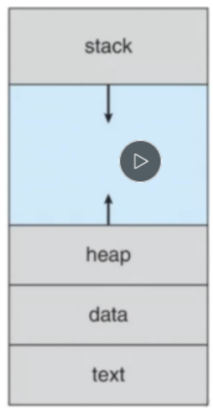
Processes

Process Overview

- A **process** is the instance of a **computer program** that is being executed by the **processor**.
- **Processes** are located in the system's **primary memory**, and are composed of:
  1. The process text (executable instructions).
  2. The process data (global and static variables).
  3. The process heap (dynamically allocated memory).
  4. The process stack (local variables and function calls).
  5. The process state (managed by the kernel).



- Each **process' memory** is **isolated**, meaning it **cannot be accessed** by **other processes** (other than the kernel).
  - This isolation is automatically enforced by the operating system.
  - One bennifit of isolation, is the ability to limit the reach of malicious processes.
- **Processes** may be composed of **several threads** allowing for **several instructions to be executed simultaneously**.

Process–Kernel Communication

- **Processes** are not able to **directly access the kernel's memory**. Instead they use system calls, which interrupt the kernel, and indicate that a process requires a service.
- The **kernel** can access all **process'** memory.

Sharing the CPU

- When **several processes** are running, system **resources need to be shared**.
- **Multiprogramming Operating Systems run processes** until they **block for an event**, when processes block for an event, they are **placed in a queue**, and the operating system executes **other processes**.
  - This scheduling technique is bad because it could result in starvation if one process runs into an infinite loop.
- **Time Sharing Operating Systems** give each **process** a **specific amount of time** to execute, and then **switch to the next**.

- This scheduling technique improves performance, and prevents starvation.
- This scheduling technique also gives the illusion that each process is running concurrently.

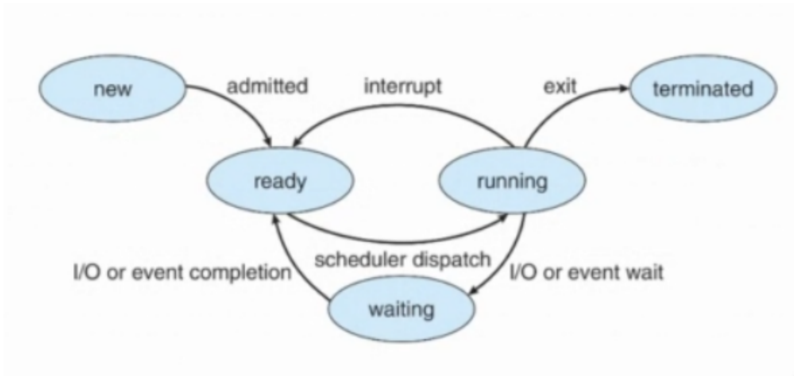
Sharing Memory

- Each process gets it's own **memory map**, which tells the **kernel** what memory belongs to the **process**.
- The **single contiguous memory model** is a model where the **RAM is occupied by one process at a time**. When that process completes, another is loaded.
  - This model limits the size of processes to the maximum amount of RAM, and can only execute one process at a time.
- The **partition model** allows **multiple processes** to **occupy** the RAM **simultaneously**.
  - As long as sufficient contiguous memory is available, new processes are allocated memory.
  - This model uses a partition table, that contains information about the allocated and unallocated memory (size, location, process occupying it).
  - The operating system should detect unallocated contiguous memory blocks, and merge them into one large block. This has a lot of overhead, and leads to poor performance.
- Modern operating systems use **virtual memory**, and **segmentation.**
  - **Virtual memory** splits ram into **fixed size partitions** called **page frames** (typically 4KB).
  - A **process** is **split** into **blocks of equal size** (block size = page frame size). Each block is numbered increamentally, however, the page frame they correspond to may or may not be consecutive.
  - **Each process** is given a **page table**, that maps **blocks** to **actual page frames**.
  - The processor **does not need to include all blocks** of a process in memory when it runs, only the ones that are required. It can **load** and **unload** them.

Process Control Block

- The **process control block (PCB)** contains information associated with a **single process**.
- The PCB contains:
  1. The **process state** (running, waiting, etc).
  2. The **program counter** (location of the next instruction to execute).
  3. **CPU register** values.
  4. **CPU scheduling information** (priorities, queue pointers).
  5. **Memory management information** (memory allocated to the process).
  6. **Accounting information** (CPU used, clock time since start, time limits).
  7. **I/O status information** (devices allocated to the process, file descriptors, etc).

Process State Diagram

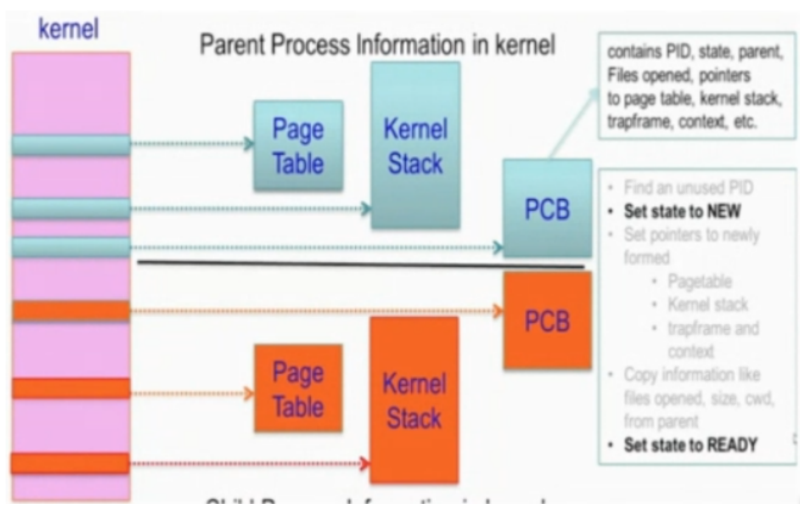


Process Creation

- **Processes** are created from **forking**, which forms a **tree of processes** as more processes call fork.
- The **main process** (PID 1) is the process that is responsible for **managing the operating system**.
- **Copy on Write (COW)** — Initally, when fork is called **the pages are shared**. When data in the **shared pages change**, the OS **makes a copy of the page**, resulting in the child and the parent process having different copies of the specific page that changed.

Kernel Process Data

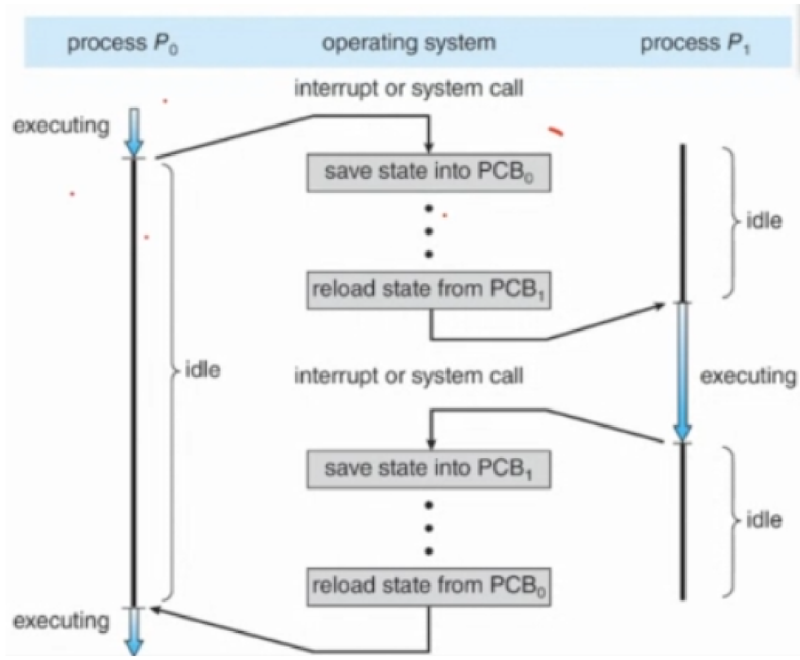
- The kernel maintains a **page table**, a **process control block (PCB)**, and a **kernel stack** for each process.
- When a **process forks**, the **new child process will contain a clone of the parents data**. This data will have minor differences (pid, page table, etc).



Process Scheduling

- **Process scheduling** is the way an **operating system** determines what **process' instructions** should be executed on a **CPU core**.
- The **goal of process scheduling** is to **maximize CPU use**, and **quickly switch between processes**.

- The **ready queue** is a queue of **processes** already in the main memory, that are **in the ready state**.
- The **process scheduler** takes **processes from the queue, and executes them**.
- The **scheduler** is triggered when a **timer interrupt occurs** or when a **process blocks for IO**. The **scheduler** will perform a **context switch** and start executing **another process**.



Process Termination

- The **exit** system call returns an exit status to the **parent process**, which can be retrieved by the parent process by the **wait** system call.
  - When the **wait** system call is invoked, the parent goes into a **blocked state** until one of it's **children terminate**. If there are no children, -1 is returned.
- After **exit** is invoked, the process' resources are **deallocated by the operating system**.
- The **parent process** may **terminate the execution** of **itsself and/or it's child processes** with the **abort** system call.
  - This may be done if the child has exceeded allocated resources, the child's task is no longer required, or the parent is exiting.

Zombie Processes and Orphan Processses

- When a **process terminates** it **still exists** in the operating system to allow the **parent process** to read the **child process' exiy status**. When a child process is in such a state, it is referred to as a **zombie process**.
- When the **parent process** reads the **child process' status** the zombie process is removed.
- If the **parent never** invokes the **wait** system call, the **zombie process** will exist **infinitely** (which is a resource leak).
- An **orphan process** is a process whose **parent has termated** before it.
- When a **process' parent termates before it**, it is **adopted by the first process (init process)**.

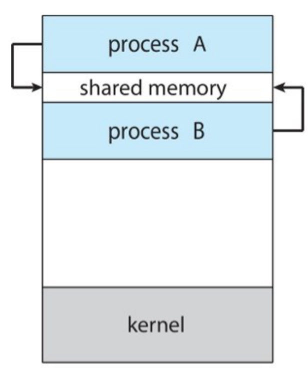
Interprocess Communication

Process Concepts

- **Each process** can only view **its virtual address**. It **cannot view others**, and **cannot determine the physical address mapping**.
- **Processes** within a system may be **independent** or **cooperating**.
- **Cooperating processes** can **affect** or be **affected** by **other processes**.
- Reasons for **cooperating processes** include:
  1. Information sharing.
  2. Computation speed increase.
  3. Modularity.
  4. Convenience.
- **Cooperating processes** need **interprocess communication (IPC)**.
- There are two **models** of **IPD**: **shared memory**, and **message passing**.

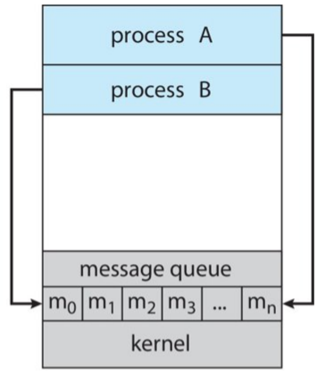
IPC Shared Memory

- **IPC shared memory** involves **allocating memory** that **several processes can access**.
- In this model, the **communication** is under the control of the **user's processes** not the **operating system**.
- This model is **fast**, but it is **very difficult** to provice a mechanism that will allow the **user's process** to **synchronize their actions**.



IPC Message Passing

- **IPC message passing** uses **shared memory** created **in the kernel**.
- **Processes** then use system calls to **send**, and **receive data**.
- This is **slower**, but allows for **processes** to easily **synchronize their communication**.



Pipes

- To allow **two processes** to **communicate** using **message passing**, we use **pipes** to establish a **communication link**.
- A **parent process** creates a pipe via the **fork** function, and can use it to communicate with **child processes**.
  - The pipe must be created before the child process.
- **Pipes** are **unidirectional communication channels** that consist of two file descriptors: **read** and **write** respectively.

Named Pipes

- **Named pipes** are **more powerful** than **ordinary pipes**.
- **Named pipes** allow for **bidirectional communication**, and can be used by **more than two processes**.
- **Named pipes** also do not require a **parent-child relationship** to establish communication.

Sockets

- A **socket** is a **communication endpoint**.
- **Sockets** are **bidirectional communication channels** that allow **two processes** to communicate **over a network**.
- **Sockets** use **ports** which are **bound to file descriptors** to enable communication.
- All **ports below 1024** are **well-known ports** that are used for standard services.

Remote Procedure Calls

- **Remote Procedure Calls (RPC)** abstracts **procedure calls** between **processes** on **networked systems**.
- **RPC** uses **sockets for communication**.