Chapter 3: Processes
Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems
Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation

- To describe the various features of processes, including scheduling, creation and termination, and communication

- To describe communication in client-server systems
Process Concept

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 – a program in execution; process execution must progress in sequential fashion.

A process includes:
- program counter
- stack
- data section
The Process

- Multiple parts
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
    - Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory

- Execution of program started via GUI mouse clicks, command line entry of its name, etc

- One program can be several processes
  - Consider multiple users executing the same program
Process in Memory

- stack
- heap
- data
- text
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
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution
Diagram of Process State

- **new**
- **admitted**
- **interrupt**
- **exit**
- **terminated**

- **ready**
  - I/O or event completion
  - scheduler dispatch

- **running**
  - I/O or event wait

- **waiting**
Process Control Block (PCB)

Information associated with each process
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information
Process Control Block (PCB)

- process state
- process number
- program counter
- registers
- memory limits
- list of open files
  - 
  - 
  -
CPU Switch From Process to Process

- Process $P_0$ is executing.
- An interrupt or system call occurs.
- The operating system saves the state into PCB$_0$.
- The operating system is idle.
- Process $P_1$ becomes executing.
- Process $P_1$ is interrupted or makes a system call.
- The operating system saves the state into PCB$_1$.
- The operating system is idle.
- Process $P_0$ reloads its state from PCB$_0$.
- Process $P_0$ becomes executing again.
Process Scheduling

- 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
  - Processes migrate among the various queues
Process Representation in Linux

- Represented by the C structure `task_struct`:
  ```c
ten 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 process */
```
Ready Queue And Various I/O Device Queues
Representation of Process Scheduling

- ready queue
- CPU
- I/O
- I/O request
- time slice expired
- fork a child
- wait for an interrupt
- child executes
- interrupt occurs
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
Schedulers (Cont.)

- 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
Addition of Medium Term Scheduling
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
  - The more complex the OS and the PCB -> longer the context switch

- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes.

- Generally, process identified and managed via a **process identifier** (pid).

- Resource sharing:
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources

- Execution:
  - Parent and children execute concurrently
  - Parent waits until children terminate
Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

- UNIX 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

fork() -> child

parent -> wait

wait -> exec()

exec() -> exit()

resumes
C Program Forking Separate Process

```c
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    /* fork another 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 */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```
A Tree of Processes on Solaris

- Sched (pid = 0)
  - init (pid = 1)
    - inetd (pid = 140)
    - telnetdaemon (pid = 7776)
    - Csh (pid = 7778)
      - Netscape (pid = 7785)
      - emacs (pid = 8105)
    - ls (pid = 2123)
  - pageout (pid = 2)
  - fsflush (pid = 3)
    - dtlogin (pid = 251)
    - Xsession (pid = 294)
      - sd_3l (pid = 340)
      - Csh (pid = 1400)
      - cat (pid = 2536)
Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait)
  - Process' resources are deallocated by operating system

- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - cascading termination
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)

(b)
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;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements
while (true) {
    /* Produce an item */
    while (((in = (in + 1) % BUFFER SIZE count) == out) ; /* do nothing -- no free buffers */
        buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
}
Bounded Buffer – Consumer

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

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
}

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)` – 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)
Implementation Questions

- 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?
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
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:
  - send\((A, \text{message})\) – send a message to mailbox A
  - receive\((A, \text{message})\) – 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** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available

- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null
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
Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    \[ \text{segment id} = \text{shmget}(\text{IPC PRIVATE}, \text{size}, \text{S_IRUSR | S_IWUSR}); \]
  - Process wanting access to that shared memory must attach to it
    \[ \text{shared memory} = (\text{char} *) \text{shmat(id, NULL, 0);} \]
  - Now the process could write to the shared memory
    \[ \text{sprintf(shared memory, "Writing to shared memory");} \]
  - When done a process can detach the shared memory from its address space
    \[ \text{shmdt(shared memory);} \]
Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation - Kernel and Notify
  - Only three system calls needed for message transfer
    msg_send(), msg_receive(), msg_rpc()
  - Mailboxes needed for communication, created via
    port_allocate()
Examples of IPC Systems – Windows XP

- Message-passing centric via local procedure call (LPC) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - The client opens a handle to the subsystem’s connection port object.
    - The client sends a connection request.
    - The server creates two private communication ports and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.
Local Procedure Calls in Windows XP

- **Client**
  - Connection request
  - Handle

- **Connection Port**
  - Handle

- **Client Communication Port**

- **Server Communication Port**
  - Handle

- **Server**

- **Shared Section Object (≤ 256 bytes)**
Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)
Sockets

- A **socket** is defined as an *endpoint for communication*

- Concatenation of IP address and port

- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**

- Communication consists between a pair of sockets
Socket Communication

host $X$
(146.86.5.20)

socket
(146.86.5.20:1625)

web server
(161.25.19.8)

socket
(161.25.19.8:80)
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.

- Stubs – client-side proxy for the actual procedure on the server.

- The client-side stub locates the server and marshalls the parameters.

- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.
Execution of RPC

<table>
<thead>
<tr>
<th>client</th>
<th>messages</th>
<th>server</th>
</tr>
</thead>
<tbody>
<tr>
<td>user calls kernel to send RPC message to procedure X</td>
<td>From: client To: server Port: matchmaker Re: address for RPC X</td>
<td>matchmaker receives message, looks up answer</td>
</tr>
<tr>
<td>kernel sends message to matchmaker to find port number</td>
<td>From: server To: client Port: kernel Re: RPC X Port: P</td>
<td>matchmaker replies to client with port P</td>
</tr>
<tr>
<td>kernel places port P in user RPC message</td>
<td>From: client To: server Port: port P &lt;contents&gt;</td>
<td>daemon listening to port P receives message</td>
</tr>
<tr>
<td>kernel sends RPC</td>
<td>From: RPC Port: P To: client Port: kernel &lt;output&gt;</td>
<td>daemon processes request and processes send output</td>
</tr>
<tr>
<td>kernel receives reply, passes it to user</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipes

- Acts as a conduit allowing two processes to communicate

- **Issues**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes?
  - Can the pipes be used over a network?
Ordinary Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style
- Producer writes to one end (the *write-end* of the pipe)
- Consumer reads from the other end (the *read-end* of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes
Ordinary Pipes
Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems
End of Chapter 3