Operating Systems Concepts

Processes

CS 4375, Fall 2025

Instructor: MD Armanuzzaman (***Arman***)
marmanuzzaman@utep.edu

September 3, 2025

Summary

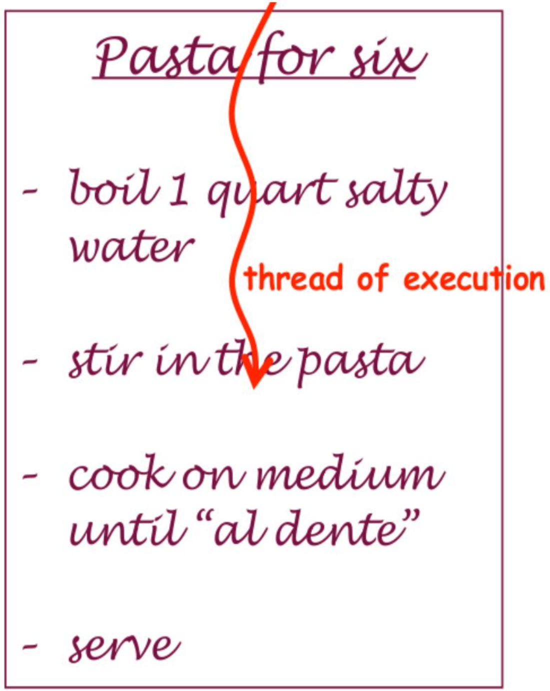
- A very brief history of computer architecture
- High level topics of the course
- Goals or Responsibilities of OS
- Tasks of an OS
- Major OS components
- OS design approaches

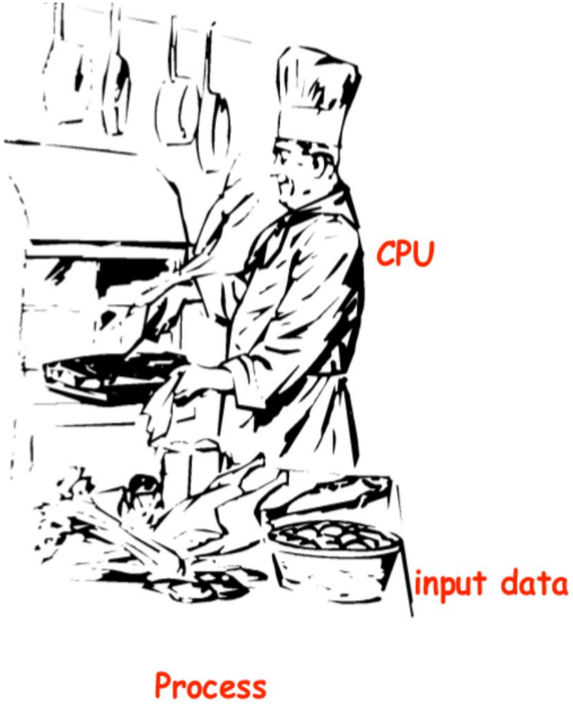
Agenda

- Processes
  - Process representation in OS
  - Process creation
  - Process termination
  - Context switching
  - Process queues
  - Process scheduling
  - Interprocess communication

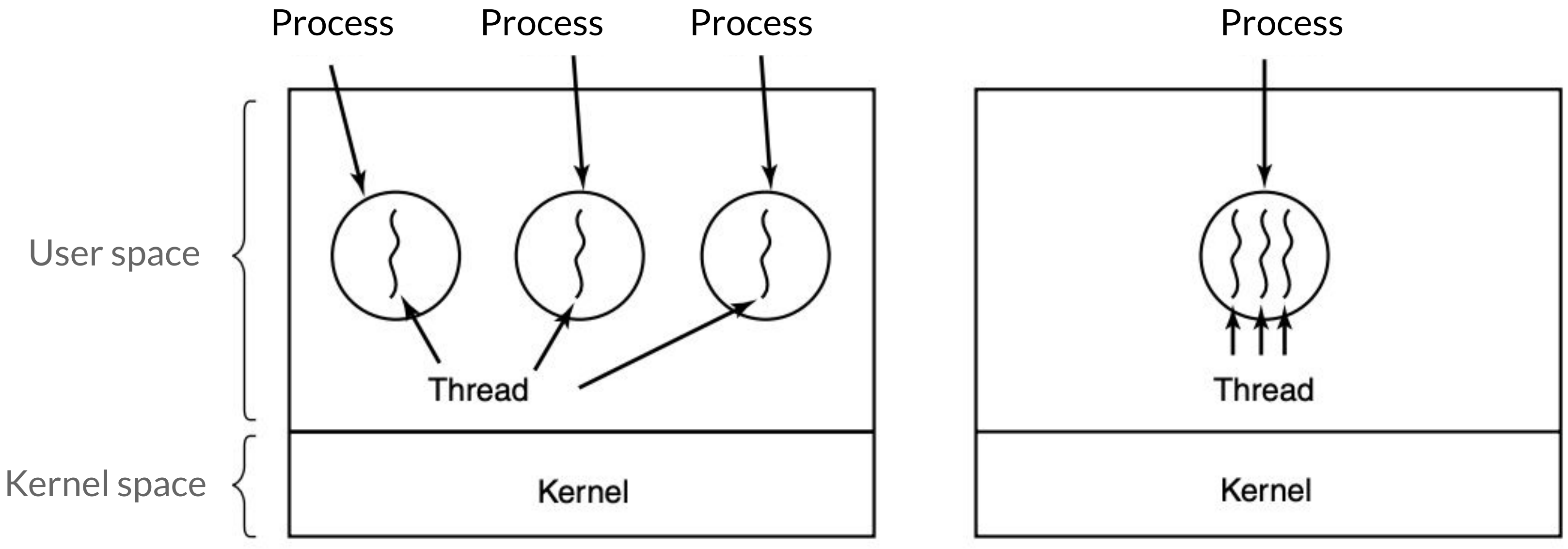
Processes Concept

- A process is a program in execution.





Process vs Thread



Processes Concept

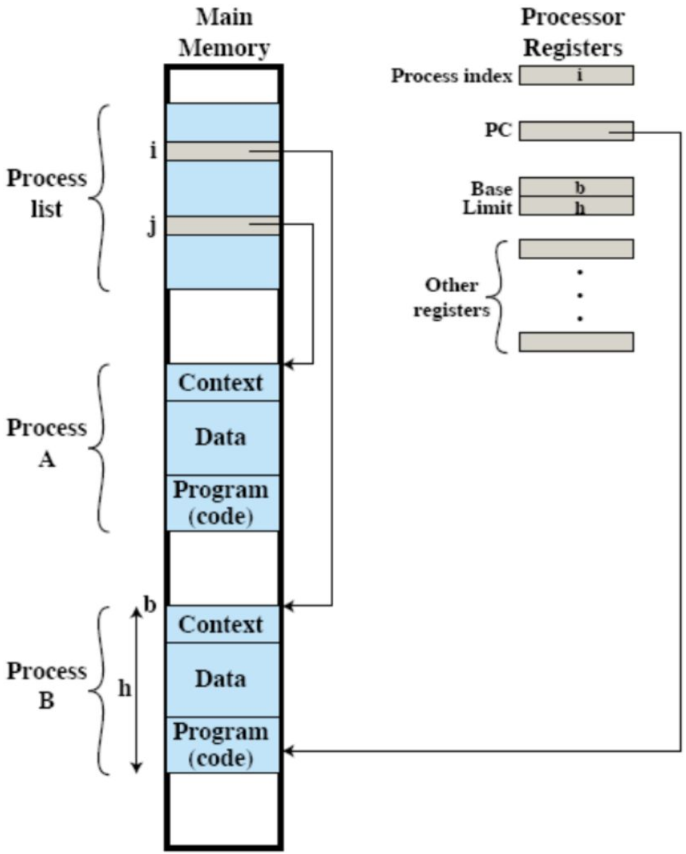
- A process image consists of three components:
  - An executable program
  - The associated data needed by the program
  - The execution context of the process which contains all information the OS needs to manage the process (ID, state, CPU registers, stack, etc.)

User Address space

Processes Concept

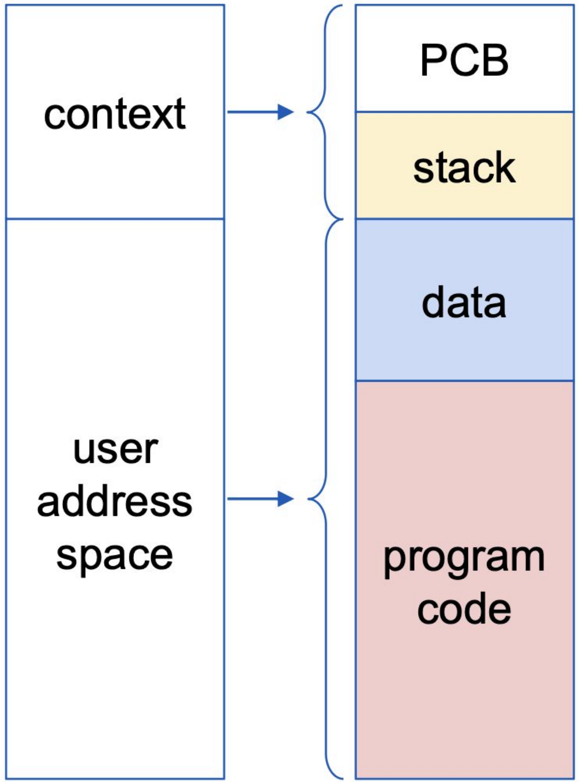
- A process image consists of three components:
  - An executable program
  - The associated data needed by the program
  - The execution context of the process which contains all information the OS needs to manage the process (ID, state, CPU registers, stack, etc.)

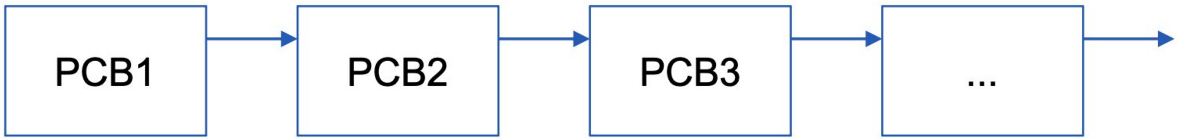
User Address space



Process Control Block (PCB)

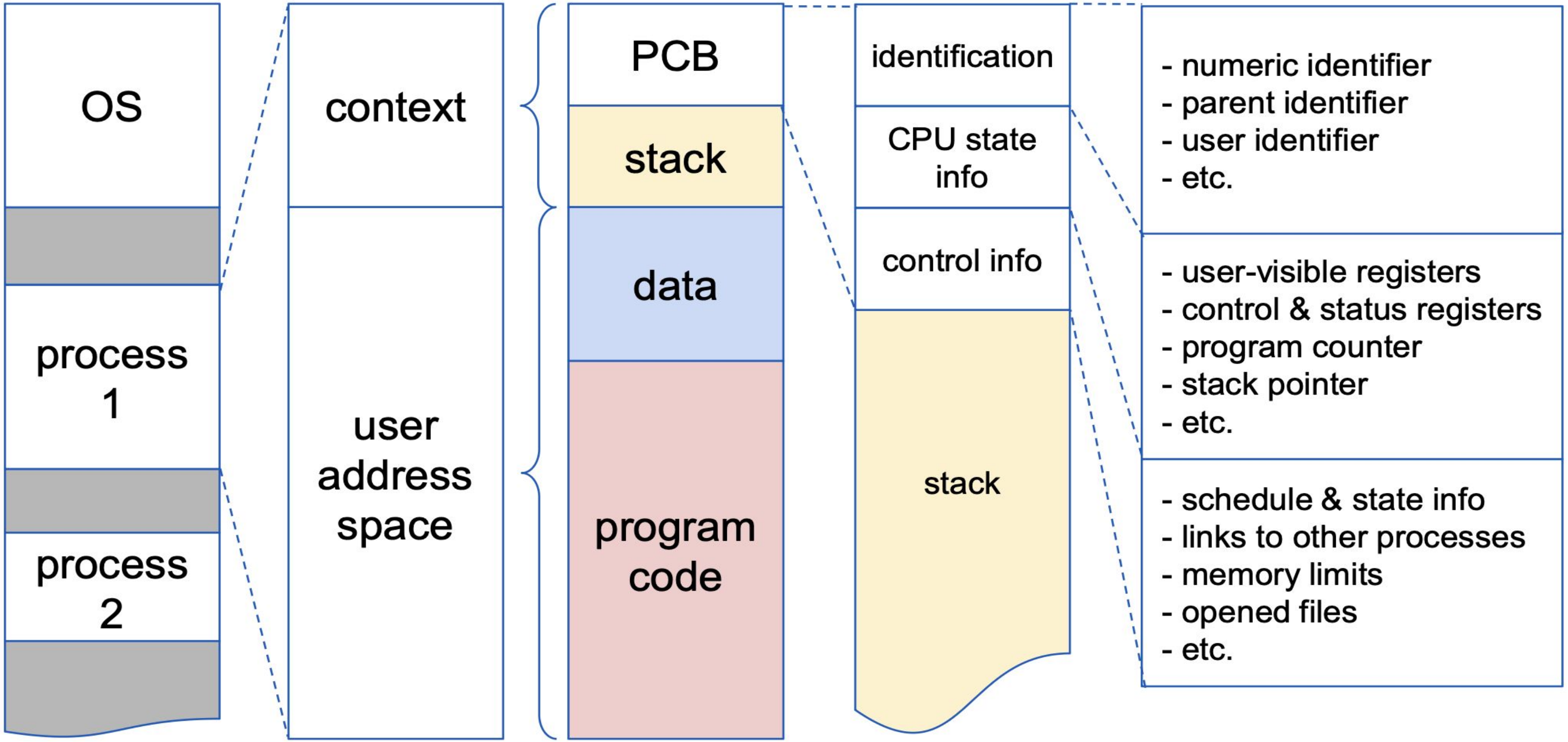
- PCB is included in the context, along with the stack
- PCB is a "snapshot" that contains all necessary and sufficient data to restart a process where it left off (ID, state, CPU registers, etc.)
- PCB is one entry in the OS's process table (array or linked list)





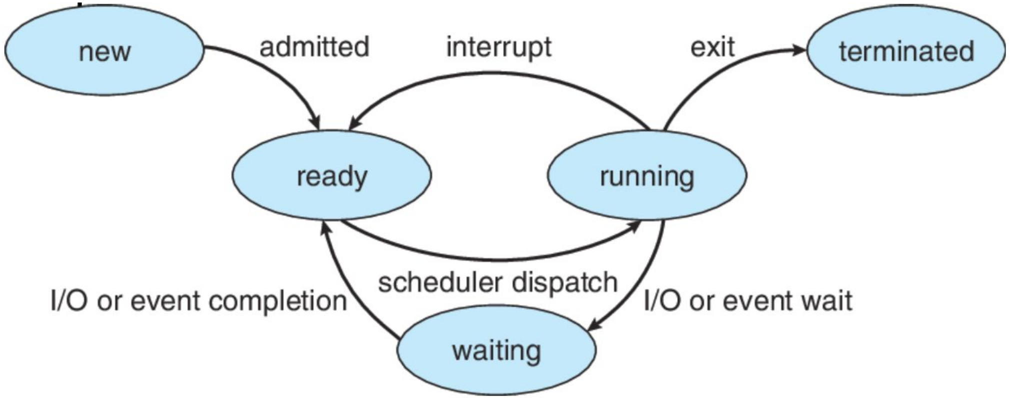
Process Control Block (PCB)

Illustrative contents of a process image in (virtual) memory



Process State

- A process changes its state during execution:
  - New: The process is being created
  - Ready: The process is waiting to be assigned to a processor
  - Running: Instructions are being executed
  - Waiting: The process is waiting for some event to occur
  - Terminated: The process has finished execution



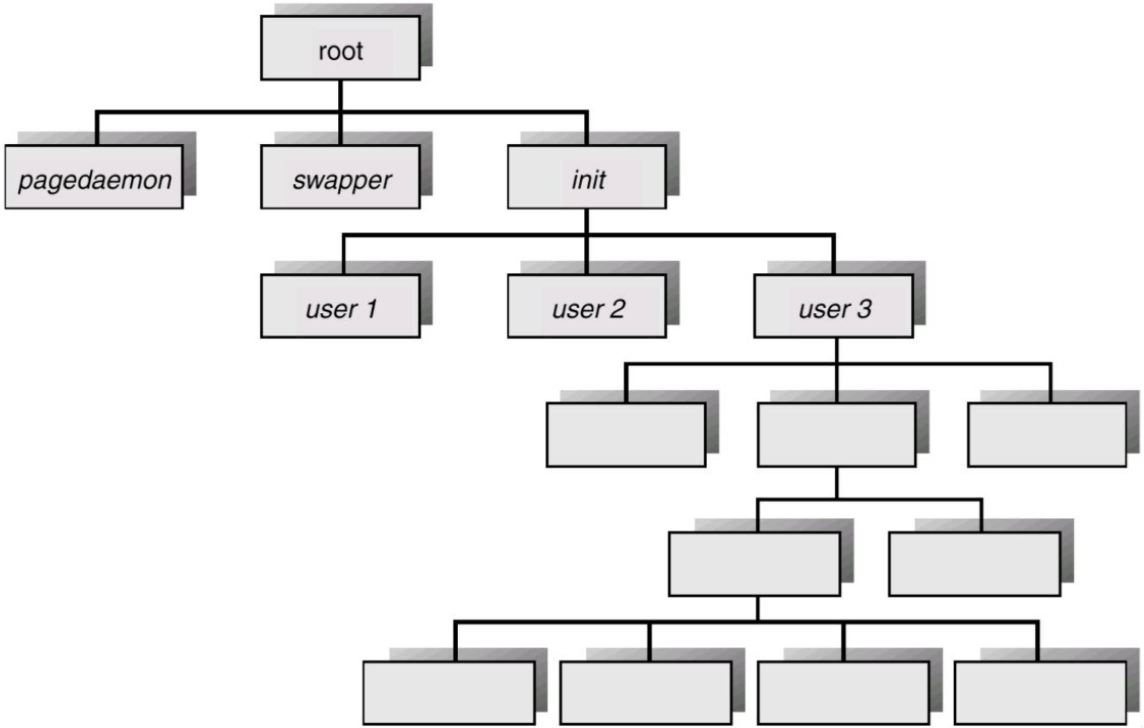
Process Creation

Some events that lead to process creation:

- The system boots
  - When a system is initialized, several background processes or "**daemons**" are started (email, logon, etc.)
- A user requests to run an application
  - By typing a command in the CLI shell or double-clicking in the GUI shell, the user can launch a new process
- An existing process spawns a child process
  - For example, a server process (e.g. web server, file server) may create a new process for each request it handles
  - The init daemon waits for user login and spawns a shell
- A batch system takes on the next job in line

Process Creation

Process creation by spawning. A tree of processes on a typical UNIX system:



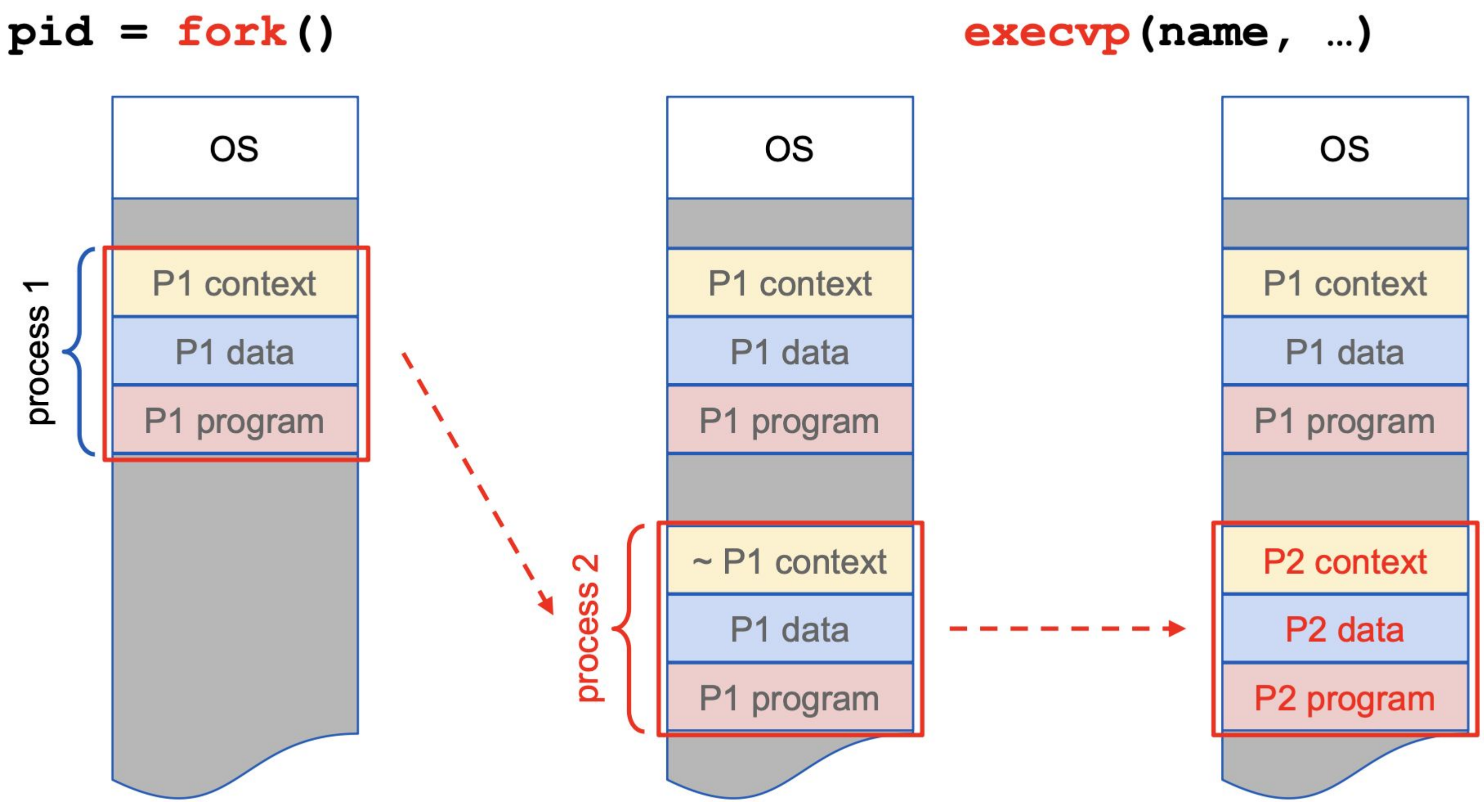
Process Creation: preliminaries

- fork() system call
    - *pid_t fork(void);*
    - Creates a new process in Unix systems, which is called the child process
        - Returns
            - **0:** On success to the child
            - **Child PID:** On success to the parent
            - **-1:** On failure and type in errno
- exece() system call
    - Family of functions replaces the current process image with a new process image
    - *int execvp(const char *file, char *const argv[]);*
    - Returns -**1** on failure

Process Creation

1. Clone child process

2. Replace child's image



Fork Example 1

```c
#include <stdio.h>
main()
{
    int ret_from_fork, mypid;
    mypid = getpid(); /* who am I? */
    printf("Before: my pid is %d\n", mypid); /* tell pid */

    ret_from_fork = fork();
    sleep(1);
    printf("After: my fork returns pid : %d, said %d\n",
                            ret_from_fork, getpid());
}
```

Process Creation

```
...
int main(...)
{
    ...
    if ((pid = fork()) == 0) // create a process
    {
        fprintf(stdout, "Child pid: %i\n", getpid());
        err = execvp(command, arguments); // execute child process
        fprintf(stderr, "Child error: %i\n", errno);
        exit(err);
    }
    else if (pid > 0) // we are in the parent
    {
        process
            fprintf(stdout, "Parent pid: %i\n", getpid());
        pid2 = waitpid(pid, &status, 0); // wait for child process
        ...
    }
    ...
    return 0;
}
```

Fork Example 2

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

int main()
{
    fork();
    fork();
    fork();
    printf("Process pid is %d\n", getpid());
    return 0;
}
```

- **How many lines of output will this code produce?**

Process Termination

Some events that lead to process termination:

- Regular completion, with or without error code.
    - The process voluntarily executes an ***exit(err)*** system call to indicate to the OS that it has finished.
- Fatal error (uncatchable or uncaught)
    - Service errors: no memory left for allocation, I/O error, etc.
    - Total time limit exceeded
    - Arithmetic error, out-of-bound memory access, etc.
- Killed by another process via the kernel
    - The process receives a **SIGKILL** signal
    - In some systems the parent takes down its children with it.

Process Pause/Dispatch

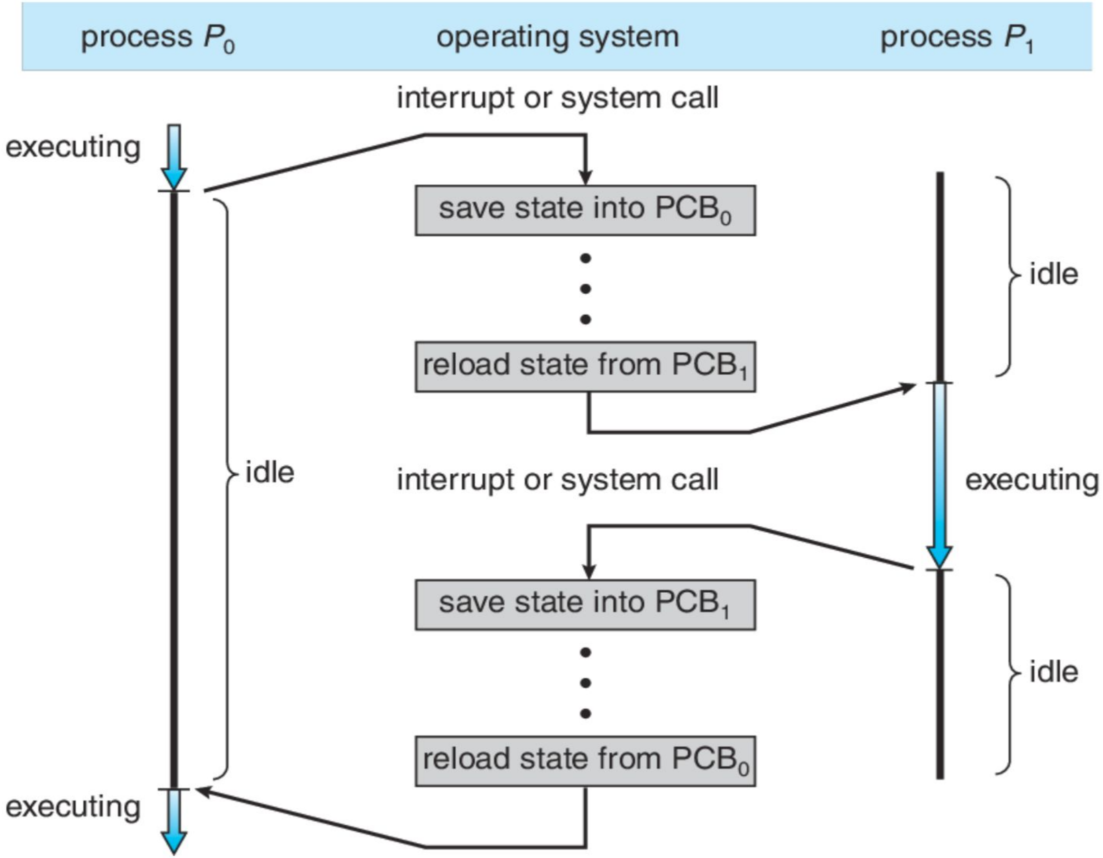
Some events that lead to process pause/dispatch:

- I/O wait - OS triggered (*e.g. system call*)
  - A process invokes an I/O system call that blocks waiting for the I/O device: The OS puts the process in "Waiting" mode and dispatches another process to the CPU.
- Preemptive timeout - Hardware interrupt triggered (*e.g. timer*)
  - The process receives a timer interrupt and relinquishes control back to the OS dispatcher: The OS puts the process in "Ready" mode and dispatches another process to the CPU.
  - Not to be confused with "**Total time limit exceeded**", which leads to process termination.

Process Context-switching

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.
- Switching time is dependent on hardware support.

Process Context-switching



Process Context-switching, Step by Step

1. Save CPU context, including PC and registers (the only step needed in a simple mode switch)
2. Update process state (to "Ready", "Blocked", etc.) and other related fields of the PCB.
3. Move the PCB to the appropriate queue
4. Select another process for execution: This decision is made by the CPU scheduling algorithm of the OS.
5. Update the PCB of the selected process (change state to "Running").
6. Update memory management structures
7. Restore CPU context to the values contained in the new PCB

Process Context-switching

What events trigger the OS to switch processes?

- Interrupts: External, asynchronous events, independent of the currently executing process instructions.
  - **Clock interrupt:** OS checks time and may block process
  - **I/O interrupt:** Data has come, OS may unblock process
  - **Memory fault:** OS may block process that must wait for a missing page in memory to be swapped in.
- Exceptions (traps): Internal, synchronous (but involuntary) events caused by instructions: OS may terminate or recover process.
- System calls (traps!): Voluntary synchronous events calling a specific OS service: After service completed, OS may either resume or block the calling process, depending on I/O, priorities, etc.

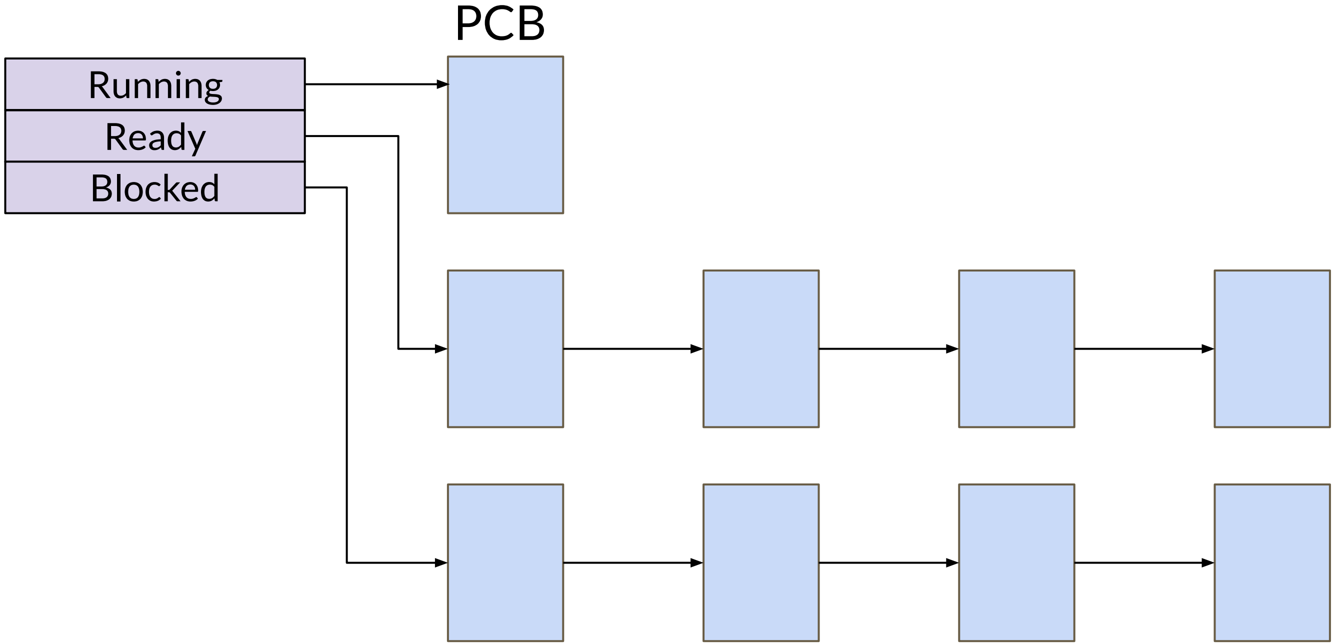
Process Queues

- Job queue: Set of all jobs in the system
- Ready queue: Set of all processes residing in main memory, ready and waiting to execute
- Device queues: Set of processes waiting for an I/O device

Processes migrate among the various queues

Process Queues

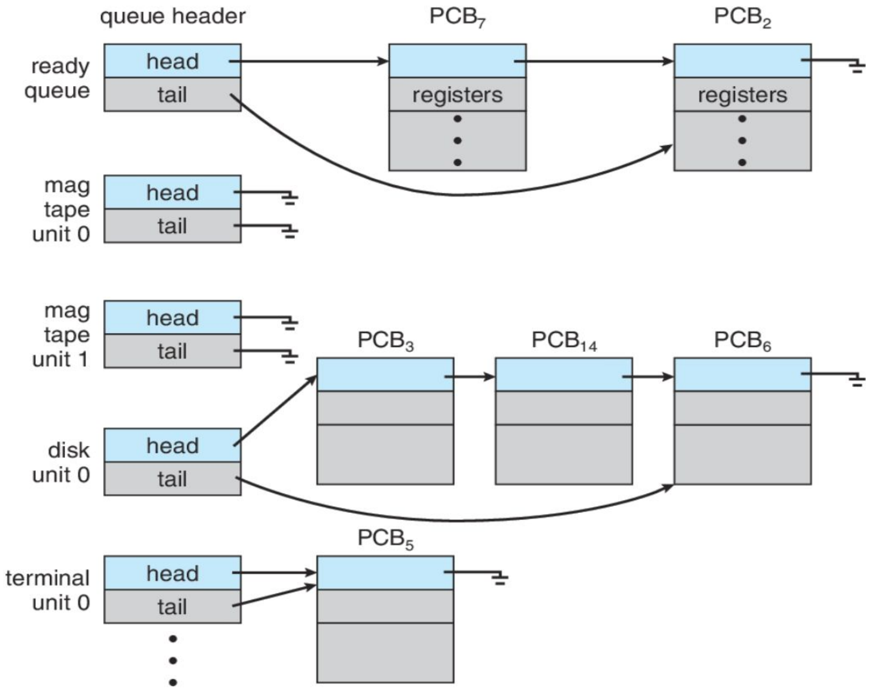
The process table can be split into per-state queues: PCBs can be linked together if they contain a pointer field



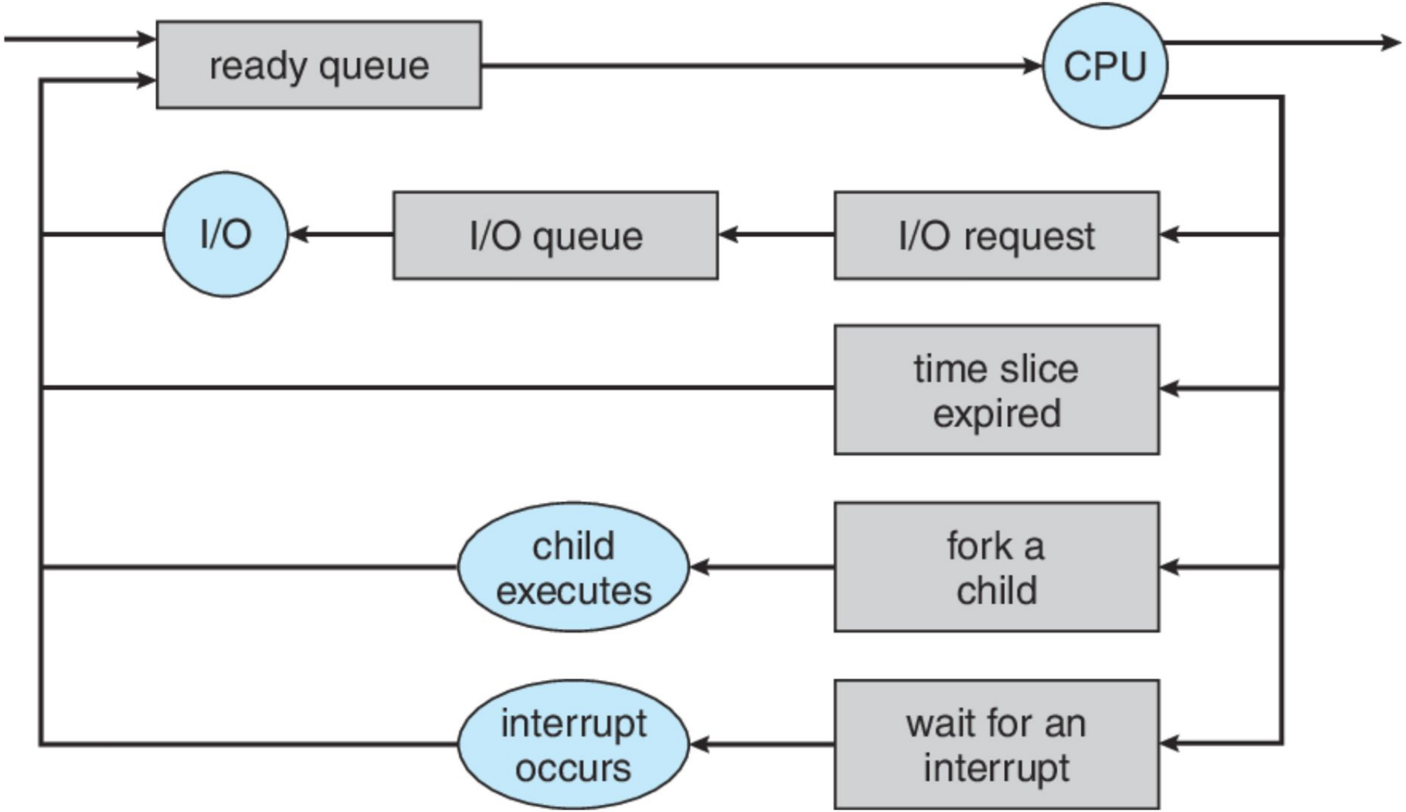
Process Queues

Ready Queue and various I/O device queues.

I/O scheduling: The decision to handle a process's pending I/O request



Representation of Process Scheduling

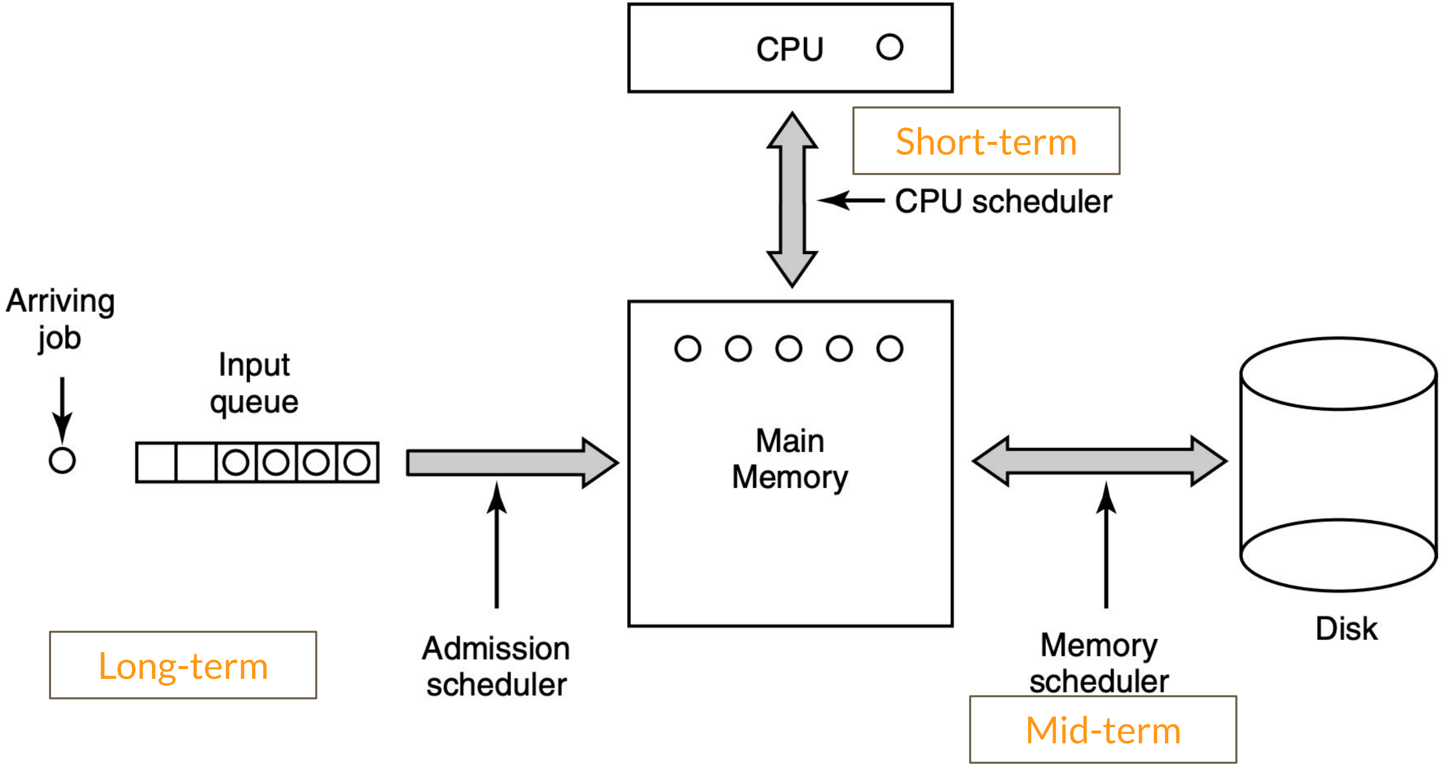


OS Scheduling

- Algorithm depends on the execution environment
  - All systems
  - Batch systems
  - Interactive systems
  - Real—time systems

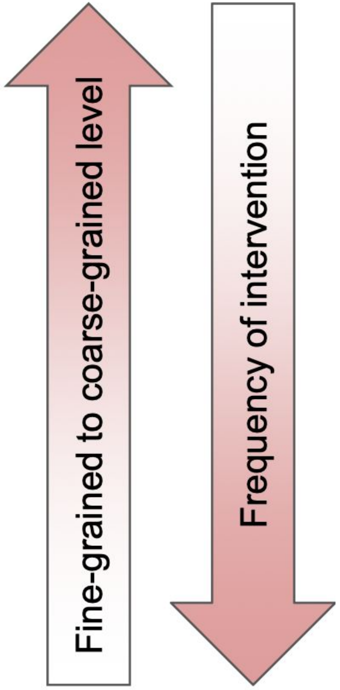
OS Scheduling

Three Level Process Scheduling



OS Scheduling

- Long-term scheduling
  - The decision to add a program to executed (job scheduling)
- Medium-term scheduling
  - The decision to add to the number of processes that are partially or fully in main memory (“swapping”)
- Short-term scheduling (CPU scheduling)
  - The decision as to which available processes in memory are to be executed by the processor (“dispatching”)

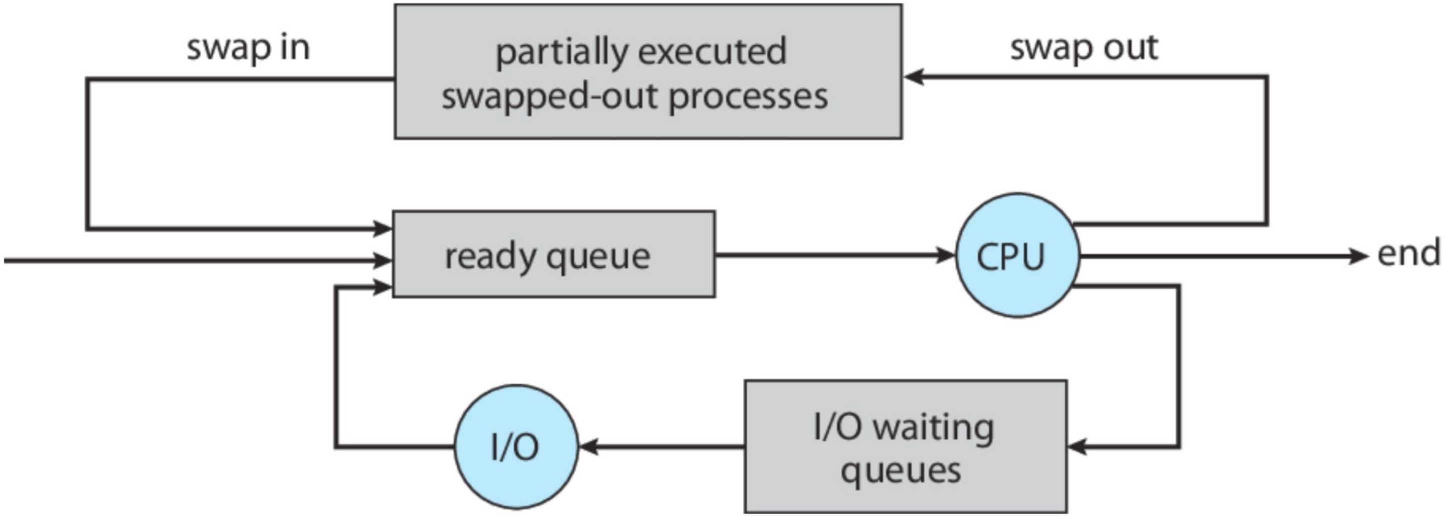


Schedulers

- Short-term scheduler is invoked very frequently ***(milliseconds) ⇒ (must be fast)***
- Long-term scheduler is invoked very infrequently ***(seconds, minutes) ⇒ (may be slow)***
- The long-term scheduler controls the degree of *multiprogramming*
- Processes can be described as either:
  - I/O-bound process: Spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process: Spends more time doing computations; few very long CPU bursts
- Long-term schedulers need to make careful decision

Addition of Medium Term Scheduling

- In time-sharing systems: remove processes from memory “temporarily” to reduce degree of multiprogramming.
- Later, these processes are resumed → Swapping



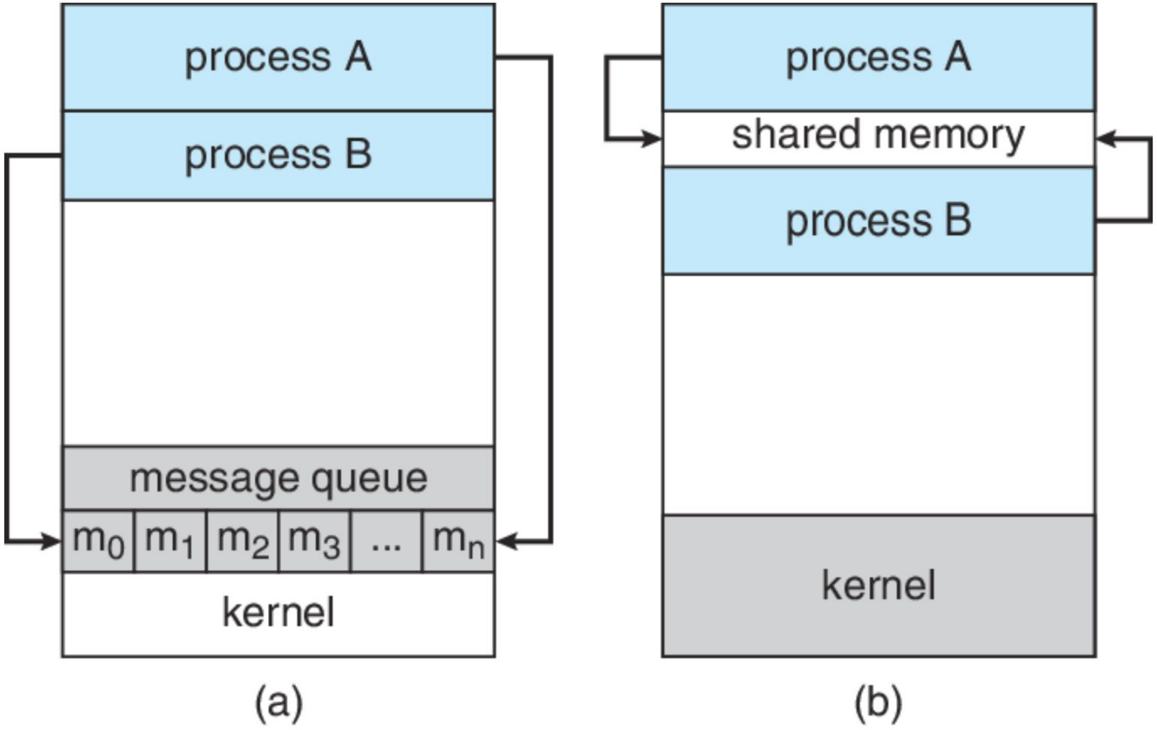
Cooperating Processes

- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by the execution of another process.
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Disadvantage
  - Synchronization issues and race conditions

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Shared Memory: by using the same address space and shared variables
- Message Passing: processes communicate with each other without resorting to shared variables

Communications Models



Message Passing

Shared Memory

Message Passing

- Message passing facility provides two operations:
  - send (*message*) – message size fixed or variable
  - receive (*message*)
- If P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive
- Two types of message passing
  - **Direct** communication
  - **Indirect** communication

Message Passing - Direct Communication

- Processes must name each other explicitly:
    - send (*P*, *message*) – send a message to process P
    - receive (*Q*, *message*) – receive a message from process Q
- Properties of communication link:
    - Links are established automatically
    - A link is associated with exactly one pair of communicating processes
    - Between each pair there exists exactly one link
    - The link may be unidirectional, but is usually bi-directional
- Symmetrical vs Asymmetrical direct communication:
    - send (*P*, *message*) – send a message to process P
    - receive (*id*, *message*) – receive a message from any process
- Disadvantage of both: limited modularity, hardcoded

Message Passing - Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Primitives are defined as:
  - send (*A*, *message*) – send a message to mailbox A
  - receive (*A*, *message*) – receive a message from mailbox A

Message Passing - Indirect Communication

- Operations
  - Create a new mailbox
  - Send and receive messages through mailbox
  - Destroy a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Message Passing - Indirect Communication

- Mailbox sharing
  - *P1* , *P2* , and *P3* share mailbox A
  - *P1* , *sends; P2* and *P3* receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a "*receive*" operation
  - Allow the system to select the receiver arbitrarily. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
    - Blocking send: The sender blocks until the message is received
    - Blocking receive: The receiver blocks until a message is available
- Non-blocking is considered asynchronous
    - Non-blocking send: The sender sends the message and continues
    - Non-blocking receive: The receiver receives a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways:
  - Zero capacity – 0 messages

    Sender must wait for receiver
  - Bounded capacity – finite length of ***n*** messages

    Sender must wait if link full
  - Unbounded capacity – infinite length

    Sender never waits

Recap

- Processes
  - Process representation in OS
  - Process creation
  - Process termination
  - Context switching
  - Process queues
  - Process scheduling
  - Interprocess communication

Acknowledgements

- "Operating Systems Concepts" book and supplementary material by A. Silberschatz, P. Galvin and G. Gagne
- "Operating Systems: Internals and Design Principles" book and supplementary material by W. Stallings
- "Modern Operating Systems" book and supplementary material by A. Tanenbaum
- R. Doursat and M. Yuksel from University of Nevada, Reno
- Farshad Ghanei from Illinois Tech
- T. Kosar and K. Dantu from University at Buffalo