CSci 4061
Introduction to Operating Systems

Synchronization Basics: Locks
Synchronization Outline

• Basics
• Locks
• Condition Variables
• Semaphores
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

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

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

- check at red point
- If (free_buffer) insert_item
- if (free_buffer) insert_item

- If (free_buffer) insert_item
- if (free_buffer) insert_item
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) =>
  • 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/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
  
  ```
  #include <pthread.h>
  //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

  ```
  pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER; // unlocked
  gcc -o myProg myProg.c -lpthread
  ```
Mutex example

```c
pthread_mutex_t acc_mtx = PTHREAD_MUTEX_INITIALIZER;
amount_t deposerter (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
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?
#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

```
#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);
```

• 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.
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

Synchronization: Condition Variables
Synchronization

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

```
<
If (free_buffer)
insert_item
>
```

Check at red point

- Critical section
  - Only 1 thread at a time in the critical section to prevent races
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
Need Richer Synchronization

```c
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?
Need Richer Synchronization

```c
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?
What is lacking?

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

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

• Does this work?

• 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);
```

- 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

3 threads
Example #2: Barrier

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

3 threads

\[ \text{\_\_\_\_\_} \]
\[ \text{\_\_\_\_\_} \]
Barrier

• Common in parallel threaded programs

for i ...
    threads work in parallel on \( i^{th} \) 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 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;
CSci 4061
Introduction to Operating Systems

Synchronization: Common Paradigms, Semaphores
Today

• Bounded-Buffer
• Reader-writer
• Semaphores

Using CVs
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)

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);
}
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
Synchronization: Semaphores
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]
initSem (&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

//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> 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()>
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
Deadlock and Synchronization

Dining philosopher

while (1) {
    get_forks();
    eat();
    put_forks();
}
