CSci 4061
Introduction to Operating Systems

Synch Lecture 1
Synchronization Basics
Basics

• Race condition: threads + shared data
• Outcome (data values) depends on who gets there first/last
  
  \[
  i=0 \\
  \text{if } i==0 \quad \text{if } i==0 \\
  \quad \Rightarrow i=5 \quad \Rightarrow i=4 \\
  \text{else } i=7 \quad \text{else } i=8
  \]
• Possible values for i at the end of execution? 7,8,4,5!
• Shared variables = heap, globals, within the process
• Races => inconsistency

  \[
  \text{if (free_buffer) } \quad \text{if (free_buffer)} \\
  \quad \text{insert_item} \quad \text{insert_item}
  \]
• 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

\[
\begin{align*}
&\text{If (free_buffer)} \\
&\text{insert_item}
\end{align*}
\]

\[
\begin{align*}
&\text{if (free_buffer)} \\
&\text{insert_item}
\end{align*}
\]

check at red point
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

```java
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) =>
  • inefficient

• 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
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/blocke
  • stuck/blocke: 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
```

```bash
gcc -o myProg myProg.c -D REENTRANT -lpthread
```
Mutex example

```
pthread_mutex_t acc_mtx = PTHREAD_MUTEX_INITIALIZER;

amount_t depositer (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;
}
```
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<stdio.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
```

• 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();
    foo();
    pthread_mutex_lock (&lock);
}
```

Can change lock attributes to be recursive: above code will work.
Also need to unlock twice!
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;
    pthread_mutex_lock (&mtx);
    if (b->next_slot_to_retrieve ==
        b->next_slot_to_store) return ERROR; // or block
        pthread_mutex_lock (&empty);
    st = b->items [b->next_slot_to_retrieve];
    b->next_slot_to_retrieve++;
    pthread_mutex_lock (&mtx);
    return st;}
```
CSci 4061
Introduction to Operating Systems

Synch Lecture 2
Condition Variables
Today

• Brief lock review
• Condition variables

• Lab #3 extended until Weds 4/14
• Lab #4 (final one) goes out 4/18
void* worker_code (void *arg) {
    int id;
    pthread_mutex_lock (&mutex);
    id = get_an_id ();
    pthread_mutex_unlock (&mutex);

    while (turn != id);

    pthread_mutex_lock (&mutex);
    do_work ();
    turn = turn + 1;
    pthread_mutex_unlock (&mutex);
}

Cont’d

// global declarations if any
int done = 0;

void child_handler (int signo) {
    done = 1;
}

void wait (int n) {
    // set up signal handler for SIGCHLD
    ...

    // block until SIGCHLD is received
    // subtle race, where is it!
    while (!done)
        pause ();
}
Cont’d

void wait (int n) {
  // set up signal handler for SIGCHLD
  ..."sigset_t set1, set2, old;

  sigfillset (&set1);
sigfillset (&set2);
sigdelset (&set2, SIGCHLD);

  // 1. block all signals, remember old set
  sigprocmask (SIG_SETMASK, &set1, &old);

  if (!done)
    // 2. wait until SIGCHLD
    sigsuspend (&set2);

  // 3. restore
  sigprocmask (SIG_SETMASK, &old, NULL);
}
Synchronization

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

[Check at red point]

\[
\text{If (free_buffer) insert_item}
\]

\[
\text{if (free_buffer) insert_item}
\]

• Critical section
  • Only 1 thread at a time in the critical section to prevent races
Locks

• Locks:
  • **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);
Need Richer Synchronization

```c
void buffer_insert(char *item){
    pthread_mutex_lock(&ring_access);
    // buffer full?
    if (count == RINGSIZE) return ERROR;
    Buffer [in] = item; // in, out initialized to 0
    in = (in + 1) % RINGSIZE;
    count++;
    pthread_mutex_unlock(&ring_access);
}
```

Want producer (and consumer) to conditionally block if buffer full/empty
void buffer_insert(char *item) {
    pthread_mutex_lock(&ring_access);
    // buffer full?
    while (count == RINGSIZE);
    Buffer [in] = item; // in, out initialized to 0
    in = (in + 1) % RINGSIZE;
    count++;
    pthread_mutex_unlock(&ring_access);
}

Problem?

deadlock
void buffer_insert(char *item) {
    pthread_mutex_lock(&ring_access);
    // buffer full?
    while (count == RINGSIZE) return ERROR;
    pthread_mutex_lock(&full_buffer);
    Buffer [in] = item;  // in, out initialized to 0
    in = (in + 1) % RINGSIZE;
    count++;
    pthread_mutex_unlock(&ring_access);
}

Problem?
still deadlock
What is lacking?

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

• if <cond> lock or spin;
• if <cond> unlock or spin;

• Does this work?
  • No, races

• 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.
Conditional Variables

- **Condition variables** allow *explicit* event notifications

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

- Associated with a **mutex** to prevent *races* on event conditions
- Atomic sleep to prevent **deadlock**
Inside wait

if lock held => {release lock; sleep}
else error
acquire lock
return
Example #1: License Management

• There are $\text{MAX}_L$ software licenses
• Must call:
  • `grab_one` to get a license (block if none free)
  • `release` when finished
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
  
  \[
  \text{init: how\_many\_threads checkin}
  \]

• called by all threads

• blocks all threads until last one checks in

3 threads
Barrier

• Common in parallel threaded programs

for i ...

threads work in parallel on i\textsuperscript{th} iteration barrier
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>

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);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

//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);
CSci 4061
Introduction to Operating Systems

Synch Lecture 3

Reader-Writer + Examples
Today

• More CVs and examples
• Reader-writer synchronization
void checkin (Barrier *B) {
    lock (&(B->L));
    B->num_ci++;
    if (B->num_ci < B->n) {
        wait (&(B->CV),&(B->L));
    } else {
        broadcast (&(B->CV));
    }
    unlock (&(B->L));
}
Example #3: 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
void buffer_insert(char *item) {
    pthread_mutex_lock(&ring_access);
    while (count == RINGSIZE)
        pthread_cond_wait(&free_slot, &ring_access);
    Buffer[in] = item; // in initialized to 0
    in = (in + 1) % RINGSIZE;
    count++;
    pthread_cond_signal(&some_content);
    pthread_mutex_unlock(&ring_access);
}
```
Example (cont’d)

```c
void buffer_remove()
{
    char *item;
    pthread_mutex_lock(&ring_access);
    while (count == 0)
    
        pthread_cond_wait(&some_content,&ring_access);
    item = Buffer[out]; // out initialized to 0
    out = (out + 1) % RINGSIZE;
    count--;
    pthread_cond_signal(&free_slot);
    pthread_mutex_unlock(&ring_access);
}
```
Analysis

1. <no while?>

2. <1 CV?>
No while

void buffer_insert(char *item) {
    pthread_mutex_lock(&ring_access);
    if (count == RINGSIZE)
        pthread_cond_wait(&free_slot,&ring_access);
    Buffer [in] = item; // in initialized to 0
    in = (in + 1) % RINGSIZE;
    count++;
    pthread_cond_signal(&some_content);
    pthread_mutex_unlock(&ring_access);
}

OK?

ONLY if 1 producer
1 CV?
Efficiency

- Taking turns

```c
pthread_mutex_t L;
pthread_cond_t CV;
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_signal (&CV);
        }
        else pthread_cond_wait (&CV, &L);
        pthread_mutex_unlock (&L);
    }
}
```
Reader-writer

- Implement using CVs
Next time

• Semaphores, synchronization wrap-up

• Start network programming (Thurs)

• Lab #4 out on Monday

• Read R&R Chapter 18

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

Synch Lecture 4
Semaphores + Examples
Today

• More examples
• Semaphores
• Lab #4
Thought exercise

Threaded-merger

Problem: merge data from multiple input streams into a single output stream

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
Threaded-Merger

```c
int done = 0;
int num_readers = N;
Lock L;
Condition free_slot, more_input_or_done;

file_reader () {
    while (!fgets(line, sz, file)) {
        lock (&L);
        while (count == MAX)
            wait (&free_slot, &L);
        add line to BB
        unlock (&L);
        signal (&more_input_or_done, &L);
    }
    lock (&L);
    num_readers--;
    if (num_readers == 0)
        signal (&more_input_or_done, &L);
    unlock (&L);
}

file_writer () {
    while (1) {
        lock (&L);
        if (num_readers != 0 && count == 0)
            wait (&more_input_or_done, &L);
        if (count != 0) {
            get line from BB; write to file
            signal (&free_slot, &L);
        }
        // no more input, no more readers
        else if (num_readers == 0) {
            unlock (&L);
            break;
        }
        else unlock (&L);
    }
```

Thread-safe: diff file

1. Only 1 writer
2. If signaled, condition cannot change

why the if?
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/wait)
  V(): atomic and indivisible (up/post)
  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 state!
  • N=1=> binary semaphore=>mutual exclusion
Posix Semaphores

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

int sem_trywait(sem_t* sem);

//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);
BB: Posix semaphore (cont’d)

```c
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> on your own
```
Semaphores using Condition Variables

```c
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();
    put_forks();
}
```
More Examples
Another Example to Ponder: Ping-Pong

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

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

Void PingPong()
{
    while(not done)
    {
        Acquire(&mtx);
        compute();
        Release(&mtx);
    }
}
Ping-Pong with 2 Semaphores?

Semaphore blue = 0;
Semaphore purple = 1;

```c
void PingPong() {
    while(!done) {
        P(&blue);
        Compute();
        V(&purple);
    }
}
```

```c
void PingPong() {
    while(!done) {
        P(&purple);
        Compute();
        V(&blue);
    }
}
```
Solve with CVs

• What synchronization problem does it resemble?
Semaphores vs. Condition Variables

• P differs from CV 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 CV 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