots of memory, then the inverted table is much much smaller.

18.14 Directory Permissions

	ReadExeDir	WriteExeDir	ExeOnlyDir
`ls -l DDDD`	ok	nope	nope
`cd DDDD`	ok	ok	ok
`ls -l`	ok	nope	nope
`cat file.txt`	ok	ok	ok
`touch file.txt`	ok	ok	ok
`touch newfile.txt`	nope	ok	nope
`mv DDDD EEEE`	nope	ok	nope
`rm DDDD`	nope	ok	nope

18.15 File Systems

18.15.1 FAT16

Relationship between A: ∑(size of all files + current free space) and B: total capacity of all data blocks.

- A < B due to internal fragmentation of files since they may not use up the last logical block

18.15.2 ext

Should data blocks for a file be stored in same block group.

- Yes! Reduces fragmentation.

18.16 Page Replacement Algorithms and Working Set

- If LRU or 2nd chance algorithm is used as the page replacement algorithm, it gives a pretty good approximation of the working sets of processes as they all exploit locality principles.

18.17 `ls -i` command

What in the I-Node actually gets accessed?

- Just the directory entry for each file in the directory
- Directory entry is where the I-Node number and file name is stored.

18.18 Advantages of Virtual Memory Without Swap

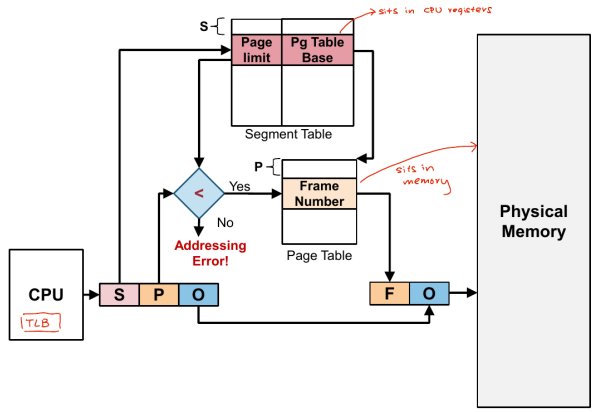
1. Separate address space protects and isolate applications from one another
2. Ability to share memory

18.19 First Page Fault

Most OS uses on-demand paging where data and code are brought into memory when needed. What is the first page fault that process encounters after a `exec()` syscall.

- **Text segment** will be the very first page fault
- Then system will load executable from disk

18.20 Segmentation + Paging



18.21 MMU Caches

- At the very least will have 1 MMU per core as it is in the critical path of execution (accessed multiple times per instruction)
- Common for modern processors to include MMU cache per level of page table (sized differently) → ensure that lower level page directory entries don't kick out the high level directory entries

18.22 File System decided by OS?

- In general file systems are defined and created by the OS but there could be different file system/OSes on the same hard disk

18.23 Why do we need file identifier?

- Note that file names do not have to be unique across the whole system
  - Just within a directory
- Unique file names are also useful for programs that reference other files!
  - Imagine you install a program on your system (`a.exe`) that references a config file in the same directory (`config.txt`) during execution
  - File identifiers are not decided by programs, but by the file system
  - `a.exe` will not know the file identifier of `config.txt`!
  - With file names, we can write code like `open(config.txt ….)`

18.24 Context Switching: Stack vs PCB

- When context switching, saving on the stack is faster since modern processors can push all registers onto the stack in a single instructions
- much faster compared to copying register values into PCB
- Context switch is also often preceded by interrupts, and handling of interrupts typically already store some context onto the stack → OS can piggyback on that

18.25 First Fit vs Next Fit

- First fit always search from beginning → holes from beginning will become too small and algorithm will have to search further down the list which increases search time
- Next fit alleviates this problem since the starting point of search changes → more uniform distribution of hole sizes and hence faster allocation

18.26 Overhead of various partitioning

Given free memory space = 16MiB (2^{24} bytes), starting address size of partition/ptr = 4 bytes, status of partition = 1 byte

- **Fixed size partitioning** (each partition 4KiB): # of partitions = $2^{24}/2^{12} = 2^{12}$, overhead = $2^{12} \times 1 = 4096$ bytes
- **Linked list**:
  - Min: Whole partition free - 1 node to keep track → 3×4 (start address, size of partition, next ptr) + 1 = 13 bytes
  - Max: # of partitions = $2^{24}/2^{10} = 2^1 4$ partitions. Overhead = $2^{14} \times 13 = 212{,}992$ bytes
- **Bitmap**: $2^{24}/2^{10}$ bits required = $2^{14}/8 = 2048$ bytes

18.27 Internal fragmentation of buddy allocation

- Min: Exact fit where request size = 2^x, 0% internal fragmentation
- Max: 50% when request size = $2^{k-1} + 1$ (1 byte above 2^x)
- Internal fragmentation cannot be >50%!

18.28 Use of working sets

- When a process changes from blocked to running we can load all the pages from the working set into RAM for that process. When it switches from running to blocked we can also migrate it to RAM to secondary storage
- Can also use total working set among all processes to decide whether we should allow more processes to run. If total > physical frame then may want to stop allowing processes to run due to thrashing.

18.29 Allocation Scheme Comparison

- Maximize for sequential access:
  1. Contiguous: No seek time to find the next block, and each block will be read sequentially as the disk head moves
  2. Linked List
  3. Indexed allocation: very large files may require multiple disk accesses to the indirect blocks
- Maximise Random Access Speed
  1. Contiguous: Just require and offset
  2. Indexed: Probably need to look at some levels of indirect blocks in order to find the right block to access (for very large files)
  3. Linked: Need to traverse through entire structure which requires many disk accesses
- Maximise Disk Capacity:
  1. Linked: Only need to store next pointer
  2. Indexed: Entire block needs to be stored
  3. Contiguous: A lot of external fragmentation!