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 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISL08]

<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; underallocates resources  | Requesting all resources at once | - Works well for processes that perform a single burst of activity  | - Sufficient  
- Delays process initiation  
- Future resource requirements must be known by processes |
|          |                                          |                   | - No preemption necessary                                                        | - Permits more often than necessary                                                 |
|          |                                          |                   | - Efficient when applied to resource whose state can be saved and restored easily | - Does not allow incremental resource requests                                       |
|          |                                          |                   | - Feasible to enforce via compile-time checks                                     | - Needs no run-time computation since problem is solved in system design           |
| Avoidance | Makeup 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  
- Process can be blocked for long periods                                             |
| Detection | Very liberal; requested resources are granted where possible |Invoke periodically to test for deadlock | - Serves delay process initiation  
- Facilitates on-line handshakes                                                      | - Inherent preemption losses                                                        |

Dining Philosophers Problem

![Dining Arrangement for Philosophers](image)
Dining Philosophers Problem

/* 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

/* 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]);
    }
}
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

```
void get_forks(int pid) { /* pid is the philosopher id number */
    int left = pid;
    int right = (pid + 1) % 5;
    // grant the left fork
    if (!forks[left]) {
        forks[left] = true;
        // grant the right fork
        if (!forks[right]) {
            forks[right] = true;
            // signal the right fork
            signal(ForkNeed[right]);
            // signal the left fork
            signal(ForkNeed[left]);
            return;
        }
    }
}
```

```
void release_forks(int pid) { /* pid is the philosopher id number */
    int left = pid;
    int right = (pid + 1) % 5;
    // release the left fork
    if (!forks[left]) {
        forks[left] = false;
        // release the right fork
        if (!forks[right]) {
            forks[right] = false;
            // signal a process waiting on this fork
            signal(ForkNeed[right]);
            // signal a process waiting on this fork
            signal(ForkNeed[left]);
            return;
        }
    }
}
```

```
void philosopher(int pid) { /* the five philosopher clients */
    while (true) {
        while (!forks[pid]) {
            // client requests two forks via monitor
            // get_spartan();
        }
        forks[pid] = true;
        // client releases forks via the monitor
        release_forks(pid);
    }
}
```

---

Dining Philosophers Problem

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

void get_forks(int pid) { /* pid is the philosopher id number */
    state[pid] = hungry;
    if (state[pid-1] == eating) {
        state[pid-1] = thinking;
        // signal a process waiting on this fork
        signal(needFork[pid-1]);
    }
    state[pid] = eating;
}
```

```
void release_forks(int pid) { /* pid is the philosopher id number */
    state[pid] = thinking;
    if (state[pid-1] == eating) {
        state[pid-1] = hungry;
        // signal a process waiting on this fork
        signal(needFork[pid-1]);
    }
    state[pid] = eating;
    // signal a process waiting on this fork
    signal(needFork[pid-1]);
}
```

```
void philosopher(int pid) { /* the five philosopher clients */
    while (true) {
        <think>
        get_forks(pid); /* client requests two forks via monitor */
        // get_spartan();
        release_forks(pid); /* client releases forks via the monitor */
    }
}
```

---

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

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>atomic_read(atomic_t *v)</td>
<td>Read integer value of v</td>
</tr>
<tr>
<td>atomic_set(atomic_t *v, int i)</td>
<td>Set the value of v to integer i</td>
</tr>
<tr>
<td>atomic_add(atomic_t *v, int i)</td>
<td>Add i to v</td>
</tr>
<tr>
<td>atomic_sub(atomic_t *v, int i)</td>
<td>Subtract i from v</td>
</tr>
<tr>
<td>atomic_inc(atomic_t *v)</td>
<td>Add 1 to v</td>
</tr>
<tr>
<td>atomic_dec(atomic_t *v)</td>
<td>Subtract 1 from v</td>
</tr>
<tr>
<td>atomic_sub_and_test(atomic_t *v, int i)</td>
<td>Subtract i from v, return 1 if the result is zero; return 0 otherwise</td>
</tr>
<tr>
<td>atomic_add_negative(atomic_t *v, int i)</td>
<td>Add i to v, return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)</td>
</tr>
<tr>
<td>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>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>

---

Linux Atomic Operations

### Atomic Bitmap Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_bit(int nr, void *addr)</td>
<td>Set bit nr in the bitmap pointed to by addr</td>
</tr>
<tr>
<td>clear_bit(int nr, void *addr)</td>
<td>Clear bit nr in the bitmap pointed to by addr</td>
</tr>
<tr>
<td>change_bit(int nr, void *addr)</td>
<td>Invert bit nr in the bitmap pointed to by addr</td>
</tr>
<tr>
<td>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>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>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>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>
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)
```

Linux Kernel Concurrency Mechanisms

- Spinlocks
  - Used for protecting a critical section

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spin_lock()</td>
<td>Acquires the specified lock, spinning if needed until it is available</td>
</tr>
<tr>
<td>spin_lock_irq()</td>
<td>Like spin_lock, but also disables interrupts on the local processor</td>
</tr>
<tr>
<td>spin_lock_irqsave()</td>
<td>Like spin_lock_irq, but also saves the current interrupt state in flags</td>
</tr>
<tr>
<td>spin_lock_bh()</td>
<td>Like spin_lock, but also disables the execution of all bottom halves</td>
</tr>
<tr>
<td>spin_unlock()</td>
<td>Releases given lock</td>
</tr>
<tr>
<td>spin_unlock_irq()</td>
<td>Releases given lock and enables local interrupts</td>
</tr>
<tr>
<td>spin_unlock_irqsave()</td>
<td>Releases given lock and restores local interrupts to given previous state</td>
</tr>
<tr>
<td>spin_unlock_bh()</td>
<td>Releases given lock and enables bottom halves</td>
</tr>
<tr>
<td>spin_lock_irq()</td>
<td>Initializes given spinlock</td>
</tr>
<tr>
<td>spin_trylock()</td>
<td>Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise</td>
</tr>
<tr>
<td>spin_is_locked()</td>
<td>Returns nonzero if lock is currently held and zero otherwise</td>
</tr>
</tbody>
</table>
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>lmb()</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>__sync_rmb()</td>
<td>On SMP, provides a rmb() and on UP provides a barrier()</td>
</tr>
<tr>
<td>__sync_wmb()</td>
<td>On SMP, provides a wmb() and on UP provides a barrier()</td>
</tr>
<tr>
<td>__sync_lmb()</td>
<td>On SMP, provides a lmb() 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</td>
<td>A notification of any file system changes</td>
<td>Change occurs in file system that matches filter criteria of this object</td>
<td>One thread released</td>
</tr>
<tr>
<td>notification</td>
<td></td>
<td></td>
<td></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</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>notification</td>
<td></td>
<td></td>
<td></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.