CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2019 L12
Synchronization

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
• **Critical section:** shared resource for 2 or more processes

• **Mutex (0 or 1):** for mutual exclusion (lock). Owned by the locking process.

• **Semaphores (any integer value):** general, may be used for counting resources/waiting processes. Shared.
  – Negative: number of processes/threads waiting
  – 0: no waiting threads
  – Positive: no waiting threads, a wait operation would not put the invoking thread in queue.

• **Semaphore implementation**
  – Hardware/software solutions to ensure wait() and signal() atomic.
  – Book uses wait() and signal() as generic terms. For posix semaphores, see documentation.

• **Why not give each philosopher 2 chopsticks?**
  – Nice and elegant solution. Widely used in Chinese restaurants. But takes all the fun away from the problem.
Problems with Semaphores

• Incorrect use of semaphore operations:
  – Omitting of wait (mutex)
    • Violation of mutual exclusion
  – or signal (mutex)
    • Deadlock!

• Solution: Monitors
Monitors

- Monitor: A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
  - Automatically provide mutual exclusion
- Originally proposed for Concurrent Pascal 1975
- Directly supported by Java but not C
Monitors

A generic monitor construct: Only one process may be active in the monitor. Implementation varies by language.

```plaintext
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {......}

    Initialization code (...) { ... }
}
```
Schematic view of a Monitor

Only one process/thread in the Monitor

Provides an easy way to achieve mutual exclusion

But ... we also need a way for processes to block when they cannot proceed.

Refinement next ...
The *condition* construct

- `condition x, y;

- Two operations are allowed on a condition variable:
  - `x.wait()` — a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` — resumes one of processes (if any) that invoked `x.wait()`

- If no `x.wait()` on the variable, then it has no effect on the variable. *Signal is lost.*
Difference between the signal() in semaphores and monitors

• Condition variables in Monitors: Not persistent
  – If a signal is performed and no waiting threads?
    • Signal is simply ignored
  – During subsequent wait operations
    • Thread blocks

• Semaphores
  – Signal increments semaphore value even if there are no waiting threads
    • Future wait operations would immediately succeed!
Monitor with Condition Variables

- Shared data
- Queues associated with x, y conditions
- Operations
- Initialization code

Entry queue
• If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  – Both Q and P cannot execute in parallel. If Q is resumed, then P must wait

• Options include
  – **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  – **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  – Both have pros and cons – language implementer can decide
  – Monitors implemented in **Concurrent Pascal ('75)** compromise
    • P executing signal immediately leaves the monitor, Q is resumed
  – Implemented in other languages including C#, Java
enum {THINKING, HUNGRY, EATING} state[5];

- state[i] = EATING only if
  - state[(i+4)%5] != EATING && state[(i+1)%5] != EATING

- condition self[5]
  - Delay self when HUNGRY but unable to get chopsticks

**Sequence of actions**

- Before eating, must invoke pickup()
  - May result in suspension of philosopher process
  - After completion of operation, philosopher may eat

  think
  DiningPhilosophers.pickup(i);
  eat
  DiningPhilosophers.putdown(i);
  think
enum {THINKING, HUNGRY, EATING} state[5];
The pickup() and putdown() operations

```java
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING} state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i); //on next slide
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test(((i + 4) % 5);
        test(((i + 1) % 5);
    }

    void test (int i) {
        if (((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING;
            self[i].signal ();
        }
    }

    initialization_code() { 
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Suspend self if unable to acquire chopstick

Check to see if person on left or right can use the chopstick

Eat only if HUNGRY and Person on Left AND Right are not eating

Signal a process that was suspended while trying to eat
Possibility of starvation

• Philosopher i can starve if eating periods of philosophers on left and right overlap
• Possible solution
  – Introduce new state: STARVING
  – Chopsticks can be picked up if no neighbor is starving
    • Effectively wait for neighbor’s neighbor to stop eating
    • REDUCES concurrency!
Monitor Implementation Mutual Exclusion

For each monitor

• Semaphore mutex initialized to 1
• Process must execute
  – wait(mutex) : Before entering the monitor
  – signal(mutex): Before leaving the monitor
Monitor Implementation Using Semaphores

• **Variables**

  ```
  semaphore mutex;   // (initially = 1) allows only one process to inside
  semaphore next;   // (initially = 0) causes signaler to sleep
  int next_count = 0; num of sleepers since they signalled
  ```

• Each procedure $F$ will be replaced the compiler by

  ```
  wait(mutex);
  ...
  body of $F$;
  ...
  if (next_count > 0)
    signal(next)
  else
    signal(mutex);
  ```

• Mutual exclusion within a monitor is ensured
Monitor Implementation – Condition Variables

• For each condition variable $x$, we have:

```c
semaphore x_sem; // (initially = 0) causes caller of wait to sleep
int x_count = 0; // number of sleepers on condition
```

• The operations $x.wait$ and $x.signal$ can be implemented as:

<table>
<thead>
<tr>
<th>The operation $x.wait$ can be implemented as:</th>
<th>The operation $x.signal$ can be implemented as:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_count++$;</td>
<td>if ($x_count &gt; 0$) {</td>
</tr>
<tr>
<td>if ($next_count &gt; 0$)</td>
<td>$next_count++;</td>
</tr>
<tr>
<td>$signal(next)$;</td>
<td>$signal(x_sem);$</td>
</tr>
<tr>
<td>else</td>
<td>$wait(next);$</td>
</tr>
<tr>
<td>$signal(mutex);$</td>
<td>$wait(x_sem);$</td>
</tr>
<tr>
<td>$wait(x_sem);$</td>
<td>$next_count--;</td>
</tr>
<tr>
<td>$x_count--;</td>
<td>}</td>
</tr>
</tbody>
</table>
Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() is executed, which should be resumed?

- FCFS frequently not adequate

- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next
Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource.

```java
R.acquire(t);
... access the resource;
... R.release;
```

Where R is an instance of type `ResourceAllocator`.

A monitor based solution next.
A Monitor to Allocate Single Resource

```java
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }

    void release() {
        busy = FALSE;
        x.signal();
    }

    initialization code() {
        busy = FALSE;
    }
}
```

Sleep, Time used to prioritize waiting processes

Wakes up one of the processes
Java Synchronization

• For simple synchronization Java provides the synchronized keyword
  – synchronizing methods
    public synchronized void increment() { c++; }
  – synchronizing blocks
    synchronized(this) {
      lastName = name;
      nameCount++;
    }

• wait() and notify() allows a thread to wait for an event. A call to notify. all() allows all threads that are on wait() with the same lock to be released

• For more sophisticated locking mechanisms, starting from Java 5, the package java.concurrent.locks provides additional locking
Java Synchronization

- Each object automatically has a monitor (mutex) associated with it.
- When a method is synchronized, the runtime must obtain the lock on the object's monitor before execution of that method begins (and must release the lock before control returns to the calling code).
- `wait()` and `notify()` allows a thread to wait for an event.
  - `wait()`: Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
  - `notify()`: Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.
- A call to `notifyAll()` allows all threads that are on `wait()` with the same lock to be released.
Java Synchronization: Dining Philosiphers

<table>
<thead>
<tr>
<th>public synchronized void pickup(int i) throws InterruptedException {</th>
<th>private synchronized void test(int i) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>setState(i, State.HUNGRY);</td>
<td>if (state[left(i)] != State.EATING &amp;&amp; state[right(i)] != State.EATING &amp;&amp; state[i] == State.HUNGRY)</td>
</tr>
<tr>
<td>test(i);</td>
<td>{</td>
</tr>
<tr>
<td>while (state[i] != State.EATING) {</td>
<td>setState(i, State.EATING);</td>
</tr>
<tr>
<td>this.wait();</td>
<td>// Wake up all waiting threads</td>
</tr>
<tr>
<td>// Recheck condition in loop,</td>
<td></td>
</tr>
<tr>
<td>// since we might have been notified</td>
<td></td>
</tr>
<tr>
<td>// when we were still hungry</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>this.notifyAll();</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>public synchronized void putdown(int i) {</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>setState(i, State.THINKING);</td>
<td></td>
</tr>
<tr>
<td>test(right(i));</td>
<td></td>
</tr>
<tr>
<td>test(left(i));</td>
<td></td>
</tr>
</tbody>
</table>
Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads
Solaris Synchronization

• Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing

• Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  – Starts as a standard semaphore spin-lock
  – If lock held, and by a thread running on another CPU, spins
  – If lock held by non-run-state thread, block and sleep waiting for signal of lock being released

• Uses **condition variables**

• Uses **readers-writers** locks when longer sections of code need access to data

• Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  – Turnstiles are per-lock-holding-thread, not per-object

• Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile
Windows Synchronization

• Uses interrupt masks to protect access to global resources on uniprocessor systems
• Uses **spinlocks** on multiprocessor systems
  – Spinlocking-thread will never be preempted
• Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  – **Events**
    • An event acts much like a condition variable
  – Timers notify one or more thread when time expired
  – Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)
Linux Synchronization

• Linux:
  – Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  – Version 2.6 and later, fully preemptive

• Linux provides:
  – Semaphores
  – atomic integers
  – spinlocks
  – reader-writer versions of both

• On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption
Pthreads Synchronization

• Pthreads API is OS-independent

• It provides:
  – mutex locks
  – condition variable

• Non-portable extensions include:
  – read-write locks
  – spinlocks
Alternative Approaches

• Transactional Memory

• OpenMP

• Functional Programming Languages
A memory transaction is a sequence of read-write operations to memory that are performed atomically without the use of locks.

```c
void update(){
    atomic{
        /* modify shared data*/
    }
}
```

May be implemented by hardware or software.
OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```c
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.
CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2018

Deadlock

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
Chapter 7: Deadlocks

• System Model
• Deadlock Characterization
• Methods for Handling Deadlocks
  – Deadlock Prevention
  – Deadlock Avoidance resource-allocation
  – Deadlock Detection
  – Recovery from Deadlock
Can you give a real life example of a deadlock?
A Kansas Law

- Early 20th century Kansas Law
  - “When *two trains approach each other at a crossing*, both shall come to a full stop and neither shall start up again until the other has gone”

- *Story of the two silly goats*: Aesop 6th cent BCE?
A contemporary example
System Model

• System consists of resources
• Resource types $R_1, R_2, \ldots, R_m$
  \( CPU \text{ cycles, memory space, I/O devices} \)
• Each resource type $R_i$ has $W_i$ instances.
• Each process utilizes a resource as follows:
  - request
  - use
  - release
Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set \( \{P_0, P_1, ..., P_n\} \) of waiting processes such that \( P_0 \) is waiting for a resource that is held by \( P_1 \), \( P_1 \) is waiting for a resource that is held by \( P_2 \), ..., \( P_{n-1} \) is waiting for a resource that is held by \( P_n \), and \( P_n \) is waiting for a resource that is held by \( P_0 \).
Deadlock with Mutex Locks

• Deadlocks can occur via system calls, locking, etc.
• See example
  – Dining Philosophers: each get the right chopstick first
  – box in text page 318 (we saw something similar earlier)

Let \( S \) and \( Q \) be two semaphores initialized to 1

\[
\begin{align*}
P_0 & \quad P_1 \\
wait(S); & \quad wait(Q); \\
wait(Q); & \quad wait(S); \\
\vdots & \quad \vdots \\
signal(S); & \quad signal(Q); \\
signal(Q); & \quad signal(S); \\
\end{align*}
\]

\( P_0 \) executes \( wait(s) \), \( P_1 \) executes \( wait(Q) \)

P0 must wait till \( P_1 \) executes \( signal(Q) \)

P1 must wait till \( P_0 \) executes \( signal(S) \)  Deadlock!
A set of vertices $V$ and a set of edges $E$.

- $V$ is partitioned into two types:
  
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system
  
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system

- **request edge** – directed edge $P_i \rightarrow R_j$

- **assignment edge** – directed edge $R_j \rightarrow P_i$
• Process

• Resource Type with 4 instances

• $P_i$ requests instance of $R_j$

• $P_i$ is holding an instance of $R_j$
Example of a Resource Allocation Graph

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Does a deadlock exist here?

P1 holds an instance of R2, and is requesting R1...

P3 will eventually be done with R3, letting P2 use it.

Thus P2 will be eventually done, releasing R1. ...