COS 318: Operating Systems

Semaphores, Monitors and Condition Variables

Jaswinder Pal Singh
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)
Today’s Topics

- Producer-consumer problem
- Semaphores
- Monitors
- Barriers
Revisit Mutex

- Mutex can solve the critical section problem
  Acquire( lock );
  *Critical section*
  Release( lock );

- Use Mutex primitives to access shared data structures
  E.g. shared “count” variable
  Acquire( lock );
  count++;
  Release( lock );

- Are mutex primitives adequate to solve all problems?
Can we solve this problem with Mutex primitives?
Use Mutex, Block and Unblock

Producer:
while (1) {
    produce an item
    if (count == N)
        Block();
    Insert item in buffer
    Acquire(lock);
    count++;
    Release(lock);
    if (count == 1)
        Unblock(Consumer);
}

Consumer:
while (1) {
    remove an item from buffer
    if (!count)
        Block();
    Acquire(lock);
    count--;
    Release(lock);
    if (count == N-1)
        Unblock(Producer);
    consume an item
}

count = 4
N = 12

Does this work?
Use Mutex, Block and Unblock

Producer:
while (1) {
    **produce an item**
    if (count == N)
        Block();
    **Insert item in buffer**
    Acquire(lock);
    count++;
    Release(lock);
    if (count == 1)
        Unblock(Consumer);
}

Consumer:
while (1) {
    **context switch**
    Block();
    remove an item from buffer
    Acquire(lock);
    count--;
    Release(lock);
    if (count == N-1)
        Unblock(Producer);
    consume an item
}

- Race condition!
- Ultimately, both block and never wake up
- Lost the unblock; any way to “remember” them?
Semaphores (Dijkstra, 1965)

- Initialization
  - Initialize to an integer value
- Never access the value directly after that, only through P(), V()
  - The operations P() and V() are atomic operations
  - System implements the atomicity
- If positive value, think of value as keeping track of how many ‘resources’ or “un-activated unblocks” are available
- If negative, tracks how many threads are waiting for a resource or unblock
Semaphores (Dijkstra, 1965)

- **P (or Down or Wait or “Probeer”) definition**
  - Atomic operation
  - Decrement value, and if less than zero block
  - Or: Wait for semaphore to become positive and then decrement
    
    ```
    P(s) {
      if (--s < 0)
        block(s);
    }
    ```

- **V (or Up or Signal) definition**
  - Atomic operation
  - Increment semaphore
  - Or increment semaphore, and if non-positive (which means at least one thread is blocked waiting on the semaphore) then unblock a thread
    
    ```
    V(s) {
      if (s <= 0)
        unblock(s);
      s++;
    }
    ```
Bounded Buffer with Semaphores

Producer:

while (1) {
    produce an item
    P(emptyCount);
    P(mutex);
    put item in buffer
    V(mutex);
    V(fullCount);
}

Consumer:

while (1) {
    P(fullCount);
    P(mutex);
    take an item from buffer
    V(mutex);
    V(emptyCount);
    consume item
    }
Uses of Semaphores in this Example

◆ Event sequencing
  ● Don’t consume if buffer empty, wait for something to be added
  ● Don’t add if buffer full, wait for something to be removed

◆ Mutual exclusion
  ● Avoid race conditions on shared variables
Example: Interrupt Handler

```c
Init(s, 0);
```

```
Device thread
while (1) {
    P(s);
    Acquire(m);
    ...  
    deal with interrupt 
    ... 
    Release(m);
}
```

```
Interrupted Thread
```

```
Interrupt handler
... 
V(s); 
... 
```

```
Interrupt
... 
```
Bounded Buffer with Semaphores (again)

```
producer() {
    while (1) {
        produce an item
        P(emptyCount);
        P(mutex);
        put the item in buffer
        V(mutex);
        V(fullCount);
    }
}

c consumer() {
    while (1) {
        P(fullCount);
        P(mutex);
        take an item from buffer
        V(mutex);
        V(emptyCount);
        consume the item
    }
}
```
Does Order Matter?

```
producer() {
    while (1) {
        produce an item
        P(mutex);
        P(emptyCount);
        put the item in buffer
        V(mutex);
        V(fullCount);
    }
}

consumer() {
    while (1) {
        P(fullCount);
        P(mutex);
        take an item from buffer
        V(mutex);
        V(emptyCount);
        consume the item
    }
}
```
Monitor: Hide Mutual Exclusion

- Brinch-Hansen (73), Hoare (74)
- Procedures are mutually exclusive

Queue of waiting processes trying to enter the monitor

Shared data

procedures
Condition Variables in A Monitor

- **Wait(condition)**
  - Block on “condition”
- **Signal(condition)**
  - Wakeup a blocked process on “condition”

Queues associated with x, y conditions

Entry queue

Shared data

procedures
Producer-Consumer with Monitors

**procedure** Producer
begin
  while true do
  begin
    produce an item
    ProdCons.Enter();
  end;
end;

**procedure** Consumer
begin
  while true do
  begin
    ProdCons.Remove();
    consume an item;
  end;
end;

**monitor** ProdCons
condition full, empty;

**procedure** Enter;
begin
  if (buffer is full)
  wait(full);
  put item into buffer;
  if (only one item)
  signal(empty);
end;

**procedure** Remove;
begin
  if (buffer is empty)
  wait(empty);
  remove an item;
  if (buffer was full)
  signal(full);
end;
Run the signaled thread immediately and suspend the current one (Hoare)

What if the current thread has more things to do?

```plaintext
monitor ProdCons
  condition full, empty;

procedure Enter;
begin
  if (buffer is full)
    wait(full);
  put item into buffer;
  if (only one item)
    signal(empty);
  end;
end;

procedure Remove;
begin
  if (buffer is empty)
    wait(empty);
  remove an item;
  if (buffer was full)
    signal(full);
  end;
```
Hansen’s Signal Implementation (MOS p 137)

- Signal must be the last statement of a monitor procedure
- Exit the monitor
- Any issue with this approach?

```plaintext
monitor ProdCons
  condition full, empty;

procedure Enter;
begin
  if (buffer is full)
    wait(full);
  put item into buffer;
  if (only one item)
    signal(empty);
end;

procedure Remove;
begin
  if (buffer is empty)
    wait(empty);
  remove an item;
  if (buffer was full)
    signal(full);
end;
```
Mesa Signal Implementation

◆ Continues its execution

```plaintext
if (only one item)
    signal (empty);
    something else
end;
```


◆ This is easy to implement!

◆ Issues?
Evolution of Monitors

- **Brinch-Hansen (73) and Hoare Monitor (74)**
  - Concept, but no implementation
  - Requires Signal to be the last statement (Hansen)
  - Requires relinquishing CPU to signaler (Hoare)

- **Mesa Language (77)**
  - Monitor in language, but signaler keeps mutex and CPU
  - Waiter simply put on ready queue, with no special priority

- **Modula-2+ (84) and Modula-3 (88)**
  - Explicit LOCK primitive
  - Mesa-style monitor

- **Pthreads (95)**
  - Started standard effort around 1989
  - Defined by ANSI/IEEE POSIX 1003.1 Runtime library

- **Java threads**
  - James Gosling in early 1990s without threads
  - Use most of the Pthreads primitives
Barrier Synchronization

- Thread A and Thread B want to meet at a particular point.
- The one to get there first waits for the other one to reach that point before proceeding.
- Then both go forward.
Using Semaphores as A Barrier

- Use two semaphores?
  
  \[
  \text{init}(s1, 0); \\
  \text{init}(s2, 0);
  \]

- What about more than two threads?

Thread A

```
... \\
V(s1); \\
P(s2); \\
... 
```

Thread B

```
... \\
V(s2); \\
P(s1); \\
... 
```
Barrier Primitive

- **Functions**
  - Take a barrier variable
  - Broadcast to n-1 threads
  - When barrier variable has reached n, go forward

- **Hardware support on some parallel machines**
  - Multicast network
  - Counting logic
  - User-level barrier variables
Equivalence

- **Semaphores**
  - Good for signaling and fine for simple mutex
  - Not good for mutex in general, since easy to introduce a bug

- **Monitors**
  - Good for scheduling and mutex
  - Maybe costly for simple signaling
# The Big Picture

<table>
<thead>
<tr>
<th></th>
<th>OS codes and concurrent applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High-Level</strong></td>
<td>Mutex</td>
</tr>
<tr>
<td><strong>Atomic API</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Low-Level</strong></td>
<td>Load/store</td>
</tr>
<tr>
<td><strong>Atomic Ops</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interrupts (I/O, timer)</td>
</tr>
</tbody>
</table>
Summary

- Mutex alone are not enough
- Semaphores
- Monitors
- Mesa-style monitor and its idiom
- Barriers