Processes

DSC 315: Computer Organization & Operating Systems

Dr. Laltu Sardar

School of Data Science,
Indian Institute of Science Education and Research Thiruvananthapuram (IISER TVM)

March 09 & 11, 2026

Table of Content

Recommended Reading From textbook: Chapter 3.1-3.6

IISER THIRUVANANTHAPURAM

3.1

Process Concept

Process Concept

- An operating system executes programs, each running as a **process**
- **Process** – a program in execution
  - Execution proceeds in a sequential manner
  - Instructions of a single process do not execute in parallel
- A process consists of several components
  - **Text section** – program code
  - **Current activity** – program counter and CPU registers
  - **Stack** – temporary data (parameters, return addresses, local variables)
  - **Data section** – global variables
  - **Heap** – dynamically allocated memory during execution

IISER THIRUVANANTHAPURAM

Process Concept (Cont.)

- **Program vs Process**
  - Program – passive entity stored on disk (executable file)
  - Process – active entity executing in memory
- A program becomes a **process** when its executable is loaded into memory
- Program execution can start through
  - GUI actions (mouse clicks)
  - Command line execution
- A single program may create **multiple processes**
  - Example: multiple users running the same program

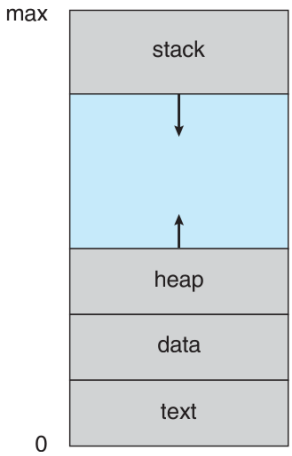


Figure: Process in Memory

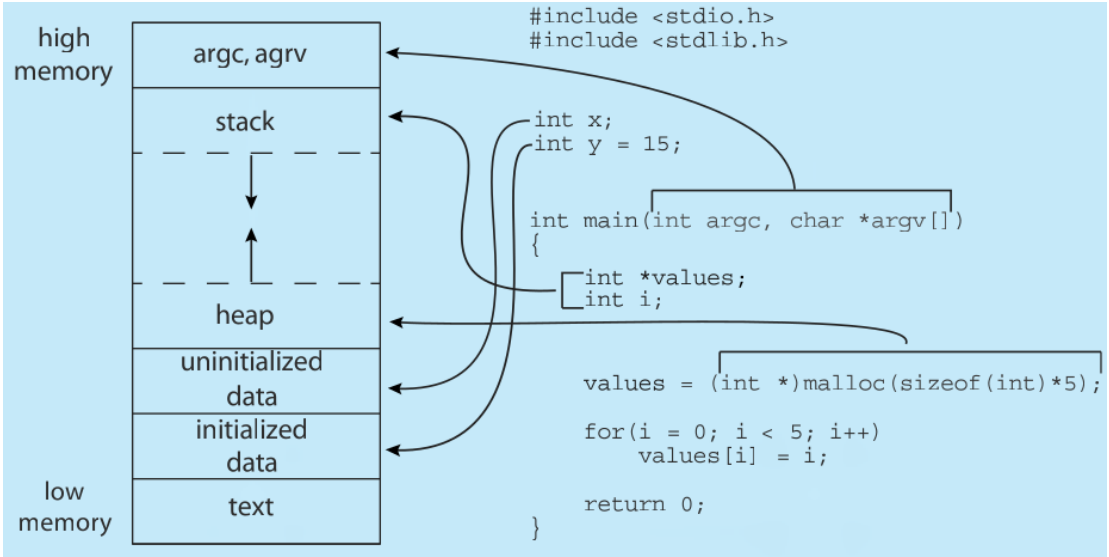


Figure: Memory Layout of a C Program

Process State

- As a process executes, it moves through different **states**
  - **New** – process is being created
  - **Ready** – process is waiting to be assigned to the CPU
  - **Running** – instructions are currently executing
  - **Waiting** – process waits for an event (for example I/O completion)
  - **Terminated** – process has finished execution

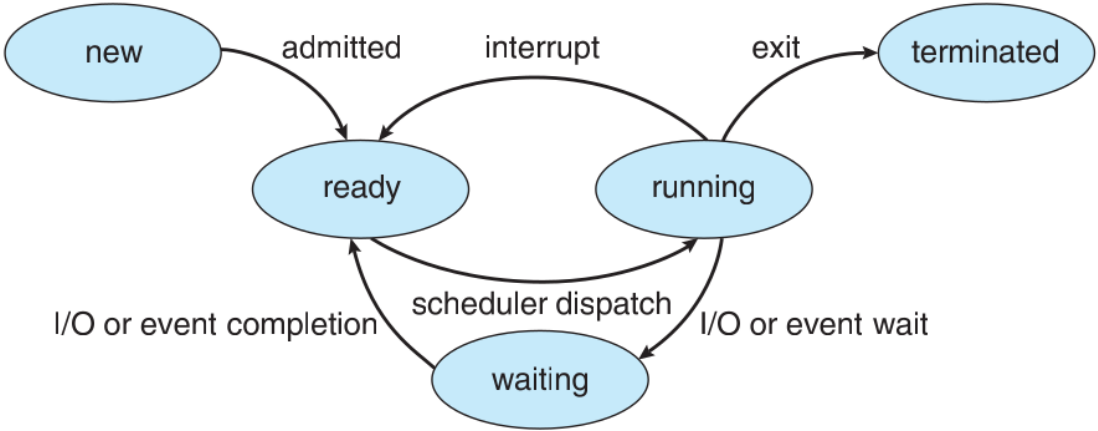


Figure: Diagram of process state.

Process Control Block

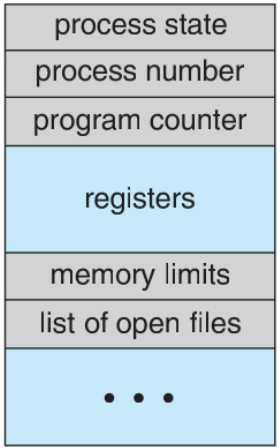


Figure: Process control block (PCB).

- **Process Control Block (PCB)** stores information associated with each process (also called **Task Control Block**)
- **Process state** – current state (running, waiting, ready, etc.)
- **Program counter** – address of the next instruction to execute
- **CPU registers** – values of registers used by the process
- **CPU scheduling information** – priority, scheduling queue pointers
- **Memory management information** – memory allocated to the process
- **Accounting information** – CPU usage, elapsed time, limits
- **I/O status information** – devices allocated and list of open files

PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

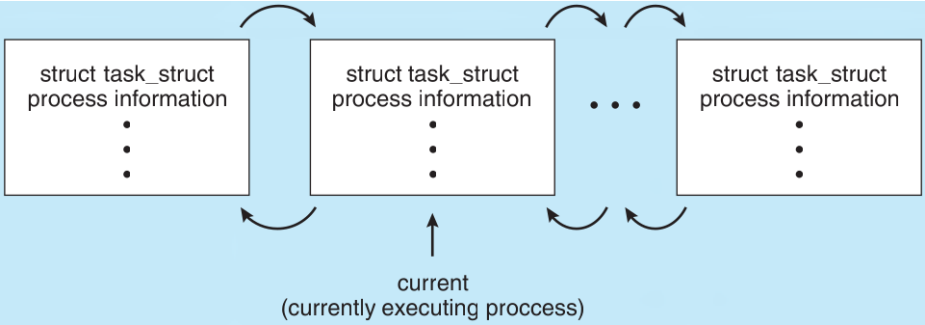
Threads

- Earlier process model assumed a **single thread of execution**
  - One program counter
  - One sequence of instructions
- Modern systems allow **multiple threads within a process**
  - Multiple program counters
  - Different parts of the program execute concurrently
  - These independent flows are called **threads**
- **Example**
  - In a word processor:
    - One thread handles user input
    - Another thread runs spell checking
- **Advantages**
  - Perform multiple tasks simultaneously
  - Efficient use of **multicore processors**
- Operating systems maintain **thread information** inside the PCB
  - Storage for multiple program counters and thread data

IISER THIRUVANANTHAPURAM

Process Representation in Linux

```
long state;                     /* state of the process */
struct sched_entity se;         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */
struct files_struct *files;     /* list of open files */
struct mm_struct *mm;           /* address space */
```



3.2

Process Scheduling

Process Scheduling

- **Process Scheduler**
  - Selects the next process to execute on the CPU
  - Goal: maximize CPU utilization and switch processes quickly
- OS maintains **scheduling queues**
- **Ready Queue**
  - Processes in main memory that are ready to execute
  - Waiting for CPU allocation
- **Wait Queues**
  - Processes waiting for events such as I/O completion
- Processes **move between queues** during execution

IISER THIRUVANANTHAPURAM

Queueing Process

- Mainly two types of queues– **Ready** & **Wait**
- **circles** represent the **resources** that serve the queues
- **arrows** indicate the **flow of processes** in the system.
- Each Queue is a **Doubly linked list**

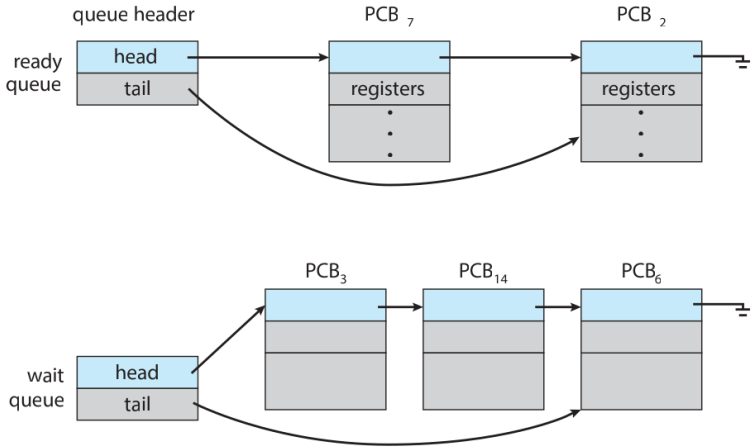


Figure: The ready queue and wait queues.

Queueing Process

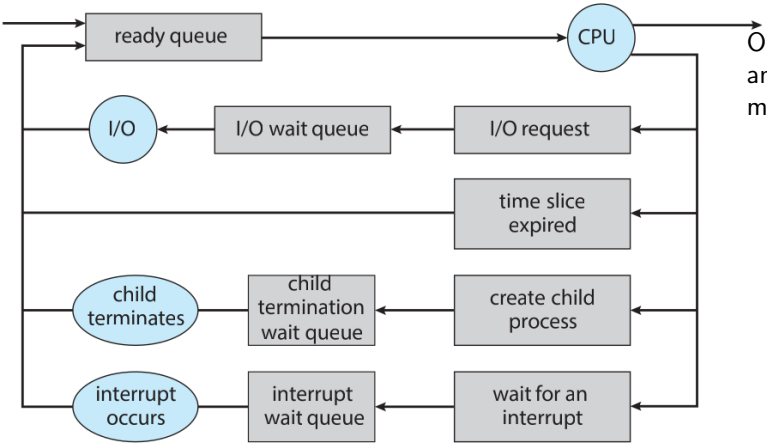


Figure: Queueing-diagram representation of process scheduling.

Once a process is assigned a CPU and starts executing, several events may occur

- Process issues an **I/O request** and moves to the **I/O wait queue**
- Process **creates a child process** and waits for the child to terminate
- Process is **preempted** (interrupt or time slice expiry) and moved back to the **ready queue**

Context Switch

- **Context switch** occurs when the CPU switches from one process to another
- The system must
  - Save the state of the current process
  - Load the saved state of the next process
- The **process context** is stored in the **Process Control Block (PCB)**
- **Context-switch time** is overhead
  - No useful computation occurs during the switch
  - More complex OS and PCB lead to longer context-switch time
- Time required depends on **hardware support**
  - Some CPUs provide multiple register sets, allowing faster switching

IISER THIRUVANANTHAPURAM

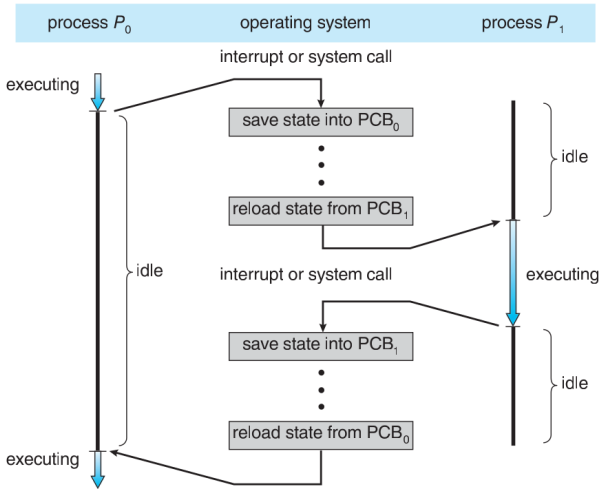


Figure: Diagram showing context switch from process to process.

3.3

Operations in Processes

Operations on Processes

- Operating systems perform several **operations on processes**
- The two main categories are:
  - **Process creation** – creating new processes to execute programs
  - **Process termination** – ending processes and releasing resources
- Processes are often created when
  - A user starts a program
  - A running process creates a child process
  - The operating system starts background services
- When a process terminates
  - The OS releases its allocated resources
  - Control returns to the parent process or operating system

IISER THIRUVANANTHAPURAM

Process Creation

- **Parent processes** create **child processes**, forming a **process tree**
- Each process is identified using a unique **Process Identifier (PID)**
- **Resource sharing options**
  - Parent and children share all resources
  - Children share a subset of the parent's resources
  - Parent and child share no resources
- **Execution options**
  - Parent and children execute concurrently
  - Parent waits until children terminate

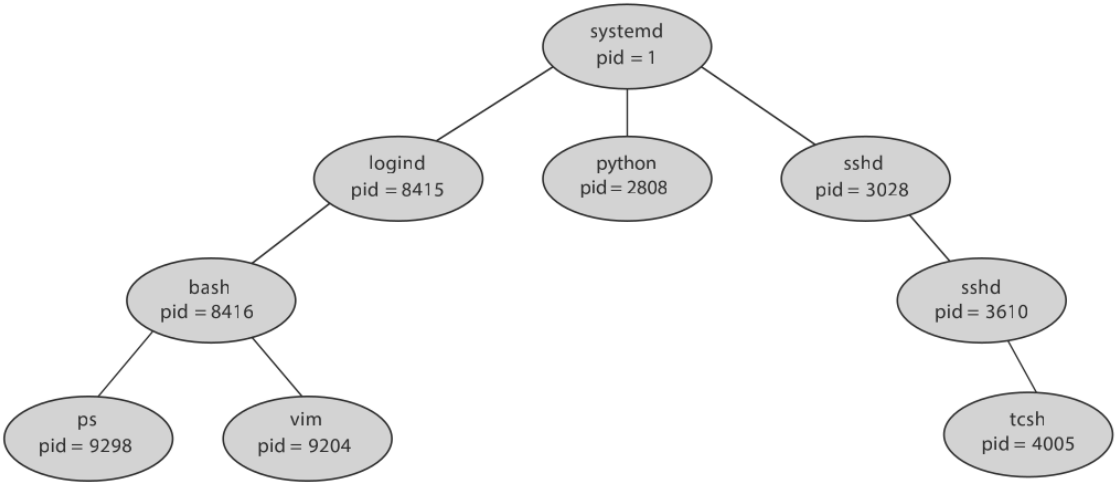


Figure: A tree of processes on a typical Linux system.

Typical Linux Process Tree

- In Linux, processes are often called **tasks**
- The **systemd** process (PID = 1) is the **root parent process**
  - First user process created during system boot
  - Parent of all other user processes
- Example child processes of **systemd**
  - **logind** – manages users logging directly into the system
  - **sshd** – manages remote logins using SSH
- Example user processes
  - **bash** shell (command-line interface)
  - Programs started by bash such as **ps** and **vim**
- Useful Linux commands
  - `ps -el` – lists detailed information about active processes
  - `pstree` – displays the hierarchical process tree

IISER THIRUVANANTHAPURAM

Resource Sharing in Process Creation

- When a **parent process creates a child process**, the child needs resources
  - CPU time, memory, files, I/O devices
- Resources can be obtained in two ways
  - Directly from the **operating system**
  - From a **subset of the parent's resources**
- Parent may
  - Partition resources among children
  - Share resources such as memory or files
- Restricting resources prevents excessive process creation
- Parent may pass **initialization data** to the child

Execution and Address Space Options

- When a process creates a new process, two **execution possibilities** exist
  - Parent and child execute **concurrently**
  - Parent **waits** until some or all children terminate
- Two **address-space possibilities**
  - Child is a **duplicate of the parent** (same program and data)
  - Child **loads a new program** into its address space

Process Creation in UNIX

- **Address space options**
  - Child is a duplicate of the parent
  - Child may load a new program into its memory
- **UNIX system calls**
  - **fork()** – creates a new child process by duplicating the parent
  - **exec()** – replaces the current process memory with a new program
  - **wait()** – parent waits for a child process to terminate and collects its status

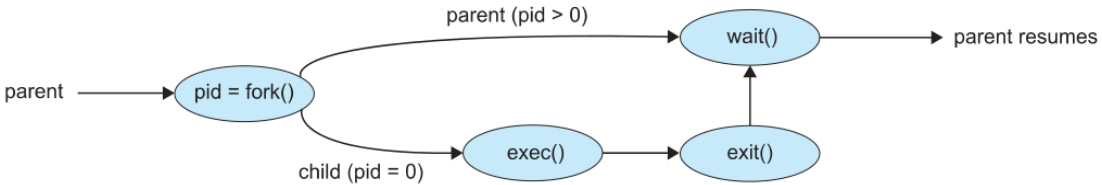


Figure: Process creation using the fork() system call.

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Figure: Creating a separate process using the UNIX fork() system call.

Process Termination

- A process finishes execution and calls **exit()** to terminate
  - Returns status information to the parent via **wait()**
  - Operating system releases all allocated resources
- A parent process may terminate a child using **abort()**
  Possible reasons:
  - Child exceeded allocated resources
  - Task assigned to the child is no longer required
  - Parent process is terminating

Process Termination (Cont.)

- Some systems enforce **cascading termination**
  - When a parent terminates, all children are also terminated
  - Initiated by the operating system
- Parent can wait for child completion using **wait()**
  - Returns child PID and termination status
  - `pid = wait(&status);`
- **Zombie process**
  - Child finished but parent has not called **wait()**
- **Orphan process**
  - Parent terminates without waiting for the child

IISER THIRUVANANTHAPURAM

Zombie and Orphan Processes

- **Zombie Process**
  - Child process has finished execution
  - Parent has not yet called **wait()** to read its exit status
  - Process entry remains in the process table
- **Orphan Process**
  - Parent process terminates before the child finishes
  - Child process is adopted by the **init/systemd** process
  - Continues execution normally

IISER THIRUVANANTHAPURAM

3.4

Inter-process Communication

Cooperating Processes and IPC

- Processes in a system may be **independent** or **cooperating**
- **Cooperating processes**
  - Can affect or be affected by other processes
  - May share data with each other
- **Reasons for cooperation**
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes require **Interprocess Communication (IPC)**
- **Two IPC models**
  - Shared memory
  - Message passing

IISER THIRUVANANTHAPURAM

3.4

IPC Shared memory system

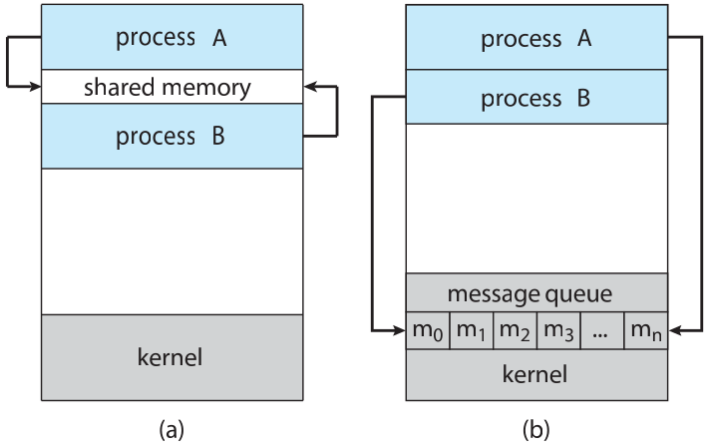


Figure: Communications models. (a) Shared memory. (b) Message passing.

IPC in Shared-Memory Systems

- Normally, the operating system prevents one process from accessing another process's memory
- **Shared memory** allows two or more processes to access a common memory region
- Processes exchange information by **reading and writing** in the shared area

- The data format and memory location are **determined by the processes**, **not by the OS**
- Processes must handle **synchronization**
  - Ensure they do not write to the same memory location simultaneously

IISER THIRUVANANTHAPURAM

Producer-Consumer Problem

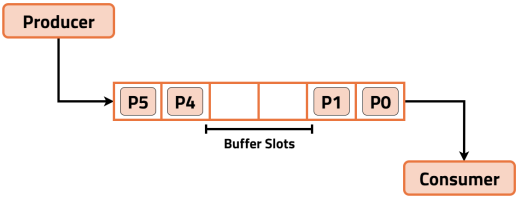


Figure: Bounded Buffer[a]

[a] link to the source

- Classic example of **cooperating processes**
- **Producer** generates data and places it in a buffer
- **Consumer** removes and uses the produced data
- **Two variations**
  - **Unbounded Buffer**
    - No practical limit on buffer size
    - Producer never waits
    - Consumer waits if buffer is empty
  - **Bounded Buffer**
    - Fixed buffer size
    - Producer waits if buffer is full
    - Consumer waits if buffer is empty

Key Issue

Main Problem: Buffer overflow or underflow
Solution: Synchronization with `shared memory`

Example of IPC with Shared Memory

- Implemented with shared array
- Array is a circular. Why?

```
#define BUFFER_SIZE 10

typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Producer process

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Consumer Process

3.4

IPC Message passing system

Example of IPC with message passing

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

Producer process

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

Consumer Process

Follow the problems discussed in the class.

References


Avi Silberschatz, Peter Baer Galvin, and Greg Gagne.
Operating system concepts, tenth edition: Lecture slides.
https://os-book.com/OS10/slide-dir/index.html, 2018.
Accessed: 2026-03-06.


Galvin P. B. Gagne G. Silberschatz, A.
Operating System Concepts.
Wiley, global edition, 2023.

Dr. Laltu Sardar, Assistant Professor, IISER Thiruvananthapuram
laltu.sardar@iisertvm.ac.in, laltu.sardar.crypto@gmail.com
Course webpage: https://laltu-sardar.github.io/courses/corgos_2026.html.