Deadlock

- 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
The Six Execution Paths

1. *Q* acquires *B* and then *A* and then releases *B* and *A*. When *P* resumes execution, it will be able to acquire both resources.

2. *Q* acquires *B* and then *A*. *P* executes and blocks on a request for *A*. *Q* releases *B* and *A*. When *P* resumes execution, it will be able to acquire both resources.

3. *Q* acquires *B* and then *P* acquires *A*. Deadlock is inevitable, because as execution proceeds, *Q* will block on *A* and *P* will block on *B*.

4. *P* acquires *A* and then *Q* acquires *B*. Deadlock is inevitable, because as execution proceeds, *Q* will block on *A* and *P* will block on *B*.

5. *P* acquires *A* and then *B*. *Q* executes and blocks on a request for *B*. *P* releases *A* and *B*. When *Q* resumes execution, it will be able to acquire both resources.

6. *P* acquires *A* and then *B* and then releases *A* and *B*. When *Q* resumes execution, it will be able to acquire both resources.
Resource Categories

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

Deadlock occurs if the multiprogramming system interleaves the execution of the two processes as follows:

\[ p_0 p_1 q_0 q_1 p_2 q_2 \]
Example - Memory Request

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

  P1
  Request 80 Kbytes;
  Request 60 Kbytes;

  Deadlock occurs if both processes progress to their second request

Consumable Resources Deadlock

- 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:

  Deadlock occurs if the Receive is blocking (receiving process is blocked until the message is received)

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

  P2
  ... Receive (P1);
  ... Send (P1, M2);
<table>
<thead>
<tr>
<th>Approach</th>
<th>Resource Allocation Policy</th>
<th>Different Schemes</th>
<th>Major Advantages</th>
<th>Major Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prevention</td>
<td>Requesting all resources at once</td>
<td>• Works well for processes that perform a single burst of activity • No preemption necessary</td>
<td>• Inefficient • Delays process initiation • Future resource requirements must be known by processes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Preemption</td>
<td>• Convenient when applied to resources whose state can be saved and restored easily</td>
<td>•Preempts more often than necessary</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resource ordering</td>
<td>• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design</td>
<td>• Disallows incremental resource requests</td>
<td></td>
</tr>
<tr>
<td>Avoidance</td>
<td>Midway between that of detection and prevention</td>
<td>Manipulate to find at least one safe path</td>
<td>• No preemption necessary</td>
<td>• Future resource requirements must be known by OS • Processes can be blocked for long periods</td>
</tr>
<tr>
<td>Detection</td>
<td>Very liberal; requested resources are granted where possible</td>
<td>Invoke periodically to test for deadlock</td>
<td>• Never delays process initiation • Facilitates online handling</td>
<td>• Inherent preemption losses</td>
</tr>
</tbody>
</table>

**Figure 6.5** Examples of Resource Allocation Graphs

- **(a) Resource is requested**
- **(b) Resource is held**
- **(c) Circular wait**
- **(d) No deadlock**
Conditions for Deadlock

**Mutual exclusion** - Only one process may use a resource at a time

**Hold and wait** - A process may hold allocated resources while awaiting assignment of other resources.

**No preemption** - No resource can be forcibly removed from a process holding it.

**Circular wait** - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain
Dealing with Deadlock

- **Prevent Deadlock** - adopt a policy that eliminates one of the conditions
- **Avoid Deadlock** - make the appropriate dynamic choices based on the current state of resource allocation
- **Detect Deadlock** - attempt to detect the presence of deadlock and take action to recover

Deadlock Prevention Strategy

- 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
Deadlock Prevention

• **Mutual Exclusion** - if access to a resource requires mutual exclusion then it must be supported by the OS

• Hold and Wait
  – Require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
  – Inefficient because of the long wait for when they are all available and the long idle times for these resources

Deadlock 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 (practical only when applied to resources whose state can be easily saved and restored later, as is the case with a processor)

• Circular Wait
  – define a linear ordering of resource types
Deadlock Avoidance

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests

Deadlock Avoidance

- Two approaches:
  - **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

Determination of a Safe State

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>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>

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>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>

Resource vector $R$

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Available vector $V$

<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>

(a) Initial state
### Determination of a Safe State

**Claim matrix**

<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>

**Allocation matrix**

<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>

**Resource vector** \( \mathbf{R} \)

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</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>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>

**Available vector** \( \mathbf{V} \)

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</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>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>

#### (b) P2 runs to completion

**Claim matrix**

<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**

<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>

**Resource vector** \( \mathbf{R} \)

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Available vector** \( \mathbf{V} \)

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>9</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

#### (c) P1 runs to completion
Determination of a Safe State

Claim matrix C
Allocation matrix A
C – A

Resource vector R
Available vector V

(a) Initial state

(b) P1 requests one unit each of R1 and R3

P3 runs to completion
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 requirem
- There must be a fixed number of resources to allocate

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
Deadlock Detection Algorithms

- 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

Deadlock Detection Algorithm

1. Mark P4, because P4 has no allocated resources.
2. Set $W = (0 \ 0 \ 0 \ 1)$.
3. The request of process P3 is less than or equal to $W$, so mark P3 and set $W = W + (0 \ 0 \ 0 \ 1) = (0 \ 0 \ 0 \ 1 \ 1)$.
4. No other unmarked process has a row in Q that is less than or equal to $W$.
   - Therefore, terminate the algorithm.
5. The algorithm concludes with P1 and P2 unmarked, indicating that these processes are deadlocked
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
Summary of Deadlock Detection, Prevention, and Avoidance Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Resource Allocation Policy</th>
<th>Different Schemes</th>
<th>Major Advantages</th>
<th>Major Disadvantages</th>
</tr>
</thead>
</table>
| Prevention | Conservative; undercommits resources | Requesting all resources at once | • Works well for processes that perform a single burst of activity  
• No preemption necessary | • Inefficient  
• Delays process initiation  
• Future resource requirements must be known by processes |
|            |                            | Preemption         | • Convenient when applied to resources whose state can be saved and restored easily | • Preempts more often than necessary                                               |
|            |                            | Resource ordering  | • Feasible to enforce via compile-time checks  
• Needs no run-time computation since problem is solved in system design | • Swallows incremental resource requests                                              |
| Avoidance  | Midway between that of detection and prevention | Manipulate to find at least one safe path | • No preemption necessary | • Future resource requirements must be known by OS  
• Processes can be blocked for long periods                                               |
| Detection  | Very liberal; requested resources are granted where possible | Invoke periodically to test for deadlock | • Never delays process initiation  
• Facilitates online handling | • Inherent preemption losses |

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)
Dining Philosophers Problem First Solution

Leads to deadlock – all reach for a first fork and get it.
They all wait for the second fork without yielding the first

Dining Philosophers Problem Second Solution
Dining Philosophers Solution Using a Monitor

```c
monitor dining_controller; /* 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]);
}
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 */
    }
}
```

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:
  - Pipes
  - Messages
  - Shared Memory
  - Semaphores
  - Signals
UNIX Concurrency Mechanisms

• Pipes, messages, and shared memory can be used to communicate data between processes.
• Semaphores and signals are used to trigger actions by other processes.

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
Pipes

- When a pipe is created, it is given a fixed size in bytes.
- When a process attempts to write into the pipe, the write request is immediately executed if there is sufficient room; otherwise the process is blocked.
- A reading process is blocked if it attempts to read more bytes than are currently in the pipe; otherwise the read request is immediately executed.
- The OS enforces mutual exclusion: that is, only one process can access a pipe at a time.

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
Messages

• The message sender specifies the type of message with each message sent, and this can be used as a selection criterion by the receiver.
• The receiver can either retrieve messages in first-in-first-out order or by type.
• A process will block when trying to send a message to a full queue.
• A process will also block when trying to read from an empty queue. If a process attempts to read a message of a certain type and fails because no message of that type is present, the process is not blocked.

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 `semWait` and `semSignal` primitives
  – no other process may access the semaphore until all operations have completed

Semaphores

• Consists of:
  – current value of the semaphore
  – process ID of the last process to operate on the semaphore
  – number of processes waiting for the semaphore value to be greater than its current value
  – number of processes waiting for the semaphore value to be zero
Signals

- 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>
<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>SIGIO</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>SIGPWR</td>
<td>Power failure</td>
</tr>
</tbody>
</table>
Linux Kernel Concurrency Mechanism

• Includes all the mechanisms found in UNIX plus:
  – Spinlocks
  – Atomic Operations
  – Semaphores
  – Barriers

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

- `ATOMIC_INIT(int i)`: At declaration, initialize an atomic_t to `i`.
- `atomic_read(atomic_t *v)`: Read integer value of `v`.
- `atomic_set(atomic_t *v, int i)`: Set the value of `v` to integer `i`.
- `atomic_add(int i, atomic_t *v)`: Add `i` to `v`.
- `atomic_sub(int i, atomic_t *v)`: Subtract `i` from `v`.
- `atomic_inc(atomic_t *v)`: Add `1` to `v`.
- `atomic_dec(atomic_t *v)`: Subtract `1` from `v`.
- `atomic_sub_and_test(int i, atomic_t *v)`: Subtract `i` from `v`; return `1` if the result is zero; return `0` otherwise.
- `atomic_add_negative(int i, atomic_t *v)`: Add `i` to `v`; return `1` if the result is negative; return `0` otherwise (used for implementing semaphores).
- `atomic_dec_and_test(atomic_t *v)`: Subtract `1` from `v`; return `1` if the result is zero; return `0` otherwise.
- `atomic_inc_and_test(atomic_t *v)`: Add `1` to `v`; return `1` if the result is zero; return `0` otherwise.

Atomic Bitmap Operations

- `void set_bit(int nr, void *addr)`: Set bit `nr` in the bitmap pointed to by `addr`.
- `void clear_bit(int nr, void *addr)`: Clear bit `nr` in the bitmap pointed to by `addr`.
- `void change_bit(int nr, void *addr)`: Invert bit `nr` in the bitmap pointed to by `addr`.
- `int test_and_set_bit(int nr, void *addr)`: Set bit `nr` in the bitmap pointed to by `addr`; return the old bit value.
- `int test_and_clear_bit(int nr, void *addr)`: Clear bit `nr` in the bitmap pointed to by `addr`; return the old bit value.
- `int test_and_change_bit(int nr, void *addr)`: Invert bit `nr` in the bitmap pointed to by `addr`; return the old bit value.
- `int test_bit(int nr, void *addr)`: Return the value of bit `nr` in the bitmap pointed to by `addr`.

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.
Spinlocks

- 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

---

### Linux Spinlocks

<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 <code>spin_lock</code>, 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 <code>spin_lock_irq</code>, 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 <code>spin_lock</code>, 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_irqsave(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>
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></th>
<th>Linux Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Traditional Semaphores</strong></td>
<td></td>
</tr>
<tr>
<td>void sema_init(struct semaphore *sem, int count)</td>
<td>Initializes the dynamically created semaphore to the given count</td>
</tr>
<tr>
<td>void init_MUTEX(struct semaphore *sem)</td>
<td>Initializes the dynamically created semaphore with a count of 1 (initially unlocked)</td>
</tr>
<tr>
<td>void init_MUTEX_LOCKED(struct semaphore *sem)</td>
<td>Initializes the dynamically created semaphore with a count of 0 (initially locked)</td>
</tr>
<tr>
<td>void down(struct semaphore *sem)</td>
<td>Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable</td>
</tr>
<tr>
<td>int down_interruptible(struct semaphore *sem)</td>
<td>Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received</td>
</tr>
<tr>
<td>int down_trylock(struct semaphore *sem)</td>
<td>Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable</td>
</tr>
<tr>
<td>void up(struct semaphore *sem)</td>
<td>Releases the given semaphore</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Reader-Writer Semaphores</th>
</tr>
</thead>
<tbody>
<tr>
<td>void init_rwlock(struct rw_semaphore *rwsem)</td>
<td>Initializes the dynamically created semaphore with a count of 1</td>
</tr>
<tr>
<td>void down_read(struct rw_semaphore, *rwsem)</td>
<td>Down operation for readers</td>
</tr>
<tr>
<td>void up_read(struct rw_semaphore, *rwsem)</td>
<td>Up operation for readers</td>
</tr>
<tr>
<td>void down_write(struct rw_semaphore, *rwsem)</td>
<td>Down operation for writers</td>
</tr>
<tr>
<td>void up_write(struct rw_semaphore, *rwsem)</td>
<td>Up operation for writers</td>
</tr>
</tbody>
</table>
Barriers

• In parallel computing, a barrier is a type of synchronization method.
  – A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

Barriers

• To enforce the order in which instructions are executed, Linux provides the memory barrier facility.
  – rmb() - insures that no reads occur across the barrier defined by the place of the rmb() in the code.
  – wmb() – insures that no writes occur across the barrier defined by the place of the wmb() in the code.
  – mb() - provides both a load and store barrier.
Barriers

• Two important points:
  – The barriers relate to machine instructions, namely loads and stores.
  – The \texttt{rmb}, \texttt{wmb}, and \texttt{mb} operations dictate the behavior of both the compiler and the processor.

Linux Memory Barrier Operations

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

\textit{SMP} = symmetric multiprocessor
\textit{UP} = uniprocessor
Synchronization Primitives

• In addition to the concurrency mechanisms of UNIX SVR4, Solaris supports four thread synchronization primitives:
  – Mutual exclusion (mutex) locks
  – Semaphores
  – Condition variables
  – Readers/writer locks

Synchronization Primitives

• Solaris implements these primitives within the kernel for kernel threads; they are also provided in the threads library for user-level threads.
• The initialization functions for the primitives fill in some of the data members.
• Once a synchronization object is created, there are essentially only two operations that can be performed:
  – enter (acquire lock)
  – release (unlock)
Synchronization Primitives

• There are no mechanisms in the kernel or the threads library to enforce mutual exclusion or to prevent deadlock.
• Threads can attempt to access "protected" data and code segments. if the appropriate synchronization primitive isn't used.
• If a thread locks an object and doesn't unlock it, no kernel action is taken.

Synchronization Primitives

• All of the synchronization primitives require the existence of a hardware instruction that allows an object to be tested and set in one atomic operation.
Mutual Exclusion 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**
Condition Variables

- A condition variable is used to wait until a particular condition is true
- Condition variables must be used in conjunction with a mutex lock. This implements a monitor of the type illustrated in the Dining Philosophers Problem

Summary

- Principles of deadlock
  - Reusable/consumable resources
  - Resource allocation graphs
  - Conditions for deadlock
- Deadlock prevention
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
Summary

- Deadlock avoidance
  - Process initiation denial
  - Resource allocation denial
- Deadlock detection
  - Deadlock detection algorithm
  - Recovery
- UNIX concurrency mechanisms
  - Pipes
  - Messages
  - Shared memory
  - Semaphores
  - Signals

Summary

- Linux kernel concurrency mechanisms
  - Atomic operations
  - Spinlocks
  - Semaphores
  - Barriers
- Solaris thread synchronization primitives
  - Mutual exclusion lock
  - Semaphores
  - Readers/writer lock
  - Condition variables