COS 318: Operating Systems

Semaphores, Monitors and Condition Variables
Today’s Topics

- Semaphores
- Monitors
- Mesa-style monitors
- Programming idiom
- Barriers
Bounded Buffer Problem
Bounded Buffer with Sleep and Wakeup

producer() {
    while (1) {
        produce an item
        if (count == N) sleep;
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

c consumer() {
    while (1) {
        if (count == 0) sleep();
        take an item from buffer
        count = count - 1;
        if (count == N-1) wakeup(producer);
        consume the item
    }
}
Bounded Buffer with Sleep and Wakeup

What if consumer is descheduled after reading count?
Lost wakeup problem
Problem: access and test of count not atomic
Semaphores (Dijkstra, 1965)

- Keep count of number of wakeups saved
- Initialization
  - Initialize a value atomically
- P (or Down or Wait) definition
  - Atomic operation
  - Wait for semaphore to become positive and then decrement
    \[
    P(s) \{
      \text{while} \ (s \leq 0) \\
      \text{if} \ (-s < 0) \\
      \text{block}(s); \\
      s--; \\
    \}
    \]
- V (or Up or Signal) definition
  - Atomic operation
  - Increment semaphore by 1
    \[
    V(s) \{
      s++; \\
    \}
    \]
Bounded Buffer with Semaphores

- **Initialization**: emptyCount = N; fullCount = 0
- **Are** \( P(\text{mutex}) \) and \( V(\text{mutex}) \) **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
    }
}
```
Use Semaphores for Interrupt Handling

Init(s, 0);

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

Interrupted Thread

Interrupt handler
    V(s);
    ...

...
Is Mutual Exclusion Enough?

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

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

- Event sequencing
  - Don’t consume if buffer empty, wait for something to be added
- Mutual exclusion
  - Avoid race conditions on shared variables
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);
    }
}
```

```c
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
  - Enforced by monitor (by compiler)

Queue of waiting processes trying to enter the monitor

- What about blocking and sequencing?
Condition Variables in A Monitor

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

- Look like semaphores, but are not
  - They don’t “count”, or accumulate signals
  - Like sleep/wakeup, but with mutual exclusion at monitor level
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;
What happens after a signal?

- Run the signaled thread immediately and suspend the current one (Hoare)
  - If the signaler has other work to do, life is complex
  - It is difficult to make sure there is nothing to do, because the signal implementation is not aware of how it is used
  - It is easy to prove things

- Exit the monitor (Hansen)
  - Signal must be the last statement of a monitor procedure

- Continues its execution (Mesa)
  - Easy to implement
  - But, the condition may not be true when the awaken process actually gets a chance to run
Mesa Style “Monitor” (Birrell’s Paper)

- Associate a condition variable with a mutex
  - \texttt{Wait}( \texttt{mutex}, \texttt{condition} )
    - Atomically unlock the mutex and enqueue on the condition variable (block the thread)
    - Re-lock the lock when it is awoken
  - \texttt{Signal}( \texttt{condition} )
    - No-op if there is no thread blocked on the condition variable
    - Wake up at least one if there are threads blocked
  - \texttt{Broadcast}( \texttt{condition} )
    - Wake up all waiting threads

- Original Mesa paper
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

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**

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

- **Make a resource available**

  ```
  Acquire( mutex );
  ...
  (make resource available)
  ...
  Signal( cond );
  /* or Broadcast( cond );
  Release( mutex );
  ```
Condition Variables Primitives

- **Wait( mutex, cond )**
  - Enter the critical section (min busy wait)
  - Release mutex
  - Put my TCB on 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, with no special priority
  - Must reevaluate the condition
- No constraints on when the waiting thread/process must run after a “signal”
- Simple to introduce a broadcast: wake up all
- No constrains 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 and then go on.
- How would you program this with a monitor?
Using Semaphores as A Barrier

- Use two semaphore?
  ```
  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
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