CSci 4061
Introduction to Operating Systems

Synchronization Basics: Locks

Motivation

- Issues
  - Threads communicate through shared variables
  - K ready or “runnable” threads => can’t predict which one is running at any particular time

Synchronization Outline

- **Basics**
- **Locks**
- Condition Variables
- Semaphores – if time
- Issues

Basics

- Race condition: threads + shared data
- Outcome (data values) depends on who gets there first/last

```
i=0

if i==0     if i==0
  => i=5      => i=4
else i=7    else i=8
```

- Possible values for i at the end of execution? 7,8,4,5!
- Shared variables = heap, globals, within the process
- Races => inconsistency or errors

```
if (free_buffer)    if (free_buffer)
   insert_item         insert_item
```

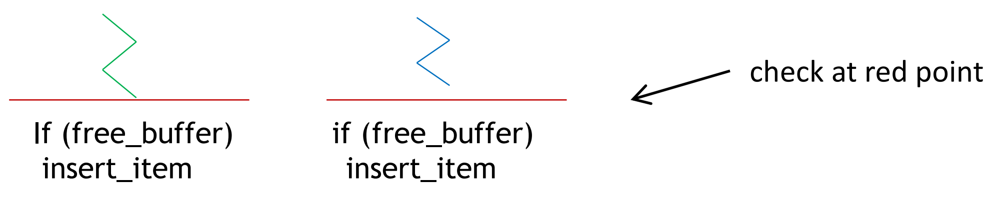
- If buffer is nearly full=> may overwrite or overflow

Problem

- Problem: we have limited control on when threads will run
- Need: orderly execution or cooperation
- Solution: synchronization
- Real life: washing dishes
  - Wash then dry
  - **No two people washing at the same time**

Synchronization

- Constrain the set of interleavings
  - Can’t prevent scheduler from switching them out
  - But threads can stay out of each others way



- Critical section
  - Region of code where shared access may lead to races
  - Constrain access to critical section
  - Only 1 thread at a time in the critical section

Critical section: How to do it?

- Threads **voluntarily** spin or block (wait) if another is in the critical section

```
entry        =>     possibly block or spin
<CS>                        <CS>
exit
```

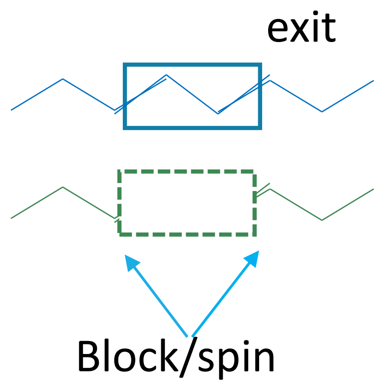
- Examples of critical section

```
If (free_buffer)
   insert_item
```

```
if i == 0             if  i == 0
 =>  i = 5             => i = 4
else i = 7            else i = 8
```

How to identify a CS: good question!

- Black art
- Conservative (too big) ?
- Too small =?



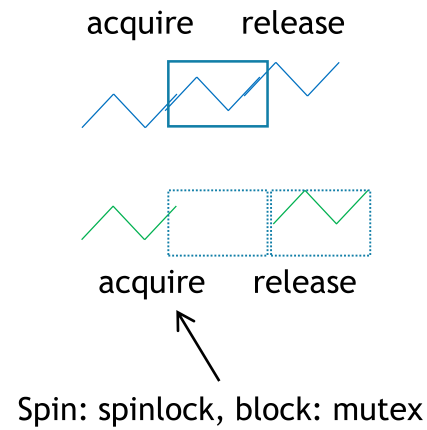
- Mutual exclusion: simplest type of synch
  - Only 1 thread allowed in CS
  - CS is “atomic” (all or nothing)—can be interrupted, but no one else can get in
- Exit
  - Crucial to make it work!

Related Issues

- Synchronization
  - Prevent bad things from happening
  - “wash then dry”, “no two washers…” (washing is a CS)
- Deadlock
  - Extreme case (misuse) of synchronization, everyone is blocked: join (self)
- Livelock
  - Everyone can run (not blocked) but no one can make progress
  - “one step forward, one step back”

Synchronization construct for mutual exclusion (ME)

- Locks:
  - Object in shared memory
  - Operations: `acquire` (lock), `release` (unlock)
  - Try to acquire a "held" lock => prevented
  - `acquire` lock before entering CS
  - `release` lock before leaving CS



Lock L;
acquire (L);
<CS>
release (L);

Lock is EXPLICIT—have to use it correctly!

T1	T2
acquire (L);	access X // this is allowed!
access to var X	
release (L);	

Use it carefully

T1	T2
acquire (L1);	acquire (L2);
access to var X	access to var X
release (L1);	release (L2);

Inside a Lock

- Lock

  boolean held;
  queue waiting_threads;

  Mutex:

  If held: acquire blocks thread and puts in on queue

  If queue is non-empty: release removes a thread from queue and makes it unblocked

Synchronization in Posix

- Posix mutex

```
#include <pthread.h>
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;  // unlocked

//acquire
int pthread_mutex_lock (pthread_mutex_t *mutex);

//release
int pthread_mutex_unlock (pthread_mutex_t *mutex);

//return 0 on success, non-0 error code otherwise



gcc -o myProg myProg.c -lpthread
```

Two Locks Deadlock; 2 Threads

- Deadlock – every thread is blocked

Lock L1, L2;

T1:	T2:
Acquire L1;	Acquire L2;
Acquire L2;	Acquire L1;
Release L2;	Release L1;
Release L1;	Release L2;

Mutex Example

```
account act;  // global shared state

// some number of deposit threads will be created
pthread_create (&t1, NULL, depositer, …);
pthread_create (&t2, NULL, depositer, …);

void *depositer (void *arg){
    amount_t amt, val;
    //determine amt somehow
    …
    val = deposit (&act, amt);
    …
}
```

Mutex example (cont'd)

```
pthread_mutex_t acc_mtx =
      PTHREAD_MUTEX_INITIALIZER;

amount_t deposit (account *act,
                  amount_t amount)
 {
      amount_t result;
      pthread mutex_lock (&acc_mtx);
      act->balance +=amount;
      result=act->balance;
      pthread_mutex_unlock (&acc_mtx);
      return result;
}
```

two threads calling deposit

Thread safety

Suppose you are not sure a library call is thread-safe?

`rand ()` –

what can you do?

Randsafe Example

```
#include <pthread.h>
#include <stdlib.h>

int randsafe (double *ramp) {
    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    int error;

    pthread_mutex_lock (&lock);
    *ranp = (rand() + 0.5)/(RAND_MAX + 1.0);
    pthread_mutex_unlock (&lock);
    return;
```

Are locks themselves safe?

- Yes!
- Must be possible for threads to concurrently call lock and unlock!
- All lock code is thread-safe

Posix mutex (cont'd)

- Can test if lock is held

```
#include <pthread.h>
 int pthread_mutex_trylock
                (pthread_mutex_t *mtx)
```

  - returns EBUSY if mtx is held

- Be careful: why?

```
if (pthread_mutex_trylock (&mtx)!= EBUSY)
   pthread_mutex_lock (&mtx);
```

Posix mutex: Bounded Buffer

Need ME, why?

```
item_t remove_item (buffer *b){
 item t st;
 if (b->next_slot_to_retrieve ==
     b->next_slot_to_store) return ERROR;
 st = b->items [b->next_slot_to_retrieve];
 b->next slot to retrieve++;
 // adjust next_slot_store if needed
 return st;
}
```

Posix mutex: Bounded Buffer

Need ME:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
item_t remove_item (buffer *b){
 item t st;
 pthread_mutex_lock (&mtx);
 if (b->next_slot_to_retrieve ==
     b->next_slot_to_store) return ERROR;
 st = b->items [b->next_slot_to_retrieve];
 b->next slot to retrieve++;
 // adjust next_slot_store if needed

 pthread_mutex_lock (&mtx);
 return st;
}
```

Synchronization

- Mutual exclusion (ME) solved with locks
    - just have to use them correctly
- Want other kinds of synchronization

Posix mutex (cont'd)

- Locks are limited to protecting shared variables only … and they are unconditional
- Want richer synchronization

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
item_t remove_item (buffer *b){
item t st;
pthread_mutex_lock (&mtx);
if (b->next_slot_to_retrieve ==
    b->next_slot_to_store) return ERROR;

st = b->items [b->next_slot_to_retrieve];
b->next slot to retrieve++;
// adjust next_slot_store if needed
pthread_mutex_lock (&mtx);
return st;
}
```

Posix mutex (cont'd)

- Locks are limited to protecting shared variables only … and they are unconditional
- Want richer synchronization

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
item_t remove_item (buffer *b){
   item t st;
   pthread_mutex_lock (&mtx);
   if (b->next_slot_to_retrieve ==
     b->next_slot_to_store) return ERROR; // block

   st = b->items [b->next_slot_to_retrieve];
   b->next slot to retrieve++;
   // adjust next_slot_store if needed
   pthread_mutex_unlock (&mtx);
   return st;
}
```

Need Richer Synchronization: ~ conditional synchronization

- Want producer (and consumer) to conditionally block if buffer full/empty

// should block if empty

```
item = remove_item (&b);
```

// should block if full

```
insert_item (&b, item);
```