COS 318: Operating Systems

Semaphores, Monitors and Condition Variables

Kai Li
Computer Science Department
Princeton University

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

- Semaphores
- Monitors
- Mesa-style monitor and its idioms
- Barriers
Semaphores (Dijkstra, 1965)

- **Initialization**
  - Initialize a semaphore s

- **P (or Down or Wait or “Probeer”) definition**
  - Atomic operation
  - Wait for semaphore to become positive and then decrement
    ```
    P(s) {
      while (s <= 0) {
        ;
        s--;
      }
    }
    ```

- **V (or Up or Signal or “Verhoog”) definition**
  - Atomic operation
  - Increment semaphore by 1
    ```
    V(s) {
      s++;  
    }
    ```
Bounded Buffer with Semaphores

- Initialization: emptyCount = N; fullCount = 0
- Are $P(mutex)$ and $V(mutex)$ necessary?

```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
    }
}
```
Example: Interrupt Handler

- A device thread works with an interrupt handler
- What to do with shared data?
- What if “m” is held by another thread or by itself?
Use Semaphore

Init(s, 0);

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

Interrupted Thread
    ...
    Interrupt handler
    ...
    V(s);
    ...
    ...
    ...
Semaphores Are Not Always Convenient

- A shared queue has Enqueue and Dequeue:

```c
Enqueue(q, item) {
    Acquire(mutex);
    put item into q;
    Release(mutex);
}

Dequeue(q) {
    Acquire(mutex);
    take an item from q;
    Release(mutex);
    return item;
}
```

- What if we want `Dequeue(q)` to block until `q` is not empty?
  - It is a consumer and producer problem

- Semaphores are difficult to use
  - Orders are important
Monitor: Hide Mutual Exclusion

- Brinch-Hansen (73), Hoare (74)
- Procedures are mutual 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”
Producer-Consumer with Monitors

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

procedure Consumer
begin
  while true do
    begin
      ProdCons.Remove();
      consume an item;
    end;
  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;
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
if (only one item)
    signal(empty);
else
    something else
end;
```

```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;
```
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?

```pascal
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

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


- This is easy to implement!

- Issues?
Mesa Style “Monitor” (Birrell’s Paper)

- Associate a condition variable with a mutex

- Wait( mutex, condition )
  - Atomically unlock the mutex and enqueued on the condition variable (block the thread)
  - Re-lock the lock when it is awaken

- Signal( condition )
  - No-op if there is no thread blocked on the condition variable
  - Wake up at least one if there are threads blocked

- Broadcast( condition )
  - Wake up all waiting threads
Consumer-Producer with Mesa-Style Monitor

```c
static count = 0;
static Cond full, empty;
static Mutex lock;

Enter(Item item) {
    Acquire(lock);
    if (count==N)
        Wait(lock, full);
    insert item into buffer
    count++;
    if (count==1)
        Signal(empty);
    Release(lock);
}

Remove(Item item) {
    Acquire(lock);
    if (!count)
        Wait(lock, empty);
    remove item from buffer
    count--;
    if (count==N-1)
        Signal(full);
    Release(lock);
}
```

Any issues with this?
Consumer-Producer with Mesa-Style Monitor

```c
static count = 0;
static Cond full, empty;
static Mutex lock;

Enter(Item item) {
    Acquire(lock);
    while (count==N)
        Wait(lock, full);
    insert item into buffer
    count++;
    if (count==1)
        Signal(empty);
    Release(lock);
}

Remove(Item item) {
    Acquire(lock);
    while (!count)
        Wait(lock, empty);
    remove item from buffer
    count--;
    if (count==N-1)
        Signal(full);
    Release(lock);
}
```
The Programming Idiom

- Waiting for a resource

```c
Acquire( mutex );
while ( no resource )
    wait( mutex, cond );
...
(use the resource)
...
Release( mutex);
```

- Make a resource available

```c
Acquire( mutex );
...
(make resource available)
...
Signal( cond );
/* or Broadcast( cond );
Release( mutex );
```
Enqueue(Queue q, Item item) {
    Acquire(lock);
    insert an item to q;
    Signal(Empty);
    Release(lock);
}

Item GetItem(Queue q) {
    Item item;
    Acquire(lock);
    while ( q is empty )
        Wait(lock, Empty);
    remove an item;
    Release(lock);
    return item;
}

Does this work?
Condition Variables Primitives

- **Wait( mutex, cond )**
  - Enter the critical section (min busy wait)
  - Release mutex
  - Put my TCB to cond’s queue
  - Call scheduler
  - Exit the critical section . . . (blocked)

- **Waking up:**
  - Acquire mutex
  - Resume

- **Signal( cond )**
  - Enter the critical section (min busy wait)
  - Wake up a TCB in cond’s queue
  - Exit the critical section
More on Mesa-Style Monitor

- Signaler continues execution
- Waiters simply put on ready queue
  - Must reevaluate the condition
- No constraints on when the waiting thread must run
- No constraints on signaler
  - Can execute after signal call (Hansen’s cannot)
  - Do not need to relinquish control to awaken thread/process
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
Example: A Simple Barrier

- Thread A and Thread B want to meet at a particular point
- Then both go forward

- How would you program this with a monitor?
Using Semaphores as A Barrier

- Use two semaphore?

  \[
  \text{init}(s1, 0); \\
  \text{init}(s2, 0);
  \]

- 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
  - Not good for mutex because it is easy to introduce a bug
- Monitors
  - Good for scheduling and mutex
  - Maybe costly for a simple signaling
Summary

- Semaphores
- Monitors
- Mesa-style monitor and its idiom
- Barriers