Process Synchronization (part 2)

ECE469, Jan 24

Yiying Zhang

11. Producer & Consumer Problem
23. Condition Variables
29. Two Options of the Signaler
32. Monitors
36. Semaphore problem 2 Readers-Writers problem

Reading

- Dinosaur Ch 6
- Comet Ch 30

Review: Process synchronization

- Cooperating processes need to
  - share data
  - synchronize access to shared data
- Accessing shared data needs to be in CS
- Other types of synchronization more complex
- Synchronization without OS help is hard
- Sync primitives supported by OS
  - Lock() is simple, but not powerful enough
  - More powerful ones were invented
    - Semaphore
    - Condition variables

[lec4] *Mutual exclusion & Critical Section*

- *Critical section* – a section of code, or collection of operations, in which only one process may be executing at a given time

- *Mutual exclusion* - mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded)

[lec4] Desirable properties of MuEx

- *Fair*: if several processes are waiting, let each in eventually

- *Efficient*: don’t use up substantial amounts of resources when waiting (e.g. no busy waiting)

- *Simple*: should be easy to use (e.g. just bracket the critical sections)

[lec4] Lock (aka mutex)

Init: lock = 1; // 0 means held; 1 means free

```
lock_acquire(lock)
{
    while (lock==0);
    lock--;
}
```

```
lock_release(lock)
{
    if (lock == 0)
        lock++;
}
```

- Each primitive is atomic
- In reality, lock is not implemented as above!
  - The waiting process is put to sleep

[lec4] Semaphore

- A synchronization variable that takes on non-negative integer values
  - Invented by Edsger Dijikstra in the mid 60’s

- Two primitve operations
  - wait(semaphore): an atomic operation that waits for semaphore to become greater than 0, then decrements it by 1
  - signal(semaphore): an atomic operation that increments semaphore by 1

Semaphores Definition (Dijkstra)

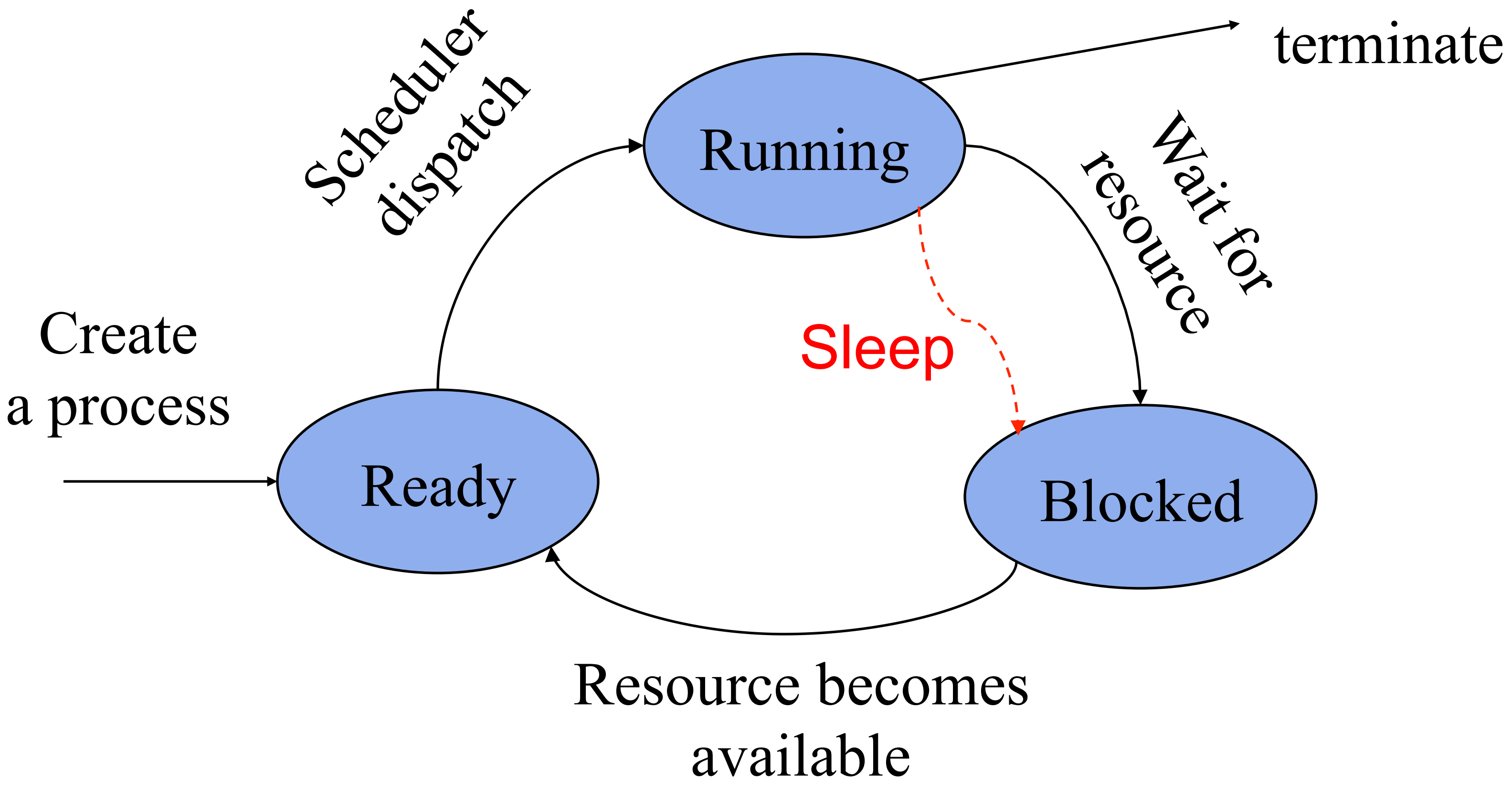
A semaphore is like an integer, with three differences:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.
   wait
2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.
3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked. done -> release -> ++

To say that a thread blocks itself (or simply "blocks") is to say that it notifies the scheduler that it cannot proceed. The scheduler will prevent the thread from running until an event occurs that causes the thread to become unblocked. In the tradition of mixed metaphors in computer science, unblocking is often called "waking".

- ACK: "The Little Book of Semaphores" by Allen B. Downey
  - Available online (free) at: http://www.greenteapress.com/semaphores/

[lec 2] Process State Transition

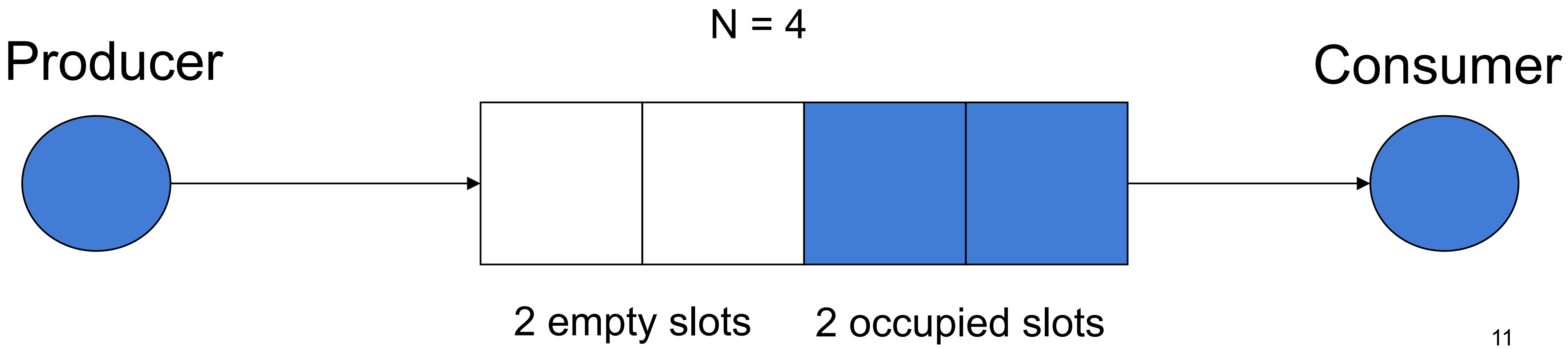


Semaphore benefits over locks

- Avoids busy waiting
- Has a value => more semantics
  - When greater than 1, can allow multiple threads to access critical resource
  - When equal to 1, can be used for mutual exclusion (only one thread in critical section)

[lec4] Producer & Consumer Problem

- Producer: creates copies of a resource
- Consumer: uses up (destroys) copies of a resource.
- Buffers: fixed size, used to hold resource produced by producer before consumed by consumer



[lec4] Producer & Consumer – semaphore, working

Producer

```
while (1) {

  produce an item;

  wait(EMPTY);

  acq(lock);
  insert(item to pool);
  rel(lock);

  signal(FULL)
}
```

Consumer

```
While (1) {

  wait(FULL);

  acq(lock);
  remove(item from pool);
  rel(lock)

  sginal(EMPTY);

  consume the item;
}
```

Init: FULL = 0; EMPTY = N; Mutex = 1;

Producer & Consumer – 2 pools

Producer

```
while (1) {
    get empty bucket  from buffer
     of empties
    produce data in bucket
    add full bucket to buffer of fulls
}
```

Consumer

```
While (1) {
    get full bucket from buffer of
     fulls
    consume data in buffer
    add empty bucket to buffer of
     empties
}
```

Producer & Consumer (2 pools)

Producer

```
while (1) {
    while (no empty buffer in Pool of Es)
        ;
    get empty buffer from pool of empties
    produce data in buffer
    add full buffer to pool of fulls
}
```

Consumer

```
While (1) {
    while (no full buffer in pool of Fs)
        ;
    get full buffer from pool of fulls
    consume data in buffer
    add empty buffer to pool of empties
}
```

Producer & Consumer (2 pools) – solution using semaphores

Producer

```
while (1) {
  wait(EMPTY);
  acq(lock);
  get empty buffer  from pool of
      empties
  rel(lock);
 produce data in buffer
 acq(lock);
 add full buffer to pool of fulls
  rel(lock);
  signal(FULL);
 }
```

Consumer

```
While (1) {
  wait(FULL);
  acq(lock);
get full buffer  from pool of
      fulls
  rel(lock);
consume data in buffer
  acq(lock);
add empty buffer to pool of
      empties
  rel(lock);
  signal(EMPTY);
}
```

FULL=0; EMPTY=N; MUTEX = 1;

Producer & Consumer (2 pools) – wrong solution using semaphores

Producer

```
while (1) {
  acq(lock);
  wait(EMPTY);
  get empty buffer  from pool of
       empties
  rel(lock);
 produce data in buffer
 acq(lock);
 add full buffer to pool of fulls
  rel(lock);
  signal(FULL);
 }
```

Consumer

```
While (1) {
 wait(FULL);
 acq(lock);
get full buffer  from pool of
  fulls
rel(lock);
consume data in buffer
acq(lock);
add empty buffer to pool of
  empties
rel(lock);
signal(EMPTY);
}
```

FULL=0; EMPTY=N; MUTEX = 1;

Deep thinking (take home)

- Why do we need to put mutex around accessing shared data
- Why does producer wait(empties) but signal(fulls)?
  - Explain in terms of creating/destroying resources
- Is the order of wait()’s important?
- Is the order of signal()’s important?
- Could we have separate mutex semaphores for each pool?
- How would this be extended to > 1 consumers?

Producer & Consumer (2 pools) – lock solution? (take home)

Producer

```
 while (1) {
 retry:
   acq(lock);
   if (no empty buffer in Pool of Es)
        {rel(lock); goto retry);}
   else
      get empty buffer  from pool of Es
   rel(lock);

   produce data in buffer

   acq(lock);
   add full buffer to pool of Fs
   rel(lock);
}
```

Consumer

```
While (1) {
retry:
   acq(lock);
   if (no full buffer in pool of Fs)
          {rel(lock); goto retry;}
   else
       get full buffer  from pool of Fs

   rel(lock);
          consume data in buffer

   acq(lock);
      add empty buffer to pool of Es
   rel(lock);
}
```

Producer & Consumer – semaphore, counting is tricky

Producer

```
while (1) {

  produce an item;

  wait(EMPTY);

  acq(lock);
  insert(item to pool);
  rel(lock);

  signal(FULL)
}
```

Consumer

```
While (1) {

  wait(FULL);

  acq(lock);
  remove(item from pool);
  rel(lock)

  sginal(EMPTY);

  consume the item;
}
```

Init: FULL = 0; EMPTY = N; Mutex = 1;

Producer & Consumer (1 pool) -- is there sth simpler than semaphore?

Producer

```
while (1) {

 produce an item;

    acquire(mutex);
    if (pool is Full) {
       release(mutex);
       wait(NotFULL);
       acquire(mutex)
    }
    record if pool was empty;
    insert(item)

    if (pool was empty)
        signal(NotEMPTY)
    release(mutex)
}
```

Put me
To sleep

If anyone is
sleeping,
wake it up
(no counting)

Consumer

```
While (1) {

acquire(mutex)
 if (pool is Empty {
     release(mutex)
     wait(NotEMPTY)
     acquire(mutex)
 }
 record if pool was full
 remove(item)

 if (pool was Full)
     signal(NotFULL)
release(mutex)

consume the item;
}
```

Producer & Consumer (1 pool) -- is there sth conceptually simpler than semaphore?

Producer

```
while (1) {

 produce an item;

    acquire(mutex);
    if (pool is Full) {

        wait(NotFULL);

    }
    record if pool was empty;
    insert(item)

    if (pool was empty)
        signal(NotEMPTY)
    release(mutex)
}
```

The simplification implies NotFull is tied to mutex

Consumer

```
While (1) {

acquire(mutex)
 if (pool is Empty {

     wait(NotEMPTY)

 }
 record if pool was full
 remove(item)

 if (pool was Full)
     signal(NotFULL)
 release(mutex)

 consume the item;
}
```

Mutual Exclusion provided by OS or language/compiler

- Semaphore
  - Powerful, but kind of low level
  - can we have a high level abstraction?
- Locks and condition variables
  - Lock alone is not flexible enough
  - Need some mechanism to sleep and wake up
- Monitor

Conditional Variable

- An explicit queue that processes/threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition)
- some other process/thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition).

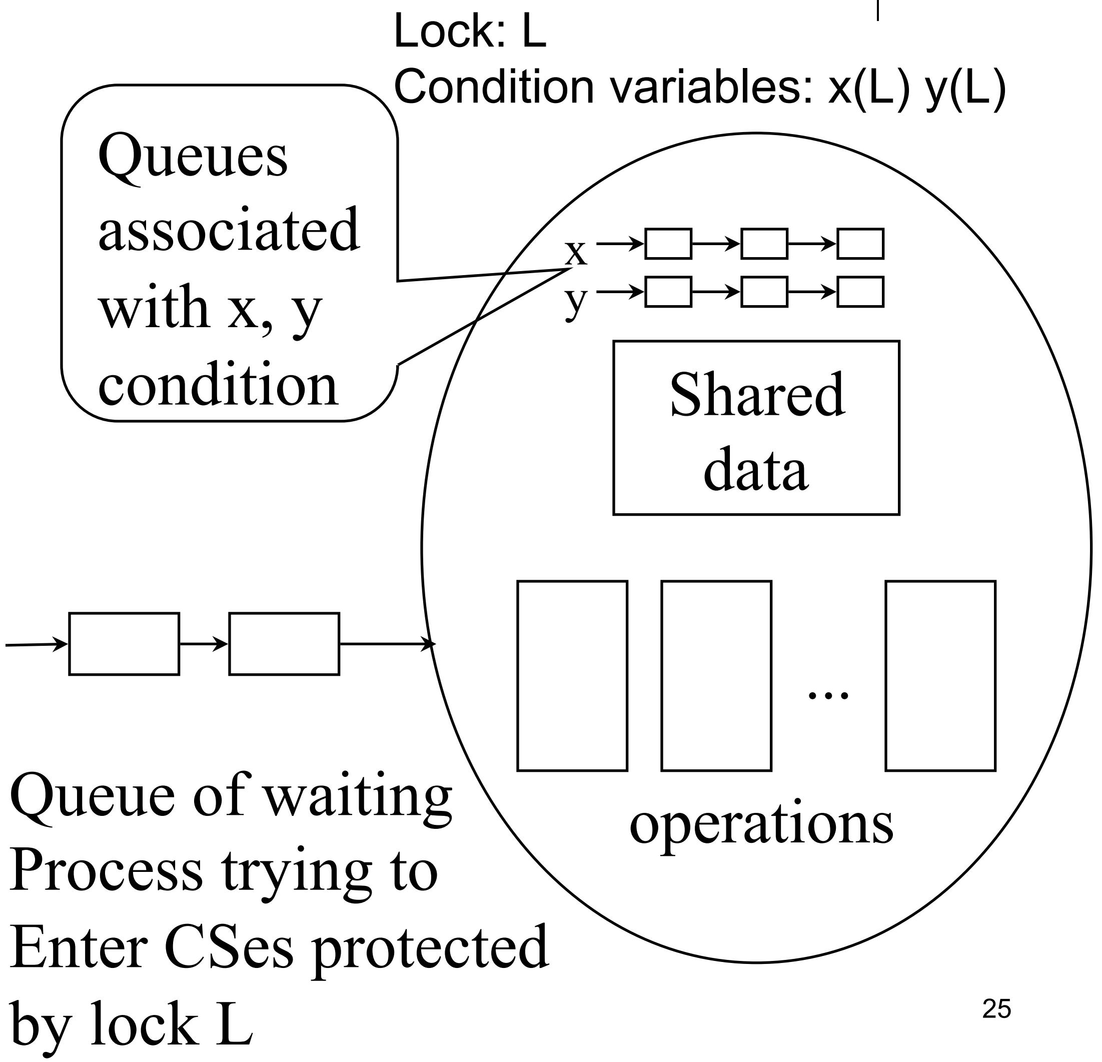
Condition Variables

- Used in conjunction with locks
- Used inside critical section to wait for certain conditions
- Contrast with Semaphore:
  - Has no counting bundled
  - More intuitive to many people

- Usage
  - On creation, specify which mutex it is associated with

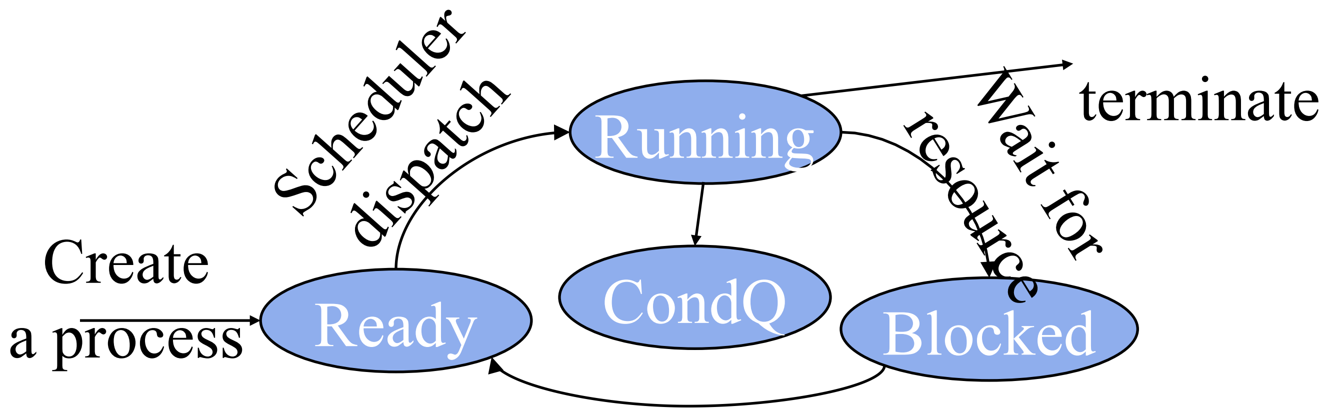
Condition Variables

- Wait (condition)
  - Block on "condition"
- Signal (condition)
  - Wakeup one or more processes blocked on "condition"
- Conditions are like semaphores but:
  - signal is no-op if none blocked
  - There is no counting!


Lock: L
Condition variables: x(L) y(L)
Queues associated with x, y condition
x
y
Shared data
...
operations
Queue of waiting Process trying to Enter CSes protected by lock L

“Wow, I like condition variables”

- One problem – what happens on wakeup?
  - Only one thing can be inside critical section
  - But wakeup implies both signaler and waiter may be in critical section, who should go on?



Two Options of the Signaler

- Relinquishes control to the awaken process; suspend signaler (Hoare-style, early time)
  - Signaler gives up lock, waiter runs immediately
  - Waiter gives back lock and CPU to signaler after critical sec.
  - Complex if the signaler has other work to do
  - In general, easy to prove things about system (e.g. fairness)

Producer & Consumer (1pool) – use condition variables

Producer

```
while (1) {

 produce an item;

    acquire(mutex);
    if (pool is Full) {
       release(mutex);
       wait(NotFULL);
       acquire(mutex)
    }
    record if pool was empty;
    insert(item)

    if (pool was empty)
        signal(NotEMPTY)
    release(mutex)
}
```

Consumer

```
While (1) {
 acquire(mutex)
 if (pool is Empty {
     release(mutex)
     wait(NotEMPTY)
     acquire(mutex)
 }
 record if pool was full
 remove(item)

 if (pool was Full)
     signal(NotFULL)
 some other work
 release(mutex)

consume the item;
}
```

```
rel(mutex)
signal(NotFull)
acq(mutex)
```

Two Options of the Signaler

- Relinquishes control to the awaken process; suspend signaler (Hoare-style, early time)
  - Signaler gives up lock, waiter runs immediately
  - Waiter gives back lock and CPU to signaler after critical sec.
  - Complex if the signaler has other work to do
  - In general, easy to prove things about system (e.g. fairness)

- Continues its execution (Mesa-style, modern)
  - Signaler keeps lock and CPU
  - Waiter put on ready queue
  - Easy to implement (e.g., no need to keep track of signaler)
  - But, what can happen when the awaken process gets a chance to run?
    - E.g. pool is full, producer 1 wait; consumer signals it; p1 in ready Q; consumer rel (lock); p2 comes along…

Producer & Consumer (1 pool) -- use condition variables – problem?

Producer

```
while (1) {

 produce an item;

     acquire(mutex);
     if  (pool is Full) {
          release(mutex);
          wait(NotFULL);
          acquire(mutex)
     }
     record if pool was empty;
     insert(item)

     if (pool was empty)
          signal(NotEMPTY)
     release(mutex)
}
```

Consumer

```
While (1) {
 acquire(mutex)
 if  (pool is Empty {
      release(mutex)
      wait(NotEMPTY)
      acquire(mutex)
  }
 record if pool was full
 remove(item)

 if (pool was Full)
     signal(NotFULL)
 some other work
 release(mutex)

consume the item;
}
```

Producer & Consumer (1 pool) – use condition variables – how to fix?

Producer

```
while (1) {

 produce an item;

    acquire(mutex);
    while (pool is Full) {

        wait(NotFULL);

    }
    record if pool was empty;
    insert(item)

    if (pool was empty)
        signal(NotEMPTY)
    release(mutex)
}
```

给consumer判断用

Consumer

Is this busy waiting?

```
While (1) {

acquire(mutex)
 while (pool is Empty {

     wait(NotEMPTY)
 }
 record if pool was full
 remove(item)

 if (pool was Full)
     signal(NotFULL)
 release(mutex)

consume the item;
}
```

NotEMPTY NotEMPTY两个conditional variable

Monitors

- Monitors are high-level data abstraction tool combining three features:
  - Like an object in OO programming language
    - Shared data
    - All procedure operate on the shared data
  - Except the procedures are all mutually exclusive!
  - Sometimes aka thread-safe object/class/module
  - Java has monitors

- Convenient for synchronization involving lots of shared state (manipulating shared data)

- Monitors hide locks, but still need condition variables

Producer-Consumer with Monitors

```
monitor ProdCons
   record pool[100];
   condition nfull, nempty;

   procedure Enter(item);
   begin
      if (pool is full)
         wait(nfull);
      put item into pool;
      if (pool was empty)
         wakeup_someone();
   end;

   procedure Remove;
   begin
      if (pool is empty)
         wait(nempty);
      remove an item;
      if (pool was full)
         wakeup_someone());
   end;
```

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

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

Mutual Exclusion provided by OS or language/compiler

- Lock
  - Alone is not powerful enough
- Semaphore (incl. binary semaphore)

  binary semaphore alone not enough
- Lock and condition variable
- Monitor (hide lock, still use condition variables)

Semaphore classic examples

1. Producer-consumer problem

2. Readers-writers problem

3. Dining philosophers problem

Semaphore classic problem 2: Readers-Writers problem

- A data object is shared among multiple processes
- Allow concurrent reads (but no writes)
- Only allow exclusive writes (no other writes or reads)

Readers-Writers problem (Solution 1)

- Constraints:
  - Writers can only proceed if there are no readers/writers
  - Readers can proceed only if there are no writers
  - ➔ use a single semaphore BlockWrite
  - To keep track of how many are reading
  - ➔ use a shared variable
  - Only one process manipulates state variable at once
  - ➔ use semaphore Mutex

- Initialization:
  - semaphore BlockWrite = 1; // used to allow ONE writer or MANY readers
  - semaphore Mutex = 1; // binary semaphore (basic lock)
  - int Readers = 0; // count of readers reading in critical section

Writer

```
P(BlockWrite); // wait to lock the shared resource for a writer

< Do the Writing >

V(BlockWrite);
```

Reader

```
P(Mutex);
Readers++;
if (Readers == 1) // first reader acquire write lock
            P(BlockWrite);
V(Mutex);

< Do the Reading >

P(Mutex);
Readers--;
if (Readers == 0) // last (only) reader releases write lock
V(BlockWrite);
V(Mutex);
```

What will happen in different scenarios?

1. The first reader blocks if there is a writer; any other readers who try to enter block on mutex.

2. The last reader to exit signals a waiting writer.

3. When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.

4. If a writer exits and a reader goes next, then all readers that are waiting will fall through.

5. Does this solution guarantee all threads will make progress?

Writes can starve
=> Read preference

[lec4] What is a good solution

- Only one process inside a critical section

- Processes outside of critical section should not block other processes
- No one waits forever

- No assumption about CPU speeds
- Works for multiprocessors

Readers-Writers problem (Solution 2)

- How do we let reads yield to writes?
  - int Readers = 0, Writers = 0; // count of readers and writers in critical section
  - semaphore BlockWrite = 1; // used to allow ONE writer or MANY readers
  - semaphore BlockRead = 1; // used to block readers
  - semaphore RMutex = 1; // binary semaphore for Readers
  - Semaphore WMutex = 1; // binary semaphore for Writers

Write

```
P(WMutex);
Writers++;
if (Writers == 1) // block readers
        P(BlockRead);
V(Wmutex);

P(BlockWrite); // ensures only one writer
< Do the Writing >
V(BlockWrite);

P(WMutex);
Writers--;
if (Writers == 0) // enable readers
        V(BlockRead);
V(WMutex);
```

Reader

```
P(BlockRead); // at most one reader can go before a pending write
P(RMutex);
Readers++;
if (Readers == 1) // first reader acquire write lock
        P(BlockWrite);
V(RMutex);
V(BlockRead);
```

Any problem?

```
< Do the Reading >

P(RMutex);
Readers--;
if (Readers == 0) // last (only) reader releases write lock
        V(BlockWrite);
V(RMutex);
```

Problem of solution 2

- Reader starvation
- Is there a solution that’s fair to both reads and writes?
    - Take-home puzzle problem (not graded)