CS2106 Cheatsheet AY24/25 —— @Jin Hang

OS: A program that acts as an intermediary between a computer user and the computer hardware.

- Provides system call interface
- Assumes total control/virtualization of the hardware
- Hide low level details → Efficiency and portability
- **Resource Allocator:** Multi-programs run simultaneously
  Control Program: Prevent Accidental/Malicious Errors
- Mem. management/Process Scheduler/Atomicity Control

Time-Sharing OS: Multics(1970s) - parent of Unix

- Multiple users to interact/job scheduling/Memory management

Kernel: Manages and protects CPU, Mem., and key resources

- OS is a user-friendly packaging of the kernel
- Kernel is a piece of **software**

Kernel Code has to be different than normal programs:

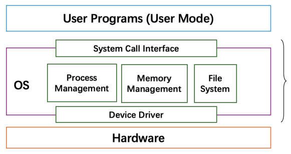
- NO use of system call in kernel code
- CANNOT use normal libraries
- NO normal I/O

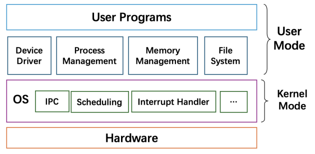
Kernel Mode: Most privileged mode of execution in **hardware**
User Mode limited privileges to prevent app direct access to hardware, ensuring system stability
Monolithic: Most Unix variants, Windows NT/XP, Linux, Ubuntu

- BIG special program, but Not compile big altogether
- Crush Code → OS crushed → May result in data leakage
- Everything keep in kernel mode → Less time penalty of switch

Microkernel: Kernel is Very small and clean. e.g. MacOS

- Provides only basic and essential facilities ⇐ Access by IPC





Layered Systems: Generalization of monolithic system
Client-Server Model: Variation of microkernel
VM(Hypervisor): Software emulation of hardware

- Virtualization of underlying hardware
- Normal (primitive) OS can run on top of the virtual machine
- Type 1 Provides individual virtual machines to guest OSes
- Type 2 Runs in host OS, Guest OS runs inside Virtual Machine

Memory: Storage for instruction and data
Cache: Duplicate part of the memory for faster access

- Usually split into instruction cache and data cache.

Fetch unit: Loads instruction from memory

- Location indicated by a special register: Program Counter (PC)

Functional units: Carry out different type of Inst. execution
Registers: Internal storage for the fastest access speed

- **General Purpose Register(GPR):** Accessible by user program (i.e. visible to compiler)
- **Special Register:** Program Counter (PC), Stack Pointer (SP), Frame Pointer (FP), Program Status Word (PSW)

Basic Instruction Execution

- An executable (binary) consists of two major components: Instructions and Data
- **Memory context:** Text, Data, Stack, Heap
  Hardware context: GPR, PC, SP, FP
  OS context: Process id/state, other resources like files, etc

Stack Memory Region: Store information function invocation.
Stack Frame: describes information of a function invocation

- Contains Return address of **Caller**

Stack Pointer: Indicate the top of stack region (first unused loc)
Frame Pointer: To facilitate access of various stack frame items

- Points to a **fixed** location in a stack frame

Stack Frame Setup / Teardown Example

1. On executing function call, Caller
   - Pass arguments with registers and/or stack
   - Save Return PC on stack
2. Transfer control from caller to callee, Callee:
   - Save registers used by callee. Save old(caller's) FP, SP
   - Allocate space for local variables of callee on stack
   - Adjust SP to point to new stack top
3. On returning from function call:
   - Callee: Restore saved registers, FP, SP
   - Transfer control from callee to caller using saved PC
   - Caller: Continues execution in caller

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

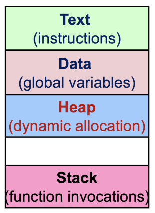
- 100 stacks, SP in the system (one per process)
- 8 stack pointer registers are in the system (one per core)
- Up to $100 \times 20 = 2000$ stack frames in the system
- 100 FP in the system (one per stack)
- 8 frame pointers registers in the system (one per core)

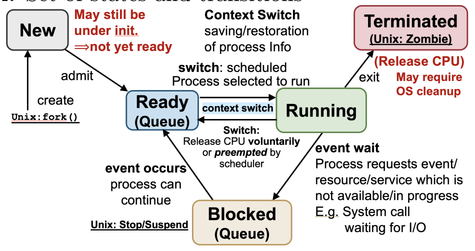
Dynamically Allocated Memory: acquire memory space during execution time(C: `malloc`, Java/C++: `new`)

- Allocated only at runtime ⇒ × in Data region
- No definite deallocation timing ⇒ × in Satck region
- heap memory region

Process: An abstraction to describe a running program

- **Process Model:** Set of states and transitions





- 1 CPU: 1 p. in running state, conceptually 1 trans. per time
  m CPUs: m process in running state, possibly parallel trans.
- Different processes may be in different states

Process Control Block (PCB), Process Table Entry

- The entire execution context for a process
- **Kernel** maintains PCB (one table) for all processes.

System Calls

1. Application Program Interface (API) to OS
   - Provides way of calling facilities/services in **kernel**
   - Need Mode Switch: User ↔ Kernel
   - OS dependent (Unix: POSIX, Win: Win API)
2. Unix Sys. Calls in C/C++ program invoked **almost directly**
   - Function Wrapper: Library version with same name & para.
   - Function Adapter: User Friendly Library version

Library Calls are provided individual languages. Some may wrap around one or more system calls.

General System Call Mechanism

1. User program invokes the library call
2. Library call places system call number in a designated location
3. Library call executes a special instruction to **switch** from user mode to kernel mode (TRAP)
4. Determine appropriate system call handler (by a dispatcher)
   - Using the system call number as index
5. System call handler is executed: Carry out the actual request
6. System call handler ended
   - Control return to the library call, kernel mode → user mode
7. Library call return to the user program: via normal function return mechanism

Exception: Executing a machine level inst. ⇒ handler execute

- **Synchronous:** Occur due to program execution

Interrupt: Usually hardware related(Timer), by external events

- Asynchronous: Events occur independent of program execution
- Program suspended → **handler** executed (Mode Switch)

Handler: Save Reg./CPU state → Perform handler routine → Restore Reg./CPU → Return from interrupt(Program execution resume May behave as if nothing happened)

Process Abstraction in Unix

`int fork()` Create an exact duplicate of parent, data not shared

- PID ← parent process
- 0 ← child process
- Child differs in: Process Id, Parent (PPID), `fork()` return value

`int excel(const char *path, const char *arg0, ..., NULL )`

- `NULL` To indicate end of argument list
- replace current executing process image with a new one.
- PID and other information still intact

`void exit(int status)` Status is returned to the parent process

- 0 ← Successful Execution
- `!0` ← Problematic
- Some process resources not releasable when exit
  PID & status needed for parent-children synchronization
  Process accounting info e.g. cpu time ⇒ P table entry still need

`init`: Special initial process created in kernel at boot up time

- `fork()` creates process tree: init is the root process

`int wait(int *status)` Parent blocks until at least one child ter.

- Returns the PID of the terminated child process
- `status`: exit status of terminated child process, can be `NULL`
- The call cleans up remainder of child system resources not removed on `exit()` ⇒ Kill zombie process

Zombie Process: Once process exit → Zombie

- Cannot kill zombie ⇐ process already dead
- Cannot delete all process info ⇒ Need to call `wait()`

1. Parent terminates before Child: Child become **Orphan**, `init` be "pseudo" parent → exit → send signal to `init` → call `wait()` ⇒ A process can both be an orphan and zombie
2. Child terminates before Parent but parent did not call wait: Child process become a zombie process → Fill up process table.

Concurrent processes: virtual/physical parallelism
Process Behavior: A typical process goes through phases of

- **CPU-Activity:** Compute-Bound Process
- **IO-Activity:** IO-Bound Process

Processing Environment Batch/Interactive/Real time
Criteria

- Fairness: fair share of CPU time (no starvation)
- Balance: All parts of the computing system should be utilized

Process Scheduling

1. Non-preemptive (Cooperative) A process stayed scheduled until it blocks / give up the CPU voluntarily
2. Preemptive: Given a fixed time quota to run(Possible to block or give up early) → Another process get picked

Scheduling Step

1. Scheduler is triggered (OS takes over)

2. Context switch if needed: saved context and placed on queue
3. Pick a suitable process P to run by Algo.
4. Setup the context for P and run P

Batch Processing - Can still have IO
- No user/No interaction/Not responsive/Non-preemptive
- **Turnaround time:** Total time taken [finish - arrive time]
- **Waiting Time**: Turnaround - (CPU+IO) Utilize Time
- **Throughput:** Number of tasks finished per unit time
- **CPU utilization:** % of time when CPU is working on a task

1. **First-Come First-Served: FCFS**
- No starvation: # of Tasks ↓⇒ task X gets its chance eventually
- Reordering can ↓ average waiting time
- Convoy Effect: if CPU-Bond Task followed by IO-Bond Task X

2. **Shortest Job First: SJF** ⇒ May Starvation
- Select task with the smallest total CPU time
- Need to know total CPU time for a task in advance
  Prediction: $\text{Predicted}_{n+1} = \alpha\text{Actual}_n + (1-\alpha)\text{Predicted}_n$
- Given a fixed set of tasks: Minimizes average waiting time

3. **Shortest Remaining Time: SRT** ⇒ Preemptive, May **S**
- Select job with shortest remaining (or expected) time

Interactive(Multiprogram.) active user/responsive/consistent
- Preemptive scheduling to ensure good resp. time
- Scheduler run periodically
- **Response time:** Time between request and response by sys
- **Predictability:** ↓ variation in response time = ↑ predictable

Timer interrupt: Interrupt goes off periodically (by hardware)
- OS ensure timer interrupt not be intercepted by other program
- Timer interrupt handler invokes scheduler

Interval of Timer Interrupt: OS scheduler is triggered every timer interrupt (1ms to 10ms)
Time Quantum: Execution duration for process (TQ%ITI==0)

1. **Round Robin (RR):** Tasks in FIFO queue
- Task in → Run until end → Place at the end of the Q
  Blocked task will be moved to other queue to wait for request
  When blocked task is ready ⇒ placed at the end of queue
- Quantum ↑ ⇒ Better CPU utilization but longer waiting time

2. **Priority Scheduling** Assign priority value → Select highest
- Preemptive/Non-# version: preempts/Late coming wait current
- May starve: High priority process keep hogging CPU
- Hard to control exact amount of CPU time given to a process
- **Priority Inversion:** H is blocked waiting on a src for L. M can progress and complete before H as L cannot preempt M to release the src. (Need at least 3 different priority)

3. **Multi-level Feedback Queue (MLFQ)**
- Minimizes Resp. T for IO bound, Turnaround T for CPU bound
- Priority First, Equal ⇒ RR || New job ⇒ Highest Priority
- Fully utilize time slice ⇒ Priority ↓, Give up/Block ⇒ Retain

4. **Lottery Scheduling**
- Give out "lottery tickets" to processes for system resources
  Chosen randomly among eligible tickets → winner get resource
- Process holding X% of tickets can win X% of lottery held, use the resource X% of time
- Responsive: Newly created process participate in next lottery
- Process can then distribute lottery tickets to its child process
- Important can be given more & different % usage/resource/task

Scheduling Algorithm Trick
- RR improve avg. turnaround time over SJF
- Arriving by order of length ↓ response time of FCFS

Threads Unique info.: Id, Register, Stack
- A single process can have multiple threads
- Threads in the **same process** shares: Mem/OS context
  Thread Switch: Only need to switch hardware context.
- Scalability: Multithreaded program can use multiple CPUs
- Economy/Resource sharing: Use less resources by sharing
- Responsiveness: Multithread ⇒ more responsive
- **Problem:** SysCall Concurrency(Parallel syscall)/Process Behavior (`fork()`/`exec()`/`exit()` when multiple threads)

User Thread (implemented as user library) Flexible→Customize
- A runtime system (in the process) will handle thread related operation (just by library calls)
- Can have multithreaded program on ANY OS
- Kernel/OS is not aware of threads in process → Scheduling is performed at process level ⇒ One thread blocked → Process blocked → All threads blocked → × exploit multiple CPUs!

Kernel Thread Implemented in OS, operation handled as syscalls
- Kernel schedule by threads, instead of by process
- Kernel may make use of threads for its own execution
- **Slower**(syscall) || More Resource Intensive || Less flexible: Used by all multithreaded programs || Many features → Expensive, overkill for simple program || Few features → Not flexible

Hybrid Thread Model: User thread binds to a kernel thread; OS schedule on kernel threads **only**; Limit concurrency
Simultaneous Multi-Threading (SMT): Multiple sets of reg. → Threads run natively and in parallel on the same core
POSIX Threads: can be implemented as user/kernel thread

```
int pthread_create(pthread_t *tidCreated,
  const pthread_attr_t *threadAttributes /* can be NULL */,
  void *(*start_routine) (void *) /* function pointer */,
  void *arg /* argument for start_routine */);
int pthread_exit(void *exitValue);
```
- If not used, a pthread will terminate automatically when the end of `startRoutine` reached
- `exitValue`: Value returned to whoever sync. with this thread

`int pthread_join(pthread_t threadID, void **status);` `!0:err`
- `status`: Exit value returned by the target pthread

`gcc XXXX.c -lpthread`

Inter-Process Communication (IPC)
Create M → Attach M → Operations → Detach M → Destroy M
- Create/Attach is OS operation, Read/Write is pointer operation

```
int shmget(key_t key /*can be IPC_PRIVATE*/, size_t size,
int shmflg /* IPC_CREAT | 600*/); → shmid (-1 ⇒ error)
//IPC_CREAT means memory will be created if nonexistent
void *shmat(int shmid, const void *shmaddr /*NULL*/, int
shmflg /*0*/); → ptr -> shm (Attach to Mem) || -1 (error)
int shmdt(const void *shmaddr); /* Detach */
int shmctl(int shmid, int cmd /*IPC_RMID*/, struct shmid_ds
*buf /*unused for IPC_RMID*/); /* Destory */
```

Message Passing: Direct/Mailbox + Block/Non-Block
- `send()` and `receive()` are sys call,
- **Direct**: Sender/receiver explicitly names the other party
  One buffer per pair of (sender, receiver)
- **Indirect** Sender/Receiver sends/receives to/from mailbox/port
- **Blocking primitives (synchronous):** `Send()` blocks until message is received; `Receive()` blocks until message has arrived
- **Non-blocking primitives (asynchronous):** `Send()` does not block; `Receive()` returns indication if no message available
  Longer Message Buffer needed
- Pros: Portable, Easier Sync. • Cons: Inefficient, Hard to use

Unix Pipes Process involves 3 standard files
- `stdin`: Connect to the keyboard, read from stdin
- `stdout`: Can redirect to a file, send to screen
- `stderr`: Cannot redirect to a file

Pipe functions as circular bounded byte buffer with implicit sync.:
- Writers/Readers wait when buffer is full/empty
- Must close the end not in use, otherwise, other processes might accidentally write/read to it ⇒ Race Condition

Variants: Multiple readers/writers; unidirectional; bidirectional
Unix SysCalls: `int pipe(int fd[])` 0 for success; `!0` for errors
- `pipe(fd)` ⇒ `fd` → `An array of file descriptors`
- `fd[0] ==` Reading End, `fd[1] ==` Writing End

Unix shell: provides the `|` symbol to link the input/output channels of one process to another, this is known as piping
Unix Signal Kill, Stop, Continue, Memory error, Arithmetic error
- Async. notification regarding an event sent to a process/thread
- Handled by default/user supplied handler(only for some signals)

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
// returns previous signal handler, or SIG_ERR on error
if (signal(SIGSEGV, myOwnHandler) == SIG_ERR) {...}
int *ip = NULL; *ip = 123; → segmentation fault
```

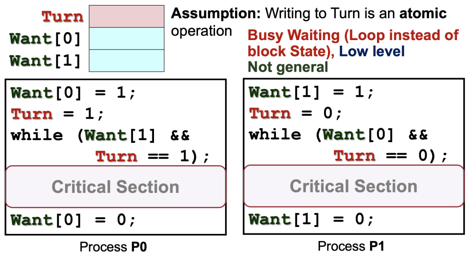
Synchronization
- Deterministic: Repeated execution gives the same result
- Race Conditions: Execution outcome depends on the order in which the shared resource is access/modified (non-determ.)

Critical Section
- **Mutual exclusion:** ∃ P in CS → all other P cannot enter CS
- **Progress:** No P in CS → one waiting P can be granted access
- **Bounded wait:** After a P requests to enter CS, ∃ upper bound of times other P can enter the CS before this P
- **Independence:** P not in CS should never block other Ps

Deadlock: All processes blocked
Livelock: Processes are not blocked, but they keep changing state to avoid deadlock and make no other progress
Starvation: Some processes are blocked forever
Test and Set: `TestAndSet Reg, Mem`. Atomically load current content at `Mem` into `Reg`, and stores 1 into `Mem`
- **Atomic:** Performed as a single machine operation

```
void EnterCS(int* Lock){while(TestAndSet(Lock) == 1);}
void ExitCS(int* Lock){*Lock = 0;}
```

Peterson's Algorithm

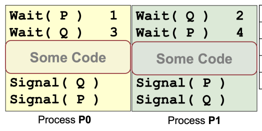

Turn
Want[0]
Want[1]
Assumption: Writing to Turn is an atomic operation
Busy Waiting (Loop instead of block State), Low level
Not general
Critical Section
Critical Section
Process P0
Process P1

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

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

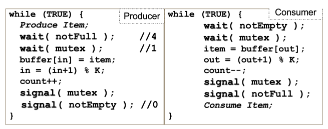
Semaphore: Contains an integer value initialized to any non-negative values. General: $S \geq 0$, Binary: $S = 0, 1$
- `Wait(S)`: S <= 0 → blocks (go to sleep) → Decrement S
- `Signal(S)`: Increments S → Wakes up one sleeping process.
- S(current) = S(initial) + #signal(S) - #wait(S)

Mutual Exclusion (`mutex`): S=0,1
- N_{CS} = # of process in CS = Process `wait()` but not `signal()` = `#Wait(S)` − `#Signal(S)`
- $S_{\text{initial}} = 1$ • $S_{\text{current}} + N_{\text{CS}} = 1$
- $S_{\text{current}} \geq 0 \Rightarrow N_{\text{CS}} \leq 1$



```
Wait( P )   1        Wait( Q )   2
Wait( Q )   3        Wait( P )   4
Some Code            Some Code
Signal( Q )          Signal( P )
Signal( P )          Signal( Q )
Process P0           Process P1
```

	P=1	Q=1
1	0	1
2	0	0
3	P0-D	
4		P1-D



```
while (TRUE) {                        Producer
  Produce Item;
  wait( notFull );         //4
  wait( mutex );           //1
  buffer[in] = item;
  in = (in+1) % K;
  count++;
  signal( mutex );
  signal( notEmpty ); //0
}
```

```
while (TRUE) {                        Consumer
      wait( notEmpty );
      wait( mutex );
      item = buffer[out];
      out = (out+1) % K;
      count--;
      signal( mutex );
      signal( notFull );
      Consume Item;
}
```

Conditional Variable: Ability to broadcast (wakes up all)
Memory: Store Transient Data and Persistent Data
Physical memory storage: Random **A**ccess **M**emory
Address Relocation: recalculate memory references by adding an offset (where process starts) to all memory ref. Slow loading time and not easy to distinguish mem. ref.
Base + Limit: `Actual = Base + Adr && Actual < Limit`
- **Compilation:** Mem Ref are offsets from base register
- **Loading:** Base reg. is starting address of process mem. space

Logical Address: Maps logical → physical. Every process has a self-contained, independent logical memory space.
Contiguous Memory
- **Multitasking:** multiple processes in memory at the same time (switch from one process to another)
- Mem. Full → Removing terminated process or Swapping blocked process to secondary storage

Memory Partition Fixed-Size Partition
- Physical memory is split into fixed number of partitions ⇒ A process will occupy one of the partitions ⇒ Internal Frag. as **Size of Mem should > largest process**
- Easy to manage. Fast to allocate.

Dynamic Partition: Base on the actual size of process
- OS keep track of the occupied and free memory regions
- **First-Fit:** Take the first hole that is large enough
- **Best-Fit:** Find the smallest hole that is large enough
- **Worst-Fit:** Find the largest hole
- Split the hole into N(for new partition) and $M - N$(new hole)
- When an occupied partition is freed, **merge** with adjacent hole.
- **Compaction:** Move the occupied partition around to create consolidate holes (Very time consuming, do not use frequently)

Buddy System
- Keep an array `A[0...K]`, where 2^K is largest allocatable size `A[J]` is a linked list keeps block(s) of the size 2^J
- Find the smallest S, such that $2^S \geq N$
- Block Exist → Allocate, Else Find the smallest $R \in [S + 1 to K]$, s.t. `A[R]` has free block B. For $[R - 1, S]$ Repeatedly split B → `A[S...R-1]` has a new free block → Check and split if possible.
- After free block B, check if buddy of B exists, merge if possible. (Buddy: S-th bit is a complement, [left,S]-th bits are same)

Buddy Address: blocks A and B are buddy of size S, if
- The S-th bit of A and B is a complement
- The **leading bits** up to S-th bit of A and B are same

Disjoint Memory: Paging Scheme
- Physical Memory → Fixed Size Physical Frame
- Logical Memory of a Process → Same Size, Logical Page
- Physical frame size = Logical page size = $\mathbf{2^n}, n \in R$
- Try to solve external fragmentation, but still have internal.

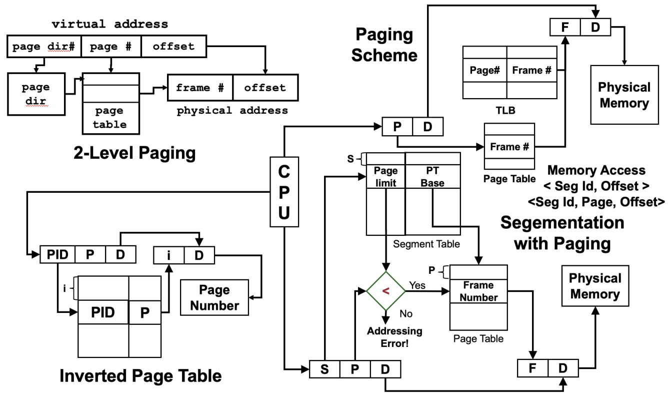
Page Table: Store **info** in PCB ⇔ memory context of a porcess.
- PA = F# × size of physical frame + Offset
- If m bits for logical address, p=Most Significant $m - n$ bits in LA, $p \to f$, d=Remaining n bits in LA ⇒ $PA = f \times 2^n + d$
- Allow several processes to share same physical memory frame
- **Usage:** Shared Code Page (Code used by many processes) / Implement Copy-On-Write (Copy page table when `fork()`, Parent and Child share PA, copy PA only when someone write)

- **Access Right Bits:** Each Table Entry has **w**ritable, **r**eadable, **e**xecutable bits. Memory access is checked against this bits.
- **Valid Bit:** Indicate whether page is valid to access (i.e. Out of Range?). OS set when a process is running.

Page Table Structure: Page Table takes physical memory space
- **Direct Paging:** All PTEs in a table. 2^P pages → P bits to specify 2^P PTEs. VM Addr. m bits, Page Size n bits ⇒ $P = m - n$
- **2-Level Paging:** Split page table of 2^P into 2^M smaller page tables of 2^{P-M} (Each with a page table number). Page directory to track smaller page tables
- **Inverted Page Table:** One table for all processes. Lookup page X ⇒ Search whole table. $[O(n)]$ `<pid, page#>` → PA
  - Entries ordered by frame number

Translation **L**ook-Aside **B**uffer: cache of a few page table entries
- TLB is part of the **hardware context** of a process
- Use page number to search TLB
  - **Hit:** Frame # retrieved to generate PA → Access Memory
  - **Miss:** Memory access to access full page table → retrieve Frame # to get PA and update TLB → Access Memory
- TLB Access Time < Memory Access
- When a process resume running, it will encounter many TLB misses to fill the TLB (place some entries initially)

Segmentation Scheme: Separate into multiple mem. segments with segment name(usually a number, segment id) and limit
- Each segment → contiguous physical memory region with a base address and a limit
- LA `<SegID, offset>` → use SegID to get `<Base, Limit>` in seg. table → `Offset < Limit` → Valid → `PA = Base + Offset`
- Segment is independent contiguous, can grow / shrink
- External fragmentation



Page Replacement Algorithms
1. **Optimal Page Replacement** (OPT): Replace the page that will not be used again for the **longest** period of time
   - Guarantees min number of page faults, but not realizable.
2. **FIFO:** Evict the oldest memory page. Using Queue
   - Remove the first page in queue if replacement is needed
   - Update the queue during page fault trap
   - physical frame ↑→ # of page faults (**Belady's Anomaly**) ← FIFO does not exploit temporal locality
3. **Least Recently Used** (LRU): Replace the page that has not been used in the longest time (use temporal locality)
   - Does not suffer from Belady's Anomaly
   - Need substantial hardware support to implement
   - Using Time Counter or Stack of Page number(Hard to implement)
4. **Second-Chance Page Replacement** (CLOCK)
   - The oldest FIFO page is selected ⇒ If reference bit = 0 replaced || bit = 1 → Reference bit cleared to 0 → Next.
   - When all pages have reference bit = 1 → FIFO algorithm

Virtual Memory: Secondary Storage >> Physical Mem Cap.
- $T_{\text{access}} = (1 - p) \times T_{\text{mem}} + p \times T_{\text{page\_fault}}$
- Split the logical address space into small chunks. Some in physical memory, other in secondary storage
- CPU can only access memory resident pages
- **Page Fault:** When CPU tries to access non-mem resident page. OS needs to bring a non-mem resident page into physical mem
- Look up page table → not found → OS Trap → Locate page and move it to physical mem → update page table and retry
- **Thrashing:** Memory access results in page fault most of the time. (unlikely to happen by Locality Principles)
  - **Temporal:** Mem addr. used is likely to be used again
  - **Spatial:** Mem addr. close to a used addr. is likely to be used
- More processes to reside in mem →↑ CPU utilization

Demand Paging: Process starts with no memory resident page
- Fast startup time for new process / Small memory footprint
- Sluggish at start / Page Faults effect on other processes

Local Replacement: Victim page are selected **among** pages of process that causes page fault
- Frames allocated to a process is constant → Stable Performance
- Thrashing can be limited to one process. But that process can hog the I/O and degrades performance of other processes
- Frames allocated not enough → hinder the progress of a process

Global Replacement: Victim selected among **all** physical frames
- Allow self-adjustment between processes
- Badly behaved process can affect others
- Frames allocated to a process can be different from run to run
- Thrashing process "steals" page from other process → other process to thrash (Cascading Thrashing)

Frame Allocation: Consider M process compete for N Frames
- **Equal Allocation:** Each get M/N frames
- **Proportional Allocation:** Each get $\frac{\text{size}_i}{\text{size}_{\text{total}}} \times N$ Frames
- Insufficient physical frames → Thrashing

Locality: Pages ref. by a process ∼ constant in a period of time
Working Set Model: With the set of pages in frame, no/few page fault until process transits to new locality
- $W(t, \Delta)$ = active pages in the interval at time t
- Allocate enough frames for pages in $W(t, \Delta)$ to ↓ P(page fault)
- Small Δ: May miss pages in the current locality ⇒ PF High.
- Big Δ: May contain pages from a different locality ⇒ Both PF and CPU utilization is low
- Well Choosen: CPU is well utilized and page fault activity is low

File System

	Memory Management	FS Management
Storage	RAM	Disk
Access Speed	Constant	Variable disk I/O time
Unit of Addr	Physical memory address	Disk sector
Usage	Address space for process **Implicit** when process runs	Non-volatile data **Explicit** access
Organization	**Paging/Segmentation:** determined by HW & OS	ext* (Linux), FAT* (Windows), HFS* (Mac)

File: Logical unit of info created by process [Abstract Data Type]

- Metadata (File Attributes): Name/Id/Type/Size/Protection ...
- Extension: `Name.Extension`. Win use to indicate file type
- Type: Each type possibly has a specific program for processing
  - Regular files: contains user information. ASCII/Binary
  - Directories: system files for FS structure
  - Special files: character/block oriented

  Unix use embedded info (a magic number) to indicate type.
- Access Control List: A list of user id and allowed access types (**R**ead/**W**rite/**E**xecute) ⇒ File Protection

File Data Structure
- Array of bytes with their offset from the start
- Array of fixed length records, can grow/shrink. Search easily
- Variable length records: Flexible but harder to locate a record

Access Methods
- Sequential Access: Read in order from start. ×skip but rewound
- Random Access: Read in any order, Unix and Win uses `seek()`
  - `Read(Offset)`: Explicitly states the position to be accessed
  - `Seek(Offset)`: Move to a new location in file
- Direct Access: Random access to any record directly

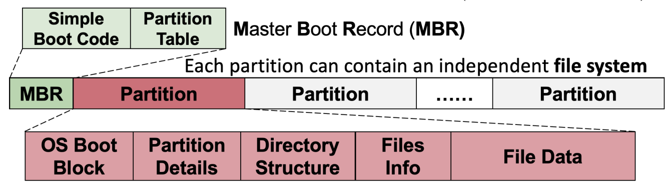
Operation on File Data: `Create, Open, Read, Write, Reposition/Seek, Truncate`(Removes data from pos to EOF)

File Operations: OS provides as sys. call (Protect/Maintain/.)
- Info of Opened File: File pointer (current location in file), Disk location, Open count (# of process that has this file opened)
- PCB → Open File Table → Actual File
  - System-Wide: One entry per unique file
  - Per-Process: One entry per file used in the process. Each entry points to the system-wide table.
- `fork()` → Sharing File (Two processes use same file descriptor)

Directory DAG Structure: [H]→File, [S]→File/Dir.
- Hard Link: Owner and Sharer have separate pointers point to the actual file on disk (Low overhead/Deletion problems) [`ln`]
- Symbolic Link: Sharer creates a special link file contains the path name of F(Simple deletion/Larger overhead) [`ln -s`]

General Graph: Create using Symbolic link of Dir. in Unix

Disk Structure: 1-d Array of logical blocks (512-bytes to 4KB)



- Logical block → disk sector [Layout is Hardware Dependent]

File Block Allocation
1. **Contiguous:** File info. store `<Name, Start, Length>`
   - Simple to keep track and Fast access (Start + Offset).
   - External Frag.; File size needs to be specified in advance
2. **Linked List:** Each disk block stores data and next block # File information stores `<Name, Start Blk#, End Blk#>`
   - Solve fragmentation problem
   - Slow random access, Less reliable, Pointer overhead
3. **FAT Allocation:** Move all the block pointers into File Allocation Table in Memory. (Used by MS-DOS)
   - File Info. stores `<Name, Start>`
   - Faster Random Access (Traversal takes place in memory)
   - Huge FAT when disk is large ⇒ Consumes memory space
4. **Indexed Allocation:** Each file has an index block (An array of disk block addr) ⇒ Block[N] is N-th Block address
   - File Info. stores `<Name, Index Blk#>`
   - Only block of opened file in memory ⇒ Less Overhead
   - Limited Max. file size (Max # of blocks = Len(IndexBlock))
   - Variation: Linked scheme - Keep a linked list of index blocks

     Multilevel index: Generalized to multi. # of levels

     Combined scheme: Direct Indexing + Multi-Level Indexing

Free Space Management: In Partition Details
1. **Bitmap** 1 bit each block. **1** - Free/**0** - Occupied
   - Need to keep in **memory** for efficiency reason
2. **Linked List** of Blocks: Each disk block contains a number of free disk block # and pointer to the next free block
   - Only first pointer is needed in memory ⇒ Easy to locate
   - High overhead

Directory Structure:
- Track File in directory(Possibly with File Metadata)
- Map File Name to Info.: File must be opened before use Open operation(Locate file info. using pathname and file name)

1. **Linear List:** Each entry represents a file
   - Store file name (**minimum**) and possibly other metadata
   - Store file information or pointer to file information
   - Linear search to locate ⇒ inefficient for large FS/deep tree
   - Use cache to remember the **latest** few searches
2. **Hash Table:** File name is hashed into index K
   - Limited size and depends on good hash function

File Info: File Name, other Metadata and Disk Blocks info.
1. Store everything in directory entry
2. Store only name and pointers to some DS for other info

File System in Action: user interacts with a file at run time need **run time info** in memory

Create File `/.../.../parent/F`
1. Use full pathname to locate the parent directory
   - Search for filename to avoid duplicates → terminates with error (could be on the cached directory structure)
2. Use free space list to find free disk block(s)
3. Add an entry with relevant file information to parent directory

Open File `/.../.../parent/F`
1. Search system-wide table for existing entry E
   - Found ⇒ Creates an entry in PCB → E and **return** pointer
2. Use full pathname to locate file F ⇒ If located, load its file info into a new entry E in system-wide table ⇒ Repeat 1.1
3. Returned pointer is used for further read/write operation

Microsoft FAT File System:
- Data allocated to data blocks. Pointers kept in FAT.
- Allocation info kept as a linked list.

File Allocation Table (FAT)
- One entry per data block/cluster
- Store disk block information (Free/Next block/EOF/Bad)
- OS will cache **in RAM** to facilitate linked list traversal

Directory: Represented as Special type of file in the data block
- Root directory is stored in a special location
- Other directories are stored in the **data blocks** ⇒ Dir. Entry

Searching From Directory
- First disk block # stored in directory entry to find the starting point of the linked disk blocks
- FAT to find out the subsequent disk blocks number
- Disk block# to perform actual disk access on data blocks

Directory Entry: File/Subdirectory within this folder

[8 bytes] File Name	[3] File Extension	[1 Bytes] Attributes	[10 Bytes] Reserved	[2] Creation Date	[2] Time	[2] First Block	[4] File Size in Bytes

- File Name: first byte may have special meaning

  Virtual FAT: Support long filenames up to 255 characters
  - Use multiple directory entries for a file with long file name
  - Still keep the 8+3 short version for backward compatibility
- Creation Time and Date: Year: 1980-2107, Accuracy ±2s
- **First Disk Block Index:** 12/16/32 bits for FAT12/16/32.

Disk Cluster: A number of contiguous disk blocks
- (Cluster size + FAT size) determines largest usable partition
- Largest partition of FAT-N = 2^N× Cluster Size

  Actual size is a bit lesser as EOF, FREE ↓ # of blocks
- FAT32: 32-bit sector count, **28-bit** disk block/cluster index

Extended-2 File System: Disk → Blocks → Block Groups
- File/Directory described by I-Node (Contains File Metadata, Data Block Addresses)

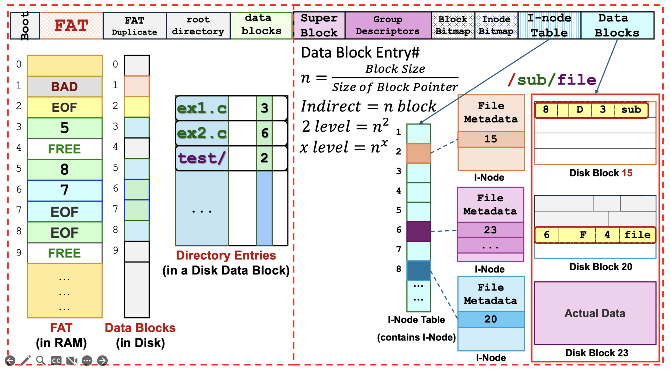
Partition Information
- Superblock: Include Total I-Nodes number, I-Nodes per group, Total disk blocks, Disk Blocks per group
- Group Descriptors: # of free blocks, free I-nodes, Bitmaps loc.
- Block Bitmap • I-Node Bitmap • I-Node table

I-Node Structure: Multi-Level, 15 disk block pointers
- First 12 pointers → Direct Blocks
- 13/14/15-th pointer → Single/Double/Triple Indirect Block
- Max File Size = $\sum$(Number of Block) × Block Size
- **Possible** Max Number of Block = $2^{\text{Length of Block Pointer}}$

Directory Structure:
- Data blocks stores a linked list of directory entries for info.
- Directory Entry contains **I-Node#**, Size of entry (To locate entry), Length of Name, Type, Name (up to 255 characters)

Hard Link Problem: Multiple references of a I-Node → Hard to determine when to delete an I-Node ⇒ Maintain I-Node reference count, decremented for every deletion.

Symbolic Link Problem: Link can be invalidated (Name Change/Deletion) ⇒ Search actual I-Node # of target file



Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		