Blocking Solutions

PENNSTATE

Where are we?

- Disable Interrupts
  - **Effectively stops scheduling other activities.**
- Busy-wait/spinlock Solutions
  - **Pure software solutions**
  - **Integrated hardware-software solutions**
- Blocking Solutions

mutex
progress
bounded wait requirement

Spinning vs. Blocking

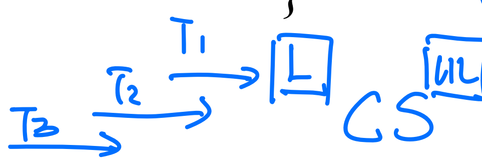
- In the previous solns., we busy-waited for some condition to change.
- This change should be affected by some other activity.
- We are "presuming" that this other activity will eventually get the CPU (some kind of pre-emptive scheduler).
- This can be inefficient because:
  - You are wasting the rest of your time quantum in busy-waiting
  - Sometimes, your programs may not work! (if the OS scheduler is not pre-emptive).

- In blocking solutions, you relinquish the CPU at the time you cannot proceed, i.e., you are put in the blocked queue.
- It is the job of the activity changing the condition to wake you up (i.e., move you from blocked back to ready queue).
- This way you do not unnecessarily occupy CPU cycles.

Example Blocking Implementation

```
Mutex_Lock(L) {
  Disable Interrupts/Use Spinlock
  Check if anyone is using L
  If not {
      Set L to being used
  }
  else {
      Move this TCB to Blocked
          queue for L
      Select another activity to run
          from Ready queue
      Context switch to that activity
  }
  Enable Interrupts/Use Spinlock
}
```

```
Mutex_Unlock(L) {
  Disable Interrupts/Use spinlock
  if (blocked queue of L == NULL)
      Set L to free
  }
  else {
      Move TCB from head of
          Blocked queue of L to
          Ready queue
  }
  Enable Interrupts/Use spinlock
}
```



① T1 gets lock
② when T2 comes in, it will have to wait for Lock to be free
③ Same for T3 as T2.

NOTE: These are OS system calls (where Disable/Enable are available), or Library calls (where Spinlocks are only option for implementation)

Until now …

- Exclusion synchronization/constraint
  - **Typical construct mutual exclusion lock**
    - **Mutex_lock(m)** different c.s. have different Lock(m;)
    - **Mutex_unlock(m)**

  - **Do a "man" on pthread_mutex_lock() for further syntactic/semantic information.**

- **But you also need synchronization constructs for other tasks than exclusion (i.e., ordering)**
  - **E.g., If printer queue is full, I need to wait until there is at least 1 empty slot**
  - **Note that mutex_lock()/mutex_unlock() are not very suitable to implement such synchronization**
  - **We need constructs to enforce orderings (e.g., A should be done after B).**

PENNSTATE

chapter 30.

enable a particular thread put itself into a queue to wait for a particular condition to be made true

Condition Variables

- C_wait() and c_signal() operations.
  → inform one number of a block queue for that particular condition variable that is now unblocked and move to ready queue
- A thread blocked on c_wait() returns when another performs a c_signal().

```
C_wait(C) {
  Disable Interrupts /Use spinlock
  Move this TCB to Blocked queue for C
  Select another thread to run
          from Ready queue
  Enable Interrupts /Use spinlock
  Context switch to that thread
}
```

```
C_signal(C) {
  Disable Interrupts/Use spinlock
  Check if blocked queue
    for L is empty
  else {
     Move TCB from head of
          Blocked queue of C to
          Ready queue
  }
  Enable Interrupts/Use spinlock
}
```

```
Cond_t  not_full, not_empty;;
Int count == 0;

Append() {
  if count == N   c_wait(not_full);

  … ADD TO BUFFER, UPDATE COUNT …

  c_signal(not_empty);
}

Remove() {
  if count == 0    c_wait(not_empty);

  … REMOVE FROM BUFFER, UPDATE COUNT

  c_signal(not_full);
}
```

wait until buffer is not full

have to run atomic to aviod rescheduling

wait until buffer is not empty

However, there is something wrong with this code!
There is a gap between checking (count == N) and c_wait()!
Similarly, for Remove.

Solution: Put c_wait() within a mutex_lock()

```
Cond_t  not_full, not_empty;
Mutex_lock m;
Int count == 0;

Append() {
   mutex_lock(m);
   if count == N   c_wait(not_full,m);

   … ADD TO BUFFER, UPDATE COUNT …

   c_signal(not_empty);
   mutex_unlock(m);
}

Remove() {
   mutex_lock(m);
   if count == 0   c_wait(not_empty,m);

   … REMOVE FROM BUFFER, UPDATE COUNT

   c_signal(not_full);
   mutex_unlock(m);
}
```

C_wait(c,m) : You give up "m" before waiting, and you regain "m" when you are signaled.

Issue

- **But, this “solution” does not really work if there are two threads that run “remove” (or “append”)**
  - **Can you identify why not?**

Solution: Put c_wait() within a mutex_lock()

```
Cond_t  not_full, not_empty;
Mutex_lock m;
Int count == 0;

Append() {
  mutex_lock(m);
  if count == N   c_wait(not_full,m);

  … ADD TO BUFFER, UPDATE COUNT …

  c_signal(not_empty);
  mutex_unlock(m);
}

Remove() {
  mutex_lock(m);
  if count == 0   c_wait(not_empty,m);

  … REMOVE FROM BUFFER, UPDATE COUNT

  c_signal(not_full);
  mutex_unlock(m);
}
```

Suppose that one thread performs a "c_wait". But, another thread runs "remove" after a "c_signal" and **steals** the buffer element.

Textbook

- The textbook goes through these scenarios in detail in Chapter 30
  - Need to understand these for P2

Solution: Put c_wait() within a mutex_lock()

```
Cond_t  not_full, not_empty;
Mutex_lock m;
Int count == 0;

Append() {
  mutex_lock(m);
  while count == N   c_wait(not_full,m);

  … ADD TO BUFFER, UPDATE COUNT …

  c_signal(not_empty);
  mutex_unlock(m);
}

Remove() {
  mutex_lock(m);
  while count == 0   c_wait(not_empty,m);

  … REMOVE FROM BUFFER, UPDATE COUNT

  c_signal(not_full);
  mutex_unlock(m);
}
```

Need to check that the condition still holds when a thread is finally scheduled.

Broadcast

- What if you do not know which thread will satisfy the condition when exiting a critical section?
  - I.e., you may "signal" a thread that does not meet the condition?
- What happens?
  - It goes back to waiting.
  - And a thread that meets the condition may not be awoken
- pthread_cond_broadcast - shall unblock all threads currently blocked on the specified condition variable *cond*