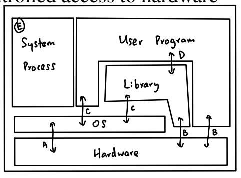
Jin Wei CS2106 Finals

Operating Systems

- OS is a software that runs in **kernel mode**: has full access to all hardware resources
- Other software that executes in **user mode**: has limited / controlled access to hardware resources → prevent User Program from crashing
  → NOT trying to minimise kernel run time

Monolithic OS

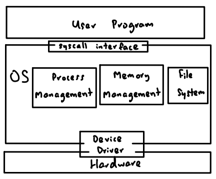
- One **BIG** program
- Most Unix variants, Windows
- Pros: Well understood; good performance
- Cons: Highly coupled components; very complicated internal structure



A: OS executing machine instructions
B: normal machine instructions by program / library code
C: syscall
D: library call
E: usually part of OS

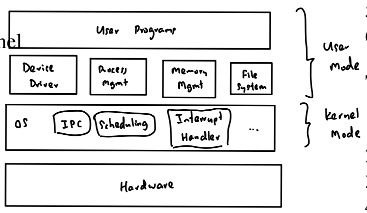
Layered Systems

- Generalisation of monolithic system
- Upper layers make use of lower layers
  - Highest: User Interface
  - Lowest: Hardware



Microkernel OS

- Kernel is very small and only provides basic and essential facilities (generally more robust)
- Inter-Process Communication (IPC)
  - i.e. File management is separate from Kernel
- Higher-level services run in User Mode

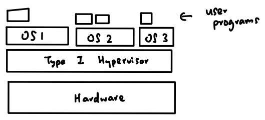


Client-Server Model

- Variation of microkernel
- 2 classes of processes
  - Client processes request service from server process
  - Server process built on top of microkernel
  - Client & Server can be on separate machines

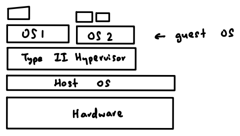
Virtual Machines

- Virtualisation of underlying hardware
- Managed by **Hypervisors**



Type 1 Hypervisor

- Similar to normal OS
- Provides individual virtual machines to guest OS



Type 2 Hypervisor

- Runs *in* a host OS
- Don't need to emulate everything, because there is a real OS below
- e.g. VirtualBox

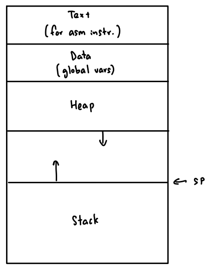
Process Management

- Memory
  - Storage for instruction and data (von Neumann)
  - Managed by OS
  - Normally accessed through load / store instructions
- Cache
  - Fast and invisible to software (purely hardware optimisation)
  - Duplicate part of memory for faster access
  - Usually split into instruction cache and data cache (Harvard Architecture)
- Fetch Unit
  - Loads instructions from memory
  - Memory location indicated by Program Counter (PC) register
- Functional Units
  - Executes instruction (read operands, compute results, write values)
  - Dedicated to different instruction types
- Registers
  - General Purpose Register (accessible by User Program)
  - Special Registers (PC, SP, FP, Program Status Word - status register)

Function Calls

Caller f() calls callee g().

1. Setup parameters
2. Transfer control to callee (via jump instruction)
3. Setup local variables
4. Store return value(s) if applicable
5. Return to caller



Stack Frame

- Return address of the caller
- Arguments for the function
- Storage for local variables
- Stack Pointer points to top of stack (first free location)
  - Either points *above first* (start of free space), or *at first* (aka last item on stack)
  - May change while a function is executing; data cannot be reliably referenced through constant offsets
  - Used for allocating space for local variables in middle of function (i.e. `int i = 0;` not at start of function)
  - Used for returning to caller function after frame pointer is restored
- Frame Pointer points to **fixed location** (doesn't change in function execution) in a stack frame
  - Other items are accessed as displacement (+-) from FP
  - Usage of FP is platform dependent (some don't have FP); FP is not actually required

One Way

These are done by compiler, not OS or HW.

Setup

1. Caller: pass parameters with registers and / or stack
2. Caller: save return PC on stack
3. Transfers control to Callee
4. Callee: save registers used by caller, old FP, old SP
5. Callee: allocate space for local variables of callee on stack
6. Callee: adjust SP to point to new stack top

Teardown

1. Callee: place return result on stack (if applicable)
2. Callee: restores saved registers, FP, SP
3. Transfers control to Caller
4. Caller: utilise return result (if applicable)
5. Caller: resumes execution

Saved Registers

- Register Spilling
  - Use memory to temporarily hold GPR values
  - GPR then used for other purpose
  - Then, restore GPR afterwards

Dynamically Allocated Memory

Acquiring memory space during execution time.

- Allocated only at runtime
  - Size not known during compilation time
  - Cannot place in Data region
- No definite deallocation timing
  - Can be explicitly freed by programmer at any time
  - Cannot place in Stack region

=> Store in Heap memory region.

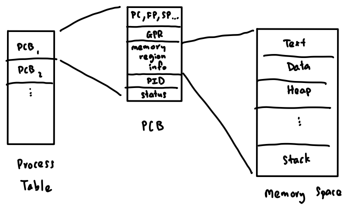
PID

PIDs are unique among processes.

- With m CPU cores, only <= m processes are in *running* state.
- Possibly parallel transitions, if m > 1

Process Control Block (PCB)

- Kernel maintains PCB for all processes, conceptually stored as one Process Table
  - Process Control Block or Process Table Entry



Context Switch

- Saves state of A into PCB for A
- Reloads state of B from PCB of B
- Hardware Context: GPR, PC, SP, FP, ...

Exceptions

- Caused by a machine level instruction
- **Synchronous** - occurs due to program execution
  - Division by zero, illegal memory access
- Effects of exception:
  - Exception handler is executed automatically
  - Similar to forced function call

Interrupts

- Caused by **external events**
- **Asynchronous** - occurs *independent* of program execution
  - Timer, mouse movement
- Effects of interrupt:
  - Program execution is suspended
  - Interrupt handler is executed automatically (with hardware help)

Interrupt Handling

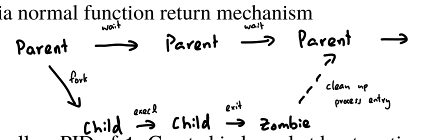
1. PUSH (PC) – hardware
2. PUSH (status register) – hardware
3. Disable interrupts in status register – hardware
4. Read Interrupt Vector Table (IVT) entry by using syscall number as index – hardware
5. Switch to **kernel mode** – hardware
6. PC <- `handler_XX()` – hardware
7. `handler_XX()` execution – software

IVT = table where OS stores addresses of all interrupt handlers, one of the first few things created by OS

System Calls

- API to OS
  - Have to change from *user mode* to *kernel mode*
- Many languages have support
  - Function wrapper - same name and same parameters
  - Function adapter - user-friendly library version (*printf* for *write* syscall)
  - `long syscall(long number, ...)` for other syscalls
- Library call places system call number & parameters in designated locations (e.g. registers)
- Library call executes a TRAP (a type of exception) to switch to kernel mode
- Then, appropriate system call handler is determined by IVT, and executed
- After it ends, control returns to library call, and switches back to user mode
- Then, returns to user program via normal function return mechanism

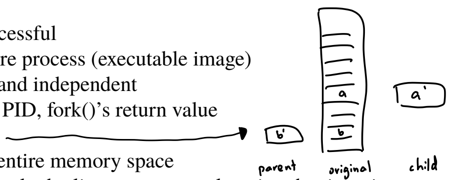
→ on Kernel Stack



Process Abstraction

Master Process called *init*, traditionally a PID of `1`. Created in kernel at boot up time.

- `int fork()`
  - returns PID (> 0) of newly created process (in parent)
  - returns 0 (in child)
  - returns < 0 if creation is unsuccessful
  - Spawns a **duplicate** of the entire process (executable image)
    - ∗ Data is copied (not shared) and independent
    - ∗ Differs only in: PID, parent PID, fork()'s return value
  - Memory Copy Optimisations
    - ∗ Very expensive to copy the entire memory space
    - ∗ Share the memory space & only duplicate a memory location that is *written* to.
  - Modern take: `clone()` (only performs partial duplication)



- `int execl(const char *path, const char *arg0, *arg1, ..., NULL)`
  - All arguments are strings.
  - Must end with NULL!
  - e.g. `execl("/bin/ls", "ls", "-l", NULL)`
- `exit(int status)`
  - 0 for normal (success) termination
  - non-zero for problematic execution
  - This function does not return (everything after it is **not** executed)
- `pid_t wait(int *status)`
  - status pointer (use *NULL* to ignore) is updated with the status
  - Blocks and waits for any *direct* child (does not block if there are **no** children)
  - Cleans up *remainder* of child system resources (those not removed on `exit()`)
  - `pid_t waitpid(pid_t pid, int *status, int options)`
    - ∗ Waits for specific child process
    - ∗ `WNOHANG` to *not* block
    - ∗ `WUNTRACED` to *block*
- `pid_t getpid()`
  - Gets PID of current process.

Zombie Process

→ Always become Zombie even if waited on.

- If child process dies before *wait*, is becomes a *Zombie* process
- Must always wait for child processes, otherwise zombies aren't cleaned up.
  - Cannot kill Zombies because they are already dead. (solution: reboot system)
  - May fill up process table (some older Unix implementations require a reboot to clear it up)

Orphan Process

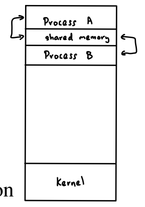
- Occurs when parent process terminates before child process.
- `init` process becomes "pseudo" parent

Process on Exit

- Most system resources are released on exit
  - e.g. file descriptors `fd`
- Not released (parent process must `wait()` to clean up information):
  - PID & status
    - ∗ Needed for parent-child synchronisation
  - Process accounting information: CPU time
  - Process table entry *may* still be needed
- Return from `main` implicitly calls `exit`

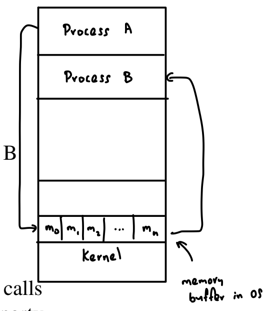
Inter-Process Communication

- memory space is independent, so need a mechanism to communicate across processes



Shared-Memory (Implicit) IPC

- Process A creates a shared memory region M
- Process B attaches M to its own memory space
- A and B can now communicate across M
  - M acts very similarly to normal memory region
  - Any writes to the region are seen by all parties that share the region
- At the end, detach after use; only one process will destroy M (can only be done if no process is attached)
- Pros
  - Efficient: OS needed only for initial setting up of shared regions
  - Ease of use: simple reads and writes to arbitrary data types (like regular memory)
- Cons
  - Limited to single machine (possible to use software abstraction to support this in distributed systems, but will be less efficient)
  - Requires synchronisation of resources: to avoid race conditions (if multiple processes write together)
- `int shmget(key_t key, size_t size, int shmflg)`
  - e.g. `shmget(IPC_PRIVATE, 40, IPC_CREAT | 0600)`
- `void *shmat(int shmid, const void *shmaddr, int shmflg)`
  - Returns a void pointer, need to cast to appropriate pointer type
- `int shmdt(const void *shmaddr)`
  - To detach: `shmdt((char*) shm)`
- `int shmctl(int shmid, int cmd, struct shmid_ds *buf)`
  - To destroy: `shmctl(shmid, IPC_RMID, 0)`



Message Passing (Explicit) IPC

Idea

- Process A prepares a message M and sends it to process B
- Process B receives the message

OS

- Messages have to be stored in Kernel memory space
- Sending and receiving is handled by OS through system calls
- **Direct communication**: need to know identity of other party
  - `send(P_2, Message)`
  - `receive(P_1, Message)`
- **Indirect communication**: only need to know address of mailbox / port (can be shared among many processes)
  - `send(MB, Message)`
  - `receive(MB, Message)`
- Need to declare in advance the capacity of the mailbox (when it is created)

Synchronisation Behaviours

Blocking (Synchronous)

- `send()`: Sender is blocked until message is received
  - Can use this to *rendezvous* (sync up processes)
  - No intermediate buffering by OS is required
- `receive()`: Receiver is blocked until message has arrived

Non-Blocking (Asynchronous)

- `send()`: Sender resumes operation immediately (if message buffer isn't full, either blocks or errors)
- `receive()`: If no message, resume without a message (rarely used)

Tradeoffs

- Pros
  - Applicable beyond a single machine
  - Portable, easily implemented on many platforms and environments; Parties on different platforms can communicate
  - Synchronisation is implicit
  - No data racing
- Cons
  - Inefficient because OS is involved with every message
  - Requires packing / unpacking the message into a supported message format

Unix Pipes

Unix Pipes is an example of a message passing IPC.

- Producer P, Consumer Q (P –> Q)
  - Writers wait when buffer is full
  - Readers wait when buffer is empty
  - Can have multiple readers / writers (although normal shell pipe has only one of each)
- Data is FIFO, like an anonymous file
- May be
  - unidirectional (half-duplex): one write-end, one read-end
  - bidirectional (full-duplex): both ends can read / write
- `int pipe(int fd[])`
  - returns 0 for success; non-0 for errors
  - `fd[0]`: reading end
  - `fd[1]`: writing end
- `close(fd[0])`, `write(fd[0])`, `read(fd[0], buf, sizeof(buf))`, where `char buf[N]`
  - Need to close the pipes not being used.
- Can redirect stdin / stdout / stderr to one of the pipes using `dup`

Unix Signal

segmentation violation

- `signal(SIGSEGV, signal_handler)`
  - `void signal_handler(int signo)`
  - Replaces the handler for Segmentation Fault with our own `signal_handler`

Process Scheduling

Concurrent Processes

- Multiple processes *progress* in execution (at the same time)
- Could be virtual parallelism (two processes sharing *one* CPU)
- Could be physical parallelism (multiple CPUs / ores)

Scheduling Algorithms

1. Batch Processing
   - No user interaction required, no need to be responsive
   - Mostly use non-preemptive scheduling
2. Interactive (or multiprogramming)
   - With active user interacting with system
   - Should be responsive: low and consistent in response time
3. Real time processing (not covered in CS2106)
   - Have deadline to meet
   - Usually periodic process

Should be **fair** (no starvation) and all parts of computing system should be **utilised**.

Scheduling Policies

- Non-preemptive
  - Process stays scheduled (in running state) until it blocks or gives up CPU voluntarily
- Preemptive (forcefully remove CPU from process)
  - Process is given a fixed time quantum (can block or give up early)
  - At end of time quantum, running process is suspended, another ready process is picked up

→ reduces average turnaround / waiting time

Batch

- Turnaround time for a task
  - Total time taken: finish time - arrival time (when it became ready)
  - Related to waiting time: time spent waiting for CPU
- Throughput
  - Number of tasks finished per unit time
- Makespan
  - Total time from start to finish to process **all** tasks
- CPU utilisation
  - Percentage of time CPU is used
- Waiting Time (for CPU)
  - turnaround time - work done - IO waiting time
- First Come First Serve (FCFS)
  - non-preemptive; does not starve (number of tasks in front of task X is always decreasing)
  - Convoy Effect: CPU bound Task A followed by many IO-bound Tasks X (Task A caused all those IO bound tasks to wait for CPU, when they could have completed first, and started waiting for IO)
- Shortest Job First (SJF)
  - Needs to know / guess **total CPU time** for a task in advance.
    - Predicted$_{n+1}$ = α * Actual$_n$ + (1 - α) * Predicted$_n$, where 0 <= α <= 1
  - non-preemptive; starves (consider a very long job, while shorter jobs keep coming in)
  - guarantees smallest average waiting time (use exchange argument)
- Shortest Remaining Time (SRT)
  - new job with shorter remaining time can preempt currently running job (good service for short jobs that arrive late)
  - preemptive version of SJF; starves (similar to SJF)
  - *does not* guarantee smallest average waiting time

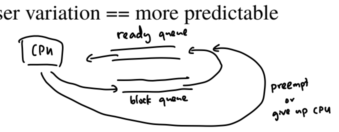
Interactive

- Scheduler takes over CPU through interrupts
- OS ensures **timer interrupt** cannot be intercepted by any other program
- Interval of Timer Interrupt (ITI)
  - OS scheduler invoked on every timer interrupt (typically 1ms to 10ms)
- Time Quantum
  - Execution duration given to process (can be constant or variable across processes)
  - Must be a multiple of ITI (commonly 5ms to 100ms)
  - Higher time quantum => less responsive / more waiting time / increased response time
  - Lower time quantum => more time wasted context switching (overhead)

Response Time & Predictability

- Response time
  - Time between request and response by system for a task
  - Preemptive scheduling algorithms are used to ensure good response time; scheduler needs to run periodically
- Predictability
  - Variation in response time, lesser variation == more predictable

Scheduling Algorithms

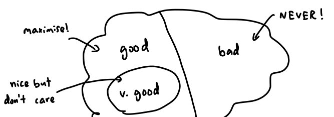


- Round Robin (RR)
  - preemptive version of FCFS; does not starve (<= `(n - 1) * time_quantum`)
- Priority-based
  - preemptive (higher priority preempts); or non-preemptive (higher priority wait for next run of scheduling)
  - starves (consider low priority job, while higher priority keeps coming in) - even worse for preemptive
    - can reduce starvation by preventing a process from running consecutively
  - **problem** of priority inversion (assume A, B, C is order of highest to lowest priority)
    - C starts & locks file
    - B preempts C
    - A arrives but file still locked (C didn't get to unlock it)
    - **solution**:
      - Temporarily increase priority of C to A until C unlocks it
      - In general, low-priority job inherits priority of higher priority process (when high-priority process requests lock held by low-priority resource)
      - Priority will be restored on successful unlock
- Multi-Level Feedback Queue (MLFQ)
  - attempts to minimise both response time for IO-bound and turnaround time for CPU-bound
  - preemptive; can starve (similar to priority), possible to prevent by detecting and increasing priority ↘ higher queue come in → preempts lower queue
  - Rules
    - Priority(A) > Priority(B): A runs
    - Priority(A) = Priority(B): A & B Round Robins
    - New job are given highest priority
    - If job uses up time quantum: reduce priority
      ↳ not reset (cumulative) if preempted before due
    - If job gives up / blocks before time quantum is up: retains priority
  - Bad for process which starts off CPU-bound, then transitions to IO-bound (fix by periodically resetting all processes to highest priority)
  - Can abuse by sleeping / blocking just before time quantum is up (fix by accumulating CPU usage across time quanta)
- Lottery Scheduling
  - preemptive; or non-preemptive; cannot starve (by probability)
  - in long run: process with X% of tickets will run X% of the time (by probability)
  - Response, newly created process can participate in next lottery; good level of control; simple to implement

Synchronisation Primitives

- Need to control interleaving accesses to shared resource
  - Allow all (or as many as possible) correct interleaving scenarios
  - Not allow any incorrect interleaving scenarios
  - Problem even for processes in different cores / threads
- Problems are only caused when a resource is read / written to.

Critical Section



Critical Section: code segment with race condition

- Mutual Exclusion (** most important)
  - Only one process can execute in critical section
- Progress
  - If no process is in critical section, one of the waiting processes should be granted access
- Bounded wait

  - After a process P requests to enter critical section, there exists an upper bound on number of times other processes can enter critical section before P
- Independence
  - Process **not** executing in critical section should never block other processes

Symptoms of Incorrect Synchronisation

- Incorrect output / behaviour: typically due to lack of mutual exclusion
- Deadlock: all processes blocked => no progress, typically due to circular dependency
- Livelock: processes not in blocked state, but keep changing stage (to avoid deadlock) => results in no progress, typically related to deadlock avoidance mechanisms
- Starvation: some processes blocked forever

Test and Set (Atomic Instruction)

- *TestAndSet Register, MemoryLocation* (atomic machine instruction)
  - Loads current content at MemoryLocation into Register
  - Stores a `1` into MemoryLocation

Semaphore

Can be initialised to any non-negative integer value. The following two operations are *atomic*.

- P() = Wait() = Down()
  - If S <= 0, blocks (goes to sleep)
  - Decrements S
- V() = Signal() = Up()
  - Increments S
  - Wakes up one sleeping process, if any (depends on library implementation)
  - Never blocks

$S_{\text{current}} = S_{\text{initial}}$ + (number of Signal() executed) - (number of Wait() **completed**, aka succeeded)

- General / Counting Semaphore
  - S >= 0 (S = 0, 1, 2, ...)
  - Only provided for convenience, can be mimicked by Binary Semaphore
- Binary Semaphore
  - S = {0, 1}
- Mutex (Mutual Exclusion)
  - Special case of binary semaphore
  - Used to guard a resource

The following can still cause deadlock (initially, P = 1, Q = 1). Consider order: A.Wait(P), B.Wait(Q), A.Wait(Q), B.Wait(P).

Condition Variable:
- Allows task to wait for certain events
- Ability to broadcast (wake all waiting tasks)
- Related to monitors

```
// Process A
Wait(P)
Wait(Q)
// code
Signal(Q)
Signal(P)

// Process B
Wait(Q) // Waits are swapped from Process A
Wait(P)
// code
Signal(P)
Signal(Q)
```

Semaphore: Critical Section

```
mutex = 1 // Binary Semaphore

wait(mutex)
// critical section
signal(mutex)
```

Semaphore: Safe-Distancing Problem

```
sem = N // General Semaphore

wait(sem)
// safely eat
signal(sem)
```

Semaphore: General Synchronisation

```
sem = 0

P1() {
  produce(X)
  signal(sem)
}
P2() {
  wait(sem)
  consume(X)
}
```

Semaphore: Barrier

```
arrived = 0, mutex = 1, waitQ = 0

Barrier(N) {
  wait(mutex)
  arrived++
  signal(mutex)
  if (arrived == N)
    signal(waitQ)

  wait(waitQ)
  signal(waitQ)
}
```

Producer-Consumer

- producers produce when buffer isn't full
- consumers consume when buffer isn't empty

```
notEmpty = 0, notFull = k, mutex = 1, in = out = 0

Produce(N) {
    while (TRUE) {
        Produce item
        wait(notFull)
        wait(mutex)
        buffer[in] = item
        in = (in+1) % k
        signal(mutex)
        signal(notEmpty)
        // Consume (for consumer)
    }
}
// Consumer same but flipped
```

Alternatively, use message passing with a fixed capacity. Everything is handled by system.

Reader-Writer

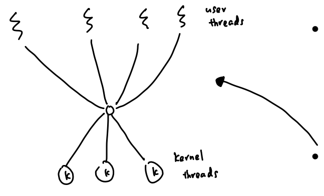
Writer has exclusive access, readers can access with other readers.

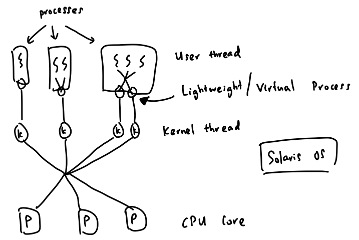
```
// Note: this version starves the Writer
roomEmpty = 1, mutex = 1, nReader = 0

Writer(N) {
    wait(revDoor) // to prevent starvation
    wait(roomEmpty)
    // modify data
    signal(roomEmpty)
    signal(revDoor)
}
Reader(N) {
    wait(revDoor) // immediately let reader through
    signal(revDoor) // if no writer
    wait(mutex)
    nReader++
    if nReader == 1 { // first reader waits
        wait(roomEmpty)
    }
    signal(mutex)
    // read data
    wait(mutex)
    nReader--
    if nReader == 0 { // last reader signals
        signal(roomEmpty)
    }
    signal(mutex)
}
```

Dining Philosophers

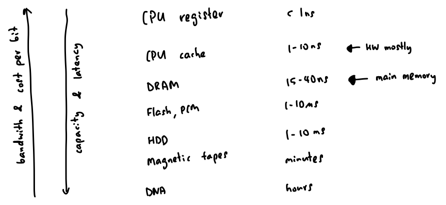
- five single chopstick between each pair of philosophers
- when he wants to eat, has to acquire both left & right chopsticks
- goal: starvation free, deadlock free, livelock free
- solution: allow at most $k-1$ of them to eat => deadlock is impossible

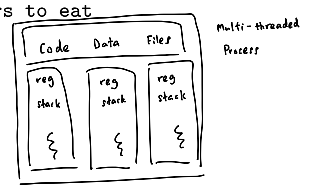
```
// Limited Eaters Solution
seats = k - 1

Philosophers() {
    while (TRUE) {
        Think()
        wait(seats)
        take(LEFT)
        take(RIGHT)
        Eat()
        put(LEFT)
        put(RIGHT)
        signal(seats)
    }
}
```





```
// Tanenbaum Solution
// no deadlock cos each philosopher only waits for own semaphore
int state[N]
sem_t mutex = 1, s[N]

Philospher(int i) {
    while (TRUE) {
        Think()
        take(i)
        Eat()
        put(i)
    }
}
take(int i) {
    wait(mutex)
    state[i] = HUNGRY
    safeToEat(i)
    signal(mutex)
    wait(s[i])
}
safeToEat(int i) {
    if state[i] == HUNGRY
        && state[LEFT] != EATING && state[right] != EATING
    {
        state[i] = EATING
        signal(s[i])
    }
}
put(int i) {
    wait(mutex)
    state[i] = THINKING
    safeToEat(left) // notify neighbours to eat
    safeToEat(right)
    signal(mutex)
}
```





Threads

- Processes are expensive (creation via `fork` & context switching & Inter-Process Communication is expensive)
- Single (multithreaded) process has multiple threads
  - shares
    - ∗ memory context: text (code), data, heap
    - ∗ OS context: process ID, other resources like files, etc… `owner | group | universe` (group = set of users who need similar access) - unique information
    - ∗ identification (thread ID)
    - ∗ registers
    - ∗ "stack" (but still share the same actual stack *space*)
  - thread switching involves only registers and "stack" (just updating FP and SP)
- benefits
  - economy: use less resources to manage than processes
  - resource sharing: most resources are shared, no need to IPC
  - responsiveness: appear more responsive
  - scalability: takes advantage of multiple CPUs (each thread in different core)
- problems
  - system call concurrency: i.e. handle concurrent `fopen` (syscalls must be thread-safe and guarantee correctness)
  - process behaviour: `fork()` (only one thread in child), `exit()` (all die), `exec()` (the rest die) -> (parenthesis is Linux behaviour, these are OS-dependent)

Thread Implementations

- User Thread
  - implemented as user library (e.g. Java threads); handled by runtime system in process; not OS dependent; more configurable and flexible; no need syscall
  - thread table in process (user mode)
  - Kernel **not aware** of existence of user threads: one thread blocked => all threads blocked; cannot exploit multiple CPUs
- Kernel Thread
  - implemented in OS (handled as syscalls)
  - thread table in kernel (kernel mode)
  - thread-level scheduling is possible
  - kernel may use threads for its own execution
- Hybrid Thread Model
  - have both Kernel and User threads
  - OS schedules on kernel threads only
  - user thread can bind to kernel thread
  - more flexibility: can control concurrency of any process / user

User Thread	:	Kernel Thread
1	:	1
M	:	1
M	:	M

Threads have hardware support on modern processors, with multiple sets of registers (GPRs, and special registers) to allow threads to run natively and in parallel on same core (known as Simultaneous Multi-Threading, i.e. Hyperthreading on Intel cores).

POSIX Threads: pthreads

- standard defined by IEEE, supported by most Unix variants
- defines API & behaviour, but not implementation, so pthread can be user OR kernel thread
- `pthread_create()` uses a function-pointer which it will call
- if `pthread_exit()` not explicitly called, pthread will terminate automatically at end of routine
- when one pthread fails / crashes, *might* bring down entire process

Memory Management

- Random Access Memory (RAM): access latency (approximately) constant regardless of address
  - can be treated as 1D array of Addressable Units (historically 1 byte)
  - each AU has a unique index - known as physical address
- Types of data (both grow / shrink during execution)
  - **Transient** data: valid only for limited duration e.g. function call (function parameters, local variables)
  - **Persistent** data: valid for duration of program or until explicitly deallocated (global variables, dynamically allocated memory)
- OS handles
  - *allocating*, *managing*, *protecting* memory space of process
  - provides memory-related syscalls, manages memory space for internal OS use
- `mmap` requires contiguous chunk of virtual address space

Memory Abstraction

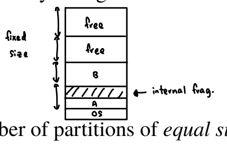
1. Use a special Base Register as base of all memory references, along with Limit Register (indicates range of memory space of current process)
   - **during compilation**, all memory references are compiled as *offset* from this register
     - Actual = BaseRegister + Addr
   - memory access is checked against the limit to protect memory space integrity
     - Check Actual < LimitRegister for validity
2. Logical Address
   - don't embed physical address, need a mapping of Logical => Physical

Contiguous Memory Management

Assumptions

1. Each process occupies a contiguous memory region (throughout the whole execution)
2. Physical memory is large enough to contain one or more processes with complete memory space
   - for multitasking: allow multiple processes in physical memory at the same time (switch between them)
   - when physical memory is full:
     - remove terminated processes
     - swap blocked processes to secondary storage

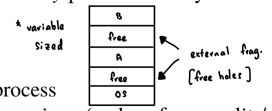
Memory Partitioning



Fixed-Size Partitions

- physical memory split into fixed number of partitions of *equal size*
- one process occupies one partition
- if process does not occupy whole partition
  - wasted leftover space can't be used by others, known as **internal fragmentation** (internal to process)
  - note: partition size must be large enough to contain **biggest** process
- pros: easy to manage, fast to allocate (can pick *any* partition as they're the same)

Variable-Size Partitions



- partition is created based on *actual size* of process
- OS keeps track of occupied / free memory regions (and performs split / merge as needed)
- free memory is called *holes* (tend to have a large number of holes due to creation / termination / swapping)
  - known as **external fragmentation** (external to process)
  - can merge neighbouring holes (but not good enough)

- more flexible, but hard to allocate memory, and OS needs to maintain more information

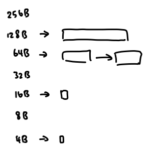
Allocation Algorithms To allocate for partition of size N, need to search for hole with size $M > N$. Splits the hole into N and $M - N$ (new hole).

- First-Fit
  - take *first* hole that fits (typically is fastest)
  - over time, holes close to beginning will become too small, resulting in longer searches
- Best-Fit
  - take *smallest* hole that's sufficient
- Worst-Fit
  - take *largest* hole
  - maximises potential for future processes

OS maintains a Linked List for partition information, stores: status (occupied or hole), start index, length of partition. OS also typically splits it into two LLs, free and occupied, for faster allocation.

Alternatively, use bitmap. Requires bit manipulation to retrieve status. Adjacent free spaces are "automatically merged".

Optimisation: Multiple Free Lists



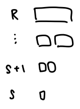
- separate list of free holes from occupied partitions
- keep multiple lists of different hole sizes
  - take hole from list that most closely matches request size
  - O(1) to get to "bucket"
  - if want 1B: search 1B, 2B, 4B, ..., break 4B into 1B (use this), 1B, 2B

Optimisation: Buddy System

- free block is halved recursively to meet request size: these blocks form buddies
  - buddies $\iff$ differ only in a single bit S (where size = 2^S)
- when buddy blocks are both free, merge back
- has little external fragmentation (compared to other algorithms)

Keep an array `A[0..K]`, where 2^K is largest allocable block size (may also have smallest allocable block size)

- each element `A[J]` is a LL which tracks free blocks with size 2^J
- indicated just by starting address

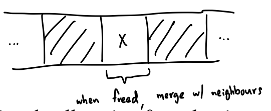


Algorithm to allocate block of size N (considered constant time)

1. Find smallest S, such that $2^S \geq N$
2. Access `A[S]` for free block
   1. If free block exists: remove it from LL and use it
   2. Find smallest R from `S+1 to K` such that `A[R]` has free block B
   3. For `(R-1 to S)`
      1. repeatedly split $B \to A[S...R-1]$ to new free blocks
   4. Goto step 2

Algorithm to deallocate block B (considered constant time)

1. Check in `A[S]`, where $2^S ==$ size of B
2. If buddy C of B exists and is free
   1. Remove B and C from LL
   2. Merge them into B'
   3. Goto step 1, set $B = B'$
3. Else, buddy of C isn't free: insert B to list in `A[S]`



Optimisations to reduce External Fragmentation

- efficient partition splitting, locating matching hole, partition de-allocation & coalescing
- when occupied partition is freed, *merge* with adjacent hole if possible
- occasionally perform *compaction*: move occupied partitions around to create bigger, consolidated holes (but very expensive)

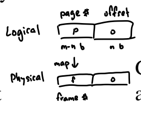
Disjoint Memory Management

Assumptions

1. Assumption 1 removed
2. Physical memory is large enough to contain one or more processes with complete memory space

Paging Scheme (* predominant)

- physical memory is split into regions of fixed size (physical frames)
  - frame sizes are decided by **hardware**
  - kept as powers of 2
- similarly, logical memory is split into regions of *same size* (logical pages)
  - physical frame size == logical page size (for now)
- at execution time, pages of process are loaded into *any available* memory frame
  - *logical* memory space remain contiguous
  - occupied *physical* memory space may be disjoint
  - requires Page Table (lookup table that translates Logical -> Physical)
    * PhysicalAddress = FrameNumber * sizeof(PhysicalFrame) + offset
      · offset = displacement from beginning of physical frame



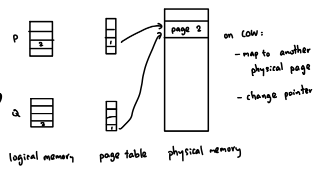
- observations
  - external fragmentation not possible, because every single free frame can be used
  - internal fragmentation is insignificant because only *one page per process* is not fully utilised, i.e. 10.5 (only 0.5 wasted)
  - clean separation of logical and physical space (more flexibility, simple address translation)

Implementing Paging Scheme

- pure software solution: OS stores page table information in PCB (or pointer to Page Table)
  - huge overhead because need to (access indexed page table entry) + (actual memory item)
- hardware support: Translation Look-Aside Buffer (TLB) - can have multiple per core (L1, L2, L3), and further split into data and instructions
  - acts as **cache** for Page Table Entries
  - very small (32 to 1024 entries), very fast (<= 1 clock cycle), very high hit rate (99%)
  - each process only requires very small number of entries (~1)
  - Logical -> Physical with TLB (use page number to query)
    * TLB-Hit: directly use frame number retrieved
    * TLB-Miss: need to access Page Table to retrieve, then update TLB
  - Average Memory Access Time = P(TLB hit) * latency(TLB hit) + P(TLB miss) * latency(TLB miss)
    * latency(TLB hit) = TLB read + read memory
    * latency(TLB miss) = TLB read + read table + read memory
  - on context switch: *flush* TLB entries to avoid correctness, security, and safety issues (don't store, because TLB is optimisation) + store pointer to Page Table in PCB

Memory Protection Included in each Page Table Entry (PTE):

- Access Right Bits
  - each PTE has writable, readable, executable bits
  - all memory accesses are checked against this in *hardware*
  - after Copy-on-Write: set child process's PTE's `W = 0` to trigger OS on write attempt; OS will allocate a new frame and copy contents over and give it to the child (update frame number & `W = 0` in child PTE)
- Valid Bit
  - because not all processes use the whole range
  - OS sets valid bits as memory is allocated
  - all memory accesses are checked against this in *hardware* (out-of-range is caught by OS)
- Dirty Bit
- Tag field to identify page in logical address space
- Offset field to identify location within page



Page Sharing

- Page Table allows several processes to share same physical memory frame
  - just point to the same physical frame number in PTE
- used for
  - shared code page (i.e. C standard library, syscalls)
  - implementing Copy-on-Write

Segmentation Scheme

Split logical memory space into disjoint physical segments: Text, Data, Heap, Stack. Set permissions for everything in a segment (as opposed to page-by-page). Enables easier growing / shrinking at execution time. Segments can be of different sizes.

Each memory segment:

- has a name: for ease of reference
- has limit: to indicate *exact range* of segment

All memory references specified as <SegmentID + Offset>, i.e. `<1 + 245>`.

- SegID used to look up <Base, Limit> of segment in Segment Table (small lookup table, don't need a TLB, just use 4 registers)
  - PhysicalAddress = Base + Offset
  - Offset < Limit for valid access

Pros & Cons

- pros
  - each segment is independent contiguous memory space
  - more efficient bookkeeping
  - segments can grow / shrink, and can be protected / shared independently
- cons
  - segmentation requires variable-sized **contiguous** memory segments
  - causes *external fragmentation*

Growing Stack and Heap *towards each other* is the best as it is the most flexible, and accounts for both heavy stack and heavy heap use cases.

Segmentation with Paging

Segmentation but solves external fragmentation issue. Each segment has own Page Table.

<Segment, Logical Page, Offset> -> <Physical Frame, Offset>

Virtual Memory

Both assumptions are removed.

- Logical Memory: 1-to-1 mapping between Logical <-> Physical
- Virtual Memory: Can have Virtual Memory » Physical Memory
- observations
  - secondary storage (HDD, SDD) » Physical memory capacity
  - some pages are accessed more often than others (principle of locality)
- split logical address into small chunks, only store *important chunks* in physical memory, store the rest in secondary storage (extension of Paging Scheme: store only some *pages* in RAM)
- use Page Table for Virtual -> Physical translation
  - new page types
    - memory resident (in physical memory)
    - non-memory resident (in secondary storage)
      - when accessing non-resident: **Page Fault** exception which OS will handle
    - denoted by *resident bit*

Accessing a Page

1. check Page Table (HW)
   1. if page X is memory resident: just access as usual. done.
   2. else: raise Page Fault exception
2. Page Fault: OS takes control (OS from here onwards)
3. locate page X in secondary storage
4. load page X into physical memory
5. update Page Table
6. go to step 1 to re-execute **same** instruction (but with the page now in-memory)

i.e. worst case requires 3 memory accesses (check Page Table, re-check Page Table, read data)

Transfer usually done by DMA (Direct Memory Access) Controller (a HW part), so CPU is free. Process will be changed to *blocked* state so other processes can run. Process will not be blocked if no Page Fault occurs!

Page Fault most of the time: known as *trashing* (not good)

- use *locality principles* to amortise cost of loading pages
  - temporal locality: likely to be used *again*
  - spatial locality: *nearby* memory addresses likely to be used soon, i.e. arrays
- higher levels of memory hierarchy stores "hotter" subset of lower level

Page Fault Algorithm Page Fault = not in Page Table (aka non-memory resident)

1. if no free memory frame, apply global / local replacement. else, just load a free frame
2. write out page to be replaced, if needed
3. locate page in secondary swap pages
4. load page into physical frame
5. update PTEs
   1. Search through all processes' PTEs (if have Inverted Page Table, use that instead)
      - Find the one with the evicted frame
      - Update ~~to~~ the resident bit to non-memory resident
   2. Use current process's Page Table
      - Update ~~to~~ the resident bit to memory resident
      - Update the frame number
6. update TLB if needed
7. return from TRAP

Demand Paging

- process starts with *no* memory resident pages (all resident bits are `F`) - even first instruction will Page Fault!
- only allocate page when Page Fault
- pros: fast startup, small memory footprint (minimise usage of in-memory)
- cons: process may appear sluggish due to multiple Page Faults, might cause other processes to trash

Page Table Structure

Problems with huge table: high overhead, Page Table can occupy several frames.

Page Table: Direct Paging Keeps all entries in single Page Table, which is stored in the PCB. With 2^p pages in logical memory space:

- p bits to specify one unique page
- 2^p PTEs with (physical frame number) + information bits (i.e. valid, permissions, etc...)
- wastes too much space!
  - 64-bit virtual address => 16 ExaBytes of Virtual space
    - $2^{64}/2^{12} = 2^{52}$ entries (if 12 bits per page)
      ↳ aka 4KB page size
  - physical memory 16GB => PhysicalAddress 34 bits
    - $2^{34}/2^{12} = 2^{22}$ entries (if 12 bits per page)
  - PTE size = 8B
    - Page Table size = $2^{52} * 8B = 2^{55}B$ per process (obviously can't fit)
- page tables must be contiguous in OS kernel memory for efficient memory access (impossible to achieve)

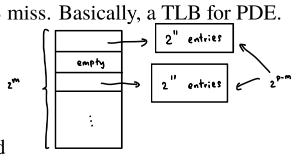
Page Table: 2-Level Paging: page the page table Use Page Directory: Page Table for Page Table. Only Page Directory is stored in PCB.

- if original page table has 2^P entries
  - with 2^M smaller page tables, M bits needed to uniquely identify a Page Table
  - each smaller page table can contain 2^{P-M} entries
- Page Directory has 2^M entries that point to the smaller page tables
- pros
  - enables page table structures to grow beyond size of a frame
    - don't need *entire* Page Table to be contiguous
  - can have empty entries in Page Directory that don't need to be allocated (saves a lot of memory)
- cons
  - requires two serialised memory addresses to get frame number (Page Directory + Page Table + Data)
  - might even Page Fault

Use TLB to eliminate Page Table accesses, but a miss still requires a longer Page Table Walk.

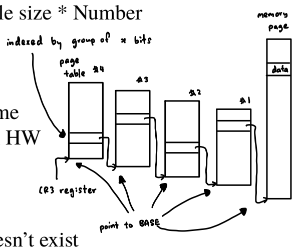
Use MMU caches (Memory Management Unit) to cache frequent *Page Directory Entries*, for speeding up Page Table Walk on TLB miss. Basically, a TLB for PDE.

Overhead with 2-Level Paging



- Number of PTEs fit in a page
- Number of entries in Page Directory
- Number of Level-2 Page Tables needed
- Overhead = Page Directory Size + Level-2 Page Table size * Number
  based on # pages allocated (NOT total)

Hierarchical Page Table (Multi-Level Paging)



- table in each level of hierarchy is sized to fit in a frame
- tables setup and wired by OS in SW, but traversed in HW
- use groups of 9 bits
  - each group is for each level of Page Table
    - each entry stores BASE of next level
  - invalid entry in any level means entire subtree doesn't exist
- some are cached in MMU

branching_factor = (page size) / (PTE size). Each level points to branching_factor number of next level. 2^k = branching_factor, where k is number of bits to represent a group

Inverted Page Table

- Keep mapping of Physical Frame -> <#, pid, page#>
  (logical page # in process)
  - pid + page# can uniquely identify a memory page
- pros: a lot smaller & one table for *all* processes
- cons: slow translation because requires linear scan
- can negate linear scan partially by hashing
- in practice: used as auxiliary structure to support certain queries

Page Replacement Algorithms

If no free physical memory frame during Page Fault, have to evict a free memory page. If dirty, have to write back.

$T_{access} = (1-p) * T_{mem} + p * T_{page\_fault}$, where p is probability of Page Fault, T_{mem} is access time for memory-resident page, T_{page_fault} is access time for Page Fault. Need $p < 0.000002$ to massively reduce access time.

- Optimal Page Replacement (magic algorithm)
  - replace page not needed for *longest period of time*
  - guarantees minimum number of Page Faults
- FIFO: evicts oldest memory page based on *loading time*
  - OS maintains queue (no need HW)
  - Belady's Anomaly: increased frames -> increased Page Faults because does not exploit temporal locality
- LRU: evicts *least recently used* memory page (good in practice)
  - two possible implementations (requires HW support)
    - PTE has "time-of-use": update whenever referred
      - replace the page with smallest "time-of-use"
      - problem: need to linearly scan, and "time-of-use" might overflow...
    - OS maintains "stack"
      - entries can be removed from anywhere in stack for updating (then pushed to top)
      - easy eviction: just pop from bottom of stack
      - hard to implement in HW (HW caches do something like this)
- Clock / Second-Chance - implemented with Circular Linked List in HW
  - maintain *reference bit* (1 = accessed since last reset, 0 = not accessed)
  - take the page if reference bit == 0

– degenerates into FIFO when all pages have reference == 1 (or all == 0)

* When replacing a frame, you take its position!

Frame Allocation

N physical memory frames, M processes that want frames

- allocation
  - equal allocation: each process gets N/M frames
  - proportional allocation: each process gets $(size_p/size_{total}) * N$ frames
- replacement
  - local replacement: select from *own process*
    - ∗ number of frames for each process is constant (stable performance between multiple runs)
    - ∗ thrashing is limited to one process
    - ∗ if not enough frames: hinders progress :/
    - ∗ can use up I/O bandwidth and still degrade performance
  - global replacement: select from *any process* (can be *other* processes)
    - ∗ allows self-adjustment between processes
    - ∗ one bad process can affect other processes!
      - · can have Cascading Trashing (Trashing page steals from others and causes others to thrash)
    - ∗ number of frames for each process can differ run to run

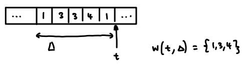
$\uparrow$ size of process

Working Set Model

Working Set = set of virtual pages that are currently resident in physical memory, is relative constant in a period of time = $W(t, \Delta)$, Δ is interval of time

- when Working Set is stable and well-defined: Page Faults are rare
- when transitioning to new Working Set: many Page Faults
- allocate enough frames for pages in $W(t, \Delta)$ to reduce possibility of Page Faults
  - transient region: working set changing in size
  - stable region: working set almost same for long time
  - Δ too small: may miss pages in current working set
  - Δ too large: may contain pages in different working set (wasted)

File System



File system provides an *abstraction* on top of physical media, *high-level resource management* scheme, *protection* between processes and users, *sharing* between processes and users.

- **self-contained**: information stored on media is enough to describe the entire organisation; should be "plug-and-play" (i.e. think thumbdrive, and other external drives)
- **persistent**: beyond lifetime of OS and processes
- **efficient**: provides good management of free and used space; minimum overhead required for book-keeping by OS

	Memory Management	FS Management
Underlying Storage	RAM	Disk
Access Speed	Constant	Variable I/O because of hard-disk rotate / seek
Unit of Addressing	Physical Memory Address	Disk Sector
Usage	Address space for process *Implicit* when process runs	Non-volatile data *Explicit* access: 'open', 'write'
Organisation	Paging / Segmentation	many different FS

File

File is smallest logical unit of persistent storage, created by process.

- **name**: human-readable name (different OS has different rules)
- **file type**: associated set of operations; possibly specific program for processing
  - regular files (ASCII files, binary files), directories (system file for FS structure), special files (character / block oriented)
  - can be distinguished by file extension (Windows), or embedded information / magic number stored at start of file (Unix)
- **size**: in bytes / words / blocks
- **protection**: access permissions
  - types of access
    - ∗ read: retrieve info from file
    - ∗ write: write / rewrite file
    - ∗ execute: load file into memory & execute
    - ∗ append: add new info to end of file
    - ∗ delete: remove file from FS
    - ∗ list: read metadata of file
  - access control list: list of user identities & allowed access types
    - ∗ very customisable; but requires a lot of info to be stored
  - permission bits
    - ∗ `owner | group | universe` (group = set of users who need similar access)
    - ∗ for Unix: `rwx`, use `ls -l` to see perms
      - · Minimal ACL = same as permission bits
      - · Extended ACL = Minimal + add (named users / group) `getfacl`
    - ∗ if no permission, `SIGSEGV` is sent to process

```
$ getfacl dir

user::rwx
user:sooyj:rwx      # permission for specific user
group::r-x
group:cohort21:rwx  # permission for specific group
mask::rwx           # permission "upper bound"
other::---
```

- **time / date /owner info**: creation / last modification time, owner ID, etc…
- **table of content**: info for FS to determine how to access file

File Data

- array of bytes, each with unique *offset* from file start
- fixed length records: array of records can grow / shrink
  - offset of Nth record = size of Record * (N-1)
  - has internal fragmentation
- variable length records
  - flexible, with no more internal fragmentation
  - but harder to locate a record

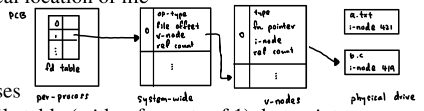
File Data: Access Methods

- Sequential Access (i.e. magnetic tapes, audio cassette)
  - must be read in order, from the start. cannot skip, but can rewind
- Random Access (byte-based access) - used by Unix & Windows
  - `Read(offset)`, `Seek(offset)`: must explicitly state offset
- Direct Access (random, but record-based instead)
  - used for fixed-length records, allows random access to any record directly
- generic operations
  - **create**: new file is created with no data
  - **open**: to prepare for further file ops
  - **read**: read data to file, usually from current position
  - **write**: write data to file, usually from current position
  - **repositioning / seek**: move current position (without read / write)
  - **truncate**: removes data between specified & end of file
- OS provides file ops as syscalls to provide protection, concurrent and efficient access (*some* are thread-safe)
- info kept for opened file (in OS)
  - **file pointer**: tracks current position in file
  - **file descriptor (fd)**: unique ID of file
  - **disk location**: actual file location on disk
  - **open / reference count**: how many processes has this file opened (so can tell when to remove from table)

File Table

Several processes can open the *same file*, *several different files* can be opened at the same time. Typically use **3 tables**:

1. **per-process** open-file table
   - track open files for a *process* (points to system-wide open-file table entries)
2. **system-wide** open-file table
   - track open files for whole *system* (points to V-node entry)
   - need to `close()` to prevent this table from being cluttered (and overflow)
3. system-wide **V-node** (virtual node) table
   - to link with file on physical drive
   - contains information about physical location of file



Process Sharing Files (Unix)

- file is opened twice from two processes
  - two entries in system-wide open-file table (with ref counts of 1), but point to same V-node (with ref count of 2)
  - requires different entries because offset can be different!
- two processes using *same* open file table entry (because of `fork`)
  - I/O changes offset for *both* processes
  - ref count = 2

Directory

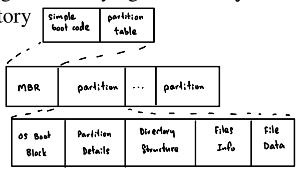
Directory (or folder) is used to provide logical grouping of files (*user view*), and to keep track of files (*actual system usage*).

- single level
  - everything is in "one" directory (e.g. Audio CDs)
- tree-structured
  - directories can be recursively embedded, naturally forming trees
  - can refer to a file via *absolute pathname* or *relative pathname* (follows from CWD)
- DAG
  - if a file can be shared, but only has one copy of actual content, would have multiple file names referring to same file content
  - Unix has **Hard Link** `ln`
    - ∗ both directories have *separate pointers* to the same file
    - ∗ only can delete file when ref count is 0 (and need to `rm` them separately)
    - ∗ data / permissions / everything else is shared, because it's the same file

- general graph
  - could have cycles, might have infinite looping
  - Unix has **Symbolic / Soft Link** `ln -s`
    - symbolic link is a *special link file*, when accessed, it will lookup the address
    - if delete original file, the *special link file* becomes a **dangling link**
    - deleting *special link file* is absolutely fine

Directory Permissions

- Read = can list files - reading `Directory` (affects tab auto-complete)
- Write = can create / rename / delete files - writing to / modifying `Directory`
- Execute = can use `Directory` as working directory



File System Implementation

Disk Organisation

Master Boot Record (MBR) at sector 0 with partition table, followed by one or more partitions, each can contain an independent FS.

FS generally contains: OS boot-up information, partition details (#blocks, number & location of free disk blocks), directory structure, files information, actual file data

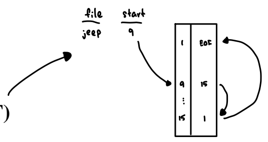
File Block Allocations

File Block Allocation: Contiguous

- only need to store table of <file, start, length>, where length is number of contiguous blocks
- makes use of sequential access, which is easy & faster (because hard-disk rotate / seek) than random access
- has external fragmentation, hard to expand file (might need to reallocate new space)

File Block Allocation: Linked List

- store table of <file, start, end>, where `start` is the starting block of the LL, `end` is used for appending to file
  - each disk block stores actual file data **&** next disk block number (as pointer)
- no more external fragmentation
- random access is slow because need to traverse linked list *in-storage* (and data might not be in same sector)
- overhead for storing pointer to next block
- less reliable (because pointer might be incorrect)



File Block Allocation: File Allocation Table (FAT)

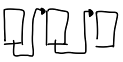
- move all block pointers into a single FAT table, in Kernel memory at all time
  - pulled & flushed on boot up & shut down respectively
- uses linked list idea, but doesn't store the pointers in the node, uses FAT instead
- faster random access because linked list is traversed in memory
- FAT can be huge & consumes a lot of memory space

File Block Allocation: Indexed Allocation

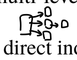
- don't track every file in the system, each file has a **index block**:
  - an *array* of disk block addresses
  - `IndexBlock[N]` = Nth block address
- less memory overhead, only index blocks of *opened files* are in-memory
- still has fast direct access
- limited maximum file size: maximum number of blocks for file == number of index block entries available
- still has overhead of index block

Indexed Block Allocation Variations

- linked scheme: keep linked list of index nodes



  - expensive because of travel cost
- multilevel index: similar to multi-level paging (can be generalised to any number of levels)



- combined scheme: combines direct indexing & multi-level index scheme
  - fast for small files; but still supports large files

Free Space Management

- need to maintain free space information, and update appropriately when allocating & freeing
- allocate: creating / enlarging file
- free: deleting / truncating file

Free Space Management: Bitmap

- each disk block represented by 1 bit (occupied = 0 by convention)
- easy to manipulate; but requires sequential scan to find first free block
- but, need to store in-memory for efficiency
- constant time updating because bitmap $\neq$ array

Free Space Management: Linked List

- each disk block contains:
  - number of free disk block numbers
  - OR pointer to next free space disk block
- easy to locate first block
- only first pointer is needed in-memory (but can cache the others)
- cons: high overhead (compared to just bits)

Directory Implementing

- keep track of files in directory (with file metadata), and maps file name to file information

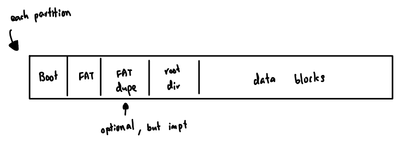
Directory: Linear List

- linear list, each entry contains:
  - file name + metadata
  - file information (or pointer to file information)
- locating files require doing a linear search through this list
  - inefficient for larger directories / deep tree traversal
  - can use cache to cache the latest few searches

Directory: Hash Table

- hash table of size N (usually resolved by chaining), hashed by file name
- fast lookup
- but relies on good hash function & need to expand table

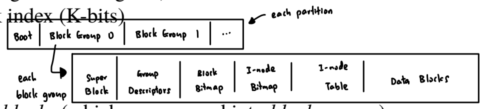
Case Studies



Microsoft FAT File System

FAT-K = use K bits

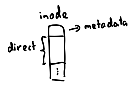
- file data are allocated to
  - number of data blocks / data block clusters (pointers kept in FAT)
  - file allocation information kept as *linked list*
- FAT ($2^K - 1$ entries, each entry is K bits)
  - one entry per data block / cluster
  - store disk block information (free?, next block (if occupied)?, damaged?)
    - `FREE` (unused block), `BlockNumber` (of next block), `EOF` (i.e. NULL pointer), `BAD` (unusable block - may be because of disk error)
  - OS caches this in RAM to facilitate LL traversal
- directory is stored as
  - special type of file
  - root directory `/` is stored in *special location* (the rest are stored in regular data blocks)
  - each file / subdirectory within directory is represented as *directory entry*
- directory entry
  - fixed-size 32-bytes: contains name + extension (8 + 3 characters), attributes (permissions, dir / file flag, hidden flag, ...), creation date + time, first disk block + file size, first disk block index (K-bits)

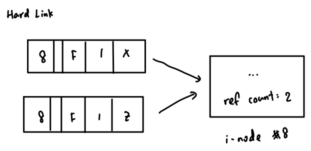


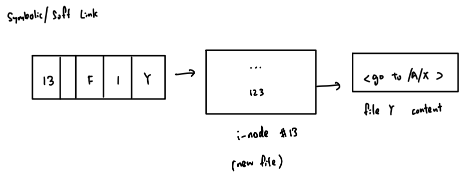
Extended-2 FS (Ext2)

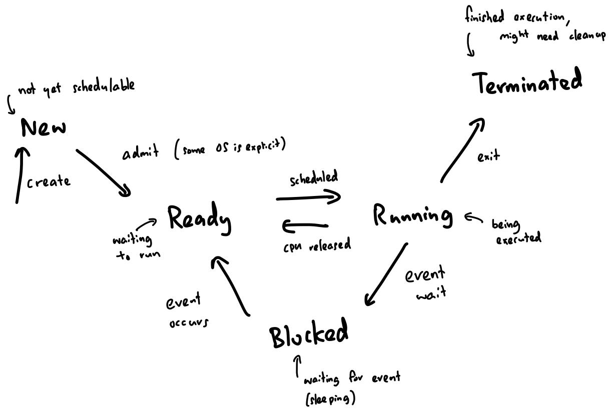
- disk space is split into *blocks* (which are grouped into *block groups*)
- **superblock** (duplicated *in each* block group for redundancy): describes whole FS, includes:
  - total i-node number, i-nodes per group
  - total disk blocks, disk-blocks per group
- **group descriptor table** (duplicated *in each* block group for redundancy): has a group descriptor for each block groups, contains:
  - number of free disk blocks, free i-nodes
  - *location* of disk block bitmaps & i-node bitmaps
- **partition information**:
  - block bitmap, i-node bitmap - keeps track of usage status of blocks in block group (1 = occupied, 0 = free)
  - i-node table: array of i-nodes (accessed by unique index)
  - contains only i-nodes of this block group
- each file / directory is described by *i-node* (index node) - each i-node is 128bytes, contains:
  - file metadata
  - 15 disk block pointers (assuming 1KiB per block)
    - first 12: direct blocks (fast for small files)
    - 13th: single indirect blocks
      - 1KiB = 2^{10} bytes, and 4 bytes for FAT-32 => stores $2^{10}/4 = 256$ entries, so 256 * 1KiB = 256KiB
    - 14th: double indirect blocks
      - 256^2 * 1KiB = 64MiB
    - 15th: triple indirect blocks (but can still handle large files)
      - 256^3 * 1KiB = 16GiB
- root directory `/` stored at special i-node number (e.g. 2)
- directory structure: data blocks of directory stores LL of directory entries for files / subdirectories in this directory, each entry contains:
  - i-node number for that file / subdirectory
  - size of directory entry (for locating next directory entry)

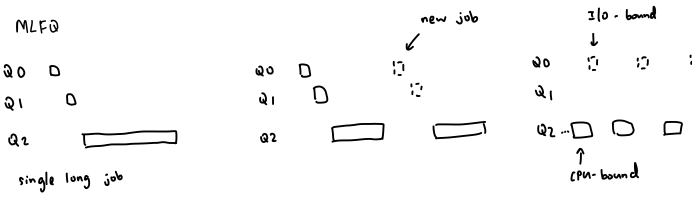
- length of file / subdirectory *name*
- type (file / subdirectory / other type)
- file / subdirectory name (up to 255 chars)











Misc

	#Bytes
1KB = 1KiB	2^{10}
1MB = 1MiB	2^{20}
1GB = 1GiB	2^{30}