CSci 4061
Introduction to Operating Systems

Synch Lecture 1
Synchronization Basics
Basics - board

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

\[
i = \begin{cases} 
0, & \text{if } i = 0 \\
5, & \text{if } i = 0 \\
4, & \text{else if } i = 7 \\
8, & \text{else if } i = 8 \\
\end{cases}
\]

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

- 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

```
If free_buffer
Insert_item
```

```
if (free_buffer
insert_item
```

check at red point

• 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 <CS> possibly block or spin <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

Block/spin
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
  • Stuck/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

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

Lock is EXPLICIT—have to use it correctly!

```text
T1
acquire (L)
access to var X
Release (L);
```

```text
T2
access X // this is allowed!
```

Spin; spinlock, block;mutex
Synchronization in Posix

• Posix mutex

    #include<pthread.h>
    //acquire
    int pthread_mutex_lock (pthread_mutex_t*mutex);
    //release
    int pthread_mutex_unlock (pthread_mutex_t*mutex);
    int pthread_mutex_destroy (pthread_mutex_t*mutex);
    const pthread_mutexattr_t*attr);

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

    pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER; // unhead

    ccc -o myProg myProg c-D REENTRANT -lpthread

Misspell -> no warning
Mutex example

```c
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.
}
```

//for dynamically created lock, need to use init
//be careful to init once—unless destroyed => safe to re-init
Example

account act;
//some number of deposit threads
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);
Thread safety

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

`rand()` - what can you do?
Randsafe Example

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

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

    if (error = pthread_mutex_lock (&lock))
        return error;
    *ramp = (rand() + 0.5)/(RAND_MAX + 1.0);
    pthread_mutex_unlock (&lock);
    return;
```
Posix mutex (cont’d)

• Can test if lock is held
  
  ```c
  #include <pthread.h>
  int pthread_mutex_trylock (pthread_mutex_t* mtx)
  ```

  • Returns EBUSY if mtx is held

• Better to create another thread to wait on it
  • Advantage of threads, need not have complex polling, logic, AND many more library/system calls.

• Be careful: why?
Posix mutex (cont’d)

• Locks are limited to protecting shared variables only ... and they are unconditional
• Want richer synchronization
• Condition variables

```c
item_t remove_item (buffer *b){
  item_t st;

  if (b->next_slot_to_retrieve ==
      b->next_slot_to_store) return ERROR // or block
  st = b->items [b->next_slot_to_retrieve];
  b->next_slot_to_retrieve++;

  return st;}
```
Next Time

• CVs!