Chapter 4: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems

Process Concept

- An operating system executes a variety of programs:
  - Batch system ≈ jobs
  - Time-shared systems ≈ processes or tasks
- We use the terms job and process interchangeably

- Process
  - A program in execution
  - Process execution must progress in sequential fashion

- A process includes:
  - Code section
  - Data section
  - Program counter
  - Stack

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

Process Control Block

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)

<table>
<thead>
<tr>
<th>pointer</th>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>process number</td>
</tr>
<tr>
<td></td>
<td>program counter</td>
</tr>
<tr>
<td></td>
<td>registers</td>
</tr>
<tr>
<td></td>
<td>memory limits</td>
</tr>
<tr>
<td></td>
<td>list of open files</td>
</tr>
</tbody>
</table>
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

A process moves between the various queues
Schedulers

- Long-term scheduler (or job scheduler):
  - selects which processes should be moved to ready queue

- Short-term scheduler (or CPU scheduler):
  - selects which process should execute next
  - allocates CPU

Also: Medium-term Scheduling
Scheduler Issues

- **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

Scheduler Issues

- 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

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
- Context-switch time is overhead:
  - the system does no useful work while switching
- Time depends on hardware support
Operations on Processes

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing choices:
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution choices:
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space choices:
  - Child duplicate of parent
  - Child has a new 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

Unix example: create new process

```c
void main(int argc, char *argv[]) {
    int pid=fork();
    if (pid < 0) {
        fprintf(stderr, "Fork failed");
        exit(1);
    } else if (pid == 0) { // child process
        execlp("/bin/ls", "ls", NULL);
    } else { // parent process
        wait(NULL);
        printf("Child complete");
        exit(0);
    }
}
```
Process Termination

- Process executes last statement
- or, asks OS for explicit exit
  - return data from child to parent (via wait)
  - process' resources are deallocated by operating system
- Parent may abort execution of children process
  - child has exceeded allocated resources
  - task assigned to child is no longer required
  - parent is exiting
    - OS does not allow child to continue if its parent terminates
      - Term: Cascading termination

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: no practical limit on the size of the buffer
  - bounded-buffer: assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

```c
#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

Bounded-Buffer – Producer Process

```c
item nextProduced;

while (1) {
  while (((in + 1) % BUFFER_SIZE) == out) ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}
```
Bounded-Buffer – Consumer Process

```c
    item nextConsumed;

    while (1) {
        while (in == out) ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    }
```

Inter-process Communication (IPC)

- A 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 operations)

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 \((P, \text{message})\) – send a message to process \(P\)
  - receive\((Q, \text{message})\) – receive a message from process \(Q\)

- Properties of communication link
  - links are established automatically
  - link is associated with exactly one pair of processes
  - between each pair there exists exactly one link
  - 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
  - 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
  - P₁, P₂, and P₃ share mailbox A
  - P₁ sends; P₂ and P₃ 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
  - Non-blocking is considered asynchronous
  - send and receive primitives may be either blocking or non-blocking

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
Example: Mach

- Message-based micro-kernel OS
- Task = process with multiple threads
- Port = mailbox
  - has port rights: receive, send, send-once
  - each task has Kernel and Notify port
- Message
  - data structure: typed collection of objects
  - contains port rights
- Message transfer primitives:
  - msg_send, msg_receive, msg_rpc
- Port set
  - group of ports (with receive right) that share a queue
  - thread can serve a port set

Example: Mach

- Port functionality:
  - allocate a new port with port rights
  - only one process has receive right
  - de-allocate a port
  - includes notification of port rights holders
  - get port status
  - create backup port
  - to receive messages if port is de-allocated
- Message content:
  - destination port
  - reply port
  - message length
  - port rights
  - pointers to parameters
- Message delivery via memory mapping

Example: Windows XP

- LPC: local procedure call
- Provides communication among Windows subsystems
- Port for
  - Connection
  - Communication
- Message passing techniques
  - Message copy (< 256 bytes)
  - Shared memory
  - Direct read/write (only to/from Win32 subsystem)
Client-Server Communication

- Sockets
- Remote Procedure Calls
- 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 occurs between a pair of sockets

Socket Communication
Java Server example

import java.net.*; import java.io.*; import java.util.*;
class Server {
    public static void main(String[] args) {
        try {
            ServerSocket server = new ServerSocket(5155);
            while (true) {
                Socket s = server.accept();
                PrintWriter out = new PrintWriter(s.getOutputStream(), true);
                out.println(new Date().toString());
                out.close(); s.close();
            }
            } catch (IOException ioe) { System.err.println(ioe); } }
    }
}

Java Client example

import java.net.*; import java.io.*;
class Client {
    public static void main(String[] args) {
        try {
            Socket s = new Socket("127.0.0.1", 5155);
            BufferedReader bin = new BufferedReader(
                       new InputStreamReader(s.getInputStream()));
            String line;
            while ((line = bin.readLine()) != null)
                System.out.println(line);
            } catch (IOException ioe) { System.out.println(ioe); }
    }
}

Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- Stubs
  - proxies for the actual procedure
  - client-side stub
    - locates the server
    - marshals the parameters
  - server-side stub
    - receives this message,
    - unpacks the marshaled parameters
    - performs the procedure on the server
Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPC
- RMI allows a Java program on one machine to invoke a method on a remote object

Marshalling Parameters

RMI example
Server Interface

```java
import java.rmi.*;
public interface QuoteServerI extends Remote {
    String getQuote() throws RemoteException;
}
```

Server Implementation (1/2)

```java
import java.rmi.*;
import java.rmi.server.*;
public class QuoteServer extends UnicastRemoteObject implements QuoteServerI {
    public QuoteServer() throws RemoteException {
        super();
    }
    public String getQuote() throws RemoteException {
        return "hello here is your quote";
    }
}
```

Server Implementation (2/2)

```java
public static void main(String[] args) {
    System.setSecurityManager(new RMISecurityManager());
    try {
        Naming.rebind("/quoter", new QuoteServer());
    } catch (Exception x) {
        x.printStackTrace();
    }
}
```
Client Implementation

```java
import java.rmi.*;
public class QuoteClient {
    public static void main(java.lang.String[] args) {
        try {
            QuoteServerI server;
            server = (QuoteServerI) Naming.lookup("quoter");
            System.out.println(server.getQuote());
        } catch (Exception x) {
            x.printStackTrace();
        }
    }
}
```

Steps to run

- Generate stubs and skeleton classes: rmic
- Compile all Java code
- Run rmiregistry
- Provide security policy
- Run QuoteServer
- Run QuoteClient

Run RMI application

- rmiregistry
- java –Djava.security.policy=policy QuoteServer
- java –Djava.security.policy=policy QuoteClient

```
policy example:
grant {
    permission java.security.AllPermission;
};
```