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 its 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

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
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

<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></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**CuuDuongThanCong.com**

https://fb.com/tailieudientucntt
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
Threads
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

Diagram showing the components of a multithreaded Web server process, including a dispatcher thread, worker threads, and a web page cache, all within the kernel and user space.
Thread Usage (3)

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

```c
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

```c
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page)
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```
### 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

Multiple user threads on a kernel thread

User space

Kernel space

Kernel thread
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
Conflicts between threads over the use of a global variable
Threads can have private global variables
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)

Proposed solution to critical region problem

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

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

while (TRUE) {
    while (turn != 1) /* loop */;
    critical_region();
    turn = 0;
    noncritical_region();
}
# 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
#define N 100  /* number of slots in the buffer */
int count = 0;  /* number of items in the buffer */

void producer(void)
{
    int item;
    while (TRUE) {  /* repeat forever */
        item = produce_item();  /* generate next item */
        if (count == N) sleep();  /* if buffer is full, go to sleep */
        insert_item(item);  /* put item in buffer */
        count = count + 1;  /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);  /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;
    while (TRUE) {  /* repeat forever */
        if (count == 0) sleep();  /* if buffer is empty, go to sleep */
        item = remove_item();  /* take item out of buffer */
        count = count - 1;  /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer);  /* was buffer full? */
        consume_item(item);  /* print item */
    }
}

Producer-consumer problem with fatal race condition
Semaphores

```c
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

The producer-consumer problem using semaphores
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
Monitors (1)

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

procedure producer();
.
end;

procedure consumer();
.
end;
end monitor;

Example of a monitor
```
Outline of producer-consumer problem with monitors

- only one monitor procedure active at one time
- buffer has $N$ slots

```verbatim
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;
function remove: integer;
begin
  if count = 0 then wait(empt)
  remove = remove_item;
  count := count - 1;
  if count = $N - 1$ then signal(full)
end;
count := 0;
end monitor;

procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
end;
```
Monitors (3)

```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
    }
    static class producer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... } // actually produce
    }
    static 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
    }
}
```

Solution to producer-consumer problem in Java (part 1)
Monitors (4)

```java
{
    synchronized (this) {
        if (count == N - 1) {  // If the producer is still working.
            count = count - 1;  // Decrease the count.
            jo = (jo + 1) % N;  // Move to the next index.
            used[jo] = false;   // Mark index as used.
            if (count == 0) {  // If count is zero, wake up the consumer.
                synchronized (this) {  // Synchronize with the consumer.
                    if (count == 1) {  // If count is one, increase it.
                        count = count + 1;  // Increase the count.
                        mu = (mu + 1) % N;  // Move to the next index.
                        used[mu] = true;  // Mark index as used.
                        break;  // Exit the while loop.
                    }  // If count is zero, sleep.
                    synchronized (this) {  // Synchronize with the consumer.
                        if (count == 0) {  // If count is zero, wake up the consumer.
                            synchronized (this) {  // Synchronize with the consumer.
                                if (count == 1) {  // If count is one, increase it.
                                    count = count + 1;  // Increase the count.
                                    mu = (mu + 1) % N;  // Move to the next index.
                                    used[mu] = true;  // Mark index as used.
                                    break;  // Exit the while loop.
                                }  // If count is zero, sleep.
                            }  // Synchronize with the consumer.
                        }  // Synchronize with the consumer.
                    }  // Synchronize with the consumer.
                }  // Synchronize with the consumer.
            }  // If count is zero, wake up the consumer.
        }  // If the producer is still working.
    }  // Synchronize with the consumer.
}
```

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

```c
#define N 100

void producer(void)
{
    int item;
    message m; /* message buffer */

    while (TRUE) {
        item = produce_item(); /* generate something to put in buffer */
        receive(consumer, &m); /* wait for an empty to arrive */
        build_message(&m, item); /* construct a message to send */
        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 empties */
    while (TRUE) {
        receive(producer, &m); /* get message containing item */
        item = extract_item(&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
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
#define N 5

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

/* number of philosophers */
/* i: philosopher number, from 0 to 4 */
/* philosopher is thinking */
/* take left fork */
/* take right fork; % is modulo operator */
/* yum-yum, spaghetti */
/* put left fork back on the table */
/* put right fork back on the table */

A nonsolution to the dining philosophers problem
Dining Philosophers (3)

```c
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i) {
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

Solution to dining philosophers problem (part 1)
Dining Philosophers (4)

void take_forks(int i) {
    down(&mutex); /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i); /* try to acquire 2 forks */
    up(&mutex); /* exit critical region */
    down(&s[i]); /* block if forks were not acquired */
}

void put_forks(i) {
    down(&mutex); /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT); /* see if left neighbor can now eat */
    test(RIGHT); /* see if right neighbor can now eat */
    up(&mutex); /* exit critical region */
}

void test(i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

Solution to dining philosophers problem (part 2)
The Readers and Writers Problem

typedef int semaphore; /* use your imagination */
semaphore mutex = 1; /* controls access to 'rc' */
semaphore db = 1; /* controls access to the database */
int rc = 0; /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) { /* repeat forever */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc + 1; /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex); /* release exclusive access to 'rc' */
        read_data_base(); /* access the data */
        down(&mutex); /* get exclusive access to 'rc' */
        rc = rc - 1;
        if (rc == 0) up(&db); /* one reader fewer now */
        up(&mutex); /* release exclusive access to 'rc' */
        use_data_read(); /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) { /* repeat forever */
        think_up_data(); /* noncritical region */
        down(&db); /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db); /* release exclusive access */
    }
}

A solution to the readers and writers problem
The Sleeping Barber Problem (1)
The Sleeping Barber Problem (2)

```c
#define CHAIRS 5          /* # chairs for waiting customers */

typedef int semaphore;  /* use your imagination */

semaphore customers = 0;  /* # of customers waiting for service */
semaphore barbers = 0;    /* # of barbers waiting for customers */
semaphore mutex = 1;      /* for mutual exclusion */
int waiting = 0;          /* customers are waiting (not being cut) */

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

void customer(void)
{
    down(&mutex);        /* enter critical region */
    if (waiting < CHAIRS) {
        waiting = waiting + 1;  /* if there are no free chairs, leave */
        up(&customers);        /* increment count of waiting customers */
        up(&mutex);            /* wake up barber if necessary */
        down(&barbers);        /* release access to 'waiting' */
        get_haircut();         /* go to sleep if # of free barbers is 0 */
        } else {
        up(&mutex);            /* be seated and be serviced */
        }
    }
}
```

Solution to sleeping barber problem.
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

Priority 4
Priority 3
Priority 2
Priority 1

Runnable processes
(Highest priority)

(Lowest priority)
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**

Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3