Chapter 3
Processes

Process Concept

- Computers can do several activities at a time
  - Executing user programs,
  - reading from disks
  - writing to a printer, etc.
- In **multiprogramming:**

  CPU switches from program to program. CPU still runs only one program at a time but gives an illusion of *parallelism*, called **pseudo parallelism or virtual CPU**
- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**

Textbook uses the terms ***job*** and ***process*** almost interchangeably

Process Concept

- **File** – A passive entity, A named sequence of bytes
- **Process** – a program in execution; process execution must progress in sequential fashion
  - An active entity
  - Needs certain resources such as CPU, memory, I/O
  - Resources are allocated either at the beginning or during execution on demand.
  - At any given point in time, System consists of:
    - System processes which execute systems code and the User processes that execute user code and all of them exist concurrently.

Process Concept (Cont.)

- An OS is responsible for following process related activities
  - Process creating and deletion
  - Scheduling processes
  - Provisions of mechanisms for synchronization, communication and deadlock handling.
  - Traditionally a process contained a single thread of controls but most modern OS now supports processes with multiple threads

Process Concept (Cont.)

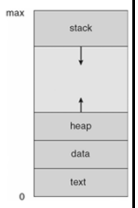
- A process includes:
  - A program counter (PC)
  - CPU registers:

    for data manipulation
  - Stack:

    for function parameters, return address, etc.
  - Data section:

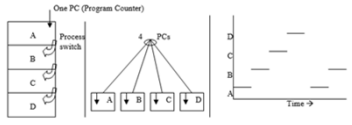
    for global variables
  - Heap:

    for dynamic allocation of memory during run time.
- Two processes may be associated with same program and are considered two execution instances
  - they have separate of PC, stack, heap, etc. even though the text section (i.e., code is same). Example: several users running mail program



Process Concept (Cont.)

- Conceptually, each process has its own **virtual CPU** but in reality, there is only one CPU, *shared* by all processes. Assume that there are 4 programs in memory



- **Abstract View:** each process has its own CPU, running independently
- Over period of time, all processes have made progress but in reality, at any given point in time, **only one** process is active.

Process Concept (Cont.)

Important: with switching back and forth,

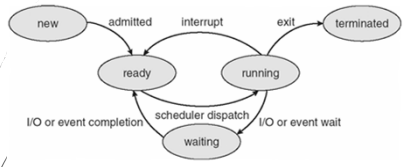
- Rate at which a process performs computation may not be uniform.
- Probably the time may not be reproducible even if same processes are run again.

Therefore, no assumption should be made about built-in timings.

Generally, most processes are not affected by underlying multiprogramming of CPU or relative speed of other processes

Process State

- As a process executes, it changes **state**



- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur (I/O completion or reception of a signal
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

Process Control Block (PCB)

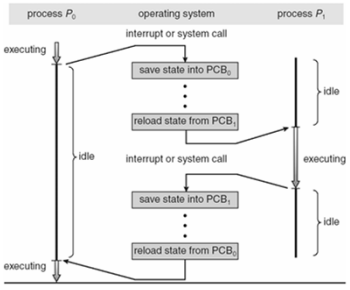
Information associated with each process

(also called **process/task control block or PCB**)

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files
- Pointer - point to the PCB of another process in the list

Pointer	Process state
Process # (PID)	
Program Counter	
Registers	
Stack Pointer	
Memory limits	
List of open files	
Process start time	
CPU time	
:	
:	

CPU Switch From Process to Process



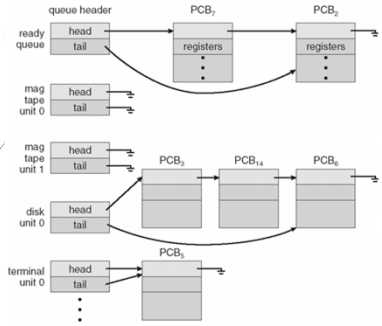
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- Will talk more about it in next chapter

Process Scheduling

- **Uni-process system:**
  - No problem since there is only one process
- **Multi-process system:**
  - Process waits until CPU is free (either because of I/O or time quantum expiration)Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes 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
- During its life, a processes migrate among various queues

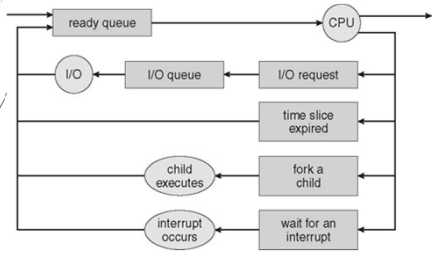
Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

Queueing diagram represents queues, resources and flows

- Rectangles: Queue
- Circles: Resources that serve queues

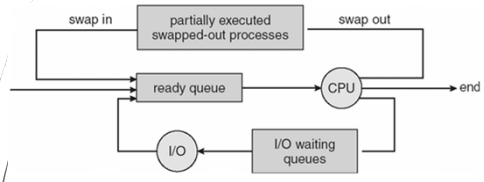


Schedulers

- **Short-term scheduler** (or **CPU scheduler or dispatcher**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked 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 scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

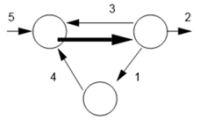
- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Advantage: may improve job mix, free up limited memory

Scheduling (contd.)

- When does a scheduler executes?



1. a process switches from running to waiting
   → select new process for CPU
2. a process terminates
   → select new process for CPU
3. a process switches from running to ready
   → select new process for CPU
4. a process switches from waiting to ready
5. a new process

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - **Foreground:**
    - A single process - controlled via user interface
  - **Background:**
    - multiple processes – in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Context Switch

- 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 via a **context switch**
- **Context** of a process is represented in the PCB
- Context-switch time is a pure overhead; CPU is busy switching between processes, not doing any 'real work' → leads to bottle neck
  - The more complex the OS and the PCB → the longer the context switch
- Time is dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

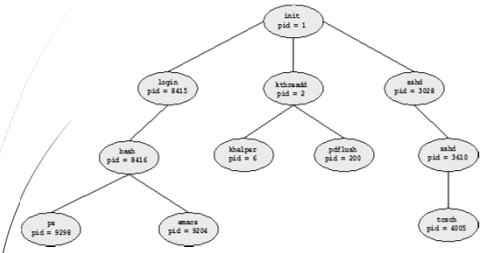
Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process execution
  - process termination,
  - ...

Process Creation

- During execution, a process may create several new processes, via a **create-process** system call.
  - The creating process is called a **Parent** process,
  - the new processes are called the **children** of that process.
- Each of these new processes may in turn create other processes, forming a **tree** of processes
- In most OS including UNIX and Windows family, Processes are identified by a unique id, called **Process ID (PID).**

A Tree of Processes in Linux



Process Creation (Cont.)

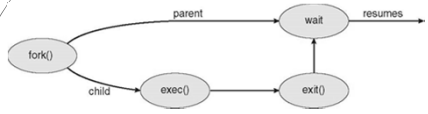
Every process needs resources (e.g., files, pointer, memory, CPU time, etc.). 3 possible scenarios

1. Parent and children share all resources
2. Children share only a subset of parent's resources
3. Parent and children share no resources. In this case, a child process may ask for its own resources from OS
   - It can get only a subset of the parent's resources.
   - **Why**?

     Prevents processes from overloading system

Process Creation (Cont.)

- UNIX examples
  - A new process is created using **fork()** system call
  - Parent and child share same address space
  - Both continue after **fork()** system call.
  - A return code: **fork()** == 0 for child and 1 for parent.
  - **execv()** system call is used by one of the two processes to replace the process's memory space with a new program.



Process Execution

Execution: 2 possibilities

1. Parent continues to execute concurrently with its children
2. Parent waits until some or all of its children are terminated

Process Termination

Process can termination either normally or abnormally

- How a process can terminate?

1. By itself
   - A process terminates itself after execution of last statement by asking OS to delete it by using the exit() system call.
   - Resources are de-allocated.
2. A parent can terminate a child process (via **abort()** system call)
3. **WHY**?
   - A child has exceeded its usage of some of the resources allocated to it
   - The task assigned to child is no longer required
   - The parent is exiting and the OS doesn't allow a child to continue if its parent terminates (e.g., VMS)
     - **Cascaded termination:** All children, grandchildren, etc. are terminated.
     - The termination is initiated by the operating system.

Process Termination (contd.)

3. The parent process may wait for termination of a child process by using the **wait()** system call**.** The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

   - If a process is terminating and its parent is not waiting for its termination (i.e., did not invoke **`wait()`** yet) process is called a **zombie**
     - All processes transition to this state when they terminate but generally they exist as zombies **ONLY** briefly
   - If parent terminated without invoking **`wait()`** , process is an **orphan**

- **UNIX**
  - Child terminates by **exit()** system call
  - **wait()** returns the process identifier of the child process so parent can identify its child
  - If parent terminates, children are assigned a **new parent**
    - the '**init**' process.

Multiprocess Architecture – Chrome Browser

- Many web browsers use to run as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multi-process with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, JavaScript. A new renderer is created for each website opened
    - Runs in **sandbox,** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Interprocess Communication

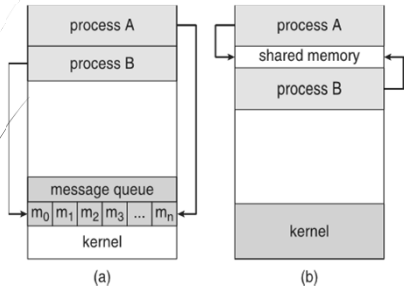
- Will talk about it when we start chapter on **Process Synchronization** (Chapter 5)

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



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

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size
- **Bounded-Buffer – Shared-Memory Solution**
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded-Buffer – Producer/Consumer Code

Producer code

```
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;
}
```

Consumer code

```
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 */
}
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

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

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
- 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

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 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 arbitrarily the receiver. 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 is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization (Cont.)

- Producer-consumer becomes trivial
  - Produce code

```
message next_produced;
while (true) {
    /* produce an item in next produced */
send(next_produced);
}
```

- Produce code

```
message next_consumed;
while (true) {
   receive(next_consumed);
/* consume the item in next consumed */
}
```

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits