CS370 Operating Systems

Colorado State University
Yashwant K Malaiya
Fall 2019 L12
Synchronization

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
Notes: we are using the terms in a generic way. There are specific implementations for POSIX and Java.

- **Critical section**: shared resource for 2 or more processes
- **Lock**: Critical section is locked by one processes at a time. Software/hardware
- **Mutex (0 or 1)**: for mutual exclusion (lock). Owned by the locking process which acquires/releases by wait( )/signal ( ).
- **Semaphores (any integer value)**: general, may be used for counting resources/waiting processes. Shared.
  - 0: no waiting threads
  - Positive: no waiting threads, a wait operation would not put the invoking thread in queue.
  - Negative: number of processes/threads waiting
- **Semaphore implementation**
  - Hardware/software solutions to ensure wait() and signal( ) atomic.
Project

• See **Document**: Schedule/Proj Proposal or Canvas/Assignments

• **Choices**: Research (topics provided) or development (IoT). Some research/original thinking required for either.

• **Deadlines**: subject to revision.
  – **D1.** Team composition and idea proposal, Thurs 10/06/2019
  – **D2.** Progress report, Thurs 10/31/2019
  – **D3.** Slides and final reports, Thurs 12/05/2019;
  – **D4A.** Posters session for Option A Tu 12/10/2019 (TBD). Also Submit poster electronically on Canvas same day.
  – **D4B.** Demos of Option B projects to be scheduled during 12/9/2019-12/13/2019.
  – **D5:** Peer Reviews due 12/14
Classical Problems of Synchronization

• Classical problems
  – Bounded-Buffer Problem
  – Readers and Writers Problem
  – Dining-Philosophers Problem

• Bounded buffer
  – n buffers, each can hold one item
  – Binary semaphore (mutex)
    • Provides mutual exclusion for accesses to buffer pool
    • Initialized to 1
  – Counting semaphores
    • empty: Number of empty slots available, Initialized to n
    • full: Number of filled slots available n, Initialized to 0
Readers-Writers Problem

• A data set is shared among a number of concurrent processes
  – Readers – only read the data set; they do not perform any updates
  – Writers – can both read and write

• Problem
  – allow multiple readers to read at the same time
  – Only one single writer can access the shared data at the same time. No readers permitted when writer is accessing the data.

• One Mutex, two counting semaphores: empty/full
Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat,
  - then release both when done
- Each chopstick is a semaphore
  - Grab by executing `wait()`
  - Release by executing `signal()`
- Shared data
  - Bowl of rice (data set)
  - Semaphore `chopstick [5]` initialized to 1
Dining-Philosophers Problem

Plato, Confucius, Socrates, Voltaire and Descartes
Dining-Philosophers Problem Algorithm: Simple solution?

• The structure of Philosopher $i$:
  
  ```
  do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5 ] );

    // eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5 ] );

    // think
  } while (TRUE);
  ```

• What is the problem with this algorithm?
  – If all of them pick up the the left chopstick first - Deadlock
• Deadlock handling
  – Allow at most 4 philosophers to be sitting simultaneously at the table (with the same 5 forks).
  – Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  – Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
Related classes

• Classes that follow CS370
  – CS455 Distributed Systems  Spring
  – CS457 Networks  Fall
  – CS470 Computer Architecture  Spring
  – CS475 Parallel Programming  Fall
  – CS435: Introduction to Big Data  Spring
Problems with Semaphores

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

• Solution: Monitors
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
• Queues:
  - for entry
  - for each condition
• 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 (...) { ... }
}
```
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:
  - \( x\text{.wait}() \) — a process that invokes the operation is suspended until \( x\text{.signal}() \)
  - \( x\text{.signal}() \) — resumes one of processes (if any) that invoked \( x\text{.wait}() \)
    - If no \( x\text{.wait}() \) on the condition variable, then it has no effect on the variable. *Signal is lost.*

Compare with semaphore. Here no integer value is associated.
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
Condition Variables Choices

• 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

```c
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
- Eat only if HUNGRY and Person on Left AND Right are not eating
- Check to see if person on left or right can use the chopstick
- 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!
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:

\[
\text{semaphore } x\_\text{sem}; \quad // \text{(initially } = 0\text{) causes caller of wait to sleep}
\]
\[
\text{int } x\_\text{count} = 0; \quad // \text{number of sleepers on condition}
\]

• The operations \( x.\text{wait} \) and \( x.\text{signal} \) can be implemented as:

The operation \( x.\text{wait} \) can be implemented as:

\[
\begin{align*}
&x\_\text{count}++;
&\text{if } (x\_\text{count} > 0) \\
&\quad \text{signal}(\text{next});
&\text{else}
&\quad \text{signal}(\text{mutex});
&\text{wait}(x\_\text{sem});
&x\_\text{count}--;)
\end{align*}
\]

The operation \( x.\text{signal} \) can be implemented as:

\[
\begin{align*}
\text{if } (x\_\text{count} > 0) \{ \\
&x\_\text{count}++;
&\text{signal}(x\_\text{sem});
&\text{wait}(\text{next});
&x\_\text{count}--;)
\end{align*}
\]
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

\[
\begin{align*}
R.\text{acquire}(t); \\
\ldots \\
\text{access the resource}; \\
\ldots \\
R.\text{release};
\end{align*}
\]

Where R is an instance of type \texttt{ResourceAllocator}

A monitor based solution next.
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;
    }
}
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

https://www.baeldung.com/java-wait-notify
Java Synchronization: Dining Philosophers

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

Colorado State University
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 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 8: Deadlocks

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance resource-allocation
  - Deadlock Detection
  - Recovery from Deadlock
Deadlock

• Can you give a real life example of a deadlock?
A Kansas Law

• Early 20\textsuperscript{th} 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”

• \textit{Story of the two silly goats}: Aesop 6\textsuperscript{th} cent BCE?
A contemporary example