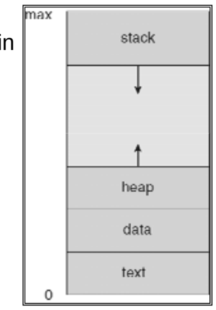
Chapter 3: Process Concept

- The concept of Process
- Process Scheduling
- Operations on Processes
- Inter-process communication (IPC)

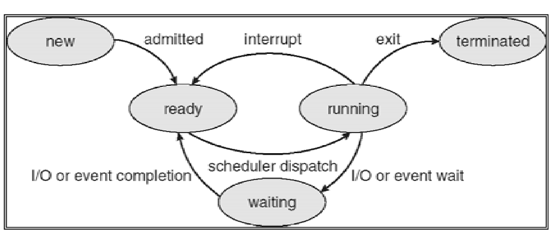
Process Concept

- Textbooks use the terms *job* and *process* almost interchangeably
- Process – an instance of a program in execution;
  - Multiple instances of the same program are different processes
- A process is more than the program code(text section)
- A process also includes:
  - program counter
  - stack
  - data section
  - Heap
- Each process identified by a unique, positive integer id (process id)



Process State

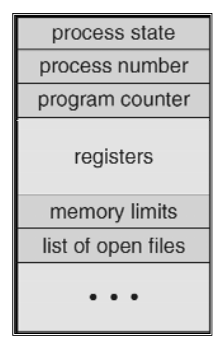
- As a process executes, it changes its ***state***
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a CPU
  - **terminated**: The process has finished execution



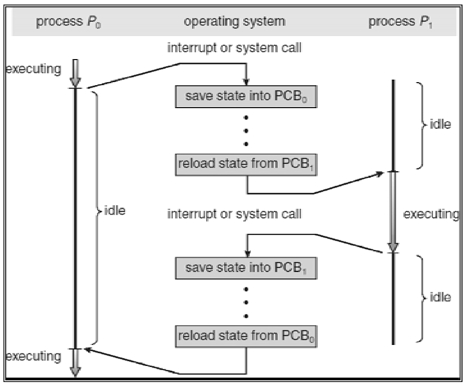
Process Control Block (PCB)

Each process is represented in the OS by a PCB. Information associated with each process includes:

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information (“process environment”)



CPU Switch From Process to Process

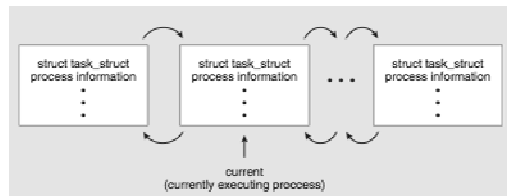


Process Representation in Linux

PCB in the Linux OS is Represented by the C structure `task_struct`

```
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* 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 of this pro */
```

Doubly linked list



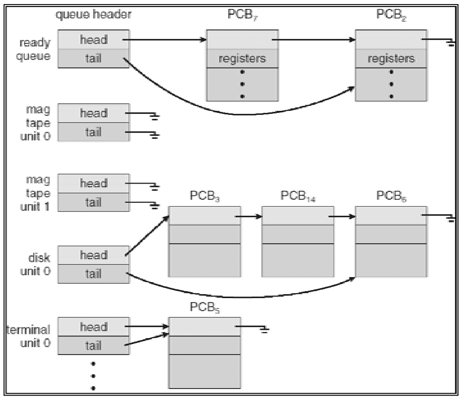
Threads

- A Thread, sometimes called lightweight process
- Thread are used for small tasks, whereas processes are used for more 'heavyweight' tasks – basically the execution of applications
- Both processes and threads are independent sequences of execution.
- Threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.
- Single thread allows the process to perform one task at a time
- Modern OSs have extended the process concept to allow a process to have multiple threads of execution to perform more than one task at a time

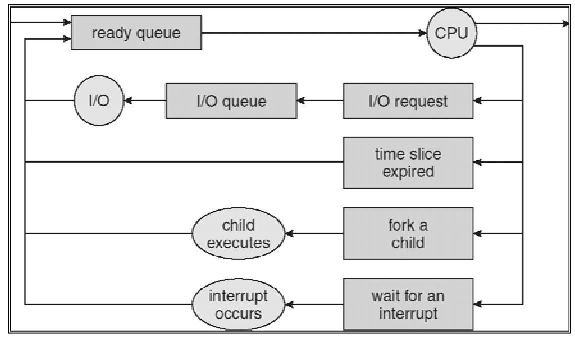
Process Scheduling Queues

- **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
- Processes migrate among the various queues

Ready Queue and Various I/O Device Queues



Simplified Model of Process Scheduling

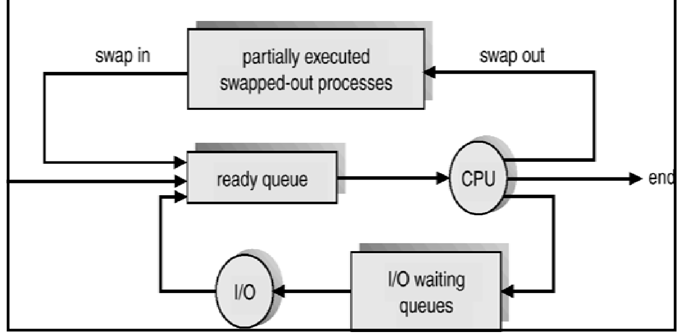


Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the *degree of multiprogramming*
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU for that process
  - Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations
  - **CPU-bound process** – spends more time doing computations

Schedulers

- **Medium-term scheduler** : To remove processes from memory and later, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called swapping



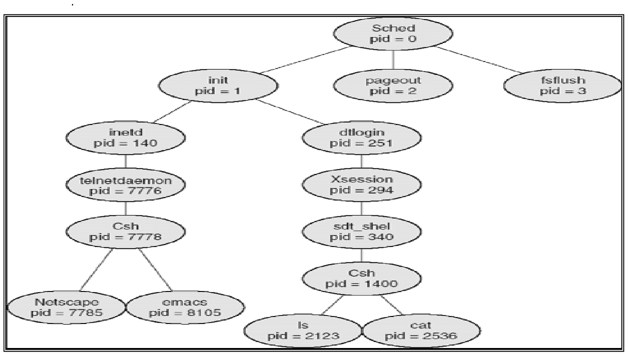
Context Switch

- 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 represented in the PCB
- Context-switch time is *overhead*; the system does no useful work while switching
- Switch time varies from system to system – dependent on hardware support(memory speed, registers)

Process Creation

- Parent process create children processes
  - Children, in turn create other processes, forming a tree of processes
- Resource sharing possibilities
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution possibilities
  - Parent and children can execute concurrently, or
  - Parent can wait until some or all its children terminate
- Memory address space possibilities
  - Address space of child duplicate of parent
  - Child has a new program loaded into it
- POSIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Process Creation Illustrated



A Tree of processes on a typical Solaris System

Process Creation

- A process can create another process by invoking fork() system call
  - The process that invokes the fork() is known as the **parent** and the new process is called the **child**
- NOTE: The fork system call does not take an argument.

A Simple program of fork()

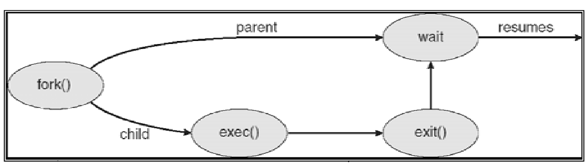
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    printf("Hello \n");
    fork();
    printf("bye\n");
    return 0;
}
```

A summary of fork() return values follows:
fork_return > 0: this is the parent
fork_return == 0: this is the child
fork_return == -1: fork() failed and there is no child

Creating a Separate Process Using fork() system call

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t  pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
        printf("Fork Failed");
        exit(-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");
        exit(0);
   }
}
```


Process creation using fork() system call



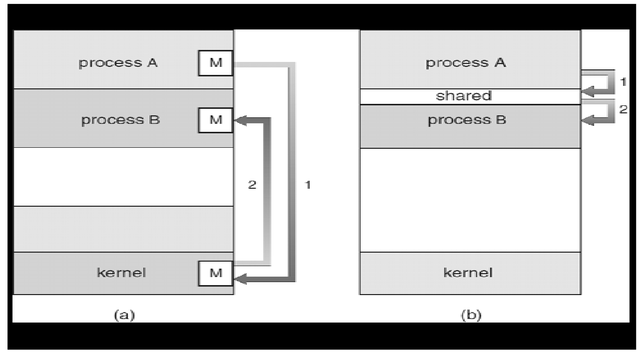
Process Termination

- A process terminates when it finishes executing its last statement and asks the operating system to terminate it using exit() system call
- Process encounters a fatal error
  - Can be for many reasons like arithmetic exception etc.
- Parent may terminate execution of children processes using kill() system call. *Some possible reasons*
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
- Parent is exiting
  - Some operating systems may not allow child to continue if its parent terminates

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

Interprocess Communication (IPC)



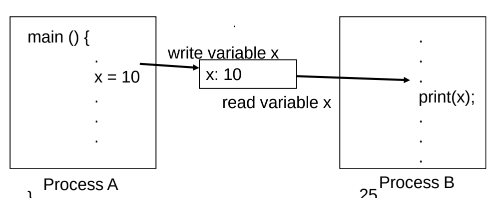
Two models of IPC: (a) Message passing (b) Shared memory

IPC

Message Passing	**Shared Memory**
Easier to implement, particularly useful in distributed environment	Not easy to implement
Not faster	Faster
Implemented using series of system calls and more time consuming	Systems calls are required only to establish shared-memory regions
No concurrent execution	Processes runs concurrently

Shared Memory IPC

- Processes share a common piece of memory either physically or virtually
- Communications via normal reads/writes



25

Shared memory

- **Eg: Producer Consumer Problem**

- Producer produces items, consumer consumes item
  - Unbounded Buffer
  - Bounded Buffer

- Shared buffer implemented as a circular array with two logical pointers in and out

```
#define BUFFER_SIZE 10
typedef  struct {
          . . .
} item;
item buffer[BUFFER_SIZE];
 int in = 0;  //next free position
int out = 0;  //first full position
```


Code for Consumer Process

```
item nextProduced;
while (true) {
  /* Produce an item in nextProduced  */
      while (((in + 1) % BUFFER SIZE)   ==  out)
              ;   /* do nothing */
     buffer[in] = nextProduced;
    in = (in + 1) % BUFFER SIZE;
   }
```

The producer process

Code Consumer Process

```
item nextConsumed;
while (true) {
        while (in == out)
             ; // do nothing -- nothing to consume

              // remove an item from the buffer
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER SIZE;
       /* consume the item in nextConsumed  */
   }
```

The consumer process

Message-Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- A message passing system 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
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Message Passing

■ Methods for implementing the communication link and the *send()*/*receive()* operations:

1. Direct or Indirect communication (*Naming*)
2. Synchronous or Asynchronous communication
3. Automatic or Explicit buffering

Process A	*Process B*
while (TRUE) {	while (TRUE) {
produce an item	receive (A, item)
send (B, item)	consume item
}	}

Direct Communication

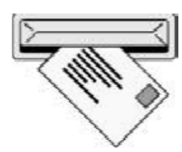
- Direct Communication
  - Processes must name each other explicitly:
    - **Symmetric Addressing**
      - **send** (*P, message*) – send a message to process P
      - **receive**(*Q, message*) – receive a message from process Q
    - **Asymmetric addressing**
      - *send (**P**, message)* – send to process P
      - *receive(**id**, message)* – rx from any; system sets id = sender
  - Properties of communication link
    - Links are established automatically between pairs
    - A link is associated with exactly two processes
    - Between each pair of processes, there exists exactly one link
    - The link may be unidirectional, but is usually bi-directional
- **Disadvantage**: a process must know the name or ID of the process(es) it wishes to communicate with

Indirect Communication

- Indirect Communication
  - Messages are directed and received from *mailboxes* (also referred to as *ports*)
    - Each mailbox has a unique *id* and is created by the kernel on request
    - 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
  - send and receive messages 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?

- Possible 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 also known as synchronous and asynchronous
- **Blocking** is considered **synchronous**
  - **Blocking send:** the sender blocks until the message is received by the other party
  - **Blocking receive:** the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send:** the sender sends the message and continues executing
  - **Non-blocking receive:** the receiver gets either a valid message or a null message (when nothing has been sent to the receiver)
- Often a combination:
  - Non-blocking send and blocking receive

Link Capacity – Buffering

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