CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 1018 L12
Synchronization

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
• Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
• Semaphore $S$ – integer variable
• Can only be accessed via two **indivisible (atomic)** operations
  
  — **wait()** and **signal()**
  
  • Originally called $P()$ and $V()$ based on Dutch words
• Definition of the **wait() operation**

  ```
  wait(S) {
    while (S <= 0) // busy wait
      S--;
  }
  ```
• Definition of the **signal() operation**

  ```
  signal(S) {
    S++;
  }
  ```

  Binary semaphore: When $s$ is 0 or 1, it is a mutex lock

  Waits until another process makes $S=1$
Wait(S) and Signal (S)

Process 0

Wait(S)
Critical section
Signal (S)

Semaphore S
S = 1
S = 0
Locked by Process 1
S = 0
S = 1

Process 1
Wait (S)
Busy waiting
Gets lock, S -
Critical section
Signal (S)

Locked by Process 1
Semaphore Usage (Review)

- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- **Counting semaphore** – integer value can range over an unrestricted domain
  - **Controls access to a finite set of resources**
  - Initialized to the number of resources
  - Usage:
    - **Wait** (S): to use a resource
    - **Signal** (S): to release a resource
  - When all resources are being used: S == 0
    - Block until S > 0 to use the resource
```c
wait(semaphore *S) {
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }
}

typedef struct {
    int value;  
    struct process *list;  
} semaphore;

signal(semaphore *S) {
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }
}
```

If value < 0, abs(value) is the number of waiting processes.
Classical Problems of Synchronization
Classical Problems of Synchronization

• Classical problems used to test newly-proposed synchronization schemes
  – Bounded-Buffer Problem
  – Readers and Writers Problem
  – Dining-Philosophers Problem

• Monitors
Bounded-Buffer Problem

- $n$ 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
Bounded-Buffer : Note

• Producer and consumer must be ready before they attempt to enter critical section

• Producer readiness?
  – When a slot is available to add produced item
    • wait(empty)
      – empty is initialized to n

• Consumer readiness?
  – When a producer has added new item to the buffer
    • wait(full)
      – full initialized to 0
The structure of the producer process

do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);  // wait till slot available
    wait(mutex);  // Allow producer OR consumer to (re)enter critical section
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex); // Allow producer OR consumer to (re)enter critical section
    signal(full);  // Signal consumer that a slot is available
} while (true);
The structure of the consumer process

Do {
    wait(full);  // wait till slot available for consumption
    wait(mutex); // Only producer OR consumer can be in critical section
    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex); // Allow producer OR consumer to (re)enter critical section
    signal(empty); // signal producer that a slot is available to add
    ...
    /* consume the item in next_consumed */
    ...
} while (true);
Readers-Writers Problem

• Shared Data
  – Data set
  – Semaphore `rw_mutex` initialized to 1 (mutual exclusion for writer)
  – Semaphore `mutex` initialized to 1 (mutual exclusion for `read_count`)
  – Integer `read_count` initialized to 0 (how many readers?)
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.

• Several variations of how readers and writers are considered – all involve some form of priorities
Readers-Writers Problem (Cont.)

• The structure of a writer process

    do {
        wait(rw_mutex);
        ...
        /* writing is performed */
        ...
        signal(rw_mutex);
    } while (true);

When: writer in critical section and if n readers waiting:
- 1 reader is queued on rw_mutex
- (n-1) readers queued on mutex
Readers-Writers Problem (Cont.)

• The structure of a reader process

```c
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    ... /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

- **mutex for mutual exclusion to readcount**
- **Cannot read if writer is writing**
- **When: writer in critical section and if n readers waiting 1 is queued on rw_mutex (n-1) queued on mutex**
Readers-Writers Problem Variations

- **First** variation – no reader kept waiting unless writer has already obtained permission to use shared object
- **Second** variation – once writer is ready, it performs the write ASAP, i.e. if a writer is waiting, no new readers may start.
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks
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 $i$:

```c
    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 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

- 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
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

- Entry queue
- Shared data
  - Queues associated with $x$, $y$ conditions
  - Operations
  - Initialization code
If process P invokes \texttt{x.signal()}, and process Q is suspended in \texttt{x.wait()}, what should happen next?

- Both Q and P cannot execute in parallel. If Q is resumed, then P must wait.

Options include:

- \textbf{Signal and wait} – P waits until Q either leaves the monitor or it waits for another condition.
- \textbf{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 \textit{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 \{\text{THINKING, HUNGRY, EATING}\} \text{state}[5];
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));
    }
}
test() to see if philosopher I can eat

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;
}

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

```c
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

```c
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:

<table>
<thead>
<tr>
<th>The operation $x$.\text{wait}$ can be implemented as:</th>
<th>The operation $x$.\text{signal}$ can be implemented as:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_\text{count}++$;</td>
<td></td>
</tr>
<tr>
<td>if ($next_\text{count} &gt; 0$)</td>
<td></td>
</tr>
<tr>
<td>$\quad$ signal($next$);</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>$\quad$ signal($mutex$);</td>
<td></td>
</tr>
<tr>
<td>$\quad$ wait($x_\text{sem}$);</td>
<td></td>
</tr>
</tbody>
</table>
| $x\_\text{count}--$; | $\quad$ if ($x\_\text{count} > 0$) {
| $\quad$ $\quad$ $next\_\text{count}++$;
| $\quad$ $\quad$ $\quad$ signal($x\_\text{sem}$);
| $\quad$ $\quad$ $\quad$ wait($next$);
| $\quad$ $\quad$ $\quad$ $next\_\text{count}--$;
| } |
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

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

Where R is an instance of type 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;
    }
}

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