Basics

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

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

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

\[ \begin{cases} 
  \text{if } (\text{free_buffer}) & \quad \text{if } (\text{free_buffer}) \\
  \quad \text{insert_item} & \quad \text{insert_item}
\end{cases} \]

• 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

```plaintext
Entry <CS> possibly block or spin <CS>
exit
```

• Examples of critical section

```plaintext
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
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

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

Lock is EXPLICIT—have to use it correctly!

```
T1
acquire (L)
access to var X
release (L);
```

```
T2
access X // this is allowed!
```
Synchronization in Posix

- Posix mutex

```c
#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);

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

pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER; // unlocked
```

```
gcc -o myProg myProg c-D REENTRANT -lpthread
```

Misspell -> no warning
Mutex example

```c
pthread_mutex_t acc_mtx = PTHREAD_MUTEX_INITIALIZER;

amount_t depoiterer (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
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

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

int randsafe(double *ramp) {
    *ramp = (rand() + 0.5)/(RAND_MAX + 1.0);
    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
  ```

- Be careful: why?
  ```c
  if (pthread_mutex_trylock (&mtx)!= EBUSY)
    pthread_mutex_lock (&mtx);
  ```

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

• In rare cases, a thread holding a lock may try to reacquire it ...

```c
void foo() {
    pthread_mutex_lock (&lock);
    foo();
    pthread_mutex_lock (&lock);
}
```

Can change lock attributes to be recursive: above code will work.
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
    pthread_mutex_lock (&mtx);

    st = b->items [b->next_slot_to_retrieve];
    b->next_slot_to_retrieve++;

    return st;}
```
Next Time

• CVs!
• Have a great weekend!
CSci 4061
Introduction to Operating Systems

Synch Lecture 2
Condition Variables
Conditional Variables

- Condition variable are a synchronization construct with simple operations:
  - **wait**: means that the process invoking this operation is suspended until another process/thread invokes **signal**
  - **signal**: operation resumes exactly one suspended process/thread. If no process/thread is suspended, then the signal operation has no effect
  - **broadcast**: wakes up all suspended/processes/threads
Conditional Variables (cont’d)

wait (CV*, Lock*)
If called with lock held: sleep, atomically releasing lock. Atomically reacquire lock before returning.

signal (CV*, Lock*)
wake up one waiter, if any

broadcast (CV*, Lock*)
wake up all waiters, if any.

some impl don’t need locks here
Conditional Variables

- *Condition variables* allow *explicit* event notifications
  - Associated with a mutex to prevent races on event conditions.

```c
condition:
acquire (lock);  acquire (lock);
if (...) wait (CV, lock);  if (...)signal (CV, lock);
release (lock);  release (lock);
```
Posix condition variables

#include <pthread.h>

int pthread_cond_signal (pthread_cond_t *cond);
int pthread_broadcast (pthread_cond_t *cond);
int pthread_cond_wait (pthread_cond_t *cond,
                       pthread_mutex_t *mutex);

// also a timed wait
int pthread_cond_destroy (pthread_cond_t *cond);
int pthread_cond_init (pthread_cond_t *cond,
                        const pthread_condattr_t *attr);

pthread_cond_t ccond = PTHREAD_COND_INITIALIZER;
Example #1: Bounded-Buffer

• There is a finite-sized buffer that producer threads want to add items to ... and consumer threads want to remove items from ... repeatedly

• Two kinds of synchronization needed:
  • *Me*—to protect integrity of the buffer
  • *Correctness*—producer must block if buffer is full and consumer must block if buffer is empty...
Example

```c
pthread_mutex_t ring_access = PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t buffer_full = PTHREAD_COND_INITIALIZER;

pthread_cond_t buffer_empty = PTHREAD_COND_INITIALIZER;
```
void buffer_insert (char* item) {
    pthread_mutex_lock (&ring_access);
    while (count == RINGSIZE)
        pthread_cond_wait (&buffer_empty, &ring_access);
    Buffer [in]=item;  // in, out initialized to 0
    in=(in +1)%RINGSIZE;
    count++;
    pthread_cond_signal (&buffer_full);
    pthread_mutex_unlock (&ring_access);
}
Analysis
Remove
Example #2: Barrier

- Barrier: synchronization construct
- Using one synch construct to implement another
  
  ```
  init(int how_many_thread)
  checkin()
  ```
- called by all threads
- blocks all threads until last one checks in
Example #3: License Management
Next Time

• More CVs
CSci 4061
Introduction to Operating Systems

Synch Lecture 3

Reader-Writer + Semaphores
Reader-writer
Threaded-Merger Example
Thought exercise
Threaded-merger
Problem: merge data from multiple input streams into a single output stream

read/write a line at a time ...

read/write a line at a time ...
Thought exercise (cont’d)

• The main program takes a set of filenames and starts a reader thread for each one
• The yellow buffer is like a ring buffer
• Readers exit when they see EOF on their input files
• Writer must distinguish between buffer empty and input done
Semaphore

- Synchronization tool does not require busy waiting
  - **Semaphore operations**

  ```
  create_sem: creates semaphore
  init_sem(ivalue): set value of semaphore to ivalue
  P(): atomic and indivisible (down)
  V(): atomic and indivisible (up)
  P: if value is 0 block, otherwise decrement value
  V: increments value, release if anyone blocked
  ```

- **Internally semaphore structure maintains**
  - Value // semaphore value, always >= 0
  - Queue // list of threads waiting in P() for the value to be >0
Example

• Counting semaphore example:
  • suppose there are $N$ free resources, $n$ threads ($n>N$)

```c
semaphore S; //create
pthread_t t[MaxT]
init_sem (&S, N);
//create N threads
for (i=1; i<n, i++)
  t[i]=pthread_create(...);

void fn(...){
  ...
P(&S);
  //got resource!
  //do something with it
  V(&S);...}
```

• Like a lock, it has a state!
  • $N=1$=> binary semaphore=>mutual exclusion
Posix Semaphores

#include <semaphore.h>
int sem_wait(sem_t* sem); //like P/down
int sem_trywait(sem_t* sem);
int sem_post(sem_t* sem); //like V or up

//pshared=0=>only threads of process can access
int sem_init(sem_t*sem,int pshared,
             unsigned value);

sem_t sem; //this is akin to create
BB with semaphores

//BB of size N with semaphores

sem_t consumer_slots, producer_slots;

sem_init (&consumer_slots, 0, 0);
sem_init (&producer_slots, 0, N);
void buffer_insert(item_t item) {
    sem_wait(&producers_slots); // this is like a P()
    pthread_mutex_lock(&ring_access);
    buffer[in] = item;
    // count++; // NOTE no external state needed
    pthread_mutex_unlock(&ring_access);
    sem_post(&consumer_slots); // this is like a V()
}

<buffer_remove>
Semaphores using Condition Variables

count condition CV;
lock mutex;
void P()
{
    acquire(&mutex);
    while(value==0)
    {
        wait(&CV,&mutex);
        value=value-1;
    }
    release(&mutex);
}

<V()>
Deadlock and Synchronization

Dining philosopher

```c
while (1) {
    get_forks();
    eat();
    eat();
    put_forks();
}
```
Another Example to Ponder: Ping-Pong

```cpp
void PingPong()
{
    while(not done)
    {
        ...
        compute();
        ...
    }
}
```

```cpp
void PingPong()
{
    while(not done)
    {
        ...
        compute();
    }
}
```
Ping-Pong with Mutexes?

```c
Void PingPong()
{
    while(!done)
    {
        Acquire(&mtx);
        compute();
        Release(&mtx);
    }
}
```
Solve with CVs
Ping-Pong with 2 Semaphores?

Semaphore blue = 0;
Semaphore purple = 1;

void PingPong() {
    while(not done) {
        P(&blue);
        Compute();
        V(&purple);
    }
}

void PingPong() {
    while(not done) {
        P(&purple);
        Compute();
        V(&blue);
    }
}
Semaphores vs. Condition Variables

• **P** differs from **wait** in that:
  - P checks the condition and blocks only if necessary
  - No need to recheck the condition after returning from P
  - Wait condition is defined internally, but is limited to a counter

• **V** differs from **signal** in that
  - Signal has no effect if no thread is waiting on the condition.
  - Condition variables are not variables! They have no value!
  - V has the same effect whether or not a thread is waiting
  - Semaphores retain a “memory” of calls to V
Lab #4
Next time

• Start network programming

• Read R&R Chapter 18