Chapter 2
Processes and Threads

Process Creation

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.

Implementation of Processes (1)

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.

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.

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.
Figure 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</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 2-11. (a) Three processes each with one thread. (b) One process with three threads.

The Classical Thread Model (1)

The Classical Thread Model (2)

The Classical Thread Model (3)

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.

<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 2-13. Each thread has its own stack.

POSIX Threads (1)

Figure 2-14. Some of the Pthreads function calls.

POSIX Threads (2)

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.

Pop-Up 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.

Proposals for achieving mutual exclusion:

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

Peterson's solution for achieving mutual exclusion:

```
define FALSE 0
define TRUE 1

if turn;  /* whose turn is it? */
  if interested[process]; /* are you interested? */
  if other = 1 - process; /* the opposite of process */
  other = interested[process]; /* tell which process is interested */
else;
  if turn = process; /* who is removing lock */
  if interested[process] = TRUE; /* indicate departure from critical region */
```

The TSL Instruction (1)

```
enter_region:
    TSL REGISTER LOCK
    CMP REGISTER AS
    JNE enter_region
    RET

leave_region:
    MOV EAX, 0
    MOV LOCK, H0
    RET
    JNE enter_region
    RET
```

**Figure 2-22. Mutual exclusion using critical regions.**

**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.

**Figure 2-24. Peterson's solution for achieving mutual exclusion.**

**Figure 2-25. Entering and leaving a critical region using the TSL instruction.**
The TSL Instruction (2)

```c
/* number of items in the buffer */
void producer(void)
{
    int b = 0;
    while (b < n)
    {
        // generate random data
        data[b] = random();
        b++;
    }
}
```

```c
/* number of items in the buffer */
void consumer(void)
{
    int b = 0;
    while (b < n)
    {
        // read data
        data[b] = random();
        b++;
    }
}
```

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

Semaphores

```c
/* number of items in the buffer */
void producer(void)
{
    int b = 0;
    while (b < n)
    {
        // generate random data
        data[b] = random();
        b++;
    }
}
```

```c
/* number of items in the buffer */
void consumer(void)
{
    int b = 0;
    while (b < n)
    {
        // read data
        data[b] = random();
        b++;
    }
}
```

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

The Producer-Consumer Problem

```
/* number of items in the buffer */
void producer(void)
{
    int b = 0;
    while (b < n)
    {
        // generate random data
        data[b] = random();
        b++;
    }
}
```

```
/* number of items in the buffer */
void consumer(void)
{
    int b = 0;
    while (b < n)
    {
        // read data
        data[b] = random();
        b++;
    }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Mutexes

```
/* number of items in the buffer */
void producer(void)
{
    int b = 0;
    while (b < n)
    {
        // generate random data
        data[b] = random();
        b++;
    }
}
```

```
/* number of items in the buffer */
void consumer(void)
{
    int b = 0;
    while (b < n)
    {
        // read data
        data[b] = random();
        b++;
    }
}
```

Figure 2-29. Implementation of mutex lock and mutex unlock.

Mutexes in Pthreads (1)

```
/* number of items in the buffer */
void producer(void)
{
    int b = 0;
    while (b < n)
    {
        // generate random data
        data[b] = random();
        b++;
    }
}
```

```
/* number of items in the buffer */
void consumer(void)
{
    int b = 0;
    while (b < n)
    {
        // read data
        data[b] = random();
        b++;
    }
}
```

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads (2)

```
/* number of items in the buffer */
void producer(void)
{
    int b = 0;
    while (b < n)
    {
        // generate random data
        data[b] = random();
        b++;
    }
}
```

```
/* number of items in the buffer */
void consumer(void)
{
    int b = 0;
    while (b < n)
    {
        // read data
        data[b] = random();
        b++;
    }
}
```

Figure 2-31. Some of the Pthreads calls relating to condition variables.
Figure 2-32. Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (3)

Figure 2-33. A monitor.

Monitors (1)

```java
monitor example

integer i;

condition c:

procedure producer()

begin

end:

procedure consumer()

begin

end;

end monitor;

Figure 2-34. An outline of the producer-consumer problem with monitors.

Monitors (2)

```

Figure 2-35. A solution to the producer-consumer problem in Java.

Monitors (3)

```java
public synchronized (setRemove) ()

if (not empty)

if (not full)

if (not blocked)

if (not empty)

if (not full)

if (not blocked)

Figure 2-35. A solution to the producer-consumer problem in Java.

public synchronized (setRemove) ()

if (not empty)

if (not full)

if (not blocked)

if (not empty)

if (not full)

if (not blocked)

```
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.

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.

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

Shortest Job First

(a)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

(b)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
</tr>
</tbody>
</table>

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

Round-Robin Scheduling

(a)

(b)

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

<table>
<thead>
<tr>
<th>Queue</th>
<th>Runnable processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority 4</td>
<td>(Highest priority)</td>
</tr>
<tr>
<td>Priority 3</td>
<td></td>
</tr>
<tr>
<td>Priority 2</td>
<td></td>
</tr>
<tr>
<td>Priority 1</td>
<td>(Lowest priority)</td>
</tr>
</tbody>
</table>

Figure 2-42. A scheduling algorithm with four priority classes.

Thread Scheduling (1)

(a)

(b)

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).

Figure 2-44. Lunch time in the Philosophy Department.

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

Figure 2-46. A solution to the dining philosophers problem.
The Readers and Writers Problem (1)

```c
typedef int semaphore;
semaphore mutex = 1;
semaphore data = 1;
int n = 0;

void reader(void) {
    while (TRUE) {
        down(&mutex);
        if (n == 1) down(&data);
        read(&data, n);
        up(&mutex);
        n = n + 1;
        if (n == 2) up(&data);
    }
}
```

The Readers and Writers Problem (2)

```c
void writer(void) {
    while (TRUE) {
        think_up_down();
        down(&data);
        update_data();
        up(&data);
    }
}
```

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

The Sleeping Barber Problem (1)

```c
typedef int semaphore;
semaphore customers = 0;
semaphore barber = 1;
semaphore hair = 1;
int n = 0;

void barber(void) {
    while (TRUE) {
        demarcate_customers();
        standing = waiting = 1;
        waiting = 0;
        up(&hair);
        if (n == 0) down(&customers);
        close_barber();
        go_to_sleep();
        up(&customers);
        if (n == 0) done_sleep();
    }
}
```

The Sleeping Barber Problem (2)

```c
void customer(void) {
    down(&customers);
    if (standing == 1) {
        down(&hair);
        if (n == 0) barber();
        if (n == 0) n = 1;
        if (n == 0) stand_barber();
        up(&hair);
        up(&customers);
    }
}```

Solution to sleeping barber problem.