Operating System

- Program that acts as an intermediary between a computer user and the computer hardware
- Must run in kernel mode

No OS	Minimal overhead, not portable, inefficient use of computer
Batch	Execute job sequentially, inefficient use of CPU
Time-sharing	Users use terminals to schedule jobs
Personal	Dedicated to user

Benefits:

1. Abstraction: same API to access all types of same category of hardware
2. Resource allocator: manages all resources like CPU, memory, I/O and arbitrate potentially conflicting requests for efficient and fair resource use
3. Control program: controls execution of programs, preventing errors and improper use

Kernel

- Deals with hardware issues, provides system call interface, and special code for interrupt handlers and device drivers

Type	Description	Pros	Cons
Monolithic (Linux)	Single program; does everything for OS to handle user <> hardware	Well understood and good performance	Highly coupled components, complex structure
Microkernel	Only implements essential functionality like IPC; higher-level services run outside of kernel and use IPC	Robust and extendible; better isolation and protection	Lower performance

Virtual machines

- Software emulation of hardware, virtualizing the underlying hardware
- Managed by hypervisor

Hypervisor (Virtual Machine Monitor): creates and manages virtual machines; is still an OS

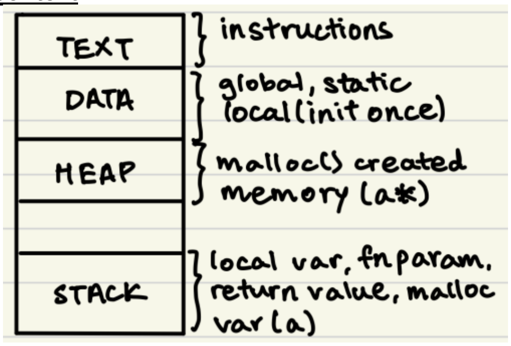
Type 1: bare metal (i.e. hypervisor is lightweight OS to create virtual machines)

Type 2: run on host OS

Hardware context

- General purpose registers, program counter, stack pointer, stack frame pointer

Memory context

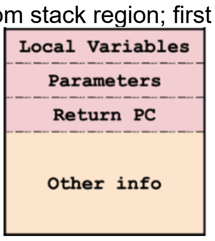

TEXT
DATA
HEAP
STACK
instructions
global, static local (init once)
malloc() created memory (a*)
local var, fn param, return value, malloc var (a)

Stack: information for function invocation in stack frame

- Deallocation timing; allow "jump" back in recursive stack

Stack frame: return address of the caller, parameters for the function, storage for local variables, etc.

Stack pointer: top/bottom stack region; first unused location



Frame pointer: fixed location in a stack frame; displace to access elements

Function call convention: non-universal ways to setup stack frame

CS2106 convention:

Caller:	Push $fp and $sp to stack
	Copy $sp to $fp
	Reserve space on stack for parameters by moving $sp
	Write parameters to stack using offsets of $fp
	JAL to callee
Callee:	Push $ra to stack
	Push current register values used in function
	Use $fp to access parameters
	Compute results
	Write result to stack (where $fp is)
	Restore registers saved
	Get $ra from stack
	Return to caller using JR $ra
Caller:	Get result from stack
	Restore $sp and $fp

Register spilling: move register value to memory temporarily when no registers available

- Registers have fast access to physical address lookup

Dynamically allocated memory

- Memory allocated during execution time using malloc()
- Size unknown during compilation time, no definite deallocation timing

Heap memory: store information for dynamically allocated values

- Trickier to manage because of variable size and allocation/deallocation timings

Process

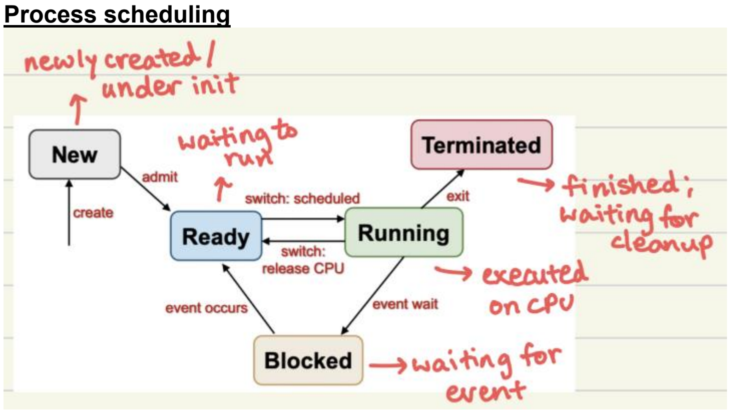
- Dynamic abstraction & information for executing program

Cores <> Processes: m CPUs has at most m processes

init: root process for all processes (PID = 1)

- Created by kernel at boot up time
- Like Supervisor in Elixir

Process scheduling



Zombie: child process terminates but parent does not call wait()

- Only stores PID and return value

Orphan: parent process terminates before child process

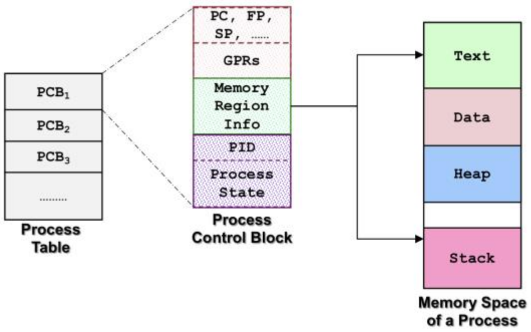
- init process becomes pseudo-parent (handles termination)

Too many processes: rebooting is best since commands may require an additional process which cannot be created

Admit: takes time to bring executable from disk to RAM and then load instructions

Process Control Block (PCB)

- Entire execution context for a process
- Maintained by kernel; stored under **process table**



System calls

- Must change from user mode -> kernel mode

Function Wrapper	Function Adapter
Syscall with library version with same name and parameters	User friendly syscall with less params/more flexible param

Mechanism:
1. User program invokes library call
2. Library call places system call number in a designated location like a register
3. Library call executes special instruction to switch from user to kernel mode (TRAP), saving the CPU state
4. Appropriate system call handler is determined by system call number (index), handled by dispatcher
5. System call handler executed
6. System call handler ends, restoring CPU state and returning to library call, switch back to user mode
7. Library call return to user program

Exception & Interruption

Exception: machine level instructions (checked exceptions)
- Synchronous and must be executed by an exception handler

Interrupt: external events, hardware related (unchecked exception)
- Asynchronous and suspends program execution, must be executed by interrupt handler

Handler: could do nothing
1. Save register/CPU state
2. Perform handler routine
3. Restore register/CPU
4. Return from disruption

fork()

```
int fork();
```

- Returns PID of child process (in parent) OR 0 (in child)
- Creates new process, duplicates the current executable machine; executes the remaining code below the fork()
- Memory in child is copied from parent (copy-on-write)

Implementation:
1. Create address space of child process (virtual address space)
2. Allocate p' = new PID
3. Create kernel process data structures (process table entry)
4. Copy kernel environment of parent process (priority)
5. Initialize child process context: PID = p', PPID = parent PID, 0 CPU time
6. Copy memory regions from parent (expensive, optimised with copy-on-write)
7. Acquires shared resources (open files, pwd)
8. Initialize hardware context for child process (copy registers from parent process)
9. Ready to run (add to scheduler queue)

Copy on Write: optimization for memory copying operation where memory locations that are written to are duplicated; noop if reading only

execl()

```
int execl(const char *path, const *char arg0, …, const *char argN, NULL);
```

- Replaces code but not process information (same PID)
- fork() + execl() on child process to separately run tasks

exit()

```
void exit(int status);
```

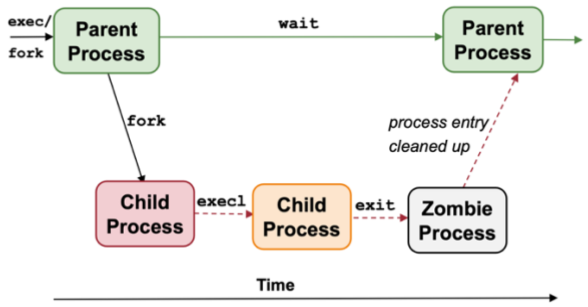
- Ends execution of process
- Status 0 success, !0 error
- Most system resources released like file descriptors
  - Left with PID & status, PCB
- Implicitly done when returning from the main() function

wait()

```
int wait(int *status);
```

- Parent waits for immediate child process (only) to terminate
- Returns PID of terminated child process
- Status is passed by address (wait(&status)); NULL if unused
- Blocking behavior on parent until one child terminates
- Clears remainder child sys resources, kills zombie processes
- execl() does not affect wait()
- If not children, terminate

Process creation



Concurrent Execution
- Multiple processes progress in execution at the same time
  - Virtual (illusion) or physical (multiple CPUs)

Time slicing: interleaving instructions for multiple processes
- Context switching between processes incur overhead

Process Environments

Batch processing: No user interaction, no need to be interactive, must have high system utilization	**Turnaround time:** finish – arrival **Waiting time:** time spent in queue **Throughput:** number of tasks finished per unit time **CPU utilization:** % time where CPU is not idling
Interactive: Active users interacting with system, must be responsive	**Response time:** first CPU use – arrival time **Predictability:** variation in response time
Real-time processing: Deadline to meet, usually periodic process	

Scheduling Algorithm Criteria
1. Process Environment
2. Fairness: all process gets fair share of CPU; no starvation
3. Utilization: all parts of system should be used (CPU, I/O)

Types of Scheduling
1. Non-preemptive (cooperative): process keeps running until blocked or gives up CPU voluntarily
2. Preemptive: process given fixed time quantum to run
   - Can be blocked or give up early
   - Forced to suspend once time quantum up
   - Ensures good response time as scheduler runs periodically
   - Periodic timer interrupt that cannot be intercepted; invokes scheduler
   - **Interval of Timer Interrupt (ITI):** OS scheduler invoked every timer interrupt (usually 1-10ms)
   - **Time quantum:** execution duration given to a process; in multiples of ITI; can be variable (usually 5-100ms)

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

Scheduling Algorithms
- Can be used in all scenarios but some better than the others

FCFS	*Batch processing, non-preemptive* Tasks stored in queue based on arrival time & tasks run till done or blocked Blocked tasks are removed from queue and when I/O completes, added back to queue Guarantees no starvation as every task runs Average waiting time can be improved and convoy effect present where long CPU bound task blocks I/O and vice versa Shortest average response time if jobs arrive in order of increasing lengths or all jobs have same completion time
SJF	*Batch processing, non-preemptive* Select task with smallest total CPU time (needs this info in advance/guess)

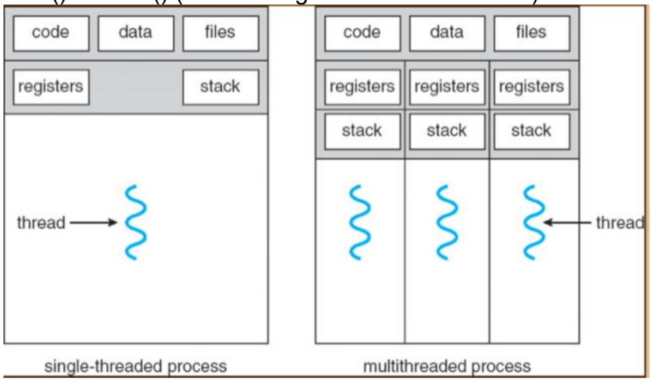
	Prediction algorithm from previous CPU-bound phases using exponential average $Predicted_{n+1} = \alpha Actual_n + (1 - \alpha) Predicted_n$ Minimizes average waiting time given a fixed set of tasks Starvation is possible due to bias towards short jobs
SRT	*Batch processing, preemptive* Variation of SJF using remaining time New jobs with shorter remaining time pre-empts currently running jobs; good for short jobs with late arrival
RR	*Interactive, preemptive* Like FCFS but preemptive so when time quantum lapses, task forced to give up CPU and placed to end of queue (after I/O) Response time (guaranteed) to be upper bounded by (n - 1)q with n tasks and time quantum q Larger time quantum yields better CPU utilization but longer waiting time, shorter time quantum yields larger context switching overhead but shorter waiting time Behaves like FCFS if job lengths are shorter than time quantum Performs worse than FCFS if job lengths are all the same and greater than time quantum (higher average turnaround time) or when there are many jobs and job lengths exceed time quantum (reduced throughput due to increased overhead)
Priority Scheduling	*Interactive, preemptive/non-preemptive* Assigns priority to processes and run highest priority first Preemptive: Higher priority process can preempt running process with lower priority Non-preemptive: Late coming waits for next round of scheduling Low priority process can starve if high priority keeps hogging CPU (worse for preemptive); resolved by decreasing priority or current process after each time quantum or giving current process a time quantum **Priority inversion:** lower priority preempts higher priority because resource is locked so higher priority is blocked
MLFQ	*Interactive, preemptive* Multiple queues with different priority levels, minimizes response time for I/O bound processes and turnaround time for CPU bound processes Highest priority queue must be empty before next queue is used Rules: 1. Priority(A) > Priority(B) -> A runs 2. Priority(A) == Priority(B) -> A and B runs in RR 3. New job -> highest priority 4. Job fully used time quantum -> reduce priority 5. Job gives up before time quantum -> retain priority **Change of heart:** process with lengthy CPU phase right before I/O phase sinks to the lowest priority and gets starved; periodically move all tasks to highest priority to fix **Gaming the system:** process repeatedly gives up CPU before time quantum lapses remains in high priority and starves other processes; accumulate total CPU use time across all quantums
Lottery Scheduling	*Interactive, preemptive*
	Give out "lottery tickets" to processes for system resources, randomly choose lottery ticket, winner gains resource; lottery tickets can be distributed to children Process holding X% of tickets has X% chance to win and use X% of resource Responsive as new processes can participate

Threading
- Units of work within a process
- Shares memory context (text, data, heap) and OS context (process ID, files, etc.)
- Uniquely identified by thread ID, registers, and stack

Thread switching: change hardware context like registers and stack
- Only memory region swapped is stack, everything else remains
- But overall stack space for the process stays the same

Benefits:
1. Economy: multiple threads in the same process requires less resources to manage
2. Resource sharing: threads share most of the resources of a process
3. Responsiveness
4. Scalability: take advantage of multiple CPUs

Problems:
1. System call concurrency: ensure that parallel system calls have no race condition
2. Process behavior: what happens when a single thread calls exit() or exec() (but Unix signals still can be used)



User Thread
- Thread implemented as user library (process handles thread related operations)
- Kernel not aware of user threads
- User thread with no kernel thread still runs on a single thread

Pros:
1. Can have multithreaded program on any OS
2. Thread operations are just library calls
3. More configurable and flexible

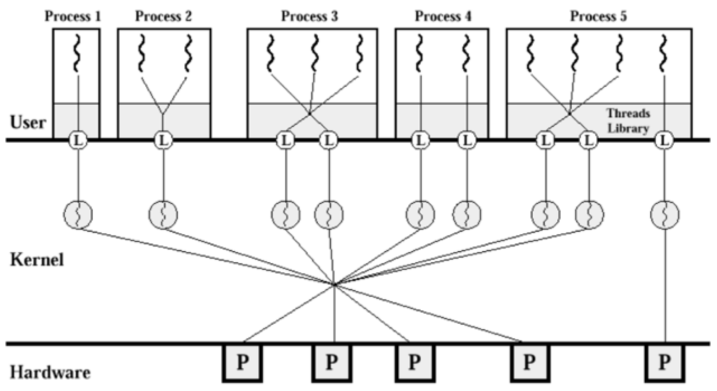
Cons:
1. OS not aware of threads and scheduling is performed at process level
   - If one user thread blocks, the entire process is blocked so all threads are blocked
   - Cannot exploit multiple CPUs

Kernel Thread
- Thread is implemented in the OS handled as system calls
- Kernel schedule by threads, not processes; may make use of threads for its own execution

Pros:
1. Kernel schedules on thread levels
   - Multiple threads can run for the same process and be non-blocking

Cons:
1. Thread operations are system calls that are slower and more resource intensive
2. Less flexible since kernel threads used by all multithreading programs

Hybrid Thread
- User threads bind to a kernel thread



Simultaneous Multi-threading

- Supply multiple sets of registers to allow threads to run natively and in parallel on the same core

POSIX thread

- Implemented as either user or kernel threads
- **pthread_t**: data type to represent thread id
- **pthread_attr:** data type to represent attributes of thread

pthread_create:

int pthread_create(pthread_t* tidCreated, const pthread_attr_t* threadAttributes, void* (*startRoutine) (void*), void* argForStartRoutine);
Return 0 if successful, !0 if errors
tidCreated: thread id for the created thread threadAttributes: control the behavior of the new thread startRoutine: function pointer to the function to be executed by thread argForStartRoutine: arguments for the startRoutine function

pthread_exit: pthread terminates automatically at the end of the startRoutine with return value defined by return statement

int pthread_exit(void* exitValue);
exitValue: value to be returned to whoever synchronize with this thread

pthread_join: wait for the termination of another pthread

int pthread_join(pthread_t threadID, void **status);
Return 0 if successful, !0 if errors
threadID: TID of the pthread to wait for status: exit value returned by the target pthread

pthread_mutex: binzary semaphore with pthread_mutex_lock() and pthread_mutex_unlock()
pthread_cond: conditional variable with pthread_cond_wait(), pthread_cond_signal(), and pthread_cond_broadcast()

IPC

Shared Memory	• Shared between processes • Process P1 creates shared memory M (shmget) and P2 attaches M to its memory space (shmat) • OS involved in creating and attaching memory region; easy to use as all operations are array operations • Requires synchronization and implementation is harder • Must detach M (shmdt) and destroy M (shmctl) after all processes detached
Message Passing	• Process P1 sends message M to P2 and P2 receives M (all as system calls) • Additional properties: identification of other party (naming scheme) and synchronization of send/receive operations • Messages are stored in kernel memory space **Direct communication:** sender/receiver of message explicitly name the other party • One link per pair of communicating processes • Identify of other party must be known **Indirect communication:** messages are sent to/received from message storage (mailbox/port) • One mailbox shared among multiple processes
	Synchronization behaviors for send() and receive(): blocking primitives (synchronous) or non-blocking primitives (asynchronous) **Advantages:** easier to implement on different processing environment and easier to synchronize **Disadvantages:** requires OS intervention so inefficient and harder to use due to limits on message size/format
Unix Pipes	• Share input/output between processes (producer/consumer) **Process communication channels:** stdin, stderr, stdout **Piping (A | B):** links the input/output channels of one process to another **Pipe in C:** circular bounded byte buffer with implicit synchronization (writer waits when buffer is full, reader waits when buffer is empty) **Unidirectional (half-duplex):** one write end, one read end **Bidirectional (full-duplex):** any end for read/write **System calls:** int pipe(int fd[]) close(int) -> side to close write(int, data, length) -> side to write to read(int, buffer, length) -> read from 0 for success, !0 for errors fd[0] is reading, fd[1] is writing • Must close the end not in use, otherwise, other processes might accidentally write/read to it **Redirecting communication channels to pipes:** use dup() and dup2() • Sets new file descriptor to refer to old as well (i.e. old and new are the same now) dup2(fd[0], STDIN_FILENO) — Redirect input to read from read end of pipe dup2(fd[1], STDOUT_FILENO) — Redirect standard output to write to write end of pipe dup2(file, STDOUT_FILENO) — Redirect standard output to write to contents of file
Unix Signals	• Asynchronous notification regarding an event sent to a process/thread ○ E.g. kill, stop, continue • Recipient of the signal must handle the signal via default handlers or user supplied handlers (signal(type, handler))

Race Conditions

- Execution outcome depends on the order in which the shared resource is accessed/modified (non-deterministic)
- General modification flow: load memory value into register, update register value, load register to memory
  - Race conditions happen when loading happens at the wrong time
- Caused by unsynchronized access to shared modifiable resources

Solution: designate code segment with race condition as critical section, only one process can execute in CS (synchronization)

Critical Section

```
// Normal code
Enter CS
// Critical work
Exit CS
// Normal code
```

Mutual exclusion: if process is executing in critical section, all other processes are prevented from entering the section
Progress: if no process is in a critical section, one of the waiting processes should be granted access
Bounded wait: after a process requests to enter the critical section, there exists an upper bound on how many other processes can enter the section before the process
Independence: process not executing in critical section should never block other process

Incorrect Synchronization

1. Deadlock: all processes blocked so no progress
2. Livelock: related to deadlock avoidance mechanism where processes keep changing state to avoid deadlock and make no other progress; processes are not blocked
3. Starvation: some processes never get to make progress in their execution as it is perpetually denied necessary resources

<u>Test and Set</u>

TestAndSet(Register, MemoryLocation)

- Machine instruction provided by processors to aid synchronization
- Load the current content at MemoryLocation into Register and stores 1 into MemoryLocation (treated as lock)
- Performed as single machine operation (atomic)
- Used to create spinlocks

```
void EnterCS(int* Lock) {
   // Cannot enter if lock value is 1
   while (TestAndSet(Lock) == 1);
}
```

```
void ExitCS(int* Lock) {
   *Lock = 0;
}
```

Busy waiting: keep checking the condition until it is safe to enter critical section which is wasteful use of processing power
Variants: compare and exchange, atomic swap, load link/store conditional

<u>Critical Section HLL Implementations</u>

Global lock (binary semaphore/mutex): Lock = 0

// disable interrupts
While (Lock != 0);
Lock = 1;
// CS
Lock = 0;
// enable interrupts
	// disable interrupts
	While (Lock != 0);
	Lock = 1;
	// CS
	Lock = 0;
	// enable interrupts

- Disable interrupts to prevent context switching entering the CS
- Buggy critical section can stall whole system
- Uses busy waiting
- Requires permission to disable/enable interrupts

Turn-based lock: Want = new int[2]

Want[0] = 1;
While (Want[1]);
// CS
Want[0] = 0;
	Want[1] = 1;
	While (Want[0]);
	// CS
	Want[1] = 0;

- P0 and P1 takes turns to enter critical section
- No deadlocks as P0 can enter CS without P1
- Other processes can also enter CS
- Deadlock can occur if both Want[0] and Want[1] are 1

Peterson's Algorithm: global lock + turn-based lock

Want[0] = 1;
Turn = 1;
While (Want[1] && Turn == 1);
// CS
Want[0] = 0;
	Want[1] = 1;
	Turn = 0;
	While (Want[0] && Turn == 0);
	// CS
	Want[1] = 0;

- Assumption: writing to Turn is atomic
- Uses busy waiting instead of going into blocked state
- Too low level; higher-level programming construct is desirable to simplify mutual exclusion and less error prone
- Not general: general synchronization mechanism is desirable, not just mutual exclusion
  - Semaphores
  - Conditional variable: allow a task to wait for certain event to happen and broadcast

<u>Semaphores</u>

- Generalized synchronization mechanism
- Only behaviors specified so different implementations are possible
- Blocks several processes and wake one or more sleeping processes
- Contains integer value (non-negative initially)

Wait(): if S <= 0, blocks, else decrement S
Signal(): increments S, if S > 0, wakes up one of the sleeping process; never blocks
Invariant: given initial S >= 0, then current S = initial S + # of signals - # of waits completed

$$S_{current} = S_{initial} + \#\, signal(S) - \#\, wait(S)$$

General/counting semaphore: S >= 0
Binary semaphore: S = 0 or 1
Mutex: S = 1

```
Wait(S);
// CS
Signal(S);
```

- Mutually exclusive, deadlocks cannot occur if properly designed (no interleaving Wait() instructions), no starvation

<u>POSIX Semaphore</u>

- Initialize semaphore with sem_init()
- Wait with sem_wait()
- Signal with sem_post()

<u>Memory Usage</u>

1. Transient data: valid only for a limited duration like parameters/local variables
2. Persistent data: valid for the duration of the program unless explicitly removed like global/constant variables/dynamically allocated memory

<u>OS Memory Management</u>

- Allocates memory space to new process, manage memory space, protect process memory space from each other, provide memory related system calls to process, manage memory space for internal use

<u>Memory Abstraction</u>

- Presenting logical interface for memory access
- Without memory abstraction, memory access can be straightforward and not require any conversion or mapping
  - But processes that occupy the same physical memory will conflict as both assume memory starts at 0

Address relocation: recalculate memory references when process is loaded into memory by adding an offset (where process starts) to all memory references

- Slow loading time and not easy to distinguish memory reference from normal integer constant

Base + Limit registers: use a base register as base of all memory references; during compilation, memory references are offsets from this register; during loading, base register is the starting address of process memory space

- Limit register indicates range of memory space of current process; protect memory space integrity using this register
- Incurs addition and comparison per memory access
- Different processes refer to different places in memory

Logical address: how process views its memory space that maps logical address to physical address

- Every process has a self-contained, independent logical memory space

<u>Progression of Memory Allocation</u>

Assuming	Topics
Contiguous & infinite	Memory partitioning, dynamic partitioning using allocation algorithms/buddy system
Disjoint & infinite	Paging, segmentation
Disjoint & finite	Virtual memory

<u>Memory Partitioning</u>

- Allocating contiguous memory region for single process

Fixed-size partition: physical memory is split into fixed number of partitions and each process occupies one partition

- Leftover unoccupied space is wasted (internal fragmentation) but too small partition cannot contain the largest process
- Easy to manage and fast to allocate (all good choice)

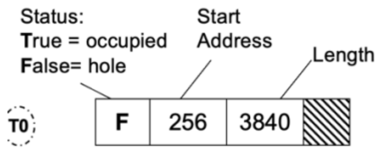
Variable-size partition: partition created based on actual process size; managed by OS

- Hole: free memory space
- Large number of holes is external fragmentation so merging the holes by moving occupied partitions can create larger holes
- Flexible and removes internal fragmentation
- Need OS to maintain information and time to locate appropriate region

<u>Dynamic Partitioning (Allocation Algorithms)</u>

- OS maintains linked list of partitions and holes
- Locate partition of size N by searching for hole with size M > N and using {first-fit, best-fit, worst-fit} and then splitting the hole into N (new partition) and M – N (leftover)
  - Merge adjacent holes if possible

- Compaction: move occupied partition around to consolidate holes but cannot be invoked too often



Dynamic Partitioning (Buddy System)

- Efficient partition splitting, locating hole, and partition de-allocation & coalescing
- Repeatedly split partition into half (original size is 2^k) and use the closest split to the required size

Implementation: array A[0..K] and each A[J] is a linked list that tracks the free blocks of size 2^J (indicates the starting memory location)

Allocation: given block size N, find smallest S where $2^S >= N$

- If A[S] has free block, remove block from free block list and allocate
- Else, find smallest R in [S+1,K] where A[R] has a free block B and then split B such that A[S..R-1] has a new free block (i.e. keep splitting the left if possible, else right)

Deallocation: free block B; check A[S] where 2^S is the size of B

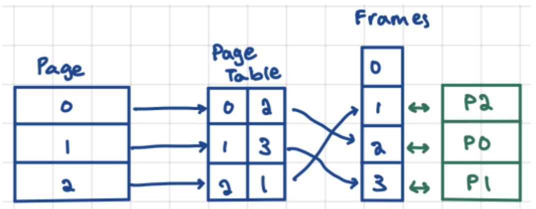
- If buddy C of B exists and is free, remove B and C from list and merge until buddy is no longer free
- Otherwise, insert B into list of A[S]

Buddy detection: buddies if S bit of B and C is a complement and the leading bits up to S bit is the same

Paging

- Split the physical memory into regions of fixed size (frame); logical memory is split into the same size (page) that are contiguous
- Pages of a process are loaded into any available memory frame
- Removes external fragmentation but internal fragmentation is possible
- Great flexibility and simple address translation
- Stored with PCB (this is equivalent to the page table)
- Stored in RAM

Paging scheme: maps logical page to physical frame using page table



- Requires 2 memory access (read page table, read memory)

Logical Address Translation: translates logical to frame address

- Physical address = frame # x frame size + offset

Paged memory reference: reading a page that is in memory

Translation Look-Aside Buffer (TLB)

- Used as cache for page table entries
- On miss, update TLB with found entry
- Part of the hardware context (cleared when context switching)
  - High initial miss after context switch
- Memory access time:

TLB hit rate x (TLB read + memory access) + TLB miss rate x (TLB read x 2 + memory access x 2)

Paging Memory Protection

Access right bits: indicates if page is writable, readable, or executable

- Check memory access against access right bit

Valid bit: attached to each page table entry, indicating if page is valid for access

- Set by OS when process is running
- Out-of-range access caught by OS, called page fault

Page Sharing

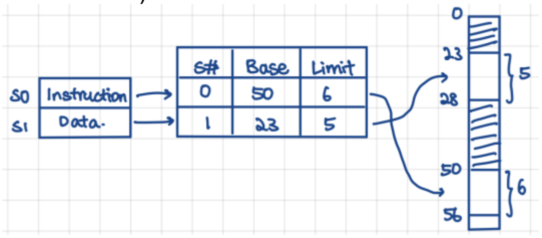
- Allow several processes to share the same physical memory frame, same physical frame

Shared code page: same code used by many processes

Implement Copy-On-Write: parent and child process share a page until one tries to change a value

Segmentation

- Separate memory regions into multiple memory segments (logical)
- Every segment has a name (segment id) and limit (range of segment)
- Segment mapped to contiguous physical memory region (base address and limit)



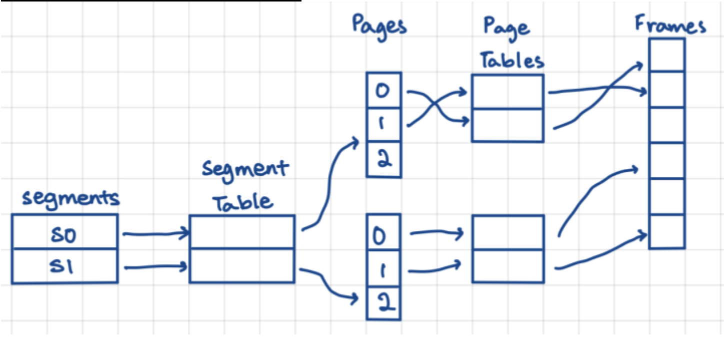
Logical Address Translation:

- Physical address = base + offset
- Check if base + offset < base + limit, else segfault

Pros: each segment is independent contiguous memory that can grow/shrink and be protected/shared separately

Cons: requires variable-size contiguous memory regions that can cause external fragmentation

Segmentation with Paging



- Segment grows by allocating new page and adding to page table

Virtual Memory

- Logical address maps to frames residing in physical memory (RAM) or secondary storage (hard disk)
- Secondary storage capacity >> physical memory capacity
- Secondary storage access time >> physical memory access time
- Adds new bit for memory resident (i.e. physical is memory resident, secondary is not)
- CPU can only access memory resident pages
  - Page fault otherwise
  - OS brings non-memory resident into memory
- Non-memory residents are stored in swap file

Swap file: special file that is not spread across different locations in storage to avoid affecting paging performance

- Should be implemented system wide so that pre-allocated swap is easier and processes can grow/shrink in swap usage

Access page X:

1. [Hardware] Convert virtual address to <page #, offset>
2. [Hardware] Search TLB for <page #>
   a. TLB miss: trap to OS
      i. [OS] Access PCB of process where <page #> is index
      ii. [OS] Check if valid, else segfault
      iii. [OS] Load PTE into TLB
      iv. [OS] Return from trap
3. [Hardware] Check page table: is page X a memory resident?
   a. If yes: access physical memory, done
   b. If not: [hardware/OS] trap to OS and page fault
      i. [OS] Locate page X in secondary storage
      ii. [OS] Load page X into physical memory
      iii. [OS] Load replaced page into secondary storage (use global/local replacement strategy)
      iv. [OS] Update page table
      v. [OS] Update TLB
      vi. [OS] Return from trap
4. [Hardware] Use <frame #><offset> to access memory

Thrashing: memory access results in page fault too many times

- Exploit locality principles to avoid

Temporal locality: memory address which is used is likely to be used again
Spatial locality: memory addresses close to used address is likely to be used

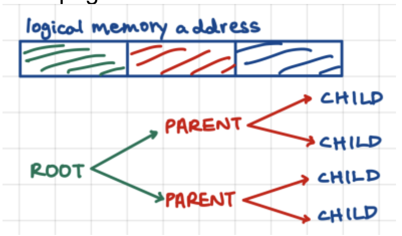
Demand Paging

- Process start with no memory resident pages and a page is allocated only when a page fault occurs

Pros: fast startup time for new process, small memory footprint
Cons: process may appear sluggish at the start due to multiple page faults (thrashing)

Page Table Structure

- Every page corresponds to one page table entry
- Too much overhead and fragmented page table

Direct paging	• Keep all entries in a single table • 2^p pages in logical memory space with p bits referring to one page, results in 2^p PTEs with physical frame number and additional information bits
n-level paging	• Split a direct paging into regions (i.e. subpage tables); each entry of a parent table points to another page table • Each subpage table has 2^{P-M} entries • Parent table contains 2^M entries that point to each subpage table  logical memory address ROOT PARENT PARENT CHILD CHILD CHILD CHILD • Segmenting memory: ○ Offset/Child: determined by power of page size (m); treat each m bit as a page (any less will result in inaccurate representation of the page) Further splitting: convert each page into a set of PTEs (i.e. subpages) so divide page size by PTE size (how many sub-pages can the 1 page represent) • Reduces overall space but can increase chance of page fault and delay time to load
Inverted page table	• Keep single mapping of physical frame to <pid, page #> instead • Given that valid entries will be << overhead of page tables • Ordered by frame number • Lookups require searching the whole table • Big savings by using one table for all processes

Page Replacement Algorithms

- Replace (evict/free) pages when no more free physical memory during page fault

On eviction: clean page is not modified and no need to write back; dirty page is modified and will need to write back
Memory access time: used to measure performance of page replacement algorithms

$$T_{access} = (1 - p) \times T_{mem} + p \times T_{page\_fault}$$

- p is the probability of failure and T is the access time for primary memory, secondary storage, or overall
- Minimise p as much as possible

Optimal Page Replacement (OPT)	• Replace the page that will not be used again for the longest period, guarantees minimum number of page faults • Not feasible to implement since "future sight" required • Used as benchmark for other algorithms
FIFO	• Evict memory pages based on their loading time, evicting the oldest first • OS maintains a queue of resident page numbers and remove the first page in queue if replacement is needed • Queue needs to be updated during page fault trap • Simple to implement as no hardware support is required • Belady's Anomaly: the more physical frames there are, the more page faults there are (counterintuitive) ○ Does not exploit temporal locality
LRU	• Replace the page that has not been used in the longest time, exploiting temporal locality • Closest approximation to OPT • Need to track "last access time" through hardware support ○ Use "time" counter that increases with every memory reference ▪ PTE stores "time of use" with that value and increments ▪ Need to search through all pages and time to use is increasing without bound (can overflow) ○ Maintain a stack of page numbers ▪ Move page from stack to top when page is referenced ▪ Replace the page at the bottom of the stack (no searching) ▪ Not a pure stack and hard to implement in hardware
Second-Chance Page Replacement (CLOCK)	• Modified FIFO with modified PTE to contain "reference bit" (1 = access, 0 = not accessed) • Algorithm 1. Oldest FIFO page is selected 2. If reference bit == 0, replace page 3. If reference bit == 1, give page a second chance a. Set reference bit to 0 b. Reset arrival time (FIFO) c. Move to next page and start at step 2 • The pointer shifts after replacing • Becomes FIFO if all reference bits is 1

Frame Allocation

- Distributing N frames to M processes

Local replacement: victim pages are selected among pages off the same process

- Keeps the frames allocated to a process constant and performance is stable between multiple runs
- Cannot expand pages allocated so thrashing can be a problem
- Keeps thrashing to one process only

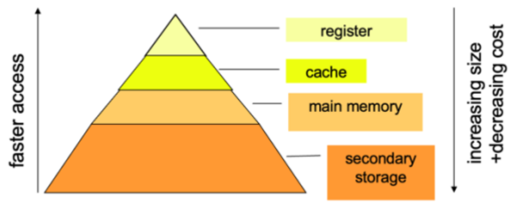
Global replacement: victim page can be chosen among all physical frames

- Processes can self-adjust to take more frames if needed
- Processes can hog the frames and the frames allocated is not consistent
  - Can cause other processes to thrash (cascading thrashing)

Equal	Every process gets N / M frames
Proportional	Processes are of different size, so each process gets $$\frac{size_p}{size_{total}} \times N$$
Working Set Model	• With the set of pages in frames, there will be no/few page faults until the process transits to a new locality • Working Set Model operates on this principle • Define Working Set Window Δ (interval of time) and W(t, Δ) is the active pages in the interval at time t • Allocate enough frames for pages in W(t, Δ) to reduce possibility of page fault • Choice of delta important ○ Too small can miss pages in current locality

	• Too big can contain pages from different localities

Memory Hierarchy



File System

- Abstraction over access to storage media (providing persistent storage)
- High level resource management scheme
- Protection between processes and users
- Sharing between processes and users

Criteria:

1. Self-contained: information stored on a media Is enough to describe the entire organization; plug and play
2. Persistent: beyond the lifetime of OS and processes
3. Efficient: provides good management of free and used space

Cluster

- Smallest amount of disk space that can be allocated to hold a file
- Page size is same as or multiple of cluster size to allow pages to be swapped out efficiently

Fixed cluster size: disk blocks are larger now but there is not changes to the file information
Variable cluster size: cluster size stored as part of the file block information
Several fixed cluster size: cluster size need to be stored but lesser number of bits needed since less to store
Pros:

1. Reduce disk accesses, overhead, and improve access speed for consecutive blocks
2. Larger cluster size reduces chance of external fragmentation
3. Smaller cluster size increases chance of internal fragementation

File System Abstractions: File

- Logical unit (abstraction) of information created by process
- Data (information structured in some way) and metadata (file attributes)

Metadata:

Name	Human readable reference to the file
Identifier	Unique ID for the file used internally by FS
Type	Indicate different type of file
Size	Current size of file (in bytes, words, blocks)
Protection	Access permissions, can be classified as reading, writing, execution rights
Time, date, owner	Creation, last modification time, owner id, etc.
Table of content	Information for the FS to determine how to access the file

File data structure:

1. Array of bytes: each byte has a unique offset from file start
2. Fixed length records: array of records that can grow/shrink
   - Jump to record via: size of record * (N – 1)
3. Variable length records: flexible but harder to locate record

Access methods:

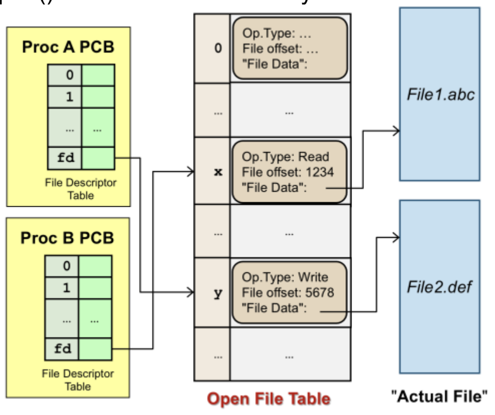
1. Sequential access: data read in order, start from beginning, cannot skip but can be rewound
2. Random access: data read in any order
   - Read(offset): every read operation states the position to be accessed
   - Seek(offset): move pointer to new location in file, all further reads based off new position
3. Direct access: allow random access to any record directly (hashmap)
   - Used for file with fixed-length records
   - 1 record == 1 byte in an array

Operations:

Create	New file is created with no data
Open	Performed before further operations
Read	Read data from file, usually starting from current position
Write	Write data to file, usually starting from current position
Repositioning	Aka Seek, move the current position to new location; no read/write is performed
Truncate	Removes data between specified position to end of file

Implementation: information needs to be preserved about opened file such as file pointer, disk location, open count
Open-file table: track information needed about open files

- System-wide: one entry per unique file
  - All open() calls create a new entry
- Per-process: one entry per file used in process; each entry points to system-wide table (PCB file descriptor table)
  - All open() calls create a new entry



Process sharing files: if two processes use different file descriptors, then I/O can occur at independent offset

- If two processes share the file descriptor, then I/O changes of one affects the other

File name: naming rules determine valid file name

- Naming rules determine valid file name
- Common rules:
  - Length
  - Case sensitivity
  - Allowed special symbols
  - File extension: Name.Extension

File type: each file type has associated set of operations and program for processing

- **Regular files:** contains user information
  - ASCII files: displayed/printed as is
  - Binary files: predefined internal structure that is processed by specific program (e.g. Java class file)
- **Directories:** system files for FS structure
- **Special files:** character/block oriented
- **Distinguishing:** using file extension (Windows) or embedded information (Unix)
  - Embedded information stored at beginning of file (magic number)

File protection: controlled access to information stored in file

- Restrict based on user identity
- User identity comprise of owner (creator of file), group (set of users with similar access to file), and universe (all other users in the system)
  - Permission bits: [rwx] [r--] [r--] [owner] [group] [universe]
- **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 file form FS
- **List:** read metadata of a file
- **Access Control List:** list of user identity and allowed access types
  - Minimal: same as permission bits
  - Extended: added named users/groups
- File access also requires syscalls so OS can intervene in case bad file access

File System Abstractions: Directory

- Logical grouping of files and keep track of files (i.e. actual system usage)

Structure:

1. Single level
2. Tree-structure
   - Directories recursively embedded into other directories
   - Absolute pathname: directory name followed from root of tree + final file
   - Relative pathname: directory name followed from current working directory
     - Set explicitly or implicitly by changing folder
3. Directed Acyclic Graph
   - File can be shared (main copy + links)

Hard Link	Symbolic Link
• File only • Directories A and B (hard link) have separate pointers to actual file F (on disk) • Low overhead since only pointers are added to directory • Deleting either A or B just deletes the pointer to the file, not the actual file • As long as 1 hard link exists, then the file is not permanently deleted • All files start with 1 hard link (original)	• File and directory • Link file, G, contains path name to F • When G is accessed, finds where F is and access F • Simple deletion as if B (symbolic link) deletes G, F is not deleted and if A deletes, then F is gone, and G remains (points to nowhere) • Larger overhead since special link file takes up actual disk space • If absolute path given, then can move symbolic file anywhere, relative file cannot work if moved

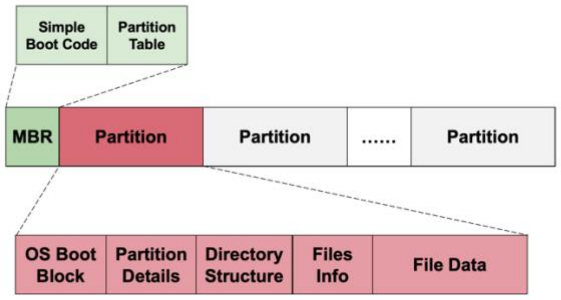
4. General Graph
   - Undesirable as hard to traverse (infinite loop) and determine when to remove file/directory
   - May be created by symbolic links due to linking directory

<u>General Disk Structure</u>

- 1-D array of logical blocks that are the smallest accessible unit (512 bytes to 4KB)
- Mapped into disk sectors and layout of disk sector is hardware dependent

<u>Disk Organization</u>

- Master Boot Record (MBR) at sector 0 with partition table
- Followed by one or more partitions with each partition containing an independent file system
- File system contains:
  - OS boot up information
  - Partition details like the total number of blocks and the number and location of free disk blocks
  - Directory structure
  - Files information
  - Actual file data



<u>File Implementation</u>

- Logical view of a file: collection of logical blocks
- File size != multiple of logical block size, last block can have internal fragmentation
- Keeps track of logical blocks, allow efficient access, and effectively utilize disk space
- Implemented in "File Info" and "File Data" section of partition

Block allocation algorithms:

Contiguous	• Allocate consecutive disk blocks to a file ◦ Simple to track and fast access (seek to first block) ◦ Causes external fragmentation and file size needs to be specified up front • File information stores the start and length of each file
Linked List	• Keep linked list of disk blocks and each disk block stores the next disk block number (pointer) and actual file data ◦ File information stores first and last disk block number ◦ Solves fragmentation problem ◦ Random access in a file is slow O(n), part of disk block used for pointer, and less reliable if pointer is invalid • Each file stores the start pointer, end pointer, and length of the file • File information stores first and last block • **Pointer information** is stored within the disk blocks themselves so the total size is actually slightly less
FAT file	• File Allocation Table, moving all block pointers into a single table that is **always in memory** ◦ Final block points to -1 ◦ Faster random access since you can traverse from anywhere ◦ FAT tracks all disk blocks in a partition which can be huge and consume valuable memory space • File size is fixed already and cannot be changed
Indexed (i-nodes)	• Every file has an index block which is an array of disk block addresses (in order) ◦ Lesser memory overhead since only index blocks of opened files needs to be in memory and fast direct access ◦ Limited maximum file size as max number of blocks is the number of index block entries and overhead for index block ◦ Increasing size ▪ Linked scheme: keep linked list of index blocks ▪ Multilevel index: like multi-level paging • Slightly smaller version of a FAT file that local to each file • Uses I-Nodes that store file metadata in file information Combined scheme: use both direct indexing and multi-level index scheme

<u>FAT Filesystem</u>

- In-memory system
- Each disk block contains some "files" (fixed number)
- Each file is either a file or folder
- Each file starts at a given address (root)
- Full file can be retrieved by using FAT to navigate (root -> next -> … -> EOF)
- Provides linear view of the contents of file/folder
- Hard linking affects the FAT entry that the hard link is going to point to
  - The hard link file then points to the same root disk block as the target file
- New disk blocks use free blocks
- 1:1 mapping for disk read and RAM read when using FAT (RAM for FAT, disk for file content)

<u>Free Space Management</u>

- Handled in "Partition Details" section of partition
- Maintains free space information for allocation and freeing

Bitmap	• Each disk block represented by 1 bit and occupied blocks are set to 0 ◦ Provide a good set of manipulations but need to keep in memory for efficiency reasons

Linked list	• Use linked list of disk blocks with each disk block containing the number of free disk block numbers and pointer to the next free disk block ○ Easy to locate free block and only first pointer needed in memory though other blocks can be cached for efficiency • High overhead introduced by storing metadata

Directory Implementation

- Implemented in "Directory Structure" section of partition
- Tracks the files in a directory with file metadata and maps file name to file information
- File information consists of file name and other metadata and disk block information
  - Either store everything in directory entry or store only file name and point to another data structure for rest of information

Linear list	• Each entry represents a file, storing file name and (optional) metadata and file information/pointer to file information ○ Requires linear search to locate file which is inefficient for large directories/deep tree traversals ○ Use cache to store latest few searches
Hash table	• Each directory contains hash table of size N ○ Hash file name into index K from [0, N-1] and HashTable[K] is inspected to match file name ○ Uses chained collision resolution (i.e. each bucket is a linked list) ○ Fast lookup ○ Limited size of hash table and depends on good hash function

File Operation: Create

1. Use full pathname to locate parent directory
   - Search for file name F to avoid duplicates
   - If duplicate found, file creation is terminated with error
   - Search can be done on cached directory structure
2. Use free space list to find free disk block(s), depending on allocation scheme
3. Add an entry to parent directory with relevant file information and disk information

File Operation: Open

1. Search system-wide table (open file table) for existing entry E
   - If found, creates an entry in P's table to point to E and return a pointer to the entry
   - Otherwise continue to step 2
2. Use full pathname to locate file F
   - If F not found, terminate with error
   - Load file information into new entry E in system-wide table
   - Creates an entry in P's table to point to E
   - Return a pointer to this entry

Read/Write Process

- Time taken to perform read/write operation

  Seek time + Rotational latency + Transfer time
- Seek time + Rotation latency >> Transfer time

1. Position the disk head over the proper track (seek time)
   - Average seek time: ~2 to 10ms
   - Total time for all possible seek over total number of possible seeks ~N/3 where N is the time for maximum seek distance
2. Wait for the desired sector to rotate under the read/write head (rotational latency)
   - Rotation speed of 4800 to 15000 rotations per minute or 12.5ms to 4ms per rotation
   - Average rotation latency: assume desired data is halfway around the track so 6.25ms at 4800 RPM and 2ms at 15000 RPM
3. Transfer the sector(s) (transfer time)x
   - Transfer size / transfer rate
   - Transfer size = size per sector * number of sectors
   - Transfer rate = 70 to 125 MBps

Disk Scheduling

- Scheduling disk I/O requests to minimize time taken
- Minimize seeking time

1. First Come First Serve (FCFS): sequential
2. Shortest Seek First (SSF)

Permissions

Operation	r-x	-wx	--x
ls -l <dir>	🟩	🟧	🟧
cd	🟩	🟩	🟩
ls -l inside folder	🟩	🟧	🟧
cat file.txt	🟩	🟩	🟩
touch file.txt	🟩	🟩	🟩
touch newfile.txt	🟧	🟩	🟧

- Read: can you read this list?
- Write: can you change this list?
- Execute: can you use this directory as the working directory?

File System Operations

Current capacity: number of free blocks indicated in bitmap
File path: walk through using the directory view
File access: find starting address position and then read from linked list data of file data
File creation: when allocating new memory to file, find all free blocks and use those as the base

Memory Management Help Sheet

Notations:

- K: number of bits for virtual memory address
- T: bits of Page Table Entry size
- S: bits of page size

Notes:

- A page is a block of memory (with offsets) that map to a physical memory frame
- Virtual memory can be larger than physical memory, just means those out of range is set to not valid
- 1-page maps to 1 frame (same size)
- N-level paging means N layers + 1 more for offset

Computations:

- # of frames needed = # of pages needed
  - Total size of process / page size
- Addressed at M byte level -> one virtual memory address points to M bytes
  - Total memory = 2^K x M
- # of pages (P) = $2^K / 2^S$
- Direct table size: P x 2^T
- N-level paging distribution
  - Offset: page size bits
  - Second last layer: $2^S / 2^T$ size (branching factor)
  - Remaining layers
    - If bits available > branching factor and N > 2, branch
    - Else, use the remaining space
- Inverted page table: number of frames available x T

Layers of OS

Layer	Abstraction	Protection
Process (CPU)	Illusion that processes execute on CPU all the time	Execution context of each process is isolated from each other
Memory	Illusion that processes own the entire memory space	Memory space of each process are mapped to different physical address, isolating them from each other
File	Files are a contiguous logical entity	Files can only be opened through system call; OS can prevent files from being opened for incompatible operations