Chapter 2
Processes and Threads
The Process Model

Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.
Process Creation

Events which cause process creation

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.
Process Termination

Events which cause process termination

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).
Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.
Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.
### Implementation of Processes (2)

<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 info</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment info</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment info</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>

Figure 2-4. Some of the fields of a typical process table entry.
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.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.
Figure 2-6. CPU utilization as a function of the number of processes in memory.
Figure 2-7. A word processor with three threads.
Figure 2-8. A multithreaded Web server.
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}

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);
}

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.
Table 2-10. Three ways to construct a server.

<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>

Figure 2-10. Three ways to construct a server.
Figure 2-11. (a) Three processes each with one thread. (b) One process with three threads.
The Classical 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>

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.
The Classical Thread Model (3)

Figure 2-13. Each thread has its own stack.
POSIX Threads (1)

<table>
<thead>
<tr>
<th>Thread call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pthread_create</td>
<td>Create a new thread</td>
</tr>
<tr>
<td>Pthread_exit</td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td>Pthread_join</td>
<td>Wait for a specific thread to exit</td>
</tr>
<tr>
<td>Pthread_yield</td>
<td>Release the CPU to let another thread run</td>
</tr>
<tr>
<td>Pthread_attr_init</td>
<td>Create and initialize a thread’s attribute structure</td>
</tr>
<tr>
<td>Pthread_attr_destroy</td>
<td>Remove a thread’s attribute structure</td>
</tr>
</tbody>
</table>

Figure 2-14. Some of the Pthreads function calls.
POSIX Threads (2)

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* This function prints the thread’s identifier and then exits. */
    printf("Hello World. Greetings from thread %d, tid\n");
    pthread_exit(NULL);
}

...  
```

Figure 2-15. An example program using threads.
Figure 2-15. An example program using threads.
Implementing Threads in User Space

Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.
Hybrid Implementations

Figure 2-17. Multiplexing user-level threads onto kernel-level threads.
Figure 2-18. Creation of a new thread when a message arrives.

(a) Before the message arrives.
(b) After the message arrives.
Figure 2-19. Conflicts between threads over the use of a global variable.
Making Single-Threaded Code Multithreaded (2)

Figure 2-20. Threads can have private global variables.
Figure 2-21. Two processes want to access shared memory at the same time.
Critical Regions (1)

Conditions required to avoid race condition

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.
Figure 2-22. Mutual exclusion using critical regions.
Mutual Exclusion with Busy Waiting

Proposals for achieving mutual exclusion

1. Disabling interrupts
2. Lock variables
3. Strict alternation
4. Peterson's solution
5. The TSL instruction
Figure 2-23. A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.
Peterson's Solution

```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 */
}
```

Figure 2-24. Peterson’s solution for achieving mutual exclusion.
The TSL Instruction (1)

Figure 2-25. Entering and leaving a critical region using the TSL instruction.
Figure 2-26. Entering and leaving a critical region using the XCHG instruction.

The TSL Instruction (2)

enter_region:
MOVE REGISTER,#1       | put a 1 in the register
XCHG REGISTER,LOCK    | swap the contents of the register and lock variable
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
The Producer-Consumer Problem (1)

```c
#define N 100
int count = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

...  
```

Figure 2-27. The producer-consumer problem with a fatal race condition.
void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();  /* repeat forever */
        item = remove_item();    /* if buffer is empty, got to sleep */
        count = count - 1;       /* take item out of buffer */
        if (count == N - 1) wakeup(producer);  /* decrement count of items in buffer */
        consume_item(item);     /* was buffer full? */
        consume_item(item);     /* print item */
    }
}
Figure 2-28. The producer-consumer problem using semaphores.
void consumer(void)
{
    int item;

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

        /* infinite loop */
        /* decrement full count */
        /* enter critical region */
        /* take item from buffer */
        /* leave critical region */
        /* increment count of empty slots */
        /* do something with the item */
    }
}

Figure 2-28. 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
ok:
    RET | return to caller; critical region entered

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

Figure 2-29. Implementation of mutex lock and mutex unlock.
## Mutexes in Pthreads (1)

<table>
<thead>
<tr>
<th>Thread call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pthread_mutex_init</td>
<td>Create a mutex</td>
</tr>
<tr>
<td>Pthread_mutex_destroy</td>
<td>Destroy an existing mutex</td>
</tr>
<tr>
<td>Pthread_mutex_lock</td>
<td>Acquire a lock or block</td>
</tr>
<tr>
<td>Pthread_mutex_trylock</td>
<td>Acquire a lock or fail</td>
</tr>
<tr>
<td>Pthread_mutex_unlock</td>
<td>Release a lock</td>
</tr>
</tbody>
</table>

Figure 2-30. Some of the Pthreads calls relating to mutexes.
### Mutexes in Pthreads (2)

<table>
<thead>
<tr>
<th>Thread call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pthread_cond_init</td>
<td>Create a condition variable</td>
</tr>
<tr>
<td>Pthread_cond_destroy</td>
<td>Destroy a condition variable</td>
</tr>
<tr>
<td>Pthread_cond_wait</td>
<td>Block waiting for a signal</td>
</tr>
<tr>
<td>Pthread_cond_signal</td>
<td>Signal another thread and wake it up</td>
</tr>
<tr>
<td>Pthread_cond_broadcast</td>
<td>Signal multiple threads and wake all of them</td>
</tr>
</tbody>
</table>

Figure 2-31. Some of the Pthreads calls relating to condition variables.
Mutexes in Pthreads (3)

```c
#include <stdio.h>
#include <pthread.h>
define MAX 1000000000
pthread_mutex_t the_mutex;
pthread_cond_t concd, condp;
int buffer = 0;

void *producer(void *ptr)
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

...
Mutexes in Pthreads (4)

... 

```c
void *consumer(void *ptr) {
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}
```

... 

Figure 2-32. Using threads to solve the producer-consumer problem.
Figure 2-32. Using threads to solve the producer-consumer problem.
Monitors (1)

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

    procedure producer();
        ...
    end;

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

Figure 2-33. A monitor.
Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.
Monitors (3)

```plaintext
procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end;
end;

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

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.
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

    ...
Message Passing (2)

... 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
}

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;  // insert an item into the buffer
        hi = (hi + 1) % N;   // slot to place next item in
        count = count + 1;   // one more item in the buffer now
        if (count == 1) notify();  // if consumer was sleeping, wake it up
    }
...

Figure 2-35. A solution to the producer-consumer problem in Java.
public synchronized int remove() {
    int val;
    if (count == 0) go_to_sleep();       // if the buffer is empty, go to sleep
    val = buffer[lo];                   // fetch an item from the buffer
    lo = (lo + 1) % N;                  // slot to fetch next item from
    count = count - 1;                  // one few items in the buffer
    if (count == N - 1) notify();       // if producer was sleeping, wake it up
    return val;
}

private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};
}
#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 */
    }
}

...

Figure 2-36. The producer-consumer problem with N messages.
Figure 2-36. The producer-consumer problem with N messages.
Barriers

Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.
Figure 2-38. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.
Categories of Scheduling Algorithms

1. Batch
2. Interactive
3. Real time
Scheduling Algorithm Goals

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

Figure 2-39. Some goals of the scheduling algorithm under different circumstances.
Scheduling in Batch Systems

- First-come first-served
- Shortest job first
- Shortest remaining Time next
Figure 2-40. An example of shortest job first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.
Scheduling in Interactive Systems

- Round-robin scheduling
- Priority scheduling
- Multiple queues
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling
- Fair-share scheduling
Figure 2-41. Round-robin scheduling.
(a) The list of runnable processes. (b) The list of runnable processes after B uses up its quantum.
Priority Scheduling

Figure 2-42. A scheduling algorithm with four priority classes.
Thread Scheduling (1)

Figure 2-43. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.
Figure 2-43. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).
Dining Philosophers Problem (1)

Figure 2-44. Lunch time in the Philosophy Department.
Dining Philosophers Problem (2)

```c
#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 */
```

Figure 2-45. A nonsolution to the dining philosophers problem.


```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);
    }
}
```

/* number of philosophers */
/* number of i’s left neighbor */
/* number of i’s right neighbor */
/* philosopher is thinking */
/* philosopher is trying to get forks */
/* philosopher is eating */
/* semaphores are a special kind of int */
/* array to keep track of everyone’s state */
/* mutual exclusion for critical regions */
/* one semaphore per philosopher */
/* i: philosopher number, from 0 to N-1 */
/* repeat forever */
/* philosopher is thinking */
/* acquire two forks or block */
/* yum-yum, spaghetti */
/* put both forks back on table */

Figure 2-46. A solution to the dining philosophers problem.
Figure 2-46. A solution to the dining philosophers problem.
Dining Philosophers Problem (5)

... 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(&s[i]);
    }
}

Figure 2-46. A solution to the dining philosophers problem.
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
        /* use your imagination */
        /* controls access to ’rc’ */
        /* controls access to the database */
        /* # of processes reading or wanting to */
        /* repeat forever */
        /* get exclusive access to ’rc’ */
        /* one reader more now */
        /* if this is the first reader ... */
        /* release exclusive access to ’rc’ */
        /* access the data */
        /* get exclusive access to ’rc’ */
        /* one reader fewer now */
        /* if this is the last reader ... */
        /* release exclusive access to ’rc’ */
        /* noncritical region */
    }
}

...
The Readers and Writers Problem (2)

... 

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

Figure 2-47. A solution to the readers and writers problem.