Operating Systems

Prof. Christophe Bobda
CSCE-Department
University of Arkansas
Chapter 2
Processes
Agenda

• 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
Process Concept

• A process includes:
  • program counter
  • stack
  • data section
Process in Memory

- Stack
- Heap
- Data
- Text

max
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 processor
  • **terminated**: The process has finished execution
Diagram of Process State

new → admitted → ready → interrupt → running → exit → terminated

ready → I/O or event completion → waiting

waiting → scheduler dispatch → running

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
  - ...

© Abraham Silberschatz
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

```
queue header
  ready queue
    head
    tail

mag tape unit 0
  head
  tail

mag tape unit 1
  head
  tail

disk unit 0
  head
  tail

terminal unit 0
  head
  tail

PCB_7
  registers

PCB_2
  registers

PCB_3

PCB_14

PCB_6

PCB_5
```
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
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 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

• Time dependent on hardware support
CPU Switch From Process to Process

- **Process $P_0$**
  - Executing
  - Save state into PCB$_0$
  - Interrupt or system call
  - Reload state from PCB$_1$
  - Idle

- **Operating System**
  - Save state into PCB$_0$
  - Idle

- **Process $P_1$**
  - Executing
  - Save state into PCB$_1$
  - Idle
  - Reload state from PCB$_0$
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
Process Creation (Cont.)

• Execution
  • Parent and children execute concurrently
  • Parent waits until children terminate

• 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 → exec() → exit() → wait → parent → resumes
int main()
{
    pid_t 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);
    }
}
A tree of processes on a typical Solaris

- init pid = 1
  - inetd pid = 140
    - telnetdaemon pid = 7776
      - Csh pid = 7778
        - Netscape pid = 7785
        - emacs pid = 8105
  - dtlogin pid = 251
    - Xsession pid = 294
      - sdt_shel pid = 340
      - Csh pid = 1400
        - ls pid = 2123
        - cat pid = 2536
- pageout pid = 2
- fsflush pid = 3

© Abraham Silberschatz
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**
Interprocess Communication

• 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
Interprocess Communication

• Cooperating processes need interprocess communication (IPC)

• Two models of IPC
  • Shared memory
  • Message passing
Communications Models

(a) process A  
  process B  
  kernel

(b) process A  
  shared  
  process B  
  kernel

© Abraham Silberschatz
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
  
  ```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

while (true) {
    /* Produce an item */
    while (((in = (in + 1) % BUFFER SIZE) == 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;
}

© Abraham Silberschatz
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)`
Interprocess Communication - Message Passing

• 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:
  • \texttt{send}(P, \textit{message}) - send a message to process P
  • \texttt{receive}(Q, \textit{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
Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
  - Process wanting access to that shared memory must attach to it
    shared memory = (char *) shmat(id, NULL, 0);
  - Now the process could write to the shared memory
    sprintf(shared memory, "Writing to shared memory");
  - When done a process can detach the shared memory from its address space
    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

Shared Section Object (≤ 256 bytes)

Server
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
Remote Procedure Calls

- 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>
</tbody>
</table>
| kernel receives reply, passes it to user | }
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