Semaphores

- Informally, multi-valued, queued, blocking lock
- Semaphore consists of a value and a queue
- Acquiring a Semaphore (aka, wait, down, or P):
  - If value is positive, decrement it and let the process in
  - If value is zero, put the process at the end of the queue
- A process releases a semaphore (aka, signal, up, or V)
  - If queue is not empty, head of the queue is allowed to acquire S
  - If the queue is empty, increment S's value
- Access to the semaphore information should be atomic (how?/why?)

Implementation

```c
void wait(Semaphore S) {
    Lock(S.Lock); // added Lock variable to Semaphore
    if (S.value == 0)
        Add to queue and block; // proceed when unblocked
    else
        S.value--;
    Unlock(S.Lock);
}

void signal (Semaphore S) {
    Lock(S.Lock);
    if (!empty(S.q))
        Unblock Top of queue;
    else
        S.value++;
    Unlock(S.Lock);
}
```

- Why are the lock and unlock needed?
- Waiting while holding the lock in wait!

Revisiting Producer Consumer

```c
int n = 0;

// Producer
while (1) {
    wait(mutex);
    produce;
    append();
    n++;
    if(n == 1)
        signal(mutex);
}

//consumer
while(1) {
    wait(mutex);
    take();
    n--;
    if(n == 0)
        wait(empty);
}
```

- Wrong Implementation
- Can you see the flaw in this program? Its tricky. Can you fix it?
Correct Implementation Using Binary Semaphores

```
//Binary semaphore Mutex, initialized to 1
//empty, initialized to 0
int n = 0, m;

// Producer
while (1) {
  produce() {
    wait(mutex);
    append();
    n++;
    if(n == 1)
      signal(empty);
    signal(mutex);
  }
}

// consumer
while(1) {
  produce an item in nextp;
  wait(empty);
  wait(mutex);
  buffer.append(nextp);
  signal(mutex);
  signal(full);
}

• Why does this additional assignment solve anything?
• A little clumsy
```

More Elegant Solution

```
//Initial values: mutex = 1, full = 0, empty = n
Producer:
while (1) {
  produce an item in nextp;
  wait(empty);
  wait(mutex);
  buffer.append(nextp);
  signal(mutex);
  signal(full);
}

Consumer:
while(1) {
  wait(full);
  wait(mutex);
  nextc = buffer.nextItem();
  signal(mutex);
  signal(empty);
}
```

Semaphore Discussion

• A more powerful and efficient mechanism for locking
• Activities interfere with each other only if they access the same semaphore
• No busy wait (or is there?)
• As with everything discussed so far, relies on well behaved processes to release the lock/semaphore on their way out
• A semaphore is a non-preemptable resource — subject to deadlock

Readers/Writers Problems

• So far, we have assumed that all accesses must be mutually exclusive
• With Readers/Writers problems
  – Any number of readers may access the resource
  – Only one writer at a time may access it
  – No readers should be reading when a writer is writing
• When is mutual exclusion necessary?
• Why not just use full mutual exclusion?
• Can you think of problems that are in this class?
Solution to Readers Writers Problem

//Semaphore mutex initialized to 1
//Semaphore write initialized to 1
readers
wait(mutex);
readers++;
if(readers == 1) readers--;
if(readers == 0) READ;

wait(mutex);
readers--;
if(readers == 0) signal(mutex);

• Problems?

Bonus HW Problem: Writers Have Priority

//All semaphores initialized to 1
reader
wait(one_reader);
wait(read); wait(mutex2);
readers++; readers--;
if(readers==0) if(readers == 0) READ;
wait(mutex1); wait(mutex2);
readers--;
if(readers==0) if(readers == 0) signal(mutex1);

• Good enough?
• What can we do to solve the problem?

One More: Dining Philosophers

A philosopher thinks for a while, eats, thinks again, etc..

Each philosopher needs two chopsticks to eat

Can you come up with a ritual that will allow all the philosophers to eat?

Dining Philosophers

Semaphore chopstick[4];

while(1) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    eat();
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
}

• Good enough?
• What can we do to solve the problem?
Condition Locks (cont’d)

- `wait(condLock)` puts the thread to sleep until it is signalled. Can only be called if the thread owns `condLock`. The thread invisibly releases `condLock`
- `signal(condLock)` wakes a single thread that is waiting on the Lock. The thread must reacquire the lock again before it can continue execution.
- Broadcast releases all the waiting threads
- Make sure that the critical region is consistent at the wait since you are letting another process in
- Mesa style vs. Hoare style

Potential Solutions

- Allow only 4 philosophers to sit on the table
- Pick up both chopsticks together (only try if they are available)
- If your other chopstick is not available, let go of the one you have in hand and try later
- If your other chopstick is not available, steal it from the philosopher who already has it
- Be creative, have at least one lefty philosopher and one righty philosopher
- More when we get to deadlock

Condition Locks (cont’d)

- `wait(condLock)` puts the thread to sleep until it is signalled. Can only be called if the thread owns `condLock`. The thread invisibly releases `condLock`
- `signal(condLock)` wakes a single thread that is waiting on the Lock. The thread must reacquire the lock again before it can continue execution.
- Broadcast releases all the waiting threads
- Make sure that the critical region is consistent at the wait since you are letting another process in
- Mesa style vs. Hoare style

Condition Locks

Condition C;
Lock condLock;
------------Producer-----------
while (1) {
produce an item in nextp;
Lock(condLock);
if(count == MAX) C.wait(condLock);
count++;
buffer.append(nextp);
if(count == 1) C.signal(condLock);
Unlock(condLock); }  
-------------Consumer-----------
while(1) {
Lock(condLock);
if(count == 0) C.wait(condLock);
nextc = buffer.nextItem();
if(count==MAX) C.signal(condLock);
count--;
Unlock(condLock);
Consume nextc;
}
Critical Regions

- Counting semaphores are a little more powerful than locks
- However, as demonstrated, they can be difficult to use
- Conditional variables makes things even more hairy
- Difficulty comes from
  - The code to acquire and release the critical region is scattered across the program, potentially in different executables
  - Inefficient/unfair to wait on conditions
- Conditional critical regions is an attempt to address these problem
  - Idea: Associate the locking with the data

Conditional Critical Regions

- High Level Synchronization construct
- A shared variable of type T is declared as:
  
  ```
  var v: shared T;
  ```
- Variable v only accessed inside a statement such as:
  
  ```
  region v when B do S;
  ```
- B is a boolean expression
  - If B is true, S is executed, otherwise, we are blocked until B becomes true
  - While S is being executed, no other process can access v

Bounded Buffer

- Definition:
  
  ```
  var buffer: shared record
    pool array [0...n-1] of item
    count, in, out: integer;
  end;
  ```
- Producer:
  
  ```
  produce nextp;
  region buffer when count < n
    pool[in] = nextp;
    in = in + 1 mod n;
    count = count + 1;
  end;
  ```
- Consumer
  
  ```
  consume nextc;
  region buffer when count > 0
    nextc = pool[out];
    out = out + 1 mod n;
    count = count - 1;
  end;
  ```
Implementation (Bonus HW)

```c
wait(mutex);
while (!B) {
    first_count++;
    if (second_count > 0)
        signal(second_delay);
    else
        signal(mutex);
    wait(first_delay);
    first_count--;
    second_count++;
    if(first_count > 0)
        signal(first_delay);
    else
        signal(second_delay);
    wait(second_delay);
    second_count--;
}
S;
if (first_count > 0)
    signal(first_delay);
else if (second_count > 0)
    signal(second_delay);
else signal(mutex);
...
```
• Hoare's original definition – activate a waiting process immediately.
  - What if signalling process is not done yet? Two additional process switches.
  - How to ensure that signalling process continues immediately after the current one (urgent queue).
  - Process scheduling must be reliable.

• A better definition – notify the process that the condition has changed.
  - Process checks condition again (since another process might have acquired the monitor and/or changed variables).
  - Broadcast causes all waiting processes on the condition to be reactivated (to recheck condition).

Examples from Real OSs – Solaris
• Supports adaptive mutex, condition variables, semaphores, reader-writer locks, and turnstiles.
  - Adaptive mutex idea: busy wait when you expect the lock to be released by a thread running on another processor.
  - Otherwise block.
• Supports turnstiles (queues ordered by priority) for semaphores and locks.
  - Turnstile queues are ordered by priority.

Examples from Real OSs – XP
• For global resources:
  - Disables interrupts on uniprocessors, uninterruptable spinlocks for multiprocessors.
  - Mutexes, semaphores, events (basically condition variables) and timer based synchronization
  - Dispatch object is similar to a semaphore.
• User level synchronization uses "dispatcher objects" to implement several synchronization mechanisms.

More Examples – XP
• For global resources:
  - Immediately after current and/or children processes exit.
  - How to ensure that signaling process continues after signal.
  - Additional process switches.
  - What if signaling process is not done yet? Two process immediately.

Some Monitor Discussion

A better definition – notify process that the condition has changed.

Broadcast causes all waiting processes on the condition to be reactivated (to recheck condition).

Examples from Real OSs – Solaris
• Supports adaptive mutex, condition variables, semaphores, reader-writer locks, and turnstiles.
  - Adaptive mutex idea: busy wait when you expect the lock to be released by a thread running on another processor.
  - Otherwise block.
• Supports turnstiles (queues ordered by priority) for semaphores and locks.
  - Turnstile queues are ordered by priority.