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 – 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
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.
Process Control Block (PCB)

Information associated with each process.
- Process ID
- 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>process number</td>
<td></td>
</tr>
<tr>
<td>program counter</td>
<td></td>
</tr>
<tr>
<td>registers</td>
<td></td>
</tr>
<tr>
<td>memory limits</td>
<td></td>
</tr>
<tr>
<td>list of open files</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></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.
- Processes migrate between the various queues.
Ready Queue And Various I/O Device Queues

Representation of Process Scheduling
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.

Addition of Medium Term Scheduling
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.

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 dependent on hardware support.
Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
- 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
  - `fork` returns 0 to child, process id of child for parent
  - `exec` system call used after a `fork` to replace the process’ memory space with a new program.
Unix Program

```c
#include <stdio.h>
main(int argc, char *argv[])
{
    int pid;
    pid=fork(); /* fork another process */
    if (pid == 0) {
        /* child */
        exclp("/bin/ls","ls",NULL);
    }
    else { /* parent */
        wait(NULL); /* parent waits for child */
        printf("Child complete\n");
        exit(0);
    }
}
```

Processes Tree on a UNIX System
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.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.
  - In Unix, if parent exits children are assigned init as parent.

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;
  ```
- Circular array
- Empty: in == out
- Full: ((in+1)%BUFFER_SIZE) == out
- 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;
}
```
Interprocess Communication (IPC)

- 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) considered later
  - logical (e.g., logical properties) now

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.
- Asymmetric variant
  - receive(id, message) – receive a message from any process, pid stored in id

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, 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; $P_3$ 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*
- **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.

Exercise: Read about Mach and Windows 2000
Mach

- Mach kernel support creation of tasks – similar to processes but with multiple threads of control
- IPC, even system calls, is by messages using mailboxes called ports
- When task created, so are Kernel and Notify mailboxes
  - The kernel communicates via kernel mailbox
  - Events are notified via Notify mailbox
- Three system calls used for message transfer
  - Msg_send, msg_receive, msg_rpc
  - Msg_rpc executes RPC by sending a message and waiting for exactly one return message
- Task creating mailbox using port_allocate owns/receives from it
- Messages from same sender are queued in FIFO order, but no other guarantees given

Mach

- Message headers contain destination mailbox and mailbox for replies
- If mailbox not full the sending thread continues (non-blocking)
- If full the sender can
  - Wait until there is room
  - Wait at most n millisecs
  - Return immediately
  - Cache the message is OS temporarily (one only)
- Receivers can receive from mailbox or mailbox set
- Similar options for receiver
- Can check # of msgs in mailbox with port_status syscall
- Mach avoids performance penalties associated with double copy (to/from mailbox) by using virtual-memory techniques to map message into receiver’s memory
Windows 2000

- W2000 consists of multiple subsystems which appl progs communicate with using communication channels
- W2000 IPC is called local procedure call (LPC)
- W2000 uses connection ports (called objects and visible to all processes) and communication ports
- Objects used to establish communication channels
  - Client opens handle to port object
  - Sends connection request
  - Server creates 2 private comm ports, and returns handle to one
  - Client and server use handles to send/receive messages

Windows 2000

- Three types of message passing:
  - For < 256 bytes, uses message queue as intermediate storage
  - For large messages uses section object (shared memory)
  - This is set up using small message with pointer to section object and size
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 is between a pair of sockets.
Java Sockets

- Java provides 3 types of socket
  - Connection-oriented (TCP) – Socket class
  - Connectionless (UDP) – DatagramSocket class
  - Multicast – MulticastSocket used to send to multiple clients
- Example: Time of day server
  - Clients request time of day from localhost (127.0.0.1)
  - Server listens on port 5155 with accept call
  - Blocks on accept until client request arrives
  - Creates new socket to communicate with client
import java.net.*; import java.io.*;
public class Server
{
 public static void main(String[] args) throws IOException {
    Socket client = null; PrintWriter pout = null; ServerSocket sock=null;
    try{
       sock = new ServerSocket(5155); //now listen for connections
       while(true){
          client = sock.accept();
          pout = new PrintWriter(client.getOutputStream(), true);
          pout.println(new java.util.Date().toString());
          pout.close();
          client.close();
       }
    } catch (IOException ioe) { System.err.println(ioe); }
    finally { if (client != null) client.close();
                if (sock != null) sock.close();
    }
}
}

import java.net.*; import java.io.*;
public class Client
{
 public static void main(String[] args) throws IOException {
    InputStream in = null; BufferedReader bin = null; Socket sock = null;
    try{
       sock = new Socket("127.0.0.1", 5155);
       in = sock.getInputStream();
       bin = new BufferedReader( new InputStreamReader(in));
       String line;
       while( (line = bin.readLine()) != null)
          System.out.println(line);
    } catch (IOException ioe) { System.err.println(ioe); }
    finally { if (sock != null) sock.close();
    }
}
}
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- Messages in RPC are addressed to daemons listening on ports on a remote system.
- **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.
- To avoid data representation problems (bigendian/littleendian) many systems use XDR (external data representation).
- RPC can be used to implement a distributed file system (DFS).

Execution of RPC
Remote Method Invocation

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

Marshalling Parameters

client

val = server.someMethod(A,B)

stub

remote object

boolean someMethod (Object x, Object y)
{
    implementation of someMethod
}

skeleton

A, B, someMethod

boolean return value