Processes

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

```
max
     \---->
      \   
       \ ->
       \--->
    stack

heap

data

text

0
```
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 process
  - **terminated**: The process has finished execution
Diagram of Process State

- new
- ready
- waiting
- running
- terminated

- admitted
- interrupt
- exit
- scheduler dispatch
- I/O or event completion
- I/O or event wait
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)

<table>
<thead>
<tr>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>process state</td>
</tr>
<tr>
<td>process number</td>
</tr>
<tr>
<td>program counter</td>
</tr>
<tr>
<td>registers</td>
</tr>
<tr>
<td>memory limits</td>
</tr>
<tr>
<td>list of open files</td>
</tr>
<tr>
<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
- Process migration between the various queues
Active Processes in Linux

```
struct task_struct
  ...
  ...
  ...
struct task_struct
  ...
  ...
  ...
  ...
struct task_struct
  ...
  ...
  ...

current
(currently executing process)
```
Ready Queue And Various I/O Device Queues
Representation of Process Scheduling

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

- Swap in
- Partially executed swapped-out processes
- Ready queue
- CPU
- I/O
- I/O waiting queues
- Swap out
- End
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 create 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
  - **exec** system call used after a **fork** to replace the process’ memory space with a new program
Process Tree for Solaris system

- **Sched**
  - **init**
    - **inetd**
      - **telnetdaemon**
        - **Csh**
          - **Netscape**
            - **ls**
          - **emacs**
            - **cat**
      - **dtlogin**
        - **Xsession**
          - **sdt_shel**
          - **Csh**
            - **ls**
            - **cat**
  - **pageout**
    - **fsflush**
      - **pid**
        - 0
        - 1
        - 2
        - 3
        - 140
        - 7776
        - 7778
        - 7785
        - 8105
        - 2123
        - 2536
        - 294
        - 340
        - 1400
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
Processes Tree on a UNIX System

root
  └── pagedaemon
  └── swapper
      └── init
          └── user 1
              └── user 2
                  └── user 3
                      └──...

Operating System Concepts 3.23

Silberschatz, Galvin and Gagne ©2005
Process Termination

- Process executes last statement and asks the operating system to decide 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
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
public interface Buffer
{
    // producers call this method
    public abstract void insert(Object item);
    // consumers call this method
    public abstract Object remove();
}
Bounded-Buffer – Shared Memory Solution

```java
import java.util.*;
public class BoundedBuffer implements Buffer {
    private static final int BUFFER_SIZE = 5;
    private int count; // number of items in the buffer
    private int in; // points to the next free position
    private int out; // points to the next full position
    private Object[] buffer;
    public BoundedBuffer() {
        // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
    }
    // producers calls this method
    public void insert(Object item) {
        // Slide 4.24
    }
    // consumers calls this method
    public Object remove() {
        // Figure 4.25
    }
}
```
Bounded-Buffer – Insert() Method

```java
public void insert(Object item) {
    while (count == BUFFER SIZE) {
        // do nothing -- no free buffers
        // add an item to the buffer
        ++count;
        buffer[in] = item;
        in = (in + 1) % BUFFER SIZE;
    }
}
```
Bounded Buffer – Remove() Method

public Object remove() {
    Object item;
    while (count == 0)
        ; // do nothing -- nothing to consume
    // remove an item from the buffer
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
}
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)
  - 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?
Interprocess Communication Models

Message Passing

Shared Memory
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, 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** 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
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
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 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

1. User calls kernel to send RPC message to procedure X
2. Kernel sends message to matchmaker to find port number
3. Matchmaker receives message, looks up answer
4. Matchmaker replies to client with port P
5. Kernel places port P in user RPC message
6. Kernel sends RPC
7. Daemon listening to port P receives message
8. Daemon processes request and processes send output
9. Daemon to client to port P: kernel <output>
10. From: RPC Port: P
11. To: client Port: kernel <output>
12. From: server Port: kernel Re: RPC X Port: P
13. From: client To: server Port: kernel Re: address for RPC X
14. From: client To: server Port: matchmaker
15. Matchmaker receives message, looks up answer
16. Matchmaker replies to client with port P
17. From user to kernel: user calls kernel to send RPC message to procedure X
Local Procedure Calls in Windows XP

Client

Connection request

Connection Port

Handle

Server

Handle

Client Communication Port

Server Communication Port

Handle

Shared Section Object (< = 256 bytes)
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

```
val = server.someMethod(A, B)

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

A, B, someMethod

boolean return value