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.
Events which cause process creation:

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

Events which cause process termination:

- Normal exit (voluntary).
- Error exit (voluntary).
- Fatal error (involuntary).
- Killed by another process (involuntary).
Process States

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.
### Figure 2-4. Some of the fields of a typical process table entry.

<table>
<thead>
<tr>
<th><strong>Process management</strong></th>
<th><strong>Memory management</strong></th>
<th><strong>File management</strong></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>
Implementation of Processes (3)

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 2-7. A word processor with three threads.
Thread Usage (2)

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);
}
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>

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.
Figure 2-13. Each thread has its own stack.
### Figure 2-14. Some of the Pthreads function calls.

<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>
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];
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d, i];
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d, status];
            exit(-1);
        }
    }

    exit(NULL);
}
```

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.
Making Single-Threaded Code Multithreaded (1)

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.
Race Conditions

Figure 2-21. Two processes want to access shared memory at the same time.
Conditions required to avoid race condition:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.
Critical Regions (2)

Figure 2-22. Mutual exclusion using critical regions.
Mutual Exclusion with Busy Waiting

Proposals for achieving mutual exclusion:

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- 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.
The Producer-Consumer Problem

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

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);
        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 */
    }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.
Figure 2-28. The producer-consumer problem using semaphores.

```c
#define N 100
typedef int semaphore; /* number of slots in the buffer */
semaphore mutex = 1; /* semaphores are a special kind of int */
semaphore empty = N; /* controls access to critical region */
semaphore full = 0; /* counts empty buffer slots */
/* counts full buffer slots */

void producer(void)
{
    int item;
    while (TRUE) { /* TRUE is the constant 1 */
        item = produce_item(); /* generate something to put in buffer */
        down(&empty); /* decrement empty count */
        down(&mutex); /* enter critical region */
        insert_item(item);
        up(&mutex);
        up(&full); /* put new item in buffer */
        /* leave critical region */
        /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;
    while (TRUE) { /* infinite loop */
        down(&full); /* decrement full count */
        down(&mutex); /* enter critical region */
        item = remove_item(); /* take item from buffer */
        up(&mutex);
        up(&empty); /* leave critical region */
        /* increment count of empty slots */
        consume_item(item); /* do something with the item */
        ..
    }
}
```

Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
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.
<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.
Mutixes in Pthreads (3)

```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */
void *producer(void *ptr) /* produce data */
{
    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 = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr) /* consume data */
{
    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(&condc); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
    return 0;
}
```

Figure 2-32. Using threads to solve the producer-consumer problem.
Figure 2-35. A solution to the producer-consumer problem in Java.
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) {};
}
Figure 2-36. The producer-consumer problem with N messages.
Figure 2-36. The producer-consumer problem with N messages.
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.
Dining Philosophers Problem (1)

Figure 2-44. Lunch time in the Philosophy Department.
Dining Philosophers Problem (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(); /* yum-yum, spaghetti */
        put_fork(i); /* put left fork back on the table */
        put_fork((i+1) % N); /* put right fork back on the table */
    }
}
```

Figure 2-45. A nonsolution to the dining philosophers problem.
Dining Philosophers Problem (3)

```c
#define N 5
#define LEFT (i+N-1)%N
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
type define HUNGRY 1
int st define EATING 2
sem typedef int semaphore;
sem int state[N];
void 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 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.
Dining Philosophers Problem (4)

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

/* 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 */
```

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

... 

```c
void put_forks(i)  /* i: philosopher number, from 0 to N−1 */
{
    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)  /* 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;  /* 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;  /* one reader fewer now */
        if (rc == 0) up(&db);  /* if this is the last reader ... */
        up(&mutex);  /* release exclusive access to ’rc’ */
        use_data_read();  /* noncritical region */
    }  
}

...  

Figure 2-47. A solution to the readers and writers problem.
The Readers and Writers Problem (2)

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 */
    }
}

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

- Batch
- Interactive
- 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.