Introduction

- Operating System: Program that acts as an intermediary between a user and the hardware

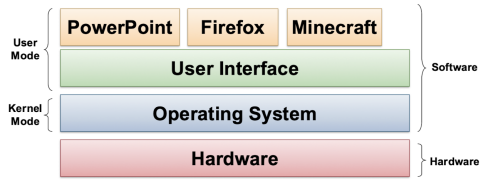
History of OS

- No OS: Programs directly interact with hardware (e.g., plugging wires)
  - Pros: Minimal overhead
  - Cons: Clunky, inefficient
- Batch OS: Executes 1 program (aka job) at a time
  - Used for mainframes: Accepts programs via punch cards
    - Minicomputer: Mini version of mainframe (e.g., Unix)
  - Batch processing: Done in a huge batch (Lots of punchcards -> 1 big tape input -> 1 big tape output)
  - Cons: Inefficient (CPU idle when performing I/O)
- Time-Sharing OS: Allows users to interact with machine using terminals (teletypes)
  - User job scheduling: By switching between apps quickly, creates illusion of concurrency on 1 processor
  - Memory management
  - **Virtualization** of hardware: Each user program (UP) executes as if it has all the resources to itself
- OS on Personal Computer: Machine dedicated to user, not timeshared between multiple users (since it's more affordable)
  - Windows model: 1 user at a time, but can have more than 1 user
  - Unix model: 1 user at machine, but other users can access remotely

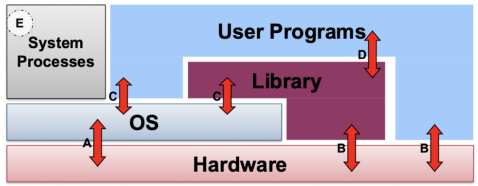
Motivations of OS

- Abstraction: Despite large variation in hardware configurations, OS abstracts away the hardware details and functionalities, so that users can perform essential tasks
- Resource allocator: Since running programs need many resources (e.g., CPU, memory, I/O)
- Control program: Prevents errors and hackers

OS Structure



- OS is a software, just that it runs in kernel mode (i.e., Access to all hardware resources)
- Other software runs in user mode (i.e., Limited access to hardware resources)



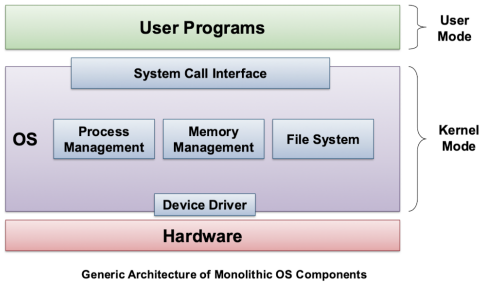
- A, B: Execute machine instructions
- C: Call OS using **system call interface**
- D: UPs call library code (e.g., data structure libraries)
- System processes: Usually part of OS

OS as a Program

- Kernel: Program with some special features:
  - Handles hardware issues
  - Provides system call interface
  - Special code for interruption handlers, device drivers
- Kernel code is different from normal programs, since it has no OS to rely on (i.e., no system calls)
- Many ways to structure an OS

Monolithic OS

> Kernel is 1 big program

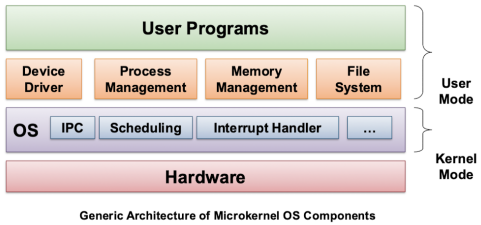


Generic Architecture of Monolithic OS Components

- Good SWE principles still possible with modularization and separation of interfaces
- E.g., DOS, Windows 9x
- Pros: Comprehensive, good performance
- Cons: Highly coupled (1 crash may crash entire kernel), complicated
- Device driver: Developed by hardware provider, so how can we handle incompatibilities?

Microkernel OS

> Kernel is very small and provides only basic functionalities



Generic Architecture of Microkernel OS Components

- Higher-level OS services built **outside** of kernel
- Pros: Robust, extensible, protection between kernel and high-level services (Fewer "blue screen of death")
- Cons: Slow

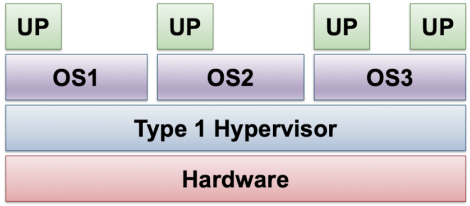
Virtual Machines

> Software emulation of hardware

- Motivation:
  - Since OS assumes total control of hardware, what if we want to run multiple OSes on same hardware at the same time (e.g., cloud computing)
  - OS hard to debug and monitor
- Virtualization of hardware: Similar to virtualization done *by* OS, but now *for* OS
  - OS can run on top of VM
- Created and managed by **Hypervisor** (aka Virtual Machine Monitor (VMM))

Type 1 Hypervisor

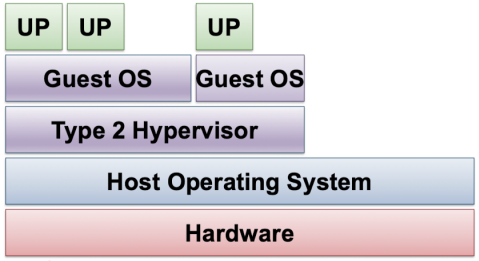
> Runs directly on hardware



- Aka bare-metal hypervisor
- Provides individual VMs to guest OSes via VM interfaces

Type 2 Hypervisor

> Runs in host OS



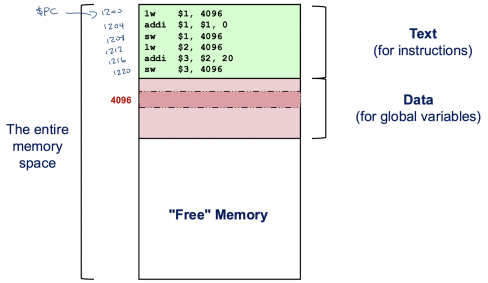
- Guest OS runs inside VM

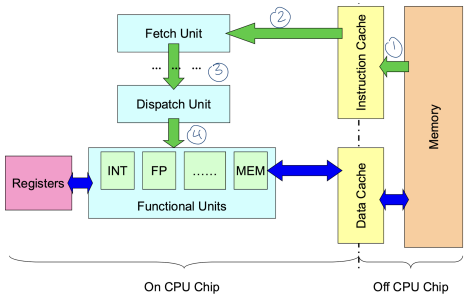
Process Abstraction

- To execute a program, need following information:
  - Memory context: Text, Data, Stack, Heap
  - Hardware context: General purpose registers, Program Counter (PC), Stack Pointer, Frame Pointer
  - OS context: Process ID, Process State

Recap: Hardware and Memory Context

- To execute a program, C code -> Assembly code
  - **Text** for instructions and **data** for global variables stored in memory





- Memory: Storage for instruction and data
- Cache: Duplicate part of memory for faster access
- Fetch Unit: Loads instruction from memory
  - Program Counter (PC): Special register indicating address of instruction to run
- Functional Units: Execute instructions
- Registers: Internal storage for fastest access speed
  - **General Purpose Register** (GPR): Accessible by user program
  - Special Register (e.g., Program Counter)

Stack Memory

> Handling function calls

Motivation

- Before: 1 huge chunk of C code

```
int i = 0;
i = i + 20;
```

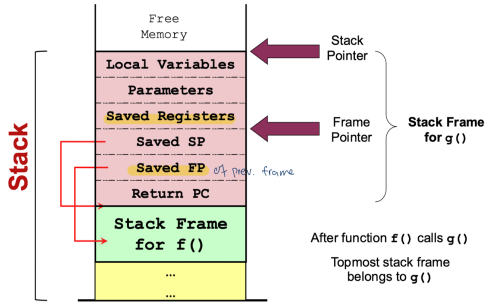
- Now: What if we use functions?

```
int g(int i, int j) {
    int a = i + j;
    return a;
}
```

- Problems:
  - Control flow: How to jump in when calling function and jump out after function call is done?
  - Data storage:
    - How to allocate memory space for **local variables** `i`, `j`, and `a`?
    - How to store return result?
- Can we just use data memory space? No, cannot differentiate between multiple calls of the same function

Solution

- Stack Memory Region: New memory region to store information for function invocations
  - `main` method included inside
- Stack Frame: Information of 1 function invocation
  - Frame added on top when function is called; Frame removed from top when function call ends
  - Stores:
    - Local variables
    - Parameters
    - Saved Stack Pointer
    - Return Program Counter
    - Saved Frame Pointer (See later)
    - Saved registers (See later)



- Stack Pointer: Top of the stack region (1st unused location) that **dynamically changes** as we add more local variables and parameters
  - Special register
  - Requires stack frame to store previous frame's SP
- Function Call Convention: Different ways to setup stack frame
  - E.g., is the information stored in stack frame or register? Prepared by caller or callee?
  - No correct answer, as long as consistent

Frame Pointer

> To facilitate access of stack frame items

- Problem: Stack Pointer is hard to use, since it's **dynamically changing from adding local variables as we run through the function** → Need to dynamically change offset for all the local variables
- Solution: Frame Pointer
  - Frame Pointer: Points to fixed location in stack frame
  - Other items have fixed displacement from FP
  - Requires stack frame to store previous frame's FP
  - Platform dependent
  - FP only for compiler's convenience

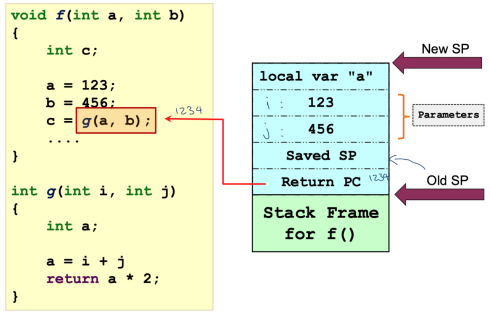
Saved Registers

- Problem: Number of General Purpose Registers limited
- Solution idea: Register spilling
  - When GPRs are all used up, use memory to temporarily hold GPR values
  - GPR can then be reused for other purposes
  - GPR values can be restored later

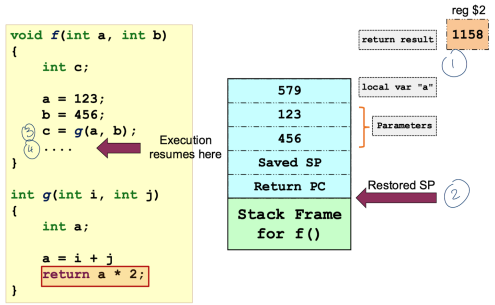
- Saved registers: Saved values from registers that the function intends to use before function starts
  - On function return, restore those registers
  - Stored in stack frame
  - Without saving registers that we intend to use in this function, we may override registers with new values, which can lead to bugs

Sample Frame Setup and Teardown

- On function call:
  1. Caller: Pass arguments with registers and/or stack
  2. Caller: Save Return PC on stack
  3. Transfer control from caller to callee
  4. Callee: Save registers used by callee
  5. Callee: Save old Frame Pointer and Stack Pointer
  6. Callee: Allocate space for local variables on stack
  7. Callee: Adjust Stack Pointer and Frame Pointer **in register**



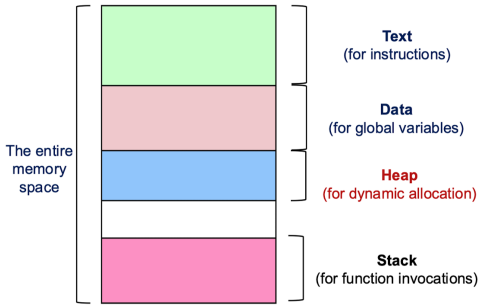
- On function return:
  1. Callee: Place return result in register (if applicable)
  2. Callee: Restore saved registers, FP, SP
  3. Transfer control from callee to caller using saved PC
  4. Caller: Use return result (if applicable)
  5. Caller: Continues execution in caller



- Is stack frame deleted explicitly? No, stack pointer suffices
  - Also why local variable needs to be initialized with new value. Without initialization, repeated calls to same function may mistakenly set variable to value from previous call

Heap Memory

> Dynamically allocated memory (i.e., needed during execution time)

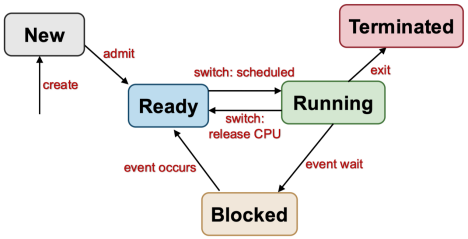


- Examples:
  - In Java, `new`
  - In C, `malloc()`
- Why not use data memory? Allocated only at runtime, so size not known during compilation
  - E.g. size of array not known until runtime
- Why not use stack memory? No definite deallocation time
- Problems:
  - Trickier to manage due to variable size
  - "Holes" in the memory

Process Management

- Motivation: Need to allow many programs to share the hardware for efficient utilization
- Solution: OS needs to allow switching from program A to B, which needs to:
  - Store information about program A

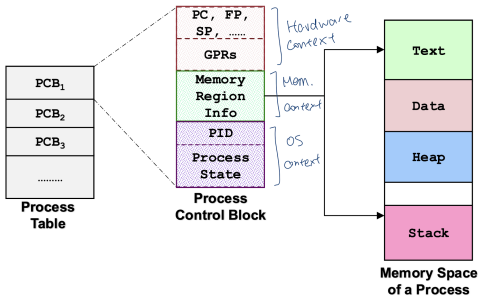
    - Replace program A's information with program B
- Process: Dynamic abstraction to describe a running program
    - Aka task or job
    - Includes memory context, hardware context, and OS context
    - Process ID (PID): Number that distinguishes processes from each other
        - OS dependent issues: Are PIDs reused? Maximum number of processes? Reserved PIDs?
    - Process State: Indication of execution status of process
        - Process Model: Set of **states** and **transitions** that describes behaviors of a process



- Given n processes:
    - With 1 CPU core:
        - ≤ 1 process in running state
        - 1 transition at a time
    - With m CPUs:
        - $\leq m$ processes in running state
        - Possibly parallel transitions
- Assumption in CS2106: Only 1 core!

Process Table

> Putting it all together

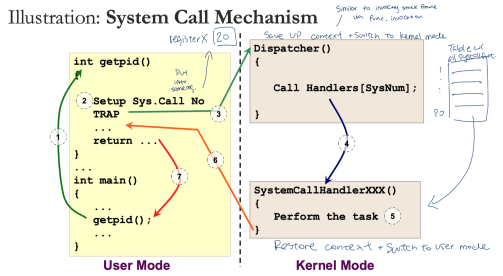


- Process Control Block (PCB): Data structure for storing entire execution context of a process
    - Aka Process Table Entry
    - Kernel maintains PCB for all processes as 1 table (i.e., process table)
- Problems:
    - Scalability: How many concurrent process can you have?
    - Efficiency: How to ensure minimum space wastage?

System Calls

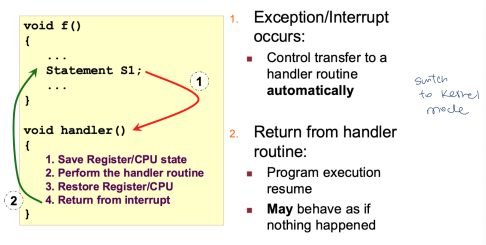
> API for OS

- Provides way of calling services in kernel
    - Different from normal function call: Need to change from user mode to kernel mode
- APIs are OS-specific
    - E.g., Unix system calls in C/C++
        - Function wrapper: Library functions that have same name and parameters as system calls
            - E.g., `getpid()`
        - Function adapter: Library functions that present a more user-friendly version of system call
            - E.g., `printf()` library call uses `write()` from system call



1. UP calls library call
2. Library call places **system call number** in designated location (e.g., register)
3. Library call switches from user mode to kernel mode using special instruction (Aka TRAP) and saves CPU state
4. In kernel mode, **dispatcher** determines system call handler using system call number as index
5. System call handler executed
6. System call handler restores CPU state, returns to library call, and switches from kernel mode back to user mode
7. Library call returns to user program

Exception and Interrupt



- Exception: Can happen when executing machine level instructions
  - E.g., Arithmetic Errors
  - Synchronous: Due to program execution
  - Effect: Have to execute **exception handler** in kernel automatically
- Interrupt: Can happen due to external events
  - Usually hardware related (e.g., keyboard)
  - Asynchronous: Independent of program execution
  - Effect: Have to execute **interrupt handler**

Process Abstraction in Unix

```
ps // Process status
ps -e // List all processes
man fork // Help manual
```

Creating a New Process

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

int fork(); // syntax
```

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

int main() {
    printf("I am ONE\n");
    fork();
    printf("I am seeing DOUBLE\n"); // Printed twice
    return 0;
}
```

- Behavior: Parent process -> Parent and child processes
  - Child process is **duplicate copy** of current executable image (i.e., same contexts)
  - Child only differs in: PID, Parent PID (PPID), `fork()` return value
  - Order of child and parent is non-deterministic
- Returns:
  - For parent process: PID of the child
  - For child process: 0
  - Use return value to distinguish parent and child
- The Master Process: The ancestor of all processes (`init` process)
  - Created in kernel at boot up time
  - Traditionally, PID = 1

Passing Arguments to a Program

```
int main(int argc, char* arv[]) {}
```

- `argc`: Number of command line arguments, including program name itself
  - E.g., `a.exe 1 2 3 hello` -> 5 arguments

Executing Another Program

- Motivation: `fork()` itself is not useful, since still need to provide full code for child process

```
#include <unistd.h>

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

- `path`: Location of executable
- `arg0, ..., argN`: Command line arguments
- `NULL`: To indicate end of argument list

```
#include <unistd.h>

// Same as running: ls -al
int main() {
    printf(...);
    execl("/bin/ls", "ls", "-al", NULL);
    printf(...); // Will not run! Since replaced
}
```

- Behavior: Replaces current executing process image (i.e., code) with new one
  - Process-related information still same
- Combining `fork()` and `execl()`: Can spawn child process and let it perform a task, while parent process is still around to accept another request

Process Termination

```c
#include <stdlib.h>

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

- `status`: Returned to parent process
  - 0: Successful execution -> Normal termination
  - Else: Indicate problematic execution
- Behavior:
  - Most system resources used by process released by `exit()`
  - Some basic process resources not releasable -> Zombie State
    - Motivation: What if parent asks for child PID and status code in `wait()` call?
- Zombie Process: Child process terminates before parent, but parent has not called wait
  - Cannot be killed
- Orphan Process: Parent process terminates before child process
  - `init` process becomes pseudo parent, not the grandparent process (if any)
  - Child termination sends signal to `init`, which uses `wait()` to clean up
- Implicit `exit()`: Return from `main()` implicitly calls `exit()`
  - If `main()` returns some number, then that number is the status code

Parent-Child Synchronization

> Parent process can wait for child process to terminate

```c
#include <sys/types.h>
#include <sys/wait.h>

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

- `status`: Pointer that stores and points to the exit status of terminated child process
  - Pass in `NULL` if don't want this
  - `$?`: Application of this in shell -> Holds exit status of last executed command
- Returns: PID of terminated child process

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    if (fork() == 0)
        printf("Hello from child\n");
    else {
        printf("Hello from parent\n");
        wait(NULL);
        printf("Child has terminated\n");
    }
    printf("Bye\n");
    return 0;
}

// Case 1: If child runs first
// Hello from child
// Bye
// Hello from parent
// Child has terminated
// Bye

// Case 2: If parent runs first
// Hello from parent
// Hello from child
// Bye
// Child has terminated
// Bye
```

- Behavior:
  - Blocks parent process until at least 1 child terminates
  - Cleans up child system resources not removed on `exit()` (including zombie processes)
- Variations:
  - `waitpid()`: Wait for specific child process
  - `waitid()`: Wait for any child process to change status
- If `wait()` is run in child process, basically ignored, since no child processes to wait for

Implementing `fork()`

- Simple way: Make almost exact copy of parent PCB
  - Problem: Memory copy very expensive
  - Observation: If only read, can share with parent; if need write, then a copy is needed
- Copy on Write: Only duplicate memory location when it is written to; otherwise, parent and child share the same memory location
  - More optimized
- `clone()` provides more control on what to copy than `fork()`, which copies everything

Process Scheduling

> Given many ready processes, which should be chosen to run?

- Concurrent processes: Multiple processes progress in execution at the same time
  - Virtual parallelism: 1 core
    - Timeslicing: Interleave instructions from both processes
      - OS incurs overhead in context switching when switching processes
  - Physical parallelism: Multi-core
- Scheduler: Part of the OS that makes scheduling decision

    - Scheduling algorithm: Algorithm used by scheduler
- Process behavior: Combination of CPU-activity and IO-activity
- Processing environment: Influences choice of scheduling algorithm
    - Batch processing: No user interaction required
    - Interactive: Active user interacting with system; should be responsive
    - Real-time processing: Have deadline to meet; periodic
- Common evaluation criteria:
    - Fairness: Each process gets fair share of CPU time
        - No starvation (Process ready, but cannot ever run)
    - Utilization: All parts of computing system should be used
- When to perform scheduling?
    - Non-preemptive (Cooperative): Process stays running until it blocks or gives up (**yields**) CPU voluntarily
    - Preemptive: Process given **fixed time quota** to run

Scheduling for Batch Processing

- Criteria for batch processing:
    - Turnaround time: Total time taken for process to finish
        - Waiting time: Time spent waiting for CPU
    - Throughput: Number of tasks finished per unit time
    - CPU utilization: Percentage of time when CPU is working on a task

First-Come First Served (FCFS)

- Idea:
    - Ready tasks stored in queue based on arrival time
    - Pick first task in queue to run until task is done or blocked
    - Blocked task removed from queue
- Pros: No starvation
- Cons:
    - Average waiting time can be reduced (SJF)
    - Convoy Effect: Cannot efficiently distribute CPU and IO tasks -> CPU/IO usually idling
        - E.g. 1 CPU task, followed by many IO tasks

Shortest Job First (SJF)

- Idea: Pick task with smallest total CPU time
- Pros: Given fixed set of tasks, minimizes average waiting time
- Cons:
    - Need to know task's total CPU time in advance
    - Starvation possible: Since short jobs always prioritized first and long jobs can starve
- Predicting CPU time using exponential average: $\text{Predicted}_{n+1} = \alpha \text{Actual}_n + (1-\alpha)\text{Predicted}_n$
    - Task usually has many phases of CPU-activity (Interleaved with IO-activity) $\rightarrow$ Use previous predicted and actual CPU-activity time to predict future

Shortest Remaining Time (SRT)

> Preemptive version of SJF

- Idea: Pick task with shortest remaining time (Basically same as SJF)
    - **Preemptive**: New jobs with shorter remaining time can cut off running job
- Pros and cons: Same as SJF

Scheduling for Interactive Systems

- Criteria for interactive systems:
    - Response time: Time between arrival and first response by system (Since interactive)
    - Predictability: Variation in response time; Lesser variation $\rightarrow$ More predictable
- Idea: To ensure good response time, **scheduler needs to run periodically to preempt**
    - Timer interrupt: Interrupt that goes off periodically based on hardware clock
    - Interval of Timer Interrupt (ITI): Interval for when OS scheduler is invoked
    - Time Quantum: Execution duration given to process
        - Must be multiples of ITI

Round Robin (RR)

> FCFS with time quantum

- Idea:
    - Ready tasks stored in queue based on arrival time
    - Pick first task in queue to run until task is done or blocked or **time quantum elapsed**
    - Blocked task removed from queue
- Choice of time quantum:
    - Big quantum: Better CPU utilization, but longer waiting time
    - Small quantum: More overhead and worse CPU utilization from context switching, but shorter waiting time
- Response time guaranteed: Since given n tasks and quantum q, response time bounded by $(n-1)q$
- Timer interrupt needed

Priority-Based Scheduling

- Idea:
    - Assign priority value to all tasks
    - Pick task with highest priority value
- Variants:
    - Preemptive: Higher priority processes can preempt running process with lower priority
    - Non-preemptive: Higher priority processes need to wait for next round of scheduling

- Cons:
  - Low priority processes can starve
    - Solution: Can decrease priority of running process after every time quantum
    - Solution: Can remove current running process in next round of scheduling after quantum
  - Hard to control CPU time for a process
- Priority Inversion: High priority task **indirectly** held up by medium priority task
  - Must be 3 priority levels
    - If only 2 levels and high priority is blocked by low priority due to locked resource, then it is **not** priority inversion, instead the high priority task is just not ready
    - 3rd medium priority task is needed to indirectly block high priority task and further delay more than necessary
  - E.g. Priority: $A = 1, B = 3, C = 5$
    - Task C starts and locks resource
    - Task B arrives and preempts C due to priority
    - Task A arrives and needs same resource as C
    - Task A blocked by C despite higher priority

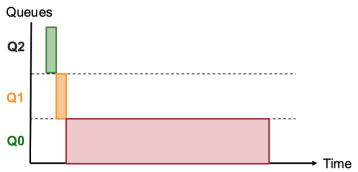
Multi-Level Feedback Queue (MLFQ)

> Lower priority for long-running jobs

- Motivation: How to schedule without perfect knowledge (e.g., process behavior, running time)?
- Adaptive: Learns the process behavior during execution
- Rules:
  - Priority of A $>$ Priority of B $\rightarrow$ A runs
  - Priority of A $==$ Priority of B $\rightarrow$ A and B runs in Round Robin
  - New job given highest priority
  - Job fully utilizes time quantum $\rightarrow$ Priority reduced
  - Job gives up or blocked before time quantum $\rightarrow$ Priority stays the same

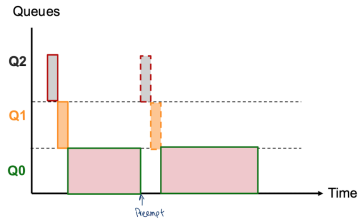
MLFQ: Example 1

- 3 Queues: Q2 (highest priority), Q1, Q0
- A single long running job
  - Try to apply the rules and check your understanding



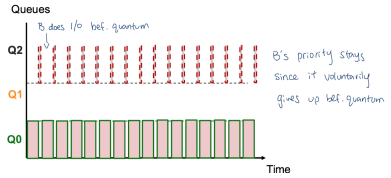
MLFQ: Example 2

- Example 1 + a short job in the middle
  - A short job appears sometime in the middle



MLFQ: Example 3

- Two jobs:
  - A = CPU bound (already in the system for quite some time)
  - B = I/O bound



- Pros:
  - Minimizes response time for IO bound processes
  - Minimizes turnaround time for CPU bound processes
- Cons:
  - Starvation for long-running jobs: Too many interactive jobs
  - Program can change behavior (e.g., CPU bound for a long time $\rightarrow$ Interactive): Low response time
    - Solution: Can periodically bump all jobs to highest priority
  - Abuse I/O: Do an I/O right before quantum ends, so priority remains same forever and hog the CPU
    - Solution: Scheduler can track total time a task has used at a priority level and reduce priority once task hits the limit

Lottery Scheduling

- Idea:
  - Give out "lottery tickets" to processes for various resources
  - When scheduling decision needed, a ticket is chosen randomly and winner is granted the resource
- In the long run, a process holding $X\%$ of tickets $\rightarrow$ Wins $X\%$ of scheduling $\rightarrow$ Uses resource $X\%$ of the time

- Pros:
  - Responsive: New process can join next lottery
  - Can prioritize: Important process can be given more tickets
  - Process can distribute tickets to child processes
  - Each resource has own set of tickets

Inter-Process Communication

- Motivation: Hard for cooperating processes to share information, since memory space is independent
  - So far, only have `wait()`: But that's only at the end of the process and it can only pass a status code

Shared Memory

- Idea:
  1. Process P_1 creates **shared memory region** M
  2. Process P_1 and P_2 attach memory region M to its own memory space
  3. P_1 and P_2 communicate via M
- Master program: Creates and tears down shared memory region

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

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

    // Create shared memory region
    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);
    }

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

    // First element in shared memory region used as control value
    // 0: Not ready, 1: Values ready
    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]);
    }

    // Detach and destroy shared memory region
    shmdt(shm);
    shmctl(shmid, IPC_RMID, 0);

    return 0;
}
```

- Slave program: Attaches and writes values to shared memory region

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

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

    // Receive shared memory id as input
    printf("Shared memory id for attachment: ");
    scanf("%d", &shmid);

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

    // Write values into shared memory
    for (i = 0; i < 3; i++) {
        scanf("%d", &input);
        shm[i+1] = input;
    }

    // Let master program know we're done
    shm[0] = 1;

    // Detach
    shmdt(shm);

    return 0;
}
```

- Pros:
  - Efficient: Only creating and attaching shared memory region involves OS
  - Ease of use: Shared memory space behaves the same as normal memory space

- Cons:
  - Lack of synchronization when accessing shared resources
    - E.g. How to ensure master program sets control value to 0 before slave program?
  - Hard to implement

Message Passing

- Idea:
  1. Process P_1 prepares message M and sends it to process P_2
  2. Process P_2 receives message M
- Send (`send()`) and receive (`recv()`) operations go through OS
  - M in P_1's PCB $\rightarrow$ M in kernel memory $\rightarrow$ M in P_2's PCB
- Naming Scheme:
  - Direct communication: Sender and receiver explicitly name other party
    - Need to know identity of other party
    - Unix: Unix domain socket
  - Indirect communication: Messages sent to and received from shared mailbox or port
    - 1 mailbox can be shared among many processes
    - Unix: Message queue
- Synchronization:
  - Blocking primitive: Receiver blocked until message arrived (Synchronous)
  - Non-blocking primitive: Receiver either receives message if available or some indication that message is not ready yet (Asynchronous)
- Pros:
  - Portable: Can work on different processing environments
  - Easier synchronization: Through blocking primitive
- Cons:
  - Inefficient: Need OS intervention and extra copying in kernel memory

Unix Pipe

> Link input and output channels of processes

- Process has 3 default communication channels:
  - Standard in: `stdin`; In C, `scanf()`
  - Standard out: `stdout`; In C, `printf()`
  - Standard error: `stderr`; In C, `perror()`
- Unix shell: `|`
- Producer-Consumer relationship:
  - P produces/writes n bytes
  - Q consumes/reads m bytes
- Queue behavior: Must access data in order
- Can be used to solve synchronization, though not common

Creating a Pipe

```c
#include <unistd.h>

int pipe(int pipefd[2]); // syntax
```

- Parameter: Array of 2 file descriptors
  - `pipefd[0]`: Read end of pipe
  - `pipefd[1]`: Write end of pipe
- Returns: 0 for success and !0 for errors

```c
#define READ_END 0
#define WRITE_END 1

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

    pipe(pipeFd);
    if ((pid = fork()) > 0) {
        // Parent process
        close(pipeFd[READ_END]);
        write(pipeFd[WRITE_END], str, strlen(str) + 1);
        close(pipeFd[WRITE_END]);
    } else {
        // Child process
        close(pipeFd[WRITE_END]);
        len = read(pipeFd[READ_END], buf, sizeof(buf));

        printf("Process %d read: %s\n", pid, buf);
        close(pipeFd[READ_END]);
    }
}
```

- `read()`: Blocking call until parent writes
- `close()`: Closes reading/writing end for process
  - Signal end of reading/writing

Unix Signal

> Asynchronous notification about an event that is sent to a process/thread

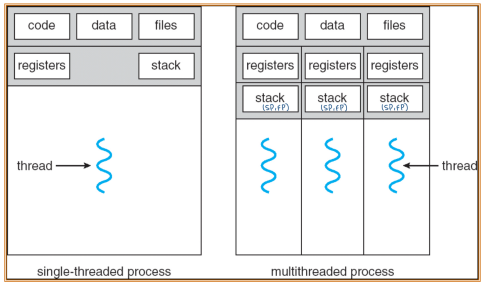
- Recipient must handle signal by using default handler or providing custom handler
- Common signals in Unix: Terminate (`SIGTERM`), kill (`SIGKILL`), interrupt (`SIGINT`)
- Can be used to solve synchronization, though not common

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

```c
#include <unistd.h>

void customHandler(int signo) {
    if (signo == SIGSEGV) {
        printf("Memory access blew up!\n");
        exit(1);
    }
}

int main() {
    int *ip = NULL;

    // Register our own handler to replace default handler
    if (signal(SIGSEGV, customHandler) == SIG_ERR) {
        printf("Failed to register handler\n");
    }

    // Will cause segmentation fault
    *ip = 123;

    return 0;
}
```

Threads

- Motivation:
  - Process creation is expensive
    - Duplicate memory space and process context
    - Context switching
  - Hard for independent processes to do IPC
- Idea:
  - Traditional process has **single thread of control**: Only 1 instruction in program is running → 1 PC
    - Multi-process model: Multiple "threads of control" using `fork()`
  - Add more threads of control to same process → Many PCs



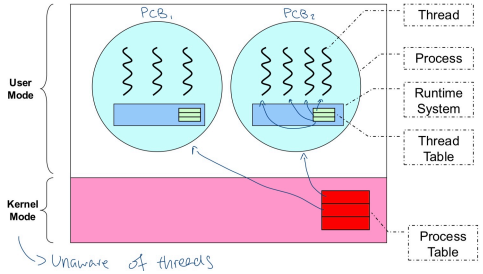
> Thread is a "lightweight process"

- Process vs. Thread:
  - Multithreaded process: Single process with multiple threads
  - Threads in the same process share:
    - Memory context: Text, data, heap
    - OS context: PID, files
  - Unique information specific to each thread:
    - Thread ID
    - General purpose registers: For calculations
    - Special registers: PC
    - "Stack": Stack still shared within process, just FP and SP copied
- Pros:
  - Less resources needed
  - No need for IPC
  - More responsive, since no need context switching
  - Scalable, since multithreaded programs can take advantage of multiple CPUs
- Cons:
  - Parallel execution of multiple threads → Parallel system call possible → Need to guarantee correct behavior
  - Impact on whole process
    - `fork()` duplicates process, then threads inside?
    - Single thread calls `exit()`, then the other threads inside the process?

Thread Models

User Thread

> Implemented as a user library, so kernel is not aware of threads

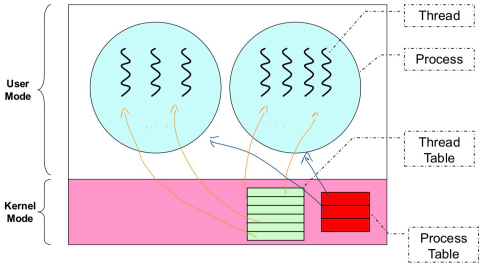


- Pros:
  - Can have multithreaded programs on any OS
  - Thread operations just library calls
  - More configurable and flexible
- Cons:
  - Scheduling done at process level

- 1 thread blocked → Entire process blocked
- Cannot exploit multiple CPUs, since scheduling done by process

Kernel Thread

> Implemented in the OS



- Pros:
  - Kernel can schedule by threads, instead of by process
    - Threads in same process can run simultaneously on multiple CPUs
- Cons:
  - Thread operations now a system call → Slower and more resource intensive
  - Less flexible, since implemented in kernel and used by all multithreaded programs
    - If too many features, then expensive and overkill for simple program
    - If too little features, then not flexible for some programs

Hybrid Thread

- Have both kernel and user threads
- Multiple user threads can bind to 1 kernel thread
- OS schedules on kernel threads only
- Pros: More flexible, since can limit concurrency on any process

POSIX Threads

- Defines API, not implementation
  - Can be implemented as user or kernel thread

Creating and Terminating Threads

```
#include <pthread.h>

gcc XXX.c -lpthread

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

- Returns: 0 if success; !0 if errors
- Parameters:
  - `tidCreated`: Thread ID for created thread
  - `threadAttributes`: Control the behavior of new thread
  - `startRoutine`: Function pointer to the function to be executed by thread
  - `argForStartRoutine`: Arguments for the `startRoutine` function
- Different from `fork()`, since forked child starts at same location as parent

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

- `exitValue`: Value to be returned to whoever syncs with this thread
- If `pthread_exit()` is not used, thread will end automatically at the end of `startRoutine`
  - If `return XYZ;` is used, then `XYZ` is captured as `exitValue`

```
int globalVar;

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

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

- Final sum may not be 5000!
  - Possible for `main()` to return first, before all the calculations

Simple Synchronization

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

- Behavior: Waits for termination of another `pthread`
- Returns: 0 if success; !0 if errors

- Parameters:
  - `threadID`: Thread ID of `pthread` to wait for
  - `status`: Exit value returned by target `pthread`

```c
int globalVar;

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

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

    for (i = 0; i < 5; i++)
        pthread_join(tid[i], NULL);
    return 0;
}
```

- Slightly better, but final sum still may not be 5000!
  - `globalVar++` is not atomic, since it involves read → calculate → write

Synchronization

Problems with Concurrent Execution

- Race condition: Execution outcome depends on order in which shared resource is accessed/modified
  - Concurrent execution may be non-deterministic (Like the above example)
- Incorrect execution due to unsynchronized access to shared modifiable resources
- Solution:
  - Critical section: Code segment with race condition
  - At any time, only 1 process can execute in critical section and other processes are prevented from entering the same critical section

Critical Section

- Properties of correct implementation:
  - Mutual exclusion: If there is a process executing in critical section, then all other processes are prevented from entering
  - Progress: If no process is in critical section, then a waiting process should be granted access
  - Bounded wait: After process requests to enter critical section, there exists a limit on number of times other processes can enter critical section beforehand
    - I should eventually enter!
  - Independence: Process not in critical section should not block other processes
- Symptoms of incorrect synchronization:
  - Deadlock: All processes blocked
  - Livelock: Processes keep changing state to avoid deadlock and make no progress
    - E.g. 2 people in sidewalk and both move in same direction
  - Starvation: Some process forever blocked

Assembly Level Implementation

```
TestAndSet Register, MemoryLocation
```

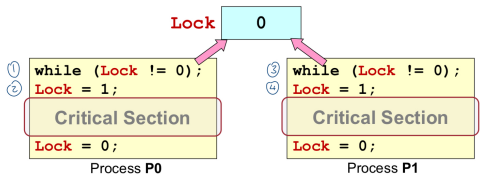
- Behavior: **Atomic**
  1. Load content at `MemoryLocation` into `Register`
  2. Stores 1 into `MemoryLocation`

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

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

- Assume `TestAndSet` has high-level language version:
  - Parameter: Memory address M
  - Saves content at M to register and sets content of M to 1
  - Returns register value
- Intuition: 1st process can escape `while` loop while other processes are stuck in loop until lock is set to 0 again
- Cons: Busy waiting (Keeps checking condition) → Wasteful processing power

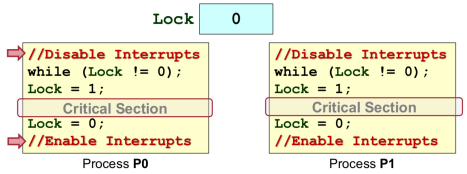
High-Level Language Implementation

- Attempt 1: Does not work

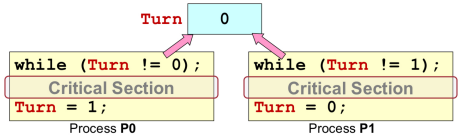


- If in sequence 1 → 3 → 2 → 4, violates mutual exclusion
  - Because load and store not atomic, unlike `TestAndSet`

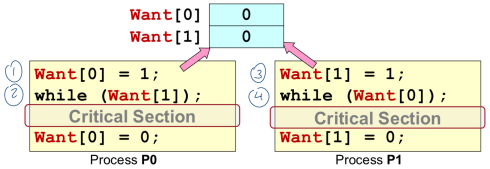
- Attempt 1a: Works, but...



- Intuition: Disable interleaving
- Cons:
  - Buggy critical section may stall whole system, since cannot preempt
  - Busy waiting
  - Need permission to disable and enable interrupts
- Attempt 2: Does not work

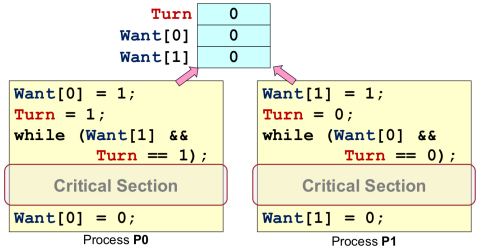


- Starvation: If P0 never enters critical section, then P1 starves
  - Violate independence
- Attempt 3: Does not work



- If in sequence $1 \to 3 \to 2 \to 4$, deadlock
  - Progress violated

Peterson's Algorithm



- Assumption: Write to `Turn` is atomic
- Intuition: Attempt 3 with deadlock solved
  - Solves deadlock with `Turn`, since `Turn` is either 0 or 1 when we reach `while`
  - Allows other process to enter once done, since `want[1] = 0` now
- Cons:
  - Busy waiting
  - Low level
  - Not general: Only for synchronization of 2 processes

Semaphore

> Data structure with 2 atomic operations: `wait()` and `signal()`

- Generalized synchronization mechanism
  - Only behavior specified; can have different implementations
- Semaphore S contains:
  - Capacity: Integer value that is initialized to any non-negative values
  - List to track waiting processes
  - **Atomic** operations:
    - `wait(S)`: Aka `P()` or `Down()`
      1. If $S \leq 0$, then blocks process/go to sleep
      2. Decrement S
    - `signal(S)`: Aka `V()` or `Up()`
      1. Increment S
      2. Wakes up a sleeping process to try again
- Given $S_{\text{initial}} \geq 0$, $S_{\text{current}} = S_{\text{initial}} + \texttt{\#signal(S)} - \texttt{\#wait(S)}$
  - `#signal(S)`: Number of `signal()` operations executed
  - `#wait(S)`: Number of `wait()` operations **completed**
- General semaphore: $S \geq 0$
- Binary semaphore: $S = 0$ or $S = 1$
  - Can build general semaphores
- Mutex: Usage of synchronization mechanism (e.g., semaphore) to ensure mutual exclusion

```
Wait(S);
// Critical section
Signal(S);
```

- Semaphores ensure mutual exclusion
  - N_{CS} = Number of processes in CS = Number of processes that completed `wait()` but not `signal()` = `#wait(S)` $-$ `#signal(S)`
  - $S_{\text{current}} + N_{CS} = S_{\text{initial}} = 1$
  - $N_{CS} \leq 1$ since $S_{\text{current}} \geq 0$
- A semaphore prevents deadlock
  - Deadlock means all processes stuck at `wait()` $\to S_{\text{current}} = 0$ and $N_{CS} = 0$

- But $S_{\text{current}} + N_{CS} = 1$
- But deadlock still possible with multiple semaphores
  - E.g. need to acquire resources from 2 semaphores P and $Q \rightarrow$ `wait()` in different order
  - Solution: Ensure order is same
- A semaphore prevents starvation
  - Suppose $P1$ blocked at `wait()`
  - $P2$ is in CS, and exits CS with `signal()`
    - If no other processes sleeping, $P1$ wakes up
    - If there are other processes, $P1$ eventually wakes up

Classic Synchronization Problems

Producer Consumer

- Processes share a bounded buffer of size K
- **Producer** processes insert items in buffer
  - Only when buffer is not full
- **Consumer** processes remove items from buffer
  - Only when buffer is not empty

```
// Producer process
while (True) {
    // Produce item
    if (count < K) {
        buffer[in] = item;
        in = (in + 1) % K;
        count++;
    }
}

// Consumer process
while (True) {
    if (count > 0) {
        item = buffer[out];
        out = (out + 1) % K;
        count--;
    }
    // Consume item
}
```

- Problems:
  - No synchronization: When running concurrently, there exists critical sections
  - Produced item wasted if buffer full
  - Consume nothing if buffer empty
- Solution 1: 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

- Intuition:
  - Wraps critical section in mutex
  - `canProduce`: Flag for blocking through busy waiting to not waste produced item
    - Initialized to 1
  - `canConsume`: Flag for blocking to not consume nothing
    - Initialized to 0
- Solution 2: No busy waiting with more semaphores!

```
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

- Intuition:
  - `notFull` semaphore: Forces producer to sleep when buffer full
    - Initialized to K
  - `notEmpty` semaphore: Forces consumer to sleep when buffer empty
    - Initialized to 0

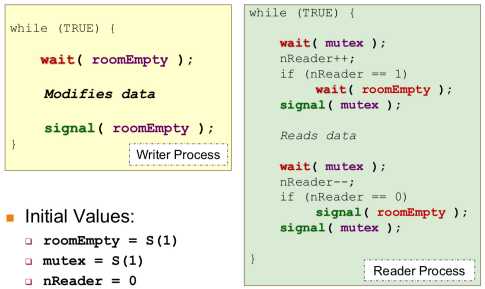
Readers Writers

- Processes share data structure D
- Reader processes retrieve information from D
  - Can share access with other readers
- Writer processes modify information in D
  - Must have exclusive access
- Attempt 1: Does not work

```
// Writer process
while (True) {
    wait(roomEmpty);
    // Modify data
    signal(roomEmpty);
```

```
}

// Reader process
while (True) {
    nReader++;
    if (nReader == 1)
        wait(roomEmpty);

    // Read data

    nReader--;
    if (nReader == 0)
        signal(roomEmpty);
}
```

- Intuition:
  - `roomEmpty` semaphore: Blocks writer if room is not empty and blocks first reader if room is not empty
    - Initialized to 1
- Problems:
  - Critical sections of `nReader++` and `nReader--`
  - If first reader is waiting on writer, and second reader comes in, it will skip `wait(roomEmpty)`
- Solution: Add a mutex



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

- Intuition: Adding mutex solves both problems, since first reader waiting for writer to finish will block other readers too
- Writer Starvation: In the above solution, as long as at least one reader, writers cannot write and starve
  - Unbounded wait time
  - Solutions:
    - If writer is waiting, new readers are blocked until writer finishes
    - Implement timeout for readers

Dining Philosophers

- 5 philosophers seated around a table and 5 **single** chopsticks placed between each pair of philosophers
- Philosopher is either thinking, hungry, or eating
- To eat, a philosopher must acquire both chopsticks to his/her left and right
- Attempt 1: Does not work

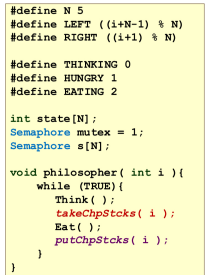
```
#define N 5
#define LEFT i
#define RIGHT ((i + 1) % N)

// For philosopher i
while (True) {
    // Think
    takeChopstick(LEFT);
    takeChopstick(RIGHT);
    // Eat
    putChopstick(LEFT);
    putChopstick(RIGHT);
}
```

- Problem: Deadlock, when all philosophers simultaneously takes up left chopstick, and none can continue
  - To fix this, make philosopher put down left chopstick if right chopstick cannot be acquired → No deadlock, but possible livelock
- Attempt 2: Add a mutex

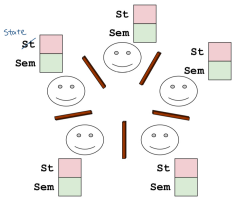
```
#define N 5
#define LEFT i
#define RIGHT ((i + 1) % N)

// For philosopher i
while (True) {
    // Think
    wait(mutex);
    takeChopstick(LEFT);
    takeChopstick(RIGHT);
    // Eat
    putChopstick(LEFT);
    putChopstick(RIGHT);
    signal(mutex);
}
```

- Only 1 philosopher eating at 1 time
- Avoids deadlocks, but does not maximize concurrency

Tanenbaum Solution



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

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

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

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

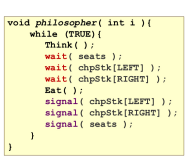



```
void takeChopsticks(i) {
    wait(mutex);
    state[i] = HUNGRY;
    safeToEat(i);
    signal(mutex);
    wait(s[i]);
}

void safeToEat(i) {
    if ((state[i] == HUNGRY) &&
        ((state[LEFT] != EATING) && (state[RIGHT] != EATING))) {
        state[i] = EATING;
        signal(s[i]);
    }
}

void putChopstick(i) {
    wait(mutex);
    state[i] = THINKING;
    safeToEat(LEFT);
    safeToEat(RIGHT);
    signal(mutex);
}
```

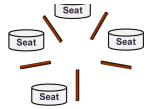
- `state[i]`: State of philosopher i
- `mutex`: For blocking critical section of updating state
- `S[i]` semaphore: Blocks philosopher i if hungry and cannot eat
  - All initialized to 0
- Intuition:
  1. If any philosopher next to me is eating, block myself
     1. When a philosopher finishes eating, they signal to me that I have a chance to eat
  2. Else, signal then block (which cancels out), then eat!
     1. Once finish eating, signal to hungry philosophers next to me to eat

Limited Eater

- Still have 5 philosophers



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



- Intuition: If at most 4 philosophers can sit at the table → There exists empty seat(s) → Attempt 1 works, since no deadlock
  - `seats` semaphore: Blocks if no more seat
    - 5th philosopher can eat after another philosopher gives up seat

Synchronization Implementations

POSIX Semaphore

- Implementation of semaphore under Unix

```
#include<semaphore.h>

gcc something.c -lrt // real time library

// Usage
// Initialize a semaphore
// Perform wait() or signal() on semaphore
```

`pthread` Mutex and Conditional Variables

- For `pthreads`
- Mutex (`pthread_mutex`)
  - Binary semaphore
  - Lock: `pthread_mutex_lock()`
  - Unlock: `pthread_mutex_unlock()`

Memory Management

- Random Access Memory (RAM): Different from hard drive
  - Treat as contiguous array of bytes, where each byte has unique index (Physical address)
- Memory hierarchy: Speed is proportional to cost, and thus inversely proportional to size and proximity to CPU
  - Spatial locality: Given a used address, program likely to access memory addresses that are adjacent
  - Temporal locality: Given a used address, program likely to access it again soon
  - Working set: Small area of memory currently in focus (e.g. loops) → Should be in faster memory
- Binding of Memory Address: Source code (compile by compiler) → Executable (load and run by OS) → Main memory

    - Compiler: Has symbol table to track variables (Variable → Address)
        - Scans through code and adds variables to symbol table
        - Helps compiler to assign memory layout (e.g. offsets)
        - Generates machine code: Replaces symbolic names with memory layout information
    - OS:
        - Maps executable into memory space (e.g. text, data segments)
        - Assigns actual memory space
- Transient data: Exists only for a short while
    - Can grow/shrink during execution
    - E.g. local variables, arguments
- Persistent data:
    - Can grow/shrink during execution
    - E.g. heap (not the pointer pointing to `malloc`)
    - Leakage: Persistent data not released
- Role of OS in managing memory:
    - Allocate space to new process
    - Manage memory space for process
    - Protect memory space of process from each other
    - Provides memory-related system calls to process (e.g. `malloc`)
    - Manage memory space for internal use (e.g. data structure for PCB table)

Memory Abstraction

- Considerations:
    - Protection of address space for a process
    - Correct sharing of physical memory by multiple processes

Without Memory Abstraction

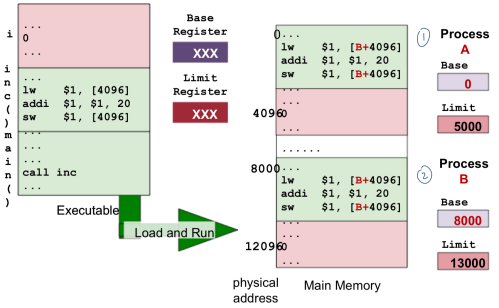
- Idea: Process directly uses physical address
- Pros: Simple, Works well for 1 process at a time
- Cons: With multiple processes, each process may access each other's memory space
    - E.g. Process 1 and 2 both access memory address 4096

Address Relocation

- Idea:
    1. OS scans and finds contiguous space in main memory to store new process
    2. Recalculates all memory references in process by adding offset
- Cons:
    - Slow to recalculate
    - Hard to distinguish memory reference from normal integer
        - E.g. address stored in register → How to determine whether this is an address for recalculation?

Base and Limit Registers

- Base Register: Special register as the base of all memory references in a process
    - During compilation, all memory references compiled as offset from starting address of process memory space
    - During loading, base register initialized to starting address of process memory space
    - During execution, hardware adds memory references and base register to get physical address
- Limit Register: Special register to indicate upper limit of memory space of current process
    - Valid range: $[BR, BR + LR)$
    - All memory access is checked against limit for protection
    - If out of range, segmentation fault



- Base and limit registers for each process stored in process's hardware context
- During execution, they are then loaded into registers
- Cons: Every memory access incurs an addition and comparison of limit

Logical Address

> Abstraction for how process views its memory space

- E.g. Memory reference 4096 means different physical addresses in Process A and B

Contiguous Memory Allocation

- Assumptions:
    - Store Memory concept: Program must be in memory during execution
    - Load-Store Memory execution model: Only interactions with memory are load and store
    - Each process occupies contiguous memory region
    - Physical memory large enough to contain multiple processes
- To support concurrency:
    - Allow multiple processes in physical memory
    - When physical memory is full, free up memory by removing terminated processes or swapping blocked processes to secondary storage
- Memory partition: Contiguous memory region allocated to single process

- Motivation: How does OS allocate memory for process?

Fixed-size Partition

> Fixed number of partitions

- Pros:
  - Easy to manage: Can use simple bitmap to track whether partition used or not
  - Fast to allocate: Every free partition is the same
- Cons: Partition size needs to be large enough to contain **largest process**
  - Internal fragmentation: Allocated memory blocks are larger than needed memory size $\rightarrow$ Wasted space internally

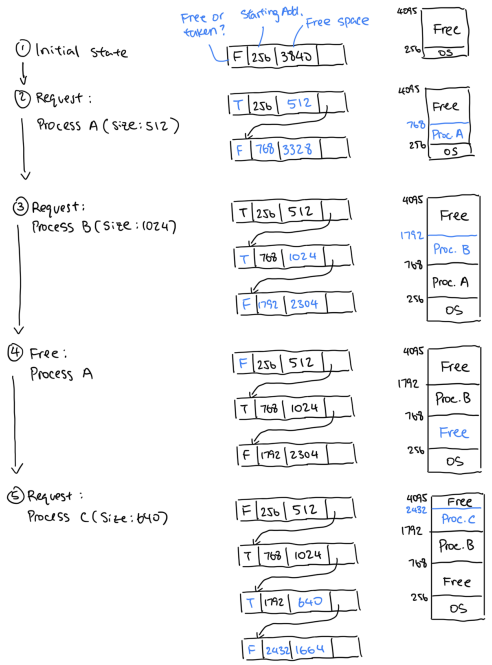
Variable-size Partition

> Partition created based on actual size of process

- Pros: Flexible, No internal fragmentation
- Cons:
  - External fragmentation: Free memory in small, non-contiguous blocks $\rightarrow$ Cannot fit new process, though it can when combined $\rightarrow$ Wasted space externally
  - Need to maintain more information in OS
  - More time and considerations to locate appropriate region

Allocation Algorithms

- Assume: OS maintains list of partitions and holes
- Goal: Locate partition of size N
- Strategies to find hole with size $M > N$:
  - First-fit: Take the first hole that's large enough
  - Best-fit: Find the smallest hole that's large enough
    - Cons: Leaves holes that are small $\rightarrow$ External fragmentation
  - Worst-fit: Find the largest hole
    - Pros: Resulting hole is the largest
- Merging: When occupied partition is freed, merge with adjacent hole if possible
- Compaction: Move occupied partitions around to consolidate holes
  - Expensive!
- Example: OS maintains linked list for partitions and uses first-fit algorithm



- Time complexity:
  - Allocation: $O(n)$, which gets slower as we allocate more
  - Deallocation: $O(1)$
  - Merging: $O(1)$ if we assume only merge neighbor

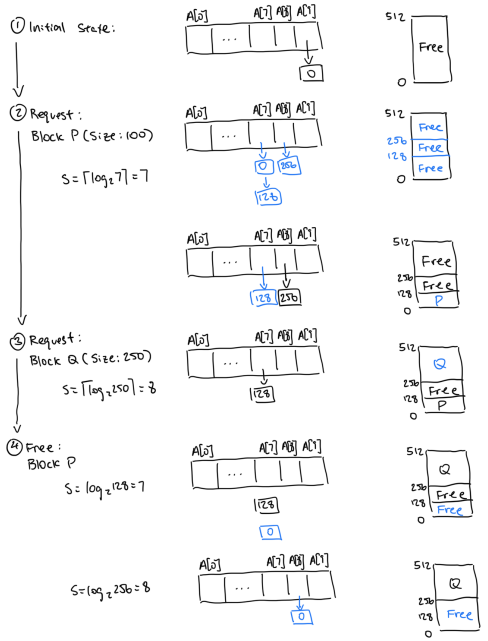
Buddy System

- Idea:
  - Free block split into half repeatedly to meet request
    - The 2 halves form buddy blocks
  - When buddy blocks are both free, they can be merged
- Algorithm:

```
A = Array of size k where 2^k is the largest allocatable block size

allocate(N):
        // Find smallest s such that 2^s >= N
        s = ceil(log2(N))

        if (Free block exists in A[s]):
                Remove block from free block list
                Allocate block
```

```
        else:
                Find smallest r from s + 1 to k, such that A[r] has free block B
                For (r - 1) to s:
                        Split B repeatedly, such that A[s:r - 1] all have a new
free block
                Remove free block from A[S] and allocate block

deallocate(B):
        s = log2(Size of B)
        if (Buddy C of B exists in A[s]):
                Remove B and C from list
                Merge B and C to get larger block B'
                deallocate(B')
        else:
                Insert B to A[s]
```

- `A[i]` is a linked list which tracks **free** blocks of size 2^i
  - Each node contains starting address of free block and next node
- How to find buddy?
  - To find buddy block C of block B with size 2^S: S-th bit of B and C are **complements** and leading (left) bits up to S-th bit of B and C are the **same**
  - E.g. given block address $A = 0 = 000000_2$ with size 32
    - After splitting, $B = 0 = 000000_2$ and $C = 16 = 010000_2$ with size 16
    - B and C are buddy blocks, since 4th bit are complements and leading bits up to 4th bit are the same



- Time complexity:
  - Allocation: $O(\log_2 n)$
  - Deallocation: $O(1)$ without merging
  - Merging: $O(\log_2 n)$

Disjoint Memory Schemes

- Assumptions:
  - Process memory space can be in **disjoint** physical memory locations
  - Physical memory large enough to contain multiple processes

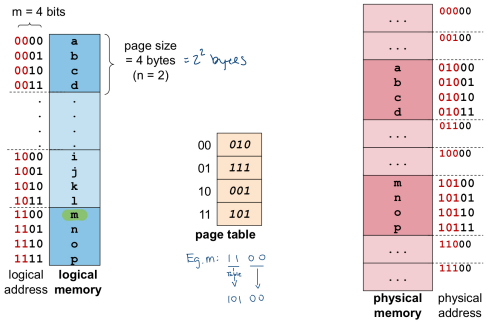
Paging Scheme

- Idea:
  - Frame: **Physical** memory split into regions of fixed size
  - Page: **Logical** memory of a process split into regions of same size
    - Size of frame = Size of page
  - During execution, process pages are loaded into any available frame
    - Logical memory space still contiguous, but occupied physical memory region can be disjointed

Logical Address Translation

- Page Table: Mapping from page number to frame number
- To locate something in physical memory, we need physical frame number (F) and offset from start of physical frame
  - $\text{Physical address} = F * \text{Frame size} + \text{Offset}$
  - But, multiplication is expensive!
- Design decision: Keep frame/page size as power of 2 $\rightarrow$ To use bit shifting
- Given:
  - Page/frame size of 2^n
  - m bits of logical address
  - Logical address LA
    - Most significant $m - n$ bits $\rightarrow$ For finding frame number f from page table
    - Remaining n bits $\rightarrow$ Offset d
- $\text{Physical address} = f * 2^n + d$
  - $f * 2^n$ can use bit shifting
  - Concatenation done by placing wires together; no need to add

Example: 4 Logical Pages, 8 Physical Frames



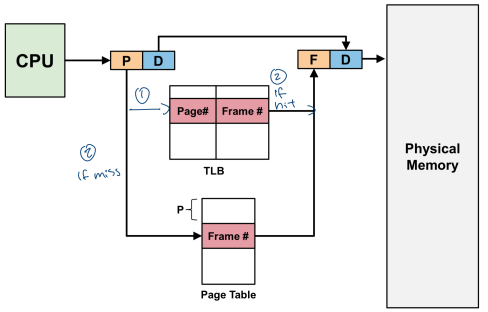
Observations

- No external fragmentation: All free frames can be used
- Internal fragmentation still possible, since logical memory space may not be multiple of page size
  - Wasted internal space in last page of process

Hardware Support

- Each process has its own page table $\rightarrow$ OS stores page table information inside process's PCB under memory context
  - Page Table Register: Base physical address of page table
  - Page table
- How to access page table from a given logical address? Hardware does this
  - When doing context switching, OS loads Page Table Register
  - Hardware adds page number to Page Table Register to access page table in physical memory

- Translation Look-aside Buffer (TLB): Cache for page table
  - Motivation: Each memory reference requires 2 memory accesses (Read page table and access actual memory item) $\rightarrow$ CPU limited by 2 memory accesses
  - Search TLB using page number:
    - If hit, retrieve cached frame number
    - If miss, retrieve from page table and update TLB
  - Impact on memory access time: Given hit rate H, TLB access time T, main memory access time M:
    - With TLB: $H(T + M) + (1 - H)(T + 2M)$
    - Without TLB: $2M$
    - High hit rate: Due to locality $\rightarrow$ 1 entry in TLB can handle all memory access in a page
    - Can have another cache for main memory to reduce M
  - Part of hardware context of a process
    - When context switching, TLB flushed $\rightarrow$ Lots of TLB misses at the start $\rightarrow$ Can place some entries initially (e.g. code pages)



Protection

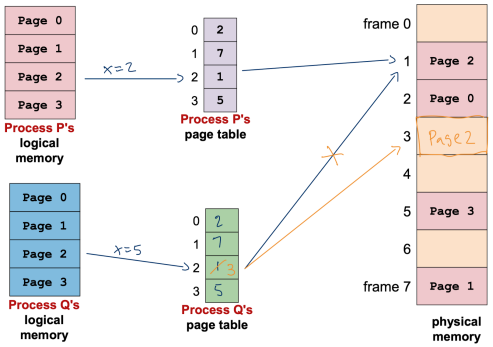
- Access-Right Bits: Attached to each page table entry to indicate whether page is readable, writable, and executable
  - E.g. page with text segment: R1W0X1, page with data segment: R1W1X0
  - Memory access checked against these bits
- Valid Bit: Attached to each page table entry to indicate whether page is valid for process to access
  - Motivation: Logical memory range usually same for all processes $\rightarrow$ Some processes do not use the whole range
  - Memory access checked against this bit $\rightarrow$ OS catches out-of-range access

Page Sharing

> Page table can allow multiple processes to share same physical memory frame

- Motivation: Wasteful to just duplicate everything when forking process
- Page sharing: No duplication of process space; only page table is duplicated
  - Hardware context different (e.g. PTB)

- Copy-On-Write:
  - Motivation: What if 1 process need to write when page sharing?

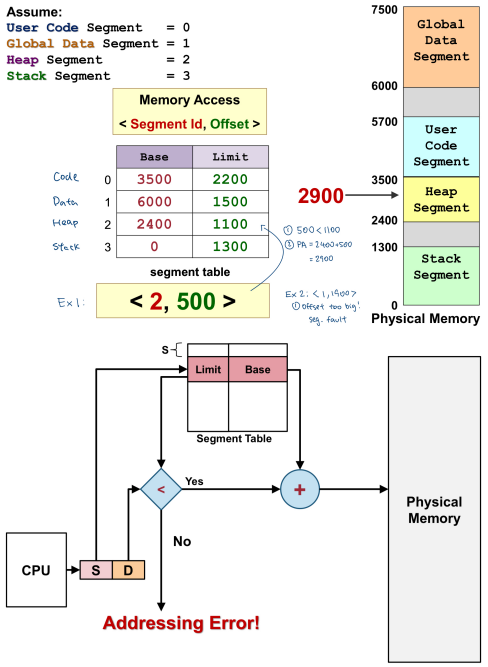


Segmentation Scheme

- Motivation: Multiple memory regions with different purposes in a process $\rightarrow$ Some regions may grow/shrink during execution
  - E.g. heap, stack
- Idea:
  - Separate logical memory regions into memory **segments**
  - Map each segment into **contiguous** physical memory region

Logical Address Translation

- Logical address: $< \text{SegID}, \text{Offset} >$
  - Segment Name/ID: To look up segment table
- Segment table: Mapping from SegID to $< \text{Base}, \text{Limit} >$
  - Base: Base physical address of segment
  - Limit: Indicates range of segment
  - $\text{Physical address} = \text{Base} + \text{Offset}$
  - $\text{Offset} < \text{Limit}$ for valid access

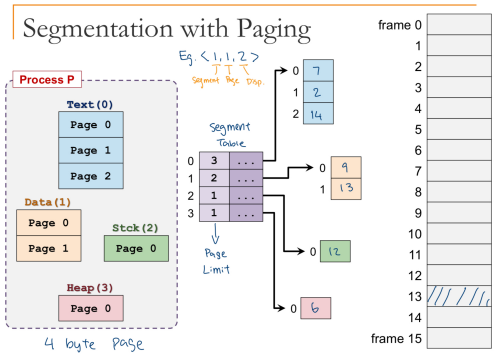


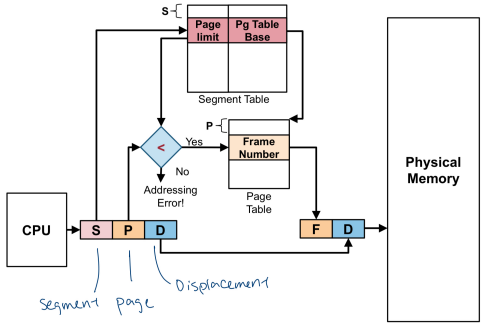
Observations

- Pros:
  - Can grow/shrink independently by adjusting limit
  - Can be protected/shared independently
- Cons: External fragmentation

Segmentation with Paging Scheme

- Motivation:
  - Paging solves external fragmentation
  - Segmentation solves growing/shrinking of segments
- Idea:
  - Each segment now composed of pages, instead of contiguous physical regions
  - Each segment in the process has a page table
  - Segments can grow/shrink by adding/deleting pages





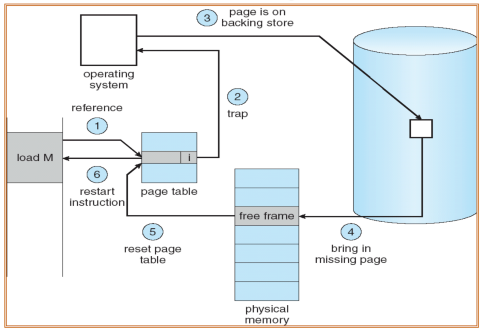
Virtual Memory Management

- Motivation:
  - Previous assumption: Physical memory is large enough to hold process's logical memory space
  - What if logical memory space of process $>$ physical memory?
- Idea:
  - Extend paging scheme to split logical address space into small chunks
  - Store some chunks in physical memory (RAM) and some in secondary storage (Hard drive), since secondary storage has much larger capacity
  - Add bit in page table entry to indicate if page is a memory resident (In physical memory) or non-memory resident (In secondary storage)
    - Page fault: CPU tries to access non-memory resident pages
      - CPU can only access memory resident pages

Accessing Virtual Memory

- By hardware:
  1. Check page table if page X is a memory resident
     1. If yes, then access it
     2. Else, trigger page fault, which traps to OS and informs OS via interrupt handler
- By OS:
  1. Locate page X in secondary storage
  2. Load page X into physical frame
  3. Update page table
  4. Go to step 1 by hardware
- Hardware handles checking page table
- OS handles loading from hard disk, because it's difficult for hardware to do

Virtual Memory Accessing: Illustration



Observations

- Secondary storage access time $>>$ Physical memory access time
  - Thrashing: Lots of time spent loading non-memory resident pages to physical memory due to page faults
    - **Unlikely** due to temporal locality and spatial locality
- More processes can reside in physical memory $\rightarrow$ Better CPU utilization
  - Idle process: Does nothing, except keeps CPU running

Demand Paging

- Application of virtual memory
- Motivation:
  - When a new process starts execution, it's costly to load all pages to physical memory
  - Which pages should be loaded?
- Idea:
  - Process starts with no memory resident page
  - Only load page into physical memory when there is page fault
- Pros:
  - Fast startup time for new process
  - Small memory footprint
- Cons:
  - Process may be slow at the start due to page faults

- Solution: Preload page 1 only, since hard to tell which pages needed before runtime
- Page faults may have cascading effect on other processes (i.e., thrashing taking up CPU time)

Page Table Structure

- Motivation:
  - Huge logical memory space → Lots of pages → Each page has a page table entry → Huge page table, which takes up physical memory
  - Issues:
    - High overhead
    - Fragmented page table: Page table split across frames

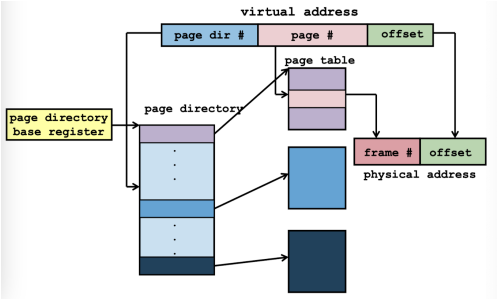
Direct Paging

> Keep all entries in single table

- Given 2^p pages in logical memory space:
  - p bits needed to specify 1 page
  - 2^p page table entries (PTE), each containing:
    - Physical frame number
    - Additional information bits (e.g., valid bit, access right bit, etc.)
- E.g., 32-bit virtual address, Page size: 4 KB, PTE size: 2 bytes
  - Number of pages: $2^{32}/2^{12} = 2^{20}$ pages
  - $p = 32 - 12 = 20$ and rest used for offset
  - Page table size: $2^{20} * 2 = 2$ MB

2-Level Paging

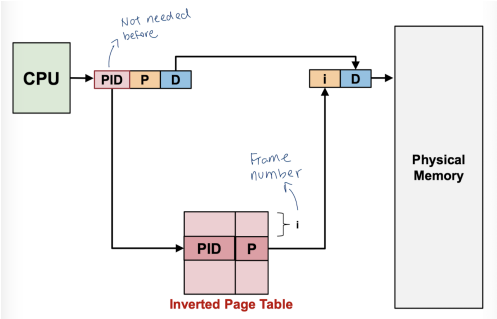
- Motivation: Process does not use everything in virtual memory space → Full page table is wasteful
- Idea: Split page table into **smaller** page tables, each with page table number
  - Single page directory to track smaller page tables
- Given original page table has 2^p entries:
  - If page directory has 2^m smaller page tables, then m bits needed to specify 1 smaller page table
  - Each smaller page table contains 2^{p-m} entries
- Virtual address format: | m | p-m | d |



- Pros: Can have empty entries in the page directory → Less space used than full page table

Inverted Page Table

- Motivation: m processes → m page tables, but only n physical frames → Generally, memory overhead of m page tables $>>$ n physical frames, which is wasteful
- Idea: Keep **inverted** mapping of physical frame to $< \text{pid}, \text{page num.} >$
  - $< \text{pid}, \text{page num.} >$ can uniquely identify memory page
  - Search whole table for matching $< \text{pid}, \text{page num.} >$, then matching entry's index is the physical frame number
- Pros: 1 table for all processes
- Cons: Slow translation, since need to search whole table to lookup a page



Page Replacement Algorithms

- Motivation:
  - Suppose no free physical frame during page fault
  - Which page should be evicted?
- When evicting page:
  - Clean page: Not modified → No need to write back
  - Dirty page: Modified → Need to write back
- Evaluation Criteria: Need to reduce page faults
  - $T_{\text{access}} = (1-p)T_{\text{mem}} + (p)T_{\text{page fault}}$

- Since $T_{\text{page fault}}$ (Access time of page fault) $>>$ T_{mem} (Access time for memory resident page), need to reduce p for minimal T_{access}

Optimal (OPT)

> Evict page that will not be used again for longest period of time

- Guarantees minimum number of page faults
- But impossible, since need future knowledge of memory references
- Useful as a base comparison for other algorithms

Example: OPT (6 Page Faults)

Time	Memory Reference	Frame			Next Use Time			Fault?
		A	B	C				
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	∞	∞	∞	

First-in First-out (FIFO)

> Evict oldest memory page based on initial loading time

- Not LRU: Only based on **initial** loading time

Example: FIFO (9 Page Faults)

Time	Memory Reference	Frame			Loaded at Time			Fault?
		A	B	C				
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

- Implementation: OS maintains queue of resident page numbers
  - Simple
- Cons:
- Belady's Anomaly: More physical frames $\rightarrow$ More page faults, which is counter-intuitive
  - Since FIFO can evict MRU frame $\rightarrow$ Does not exploit temporal locality

Least Recently Used (LRU)

> Evict page that has not been used in the longest time

- Exploits temporal locality heuristic: Have not used for long time $\rightarrow$ Likely will not be used again

Example: LRU (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	

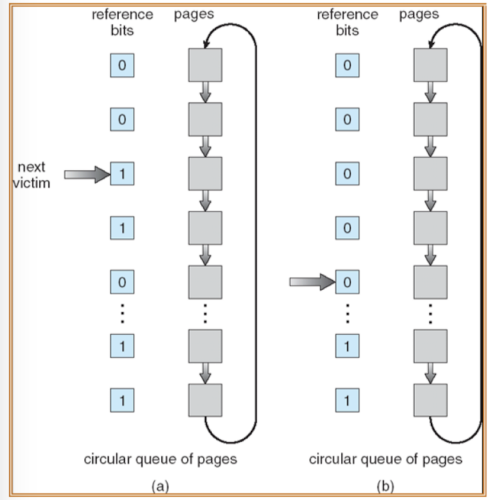
- Pros:
  - Does not suffer from Belady's Anomaly
  - Similar performance to OPT algorithm
- Cons: Hard to implement in hardware
- Implementation:
  - Approach #1: Use a fake "time" counter, which is incremented for every memory reference
    - Update "time-of-use" field in page table entry whenever reference occurs
    - Problems:
      - Need to search through all page for smallest "time-of-use" field
      - "Time-of-use" forever increasing $\rightarrow$ Overflow possible
  - Approach #2: "Stack"
    - If page is referenced, then remove from stack and push on top of stack
    - Replace page from bottom of stack $\rightarrow$ No need to search through all entries
    - Problems: Not pure stack, since entries can be removed from anywhere in stack

Second-Chance (Clock)

> Modified FIFO to give second chance to pages that are accessed

- Each PTE maintains a reference bit
  - 0 when newly loaded and not accessed again
  - 1 when accessed again
- Replacement algorithm:

    1. Maintain pointer to oldest FIFO page
    2. If page's reference bit is 0, then page is replaced
    3. If page's reference bit is 1, then page is given a 2nd chance
        1. Reference bit cleared to 0 → Page taken as newly loaded
        2. Point to next FIFO page and return to step 2
- If all pages have reference bit as 1 or 0, then basically FIFO
- Implementation: Circular queue of pages with pointer to oldest page (Victim page)



Example: CLOCK(6 Page Faults)

Time	Memory Reference	Frame (with Ref Bit) A	B	C	Fault?
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)	

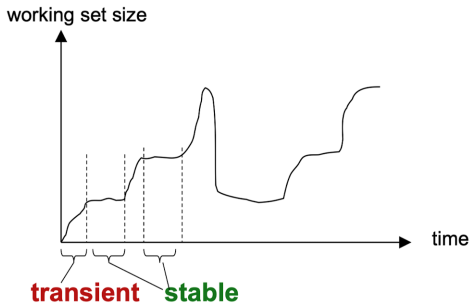
▶ Victim Page

Frame Allocation

- Motivation: With N limited physical memory frames, how do we distribute them among M processes?
- Approaches:
    - Equal allocation: Each process gets N/M frames
    - Proportional allocation: Each process gets $\text{size}_p/\text{size}_{\text{total}} * N$
- Local replacement: Victim page selected among pages of process that caused page fault
    - Implicit assumption for above page replacement algorithms discussed
    - Pros: Number of frames allocated to process is constant → Performance is stable between runs
    - Cons: If not enough allocated frames → Lots of page faults → Slow
- Global replacement: Victim page can be chosen among all frames
    - Pros: Allow self-adjustment between processes (i.e., process that needs more frames can get from others)
    - Cons:
        - Badly behaved processes (e.g., hello world with 1 GB array) will steal pages from other processes
        - Number of frames allocated to a process can vary

Working Set Model

- Motivation: Hard to find right number of frames! → Lack of frames lead to thrashing
    - For local replacement: Thrashing limited to 1 process → Can hog I/O and slow down other processes
    - For global replacement: Thrashing steals frames from other processes → Other processes thrash too (Cascading thrashing)
- Observation:
    - Locality: Set of pages referenced by process is relatively constant in a period of time
    - In a new locality, lots of page faults for the set of pages
    - Once set of pages loaded in, little page faults
- Solution:
    - Working Set Size $W(t, \Delta)$: Active pages in the Working Set Window Δ (Past time interval) at time t
    - Allocate enough frames for pages in $W(t, \Delta)$ to reduce page faults

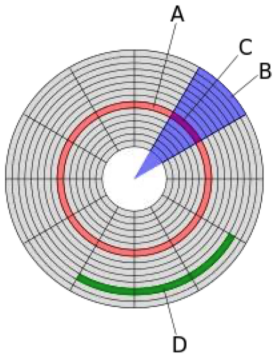


- Transient region: Transition between localities → Working set changing in size
- Stable region: Working set size stable
- Choice of Δ important:

- Too small: May miss pages in current locality
- Too big: May contain pages from different locality

File System Management

- Motivation: Physical memory is volatile → Use external storage to store persistent information, even after OS terminates → Need some abstraction to manage storage
- Criteria:
  - Self-contained: Can "plug-and-play" on any system
  - Persistent: Beyond lifetime of OS and processes
  - Efficient: Good management of free and used space with minimum overhead for bookkeeping information



- Hard disk layout:
  - Track (A)
  - Sector (C): Smallest physical storage units
  - Cluster (D): (or block) Group of contiguous sectors that acts as the logical unit of file storage
  - Platter: Entire disc; both sides can store data
  - Cylinder: Tracks in same location on each platter

	Memory Management	File System Management
Underlying Storage	RAM	Disk; SSD: Similar to RAM but organized like disk for uniform interface
Access Speed	Constant	Disk: Variable I/O time since need to move arm; SSD: Constant
Unit of Addressing	Physical memory address	Disk sector
Usage	Address space for process (Implicit when process runs)	Non-volatile data (Explicit access)
Organization	Paging/segmentation	ext *(Linux), FAT* (Windows), HFS* (MacOS), etc.

File System Abstraction

- File system: Collection of files and directories with abstraction of accessing and using them

File

> Abstraction representing logical unit of information created by process

- Contains metadata and data

File Metadata

> Additional information associated with the file (aka file attributes)

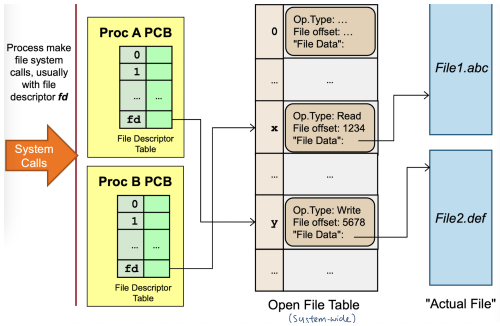
- File name: Different file systems have different naming rules
  - E.g., `Name.Extension`
- Identifier: Unique ID for file used internally by FS
- Size: Current size of file
- Time, date, and owner information: Creation, last modification time, owner ID, etc.
- Table of content: Information for FS to check how to access the file
- File type: Each type supports different operations and requires different programs for processing
  - Common types:
    - Regular files: User information
      - ASCII files: Readable as is (e.g., text files, source code)
      - Binary files: Have predefined structure that can be processed by specific program (e.g., Java class file)
    - Directories: System files for FS structure
    - Special files: Character/block oriented
  - Distinguishing file type:
    - Use file extension (Windows)
    - Magic number stored at beginning of the file data (Unix)
- File protection: Access control to file data
  - Type of access: Read, write, execute, append, delete, list (Read metadata of file)
  - How?
    - Permission Bits: Classify users into 3 classes
      - Owner: User who created the file
      - Group: Set of users who need similar access to a file
      - Universe: All other users in system
    - Access Control List: File stores list of user identities and allowed access type → More customizable, but long
      - Unix: Minimal ACL (Permission bits) vs. Extended ACL (Permission bits + Specific users/groups)
- Operations on file metadata: Rename, change attributes, read attribute

File Data

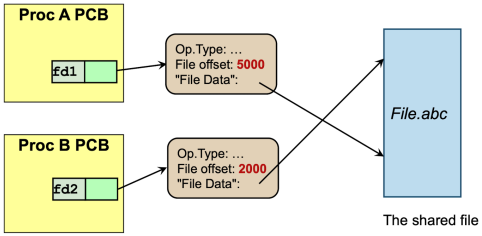
- Structure:
  - Array of bytes: Each byte has unique offset from file start
  - Fixed length records: Array of records (sections) that can grow and shrink
    - Can jump to any record easily: $\text{Size of record} * (N - 1)$
  - Variable length records: More flexible but harder to locate a record
- Access methods:
  - Sequential access: Data read in order from the start; cannot skip but can rewind (like a tape drive)
  - Random access: Data can be read in any order
    - Read: Specify offset position for every read/write
    - Seek: Jump to offset position in file, then do read/write
  - Direct access: Random access for **fixed-length** records
    - Random access is special case of direct case where each record is 1 byte
- Operations: Create (New file with no data), open (OS creates structures to read/write file), read, write, repositioning (aka seek), truncate (Remove from position to end of file)

File Operations

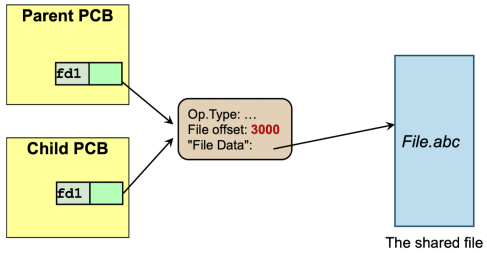
- OS provides file operations as system calls
- Information for an opened file:
  - File pointer: Current location in file
  - Disk location: Actual file location on disk
  - Open count: How many processes have this file opened
- Problem: Several processes can open several files → How to organize this open-file information?
- Approach:
  - System-wide open-file table: One entry per unique file
  - Per-process open-file table (aka file descriptor table): One entry per file used in the process
    - Each entry points to system-wide table



- 2 processes share a file using **different** file descriptors
  - E.g., 2 separate open commands
  - I/O (read/write) occurs at different and independent offsets



- 2 processes share a file using the **same** file descriptor
  - E.g., `fork()` after file is opened
  - Only 1 offset: I/O (read/write) **shifts** offset for other process



Directory

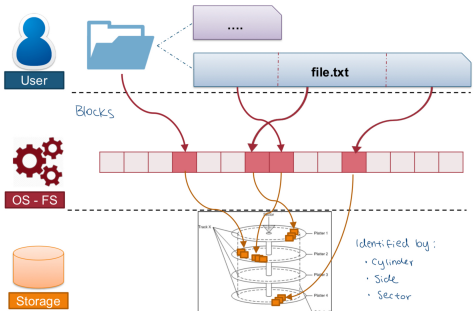
> Provides logical grouping of files and tracks file locations

- Actually just a file
- How to structure directories?
  - Single-level: Only root directory with all files inside
  - Tree-structured: Directories recursively embedded in other directories
    - Absolute pathname: Path from root directory (`/`) to the file
    - Relative pathname: Directory names followed from the current working directory (`./`)
      - Parent directory: `../`
  - Directed Acyclic Graph (DAG): Tree-structured, but a file can be shared and appears in multiple directories
    - Consider: Directory A is owner of file F and directory B wants to share F
    - Hard Link: Directory A and B have separate pointers pointing to file F in disk
      - Can link to files only
      - Pros: Low overhead since only pointers are added in directory
      - Cons:
        - Directory can only point to files in the same FS
        - Deletion problem: Each file has a reference counter → Need to delete all pointers to delete file
      - Application: Installation scripts create hard links to installation binary, instead of copy

    - Unix: `ln`
  - Symbolic Link: Directory B creates **special link file** G, which contains path name of file F
    - Can link to files and directories $\rightarrow$ Can introduce cycles!
    - Pros:
      - Path can point to different FS (e.g., network FS)
      - Deletion:
        - If B deletes, then G is deleted and not F
        - If A deletes, then F is deleted and not G (but not working)
    - Cons: Larger overhead
    - Unix: `ln -s`
- General Graph: Has cycles
  - Hard to traverse since need to prevent infinite loops
  - Hard to determine when to remove a file or directory

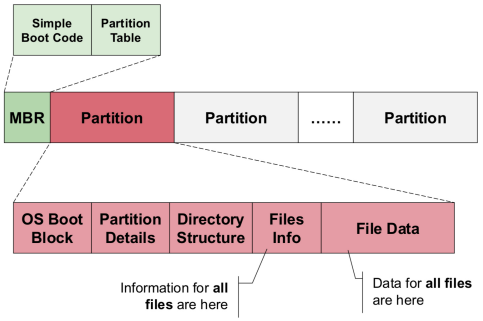
File System Implementations

Disk Structure



- Treat as array of logical blocks
  - Logical block: Smallest accessible unit
- Logical block is mapped into disk sectors

Disk Organization



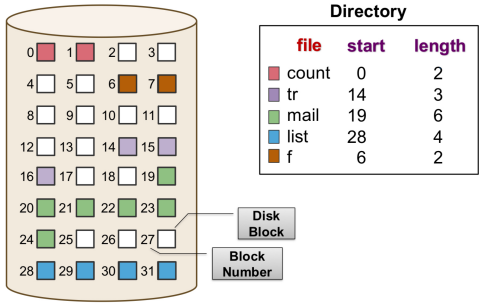
- Master Boot Record (MBR): Boot code to start up OS boot code in individual partitions
  - Each partition can have its own OS and file system
- Partition table: Stores location of each partition
- Partition Details: Total number of blocks + Number and location of free disk blocks
- Directory Structure: Tracks files in a directory by **mapping file name to File Information entry**
  - Windows: File Information entry (including metadata) stored in directory entry
  - Unix: Mapping of file name to i-node number
- Files Information: Information for all files
  - Unix: List of indexed nodes (I-node)
    - I-node: Stores file meta data and pointers to file data (See indexed allocation for data structure)
- File Data: Actual data

File Block Allocation

> How to allocate file data on disk?

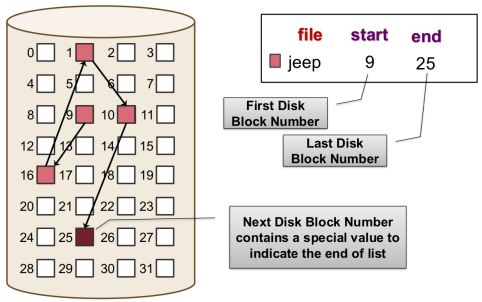
- If file size is not a multiple of logical blocks, then internal fragmentation in last block
- Good file implementation:
  - Track logical blocks and whether they are used or free
  - Allow efficient access
  - Utilize disk space efficiently

Contiguous Block Allocation



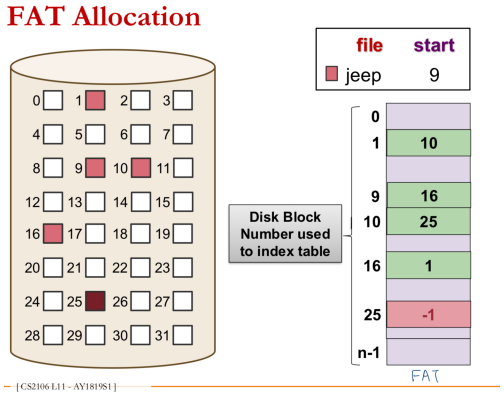
- Idea: Allocate consecutive disk blocks to a file
  - Directory Structure stores mapping from file name to entry in Files Information, which contains `start` and `length`
- Pros:
  - Simple to track: Each file only needs starting block number and length
  - Fast access: Only need to seek to first block
- Cons:
  - External fragmentation
  - File size needs to be specified in advance

Linked List Allocation



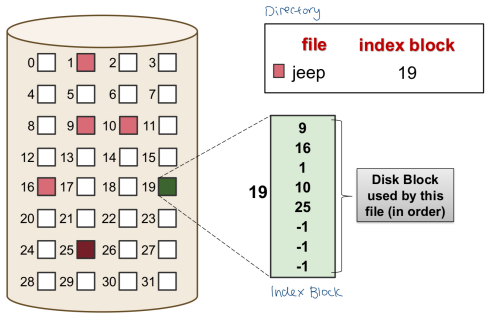
- Idea: Linked list of disk blocks
  - Each disk block stores file data and **next disk block number** (like a pointer)
  - File Information stores start and end (optional) disk block number of a file
- Pros: No external fragmentation
- Cons:
  - Random access in a file is slow, since reading any block requires sequential access in hard drive, which moves disk arm
  - Part of disk block is used for pointer
  - Less reliable (e.g., 1 wrong pointer)

File Allocation Table (FAT) Allocation



- Idea: Move all block pointers to a single table (aka FAT)
  - FAT is in **main memory** at all times
  - File Information entry stores start disk block number of a file to access FAT
- Pros: Faster random access
  - Still sequential, but much faster in main memory
- Cons:
  - FAT tracks all disk blocks in a partition, which can be huge
  - Does not solve linked list's reliability issue

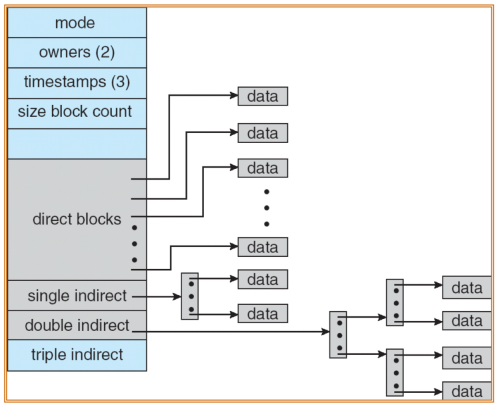
Indexed Allocation



- Idea: Each file has an **index block** occupying a disk block
  - Index block: Array of disk block addresses used by file in sequence
  - File Information entry stores disk block number of file's index block
- Pros:
  - Less memory overhead, since only index block of opened file needs to be in main memory, instead of all disk blocks
  - $O(1)$ direct access
- Cons:

- Maximum file size limited by size/number of entries in an index block
- Overhead from storing index block

- Variations to allow larger file size:
  - Linked scheme: Linked list of index blocks
    - Each index block contains pointer to next index block
  - Multi-level index: Similar to multi-level paging
    - First-level index block points to number of second-level index blocks
  - Combination of direct indexing and multi-level indexing: Can store small files efficiently while still allowing large files (Trade-off)
    - E.g., I-node, apart from containing file metadata
      - 12 direct pointers
      - 1 single indirect block
      - 1 double indirect block
      - 1 triple indirect block



Free Space Management

- Motivation: Before allocating file, need to know which disk block is free
- Free space information maintained by **Partition Details**

Bitmap

- Idea: Each disk block represented by 1 bit
  - E.g., 1 $\rightarrow$ Free; 0 $\rightarrow$ Used
- Pros: Good set of manipulations using bit-level operations
- Cons: Need to keep in memory for efficiency reasons
  - Bitmap can be huge depending on number of blocks

Linked List

- Idea: Linked list of disk blocks (aka free space disk blocks), where each disk block contains:
  - Free disk block numbers
  - Pointer to next free space disk block
- Pros:
  - Easy to locate free block, since just need to load a disk block number from free space disk block
  - Only first pointer needs to be in memory
- Cons: High overhead, since takes up disk blocks

Directory Implementation

> Given full path name, search directories recursively along the path to get file information

- File must be opened before use
  - Purpose: Find file information and load into main memory
- Only **root directory** has a dedicated section in disk drive (i.e., **Directory Structure**)
  - Subdirectories are in the form of files, with similar structure as the root directory

Linear List

- Idea: Directory consists of a list, where each file entry contains:
  - File name
  - Pointer to file information in Unix (or file information itself in Windows)
- Cons: Requires linear search, which is inefficient for large directories
  - Solution: Cache last few searches

Hash Table

- Idea: Directory contains hash table of size N
  - If chained collision resolution used, then check hash table entry's linked list for matching file name
- Pros: Fast lookup
- Cons:
  - Hash table has limited size
  - Depends on good hash function

File System in Action

- Run-time information is needed when user interacts with file during runtime
  - E.g., load file information into memory when opening file
  - Maintained in OS context of process
- Create `/parent/F`:
  1. Use path to locate parent directory
  2. Search for F to avoid duplicates
     1. If found, file creation terminates with error
  3. Use free space list to find free disk blocks
  4. Add entry to parent directory

- Process P opens file `/parent/F`:
    1. Search **system-wide open-file table** for existing entry E
        1. If found, create an entry in P's **per-process open-file table** to point to E and return a pointer to this entry
    2. Use path to locate file F
        1. If not found, open operation terminates with error
    3. When F is located, file information loaded into new entry E in system-wide table
    4. Create entry in P's per-process table to point to E and return a pointer to this entry
    5. Pointer used for further read/write operations