Chapter 2
Processes and Threads

2.1 Processes
2.2 Threads
2.3 Interprocess communication
2.4 Classical IPC problems
2.5 Scheduling

Processes
The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Process Creation

Principal events that cause process creation
1. System initialization
   • Execution of a process creation system
1. User request to create a new process
2. Initiation of a batch job

Process Termination

Conditions which terminate processes
1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)
Process Hierarchies

- Parent creates a child process, child processes can create their own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - All processes are created equal

Process States (1)

- Possible process states
  - Running
  - Blocked
  - Ready
- Transitions between states shown

Process States (2)

- Lowest layer of process-structured OS
  - Handles interrupts, scheduling
- Above that layer are sequential processes

Implementation of Processes (1)

Fields of a process table entry

- Process management
  - Registers
  - Program counter
  - Program status word
  - Stack pointer
  - Process state
  - Priority
  - Scheduling parameters
- Process ID
- Parent process
- Process group
- Signals
- Time when process started
- CPU time used
- Children's CPU time
- Time of next alarm

- Memory management
  - Pointer to text segment
  - Pointer to data segment
  - Pointer to stack segment

- File management
  - Root directory
  - Working directory
  - File descriptors
  - User ID
  - Group ID
Implementation of Processes (2)

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Skeleton of what lowest level of OS does when an interrupt occurs

The Thread Model (1)

(a) Three processes each with one thread
(b) One process with three threads

The Thread Model (2)

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
- Items private to each thread

The Thread Model (3)

Each thread has its own stack
Thread Usage (1)

A word processor with three threads

Thread Usage (2)

A multithreaded Web server

Thread Usage (3)

• Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker thread

Thread Usage (4)

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls, interrupts</td>
</tr>
</tbody>
</table>
Implementing Threads in User Space

A user-level threads package

Implementing Threads in the Kernel

A threads package managed by the kernel

Hybrid Implementations

Multiplexing user-level threads onto kernel-level threads

Scheduler Activations

- Goal – mimic functionality of kernel threads
  - gain performance of user space threads
- Avoids unnecessary user/kernel transitions
- Kernel assigns virtual processors to each process
  - lets runtime system allocate threads to processors
- Problem:
  Fundamental reliance on kernel (lower layer) calling procedures in user space (higher layer)
**Pop-Up Threads**

- Creation of a new thread when message arrives
  - (a) before message arrives
  - (b) after message arrives

**Making Single-Threaded Code Multithreaded (1)**

Conflicts between threads over the use of a global variable

**Making Single-Threaded Code Multithreaded (2)**

Threads can have private global variables

**Interprocess Communication**

Race Conditions

Two processes want to access shared memory at the same time
Critical Regions (1)

Four conditions to provide mutual exclusion
1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Critical Regions (2)

Mutual exclusion using critical regions

Mutual Exclusion with Busy Waiting (1)

```c
while (TRUE) {
    while (turn != 0) /* loop */;
    critical_region();
    turn = 1;
    noncritical_region();
}

(a) Process 0.        (b) Process 1.
```

Proposed solution to critical region problem
(a) Process 0.        (b) Process 1.

Mutual Exclusion with Busy Waiting (2)

```c
#define FALSE 0
#define TRUE 1
#define N 2

int turn;           /* whose turn is it? */
int interested[N];  /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
    int other;        /* number of the other process */
    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;      /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}

void leave_region(int process) /* process: who is leaving */
    interested[process] = FALSE; /* indicate departure from critical region */
```

Peterson's solution for achieving mutual exclusion
Mutual Exclusion with Busy Waiting (3)

Entering and leaving a critical region using the TSL instruction

Sleep and Wakeup

Producer-consumer problem with fatal race condition

Semaphores

The producer-consumer problem using semaphores

Mutexes

Implementation of \texttt{mutex\_lock} and \texttt{mutex\_unlock}
Monitors (1)

Example of a monitor

```java
monitor example
    integer c;

    procedure producer();
        
    procedure consumer();
        
end monitor;
```

Monitors (2)

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has $N$ slots

```java
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
    end;

    procedure remove; integer;
    begin
        if count = 0 then wait(empty);
        remove_item;
        count := count - 1;
    end;

end

while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item);
    end

end

procedure consumer;
    begin
        item = ProducerConsumer.remove;
        consume_item(item);
    end
end
```

Monitors (3)

Solution to producer-consumer problem in Java (part 1)

```java
public class ProducerConsumer {
    static final int N = 100; // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor
    public static void main(String args[]) {
        p.start(); // start the producer thread
        c.start(); // start the consumer thread
    }
}

public class producer extends Thread {
    public void run() {
        // run method contains the thread code
        int item;
        while (true) {
            // producer loop
            item = producer_item();
            mon.insert(item);
        }
    }
}

private int producer_item() { ... } // actually produce

public class consumer extends Thread {
    public void run() {
        // run method contains the thread code
        int item;
        while (true) {
            // consumer loop
            item = mon.remove();
            consume_item(item);
        }
    }
}

private void consume_item(int item) { ... } // actually consume
```

Monitors (4)

Solution to producer-consumer problem in Java (part 2)

```java
public class ProducerConsumer {
    public class ProducerConsumer {
        
    public class ProducerConsumer {
        
    public class ProducerConsumer {
        
```
Message Passing

```c
#define N 100

void producer(void)
{
    int item;
    message m;
    while (TRUE) {
        /* generate something to put in buffer */
        send(item, &m);
        build_item(&m, item);
    }
}

void consumer(void)
{
    int item, i;
    message m;
    for (i = 0; i < N; i++) {
        send_empty(&m);
        while (TRUE) {
            receive(&item, &m);
            /* do something with the item */
            consume_item(item);
        }
    }
}
```

The producer-consumer problem with N messages

---

Barriers

- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through

---

Dining Philosophers (1)

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock

---

Dining Philosophers (2)

```c
#define N 5

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i + 1) % N);
        eat();
        put_fork((i + 1) % N);
    }
}
```

A non-solution to the dining philosophers problem
Dining Philosophers (3)

```c
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define HUNGRY 1
#define THINKING 0
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore e[N];
void philosopher(int i)
{
    while (TRUE) {
        // repeat forever
        think();
        take_forks();
        eat();
        put_forks();
    }
}

Solution to dining philosophers problem (part 1)
```

Dining Philosophers (4)

```c
void take_forks(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test();
    up(&mutex);
    down(&e[i]);
}

void put_forks(i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i)
{
    // i: philosopher number, from 0 to N-1 +
    if (state[i] == HUNGRY && state[LEFT] == EATING && state[RIGHT] == EATING) {
        state[i] = EATING;
        up(&mutex);
    }
}
```

Solution to dining philosophers problem (part 2)

The Readers and Writers Problem

```c
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int c = 0;

void reader(void)
{
    while (TRUE) {
        // repeat forever
        down(mutex);
        c = c + 1;
        if (c == 1) down(db);
        up(&mutex);
        read_data(base); down(mutex);
        c = c - 1;
        if (c == 0) up(db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        // repeat forever
        think_up(&data);
        down(db);
        write_data(base);
        up(&db);
    }
}
```

A solution to the readers and writers problem

The Sleeping Barber Problem (1)

![Sleeping Barber Diagram]

```c
/* i: philosopher number, from 0 to N-1 */
/* enter critical region */
/* record fact that philosopher i is hungry */
/* try to acquire 2 forks */
/* exit critical region */
/* block if forks were not acquired */
```
The Sleeping Barber Problem (2)

```c
#define CHAIRS 5
typedef int semaphore;

semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;

void barber(void)
{
    while (TRUE)
    {
        down(&customers);
        down(&mutex);
        waiting = waiting - 1;
        up(&barbers);
        up(&mutex);
        cut_hair();
        /* go to sleep if the number of customers is 0 */
        /* acquire access to 'waiting' */
        /* decrement count of waiting customers */
        /* one barber is now ready to cut hair */
        /* release 'waiting' */
        /* cut hair outside critical region */
    }
}

void customer(void)
{
    down(&mutex);
    if (waiting < CHAIRS) {
        waiting = waiting + 1;
        up(&customers);
        up(&mutex);
        get_haircut();
        /* enter critical region */
        /* if there are no free chairs, leave */
        /* increment count of waiting customers */
        /* wake up barber if necessary */
        /* release access to 'waiting' */
        /* go to sleep if 8 of two barbers are 0 */
        /* be seated and be serviced */
    } else {
        up(&mutex);
        /* shop is full; do not wait */
    }
}
```

Solution to sleeping barber problem.

Scheduling Introduction to Scheduling (1)

- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O bound process

Introduction to Scheduling (2)

All systems
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems
- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Scheduling Algorithm Goals

Scheduling in Batch Systems (1)

An example of shortest job first scheduling
Scheduling in Batch Systems (2)

Three level scheduling

Scheduling in Interactive Systems (1)

• Round Robin Scheduling
  – list of runnable processes
  – list of runnable processes after B uses up its quantum

Scheduling in Interactive Systems (2)

A scheduling algorithm with four priority classes

Scheduling in Real-Time Systems

Schedulable real-time system

• Given
  – $m$ periodic events
  – event $i$ occurs within period $P_i$ and requires $C_i$ seconds

• Then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$
Policy versus Mechanism

- Separate what is allowed to be done with how it is done
  - a process knows which of its children threads are important and need priority

- Scheduling algorithm parameterized
  - mechanism in the kernel

- Parameters filled in by user processes
  - policy set by user process

Thread Scheduling (1)

Possible scheduling of user-level threads
- 50-msec process quantum
- threads run 5 msec/CPU burst

Thread Scheduling (2)

Possible scheduling of kernel-level threads
- 50-msec process quantum
- threads run 5 msec/CPU burst