COS 318: Operating Systems

Synchronization: Semaphores, Monitors and Condition Variables
Today’s Topics

- Mutex Isn’t Enough
- Semaphores
- Condition Variables
- 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 synchronization problems?
Producer-Consumer (Bounded Buffer) Problem

Produce: while (1) {
    produce an item
    Insert item in buffer
    count++;
}

Consumer: while (1) {
    remove an item from buffer
    count--;
    consume an item
}

count = 4
N = 12

✿ Can we solve this problem with Mutex primitives?
Producer-Consumer (Bounded Buffer) Problem

Producer:

```c
while (1) {
    produce an item
    Acquire(lock);
    Insert item in buffer
    count++;
    Release(lock);
}
```

Consumer:

```c
while (1) {
    Acquire(lock);
    remove an item from buffer
    count--;
    Release(lock);
    consume an item
}
```

- Does this work?

N = 12

count = 4
Limitations of Locks

- Provide mutual exclusion: only one process/thread can be in the critical section at a time
- Do not provide ordering or sequencing (aka event synchronization)
  - Who gets to be in critical section first?
  - How does thread A wait for thread B (or C, D, E) to do X before A does Y?
    - How does producer know when to stop inserting, or consumer when to remove?
Use Mutex, Block and Unblock

Producer:

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

Consumer:

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

- Use block/unblock for ordering
- 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) {
    if (!count)
        {context switch}
    Block();
    remove an item from buffer
    Acquire(lock);
    count--;
    Release(lock);
    if (count == N-1)
        Unblock(Producer);
    consume an item
}

- Ultimately, both block and never wake up
- Lost the unblock; any way to “remember” them?
Limitations of Locks and Block/Unblock

- Need some way of counting or remembering number of events

- Need additional synchronization mechanisms
  - Semaphores
  - Condition Variables
  - Monitors
  - (Higher level constructs composed from these)
Semaphores (Dijkstra, 1965)

- A semaphore is a synchronization variable that contains an integer value
  - Cannot access the integer value directly (only via semaphore operations)
  - Initialized to some integer value
  - Supports two atomic operations other than initialization
    - P(), (or down() or wait()); P for Proberen
    - V() (or up() or signal()); V for Verhogen

- 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

- Provides ordering and counting (of ‘surplus’ events/resources)
**Semaphores (Dijkstra, 1965)**

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

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

**Producer:**
```
while (1) {
    produce an item
    P(emptCount);

    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(emptCount);
    consume item
}
```

- Initialization: emptyCount = N; fullCount = 0
- Are $P\,(\text{mutex})$ and $V\,(\text{mutex})$ necessary?
Uses of Semaphores in this Example

- **For Event sequencing:** emptyCount, fullCount
  - Don’t consume if buffer empty, wait for something to be added
  - Don’t add if buffer full, wait for something to be removed

- **For Mutual exclusion:** mutex
  - Avoid race conditions on shared variables
Example: Interrupt Handler

init(s, 0);

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

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

Interrupted Thread
    ...
    Interrupt
    ...

Bounded Buffer with Semaphores (again)

```c
producer() {
    while (1) {
        produce an item
        P(emptyCount);
        P(mutex);
        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
    }
}
```
Does Order Matter?

```c
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
    }
}
```

➢ Q: What problem can happen if the order of
P(mutex) and P(emptyCount) are reversed as here?
Different Example: Waiting in Critical Section

- A lock provides mutual exclusion to the shared data
- Rules for using a lock:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock is initially free.

- Simple example: a synchronized queue

```cpp
bool tryInsert()
{
    lock.Acquire();  // lock before use
    … put item on queue;  // ok to access
    lock.Release();  // unlock after done
    return success;
}

bool tryRemove()
{
    lock.Acquire();
    if something on queue   // can we wait?
        remove it;
    lock->Release();
    return success;
}
```
Condition Variables

- Make `tryRemove` wait until something is on the queue?
  - Can’t just sleep while holding the lock
  - Key idea: make it possible to go to sleep inside critical section, by atomically releasing lock at same time we go to sleep.

- **Condition variable**: enables a *queue of threads waiting for something inside a critical section.*
  - **Wait()** --- Release lock, go to sleep, re-acquire when woken
    - release lock and going to sleep is **atomic**
  - **Signal()** --- Wake up a waiter, if any
  - **Broadcast()** --- Wake up all waiters
Synchronized Queue

- **Rule**: must hold lock when doing condition variable operations

AddToQueue()
{
    lock.acquire();
    put item on queue;
    condition.signal();
    lock.release();
}

RemoveFromQueue()
{
    lock.acquire();

    while nothing on queue
        condition.wait(&lock);
        // release lock; go to
        // sleep; reacquire lock
        // when woken

    remove item from queue;
    lock.release();
    return item;
}
**Condition variable design pattern**

```java
methodThatWaits() {
    lock.acquire();

    // Read/write shared state
    while (!testSharedState()) {
        cv.wait(&lock);
    }

    // Read/write shared state
    lock.release();
}

methodThatSignals() {
    lock.acquire();

    // Read/write shared state
    if (testSharedState() is now true) {
        cv.signal(&lock);
    }

    // Read/write shared state
    lock.release();
}
```
Condition variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is synchronization FOR shared state
  - Remember: ALWAYS hold lock when accessing shared state

- Unlike semaphore, condition variable is memory-less
  - If signal when no one is waiting, no op
  - If signal after a wait is posted, a waiter wakes up

- Wait atomically releases lock
Structured synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
- Add locks to object/module
  - Obtain lock on start to every method/procedure
  - Release lock when finished
- If need to wait for something inside critical section
  - while(needToWait()) { condition.Wait(lock); }
- If do something that should wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting
Monitors

- **Monitor definition:**
  - *a lock and zero or more condition variables for managing concurrent access to shared data*

- **Monitors make things easier:**
  - “locks” for mutual exclusion
  - “condition variables” for scheduling constraints
Monitors Embedded in Languages

- High-level data abstraction that unifies handling of:
  - Shared data, operations on it, synchronization and scheduling
    - All operations on data structure have single (implicit) lock
    - An operation can relinquish control and wait on a condition

```java
// only one process at time can update instance of Q
class Q {
    int head, tail;  // shared data
    void enqueue(v) { locked access to Q instance }
    int dequeue() { locked access to Q instance }
}
```

- Java from Sun; Mesa/Cedar from Xerox PARC

- Monitors are easy and safe
  - Compiler can check, lock is implicit (cannot be forgotten)
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.Add();
    end;
  end;
end;

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

monitor ProdCons
  condition full, empty;

procedure Add;
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;
Hoare’s Signal Implementation (MOS p137)

- Run the signaled thread immediately and suspend the current one (Hoare)

- What if the current thread has more things to do?

```plaintext
ten 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;
```

if (only one item)
  signal(empty);
else
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;
```
Continues its execution

```java
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 waiting signaled thread (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?

init(s1, 0);
init(s2, 0);

- What about more than two threads?
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

```c
Thread 1
    ...
    Barrier(b);
    ...
```

```c
Thread n
    ...
    Barrier(b);
    ...
```

 Barrier variable
Equivalence

• Semaphores
  ● Good for signaling and fine for simple mutex
  ● Not good for mutex in general, since easy to introduce a bug with ordering against other semaphores
    • Locks are only for mutex, so clearer and less bug-prone

• Monitors
  ● Good for scheduling and mutex
  ● May be costly for simple signaling
**The Big Picture**

<table>
<thead>
<tr>
<th>High-Level Atomic API</th>
<th>Mutex</th>
<th>Semaphores</th>
<th>Monitors</th>
<th>Barriers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-Level Atomic Ops</td>
<td>Load/store</td>
<td>Interrupt disable/enable</td>
<td>Test&amp;Set</td>
<td>Other atomic instructions</td>
</tr>
<tr>
<td></td>
<td>Interrupts (I/O, timer)</td>
<td>Multiprocessors</td>
<td>CPU scheduling</td>
<td></td>
</tr>
</tbody>
</table>
Summary

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