# CS2106 Intro Op. Systems Notes

AY23/24 Sem 2, github.com/gerteck

## 1. Introduction

**Course objectives:** Introduces basic concepts in operating systems.
Focusing on:
- OS structure and architecture, process management, memory management, file management and OS protection mechanism.
- Identify and understand major functionalities of modern operating systems.
- Extend and apply the knowledge in future courses.

**Supplementary Text**: Modern Operating System (5th Edition), by Andrew S. Tanenbaum, Pearson, 2023.

### Learning Outcomes

- Understand how an **OS manages computational resources for multiple users and applications, and the impact on application performance**
- Appreciate the **abstractions and interfaces provided by OS**
- Write **multi-process / thread programs** and avoid common pitfalls such as **deadlocks, starvation and race conditions.**
- Write system programs that utilizes **POSIX** syscall for process, memory and I/O management.
- Self-learn and explore advanced OS topics.
- Understand important design principles in complex systems.

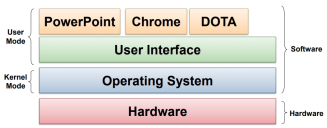
Areas to focus on: Try to understand how things are running in parallel, since we naturally think sequentially. Secondly, how we can manage memory and how they combine and interact (in strange ways), synchronization.

### Operating System OS

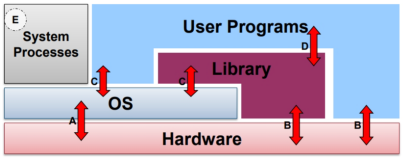
An OS is a program that acts as an intermediary between a computer user and the computer hardware. Motivation for OS:

- Manage resources and coordination. (Resource Allocator: Process synchronization, resource sharing)
- Simplify programming (Abstraction of hardware / hardware virtualization, convenient services)
- Enforce usage policies
- Security and protection
- User Program Portability (across different hardware)
- Efficiency (Optimize for particular usage and hardware).

**Kernel Mode**: Complete access to all hardware resources.
**User Mode**: Limited / Controlled access to hardware resources.




### Generic OS Components




- **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 is known as the **kernel**.
  Program that deals with hardware issues, provide system call interface and special code for interrupt handlers, device drivers.
- Kernel code is different from normal programs:
  No use of system call in kernel code, can't use normal libraries, no normal I/O (must do I/O itself).
- **Implementing OS**: Historically in assembly/machine, now in HLLs (C, C++). Heavily hardware architecture dependent. Challenges include complexity, debugging, codebase size.

### OS Structures

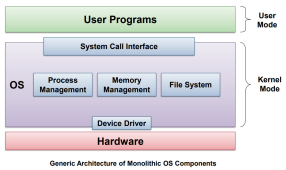
**Monolithic OS**: One Big program.

- Well understood, good performance, but highly coupled components (everything running in kernel mode) and usually devolved into very complicated internal structure.

**Microkernel OS**:

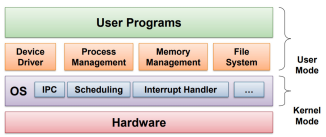
- Kernel is very small and clean, only providing basic and essential facilities.
- Inter-Process Communication (**IPC**), Address space management, Thread management etc.
- Higher level services are built on top of basic facilities, run as server process *outside* of OS, use IPC to communicate.
- Kernel is more robust and extendible, better isolation and protection between kernel and high level services. But, lower performance. (Latency)

Monolithic Kernel Illustration




Generic Architecture of Monolithic OS Components

Microkernel Components




Generic Architecture of Microkernel OS Components

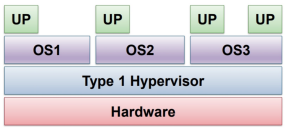
**Other OS Structure**

- **Layered Systems**: Generalization of monolithic system, organize components into hierarchy of layers. Lowest is hardware, highest is user interface.
- **Client-Server Model**: Variation of microkernel. Two classes of processes: Client p. request service from server process, server process built on top of microkernel. Client & Server process can be on separate machine.

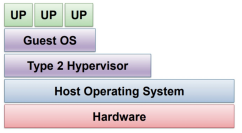
### Virtual Machines

- **Motivation**: OS assumes total control of hardware, making it hard to run several OS on same hardware at same time. OS is also hard to debug / monitor, hard to observe working of OS, test potentially destructive implementation.
- **Virtual Machine**: Software emulation of hardware.
- **Virtualization of underlying hardware**: Illusion of complete hardware to level above. (Memory, CPU etc.) Normal OS can then run on top of virtual machine. Aka **Hypervisor**.
  - **Type 1 Hypervisor**: Provides individual virtual machines to guest OSes (e.g. IBM VM/370)
  - **Type 2 Hypervisor**: Runs in host OS, Guest OS runs inside Virtual Machine, (e.g. VMware)

Type 1 Hypervisor




Type 2 Hypervisor




## - Upcoming Topics -

**OS Process Management**: As OS (to maximise efficiency hardware resources), to be able to switch from running one program to the other (share hardware, e.g. CPU), requires information regarding execution of A stored, and A's information replaced with B's information to run. (E.g. the registers in CPU replaced)

- **2. Process Abstraction**: Info describing executing program
- **3. Process Scheduling**: Deciding which process gets to execute
- **4. Inter-Process Communication**: Passing information between processes (tough)
- **5. Threads + Synchronization**: Alternative to Process (Light-weight process aka Thread)
- **6. Memory Management**
- **7. Disjoint Memory Management**
- **8. File System Management**
- **9. File System Implementation**

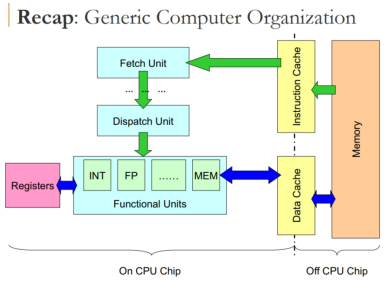
# 2. Process Abstraction

To switch programs, requires information of both programs. Hence, we need abstraction to describe running program, aka **process**.

- **(Process / Task / Job)** is a dynamic abstraction for executing program.
- It is info required to describe a *running program*:
  - Memory Context (Code/Text, Data, Stack, Heap),
  - Hardware Context (Register/PC, Stack/Frame Pointer),
  - OS Context (Process Properties (PID, State), Resources Used).

## Computer Organization (Recap)

- **Components**: Memory, Cache, Fetch Unit (Loads instruction, location indicated by special register **PC**.)
- **Functional Units** (Carry out instruction execution, dedicated to diff. instr. type) (CS2100 looked at INT func. unit)
- Registers (Internal storage, fastest access speed).
  - **GPR**: General Purpose Register, accessible by user program / compiler.
  - **Special Registers**: PC, Stack/Frame Pointer, PSW etc.
- **Binary Executable File**: file in machine language (built by compiler) for specific processor:
  - Executable (binary) consists two major components: Instr. (Text) & Data
  - When under execution, more info: Memory, Hardware, OS context.




## Memory Context for Function Call (Stack Memory)

### Memory Context Challenges of Functional Calls:

- Control Flow Issues: Need to jump to function body, resume after, need to store PC of caller.
- Data Storage Issues: Need to pass params to function, capture return result, may need declare local variables.
- Require region of memory dynamically used by function invocations.

Hence, portion of memory space used as **stack memory** that stores executing function using **stack frame**, which includes usage of *Stack Pointer, Frame Pointer*.

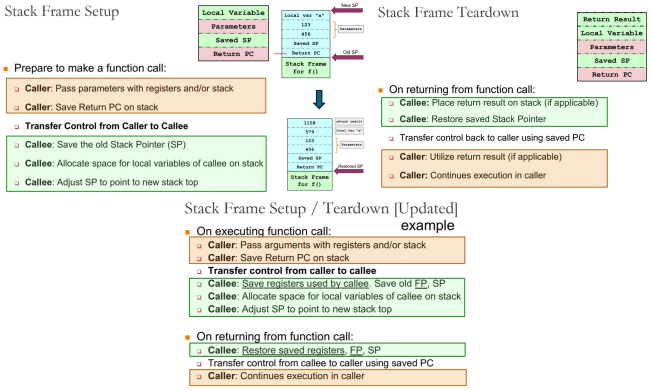
### Stack Memory Region

- **Memory region to store information function invocation.**
- **Stack Frame**: Describes information of function invocation.
- Stack frame added on top when function is invoked, stack "grows", removed from top when function call ends, stack "shrinks".
- Stack Frame contains return PC address of caller, arguments for function, storage for local variable, etc.
- **Stack Pointer**: Indicates top of stack region (first unused memory location). Usually indicated in specialized register.

## Function Call Convention: Stack Frame Setup / Teardown

There are different ways to setup stack frame, known as function call convention, differences about (info stored in frame, which portion of stack frame prepared & cleared by caller / callee etc). Dependent on hardware & programming language.

### Example Scheme:



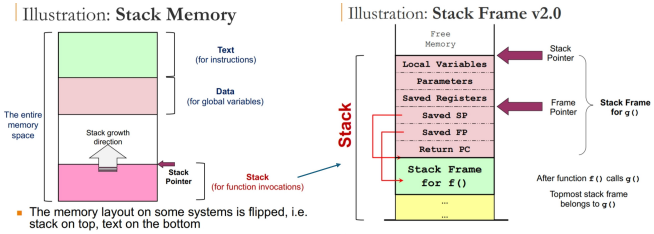

## Stack Frame: Other Information

### Frame Pointer

- To facilitate access of various stack frame items. As stack pointer hard to use as it can change, some processors provide dedicated register Frame Pointer.
- Frame Pointer points to fixed location in stack frame, other items accessed as displacement from frame pointer, usage of FP is platform dependent.

### Saved Registers

- Since number of GPR limited, when GPR exhausted, use memory to temp. hold GPR values for reuse.
- **Known as Register Spilling**. Function can spill registers it intends to use before function starts, then restore registers at end of function.



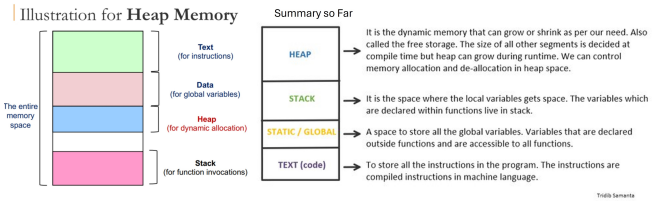

## Memory Context for Dynamically Allocated Mem. (Heap)

Most programming languages allow dynamically allocated memory, i.e. acquire memory space during execution time.

- In C, `malloc()` function call, while C++ / Java: `new` keyword.
- Cannot place in *Data region*, as allocated at runtime, size not known during compilation.
- Cannot place in *Stack region*, as no definite deallocation time, cannot be freed by garbage collector.
- Solution: Set up separate **Heap Memory Region**

## Heap Memory

- Managing heap memory trickier due to variable size, variable allocation / deallocation timing.
- Common situation where heap memory alloc/dealloc creating "holes" in memory. Free memory block squeezed between occupied memory blocks.
- Covered in memory management.



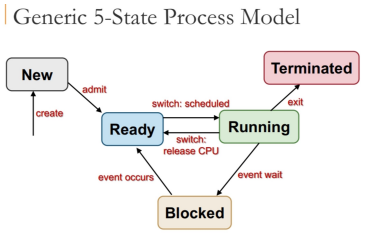

## OS Context: Process ID, Process State

### Process Identification:

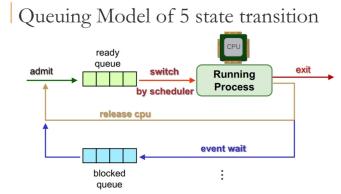
- **Process ID**: To distinguish processes from each other (Just a number, unique among processes)
- PIDs are OS dependent as well, including if PIDs reused, if limits maximum no. of processes or any PIDs reserved.

### Process State:

- Processes require a process state as indication of execution status. (Running / Not Running / Ready to Run etc.)
- **Process Model**: Set of states and transitions, describes behaviors of a process.
- **Global View of Process States**: Given $n$ processes,
  - With 1 CPU, $\leq 1$ process in running state, 1 transition at a time.
  - With $m$ CPUs, $\leq$ m processes running state, possibly parallel transitions.
- Different processes may be in different states, each process may be in different part of its state diagram.
- **5-State Process Model**:




Notes: generic process states, details vary in actual OS




Notes:
- More than 1 process can be in ready + blocked queues
- May have separate event queues
- Queuing model gives global view of the processes, i.e. how the OS views them

### Process States for 5-Stage Model

- **New**:
  - New process created
  - May still be under initialization → not yet ready
- **Ready**:
  - process is waiting to run
- **Running**:
  - Process being executed on CPU
- **Blocked**:
  - Process waiting (sleeping) for event
  - Cannot execute until event is available
- **Terminated**:
  - Process has finished execution, may require OS cleanup

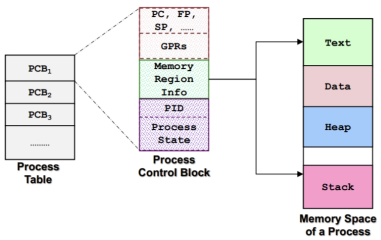
### Process State Transitions in 5-Stage Model

- **Create** (nil → New):
  - New process is created
- **Admit** (New → Ready):
  - Process ready to be scheduled for running
- **Switch** (Ready → Running):
  - Process selected to run
- **Switch** (Running → Ready):
  - Process gives up CPU voluntarily or *preempted* by scheduler
- **Event wait** (Running → Blocked):
  - Process requests event/resource/service which is not available/in progress
  - Example events:
    - System call, waiting for I/O, *(more later)*
- **Event occurs** (Blocked → Ready):
  - Event occurs → process can continue

## Process Table & Process Control Block

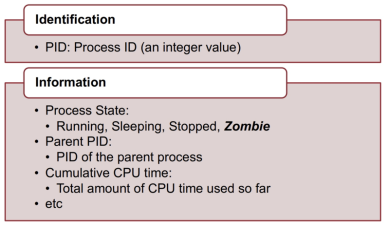
- Since the OS is just a program as well, need to make use of data structures to track these processes.
- **Process Control Block (PCB) or Process Table Entry**: Entire Execution Context for a process.
- Kernel maintains **PCB** for all processes. (Conceptually stored as one table representing all processes.)
- **Factors to consider**:
  - Scalability (how many concurrent processes at once).
  - Efficiency (should provide efficient access with minimum space wastage).

Illustration of a Process Table




## Process Abstraction in Unix

- **Process Identification, Information**




- Unix Command for process information:
  - `ps` (short for process status)

- **Process Creation, Termination, Parent-Child Synchronization**

## Note: Command Line Argument in C

- We can pass arguments to a program in C.
- `argc` : Number of CL arguments, including program name.
- `argv` : A char strings array, each element in `argv[]` is a C character string.

```
int main( int argc, char* argv[] )
{ int i;
   for (i = 0; i < argc; i++){
   printf("Arg \%i: \%s, ",i, argv[i] );
   }
   return 0;
}
```

- Example Run: "a.out 123 hello world"
- Output: "Arg 0: a.out, Arg 1: 123, Arg 2: hello, Arg 3: world"

## Process Creation: `fork()`

Process Creation in Unix: `fork()`

- The main way to create a new process

Header File
```
#include <unistd.h>
```

Syntax
```
int fork( );
```

- Returns:
  - PID of the newly created process (for parent process) OR
  - 0 (for child process)
- Header files are system dependent
  - *"man fork"* to locate the right files for your system!

Process Creation in Unix: `fork()` (cont)

- Behavior:
  - Creates a new process (known as ***child process***)
  - Child process is a **duplicate** of the current executable image
    - i.e. same code, same address space etc
    - Data in child is a COPY of the parent (ie.not shared)
  - Child differs **only in:**
    - Process id (PID)
    - Parent (PPID )
      - Parent = The process which executed the fork()
    - `fork()` return value

`fork()` : Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    printf("I am ONE\n");
    fork();
    printf("I am seeing DOUBLE\n");

    return 0;
}
```

- Question:
  - What do you think is the output?

- `fork()` : Create exact copy of the parent, including any variables.
- **Output**: Both parent and child resume execution after the point `fork()` .
- **Note** `clone()` : `fork()` not versatile, for scenarios where partial duplication preferred, `clone()` , which supersedes `fork()` .
- Both parent and child processes continue executing, common usage is to use the parent/child process differently. (Parent spawn off child to carry out some work, parent ready to take another order.)
  - Use return value of `fork()` to distinguish parent and child.

`fork()`: Parent and Child Example

```
... ...
int result;

result = fork();
if (result != 0){
    printf("P:My Id is %i\n",getpid());
    printf("P:Child Id is %i\n",result);
} else {
    printf("C:My Id is %i\n", getpid() );
    printf("C:Parent Id is %i\n", getppid() );
}
... ...
```

Parent Process | Using fork() result to differentiate parent and child | Child Process

## Process Replacement: `execl()` System Call

- Function **replaces current executing process image** with a new process image specified by path. No return is made because the calling process image is replaced by the new process image.
- **Code Replacement**, but **PID and other information still intact**.

`execl()` System Call

Header File
```
#include <unistd.h>
```

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

- `path`: Location of the executable
- `arg0`, ..., `argN`: Command Line Argument(s)
- `NULL`: To indicate end of argument list

`execl()`: Simple Example

```
int main()
{
    execl( "/bin/ls", "ls", "-l", NULL);
}
```

- Note:
  - `Path` = "`bin/ls`"
    - The "`dir`" command in unix, to list the files in directory
  - `arg0` = "`ls`"
    - The program name
  - `argl` = "`-l`"
- The above is exactly the same as executing:
  `ls -l`

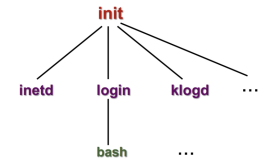
- **By combining** `fork()` **and** `exec()` , we can spawn off a child process (let it perform task through `exec()` ), while parent process around to accept another request.
- This combination of mechanisms is main way in Unix to get new process for running new program!

## The Master Process: `init`

- Every process has parent process, consider special initial process.
- `init` **process**: Created in kernel at boot up time, usually PID = 1.
- **Purpose**: Watches and respawns other (critical) processes where needed.
- `fork()` creates the process tree, where `init` is the root process.

## Simplified Process Tree Ex.

Process Tree Example (simplified)




Note: just a simple example, actual process tree varies according to Unix setup

- **d**, (e.g. `klogd` ) at end of process name usually means server process (Daemon, background process.)

## Process Termination in Unix: `exit`

Process Termination in Unix

- To end execution of process:

Header File
```
#include <stdlib.h>
```

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

- Status is returned to the parent process (more later)
- Unix Convention:
  - 0 = Normal Termination (successful execution)
  - ! 0 = To indicate problematic execution
- The function **does not return!**

Implicit `exit()`

- Most programs have no explicit exit() call
- Example:

```
int main()
{
    printf("Just to say goodbye!\n");
}
```

- Return from `main()` implicitly calls exit()
  - Open files also get flushed automatically!

- **Process finished execution**: **Most** system resources used by process are released on exit. (e.g. file descriptors).
- **Certain basic process resources not Releasable**: PID, status needed. For parent-children synchronization, for parent to check status of child, For process accounting info (e.g. cpu time).
- Process table entry may still be needed after termination.

## Parent/Child Synchronization in Unix

- Parent process can wait for child process to terminate.
- Argument is a pointer to a variable that will store the return value ( `*status` ).

Parent/Child Synchronization in Unix

Header File
```
#include <sys/types.h>
#include <sys/wait.h>
```

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

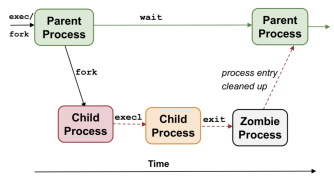
- Returns the PID of the terminated child process
- status (passed by address):
  - Stores the exit status of the terminated child process
  - Use **NULL** if you do not need/want this info

Parent/Child Synchronization in Unix

- Behavior:
  - The call is blocking:
    - Parent process blocks until at least one child terminates
  - The call cleans up ***remainder*** of child system resources
    - Those not removed on `exit()`
    - Kill zombie process ☺
- Other variants of `wait()` :
  - `waitpid()`
    - Wait for a specific child process
  - `waitid()`
    - Wait for any child process to **change status**
  - etc…

- Kills zombie processes! With enough zombies, process table finite size, run out of space.

Process Interaction in Unix




Note: example uses one ordering of execution, others are possible!

### Zombie Processes (2 Cases)

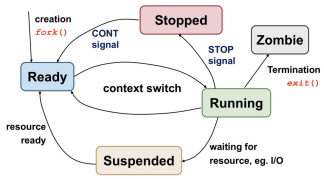
- `wait()` "creates" the zombies (and later cleans it up) as on process exit, process becomes zombie.
- Since it cannot delete all process info (if parent asks for info in `wait()` call, remainder of process data structure can be cleaned up only when `wait()` happens.)
- We cannot `kill PID` zombie process, is already dead. Until restart system or modern OS look through table and remove them.

1. **Parent process terminates before child process**
   - `init` process becomes "pseudo" parent of child processes.
   - Child termination sends signal to `init`, which utilizes `wait()` to cleanup
2. **Child process terminates before parent but parent did not call wait**
   - Child process become a zombie process
   - Can fill up / hog process table, May need a reboot to clear the table on older Unix implementations

Summary of Unix Process System calls

- `fork()`:
  - Process creation
- `exec()` family:
  - Change executing image/program
  - `execl, execv, execve, execle, execvp`
- `exit()`:
  - Process termination
- `wait()` family:
  - Get exit status, synchronize with child
  - `wait, waitpid, waitid, etc`
- `getpid()` family:
  - Get process information
  - `getpid, getppid`, etc

Process State Diagram in Unix




# Implementation Issues

## Implementing `fork()`

Implementing `fork()`

- Behavior of `fork()`:
  - Makes an almost exact copy of parent process
- Simplified implementation:
  1. Create address space of child process
  2. Allocate `p'` = new `PID`
  3. Create kernel process data structures
     - E.g. Entry in Process Table
  4. Copy kernel environment of parent process
     - E.g. Priority (for process scheduling)
  5. Initialize child process context:
     - `PID=p', PPID=parent` id, zero CPU time

Implementing `fork()` (cont)

6. Copy memory regions from parent
   - Program, Data, Stack
   - Very expensive operation that can be optimized (more later)
7. Acquires shared resources:
   - Open files, current working directory etc
8. Initialize hardware context for child process:
   - Copy registers, etc. from parent process
9. Child process is now ready to run
   - add to scheduler queue

Memory copy is very expensive:

- Potentially need to copy the whole memory space

- **Copying entire memory space is wasteful** and not always needed! (E.g. copy entire 200mb program image of Zoom etc). Mostly, only need contents, and PC, register values.
- **Give Rise to COW.** (copy on write)

## Memory Copy Operation

- If child just read from location, unchanged, just use a shared version.
- **Only when write is perform on a location, then two independent copies needed.**
- **Copy on Write** is possible optimization, only duplicate "memory location" when it is written to, otherwise parent and child share same "memory location".
- Note: memory organized into memory pages (consec range of mem locations), memory managed on page level instead of individual location.

# System Calls (Process Interaction with OS)

## API to OS: Application Program Interface to OS

- OS API provides way of calling facilities/services in kernel.
- **Not same as normal function call**: Change from *user mode to kernel mode*.
- Different OS have different APIs: Unix Variants most follow POSIX standards, small no. of calls 100. Windows family uses Win API across diff. windows, huge no. of calls 1000.

## Unix System Calls in C/C++ program

- In C/C++ program, **system call can be invoked almost directly**, as library version very closely reflects these calls.
- Majority of system calls have library version with **same name** and parameters, library version acts as **function wrapper**.
- A few library functions present more user friendly version, e.g. less no. /more flexible parameters). Library version acts as **function adapter**.

Example

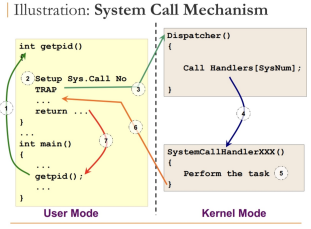
```
#include <unistd.h>
#include <stdio.h>

int main()
{
    int pid;

    /* get Process ID */
    pid = getpid();

    printf("process id = %d\n", pid);

    return 0;
}
```

Library call that has the same name as a system call

Library call that make a system call

- System Calls invoked in this example:
  - `getpid()`
  - `write()` – made by `printf()` library call

Illustration: System Call Mechanism




## General System Call Mechanism

General System Call Mechanism

1. User program invokes the library call
   - Using the normal function call mechanism as discussed
2. Library call (usually in assembly code) places the **system call number** in a designated location
   - E.g. Register
3. Library call executes a special instruction to switch from user mode to kernel mode
   - That instruction is commonly known as **TRAP**

General System Call Mechanism (cont)

4. Now in kernel mode, the appropriate system call handler is determined:
   - Using the system call number as index
   - This step is usually handled by a **dispatcher**
5. System call handler is executed:
   - Carry out the actual request
6. System call handler ended:
   - Control return to the library call
   - Switch from kernel mode to user mode
7. Library call return to the user program:
   - via normal function return mechanism

# Exception and Interrupt

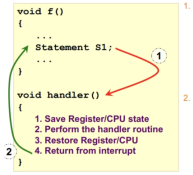
## Exception

- **Executing machine level instruction** can cause **exception**. For example:
  - Airthmetic Errors: Overflow, Underflow, Division by Zero
  - Memory Accessing Errors: (accessing memory not belonging to program), Illegal memory address, mis-aligned memory access.
- Exception is **Synchronous**: Determinate, occurs due to program execution at exact points of time.
- **Effect of Exception**: Have to execute **exception handler**, which is similar to a **"forced function call"!**
- Exception ≠ Interrupt!

## Interrupt

- **External events** can interrupt the execution of a program.
- Interrupt request lines connected to CPU, lines are checked during instruction execution cycle.
- Usually hardware related, e.g.: Timer, Mouse move, Keyboard press etc
- Interrupt is **asynchronous**: Events occurs independent of program execution.
- **Effect of interrupt:** Program execution, suspended, execute an interrupt handler.

Exception/Interrupt Handler: Illustration




1. Exception/Interrupt occurs:
   - Control transfer to a handler routine **automatically**
2. Return from handler routine:
   - Program execution resume
   - **May** behave as if nothing happened

## Summary

- Using **process as an abstraction** of running program.
- Includes necessary information (environment) of execution, Memory, Hardware and OS contexts.
- **Process from OS perspective**: PCB and process table
- **How OS & Process interact**: System calls, Exception / Interrupt

## REFER TO TEXTBOOK:

- **Modern Operating System (3rd Edition)**
  - Section 2.1: Processes
  - Section 2.4: Process Scheduling
  - Section 2.2: Threads
- **Operating System Concepts (8th Edition)**
- Section 3.1

# 3. Process Scheduling

A multiprogrammed computer frequently has multiple processes/threads computing for CPU at the same time. Occurs whenever $\geq 2$ simultaneously in ready state.

- **Scheduler**: Part of OS to decide which process to run next.
- **Scheduling Algorithm**: Algo used.
- **Scheduling Problem**: Choosing, ready process $>$ available CPUs.
- In addition to picking right process, need **efficient use of CPU** as process switching is expensive. (Switch user to kernel mode, save state of process, memory map, memory cache may need to reload, etc.)
- **I/O Input/Output** (disk or network): When process enters blocked state waiting for external device to complete work.

## Process Behavior

- Process' unique **requirement of CPU time.**
- Process goes through phases of CPU-activity & IO-activity.
- **Compute/CPU-Bound Process** (computation, e.g. number crunching) vs. **IO-bound Process** (e.g. read/write to file, print to screen)

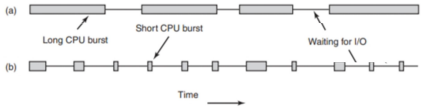



**Figure 2-39.** Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

## Process Environment

- **Batch Processing**: No user, no interaction / responsiveness required.
- **Interactive / Multiprogramming**: Active user interacting with system. Need responsive, consistent in response time.
- **Real Time Processing**: Deadline to meet, usually periodic process.

## Scheduler Evaluation Criteria

- Many criteria to evaluating algo, largely influenced by p. environment. May be conflicting.
- **All Systems**:
  - *Fairness*: CPU time (per process basis / per user basis). **No starvation**.
  - *Balance*: All parts of computing system utilized.
- **Batch Systems**:
  - *Throughput*: Maximize jobs per hour.
  - *Turnaround time*: Minimize time btwn. submission & termination.
  - *++(Waiting Time)*: Related to turnaround, time waiting for CPU
  - *CPU utilization*: keep CPU busy all the time.
- **Interactive Systems**:
  - *Response time*: respond to requests quickly.
  - *Proportionality*: meet users' expectations.
- **Real-time Systems**:
  - *Meeting deadlines*: avoid losing data. (e.g. livestream)
  - *Predictability*: avoid quality degradation in multimedia.

## Concurrent Execution

- **Concurrent Processes**: Logical concept to cover multitasked processes.
- **Virtual parallelism**: Illusion of parallelism (pseudo-parallelism)
- **Physical parallelism**: Multiple CPUs/Cores, multiple parallel exec

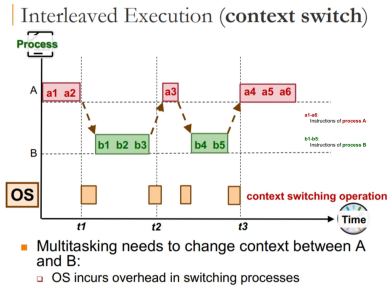
## When to Schedule

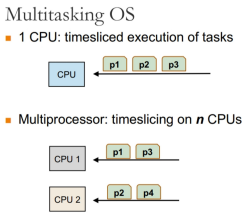
Key issue, when to make schedule decisions.
E.g. When new process created, run parent or child. When process exits, when process blocks on I/O, or I/O interrupt.

- **Non-preemptive (cooperative)**: Process stays scheduled (running state) until it blocks, or gives up CPU voluntarily.
- **Preemptive (Fixed Quota)**: Process given fixed time quota to run (possible to block / give up early). At end of quota, running process suspended, another picked if available.

## Interleaved Execution (Timeslicing)

- **Concurrent Execution on 1 CPU**: Interleave instructions from both processes.
- **OS overhead**: Multitasking needs to change context between programs, incurs overhead.




- **Scheduling a Process**:
  1. Scheduler triggered (OS takes over)
  2. If Context Switch needed, save context, place on blocked/ready queue.
  3. Pick suitable process $P$ to run base on scheduling algo.
  4. Setup context for $P$.
  5. Let process $P$ run.

# Scheduling in Batch Systems

**Environment**: No user interaction, non-preemptive scheduling predominant.
Scheduling algoritms generally easier to understand and implement, with variants and improvements for other type of system.
**Criteria**: Turnaround time (related to waiting time, time spent waiting for CPU). Throughput. CPU utilization.

## Batch Systems Scheduling Algorithms

## FCFS: First-Come First-Served

- Tasks stored on **FIFO queue based on arrival time**.
- Pick first task in queue to run until done / blocked. Blocked task removed from FIFO queue, when ready, place at back of queue ("newly arrived").
- **Evaluation**:
  - *No starvation*. (Every task eventually processed)
  - *Covoy Effect*. (CPU-Bound followed by IO-Bound tasks heavily inefficient.) Simple reordering can reduce average waiting time.

## SJF: Shortest Job First (Nonpreemptive)

- Select task with **smallest total CPU time**.
- Need to know total CPU time for task in advance.
  - Possible to guess future CPU time by previous CPU-bound phases.

Common approach (Exponential Average):

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

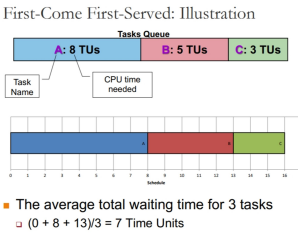
- $Actual_n$ = The most recent CPU time consumed
- $Predicted_n$ = The past history of CPU Time consumed
- $\alpha$ = Weight placed on recent event or past history
- $Predicted_{n+1}$ = Latest prediction

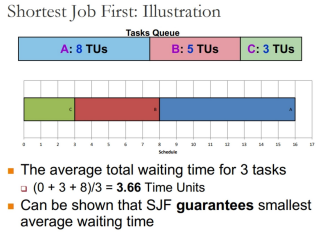
- **Evaluation**:
  - *Starvation Possible* (Biased towards short jobs) - *Minimize average waiting time*. - Optimal only when all jobs available simultaneously.

## SRT: Shortest Remaining Time Next (Preemptive)

- **Preemptive ver of SJF**.
  - Scheduler chooses process whose remaining run time is shortest.
- When new job arrives, total time compared to current process' remaining (or expected) time.

## Batch System Scheduling Examples

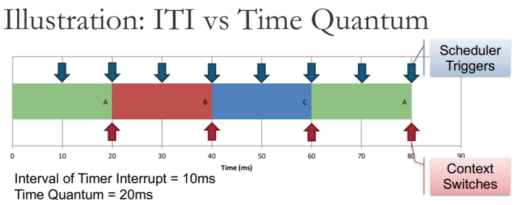


# Scheduling in Interactive Systems

**Criteria**: Response time (time btwn request & response). Predictability (Less variation in response time).
**Environment**: User interaction. Preemptive scheduling used to ensure good response time. Scheduler needs to run periodically.
**Ensuring Periodic Scheduler**: Use timer **interrupts** (based on hardware clock). OS ensures timer interrupt cannot be intercepted by any other program. Interrupt handle **invokes scheduler**.

- **ITI: Interval of Timer Interrupt**: Timing Interval that interrupt happens, and OS scheduler triggered. Typical values (1ms - 10ms).
- **Time Quantum**: Execution duration given to each process, constant / variable. Must be multiples of ITI. Typical values (5ms - 100ms).




## Interactive Systems Scheduling Algorithms

## RR: Round Robin

- Preemptive ver. of FCFS.
- **Tasks stored in FIFO queue**. Each process assigned time quantum.
- If task still running at end of quantum / blocked / gives up CPU voluntarily, CPU preempted, given to another process.
- Task placed at end of queue to wait for another turn.
- **Response time guarantee**: $n$ tasks, $q$ quantum.
  Time before task gets CPU bounded by $(n-1)q$.
- **Evaluation**:
  - Too short quantum = many process switches, lower CPU efficiency
  - Too long quantum causes poor response to short interactive requests.

## Priority Scheduling: Priority Based

- Each process assigned a priority, runnable process with highest priority p allowed to run.
  - **Preemptive ver.**: Higher p. process can preempt running low p. process.
  - **Non-preemptive ver.**: Late high p. process wait for next scheduling.
- **Evaluation**:
  - **Possible Starvation**: High p. process may hog CPU.
  To prevent this, scheduler may **decrease priority** of currently running process at each clock tick (clock interrupt). Or, give some max time quantum, when used up, allow next in line to run.
  - **Priority Inversion**: When lower priority task preempts higher priority task. (If lower priority task locks some resource, e.g. file, gets switched, higher priority task cannot run)

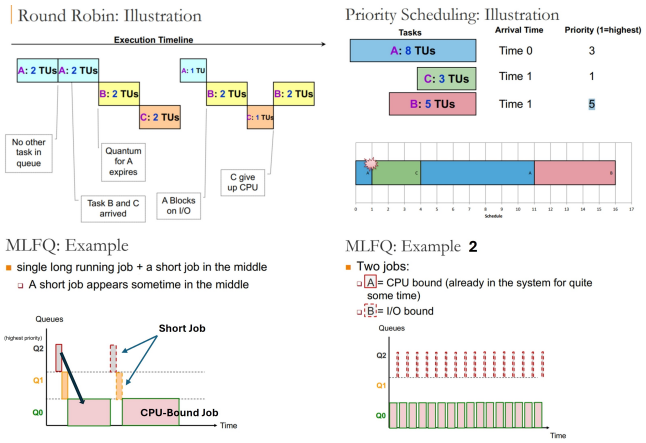
## MLFQ: Multi-Level Feedback Queue

- More efficient to give CPU-bound process large quantum once in a while, rather than small quanta frequently to reduce swapping. Also, giving all processes large quantum means poor response time.
  Hence, solution to set **multiple priority queues**.
- As (CPU-bound) process sinks deeper into priority queues, run less frequently, saving CPU for short, interactive processes.
- Hence, **adaptively learn process behavior**, minimize both:
  - Min. Response time for IO bound processes
  - Min. Turnaround time for CPU bound processes.
- **MLFQ Rules**:
  - **Basic Rule**: Higher priority process runs. If same p, run in RR.
  - **Priority Setting**: New job given highest priority. If job fully utilize time slice, priority reduced. If job gives up / blocks before time slice finished, priority retained.
- **Evaluation**: - **Can be gamed**: User typing carriage returns at random every few seconds doing wonders for his response time.

## Lottery Scheduling

- Give processes lottery tickets for system resources. When scheduling, chose ticket at random.
- "All processes equal, some processes more equal." More important processes given extra tickets.
- **Evaluation**:
  - *Responsive*: New process can participate in next lottery.
  - *Good Control*: Each process can distribute to child processes proportionally w.r.t to need.

## Interactive System Scheduling Examples




## Others

- **Shortest Process Next**: Estimate (using calculated aging).
- **Guaranteed / Fair-Share Scheduling**: Track CPU usage, run accordingly. Each user gets agreed allocation of CPU.

# 4. Threads

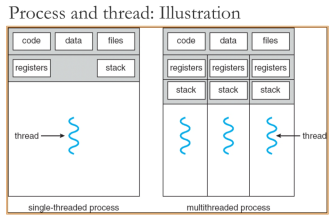
In trad. OS, each process has an address space and single thread of control. Desirable instead to have multiple threads of control in the same address space running in quasi-prallel, as though (almost) separate processes (except for shared address space).

## Motivation for Thread

- **Process is Expensive**: Process creation under `fork()` model duplicate memory space, most of process context.
- Context switch also requires overhead, save/restore process info.
- **Communication**: Hard for independent processes to communicate with each other. Independent memory space, no shared variable, requires IPC.
- **Thread**: "quick hack" into popular mechanism.
- **Basic Idea**: Traditional process only single thread of control. (Only one instruction executing at a time).
- Add more threads (of control) to same program, multiply parts of programs executing at same time conceptually.

## Process and Thread

- **Multi-threaded Process**: Single process can have multiple threads.
- Threads in same process share same: *Memory Context* (text, data heap), *OS Context* (Process id, files etc.)
- **Unique info per thread**: Id (thread id), Registers (GPR and special), "Stack".
- **Process Context Switch vs. Thread Switch**:
  - *Process Context Switch*: Switch OS, Hardware, Memory Context.
  - *Thread Switch (same process)*: Just hardware context (registers, stack).
  - Thread is **lightweight process**.




## Threads: Benefits

- **Resource Sharing**: Ability for parallel entities to share address space and all of data. No need add. mechanism for passing info.
- **Economy**: Threads are lighter weight than process, faster to create and destroy. Less resources to manage.
- **Responsiveness**: No performance gain when all CPU bound, but when substantial computing and substantial I/O, threads allows activities to overlap, speeding up application.
- **Scalability**: Multithreaded program can take advantage of multiple CPUs.

## Threads: Problems

- **System Call Concurrency**: Parallel execution of multiple threads: parallel system call possible. Need to guarantee correctness and determine correct behavior.
- **Process Behavior (OS dependent)**: Impact on process operations, (e.g. If one thread executes exec() / exit(), what about other threads / whole process).

## Thread Models (ways to support threads)

- **User Thread**: Thread implemented as a **user library**. (Runtime system (in the process) will handle thread related operation.)
- Kernel not aware of threads in process.
- **Kernel Thread**: Thread implemented in the OS. Operation handled as system calls.
- Kernel thread-level scheduling possible, where kernel schedule by threads instead of by process.
- Kernel may make use of threads for own execution.
- **Threads on Modern Processor**: Threads started as software mechanism, now exists hardware support on modern processors (multiple register sets on same core, *simultaneous multi-threading (SMT)*, "Hyperthreading" on Intel).

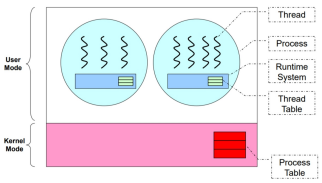
## User Thread

- **Advantages**: Can have any multithreaded program on any OS, thread operations are just library calls, generally more configurable and flexible (e.g. customized thread scheduling policy.)
- **Disadvantages**: OS not aware of threads, scheduling is performed at process level. (One thread blocked, process blocked, all threads blocked). Cannot exploit multiple CPUs.

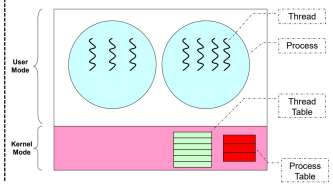
## Kernel Thread

- **Advantages**: Kernel can schedule on thread levels: $>1$ thread in same process can run simultaneouly on multiple CPUs.
- **Disadvantages**: Thread operations now system calls, slower and more resource intensive. Generally less flexible, used by all multithreaded programs, so too many / few features, overkill / not flexible enough for different programs.

User Thread: Illustration



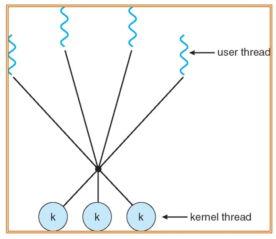

Kernel Thread: Illustration



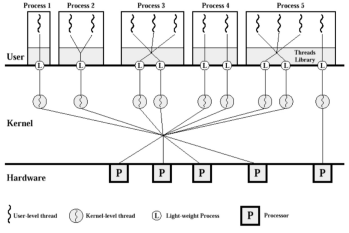

## Hybrid Thread Model

- Have both Kernel and User threads, where OS schedule on kernel threads only, and user thread binds to a kernel thread.
- Offers flexibility, limit concurrency of any process / user.

Hybrid Thread Model




Hybrid Model Example: **Solaris**




# POSIX Threads

IEEE defined standards (1003.1c) for threads, called *Pthreads*, most UNIX systems support it. Defines over 60 function calls.

## Pthread

- **POSIX**: Portable Operating System Interface. (family of standards)
- Defines the API as well as the behavior, but not implementation.
- pthread can be implemented as user / kernel thread.

Basics of pthread

- Header File:
  - `#include <pthread.h>`
- Compilation (flag is system dependent):
  - `gcc xxxx.c -lpthread`
- Useful datatypes:
  - `pthread_t` : Data type to represent a thread id (TID)
  - `pthread_attr`: Data type to represents attributes of a thread

pthread Creation Syntax

```
int pthread_create(
        pthread_t* tidCreated,
        const pthread_attr_t* threadAttributes,
        void* (*startRoutine) (void*),
        void* argForStartRoutine  );
```

- Returns (0 = success; !0 = errors)
- Parameters:
  - `tidCreated`: Thread Id for the created thread
  - `threadAttributes`: Control the behavior of the new thread
  - `startRoutine`: Function pointer to the function to be executed by thread
  - `argForStartRoutine`: Arguments for the `startRoutine` function

pthread Termination Syntax

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

- Parameters:
  - `exitValue`: Value to be returned to whoever synchronize with this thread (more later)
- If `pthread_exit()` is not used, a *pthread* will terminate automatically at the end of the `startRoutine`
  - If a "return XYZ;" statement is used, then "XYZ" is captured as the exitValue
  - Otherwise, the exitValue is not well defined

pthread Creation & Termination: Example

```
//header files not shown

void* sayHello( void* arg )          Function to be executed by a pthread
{
    printf("Just to say hello!\n");
    pthread_exit( NULL );            Pthread Termination
}

int main()
{
    pthread_t tid;                   Pthread Creation

    pthread_create( &tid, NULL, sayHello, NULL );
    printf("Thread created with tid %i\n", tid);

    return 0;
}
```

pthread: Sharing of memory space

```
//header files not shown
int globalVar;          Variable shared between pthreads

void* doSum( void* arg)
{
    int i;
    for (i = 0; i < 1000; i++)
        globalVar++;                 Using a shared variables
}

int main()
    pthread_t tid[5];    //5 threads id
    int i;
    for (i = 0; i < 5; i++)
        pthread_create( &tid[i], NULL, doSum, NULL );
    printf("Global variable is %i\n", globalVar);
    return 0;
}
```

Bc. **Race Condition**, Sum unpredictable

```
#include <stdio.h>
#include <pthread.h>
int globalVar;

void* doSum( void* arg){
    int i;
    for (i = 0; i < 1000; i++) {
        globalVar++; }
}


int main() {
    pthread_t tid[5]; //5 threads id
    int i;

    for (i = 0; i < 5; i++) {
        pthread_create( &tid[i], NULL, doSum, NULL ); }

    //Wait for all threads to finish
    for (i = 0; i < 5; i++) {
        pthread_join( tid[i], NULL ); }

    printf("Global variable is %i\n", globalVar);
    return 0;
}
```

pthread Simple Synchronization - Join

```
int pthread_join( pthread_t threadID,
                  void **status );
```

- To wait for the termination of another ***pthread***:
- Returns(0 = success; !0 = errors)
- Parameters:
  - `threadID`: `TID` of the ***pthread*** to wait for
  - `status`: Exit value returned by the target ***pthread***

Pthread: A lot more!

- There are more interesting stuff about ***pthread***:
  - Yielding (giving up CPU voluntarily)
  - Advanced synchronization
  - Scheduling policies
  - Binding to kernel threads
  - Etc
- As we cover new topics, you can explore the ***pthread*** library to see the application!

## Summary of Pthread

- **All Pthread threads have certain properties**: each one has identifier, set of registers (including program counter), set of attributes which are stored in a structure.
- Attributes include stack size, scheduling params etc.
- *pthread_create call*: New thread created, returns thread identifier of newly created thread as function value.
- *pthread_exit*: Thread terminate by calling, stops thread and releases stack.
- *pthread_join*: Thread needing to wait for another thread to finish work and exit before continuing can call to wait for specific other thread to terminate. (TID) of thread to wait for given as parameter.

| Thread call | Description |
|---|---|
| Pthread_create | Create a new thread |
| Pthread_exit | Terminate the calling thread |
| Pthread_join | Wait for a specific thread to exit |
| Pthread_yield | Release the CPU to let another thread run |
| Pthread_attr_init | Create and initialize a thread's attribute structure |
| Pthread_attr_destroy | Remove a thread's attribute structure |

**Figure 2-14.** Some of the Pthreads function calls.

# 5. Inter-Process Communication (IPC)

Given that processes frequently need to communicate, some need for communication, preferably in some well-structured way not using interrupts. Consider that cooperating processes have independent memory space, making IPC necessary.

## Common Communication Mechanisms

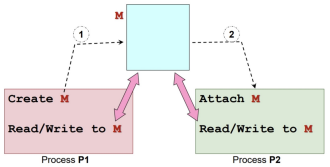
## Shared Memory

- **General Idea**: Process 1 creates shared memory region $M$, process 2 attaches $M$ to its own memory space. Both processes can now communicate using memory region $M$. Applicable to multiple processes.
- $M$ behaves similar to normal memory region, any writes to region can be seen by all other parties.

POSIX Shared Memory in *nix

- Basic steps of usage:
  1. Create/locate a shared memory region **M**
  2. Attach **M** to process memory space
  3. Read from/Write to **M**
     - Values written visible to all process that share **M**
  4. Detach **M** from memory space after use
  5. Destroy **M**
     - Only one process need to do this
     - Can only destroy if **M** is not attached to any process

Shared-Memory: Illustration




- OS is involved in step 1 and 2 only

- **Advantages**:
  - Efficient, as only initial steps of create & attach $M$ involves OS.
  - Ease of use, as $M$ behaves as normal memory space, info of any size/type writable easily.
- **Disadvantages**:
  - Synchronization: Shared resource, need to sync access still.
  - Implementation is harder.

Example: Master program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    int shmid, i, *shm;

    shmid = shmget( IPC_PRIVATE, 40, IPC_CREAT | 0600);

    if (shmid == -1){
        printf("Cannot create shared memory!\n");
        exit(1);
    } else
        printf("Shared Memory Id = %d\n", shmid);

    shm = (int*) shmat( shmid, NULL, 0 );
    if (shm == (int*) -1){
        printf("Cannot attach shared memory!\n");
        exit(1);
    }

    shm[0] = 0;

    while(shm[0] == 0){
        sleep(3);
    }

    for (i = 0; i < 3; i++){
        printf("Read %d from shared memory.\n", shm[i+1]);
    }

    shmdt( (char*) shm);
    shmctl( shmid, IPC_RMID, 0);

    return 0;
}
```

The master program create the shared memory region and wait for the "slave" program to produce values before proceeding.

Step 1. Create Shared Memory region.

Step 2. Attach Shared Memory region.

The first element in the shared memory region is used as "control" value in this example (0: values not ready, 1: values ready).

The next 3 elements are values produced by the slave program.

Step 4+5. Detach and destroy Shared Memory region.

Example: Slave program

```
//similar header files
int main()
{   int shmid, i, input, *shm;

    printf("Shared memory id for attachment: ");
    scanf("%d", &shmid);

    shm = (int*)shmat( shmid, NULL, 0);
    if (shm == (int*)-1) {
        printf("Error: Cannot attach!\n");
        exit(1);
    }

    for (i = 0; i < 3; i++){
        scanf("%d", &input);
        shm[i+1] = input;
    }
    shm[0] = 1;

    shmdt( (char*)shm );
    return 0;
}
```

Step 1. By using the shared memory region id directly, we skip ***shmget***() in this case.

Step 2. Attach to shared memory region.

Write 3 values into shm[1 to 3]

Let master program know we are done!

Step 4. Detach Shared Memory region.

## Message Passing

- **General Idea**: Process 1 prepares, send message $M$. Process 2 receives it.
- Message sending / receiving usually provided as system calls.
- **Additional properties**: Naming (identify other party), synchronization (behavior of send/rec ops).
- **Msg** has to be stored in kernel memory space. Each send/rec op needs to go through OS (system call).

### Direct Communication (Naming Scheme)

- Sender/Receiver of message **explicitly names other party**.
- E.g. `Send(P2, Msg)` & `Receive(P1, Msg)`.
- One link per pair of process, need know identity of other party.

### Indirect Communication (Naming Scheme)

- Messages sent / recieved from message storage, **aka mailbox / port**.
- E.g. `Send(MB, Msg)` & `Receive(MB, Msg)`.
- One mailbox can be shared among a number of processes

### Synchronization Behaviors

- **Blocking Primitives (Synchronous)**: Sender blocked until message received, receiver blocked until message has arrived.
- **Non-Block Primitives (Asynchronous)**: Sender resume operation immediately, receiver either receive, or indicate message not ready yet.

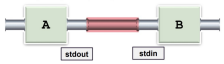
- **Advantages**:
  - Portable, easily implemented on diff. processing env.
  - Easier synchronization: Esp. when synchronous primitive used.
- **Disadvantages**:
  - Inefficient, require OS intervention
  - Harder to use, less flexi, messages limited in size / format.

## Pipe (Unix Specific)

- Unix process has 3 default communication channels:
  - **stdin** (standard in): Commonly linked to keyboard input.
  - **stderr** (standard error): Only used print out error msg.
  - **stdout** (standard out): Commonly linked to screen.

Piping in Shell

- For example ( " A | B" ):



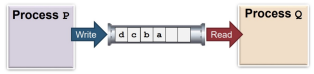

- The output of A (instead of going to screen) directly goes into B as input (as if it come from keyboard)

- General Idea:
  - A communication channel is created with 2 ends:
    - 1 end for reading, the other for writing
    - Just like a water pipe in the real world


- The piping "|" in shell is achieved using this mechanism internally

- **Unix Shell Piping**: Unix shell provides | symbol to link input/output channels of one process to another, aka *piping*.

Unix Pipes: as a IPC Mechanism




- A pipe can be shared between two processes
- A form of Producer-Consumer relationship
  - P produces (writes) **n** bytes
  - Q consumes (reads) **m** bytes
- Behavior:
  - Like an anonymous file
  - FIFO → must access data in order

Unix Pipes: Semantic

- Pipe functions as **circular bounded byte buffer** with **implicit synchronization**:
  - Writers **wait** when buffer is **full**
  - Readers **wait** when buffer is **empty**

- Variants:
  - Can have multiple readers/writers
    - The normal shell pipe has 1 writer and 1 reader
  - Depends on Unix version, pipes may be **half-duplex**
    - **unidirectional**: with one write end and one read end
  - Or **full-duplex**
    - **bidirectional**: any end for read/write

## Unix Pipe: System Calls

| Header File | Syntax |
|---|---|
| `#include <unistd.h>` | `int pipe( int fd[] )` |

- Returns:
  - 0 to indicate success; !0 for errors
  - An array of file descriptors is returned:
    - fd[0] == reading end
    - fd[1] == writing end


Unix Pipes: Example Code

```
#define READ_END 0
#define WRITE_END 1

int main()
{
   int pipeFd[2], pid, len;
   char buf[100], *str = "Hello There!";

   pipe( pipeFd );
   if ((pid = fork()) > 0) { /* parent */
     close(pipeFd[READ_END]);
     write(pipeFd[WRITE_END], str, strlen(str)+1);
     close(pipeFd[WRITE_END]);
   } else {                 /* child */
     close(pipeFd[WRITE_END]);
     len = read(pipeFd[READ_END], buf, sizeof buf);
     printf("Proc %d read: %s\n", pid, buf);
     close(pipeFd[READ_END]);
   }
}
```

## Signal (Unix Specific)

- Form of IPC. **Asynchronous notification** regarding an event. Sent to process / thread.
- Recipient of signal must handle signal by: Default set of handlers or user supplied handler.
- **Common UNIX signals**: Kill, Stop, Continue, Arithm. error etc.

Example: Custom Signal Handler

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

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;

    return 0;
}
```

User defined function to handle signal. In this example, we handle the "SIGSEGV" signal, i.e. the memory segmentation fault signal.

Register our own code to replace the default handler.

This statement will cause a segmentation fault.

# 6. Synchronization

**Problems with Concurrent Execution**: When process execute in interleaving fashion AND share a modifiable resource, can cause **synchronization problems**.

- Single sequential process execution deterministic, concurrent processes execution may be non-deterministic.
- **Race Condition**: When final result depends on who runs precisely when.

## Critical Regions / Sections

- **Critical Section**: Part of the program where shared memory is accessed (with race condition).
- Incorrect execution is due to **unsynchronized access to shared modifiable resource**.
- If no two processes in their CS at same time, we can avoid races.
- **4 Conditions of good Critical Section / Region (CS) Implementation:**
  1. **Mutual Exclusion**: No two processes simultaneously inside their CS.
  2. **Progress**: If no process in CS, one waiting process granted access.
  3. **Bounded Wait**: No process waits forever to enter its CS.
  4. **Independence**: No process outside CS may block any process.
- **Symptoms of Incorrect Synchronization:**
  - **Deadlock**: All processes blocked, no progress.
  - **Livelock**: Process keep changing state to avoid deadlock, make no progress (dl avoidance mechanism), typically process not blocked.
  - **Starvation**: Some processes blocked forever.

## Critical Section Implementations

- **Assembly Level Implementations:** Mechanism provided by processor.
- **High Level Language Implementations:** Utilizes only normal programming constructs. (E.g. CS lock using normal variables)
- **High Level Abstraction:** Abstracted mechanisms that provide additional useful features (E.g. abstract data types, provided as library calls, and involve system calls).

# 6.1 Assembly Level Implementation

## Test and Set (TSL) Instruction

Common machine instruction to aid synchronization: TSL RX,LOCK. (Test and Set Lock). `TestAndSet Register , MemoryLocation`

- **Behavior**: Load current content at **MemoryLocation** into **Register**, and stores a 1 into **MemoryLocation**
- **Atomic**: Performed as a single machine operation, indivisible.
- Assume equivalent high level lanugage ver of **TSL**:

**Using Test and Set**

TestAndSet() takes a memory address M:
- Returns the current content at M
- Set content of M to 1

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

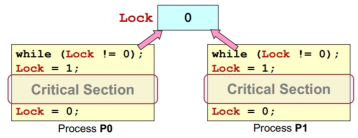
```
void ExitCS( int* Lock )
{
   *Lock = 0;
}
```

- **How Implementation works**: Before entering CS, a process calls enterCS, busy wait until lock is free; then it acquires the lock and returns. After leaving CS process calls ExitCS, which stores a 0 in lock.
- **Inefficient**: Employs busy waiting, wasteful use of processing power.
- **Variants of instruction**: `CompareandExchange` , `AtomicSwap` , `Load Link` .

# 6.2 High Level Language Implementation

## Bad: Lock Variables

Have a single, shared (lock) variable, initially 0. When entering CS, test lock and sets to 1.Thus, 0 means no process in CS, 1 means some process in CS.

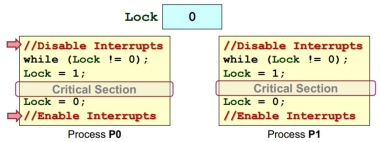



**doesn't work properly ☹**

Suppose that one process reads the lock and sees that it is 0.
Before it can set the lock to 1, another process is scheduled, runs, and sets lock to 1. When the first process runs again, it will also set lock to 1.
**Two processes will be in their critical regions at the same time.**

## Bad: Lock Variables with Interrupts Disabled

Solve the problem by preventing context switch, by disabling interrupts.

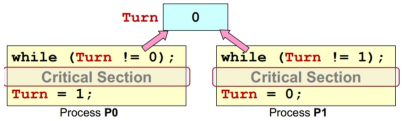



- Buggy critical section may stall the WHOLE system
- Busy waiting
- Requires permission to disable/enable interrupts

**Unwise to give user processes power to turn off interrupts.**
**If system is multiprocessor, disables only one CPU, others will continue to run and access shared memory.**

## Bad: Strict Alternation

The integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.

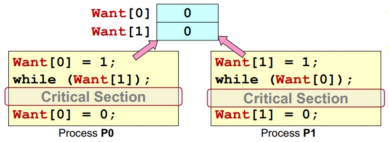



- **Assumption:**
  - P0 and P1 executes the above in loop
  - Take turn to enter **critical section**
- **Problems:**
  - Starvation:
    - E.g. If P0 never enters CS, P1 starve
  - Violate the **independence** property!

**Independence property violated**: After P0 enters CS, sets turn to 1, and wants to enter CS again, needs to wait for P1 to enter its CS to set turn to 0. If P1 does not enter CS, turn stuck at 1, P0 starves.

## Bad: Sub-Peterson Algorithm (without turn)

Leading up to peterson's algorithm. Use global shared array Want, if process wishes to enter CS, set Want[n] to 1.



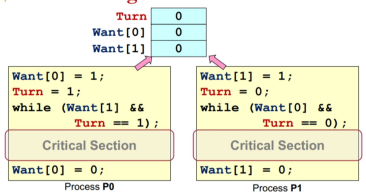

- Solve the independence problem
  - If P0 or P1 is not around, another process can still enter CS
- Problem:
  - Deadlock! Try identify the execution sequence that causes deadlock

If P0 sets Want[0] = 1, preempted, P1 sets Want[1] = 1, both will get stuck and deadlock situation occurs.

## Good: Peterson's Algorithm

Assume that writing to Turn is an **atomic operation**.

**Peterson's Algorithm**




- Busy Waiting:
  - The waiting process repeatedly test the while-loop condition instead of going into blocked state:
- Low level:
  - Higher-level programming construct is desirable
    - simplify mutual exclusion
    - less error prone
- Not general:
  - General synchronization mechanism is desirable
    - Not just mutual exclusion

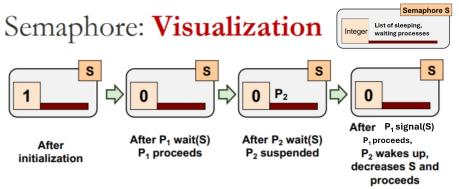
Process can store their process number, or the other process number in turn. Consider both processes trying to enter CS almost simultaneously. Both store in turn variable. Whichever store done last is the one that counts; first one overwritten and lost. After, only one can enter CS, the other will loop.

# 6.3 High Level Abstraction Implementation

## Semaphore

### Understanding

- E.W. Dijkstra proposed using semaphore to count the number of wakeups saved for future use. (Processes sleep while waiting for shared resource).
- **Atomic**: Once a semaphore operation started, no other process can access semaphore until operation has completed or blocked.




**Semaphore**: A generalized synchronization mechanism that specify behavior and not implementation. Provides a way to block a number of processes, which will then be known as **sleeping processes**.

- Semaphore S seen as a "protected integer", with non-negative initial value.
- A general semaphore (aka counting semaphore) can have values $S \geq 0$. A binary semaphore or mutex has values $S = 0$ or 1.

### Wait(S)

Takes in a semaphore. If semaphore value is $\leq 0$, blocks current process. Decrements the value when it proceeds. Atomic operation. Aka **down()**.

### Signal(S)

Takes in a semaphore, wakes up one sleeping process (if any) and increments the semaphore value. This is an atomic operation and never blocks. Aka **up()**.

### Semaphore Invariant

- $S_{current} \geq 0$
- $S_{current} = S_{initial} + \#signal(S) - \#wait(S)$
- $\#signal(S)$ is number of $signal()$ executed, $\#wait(S)$ is number of $wait()$ operations completed.

### Mutex (Binary Semaphore Usage)

- Set binary semaphore $s = 1$, for any process, do **wait(s)** before entering CS, and **signal(s)** after finishing CS.
- S can only be 0 or 1, deduced by semaphore invariant.
- This usage known as **mutex** (Mutual Exclusion)

Mutex: Correct CS - Informal Proof

- Mutual Exclusion:
  - $N_{CS}$ = Number of process in critical section
    = Process that completed `wait()` but not `signal()`
    = `#Wait( S ) - #Signal( S )`
  - $S_{initial} = 1$
  - $S_{current} = 1 + \#Signal(S) - \#Wait(S)$
  - $S_{current} + N_{CS} = 1$
  - Since $S_{current} \geq 0$ → $N_{CS} \leq 1$
- **Deadlock:**
  - Deadlock means **all** processes stuck at `wait( S )`
    → $S_{current} = 0$ and $N_{CS} = 0$
  - But $S_{current} + N_{CS} = 1$
  - →← ( contradiction)
- **Starvation:**
  - Suppose `P1` is blocked at `wait( S )`
  - `P2` is in CS, exits CS with `signal( S )`
    - If no other process sleeping, `P1` wakes up
    - If there are other process, `P1` eventually wakes up (assuming fair scheduling)

- **Possible Deadlock**: Note incorrect usage of semaphore may still result in deadlock, e.g. incorrect interleave usage of semaphore.
- **Conditional Variable**: Similar to semaphore, but allows process to wait for some event to happen. Once event happens, broadcast made to wake up all waiting tasks.

# 6.4 Classical Synchronization Problems

## Producer-Consumer Problem

Processes share a bounded buffer of size K. Producers produce items to insert in buffer, only when the buffer is not full. Consumers remove items from buffer, only when the buffer is not empty. *How to sync the two?*

### Busy Waiting Solution (Inefficient)

Producer Consumer: **Busy Waiting**

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

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

- **canConsume**:
  - Triggers consumer to *try* to get item
- **canProduce**:
  - Triggers producer to *try* to produce item
- `wait(mutex) + signal(mutex)` : Creates a CS
- `in = (in+1) % K` :
  `out = (out+1)% K` : Wraps around, circular array
- Initial Values:
  - `count = in = out = 0`
  - `mutex = S(1)` //semaphore with initial value 1
  - `canProduce = TRUE` and `canConsume = FALSE`;
- Evaluation:
  - The code **correctly solve** the problem
  - However, **busy-waiting** is used

### Blocking Solution

Solution uses three semaphores, one called (*notFull*) for counting empty slots, one called (*notEmpty*) for counting full slots, and one called *mutex* to make sure producer and consumer do not access the buffer at the same time.

- *NotFull* is initially equal to the number of slots in the buffer, *NotEmpty* is initially 0, and *mutex* is initially 1.
- When producing item, wait(notFull), which decrements. Then, acquire mutex. After producing, signal (release) mutex, and signal(notEmpty), causing it to increment.

Producer Consumer: **Blocking Version**

```
while (TRUE) {
    Produce Item;

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

```
while (TRUE) {

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

    Consume Item;
}
```
Consumer Process

- `wait( notFull )` : Forces producers to go to sleep
- `wait( notEmpty )` : Forces consumers to go to sleep
- `signal( notFull )` : 1 consumer wakes up 1 producer
- `signal( notEmpty )` : 1 producer wakes up 1 consumer
- Initial Values:
  - `count = in = out = 0`
  - `mutex = S(1), notFull = S(K), notEmpty = S(0)`
- Evaluation:
  - This code correctly solve the problem
  - No busy-waiting, "unwanted" producer/consumer will go to sleep on respective semaphores

## Readers and Writers Problem

Processes share a data structure $D$, where readers can access and read information from D together, while writers must have exclusive access to $D$ to write information. *How to sync the two?*

Readers Writers: **Simple Version**

```
while (TRUE) {

    wait( roomEmpty );

    Modifies data

    signal( roomEmpty );
}
```
Writer Process

```
while (TRUE) {

    wait( mutex );
    nReader++;
    if (nReader == 1)
        wait( roomEmpty );
    signal( mutex );

    Reads data

    wait( mutex );
    nReader--;
    if (nReader == 0)
        signal( roomEmpty );
    signal( mutex );
}
```
Reader Process

- Initial Values:
  - `roomEmpty = S(1)`
  - `mutex = S(1)`
  - `nReader = 0`

In this solution, the first reader to get access to the database does a wait (down) on semaphore **roomEmpty**.

Subsequent readers merely increment counter, **nReaders**.

As readers leave, they decrement **nReaders**, and the last to leave does a signal (up) on **roomEmpty**, allowing a blocked writer, if there is one, to get in.

- **Problem**: As long as at least one reader active, subsequent readers admitted. Writer kept suspended until no reader is present. Writer may never get in.
- **Rectification**: Program could be written slightly differently: when reader arrives and a writer is waiting, the reader suspended behind writer instead of being admitted immediately. Writer waits for readers active to finish, but not readers after. Disadvantage is less concurrency and thus lower performance.

## Dining Philosophers

Five philosophers are seated around a table, and there are five single chopsticks placed between each pair of philosophers. When any philosopher wants to eat, he/she will have to acquire both chopsticks from his/her left and right.
*How can we have a deadlock-free and starvation-free way to allow the philosophers to eat freely?*

### Obvious Wrong Solutions

- **Wait till fork is available**: Obvious solution is wrong. If all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.
- **After taking left fork, check right fork. If not available, put down left one, wait, and repeats**: Also fails, for different reason, if all philosophers start algorithm simultaneously, will cause livelock, fail to make progress, cause starvation.
- **Use single mutex**: Before acquiring forks, wait(mutex), after putting down, signal(mutex). While no deadlock, no starvation, It has perfomance bug, only one can eat at an instant. With five forks, at least two philosophers can eat at same time.

### Tanenbaum Solution

The solution presented is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers.

- It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire cultery forks).
- A philosopher may move into eating state only if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.
- The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy.

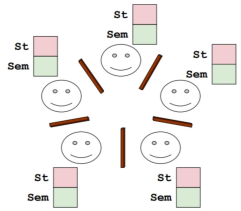
Dining Philosopher: **Tanenbaum Solution**

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

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

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

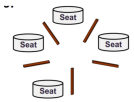
```
void putChpStcks( i )
{
    wait( mutex );

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

    signal( mutex );
}
```




### Limited Eater Solution

Dining Philosopher: **Limited Eater**

- If **at most 4 philosophers are allowed to sit at the table (leaving one empty seat)**
  ➔ **Deadlock is impossible!**




- Initial Values:
  - `seats = S(4)`
  - `chpStk = S(1)[5]`

```
void philosopher( int i ){
    while (TRUE){
        Think( );
        wait( seats );
        wait( chpStk[LEFT] );
        wait( chpStk[RIGHT] );
        Eat( );
        signal( chpStk[LEFT] );
        signal( chpStk[RIGHT] );
        signal( seats );
    }
}
```

# 6.5 Synchronization Implementations

POSIX Implementations in Unix:

### POSIX Semaphore

- Popular implementation of semaphore under Unix
- Header File:
  - `#include <semaphore.h>`
- Compilation Flag:
  - `gcc something.c -lrt`
  - Stand for "real time library"
- Basic Usage:
  - Initialize a semaphore
  - Perform `wait()` or `signal()` on semaphore

### *pthread* Mutex and Conditional Variables

- Synchronization mechanisms 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()`

- Programming languages with thread support will have some forms of synchronization mechanism
- **Examples:**
  - Java: all object has built-in lock (mutex) (monitor), synchronized method access, etc
  - Python: supports mutex, semaphore, conditional variable, etc
  - C++: Added built-in thread in C++11; Support mutex, conditional variable

# 7. Memory Management

How OSes create abstractions from memory and how they manage them.

## 7.1 Memory

- **Physical memory storage**: Random Access Memory (RAM) accessible in O(1), think as an array of bytes, with **physical address** as "unique index".
- **Contiguous memory region**: Interval of consecutive addresses.
- **Memory hierarchy**: OS abstracts hierarchy into useful mode, and manages it.

**Binding of Memory Address for executable:** Executable contains code (text), data layout (data), that has been compiled by the compiler into machine code. This is *load and run* into the main memory when we execute.

**Two types of data in a process**:

1. **Transient** Data: Valid for limited duration (e.g. function params, var)
2. **Persistent** Data: Valid for duration of program unless removed (e.g. global var, dynamically allocated memory).

**OS needs to perform the following tasks**:

- Allocate memory space to new processes
- Manage memory space for processes
- Protect memory space of processes from each other
- Provides memory related system calls to processes
- Manage memory space for internal use

## 7.2 Memory Abstraction

Memory abstraction: presenting a logical interface for memory accesses.

### No Memory Abstraction

- **Pros**: Straightforward, fast, addresses fixed during compile time.
- **Cons**: Both processes assume they start at 0, resulting in conflicts. Hard to protect memory.

#### Solution 1: Relocate Addresses

- Recalculate memory references when process is loaded into memory, e.g. if process B is located at address 8000, add 8000 to all memory addresses.
- However, the loading time is slow and it is not easy to distinguish a memory address from any arbitrary integer.

#### Solution 2: Base + Limit Registers

- Use a special **Base Register** that stores the starting address of the process memory space. All memory references will be offset by this register value. We then use a **Limit Register** to indicate the range, i.e. cannot access past the limit.
- However, there is a lot of overhead since we need to perform an addition and comparison per access. This is later generalised in segmentation.

### Memory Abstraction: Physical and Logical Addresses

- **Physical Address**: Generally, embedding **physical addresses** in programs is a bad idea.
- **Logical Address**: Thus let each process have a self contained, independent logical memory space that they will reference using logical addresses, then the OS will do the mapping.
- **Multitasking**: Need ways to partition memory, switch between processes using different memory partitions.
- When physical memory is full, either remove partitions used by terminated processes or swap blocked process to secondary storage.

## 7.3 Contiguous Memory Allocation

Allocating and managing **continuous** chunk of memory.

### Assumptions

- Each process uses a **memory partition** of a contiguous memory region.
- Physical memory **large enough** to contain processes with complete memory space.

### Fixed-Size Partitions

Physical memory split into fixed number of partitions, process occupies one.

- **Internal fragmentation**: When process does not occupy entire partition.
- **Easy to manage, fast to allocate**: Give any free partition, all same.
- **Need to fit largest process:** All smaller processes will waste space in partition large enough to accomodate largest process.

### Variable-Size (Dynamic) Partitions

Allocate just the right amount of space needed, forming **holes** (free memory spaces) between such partitions.

- **No internal fragmentation**: All processes get the exact space required.
- **External fragmentation**: When holes between processes become unusable, wasting space. Can be fixed via compaction, but slow (running processes need to stop). Better than internal, as internal is unreachable.
- **Maintain more info** in OS, **slower to allocate** to find appropriate region.

### Allocation Algorithms

Assuming the OS maintain a list of partitions and holes. Algorithm to locate partition of size N: Search for hole with size $M > N$. **Variants:**

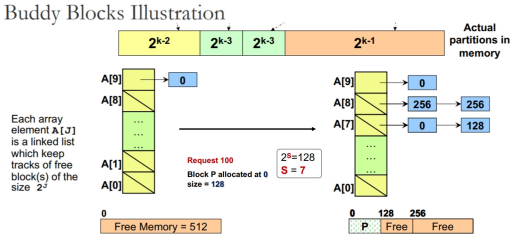
- **First-Fit**: Take the first hole that is large enough. Split it into two, give the process the required space, and the remaining space is a new hole.
- **Best-Fit**: Take the smallest hole that is large enough.
- **Worst-Fit**: Take the largest hole. (Creates larger holes for next process).
- **Merging and Compaction**: Try to merge freed partition with adjacent holes. Can also do compaction i.e. move partitions around, but this is expensive and should not be done frequently.

**Partition info** can be maintained as either a **linked list** or a bitmap. For **linked list**, each node contains: 1. Status: True = Occupied, False = Free, 2. Start Address, 3. Length of partition/hole, 4. Pointer to next node.

### Dynamic Allocation: Buddy System

**Buddy Memory Allocation**: Implementation of multiple free lists, provides efficient partition splitting, locating free partition (O(1)), partition deallocating and coalescing (merging).

- Free block is **split into half repeatedly** to meet request size. The two halves forms a sibling blocks (buddy blocks). Later, when buddy blocks both free, can merge to form larger block.
- **Buddy Blocks**: Two blocks B and C are buddy of size $2^k$, if the lowest K bits (bits 0 .. K-1) of B and C identical, and Bit $K$ of B and C is different.




## 7.4 Disjoint Memory Allocation

Allocating, managing memory in disjoint areas. (Remove assumption, process memory space now **split into disjoint** physical memory ranges.)

### 7.4.1 Paging Scheme

Split physical memory into **physical frames** and logical memory into **logical pages** of same size. Logical memory remains contiguous (page-wise), while each page can be mapped (loaded) into any available disjoint memory frame.

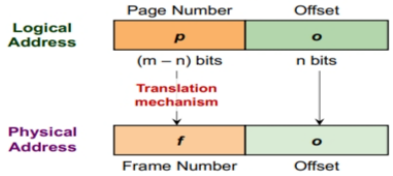
#### Page Table (Lookup Mechanism)

To support translation, use **page table** (one per process), array where the index is the page number and value is the frame number.

**Physical Address** = Frame_number $\times$ sizeof(physical_frame) + offset

Two design decisions to simplify address translation:
1. Keep frame/page size as a power of 2
2. Keep frame and page size equal

For page/frame size of $2^n$ and address of $m$ bits, to translate, we copy the last $n$ bits (the offset), then for the rightmost $m - n$ bits (page number), we index it into the page table and replace it with the value (frame number).




#### Paging: Analysis

- **No external fragmentation**: No leftover physical memory region
- **Has internal fragmentation**: When a logical memory space is not a multiple of page size. (Max one page per process not fully utilized).
- **Clear separation of logical and physical**: Flexible, simple translation

#### Paging: Implementation

Each process has own page table, stored in its PCB (*Memory Context*). This PCB is in RAM. However, require **two RAM accesses** for every mem reference.

1. Read the indexed page table entry to get frame number
2. Access to actual memory item

#### Paging: Translation Look-Aside Buffer

The TLB is on the chip, and provides hardware support for paging. It caches 4KB of page table entries, cache multiple at once, like associative cache.

- **Fast**: Takes 1ns vs 50ns for RAM
- **TLB-Hit vs TLB-Miss**: RAM is only accessed upon the latter, and TLB is updated after that
- **Context switching**: In CS2106, can assume TLB is fully flushed when context switch (so new process does not get incorrect translation), as it is part of a process' hardware context.

### Paging: Protection

Extend the paging entries to include bits to support memory protection.

1. **Access right bits**: On whether the page itself is writable, readable or executable, e.g. cannot write over text of process, but can execute
2. **Valid bit**: Some pages may be out of range for certain processes for certain reasons. OS will set these valid bits when running. If out-of-range access is done, OS will catch.

### Paging: Page Sharing

Allow processes to share same frame, hence multiple pages pointing to same physical frame. E.g. some library code shared between processes. Use a shared bit in the page table entry to track whether the page is shared.

- **Shared code page**: When shared library code or system calls, etc.
- **Copy-On-Write**: When a parent forks, the parent and child share the same pages, i.e. page table is copied but frames remain. Using a shared bit, when the child needs to update a shared page, then the frame is duplicated and page "unshared".

## 7.4.2 Segmentation Scheme

The memory space of a process contains **multiple logical regions** with different usages, permissions, lifetime, scope etc. (E.g. 4 types: Text, Data, Heap, Stack). No need to put all the regions together in a contiguous logical memory space. Difficult to allow them to shrink/grow freely individual if together. Also hard to check if a memory access is in-range. Hence, **manage memory at the level of memory segments**.

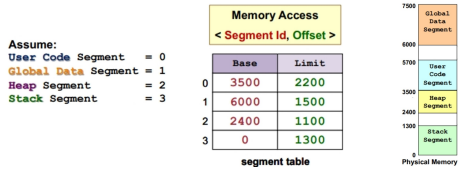
### Segmentation Scheme

All program memory references specified as (Segment name + Offset).
**Memory segments**: Split the regions into their own segments

- **Name**: For reference, usually translated to an index, e.g. Text = 0, Data = 1, Heap = 2, etc.
- **Base**: The physical base address
- **Limit**: Indicate range of segment

### Segment Table

Keep a table of [Base, Limit], indexed by the name indices / segment ids. Store segment table inside registers as size is fixed and small, for fast use.




1. With [SegId, Offset], we use SegId to get the [Base, Limit].
2. Check if Offset $<$ Limit, if so, segmentation fault.
3. Else access Base + Offset

### Segmentation Analysis

- **Independent segments**: Each segment is contiguous and independent, and can shrink and grow independently
- **External fragmentation**: Variable size, contiguous memory regions result in the same problem
- **Not the same as paging**: Solving different problems

## 7.4.3 Segmentation with Paging

Each segment is now composed of pages and has a page table of its own. The segment table now points to the page table address instead of the base address. Page limit remains unchanged.

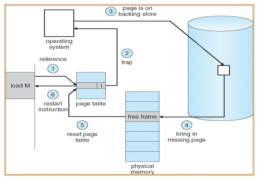
# 7.5 Virtual Memory Management

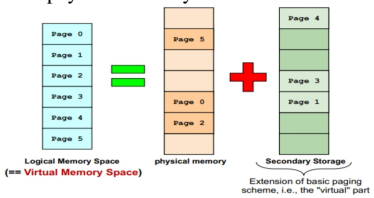
Fully separate **logical memory** from **physical memory**. Use of secondary storage (HDD, SSD etc.) as extended memory region, as physical memory (RAM) may not be large enough to hold processes' logical memory space completely.

- A popular solution: keep some pages in physical memory, others on secondary storage. Bringing pages in and out of memory called **swapping**.
- **Logical Memory Address Size not restricted by physical**
- **Efficient use of physical memory**: Only needed pages swapped in.
- **More processes in memory**: Improves CPU utilisation since more processes can reside in memory, and to be chosen to run

## 7.5.1 Extended Paging Scheme

We add a "Is Memory Resident" bit in page table entries. When CPU tries to access non-memory resident pages, a **page fault** occurs. OS then needs to bring non-memory resident page into physical memory.




### Page Fault (See diagram above)

Updated page accessing process, now with page faults.
1. (Hardware) Check page table. If memory resident, done, else continue
2. (Hardware) Page fault: TRAP to OS
3. (OS) Locate page in secondary storage
4. (OS) Load copy of page in physical memory frame
5. (OS) Update page table
6. (OS) Go back to step 1 to retry

- **Thrash**: Poor performance of virtual memory / paging system when same pages loaded repeatedly due to lack of main memory.
- **Thrashing**: Constants state of paging and **page faults**.

### Virtual Memory and Locality

- *Temporal Locality*: Memory address used now likely to be used again. Cost of loading is amortized.
- *Spatial Locality*: Memory addresses close to the address used now likely to be used soon. A page contains these many consecutive locations.

### Demand Paging

What we have described thus far is demand paging, i.e. OS only copies a page into memory if a page fault occurs, as compared to anticipatory paging. Unneeded pages never loaded, more efficient use of physical memory.

- Process start with **no memory resident page**, only allocate page when there is page fault. Fast startup time, small memory footprint.
- Process may be sluggish at start due to multiple page faults, may cause cascading thrashing on other processes.

## 7.5.2 Page Table Structure

**Page Table** exists with process info, takes up **physical memory space**! Even if large number of pages are on disk (secondary storage), page tables themselves still take up a lot of space in memory. This results in **high overhead and fragmentation**, since the table itself needs to occupy several pages.

### A. Direct Paging

We keep all entries in a single page table, and we allocate each process this huge page table. To compute the size of this table:

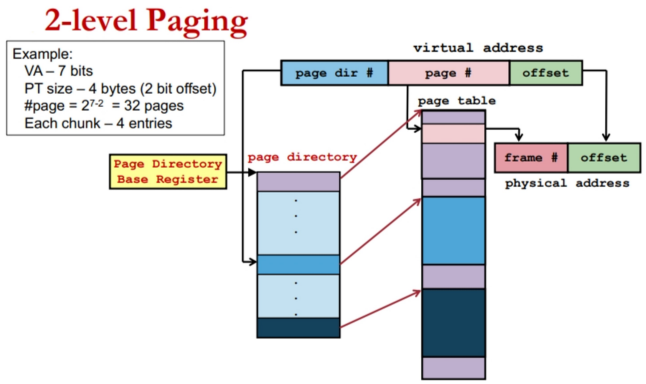
1. Find number of pages from virtual address (byte-addressed) / Page Size.
2. For $2^p$ pages (b-bit addresses), need $p$ bits to specify one unique page.
3. If virtual address has 32 bits, and each page/frame is $4KB = 2^{12}B$, then $p = 32 - 12 = 20$ bits.
4. This also means $2^{20}$ pages and page table entries.
5. If each page table entry is 2 bytes, then page table size is $2^{20} \times 2\text{B} = 2\text{MB}$.

But not all processes use the full virtual memory space, so much of this table will be unused

### B. 2-Level Paging

We further split the page table into regions, and only allocate these regions when needed. Keep directory of these regions. Each table entry points to the base address of the next page table. Further extendable to multilevel paging.

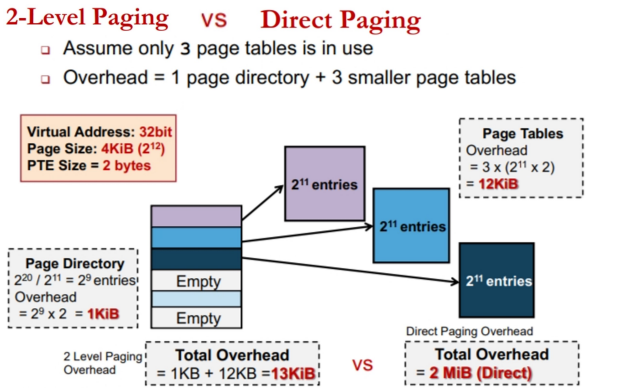
- **Size of page table**: Generally we want each (region) table to be same size as a page, to reduce fragmentation.
- **Page directory base register**: One register that stores base address of process' page directory, update during context switching.



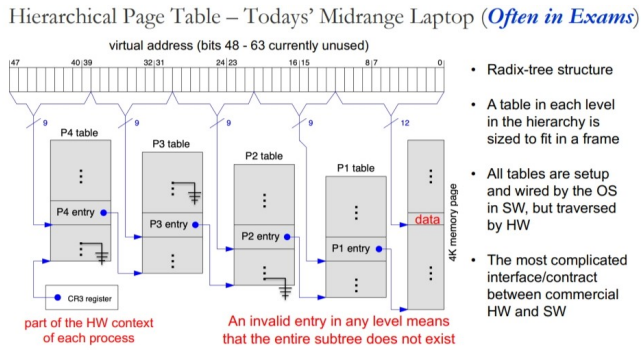

- **Pros**: Enables page table structures to grow beyond size of a frame. Can have empty entries in page directory, corresponding page tables need not be allocated.
- **Cons**: Requires two serialized memory accesses just to get the frame number (directory + page table), only then to access data.
- Use TLB? TLB hits eliminate page-table accesses, but TLB misses experience longer *page-table walks* (traversal of page-tables in hardware).

### Overhead calculation (2 Level Paging)

1. E.g. all tables/pages = 4KB, each entry = 2B, virtual address = 32 bits.
2. Since each frame/page is 4KB, the offset is $2^{12}$B, i.e. we only look the first 20 bits of the address.
3. Since each page table (the ones that are split) is 4KB big, and each entry is 2B, then each table has $2^{12} \div 2 = 2^{11}$ entries.
4. That means directory has $2^{20} \div 2^{11} = 2^9$ entries. This means it will take up $2^9 \times 2\text{B} = 2^{10}\text{B} = 1\text{KB}$.
5. Although technically directory itself would also take 1 page, we can consider it to only have an overhead of 1KB. Let's say we only allocated and are using 3 page tables.
6. Then total overhead = 1KB+12KB = 13KB.




### Hierarchical Page Table




### C. Inverted Page Table

We want to keep a page table that tells us which process is using which frame. We can thus keep an array with the same size as number of frames. In each entry, we store the **PID** and **page number**.

- **Fast lookup by frame**: Often to support RAM management operations
- **Ease of use**: Frame management is a lot easier + one table for all processes
- **Slow translation**: From page number to frame number, since we need to search through the entire table. However, normally we use page directories and tables for that. Inverted page table used as auxiliary structure, e.g. to show processes using a frame.

## 7.5.3 Page Replacement Algorithms

When there are no free pages during a page fault, we will need to replace an existing page. Based on a **dirty bit** for the page entry, if the page has been modified, it will need to be written back to disk.

### Memory Reference Strings

The offset does not matter when talking about page replacement, only page numbers. A sequence of page numbers is called a memory reference string. (E.g. 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4).

### Memory Access Time

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

Want to reduce $p$ (probability of page fault) since $T_{fault} >> T_{mem}$.

### A. Optimal Page Replacement (OPT)

Replace the page that will not be used again for the longest period of time. **Minimum page faults guaranteed**, used as a benchmark. However, **need to know the future**, not feasible irl.

### B. FIFO Page Replacement

Evict oldest memory page (based on loading time, NOT access time).

- **Queue**: Maintain queue of resident pages, update during page fault.
- **Simple to implement**: No need hardware support
- **Belady's anomaly**: Generally, if number of frames in RAM increases, page faults should occur less frequently. However, for FIFO, we may actually see the opposite.
- **Does not exploit temporal locality**: This is the reason for above.

### C. LRU: Least Recently Used Page Replacement

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

- **Temporal locality**: Makes use of it. Does not suffer Belady's anomaly.
- **Close to optimal algorithm**: Good approximation
- **Difficult implementation**: Need to keep track of last access time, thus need substantial hardware support.

#### LRU Implementation Details

1. **Time counter**: A logical time counter that increments for every stored reference and is stored along with it. However, deletion is O(n) since we need to find the page with the lowest counter. We may also have overflow issues with the counter.
2. **Stack**: Maintain a stack of page numbers. When a page is referenced, we pop it out from the stack (if inside), and push it to the top. For replacement, we remove the bottom most page. Hard to implement in hardware, and not exactly a stack, since we can pop from anywhere.

### D. Second-Chance Page Replacement (CLOCK)

Modified FIFO, maintain a separate reference bit for each page entry. When all page entries have same reference bit, effectively becomes a FIFO algo.
1. We maintain a circular queue of page numbers, and a pointer to the "oldest", or victim page.
2. When we load a page entry, we set the reference bit to 0 (for CS2106).
3. Upon accessing the page entry, we will set the reference bit to 1
4. When a page replacement is required, we check the current victim page
(a) If the reference bit is 0, we replace it.
(b) Else if the reference bit is 1, we set it to 0 and move the pointer to the next page
5. Repeat step 4 until a victim page with reference bit 0 is found.

## 7.5.4 Frame Allocation

If there are N physical frames and M processes competing for frames, we need to distribute these frames.

1. **Equal allocation**: Each process gets $N/M$ frames.
2. **Proportional allocation**: Each process gets $\frac{size_p}{size_{total}} \times N$ frames.

### Local (Page) Replacement

Victim page selected among **pages of process** that caused the page fault.

- **Constant frame allocation**: Number of frames is the same, performance is stable between runs.
- **Insufficient allocation**: Will result in hindering of the process
- **Thrashing**: Limited to one process, but that single process can hog the I/O (to bring non-resident pages into RAM) and degrade the performance of other processes.

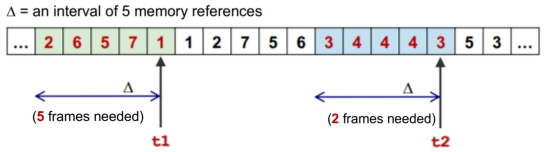
### Global (Page) Replacement

Victim page is selected among pages of all processes.

- **Self-adjustment**: Processes that need more frames can get them from other processes
- **Malicious processes**: Can affect others.
- **Inconsistent performance**: Between runs
- **Cascading thrashing**: One process that thrashes will steal pages from others, resulting in other processes also thrashing

### Working Set Model

Generally, the **(working) set** of pages referenced by a process is quite constant in a period of time due to locality. The number of page faults is minimal until process transmits to new locality, e.g. new function call, etc. **Working Set Model**: monitors the memory usage patterns of processes **(look back last $x$ frames and see how many unique frames used)** and adjusts the frame allocations based on the working set of each process.




- Define a **working set** window $\delta$, which is an interval of time
- $W(t, \delta)$ is the set of active pages in the interval at time $t$
- We thus want to allocate enough frames for pages in $W(t, \delta)$ to reduce page fault
- Accuracy of model depends on choice of $\delta$: Too small, miss pages in current locality. Too large, contains pages from different localities.

### All Page Entry Bits Thus Far

1. Access Right Bits
2. Valid Bit
3. Is-Shared Bit
4. Is-Memory Resident Bit
5. Dirty Bit
6. Reference Bit (for CLOCK)

# 8. File Systems

**Motivation**: Physical memory (RAM) is violate, use external storage for *persistent* info. Direct access to storage media **not portable / standard**. **File systems** consists of files and directories, and provides an abstraction for access, high level resource management scheme, protection and sharing between processes and users.

## General Criteria (of File System)

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

# 8.1 File System Abstractions

## 8.1.1 File

- **File**: A logical unit of information created by a process, an ADT.
- **File Metadata**: Attributes associated with the file.
- Metadata includes: Name, Identifier (UID), Type, Size, Protection (permissions), Time/Date/Owner, Table of Content.
- *Abstract Data Type (ADT)*: Set of common operations with various possible implementation.

### File Name

A human readable reference to the file. Different FS have different rules. Rules include length, case-sensitivity, allowed special symbols and file extension. Some FS use extension to indicate file type. (Name.extension)

### File Type

Each file type has an associated set of operations and possibly specific program for processing. Common file types:

- **Regular files**: contains user info
  - **ASCII files**: text files, source codes, etc. Can be printed as is.
  - **Binary files**: executables, mp3, pdf, etc. Have a predefined internal structure that needs a specific program to process
- **Directories**: system files for FS structure
- **Special files**: character/block oriented

**Distinguishing File Type**:

- **Windows**: Uses file extension as file type
- **UNIX**: Uses magic number embedded at beginning of file

### File Protection

Control access to information stored in file. Most common approach: restrict access based on user identity.
Type of accesses: (R/W, Append, Execute[load into memory, execute], Delete, List[read metadata]).
*Related to CS2107 Access Control List, Permission Bits.*

- **Access Control Scheme** A list of user identity & allowed access types.
- Very customizable, but too much information associated with file.
- Unix: Classify users into three classes (owner, group, universe), and define permission of R/W/E for these 3 classes of users.
- **Access Control List (ACL)**: Minimal ACL (same as permission bits), extended ACL (added named users/groups).

## 8.1.2 File Data

### Different Structure

- **Array of bytes**: Just raw bytes having a certain offset from the start of file.
- **Fixed length records**: Each record has a fixed size. This array of records can grow / shrink, and easy to jump to any record using offset.
- **Variable length records**: Flexible but hard to locate a record.

### Access Methods

- **Sequential access**: Read in order from beginning. Can't skip, can rewind.
- **Random access**: Read in any order, via read(offset) or seek(offset). UNIX and Windows both use seek.
- **Direct access**: Basically random access but with records instead of bytes. (For files with fixed-length records).

## 8.1.3 File Operations

Generally these operations are **system calls**, as it provides protection, and concurrent and efficient access.

### Metadata Operations

1. Rename
2. Change attributes, e.g. file access permissions, dates, ownerships, etc.
3. Read attribute, e.g. get file creation time

### Data Operations

1. **Create** a new file with no data
2. **Open** a file to prepare the necessary information for file operations later
3. **Read** data from file, usually from current position
4. **Write** data to file, usually from current position
5. **Reposition** (seek) to new location
6. **Truncate** removes data between specified position to end of file

### Representation of Open File

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

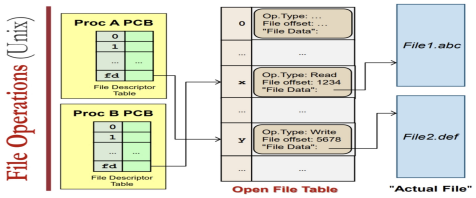
### Two Table Approach (One system-wide, PCB per process)

1. **System-wide open-file table**

- Keeps track of open files in the system, one entry per unique file.
- If one process opens the same file twice, or two processes open the same file, there will be two separate entries
- If one process opens a file then forks, only one entry here.

2. **Per-process open-file table**

- **File Descriptor Table**: Keeps track of open files for a process, also known as file descriptors
- Each entry points to a system-wide table entry
- If one process opens a file then forks, there will be two fds pointing to the same system-wide table entry. They will thus share the same offset.




## 8.1.4 Directory

Helps user group files, and helps system keep track of files.

### Single-Level Directory

All files are contained under the root directory. (E.g. Audio CD)

### Tree-Structured Directory

Since directories can contain directories, they are like trees or subtrees. Files are like leaves.

- **Absolute pathname**: Can refer to file by path from root to file.
- **Relative pathname**: Can refer to file by path from current working directory (CWD).

### DAG Directory Structure

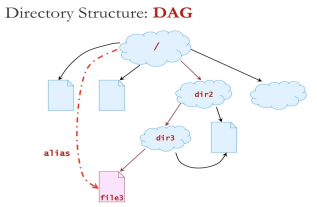
If a **file can be shared**, might get a DAG (Directed Acyclic Graph) from a tree, "skips" levels, i.e. share a file/directory such that it appears in multiple directories but refers to the same copy of actual content.

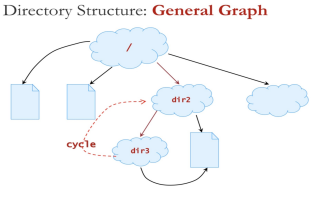
- **UNIX Hard link**: Directories A and B have separate pointers to file F . Only works with files.
  - **Pros**: Low overhead, only adds pointers
  - **Cons**: Deletion problems - what if one directory deletes F?
- 'ln' command in UNIX
- **Symbolic link**: Directory B creates a special link file G that contains path name of F . When G is accessed, we access F instead.
  - Pros: Simple deletion. B deletes G, not F . A deletes F , and G remains but does not work.
  - Cons: Larger overhead - G takes up actual disk space
- ln -s command in UNIX

### General Graph Directory Structure

General Graph Created when the tree has a cycle. Though possible in UNIX, it is not desirable.

- **Hard to traverse**: Need to prevent infinite loops.
- **Hard to determine** when to remove a file/directory
- **In UNIX**: general graph created when symbolic link allowed to link to directory.



### UNIX Context: File Operations

- **int open(char *path, int flags)**: Takes in a path and flags, and returns a file descriptor (fd) integer.
- **int read(int fd, void *buf, int n)**: Takes in a fd, a buffer and an integer n and reads up to n bytes into the buffer. It is sequential - starts at current offset and increments offset by bytes read.
- **int write(int fd, void *buf, int n)**: Takes in a fd, a buffer and an integer n and write up to n bytes from buffer into the file. It is sequential - starts at current offset and increments offset by bytes written. Appends new data if file size goes beyond EOF. Throws error if file size limit, quota, disk space, etc. are exceeded.
- **off_t lseek(int fd, off_t offset, int whence)**: Takes in a fd, an offset and a whence and moves current position in file. If offset is positive, move forward, else move backward. The value of whence can be: SEEK_SET: set absolute offset (from start of file), SEEK_CUR: set relative offset from current position, SEEK_END: set relative offset from end of file.
- **int close(int fd)** Closes an opened fd, and kernel can remove associated data structures. Process termination automatically closes all open files.

## 8.1.5 Disk I/O

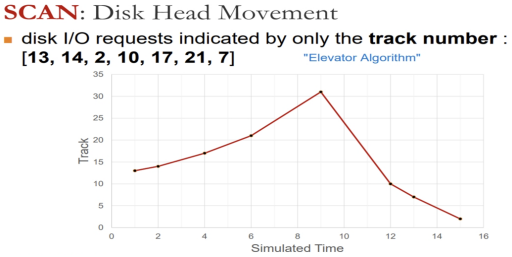
### Disk Structure (Recap)

- **Track**: One ring around the disk. One disk can have many tracks of different radii.
- **Sector**: Much like the sector of a circle, this is a sector of a track
- **Disk head**: The reader stick that moves above the disk and transform the disk's magnetic field into electrical current or vice versa
- **Rotation**: By rotating, we can change sector
- **Seek**: By shifting the disk head towards and away from the centre of the disk, we can change track

### Disk Scheduling Algorithms

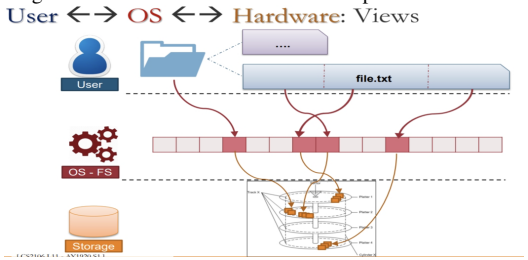
Due to significant seek and rotational latency, OS should schedule disk I/O requests. This is to **reduce overall waiting time**, focus on **reducing seeking time** (aka moving between tracks).

- **First-Come First-Serve**: Move disk head to next track.
- **Shortest Seek First**: Move head to nearest track within schedule.
- **SCAN**: Bidirectional, go from innermost to outermost, then back to innermost. Much like lift algorithms.
- **C-SCAN**: Unidirectional, go from outermost to innermost, then reset.
- **Deadline**: 3 queues for requests: 1. Sorted, 2. Read FIFO (sorted chronologically), 3. Write FIFO (sorted chronologically).
- **noop**: No sorting involved. Actually same as FCFS
- **cfq**: Completely fair queuing, time sliced and per process sorted queues
- **bfq**: Budget fair queuing or multiqueue, fair sharing based on number of sectors requested



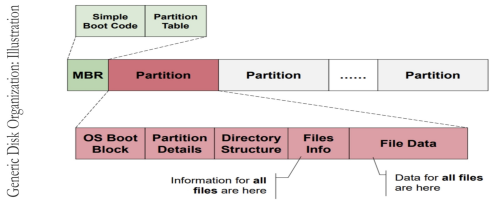

# 8.2 File System Implementation

Earlier, we saw that a disk has various sectors. To the OS, what it sees is a 1-D array of logical blocks, usually 512B to 4KB big. The mapping between logical blocks and sectors is hardware dependent.




## 8.2.1 Disk Organisation

- **Master boot record**: Found at sector 0. Contains: Simple boot code and Partition table.
- **Partitions**: Each partition is an independent file system. Contains: 1. OS boot block (information for boot-up), 2. Partition details, e.g. total number of blocks, number and location of free disk blocks, 3. Directory structure, 4. File information, 5. Actual file data.




## 8.2.2 Files Info and File Data

We view a file logically as a collection of logical blocks. There will be **internal fragmentation** as file size may not be a multiple of logical block size.
**Good file implementation**: When allocating file data, need to keep track of logical blocks, allow efficient access and utilise disk space efficiently.

### A. Contiguous File Block Allocation

Allocate consecutive disk blocks to a file. File information will just store start + length for each file.

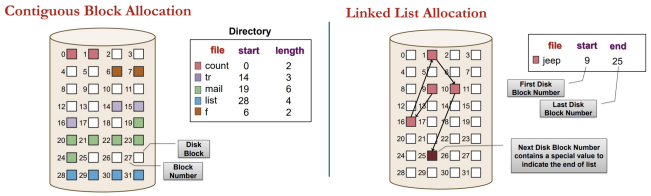
- **Easy to keep track**: Each file just needs starting block + length.
- **Fast access**: Just need to seek first block
- **External fragmentation**: After a lot of creation/deletion, disk can have many small "gaps".
- **File size**: Needs to be specified in advance, hard to change.

### B. Linked List (stored in block itself)

Maintain disk blocks as linked list. Each disk block stores the next disk block number besides the file data.
**File information** will store first and last disk number for each file.

- **No external fragmentation**: Solves the previous problem
- **Slow access**: O(n) access
- **Less usable space**: Need to use space for pointer
- **Less reliability**: Fails if one pointer is incorrect

### A. Contiguous. B. Linked List V1




### C. Linked List V2.0 (FAT)

Same as linked list but now we move all pointers into a **file allocation table (FAT)** in memory. This is used by MS-DOS.

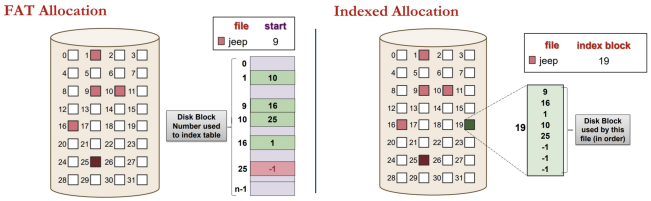
- **Faster access**: Linked list traversal is now all in memory.
- **Takes up space**: FAT tracks all disk blocks. This number can be huge when disk is large, consuming valuable memory.

### D. Indexed Allocation (Separate Index Block)

We use one block per file to store an array of indices of all other blocks. For example, index of 0-th block will be value of IndexBlock[0].

- **Less memory overhead**: Only index block of opened files need to be in memory (compared to whole FAT).
- **Fast direct access**: No need for traversal.
- **Limited max file size**: Max number of blocks is the max array size of index block.
- **Index block overhead**: Consumes space, keep updated.
- **Variations**:

1. **Linked Scheme**: Keep linked list of index blocks, index block contains pointer to next index block.
2. **Multilevel Index**: Similar to multi-level paging. FIrst level index points to a number of second level index blocks etc.
3. **Combined Scheme**: Both direct indexing and multi-level scheme. **Unix I-node** does this. (Direct pointers, single indirect block, double indirect, triple indirect)



## 8.2.3 Free Space Management

**Free List**: We want to know which disk blocks are free or not free, so that we can easily allocate (remove from free list) or free blocks (add to free list space).

### A. Bitmap (to track free blocks)

Each block is 1 bit, if it's 1 means it's free, else it's taken.

- **Easy to manipulate**: Use simple bit operations to find first free block or n-consecutive free blocks
- **Need to keep it memory**: For efficiency reasons.

### B. Linked List

We basically have a linked list of blocks used to contain indices of free blocks.

- **Easy to locate free block**: Just use first block.
- **Just need pointer to first block in memory**: Though we can cache other linked list blocks for efficiency.
- **High (Low) overhead**: We can mitigate this by using free blocks to store this list. Just make sure to give up free block holding linked list last.
- **Runs alternative**: If the free space tends to be in contiguous runs, we can instead store first index + length. But may not work as well as fragmentation worsens.

## 8.2.4 Directory Structure Implementation

The goal of a directory structure is to keep track of files in directory, and map file name to file information. A file is opened before use using some pathname (E.g. `open(''data.txt'')`)

- **Path name**: List of directory names traversed from root.
- **A sub-directory**: usually stored as file entry with special type in a directory.

### A. Linear List (Indexed List)

Directory contains a list of entries, one entry per file. The entry contains the file name (and possibly other metadata) and file information or pointer to file information.

- **Inefficient searching**: Especially for large directories or deep tree traversal, e.g. (search all entries to see if file exists).
- **Solution**: Cache the last few searches

### B. Hash Table

Directory contains a size N hash table. We hash the file name from 0 to $N-1$, and use chained collision resolution.

- **Fast lookup**
- **Cons: Limited size for hash table, depends on a good hash function**

### C. File Information

Two approaches to store file information in a directory entry:

1. **Store everything in directory entry**: Likely have some fixed size entry
2. **Store only file name and pointer**: Pointer to some other data structure for more information

## 8.2.5 Walkthrough on File Operation

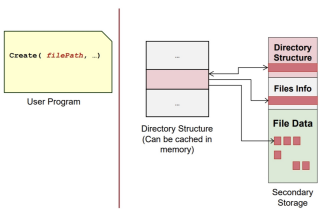
We have looked at static information for a FS stored on media. At runtime, when user interacts with file, run-time information is needed, which is maintained by OS in memory. Involves common in-memory information (System-wide open file table, per-process open file table, buffer for disk blocks read from/written to disk.)

### Create [touch, New-Item etc.]

Walkthrough on file operation: **Create**

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

File Creation: Illustration

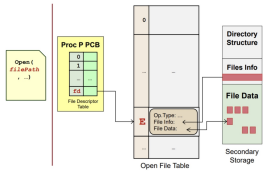



### Open

Walkthrough on file operation: **Open**

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

**File Open**: Improved Understanding




## 8.2.6 Extra Knowledge

### FS Consistency Check

Performed when sudden power loss or crash happens. *CHKDSK* for Windows, *fsck* for Linux.

### Defragmentation

When fragmentation on disk gets too severe, I/O performance may suffer. On Windows, official and 3rd party software are used to solve this problem. On Linux, the file allocation algorithm allocates files further apart, and free blocks near existing data blocks are used if possible, keeping fragmentation low when disk occupancy is ¡ 90%.

### Journaling

Like saving state. Information or actual data is written into a separate log file before an operation is performed, so that it can recover to an earlier stable state or re-perform the interrupted file operation.

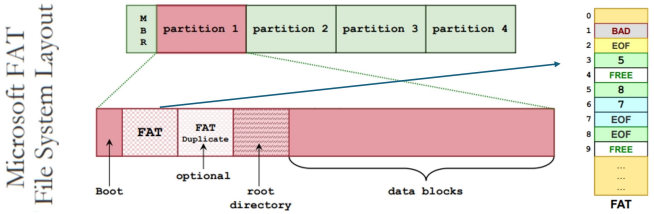
### Interesting FS

- **Virtual FS**: Provides another layer of abstraction on top of existing file systems, allowing applications to access files on different file systems
- **Network FS**: Allows files to reside on different machines, and file operations are now network operations
- **New Technology FS NTFS**: Used in Windows XP onwards, provides file encryption, compression, versioning, and hard/symbolic link
- **Ext3/Ext4**: Variant of Ext2 with journaling and expanded max file and file system sizes
- **Hierarchical FS Plus (HFS+)**: Used in Mac OS X, provides compression, encryption, large FS/- files/number of files support, metadata journaling.

# 8.3 File System Case Studies

## MS-DOS FAT

Microsoft Disk Operating System File Allocation Table. Works with linked list block allocation and the FAT is cached in RAM for traversal. The index of the block is the index of its FAT entry. A block is also called a cluster.

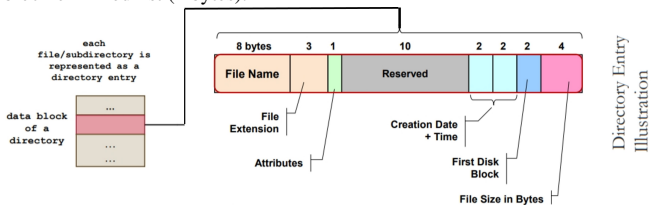



### FAT FS:

- **Each partition** has: OS boot block (to start file system), FAT, FAT duplicate (optional), Root directory, Data blocks.
- **Each FAT Entry has values**: FREE code (block is unused), Block number of next block of file, EOF code i.e. NULL, BAD block (unusable).
- **Directory**: Special type of file. Root directory in special location, rest in data blocks. Each file or subdirectory **represented by a directory entry**.

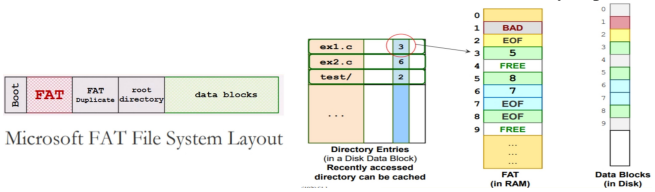
### Directory Entry

Each entry has a fixed size of 32 bytes. The first byte of name may have meaning. Attributes (e.g. read-only, is-Dir/file flag, hidden). First disk block of linked list (2 bytes).




### MS-DOS FAT Tracing Process

1. Get required directory table from disk blocks. Likely cache at this point.
2. Get required directory entry and find first disk block number.
3. Use FAT to find remaining disk block numbers, until EOF code reached.
4. Use numbers for disk access on file data blocks. If subdirectory, repeat 1.



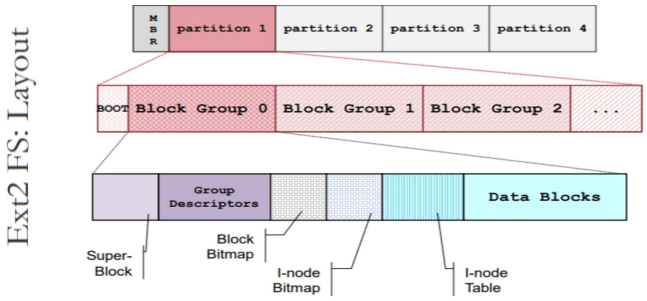

### MS-DOS FAT Common Tasks

- **File Deletion**: Soft delete, set first letter in filename to a special value 0xE5, free data blocks in link list.
- **Free space management**: No record, calculated by going through FAT.
- **Fat12/16/32**: Bigger FAT, (Disk block index 12 for FAT12 etc.), more disk block, more bits to represent cluster. Larger usable partition size.
- **Long File Name Support, VFAT**: Virtual FAT, use multiple directory entries for file with long file name.

## Linux Ext2

Popular intricate file system in Linux, embeds traditional Unix FS ideas. The layout of the file system is different.

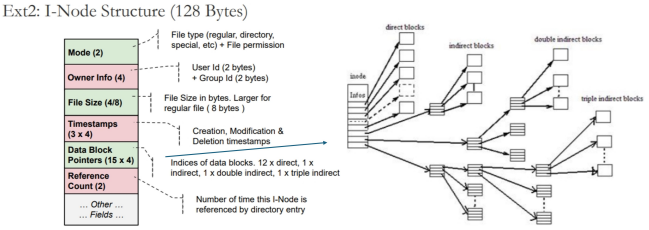
- Blocks are grouped into **block groups**.
- each file/directory is described by a single special structure called **I-Node (Index Node)**, containing file metadata and data block addresses.




### Partition Information

- **MBR: Master boot record**: Same as before
- **Block Groups**: OS boot block.
  **Superblock**: Describes the whole file system, e.g. total I-Node number, I-Nodes per group, total disk blocks, disk blocks per group, etc. Duplicated in each block group for redundancy.
  **Group descriptors**: Describe each of the block group, e.g. number of free disk blocks and I-Nodes per group, location of bitmap etc. Duplicated in each block group for redundancy.
  **Block bitmap**: Keeps track of usage status of blocks in this block group (1 = Occupied, 0 = Free)
  **I-Node bitmap**: Same but for I-Nodes
  **I-Node table**: Array of I-Nodes in this block group
  **Data blocks**

### I-Node Structure

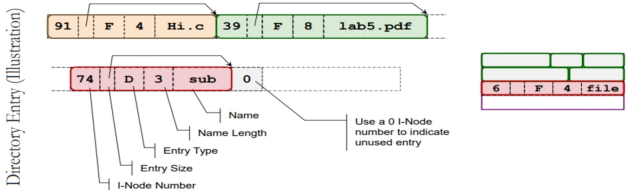



### Multilevel Data Block Pointers

Allows for fast access to small files and flexibility to handle huge files. The first 12 data block pointers directly point to the disk block. The next pointer points to a single indirect block, which contains direct pointers. The next pointer points to a double indirect block, which contains pointers to single indirect blocks. Finally, we have one triple indirect block. Assuming each disk block address is 4B, each disk block is 1KB, then in total, we have 12 × 1KB + 256 × 1KB + 2562 × 1KB + 2563 × 1KB = 16843020KB = 16GB.

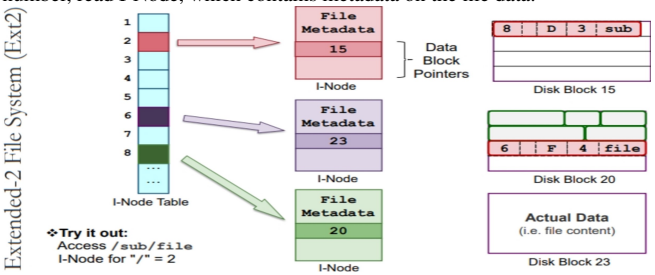
### Directory Structure

A directory is just another file. Fata blocks form a "linked list" of directory entries for files and subdirectories. Each entry has: I-Node number for file/subdirectory, 0 for unused entry, Size to traverse to next entry, length of name, Type (file, subdirectory, special etc), Name of file/subdirectory (up to 255) characters.




### Linux Ext2 Tracing Process

1. Get root directory I-Node. Generally a fixed number e.g. 2. Read it.
2. If next part in pathname is **subdirectory**: (a) Locate directory entry in curr directory, (b) Get I-Node number, read I-Node, contains metadata on subdirectory data, (c) Current directory is now this subdirectory.
3. Else if is **file**: (a) Locate directory entry in curr directory, (b) Get I-Node number, read I-Node, which contains metadata on the file data.




## Some Ext2 Operations

### Open

1. Go through tracing process. If F is not found, terminate with error.
2. Load file info into a new entry E in the system-wide table (i.e. store the I-Node pointer)
3. Create an entry (file descriptor) in process P 's table to point to E
4. Return fd of this entry

### Delete

1. Remove its directory entry by pointing the previous entry to the next entry. If first entry, make it a blank record.
2. Update I-Node bitmap and mark corresponding I-Node as free
3. Update block bitmap and mark the corresponding blocks as free

### Hard Link

1. Create a new directory entry in B with same I-Node number as X in A
2. Update X I-Node's reference count
3. For deletion, decrement the reference count, then when it goes to 0, perform actual deletion (or cleanup).

### Symbolic Link

1. Create new file Y in directory B, i.e. new Y I-Node + new dir. entry for Y
2. Store pathname of file X in file Y , i.e. as the file content
3. As only pathname is stored, the link easily invalidated.