Chapter 3: Processes

肖 卿 俊
办公室：计算机楼212室
电邮：csqjxiao@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/csqjxiao
电话：025-52091022
Chapter 3: Processes

- Process Concept
- Operations and APIs on Processes
- Process Scheduling
- 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.

- Q: Why process, not program? What is a program?

- Process: running program
  - A program is lifeless, the OS makes it running (as a process).
  - A process can be viewed as a running program with machine states.
Loading into Memory: From Program To Process

Virtualizing the Memory

- Operating system
- Job 1
- Job 2
- Job 3
- Job 4

Loading: Takes on-disk program and reads it into the address space of process
Processes
The Process Model

■ Virtualizing the CPU:
   By running one process, then stopping it and running another, and so forth.

■ An Example: Multiprogramming of four programs
   Conceptual model of 4 independent, sequential processes
   Only one program active at any instant

![Diagram of process model](image)
Process Concept (Cont.)

- Process – a program in execution; process execution must progress in sequential fashion.
- The running state of a process includes:
  - Memory
    - Address space: Instructions and data.
  - Registers
    - Program counter (PC) / instruction pointer (IP): current instruction.
    - Stack pointer, frame pointer: management of stack for parameters, local variables, and return addresses.
    - Contents of the processor’s other registers
  - I/O information
    - A list of the files the process currently has open.
Process in Memory

//main.cpp
int a = 0;  
data segment
char *p1;  
data segment
main()
{
    int b;  
    stack
    char s[] = "abc";  
    stack
    char *p2;  
    stack
    char *p3 = "123456";  
    stack
    p1 = (char *)malloc(10);  
    heap
    p2 = (char *)malloc(20);  
    heap
}
Heap is memory that is dynamically allocated during process run time.
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
- scheduled dispatch
- waiting
- running
- interrupt
- terminated
- exit
- I/O or event completion
- I/O or event wait
### Tracing Process State

<table>
<thead>
<tr>
<th>Time</th>
<th>Process(_0)</th>
<th>Process(_1)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Running</td>
<td>Ready</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Running</td>
<td>Ready</td>
<td>Process(_0) initiates I/O</td>
</tr>
<tr>
<td>4</td>
<td>Blocked</td>
<td>Running</td>
<td>Process(_0) is blocked, so Process(_1) runs</td>
</tr>
<tr>
<td>5</td>
<td>Blocked</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Blocked</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Ready</td>
<td>Running</td>
<td>I/O done</td>
</tr>
<tr>
<td>8</td>
<td>Ready</td>
<td>Running</td>
<td>Process(_1) now done</td>
</tr>
<tr>
<td>9</td>
<td>Running</td>
<td>–</td>
<td>Process(_0) now done</td>
</tr>
<tr>
<td>10</td>
<td>Running</td>
<td>–</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.4: Tracing Process State: CPU and I/O**
Discussion

Q1: Draw on the blackboard the Diagram of Process State

Q2: 下列哪一种情况不会引起进程之间的切换？

◆ A. 进程调用本程序中定义的函数进行计算
◆ B. 进程处理I/O请求
◆ C. 进程创建子进程并等待子进程结束
◆ D. 产生中断
Data Structure

- OS is a program, so it has some key data structures that track the state of each process.
  - Process lists for all ready / running / waiting processes

- An example: xv6 kernel
  - Types of information an OS needs to track processes
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                 RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;              // Start of process memory
    uint sz;                // Size of process memory
    char *kstack;           // Bottom of kernel stack
    enum proc_state state; // Process state
    int pid;                // Process ID
    struct proc *parent;   // Parent process
    void *chan;             // If non-zero, sleeping on chan
    int killed;             // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;      // Current directory
    struct context context; // Switch here to run process
    struct trapframe *tf;   // Trap frame for the
                           // current interrupt
};
Process Control Block (PCB)

Information associated with each process.
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- File usage and 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></td>
</tr>
</tbody>
</table>

- Process number
- Program counter
- Registers
- Memory limits
- List of open files
  - 
  -
Context Switch

What is a process context?

The context of a process includes the values of CPU registers, the process state, the program counter, and other memory/file management information.
What is a context switch?

- After the CPU scheduler selects a process (from the ready queue) and before allocates CPU to it, the CPU scheduler must
  - save the context of the currently running process,
  - put it into a queue,
  - load the context of the selected process, and
  - let it run.
CPU Switch From Process to Process

process $P_0$         operating system         process $P_1$
executing
interrupt or system call
save state into PCB$_0$
idle
save state into PCB$_1$
reload state from PCB$_1$
executing
interrupt or system call
save state into PCB$_1$
idle
reload state from PCB$_0$
Context Switch (Cont.)

- 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.
Chapter 3: Processes

- Process Concept
- Operations and APIs on Processes
- Process Scheduling
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems
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.
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 (UNIX)

fork() → parent → wait → resumes

child → exec() → exit()
The fork() System Call

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {         // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

ODD?
The fork() System Call

- The process that is created by using the fork() system call is an (almost) exact copy of the calling process.

Discussion:

what is the output?

```c
int rc = fork();
if (rc < 0) {
    printf("A");
    exit(1);
} else if (rc == 0) {
    printf("B");
} else {
    printf("C");
}
return 0;
```
The fork() System Call

Discussion: What is the output if we add a loop command before the screen print command?

```c
int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        int sum = 0;
        for (int i = 0; i < 100000000; i++)
            sum += i;
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        int sum = 0;
        for (int i = 0; i < 100000000; i++)
            sum += i;
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```
The fork() System Call

Qingjuns-MacBook-Pro:OSC3_code_cpu-api csqjxiao$ ./p1-2
hello world (pid:43349)
hello, I am parent of 43350 (pid:43349)
hello, I am child (pid:43350)
Qingjuns-MacBook-Pro:OSC3_code_cpu-api csqjxiao$ ./p1-2
hello world (pid:43352)
hello, I am child (pid:43353)
hello, I am parent of 43353 (pid:43352)
Qingjuns-MacBook-Pro:OSC3_code_cpu-api csqjxiao$  

- Discussion: why not deterministic?
Processes Tree on Solaris

```
  Sched
     pid = 0

  init
     pid = 1

  pageout
     pid = 2

  fsflush
     pid = 3

  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
```
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 OS.

- 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.
The wait() System Call

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc, (int) getpid());
    }
    return 0;
}
```

parent waits for child process to finish

```bash
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```
The exec() System Call

- The process that is created by using the exec() system call can be a different program.

- Some details in exec()
  - It does not create a new process; rather, it transforms the currently running program into a different running program.
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {  // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {  // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");  // program: "wc" (word count)
        myargs[1] = strdup("p3.c");  // argument: file to count
        myargs[2] = NULL;  // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn’t print out\n");
    } else {  // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (%d)\n",
               rc, wc, (int) getpid());
    }
    return 0;
}
prompt> . /p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
Review

- Process creation APIs
  - `fork()`
  - `wait()`
  - `exec()`

- What are the differences?
Consider the following C program. Guess how many lines of output will be printed.

```c
int main(int argc, char * argv[]) {
    int i, id1, id2;
    for (i = 1; i < 2; i++) {
        id1 = fork();
        id2 = fork();
        if (id1 == 0 || id2 == 0) fork();
    }
    printf("I am %d\n", getpid());
}
```

What if we change the initial value $i=1$ to $i=0$?
Chapter 3: Processes

- Process Concept
- Operations and APIs on Processes
- Process Scheduling
- Cooperating Processes
- Interprocess Communication
- Communication in Client-Server Systems
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.
Ready Queue And Various I/O Device Queues

- Ready Queue
  - head
  - tail

- Magazine Tape Unit 0
  - head
  - tail

- Magazine Tape Unit 1
  - head
  - tail

- Disk Unit 0
  - head
  - tail

- Terminal Unit 0
  - head
  - tail

- PCBs
  - PCB7
  - PCB2
  - PCB3
  - PCB14
  - PCB6
  - PCB5

- Registers
  - :
  - :
  - :
  - :
  - :

Operating System Concepts

Southeast University
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 loaded into memory for execution.

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU.
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 ________ scheduler controls the degree of multiprogramming.
  - long-term
  - short-term
Schedulers (Cont.)

- The long-term scheduler controls the *degree of multiprogramming*.

- Long-term scheduling performs a *gatekeeping function*. It decides whether there's enough memory, or room, to allow new programs into the system.

- Dispatching affects processes:
  - running;
  - ready;
  - blocked;

- Long-term scheduling affects processes:
  - new;
  - exited.
Schedulers (Cont.)

- 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.

- The period of computation between I/O requests is called the **CPU burst**.

![Diagram showing CPU and I/O bursts]

Operating System Concepts
Discussion: If you design a CPU scheduler, which type of processes will you give a higher priority of granting CPU resource? I/O-bound processes, or CPU-bound processes?
The resource needs of a process may vary during its runtime. When the system resources become insufficient, some processes may need to swap out.

- Addition of Medium-Term Scheduling

Chapter 3: Processes

- Process Concept
- Operations and APIs on Processes
- Process Scheduling
- Interprocess Communication
- Communication in Client-Server Systems
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
A Common Cooperating Pattern: 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 – Share-memory Solution

```c
#define BUF_LEN 10

typedef struct {
    ...
} item;

item buffer[BUF_LEN];
int in = 0, out = 0;

Producer Process
item nextProduced;
while (1) {
    while (((in+1)%BUF_LEN) == out)  
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUF_LEN;
}

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

Implementation of Communication Link by Shared Memory

- Frame buffer
- CPU1
- CPU2
- Memory
- System Bus
- I/O Bus
- Bridge
- Contr.
- Contr.
- Contr.
- Contr.
- Storage
- Network
- Producer
- Consumer
Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.

- Message-passing system – processes communicate with each other without resorting to shared variables.
Interprocess Communication (Cont.)

- 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.
  ◆ 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:
  - $\text{send}(A, \text{message})$ – send a message to mailbox $A$
  - $\text{receive}(A, \text{message})$ – 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**
- **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 blocks.
Pipes in Unix

- UNIX pipes are implemented in a similar way, but with the `pipe()` system call.
  
  - The output of one process is connected to an in-kernel pipe.
  - The input of another process is connected to that same pipe.
  
  E.g.,
  
  `ls | wc`
if (pid > 0) { /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

   /* write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the write end of the pipe */
   close(fd[WRITE_END]);
}
else { /* child process */
   /* close the unused end of the pipe */
   close(fd[WRITE_END]);

   /* read from the pipe */
   read(fd[READ_END], read_msg, BUFFER_SIZE);
   printf("read %s", read_msg);

   /* close the write end of the pipe */
   close(fd[READ_END]);
}
Discussion

- What if the parent wants to write something to child, while child also wants to write something to parent?

HOW?

- Hints, ordinary pipes are unidirectional
Chapter 3: Processes

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

(146.86.5.2/1625)

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

**Diagram Description:**
- **User:** Calls kernel to send RPC message to procedure X.
- **Kernel:** Sends message to matchmaker to find port number.
- **Matchmaker:** Receives message, looks up answer.
- **Server:** Places port P in user RPC message.
- **Daemon:** Listens to port P and receives message.
- **RPC:** Processes request and sends output.
- **Kernel:** Receives reply, passes it to user.
- **Client:** Calls server to handle port P.
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

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

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

Diagram:

- Client
  - `val = server.someMethod(A, B)`
  - Stub
  - A, B, someMethod
  - boolean return value

- Remote Object
  - boolean someMethod (Object x, Object y)
  - Skeleton