Synchronization Basics: Locks
Today

- Thread wrap-up
- Synchronization
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
Basics

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

  ```
  i = 0
  if i == 0
    => i = 5
  else
    i = 7
  
  if i == 0
    => i = 4
  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)
    insert_item
  if (free_buffer)
    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

```plaintext
If (free_buffer)
  insert_item
=> i = 5
else i = 7
```

```plaintext
if i == 0
  => i = 5
else i = 7
```
How to identify a CS: good question!

• Black art

• Conservative (too big) =>
  • Inefficient/lose concurrency

• Too small =?
  • races

• 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 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

```java
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!
Inside a Lock

• Q1
• Q2
Synchronization in Posix

- Posix mutex

```c
#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
```
CSci 4061
Introduction to Operating Systems

Synchronization: Locks, Condition Variables

“Are you OK? You look a little pixelated.”
Today

• Locks finished
• Condition Variables started
Two Locks Deadlock; 2 Threads

• Deadlock - every thread is blocked
Mutex Example

```c
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)

```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;
}
```

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);
    *ramp = (rand() + 0.5)/(RAND_MAX + 1.0);
    pthread_mutex_unlock (&lock);
    return;
}
Randsafe Example

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

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

    pthread_mutex_lock (&lock);
    *ramp = (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
  ```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
  ```c
  • 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_unlock (&lock);
}
```

Can change lock attributes to be recursive: above code will work.

Also need to `un`lock twice!

Limited to the same thread
Posix mutex: Bounded Buffer

Need ME, why?

```c
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;
}
```
Need ME:

```c
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

```c
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

```c
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);
Need Richer Synchronization

// assume we initialize these to ‘held’
pthread_mutex_t full_buffer = …;
pthread_mutex_t empty_buffer = …;

item_t remove_item (buffer *b){
    item_t st;
    pthread_mutex_lock (&mtx);
    if (b->next_slot_to_retrieve == b->next_slot_to_store)
        pthread_mutex_lock(&empty_buffer);
    st = b->items [b->next_slot_to_retrieve];
    b->next_slot_to_retrieve++;
    // adjust next_slot_store if needed
    pthread_mutex_unlock (&full_buffer);
    pthread_mutex_lock (&empty_buffer);

    pthread_mutex_unlock (&mtx);
    return st;
}

Problem?
deadlock

int insert_item (buffer *b, item_t st) {
    pthread_mutex_lock (&mtx);
    if (b->next_slot_to_store == MAX)
        pthread_mutex_lock(&full_buffer);

    // insert_item
    pthread_mutex_unlock (&empty_buffer);
    pthread_mutex_unlock (&mtx);
}
...
Does this work?

```c
item_t remove_item (buffer *b){
    item_t st;
    pthread_mutex_lock (&mtx);
    while (b->next_slot_to_retrieve == b->next_slot_to_store);

    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;
}
```

```c
int insert_item (buffer *b, item_t st) {
    pthread_mutex_lock (&mtx);
    while (b->next_slot_to_store == MAX);

    ... // insert_item

    pthread_mutex_unlock (&mtx);
    ...
}
```
What is lacking?

• Cannot suspend/spin while holding a lock
• OK, let’s try conditional synchronization & remove the mutex

• if <cond> block or spin;
• if <cond> unblock or stop spin;
Does this work?

```c
item_t remove_item (buffer *b) {
    item_t st;
    pthread_mutex_lock (&mtx);
    while (b->next_slot_to_retire == b->next_slot_to_store);
    st = b->items [b->next_slot_to_retire];
    b->next_slot_to_retrieve++;
    // adjust next_slot_store if needed
    pthread_mutex_lock (&mtx);
    return st;
}
```

```c
int insert_item (buffer *b, item_t st) {
    pthread_mutex_lock (&mtx);
    while (b->next_slot_to_store == MAX);
    ...
    // insert_item
    pthread_mutex_lock (&mtx);
    ...
}
```

Does this work?

No, races on <cond>

Need something more powerful
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

• Sounds like a lock!

• Almost ...
Conditional Variables (cont’d)

wait (CV*,Lock*)
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

```c
acquire/lock (&lock);
if/while (<cond>) wait (CV, &lock);
release/unlock (&lock);
```

```c
acquire/lock (&lock);
if/while (!<cond>) signal(&CV, &lock);
release/unlock (&lock);
```

• Associated with a *mutex* to prevent *races* on event conditions

• Atomic sleep to prevent *deadlock*
Example: hello world

Condition CV;
Lock L;
int turn = 1; // hello

T1:
lock (&L);
if (turn == 1)
    print ("hello");
turn = 2;
signal (&CV, &L);
unlock (&L);

T2:
lock (&L);
if (turn != 2)
    wait (&CV, &L);
p
print ("world");
unlock (&L);
Wash then Dry; forever using CVs

```c
enum sink_t {wash, dry} sink = wash;
Condition CV;
Lock L;
T1 (washer):
while (1) {
T2 (dryer):
while (1) {
```
Example #1: License Management

• There are \texttt{MAX\_L} software licenses
• Must call:
  • \texttt{grab\_one} to get a license (block if none free)
  • \texttt{release} when finished

\texttt{grab\_one ();}

\texttt{...}

\texttt{release ();}
Inside \texttt{wait}

\texttt{if lock held => \{release lock; sleep\}}

\texttt{else error}

\texttt{acquire lock}

\texttt{return}

atomic

wakeup and acquire, are \texttt{not} atomic
Example #2: Barrier

• Barrier: synchronization construct
  
  init: how_many_threads
  checkin

• called by all threads
• blocks all threads until last one checks in
Example #2: Barrier

- Barrier: synchronization construct
  - init: `how_many_threads`
  - checkin
- called by all threads
- blocks all threads until last one checks in
Barrier

• Common in parallel threaded programs

for i ... threads work in parallel on i\textsuperscript{th} iteration

\[
x^{(k+1)}_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1, i \neq j}^{n} a_{ij} x^{(k)}_j \right)
\]
typedef struct {
    int n;
    int num_ci;
    lock L;
    condition CV;
} Barrier;

void init (Barrier *B, int num) {
    B->n = num;
    B->num_ci = 0;
}

void checkin (Barrier *B);

//USAGE
Barrier B;

void *thread_fn (...)
{
...
    checkin (&B);
    ...
}

void main (...)
{
    ...
    init (&B, n);
    // launch threads
    ...
}

void checkin (Barrier *B);
Posix condition variables

#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

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);
Bounded-Buffer (two CVs)

• 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;

// consumer: wait for content
pthread_cond_t some_content = PTHREAD_COND_INITIALIZER;

// producer: wait for a free slot
pthread_cond_t free_slot = PTHREAD_COND_INITIALIZER;
```
Example (cont’d)

```c
item_t remove_item (buffer *b){
    item_t st;
    pthread_mutex_lock (&mtx);
    while (b->next_slot_to_retrieve ==
        b->next_slot_to_store)
        pthread_cond_wait(&free_slot,&mtx);

    st = b->items [b->next_slot_to_retrieve];
b->next_slot_to_retrieve++;
    // adjust next_slot_store if needed
    pthreadCond_signal(&some_content);
    pthread_mutex_unlock(&mtx);
    return st;
}
```
Example (cont’d)

```c
void insert_item (buffer *b, item_t st) {
    pthread_mutex_lock (&mtx);
    while (b->next_slot_to_store == MAX)
        pthread_cond_wait (&some_content, &mtx);
    ... 

    // insert_item
    pthread_cond_signal(&free_slot);
    pthread_mutex_unlock(&mtx);
}
```
void insert_item (buffer *b, item_t st) {
    pthread_mutex_lock (&mtx);
    while (b->next_slot_to_store == MAX)
        pthread_cond_wait (&some_content,&mtx);
    // insert_item
    pthread_cond_signal(&free_slot);
    pthread_mutex_unlock(&mtx);
}

item_t remove_item (buffer *b){
    item_t st;
    pthread_mutex_lock (&mtx);
    while (b->next_slot_to_retrieve ==
        b->next_slot_to_store)
        pthread_cond_wait(&free_slot,&mtx);
    st = b->items [b->next_slot_to_retrieve];
    b->next slot to retrieve++;
    // adjust next_slot_store if needed
    pthread_cond_signal(&some_content);
    pthread_mutex_unlock(&mtx);
    return st;
}
Efficiency

• Taking turns

```c
pthread_mutex_t L = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t CV = PTHREAD_COND_INITIALIZER;
int turn = 0;

void* ring (int my_id) {
    while (1) {
        pthread_mutex_lock (&L);
        if (turn == my_id) {
            printf ("%d,", my_id);
            turn = (turn + 1) % N;
            pthread_cond_broadcast (&CV);
        }
        else pthread_cond_wait (&CV, &L);
    }
    pthread_mutex_unlock (&L);
}
```
Lab #3
Exam #2 Coverage

• Closed everything, will provide code APIs

• Material since last exam

• Will post a previous exam and the API
Exam (cont’d)

• Topics:
  • IPC (after pipes): shared memory, messaging passing
  • signals
  • threads
  • locks, CVs

• Closed book
  • You can bring a 1-sided cheat sheet; no magnifying glass allowed.
  • 40% short answer
  • 60% programming/medium (3 questions)
    • IPC
    • signals
    • threads
    • Locks, CVs
Exam (cont’d)

• IPC
  • shared memory, message-passing, issues, programming interfaces

• Signals—how are they used?
  • Dealing with them
  • Issues—reentrancy, races
  • Programming models

• Threads—what good are they?
  • issues: models, implementations, race conditions, pitfalls, programming
  • alternative ways to get concurrency

• Synchronization—what is it?
  • locks
  • condition vars
  • deadlock
Sample

• Short answer:
  • What is a race condition?
  • List two main differences between Unix Pipes and Unix Message Queues?

• Longer
  • Write a multithreaded program to do X?
  • Use signals (e.g. ALARMS) to do Y?
This program contains two threads T1 and T2, and has a serious flaw. What is it? How could you fix it? **Hint:** there are numerous ways to fix this.

```c
int x, y;
pthread_mutex_t xlock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t ylock = PTHREAD_MUTEX_INITIALIZER;
```

T1:
```c
... 
pthread_mutex_lock (&xlock);
x = x + 1;
pthread_mutex_lock (&ylock);
y = y + 1;
pthread_mutex_unlock (&ylock);
pthread_mutex_unlock (&xlock);
```

T2:
```c
... 
pthread_mutex_lock (&ylock);
y = y + 2;
pthread_mutex_lock (&xlock);
x = x + 2;
pthread_mutex_unlock (&xlock);
pthread_mutex_unlock (&ylock);
... 
```
Programming

Make take turns more efficient

Write a program that takes an action every $X$ time units
Study

- Lab 2, 3
- Lecture notes
- Read the book - write multithreaded programs