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

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)

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td>Parent process</td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td>Process group</td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td>Signals</td>
<td>Time when process started</td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td>Time CPU used</td>
<td>CPU time used</td>
</tr>
<tr>
<td>Process ID</td>
<td>Signals</td>
<td>Children's CPU time</td>
</tr>
<tr>
<td>Parent process</td>
<td>Time of next alarm</td>
<td>Time of next alarm</td>
</tr>
</tbody>
</table>
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 (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>

Three ways to construct a server
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 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)

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

(a)
```

Proposed solution to critical region problem

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

Mutual Exclusion with Busy Waiting (2)

```
#define FALSE 0
#define TRUE 1
#define N 2
/* number of processes */

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)

enter_region:
TSL REGISTER LOCK | copy lock to register and set lock to 1
CMP REGISTER,#0     | was lock zero?
JNE enter_region    | if it was non zero, lock was set, so loop
RET                   | return to caller; critical region entered

leave_region:
MOVE LOCK,#0         | store a 0 in lock
RET                   | return to caller

Entering and leaving a critical region using the TSL instruction

Semaphores

// define N to 100
#define N 100
typedef int semaphore;
semaphore empty = 1;
semaphore full = 0;

void producer(void)
{
  int item;
  while (TRUE) {
    item = produce_item(); // generate something to put in buffer
    decrease_item_count();
    enter_critical_region();
    insert_item(item);
    leave_critical_region();
    increase_item_count();
  }
}

void consumer(void)
{
  int item;
  while (TRUE) {
    item = consume_item(); // take item out of buffer
    decrease_item_count();
    enter_critical_region();
    consume_item(item);
    leave_critical_region();
    increase_item_count();
  }
}

Producer-consumer problem with fatal race condition

Mutexes

mutex_lock:
TSL REGISTER_MUTEX | copy mutex to register and set mutex to 1
CMP REGISTER,#0     | was mutex zero?
JZE ok               | if it was zero, mutex was unlocked, so return
CALL thread_yield    | mutex is busy; schedule another thread
JMP mutex_lock       | try again later
ok: RET               | return to caller; critical region entered

mutex_unlock:
MOVE Mutex,#0        | store a 0 in mutex
RET                   | return to caller

Implementation of mutex_lock and mutex_unlock

The producer-consumer problem using semaphores
Monitors (1)

```java
monitor example
    integer i;
    condition c;

    procedure producer();
    ...
    ...
    end;

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

Example of a monitor

Monitors (2)

```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;
        if count = 1 then signal(empty)
    end;

    procedure consumer();
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
end;
```

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

Monitors (3)

```java
public class ProducerConsumer {
    static final int N = 100;
    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() {
        int item;
        while (true) {
            // producer loop
            item = produce_item();
            mon.insert_item(item);
        }
    }
}

private void produce_item() {
    ...
}

public class consumer extends Thread {
    public void run() {
        int item;
        while (true) {
            // consumer loop
            item = remove_item();
            mon.consume_item(item);
        }
    }
}

private void consume_item(int item) {
    ...
}
```

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

Monitors (4)

```java
static class our_monitor {
    // this is a monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices

    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
        buffer[hi] = val;
        hi = (hi + 1) % N; // insert an item into the buffer
        count = count + 1;
        if (count == 1) notify(); // if consumer was sleeping, wake it up
    }  

    public synchronized void remove() {
        int val;
        if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
        val = buffer[lo];
        lo = (lo + 1) % N; // fetch an item from the buffer
        slot_to_fetch_next_item_from
        count = count - 1;
        if (count == N - 1) notify(); // if producer was sleeping, wake it up
        return val;
    }

    private void go_to_sleep() { try{wait()} catch(InterruptedException ex){} }
}
```

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

```c
define N 100 // number of slots in the buffer */
void producer (void) // message buffer */
{
    int item;
    message m:
    while (TRUE) {
        item = produce_item ( ); // generate something to put in buffer */
        receive ( consumer, &m ); // wait for an empty to arrive */
        send ( consumer, &m ); // send a message to sender */
        send ( consumer, &m ); // send item to consumer */
    }
}
void consumer ( void )
{
    int item, i;
    message m:
    for ( i = 0, i < N, i++ ) send ( producer, &m ); // send N requests */
    while (TRUE) {
        item = extract_item ( &m ); // get message containing item */
        receive ( producer, &m ); // extract item from message */
        send ( producer, &m ); // send back empty reply */
        consume_item ( item ); // do something with the item */
    }
}

The producer-consumer problem with N messages
```

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 // number of philosophers */
void philosopher ( int i ) // i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think ( ); // /* philosopher is thinking */
        take_fork ( i ); // /* take left fork */
        take_fork ( ( i + 1 ) % N ); // /* take right fork; % is modulo operator */
        eat ( ); // yumm-yumm, spaghetti */
        put_fork ( i ); // /* put left fork back on the table */
        put_fork ( ( i + 1 ) % N ); // /* put right fork back on the table */
    }
}
```

A non-solution to the dining philosophers problem

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 (3)

```c
#define N 5 /* number of philosophers */
#define LEFT ((i+1)%N) /* number of i's left neighbor */
#define RIGHT ((i+1)%N) /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */
typedef int semaphore; /* semaphores are a special kind of int */
int state[N]; /* array to keep track of everyone's state */
semaphore mutex = 1; /* mutual exclusion for critical regions */
semaphore s[N]; /* one semaphore per philosopher */

void philosopher(int i) { /* i: philosopher number, from 0 to N-1 */
    while (TRUE) {
        /* repeat forever */
        think();
        /* philosopher is thinking */
        take_forks();
        /* acquire two forks or block */
        eat();
        /* yum-yum, spaghetti */
        put_forks();
        /* put both forks back on table */
    }
}
```

Solution to dining philosophers problem (part 1)

The Readers and Writers Problem

```c
typedef int semaphore; /* use your imagination */
semaphore mutex = 1; /* for critical sections */
semaphore db = 1; /* for database access */
int n = 0; /* number of processes reading or writing */

void readers(n) { /* n: number of readers */
    while (TRUE) { /* repeat forever */
        down(&mutex); /* get exclusive access to db */
        if (n == 0) down(&db); /* if this is the first reader... */
        up(&mutex); /* release exclusive access to db */
        n--; /* access the data */
        up(&db); /* if this is the last reader... */
    }
}

void writers(valid) { /* valid: number of writers */
    while (TRUE) { /* repeat forever */
        up(&mutex); /* go into critical region */
        down(&db); /* get exclusive access */
        update_data(); /* update the data */
        up(&db); /* go into critical region */
    }
}
```

A solution to the readers and writers problem

Dining Philosophers (4)

```c
void take_forks(int i) { /* i: philosopher number, from 0 to N-1 */
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks() { /* i: philosopher number, from 0 to N-1 */
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

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

Solution to dining philosophers problem (part 2)

The Sleeping Barber Problem (1)

void take_barber() { /* barber number */
    down(&mutex);
    state[i] = HUNGRY;
    test();
    up(&mutex);
    down(&s[i]);
}

void put_barber() { /* barber number */
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test() { /* barber number */
    if (state[i] == HUNGRY && state[LEFT] == EATING && state[RIGHT] == EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

A solution to the sleeping barber problem
The Sleeping Barber Problem (2)

```c
#include <semaphore.h>

typedef int semaphore;

semaphore chairs = 5;
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;
semaphore cut_hair = 0;

void barber(void)
{
    while (TRUE)
    {
        down(chairs);
        acquire(waiting);
        waiting = waiting - 1;
        up(chairs);
        up(mutex);
        cut_hair();
    }
}

t void customer(void)
{
    down(mutex);
    if (waiting < CHAIRS)
    {
        waiting = waiting + 1;
        increment count of waiting customers
        up(chairs);
        up(mutex);
        go to sleep
        get haircut;
        sleep for cut hair
        up(mutex);
        shop is full; do not wait
    }
}
```

Solution to sleeping barber problem.

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)

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

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

Scheduleable 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