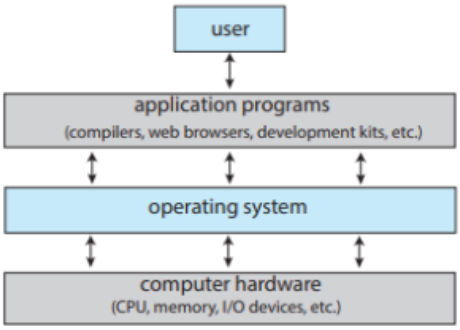
CSOPESY Notes
Term 1, AY 2024-2025

Introduction to Computing Systems

Operating System

- program that acts as an intermediary between user of a computer and computer hardware
- Goals:
  - execute user programs and make solving user problems easier
  - make computer system convenient to use
    - provides API for drivers
  - use computer hardware in an efficient manner
    - sharing resources, having system updates

Computer Structure



- **Hardware Components**
  - physical components and drivers
  - Ex: CPU, memory, I/O, storage
- **Operating System**
  - controls and coordinates use of the hardware with user applications
- **Application Programs**
  - define ways which the system resources are used to solve the problem of the users
  - Ex: word processors, browsers, compilers, video games
- **Users**
  - people, machines, other computers

Why study OS?

- all code runs on top of the OS (legacy code)
- additional knowledge that can lead to proper, efficient, effective, and secure programming

Primary Tasks of OS

- **bridge/API** to hardware resources
- has a program that runs at all times on the computer called a **kernel**, alongside others:
    - **System programs**: software not part of the kernel but associated with the OS (different from application programs)
    - **Middleware**: set of software frameworks that provide additional services to application developers
- **Resource Allocator**
    - manages all resources, how they are shared among programs
    - decided between conflicting requests for efficient and fair resource use
    - allocates resources such as CPU and memory to an application
- **Control Program**
    - controls execution of programs to prevent error and improper use of computer resources (CPU and memory hoggers, infinite loops, etc.)
    - OS ensures all applications receive fair number of resources
    - each has components where allocations are already pre-defined:
        - **Main Program**: points to main function
        - **Subroutine**: contains all functions declared in code
        - **Libraries**: external functions or dependencies required
        - **Symbol Table**: holds variables (hash table)
        - **Stack**: holds temporary variables for function calls, also stores return address of a given function call
        - **Heap**: memory for dynamically allocated objects
- **Security**
    - must protect application's data from one another (unless designed to share data)
    - allocate resources fairly and efficiently, settles conflicting requests for resources
    - prevent errors and improper use of hardware
- **Improvisation**
    - find a way to improvise when resources are scarce
    - provides illusion of dedicated machine with infinite memory and processors
    - if there are multiple programs utilizing most/all memory and CPU, it still works because of **virtual memory**, wherein some processes that are less active would be stored in a secondary storage (i.e. disk) to swap it with another process whenever RAM is used up
- **Process Management**
    - creating and deleting both user and system process
    - scheduling process and threads on the CPU
    - suspending and resuming processes
    - providing mechanisms for process synchronization and communication

- **Memory Management**
  - keeping track of which parts of memory are currently being used and which process is using them
  - allocating and deallocating memory space as needed
  - deciding which processes (or their parts) and data to move in and out of memory
- **File-System Management**
  - creating and deleting files
  - creating and deleting directories to organize files
  - supporting primitive for manipulating files and directories
  - mapping files onto mass storage
  - backing up files on stable (nonvolatile) storage media
- **Cache Management**
  - **Cache**: temporary storage that is fast (i.e. main memory)
  - registers
  - manages hierarchy of storage
- **I/O System Management**
  - memory-component that includes buffering, caching, and spooling
  - general device-driver interface
  - drivers for specific hardware devices

OS Services

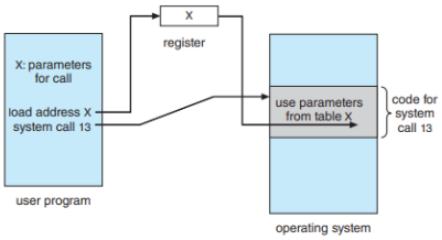
- OS provides execution environment for programs by providing services available
- available services varies depending on the OS and hardware
- User
  - **User Interface**
    - refers to display interface and reading of peripheral inputs (keyboard, mouse, touch screen), allows user to interact with computer
    - early OS started with just **command-line interface (CLI)**, which uses text commands and keyboard for entering them; nowadays, CLI and GUI
    - Examples:
      - Start menu and taskbar
      - Changing taskbar layout and wallpaper
      - Snap feature to show two programs side-to-side
      - Sidebar for notifications
      - Drawing NVIDIA Geforce experience overlay
  - **Process Control**
    - Examples:
      - Viewing active programs in task manager
      - Elevating priority for a certain process

- **Program Execution**
  - Examples:
    - Double-clicking an executable file
    - Training neural network using NVIDIA GPU and Pytorch
    - Running the built-in calculator
- **I/O Operations**
  - Ex: Copy pasting text across different files
- **File System Manipulation**
  - Examples:
    - Deleting browser history
    - Saving PNG images via MS Paint
    - Emptying recycling bin
- **Communications**: shared memory, message passing
  - Examples:
    - Choosing between WiFi or Ethernet cable
    - Printer sharing in a LAN network
    - Windows background service for checking for OS updates
    - Socket implementation
- **Device Management**
  - Examples:
    - Expanding specific device categories
    - Printing a PDF file
    - Increasing CPU/GPU fan speed
    - Changing behavior of RGB lights
    - Setting up monitor refresh rate
- **Error Detection**

- Operations
  - **Resource Allocation**
    - Examples:
      - Viewing the RAM usage in task manager
      - Setting up pages/frames in Windows
  - **Logging**
    - Examples:
      - Generation of error reports of non-responsive programs
      - Print statements in a C++ program
  - **Protection and Security**
    - Examples:
      - Adding a new family account
      - File permissions (read/write access)

System Calls

- programming interface provided by the OS
- usually written in high-level language (C/C++)
- mostly accessed via API
- Implementation:



  - **system call interface** serves as a link to system call made available to OS and the **Run Time Environment (RTE)**
  - typically, system call is assigned a number and the **System Call Interface** maintains a table indexed according to these numbers
  - Parameter passing: registers, block or table from memory
- Example:

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

  - they are generally abstracted (usually in C)
- Types:
  - Process Control
    - end, abort
    - load, execute
    - create process, terminate process
    - get process attributes, set process attributes
    - wait for time
    - wait event, signal event
    - allocate and free memory

- File Management
    - create file, delete file
    - open, close file
    - read, write, reposition
    - get and set file attributes
- Device Management
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices
- Information Maintenance
    - get time or date, set time or date
    - get system data, set system data
    - get and set process, file, or device attributes
- Communications
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach and detach remote devices

OS Portability

- due to variances of system calls from OS to another, programs compiled for one are not executable on other OSses
- portable apps are still possible using following techniques:
    - interpreted programming languages (Python/Ruby) has an **interpreter** that reads each line and executes equivalent instructions native to OS
    - having their own full **RTE** (Java) that abstracts the system calls from the program
    - **standardized use of API** related to compilation of binaries (POSIX/UNIX)

OS Structures

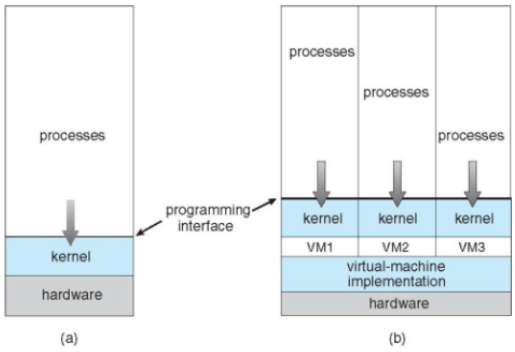
- all components are interconnected and connected to create a kernel
- **Monolithic**
    - all functionality is in a single, static binary file
    - tightly coupled
    - Ex: Linux
- **Layered**
    - modular components are layered where the **lowest** layer is the **hardware** and the **highest** layer is the **UI** (structure of most OS)
    - **loosely coupled**, changes can be modular (making debugging easier)

- **Microkernels**
  - **Carnegie Mellon University** developed an OS called **Mach** that modularized the kernel using this approach:
    - removing nonessential components from kernel and implementing them as a user-level program on a separate address space
    - smaller kernel, usually process and memory management, in addition to a communication facility, are retained
      - makes extending of OS easier
  - current model of most OS
  - services are added to use space without modification to the kernel (modular)
  - if service fails, rest of OS is untouched
  - increased system function overhead as messages must be copied between services, which reside in separate address spaces.
  - likewise, additional process switching maybe needed to exchange the services
  - Ex: **Darwin** (in iOS, macOS)

Sample OS Structures

- macOS and iOS Structure
  - **User Experience Layer**
    - defines software interface that allows users to interact with device
    - Ex: Aqua (macOS) for mouse/trackpad; Springboard (iOS) for touch
  - **Application Framework Layer**
    - includes Cocoa (for macOS apps) & Cocoa Touch (for iOS apps) frameworks, which provide API for Objective-C and Swift
  - **Core Frameworks**
    - defines frameworks that support graphics & media (Quicktime, OpenGL)
  - **Kernel Environment** (Darwin)
    - includes Mach microkernel and BSD UNIX kernel
- Android Structure
  - virtualized execution environment via Android Runtime ART (formerly Dalvik)
    - performs **ahead-of-time** compilation for performance considerations
  - Java programs compiled to Java bytecode .class file and translated to executable .dex file (will be compiled to native machine code after installation on device)
  - Java Native Interface JNI is available such that developers can bypass the virtual machine, but not portable
  - **Hardware Abstraction Layer (HAL)** abstracts hardware to the developers as Android may run on almost unlimited number of devices
  - **Glibc** was replaced by **Bionic C** library for Android
  - based on UNIX as it has mobile optimized **Linux kernel** at bottom of stack

Virtual Machines

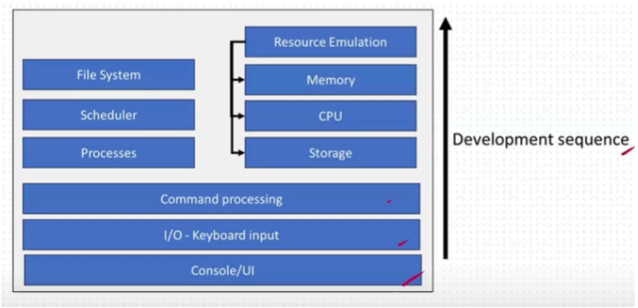


(a) Nonvirtual machine (b) virtual machine

- takes layered approach to its logical conclusion, as it treats hardware and operating system kernel as though they were all hardware
- provides interface identical to underlying bare hardware
- OS host creates illusion that a process has its own processor (virtual memory)
- each guest is provided with a (virtual) copy of the underlying computer
- first appeared commercially in IBM mainframes in 1972
- fundamentally, multiple execution environments (different operating systems) can share the same hardware
- protect from each other
- some sharing of files can be permitted, controlled
- communicate with each other, and other physical systems via networking
- useful for development, testing
- consolidation of many low-resource use systems onto fewer busier systems
- “Open Virtual Machine Format”, the standard format of virtual machines, allows a VM to run within many different virtual machines (host) platforms

Overview of OS Emulator

OS Emulator



- resources are being emulated since no access to the hardware (since not a kernel)
- CPU = Thread
- Process = { pid, pname, # of instructions, list of instructions }
    - often handled by the compiler
- Memory Manager: allocates chunks of memory needed by processes
    - entails using malloc(), etc.

Pseudocode of Sequence

```
// in immediate-mode (real-time), unlike web apps and browsers which are in retained mode
void enterMainLoop() {
        while (< user hasn't initiated shutdown>) {
                // Continuously handle I/O handling and system calls
                // Dispatch user processes and manage execution
                // If idle for X seconds, start a certain background service
                // Check for updates...
        }
}

int main() {
        // POST (hardware-level) first, then bootstrap

        bootstrap();
        initializeKernel(); // Compiled version of the OS (ex: .ISO)
        startSystemServices(); // Inside kernel : starts background processes
        showWelcomeUI();
        enterMainLoop(); // Lifecycle of OS

        shutdownAndCleanup(); // Gracefully terminate running processes and services
                                // Halt or reboot the system
}
```

Typical Sequence

- **Bootstrapping**
  - perform **low-level initialization** (hardware setup, memory initialization, etc.)
  - links with the BIOS (basic input/output system) to start up the OS
  - load kernel into memory and start executing it
  - Examples:
    - perform hardware initialization
    - initializing Hardware Abstraction Layer (HAL)
    - enabling Floating Point Unit (FPU)
    - setting up boot loader environment
    - configuring the Memory Management Unit (MMU)
    - initializing the System Management Mode (SMM)
    - configuring the Unified Extensible Firmware Interface (UEFI)
    - setting up kernel command line parameters
    - setting up the Real Mode Interrupt Vector Table (IVT)
- **Kernel Initialization**
  - initialize **data structures** (process table, file system, etc.)
  - set up interrupt handlers and device **drivers**
    - mapping of the hardware to software (OS components)
  - initialize **memory management** and **scheduling** algorithms
  - Examples:
    - initialize process table and memory manager (e.g. memory allocation, demand paging)
    - setting up the Global Descriptor Table (GDT)
    - configuring the Interrupt Descriptor Table (IDT)
    - initializing the Advanced Configuration and Power Interface (ACPI)
    - enabling virtual memory and setting up page tables
    - initializing the Programmable Interval Timer (PIT)
    - starting the initial process scheduler
    - configuring the Memory Management Unit (MMU)
    - enabling symmetric multiprocessing (SMP) support
    - initializing the Real-Time Clock (RTC)
    - configuring input/output ports and devices
- **Start System Services**
  - **System Services**: user side representation
  - start essential system services (file system, networking, etc.)
  - launch system daemons and background processes
  - processes persistent in the OS, which means they are **always alive**/periodically invoking functions

- Examples:
    - start file system service and network service
    - loading the initial RAM disk (initrd)
    - loading kernel modules and drivers
    - starting the initial RAM disk filesystem
    - initializing the system logger
    - setting up the initial file system
    - configuring network interfaces and protocols
    - initializing user authentication mechanisms
    - loading and initializing system libraries
    - setting up inter-process communication (IPC) mechanisms
    - configuring power management features
    - initializing system time and date settings
    - starting system monitoring and performance analysis tool

- **Enter Main Loop**
    - keep running programs meant to run either something interrupts or it shuts down
    - continuously handle interrupts and system calls
    - dispatch user processes and manage their execution
    - handle user input and manage I/O operations (e.g. open, close files)
    - Examples:
        - enter an infinite loop to handle system events
        - starting the initial process scheduler
        - starting the graphical user interface (GUI)
- **Shutdown and Cleanup**
    - gracefully **terminate** running processes and services
    - **clean** up allocated resources and release memory
    - halt or reboot the system
    - Examples:
        - clean up resources and prepare for shutdown
        - saving system configuration settings to persistent storage
        - flushing system buffers and caches
        - terminating user processes and services gracefully
        - releasing allocated system resources and memory
        - closing open file handles and network connections

Scheduling

- backend of OS
- shows **CPU** utilization (percent, cores available vs used), running and finished processes, timestamp of processes (when did it start), current line of instruction per process
- scheduling simulation → simulates various CPU scheduling algorithms, given *N* processes and *X* CPU cores (simulated/declared during compile-time)
- arbitrary number of processes are scheduled, each containing *Z* commands
    - commands can either be simple print or an I/O operation such as searching for a file or a request for a driver/hardware
- can display a summary of the order of processes finished/still executing

Memory Management

- simulation that combines memory management + scheduling
    - using existing CPU schedulers implemented, processes only execute if sufficient memory is available
- has command that displays memory utilization and running process (e.g. nvidia-smi)
- processes have a predefined amount of memory needed to execute
    - emulator allocates needed memory if available
- allows user input-driven creation of processes on the fly
    - while having a predefined list of running processes, the user can add new ones
    - all processes must finish eventually → CPU & memory utilization should 0%
- allows real-time viewing of CPU and memory utilization while still typing commands

Crash Course in C++

Reference
- Documentation: cppreference.com

Dear IMGUI
- an immediate-mode application (constantly being refreshed)

Filters
- serves as actual filter for organizing in Visual Studio (done through the Solution)

Classes
- can inherit from Abstract class
- can have enum attributes

Virtual Functions
- an abstract function without any real use (read-only)

Forward Declaration
- goes to the stack
- deallocate immediately when variable goes out of scope
- for dynamic memory allocation (e.g. smart pointers), it goes to the heap
    - *Tip: Only use smart pointers if the class will likely persist for a long time or used in different functions*

Some Data Structures
- Linked List (Vector): vector<type> name;
- Unordered Map (Hash Table): unordered_map<type> name;

I/O and Display Interfaces

Console Layouting

- CLIs typically have multiple layouts that can be drawn on screen depending on context
- Basic Implementation: have a mechanism to draw new screen → back to previous screen
- Example:
    - a **Console Manager**
        - has multiple AConsole (abstract class of a console), each having UI/screens of their own
        - in summary, it is a **singleton** instance
            - a design pattern
            - a class where once instantiated, is accessible in global and is only one instance
        - Attributes: list of consoles
    - the **Consoles** (abstract class)
        - has ConsoleManager as a **friend** class (can access public, protected, and private attributes)
    - Doing initialization (constructor)
        - in CLI, each UI is registered under a function to register screen
        - all the screens are then stored to the hash table
        - screen would be set to the main menu
        - there would be a BaseScreen (a reference to Process) in AConsole

I/O Systems

- I/O management is a major component of OS design and operation:
    - important aspect of computer operation
    - I/O devices vary greatly
    - various methods to control them
    - performance management
    - new types of devices frequent
- ports, busses, device controllers connect to various devices
- **Device drivers** encapsulate device details, present uniform device-access interface to I/O
- incredible variety of I/O devices: storage, transmission, human-interface
- Common concept: signals from I/O devices interface with computer
    - **Port**: connection point for device
    - **Bus**: daisy chain or shared direct access
        - PCI bus common in PCs and servers, PCI Express (PCIe)
        - expansion bus connects relatively slow devices
        - serial-attached SCSI (SAS) common disk interface

    - **Controller** (host adapter): electronics that operate port, bus, device
        - sometimes integrated
        - sometimes separated circuit board (host adapter)
        - contains processor, microcode, private memory, bus controller, etc.
        - some talk to per-device
- devices have addresses, used by:
    - **direct I/O instructions**
    - **memory-mapped I/O**
        - device data and command registers mapped to processor address space, especially for large address spaces (graphics)

Application Behavior

- **Real-Time**: screen always refreshes even if there's no event/user inputs
    - Ex: games, interactive applications (e.g. streaming)
- **Event-Driven**: screen refreshes after user input
    - Ex: CLI

Polling

- for each byte of I/O:
    - read busy bit from status register until 0
    - host sets read or write bit and if write copies data into data-out register
    - host sets command-ready bit
    - controller sets busy bit, executes transfer
    - controller clears busy bit, error bit, command-ready bit when transfer done
- Step 1 is busy-wait cycle to wait for I/O from device
    - reasonable if device is fast, but inefficient if device slow
    - CPU switches to other tasks, but if miss a cycle data is overwritten/lost
- can happen in 3 instruction cycles
    - read status, logical-and to extract status it, branch if not zero
- **Interrupts**
    - CPU interrupt-request line triggered by I/O device (checked by processor after each instruction)
    - interrupt handler receives interrupts (maskable to ignore/delay some interrupts)
    - interrupt vector to dispatch interrupt to correct handler
        - context switch at start and end
        - based on priority
        - some nonmaskable
        - interrupt chaining if more than one device at same interrupt number
    - used for exceptions

- terminate process, crash system due to hardware error
- page fault executes via trap to trigger kernel to execute request
- multi-CPU systems can process interrupts concurrently (if OS designed to handle)
- used for time-sensitive processing, frequent, must be fast

Direct Memory Access

- used to avoid programmed I/O (one byte at a time) for large data movement
- requires DMA controller
- bypasses CPU to transfer data directly between I/O device and memory
- OS writes DMA command block into memory

Application I/O Interface

- I/O system calls encapsulate device behaviors in generic classes
- device-driver layer hides differences among I/O controllers from kernel
- new devices talking already-implemented protocols need no extra work
- each OS has its own I/O subsystem structures and device driver frameworks
- devices vary in many dimensions
  - character-stream or block
  - sequential or random-access
  - synchronous or asynchronous (or both)
  - sharable or dedicated
  - speed of operation
  - read-write, read-only, or write-only

Network Devices

- varying enough from block and character to have own interface
- Linux, Unix, Windows, and many others include socket interface
  - separates network protocol from network operation
  - includes select() functionality – returns info about which sockets have a packet waiting; eliminates polling
- approaches vary widely (pipes, FIFOs, steams, queues, mailboxes)

Clocks and Timers

- provide current time, elapsed time, timer
- normal resolution about 1/60 second
- some systems provide higher-resolution timers
- programmable interval timer used for timings, periodic interrupts
- ioctl() (on UNIX) covers odd aspects of I/O such as clocks and timers

Nonblocking and Asynchronous I/O

- **Blocking**: process suspended until I/O completed
  - easy to use and understand
  - insufficient for some needs
- **Nonblocking**: I/O call returns as much available
  - user interface, data copy (buffered I/O)
  - implemented via multi-threading
  - returns quickly with count of bytes or written
  - select() to find if data ready then read() or write() to transfer
- **Asynchronous**: process runs while I/O executes
  - difficult to use
  - I/O subsystem signals process when I/O completed

Vectored I/O

- allows one system call to perform multiple I/O operations
- single procedure call sequentially reads data from multiple buffers or reads data from a data stream and writes it to multiple buffers
- scatter-gather method better than multiple individual I/O calls
  - decreases context switching and system call overhead
  - some versions provide atomicity

Kernel I/O Subsystem

- **Scheduling**
  - some I/O request ordering via per-device queue
  - some OSs try fairness
  - some implement Quality of Services (i.e. IPQOS)
- **Buffering**: store data in memory while transferring between devices
- **Caching**: faster device holding copy of data
- **Spooling**: hold output for a device
- **Device Reservation**: provides exclusive access to a device

Power Management

- not strictly domain of I/O, but much is I/O related
- computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
- Ex: wake locks, power collapse

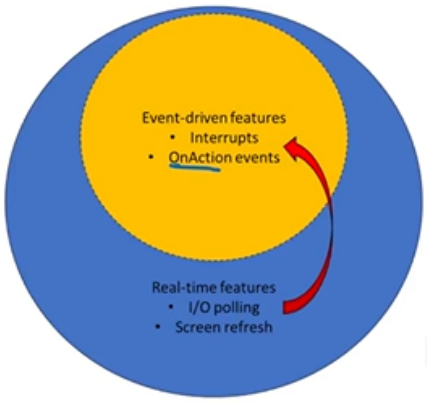
Keyboard Polling

- means to support keyboard input, both real-time and event-driven
    - **Real-time**: user can always type characters, and this displays on-screen immediately (e.g. word processors); OS-driven (e.g. Windows UI)
    - **Event-driven**: wait for an event, such as an enter command, and then perform an associated operation; user applications-driven (e.g. Java GUI)
        - all are derived from real-time applications; still require functionality that needs to run in real time
        - Ex: web applications, mobile applications, networking applications
    - applications can "switch" real-time mode to event-drive mode and vice versa, and they can also run together

Polling in OS

- must be done in real-time rather than event-driven
- if done with std::cin for instance, it will indefinitely wait for input
- without using a background worker, emulator will stall and not do anything

Family of Implementation Features



- Ex: a background in hardware-level I/O systems

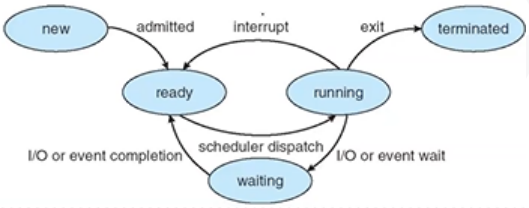
Interrupt Mechanism

- present in all hardware
- in an interrupt-driven system, CPU is interrupted when a key is pressed
- OS or a low-level handler responds to the interrupt and takes appropriate action:
    - a system API call could be exposed where events can be handled by the developer (e.g. onKeyDown(), onKeyUp())
- more event-driven and can be more efficient as CPU is not constantly checking for input
- it first interrupts ("pauses") to poll for a response, then refreshes to draw/update

Processes

Process

- program in execution
  - must progress in sequential fashion
  - no parallel execution of instructions of a single process
- OS executes a variety of programs that run as processes
- Parts
  - **Program Code** (also called text section)
  - Current activity, including **program counter** and processor registers
  - **Stack** containing temporary data
    - Ex: function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
- **Program**
  - passive entity stored on disk (executable file)
  - becomes process (which is active) when an executable file is loaded into **memory**
  - execution started via GUI mouse clicks, command line entry of name, etc.
  - can be several processes (consider multiple users executing the same program)
- as process executes, it changes state per cycle scheduled by CPU:

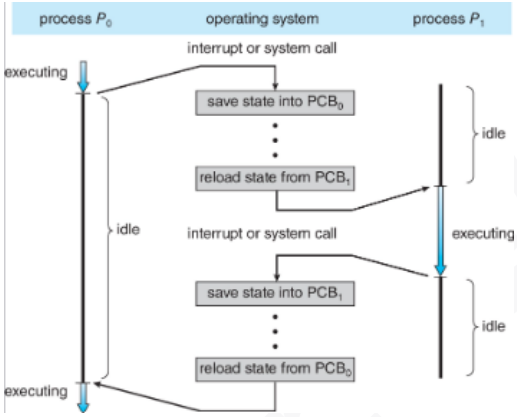


  - **New**: process is being created
  - **Running**: instructions are being executed
  - **Waiting**: process is waiting for some event to occur
  - **Ready**: process is waiting to be assigned to a processor
  - **Terminated**: process has finished execution
- **Process Control Block**
  - information associated with each process (also called **task control block**):
    - Process state: running, waiting, etc.
    - Program counter: location of instruction to next execute
    - CPU registers: contents of all process-centric registers
    - CPU scheduling info: priorities, scheduling queue pointers
    - Memory-management info: memory allocated to the process
    - Accounting info: CPU used, clock time elapsed since start, time limits
    - I/O status info: I/O devices allocated, list of open files

- **Threads**
  - 1 process = 1 thread of execution
  - think of it as having multiple program counters per process
    - multiple locations can execute at once
  - multiple threads of control → threads
  - must have storage for thread details, multiple thread program counters in PCB

Process Scheduler

- selects among available processes for next execution on CPU core
- Goal: maximize CPU use, quickly switch processes onto CPU core
- maintains scheduling queues of processes
  - **Ready Queue:** set of all process in main memory, ready and waiting to execute
  - **Wait Queues:** set of processes waiting for an event (i.e. I/O)
  - processes migrate among the various queues
- **Context Switch**



  - used when CPU switches from one process to another
  - system would **save the state** of the old process and load the **saved state** for the new process
    - **Context**: represented in the PCB
  - pure overhead; the system does no useful work while switching
    - the more complex the OS & PCB, the longer it takes
  - time-dependent on hardware support
    - some hardware provides multiple sets of registers per CPU, allowing multiple contexts to be loaded at once

Multitasking in Mobile Systems

- some allow only one process to run, others suspended
- due to screen real estate, user interface limits iOS provides for a:
    - Single **foreground** process: controlled via UI
    - Multiple **background** process: in memory, running, but not on display
        - Limits: single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
    - background process uses a **service** to perform tasks
    - service can keep running even if background process is suspended
    - service has no UI, small memory use

Process Creation

- **Parent** process create **children** processes which, in turn create other processes, forming a **tree** of processes
- generally, process identified and managed via **process identifier (pid)**
- Resource sharing options
    - parent and children share all resources
    - children share subset of parent's resources
    - parent and child share no resources
- Execution options
    - parent and children execute concurrently
    - parent waits until children terminate
- **Address space**
    - child duplicate of parent
    - child has program loaded onto it
- UNIX
    - **fork()** system call creates new process
    - **exec()** system call used after a fork() to load a new program
    - **wait()** is called by parent to wait for child to terminate

Process Termination

- process executes last statement and then asks OS to delete it using the **exit()** system call
    - returns status data from child to parent (via **wait()**)
    - process resources are deallocated by OS
- parent may terminate the execution of children processes using the **abort()** system call if:
    - Child has **exceeded allocated** resources
    - **Task** assigned to child is **no longer required**
    - Parent is existing, and OS does not allow child to continue if **parent terminates**

- some OS do not allow child to exist if parent is terminated, so OS terminates children:
  - **Cascading Termination**: all children, grandchildren, etc; initiated by OS
- parent process may wait for termination of child process by using the wait() system call, returning the status info and pid of the terminated process via pid = wait(&status)
- if no parent waiting (did not invoke wait()), process is a **zombie**
- if parent terminated without invoking wait(), process is an **orphan**

Process Termination in Mobile OS (Android)

- mobile OS often have to terminate processes to reclaim system resources such as memory, from most to least important:
  - **Foreground process**
  - **Visible process**
  - **Service process**
  - **Background process**
  - **Empty process**
- Android will begin terminating processes deemed least important

Multiprocess Architecture (Chrome Browser)

- many web browsers ran as single process (some still do)
  - if one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 types of processes:
  - **Browser**: manages UI, disk, and network I/O
  - **Renderer**: renders web pages, deals with HTML/JS
    - new one created for each website opened (**1 tab = 1 process**)
    - runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in**: for each type of plug-in

Types of Processes

- **Independent**
- **Cooperating**: can affect or be affected by other processes, including sharing data
  - for information sharing, computation speedup, modularity, and convenience
  - requires **interprocess communication (IPC)**, whose models are either:
    - **Shared Memory**
      - area of memory shared among processes that wish to communicate
      - communication is under the control of the users processes (not OS)
      - major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory

- **Message Passing**
  - processes communicate with each other without resorting to shared variables
  - IPC facility provides two operations: send(msg), receive(msg)
    - message size is either fixed or variable
  - if processes P & Q wish to communicate, they need to:
    - establish a **communication link** between them
      - **Physical**: shared memory, hardware bus, network
      - **Logical**: direct/indirect, sync/async, auto/explicit buffering
    - Types of Communication
      - **Direct Communication**: msg via **send/receive**
        - processes must **name** each other explicitly
        - Properties of the Link:
          - links are established automatically
          - link is associated with exactly one pair of communicating processes
            - between each is **one** link
          - may be unidirectional, but usually **bi-directional**
      - **Indirect Communication**: msg via **mailboxes**
        - each mailbox has a unique id
        - processes can communicate only if they share a mailbox/port
        - Properties of the Link:
          - link established only if processes **share** a common mailbox
          - link may be associated with many processes
          - each pair of processes may share **several** communication links
          - link may be unidirectional or bi-directional
        - Operations:
          - create/delete mailbox
          - send/receive through mailbox
        - primitives are defined as:
          - send(A, message)
          - receive(A, message)

    - Synchronization
        - **Blocking (Synchronous)**
            - **Blocking send**: sender blocked until msg received
            - **Blocking receive**: receiver blocked until msg is up
        - **Non-blocking (Asynchronous)**
            - **Non-blocking send**: senders send msg and continue
            - **Non-blocking receive**: receiver gets valid/null msg
        - different combinations possible
            - if both send and receive are blocking = **rendezvous**
- **Buffering**
    - queue of messages attached to the link
    - implemented in one of three ways:
        - **Zero capacity**: no messages queued on a link
            - sender must wait for receiver (rendezvous)
        - **Bounded capacity**: finite length of *n* messages
            - sender must wait if link is full
        - **Unbounded capacity**: infinite length
            - sender never waits
- **Producer-Consumer Problem**
    - paradigm for cooperating processes
    - producer processes produces information that is consumed by a consumer process
    - Variations:
        - **Unbounded-buffer**
            - places no practical limit on the size of the buffer
            - producer **never waits**
            - consumer waits if there is no buffer to consume
        - **Bounded-buffer**
            - assumes that there is a fixed buffer size
            - producer must **wait if all buffers are full**
            - consumer waits if there is no buffer to consume
            - implemented by having an integer counter that keeps track of the number of full buffers (initially 0)
                - incremented by producer after produces new buffer
                - decremented by consumer after consumes a buffer

Examples of IPC Systems

- POSIX Shared Memory: process first creates shared memory segment
- Mach: messaged based (including system calls), ports for kernel and notify
- Windows: message-passing centric, only works between processes on same system

Other Parts of IPC

- **Pipes**: acts as conduit allowing two processes to communicate
    - **Ordinary pipes**: cannot be accessed from outside the process that created it
        - typically, parent process creates pipe and uses it to communicate with a child process that it created
        - allow communication in standard producer-consumer style:
            - Producer writes to one end (the write-end of the pipe)
            - Consumer reads from the other end (the read-end of the pipe)
        - **unidirectional**
        - require parent-child relationship between communicating processes
        - Windows calls these **anonymous pipes**
    - **Named pipes**: can be accessed without a parent-child relationship
        - more powerful
        - **bidirectional**
        - no parent-child relationship necessary
        - several processes can use it for communication
        - provided on both UNIX and Windows systems

Client Server Systems

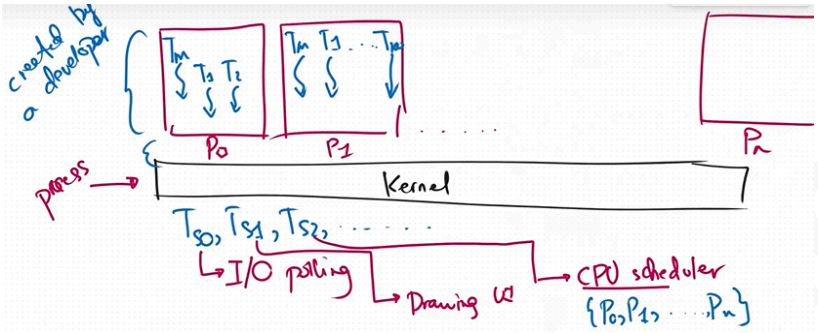
- **Sockets**
    - endpoint for communication
    - concatenation of **IP address** and **port**
        - a number included at start of message packet to differentiate network services on a host
        - Ex: socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
    - communication consists between a pair of sockets
    - all ports below 1024 are **well known**, used for standard services
        - special IP address 127.0.0.1 (loopback) to refer to system on which process is running
    - Types:
        - **Connection-oriented (TCP)**
        - **Connectionless (UDP)**
        - **MulticastSocket class**
            - data can be sent to multiple recipients

- **Remote Procedure Calls (RPC)**
  - abstracts procedure calls between processes on networked systems
    - uses ports for service differentiation
  - **Stubs**: client-side proxy for actual procedure on the server
    - **Client-side**: locates server and **marshalls** the parameters
    - **Server-side**: receives this message, unpacks marshaled parameters, and performs procedure on the server
    - on Windows, stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**
  - data representation handled via External Data Representation (XDL) format to account for different architectures
    - either big-endian and little-endian
  - remote communication has more failure scenarios than local
    - messages can be delivered exactly once rather than at most once
  - OS typically provides a rendezvous (or matchmaker) service to connect client and server

Threading

Threads

- lightweight processes that can handle smaller operations
- a sequence of instructions to execute in **parallel** with another thread
- since it is within a process, **memory is shared**
- multiple threads can access the same data structure, functions, etc.
- in an OS, it should be able to manage processes and threads:



  - each process has a main thread, followed by the other smaller threads
  - in the kernel, it also handles system threads aside from the ones of processes
    - Ex: I/O polling, drawing UI, and CPU scheduler
- Benefits
  - **Responsiveness**: may allow continued execution if part of process is blocked, especially important for user interfaces
  - **Resource Sharing**: threads share resources of process, easier than shared memory or message parsing
  - **Economy**: cheaper than process creation, thread switching has lower overhead than context switching
  - **Scalability**: process can take advantage of multicore architectures

Multicore Programming

- implementation of threads
- challenges include dividing activities, balance, data splitting, data dependency, testing and debugging
- **Concurrency**: supports more than one task making progress
  - single processor/core, scheduler providing concurrency
- **Parallelism**: implies system can perform more than one task simultaneously
  - **Data Parallelism**: distributes subsets of same data across multiple cores, same operation on each
  - **Task Parallelism**: distributing threads across cores, each thread performing unique operation

- **Amdahl's Law**
  - identifies performance gains from adding additional cores to an application that has both serial and parallel components
  - Given S – serial portion, N – processing cores: $speedup \leq \frac{1}{S+\frac{(1-S)}{N}}$
    - as N approaches infinity, speedup approaches 1/S
    - Ex: if application is 75% parallel / 25% serial, moving 1 to 2 cores results in speedup of 1.6 times
  - **serial portion** of an application has disproportionate effect on performance gained by adding additional cores

Types of Threads

- **User Threads**: management done by user-level threads library, such as:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- **Kernel Threads**: supported by the Kernel (in virtually all general-purpose OS)

Multithreading Models

- **Many-to-One**
  - many user-level threads mapped to single kernel thread
  - one thread blocking causes all to block
  - multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
  - few systems currently use this model
  - Ex: Solaris green threads, GNU portable threads
- **One-to-One**
  - each user-level thread maps to kernel thread
  - creating a user-level thread creates a kernel thread
  - more concurrency than many-to-one
  - number of threads per process sometimes restricted due to overhead
  - Ex: Windows, Linux
- **Many-to-Many**
  - allows many user level threads to be mapped to many kernel threads
  - allows the operating system to create a sufficient number of kernel threads
  - not very common
  - Ex: Windows with the ThreadFiber package, Java on Solaris
- **Two-Level Model**
  - similar to M:M, except that it allows a user thread to be bound to kernel thread

Thread Libraries

- provides programmer with API for creating and managing threads
- two primary ways of implementing
  - library entirely in user space
  - kernel-level library supported by the OS
- **Pthreads**
  - may be provided either as user-level or kernel-level
  - a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - specification, not implementation
  - API specifies behavior of the thread library, implementation is up to development of the library
  - common in UNIX operating systems (Linux & Mac OS X)
- **Java Threads**
  - managed by the JVM
  - typically implemented using the threads model provided by underlying OS
  - may be created by extending Thread class or implementing the Runnable interface
    - standard practice is to implement Runnable interface
- **OpenMP**
  - set of compiler directives and an API for C, C++, FORTRAN
  - provides support for parallel programming in shared-memory environments
  - identifies parallel regions – blocks of code that can run in parallel
    - #pragma omp parallel
    - create as many threads as there are cores
- **Intel Threading Building Blocks (TBB)**
  - template library for designing parallel C++ programs
- Windows Threads
  - **Windows API**: primary API for Windows applications
  - implements one-to-one mapping, kernel-level
  - each thread contains:
    - thread id
    - register set representing state of processor
    - separate user and kernel stacks for when thread runs in user mode or kernel mode
    - private data storage area used by run-time libraries and dynamic link libraries (DLLs)
  - register set, stacks, and private storage area are known as the **context** of the thread
  - primary data structures include:
    - **ETHREAD (executive thread block)**

        - includes pointer to process to which thread belongs and to KTHREAD, in kernel space
    - **KTHREAD (kernel thread block)**
        - scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
    - **TEB (thread environment block)**
        - thread id, user-mode stack, thread-local storage, in user space
- Linux Threads
    - refers to threads as **tasks**
    - creation is done through **clone()** system call
        - allows child task to share address space of parent task (process)
        - flags control behavior: CLONE_FS, CLONE_VM, CLONE_SIGHAND, CLONE_FILES
- fork() and exec()
    - fork means to duplicate thread
        - some UNIXes have two versions, one for only calling thread and another calling all threads
    - exec() usually works as normal – replace running process including all threads

Implicit Threading

- **Thread Pools**
    - create a number of threads in a pool where they await work
    - Advantages:
        - usually slightly faster to service a request with an existing thread than create a new thread
        - allows the number of threads in the application(s) to be bound to the size of the pool
        - separating task to be performed from mechanics of creating task allows different strategies for running task
            - Ex: Tasks could be scheduled to run periodically
    - supported in Windows API
- **Fork-Join Parallelism**
    - multiple threads (tasks) are **forked**, and then **joined**
    - supported by many libraries
- **Grand Central Dispatch**
    - Apple technology for macOS and iOS
    - extensions to C, C++ and Objective-C languages, API, and run-time library
    - allows identification of parallel sections
    - manages most of the details of threading

- block is in "^{ }" :
  - placed in dispatch queue
  - assigned to available thread in thread pool when removed from queue
- Types of Dispatch Queues
  - **Serial**: blocks removed in FIFO order, queue is per process (main queue)
  - **Concurrent**: also FIFO order but several may be removed at a time

Signal Handling

- used to process **signals**:
  - used in UNIX systems to notify a process that a particular event has occurred.
- signal is generated by a particular event, delivered to a process, then handled by one of the two handlers: default or user-defined
- every signal has **default handler** that kernel runs when handling signal
  - **user-defined** signal handler can override default
  - for single-threaded, signal delivered to process
  - for multi-threaded:
    - deliver signal to thread to which signal applies
    - deliver signal to every thread in process
    - deliver signal to certain threads in the process
    - assign a specific thread to receive all signals for the process

Thread Cancellation

- terminating thread before it has finished
- thread to be canceled is called a **target thread**
- two general approaches:
  - **Asynchronous Cancellation**: terminates target thread immediately
  - **Deferred Cancellation**: allows target thread to periodically check if should
- invoking it requests cancellation, but actual cancellation depends on thread state
  - if cancellation disabled, it remains pending until enabled
  - default type is **deferred**
    - cancellation only occurs when thread reaches cancellation point
    - then, cleanup handler is invoked
  - on Linux systems, thread cancellation is handled through signals

Thread-Local Storage (TLS)

- allows each thread to have its own copy of data
- useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- different from local variables, as TLS is **visible across function invocations**

    - similar to static data
    - meanwhile, local variables are visible only during single function invocation
- **unique** to each thread

Scheduler Activations

- both M:M and two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- typically use an intermediate data structure between user and kernel threads called **lightweight process (LWP)**
    - appears to be a virtual processor on which process can schedule user thread to run
    - each LWP attached to kernel thread
- provide **upcalls**
    - a communication mechanism from the kernel to the upcall handler in the thread library
    - allows an application to maintain the correct number kernel threads

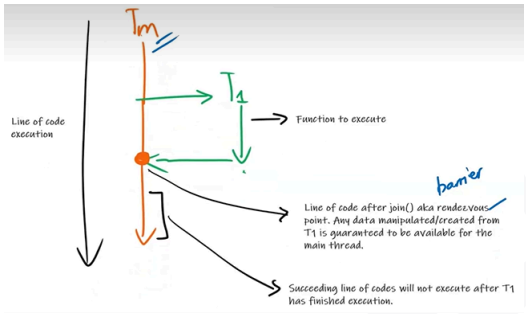
Threads and Concurrency Summary

- A **thread** represents a basic unit of CPU utilization, and threads belonging to the same process share many of the process resources, including code and data
- There are four primary benefits to multithreaded applications: responsiveness, resource sharing, economy, and scalability
- **Concurrency** exists when multiple threads are making progress
- **Parallelism** exists when multiple threads are making progress simultaneously
    - on a system with a single CPU, only concurrency is possible
    - requires a **multicore** system that provides multiple CPUs
- There are several challenges in designing multithreaded applications, including dividing and balancing the work, dividing the data between the different threads, and identifying any data dependencies
    - multithreaded programs are especially challenging to test and debug
- **Data parallelism** distributes subsets of the same data across different computing cores and performs the same operation on each core, while **task parallelism** distributes not data but tasks across multiple cores. Each task is running a unique operation
- User applications create **user-level threads**, which must ultimately be mapped to **kernel threads** to execute on a CPU
- The **many-to-one model** maps many user-level threads to one kernel thread, but other approaches include the one-to-one and many-to-many models.
- A **thread library** provides an API for creating and managing threads
    - Ex: Windows, Pthreads (POSIX), and Java threading (JVM)

- **Implicit threading** involves identifying tasks—not threads—and allowing languages or API frameworks to create and manage threads
    - Approaches: **thread pools**, **fork-join frameworks**, and **Grand Central Dispatch**
    - becoming an increasingly common technique for programmers to use in developing concurrent and parallel applications.
- Threads may be terminated using either asynchronous or deferred cancellation:
    - **Asynchronous cancellation** stops a thread immediately, even if it is in the middle of performing an update
    - **Deferred cancellation** informs a thread that it should terminate but allows the thread to terminate in an orderly fashion
    - In most circumstances, **deferred** cancellation is preferred to asynchronous termination

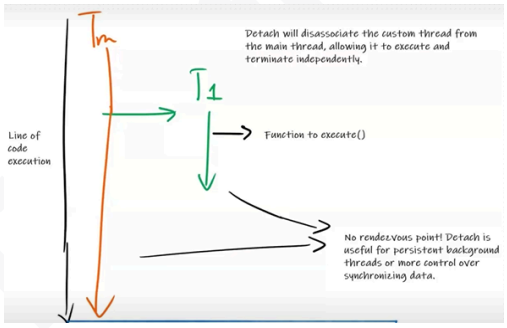
Implementation of Threads

- thread.join() In C++:



std::thread <name>(function);
<thread_name>.join(); // either .join() or .detach()

- thread.detach() in C++:



    - thread executes and terminates independently from the main thread
    - in Java/C#, threads are detached by default
- Default Thread Handling (in Windows):
    - has main thread and UI thread, even in the simplest programs
    - for those with a Window screen, such as games, there are 3 by default

  - **Main Thread**
    - instructions called from the main() function
  - **Console Thread**
    - handles prints
    - persistently runs until the main program is closed
  - **UI Thread**
  - Ex: “Hello world” program has 2 threads
- Customized Threads
  - 1 thread = 1 class (object-oriented)
  - start() to create a thread
  - sleep() to avoid premature halting (waiting for other threads to finish)
- Sequence
  - threads are scheduled for execution
  - **OS completely decides sequence** (non-deterministic)
    - may result in disorganized layout, as simultaneous std::couts are happening concurrently, resulting in a queue

CPU Scheduling

CPU Scheduling

- subset of **process scheduling**
  - deals with overall lifecycle and interactions of processes, and they may not always require the CPU (e.g. GPU rendering, I/O operation)
- deals with deciding which process to execute next on **central processing unit (CPU)**
  - logical unit for executing a process (not the physical CPU hardware)
- involves **selecting a process from the ready queue and allocating CPU time** to that process
  - **Ready Queue**: holds processes that should be executed in the CPU
  - all processes need the CPU at least once in their lifetime; so for a process to be scheduled on the GPU, it must first have a CPU instruction requesting the transfer of data and GPU-specific instructions into the GPU
- Types of Schedulers
  - **Short-Term Scheduler**
    - selects from among the processes in ready queue and allocates to the CPU
  - **Long-Term Scheduler**
    - decides which processes should go first to the short-term scheduler
    - not all OS have this
- Goal: **maximize CPU utilization and throughput** while minimizing waiting times and response times for processes (obtained with multiprogramming)

CPU-I/O Burst Cycle

- process execution by the CPU
- consists of a cycle of CPU execution and I/O wait (CPU burst followed by I/O burst)

Scheduling Decisions

- multiple processes compete for CPU time, and CPU scheduling determines which process to execute next in a **ready queue**
- scheduler allocates a CPU core to one of the processes in the queue when a process:
  1. switches from running to waiting state
  2. switches from running to ready state
  3. switches from waiting to ready state
  4. terminates
- for 1 and 4, there is no choice in terms of scheduling, for a new process (if one exists in the ready queue) must be selected for execution (**nonpreemptive**)
- for 2 and 3, there is a choice (**preemptive**)

Preemptive and Nonpreemptive Scheduling

- **Nonpreemptive**
  - once CPU has been allocated to a process, process keeps the CPU until it releases it either by terminating or switching to a waiting state
  - doesn't need special hardware (i.e. clock/signal generator)
- **Preemptive**
  - given processes, -1 burst to put in CPU, then put in queue, then run each
  - can result in race conditions when data are shared among several processes
    - Ex: two processes share data, and while one is updating data, it is preempted so that second can run, but when second tries to read data, it is in an inconsistent state
  - virtually all modern OS use an algorithm that is **preemptive**

Dispatcher

- gives control of the CPU to the process selected by CPU scheduler, and involves
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart the program
- **Dispatch latency**: time it takes for dispatcher to stop one process and start another run

Scheduling Criteria

- **CPU utilization**: keep CPU as busy as possible
- **Throughput**: # of processes that complete their execution per time unit
- **Turnaround time**: amount of time to execute a particular process
- **Waiting time**: amount of time a process has been waiting in the ready queue
- **Response time**: amount of time it takes from when a request was submitted until first response is produced, not output (for time-sharing environment)
- **Fairness**: share CPU among users in equitable manner

CPU Scheduling Problem

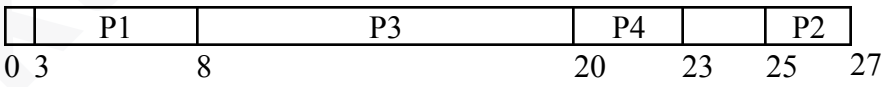
- Given a set of processes $P = \{P_1, P_2, \ldots, P_N\}$ where $P_i \in P$ contains (A,C,X/null) where A is arrival time, C is CPU time, and X is priority, and a CPU scheduling algorithm, find O which is a set of ordered P according to sequence of execution
- Optimization Criteria:
  - **Max CPU utilization**
  - **Max throughput**
  - **Min turnaround time**
  - **Min waiting time**
  - **Min response time**

CPU Scheduling Algorithms

- *Note: Autosolver Link – Process Scheduling Solver (boonsuen.com)*
- **First-Come, First-Served (FCFS) Scheduling**
  - FIFO, run until it's done
  - a **nonpreemptive** scheduling algorithm
  - in early systems, meant one program scheduled until done
  - Examples:
    1. Processes P_1, P_2, P_3 have burst times of 24, 3, and 3 respectively. Assume they have the same arrival time of 0
       - Suppose they arrive in order P_1, P_2, P_3, Gantt Chart for schedule:

       Waiting Time for P_1 = 0; P_2 = 24; P_3 = 27
       Average Waiting Time: (0+24+27) / 3 = 17
       Average Completion Time: (24+27+30) / 3 = 27
       - Suppose they arrive in order P_2, P_3, P_1, Gantt Chart for schedule:

       Waiting Time for P_1 = 6; P_2 = 0; P_3 = 3
       Average Waiting Time: (6+0+3) / 3 = 3
         - much better than the previous case due to **Convoy effect**
           - short process behind long process
    2. Processes P_1, P_2, P_3, P_4 have burst times of 5, 2, 12, 3 respectively. Find their waiting time and average waiting time assuming their arrival times are 3, 25, 4, 10 respectively.

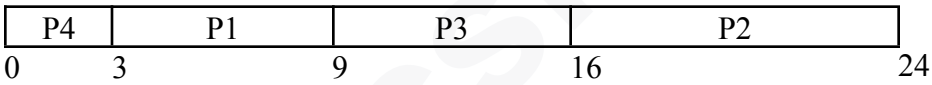


       Waiting Times:
       - P_1 = (actual arrival time - arrival time) = 3 - 3 = 0
       - P_2 = 25 - 25 = 0
       - P_3 = 8 - 4 = 4
       - P_4 = 20 - 10 = 10

       Average Waiting Time: (0+0+4+10) / 4 = 3.5
       Throughput$_{20}$ (# of processes that start and finish at N bursts): 2

       Note: it is possible for the scheduler to be idle (no process)

- **Shortest-Job-First (SJF) Scheduling**
  - associate with each process the length of its next CPU burst
  - use these lengths to schedule the process with the shortest time
  - a **nonpreemptive** scheduling algorithm
  - optimal as it gives **minimum average waiting time** for a given set of processing
  - for each CPU cycle:
    1. Update *R* if there's any pending process to be scheduled
    2. If applicable, sort *R* where first element satisfies $P_0 = \min(C)$
       - select minimum CPU time from the processes as $P_{candidate}$
       - a batch of processes can potentially arrive per CPU cycle
    3. If a current *P* is finished/null, select $P_{candidate}$, such that $P_{candidate} = \min(C)$
    4. Execute $P_{candidate}$. $P_{candidate}.C$--;
  - Examples:
    1. Processes P1, P2, P3, P4 have burst times of 6, 8, 7, 3 respectively. Assume their arrival times are all at 0.

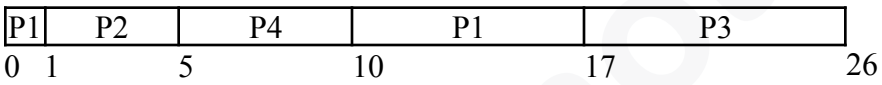


       Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

       For every process that arrives, put in ready queue and sort.
       If no running P, choose min(burst time) in queue and finish it.

  - Predicting length of next CPU burst
    - can only estimate the length, then pick process with shortest predicted next CPU burst
    - can be done by using length of previous CPU bursts, using exponential averaging: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
      - τ_{n+1} = predicted value for the next CPU burst
      - t_n = actual length of n^{th} CPU burst
      - α, where $0 \leq \alpha \leq 1$ (commonly set to ½)
- **Shortest-Remaining-Time-First (SRTF) Scheduling**
  - **preemptive SJF**
  - whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJF algorithm
  - allows for running processes to return to queue if ready queue has a process with a shorter burst
    - means waiting time for a process could increase (since waiting in queue)

- for each CPU cycle:
    1. Update *R* if there's any pending process to be scheduled
    2. If applicable, sort *R* where first element satisfies $P_0 = \min(C)$
        - select minimum CPU time from the processes as $P_{candidate}$
        - a batch of processes can potentially arrive per CPU cycle
    3. Select $P_{candidate}$, such that $P_{candidate} = \min(C)$; put current *P* running back to *R* if $P_{candidate}$ has a shorter remaining burst time
    4. Execute $P_{candidate}$. $P_{candidate}.C$--;
- Examples:
    1. Processes P_1, P_2, P_3, P_4 have burst times of 8, 4, 9, 5 respectively. Assume they have arrival times of 0, 1, 2, 3 respectively.

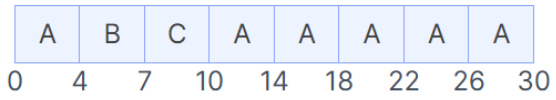


Waiting Times: P_1 =(10-1)= 9; P_2 =(1-1)= 0; P_3 =(17-2)= 15; P_4 =(5-3)= 2
Average Waiting Time: (9+0+15+2) / 4 = 26/4 = 6.5

P_1 had to queue again as P_2 had burst time of 4 when it arrived (P_1 had 7)
When P_3 and P_4 arrived, P_2 still had a shorter remaining time, so it stays
By end of P_2, shortest burst in the queue was P_4, then P_1, then P_3

- **Round-Robin (RR)**
    - time/quantum sharing (equal bursts)
    - each process gets a small unit of CPU time (time quantum *q*), usually 10-100 ms; after it elapses, process is **preempted** and added to end of ready queue
    - if there are *n* processes in ready queue and time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once
        - No process waits for more than **(*n*-1)*q*** time units
    - Timer interrupts every quantum to schedule next process
    - Performance:
        - *q* large → FIFO (FCFS); *q* small → RR
        - *q* be large with respect to context switch; otherwise, overhead too high
    - for each CPU cycle (*C* means CPU cycle):
        1. Update *R* if there's any pending process to be scheduled
        2. Select the first process in *R* (or randomly) to be the $P_{candidate}$
        3. Execute $P_{candidate}$. $P_{candidate}.C$++;
        4. If $P_{candidate}.C = T$, then perform #2. Put $P_{candidate}$ at the end of *R*; otherwise, perform #3

- Examples:
    1. Processes *A*, *B*, *C* have burst times of 24, 3, and 3 respectively. Assume they have the same arrival time of 0. $q = 4$

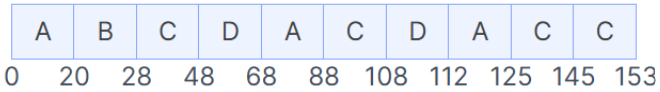


Waiting times:
- *A* = (0-0) + (10-4) = 6
- *B* = (4-0) = 4
- *C* = (7-0) = 7

Average waiting time: (6+4+7) / 3 =5.667
Turnaround time (total time it takes to start from arrival to finish):
- *A* = 30; *B* = 7; *C* = 10

No sorting. Typically higher turnaround than SJF, but better response.

2. Processes *A*, *B*, *C*, *D* have burst times of 53, 8, 68, 24 respectively. Assume all have arrival time of 0. $q = 20$

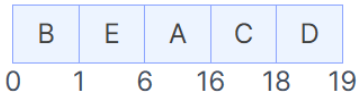


Waiting times:
- *A* = (68-20) + (112-88) = 48 + 24 = 72
- *B* = (20-0) = 20
- *C* = (28-0) + (88-48) + (125-108) = 85
- *D* = (48-0) + (108-68) = 88

Average waiting time = (72+20+85+88) / 4 = 66.25
Average completion time = (125+28+153+112) / 4 = 104.5

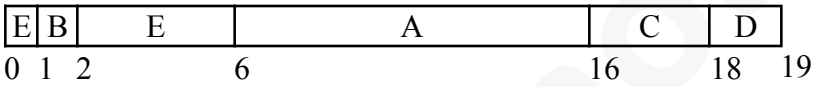
- **Priority Scheduling**
    - priority number (**integer**) is associated with each process
    - CPU is allocated to the process with the **highest priority** (**smallest integer**)
    - SJF is priority scheduling where priority is inverse of predicted next CPU burst time
    - can be **either preemptive or nonpreemptive**
    - Problem: **Starvation** (low priority processes may never execute)
    - Solution: **Aging** (as time progresses, increase the priority of process)

- Examples:
    1. Processes *A*, *B*, *C*, *D*, *E* have burst times of 10, 1, 2, 1, 5 respectively. Each has a priority of 3, 1, 4, 5, 2 respectively. Nonpreemptive priority.



Average waiting time = (1+6+16+18) / 5 = 8.2

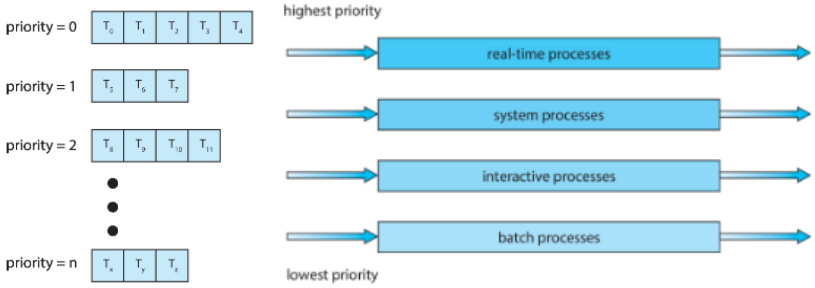
    2. Processes *A*, *B*, *C*, *D*, *E* have burst times of 10, 1, 2, 1, 5 respectively. Each has a priority of 3, 1, 4, 5, 2 respectively. Arrival time is all 0 except for *B*, whose arrival is at 1. Preemptive priority.



Average waiting time = (1+0+6+16+18) / 5 = 8.2

    3. Run the process with the highest priority. Processes with the same priority run round-robin. Given processes P_1, P_2, P_3, P_4, P_5 with burst times 4, 5, 8, 7, 3 respectively. Priority is 3, 2, 2, 1, 3 respectively. Arrival = 0. q = 2.

- **Multilevel Queue**
    - ready queue consists of multiple queues
    - scheduler defined by following parameters:
        - Number of queues
        - Scheduling algorithms for each queue
        - Method used to determine which queue a process will enter when that process needs service
        - Scheduling among the queues
    - with priority scheduling, separate queues for each priority (e.g. process type):



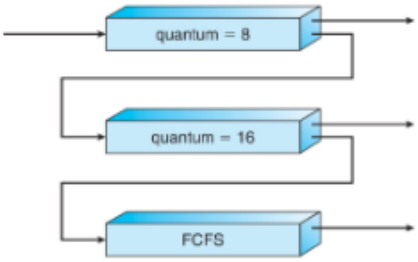
- as long as upper RQ is not empty, always select processes there
    - selection is based on whichever algorithm is being used
- Con: if there's always a higher priority process that arrives, starvation happens on lower priority processes
- Alternative: **Multi-Level Feedback Scheduling**
    - there's a promotion scheme for elevating certain processes, based on certain conditions
    - Ex: if process P_k stays too long in level 1, can be promoted to level 0
    - Example of Multilevel Feedback Queue

Three queues:

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

Scheduling

- A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
- At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

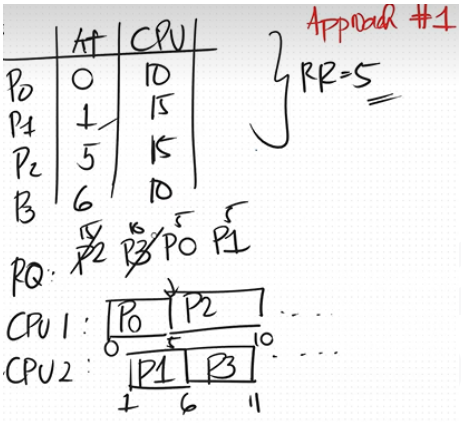


Thread Scheduling

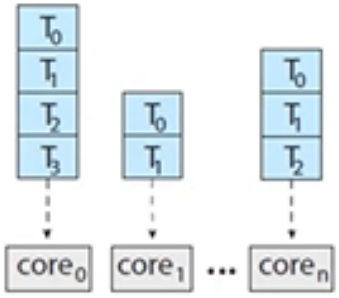
- distinction between user-level and kernel-level threads
- when threads supported, threads scheduled, not processes
- many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
    - known as **process-contention scope (PCS)** since scheduling competition is within the process
    - typically done via priority set by programmer
- kernel thread scheduled onto available CPU is **system-contention scope (SCS)**, wherein competition among all threads in system
- API allows specifying either PCS or SCS during thread creation:
    - PTHREAD_SCOPE_PROCESS
    - PTHREAD_SCOPRE_SYSTEM
- can be limited by OS
    - Linux and macOS only allow PTHREAD_SCOPE_SYSTEM

Multiple-Processor Scheduling

- may be any of the following architectures:
  - multicore CPUs
  - multithreaded cores
  - NUMA systems
  - heterogeneous multiprocessing
- **Symmetric Multiprocessing (SMP)**
  - each processor is self scheduling
  - scheduling strategies:
    - **Common Ready Queue**: one ready queue and 1 CPU = 1 slot



    - **Pre-Core Run Queues:** 1 CPU = 1 RQueue (a private queue of threads):



- **Asymmetric Multiprocessing (AMP)**
- **Multithreaded Multicore System**
  - chip-multithreading (CMT) assigns each core multiple hardware threads (Intel refers to this as **hyperthreading**)
  - on a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.
- **Load Balancing**
  - Two types of scheduling:
    - OS deciding which software thread to run on logical CPU
    - how each core decides which hardware thread to run on the physical core

- **Processor Affinity**
  - refers to when a thread has been running on one processor and the cache contents of that processor stores the memory accessed by that thread
  - **load balancing** may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of
  - **Soft Affinity**
    - OS attempts to keep a thread running on the same processor, but no guarantees.
  - **Hard Affinity**
    - allows a process to specify a set of processors it may run on.
- NUMA and CPU Scheduling
  - if OS is NUMA-aware, it assigns memory closest to CPU the thread is running on

Real-Time CPU Scheduling

- **Soft Real-Time Systems**: critical real-time tasks have highest priority, but no guarantee as to when tasks will be scheduled
- **Hard Real-Time Systems**: task must be serviced by its deadline
- **Event Latency**: amount of time that elapses when event occurs to when it is serviced
  - types that affect performance:
    - **Interrupt Latency**: time from arrival of interrupt to start of routine that services interrupt
    - **Dispatch Latency**: time for schedule to take current process of CPU and switch to another
      - Conflict Phase: preemption of any process running in kernel mode, release by low-priority process of resources needed by high-priority processes

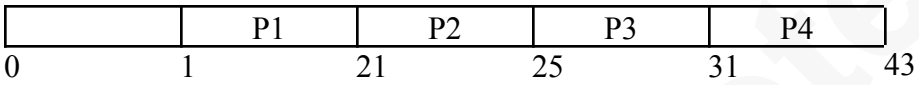
Other Types of Priority-based Scheduling

- **Rate Monotonic Scheduling**
  - priority is assigned based on inverse of its period
  - shorter periods = higher priority
- **Earliest Deadline First Scheduling (EDF)**
  - priorities are assigned according to deadlines
  - earlier deadline = higher priority
- **Proportional Share Scheduling**
  - T shares are allocated among all processes in the system
  - an application receives N shares where $N < T$
    - ensures each application will receive N / T of the total processor time

Get Gantt Chart, Average Waiting & Turnaround Time, and Throughput after 20 time bursts:

Process	Arrival Time	Burst Time	Priority
P_1	1	20	1
P_2	3	4	6
P_3	8	6	2
P_4	11	12	3

1. FCFS
   A. 1 CPU



Waiting Times:
P1 = (1-1) = 0
P2 = (21-3) = 18
P3 = (25-8) = 17
P4 = (31-11) = 20

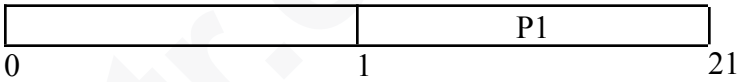
Average: (0+18+17+20)/4 = 13.75

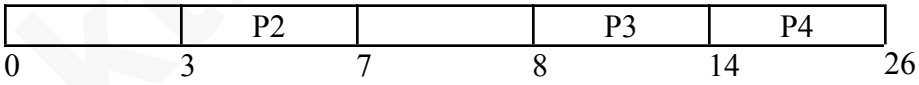
Turnaround Time:
P1 = (21-1) = 20
P2 = (25-3) = 22
P3 = (31-8) = 23
P4 = (43-11) = 32

Average: (20+22+23+32)/4 = 24.25

$Throughput_{20} = 0$

   B. 2 CPU





Waiting Times:
P1 = (1-1) = 0
P2 = (3-3) = 0
P3 = (8-8) = 0
P4 = (14-11) = 3

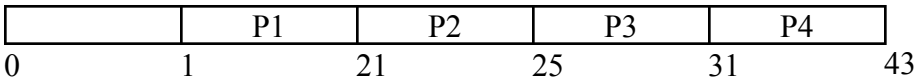
Average: 3/4 = 0.75

Turnaround Time:
P1 = (21-1) = 20
P2 = (7-3) = 4
P3 = (14-8) = 6
P4 = (26-11) = 15

Average: (20+4+6+15) = 45/4 = 11.25

$Throughput_{20} = 2$

2. SJF
   A. 1 CPU



Waiting Times:
P1 = (1-1) = 0
P2 = (21-3) = 18
P3 = (25-8) = 17
P4 = (31-11) = 20

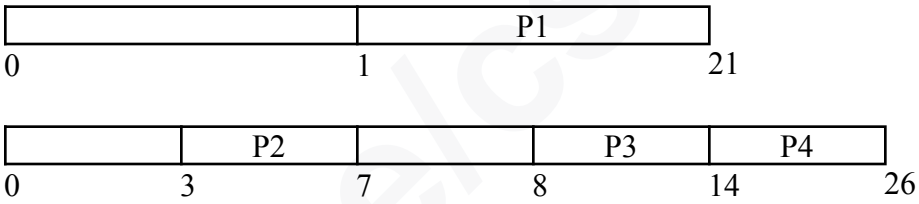
Average: (0+18+17+20)/4 = 13.75

Turnaround Time:
P1 = (21-1) = 20
P2 = (25-3) = 22
P3 = (31-8) = 23
P4 = (43-11) = 22

Average: (20+22+23+22)/4 = 21.75

$Throughput_{20} = 0$

   B. 2 CPU



Waiting Times:
P1 = (1-1) = 0
P2 = (3-3) = 0
P3 = (8-8) = 0
P4 = (14-11) = 3

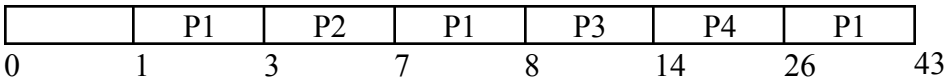
Average: 3/4 = 0.75

Turnaround Time:
P1 = (21-1) = 20
P2 = (7-3) = 4
P3 = (14-8) = 6
P4 = (26-11) = 15

Average: (20+4+6+15) = 45/4 = 11.25

$Throughput_{20} = 2$

3. SRTF
   A. 1 CPU



Waiting Times:
P1 = (1-1)+(7-3)+(26-8)= 22
P2 = (3-3) = 0
P3 = (8-8) = 0
P4 = (14-11) = 3

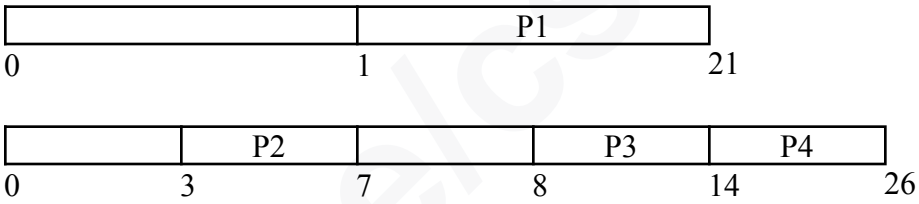
Average: (22+3)/4 = 6.25

Turnaround Time:
P1 = (43-1) = 42
P2 = (7-3) = 4
P3 = (14-8) = 6
P4 = (26-11) 15

Average: (42+4+6+15) = 67/4 = 16.75

$Throughput_{20} = 2$

   B. 2 CPU



Waiting Times:
P1 = (1-1) = 0
P2 = (3-3) = 0
P3 = (8-8) = 0
P4 = (14-11) = 3

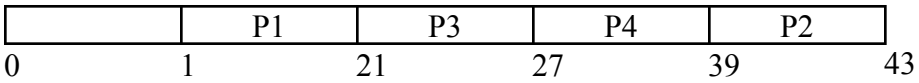
Average: 3/4 = 0.75

Turnaround Time:
P1 = (21-1) = 20
P2 = (7-3) = 4
P3 = (14-8) = 6
P4 = (26-11) = 15

Average: (20+4+6+15) = 45/4 = 11.25

$Throughput_{20} = 2$

4. Non-preemptive Priority Scheduling
   A. 1 CPU



Waiting Times:
P1 = (1-1) = 0
P2 = (39-3) = 36
P3 = (21-8) = 13
P4 = (27-11) = 16

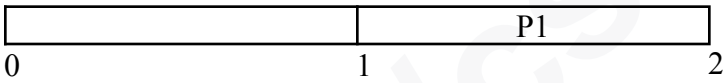
Average: (36+13+16)/4 = 16.25

Turnaround Time:
P1 = (21-1) = 20
P2 = (43-3) = 40
P3 = (27-8) = 19
P4 = (39-11) = 28

Average: (20+40+19+28)/4 = 26.75

Throughput$_{20}$ = 0

   B. 2 CPU



Waiting Times:
P1 = (1-1) = 0
P2 = (3-3) = 0
P3 = (8-8) = 0
P4 = (14-11) = 3

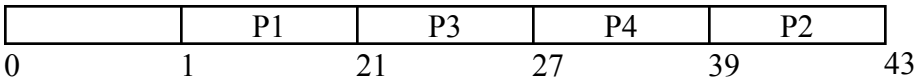
Average: 3/4 = 0.75

Turnaround Time:
P1 = (21-1) = 20
P2 = (7-3) = 4
P3 = (14-8) = 6
P4 = (24-11) = 13

Average: (20+4+6+13) = 43/4 = 10.75

Throughput$_{20}$ = 2

5. Preemptive Priority Scheduling
   A. 1 CPU



Waiting Times:
P1 = (1-1) = 0
P2 = (39-3) = 36
P3 = (21-8) = 13
P4 = (27-11) = 16

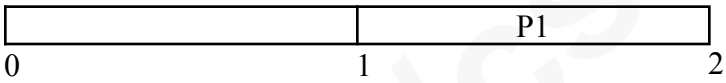
Average: (36+13+16)/4 = 16.25

Turnaround Time:
P1 = (21-1) = 20
P2 = (43-3) = 40
P3 = (27-8) = 19
P4 = (39-11) = 28

Average: (20+40+19+28)/4 = 26.75

$Throughput_{20} = 0$

   B. 2 CPU



Waiting Times:
P1 = (1-1) = 0
P2 = (3-3) = 0
P3 = (8-8) = 0
P4 = (14-11) = 3

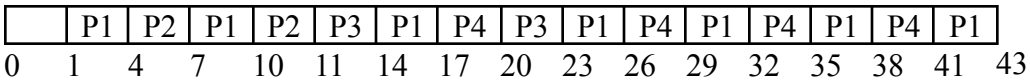
Average: 3/4 = 0.75

Turnaround Time:
P1 = (21-1) = 20
P2 = (7-3) = 4
P3 = (14-8) = 6
P4 = (26-11) = 15

Average: (20+4+6+15) = 45/4 = 11.25

$Throughput_{20} = 2$

6. Round Robin (q = 3)
   A. 1 CPU



Waiting Times:
P1 = (1-1)+(7-4)+(14-10)+(23-17)+
(29-26)+(35-32)+(41-38) = 22
P2 = (4-3) + (10-7) = 4
P3 = (11-8) + (20-14) = 9
P4 = (17-11)+(26-20)+(32-29)+(38-35)
= 18

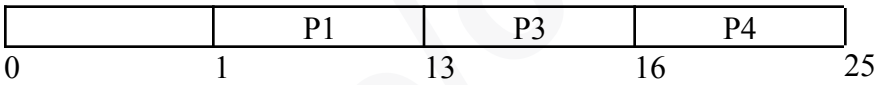
Average: (22+4+9+18) = 53/4 = 13.25

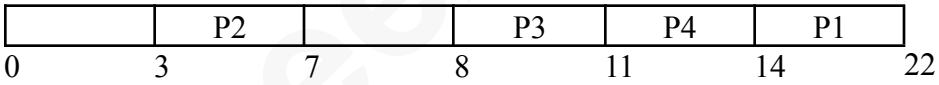
Turnaround Time:
P1 = (43-1) = 42
P2 = (11-3) = 8
P3 = (23-8) = 15
P4 = (41-11) = 30

Average: (42+8+15+30) = 95/4 = 23.75

$Throughput_{20} = 1$

   B. 2 CPU





Waiting Times:
P1 = (1-1) + (14-13) = 1
P2 = (3-3) = 0
P3 = (8-8) + (13-11) = 2
P4 = (11-11) + (16-14) = 2

Average: (1+2+2)/4 = 1.25

Turnaround Time:
P1 = (22-1) = 21
P2 = (7-3) = 4
P3 = (16-8) = 8
P4 = (25-11) = 14

Average: (21+4+8+14)/4 = 47/4 = 11.75

$Throughput_{20} = 2$

Memory Management

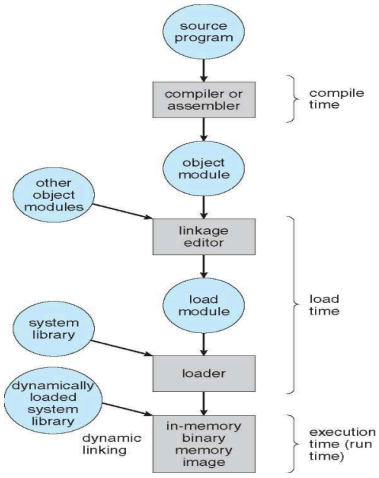
Main Memory

- Program must be brought (from a disk) into memory and placed within a process to run
- **Main memory** and **registers** are the only storage CPU can access directly
  - Register access is done in **one** CPU clock (or less)
  - Main memory can take **many** cycles, causing a stall
  - **Cache** sits between main memory and CPU registers
- Memory unit only sees a stream of:
  - **addresses + read** requests
  - **address + data and write** requests
- **protection** of memory required (done by OS) to ensure correct operation
  - need to ensure that a process can access only those addresses in its address space
  - can be done by using a pair of **base** and **limit** registers to define the logical address space of a process
  - Hardware Address Protection
    - **CPU** must check every memory access generated in user mode to be sure it is between base and limit of that user
    - instructions to loading the base and limit registers are privileged

Address Binding

- programs on disk are ready to be brought into memory to execute from an input queue
  - without support, it must be loaded into address 0000
  - inconvenient to have first use process as physical address is always at 0000
- Binding of instructions and data to memory, can happen at three different stages:
  1. **Compile time**: if memory location is known a priori, absolute code can be generate; must recompile code if starting location changes
  2. **Load time**: must generate relocatable code if memory location is not known at compile time (placed in a stack)
     - usually refers to loading libraries
     - Ex: LEA in NASM to access printf from the C library
  3. **Execution time**: binding delayed until run time if the process can be moved during its execution from one memory segment to another
     - need hardware support for address maps (e.g. base and limit registers)

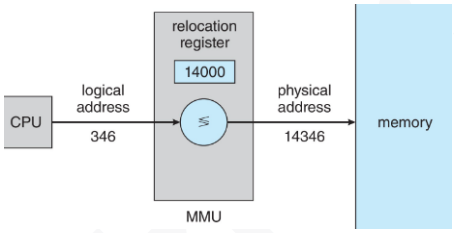
Multistep Processing of a User Program



Logical vs Physical Address Space

- In memory management, logical address space is bound to a separate physical address space:
  - **Logical address**: generated by the CPU; also referred to as virtual address
  - **Physical address**: address seen by the memory unit
- same in compile-time and load-time address-binding schemes, but they differ in **execution-time** address-binding scheme
- **Logical Address Space**: set of all logical addresses generated by a program
- **Physical Address Space**: set of all physical addresses generated by a program

Memory Management Unit



- hardware device that at run time maps virtual to physical address
- consider a simple scheme (base-register), it is now called a **relocation register**:
  - value is added to every address generated by a user process at the time it is sent to memory
  - user program deals with logical addresses, and it never sees the real physical addresses
- execution-time binding occurs when reference is made to location in memory
- logical addresses are bound to physical addresses

Dynamic Loading

- entire program does not need to be in memory to execute
- routine is not loaded until called
    - for better memory-space utilization, unused routine is never loaded
- all routines kept on disk in relocatable disk format
    - useful when need large amounts of code to handle infrequently occurring cases
- no special support from OS is required, as it is implemented through program design
    - OS can help by providing libraries to implement dynamic loading

Linking

- **Static Linking**: system libraries and program code combined by the loader into the binary program image
- **Dynamic Linking**: linking postponed until execution time
    - the one typically used in memory management
    - small piece of code, **stub**, locates appropriate memory-resident library routine:
        - stub replaces itself with address of routine, and executes it
        - OS checks if routine is in processes' memory address (if not, add to space)
    - particularly useful for libraries

Contiguous Allocation

- main memory must support both OS and user processes, with this being on early method
- usually divides memory into **two partitions**:
    - Resident OS, usually held in low memory with interrupt vector
    - User processes held in high memory (each contained in single contiguous mem)
- relocation registers are used to protect user processes form each other, and from changing OS code and data:
    - base register: value of smallest physical address
    - limit register: range of logical address (each must be less than limit register)
    - MMU maps logical address dynamically, and can then allow actions such as kernel code being transient and kernel changing size

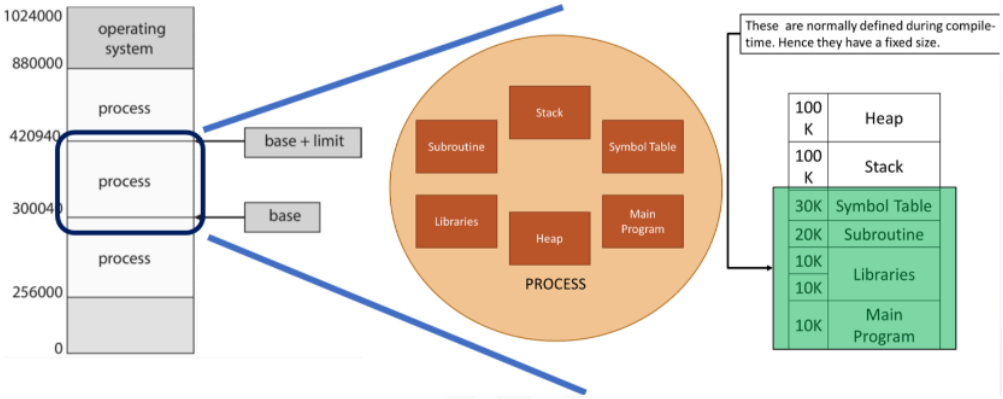
Variable Partition

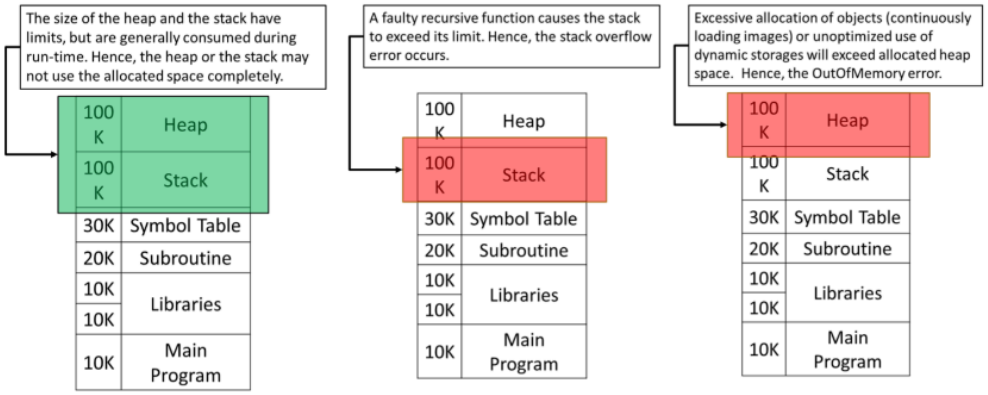
- multiple-partition allocation:
    - degree of multiprogramming limited by number of partitions
    - variable-partition sizes for efficiency (sized to given process' needs)
- **Hole**: block of available memory, scattered with various sizes throughout memory
    - when process arrives, it is allocated memory from hole large enough to fit
    - process exiting frees its partition, adjacent free partitions combined
    - OS maintains info about allocated partitions and free partitions (hole)

Implementing Memory Management

- most OS pre-allocate memory for each process, depending on its relative size and computational demand
  - can be written/freed when executing instructions → **malloc(), dealloc(), pointers**
  - each **process** have a separate memory space
  - start address and end address present in process info
- OS also has a share of its memory space
- Processes cannot execute without their resources loaded into memory
  - similar to how the OS/Kernel has a fixed amount of memory allocated (since it is also a process)

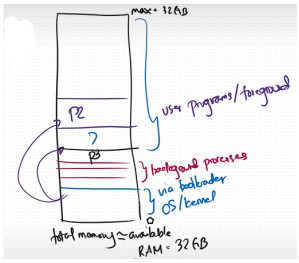
Diagram of the allocation:





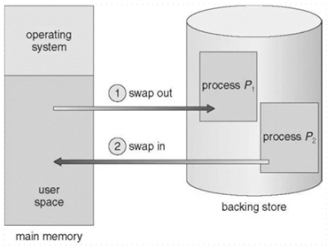
- allocation typically done by the OS

Flat Memory Model



- Assume at the very start of the OS program, memory are divided into *N* partitions, with predefined sizes
    - depending on the implementation
- Before process executes in the CPU, **OS allocates a hole big enough** for the process
    - Processes that require larger memory should need more holes
    - If the memory is not divisible, it still occupies the additional hold (although with an **internal fragmentation**, which is the extra space not used in the block)
- If a process arrives, but memory is full:
    - Possible implementations:
        - Wait until a process finishes execution, to free up memory space (**non-preemptive**)
            - Tradeoff: **lower CPU throughput**
        - Perform **swapping**, wherein a process in the waiting queue will have to release its memory and transfer it back to a backing store
            - Consider that OS create the illusion that there is infinite memory; the OS would typically follow the **scheduler algorithm** (i.e. Round Robin), and then from the main memory, save it to a **backing store** (SSD/HDD) with the progress logged in it
            - Once the process is relinquished, the new process is given allocation (depending on the implementation, it may look at the larger process instead if the new process requires a lot of memory)
            - typical behavior in current OSs
            - Tradeoff: **additional CPU cost**
- When a process finishes execution, it can immediately **free memory** for a future process
    - Bigger holes demanded by processes may merge, upon finish (depends on OS)
    - Partitions that are not continuous are called **external fragmentation** (often happens when a process to be allocated cannot since partition is not continuous)
    - OS may choose to do the following
        - Revert back to the original partitioning scheme
        - Maintain new partitions as is
        - Continuous partitions may be merged to avoid internal fragmentation

Possible Implementation in the OS Emulator



- Continuously allocate processes in the user space
- If there are no free user space, select a process that could be put in the backing store
    - To put in the backing store, save necessary information about the process, such as the pid, program counter, certain allocations/variable declarations performed
- Free up the block of the selected process
- Put the new process in the recently freed up block
- Factors to consider:
    - Minimize fragmentation
    - Maximize CPU utilization
    - Avoid swapping processes actively executing in CPU
        - it might get put in the backing store, and then loaded again (context-switching)

Fit Approaches

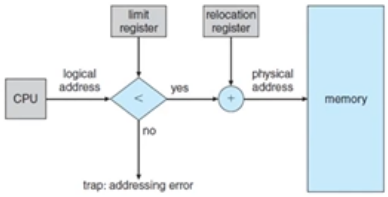
- aims to reduce fragmentation (cannot be fully eliminated)
- **First Fit**
    - OS allocates the first hole large enough to accommodate the process's size
    - Advantages: simple and easy to implement, quick allocation of memory
    - Disadvantages: may result in fragmentation, both internal and external, over time
    - Example:
        - With Round-Robin, P_0 is allocated 32KB, and P_1 with 1024KB, filling up the entire main memory.
        - When P_2 arrives and requires 512KB, P_1 is put in backing store to accommodate P_2 and allow it to run
            - This lasts for a time quantum, in which if P_1 would be put back into execution after since the scheduling is Round-Robin
- **Best Fit**
    - allocates smallest available block of memory that is large enough to fit the process
    - Advantages: minimizes wasted memory by selecting closest match in size, reducing external fragmentation
    - Disadvantages: may lead to fragmentation and may require searching through entire list of available blocks (linear)

    - Example:
        - There are four processes with both 22KB and 4KB blocks open, and the new process (assuming it can fit in either) would go to the 4KB block
- **Worst Fit**
    - allocates largest available block of memory, aiming to leave largest holes for future allocations
    - Advantages: can lead to larger available blocks for subsequent allocations
    - Disadvantages: may result in more wasted memory due to fragmentation, and can be time-consuming since it also searches
- **Next Fit**
    - similar to first-fit, but it starts searching for available memory from the last allocation point, and continues until a suitable block is found
    - Advantages: reduces fragmentation compared to first fit and simplest it
    - Disadvantages: still susceptible to fragmentation
        - Ex: a process finishes and frees up space, but it is before the last point
- **Quick Fit**
    - memory is divided into predefined block sizes, and each size has its own list of available blocks
    - allocation involves finding the appropriate size class and selecting the first available block
    - Advantages: quick allocation, especially for commonly used block spaces
    - Disadvantages: may lead to fragmentation within size classes
    - Ex: A certain block is dedicated to high priority
- in speed and storage utilization, first-fit and best-fit are better than worst-fit

Fragmentation

- **External Fragmentation**: total memory space exists to satisfy a request, but not contiguous
    - can be reduced by **compaction**
        - shuffle memory contents to place all free memory together in one large block
        - possible only if relocation is dynamic, and is done at execution time
- **Internal Fragmentation**: allocated memory may be slightly larger than request memory
    - size difference is memory internal to a partition, but not being used
    - first fit analysis reveals that given N blocks allocated, 0.5 N blocks are lost to fragmentation
        - ⅓ may be unusable → 50% rule
- Consideration: also a problem for the backing store

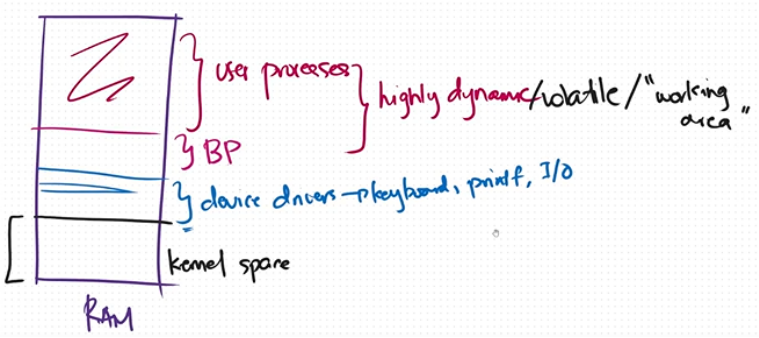
Implementing the Memory Management Unit



- typically tied in to the hardware architecture, wherein the CPU tries to add an offset to the location of the process memory, which is called a **logical address space**
    - When debugging, you would see a x-bit memory address, which would be generally the same for security purposes (to avoid malicious access to resources)
    - The address is a fixed minimum and maximum for P_0 to P_n
- once there is a logical address space, a translator will allocate the process to the **physical address space**
    - if there is a violation or overflow on the heap that may take space of another process, it is thrown to a trap or **addressing error** (i.e. segfault)
- Why?
    - In a kernel/OS, processes have a relatively large memory footprint, with some requiring dynamic memory allocation (e.g. variable declaration command)
    - Without one, memory is being blindly used as it is relying on language's built-in memory manager (i.e. current implementation of the OS emulator)
        - if careless, the OS could suddenly crash
    - For a kernel not having a memory manager would mean relying on hardware handling of memory, which usually only offers basic memory management and error handling
        - could also lead to crashing, as the OS is unable to facilitate resources
- Considerations:
    - a clever way to track memory for debugging, profiling, and warning of potential memory violations (e.g. Task Manager)
    - provide friendlier/meaningful error messages

Placement New

- operator that is an expansion of "new" keyword in C++; by extension, an expansion of pointers and addresses
    - Normal new allocates memory and constructs an object in allocated memory
    - Placement new **passes a preallocated memory** and **construct an object** in the passed memory (separating them)
- also **returns a reusable address space** (which can be used for succeeding allocations)
- allows defining of fixed memory spaces
    - can be helpful for debugging as you can organize processes like this:



- Example:

```
// Allocate block of memory to store MAX_SIZE * sizeof(uint8_t) bytes
uint8_t* data = new uint8_t[MAX_SIZE * sizeof(uint8_t)];

// Pointer to track next available position
uint8_t* top = data; // returns the minimum address

// Push 10 values onto stack using placement new
for (int i = 0; i < 10; ++i) {
        new (top) uint8_t(200); // it will basically increment with a dummy value
        top++;
}

// Access and print elements (stack order - LIFO)
while (top != data) {
        top++;
        std::cout << “Value: “ << static_cast<int>(*top) << “, Address: “ <<
        reinterpret_cast<void*>(top) << std::endl;
}

// Deallocate memory block
delete[] data;
```

```
void *apic_address = reinterpret_cast<void *>(<address>); // Specify “actual” address
APIC *apic = new (apic_address) APIC;

// However, it is unsafe; recommended to allocate some dummy value instead
// Another good practice is to pre-allocate memory
```

- *Note: Not necessary for OS emulator, just one of the possible ways. Alternatives include using data structures like std::vector*

Sample Partition Problem

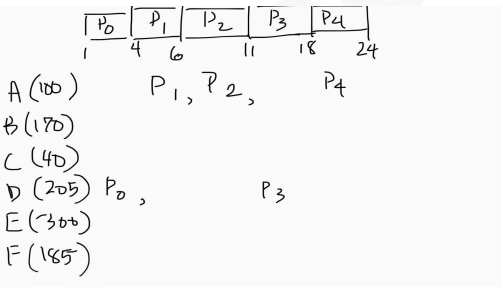
1. Given six (6) memory partitions of A (100 MB), B (170 MB), C (40 MB), D (205 MB), E (300 MB), and F (185 MB) (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes P0 (200 MB), P1 (15 MB), P2 (75 MB), P3 (175 MB), and P4 (80 MB) (in order)? Write the letter representing the partition size where each process will be placed. Indicate if it will be placed in a new one by writing the new partition size and letter of the partition it came from. If a process would not be placed in any of the partitions, just indicate None.

Process/ Algorithm	P0 (200 MB)	P1 (15 MB)	P2 (75 MB)	P3 (175 MB)	P4 (80 MB)
First-fit	D	A	B	E	F
Best-fit	D	C	A	F	B
Worst-fit	E	D	F	None	B

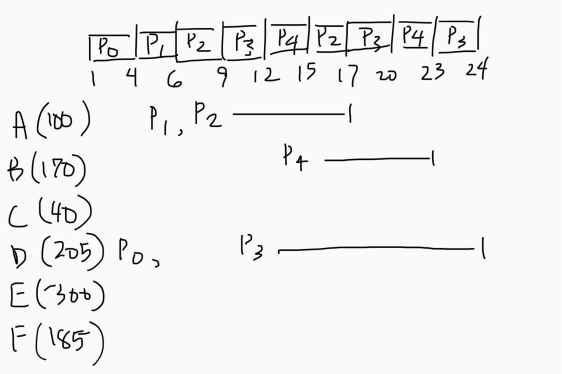
2. Same given as #1 but with the following Arrival Time and Burst Time:

	Arrival Time	CPU Bursts
P0	1	3
P1	3	2
P2	5	5
P3	6	7
P4	7	6

A. FCFS, First-Fit



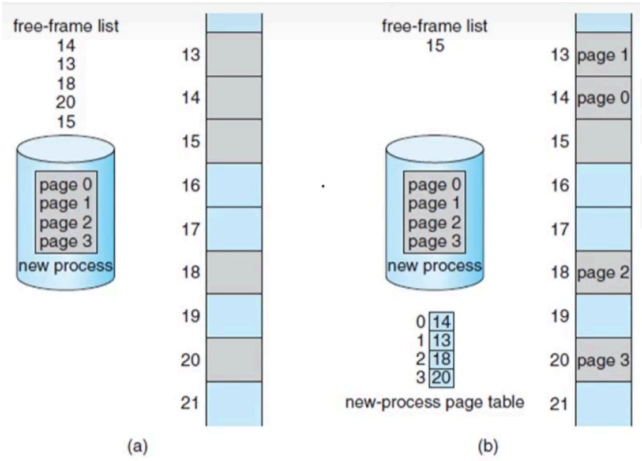
B. RR (q = 3), First-Fit



Paging

Paging

- used by most OS, more versatile than the flat memory model
- divide processes into chunks (**pages**), and physical memory space into **frames**
- allows **non-contiguous** allocation of memory for processes to **reduce fragmentation**, although internal fragmentation may still occur
- Recall that a process is stored in memory and divided into subspaces, and they are referred to as pages:
    - The OS maintains a **list of free frames** (typically sequential and defined at start)
    - All processes that arrive have a set number of pages
    - The **OS selects free frame lists to assign the pages and creates the page table**
    - Processes that finishes execution returns the free frames
- Example:



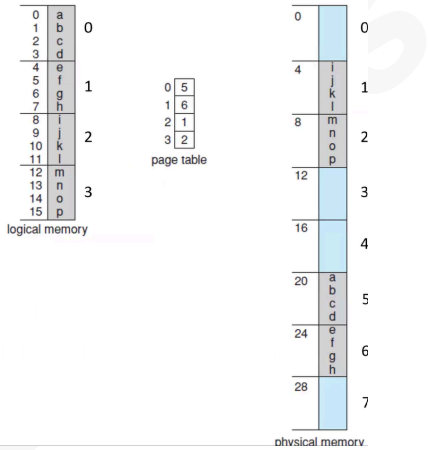
    - A new process that arrives that requires 4 pages
    - A free-frame list is managed by the OS, and take the first four listed to assign it to the pages of the newly arrived process (14, 13, 18, 20)
    - A page table is then made containing the allocating pages
        - can be implemented as a hash table (int-int), depending on the information being stored
    - This occurs for every newly arrived process

What if there are no free frames?

- **Backing store operation**
    - find a process that it will store in the backing store, to be loaded later (an I/O operation akin to saving a text file)
- or, put the new process **back to the ready queue**

Paging Mechanism

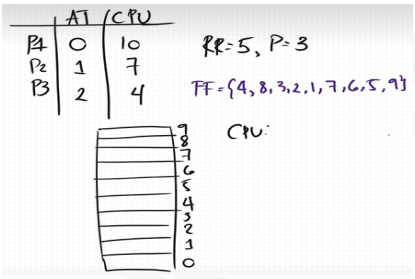
- defining page size and frame size is defined by the hardware, but is **always a power of 2**
  - makes translation of logical address to a page number and offset much simpler
  - if size of logical address space is 2^m, and a page size is 2^n bytes, then the high-order ***m-n*** bits of a logical address designate the **page number**, and the ***n*** low-order bits designate the **page offset**
- Example:
  Given n = 2 and m = 4 and 32 bytes of physical memory. Hence, page size is 4, logical memory size is 16.



1. What is the physical memory of the string “m”?
   Address 8
2. What is the range of the physical memory address of page index 2?
   Address 4-8
3. How much free memory space does the OS have?
   16 bytes

Paging Problems

1. Given the following specs and processes:

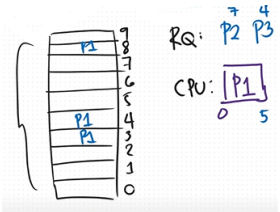


Follow the CPU scheduling, but with an **added condition to check if the memory/free frames are enough** for incoming process

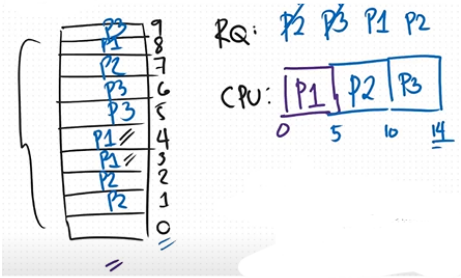
- If enough, put it to memory, then to CPU

Solution (given 1-core):

- Do typical scheduling



- Reaching the first quantum slice, although swapping will occur with P_2, P_1 will remain to have an **allocation in the memory until it is finished**
  - *Note: This scheme will likely fill up memory quickly if multiple processes are not finished, resulting in the potential need for backing store operation*
- Once the process finished (like P_3 at burst 14), remove the pages in the allocation, meaning frames 6, 5, 9 (in this case) are back as free frames in the list (in a queue)



Possible Questions:

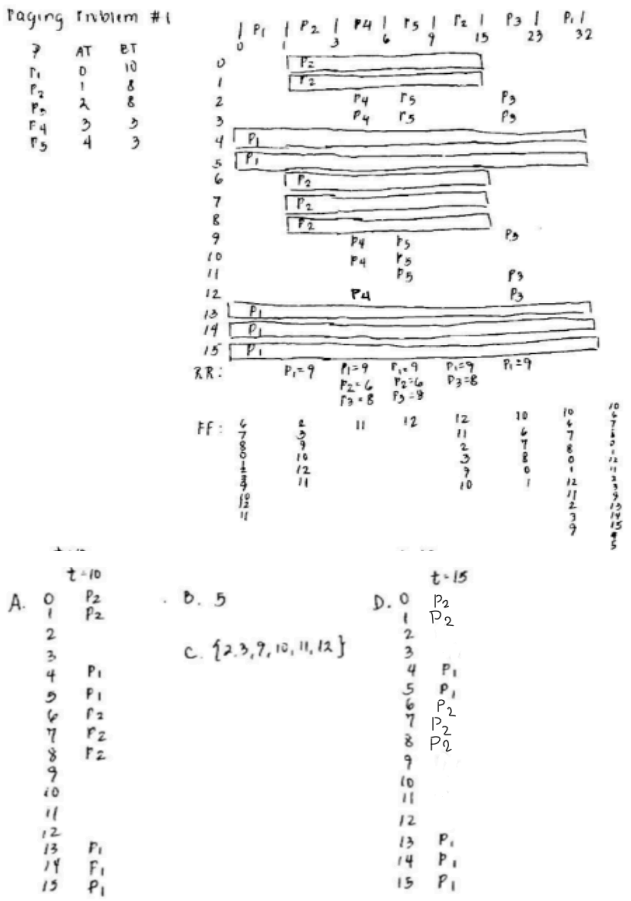
A. Draw the physical memory map by time slice 10. (*Note: Assume that these questions look at it before memory is freed*)
B. What will be the last value of the free-frame-list, after all processes have finished executing?
C. What will be the last value of the free-frame list, after time slice = *N*?
D. Include all the free frames by time slice = *N*

2. Consider processes wherein each needs 5 pages to execute. The OS has 15 frames available and the free-frame list is treated as a queue and has the following values: 13, 14, 15, 4, 5, 6, 7, 8, 0, 1, 2, 3, 9, 10, 12, 11. If the free-frame list is empty, the process will need to wait. Consider the process table and using a CPU scheduling scheme of preemptive shortest-job first.

Process	Arrival Time	CPU Burst
P1	0	10
P2	1	8
P3	2	8
P4	3	3
P5	4	3

A. Draw the physical memory map by time slice 10
B. What will be the last value of the free-frame list after all processes have finished executing?
C. Include all free frames by time slice 10
D. Draw the physical memory map by time slice 15

(Solution below)



3. Consider processes wherein each need 5 pages to execute. The OS has 10 frames available and the free-frame list is treated as a queue and has the following values: 4, 5, 6, 7, 0, 1, 2, 3, 8, 9. The OS has swapping enabled and performs swappin of the oldest process. Consider the process table and using a CPU scheduling scheme of preemptive shortest job first, answer the following questions.

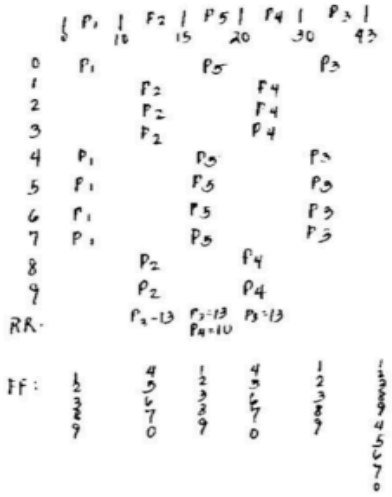
Process	Arrival Time	CPU Burst
P1	0	10
P2	5	5
P3	10	13
P4	12	10
P5	14	5

   A. What will be the last value of the free-frame list after all processes have finished executing?
   B. Indicate the allocated frames for P1.
   C. Indicate the allocated frames for P2.
   D. Indicate the allocated frames for P3.
   E. Indicate the allocated frames for P4.
   F. Indicate the allocated frames for P5.

(Solution below)

Paging Problem # 2

P	AT	BT
P_1	0	10
P_2	5	5
P_3	10	13
P_4	12	10
P_5	14	5



A. 0
B. {0, 4, 5, 6, 7}
C. {1, 2, 3, 8, 9}
D. {0, 4, 5, 6, 7}
E. {1, 2, 3, 8, 9}
F. {0, 4, 5, 6, 7}

Backing Store Operation

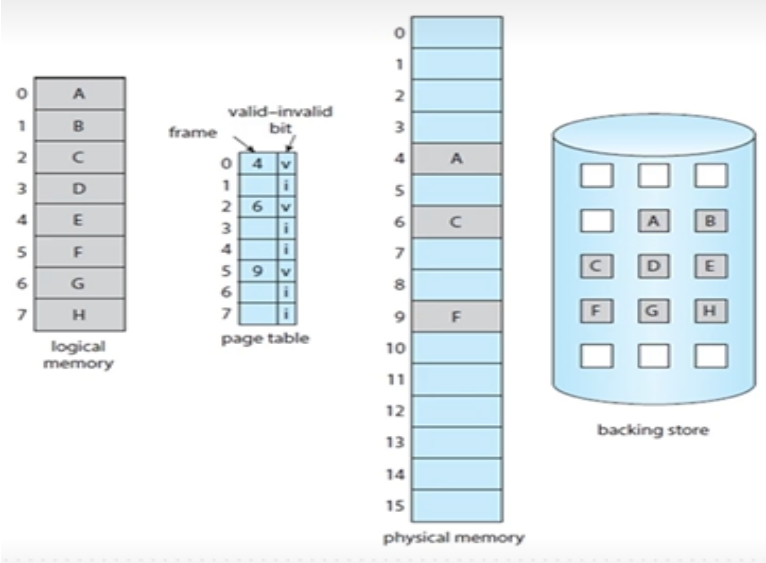
- for initial implementation, one could simply **select a random process** that resides in memory, resulting in the removal of its associated pages
- the process will then be put in a backing store, while the new process will be allocated memory that was previously from the pages of the stored process
- if this executes again on the CPU (such as in RR), then it may need to be pulled back from the backing store, and another process could be placed to the backing store (preventing processes remaining stuck and unfinished)
- Possible Implementation: Text files saved in a certain directory (relative to emulator path)
    - done to free main memory usage
    - must contain necessary process info such as the id, name, command counter, number of pages, and associated memory sizes, as dictated by its last command execution
        - *Note: In the emulator, since process memory and page size is fixed, simply store these values*

Demand Paging

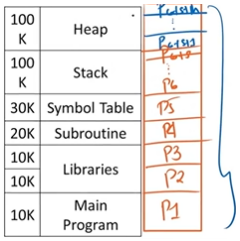
Demand Paging

- a more flexible approach to reduce external fragmentation
- addresses the limitation of paging of potentially inefficient memory usage:
    - Consider two programs with equal number of pages
        - To compensate for processes that require large memory, typically allocate or determine the number of pages and page size according to max known memory
        - This implies that lightweight processes will have unutilized number of pages
- load pages only as they are needed (minimum number of pages possible)

How it works



- The page table is extended to indicate a **valid/invalid bit**; if page index is marked as valid, it means it is in main memory, otherwise, it resides in the backing store
- At start of execution, active process will have **all its pages** in the backing store and marked as invalid
- At any point of execution, an active process can have **some of its pages** put to main memory, and some put back into a backing store if it's not needed
    - Another active process can then load its needed pages into main memory



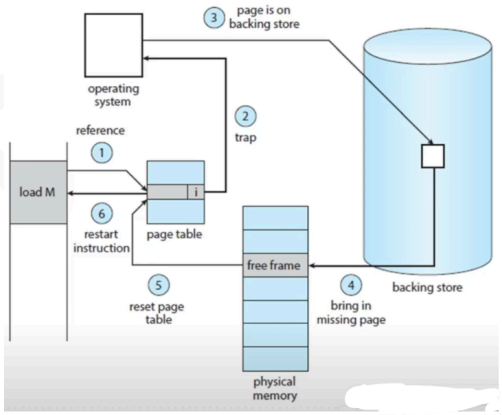
- Required pages per process can be fine-grained, where subcomponents may require multiple pages
- Purpose: Allow processes with smaller memory footprint to run with other processes that need larger memory
- The compiler and/or OS typically decides for the page allocation of each process

Structure of the Page Table

- memory structures for paging can get huge using straight-forward methods
- consider a 32-bit address space as on modern computers:
    - if page size of 4 KB (2^{12}), page table would have 1 million entries ($2^{32}/2^{12}$); and if each entry is 4 bytes, each process 4 MB of physical address space for the page table alone
        - don't want to allocate that contiguously in main memory
            - Solution: divide the page table into smaller units
                - Ex: hierarchical paging, hashed page tables, inverted page tables

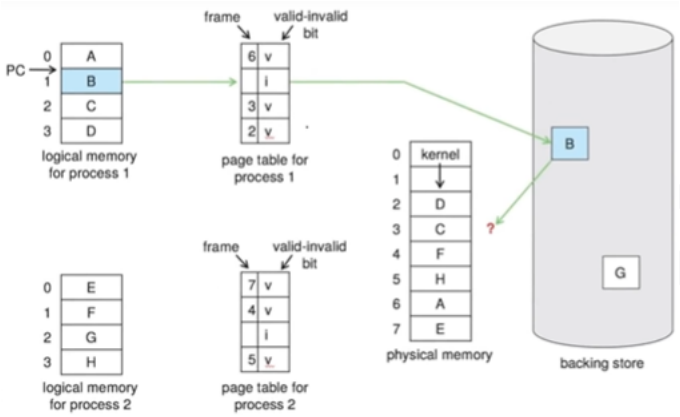
Page Fault

- if some pages are in a backing store, then this implies that a process may inevitably access a page that is not yet in the main memory
- access to a page marked invalid
- Handling procedure:

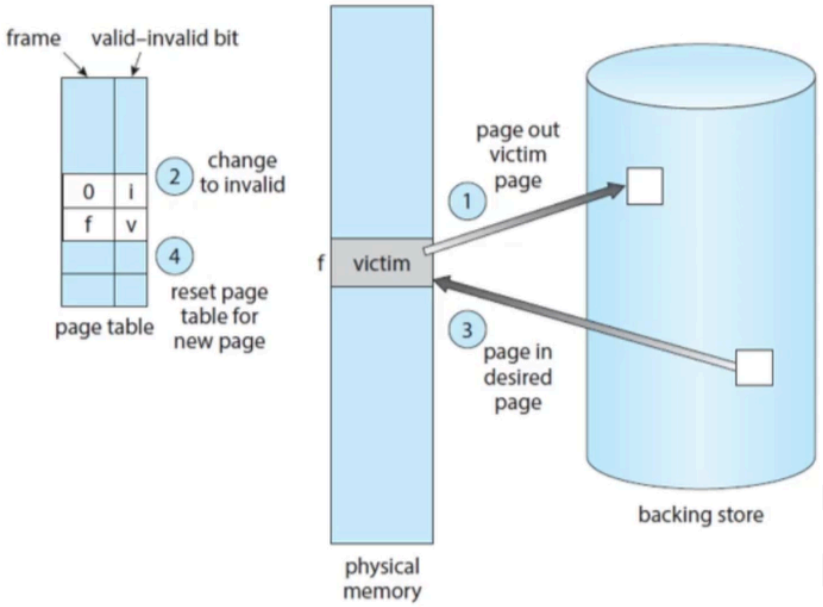


1. Check an internal table (page table) for this process to determine whether the reference was a valid or invalid memory access
2. If the reference is invalid, terminate the process (access violation error); if it was valid but not yet brought in that page, page it in
3. Find a free frame
4. Schedule a secondary storage operation to read desired page into the newly allocated frame
5. When the storage read is complete, modify the internal table kept with the process and page table to indicate that the page is now memory
6. Restart the instruction that was interrupted, as the process can now access the page as though it has always been in memory

Page Replacement



- As OS tries to run multiple processes, degree of multiprogramming increases, which means a high risk of running out of free frames
- OS has several options:
  - Terminate the process
    - easy to implement, but not good for the user
    - demand paging is the OS attempt to improve computer system's utilization and throughput
    - users should not be aware that their process are running on a paged system – paging should be logically transparent to the user (so not the best choice)
  - Perform page replacement
    - if no frame is free, then find a candidate page being used and free it
- General Algorithm:

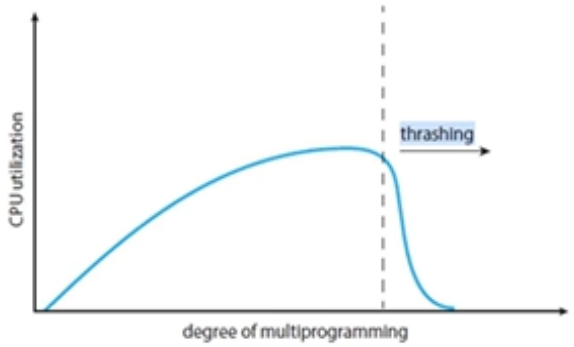


1. Find the location of the desired page on secondary storage
2. Find a free frame:
    - If there is a free frame, use it
    - If there is no free frame, use a page-replacement algorithm to select a victim frame
    - Write the victim frame to secondary storage; change the page and frame tables accordingly
3. Read the desired page into the newly freed frame; change the page and frame tables
4. Continue the process from where the page fault occurred

- technically means there is **infinite virtual memory**:
    - memory seen by OS and processes is not directly tied to the limits of physical memory
    - assume there are no free frames, OS will always find a victim frame to replace
- Different Page Replacement Algorithms (to choose the victim)
    - **Random**
        - Principle: randomly selects a page to evict
        - Pros: simplicity
        - Cons: unpredictable and may not perform well in practice
    - **FIFO (First-In-First-Out)**
        - Principle: evict the oldest page in the memory queue
        - Pros: simple and easy to implement
        - Cons: may suffer from "**Belady's Anomaly**", where increasing the number of frames does not guarantee a reduction in page faults

- **Optimal Page Replacement**
  - Principle: evict the page that will not be used for the longest period (optimal, but not practically implementable)
  - Pros: theoretical optimal option
  - Cons: require future knowledge of page accesses, which is not feasible in a real system
- **LRU (Least Recently Used)**
  - Principle: evict the page that has not been used for the longest time
  - Pros: intuitive and often effective
  - Cons: implementation complexity, and keeping track of usage timestamps can be resource-intensive, may result in thrashing

Thrashing



- occurs if a process does not have "enough" frames (does not have minimum number of grams to support pages in the working set), resulting in a page-fault
- a high paging activity where a process is spending more time paging than executing:
  - at this point, must replace some page; however, since all its pages are in active use, it must replace a page that will be needed again right away
  - consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately
- typically occurs when there are already so many processes running in a system

Differences between Paging and Demand Paging

- Paging
  - Divides the memory into fixed-sized blocks called pages
  - When a process executes, it splits into pages
  - When the process requests memory, the OS allocates page frames from the primary memory
  - The OS moves program pages from the secondary memory (SSD/HDD) o the primary memory frames

    - This technique improves efficiency of memory management by eliminating external fragmentation
- Demand Paging
    - specific implementation of paging
    - The OS loads only the necessary pages of a program into memory at runtime
    - A page fault occurs when the program needs to access a page that is not currently in memory
    - The OS then loads the required pages from the disk into memory
    - The OS updates the page tables accordingly
    - This process is transparent to the running program

Memory-Mapped Files and Shared Libraries

- To further reduce the need for page replacement and maximize the efficient use of main memory, one possible way is to dedicate portions of main memory for **shared data**
- Summary: Processes should only have their dedicated memory space used for unique data; and any data that could be potentially shared must reside only once in the main memory
- **Shared Libraries**
    - System libraries such as the standard C library can be shared by several processes through mapping of shared object into the virtual address space
    - Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all processes
        - typically, a library is mapped **read-only** onto the space of each process that is linked with it
- **Memory-Mapped Files**
    - Recall that each process will have some pre-allocated memory
        - Typically the heap/stack is pre-allocated and can run of space, especially for operations that perform reading/writing of big datasets
    - mechanism that enables a file or portion of a file to be directly mapped into the virtual memory of a process
        - allows the process to access the file using memory references rather than traditional read and write operations
    - commonly used in scenarios where a large dataset needs to be accessed in a way that is more efficient than traditional I/O, such as DB systems, certain types of file processing, and shared-memory IPC (inter-process communication)

Synchronization

Background

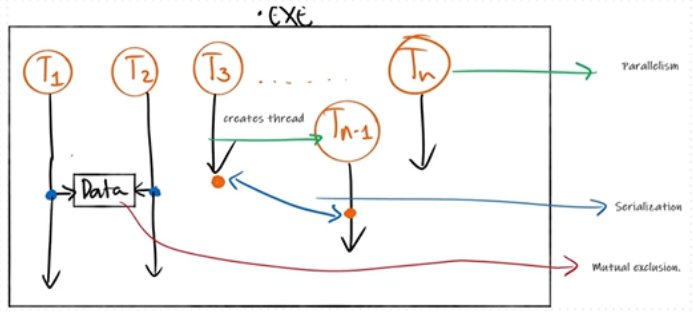
- All processes would often require one or more resources, but they are often limited (number of CPUs, amount of memory, storage available, I/O devices available)
    - At some point in time, there will be **resource contention** where multiple processes would attempt to get a certain resource
        - can be resolved using **mutex** (makes it thread-safe)
- CPU cores will simply perform their usual pipelined instructions (fetch, decode, execute), so there must be an arbiter (the OS) that decides what set of processes will execute concurrently
    - Often, this is done through threads to maximize parallelism and increase CPU utilization

Process Synchronization

- coordinates and controls concurrent processes or threads to ensure orderly execution and prevent conflicts in shared resources
- in a multitasking environment, where multiple processes or threads execute concurrently, it becomes crucial to synchronize their activities to maintain data consistency, prevent race conditions, and avoid conflicts that could lead to errors or system instability
- Motivations:
    1. **Shared Resources**: refers to CPU, memory, and devices
    2. **Data Consistency**: deals with transactions that occur at the same time
    3. **Race Conditions**: final outcome depends on order of execution of concurrent processes (synchronization mechanisms eliminate these)
    4. **Deadlock Avoidance**: process waiting for resources
        - Ex: circular dependency
            - a process needs a resource A that another process is using, but cannot unload it unless it gets resource B that the first process is currently using
    5. **Preventing Starvation**
        - Starvation: little progress because a process keeps using a resource for itself
    6. **Orderly Communication**: through shared data structures (to prevent corruption)
    7. **Concurrency Control**
    8. **System Stability**

Synchronization

- attempting to perform multiple tasks at the same time in an organized manner
- developers who create concurrent systems are primarily concerned with:



  - **Parallelism**: Task A and B should execute at the same time to speed up performance
  - **Serialization**: Task A should execute before Task B
  - **Mutual Exclusion**: Data type X should only be modified by one task, A or B

Why Concurrency Programming is Difficult

- Consider this sample program with two threads:

```
1  function Alice() {
2      eatBreakfast()
3      work()
4      eatLunch()
5      callBob()
6  }
```

```
8  function Bob() {
9      eatBreakfast()
10     eatLunch()
11 }
```

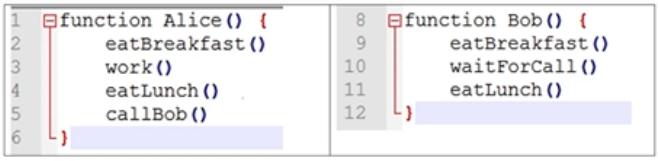
  - 1st scenario: A2 and B9 may execute in parallel through multiple cores, but it is unclear which will execute first
  - 2nd scenario: In a single-core Round-Robin, context switch would occur between the two, wherein either A2→B9 or B9→A2

Concurrency

- developer has no control over when each thread runs, so the OS makes those decisions (depending on the scheduling scheme)
- two events are **concurrent** if it is not clear looking at the code which will happen first

Attempting Serialization

- Goal: Task A executes before Task B
- callBob() passes a message to Bob, which is received by waitForCall()
  - as long as waitForCall() does not receive any message from callBob(), it cannot continue executing (thread sleeps)
- To ensure Alice calls eatLunch() first, perform **message passing**:

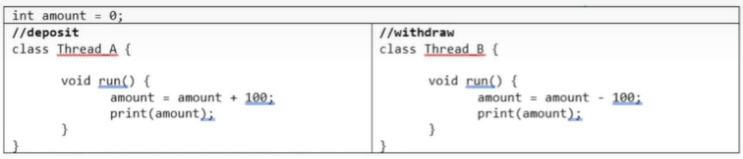


  - Adding waitForCall at line 10 puts Bob thread to sleep, until callBob is called by Alice thread
  - As a result, A4 < B11
    - < and > means if a task occurred before or after another task respectively

Non-determinism in Concurrency

- Concurrent programs are **non-deterministic**, as it is impossible to tell by looking at the code which will execute first (making it hard to debug)
- can only be addressed by careful programming

Sample Problem: Withdraw-Deposit

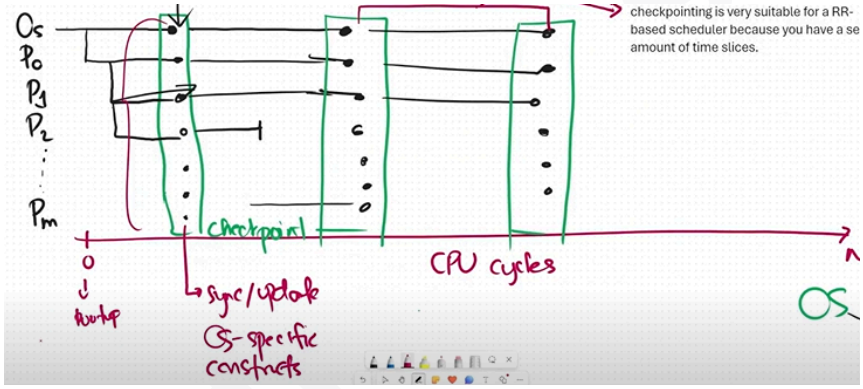


- **Critical section**: refers to a part of the code where there is shared, modifiable data
  - Ex: accessing the data can cause the following: 0, -100, +100 (goal: 0)
- Criteria for solving the critical section problem:
  - **Mutual Exclusion**: if process P_1 is executing in its critical section, then no other processes can be executing in their critical section
    - Ex: thread A runs first, and then thread B can only run once A is done
  - **Progress**: if no process is executing in the critical section and there exist some processes that wish to enter it, then the selection of processes that will enter next cannot be postponed indefinitely
    - Ex: if no thread is in the critical, the newly arrived thread runs it ASAP

- **Bounded Waiting**: a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
    - assume each process executes at nonzero speed
    - no assumption concerning relative speed of the *n* processes
    - Ex: assuming persistent threads, there is a wait time before accessing

Cooperative vs Preemptive Multitasking

- **Cooperative Multitasking**: all processes must be programmed (by the developer) to yield control voluntarily
    - in a process, there would be a for loop for execution, and after the loop is a yield (return execution to the OS to allow other processes to update)
    - Problem: a process that spawns processes in an infinite while loop will hog the system and cause it to crash
- **Preemptive Multitasking**: OS completely manages scheduling, which decides when a process should be paused, and another process should be given CPU time (like a checkpoint to sync/update)



    - allows OS to preempt a running process to ensure fair allocation of CPU resources among all processes
    - improves system responsiveness and stability, as no single process can monopolize the CPU
    - very suitable for RR-based scheduler
    - employed by most modern OS
    - Ex: In Android, OS would throw not responding dialog box to close the app if needed when it has not refreshed for a long time

Peterson's Solution

```
1  int amount = 0;
2
3  int turn = 0;
4  bool flag[2] = {false, false};
```

```
7  //deposit
8  class Thread_A {
9      void run() {
10         flag[0] = true;
11         turn = 1;
12
13         while(flag[1] && turn  == 1) {
14             //busy wait
15         }
16
17         amount = amount + 100;
18         print(amount);
19
20         flag[0] = false;
21     }
22 }
```

```
24 //withdraw
25 class Thread_B {
26     void run() {
27         flag[1] = true;
28         turn = 0;
29
30         while(flag[0] && turn  == 0) {
31             //busy wait
32         }
33
34         amount = amount - 100;
35         print(amount);
36
37         flag[1] = false;
38     }
39 }
```

- two process solution
- amount, turn, and flag are shared data types
- Trace Scenarios:
    - Assume parallel:
        - At line 11, there is an intentional race condition to determine whoever will win ends up going first
        - The while loop is a busy wait that allows the other thread to execute the critical section first
    - If T_a executes first
        - if flag[0] and turn are true and 0 respectively
        - T_b would go on busy wait until T_a runs the critical section
    - If T_b executes first
        - same idea as T_a
- based on the scenarios, the program has proven to work on a multithreaded scenario as it satisfies the 3 criteria:
    - Mutual exclusion:
      If T_a is in the critical section → flag[0] = true and T_b will have $turn_b = 0$, which causes T_b to perform busy waiting
    - Progress:
      If T_a finishes executing critical section, T_b guarantees access to the critical section because flag[0] becomes false, and vice versa
    - Bounded waiting:
      Since flag[0]/flag[1] becomes false in the future, this guarantees bounded waiting because the next thread inside the busy-waiting while-loop gets to enter the critical section
- *Note: Shared data types in this case is only applicable for 2 threads; however, a variant exists for N threads where N > 2*