Section 1 : Processes and Threads

What is a process?

A process is like a box that holds everything needs to run a program

- A process is a program running on your computer.
- Each process has its own memory and resources
- It runs independently, it doesn't share memory with other processes.
- Processes include at least one thread (usually a main thread)
- Created using system calls like fork() or exec()

Def : Main thread- The first thread that is created when a process starts.

What are the components of a process?

- Text segment (the code)
- Data segment (global and static variables)
- BSS segment (uninitialized globals)
  - These are the variables that are declared but not initialized yet
- Heap : Dynamically allocated memory
- Stack : Function call frames, local variables, return addys
  - **Def - Function call frames** : When you call a function, a stack frame is created. A stack frame is a section of the stack that stores things.

Interprocess Communication (IPC)

What is it - How different process talk to each other.

Why important in OS design? - Important for process coordination/resource sharing in multi process environment.

Types of IPC

1. **Shared Memory** - A region of memory is mapped so that multiple processes can access it
   - Fast, but needs synchronization (***semaphores***) to avoid conflicts
   - Functions like mmap() or memory segments are used to do this
2. **Message Passing** - Used when process can't share data directly
   - Slow due to overhead (copying), but safe and structured
     - Safe - Processes don't accidentally overwrite each others memory
   - Includes
     - Pipes - Connecting output of one process to another
     - Sockets - Like a plug on the wall. Lets two programs connect & talk to each other
     - Message Queues - Data structure where processes can send/receive messages

Different types of Pipes

- Unnamed Pipes - Used between parent and child processes, created with pipe()
- Named Pipes (FIFOs) - Allows unrelated processes to communicate, exists in file system.

How does the OS represent a process?

Represents a process through the PCB. OS's record of everything it needs to manage a process

The PCB contains

- Process ID - Unique number given to each process. Used to tell processes apart
- Program Counter - Address of the next instruction the process will execute (bookmark)
- CPU register contents - Current values inside the CPUs register while the process is running. This is used to resume a process during a context switch
- Scheduling info - Info OS scheduler uses to decide when the process should run (priority, runtime)
- Memory maps - Details about how the processes memory is organized (where it's code, data, stack, and heap are located)
- Open file table - A list of files the process has open
- I/O state - Info about input/output devices being used (if the process is waiting for something)
- Execution state (ready, waiting, etc) - Current state of the process
  - Running : its on the CPU now
  - Ready: Waiting for the CPU
  - Waiting : paused, waiting for something (like input)

Context Switch

When the OS switches between processes, pauses one and starts another. it saves/restores the PCBs. This is expensive, since it entails copying a lot of the state.

Process States and Transitions

Common States:

- New - Just created
- Ready - waiting for CPU
- Running - executing
- Waiting - blocked (i.e for I/O)
- Terminated - done

Transitions:

- Ready → Running (when CPU assigned)
- Running → waiting (i.e. for I/O call)
- Waiting → Ready (I/O finishes)
- Running → Terminated (exit)

Difference between Process States and Transitions

- Process state = what the process is doing at a specific moment
  - Current condition of a process
- Process transitions = How a process moves between states
  - The actions of events that move the process from one state to another

Difference between fork() and vfork()

- fork()
  - Creates a child process, child gets a copy of the parents address space
  - Both process continue to execute independently from the point of the fork() call

Analogy: Like photocopying your document and letting the new version go off and do its own thing

- vfork()
  - Also creates new child process, but doesn't copy the address space
  - Child shares the parents memory temporarily, but cannot change it
  - Parent is suspended until the child calls exec() or exits

Analogy: Like borrowing your parents room for a second while they wait in the hall, but you cant mess anything up before you leave

When to use which

- Use fork() for general for general purpose process creation when the child might need to run code before replacing itself
- Use vfork() for performance when the child will immediately call exec() and doesn't touch memory, when we want to create a child process whose only job is to replace itself with another program

Def - Wait() is a system call that stops the parent from executing until the child is done.

Def - Clone() - Custom version of fork() where you can decide which parts of the parent's execution context the child process will share.

User and System Mode

Two modes the CPU can operate in

User Mode : Used by regular programs

- Runs normal instructions, cannot access hardware or OS internal

System Mode: Used by the OS kernel

- Has full access to memory, devices, and CPU instructions
- Has the ability to access more "privileged" things.

What is mmap()?

mmap() can be used to map files or devices into memory. Its used to create shared memory regions between processes.

Process Address Space Layout

```
+-------------+
|  Stack      | <- grows downward
+-------------+
|  Heap       | <- grows upward
+-------------+
|  BSS        |
+-------------+
|  Data       |
+-------------+
|  Text       | <- code
+-------------+
```

Process vs Thread

- Processes are independent units with their own address space and PCB (Process Control Block)
  - The process control block is used by the OS to keep track of all the important info about a process
- Threads are lightweight processes, sharing address space and resources of the process.

Feature	Process	Thread
Address Space	Separate	Shared with other threads
PCB	Yes	Shares PCB with other threads
Creation Overhead	High (due to memory copying, etc.)	Low (uses same memory/resources)
Communication	Requires IPC mechanisms	Can communicate via shared memory
Fault Isolation	Strong	Weak

How are Processes Created?

1. **Fork()** - Clones the calling process
   - Child gets a copy of the parent's memory (copy on write)
     - Not actually copying everything (too slow and wasteful)
     - **Copy on write** means the parent and child share the same physical memory pages. These pages are marked as read only. If either process tries to modify a page, only then a copy is made (the write triggers the copy)
   - Same code, same stack, same heap - different PID (Process ID)
   - Often used when a process wants to create a new process

   Analogy : Like duplicating a document
2. **Exec()** - Replaces current process with a new program
   - Overwrites the calling process with a new program
   - PID stays the same, but the code, stack, heap, etc are replaced
   - Used after fork() to load new code into the child - when the child wants to run a different program than the parent

   Analogy : Like erasing all the content in your document and pasting in a new one

What is a thread?

A thread is like a worker inside the process box. It's what actually does the work. Its the smallest unit of execution

- Threads share the same memory and tools of the process that they belong to
- Each thread has its own
  - Stack - stores variables and information
  - Program counter - Register that keeps track of where the thread is in the program
    - Stores memory address of the next instruction the thread will execute
    - Every time the thread runs an instruction, PC updates to point to the next instruction
  - CPU register state - Values stored in the CPUs registers at a specific point in time when the thread is running
    - Registers are tiny super fast storage areas inside the CPU that hold
      - Program counter,
      - stack pointer (points to the top of the current stack frame) ,
      - general purpose registers (used for things like holding variable values or temp data

Types of threads

- User level: Managed by a user space library (Pthreads). Fast but invisible to kernel
- Kernel level: Managed by OS. Scheduler is aware of these and can assign them to CPUs

Process Address Space: One Thread vs Multiple Threads

Feature	One Thread	Multiple Threads
Address Space	Shares text, data, BSS, and heap — only one thread uses them	All threads share text, data, BSS, and heap
Stack	One stack	Each thread has its own stack
Program Counter	One program counter	Each thread has its own program counter
CPU Register State	One CPU register state	Each thread has its own CPU register state
Execution	No concurrency — only one thread executes	Multiple threads can execute concurrently
Communication	No communication needed (only one thread)	Threads communicate via shared memory (synchronization required)
Complexity	Simple	More complex (must manage race conditions and synchronization like mutexes)

Multiprogramming - Running multiple programs in mem, switching between to keep CPU busy.
Multiprocessing - Using multiple CPUs to run processes simultaneously.
Multithreading: Running multiple threads within single program to perform tasks concurrently.

System Calls - System calls let user programs request services from the OS, like file access or process control. Examples - read() for reading data and fork() for creating new processes.

Address Space fork vs thread - Threads share the entire address space, while forked processes receive a separate copy of the address space.

Traps - Software-generated interrupts triggered by a program when it needs to request a service from the operating system (like a system call) or when an error occurs. Switches CPU from user mode to kernel mode, allowing the OS to take control and handle the event safely.

Processes are called **heavyweight** because they have more information for the OS to manage, like memory, open files, and full CPU state stored in the **PCB** (Process Control Block). Threads are **lightweight** since they only need a small amount of info stored in the **TCB** (Thread Control Block) and share most resources with their parent process. This makes **threads faster to create and switch between**, while **processes are slower but more isolated**.

Threading Models

How user threads map to kernel threads.User threads cant run unless backed by kernel threads

1. **1:1 (One to One) -** Each user level thread is directly mapped to a unique kernel level thread

   How it works: The OS is aware of each thread, if a user thread blocks (is waiting on I/O), the kernel can switch to another thread

   Pros:
   - True parallelism on multicore syst - multiple threads can run on multiple CPUs at once

   Cons
   - High overhead - Creating/managing multiple kernel threads takes more time/memory
   - There may be limits on how many kernel threads can be created
2. **M:1 (Many to One)** - Many user threads are mapped to one kernel thread

   How it works:
   - The kernel sees only one thread - all scheduling and switching happens in user space.
     - User space is part of mem where user programs run
   - If one thread blocks, all threads block. The kernel cant mange them individually
     - **Pros:**
       - Low overhead, simple implementation
       - No kernel involvement = fast thread switching
     - **Cons:**
       - No parallelism on multicore systems - only one thread runs at a time
       - One blocking thread can halt the entire process
3. **M:N (Many to Many)** - Maps M user level threads to N kernel threads

   How it works
   - The OS can schedule multiple kernel threads across cores
   - User level thread library handles creation and scheduling of threads - allows system to balance between flexibility and efficiency

   Pros :
   - can run multiple threads in parallel
   - More scalable than 1:1 - OS only creates kernel threads when needed

   Cons : Very complex to implement

Interrupts

Signals sent to the CPU by hardware or software to indicate something needs immediate attention. Allow OS to regain control from current running process when certain events happen.

- Hardware Interrupt - Triggered by devices
- Software Interrupt - Generated by programs
- Timer interrupt - Used by the OS scheduler to regain control after a time slice

Purpose: Interrupts notify the OS of important events and allow it to react

Mechanism: When interrupt occurs, the CPU pauses whatever program is running and switches to a special piece of OS code designed to handle that interrupt, called an interrupt handler

Timer interrupt is basically like an alarm clock set by the OS that goes off every few milliseconds to take back control of the CPU. **Prevents an infinite loop.**

- **User programs cannot disable interrupts**. Only the OS in kernel mode can do this. If user code could disable them, it could take over the entire system.
- Mechanism to get back crontrol from OS from a running process is the timer interrupt.