Deadlock Detection

- Two phase process
  - deadlock detection
    - figure out that deadlock occurred
  - deadlock resolution
    - do something to resolve it

**Figure 6.10** Example for Deadlock Detection
Strategies once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may reoccur
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Selection Criteria Deadlocked Processes

- Many criteria to select from, e.g.
  - Least amount of processor time consumed so far
  - Least number of lines of output produced so far
  - Most estimated time remaining
  - Least total resources allocated so far
  - Lowest priority
## Strengths and Weaknesses of the Strategies

<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>Conservative; underestimates resources</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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preemption</td>
<td></td>
<td></td>
<td>Conveniences when applied to resources whose state can be saved and restored easily</td>
<td>Preempts more often than necessary</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource ordering</td>
<td></td>
<td></td>
<td>Fluid to enforce via compile-time checks; Needs no run-time computation since problem is solved in system design</td>
<td>Elevates incremental resource requests</td>
</tr>
<tr>
<td>Avoidance</td>
<td></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 when possible</td>
<td>Involve periodically to test for deadlock</td>
<td>Never delays process initiation; Facilitates on-line handling</td>
<td>Inherent preemption losses</td>
</tr>
</tbody>
</table>

### Dining Philosophers Problem

![Diagram of the Dining Philosophers Problem](image)

*Figure 6.11 Dining Arrangement for Philosophers*
Dining Philosophers Problem

```c
/* 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));
}
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

---

Dining Philosophers Problem

```c
/* 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 6.13 A Second Solution to the Dining Philosophers Problem
Dining Philosophers Problem

```c
monitor dining_controller;  // condition variable for synchronization
shared fork[5];            // availability status of each fork
void get_forks(int pid)     // pid is the philosopher id number
{
    int left = pid;
    int right = (pid+1) % 5;
    //Grant the left fork
    if (fork[left] == false)
    {                                            // queue on condition variable
        fork[left] = true;
        fork[FOREVER] = false;
        if (fork[right] == false)
        {                                        // queue on condition variable
            fork[right] = true;
            fork[FOREVER] = true;
            fork_release_forks(pid);
        }
    }                                           // release the forks
    else
    {
        fork_release_forks(pid);
    }
}

void release_forks(int pid)
{
    fork[left] = false;
    fork[right] = false;
    fork_release_forks(pid);
}

void philosopher(int i = 0 to 4)     // the five philosopher clients
{                                       // the five philosopher clients
    while (true)
    {
        lock;
        get_forks(i);                  // client requests two forks via monitor
        wait_for_fork[i];              // client requests forks via the monitor
        if (fork[i] == false)
        {
            fork[i] = true;
            fork[FOREVER] = false;
            fork_release_forks(i);
        }
        else
        {
            fork_release_forks(i);
        }
    }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor

---

Dining Philosophers Problem

```c
monitor dining_controller;
shared states(thinking, hungry, eating); state[5];
shared needFork[5];  // condition variable

void get_forks(int pid)     // pid is the philosopher id number
{
    state[pid] = hungry;     // announce that I'm hungry
    if (state[pid] == eating)
    {  // state[pid-1] % 5 == eating
        state[pid] = eating;
        state[needFork[pid]] = eating;
    }
    else
    {                                     // wait if either neighbor is eating
        state[pid] = eating;
        state[needFork[pid]] = eating;
    }
    state[pid] = thinking;              // give right (higher) neighbor a chance to eat
    if (state[pid] == hungry)
    {
        state[needFork[pid]] = eating;
        state[needFork[pid-1]] = eating;
    }
    else if (state[pid] == hungry)
    {
        state[needFork[pid]] = eating;
        state[needFork[pid-1]] = eating;
    }
}

void release_forks(int pid)
{
    state[pid] = thinking;
    if (state[pid-1] % 5 == eating)
    {
        state[needFork[pid]] = eating;
        if (state[pid] == eating)
        {
            state[needFork[pid]] = eating;
            state[needFork[pid-1]] = eating;
        }
    }
}

void philosopher(int i = 0 to 4)     // the five philosopher clients
{                                       // the five philosopher clients
    while (true)
    {
        lock;
        get_forks(i);                  // client requests two forks via monitor
        wait_for_fork[i];              // client requests forks via the monitor
        if (fork[i] == false)
        {
            fork[i] = true;
            fork[FOREVER] = false;
            fork_release_forks(i);
        }
        else
        {
            fork_release_forks(i);
        }
    }
}
```

Figure 6.17 Another Solution to the Dining Philosophers Problem Using a Monitor
UNIX Concurrency Mechanisms

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

UNIX Pipes

- used to carry data from one process to another
- one process writes into the pipe
- the other reads from the other end
- essentially FIFO
UNIX Pipes

• Examples
  – ls | pr | lpr
    • pipe ls into the standard input of pr
    • pr pipes its standard output to lpr
    • pr in this case is called a filter
  – ls > filea
  – pr < filea > fileb
    • read input from filea and output to fileb

Signals

• Signals are a facility for handling exceptional conditions similar to software interrupts
• Generated by keyboard interrupt, error in a process, asynchronous events
  – timer
  – job control
• Kill command can generate almost any signal
Linux Kernel Concurrency Mechanisms

- Includes all the mechanisms found in UNIX
- Atomic operations execute without interruption and without interference
Linux Atomic Operations

### Atomic Integer Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atomic.ReadInt(atomic_t *v)</code></td>
<td>Read integer value of <code>v</code></td>
</tr>
<tr>
<td><code>atomic.WriteInt(atomic_t *v, int i)</code></td>
<td>Set the value of <code>v</code> to integer <code>i</code></td>
</tr>
<tr>
<td><code>atomic.AddInt(atomic_t *v)</code></td>
<td>Add <code>i</code> to <code>v</code></td>
</tr>
<tr>
<td><code>atomic.SubInt(atomic_t *v)</code></td>
<td>Subtract <code>i</code> from <code>v</code></td>
</tr>
<tr>
<td><code>atomic.IncInt(atomic_t *v)</code></td>
<td>Add <code>1</code> to <code>v</code></td>
</tr>
<tr>
<td><code>atomic.DecInt(atomic_t *v)</code></td>
<td>Subtract <code>1</code> from <code>v</code></td>
</tr>
<tr>
<td><code>atomic.SubAndTest(atomic_t *v, int i)</code></td>
<td>Subtract <code>i</code> from <code>v</code>; return <code>1</code> if the result is zero; return <code>0</code> otherwise</td>
</tr>
<tr>
<td><code>atomic.AddAndTest(atomic_t *v)</code></td>
<td>Add <code>i</code> to <code>v</code>; return <code>1</code> if the result is negative; return <code>0</code> otherwise</td>
</tr>
<tr>
<td><code>atomic.DecAndTest(atomic_t *v)</code></td>
<td>Subtract <code>1</code> from <code>v</code>; return <code>1</code> if the result is zero; return <code>0</code> otherwise</td>
</tr>
<tr>
<td><code>atomic.IncAndTest(atomic_t *v)</code></td>
<td>Add <code>1</code> to <code>v</code>; return <code>1</code> if the result is zero; return <code>0</code> otherwise</td>
</tr>
</tbody>
</table>

### Atomic Bitmap Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_bit(int nr, void *addr)</code></td>
<td>Set bit <code>nr</code> in the bitmap pointed to by <code>addr</code></td>
</tr>
<tr>
<td><code>clear_bit(int nr, void *addr)</code></td>
<td>Clear bit <code>nr</code> in the bitmap pointed to by <code>addr</code></td>
</tr>
<tr>
<td><code>change_bit(int nr, void *addr)</code></td>
<td>Invert bit <code>nr</code> in the bitmap pointed to by <code>addr</code></td>
</tr>
<tr>
<td><code>get_bit(int nr, void *addr)</code></td>
<td>Get bit <code>nr</code> in the bitmap pointed to by <code>addr</code>; return the old bit value</td>
</tr>
<tr>
<td><code>set_bit_and_test(int nr, void *addr)</code></td>
<td>Set bit <code>nr</code> in the bitmap pointed to by <code>addr</code>; return the old bit value</td>
</tr>
<tr>
<td><code>clear_bit_and_test(int nr, void *addr)</code></td>
<td>Clear bit <code>nr</code> in the bitmap pointed to by <code>addr</code>; return the old bit value</td>
</tr>
<tr>
<td><code>change_bit_and_test(int nr, void *addr)</code></td>
<td>Invert bit <code>nr</code> in the bitmap pointed to by <code>addr</code>; return the old bit value</td>
</tr>
<tr>
<td><code>test_bit(int nr, void *addr)</code></td>
<td>Return the value of bit <code>nr</code> in the bitmap pointed to by <code>addr</code></td>
</tr>
</tbody>
</table>
Linux Spinlocks

- Used for protecting a critical section
- Only one thread at a time can acquire a spinlock, other threads will “spin” on that lock
  - internally, integer local in memory
    - if value is 0, the thread sets it to 1 and enters critical section
  - spinlocks are not very efficient
    - why? waiting threads are in busy-waiting mode
    - use when wait-times are expected to be very short

spin_lock(&lock)
/*critical section*/
spin_unlock(&lock)
Memory Barrier

- A class of instructions
- Enforces that CPU executes memory operations in order

Why would one need to enforce in-order execution?
Memory Barrier Operations

• Consider the following 2 processes
  Proc #1:
    loop: load the value of location y,
    if it is 0 goto loop
    print the value in location x
  Proc #2:
    store the value 55 into location x
    store the value 1 into location y

• What is the output?

Linux Kernel Concurrency Mechanisms

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>sys_rmb</td>
<td>On SMP, provides a rmb () and on UP provides a barrier ()</td>
</tr>
<tr>
<td>sys_wmb</td>
<td>On SMP, provides a wmb () and on UP provides a barrier ()</td>
</tr>
<tr>
<td>sys_mb</td>
<td>On SMP, provides a mb () and on UP provides a barrier ()</td>
</tr>
</tbody>
</table>

SMP = symmetric multiprocessor
UP = uniprocessor
Solaris Thread Synchronization Primitives

- Mutual exclusion (mutex) locks
- Semaphores
- Multiple readers, single writer (readers/writer) locks
- Condition variables

Figure 6.15 Solaris Synchronization Data Structures
<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>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>Mutex</td>
<td>A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore</td>
<td>Owing 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 change notification</td>
<td>A notification of any file system changes that match filter criteria of this object</td>
<td>Change occurs in file system that matches filter criteria of this object</td>
<td>One thread released</td>
</tr>
<tr>
<td>Console input</td>
<td>A text window screen buffer (e.g., used to handle screen I/O for an MS-DOS application)</td>
<td>Input is available for processing</td>
<td>One thread released</td>
</tr>
<tr>
<td>Job</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>Memory resource notification</td>
<td>A notification of change to a memory resource</td>
<td>Specified type of change occurs within physical memory</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: Colored rows correspond to objects that exist for the sole purpose of synchronization.