Chapter 6
Concurrency: Deadlock and Starvation

Ninth Edition
By William Stallings
The permanent blocking of a set of processes that either compete for system resources or communicate with each other.

A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.

Permanent because none of the events is ever triggered.

No efficient solution in the general case.
Figure 6.1  Illustration of Deadlock
Figure 6.2 Example of Deadlock

- both $P$ and $Q$ want resource $A$ (represented by a striped pattern)
- both $P$ and $Q$ want resource $B$ (represented by a checkered pattern)
- deadlock-inevitable region (represented by a gray shaded area)

Possible progress path of $P$ and $Q$:
- Horizontal portion of path indicates $P$ is executing and $Q$ is waiting.
- Vertical portion of path indicates $Q$ is executing and $P$ is waiting.

Legend:
- $A$ Required
- $B$ Required

Progress of $P$ and $Q$:
- Get $A$
- Get $B$
- Release $A$
- Release $B$
- Progress of $Q$
Figure 6.3 Example of No Deadlock

- Both P and Q want resource A
- Both P and Q want resource B

Possible progress path of P and Q:
- Horizontal portion of path indicates P is executing and Q is waiting.
- Vertical portion of path indicates Q is executing and P is waiting.
Reusable

- Can be safely used by only one process at a time and is not depleted by that use
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- One that can be created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information
- In I/O buffers
<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_0$</td>
<td>Request (D)</td>
<td>$q_0$</td>
<td>Request (T)</td>
</tr>
<tr>
<td>$p_1$</td>
<td>Lock (D)</td>
<td>$q_1$</td>
<td>Lock (T)</td>
</tr>
<tr>
<td>$p_2$</td>
<td>Request (T)</td>
<td>$q_2$</td>
<td>Request (D)</td>
</tr>
<tr>
<td>$p_3$</td>
<td>Lock (T)</td>
<td>$q_3$</td>
<td>Lock (D)</td>
</tr>
<tr>
<td>$p_4$</td>
<td>Perform function</td>
<td>$q_4$</td>
<td>Perform function</td>
</tr>
<tr>
<td>$p_5$</td>
<td>Unlock (D)</td>
<td>$q_5$</td>
<td>Unlock (T)</td>
</tr>
<tr>
<td>$p_6$</td>
<td>Unlock (T)</td>
<td>$q_6$</td>
<td>Unlock (D)</td>
</tr>
</tbody>
</table>

**Figure 6.4 Example of Two Processes Competing for Reusable Resources**
Example 2: Memory Request

- Space is available for allocation of 200Kbytes, and the following sequence of events occur:

  - P1
    - ... Request 80 Kbytes;
    - ... Request 60 Kbytes;
  
  - P2
    - ... Request 70 Kbytes;
    - ... Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request
Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

- **P1**
  - ...
  - Receive (P2);
  - ...
  - Send (P2, M1);

- **P2**
  - ...
  - Receive (P1);
  - ...
  - Send (P1, M2);

Deadlock occurs if the Receive is blocking
There is no single effective strategy that can deal with all types of deadlock.

Three approaches are common:

- **Deadlock avoidance**
  - Do not grant a resource request if this allocation might lead to deadlock.

- **Deadlock prevention**
  - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening.

- **Deadlock detection**
  - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover.
Figure 6.5 Examples of Resource Allocation Graphs
Figure 6.6  Resource Allocation Graph for Figure 6.1b
## Conditions for Deadlock

<table>
<thead>
<tr>
<th>Mutual Exclusion</th>
<th>Hold-and-Wait</th>
<th>No Pre-emption</th>
<th>Circular Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Only one process may use a resource at a time</td>
<td>• A process may hold allocated resources while awaiting assignment of other resources</td>
<td>• No resource can be forcibly removed from a process holding it</td>
<td>• A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain</td>
</tr>
</tbody>
</table>
Design a system in such a way that the possibility of deadlock is excluded

Two main methods:

- Indirect
  - Prevent the occurrence of one of the three necessary conditions
- Direct
  - Prevent the occurrence of a circular wait
Mutual exclusion
- If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS
- Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes
- Even in this case, deadlock can occur if more than one process requires write permission

Hold and wait
- Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
Deadlock Condition Prevention

- No Preemption
  - If a process holding certain resources is denied a further request, that process must release its original resources and request them again.
  - OS may preempt the second process and require it to release its resources.

- Circular Wait
  - The circular wait condition can be prevented by defining a linear ordering of resource types.
Deadlock Avoidance

- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock

- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached

- Requires knowledge of future process requests
**Two Approaches to Deadlock Avoidance**

**Deadlock Avoidance**

- **Resource Allocation Denial**
  - Do not grant an incremental resource request to a process if this allocation might lead to deadlock

- **Process Initiation Denial**
  - Do not start a process if its demands might lead to deadlock
Resource Allocation Denial

- Referred to as the *banker’s algorithm*

- *State* of the system reflects the current allocation of resources to processes

- *Safe state* is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock

- *Unsafe state* is a state that is not safe
**Figure 6.7 Determination of a Safe State**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim matrix C

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Allocation matrix A

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

C – A

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Resource vector R

Available vector V

(a) Initial state
Figure 6.7 Determination of a Safe State

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Claim matrix C

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Allocation matrix A

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

C – A

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Resource vector R

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Available vector V

(b) P2 runs to completion
Figure 6.7 Determination of a Safe State

(c) P1 runs to completion
(d) P3 runs to completion

Figure 6.7 Determination of a Safe State
Figure 6.8 Determination of an Unsafe State

(a) Initial state

(b) P1 requests one unit each of R1 and R3
struct state {
    int resource[m];
    int available[m];
    int claim[n][m];
    int alloc[n][m];
}

(a) global data structures

if (alloc[i,+] + request[*] > claim[i,+])
    < error >; /* total request > claim*/
else if (request[*] > available[*])
    < suspend process >;
else
    /* simulate alloc */
    < define newstate by:>
    alloc[i,+] = alloc[i,+] + request[*];
    available[*] = available[*] - request[*];
} if (safe(newstate))
    < carry out allocation >;
else
    < restore original state >;
    < suspend process >;

(b) resource allocation algorithm

boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = (all processes);
    possible = true;
    while (possible) {
        <find a process Pk in rest such that>
        claim[k,+] = alloc[k,+] <= currentavail;>
        if (found) {
            /* simulate execution of Pk */
            currentavail = currentavail + alloc[k,+];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic
Deadlock Avoidance

Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection.
- It is less restrictive than deadlock prevention.
Deadlock Avoidance Restrictions

- Maximum resource requirement for each process must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources
Deadlock Strategies

Deadlock prevention strategies are very conservative

- Limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

- Resource requests are granted whenever possible
Deadline Detection Algorithm

- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur.

**Advantages:**
- It leads to early detection
- The algorithm is relatively simple

**Disadvantage**
- Frequent checks consume considerable processor time
### Figure 6.10  Example for Deadlock Detection

**Request matrix Q**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Allocation matrix A**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Resource vector**

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Allocation vector**

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Recovery Strategies

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists
Integrated Deadlock Strategy

Rather than attempting to design an OS facility that employs only one of these strategies, it might be more efficient to use different strategies in different situations.

- Group resources into a number of different resource classes
- Use the linear ordering strategy defined previously for the prevention of circular wait to prevent deadlocks between resource classes
- Within a resource class, use the algorithm that is most appropriate for that class

Classes of resources

- Swappable space
  - Blocks of memory on secondary storage for use in swapping processes
- Process resources
  - Assignable devices, such as tape drives, and files
- Main memory
  - Assignable to processes in pages or segments
- Internal resources
  - Such as I/O channels
Within each class the following strategies could be used:

- **Swappable space**
  - Prevention of deadlocks by requiring that all of the required resources that may be used be allocated at one time, as in the hold-and-wait prevention strategy.
  - This strategy is reasonable if the maximum storage requirements are known.

- **Process resources**
  - Avoidance will often be effective in this category, because it is reasonable to expect processes to declare ahead of time the resources that they will require in this class.
  - Prevention by means of resource ordering within this class is also possible.

- **Main memory**
  - Prevention by preemption appears to be the most appropriate strategy for main memory.
  - When a process is preempted, it is simply swapped to secondary memory, freeing space to resolve the deadlock.

- **Internal resources**
  - Prevention by means of resource ordering can be used.
Dining Philosophers Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)

- No philosopher must starve to death (avoid deadlock and starvation)

Figure 6.11 Dining Arrangement for Philosophers
/* program diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
        philosopher (3), philosopher (4));
}
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
Figure 6.14

A Solution to the Dining Philosophers Problem Using a Monitor

```c
void philosopher[k=0 to 4] /* the five philosopher clients */
{
    while (true) {
        <think>;
        get_forks(k); /* client requests two forks via monitor */
        <eat spaghetti>;
        release_forks(k); /* client releases forks via the monitor */
    }
}

monitor dining_controller;
cond ForkReady[5]; /* condition variable for synchronization */
boolean fork[5] = {true}; /* availability status of each fork */

void get_forks(int pid) /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /* grant the left fork*/
    if (!fork[left])
        cwait(ForkReady[left]); /* queue on condition variable */
    fork[left] = false;
    /* grant the right fork*/
    if (!fork[right])
        cwait(ForkReady[right]); /* queue on condition variable */
    fork[right] = false;
}

void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /* release the left fork*/
    if (!empty(ForkReady[left]) /* no one is waiting for this fork */
        fork[left] = true;
    else /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /* release the right fork*/
    if (!empty(ForkReady[right]) /* no one is waiting for this fork */
        fork[right] = true;
    else /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}
```
UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals
Pipes

- Circular buffers allowing two processes to communicate on the producer-consumer model
  - First-in-first-out queue, written by one process and read by another

Two types:

- Named
- Unnamed
Messages

- A block of bytes with an accompanying type
- UNIX provides `msgsnd` and `msgrcv` system calls for processes to engage in message passing
- Associated with each process is a message queue, which functions like a mailbox
Shared Memory

- Fastest form of interprocess communication
- Common block of virtual memory shared by multiple processes
- Permission is read-only or read-write for a process
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory
Semaphores

- Generalization of the \texttt{semWait} and \texttt{semSignal} primitives
  - No other process may access the semaphore until all operations have completed

<table>
<thead>
<tr>
<th>Consists of:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Current value of the semaphore</td>
</tr>
<tr>
<td>• Process ID of the last process to operate on the semaphore</td>
</tr>
<tr>
<td>• Number of processes waiting for the semaphore value to be greater than its current value</td>
</tr>
<tr>
<td>• Number of processes waiting for the semaphore value to be zero</td>
</tr>
</tbody>
</table>
A software mechanism that informs a process of the occurrence of asynchronous events
- Similar to a hardware interrupt, but does not employ priorities

A signal is delivered by updating a field in the process table for the process to which the signal is being sent

A process may respond to a signal by:
- Performing some default action
- Executing a signal-handler function
- Ignoring the signal
Table 6.2

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>SIGHUP</td>
<td>Hang up; sent to process when kernel assumes that the user of that process is doing no useful work</td>
</tr>
<tr>
<td>02</td>
<td>SIGINT</td>
<td>Interrupt</td>
</tr>
<tr>
<td>03</td>
<td>SIGQUIT</td>
<td>Quit; sent by user to induce halting of process and production of core dump</td>
</tr>
<tr>
<td>04</td>
<td>SIGILL</td>
<td>Illegal instruction</td>
</tr>
<tr>
<td>05</td>
<td>SIGTRAP</td>
<td>Trace trap; triggers the execution of code for process tracing</td>
</tr>
<tr>
<td>06</td>
<td>SIGIOT</td>
<td>IOT instruction</td>
</tr>
<tr>
<td>07</td>
<td>SIGEMT</td>
<td>EMT instruction</td>
</tr>
<tr>
<td>08</td>
<td>SIGFPE</td>
<td>Floating-point exception</td>
</tr>
<tr>
<td>09</td>
<td>SIGKILL</td>
<td>Kill; terminate process</td>
</tr>
<tr>
<td>10</td>
<td>SIGBUS</td>
<td>Bus error</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Segmentation violation; process attempts to access location outside its virtual address space</td>
</tr>
<tr>
<td>12</td>
<td>SIGSYS</td>
<td>Bad argument to system call</td>
</tr>
<tr>
<td>13</td>
<td>SIGPIPE</td>
<td>Write on a pipe that has no readers attached to it</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Alarm clock; issued when a process wishes to receive a signal after a period of time</td>
</tr>
<tr>
<td>15</td>
<td>SIGTERM</td>
<td>Software termination</td>
</tr>
<tr>
<td>16</td>
<td>SIGUSR1</td>
<td>User-defined signal 1</td>
</tr>
<tr>
<td>17</td>
<td>SIGUSR2</td>
<td>User-defined signal 2</td>
</tr>
<tr>
<td>18</td>
<td>SIGCHLD</td>
<td>Death of a child</td>
</tr>
<tr>
<td>19</td>
<td>SGPWR</td>
<td>Power failure</td>
</tr>
</tbody>
</table>

(Table can be found on page 288 in textbook)
Real-time (RT) Signals

- Linux includes all of the concurrency mechanisms found in other UNIX systems
- Linux also supports real-time (RT) signals
- RT signals differ from standard UNIX signals in three primary ways:
  - Signal delivery in priority order is supported
  - Multiple signals can be queued
  - With standard signals, no value or message can be sent to the target process – it is only a notification
  - With RT signals it is possible to send a value along with the signal
Atomic Operations

- Atomic operations execute without interruption and without interference

- Simplest of the approaches to kernel synchronization

- Two types:

  **Integer Operations**
  - Operate on an integer variable
  - Typically used to implement counters

  **Bitmap Operations**
  - Operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable
### Atomic Integer Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOMIC_INIT(int i)</td>
<td>At declaration: initialize an atomic to i</td>
</tr>
<tr>
<td>int atomic_read(atomic_t *v)</td>
<td>Read integer value of v</td>
</tr>
<tr>
<td>void atomic_set(atomic_t *v, int i)</td>
<td>Set the value of v to integer i</td>
</tr>
<tr>
<td>void atomic_add(int i, atomic_t *v)</td>
<td>Add i to v</td>
</tr>
<tr>
<td>void atomic_sub(int i, atomic_t *v)</td>
<td>Subtract i from v</td>
</tr>
<tr>
<td>void atomic_inc(atomic_t *v)</td>
<td>Add 1 to v</td>
</tr>
<tr>
<td>void atomic_dec(atomic_t *v)</td>
<td>Subtract 1 from v</td>
</tr>
<tr>
<td>int atomic_sub_and_test(int i, atomic_t *v)</td>
<td>Subtract i from v; return 1 if the result is zero; return 0 otherwise</td>
</tr>
<tr>
<td>int atomic_add_negative(int i, atomic_t *v)</td>
<td>Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)</td>
</tr>
<tr>
<td>int atomic_dec_and_test(atomic_t *v)</td>
<td>Subtract 1 from v; return 1 if the result is zero; return 0 otherwise</td>
</tr>
<tr>
<td>int atomic_inc_and_test(atomic_t *v)</td>
<td>Add 1 to v; return 1 if the result is zero; return 0 otherwise</td>
</tr>
</tbody>
</table>

### Atomic Bitmap Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void set_bit(int nr, void *addr)</td>
<td>Set bit nr in the bitmap pointed to by addr</td>
</tr>
<tr>
<td>void clear_bit(int nr, void *addr)</td>
<td>Clear bit nr in the bitmap pointed to by addr</td>
</tr>
<tr>
<td>void change_bit(int nr, void *addr)</td>
<td>Invert bit nr in the bitmap pointed to by addr</td>
</tr>
<tr>
<td>int test_and_set_bit(int nr, void *addr)</td>
<td>Set bit nr in the bitmap pointed to by addr; return the old bit value</td>
</tr>
<tr>
<td>int test_and_clear_bit(int nr, void *addr)</td>
<td>Clear bit nr in the bitmap pointed to by addr; return the old bit value</td>
</tr>
<tr>
<td>int test_and_change_bit(int nr, void *addr)</td>
<td>Invert bit nr in the bitmap pointed to by addr; return the old bit value</td>
</tr>
<tr>
<td>int test_bit(int nr, void *addr)</td>
<td>Return the value of bit nr in the bitmap pointed to by addr</td>
</tr>
</tbody>
</table>

(Table can be found on page 289 in textbook)
Spinlocks

- Most common technique for protecting a critical section in Linux
- Can only be acquired by one thread at a time
  - Any other thread will keep trying (spinning) until it can acquire the lock
- Built on an integer location in memory that is checked by each thread before it enters its critical section
- Effective in situations where the wait time for acquiring a lock is expected to be very short
- Disadvantage:
  - Locked-out threads continue to execute in a busy-waiting mode
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void spin_lock(spinlock_t *lock)</code></td>
<td>Acquires the specified lock, spinning if needed until it is available</td>
</tr>
<tr>
<td><code>void spin_lock_irq(spinlock_t *lock)</code></td>
<td>Like spin lock, but also disables interrupts on the local processor</td>
</tr>
<tr>
<td><code>void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)</code></td>
<td>Like spin lock irq, but also saves the current interrupt state in flags</td>
</tr>
<tr>
<td><code>void spin_lock_bh(spinlock_t *lock)</code></td>
<td>Like spin lock, but also disables the execution of all bottom halves</td>
</tr>
<tr>
<td><code>void spin_unlock(spinlock_t *lock)</code></td>
<td>Releases given lock</td>
</tr>
<tr>
<td><code>void spin_unlock_irq(spinlock_t *lock)</code></td>
<td>Releases given lock and enables local interrupts</td>
</tr>
<tr>
<td><code>void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)</code></td>
<td>Releases given lock and restores local interrupts to given previous state</td>
</tr>
<tr>
<td><code>void spin_unlock_bh(spinlock_t *lock)</code></td>
<td>Releases given lock and enables bottom halves</td>
</tr>
<tr>
<td><code>void spin_lock_init(spinlock_t *lock)</code></td>
<td>Initializes given spinlock</td>
</tr>
<tr>
<td><code>int spin_trylock(spinlock_t *lock)</code></td>
<td>Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise</td>
</tr>
<tr>
<td><code>int spin_is_locked(spinlock_t *lock)</code></td>
<td>Returns nonzero if lock is currently held and zero otherwise</td>
</tr>
</tbody>
</table>

**Table 6.4  Linux Spinlocks**

(Table can be found on page 291 in textbook)
Semaphores

- User level:
  - Linux provides a semaphore interface corresponding to that in UNIX SVR4

- Internally:
  - Implemented as functions within the kernel and are more efficient than user-visible semaphores

- Three types of kernel semaphores:
  - Binary semaphores
  - Counting semaphores
  - Reader-writer semaphores
<table>
<thead>
<tr>
<th><strong>Traditional Semaphores</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>*<em>void sema init(struct semaphore <em>sem, int count)</em></em></td>
</tr>
<tr>
<td>*<em>void init MUTEX(struct semaphore <em>sem)</em></em></td>
</tr>
<tr>
<td>*<em>void init MUTEX LOCKED(struct semaphore <em>sem)</em></em></td>
</tr>
<tr>
<td>*<em>void down(struct semaphore <em>sem)</em></em></td>
</tr>
<tr>
<td>*<em>int down interruptible(struct semaphore <em>sem)</em></em></td>
</tr>
<tr>
<td>*<em>int down trylock(struct semaphore <em>sem)</em></em></td>
</tr>
<tr>
<td>*<em>void up(struct semaphore <em>sem)</em></em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Reader-Writer Semaphores</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>*<em>void init rwsem(struct rw_semaphore, <em>rwsem)</em></em></td>
</tr>
<tr>
<td>*<em>void down read(struct rw semaphore, <em>rwsem)</em></em></td>
</tr>
<tr>
<td>*<em>void up read(struct rw semaphore, <em>rwsem)</em></em></td>
</tr>
<tr>
<td>*<em>void down write(struct rw_semaphore, <em>rwsem)</em></em></td>
</tr>
<tr>
<td>*<em>void up write(struct rw semaphore, <em>rwsem)</em></em></td>
</tr>
</tbody>
</table>

(Table can be found on page 293 in textbook)
# Table 6.6

Linux Memory Barrier Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmb()</td>
<td>Prevents loads from being reordered across the barrier</td>
</tr>
<tr>
<td>wmb()</td>
<td>Prevents stores from being reordered across the barrier</td>
</tr>
<tr>
<td>mb()</td>
<td>Prevents loads and stores from being reordered across the barrier</td>
</tr>
<tr>
<td>Barrier()</td>
<td>Prevents the compiler from reordering loads or stores across the barrier</td>
</tr>
<tr>
<td>smp_rmb()</td>
<td>On SMP, provides a rmb() and on UP provides a barrier()</td>
</tr>
<tr>
<td>smp_wmb()</td>
<td>On SMP, provides a wmb() and on UP provides a barrier()</td>
</tr>
<tr>
<td>smp_mb()</td>
<td>On SMP, provides a mb() and on UP provides a barrier()</td>
</tr>
</tbody>
</table>

SMP = symmetric multiprocessor  
UP = uniprocessor  

(Table can be found on page 294 in textbook)
The RCU mechanism is an advanced lightweight synchronization mechanism which was integrated into the Linux kernel in 2002.

The RCU is used widely in the Linux kernel.

RCU is also used by other operating systems.

There is a userspace RCU library called liburcu.

The shared resources that the RCU mechanism protects must be accessed via a pointer.

The RCU mechanism provides access for multiple readers and writers to a shared resource.
In addition to the concurrency mechanisms of UNIX SVR4, Solaris supports four thread synchronization primitives:

- Mutual exclusion (mutex) locks
- Semaphores
- Readers/writer locks
- Condition variables
(a) MUXEX lock

- owner (3 octets)
- lock (1 octet)
- waiters (2 octets)
- type specific info (4 octets)
  (possibly a turnstile id, lock type filler, or statistics pointer)

(b) Semaphore

- Type (1 octet)
- wlock (1 octet)
- waiters (2 octets)
- count (4 octets)

(c) Reader/Writer lock

- Type (1 octet)
- wlock (1 octet)
- waiters (2 octets)
- thread owner (4 octets)
- union (4 octets)
  (statistic pointer or number of write requests)

(d) Condition variable

Figure 6.15 Solaris Synchronization Data Structures
Mutual Exclusion (MUTEX) Lock

- Used to ensure only one thread at a time can access the resource protected by the mutex
- The thread that locks the mutex must be the one that unlocks it
- A thread attempts to acquire a mutex lock by executing the `mutex_enter` primitive
- Default blocking policy is a spinlock
- An interrupt-based blocking mechanism is optional
Semaphores

Solaris provides classic counting semaphores with the following primitives:

- `sema_p()` Decrements the semaphore, potentially blocking the thread
- `sema_v()` Increments the semaphore, potentially unblocking a waiting thread
- `sema_tryp()` Decrements the semaphore if blocking is not required
Readers/Writer Locks

- Allows multiple threads to have simultaneous read-only access to an object protected by the lock.

- Allows a single thread to access the object for writing at one time, while excluding all readers.
  - When lock is acquired for writing it takes on the status of write lock.
  - If one or more readers have acquired the lock its status is read lock.
A condition variable is used to wait until a particular condition is true.

Condition variables must be used in conjunction with a mutex lock.
Windows Concurrency Mechanisms

- Windows provides synchronization among threads as part of the object architecture

Most important methods are:

- Executive dispatcher objects
- User mode critical sections
- Slim reader-writer locks
- Condition variables
- Lock-free operations
Wait Functions

- Allow a thread to block its own execution.
- Do not return until the specified criteria have been met.
- The type of wait function determines the set of criteria used.
<table>
<thead>
<tr>
<th>Object Type</th>
<th>Definition</th>
<th>Set to Signaled State When</th>
<th>Effect on Waiting Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notification event</td>
<td>An announcement that a system event has occurred</td>
<td>Thread sets the event</td>
<td>All released</td>
</tr>
<tr>
<td>Synchronization event</td>
<td>An announcement that a system event has occurred</td>
<td>Thread sets the event</td>
<td>One thread released</td>
</tr>
<tr>
<td>Mutex</td>
<td>A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore</td>
<td>Owning thread or other thread releases the mutex</td>
<td>One thread released</td>
</tr>
<tr>
<td>Semaphore</td>
<td>A counter that regulates the number of threads that can use a resource</td>
<td>Semaphore count drops to zero</td>
<td>All released</td>
</tr>
<tr>
<td>Waitable timer</td>
<td>A counter that records the passage of time</td>
<td>Set time arrives or time interval expires</td>
<td>All released</td>
</tr>
<tr>
<td>File</td>
<td>An instance of an opened file or I/O device</td>
<td>I/O operation completes</td>
<td>All released</td>
</tr>
<tr>
<td>Process</td>
<td>A program invocation, including the address space and resources required to run the program</td>
<td>Last thread terminates</td>
<td>All released</td>
</tr>
<tr>
<td>Thread</td>
<td>An executable entity within a process</td>
<td>Thread terminates</td>
<td>All released</td>
</tr>
</tbody>
</table>

Note: Shaded rows correspond to objects that exist for the sole purpose of synchronization.

(Table can be found on page 299 in the textbook)
Critical Sections

- Similar mechanism to mutex except that critical sections can be used only by the threads of a single process

- If the system is a multiprocessor, the code will attempt to acquire a spin-lock
  - As a last resort, if the spinlock cannot be acquired, a dispatcher object is used to block the thread so that the kernel can dispatch another thread onto the processor
Windows Vista added a user mode reader-writer

The reader-writer lock enters the kernel to block only after attempting to use a spin-lock

It is *slim* in the sense that it normally only requires allocation of a single pointer-sized piece of memory
Windows also has condition variables

The process must declare and initialize a CONDITION_VARIABLE

Used with either critical sections or SRW locks

Used as follows:
1. acquire exclusive lock
2. while (predicate() == FALSE) SleepConditionVariable()
3. perform the protected operation
4. release the lock
Lock-free Synchronization

- Windows also relies heavily on interlocked operations for synchronization
  - Interlocked operations use hardware facilities to guarantee that memory locations can be read, modified, and written in a single atomic operation

“Lock-free”

- Synchronizing without taking a software lock
- A thread can never be switched away from a processor while still holding a lock
Android adds to the kernel a new capability known as Binder

- Binder provides a lightweight remote procedure call (RPC) capability that is efficient in terms of both memory and processing requirements
- Also used to mediate all interaction between two processes

The RPC mechanism works between two processes on the same system but running on different virtual machines

The method used for communicating with the Binder is the ioctl system call

- The ioctl call is a general-purpose system call for device-specific I/O operations
Figure 6.16 Binder Operation
Summary

- Principles of deadlock
  - Reusable/consumable resources
  - Resource allocation graphs
  - Conditions for deadlock
- Deadlock prevention
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Deadlock avoidance
  - Process initiation denial
  - Resource allocation denial
- Deadlock detection
  - Deadlock detection algorithm
  - Recovery
- Android interprocess communication
- Integrated deadlock strategy

- UNIX concurrency mechanisms
  - Pipes
  - Messages
  - Shared memory
  - Semaphores
  - Signals

- Linux kernel concurrency mechanisms
  - Atomic operations
  - Spinlocks
  - Semaphores
  - Barriers

- Solaris thread synchronization primitives
  - Mutual exclusion lock
  - Semaphores
  - Readers/writer lock
  - Condition variables

- Windows concurrency mechanisms
  - Wait functions
  - Dispatcher objects
  - Critical sections
  - Slim reader-writer locks
  - Lock-free synchronization