CS2106 TABLE OF CONTENTS

Learning Objectives

- OS Structure and Architecture
- Process Management
- Memory Management
- File Management
- OS Protection Mechanism

- 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/threaded programs and avoid common pitfalls such as deadlocks, starvation and race conditions
- Write system programs that utilizes POSIX system calls for process, memory and I/O management

1. INTRODUCTION

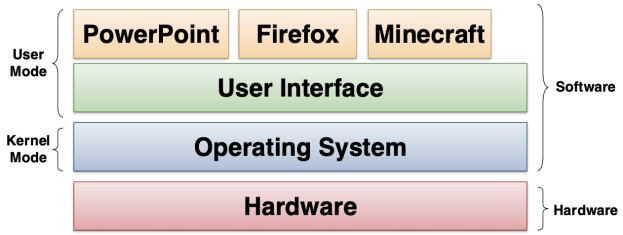
Operating System (OS) is a **software program** that acts as an intermediary between a computer user and the computer hardware.

No OS	• Minimal overhead • Not portable, Inefficient use of computer
Batch OS	• Execute job sequentially, inefficient use of CPU
Time-sharing OS	• Multiple users use terminals to schedule jobs • Virtualization of hardware: Each program executes as if it has resources all to itself
Personal computer	• Dedicated to user (not timeshare between multiple users)

Motivations for OS

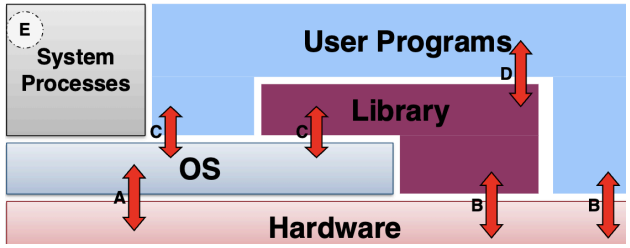
- **Resource Allocator**
  - Manage resources (CPU, memory, I/O devices) and coordination
  - Process synchronization
  - Resource sharing: Multiple programs allowed to execute simultaneously
  - Arbitrate potentially conflicting requests for efficient and fair resource use
- Simplify programming
  - High level **abstraction** (same API to access all types of same category of hardware) of hardware → Efficiency, programmability, portability
  - Hardware virtualization: Each program executes as if it has resources all to itself
  - Convenient services
- **Control program** , controls execution of programs
  - Enforce usage policies
  - Security and protection
  - User program portability: Across different hardware
- Efficiency
  - Sophisticated implementations
  - Optimized for particular usage and hardware

High-level view of OS



KERNEL mode of OS: Direct access to all hardware resources
User mode of other software: Limited (or controlled) access to hardware resources

Generic OS Components



A: OS executing machine instructions
- Between OS and hardware

B: Normal machine instructions executed (program/library code)
- Between Library / User Programs and Hardware

C: Calling OS using system call interface
- Between Library / User Programs and OS

D: User program calls library code
- Between Library and User Programs

E: System processes (Provide high-level services, usually part of OS)

Kernel

Deals with hardware issues; Provides system call interface; Special code for interrupt handlers, device drivers

Kernel code has to be different than normal programs:
- ❑ Can't use system calls in kernel code
- ❑ Can't use normal libraries
- ❑ No "normal" I/O

Implementing Operating System
- ❑ Programming Language: Higher Level Languages (HLLs), especially C, C++
- ❑ Common code organization: Machine independent HLL, Machine dependent HLL, Machine dependent assembly code
- ❑ Challenges: Debugging is hard, Complexity, Enormous Codebase

Operating System Structure

Layered Systems

❑ Generalization of **monolithic** system
❑ Organize the components into hierarchy of layers
■ Upper layers make use of the lower layers
■ Lowest layer is the hardware
■ Highest layer is the user interface

Client-Server Model

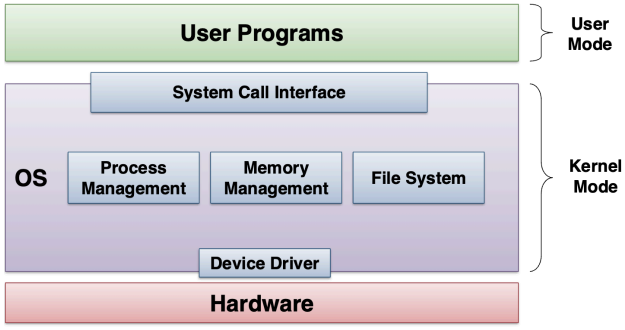
❑ Variation of **microkernel**
❑ Two classes of processes:
■ Client process request service from server process
■ Server process built on top of the microkernel
■ Client and server process can be on separate machine!

Monolithic OS	Microkernel OS

Kernel
• One big special program
• Good software engineering principles possible with:
  • modularization
  • separation of interfaces and implementation
	Kernel
	• Very small and clean
	• Provides basic and essential facilities:
	  • Inter-Process Communication (IPC)
	  • Address space management
	  • Thread managemenT

	Higher-level OS services: • Are built on top of the basic facilities • Run as server process **outside** of the kernel • Use IPC to communicate
Advantages • Well understood • Good performance • Fast because everything is in kernel mode, and not needed to keep switching between user and kernel modes	Advantages • Kernel is generally more robust and more extensible • Better isolation and protection between kernel and high-level services
Disadvantages • Highly coupled components • Usually very complicated internal structure	Disadvantages • Lower performance

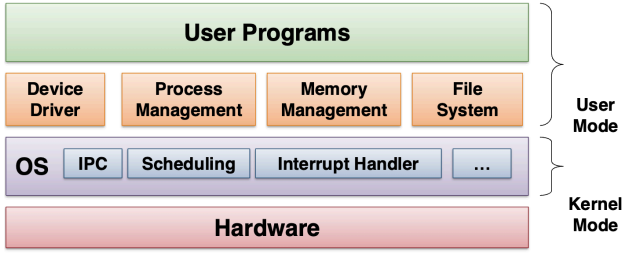
MONOLITHIC KERNEL: all components of the OS runs in kernel mode



Generic Architecture of Monolithic OS Components

- User program can interact with the OS through the **system call interface** (POSIX in Unix, win64 in Windows)
- Interaction between OS and hardware is primarily through the **interrupt handler**
- **Kernel mode**: Programs in this mode can access any memory location and any piece of hardware in the machine.
- **User mode:** Only confined to the memory allocated by OS

MICROKERNEL COMPONENTS



Generic Architecture of Microkernel OS Components

Virtual Machines

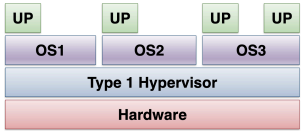
Motivation for Virtual Machines

OS assumes total control of the hardware, making it hard to run several OS on same hardware at the same time. OS is hard to debug / monitor, hard to observe working of OS, test potentially destructive implementation.

Definition of Virtual Machine

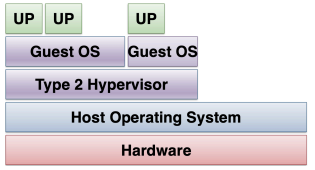
- A software emulation of hardware
- **Virtualization** of underlying hardware - Illusion of complete hardware to level above: memory, CPU, hard disk, …
- Normal (primitive) OS can then run on top of the virtual machine
- Created and managed by **Hypervisor**, aka Virtual Machine Monitor (VMM)

Type 1 Hypervisor



• Bare-metal hypervisor
• Provides individual virtual machines to guest OSes (eg IBM VM/370, VMware ESXi)

Type 2 Hypervisor



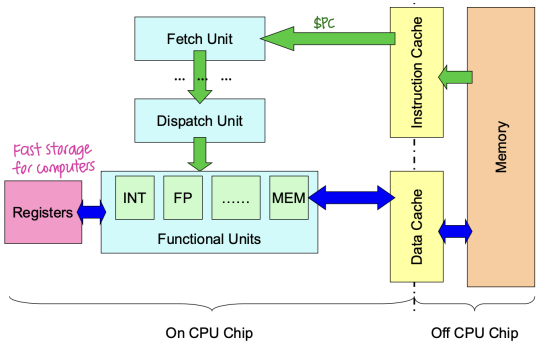
• Runs in host OS
• Guest OS runs inside Virtual Machine (eg VMware Workstation, VirtualBox, QEMU)

2. PROCESS ABSTRACTION

Switching programs requires information of both programs. Hence abstraction is needed to describe running programs, aka **process**.

- **Process / Task / Job** is a dynamic abstraction for executing program
- Info required to describe a *running (under execution) program*:
  - **MEMORY CONTEXT**
    Code/Text, Data, Function call, Dynamically Allocated Memory, **Stack**, **Heap**
  - **HARDWARE CONTEXT**
    General purpose registers, PC, **Stack Pointer, Frame Pointer**
  - **OS CONTEXT**
    Process Properties (PID, State), Resources (CPU time etc) used, **Process ID**, **Process State**

Computer Organisation (recap)



- Memory
  - Storage for instruction and data
- Cache
  - Duplicate part of the memory for faster access
  - Instruction cache and Data cache
- Fetch unit
  - Loads instruction from memory
  - Location indicated by Program Counter (PC) register
- Functional units
  - Carry out instruction execution
  - Dedicated to different instruction type
- Registers
  - Internal storage for the fastest access speed
  - General Purpose Register (GPR) - accessible by user program / compiler
  - Special Register - PC, Stack Pointer, Frame Pointer, PSW

- Executable (binary) has 2 major components: Instructions and Data
- When program is under execution, there are more info (memory, hardware, OS context)

Basic Instruction Execution

- Instruction X is fetched .
  - Memory location indicated by PC.
- Instruction X dispatched to corresponding Functional Unit.
  - Read operands if applicable (usually from memory / GPR).
  - Result computed.
  - Write value if applicable (usually to memory / GPR).
- Instruction X completed. PC updated for next instruction.

2.1 Memory Context for FUNCTION CALL

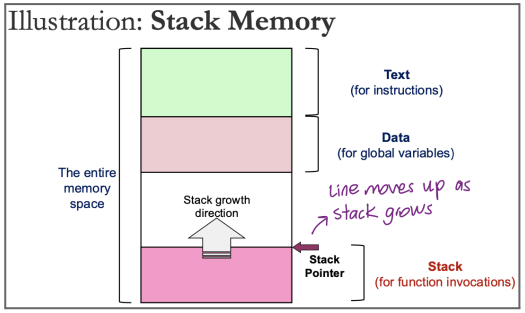
Memory Context Challenges of Functional Calls

- Control Flow issues: Need to jump to function body, resume when function call is done, store PC of caller
- Data Storage issues: Need to pass parameters to function, capture return result, may need to declare local variables
- Need a new region of memory dynamically used by function invocations

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

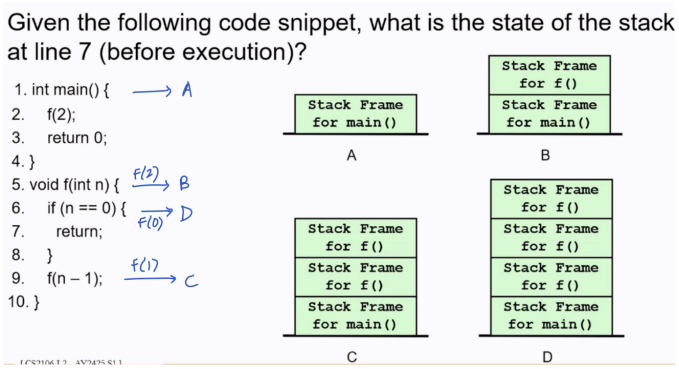
Stack Memory Region

- Memory region to store information for function invocation



Stack Frame (dynamic):

- Describes information of function invocation.
- Contains
  - **Return address of caller**
  - **Arguments (parameters) for the function**
  - **Storage for local variables**
  - **Frame Pointer**
  - **Saved registers**
- Added on top when function invoked → Stack "grows"
  - Push stack frame; Stack pointer address decreases
- Removed from top when function call ends → Stack "shrinks"
  - Pop stack frame; Stack pointer address increases



main → f(2) → f(1) → f(0), return 0. Ans: D.

Stack Pointer (dynamic):

- Top of stack region (first unused memory location)
  - The SP usually points to the start of free space on the stack or to the last item on the stack.
- Most CPU has a specialised register for this purpose
- A function invocation that uses the stack can operate solely using the SP (i.e., without using the FP at all).
  - Frame pointer, where it exists, is only there for the convenience of the compiler.

Frame Pointer:

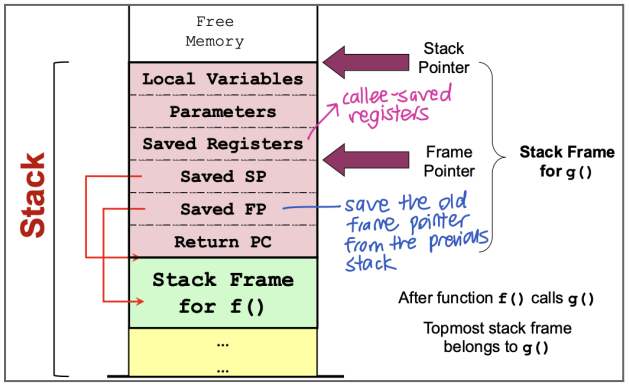
- To facilitate the access of various stack frame items. As Stack Pointer is hard to use as it can change, some processors provide a dedicated register **Frame Pointer**.
- Points to a fixed location in a stack frame
- Other items are accessed as a displacement from the Frame Pointer
- The usage of FP is platform / ISA dependent

Saved Registers:

- The number of general purpose register (GPR) on most processors are very limited (e.g., MIPS has 32 GPRs, x86 has 16 GPRs)
- When GPRs are exhausted:
  - Use memory to temporary hold the GPR value
  - GPR can then be reused for other purposes
  - GPR value can be restored afterwards
  - Known as **register spilling**

- Similarly, a function can spill the registers it intend to use before the function starts (callee-saved)
  - Restore those registers at the end of function

Stack Frame Illustration



What can be found in the **stack frame created** when f() calls g():

- [x] **Local variable that g() uses**
- [x] **General purpose registers that g() modifies**
  - g()'s responsibility to save (and later restore) any registers that it modifies, so that f() is not affected.
- [ ] Local variable that f() uses
- [ ] General purpose registers that g() modifies
- [ ] Address where f() begins
- [ ] Address where g() begins

Function Call Convention:

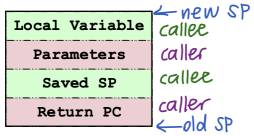
Stack Frame Setup / Teardown

Different ways to set up stack frame

- Info stored in stack frame vs registers; Portion of stack frame prepared, cleared by caller / callee; Caller or callee adjust the stack pointer
- Dependent on hardware and programming language

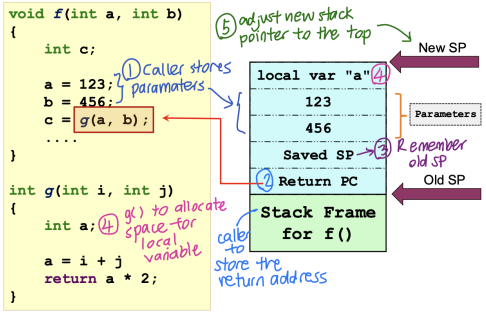
Stack Frame Setup

Prepare to make a function call:



- **CALLER**: Pass parameters (arguments) using registers and/or stack
- **CALLER**: Save Return PC on stack
- **Transfer Control from Caller to Callee**
- **CALLEE**: Save registers used by callee. Save the old Frame Pointer (FP), Stack Pointer (SP)
- **CALLEE**: Allocate space for local variables of callee on stack
- **CALLEE**: Adjust SP to point to new stack top; Adjust FP

Example: Calling function g()

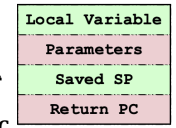


The **CALLEE saves registers it intends to use onto the stack**, and **restores** them after that. **If not**, the callee does not know what registers the caller is using, and may accidentally change the contents of the register that the caller was using, resulting in incorrect execution.

Can do this for main, but not necessarily since main is likely invoked by the OS. The **OS would have saved registers** that it (or other processes) needed during **context switching.**

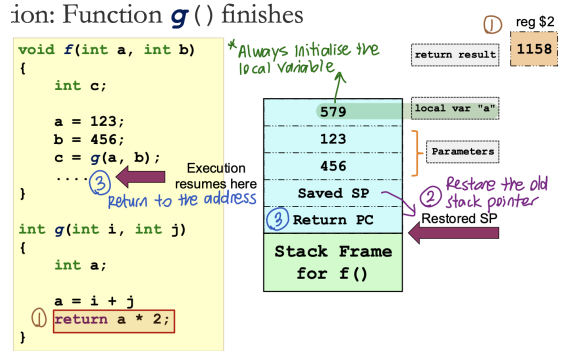
Stack Frame Teardown

On returning the function call:



- **CALLEE**: Place return result in register (if applicable)
- **CALLEE**: Restore saved registers, Frame Pointer, Stack Pointer
- **Transfer control back to caller using saved PC**
- **CALLER**: Utilize return result (if applicable)
- **CALLER**: Continues execution in caller

Example: Function g() finishes



and $v0. You may find it easier if you save $sp to $fp, then used $fp to access the arguments.

```
f:    lw $t0, 0($fp)        ; Get first parameter
      lw $t1, 4($fp)        ; Get second parameter
      add $v0, $t0, $t1     ; $v0 = $t1 + $t2
      sll $v0, $v0, 1       ; $v0 = 2($t1 + $t2)
      sw $v0, 0($fp)        ; Store result
      jr $ra                ; Return to caller

main: addi $fp, $sp, 0      ; Save $sp. Note you can also do mov $fp, $sp
      addi $sp, $sp, 8      ; Reserve 2 integers on the stack for stack frame
      la $t0, a             ; Load a
      lw $t0, 0($t0)        ;
      sw $t0, 0($fp)        ; Write to stack frame
      la $t0, b             ; Load b
      lw $t0, 0($t0)
      sw $t0, 4($fp)        ; Write to stack frame
      jal f                 ; call f
      lw $t1, 0($fp)        ; Get result from stack frame
      la $t0, y             ; Store into y
      sw $t1, 0($t0)
      addi $sp, $sp, -8     ; Pop off stack frame
      li $v0, 10            ; Exit to OS
      syscall
```

```
int f(int x, y) {
        return 2 * (x + y);
}

int a = 3, b = 4, y;

int main() {
        y = f(a, b);
}
```

Note to TA: Notice how we increment $sp by 8 to make space for the two arguments, then store using offsets from $fp, rather than doing this (which students are likely to do).

```
sw $t0, 0($sp)
addi $sp, $sp, 4
sw $t0, 0($sp)
addi $sp, $sp, 4
```

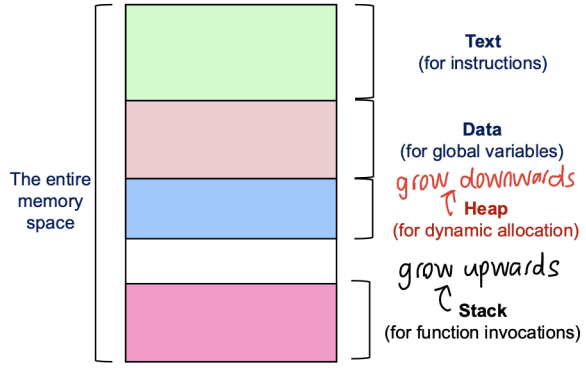
more efficient, saves the number of adds to $sp.

2.2 Memory Context for DYNAMICALLY ALLOCATED MEMORY (HEAP memory region)

Dynamically allocated: Acquire memory space during **execution time**

- In C: `malloc()` function call; In C++, Java: `new` keyword
- Allocated only at runtime, i.e., size is not known during compilation time
  → Cannot place in **Data** region (which is fixed, based on compile time)
- **No definite deallocation timing**, e.g., can be explicitly freed by programmer in C/C++, can be implicitly freed by garbage collector in Java
  → Cannot place in **Stack** region (where stack frame is deallocated automatically once function returns)
- Solution: Setup a separate **heap memory region**

HEAP MEMORY




```
int fun1(int x, int y) {
        int z = x + y;
        return 2 * (z – 3);
}

int c;

int main() {
   int *a = NULL, b = 5;
   a = (int *) malloc(sizeof(int));
   *a = 3;
   c = fun1(*a, b);
}
```

Item	Where it is stored/created
a	Stack
*a	Heap
b	Stack
c	Data memory
x	Stack
y	Stack
z	Stack
fun1's return result	Stack
main's code	Text
Code for f	Text

Managing Heap Memory

- Heap memory is a lot trickier to manage due to its:
  Variable size; and Variable allocation / deallocation timing
- Heap memory allocated /deallocated in a way that creates "holes" in the memory (since there is no specific deallocation timing). Free memory block squeezed in between of occupied memory block

```
int f1(int a){
   int x=7;
   return a+x;
}


int main(){
   int i=0, j=0;

   int *p= malloc(10);

   while(i<100){
       j = j + f1(i);
       i++;
   }

   free(p);

   return 0;
}
```

- Variable x is allocated on the stack.
- Variable x will be allocated 100 times. Every invocation of f1() will allocate x.
- Variable x will be allocated at the same address every time.
  When invoked f, push a frame, allocate the space just nicely, pop frame, repeat.
  Even if main called another function, let's say g(), but before it invokes the f, because it popped g's frame before it calls f and goes into the loop.

- Local variable *p* is allocated on the **stack**.
- p is of type (int *). p is a pointer, but the **address / memory location that p points to** is in the **heap** segment.

2.3 OS Context: Process Id (PID), Process State

Process Identification

- Process ID (PID) to distinguish processes from each other
  - A number unique among processes
- OS dependent issues: If PIDs reused; if it limits the maximum number of processes; if there are reserved PIDs

Process State

- Indicate execution status (Running / Not running / Ready to run)

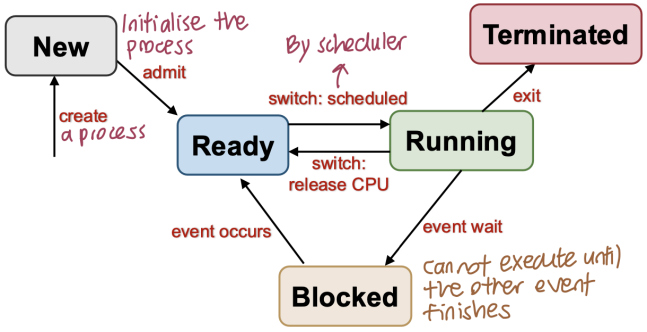
Global view of Process States:

- Given n processes,
  - With 1 CPU (core): ≤ **1** process in **running** state, conceptually 1 transition at a time. **Any** number of processes can be in **blocked** state.
  - With m CPUs (cores): ≤ **m** process in **running** state, possibly parallel transitions
- Different processes may be in different states, each process may be in different part of its state diagram
- Assumption in CS2106: Our CPU has 1 core!

Process Model

- Set of states and transitions; Describes the behaviors of a process

Generic 5-Stage Process Model



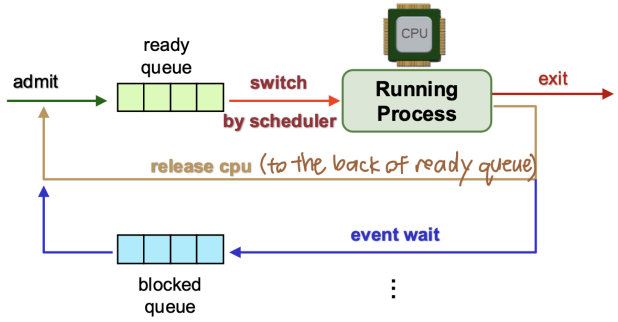
Process States for 5-Stage Model

- 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 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: Acquiring lock, waiting for I/O…
- Event occurs (Blocked → Ready)
  - Event occurs → Process can continue

Queuing Model of 5 state transition



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

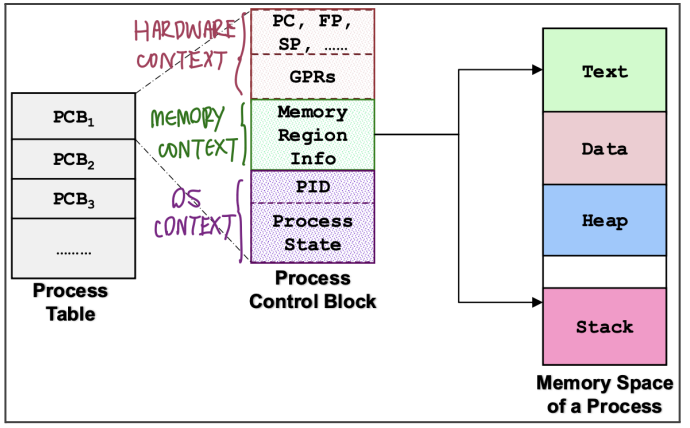
2.4 Process from OS perspective: PROCESS TABLE & PROCESS CONTROL BLOCK

Shell executes as a process, hence it would have a PCB.

Process Control Block (PCB) or Process Table Entry

- The entire execution context for a process
- Kernel maintains PCB for all processes
  - Conceptually stored as one table representing all processes
- Issues
  - Scalability: How many concurrent processes can you have?
  - Efficiency: Should provide efficient access with minimum space wastage

Process Table



2.5 Process Interaction with OS: SYSTEM CALLS

Application Program Interface (API) to OS

- Provides way of calling facilities/services in kernel
- **NOT** the same as normal function call
  - Have to change from *user* mode to ***kernel*** mode (higher privilege, access to hardware)

Different OS have different APIs:

- Unix Variants: Most follows POSIX standards (relatively stable); Small number of calls: ~100
- Windows Family: Uses WinAPI across different Windows versions; New version of windows usually adds more calls; Huge number of calls:~1000

Unix System Calls in C/C++ program

In C/C++ program, system call can be invoked almost directly

- Majority of the system calls have a library version with the **same name** and the same parameters
  - The library version act as a **function wrapper**
- A few library functions present a more user friendly version to the programmer eg., lesser number of parameters, more flexible parameter values…
  - The library version acts as a function adapter

Example

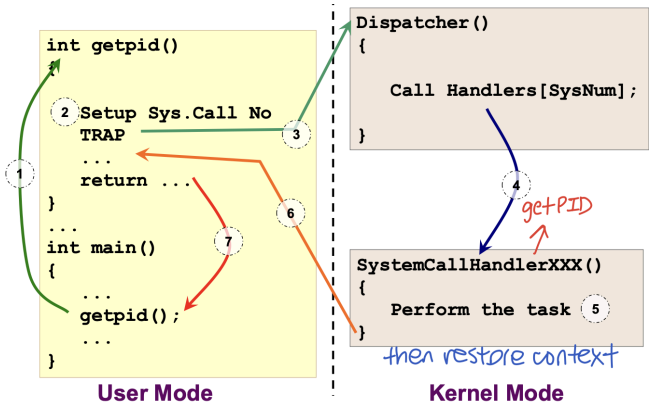
```
#include <unistd.h>
#include <stdio.h>

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
- System call returns a process id

General System Call Mechanism



1. User program invokes the library call
   - Using the normal function call mechanism
2. Library call (usually in assembly code) places the **system call number** in a designated location.g., a register
3. Library call executes a special **instruction (TRAP)** to save the user program's context, and switch from **user mode to kernel mode**
   - Saves CPU state
   - TRAP causes the CPU to hand control over to a fixed memory location, which has the OS dispatcher.
4. In **kernel** mode, the appropriate system call handler is determined using the **system call number** as index
   - This is usually handled by a **dispatcher**
5. System call handler is executed
   - Carry out the actual request
6. System call handler ended:
   - Restore CPU state, (context) and 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

NOT a context switch →

Context switch requires loading of the context of a new process into the register. The interrupt service routine is not a process, it is a service routine. It restores the context of the interrupted process, not a new process.

2.6 Process Interaction with OS: EXCEPTION & INTERRUPT

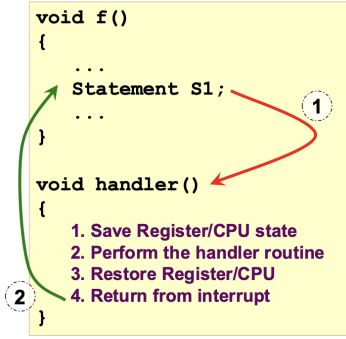
Exception (synchronous)

- Executing a **machine level instruction** can cause exceptions, e.g. Arithmetic Errors (Overflow, Division by Zero), Memory Accessing Errors (Illegal memory address, Misaligned memory access)...
- Exception is **Synchronous** - occur due to program execution
- False that exceptions can happen at any moment during the execution of a program.
- Effect of exception:
  - Have to execute an **exception handler**
  - Similar to a **forced function call**

Interrupt (asynchronous)

- External events can interrupt the execution of a program
- Usually hardware related, e.g.Timer, mouse movement, keyboard pressed (`ctrl + Z` to suspend, `fg` to continue, `ctrl + C` to terminate).
- Interrupt is **asynchronous**
  - Events that occurs **independent** of program execution
- Effect of interrupt:
  - Program execution is suspended
  - Have to execute an **interrupt handler** (executed immediately)

Exception/Interrupt Handler: Illustration



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

2.7 Process Abstraction in Unix

- **PROCESS IDENTIFICATION**
  - PID: Process ID (an integer value)
- **PROCESS INFORMATION →** `getpid()`, `getppid`
  - Process State
    - Running, Sleeping, Stopped, ***Zombie***
  - Parent PID
    - PID of parent process
  - Cumulative CPU time
    - Total amount of CPU time used so far
  - Unix Command: `ps` (process status)
- **PROCESS CREATION →** `fork()` ❗
- **PROCESS TERMINATION →** `exit()`
- **PARENT-CHILD SYNCHRONISATION →** `wait()` ❗
  - Get exit status, synchronize with child

Another Unix Process System call: `exec()`
- Change executing image / program

Note: Common Line Argument in C

- Can pass arguments to a program in C
  - `argc`: Number of command line arguments, including program name itself
  - `argv`: A char strings array
    - Each element in `argv[]` is a C character string
  - Eg: `a.exe 1 2 3 hello`
    - `argc`: 5
    - `argv[]`: `[a.exe, 1, 2, 3, hello]`

```
int main( int argc, char* argv[] )
{   int i;
    for (i = 0; i < argc; i++){
        printf("Arg %i: %s\n",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 in Unix: `fork()` ❗

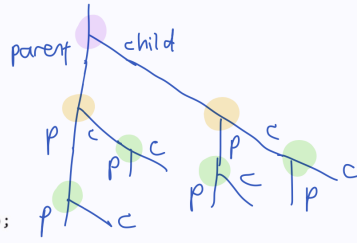
`fork()` involves a **system call**

Header File	Syntax
`#include <sys/types.h>` `#include <unistd.h>`	`int fork();`

- Use the ***RETURN VALUE*** of `fork()` to distinguish parent and child:
  - **PID** of the **newly created child process** (for **parent** process) OR
  - **0** (for **child** process)
- Behavior
  - Creates a new process (**child** process)
  - Child process is a **duplicate** of the current executable image
    - i.e., same code, same address space, etc.
    - Memory in child is a COPY of the parent (i.e., not shared)
      - Same memory state, variables, and register content. Except return value.
      - Implemented using copy-on-write

    ⇒ Parent and child can operate independently of each other.
  - Child DIFFERS (some OS state are different) only in:
    - Process id (PID)
    - Parent (PPID)
      - Parent = The process which executed the `fork()`
    - `fork()` return value
- Output
  - **PC register** will be the **same** after returning
  - Both parent and child resume execution after the point `fork()`
  - Indeterminate if parent or child runs first
    - Order of printing is not deterministic
  - Common usage: Use the parent/child process differently eg
    - Parent spawn off a child to carry out some work
    - And then the parent is ready to take another task

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

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

    fork();

    fork();

    printf("I am seeing DOUBLE\n");
    return 0;
}
```



Program prints "I am seeing DOUBLE" 8 times

```
int result;

result = fork();          → Not the child.
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 | Child Process

`fork()`: Independent Memory Space

```
int var = 1234;
int result;

result = fork();
if (result != 0){
    printf("Parent: Var is %i\n", var);   1234
    var++;
    printf("Parent: Var is %i\n", var);   1235
} else {
    printf("Child: Var is %i\n", var);    1234
    var--;
    printf("Child: Var is %i\n", var);    1233
}
```

- Child is a COPY of the parent, NOT a reference to
- Processes have independent memory space, i.e. updates (var++) do not impact each other's memory space.

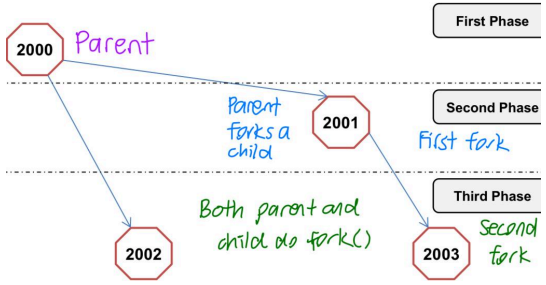
- Note `clone()`:
  `fork()` not versatile (a thorough duplication of the parent process), for scenarios where partial duplication preferred (eg parent and child shares some memory regions or other resources),
  ⇒ `clone()` supersedes `fork()`
- `fork()` itself is not useful
  - Still need to provide the full code for the child process
  - What if we want to execute another existing program instead?

`fork()`: Ordering and Process Tree

```
// First phase
printf ("1");


// Second phase
childPID=fork();

printf("2*");
printf("2#");


// Insertion
point


// Third phase
childPID=fork();

printf("3*");
printf("3#");

return 0;
```

- **Sequential ordering** in **SAME** process: `*` and `#` from the same process can never change place.
- Messages from the **same process** will follow the phases: `2*` , `2#` before `3*` , `3#`
- Message from the first phase (only one) must precede all other messages as there is only one process executing at that time.
- **Parent and child are scheduled independently.** No fixed execution order between them. Messages from child process need not always precede / come after the direct parent's message.
- **Messages from the same phase need not precede messages from next phase**. Parent process can execute to the end i.e. printing all 3 phases before any of the forked processes have a chance to execute.

Process tree showing different PIDs of parent and child processes:



Sleep code in insertion point:

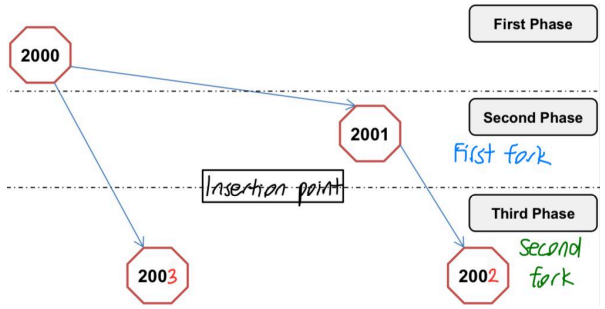
```
if (childPID == 0) {          // child process 2001
     sleep(5);                // sleep for 5 seconds
}
```

Pause the first child process, 2001, for 5 seconds. Assuming process 2002 takes <5 seconds to create and run, it is likely that both processes 2000 and 2002 will finish execution before 2001 and 2003.

Wait code in insertion point:

```
if (childPID != 0) {    // parent process 2000
     wait(NULL);        // NULL: Don't care about return result
```

```
}
```



1. Code pauses parent process 2000 after printing `2*` , `2#` .
2. Process 2001 carry on to spawn child process 2002.
3. Process 2001 continue to print `2*` , `2#` and exits.
4. Once 2001 exits, then 2000 also exits, and 2000 can continue to spawn 2003.
5. Processes 2000 and 2003 will print their `3*` , `3#` messages independently.
6. Note process 2002 will continue printing `3*` , `3#` messages independently of steps 4 and 5.

Process Replacement: `execl()` System Call

- `execl, execv, execve, execle, execvp`
- **Replace** current executing process image with a new one
  - **Code replacement**, new process.
  - **PID and other information still intact**

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

Example:

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

- path = "`/bin/ls`"
  - "dir" command in unix, to list the files in directory
- arg0 = "`ls`"
  - Program name
- arg1 = "`-al`"
- Same as executing: `ls -al`

```
int main() {
  printf ("I am here!\n"); // printed
  execl("/bin/ls", "ls", "-al", NULL); // execl()
  printf ("Am I still here?\n"); // not printed
```

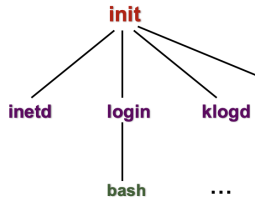
Combining `fork()` and `exec()`

- Can spawn off a child process
  - Let the child process perform a task through `exec()`
- Meanwhile, the parent process is still around
  - To accept another request

⇒ For Unix to get a new process for running a new program

The Master Process `init` process (no parent)

- Created in kernel at boot up time
- PID = 1 (no other processes with PID = 1)
- Watches for other processes and respawns where needed
- `fork()` creates process tree VS `init` is the root process



Process Tree example:

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

Process Termination in Unix: `exit()`

Header File	Syntax
`#include <stdlib.h>`	`void exit( int status );`

- Status (`echo $?` to get exit status) is returned to the parent process
- Unix convention
  - **0** = **Normal Termination** (successful execution)
  - !0 = To indicate problematic execution
- Function does not return

Process finished execution

- Most system resources used by process are released on exit, eg file descriptors
- Some basic process resources not releasable (process is now a **ZOMBIE**)
  - **PID** and **status** needed for **parent-children synchronisation**
  - Process accounting info eg cpu time
  - Remaining resources are later cleaned up by `wait()` system call by parent
- Process table entry may still be needed

Implicit `exit()`

- Most programs have no explicit `exit()` call

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

- Return from main() implicitly calls `exit()`
  - *If there is a return value in main() (eg return 10;), then the exit status would be the return value from main() (ie 10)*
- Open files also get flushed automatically!
  - Open files also get flushed automatically!

❗ Parent/Child Synchronization in Unix: `wait` ❗

- **Parent** process (**blocked**) `wait` for child process to terminate (exit)

Header File	Syntax
`#include <sys/types.h>` `#include <sys/wait.h>`	`int wait( int *status );`

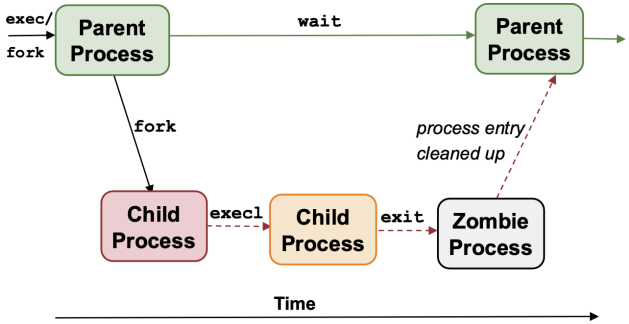
- Returns: the **PID** of the terminated **child** process
- Status (*passed by address*):
  - Stores the exit status of the terminated child process
  - Use NULL if you do not need/want this info
  - Terminated child returns status number to parent in `*status`
- `wait(*result)` stores the child's exit status in `result`
- Number of `wait()` system calls
  - One `wait()` → If any child exits, the `wait()` returns
  - Call `wait()` twice to wait for both children to exit

Behaviour:

- Call is blocking:
  - Parent process blocks until at least one child terminates
- Call cleans up **remainder** of child system resources
  - Those not removed on `exit()`
  - **Kill zombie process**

- `waitpid()`: Wait for a specific child process
- `waitid()`: Wait for any child process to **change status**

Process Interaction in Unix



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

Zombie Processes (2 cases)

`exit()` "creates" zombies:

- On process `exit`:
  - Most resources are released → becomes **ZOMBIE**
  - **Cannot delete** all process info
    - If parent asks for the info in a `wait()` call, the remainder of process data structure can be cleaned up only when `wait()` happens
  - **Cannot kill** zombie. The process is already dead.

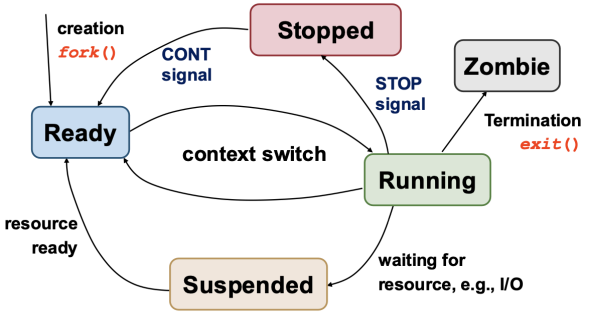
1. **ORPHAN Process: Parent process terminates BEFORE child process**

- `init` process becomes "pseudo" parent of child processes
- Child termination (by calling `exit()` and becoming a zombie process) sends signal to `init`, which utilizes `wait()` to cleanup the resources and kill the zombie process

2. **ZOMBIE Process: Child process terminates before parent but parent did not call `wait`**

- Child process become a zombie process
- Can fill up process table; May need a reboot to clear the table on older Unix implementations

Process State Diagram in Unix

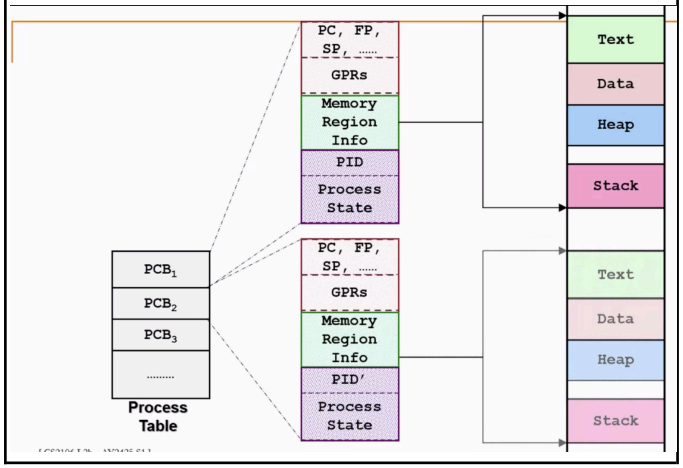


2.8 Implementation Issues

Implementing `fork()`

Behaviour of `fork()`
- Makes an almost exact copy of parent process

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
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 Operation (very expensive)
- Potentially need to copy the whole memory space
- The child process will not access the whole memory range right away
- Additionally:
  - If child just read from a location, remains unchanged
    - Can use a shared version
- ***Only when write is perform on a location***:
  - Then two independent **copies** are needed

Eg

```
int a = 5, b = 10;
if  (fork() == 0) {     // child
    printf("%d", a);
    b++;
} else {
    printf("%d, %d", a, b);
}
```

Only need to copy the memory address of b , not a, to the child. Value of b will be modified by the child. Child only needs reference to a, no need to modify it, so no need to copy memory address of a.

COPY ON WRITE to optimise Memory Copy Operation

- Only duplicate a "memory location" when it is written to, otherwise parent and child share the same "memory location"

Note:
- Memory is organized into memory pages
  - A consecutive range of memory locations
- Memory is managed on a page level, instead of individual location

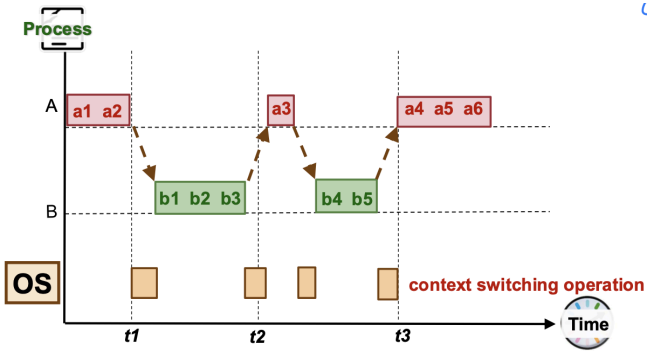
3. PROCESS SCHEDULING

Concurrent Execution

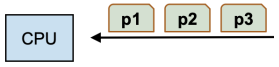
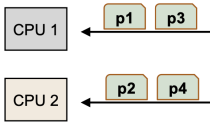
Concurrent processes: Multiple processes **progress** in execution (at the same time)

- Virtual parallelism: Illusion of parallelism (pseudo-parallelism); 1 CPU
- Physical parallelism: E.g., Multiple CPUs / Multi-Core CPU to allow parallel execution of multiple processes

Interleaved Execution (Timeslicing)



- Concurrent execution on 1 CPU (core): Interleave instructions from both processes
- **Context switch** (hardware, memory, OS):
  - Multitasking needs to change context between programs
    - OS incurs overhead in switching processes

Multitasking OS	
1 core (CPU) Timesliced execution of tasks	**Multiprocessor** Timeslicing on ***n*** CPUs
	

Scheduling a Process

1. Scheduler is triggered (OS takes over)
2. If context switch is needed:
   • Context of current running process is saved and placed on blocked queue / ready queue
   • Old register values saved to process' PCB before context switching to another process; remember hardware context.
3. Pick a suitable process P to run based on scheduling algorithm
4. Setup context for P
5. Let process P run

Scheduling in OS

- **Scheduling problem** with having multiple processes: If ready-to-run process is more than available CPUs, which should be chosen to run?
- **Scheduler**: Part of the OS that makes scheduling decision
- **Scheduling Algorithm**: Algorithm used by scheduler to allocate

Process Behaviour

- Process' unique **requirement of CPU time**
- A typical process goes through phases of:

CPU-Activity	IO-Activity
• Computation, eg number crunching • **Compute-Bound Process** spends majority of its time here, consumes a lot of CPU time	• Requesting and receiving service from I/O devices E.g., print to screen, read from file. • **IO-Bound Process** spends majority of its time here, consumes little CPU time, spend time waiting for I/O devices

* For a process to **access I/O** devices or other system level events, the process needs to make a **SYSTEM CALL** i.e. OS will be notified, hence it is possible to **let OS intercept** those events and **call the scheduler directly** from the system call routines.

Processing Environment

- **Batch Processing**: No user interaction required, no need to be responsive
- **Interactive** (Multiprogramming): Active user interacting with the system. Should be responsive: low and consistent in response time.
- **Real time processing**: Deadline to meet, usually periodic process.

3.1 Criteria for Scheduling Algorithms

Many criteria, largely influenced by processing environment

Criteria for **all processing environments**:
- **Fairness**: Fair share of CPU time (per process / user). **No starvation** (all ready processes have the chance to run)
- **Utilisation**: All parts of computing system should be utilised.

Criteria for **BATCH PROCESSING**:
- **Turnaround time** (lower → better)
  - Total time taken (from time of submitting job to finishing job)
  - Includes waiting time (from time of submitting job to execution of job): Time spent waiting for CPU
  - Waiting time = Turnaround time - Process time
- **Throughput** (higher → better)
  - Number of tasks finished per unit time i.e. rate of completing tasks
- **CPU Utilisation**
  - Percentage of time when CPU is working on a task

Criteria for **INTERACTIVE SYSTEMS**:
- **Response time**: Time between request and response by system
- **Predictability**: Less variation in response time → More predictable

Criteria for REAL-TIME SYSTEMS:
- Meeting deadlines Avoid losing data (eg livestream)
- Predictability: Avoid quality degradation in multimedia

When to perform scheduling

- **NON-PREEMPTIVE (Cooperative)**
  - A process stayed scheduled (in running state) until it
    - blocks (performs IO, wait() call…); or
    - gives up the CPU voluntarily (yield(), tell system that no OS is needed)
  - Can cause starvation
- **PREEMPTIVE**
  - Process given a **fixed time quota** to run (Possible to block or give up early)
  - At the end of the time quota, the running process is suspended. Another ready process gets picked if available

3.2 Scheduling for Batch Processing

Processing environment
- No user interaction. **Non-preemptive** scheduling is predominant.

Scheduling Algorithms for Batch Processing

Generally easier to understand and implement; Commonly resulted in variants/improvements that can be used for other types of systems.

First-Come First Served (FCFS)

- Tasks stored in **FIFO queue based on arrival time**
- Pick first task in the queue to run until task is done or blocked.
- Blocked task removed from FIFO queue
  - When it is ready again, it is placed at the back of the queue like a newly arrived task
- Guaranteed **no starvation** (Every task eventually processed)
  - Number of tasks in front of task X in FIFO is always decreasing
  - Assuming that the running time of each process is finite, task X will get its chance eventually
  - **Only Round Robin and FCFS have bounded waiting time → Prevent starvation** by ensuring every process in the ready queue gets to run eventually
- When FCFS allows **shortest possible average response time**:
  - Jobs arrive in ready queue in order of increasing job lengths; avoids short jobs arriving later from waiting substantially for an earlier longer job
  - All jobs have same completion time

Shortcomings
- Simple reordering can reduce the average waiting time
- **Convoy effect (CPU-Bound then IO-Bound tasks → Inefficient)**
  - First task (CPU-bound) is followed by IO-Bound tasks
  - CPU-Bound task runs while all IO-Bound tasks (idling) wait in ready queue
  - CPU-Bound task blocked on I/O → All IO-Bound tasks execute quickly and blocked on I/O (CPU idling)

Shortest Job First (SJF)

- Select task with **smallest total CPU time**
- Need to know total CPU time for a task in advance
  - Can 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
- α = Weight placed on recent event or past history
- $Predicted_{n+1}$ = Latest prediction

- Given a fixed set of tasks, **minimizes average waiting time**
- **SHORTCOMING - Starvation is possible:** Biased towards short jobs, Long jobs may never get a chance.
- Optimal only when all jobs available simultaneously

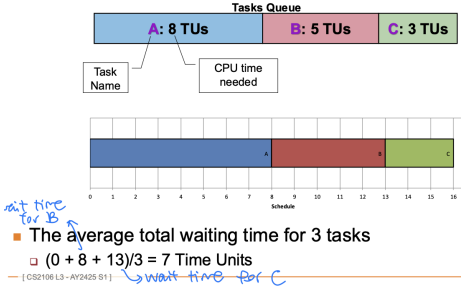
Shortest Remaining Time (SRT)

- **Preemptive** version of SJF
- Select job with **shortest remaining / expected time**
- When a new job arrives, the total time is compared to the current process' remaining / expected time. New job with shorter remaining time can preempt the currently running job. Provide good service for a short job even when it arrives late.

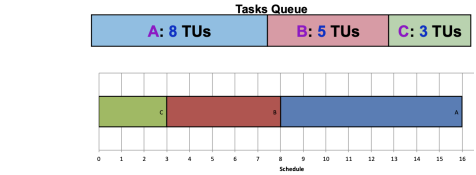
- If all tasks arrive at the beginning, SRT will give the same schedule as SJF.
- **SHORTCOMING - Starvation is possible**

Batch System Scheduling Examples

First-Come First-Served: Illustration

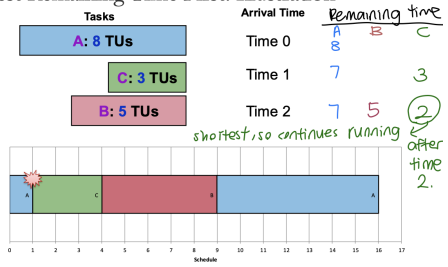


Shortest Job First: Illustration



- The average total waiting time for 3 tasks
  - (0 + 3 + 8)/3 = **3.66** Time Units

Shortest Remaining Time First: Illustration



3.3 Scheduling for Interactive Systems

Processing environment
- User interaction. **Preemptive** scheduling algorithms are used to ensure good response time. Scheduler needs to run **periodically**.

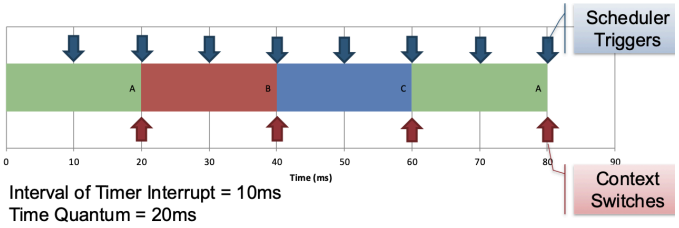
Every **ITI** –*invoke*→ **Timer Interrupt** -*invoke*→ **Scheduler** → Scheduling

Ensuring periodic scheduler using **Timer Interrupt**
- **Timer interrupt** goes off periodically (based on hardware clock).
- OS ensures timer interrupt cannot be intercepted by any other program ➔ Timer interrupt handler **invokes scheduler**

Interval of Timer Interrupt (ITI)
- Timing interval for every timer interrupt, OS scheduler is invoked.
- Typical values (1ms to 10ms)
- Checks if process is done with the time quantum

Time Quantum
- Execution duration given to a process
- Could be constant or variable among the processes
- Must be **multiples of interval of timer interrupt**
- Large range of values (commonly 5ms to 100ms)

ITI (Interval of Timer Interrupt) vs Time Quantum



Interval of Timer Interrupt = 10ms
Time Quantum = 20ms

Scheduling Algorithms for Interactive Systems

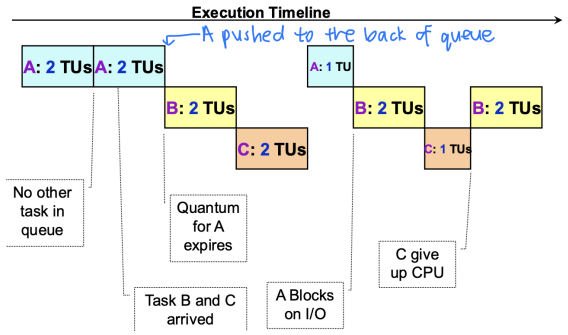
Round Robin (RR)

- Preemptive version of FCFS
  - Behave identically to FCFS if job lengths shorter than time quantum
- Tasks stored in **FIFO** queue. Each process assigned a **time quantum** / slice.
- If task is still running at end of **time quantum** / blocked / gives up CPU voluntarily, the CPU is preempted, given to another process.
- Task placed at the end of the queue to wait for another turn.
  - Blocked task moved to other queue to wait for requested resource. When a blocked task is ready again, it is placed at the end of the queue.

- Guaranteed **no starvation**
- **Response time guarantee**
  - Given ***n tasks*** and ***quantum q***, the time before a task get CPU is bounded by ***(n-1)q***
- **Timer interrupt** needed: For scheduler to check on quantum expiry
- **Choice of time quantum duration**
  - Big quantum: Better CPU utilization (less context switch → less overhead → more efficient) but longer waiting time
  - **Small quantum: Bigger overhead** (worse CPU utilization; more CPU time spent by OS for context switching → reduce throughput, increase turnaround time) but **shorter waiting time**

Cases where RR performs poorly than FCFS (FIFO):

- Job lengths all the same and much greater than time quantum: RR performs **poorly in average turnaround time**
- Many jobs and job lengths exceed time quantum: RR results in **reduced throughput** due to **greater overhead** from the OS incurred due to **context switches** when jobs are **preempted**

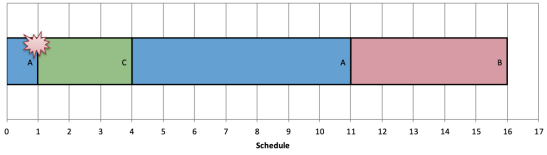


Priority Based Scheduling

- Assign a **priority value** to all tasks. Runnable task with the highest **priority** is allowed to run.
  - **Preemptive** version: Higher priority process can preempt running process with lower priority
  - **Non-preemptive** version: Late coming high priority process has to wait for next round of scheduling

Tasks	Arrival Time	Priority (1=highest)
A: 8 TUs	Time 0	3
C: 3 TUs	Time 1	1
B: 5 TUs	Time 1	5

lowest priority



- **Shortcoming - Low priority process can STARVE**
  - High priority process keep hogging the CPU, worse in preemptive variant
  - Solutions
    - Decrease the priority of currently running process after every time quantum → Eventually dropped below the next highest priority
    - Give the current running process a time quantum. This process is not considered in the next round of scheduling
  - Generally, it is hard to guarantee or control the exact amount of CPU time given to a process using priority

- **Priority Inversion: Lower priority task preempts higher priority task**
  - If lower priority task that locks some resource eg file, gets switched, the higher priority task cannot run
  - Eg Priority: {A = 1, B = 3, C = 5} (1 is highest). Task C starts and locks a resource (e.g., file). Task B preempts C. C is unable to unlock the resource. Task A arrives and needs the same resource as C, but the resource is locked. Task B continues execution even if Task A has higher priority.

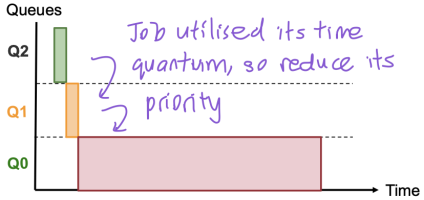
Multi-Level Feedback Queue (MLFQ)

- More efficient to give CPU-bound processes large quantum once in a while, rather than small quanta frequently to reduce swapping. Also, giving all processes large quantum means poor response time. Solution is to set **multiple priority queues**.
- As (CPU-bound) process sinks deeper into priority queues, run less frequently, saving CPU for short, interactive processes.
- Adaptively learn process behaviour automatically, minimising:
  - Response time for IO bound processes
  - Turnaround time for CPU bound processes
- **SHORTCOMING - Starvation is possible**

MLFQ Rules

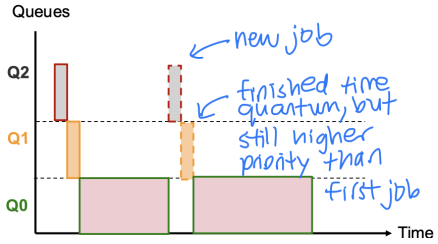
- Basic rules
  - If Priority(A) > Priority(B) ➔ A runs
  - If Priority(A) == Priority(B) ➔ A and B runs in RR
- Priority Setting/Changing rules:
  - New job given highest priority.
  - If a job utilized its time quantum ➔ Priority reduced
  - If a job gives up / blocks before finishes its time quantum ➔ Priority retained

Eg 1. 3 Queues: Q2 (highest priority), Q1, Q0. A single long running job



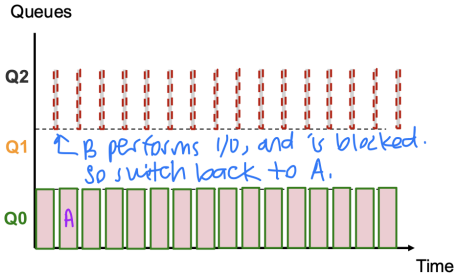
- A process with a **lengthy CPU phase followed by I/O-intensive** phase: The process can sink to the **lowest priority during the CPU intensive** phase. With low priority, the process may not receive CPU time in a timely fashion during the I/O phase, which **degrades responsiveness**.

Eg 2. Long running job + A short job in the middle.



Eg 3. A is CPU bound and already in the system for some time). B is I/O bound.

- I/O bound
  - New job has higher priority, can immediately preempt CPU bound processes.
  - Very short CPU time. Never use up time quantum, so remains in Priority 2.
  - Very good response time. Every time it needs CPU, it can always get scheduled as it has a higher priority.
- CPU bound: Gets preempted, but I/O bound process only uses a small amount of CPU time, so turnaround time of CPU-bound process is still good, not worse by much.



Lottery Scheduling

- Give lottery tickets to processes for various system resources eg CPU time, I/O devices
- When a scheduling decision is needed
  - A lottery ticket is chosen randomly among eligible tickets
  - The picked ticket is granted the resource
- In the long run, a process holding X% of tickets can win X% of the lottery held, and uses the resource X% of the time
- **Responsive**: A newly created process can participate in the next lottery
- Provides **good level of control**
  - A process can be given Y lottery tickets. It can then distribute to its child process
  - Important process can be given more lottery tickets → Can precisely control the proportion of usage of resource (eg can give precisely ⅓ of CPU time if give out ⅓ of the tickets)
  - Each resource can have its own set of tickets. Different proportion of usage per resource per task.
- Simple implementation

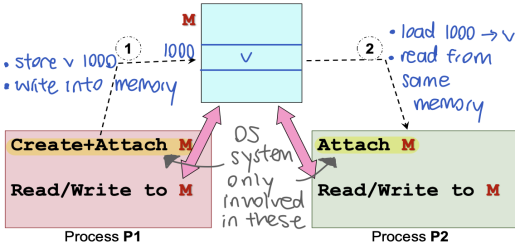
4. INTER-PROCESS COMMUNICATION (IPC)

Memory space is independent (parent and child do not have the same address) → Hard for cooperating processes to share information → IPC needed

4.1 IPC Mechanism: SHARED-MEMORY

Applicable to multiple processes sharing the same memory region.

- Process 1 ***creates*** shared memory region M ***[involves OS]***
- Process 1 and Process 2 ***attach*** memory region M to its own memory space ***[involves OS]***
- Processes 1 and 2 can now communicate using memory region M
  - M behaves very similar to normal memory region (i.e. through load and store information)
  - There is **no need to involve the Operating System to access memory**, the processes can use load and store instructions.
  - Any writes to the region are visible to the other process
  - The programmer (NOT OS) must ensure that reads and writes by multiple programs to a shared variable are done in a synchronised manner to avoid race conditions.



OS involved in steps 1 and 2 only

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

MASTER program

```
#include <stdio.h>
#include <stdlib.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 creates the shared memory region and waits 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.

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.

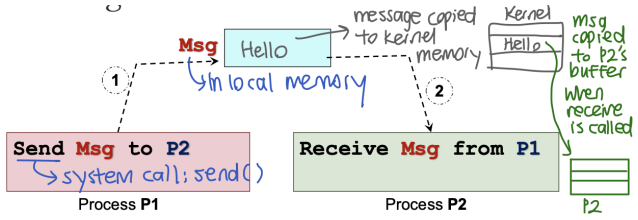
Write 3 values into shm[1 to 3]

Let master program know we are done!

Step 4. Detach Shared Memory region.

4.2 IPC Mechanism: MESSAGE PASSING

- Process 1 prepares, sends message M to process 2. Process 2 receives.
- Message sending and receiving are usually provided as system calls.
- Additional properties: Naming (how to identify other party). Synchronisation (behaviour of sending / receiving operations).



- Msg has to be stored in kernel memory space.
- **Every send/receive** operation needs to go through the **OS** (i.e., a **system call**).
- P2 calls a receive call (has a **BUFFER** - contains msg - specified here, inside local memory).

(Naming scheme) *DIRECT* COMMUNICATION

- Sender/Receiver of message explicitly names the other party
  - `Send(P2` (P2's address)`, Msg)`: Send Msg to Process P2
  - `Receive(P1` (address)`, Msg)`: Receive Msg from Process P1
- One link (communication channel) per pair of communicating processes. Need to know the identity of the other party

(Naming scheme) *INDIRECT* COMMUNICATION

- Messages sent to / received from message storage (**mailbox / port**)
  - `Send( MB, Msg )` : Send Message Msg to Mailbox MB
  - `Receive( MB, Msg )` : Receive Message Msg from Mailbox MB
- Unix: Message queue (Push msg here. Receiver pops it from queue)
- One mailbox can be *shared* among ***a number*** of processes

2 SYNCHRONISATION Behaviours

- **Blocking primitives (synchronous)**
  - Receive(): Receiver is blocked until a message has arrived
- **Non-Blocking Primitives (asynchronous)**
  - Receive(): Receiver either receive the message if available or some indication that message is not ready yet

Example:

Suppose process A is passing a message using a **non-blocking send()** to Process B, which receives it using a **blocking receive().**

- Because of the **non-blocking send**, A will not wait for the received after sending, which is why **buffering** is needed.
- **Blocking receive**: If B calls receive() before the message is sent, it will block until the message is received. However, it will not block if a message is already available.
- Send/receive goes through the OS (these are system calls).

Message Passing: Pros and Cons

Advantages:

- Portable: Can easily be implemented on different processing environments, e.g., distributed system, wide area network, etc.
- Easier synchronization e.g., when synchronous primitive is used, sender and receiver are implicitly synchronized

Disadvantages :

- Inefficient
  - Usually requires OS intervention
  - Extra copying: Go through the OS each time you want to send info to a process. When calling send() or receive(), copy local memory to kernel memory, then kernel memory to receiving memory.

4.3 IPC Mechanism: Unix PIPE (Unix-specific)

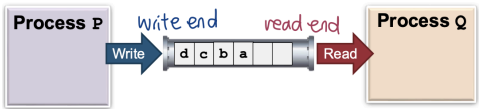
Process: 3 default communication channels in Unix

- ***stdin*** (standard in) (used by ***scanf()***)
  - Commonly linked to keyboard input
- ***stdout*** (standard out) (used by ***printf()***)
  - Commonly linked to terminal
- ***stderr*** (standard error) (used by ***perror()***)
  - Only used to print out error messages

Piping in Shell: `input channel | output channel`

- **A | B**: Output of A (instead of going to the terminal) directly goes into B as input (as if it comes from the keyboard).

Unix Pipes as an **IPC mechanism**



- A pipe can be shared between two processes
- 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
    - Normal shell pipe has 1 writer and 1 reader
  - Depends on Unix version, pipes may be **half-duplex**
    - **Unidirectional**: 1 write end and 1 read end
  - Or **full-duplex**
    - **Bidirectional**: 1 write end

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
  - **fd[0] == reading end** **fd[1] == writing end**

```
#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]);
   }
}
```


4.4 IPC Mechanism: Unix SIGNAL (Unix-specific)

- Form of inter-process communication. An **asynchronous** notification regarding an event. Sent to a process/thread
- Recipient of the signal must handle the signal by:
  - A default set of handlers OR
  - User supplied handler (only applicable to some signals)
- Eg Kill, Interrupt, Stop, Continue, Memory error, Arithmetic error …

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;
}
```

5 ❗ [Process Management] Process Alternative - THREADS ❗

Motivation for Thread

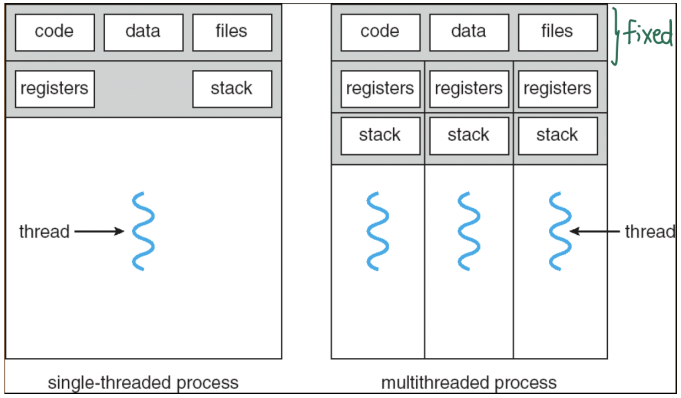
- Process is expensive
  - Process creation under `fork()`: Duplicate memory space and most of the process context etc
  - Context switch: Requires overhead, save / restore process info
- Hard for independent processes to communicate with each other
  - Independent memory space → No easy way to pass information
  - Requires **Inter-Process Communication (IPC)**

Basic idea

- Traditional process has a single thread of control: Only one instruction of the whole program is executing at any time
- Add **more threads of control** to the **same process**
  - Multiple parts of the program are executed at the same time conceptually. Same process but at different PC locations

5.1 Process and Thread

- **Multithreaded process**: A single process can have multiple threads.
- Threads in the same process share:
  - Memory Context: Text (code), Data, Heap
  - OS Context: Process id, other resources like files, etc.
- UNIQUE information needed for each thread:
  - Identification (usually **thread id**)
  - Registers (General purpose and special)
    - Registers are independent across different threads, can execute different functions
    - PC register different because executing at different location
  - "Stack"



Process Context Switch vs Thread Switch

Process Context Switch	Thread Switch
• OS context • Hardware context • Memory Context	Hardware Context • Registers • "Stack" (just changing FP & SP registers)

  - Thread is much "lighter" than process, a.k.a. **lightweight process**

Threads: Benefits

- **Economy**: Multiple threads in the same process requires much less resources to manage compared to multiple processes
- **Resource Sharing**: Threads share most of the resources (eg share the same memory) of a process. No need for additional mechanisms for passing information around.
- **Responsiveness**: Multithreaded programs can be more responsive.
- **Scalability**: Multithreaded programs can take advantage of multiple CPUs.

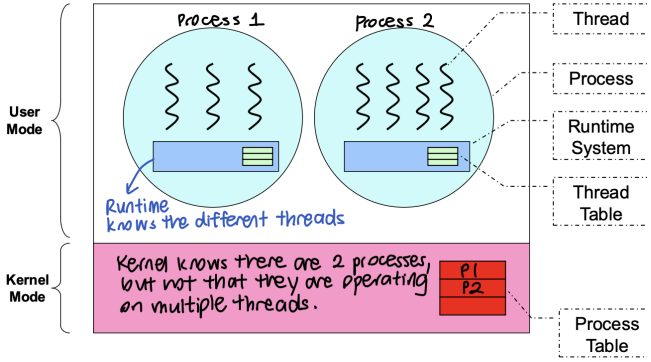
Threads: Problems

- **System Call Concurrency**:
  - Parallel execution of multiple threads → Parallel system call possible across different threads.
    - If a library call is thread-safe, multiple threads can call the library call at the same time.
    - Have to guarantee correctness and determine the correct behaviour.
- **Process Behaviour (OS Dependent)**
  - Impact on process operations
  - `fork()` duplicate process, how about threads?
  - If a single thread executes `exit()`, how about the whole process?
  - If a single thread calls `exec()`, how about other threads?

5.2 Thread Models (ways to support threads)

User Thread	Kernel Thread
• Thread is implemented as a **user library**. A runtime system (in the process) will handle thread related operations. • Kernel is ***not aware*** of the threads in the process.	• Thread is implemented in the **OS**: Thread operation is handled as ***system calls*** • Thread-level scheduling is possible: Kernel schedule by threads, instead of by process • Kernel may make use of threads for its own execution

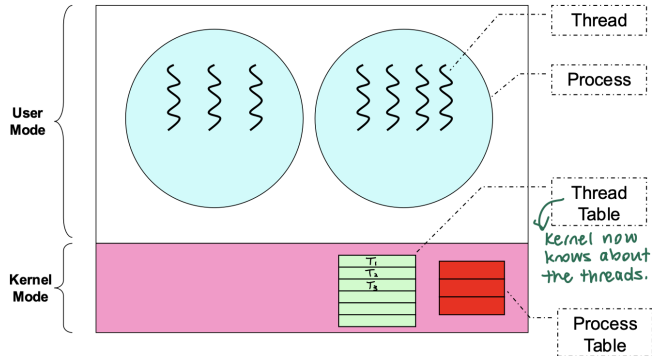
USER Thread



Advantages:
- Can have multithreaded program on ANY OS
- Thread operations are just library calls
- Generally more configurable and flexible e.g., customized thread scheduling policy

Disadvantages:
- OS is not aware of threads, scheduling is performed at process level
  - One thread blocked → Process blocked → All threads blocked (assumed by OS, although other threads may not be blocked)
  - Cannot exploit multiple CPUs!

KERNEL Thread

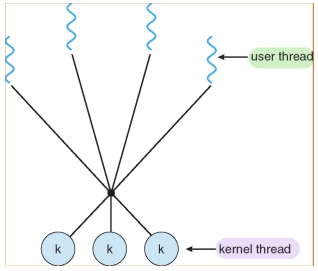


Advantages:
- Kernel can schedule on thread levels: More than 1 thread in the same process can run simultaneously on multiple CPUs

Disadvantages:
- Thread operations is now a system call
  - Slower and more resource intensive
- Generally less flexible
  - Used by all multithreaded programs
  - If implemented with many features: Expensive, overkill for simple program
  - If implemented with few features: Not flexible enough for some programs

HYBRID THREAD MODEL

- Have both Kernel and User threads
  - OS schedule on kernel threads only
  - User thread can bind to a kernel thread
- Offer great flexibility
  - Can limit the concurrency of any process / user



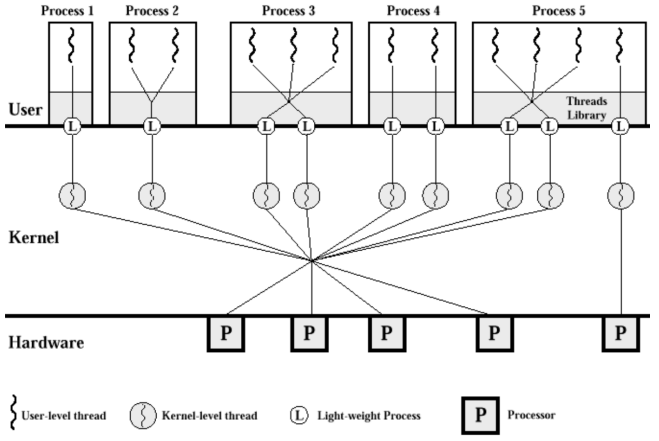
Example:
Process P is running on System X and has **three user-level threads** (A, B, C). System X uses the **hybrid thread model**, and has assigned **two kernel threads** to process P. System X has a four-core CPU, with each core having a single hardware context, i.e., each core can only run **one thread at a time**.

If there are no other processes in the system, what is the maximum number of cores process P can utilize at any given moment?

—-

Since process P is assigned **only two kernel threads**, it can only have **2 independent threads running at any given time**. Threads A, B, and C can take turns, but at any given time **at most two** of them will be running.

Hybrid Model Example: Solaris



Threads on Modern Processor
- Threads started off as a software mechanism
  - User space library → OS aware mechanism
- Now exists hardware support on modern processors
  - **Simultaneous multithreading (SMT)**: Supply multiple sets of registers (GPRs, and special registers) to allow threads to run natively and in parallel on the same core. Eg hyperthreading on Intel.
    - 2 hyperthreads on each CPU ⇒ can have concurrent threads run on each CPU core

5.3 POSIX Threads: `pthread`

- Standard defined by the IEEE. Supported by most Unix variants
- Defines the **API** and **behavior**
  - But implementation is not specified. So, `pthread` can be implemented as user / kernel thread

- Header file `#include <pthread.h>`
- Compilation (flag is system dependent) `gcc XX.c -lpthread`
- Useful data types
  - `pthread_t` : Represents a **thread id (TID)**
  - `pthread_attr`: Represents attributes of a thread

`pthread_create` Creation Syntax

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

contains thread id of new thread

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

`pthread_exit` Termination Syntax

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

- Parameters:
  - `exitValue` (can be specified): Value to be returned to whoever (only to parent for exit call) synchronize with this thread
- 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 and Termination example

thread

main

```
//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;
}
```

new pid of new thread

no attributes

no input params for sayHello function

wont the new thread to start executing the sayHello function

`pthread_join` Simple Synchronisation - 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` : 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;
}
```

main thread creates 5 threads

function executed by each thread

might not be

- `globalVar` might not be 500 as we are not sure when the `main` function will execute `printf` with respect to the thread.
- Because of **RACE CONDITION**, the sum is unpredictable.

`pthread` : Sharing of memory space v2.0

```
int globalVar;

void* doSum( void* arg )
{ //same as before }

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 Synchronization
        pthread_join( tid[i], NULL );

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

to

- `main` waits for **all** 5 threads to **terminate** before executing `printf`
- NOTE: Accessing shared variable is much more tricky. "`globalVar++`" is NOT a single machine operation: Read globalVar from memory, Increase globalVar, Store globalVar back to memory.

6 ❗[Process Management] Synchronisation❗

Problems with concurrent execution

- When two or more processes execute concurrently in interleaving fashion AND share a modifiable (read, write) resource
  ➔ Can cause **synchronization problems**
- Execution of a single sequential process is **deterministic**: Repeated execution gives the same result
- Executing concurrent processes may be non-deterministic:
  - **RACE CONDITION**: Execution outcome depends on the **order in which the shared resource is accessed/modified**; possible when there is at least one write.
  - No race conditions when multiple threads read the same variable, as the *order of reads doesn't impact the result.*

Race condition: Good behaviour

Time	Value of X	P1	P2
1	345	Load X ➔ Reg1	
2	345	Add 1000 to Reg 1	
3	1345	Store Reg1 ➔ X	
4	1345		Load X ➔ Reg1'
5	1345		Add 1000 to Reg1'
6	**2345**		Store Reg1' ➔ X

Race condition: Bad behaviour

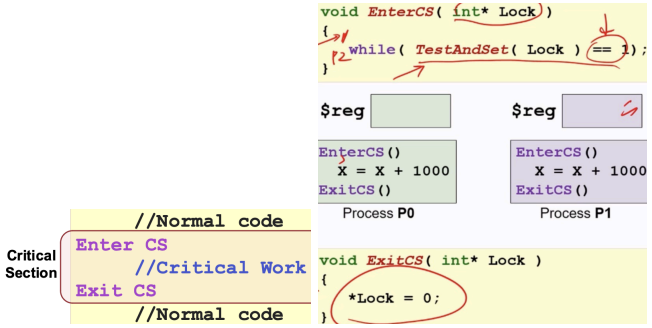
Time	Value of X	P1	P2
1	345	Load X ➔ Reg1	
2	345	Add 1000 to Reg1	
3	345		Load X ➔ Reg1'
4	345		Add 1000 to Reg1'
5	1345	Store Reg1 ➔ X	
6	**1345**		Store Reg1' ➔ X

Time quantum finished, so the OS switch to P2

- Incorrect execution is due to the **unsynchronized access to shared modifiable resources**

6.1 Critical Section (CS)

Critical Section (CS): Part of the memory where shared memory is accessed (with race condition)



Properties of Correct Critical Section Implementation

- **MUTUAL EXCLUSION**
  - If process P is executing in the critical section, all other processes are prevented from entering the critical section. *No two processes simultaneously inside their CS.*
- **PROGRESS**
  - If no process in a critical section, one of the waiting processes granted access.
- **BOUNDED WAIT**
  - After process P requests to enter the critical section, there exists an upper bound on the number of times other processes can enter the critical section before P. *No process waits forever to enter its CS.*
- **INDEPENDENCE**
  - Processes not executing in the critical section should never block other processes. *No process outside CS may block any process.*

Symptoms of Incorrect Synchronisation

- **DEADLOCK**
  - All processes blocked → No progress
- **LIVELOCK**
  - Usually related to deadlock avoidance mechanism
  - Processes keep changing state to avoid deadlock and make no other progress
  - Typically processes are *not blocked*
- **STARVATION**
  - Some processes never get to make progress in their execution as it is perpetually denied necessary resources

6.2 CS Implementations Overview

- **Assembly level implementations**:
  - Mechanisms provided by the processor
- **High level language implementations**:
  - Utilizes only normal programming constructs (Eg CS lock using normal variables)
- **High level abstraction**:
  - Provide abstracted mechanisms that provide additional useful features (eg abstract data types, provided as library calls, and involve system calls)
  - Commonly implemented by assembly level mechanisms

6.2.1 Assembly Level Implementation

Test and Set (Atomic instruction)

- Common machine instruction provided by processors to aid synchronisation

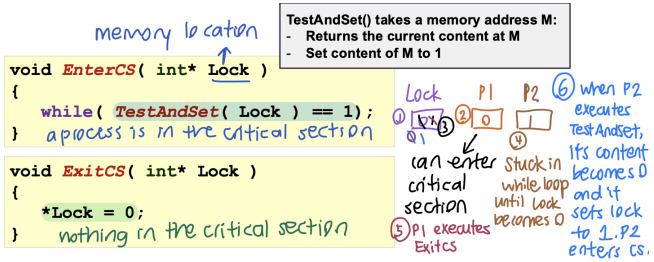
```
TestAndSet Register, MemoryLocation
```

- **ATOMIC**: Performed as a **single machine operation**, indivisible. No instructions interleaved in between.

Behaviour:

- **Load** current content at `MemoryLocation` into **`Register`**
  - To know the current value of the `Lock`
- Store a 1 into `MemoryLocation`
  - If current value of `Lock` is 0, need to **atomically** *set* the value of lock so no other process can enter

Assume ***`TestAndSet`*** machine instruction has an equivalent high level language version:



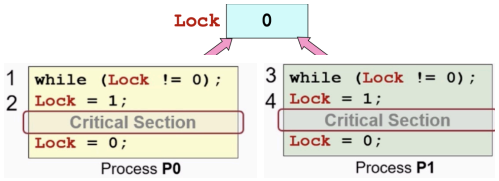
Before entering CS, a process calls ***`EnterCS`***, busy wait until ***`Lock`*** is free; then it acquires the lock and returns. After leaving CS, the process calls ***`ExitCS`***, which stores a 0 in the ***`Lock`***.

INEFFICIENT: ***Busy waiting*** (keep checking the condition until it is safe to enter the Critical Section) → Wasteful use of processing power

- Variants: Compare and Exchange, Atomic Swap, Load Link / Store Conditional

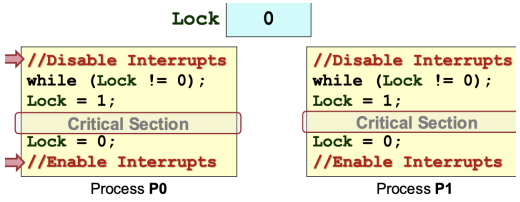
6.2.2 High Level Language Implementation

Bad: Lock Variables



- Interleaving in 1,3,4,2 violates **mutual exclusion**
- Having a single, shared (lock) variable, initially 0. When entering CS, test lock and set to 1. Thus, 0 means no process in CS, 1 means some process in CS.
- Suppose that one process reads the locks to be 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1. **Two processes will be in their critical regions at the same time.**

Bad: Fix Lock Variables with Interrupts Disabled

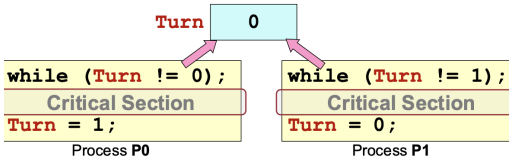


- **Disable interrupt → Prevent Context Switch** → Solve problem above
- Reason for interrupts is the timer interrupt which preempts a process and schedules another process. **Disable interrupts to prevent interleaving**

BUT

- Buggy critical section may stall the WHOLE system. Eg code is stuck, but other processes and even the OS cannot preempt due to disable interrupt
- Busy waiting
- Requires permission to disable / enable interrupts (privileged instructions, handled only by OS, not processes)
- If the system is a multiprocessor, disables only one CPU, others will continue to run and access shared memory.

Bad: Strict Alternation



- Each process checks for its own turn. Achieves **mutual exclusion**.
- The integer variable `Turn`, initially 0, keeps track of whose turn it is to enter the critical region and examine / update the shared memory.

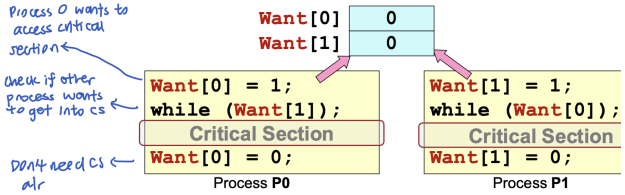
Assumption

- P0 and P1 execute the above in loop
- Take turn to enter **critical section**

Problems

- **Violates** the ***independence*** property. 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: `Want` variable per process, without `Turn`

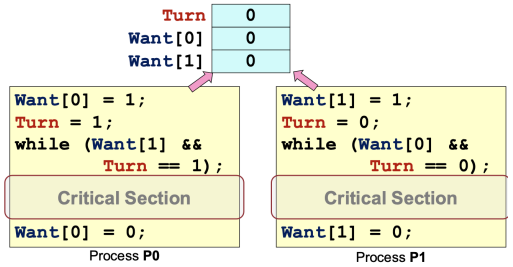


- `Want` int variable indicates if process wants to get locked
- Solve **independence** problem: If P0 or P1 is not around, another process can still enter the Critical Section.
- Problem: **Deadlock** when interleaving.

```
Want[0] = 1;
Want [1] = 1;
while ( …
```

Both processes are stuck in a while loop.
Violates *progress*.

Peterson's Algorithm



- Assume: Writing to `Turn` is an **atomic** operation
- `Want[1] && Turn == 1`: Check if the other process wants the lock. If no, then this process exits the while loop. If the other process wants, then check the turn variable for whose turn it is.
- Processes can store their / the other process number in `Turn`. consider both processes trying to enter CS almost simultaneously. Both store in the `Turn` variable. Whichever stores last is the one that counts; first one is overwritten and lost. After, only one can enter CS, and the other will loop.

Disadvantages of 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.

6.2.3 High Level Abstraction

SEMAPHORE

A generalised synchronization mechanism that ***only specifies BEHAVIOR***, **not the implementation**. Can have different implementations.

- Can block processes, which becomes **sleeping process**
- Can unblock / wake up one or more sleeping process

Semaphore S contains an integer value, and can be initialised to any non-negative values.

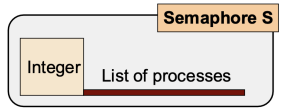
- **GENERAL (counting) SEMAPHORE** $S \geq 0$, S = 0, 1, 2, 3 …
- **BINARY SEMAPHORE (MUTEX)** S = 0 or 1
- Binary semaphore is sufficient ie general semaphore can be mimicked by binary semaphores. General semaphore provided for convenience.

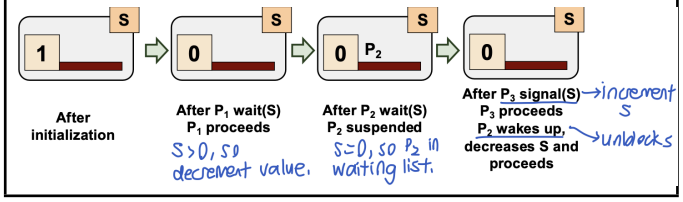
Atomic Semaphore Operations: `wait()`, `signal()`

ATOMIC: Once a semaphore operation starts, no other process can access semaphore until operation has completed or blocked.

`wait(S)` (or *P()* or `Down()`)	`signal(S)` (or *V()* or `Up()`)
• Takes in a semaphore. If semaphore value **S ≤ 0**, **blocks** current process (go to sleep). • **Decrements** semaphore value S when it proceeds.	• Takes in a semaphore. • Wakes up one sleep process if any. **Never blocks**. • **Increments** semaphore value S when it proceeds.

Semaphore as
- A protected integer
- A list to keep track of waiting processes





Semaphore Invariant given $S_{initial} \geq 0$

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

Increment S *Decrement S*

- `#signal(S)`: Number of `signal(S)` operations executed
- `#wait(S)`: Number of `wait(S)` operations **completed**

MUTEX: Mutual Exclusion (Binary Semaphore Usage)

`Wait( S );`
Critical Section
`Signal( S );`
	Set binary semaphore S = 1. For any process, do `wait(S)` (decrement S) before entering CS, and `signal(S)` (increment S) after finishing CS. **S can only be 0 or 1**, as deduced by semaphore invariant.

Not busy looping. When process is waiting, it is blocked.

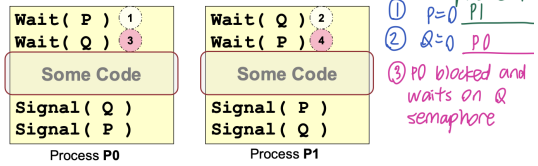
Mutex: Correct CS - Informal Proof

- **Mutual Exclusion**
  - N_{CS} = number of processes in Critical Section
    = #processes that completed `wait(S)` but not `signal(S)`
    = #`wait(S)` - #`signal(S)`
  - $S_{initial} = 1$
  - $S_{initial} = 1 + \#signal(S) - \#wait(S)$
  - $S_{initial} + N_{CS} = 1$
  - Since $S_{initial} \geq 0$, then $N_{CS} \leq 1$
- **Mutex has no deadlock**
  - Deadlock = All processes stuck at `wait(S)`
    $\Rightarrow S_{current} = 0$ and $N_{CS} = 0$
  - But $S_{initial} + N_{CS} = 1 \Rightarrow$ contradiction!
- **Starvation**
  - Suppose P1 is blocked at `wait(S)`. P2 is in CS, exits CS with `signal(S)`.
  - If no other process is sleeping, P1 wakes up.
  - If there are other processes, P1 eventually wakes up (assuming fair scheduling, then starvation-free)

Incorrect use of Semaphore → Deadlock

Incorrect interleaving: P0 does `wait(P)`, then P1 does `wait(Q)`:

Assume semaphores **P = 1**, **Q = 1** initially



Other High Level Abstractions: Conditional Variable

- Allow a task to wait for certain event to happen
- Has the ability to **broadcast**, i.e., wakes up all waiting tasks
- related to **monitor**

6.3 Classical Synchronisation Problems

6.3.1 Producer Consumer problem

Specification

- Processes share a bounded buffer of size K
- **Producers** produce items to insert in buffer
  - Only when the buffer is **not full** (< K items)
- **Consumers** remove items from buffer
  - Only when the buffer is **not empty** (> 0 items)

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

Initial Values:

- `count` = `in` = `out` = 0
- `mutex` = `S(1)` //semaphore with initial value 1
- `canProduce` = TRUE and `canConsume` = FALSE;

Code correctly solves the problem, but **busy waiting** is used.

- `canConsume` = Triggers consumer to *try* to get item
- `canProducee` = Triggers consumer to *try* to get item
- `wait(mutex) + signal(mutex)`: Creates a CS
- `in=(in+1)%K` | `out=(out+1)%K` : Wraps around, circular array

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

Initial Values:

- `count` = `in` = `out` = 0
- `mutex` = `S(1)`, `notFull` = `S(K)`, `notEmpty` = `S(0)`

Code correctly solves the problem. **No busy-waiting**, "unwanted" producer/consumer will go to sleep on respective semaphores.

- `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

6.3.2 Readers Writers problem

Specification - How to sync reader and writer?

- Processes share a data structure D.
- Reader retrieves information from D.
- Writer modifies information in D.
- Writer must have exclusive access to D (**WRITE ALONE**)
- Reader can access with other readers (CAN READ TOGETHER)

```
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`

The first reader to get access to the database does a ***`wait(S)`*** (down) on semaphore ***`roomEmpty`***. Subsequent readers merely increment counter ***`nReader`***. As readers leave, they decrement ***`nReader`***. The last reader to leave does a ***`signal(S)`*** (up) on ***`roomEmpty`***, allowing a blocked writer, if there is one, to get in.

PROBLEM: As long as at least one reader is active, subsequence readers admitted. Writer kept suspended until no reader is present. Writer may never get in.

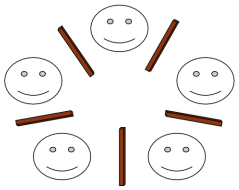
RECTIFICATION: When a reader arrives and the writer is waiting, the reader should be suspended behind the writer instead of being admitted immediately. Writer waits for active readers to finish, but not readers after. Disadvantage is less concurrency → lower performance.

6.3.3 Dining Philosophers problem

Specification: 5 philosophers are seated around a table. There are 5 single chopstick 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. Devise a **deadlock-free** and **starve-free** way to allow the philosopher to eat freely.

BAD: All philosophers simultaneously take up the left chopstick, and none can proceed. **DEADLOCK**.

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

//For philosopher i
while (TRUE){

    Think( );
    //hungry, need food!
    takeChpStk( LEFT );
    takeChpStk( RIGHT );

    Eat( );

    putChpStk( LEFT );
    putChpStk( RIGHT );
}
```



BAD: Make the philosopher put down the left chopstick if the right chopstick cannot be acquired. Try again later. **LIVELOCK.** All take up the left chopstick, put it down, take it up, put it down …

BAD: Use single mutex.

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

//For philosopher i
while (TRUE){
       Think( );

       wait( mutex );

       takeChpStk( LEFT );
       takeChpStk( RIGHT );
       Eat( );
       putChpStk( LEFT );
       putChpStk( RIGHT );

       signal( mutex );
}
```

Before acquiring chopsticks, ***wait(mutex)***. After putting down, ***signal(mutex)***.

No deadlock. No starvation. Works because there is at most 1 eating at a time.

But performance bug, only one can eat at an instant. Concurrency can be maximised. With five chopsticks, at least two philosophers can eat at the same time.

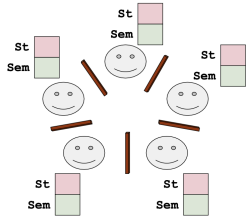
Dining Philosopher: Tanenbaum Solution

```
#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];
one for each philosopher
void philosopher( int i ){
    while (TRUE){
       Think( );
       takeChpStcks( i );
       Eat( );
       putChpStcks( i );
    }
}
```

```
void takeChpStcks( i )
{ shared variable, so put in semaphore
       wait( mutex );
       state[i] = HUNGRY;
       safeToEat( i );
       signal( mutex );
       wait( s[i] );
}
```

```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
Check  (state[LEFT] != EATING) &&
that left (state[RIGHT] != EATING) ) {
and right are not eating
        state[ i ] = EATING;
        signal( s[i] );
    }
}
```

```
void putChpStcks( i )
{
       wait( mutex );
Right
philosopher
blocked  state[i] = THINKING;
by wait  safeToEat( LEFT );
is now   safeToEat( RIGHT );
unblocked
       signal( mutex );
}
```



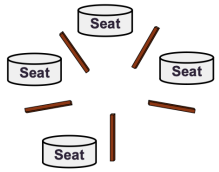
- **DEADLOCK-FREE**, allows maximum parallelism for an arbitrary number of philosophers
  - Each philosopher waits only on its own semaphore
  - The **mutex** is necessary for exclusive accesses to shared variables and the correctness overall, but it does not contribute to the freedom from deadlocks here.
- Array `state` keep track if philosopher is eating/thinking/hungry
- Move into an eating state only if neither neighbor is eating. Philosopher i's neighbors defined by `LEFT` and `RIGHT`. If i is 2, then `LEFT` = 1 and `RIGHT` = 3.
- After eating, each philosopher does send the signal to the neighbors (it also pre-signals to itself before it calls wait() on its semaphore)
- Array of semaphores, one per philosopher, so hungry philosophers can block if the chopsticks are busy.

Dining Philosopher: Limited Eater Solution

If at most 4 philosophers are allowed to sit at the table (leaving one empty seat)

➧ Deadlock is impossible!

```
void philosopher( int i ){
    while (TRUE){
       Think( );
       wait( seats );   N-1
       wait( chpStk[LEFT] );
       wait( chpStk[RIGHT] );
       Eat( );
       signal( chpStk[LEFT] );
       signal( chpStk[RIGHT] );
       signal( seats );
    }
}
```



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

6.4 Synchronisation Implementations

POSIX Semaphore

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

Others

- Programming languages with thread support will have some forms of synchronization mechanisms
- Java: All objects have built-in lock (mutex), **synchronized** method access (method protected by synchronisation mechanisms), etc.
- Python: supports mutex, semaphore, conditional variable, etc.
- C++: Added built-in thread in C++11; Support mutex, conditional variable

7. [Memory Management] Memory Abstraction

7.1 Memory

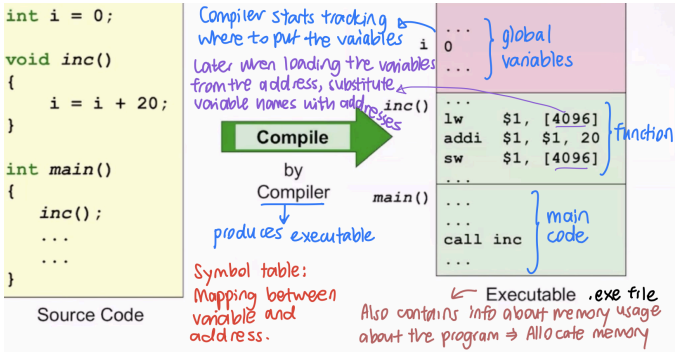
Memory hardware

- Physical memory storage: **Random Access Memory (RAM)** accessible in O(1). Like an array of bytes, each byte with a **physical address** as a unique index.
- A **contiguous** memory region: Interval of consecutive addresses.

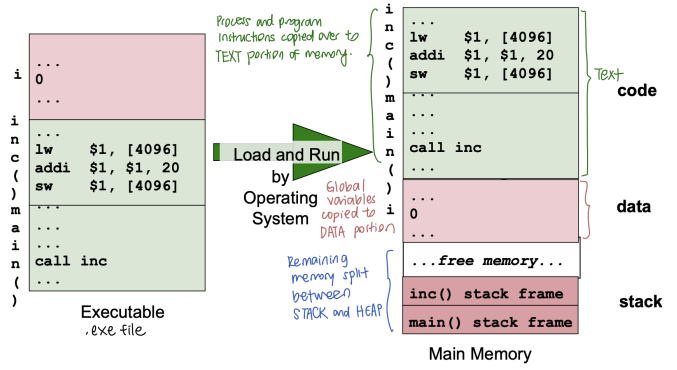
Memory Hierarchy: OS abstracts hierarchy into useful mode, and manages it. Fast → More expensive → Cannot have a lot (smaller size).

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

Binding of memory address:



Actual execution:



Memory usage: Two types of data in a process

TRANSIENT data	PERSISTENT data
Data that only exists in the lifetime of the function. Valid for a limited duration.	Can exist forever. Valid for duration of program unless removed.
Local variables. **Function parameters.**	**Dynamic variables. Dynamically allocated memory** written by malloc. (`(int*)malloc(4)`) → If not freed, PC can run out of memory → Memory leakage
Both types of data sections can **grow / shrink** during execution.	

Managing Memory: OS handles the following memory related tasks

- **Allocate** memory space to new process
- **Manage** memory space for process
- **Protect** memory space (security) of process from each other
- Provides memory related **system calls** to process
- Manage memory space for **internal use** (PCB is a structure)

7.2 Memory abstraction

Memory Abstraction: Presenting a logical interface for memory accesses

7.2.1 No memory abstraction

Process **directly uses physical address**

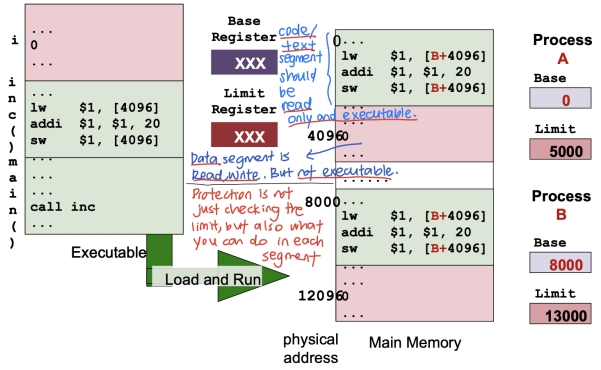
- Pros: Straightforward with 1 process running. Fast. addresses fixed during compile time. For lw, go to address, read the data, and put it into register. For sw, go to address and store the data there.
- Cons: Both processes assume they start at 0, resulting in conflicts. Hard to protect memory. Both processes access at the same time → Race condition, cannot guarantee exclusive access to physical address

7.2.2 Fix Attempt 1: Address Relocation

- Recalculate memory references when process is loaded into memory
- Eg If a process is at address 800, add 8000 to all memory addresses.
- Problems: Slow loading time. Not easy to distinguish memory reference from normal integer constant (OS does not know if register content is an address, only hardware knows.)

7.2.3 Fix Attempt 2: Base + Limit Registers

- **BASE REGISTER** as the base of all memory references [relocation]
  - At compilation time: All memory references compiled as offset from base register.
  - At loading time: Base register initialised to the starting address of the process memory space.
- **LIMIT REGISTER** to indicate the range of memory space of current process [protection]
  - All memory access is checked against the limit (and what can be done in the segment) to **protect** memory space integrity
  - Outside of range: Segmentation fault, OS kills program.



- Problems: Overhead; each memory access incurs:
  - Addition (Actual = Base + Address)
  - Comparison (Check Actual < Limit if Limit is the ending address of process) (Check Actual < Base + Limit - 1 if Limit is the length)
- The above idea is useful
  - Later generalized to **segmentation mechanism**
  - Provides a **crude memory abstraction**: Address 4096 (**LOGICAL ADDRESS**) in Process A and B are no longer the same physical location. **PHYSICAL ADDRESS (= BASE + LOGICAL ADDRESS)** sent to memory to get data.

7.2.4 Memory Abstraction: LOGICAL ADDRESS

- Embedding actual **Physical Address** is program is a bad idea
- **LOGICAL ADDRESS**: How the process VIEWS its memory space.
  - Each process has a self-contained, independent logical memory space that they will reference using logical address, then the OS will do the mapping (**logical address + base)** in contiguous memory) between logical and physical address. *Allow multiple processes to share **same logical address**, but mapped to **different physical addresses**, so that processes do not override each other.*
- **Multitasking, Context Switching, and Swapping**
  - To support **multitasking**, allow multiple processes in the physical memory at the same time. Need ways to partition memory, so we can switch between processes using different memory partitions.
- When physical memory is **full**, free up memory by:
  - removing terminated process or
  - swap blocked process to secondary storage

7.3 Contiguous Memory Allocation

Allocating and managing **continuous** chunk of memory

- Problem: External / Internal fragmentation
- Advantage: Simple, just need the starting address and length

7.3.1 Contiguous Memory Management

- Process must be in memory during execution
  - **Store Memory** concept: Stored program. Program and data to be run must be present in memory for it to be executed.
  - **Load-Store Memory** execution model: Only instructions that interact with memory are *load* and *store*. Everything else interacts with registers.
- ASSUMPTIONS
  1. Each process uses a **MEMORY PARTITION** of a contiguous memory region
  2. Physical memory is **large enough** to contain one or more processes with complete memory space.

Fixed-Size Partitioning

- Physical memory is split into a ***fixed number of partitions***.
- A process will occupy one of the partitions
- Cons: **INTERNAL FRAGMENTATION** when a process does not occupy the whole partition.
  - Partition size needs to be large enough to contain the largest process → Smaller process will waste memory space.
- Pros: Easy to manage. Fast to allocate (every free partition is the same, no need to choose)

Variable-Size (Dynamic) Partitioning

- Partition is created based on the ***actual size*** of process
- OS keep track of the occupied and free memory regions
  - Perform splitting and merging when necessary

Pros:

- **NO INTERNAL FRAGMENTATION**. All processes get exact space.
- Flexible

Cons:

- With process creation / termination / swapping → Large number of holes (free memory) between partitions → **EXTERNAL FRAGMENTATION** → Possible to **merge** the holes. Can be fixed via **compaction**, but slow (running processes need to stop). But better than internal fragmentation which is unreachable
- Maintain more info in OS → Slower to locate appropriate region

Dynamic Partitioning: ALLOCATION ALGORITHMS

Assuming OS maintains a list of partitions (given process) and holes (free memory). Algorithm to locate partition of size `N`:

- Search for hole with size `M` > `N`. Variants:
  - **FIRST-FIT**: Take the first hole that is big enough.
  - **BEST-FIT**: Take the smallest hole that is large enough. (Tend to leave very small holes → **EXTERNAL FRAGMENTATION)**
  - **WORST-FIT**: Take the largest hole. (Creates larger holds for next process) (Likely to be most useful)
- Split the hole into `N` (new partition) and `M-N` (left over space → **NEW HOLE**)

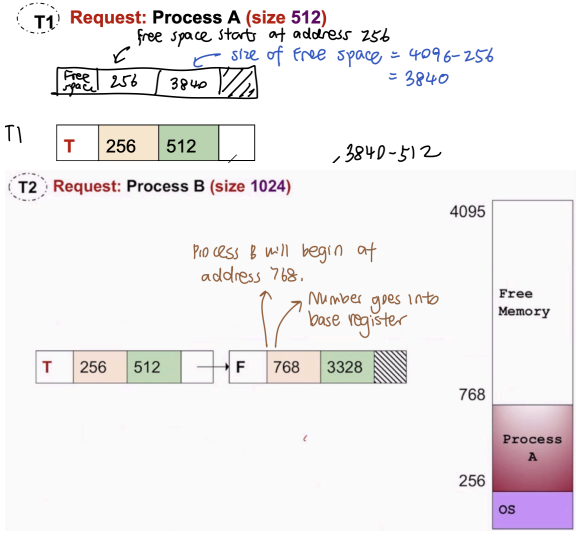
Partition info can be maintained as a **linked list** or bitmap. For the **linked list**, each node contains: Status (True = Occupied, False = Free), Start Address, Length of partition / hole, Pointer to next node.

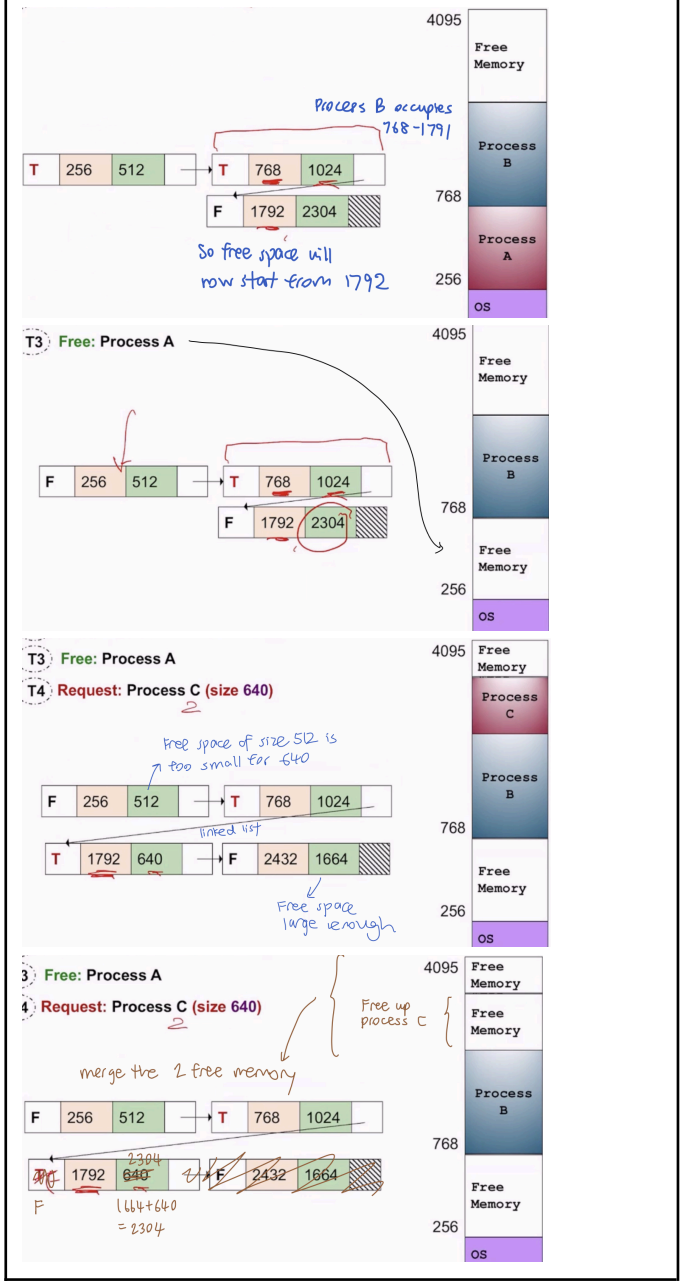
Allocation: O(N). Deallocation: O(1). Merging: O(1).

Dynamic Partitioning: MERGING and COMPACTION

- **MERGE** freed partition with adjacent hole if possible
- **COMPACTION**
  - Move occupied partition around to create consolidated holes
  - Time consuming, cannot be invoked too frequently

Example: First-Fit algorithm

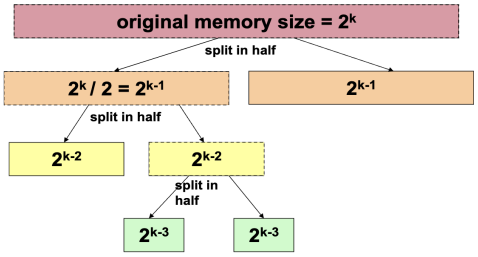




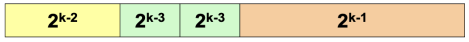
Dynamic Allocation Algorithm: BUDDY SYSTEM

Buddy memory allocation provides efficient
• Partition splitting; • Locating good match of free partition (hole)(O(1)); • Partition de-allocation and coalescing (merging).

- Free block **split into half repeatedly** to meet request size.
- The two halves forms a sibling blocks (**buddy blocks**)
- Free buddy blocks can be merged to form larger block



Buddy blocks



!! BUDDY BLOCKS !!

Two blocks B and C are buddies of size 2^S, if the S^{th} *bit* of B and C is a **complement**, and the leading bit up to S^{th} *bit* are the **same**.

```
A = 0 (0000002), size = 32
After splitting:                    log2 16 = 4 = S
• B = 0  (0000002), size = 16       4th bit would
• C = 16 (0100002), size = 16       be inverses
```

Implementation

- Array *A[0… k]*, where 2^K is the largest allocatable block size
  - Each array element *A[J]* is a linked list which keep tracks of free block(s) of size 2^J, and are indicated by starting address

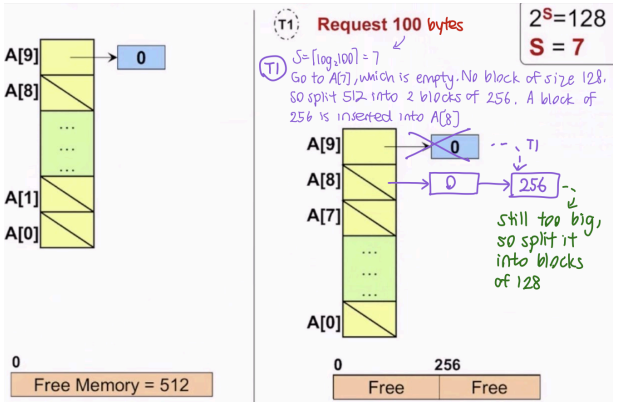
Buddy System: Allocation Algorithm $O(\log_2 N)$ (binary splitting)

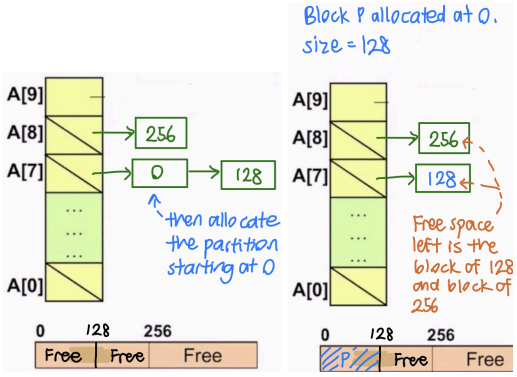
- To allocate a block of size *N*
  1. Find the smallest *S* where $2^S \geq N$; $S = \lceil \log_2 N \rceil$
  2. Access *A[S]* (points to linked list of size 2^S) for a free block
  - If free block exists, remove the block from free block list and allocated the block
  - Else find the smallest *R* from *S+1* to *K*, such that *A[R]* has a free block *B*. For *(R-1 to S)*, repeatedly split *B* → *A[S … R-1]* has a new free block. Go to step 2.

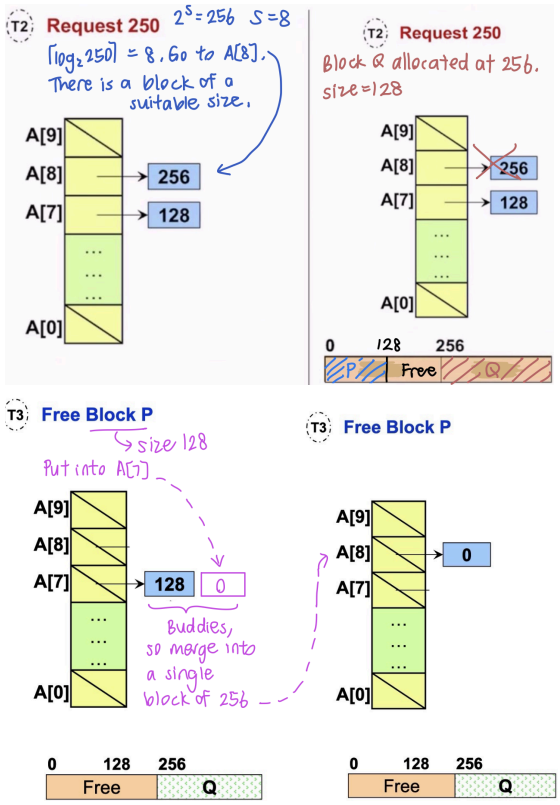
Buddy System: Deallocation Algorithm $O(1)$

- To free a block B:
1. Check in `A[S]`, where 2^S `== size of B`
2. If the buddy C of B exists (also free)
   a. Remove B and C from list
   b. Merge B and C to get a larger block B'
   c. Goto step 1, where B ← B'
3. Else (buddy of B is not free yet)
   - Insert B to the list in `A[S]`

Merging: $O(\log_2 N)$







7.4 DISJOINT Memory Allocation

Both **paging** and **segmentation** allow the logical address space to be in **disjoined physical region**

- **PAGING** splits the logical address into **fixed** size pages, stored in same fixed size physical memory frames
- **SEGMENTATION** splits the logical address into **variable** size **segments** according to their usage. Stored in variable-sized physical partitions.

Remove assumption 1, Process memory space is now in **DISJOINT physical memory locations,** via **PAGING**.

7.4.1 PAGING Scheme

- Split **physical memory** into fixed-size regions ⇒ ***PHYSICAL FRAME***
- Split **logical memory** into same-size regions ⇒ ***LOGICAL PAGE***
- !! Size of ***FRAME*** and ***PAGE*** must be identical !!
- At execution time, the **pages** of a process are loaded into **any available memory frame**
  - **Logical** memory space remain **contiguous**
  - **Occupied physical** memory region can be **disjoined**

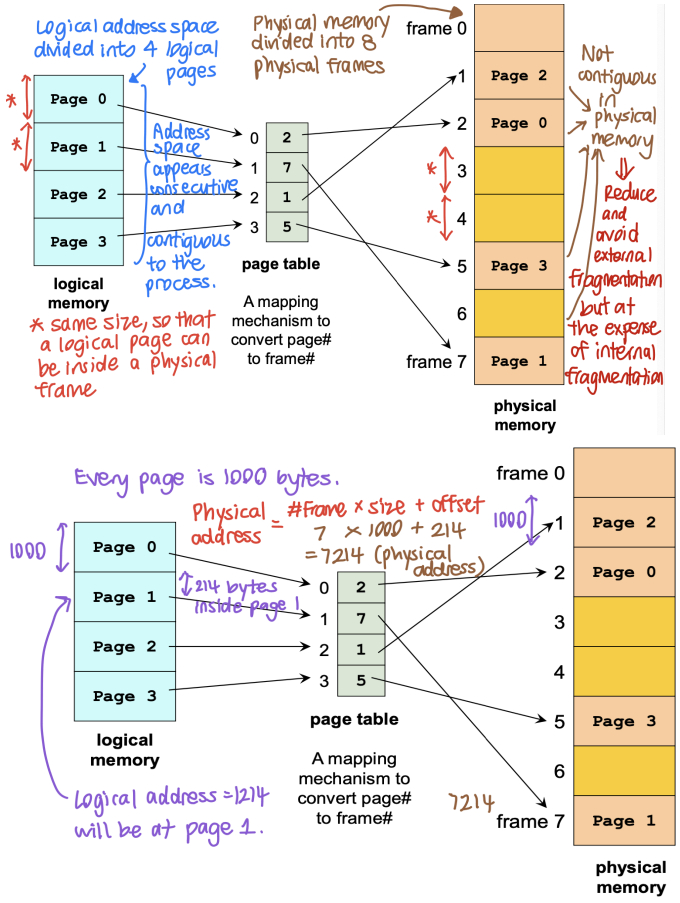
- **NO EXTERNAL FRAGMENTATION**. No left-over physical memory region. All free frames can be used with no wastage.
- Has **INTERNAL FRAGMENTATION** (process does not occupy whole region). Logical memory space may not be multiple of page size.
- Clear separation (assign different frames to different processes) of logical and physical address space → Flexibility, simple translation.

PAGE TABLE (convert page# to frame #) - Lookup Mechanism

- Use starting address (in base register) and size of process (in limit register)
- One per process
- Array where index is the page number, value is the frame number

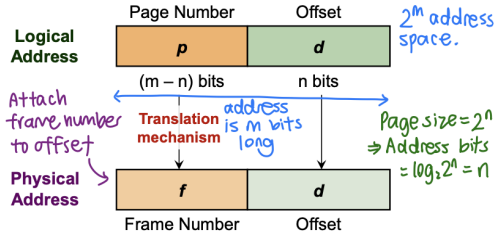
Physical Address = Frame_number x Physical_frame_size + Offset

- Frame_number: Physical frame number
- Offset: Displacement from beginning of physical frame
- NOTE multiplication is expensive.



Simplify address translation calculation:

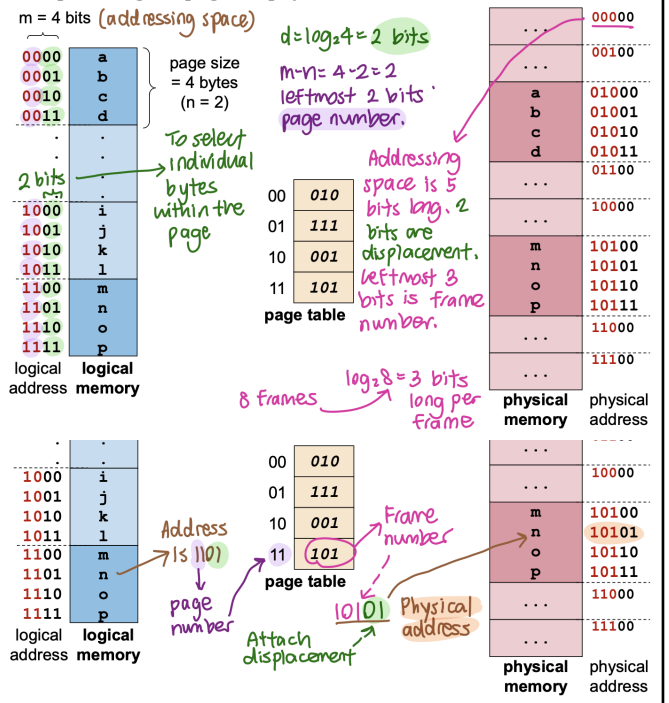
- Keep **frame size (page size) as a power of 2**
- **Physical frame size == Logical page size**



- Given page / frame size of 2^n and ***m bits*** of logical address,
- Logical Address
  - **page number** p = Most significant ***m - n bits***
  - **offset** d = Remaining ***n bits*** of logical address = $\log_2 m$
- Use ***page number p*** to find ***frame number f*** (via page table)

⇒ ***Physical address = frame_number_f * 2^n + offset_d***

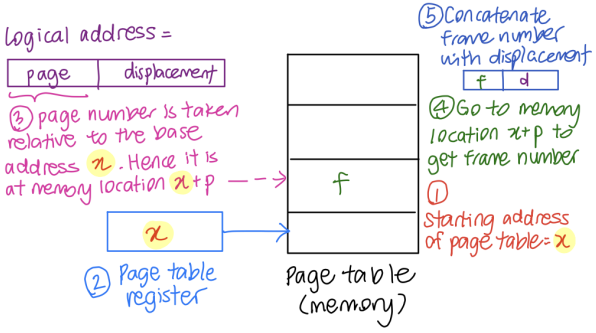
Example: 4 logical pages, 8 physical frames



Implementing Paging Scheme

- OS is a program:
  - Cannot see addresses accessed by other programs. If process is running, then the OS is not running. Addresses are generated by datapath (eg lw instruction) which are ***hardware***, not software.
  - Cannot see its own physical address. Can see its logical address
  - Hence occupies a logical address space.
  - Whenever the OS runs, it has to go through the translation process. Hence, **translation has to be done by the *HARDWARE*, not software**.
- Hardware: Page table register, pointing to the start of the page table

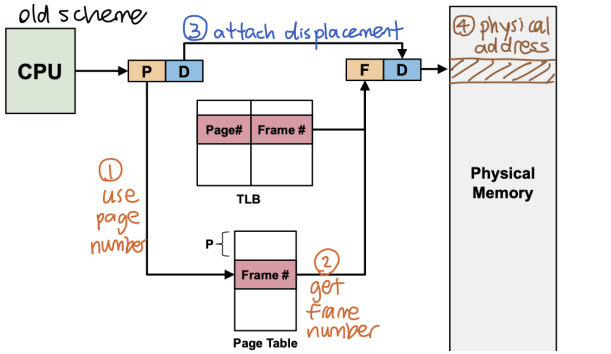
- This whole mechanism is handled by ***HARDWARE***:

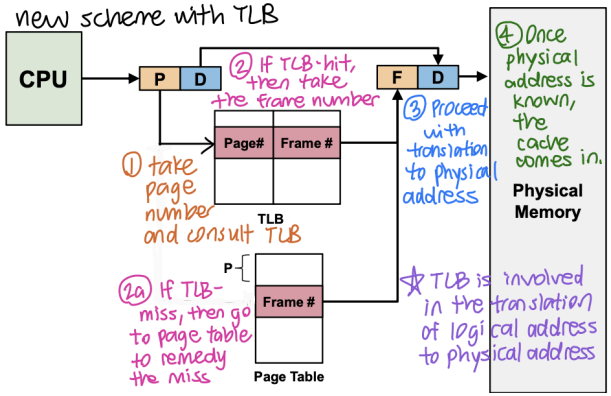


- Common **pure-software** solution:
  - OS stores page table information with the process information (eg *PCB*). Page table information is usually the starting address of the page table. This starting address has to be copied into the **page table register, which points to the start of the page table**. The page table is part of the OS, which is in logical memory space, which is in physical memory.
  - Note that as *every process has its own page table*, the OS stores the starting address of the page table *for each process in the PCB*.
  - ⇒ **MEMORY CONTEXT** of a process == **PAGE TABLE**
    - Memory context consists of information about the page table
  - **ISSUES**: Requires **2 RAM accesses** for every memory reference
    - 1st access: Read the indexed page table entry to get frame number
    - 2nd access: Access actual memory item (instruction / data)

Paging Scheme - *HARDWARE* support: *Translation Look-Aside Buffer (TLB)*

- On-chip TLB acts as a **cache** for *a few* page table entries
- Logical address translation: Use page number to search TLB associatively
  - **TLB-Hit**: Entry found.
    - Frame number retrieved to generate physical address
  - **TLB-Miss**: Entry not found.
    - Only the **FIRST access to a page** will be a **TLB-Miss**. **Remaining access will be a TLB-Hit**.
    - **RAM is only accessed during TLB-Miss**. Memory access to access the full page table
    - Retrieve frame number is used to generate physical address and update TLB





Impact of TLB on Memory Access Time

Suppose

- TLB access = 1ns
- Main memory access = 50ns (assuming no cache)
- TLB contains 40% of the whole page table

⇒ **Memory Access Time = TLB Hit + TLB Miss**

= 40%×(1ns + 50ns) + 60%×(1ns + 50ns + 50ns) = 81ns

Time to remedy TLB Miss

- Note: *Overhead of filling in TLB entry and impact of cache ignored.*
  - With cache:

    Memory access time = cache_hit_time × cache_access_time + (1 - hit_time) × cache_access_time + memory_access_time

TLB and Process Switching

- **TLB** is part of the **HARDWARE CONTEXT** of a process
- Hence, when a context switch occurs, TLB entries are flushed, so the new process will not get an incorrect translation.
- Hence, when a process resumes running, it will have many TLB misses (miss the first few accesses to the first few DIFFERENT PAGES, not the first few consecutive addresses) to fill the TLB
  - Possible to place some entries initially, e.g. the code pages to reduce TLB misses

Paging Scheme - Memory protection between processes

Access Right Bits

- Each Page Table Entry has several bits attached to it to indicate if it is writable, readable, executable (*rwx*)
  - E.g. Page containing code → executable (*rwx* = 001)
  - Page containing data → readable and writable (*rwx* = 110), no instructions inside data hence not executable
- Check memory access against Access Right Bits → Go through or get an exception

Observe that the logical memory range is usually the same for **all** processes. But not all processes utilized the whole range → Some pages are out-of-range for a particular process.

Valid Bit

- Attached to each page table entry to indicate if the page is valid for the process to access
- When a process is running and the OS creates space in the logical space for the process, the **OS will set the valid bits** and mark these pages as valid
- Memory access is checked against this bit
  - Out-of-range access will be caught by OS
  - If you load memory access for a page that is not valid ⇒ Segmentation fault

Paging Scheme - Page Sharing

Page table can allow several processes to **share the same physical memory frame**

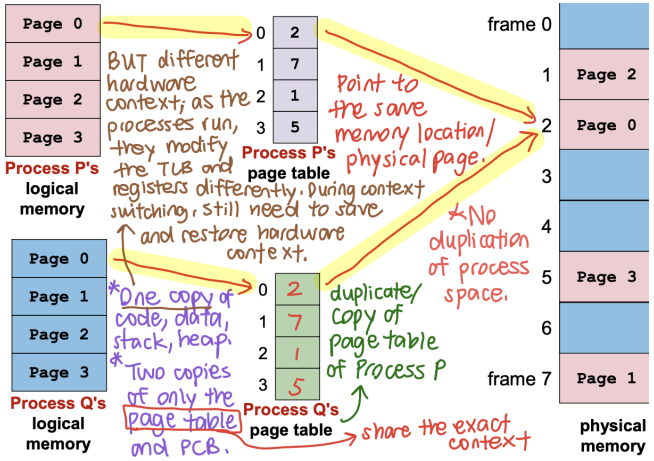
- Use the same physical frame number in the page table entries
- Multiple pages point to the same physical frame
- Use a shared bit in the page table entry to track if the page is shared

Possible usage:

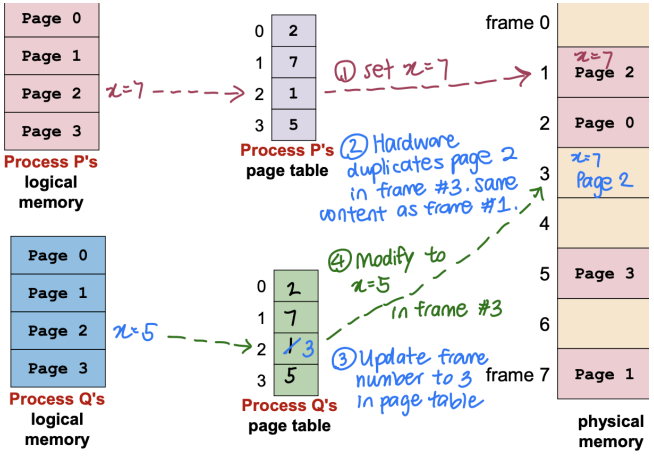
- **Shared code page**: Some code are being used by many processes, e.g. C standard library, system calls etc

- **Implement Copy-On-Write**: [refer to Process Abstraction]
  - When a parent forks, the memory space of the parent process need not be duplicated. The parent and child share the same pages, (i.e. page table is copied but frames remain), until one tries to change a value in it.
  - Using a shared bit, when the child needs to update a shared page, then the frame is duplicated and the page is "unshared".

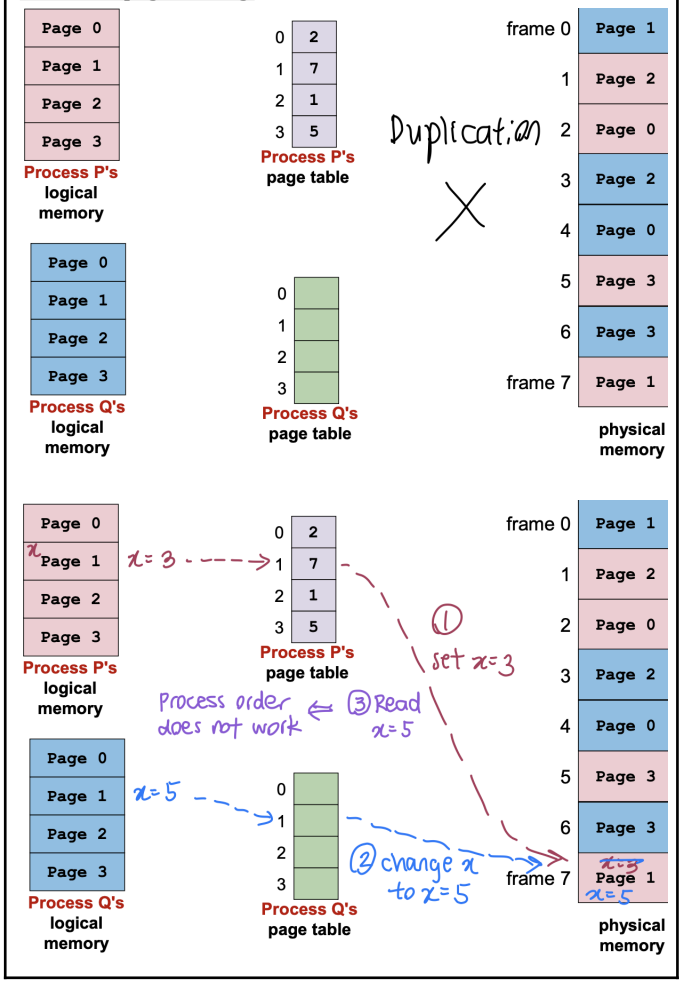
PAGE SHARING example:



COPY-ON-WRITE with PAGE SHARING example:



WITHOUT page sharing:



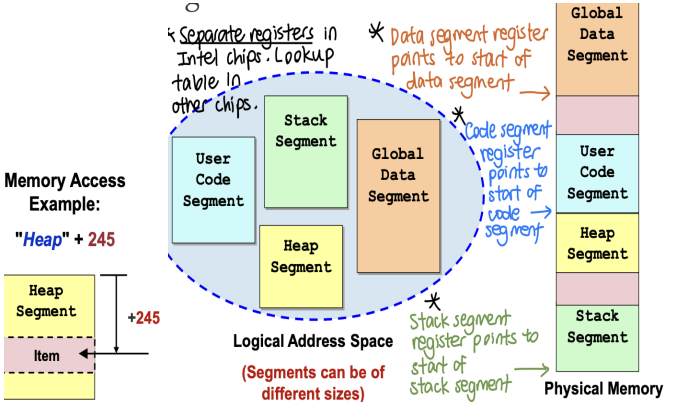
7.4.2 SEGMENTATION Scheme

- So far, the memory space of a process is treated as a single entity. But there are actually a number of memory regions with different usage (code region, global variables region, heap region with dynamic data, stack region with local variables) in a process
- If regions are in contiguous memory space,
  - difficult to allow regions to grow / shrink at execution time (eg stack, heap, library code regions)
  - difficult to check if a memory access if in range

⇒ Hence, separate the regions into multiple **MEMORY SEGMENTS**

- Logical memory space of a process is a collection of **segments**
- Each **MEMORY SEGMENT** has a
  - name: for ease of reference (shows the start of segment)
  - limit: range of segment

  ⇒ All memory reference specified as **Segment name + Offset**



Segmentation - Pros and Cons

Pros: Each segment is an **independent** contiguous memory space

- Can grow / shrink independently
- Can be protected / shared independently

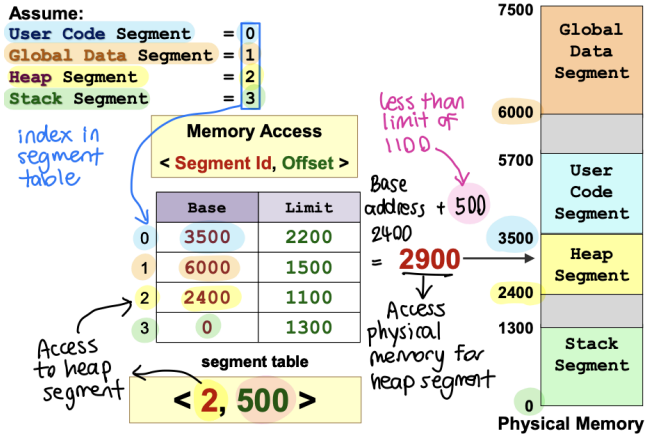
Cons: Segmentation requires variable-size **contiguous memory regions** ⇒ Can cause **EXTERNAL FRAGMENTATION**

Note: Segmentation is **NOT** the same as Paging.

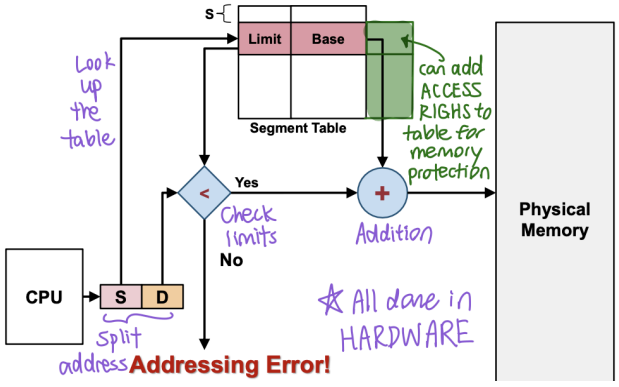
- **Segmentation solves the problem of dependence** between segment spaces. Separating the segments means the segments can grow and shrink independently.
- **Paging solves the problem of external fragmentation**.

Segmentation: Logical Address Translation

- Each segment mapped to a contiguous physical memory region with a **base address (in segment register)** and **limit (in limit register of segment)**
- **segment id** : Number to represent segmentation name
- ***Logical address <SegID, Offset>***
  - ***SegID*** (index into segment table) → To look up ***<Base, Limit>*** of the segment in a segment table
  - **Physical Address = Base + Offset**
  - **Offset < Limit for valid access. If offset > limit, then segmentation fault.**

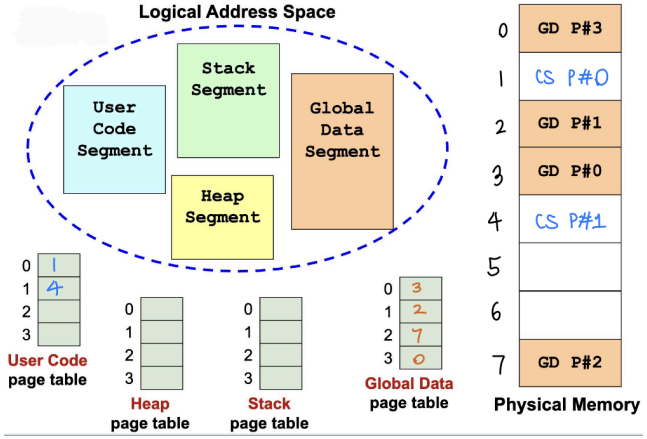


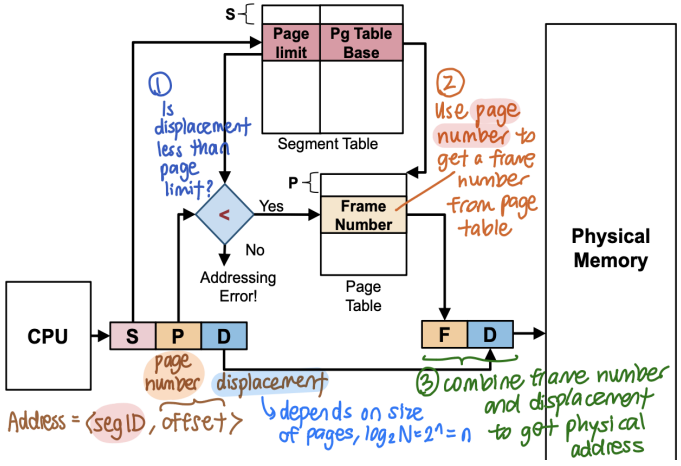
Hardware support in Segmentation

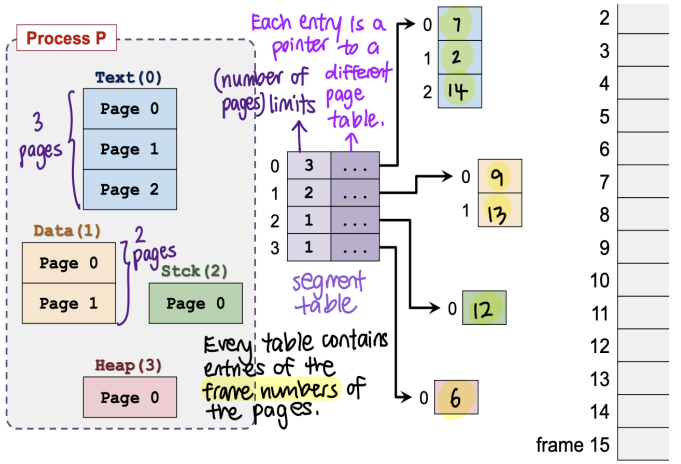


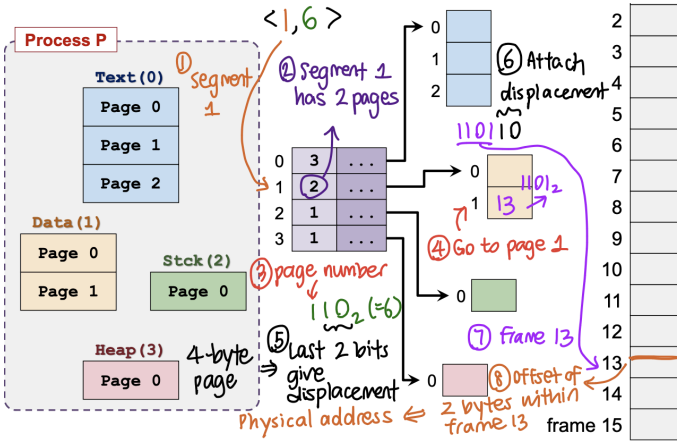
7.4.2 SEGMENTATION with PAGING

- **Each segment** is now composed of **several pages** instead of a contiguous memory region ⇒ **Each segment has a page table**
  - **Segment table** now points to the **page table address** instead of the base address.
  - Page limit remains unchanged.
- Entries in each page table cannot be duplicated. Segments should not occupy the same stretch of memory.
- Segment grows by allocating new page and adding to its page table. Similarly for shrinking.





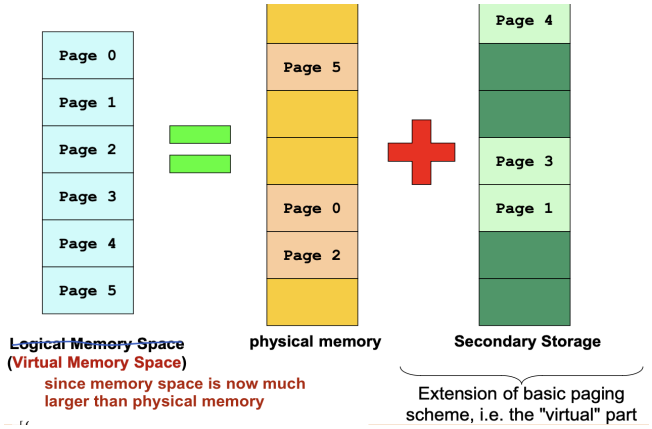




7.5 Virtual Memory Management

Remove assumption 2. Physical memory may not be large enough to contain one or more processes with complete memory space.
⇒ Split logical memory space into fixed size pages (*extension of paging*)

- Some pages may be in physical memory
- Others stored in **SECONDARY STORAGE** (HHD, SSD etc), which has a much larger capacity than physical memory



7.5.1 Extended Paging Scheme

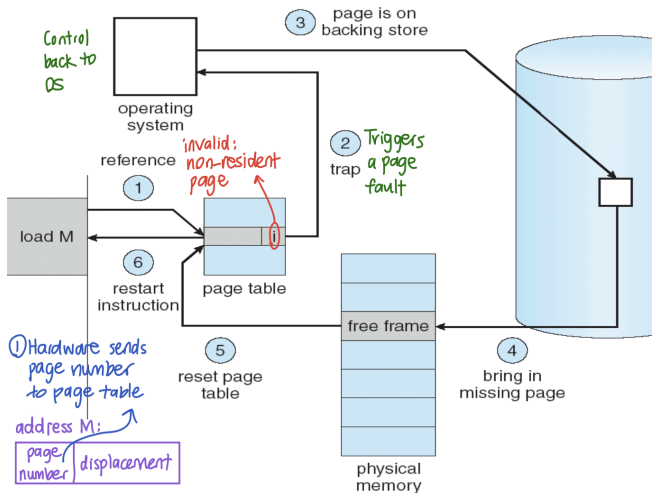
- **Page table** to translate **VIRTUAL** address to physical address
- Secondary Storage access time >> Physical memory access time (~50ns, which is long)

- **Thrash**: Poor performance of virtual memory / paging system when same pages loaded repeatedly due to lack of main memory
- If memory access results in **page fault** most of the time → Non-memory resident pages need to be loaded → **THRASHING** (constant state of paging and page faults)

VIRTUAL MEMORY

- Completely separate logical memory address from physical memory
  - Amount of physical memory no longer restrict the size of logical memory address
- **More efficient use of physical memory**. Page currently not needed can be on secondary storage
- Allow more processes to reside in memory → CPU has more processes to choose to run → Less likelihood of CPU being idle → **Improve CPU utilization**
  - *When waiting for I/O → Cannot use CPU, runs another process → Will run idler process if it cannot run anything → Burns CPU cycles*

Virtual Memory Accessing



1. [Hardware] Check the page table. Use a ***(is memory resident?)*** bit in the page table entry to check if page X is a **memory resident (in physical memory)**.
   - Yes → Access physical memory location → Hardware performs address translation and gets data from memory → Software sets up page table. Software allocates space for the page table and points the hardware page table register to the base of the page table. DONE.
   - No, page X is a **non-memory resident (in secondary storage)**. Continue to the next step.

2. [Hardware] **PAGE FAULT (when CPU tries to access non-memory resident page). Trap** to OS (OS is informed).
   - Page fault is an exception generated by *hardware (invokes page fault handler, which is part of OS)*, called by software (OS) through the interrupt service routine mechanism

3. [OS] Locate page X in **secondary storage**.

4. [OS] Load page X into a physical memory frame. (Since the CPU can only access memory resident pages).

5. [OS] Update page table.

6. [OS] Go back to step 1 to retry. (where hardware takes over)

Virtual Memory and Locality

Most programs exhibit these behaviors:

- Most time are spent on a relatively small part of **code** only
- In a period of time, accesses are made to a relatively small part of **data** only

⇒ Actual code and data used occupy only a small portion of memory.
⇒ **Locality principles**

Exploiting **Temporal Locality**

(*Memory address which is used is likely to be used again*):

- After a page is loaded to physical memory, it is likely to be accessed in the near future.
  - Cost of loading page is **amortized**

Exploiting **Spatial Locality**

(Memory addresses close to a used address is likely to be used)

- A page contains contiguous locations that are likely to be accessed in the near future.
  - Later access to nearby locations will not cause page fault

Most pages are between 8KB to 32KB. If instructions are 4 bytes each, then a page can hold 2048 instructions. Due to locality, most of the instruction accesses are likely within one page. Hence, no need to swap pages in and out very frequently. Can make use of disk drive as additional memory without incurring too much overhead.

Note: Exceptions where program behaves badly due to poor design / malicious intent

Demand Paging

- Process starts with **no memory resident page**. All in virtual memory.
- Only allocate a page when there is a **page fault**.
  - FYI: .exe files gives info about memory usage and starting point of program. Based on the starting point, OS can calculate the starting page for the process. OS can load the starting page into main memory before starting it, avoiding the initial page fault.

Pros:

- Fast startup time for new processes. (vs Large number of pages to allocate and initialise → Large startup cost, occupy more space in main memory, can only run fewer processes)
- Small memory footprint. (reduce footprint of process in physical memory so that more processes can be memory resident)

Cons:

- Process may appear sluggish at the start due to multiple page faults
- Page faults may have cascading effect on other processes ie **thrashing**

7.5.2 Page Table Structure (to reduce page table overhead)

Page table (present for every process) information is kept with the process information and **takes up physical memory space**

- Huge logical memory space → Huge number of pages
- Each page has a page table entry → Huge page table
- Problems with huge page table
  - **High overhead**. **Fragmented page table** (Page table occupies several memory pages).

DIRECT PAGING : Keep ALL entries in a SINGLE table

1. Page size = Frame size = 2^d **bytes**
   ⇒ Displacement = $\log_2$(frame_size) = d bits
2. Number of pages = Number of page table entries (PTEs)
   = Virtual address (byte-addressed) ÷ Page size
3. With 2^p **pages (**and virtual address of **b bits)** in logical memory space
   ⇒ b - d = p bits to specify one unique page
   (Since memory address = p-bit page number + d-bit displacement)
4. 2^p **pages** and 2^p **page table entries (PTEs)**, each with
   - Physical frame number
   - Additional information bits (valid/invalid, access rights etc)
5. If size of each page table entry = *x* bytes
   ⇒ **Page table size** = 2^p × *x* bytes

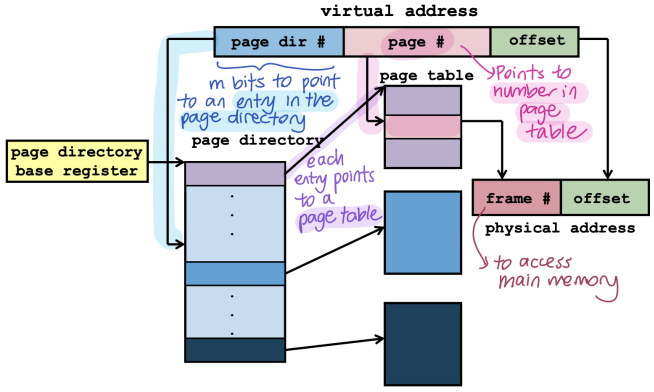
Example:

- Virtual address = 32 bits
- Page size = 4KB = 2^{12} bytes or $\log_2(4096)$ = 12 bits for displacement
- p = 32 - 12 = 20 bits to specify one unique page
- If size of PTE = 2 bytes ⇒ Page table size = 2^{20} × 2 bytes = 2 MB

2-Level Paging

- Motivation: With direct paging, not all processes use the full range of virtual memory space → Full page table is a waste
- **Split the full page table into regions.** Only a few regions used.
  - As memory usage grows, new region can be allocated.
  - Similar to splitting logical memory space into pages.
- Need directory to keep track of regions (like page table ↔ pages)
  - Each table entry points to the base address of the next page table
  - Further extendable to multilevel paging

- Split page table into **smaller page tables**, each with a **page table number**
- If the original page table has 2^P entries:
  - With 2^M smaller page tables, M bits is needed to uniquely identify one page table
  - Each smaller page table contains $2^P / 2^M = 2^{(P-M)}$ **entries**
- To keep track of the smaller page tables,
  - Need a single page directory with 2^M indices to locate each of the smaller page tables

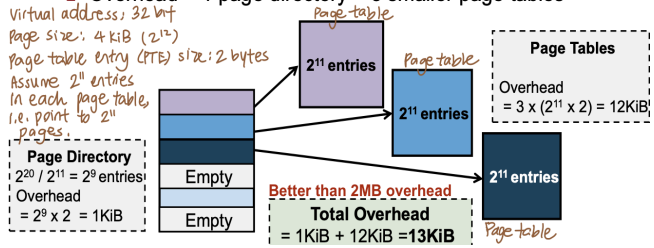


Example:

- Virtual address = 7 bits
- Page table size = 4 bytes (⇒ 2 bits offset)
- Number of pages = $2^{7-2} = 2^5$ = 32 pages
  ⇒ P = 5 entries in original page table
- If each smaller page table has 4 entries
  ⇒ Number of page tables = 32/4 = 8 = 2^3 ⇒ M = 3

Advantages of 2-Level Paging

- Page table structure can grow beyond the size of a frame
- Can have empty entries in the page directory, the corresponding page tables need not be allocated
  - Assume only 3 page tables is in used
  - Overhead = 1 page directory + 3 smaller page tables

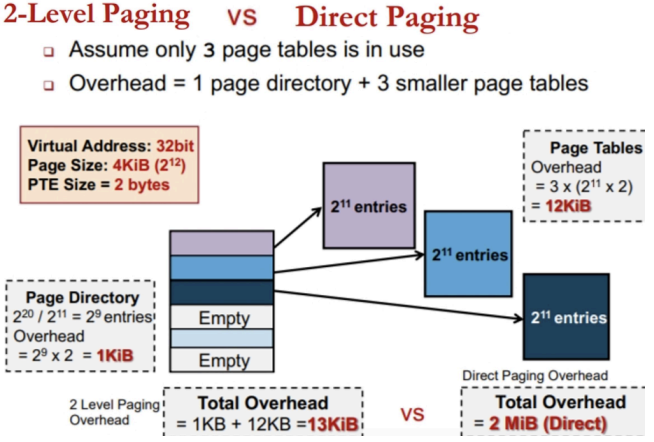


32-bit address

- 9 bits for directory
  - Page directory has 2^9 = 512 entries
  - Out of 512, only 3 entries active
- 11 bits
  - 2^{11} entries. So 11 bits for page number
- Last 12 bits
  - One page = 4 KiB

Overhead calculation for 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.



Disadvantages of 2-Level Paging

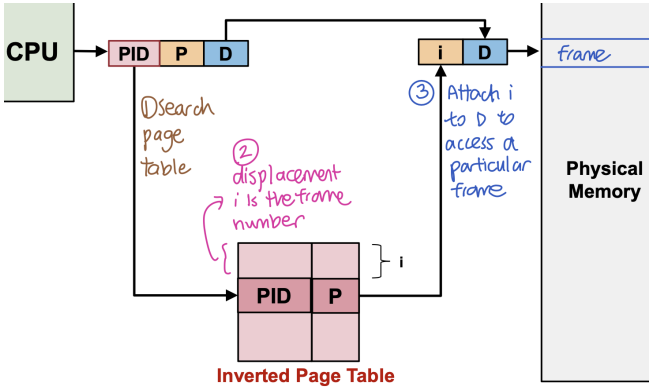
- Requires two serialized memory accesses just to get the frame number (directory + page table), only then to access data

Use TLB? TLB hits eliminate page-table accesses, but TLB misses experience *longer page-table walks (traversal of page-tables in hardware)*.

Inverted Page Table

Motivation: Page table is a per-process information
- M processes in memory ⇒ M independent page tables
- Observation: Only N physical memory frames can be occupied
  - Out of the M page tables, only N entries is valid
  - Huge waste: N << Overhead of M page tables

⇒ Keep a **single** mapping of physical frame to **<pid, page#>**
- pid = process id
- page# = page number, ***not unique*** among processes
- pid + page# can uniquely identify a memory page
- Array with the same size at the number of frames, with each entry storing **pid** and **page#**



Normal page table	Inverted page table
Entries ordered by page number	Entries ordered by frame number
To lookup page X, simply access the Xth entry	To lookup page X, need to search the whole table

Advantages:
- Fast lookup by frame: Often to support RAM management operations
- Ease of use: Frame management is a lot easier, one table for all processes

Disadvantage: Slow translation. From page number to frame number, since we need to search through the entire table for every memory access. However, normally 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 (to reduce page fault)

If there is no free physical memory frame during a **page fault** → Need to **evict (free) a memory page**:
- **Clean page**: Not modified → No need to write back
- **Dirty page**: Modified → Need to write back

Memory Reference Strings

- In actual memory reference: Logical Address = Page Number + Offset
- For page replacement algorithm, only page number is important

⇒ **Memory reference string**: Sequence of page numbers (eg 3,2,1,0,3,4)

Memory Access Time

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

- p = probability of page fault
- T_{mem} = access time for memory resident page
- T_{page_fault} = access time if page fault occurs
- $T_{page\_fault} >> T_{mem}$ ⇒ Reduce p to keep T_{access} reasonable.

⇒ Good algorithm should **reduce the total number of page faults**

Optimal Page Replacement (OPT) Algorithm

Replace the page that **will not** be used again for the **longest period of time**. **Guarantees minimum number of page faults.**
- ***BUT** need **future knowledge** of memory references*
- Useful as a base of comparison for other algorithms. Closer to OPT means a better algorithm.

Example: Optimal Page Replacement (OPT) Algorithm (6 page faults)

Time	Memory Reference	Frame A	Frame B	Frame C	Next Use Time A	Next Use Time B	Next Use Time C	Fault?
1	2	2			3	∞	∞	Y
2	3	2	3		3	9	∞	Y
3	2	2	3		6	9	∞	
4	1	2	3	1	6	9	∞	Y
5	5	2	3	5	6	9	8	Y
6	2	2	3	5	10	9	8	
7	4	4	3	5	∞	9	8	Y
8	5	4	3	5	∞	9	11	
9	3	4	3	5	∞	∞	11	
10	2	2	3	5	12	∞	11	Y
11	5	2	3	5	∞	∞	11	
12	2	2	3	5	∞	∞	∞	

FIFO Page Replacement Algorithm

Evict **oldest** memory page (based on **LOADING TIME**, not access time)
- OS maintains a **queue** of resident page numbers. Remove the first page in the queue if replacement is needed. Update the queue during page fault trap
- Simple to implement. No need hardware support.

Example: FIFO Page Replacement Algorithm (9 page faults)

Time	Memory Reference	Frame A	Frame B	Frame C	Loaded at Time A	Loaded at Time B	Loaded at Time C	Fault?
1	2	2			1			Y
2	3	2	3		1	2		Y
3	2	2	3		1	2		
4	1	2	3	1	1	2	4	Y
5	5	5	3	1	5	2	4	Y
6	2	5	2	1	5	6	4	Y
7	4	5	2	4	5	6	7	Y
8	5	5	2	4	5	6	7	
9	3	3	2	4	9	6	7	Y
10	2	3	2	4	9	6	7	
11	5	3	5	4	9	11	7	Y
12	2	3	5	2	9	11	12	Y

Disadvantages:
FIFO does not exploit **temporal locality** and does not take into account the actual usage pattern. Hence, FIFO violates the intuition that if the number of physical frames increases (e.g. more RAM), the number of page faults should decrease.
⇒ **Belady's Anomaly: Increased number of frames causes increased number of page faults.**

Least Recently Used (LRU) Page Replacement

- [**Temporal locality**] Replace page that has **not been used (accessed) in the longest time**.
- Expect a page to be reused in a short time window
  - Have not used for some time → most likely not used again
- Good approximation with the optimal Optimal Page Replacement (OPT) Algorithm
- Does not suffer from Belady's Anomaly

Example: Least Recently Used (LRU) Page Replacement (7 page faults)

Time	Memory Reference	Frame			Last Use Time			Fault?
		A	B	C				
1	**2**	**2**			1			Y
2	**3**	2	**3**		1	2		Y
3	**2**	**2**	3		3	2		
4	**1**	2	3	**1**	3	2	4	Y
5	**5**	2	**5**	1	3	5	4	Y
6	**2**	**2**	5	1	6	5	4	
7	**4**	2	5	**4**	6	5	7	Y
8	**5**	2	**5**	4	6	8	7	
9	**3**	**3**	5	4	9	8	7	Y
10	**2**	3	5	**2**	9	8	10	Y
11	**5**	3	**5**	2	9	11	10	
12	**2**	3	5	**2**	9	11	12	

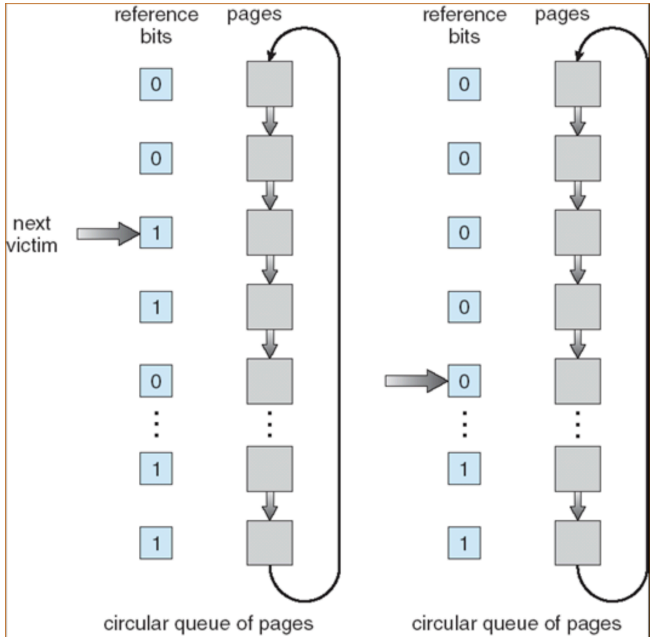
Not easy to implement LRU, need to keep track of the "last access time", need substantial hardware support:

- **Option 1: Use a counter**
  - Logical "time" counters, incremented for every memory reference
  - Page table entry has a "time-of-use" field
    - Store the time counter value whenever reference occurs
    - Replace the page with smallest "time-of-use" (but deletion is O(n))
  - Problems:
    - Need to search through all pages
    - "Time-of-use" is forever increasing (overflow possible)
- **Option 2: Use a Stack**
  - Maintain a stack of page numbers
  - If page X is referenced
    - Remove from the stack (for existing entry)
    - Push on top of stack
  - Replace the page at the bottom of the stack
    - No need to search through all entries
  - Problems:
    - Not a pure stack as the entries can be removed from anywhere in the stack. In a real stack (LIFO), we remove the most recently added element.
    - Hard to implement in hardware

Second-Chance Page Replacement (CLOCK)

- **Modified FIFO** to give a second chance to pages accessed
- Each PTE maintains a "**reference bit**"
- When all pages has **reference bit == 1 (accessed)** ⇒ Degenerate into FIFO algorithm

Implementation:



- **Circular queue** of page numbers, with a pointer pointing to the oldest page (**victim page**)
- To find a page to be replaced, advance until a page with reference bit == 0. Clear the reference bit to 0 as the pointer passes through.
- Algorithm
  1. Oldest FIFO page selected.
  2. When loading a page entry, set **reference bit == 0 (not accessed)**.
  3. Upon accessing the page entry, set **reference bit == 1 (accessed)**
  4. When a page replacement is required, check the current victim page (oldest FIFO page)
     a. If **reference bit == 0** → replace it.
     b. If **reference bit == 1** → give the page a second chance. Clear reference bit to 0 and move the pointer to the next page
  5. Repeat step 4 until a victim page with reference bit == 0 is found.

Example: Second-Chance Page Replacement (CLOCK) (6 page faults)

Time	Memory Reference	Frame (with `Ref Bit`)			Fault?
		A	B	C	
1	**2**	▶**2** (0)			Y
2	**3**	▶**2** (0)	**3** (0)		Y
3	**2**	▶**2** (1)	**3** (0)		
4	**1**	▶**2** (1)	**3** (0)	**1** (0)	Y
5	**5**	**2** (0)	**5** (0)	▶**1** (0)	Y
6	**2**	**2** (1)	**5** (0)	▶**1** (0)	
7	**4**	▶**2** (1)	**5** (0)	**4** (0)	Y
8	**5**	▶**2** (1)	**5** (1)	**4** (0)	
9	**3**	**2** (0)	**5** (0)	▶**3** (0)	Y
10	**2**	**2** (1)	**5** (0)	▶**3** (0)	
11	**5**	**2** (1)	**5** (1)	▶**3** (0)	
12	**2**	**2** (1)	**5** (1)	▶**3** (0)	

▶ Victim Page

7.5.4 Frame Allocation (affects page fault)

Distribute N physical memory frames among M processes

- **Equal allocation**
  - Each process gets N/M frames
- **Proportional allocation**
  - Processes are different in size (memory usage)
  - Each process gets $(size_p / size_{total}) \times N$ frames
    - $size_p$ = size of process p
    - $size_{total}$ = total size of all processes

Local (Page) Replacement

Victim page are selected **among pages (from a frame given to the process) of the process** that causes page fault (implicit assumption for page replacement algorithms discussed)

Pros:

- Frames allocated to a process remain constant → Performance is **stable between multiple runs**. Avoids the situation where one process causes so many page faults such that it eats up the frames of every other process.

Cons:

- If frame allocated is not enough → Hinder the progress of a process.
- **Insufficient physical frame** → Process triggers many page faults as you only replace the pages that belong to the process rather than increasing the number of frames → **THRASHING in the process, heavy I/O to bring non-resident pages into RAM**, hogs disk drive, difficult for other processes to read and write to disk drive
- **Thrashing** can be limited to one process, but that single process can hog the I/O and degrades the performance of other processes

Global (Page) Replacement

Victim page chosen **among all physical frames**. Process P can take a frame from Process Q by evicting Q's frame during replacement

Pros:

- Allow **self-adjustment** between processes
  - Process that needs more frame can get from other

Cons:

- **CASCADING THRASHING: A thrashing process steals a page from other process, resulting in other processes also thrashing**
- Badly behaved (malicious) process can affect others
- Frames allocated to a process can be different from run to run (inconsistent performance)

Working Set Model: Finding the right number of frames

Motivation:

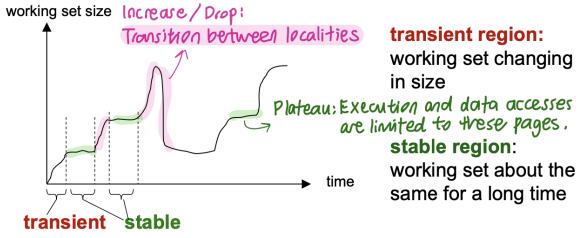
- **LOCALITY**: The **set** of pages referenced by a process is relatively constant in a period of time (in the short run)
- However, as time passes, the set of pages can change
- When a function is executing, the references are likely on local variables, parameters, code in that function.
  - These pages define the locality for the function.
  - After the function terminates, the references will change to another set of pages.

⇒ In a **new locality**, a process will cause **page fault** for the **set** of pages. With the **set** of pages in frame, **no/few** page faults until the process transits to new locality.

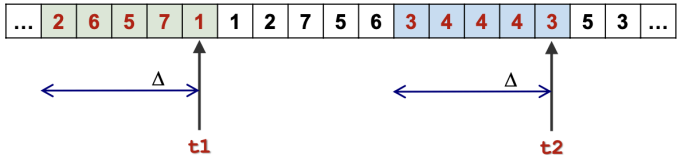
Working Set Model $W(t, \Delta)$

$W(t, \Delta)$ = active pages in the interval at time t

- Allocate **enough frames for pages in $W(t, \Delta)$ to reduce possibility of page fault**
- **Δ : Working Set Window** (an interval of time) (*always in the past*)
  - Directly affects the accuracy of Working Set Model
    - **Too small** Δ → May **miss** pages in the **current** locality
    - **Too big** Δ → May **contain** pages from **different** locality



Example memory reference strings:



- Assume Δ = an interval of 5 memory references
- $W(t1, \Delta)$ = {1,2,5,6,7} (5 frames needed)
- $W(t2, \Delta)$ = {3,4} (2 frames needed)

7.6 Summary - Page Entry Bits

7.4.1 Disjoint Memory Allocation: Paging scheme

Access Right Bits [Memory protection between processes]:
Attached to each Page Table Entry to indicate writable, readable, and/or executable (*rwx*)

Valid Bit [Memory protection between processes]:
Attached to each page table entry to indicate if the page is valid for the process to access

Is-Shared Bit [Page Sharing]:
in the page table entry to track if the page is shared

7.5 Virtual Memory Management

Is-Memory Resident Bit [7.5.1 Extended Paging Scheme]:
In the page table entry to check if page X is a memory resident (in physical memory)

Dirty Bit [7.5.3 Page Replacement Algorithms):
Check if a memory page is a clean page (Not modified → No need to write back) or dirty page (Modified → Need to write back) before freeing it

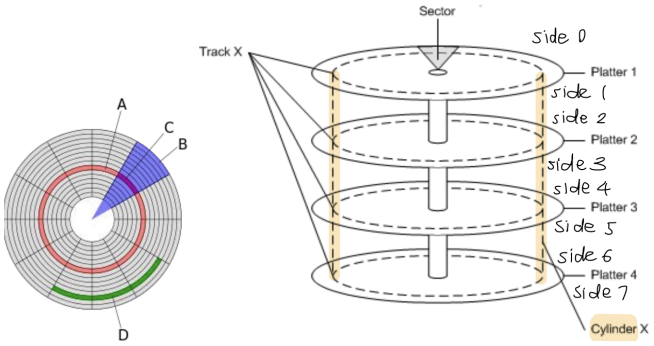
Reference Bit [Second-Chance Page Replacement (CLOCK)]:
Check if a page has been accessed. Replace it if reference bit == 0 (not accessed)

8. File System Management

Motivation:

- Physical memory is **volatile**
  - Use external storage to store **persistent** information
- Direct access to the storage media is **not portable**
  - Dependent on hardware specification and organization:

Hard Disk Layout:



- A = Track (one ring on its own)
- B = Geometric Sector
  (individual tracks are divided into sectors/blocks)
- C = Track Sector
- D = Cluster (Transfer units in >1 sector ⇒ Block)
  (>1 block ⇒ Logical block)

FILE SYSTEM provides

- An abstraction on top of the physical media
- A high level resource management scheme
- Protection between processes and users
- Sharing between processes and users

File System: General Criteria

- **Self-contained**
  - Information stored on a **media** is enough to describe the entire organization
  - Hence, should be able to "**plug-and-play**" on another system. If you unplug a drive from a computer and plug it into another computer, you can still access all the files.
- **Persistent**
  - Data persists beyond the lifetime of OS and processes. When you switch off a system and switch it back on, the contents in the drive should still be accessible.
- **Efficient**
  - Good management of free and used space
  - Minimum overhead for bookkeeping information

Memory Management vs File Management

	Memory Management	File System Management
Underlying storage	RAM	Disk (SSD: Solid state drive)
Access Speed	Constant	Variable disk I/O time
Unit of Addressing	Physical memory address	Disk sector / Block number (a group of sectors)
Usage	Address space for process. **Implicit** when the process runs. Determined by address translation mechanism, done purely by hardware.	Non-volatile data. **Explicit** access (intentionally request)
Organisation	**Paging / Segmentation**: determined by hardware and OS	Many different file systems: ext* (Linux), FAT* (Windows), HFS* (macOS)..

8.1 File System Abstraction

Abstraction: Hide away the idea of starting block number and length

8.1.1 File (abstract storage of data)

- Represent a logical unit of information created by process
- An **abstraction**, an **ABSTRACT DATA TYPE (ADT)** i.e. has a set of common operations with various possible implementation
- Contains:
  - Data: Information structured in some ways
  - **Metadata (file attributes)**: Additional info associated with file:

Name:	A human readable reference to the file
Identifier:	A unique id for the file used internally by FS (inode number)
Type: (most OS excludes this)	Indicate different type of files E.g. executable, text file, object file, directory etc
Size:	Current size of file (in bytes, words or blocks)
Protection:	Access permissions, can be classified as reading, writing and execution rights Mac/Unix: rwx rwx rwx (owner group others)
Time, date and owner information:	Creation, last modification time, owner id etc
Table of content:	(Windows: Fat (Linked list) Mac/Linux: Indexes) Information for the FS to determine how to access the file

File name

Different file system has different **naming rule** to determine valid file name

- Length of file name
- Case sensitivity
- Allowed special symbols
- File extension (usually `Name.Extension`)
  - Indicates file type in some file systems

File Type

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

- **Regular files**: Contains user information
  - **ASCII files** (eg ext file, programming source codes, etc)
    - Can be displayed or printed **as is**
  - **Binary files** (eg executable, Java class file, pdf file, mp3/4, png/jpeg/bmp)
    - Have a predefined internal structure that can be processed by specific program (JVM to execute Java class file, PDF reader for pdf file etc)
- **Directories**: System files for file system structure
  - Every directory, except the root directory, is a file.
- **Special files**: Character / Block oriented
  - Keyboard is a special file for stdin. Screen is a standard file for stdout. Can do redirection to make the files point to something else to get data from other places.

Distinguishing file type:

- Windows OS: Use **file extension** (.docx → Words document)
  - Change of extension implies a change in file type
- Unix: Use **magic number** (**embedded information** in the file)
  - Usually stored at the beginning of the file

File Protection

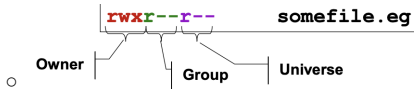
Controlled access to the information stored in a file.

- Types of access: **Read** (retrieve information), **Write**, **Execute** (load file into memory and execute it), **Append** (add new information to the end of the file), **Delete**, **List** (read metadata of a file)
- Most common is to restrict access based on user identity
- Most general scheme: **Access Control List**
  - A list of user identity and the allowed access types
  - Pros: Very customizable
  - Cons: Additional information associated with file

Permission bits classify users into 3 classes:

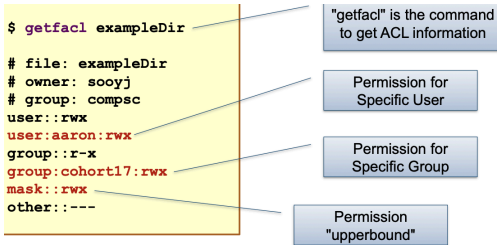
- Owner: User who created the file

- Group: A set of users who need similar access to a file
- Universe: All other users in the system
- E.g. Unix - define permission of 3 access types (Read / Write / Execute) for the 3 classes of users



Access Control List (ACL) (Unix)

- Minimal ACL (same as permission bits)
- Extended ACL (added named users / group)



8.1.2 File Data

Structure

- **Array of bytes** (traditional Unix view)
  - No interpretation of data: just **raw bytes**
  - Each byte has an unique **offset (distance)** from the file start
- **Fixed length records**
  - Array of records, can grow/shrink
  - Can jump to any record easily:
    Offset of the Nth record = size of Record * (N-1)
- **Variable length records**
  - Flexible but harder to locate a record

Access methods

- **Sequential Access**
  - Data read in order, starting from the beginning
  - Cannot skip but can be rewound
- **Random Access**
  - Data can be read in any order. Can be provided in two ways:
    1. Read(Offset) : Every read operation explicitly state the position to be accessed
    2. Seek(Offset) : A special operation is provided to move to a new location in file (used by Unix and Windows)
- **Direct Access**
  - Used for file contains **fixed-length records**
  - Allow **random access to any record directly**
  - Very useful where there is a large amount of records e.g. in database
  - Special case: Basic **random access** method where **each record == one byte**

8.1.3 File Operations

OS provides file operations as **SYSTEM CALLS**:

- Provide protection, concurrent and efficient access
- Maintain information

Operations on File Metadata

- **Rename**: Change filename
- **Change attributes**: File access permissions, dates, ownership etc
- **Read attributes**: Get file creation time

Operations on File Data

- **Create** a new file with no data
- **Open** a file to prepare the necessary information for file operations later
- **Read** the data from the file, usually starting from the current position
- **Write** data to the file, usually starting from the current position
- **Reposition (Seek)** the file to a new location (No actual read / write)
- **Truncate** removes data between specified position to end of file

Information kept for an open file

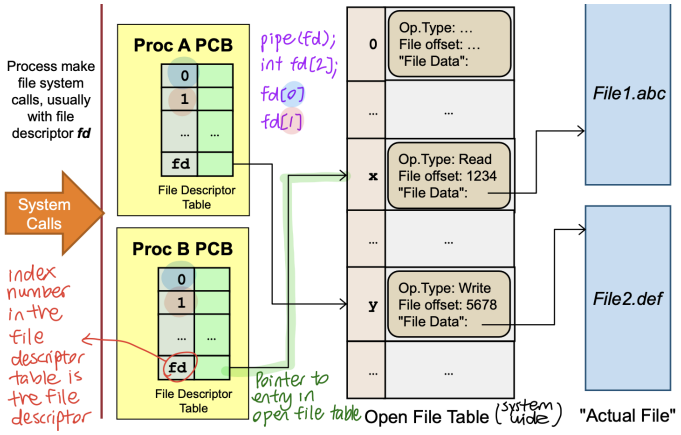
- **File Pointer**: Current location in file
- **Disk Location**: Actual file location on disk
- **Open Count**: How many processes has this file opened?
  - Useful to determine when to remove the entry in table

Approaches to organise the open-file information

- Several processes can open the same file
- Several different files can be opened at any time

System-wide open-file table

- 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.



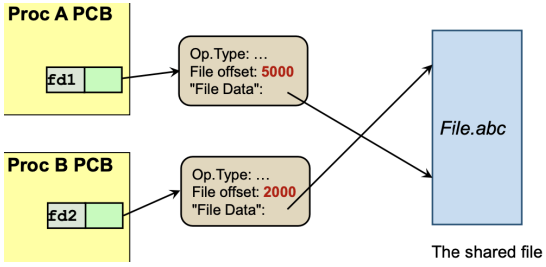
Per-process open-file table

- One entry per file used in the process.
- **File Descriptor Table**: Keeps track of open files for a process, also known as **file descriptors**
- Each entry points to the **system-wide table.**
- 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.

Process Sharing File in Unix: Cases

Two processes using **DIFFERENT file descriptors** to access the same file

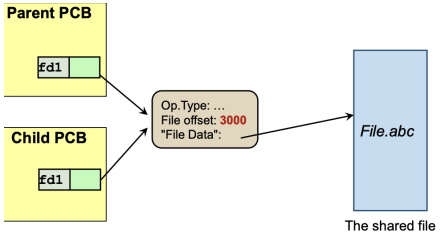
- Each process has its own entry in the **per-process open file table**, with each entry pointing to a separate entry in the **system-wide open file table.**
- This allows the processes to maintain ***INDEPENDENT file offsets***, so I/O operations in one process do not affect the offset in the other.



- Examples:
  - Two process open the same file
  - Same process open the file twice

Two processes using the **SAME file descriptor (after `fork()`)**

- Both processes share the same entry in the **per-process open file table**, which in turn points to a single entry in the **system-wide open file table**.
- Share the ***SAME file offset*** ⇒ I/O in one process affects the offset for the other process



- Example:
  - `fork()` after file is opened

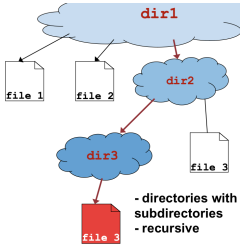
8.1.4 Directory (Folder) (organisation of files)

Provides the user with a logical grouping of files. Keeps track of files in the system.

Single-Level Directory

All files in the root directory

Tree-Structured Directory

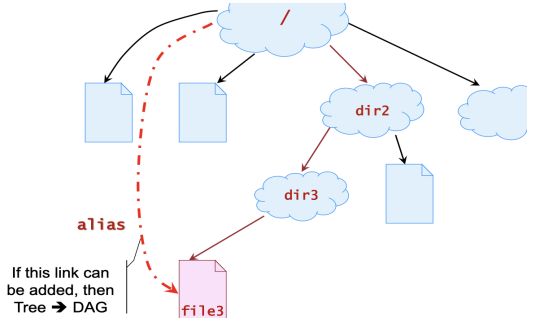


Directories can be recursively embedded in other directories.

- **Absolute Pathname**
  - Directory names followed from **root** directory "/ " to file
  - `/dir2/dir3/file3`
- **Relative Pathname**
  - Directory names followed from **current working directory** (CWD)
  - Eg CWD is dir2
    ⇒ `dir3/file3` OR `./dir3/file3` (where ***. is the CWD***)
  - `../dir2/dir3/file3`
    (.. → access through previous directory, parent of CWD)
  - CWD can be set explicitly or implicitly changed by moving into a new directory under shell prompt

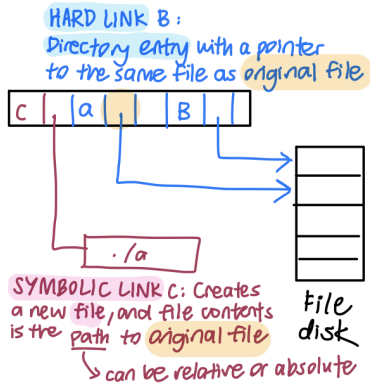
Directed Acyclic Graph (DAG) Directory Structure

If a **file *can be shared***, i.e. it appears in multiple directories with different path names, but refers to the same copy of actual content. ⇒ **DAG**



DAG: Unix Hard Link (directory entry with pointer to a FILE)

- If Directory A is the owner of file F and Directory B wants to share F…
- A and B have **separate pointers** point to the actual file F in disk
- Unix command: "`ln`"
- Pros: Low overhead, only pointers are added in directory
- Cons: When one directory deletes the file F
  - Delete link → Entry (eg A) is removed. File F is still there.
  - Remove last link → Data file F will also be removed.
  - File has *reference count*, tracks the number of links pointing to the file. Deleting a link from the directory to file F will also decrease the reference count by 1. When the reference count is 0, file F is deleted.

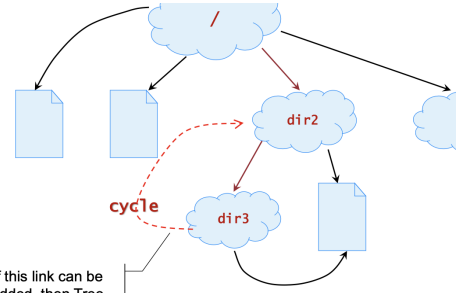


Symbolic links can point to **directories (** → can create **GENERAL GRAPH). Hard links cannot.**

DAG: Unix Symbolic Link (can link to DIRECTORIES)

- B creates a **special link file, G, which contains the path name of F**.
- When G is accessed, find out where F is, then access F.
- Unix command: "`ln -s`"
- Pros: Simple deletion.
  - If B deletes: G is deleted, not F. File F remains.
  - If A deletes: F is gone, G remains (but not working).
- Cons: Larger overhead, special link file G takes up actual disk place
- Symbolic link is not just a DAG, can be a full graph.

General Graph Directory Structure (not desirable)



Tree has a cycle.

- **Hard to traverse.**
  - Need to prevent infinite looping (A point to B, B point to A).
- **Hard to determine when to remove a file / directory** (since they point to each other)

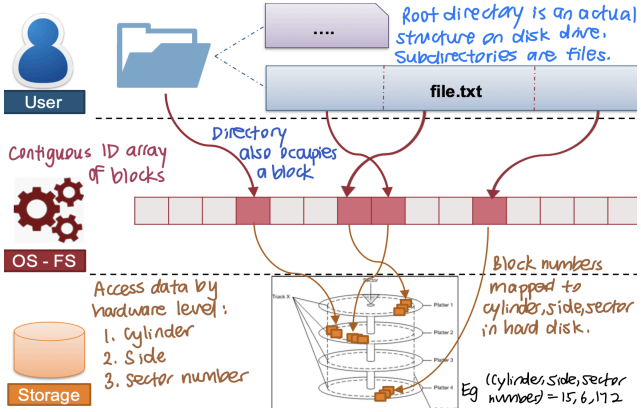
8.2 File System Implementation

File systems stored on storage media such as **hard disk**.

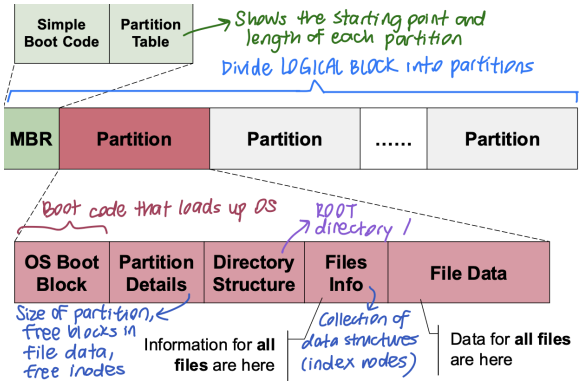
Disk Structure

- Can be treated as a 1-D array of **logical blocks**, with smallest accessible unit = 512-bytes to 4KB
- Logical block is mapped into **disk sector(s)**
  - Layout of disk sector is **hardware dependent**

User ↔ OS ↔ Hardware: Views



8.2.1 Disk Organisation



- Files Info + File Data: Refer to 8.2.2 File Implementation
- Partition Details: Refer to 8.2.3 Free Space Management
- Directory Structure: Refer to 8.2.4 Implementing Directory

- **Master Boot Record (MBR)** at sector **0** with **partition table**
  - Partition table → Know where the file system starts → Can access file systems (which contains all files, including directories)
  - Lose partition table = Lose all data
- Followed by one or more **partitions**
  - **Each partition** can contain an independent **file system**

A **file system** generally contains

- OS Boot-Up information
- Partition details: Total Number of blocks, Number and location of free disk blocks
- Directory Structure (info on root directory)
- Files Information (a set of data structures to help find data)
- Actual File Data

8.2.2 File Implementation (File Info and File Data) - *how to allocate* file data on disk

- Logical view of a file: A collection of **logical blocks**
- Disk space divided into fixed-size blocks, and when **file size != multiple of logical blocks** → Last block may contain wasted space i.e. **INTERNAL FRAGMENTATION**
- Good file implementation
  - **Keep track of the logical blocks**
    - Know where used blocks and free blocks are
    - Do not put another file in the actual data in the block as it is computationally expensive to keep track of files that start within the start block. Easier to assume file begin at the start of the block.
  - **Allow efficient access**
  - **Disk space is utilized effectively**
    - File system data should not occupy too much space
    - Block allocation should have efficient allocation and runtime

Contiguous File Block Allocation

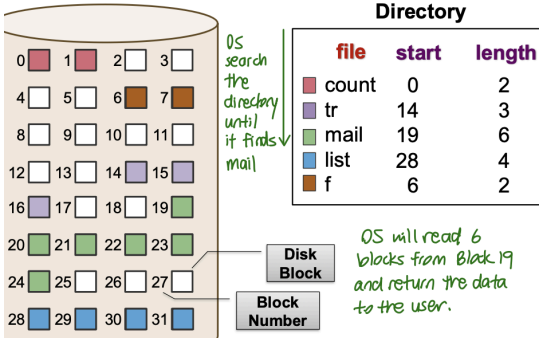
Allocate consecutive disk blocks to a file.

Pros:

- **Simple to keep track**. Each file only needs: Starting block number + Length.
- **Fast access** (only need to seek to first block)

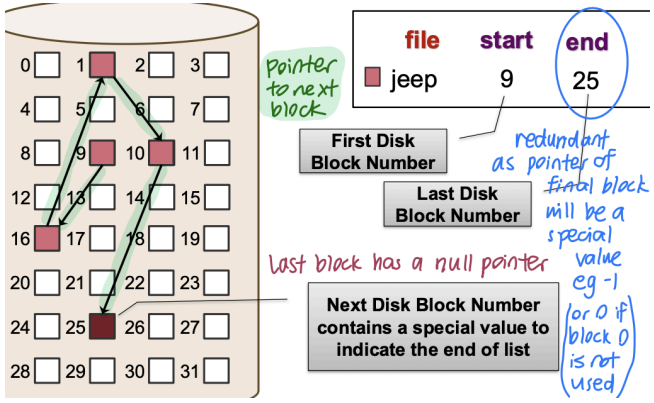
Cons:

- **External Fragmentation.** With each file as a variable-size "partition" - over time, with file creation/deletion, disk can have many small "holes", little pockets of free space that may not be big enough for the next file.
- File size needs to be specified in advance.



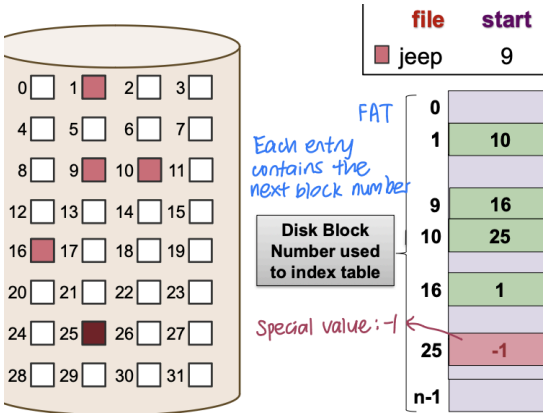
Linked List Block Allocation (stored in block)

- Keep **a linked list of disk blocks**. Each disk block stores the **next disk block number** (act as **pointer**) and actual file data.
- File information stores first and last disk block number
- Pros:
  - Solves fragmentation problems. **No external fragmentation.**
- Cons:
  - Random access in a file is **very slow** (O(n)), need to sequentially access all blocks.
  - Part of a disk block is used for pointer → Smaller disk block → **Less usable space**.
  - **Less reliable** if one of the pointer is incorrect (eg OS accidentally overwrite pointer data → corrupted pointer data)



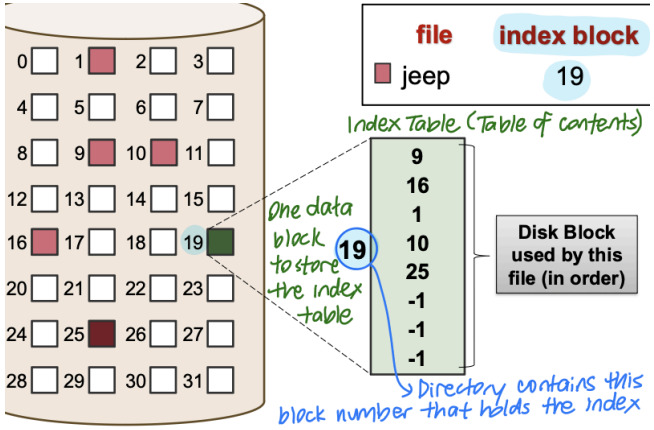
Linked List V2.0 (File Allocation Table, FAT)

- Move all block pointers into a single table (**File Allocation Table**).
- FAT is in memory all the time, not in disk drive
  - Following links in the FAT is much faster
- Pros:
  - **Faster random access**, linked list traversal now takes place in memory
- Cons:
  - FAT keeps tracks of **all disk blocks** in a partition → Can be huge when disk is large → **Consume valuable memory space** → Cannot run as many processes or result in thrashing.
  - Linked list can still become corrupted → Less reliable.



Indexed Allocation (Separate Index Block)

- Each file has an **index block** to store an **array** of disk block addresses. ⇒ `IndexBlock[N] == Nth Block address`
- Pros:
  - **Less memory overhead**. **Only the index block of the opened file** needs to be in memory (compared to whole FAT). No need to maintain information of all blocks in the disk drive.
  - **Fast direct access.** No need for traversal.
- Cons:
  - **Limited maximum file size**. Maximum number of blocks == Number of index block entries
  - **Index block overhead**: Consumes space, keep updated.

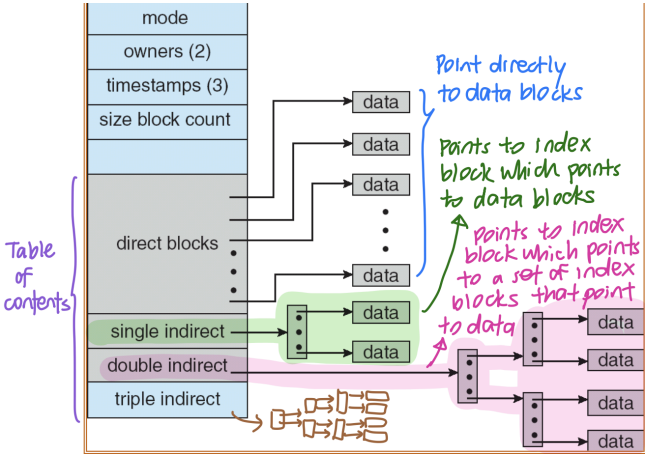


Variations of Indexed Block Allocation

- Several schemes to allow larger file size
- **Linked scheme**
  - Keep a **linked list** of index blocks
  - Each index block contains the pointer to next index block
- **Multilevel index** (similar to **multi-level paging)**
  - First level index block points to a number of **second level index blocks**. Each second level index block points to the actual disk block. Can be generalised to any number of levels.
- **Combined scheme (direct indexing + multi-level index)**

  Unix I-Node (Indexed Node):

  Eg 512 block numbers in each block → 512 pointers per block.
  - 12 **direct pointers** that point to disk block directly
  - 1 **single indirect block** which contains a number of direct pointers → 512 data blocks
  - 1 **double indirect block** which points to a number of single indirect blocks → 2 level index, points to 512 index blocks, and each index block points to 512 data blocks. Hence 512^2 data blocks.
  - 1 **triple indirect block** which points to a number of double indirect blocks → 512^3 data blocks.
  - A combination of efficiency (for small file) and flexibility (still **allow large file** eg size = $12 + 512 + 512^2 + 512^3$)



8.2.3 Free Space Management

Free space list to maintain free space information and know which disk block is free so as to perform file allocation.

- Allocate:
  - Remove free disk block from free space list
  - Needed when file is created or enlarged (appended)
- Free:
  - Add free disk block to free space list
  - Needed when file is deleted or truncated

Free Space Management: BITMAP

- Each disk block is represented by 1 bit
  - 1 == free, 0 == occupied

    `0 1 0 1 1 1 0 0 1 0 1 1 ...`
    - Occupied Blocks = **0, 2, 6, 7, 9, …**
    - Free Blocks = **1, 3, 4, 5, 8, 10, 11, …**
- Pros: **Easy to manipulate**. Use simple bit operations to find first free block or n-consecutive free blocks
- Cons: Need to **keep in memory** for efficiency reasons. Bitmap can become very big.

Free Space Management: LINKED LIST

- Use a linked list of disk blocks: Each disk block contains:
  - A number of free disk block numbers
  - A pointer to the next free space disk block
- Pros:
  - **Easy to locate free block**. Just use the first block.
  - **Only the first pointer is needed in memory** (though other blocks can be cached for efficiency)
- Cons:
  - **High overhead**. But can be mitigated by storing the free block list in free blocks.
- Runs alternative: If the free space tends to be in contiguous runs, can instead store first index + length. But may not work as well as fragmentation worsens.

8.2.4 Implementing Directory

Directory Structure

- Main tasks:
  - Keep tracks of the files in a directory, possibly with the file metadata
  - Map the file name to the file information
- Note: **File must be opened before use**. Eg `open( "data.txt" );`
  - Open operation: Locates file info using pathname + file name
- **Path name**: List of directory names traversed from root
  - Given a full path name, need to recursively search the directories along the path to arrive at the file information.
    - Full path name: `/dir2/dir3/data.txt`
      1. Find "dir2" in directory "/"
         - Stop if not found (or incorrect type)
      2. Find "dir3" in directory "dir2"
         - Stop if not found (or incorrect type)
      3. Find "data.txt" in directory "dir3"
         - Stop if not found (or incorrect type)
- **Sub-directory**: Usually stored as file entry with special type in a directory

Directory Implementation: LINEAR LIST

- Directory consists of a list. Each entry represents a file:
  - Store file name (minimum) and possibly other metadata
  - Store file information or pointer to file information
- Locating a file using a list requires a **linear search**.
  - **Inefficient** for large directories and/or deep tree traversal, eg search all entries to see if file exists.
  - Solution: Use **cache** to remember the last few searches. Users usually move up/down a path.

Directory Implementation: HASH TABLE

- Each directory contains a hash table of size N.
- To locate a file by file name:
  - File name is hashed into index ***K*** from ***0*** to ***N-1***
  - `HashTable[K]` is inspected to match file name
    - Usually **chained collision resolution** used, i.e. file names with same hash value is chained together to form a linked list with list head at `HashTable[K]`
- Pros:
  - **Fast lookup**. Good hash function → File names evenly distributed across all entries → Each table points to a very short linked list → More efficient search.
- Cons:

- Hash table has limited size. **Smaller hash table → More collisions, longer linked list.**
- Depends on good hash function.

Directory Implementation: FILE INFORMATION

- File information consists of:
  - File name and other metadata
  - Disk blocks information
- Approaches:
  - **Store everything in directory entry**. Can have a fixed size entry where all files have the same amount of space for information.
  - **Store only file name and pointer** which points to some data structure for other info

8.2.5 Walkthrough on File Operation

- Previous sections are on **static information** for a file system stored on media so that they are persistent (eg free list, free space, in iNode)
- At runtime, when a user interacts with a file, **run-time information** is needed. Maintained by OS in memory (OS context: information about the file we are using)
- [Recap] Common in-memory information:
  - System-wide open-file table: Contains a copy of file information for each open file + other info.
  - Per-process open-file table: Contains pointer to system-wide table + other info
  - Buffers (⇒ efficiency) for disk blocks read from/written to disk

Walkthrough on file operation: *Create*

To create a file */…/…/parent/F* :

1. Use full pathname to locate the PARENT directory.

- Search for filename F to avoid duplicates.
  - If found, file creation terminated with error.
    - Cannot have 2 files with the same name.
  - Search could be on the cached directory structure, which is cached in memory. Directory is stored in media for it to be persistent, but as we load up files, we can store parts of the directory in **memory**.

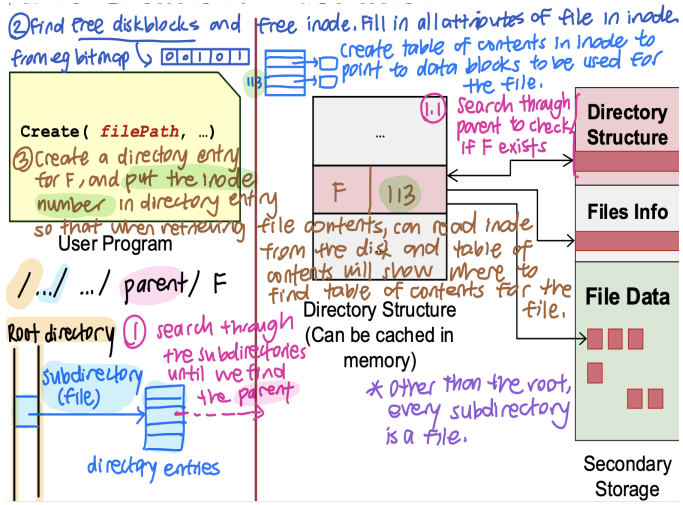
2. Use free space list (bitmap in Linux) to find free disk block(s).

- Depends on the allocation scheme. When using file allocation table, the blocks do not have to be contiguous, hence there is no problem of external fragmentation.

3. Add an entry to the PARENT directory.

- With relevant file information such as file name, disk block information etc.

File CREATION illustration:

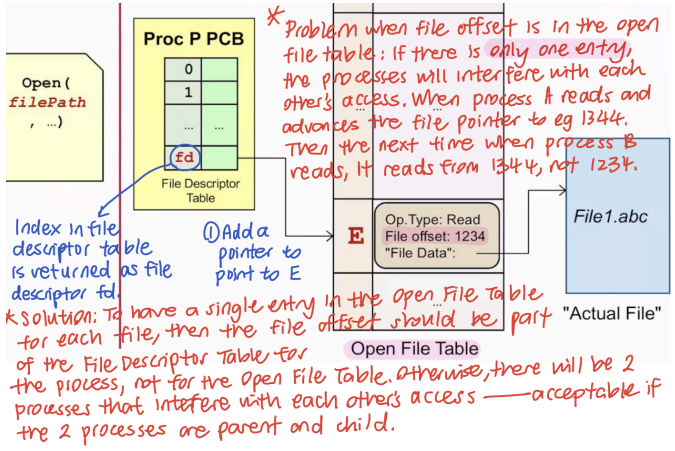


Walkthrough on file operation: *Open*

Process P open file */…/…/…/F* :

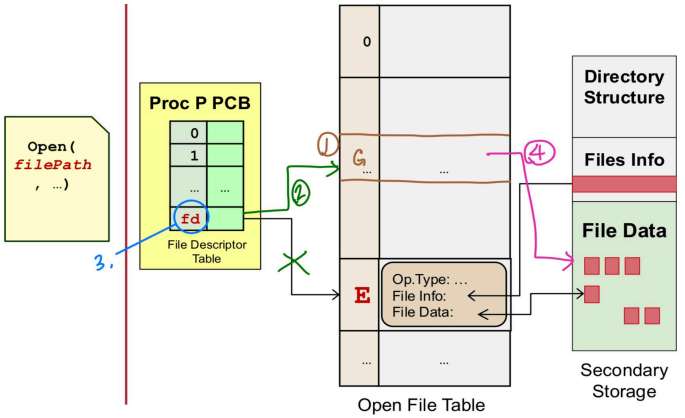
ALTERNATIVE 1. Search SYSTEM-WIDE TABLE for *existing* entry E.

- We have one entry in the open-file table for each file.
- If found → Creates an entry in P's table to point to E. Return a pointer to this entry.
- If not found, continue to the next step (**ALTERNATIVE 2**).



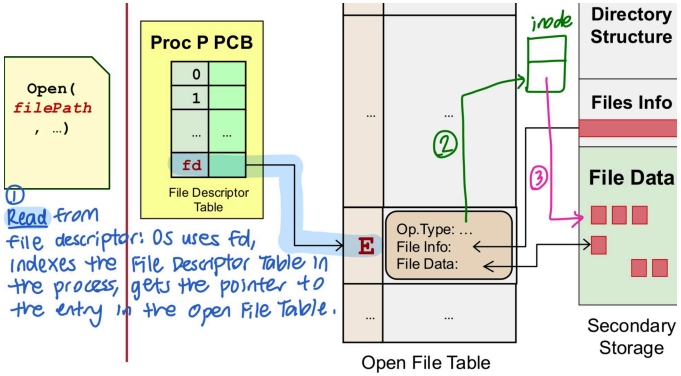
ALTERNATIVE 2. Use full pathname to locate file F.

- If not found, open operation terminates with error.
- When F is located, its file information is loaded into a **new entry** E in the **system-wide table**.
  - Creates an entry in P's table to point to E.
  - Return a pointer to this entry in the file descriptor table. This is the file descriptor we get when we do the ***open*** operation.
- Returned pointer is used for further read/write operation.



1. When new process P opens the file, this creates a new entry in the open file table, eg Entry G - with operation type, file offset etc. As there is now a separate file table entry for process P, process P will now have its own file offset that does not interfere with the original process that opened the file.
2. Process P File Descriptor Table would point to the new Open File Table entry G instead of E.
3. The index of the File Descriptor Table, fd, would be returned as the file descriptor that allows us to access the file.
4. The new Open File Table entry G would also allow access to the file data of the same file as the initial entry E.

What can we do with the file descriptor?



1. Read from file descriptor. OS uses fd, indexes the File Descriptor Table in the process, gets the pointer to the entry in the Open File Table.
2. Use, for example the inode structure, and table of contents, to find the correct disk block to read from. For example, if we read from byte number 11455, and suppose each block is 4KB. The block that we want to read would be in the 3rd data block (as the 1st data block is from bytes 0 to 4095, the second block is from bytes 4096 to 8191, and the 3rd data block covers from bytes 8192 to 12000+).
3. To read from byte number 11455, make use of the Open File Table entry which would have cached the inode. Look at the block number from the 3rd data block, and use that to directly access the data block in the Secondary Storage / drive.

Writing is similar.

1. Use the file descriptor, fd, as the index to the process' File Descriptor Table, find the pointer to the Open File Table.
2. Make use of the "File Data", where we cache the iNode, to find out which data block to access to read or write based on the byte offset.