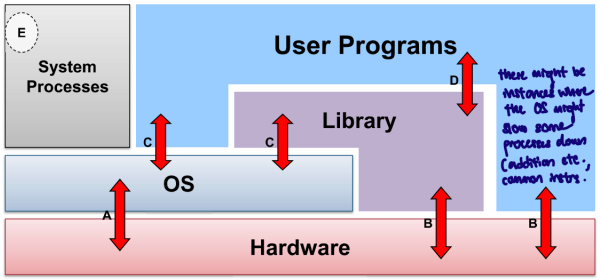
CS2106 Cheatsheet AY22/23 —— @JasonYapzx

Motivation for OS

- Abstraction → ease of programming, portability, efficiency
- Resource Allocator → asynchronous programs running
- Control Program → prevents errors, provides security

OS Components

Operating systems are essentially software, where they run in *kernel mode*, giving full access to all hardware resources. Other software usually executes in *user mode* with limited access to hardware resources.



- **A**: OS executing machine instructions
- **B**: normal machine instructions executed (program/library code)
- **C**: calling OS using ***system call interface***
- **D**: user program calls library code
- **E**: system processes (provide high level services, usually part of OS)

OS Structures

Monolithic OS

Monolithic as the name suggests, has ONE BIG program as its OS, where all its process, memory, file system and system call interface are within the same program.

- Advantages: well understand & great performance
- Disadvantages: highly coupled, complicated internal structure

Microkernel OS

Kernel is small and clean, and only provides the basic and essential facilities. The higher level services are built on top of the basic facilities.

- Advantages: kernel is more modular + robust → easily extendible, good isolation between kernel / high-level services.
- Disadvantages: less performance

Virtual Machines / Hypervisors

Software emulation of hardware (**virtualization**).

- **Type 1 hypervisor OS:** provides individual **virtual machines** to guest OSes → similar to on-premise hosting of different OSes, cloud providers giving resources for people to run containers/servers.
- **Type 2 hypervisor OS:** Runs in the host OS guest OS runs inside Virtual Machine → similar to VirtualBox emulation, it uses resources as part of a program to emulate the virtual OS.

Process Abstraction

Components

- Memory context: Text, Data, **Stack** and **Heap**
- Hardware context: **General Purpose Registers (GPR)**, PC, Stack/Frame Pointer (SP/FP)

Control Flow and Data

Stack Memory Region: where information and function invocation is stored. Stack Pointer points to the top of stack region (first free location). Information needed for function invocation is described as a stack frame and contains:

- return address of caller
- arguments (params)
- local variable storage
- **common additional information**: frame pointer/saved registers

Frame pointer: facilitates access of various stack frame items, points to a fixed location in the stack frame.
Saved Registers: prevent *register spilling*, GPRs are usually limited on processers. Functions spill registers it intends to use before starting, restoring them at the *end of the function*.

Stack Frame Setup / Teardown

On executing function call:

- **Caller:** Pass arguments with registers and/or stack
- **Caller:** Save Return PC on stack

Transfer control from caller to callee:

- **Callee:** Save registers used by called. Save old FP and SP
- **Callee:** Allocated space for local variables of callee on stack
- **Callee:** Adjust SP to point to new stack top

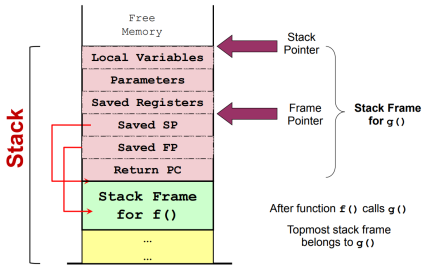
On returning from function call:

- **Callee:** Restore saved registers, FP and SP

Transfer control from callee to caller using saved PC:

- **Caller:** Continues execution in caller

Illustration: **Stack Frame v2.0**



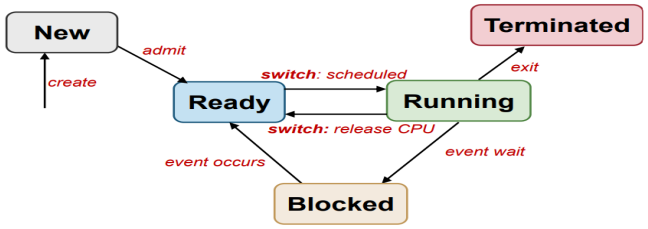
Dynamicaly Allocated Memory

- Allocated only at runtime → size not known at compile time (cannot put int `data` region)
- No definite deallocation timing → cannot be explicitly freed any time (cannot put int `stack` region)
- Solution: setup a separate **heap memory region**

Processes

5-STAGE PROCESS MODEL (PROCESS State)

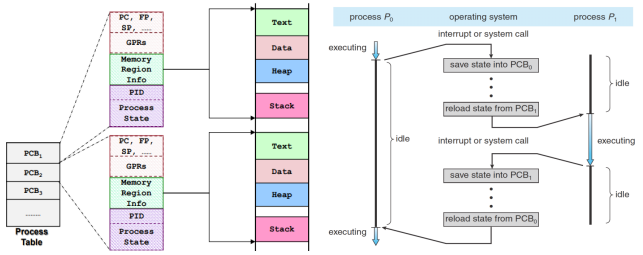
1. **New:** process created / still initializing
2. **Ready:** process waiting to run
3. **Running:** process executed on CPU
4. **Blocked:** waiting for event → × execute until event available
5. **Terminated:** finished execution, may require OS cleanup



Process Table

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

- HW Context: Only updated in PCB when process swapped out
- Mem Context: Not actual memory space used by the process
- OS Context: May contain information used for scheduling



Exceptions and Interrupts

Executing a **machine level instruction** can cause exceptions. They are synchronous and occur due to program execution. An **exception handler** is executed automatically, similar to a forced function call. Examples:

- Arithmetic Errors: Overflow, Underflow, Division by zero
- Memory Errors: Illegal/Misaligned memory address

Interrupts

External events that can interrupt the execution of a program. Intterupts are asynchronous and occur in events that are indepedent of program execution. The program exeuction is suspended, and an **interrupt handler** is exeucted automatically.

- Hardware related: Timer, mouse movement, keyboard press etc.
- Can be enabled/disabled (selectively or globally), prioritized, nested.

Interrupt Handling Steps: HW/SW Co-design

The contract between HW and SW around the IVT. OS populates IVT with address of interrupt routines. HW reads the IVT automatically to locate the handler.

1. PUSH (PC) – hardware
2. PUSH (status register) – hardware
3. Disable interrupts in the status register – hardware
4. Read Interrupt Vector Table (IVT) entry number #42 – hardware → IVT a table where the OS stores the address of all interrupt handlers
5. Switch to kernel mode – hardware
6. PC ← `handler_42()` – hardware
7. `handler_42()` execution – software

System Calls

Application Programming Interface (API) to OS, provides a way of calling facilities or services to the kernel. OS has to change from user mode to kernel mode. In C/C++ program, system calls can be invoked *almost directly*:

1. Library version with the same name and the same parameters (function wrapper)
2. User Friendly Library version (function adapter)
3. `long syscall(long number, ...)`

General System Call Mechanism

1. User invokes library call
2. Library call (in assembly code) places the **system call number** in designated location
3. Library call executes a special instruction to switch from user mode to kernel mode (e.g TRAP or syscall)
4. Now in kernel mode the appropriate system call handler is determined:
   - Using system call number as index, (similar to accessing the IVT) → handled by dispatcher
5. System call handler executed
6. System call handler ended: control return to the library call, switch back to user mode
7. Library call return to user program via normal function return mechanism

UNIX Context

A UNIX process abstraction has PID for identification, and the following for information: Process State, Parent PID, Cumulative CPU time (total amount of CPU time used so far)

`ps`: short for process status

`init`: Root of all process (UNIX) - PID 1

`fork()`: Creates child process with copy of parent's executable image. Child executes remaining code from where the `fork()` was called. Data not shared. Function returns `0` for child, `> 0` for parent. Child differs in: PID, Parent PID, `fork()` return value.

`exec()`: Many variants: `execv, execl, execle, execlv, execlp` Replaces current executing process image with a new one. Only replaces code, PID and other information remains.

- `path`: location of executable, if not executable, return `0`
- `arg0, ..., argN`: command line argument(s)
- `NULL`: indicate end of argument list

`exit()`: Takes in a status, terminates process and returns status to parent process. Upon process termination, some resourcess are not released: PID, status, CPU time. Generally PCB remains.

`wait()`: Waits for child (blocks), if a child exists, else it returns `-1`. Cleans up on child process (those that were nor removed on `exit()`), e.g. removes PCB of child. Also kills **zombie** processes.

- **Zombie:** When child exits but parent doest not call `wait()`
  - On `exit()`, becomes zombie. Cannot delete all process info until `wait()` is called, to allow data to be cleaned up.
- **Orphan:** Subset of Zombie. When parent terminates before child process. `init` becomes parent and does the clean-up

Inter-Process Communication

$\mathbf{P}_1$ creates a shared memory region $\mathbf{M}$. $\mathbf{P}_2$ attaches memory region $\mathbf{M}$ to its own memory space. $\mathbf{P}_1$ and $\mathbf{P}_2$ can now communicate using memory region $\mathbf{M}$. $\mathbf{M}$ would behave like a normal memory region.

ADVANTAGES / DISADVANTAGES

1. **Efficient:** Only `create` and `attach` require OS
2. **Ease of use:** Shared memory region behaves like a regular one, write info of any size/type
3. BUT **Synchronisation** issues and **Hard to implement**

Race Condition

Final outcome depends on the interleaving of `read` and `write` operations. Possibly huge number of interleaving scenarios, this could grow exponentially w.r.t number of processes/shared variables/operations on shared variables.

Shared Memory (UNIX)

Basic steps of usage:

1. Create/locate a shared memory region $\mathbf{M}$
2. Attach $\mathbf{M}$ to process memory space
3. Read from/Write to $\mathbf{M}$ → values written visible to all processes that share $\mathbf{M}$
4. Detach $\mathbf{M}$ from memory space after use
5. Destory $\mathbf{M}$ (only one processes needs to do this) → can only destroy if $\mathbf{M}$ not attached to any process

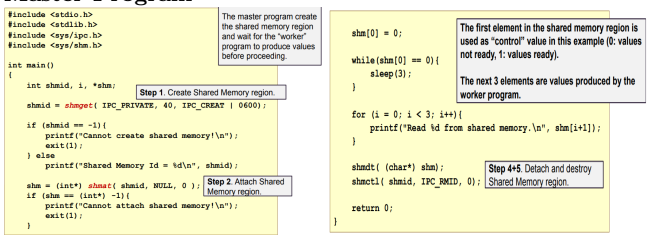
`shmget()`: (master) Creates shared memory region, returns the id.

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

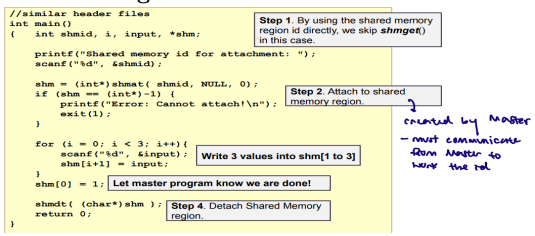
`shmdt()`: Detaches shared memory region using the pointer from attaching. Done by both the master and slave program.

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

Master Program



Worker Program



Message Passing

Naming Scheme

1. **Direct Communication:** Sender and receiver explicitly name the other party. Needs one link per pair and processes to know the identity of the other part.
   - `Send(P2, Msg)` - to send `Msg` to process `P2`
   - `Receive(P1, Msg)` - to receive `Msg` from process `P1`
2. **Indirect Communication:** Send and receive from a port. One port can be shared among a number of processes.
   - `Send(MB, Msg)` - to send `Msg` to mailbox `MB`
   - `Receive(MB, Msg)` - to receive `Msg` from mailbox `MB`

Synchronisation

1. **Blocking Primitives (Synchronous):** Also known as rendezvous, no intermediate buffering required
   - `send()`: sender is blocked until the message is received
   - `receive()`: receiver is blocked until a message has arrived
2. **Non-Blocking Primitives (Asynchronous):** Too much freedom for programmer, complex program. Finite uffer size means system is not truly async
   - `send()`: sender resumes opertaion immediately
   - `receive()`: if message hasn't arrived yet, proceeds empty-handed but doesn't block
3. Usually `receive()` is blocking → behavior of sender critical
   - `send()`: blocks until matching `receive()` ran → synchronous
   - `receive()`: proceeds regardless → asynchronous

Message Buffers

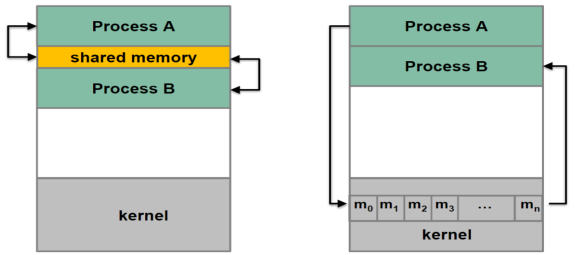
Intermediate buffers between sender and receiver in asynchronous communication, under OS control. Large buffer decouples sender and receiver not need to wait for one another unnecessarily.

ADVANTAGES OF MESSAGE PASSING

1. **Portable:** Easily implemented across different processing envs
2. **Ease of sync:** Using blocking primitives force sender and receiver to be synchronous

DISADVANTAGES OF MESSAGE PASSING

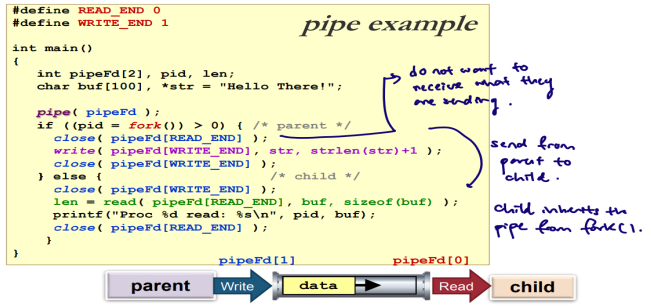
1. **Inefficient:** Need OS intervention
2. **Hard to use:** Messages are limited in size and format



Shared memory Message passing

Unix Pipes

In UNIX, process has 3 default communication channels, `stdin`, `stdout` and `stderr`. The | in shell directs one process' output to another's input. e.g. `A | B`, output of A goes into B. More generally, a pipe is a FIFO circular bounded byte buffer with implicit synchronisation - writers wait when buffer isolation full, readers wait when buffer is empty.



VARIANTS

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

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

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

`dup()`: Takes in a file descriptor (fd) and finds the lowest unused fd and duplicates it there. (lowest unused fd is now an alias for the argument fd). `dup2()`: Like `dup()` execpt that you specify the new file descriptor to duplicate into.

Signal (UNIX)

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

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

```c
void myOwnHandler( int signo ) {
    if (signo == SIGSEGV) {
        printf("Memory access blows up!\n");
        exit(1);
    }
}
int main() {
    int *ip = NULL;
    if (signal(SIGSEGV, myOwnHandler) == SIG_ERR)
        printf("Failed to register handler\n");
    *ip = 123; // seg fault
    return 0;
}
```

Process Scheduling

Concept of multiple processes **progress** in execution (at the same time): virtual / physical parallelism. 1 core (CPU) - Time sliced execution of tasks. OR Multiprocessor: time slicing of **n** CPUs

Process Behaviour

CPU-Activity:

- Computation - e.g. Number crunching
- **Compute-Bound Process** spends majority of its time here

IO-Activity:

- Requesting and receiving service from I/O devices - e.g. Print to screen, read from file
- **IO-Bound Processes** spends majority of its time here

Processing Environment

1. Batch Processing: No user interaction, no need to be responsive
2. Interactive / Multiprogramming: With active user interacting system, should be responsive (low/consistent response time)
3. Real time processing: Have deadline to meet - periodic process

Criteria for Scheduling Algorithms

Largely influenceed by processing environment, may be conflicting
Fairness: Get a fair share of CPU time, per process/user basis
Utilization: All parts of the computing system should be utilized

When to perform scheduling

Non-preemptive: (Cooperative) A process stays scheduled (in running state) untl it blocks or gives up CPU voluntarily
Preemptive: Process given a fixed time quota to run, possible to block / give up early. At the end of time quota, running process suspended, another ready process gets picked.

⋆ The moment PC set to one of the addresses of the instruction in a process, it is said to be in the running state.

1. Scheduler is triggered (OS takes over)
2. If context switch needed, context of currently 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 the context for **P**, let process **P** run

Batch Processing

No user interaction, non-preemptive scheduling is predominant.

Criteria for batch processing

1. **Turnaround time:** Total time taken (finish time - arrival time), related to waiting time (time spent waiting for CPU)
2. **Throughput:** Number of tasks finished per unit time (rate of task completion)
3. **Makespan:** Total time from start to finish to process all tasks
4. **CPU Utilization:** Percentage of time when CPU is working on a task
5. **Wait time**: Turnaround time - Burst time

First-Come First-Served – Non-Preemptive

Tasks stored in FIFO queue based on arrirval time. Pick task in queue to run until DONE / BLOCKED. Blocked task removed from FIFO queue, when ready again place at back of queue.

⋆ Guaranteed to have no **starvation**: number of tasks in front of task X in FIFO is always decreasing.

Shortcomings: Simple reordering can reduce average waiting time - Convoy Effect.

Shortest Job First (SJF)

Select task with the smallest total CPU time upon arrival (not preemptive). Need to know total CPU time for a task in advance. Given a fixed set of tasks, minimizes average waiting time.

⋆ Starvation is possible as it is biased towards shorter jobs. Long jobs may not get a chance to run.

A task usually goes through several phases of CPU-Activity. Possible to guess the future CPU time requirement by previous CPU-Bound phases.

- Actual_n - most recent CPU time consumed
- Predicted_n - the previous CPU time consumed
- α - Weight placed on recent event or past history
- Predicted_{n+1} - Latest prediction

$$Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$$

Shortest Remaining Time (SRT) – Preemptive

Variation of SJF, use remaining time and preemptive. New job with shorter remaining time can preempt currently running job. Provide good service for short job even when arrived late.

Interactive Systems

Criteria for interactive environment

1. **Response time:** time between request / response by system
2. **Predictability:** variation in response time, lesser variation == more predictable

Preemptive scheduling algorithms are used to ensure good response time → scheduler needs to run **periodically**

Interval of Timer Interrupt: OS scheduler is invoked on every timer interrupt, typically 1ms to 10ms. (freq of scheduler)
Time Quantum: Execution duration given to a process, could be constant or variable among the processes. Must be multiples of interval of timer interrupt. Large range of values (commonly 5ms to 100ms). This is the actual time for processes to run.

Round Robin (RR)

Tasks stored in a FIFO queue. Pick the first task from queue front to run until fixed *time slice (quantum)* elapsed or task gives up CPU voluntarily or task blocks. Task is then placed to back of queue to wait another turn. (Block task moved to another queue to wait for requested resource). When blocked task is ready again, placed at back of queue.

⋆ Preemptive version of FCFS.

- Response time guarantee: given n tasks and quantum q, time before a task gets CPU is bounded by $(n-1)q$
- Timer interrupt needed: Scheduler to check on quantum expiry
- Choice of time quantum
  - Big: Better CPU utilization, longer waiting time
  - Small: Worse CPU utilization, shorter waiting time

Priority Scheduling

Some processes more important than others, assign priority value to all tasks, selecting that with the highest priority value.
Preemptive: Higher priority process preempts running process with lower priority
Non-preemptive: Late coming high priority process has to wait for next round of scheduling

⋆ Low prioirty process can starve, high priority process hogging CPU (worse for preemptive version). Hard to guarantee or control the exact amount of CPU time given to a process using priority.

Possible solutions:

- Decrease priority of currently running process after every quantum (eventually dropped below next highest priority)
- Give current running processing a time quantum, not considered in the next round of scheduling

Priority Inversion: Lower priority task preempts higher priority

- Priority: {A=1, B=3, C=5} (1 highest). Task **C** starts and locks a resource. **B** preempts **C** (unable to unlock resource)
- Task **A** arrives and needs same resource as **C**, but it is locked!
- Task **B** continues execution even if Task **A** has higher priority.

Multi-level Feedback Queue (MLFQ)

Adaptive Learn the process behavior automatically
Minimizes: Response time for IO-Bound processes, turnaround time for CPU-Bound processes.
Basic Rules:

- If `Priority(A)` > `Priority(B)` → A runs
- If `Priority(A)` == `Priority(B)` → A and B runs in RR

Priority Setting/Changing Rules:

- New job → highest priority

- If a job fully utilized its time quantum → priority reduced
- If a job gives up / blocks before finishes its time quantum → priority retained

Abusing MLFQ Algorithm: `sleep()` right before the quantum, which is blocking. Since it is no longer using CPU, the scheduler schedules another task. Ensures that the process always finishes RIGHT BEFORE end of quantum. This allows process to still retain its high priority.

- **Solution**: Track total CPU time instead. If we want to further abuse this, we can `fork()` every child process to run the task so priority is kept. Parent exits to prevent system from clogging.
- **Solution to Solution's problem**: Inherit parent's CPU usage statistic, ensure that a child would retain it's parent's priority.

Algorithm	Batch / Interactive?	Pre-emptive?	Comments
First-Come-First-Served (FCFS / FIFO)	Batch	No	Simple, but could underutilize CPU
Shortest Job First		No	"Theoretically optimal" but hard to know length of task
Shortest Remaining Time		Yes	Fixes late short job issue in SJF
Round Robin	Interactive	Yes	Pre-emptive FCFS
Priority-based		Both possible	Better allocation but could have starvation / priority inversion
Multilevel Feedback Queue (MLFQ)		Yes	Quite balanced solution by managing CPU/IO tasks
Lottery Scheduling		Yes	Simple theoretical idea, not covered

Lottery Scheduling

Give out 'lottery tickets' to processes for various system resources. When a scheudling decision is needed, lottery ticket is chosen randomly among eligible tickets, winner granted resource. In the long run, a process holding **X**% of tickets can win **X**% of lottery held, and it uses the resources **X**% of the time

- **Responsive:** newly created process can participate in the next lottery
- **Good level of control:** Process can be given Y lottery tickets, distribute to child process
- **Important process given more lottery tickets**
- **Each resouce can have its own set of tickets**
- **Simple to implement**

Synchronisation Primitives

Execution of single sequential process is *deterministic*. However execution of concurrent processes may be **non-deterministic** as the execution outcome depends on the order in which the shared resources is accessed or modifed.

Race Condition Solution

Needed: synchronization to control the interleaving of accesses to a shared resource.

- Allow all (as many as possible) correct interleaving scenarios
- Not allow any incorrect interleaving scenario

Solution Outline

- Desginate code segment with race condition as critical section
- At most 1 process in critical section

Critical Section

Properties of CS

- **Mutual Exclusion**: If process `P` is executing in critical section, all other processes are prevented from entering critical section.
- **Progress**: If no process in critical section, one of the waiting processes should be granted accesses.
- **Bounded Wait**: After process `P` request to enter critical section, there exist an upper bound on the number of times other processes can enter the critical section before `P`.
- **Independence**: Process not executing in critical section should never block other processes.

Symptoms of Incorrect Synchronisation

- **Incorrect output/behaviour** - to lack of Mutual Exclusion
- **Deadlock** - all processes blocked (circular dependency loop) → no progress
- **Livelock** - Process not in a blocked state, keep changing state to avoid deadlock but make no progress. Usually related to *deadlock avoidance*.
- **Starvation** - processes blocked forever, algo is favouring certain processes

Peterson's Algorithm:

```
// Process 1
Want[0] = 1;
Turn = 1;
while (Want[1]
    && Turn == 1);
// CRITICAL SECTION
Want[0] = 0
```

```
// Process 1
Want[1] = 1;
Turn = 0;
while (Want[0]
    && Turn == 0);
// CRITICAL SECTION
Want[1] = 0
```

- If ONLY the `Turn` based system. P0 and P1 takes turn to enter the **critical section**. This could lead to starvation as if P0 never enters CS, P1 starves, violating the independence property.
- If ONLY the `Want[]` based system. P0 or P1 not around, another process can still enter CS, however could result in a DEADLOCK, violating the progress property. Both systems wanting, repeated the while loop.

Assembly Level Implementation

Test and Set: Atomic/machine instruction, aid synchronisation.
Behavior: Entirely behaves like a single atomic machine operation (cannot be broken).

- Load the current content at `MemoryLocation` into `Register`
- Store a 1 into `MemoryLocation`

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

/* Process P0 */          | /* Process P1 */
EnterCS();x+=1;ExitCS(); | EnterCS();x+=1;ExitCS();
```

Employs busy waiting, keeps checking the condition until it is safe to enter, might waste processing power. Does not guarantee bounded-wait out of the box (unless scheduling) is fair.

Semaphore

S contains int value, can be initialized to any non-negative values.

- `Wait( S )`: If `S` ≤ 0, blocks (go to sleep), Decrement `S`. – aka `P()` or `Down()`
- `Signal( S )`: Increments `S`, wakes up one sleeping process if any, this operation **never blocks**. – aka `V()` or `Up()`

Given `S >= 0`, then the following **invariant** must be true:

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

`#signal(S)` – number of signal() operations executed
`#wait(S)` – number of wait() operations completed
Binary Semaphores `S = 1` S an only be 0 or 1. (aka MUTEX, mutual exclusion)

Priority Inversion Solution

Priority inheritance

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

EXAMPLE: The highest and lowest priority tasks running code A and a middle priority task running task B.

```
// Code A
wait(S);
<do some work>
signal(S);

// Code B
<heavy computation>
```

Process Alternative - Threads

Processes are expensive as when creating child processes using `fork()`, it duplicates both memory space and process context. Context switch also requires saving/restoration of process information.
Independent processes have no easy way to pass information to one another, and requires IPC. These requires syscalls to be done which are also expensive.

Multiple threads of control

Using fork() vs threads

```
int result;
result = fork();
if (result != 0) {
    result = fork();
    if (result != 0) {
        steamRice();
    } else {
        fryFish();
    }
} else {
    cookSoup();
}
```

```
int main() {
    Thread 1 { steamRice() };
    Thread 2 { fryFish() };
    Thread 3 { cookSoup() };
    Wait all threads finish;
    printf("Lunch Ready\n");
    return 0
}
```

Process and Thread

- A single process can have multiple threads, threads of same process shares:
  - **Memory Context:** Text, Data Heap
  - **OS Context:** Process id, other resources like files, etc.
- Unique Information needed for each thread:
  - Identification (usually **thread id**), Registers (General purpose and special)

– "Stack" – Stack and CPU cannot be shared. Each thread implemented like a function thus each thread should hvae its own stack and store its own local data and values.

Process Context Switch vs Thread Switch

- Process Context switch involves:
  - OS Context: syscalls
  - Hardware Context: Register values change
  - Memory Context: Remove process from memory and place in a different place
- Thread switch within the same process involves:
  - Hardware context: Registers and "Stack" – changing FP and SP registers

Benefits of Threads

1. Economy: Multiple threads require much less resource than multiple processes
2. Resource sharing: Share most of the resources in a process, no need for IPC to pass information
3. Responsiveness: Multi-threaded programs can appear more responsive
4. Scalability: Multi-threaded programs can take advantage of multiple CPUs

Problems of Threads

1. **System Call Concurrency**: Parallel execution of multiple threads → parallel syscall possible. Have to gurantee correctness and determine the correct behaviour.
2. **Process Behavior**: Impact on process operations. `fork()`, `exit()` and `exec()` called on a single thread, what about other threads or other processes.

User Thread

Thread is implemented as a **user library**. A runtime system (in the process) handles thread related operation. Kernel is **NOT AWARE** of the threads in a process.

ADVANTAGES

- Can have multi-threaded program on ANY OS
- Thread operation are just library calls
- Generally more configurable and flexible

DISADVANTAGES

- OS unaware of threads, scheduling performed at process level
- One thread blocked → Process blocked → all threads blocked, cannot exploit multiple CPU

Kernel Thread

Thread is implemented in the OS, thread operations handled as system calls. Thread-level scheduling is possible (schedules by threads instead of processes.) Kernel may make use of threads for own execution.

ADVANTAGES

- Kernel can schedule on thread levels: More than 1 thread in the same process can run simultaneously on multiple CPUs

DISADVANTAGES

- Thread operations are now a syscall: slower, more resource intensive. Less flexible: Used by all multi-threaded processes

Hybrid Thread Model

Have both Kernel and User threads. OS can schedule on kernel threads only. We can have the User thread bind to a Kernel Thread. Offers greater flexibility, limiting concurrency of any process or user.

- 1-1 binding in hybrid thread model, acts more like a pure kernel thread, since kernel threads can be scheduled.

Hardware Support on Modern Processes: Supply multiple sets of registers (GPRs, Special registers) to allow threads to run natively and in parallel on the same core → simultaneous multi-threading (SMT) or hyperthreading on Intel processor.

POSIX Threads

```
1 int pthread_create( // 0 = success, !0 = error
2     pthread_t* tidCreated, // threadId
3     const pthread_attr_t threadAttribtues, // control↩
          behaviour
4     void* (*startRoutine) (void*). //pointer to fn
5     void* argForStartRoutine );
```

```
1 int pthread_exit( void* exitValue );
```

- If `pthread_exit()` is not used, pthread terminate automatically at end of `startRoutine`. If a "return XYZ;" statement is not used, then "XYZ" captured as exitValue
- Otherwise, exitValue not well defined

pthread Synchronisation

```
1 int pthread_join( pthread_t threadId,
2     void **status ); // exit value returned by ↩
          threadId
3     // used to wait for termination of another ↩
          pthread
```

Classical Synchronisation Problems

Producer-Consumer

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

```
1 //zeroUponFull = N, zeroUponEmpty = 0
2 //Produce              | //Consume
3 wait(zeroUponFull)     | wait(zeroUponFull)
4 wait(mutex)            | wait(mutex)
5 buffer[in] = item      | buffer[out] = item
6 in = (in + 1) % N      | out = (out + 1) % N
7 signal(mutex)          | signal(mutex)
8 signal(zeroUponEmpty)| signal(zeroUponEmpty)
```

Readers Writers

The solution allows either one writer or multiple readers in the "room". It works but can cause starvation of the writer.

```
1 wait(roomLock)       | wait(mutex)
2 //Write data         | nReader++
3 signal(roomLock)     | if nReader == 1:
4                      |   wait(roomLock)
5                      | signal(mutex)
6                      | //Read data
7                      | wait(mutex)
8                      | nReader--
9                      | if nReader == 0:
10                     |   signal(roomLock)
11                     | signal(mutex)
```

Dining Philosophers

5 philosophers seated around a table, 5 single chopsticks placed between each pair of philosophers. When any philosophers wants to eat, they will have to acquire both chopsticks to their right.

- **deadlock-free**, **starve-free** to allow philosophers to eat freely.

TANENBAUM SOLUTION

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

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
        Think( );
        takeChpStcks( i );
        Eat( );
        putChpStcks( i );
    }
}
```

```
void takeChpStcks( i )
{
        wait( mutex );
        state[i] = HUNGRY;
        safeToEat( i );
        signal( mutex );
        wait( s[i] );
}
```

```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

        state[ i ] = EATING;
        signal( s[i] );
    }
}
```

```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

        state[ i ] = EATING;
        signal( s[i] );
    }
}
```

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

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

        signal( mutex );
}
```

- Previously we block on the chopstick → which is between 2 philosophers.
- This solution, each philosopher has its own semaphore, which blocks itself if its partners are eating.
- Being blocked waiting on others could cause a deadlock, this solution gurantees no deadlock.
- OR: 4 seated philisophers, treating pickup of both chopsticks as atomic, or asymmetric (one pick left, one pick right first)

Memory Abstraction

Contiguous Memory Management

Process must be in memory as one piece during the whole execution. To support *multitasking* we allow multiple processes in the physical memory at the same time (switch from one to another). When physical memory is full, we free up memory by removing terminated processes, or swapped blocked processes to secondary storage.

Memory Partitioning

Fixed Partitioning: Leftover space is wasted for any process that does not occupy whole partition: ***internal fragmentation***

- **Pros:** Easy to manage, fast to allocate. Every free partition is the same (no choosing needed).
- **Cons:** Partition size needs to be large enough to contain the largest of processes → smaller process will waste memory space.

Dynamic Partitioning: Free memory space is known as **hole**. With process creation, termination or swapping tend to have large number of holes, also known as: ***external fragmentation***

- **Pros:** Flexible and remove internal fragmentation.

- **Cons:** Need to maintain information in OS, takes more time to locate appropriate region.

Allocation Alrogithms

Assume OS maintain list of partitions and holes, algorithm to locate partition of size `N`. Search for hole with size `M` > `N`.

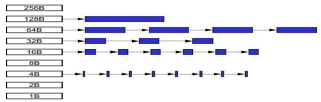
- ***First-Fit*:** Take first hole large enough, minimize search time.
- ***Best-Fit*:** Take smallest hole large, reduce size of hole created.
- ***Worst-Fit*:** Find largest hole, make bigger hole for future process after partition.
- `N` will be new partition, `M - N` will be left over space

Reducing External Fragmentation

- **Merge:** Occupied partition freed, merge with adjacent hole.
- **Compaction:** Move occupied partitions around to create bigger consolidated holes → cannot be invoked too frequently as it is very time consuming.

Multiple Free Lists

- Separate list of free holes from list of occupied partitions
- Keep multiple lists of different hole sizes



Buddy System

- **Buddy** memory allocation provides efficient partition splitting, locating good match for free partition (hole), partition de-allocation and coalescing
- **Main Idea:**
  - Free block is split into half repeatedly to meet request size
  - 2 halves forms a sibling blocks (buddy blocks)
  - When buddy blocks **both free**, merged into larger block

Disjoint Memory Schemes

- Logical page ↔ physical page (frame) mapping not as straightforward
- Need a lookup table to provide translation: **page table**
- Physical Add = *FrameNum* × `sizeof`(physicalframe) + *offset*
- Offset is the displacement from beginning of the physical frame
- Frame sizes (page size) as a power-of-2 → for bit level addressing

Formula

- Given: Page/Frame size of `2^n`, `n` bits of logical addresses
- Logical Address `LA`:
  - `p` = Most significant `m-n` bits of `LA`
  - `o` = Remaining `n` bits of `LA`
- Use `p` to find frame number `f` from mapping mechanism like page table
- Physical Address `PA`: `PA = f * 2^n + o`
- **External Fragmentation:** Not possible, no left-over physical memory region, every single free frame can be used
- **Internal Fragmentation:** Logical memory space not multiple of page size, max one page per process not fully utilized
- Clean separation of *logical* and *physical* address space, good flexibility and simple address translation

Paging Scheme Implementation

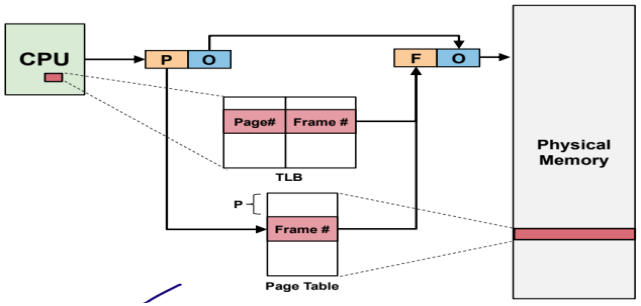
- **Pure-software** solution, OS stores page table information in PCB, memory context of process includes the page table
- Issues: requires 2 memory access for every memory reference (1 to red indexed page table entry to get frame number, 2 to access actual memory item) —— HUGE OVERHEAD

Translation Look-Aside Buffer (TLB)

- **TLB** provided by modern processors, specialized on-chip component to stop paging.
- TLB acts as **cache** for page table entries, reducing redudnat access, very small (10s of entries) and fast (<= 1 clock cycle)

Logical-to-physical address translation with TLB

- Use page number to search TLB associatively
  - **TLB-Hit**: Frame number to generate physical address
  - **TLB-Miss**: Memory access to access the page table, retrieved frame number used to generated physical address and update TLB



TLB: Impact on Memory Access Time

TLB access takes **1ns**, main memory access takes **50ns**, average memory access time if hit ratio is 90%:

$$P(TLBhit) \times latency(TLBhit) + P(TLBmiss) \times latency(TLBmiss)$$
$$= 90\% \times (1 + 50) + 10\% \times (1 + 50 + 50) = 56ns$$

TLB and Process Switching

1 TLB for every physical core (minimally), on context switch:

- TLB entries flushed, new process won't get incorrect translation
- Avoids correctness, security and safety issues
- When a process resumes running, encounter many TLB misses to fill TLB

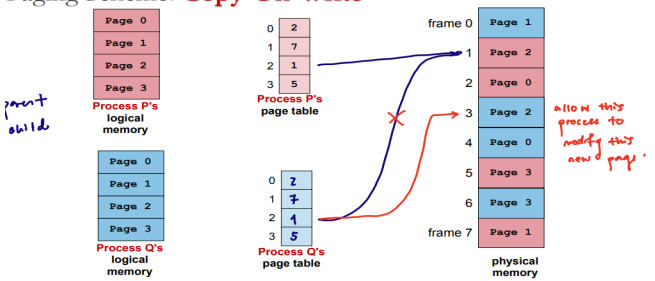
Paging Scheme: Protection

- ***Access-Right Bits***
  - Page Table Entry has writable, readable, executable bits
  - Page containing code: **executable and read only**
  - Page containing data: **readable and writable**
  - Every memory access checked against access bits in hardware
- ***Valid Bit***: Logical memory range is the same for all processes, not all processes utilize the whole range, some pages out-of-range for a particular process
  - Indicate whether the page is valid to access by the process
  - OS set valid bits as memory is being allocated to the process
  - *Every* memory access checked against this bit in hardware, out-out-range access caught by OS

Page Sharing

- Shared code page, some code is being used by many processes: C standard lib, syscalls etc.
- Implementing **Copy-On-Write**: parent and child process can share a page until 1 tries to change a value in it. Mark all pages a READ-ONLY when copied, generate exception if Q wants to write. OS catches, can generate a new copy of pages for Q to write on.

Paging Scheme: **Copy-On-Write**



Segmentation Scheme

Manage memory at level of **memory segments**

- Logical memory of a process: collection of **segments**
- Mapped into contiguous physical partitions of the same size
- Each memory segment has a name and limit (for range of segment), They are specified as `Segment Name + Offset`

Logical Address Translation

- Each segment mapped to a contiguous physical memory region with **base address** and **limit/size**
- Segment name usually represented as a single number → **segmentid**
- Logical address `SegID, Offset`:
  - `SegID` is used to look up `Base, Limit` of the segment
  - Physical Address = `PA = Base + Offset`
  - `Offset` < `Limit` for valid access

ADVANTAGES

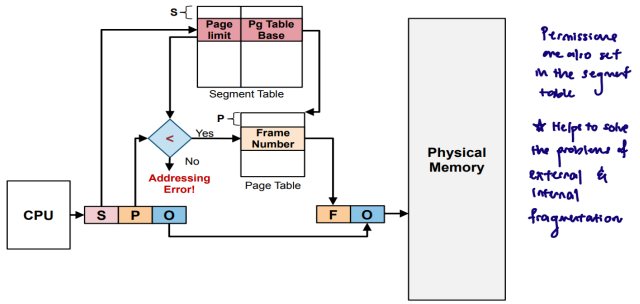
- Each segment is an indepedent contiguous memory space, match programmers view of memory
- More efficient bookkeeping, segment scan grow/shrink and be protected/shared independently

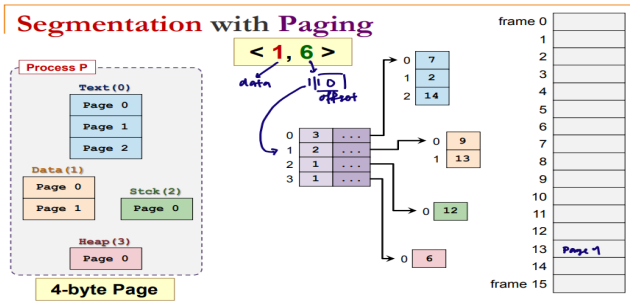
DSIADVANTAGES

- Segmentation requires variable-size **contiguous** memory region → external fragmentation

Segmentation WITH Paging

Each segment is now composed of several pages instead of a contiguous memory region, each segment has a page table. Segment can grow by allocating new page then add to its page table (same for shrinking).





Virtual Memory Management

Split the logical address space into small chunks, some reside in **physical memory**, others are stored in secondary storage.

1. Secondary Storage (HDD/SDD) > Physical Memory Capacity
2. Some pages are access much more often than others

Extended Paging System

New addition to the paging scheme: 2 page types

- **Memory Resident** (pages in physical memory)
- **Non-memory Resident** (pages in secondary storage)
- Use a *resident bit* in the PTE → indicates whether the page is resident in memory
- CPU can only access memory resident pages → Page Fault when CPU tries to access non-memory resident page. OS brings non-memory resident page into physical memory

Access Page X: Steps

1. Check page table: Is page X memory resident?
   - Yes: Access physical memory location. No: raise exception
2. Page Fault: **OS takes control**
3. Locate page X in secondary storage
4. Load page X into physical memory and update page table
5. Back to step 1, **re-execute same** instruction

Issues with Virtual Memory

Secondary Storage access time >> Physical memory access time (5 orders of magnitude).

- If memory access results in **page faults** often to load non-resident pages, slow system down, leading to thrashing

Locality Principles

- **Temporal:** Address used now ***likely to be used again***
- **Spatial:** Addresses close to address is likely to be used soon

Demand Paging —— Laziness

Processes start with no memory resident page, only allocate a page when there is a page fault.

- **Pros**: Fast startup, small memory footprint
- **Cons**: Process appears slow initially due to many page faults, page faults might have a cascading effect on other processes (thrashing)

Page Table Structures

Direct Paging $\#pages = \frac{2^{Virtual\ Address\ Space}}{Page\ Size}$

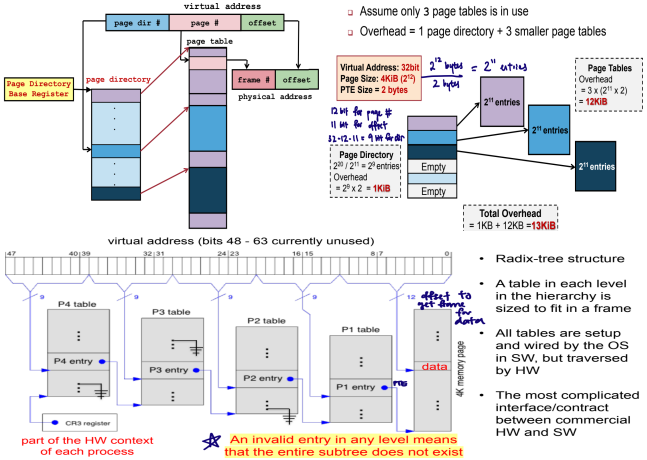
Keep all entries in a single table AND page tables must be **contiguous** in OS memory.

- With 2^p pages in logical memory space: **p** bits to specify one unique page. 2^p PTE with Physical Frame Number + Additional Information (valid, readonly etc)
- Virtual Address: 32 bits, Page Size = 4KB $\approx 2^{12}$
- P = 32 - 12 = 20, size of PTE = 4 bytes, Page Table Size = 2^{20} * 4 bytes = 4MB
- Even if page tables do not fit in a frame, OS must ensure that it is laid-out consecutively in its own memory.

2-Level Paging — Paging the page table

$Overhead = (\#PDE \times PTE\ Size) + (\#PT \times Page\ Size)$

- Split whole page table into regions, only a gew regions are used.
- As memory usage grows, new regions can be allocated.
- Directory is used to keep track of regions, analogues to the relationship between page tables and data pages.
- If original table has 2^p entries,
  - With 2^M smaller pages tables, **M** bits used to uniquely identify one page table
  - Each smaller page table contains 2^{p-M} entries



ADVANTAGES

- Enables page tables to grow beyond size of frame, no need to be contiguous in OS memory
- Can have empty entries in the page directory

DISADVANTAGES

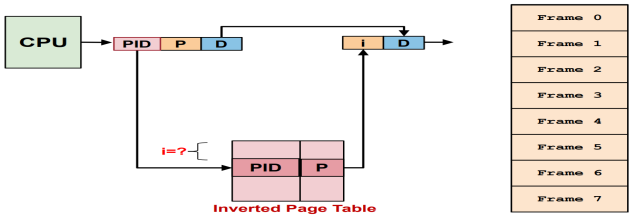
- Requires 2 serialized memory address just to get frame number (1 dir, 1 page table)
- **TLB:** TLB hits eliminate page-table accesses, but TLB misses experience longer **page-table walks**.
- **MMU Caches**: additional hardware in modern processors
  - tiny cache that caches frequent page directory entries, speed up page-table walk upon TLB miss.
  - We do not want directory to kick out a PTE in TLB, thus use MMUs specifically for Directories.

Inverted Page Table $\#\ frames = \frac{RAM\ size}{frame\ size}$

Keep a **single** mapping of physical frame to `<pid, page#>`

- pid = process id, page# = logical page number in process
- page# is not unqiue among processes

Normal vs Inverted Page Table: Entries ordered by page number vs frame number, lookup page X, simply access X^{th} entry directly vs search whole table. ADVANTAGES: Huge saving, one table for all processes. DISADVANTAGES: Slow translation



Page Replacement Algorithms

Which page should be replaced when needed. No free physical memory frame during a page fault, need to evict a memory page.

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

Evaluation

A good algorithm should minimize the total number of **page faults** and be fast! **Memory access time:**

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

- `p` = probability of page fault
- `T_mem` = access time for memory resident page
- `T_pageFault` = access time if page fault occurs
- Since `T_pageFault` >> `T_mem`, need to reduce `p` to keep `T_access` reasonable

1. Optimal Page Replacement

Replace the page that **will not** be needed again for *longest period of time*. **Guarantees minimum** number of page faults. Not realizable as **need future knowledge** of memory references. Closer to OPT, better algorithm.

2. FIFO Page Replacement

Memory pages evicted based on loading time → evict oldest memory page.

- OS maintains queue of resident page numbers
- Remove the first page in queue if replacement is needed
- Update queue during page fault trap
- **Simple** to implement, no hardware support needed

Problems with FIFO:

- Number of physical frame increases, number of page faults should increase
- Opposite behavior (↑ frames → ↑ page faults)
  - *Belady's Anomaly*, does not exploit **temporal locality**
  - Bad performance in practice

3. Least Recently Used (LRU)

Make use of temporal locality, honour notion of *recency*.

- Replace page that has not been used in the longest time
- Aims to approximate OPT algorithm, predicting the future by mirroring the past

IMPLEMENTATION DETAILS

- Need to keep track of 'last access time', need substantial hardware support
- **A. Use a Counter**
  - Logical time counter, increment on every memory reference
  - Page table entry has time-of-use field, store time counter value whenever references occurs
  - Replaced page with smallest time-of-use

  - **Problems:** Need to search through all pages, time-of-use is forever increasing (overflow possible)
- **B. Use a Stack**
  - Maintain stack of page numbers, if **X** referenced, remove from stack and push to top.
  - Replace the page at **bottom of stack**, no need to search through all entries
  - **Problems:** Not a pure stack, since entries can be moved from anywhere in stack → hard to implement in hardware.

3. Second-Chance Page Replacement — CLOCK

Modified FIFO to give second change to pages accessed. Each PTE now maintains a *referenced bit.*

- 1 = Accessed since last reset, 0 = not accessed
- Algorithm:
  1. Oldest FIFO page selected
  2. If reference bit == 0 → page replaced, done.
  3. If reference bit == 1 → page skipped, given 2^{nd} chance
     - Reference bit cleared to 0, resets arrival time, page taken as newly loaded. Next FIFO page selected, go to Step 2
- Becomes a FIFO algorithm when all page reference bits == 1
- Adds notion of recency, works well in practice

Page Table Structures

- **Equal Allocation:** Each process gets `N / M` frames
- **Proportional Allocation:**
  - Processes different in size (memory usage)
  - Let `size_p` = size of process `p`, `size_total` = total size of all processes
  - Each process get `size_p/size_total * N` frames

Local replacement: Victim page are selected among pages of the process that causes page fault

- **Pros**: Number of frames allocated to a process remain constant → Performance is stable between multiple runs.
- **Cons**: If the frames allocated are not enough → hinder the progress of a process.

Global replacement: Victim page can be chosen among all physical frames: → Process P can take a frame from Process Q by evicting Q's frame during replacement!

- **Pros**: Allow self-adjustment between processes → Process that needs more frame can get from others that need less.
- **Cons**: Badly behaved processes can affect others. Number of frames allocated to a process can be different from run to run.

Thrashing

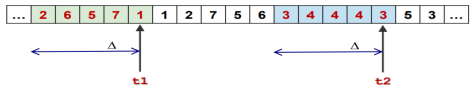
Insufficient physical frames → Thrashing. Heavy I/O to bring non-resident pages into RAM.

- Hard to find right number of frames:
  - **Global used**: thrashing process 'steals' page from other process, cause other process to thrash (cascading thrashing)
  - **Local used**: thrashing limited to one process, but single process can use up I/O bandwidth, degrade performance of other processes

Working Set Model

Set of pages referenced by process is relatively constant in a period of time —— known as **working set**. As time passes, set of pages can change as different program phases require different data.

- Working set stable and well-defined, rare for page faults
- When transitioning to new working set, many page faults for new set of pages
- Defines a **Working Set Window** △ → an interval of time
- `W(t, △ )` = active pages in interval time `t`
- Allocate enough frames for pages in `W(t, △ )` to reduce possibility of page fault.
- Too small, miss pages in current working set, Too big, contain page in different working set
- Assume △ = an interval of 5 memory references
- `W(t1, △ ) = {1,2,5,6,7}` (5 frames needed)
- `W(t2, △ ) = {3,4}` (2 frames needed)



File System Introduction

It is an abstraction on top of physical media, high-level resource management scheme. It must ensure protection between processes and users, and be shared between processes and users.

- **Self-Contained:** Information stored on media enough to describe entire organisation → plug and play
- **Persistent:** Beyond the lifetime of OS and processes
- **Efficient:** Provides good manangement of free and unused space + minimum overhead of bookkeeping information

File Overview

File Name: Different FS have different naming rules: length of file name, case sensitivity, allow special symbols and file extensions.

File Type:

1. **Regular files:** contains user info (ASCII/binary files)
   - *ASCII:* text file, programming source code
   - *Binary files:* executable, Java `.class` files, `.pdf` file, `png` etc. Have a predefined internal structure that can be pprocess by a specific program (JVM for Java `.class`)
2. **Directories:** system files for FS structure
3. **Special files:** character/block oriented
4. Distinguished by **file extension** (Windows → `XXX.docx` Word Document) OR **embedded information** (Unix), usually stored at beginning of file → magic number

File Protection:

Accessing information: Read/Write/Execute/Append/Delete/List

- Permission Bits
  - **Owner:** User who created the file
  - **Group:** Set of users who need similar access to a file
  - **Universe:** All other users in the system
- Access Control List: Minimal ACL (same as permission bits) or Extended ACL (added named users/group)

File Structure

- **Array of bytes**: each byte has unique offset from the file start
- **Fixed length records:** Array of records can grow/shrink, can jump to any record easily.
- **Variable length records:** Flexible but hard to locate.

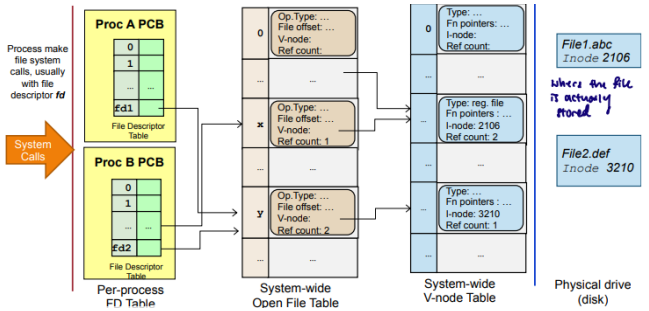
Access Methods

- **Sequential Access:** Data read in order, start from beginning but cannot skip can rewound
- **Random Access:** Read in any order, `Read( Offset )`, `Seek( Offset )` to move to new location in file
- **Direct Access:** Used for file that contains fixed-length records, allow random access to any record directly. Useful when there is large amount of records.

File Information:

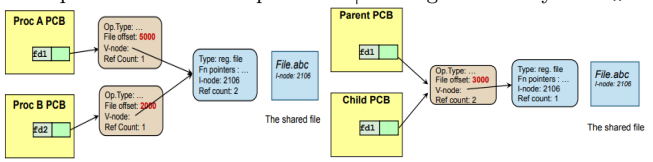
Operations: Create, Open, Read, Write, Repositioning (seek), Truncate Information kept for an open file

- **File pointer:** keep track of current position within file
- **File descriptor:** unique identifier of the file
- **Disk Location:** actual file location on disk
- **Open Count/Reference count:** how many processes has this file open (used to determine when can remove from table.)
- **Per-process open-file table:**
  - Keep track of open files for a process
  - Each entry points to the **system-wide open-file table** entries → if `open()` is called twice, there will be 2 seperate `fd`
- **System-wide open-file table:**
  - Keep track of all open files in the system
  - Each entry points to a **V-node** entry
- **System-wide V-node(virtual node) table:**
  - Link with the file on physical drive
  - Contains the information about the physical location of the file



- V-Node table consists of Files that are **open**, in RAM
- I-Node is in harddisk, that consists of data of ALL files.

File opened twice from 2 processes | 2 using same entry `fork()`



Directories

Tree-Structured

- Directories recursively embedded in other directories
- **Absolute Path:** Dir names followed from root of tree + final file, path from root dir to file
- **Relative Path:** Dir names from the current working directory (CWD), can be set explicitly or implcitly changed by moving into a new dir under shell prompt

DAG —— Unix Hard Link `ln`

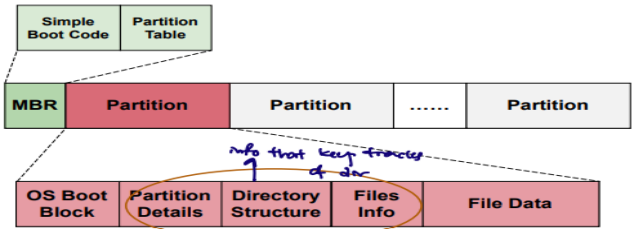
- Dir `A` owns file `F`, Dir `B` wants to share `F`
- `A` and `B` has separate pointers point to actual file F in disk
- **Pros:** Low overhead, only pointers added in directory.
- **Cons:** Deletion problems, if use reference count, if count > 1 do not delete.

DAG —— Unix Sym/Soft Link `ln -s`

- Symbolic link is a **special link file** `G`, contains path name of `F`
- When `G` accessed: find out where is `F`, then access `F`
- **Pros:** Simple deletion, if symbolic link deleted (`G` deleted), `F` not deleted. If linked file is deleted: `F` is gone, `G` remains but is dangling link.
- **Cons:** Larger overhead, special link file take up space.

File System Implementation

General **Disk Structure** is a 1D array of logicial blocks (smallest accessible unit). Logical block is mapped into disk sector(s), hardware dependent.



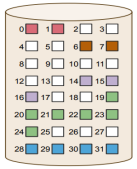
- **Master Boot Record** at sector 0 with partition table
- Each partition can contain an indepedent file system
- File system generally contains the following:

Implementing File

Within the `Files Info` + `Files Data` in Disk Organisation.

File Block Allocation: *Contiguous*

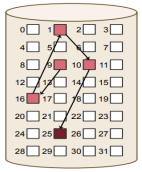
Allocate consecutive disk blocks to files



- **Pros:** Simple & fast access (only need to seek first block)
- **Cons:** External Frag. File size needs to be specified in advance.

File Block Allocation: *Linked List*

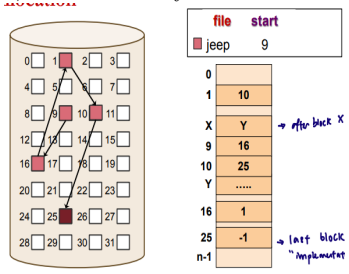
Keep linked list of disk blocks. Each disk block stores next disk block number (act as **pointer**) + actual file data. File information stores first and last disk block number.



- **Pros:** Solve fragmentation problem.
- **Cons:** Random file access is very slow. Part of disk block used for pointer → what if one of the pointers is wrong (less reliable)

File Block Allocation: *File Allocation Table (FAT)*

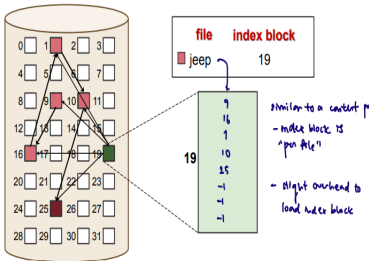
All block pointers in a single table, FAT in memory all the time.



- **Pros:** Faster than random access, linked list traversal now takes place in memory.
- **Cons:** FAT keeps track of **all disk blocks** in partition. Can be huge when disk is large, expensive overhead.

File Block Allocation: *Indexed Allocation*

Instead of keeping track of every file in system, maintain blocks for each file. Each file has **index block**. Index block is an array of disk block addresses, `IndexBlock[N]` == N^{th} block address.



- **Pros:** Less memory overhead, only index block of opened file needs to be in memory. Fast direct access.
- **Cons:** Limited maximum file size and Index block overhead. Max number of blocks == Number of index block entries

VARIATIONS of indexed block allocation

- **Linked:** Linkedlist of index nodes, expensive traversal cost
- **Multi-level Index:** Similar idea to multi-level paging, can be generalized to any number of levels
- **Combined scheme:** Combination of direct indexing and multi-level index scheme, fast access for small files but can handle large files

Free Space Management

Within `Partition Details` of Disk Organisation. To perform file allocation, we need to know which disk blocks are free.

- **Allocation:** Remove free disk blocks from free space list. (when file created/enlarged)
- **Free:** Add free disk blocks to free space list (when file deleted/truncated)

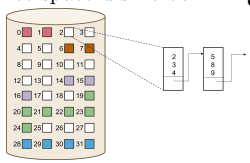
Bitmap

Each disk block represented by 1 bit: `0 1 0 1 1 1 0 1 0 1 1`
Occupied: 0, 2, 6, 7, 9 ... — Free: 1, 3, 4, 5, 8, 10, 11 ...

- **Pros:** Good set of manipulations
- **Cons:** Need to be kept in memory for efficiency

LinkedList

Use linkedlist of blocks, each disk block contains (1) number of free disk blocks OR (2) pointer to the next free space disk block.



- **Pros:** Easy to locate free block, only first pointer needs to be in memory. (other blocks cached)
- **Cons:** High overhead.

Directory Structure

Within `Directory Structure` of Disk Organisation. Keeps tracks of files in directory and map file name to file information. `open()` operation is to locate file info using `pathname` + `file name`

- Given full path name, need to recursively search directories along path to arrive at file information
- Sub-directories stored as file entry with special type

Linear List

Each entry in list represents file. Stores file name (minimally) and file metadata → **file information** / **pointer** to file information. File located by **linear search**, inefficient for larger directories / deep traversals. (use cache to remember recent searches).

Hash Table

Each directory contains hash table of size `N`. To locate, filename hashed into index `K` from `0` to `N-1`. `HashTable[K]` inspected to match file name. (usually chained collision resolution used.)

- **Pros:** Fast lookup, *O(1)* versus *O(n)* in linear list.
- **Cons:** Limited size Hash table & depend on good hash functions.

File Information

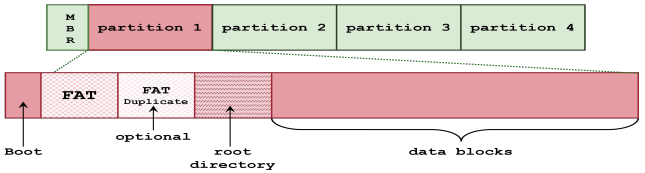
File info consists of file name and other metadata + disk blocks information.

- Store everything in directory entry
- Store only file name, point to DS for other information

Microsoft FAT File System

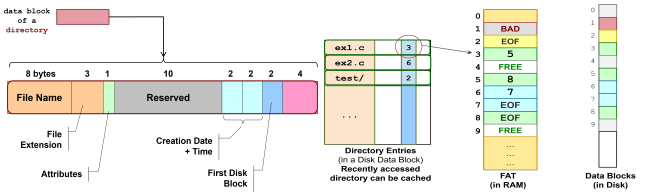
File Data + FAT Implementation

- File data allocated to some data blocks / block clusters
- Allocation info kept as **linked list**, data block pointers kept separately in FAT
- One entry per data block/cluster. Store disk block info. Os will cache in RAM to facilliate linkedlist traversal



Directory Structure and File Info

- Folder represented as special type of file. Root directory stored in special location (others stored in data blocks.)
- File/subdirectory in a folder represented as 1 directory entry.
- **Directory Entry** are fixed size 32-bytes. Contains: Name + Ext, Attributes (RDONLY, Dir/File flag, Hidden etc), Creation Date + time, First disk block + File size.

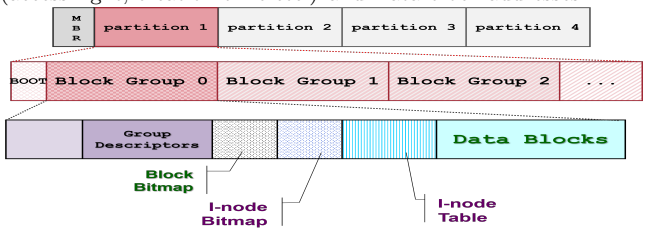


Directory Entry Fields

- **File Name + Extension:** Limit to 8+3 chars. First byte of file name may mean: deleted, end of dir entries, parent dir etc.
- **File Creation Time and Date:** Year limtied to 1980 to 2107, accuracy of seconds +- 2s
- **First Disk Block Index:** Different variants use different number of bits, FAT32 = 32bits

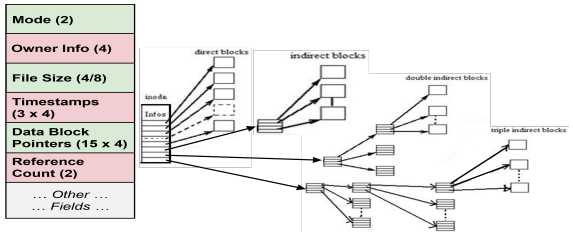
Extended-2 File System — Linux

Disk space split into **Blocks**, grouped into **Block Groups**. Each File/Dir is described by an **I-Node**, containing File metadata (access right, creation time etc.) and Data block addresses.



Partition Information

- **Super Block:** Describes whole FS: Total I-Node number, I-Node per group. Total disk blocks, disk blocks per group.
- **Group Descriptors:** Describe each of the group, number of free disk blocks, free I-Nodes, location of bitmaps. Duplicated in each block group as well.
- **Block Bitmap:** Track usage of status of blocks in block group (1=Occupied, 0=Free)
- **I-Node Bitmap:** Track usage of status of I-Nodes in block gorup
- **I-Node Table:** Array of I-Nodes, each I-Node access by its index, contains only I-nodes of this block group



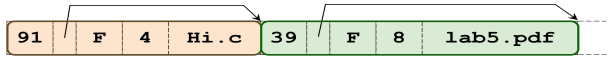
I-Node Data Blocks

15 Disk block pointers, first 12 pointers are **Direct Blocks**, 13^{th} **Single Indirect**, 14^{th} **Double Indirect**, 15^{th} **Triple Indirect**. Direct blocks point directly to disk blocks, indirect blocks points to a block of pointers (also stored on data blocks).

Directory Structure

Directory data blocks stores a **linked list of dir entries** for file / subdirectories information. Each directory entry contains:

- I-Node number for file/subdirectory
- Size of directory entry
- Length of file/subdirectory name
- Type: File or Subdirectory or Special File
- File/Subdirectory Name (255 chars max)



Ext2: Hard vs Symbolic Link

- **Hard Link:** With multiple references of a I-Node, hard to determine when to delete an I-Node. Maintain I-Node reference count. decrement for every deletion.
- **Sym Link:** Only the pathname is stored, the link can be easily invalidated: File name changes / deletion etc. Involve a search to locate actual I-Node number of target file.

Tutorial + PYP Questions

pthread Mutex and Conditional Variables

- Synchronisation mechanism for `pthreads`
- Mutex (`pthread_mutex`)
  - Binary semaphore (i.e. equivalent Semaphore(1))
  - Lock: `pthread_mutex_lock()`
  - Unlock: `pthread_mutex_unlock()`
- Conditional Variables (`pthread_cond`)
  - Wait: `pthread_cond_wait()`
  - Signal: `pthread_cond_signal()`
  - Broadcast: `pthread_cond_broadcast()`
- **Semaphore vs POSIX thread condition variable signal:** Semaphore signal increments count and may wake a waiting thread. If there is no thread sleeping on semaphore, it allows next thread requesting a "down" to proceed without blocking. A conditional signal has no concept of count. It wakes up one thread currently sleeping on the condition, has no (lasting) impact on later threads that may be waiting on condition.

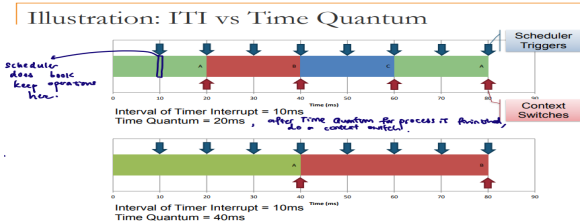
Context Switching

- Involve process transitioning from READY state to RUNNING state, other process **being switched out** of is still in the RUNNING state.
- Before switching, current PC should be saved in the PCB to restore process execution when process gets CPU back.
- Context switching can still occur even if interrupts are disabled – though a context switch can occur as a result of an interrupt.

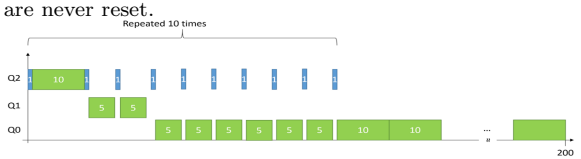
Critical Section —— Ordering of Mutex

Suppose a critical section requires M mutex locks before it can be executed. Show why if every process acquires the locks in the same order, then a deadlock cannot occur. Would it also matter if the processes *unlock* the mutex lock in different orders?

- Suppose the M locks are labelled L_1 to L_M. All threads competing for the locks in the same order ensures that L_1 in effect acts like the lock for L_2 to L_M as well as the critical section. There can be no deadlock because no other thread can acquire any of L_2 to L_M without first contending for L_1.
- Doesn't matter because the locks has to be acquired in the same order. Even if another thread release lock L_i first before L_1, until it releases L_1, no other thread can proceed beyond L_1.



MLFQ Task Scheduling: Suppose there are three priority queues, Q0, Q1, and Q2 with Q0 having the lowest and Q2 having the highest priority. Now suppose there are 2 tasks, both present at the start. Task A is I/O bound, it will run for 1 time unit on the CPU before blocking for I/O for 5 time units. This is repeatedly 9 times before 1 final 1 time unit CPU execution to finish off the task. Task B is CPU bound and will run for 200 time units. Assuming that the time quantum is set at 10 time unit and that the OS takes no overhead, show the schedule under Multi-level Feedback Queue for 200 time units with Task A starting the schedule. For simplicity, we assume that the queues are never reset.



Task A (in blue) gets to run in Q2 throughout its lifetime coz it does not use up 10 time units – within the time quantum. In the first time that it blocks, Task B (in green) will be in Q2 with the same priority as Task A. So even though Task A is runnable at time unit 6, it will have to wait till Task B completes its time quantum. However, after this Task B will be demoted to Q1, exhaust it time quantum then gets demoted again to Q0, resulting in the above. While in Q1, Task B will be interrupted when Task A is ready to run and will have to give up the CPU to Task A.

File Descriptors: A process has 3 threads, T1, T2, and T3, and its code does not contain any `exec*()` commands. T1 opens 3 files and T2 creates a pipe. After these actions, T1 executes a `fork()` command and T2 executes a `fork()` whose child immediately calls `execvp()` to execute a single-threaded program. T1 closes the three files and then terminates. T3 terminates without opening any file. Note that all threads are POSIX threads.

⋆ On a Unix-like operating system, the first three file descriptors, by default, are STDIN, STDOUT and STDERR. T1 creates 3 fds, T2 creates 2 fds. There are 8 fds in the process now. After this, there are 2 fork() calls → each process will duplicate the fd table with 8 fds: 24 fds are created in total.

- If threads read from same fd, they would read different content, since offset is shared in the system-wide open file table.
- Suppose each thread opens file indepedently, fd is different, thus content read is the same, as both reads the entire file.

Virtual Memory Address vs Virtual Address Space: 32 bit virtual *memory addresses*, and question 5 mentions 2^{64} bit *address space*. Each address points to 1 byte in memory, so 32 bit memory address will be able to represent 2^{32} bytes. Therefore the *address space* is 2^{32} bytes or 2^{35} bits. However, 2^{64} bit address space is the total amount of data *memory addresses* point to, and hence we have to convert to bytes, as 2^{64} bit address space doesn't mean 64 bits are used for addressing.

Single-Core Architectures: Every instruction on a single-core machine is effectively atomic, including the ordinary increment instruction on S1. Therefore, for S1, as long as the entire critical section fits in one instruction, which is the case here, there is no way to violate mutual exclusion. In all other cases, mutual exclusion is not guaranteed and race conditions are possible.

Page Replacement Algo: Given 5 physical frames for a process. After the following sequence of page accesses(memory reference string), what are the pages in RAM sorted from first frame to last frame? Sequence: 3, 2, 4, 5, 1, 5, 7, 4, 7, 6, 3, 5

1. **LRU:** 7, 6, 4, 5, 3 → ~~3~~ 7, ~~2~~ 6, 4, 5, ~~1~~ 3
2. **FIFO:** 7, 6, 3, 5, 1 → ~~3~~ 7, ~~2~~ 6, ~~4~~ 3, 5, 1

```
1  clock_algo(incoming_page):
2      if (present_in_frame(incoming_page)):
3          update same_page_in_frame.bit to 1
4          clock_algo(receive_next_page)
5      else: // incoming page not present in frame
6          if (victim_pointer.page.bit == 1):
7              victim_pointer.page.bit = 0
8              victim_pointer = next_page_in_frame
9              clock_algo(incoming_page) // REDO PAGE
10         else: // MOVE ON TO NEXT PAGE
11             swap(victim_pointer.page)
12             victim_pointer = next_page_in_frame
13             clock_algo(receive_next_page)
```