Process Synchronization (part 3)

ECE595, Jan 25

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

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;

  local_flag = 0;
  while (local_flag == 0) {
   acq(lock)
   if (pool is not full) {
         local_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

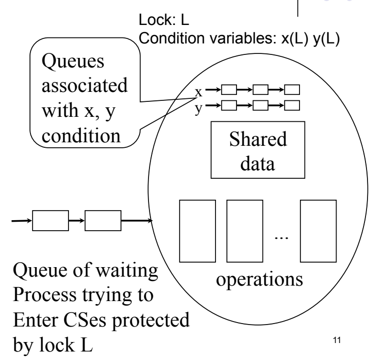
```
While (1) {

  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

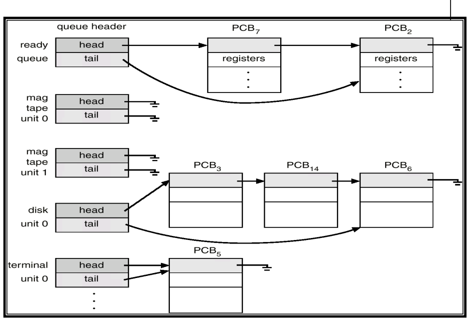
- Used in conjunction with locks
- Used inside critical section to wait for certain conditions
- Contrast with Semaphore:
  - Has no counting bundled
  - More intuitive to many people
- Usage
  - On creation, has to specify which mutex it is associated with

Condition Variables

- Wait (condition)
  - Block on “condition”
- Signal (condition)
  - Wake up 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

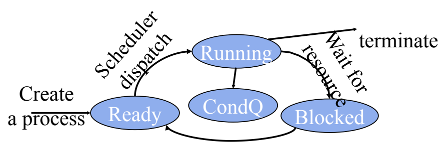
```
while (1) {

 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