Process Synchronization (part 2)

ECE595

Sep 4

Y. Charlie Hu

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] 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] "Too much milk" problem with locks

```
Acquire(lock);
  if (noMilk)
    buy milk;
Release(lock);
```

} Critical Section

- There is one problem with this solution?

Often times, we have to wait for shared resources

- In "too much milk", we just needed to check
- Often times, before accessing shared resources, we have to wait …

Producer & Consumer Problem (1-pool version)

- Producer: creates copies of a resource
- Consumer: uses up (destroys) copies of a resource. (may produce something else)
- Buffer: used to hold resource produced by producer before consumed by consumer
- Synchronization: keeping producer & consumer in sync
- Happens inside OS all the time (e.g. I/Os)
  - How about in real life?

Producer & Consumer – solution using locks?

Producer

```
while (1) {

 produce an item;

  while (buffer is full);

  insert item into buffer

  }
```

Consumer

```
While (1) {

while (buffer is empty);

 remove an item;

 consume the item

 }
```

Often times, we have to wait for shared resources

- (Busy waiting is a bad idea)

- Checking resources needs to be in critical section
- Busy waiting & checking inside CS even worse!
  - No one else can check!

→ Need a more powerful sync. primitive!
→ Want the simplest primitive that can check & wait

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

Semaphore

```
wait(S) {
   while (S<=0);
   S--;
}
```

```
signal(S) {
   S++;
}
```

- In reality, wait(S) is not implemented as above!
- Semaphore aren't provided by hardware (why not?) – we'll discuss OS implementations next time

Binary Semaphore

Init: S =1;

```
wait(S) {
   while (S==0);
   S--;
}
```

```
signal(S) {
   if (S == 0) S++;
}
```

- Binary semaphores: only take 0 or 1
- Sounds familiar?
  - S=0 → someone is holding the lock!

semaphores vs. locks: fundamental difference?

Semaphores

```
wait(S) {
   while (S<=0);
   S--;
}
```

```
signal(S) {
   S++;
}
```

Binary Semaphore (lock)

```
wait(S) {
   while (S==0);
   S--;
}
```

```
signal(S) {
   if (S == 0) S++;
}
```


semaphore has built-in counting!

- signal(S) simply increments S
  - "just produced an item"
  - S value == how many items have been produced
- wait(S) will return without waiting only if S > 0;
  - Wait(S) is saying "waited until there is at least one item, and then just consumed an item"

Two usages of semaphores

- For mutual exclusion:
  - to ensure that only one process is accessing shared info at a time.
  - Semaphores or binary semaphores?
- For condition synchronization:
  - to permit processes to wait for certain things to happen
  - Semaphores or binary semaphores?

Producer & Consumer (1-pool version)

- Define constraints (what is "correct")
  - Consumer must wait for producer to fill buffer
    (mutual excl. or condition sync?)
  - Producer must wait for consumer to empty buffer, if all buffer space is in use
    (mutual excl. or condition sync?)
  - Only one process must manipulate buffer at once
    (mutual excl. or condition sync?)
- Use a separate semaphore for each constraint
  - Full = 0
  - Empty = N
  - Mutex = 1

Producer & Consumer – solution using locks?

Producer

```
while (1) {

 produce an item;

  while (buffer is full);

  insert item into buffer

 }
```

Consumer

```
While (1) {

 while (buffer is empty);

 remove an item;

 consume the item

 }
```

EMPTY=N; FULL=0, mutex=1

Producer & Consumer – solution using locks?

Producer

```
while (1) {

 produce an item;

 wait(EMPTY);

 acq(mutex)
  insert item into buffer
  rel(mutex)

  signal(FULL);
 }
```

Consumer

```
While (1) {

 wait(FULL);

 acq(mutex);
 remove an item;
 rel(mutex);

 signal(EMPTY);

 consume the item

 }
```


Deep thinking

- Why does producer wait(empties) but signal(fulls)?
  - Explain in terms of creating / destroying resources
- Is the order of signal()'s important?
- Is the order of wait()'s important?
- How would this be extended to have >1 consumers?

Producer & Consumer – solution using locks?

Producer

```
while (1) {

 produce an item;

 acq(mutex)
 wait(EMPTY);
  insert item into buffer
  rel(mutex)

  signal(FULL);
 }
```

Consumer

```
While (1) {

 wait(FULL);

 acq(mutex);
 remove an item;
 rel(mutex);

 signal(EMPTY);

 consume the item

 }
```

Break

Producer & Consumer (1 pool)

Producer

```
while (1) {

  produce an item;

  while (pool is full);

  insert(item to pool)

}
```

Consumer

```
While (1) {

  while (pool is empty);

  remove(item from pool);

  consume the item;

}
```


Producer & Consumer (1 pool) -- needs mutual excl, try lock

Producer

```
while (1) {

  produce an item;

  flag = 0;
  while (flag == 0) {
   acq(lock)
   if (pool is not full) {
         flag = 1;
         insert(item to pool)
   }
   rel(lock)
}
```

Consumer

```
While (1) {

  while (pool is empty);

  remove(item from pool);

  consume the item;



}
```


Producer & Consumer (1 pool) -- try 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;

}
```

EMPTY=N; FULL=0;

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;
```


Producer & Consumer (1 pool) -- is there sth 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)
}
```

Consumer

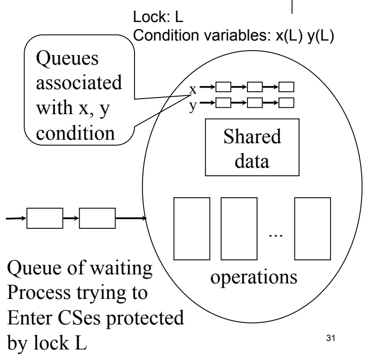
```
While (1) {

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;
}
```


Condition Variables

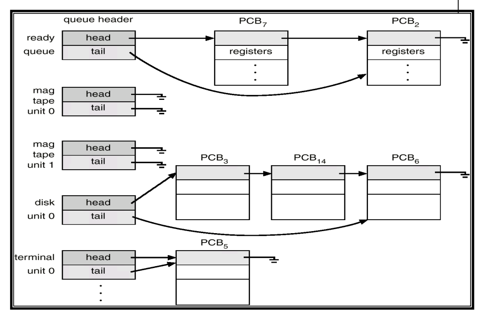
- Used in conjunction with locks
- Used inside critical section to wait for certain conditions
- Contrast with Semaphore:
  - Has no counting bundled
  - More intuitive to many people
- Usage
  - On creation, has to specify which mutex it is associated with

Condition Variables

- Wait (condition)
  - Block on “condition”
- Signal (condition)
  - Wakeup a process blocked on “condition”
- Conditions are like semaphores but “not sticky” :
  - signal is no-op if none blocked
  - There is no counting!



[lec4] Ready Queue And Various I/O Device Queues



Producer & Consumer (1 pool) – use condition variables

Producer

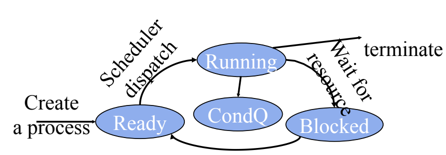
```
while (1) {

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)
}
```

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;
}
```


“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;
}
```

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!
  - 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)

Reading assignment

- Read Chapter 6