Process Scheduling, Process Synchronization, and Deadlock will be discussed further in Chapters 5, 6, and 7, respectively.

**1. Process Concept**

**1.1 What is a Process?**

A process is a program in execution. Processes exist in main memory.

![Diagram of process components](figure4.1)

- **Stack Section**: Local variables, function/procedure parameters, and return address
- **Dynamic Data Section**: malloc() or calloc() calls
- **Static Data Section**: Global variables and arrays
- **Code Section**: Sequence of instructions
- **PC (Program Counter)**: Next instruction

Note that a single text program on HDD can be loaded into several processes. For example, multi-users can execute mail program; the process of a program including a “fork()” system call will have child process at run time.

**1.2 Type of Process**

A process may be:

- Either OS process (e.g., system call) or User process
- Either I/O bound process or CPU bound process
  - I/O bound processes: Issue lots of I/O requests and little computation
  - CPU bound processes: Use CPU frequently and I/O infrequently
- Either Independent process or Cooperating process
Independent processes

- Does not affect the execution of other processes
- Is not affected by other processes
- Does not share any data with other processes

Cooperative processes (e.g., producer/consumer example)

- Can affect the execution of other processes
- Can be affected by other processes
- Share data with other processes

Either Foreground process or Background process

- Foreground processes:
  - Hold the terminal.
  - Can receive input and return output from/to the user

- Background processes
  - Detached from the terminal it was started
  - Runs without user interaction.
  - The “&” sign will allow you to execute a process as a background process in UNIX.
  - Run “netscape” and “netscape &” and observe the difference.

2. Process Scheduling

- Multiprogramming: Switch the CPU among processes so efficiently that the CPU is always running

- Time-sharing: Switch the CPU among processes so frequently that users can interact with their programs while they are running

2.1 Process State

![Process State Diagram](image)

- New process has “ready” state
- Ready state: The process is waiting to be assigned to the CPU
- Running state: An instruction of the process is being executed by the CPU
• Waiting (blocked) state: The process is waiting for an event (e.g., an I/O completion, termination of a child process, a signal/message from another process).

2.2 PCB (Process Control Block)

Each process is represented by a PCB in the OS.

• Process state
• PID (Process ID)
• PC (Program counter)
• Contents of the processor’s registers
• Memory limits
• List of I/O devices allocated to the process
• List of open files
• etc. (e.g., priority of the process, amount of CPU time used, time limits, …)

2.3 Scheduling Queues

• Ready queue: a list of PCBs. Each PCB represents a “ready” process
• Several device queues: a list of PCBs. Each PCB represents a “waiting” process who is waiting for an I/O completion.
  o Examples: Tape#1 queue, Tape#2 queue, FDD queue, HDD queue, Printer queue, Terminal queue, etc
• Several waiting queues: a list of PCBs. Each PCB represents a “waiting” process who is waiting for an event (e.g., termination of a child process or reception of a signal/message)

Note that a process exists in only one of the above queues at a time.

2.4 Long-term, Short-term, and Medium-term Schedulers

• Long-term scheduler:
  o Job pool: In multiprogrammed batch systems, there are often more processes submitted than can be executed immediately. These processes are spooled to the HDD (secondary storage).
  o Long-term scheduler selects processes from the job pool and fills the ready queue.
    ▪ Executes infrequently ➔ can take more time to select a process
    ▪ Mainly used in Batch systems
    ▪ Controls the number of processes in main memory\(^1\)

\(^1\) When the number of processes in the main memory is always the same, the long-term scheduler will need to be invoked only when a process is terminated to select a process from the job pool.
- Select a good mix of I/O bound processes and CPU bound processes to maximize both the I/O devices utilization and the CPU utilization

- **Short-term scheduler** (CPU scheduler):
  - Executes very frequently \( \rightarrow \) must be very fast
  - Select a process from the ready queue when the current process releases the CPU

- **Medium-term scheduler**:
  - Swap in/out\(^3\) partially executed processes to temporarily free up main memory by reducing the degree of multiprogramming (i.e., the number of processes in main memory).

We may need this scheduler to improve the mix of I/O bound processes and CPU bound processes in main memory or to react to an increase in dynamic memory requests (e.g., malloc())

---

2.5 Context Switching
- Occurs when the current process releases the CPU
- Procedure:
  - The current process releases the CPU voluntarily or involuntarily

---

\(^2\) Some scheduling techniques force the current process to release the CPU (i.e., Preemptive scheduling)

\(^3\) Swap out: remove a process from the main memory and store the image on HDD for later scheduling. Swap in: load a swapped-out process into the main memory and construct (or update) the PCB.
o OS updates the PCB of the current process with the current information (e.g., PC, contents of CPU registers, etc.)
o Short-term scheduler selects the next process from the ready queue
o OS load CPU registers with new data from the PCB of the next process to be executed
o Start the instruction pointed to by the PC

- Frequently occurs and expensive \( \rightarrow \) decrease the system performance
- Some solutions
  - Multiple sets of registers
  - Special HW instruction (e.g., a single instruction to load or store all registers)
  - Threads whenever possible (e.g., A single process includes both producer and consumer – Will be discussed in Chapter 6 in much more detail)

### 3. Process Creation and Process Termination

Linux/UNIX System Calls “fork”, “execv”, “wait”, “abort/exit(1)/exit(2)/exit(3)/kill”, “getpid”, and “getppid”

#### 3.1 Process Creation

<table>
<thead>
<tr>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stack (copy)</td>
</tr>
<tr>
<td>Data</td>
<td>Data (copy)</td>
</tr>
<tr>
<td>Code Section (Shared read-only)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4 The UNIX “fork” operation

- If you use “wait” system call, the parent waits until some or all of its children terminate
- If you don’t use “wait” system call, the parent continues to execute concurrently with its children

- If you use only “fork” call, the child process is a duplicate of the parent process

• If you use “fork” in the parent and “execv” in the child, the child process will have a program loaded into it.

Note that a parent can call “fork” several times to create more children. Child process can also call “fork” to a hierarchy of processes

```
<table>
<thead>
<tr>
<th>Parent</th>
<th>Child</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Stack of P2</td>
</tr>
<tr>
<td>Data</td>
<td>Data of P2</td>
</tr>
<tr>
<td>Code Section</td>
<td>Code of P2</td>
</tr>
</tbody>
</table>
```

Figure 4.5 After the child calls “execv” to load P2, the stack, data, and code of the child are overwritten.

### 3.2 Process Termination

- In UNIX, a process terminates when it calls “exit” (Normal termination) or “abort” (abnormal termination).
- “exit” terminates the calling process immediately.
  - Every open file will be closed.
  - All the resources will be freed.
  - The children of the process are inherited by process 1 (i.e., “init” process).
- Parent can terminate any of its children with “kill(child PID, SIGKILL)” operation for a variety of reasons.
- Note that a child can terminate its parent with “kill(getppid(), SIGKILL)” operation.
4. Threads

Linux System Call “clone” and Linux/Unix System calls “pthread_create/pthread_exit/pthread_join/pthread_detach/pthread_attr_init”

- **TASK**: a set of peer threads.

- **PEER THREADS**: share code section, data section, and OS resources such as table of file descriptors and table of signal handlers. Each thread has its own stack and PC (see Figure 4.6) → Good for the multi-processor systems in which the processors share main memory.

- **USER THREAD**: By calling special library functions, we can develop multiple threads of controls in our program that run concurrently (see Figure 4.6).
  - **Fast switching**: Switching among peer threads does not incur an interrupt to the Kernel (no short-term scheduling, no address space change, …). Only PC & stack-address switch among threads is required)
    - Sometimes, we can modify multiple processes sharing some memory space to a task of multi-threads → Faster, partially solve the context switch problem
  - **Limitation**: (CPU scheduling units are not threads but tasks):
    - When thread 1 of task 1 waits for I/O completion or a signal, the scheduler switches the CPU to another task. Thus, all peer threads in task 1 will be blocked because the task 1 releases the CPU and gets into a device (or waiting) queue.
    - Scheduler allocates same amount of time to 100-thread task and 2-thread task.
    - Motivation for Multi Thread Kernel
Figure 4.6 An example of 3-thread task: a user program, which consists of three subsections that can run concurrently, can be a task that consists of three threads. (In a Linux, type “man clone” and read the man page.) Note, peer threads can read/write any other thread’s stack.

- **Multi-thread kernel:** Kernel is a task of multiple threads. Therefore,
  - Fair scheduling: solve the limitations of user threads
  - Increased Kernel utilization: While one kernel thread is waiting for an I/O completion, the kernel can accept another request.

- **Hybrid Approach:** Both user threads and kernel threads are implemented (e.g., Solaris 2)
5. Inter-Process Communication (IPC)

Linux/UNIX System Calls “pipe/read/write”, “shmget/shmat/shmct1”, “socket/socketspair”

5.1 Introduction

- Cooperative processes: Why do we need them?
  - Information sharing
  - Computation speedup for multi-processor systems
  - Modularity
  - Convenience
- OS should provide Cooperative processes with communication mechanisms so that they can communicate with each other. (At least read-write or send-receive)
- Example of Cooperative processes: Producer-Consumer problem
  - Unbounded buffer or bounded buffer
  - Buffer can be provided by OS (i.e., message-passing) e.g., pipe()
  - Buffer can be programmed in the cooperative processes (i.e., shared memory) e.g., shmget()

5.2 Direct Communication

- Both sender and receiver must name each other. (Symmetric addressing)
  - Process Q: send (P, message)
  - Process P: receive (Q, message)
- Only the sender names the recipient. (Asymmetric addressing)
  - Process Q: send (P, message)
  - Process P: receive (id, message)
- Pros: provide some degree of protection
- Cons: Limited modularity (changing name of P)

5.3 Indirect Communication

- Messages are sent and received to/from ports (or mailboxes)
- Each port has a unique ID
- Process P: send (Port ID, message)
- Process Q: receive (Port ID, message)
- Mailbox can belongs to OS or programmed in cooperative processes

5.4 Buffering

- Port has Zero capacity:
  - Synchronization is necessary: Sender wait for receiver to send a message. When both of them are ready (i.e., Rendezvous), the sender can send a message.
- Port has Bounded capacity:
  - N messages can be stored in Port temporary. Note, N is the same as the port size.
  - Sender does not wait when the port is not full
- When the port is full, sender wait until one message in the port is consumed

- Port has Unbounded capacity:
  - Sender never has to wait for receiver

### 5.5 Issues with IPC

These issues apply to both message passing and shared memory schemes

- Sender or receiver has been terminated and the other one is waiting for a message from the terminated one. Receiver has been terminated and the sender sent a message
  - Best solution: One of the error return value of the send/receive function is “the sender/receiver has been terminated”
  - So-So: OS terminates the other one.
  - Worst solution: OS does not care about it.

- Message was lost
  - Best: Its OS responsibility
  - So-so: OS detect it and tells the sender about it.
  - Worst: Its sender’s responsibility.

- Message was damaged
  - Parity check, …