Concurrency review

Glenn Bruns
CSUMB

Exam structure

50 minutes

Concurrency concepts

Synchronization primitives

Pthreads programming

about 60%

Command-line (bash, awk, etc.) about 20%

Process mgmt. and memory mgmt. about 20%

You should be able to write and understand pthreads code.

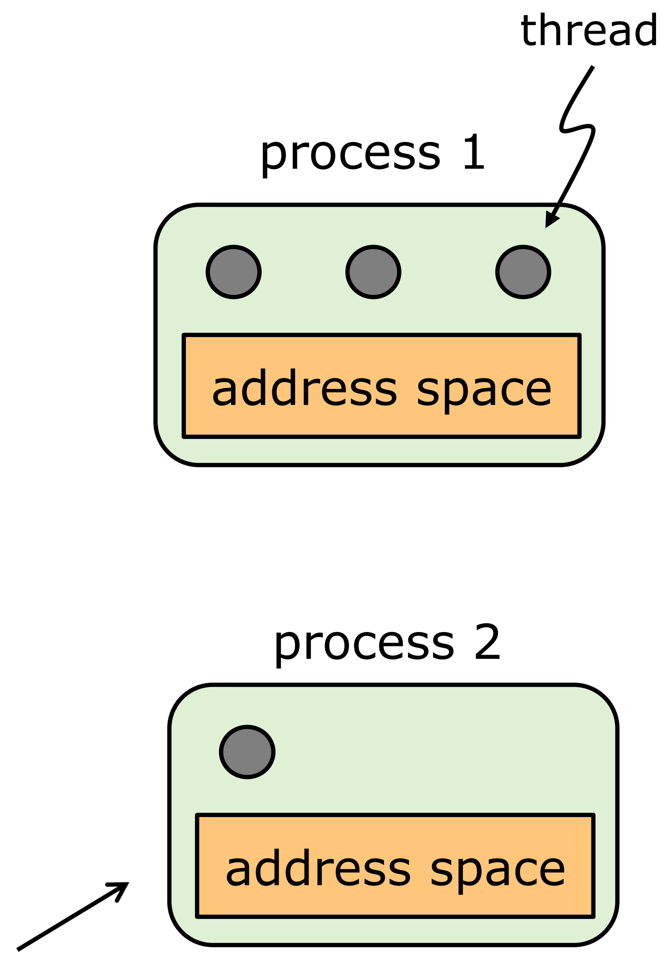
Concurrency concepts

concurrency
process
thread
shared memory
non-determinism
deadlock
critical section
race condition
mutual exclusion
lock
blocking operation
thread-safe data structure

condition variable
spurious wakeup
semaphore
synchronization primitive
synchronization variable
shared data
bounded buffer
synchronization
serialization
execution path
atomicity

Threads and processes

- A process contains one or more threads
- Threads and processes both have state
- Threads are faster to create, destroy, and context switch
- The threads within a process share a virtual address space
- But, each thread has its own stack, and its own registers



a “classic”, single-threaded process

Thread creation

```
int
pthread_create(pthread_t *thread,
               const pthread_attr_t *attr,
               void *(*start_routine)(void*),
               void *arg);
```

thread:	pointer to a pthread_t structure
attr:	thread attributes (or NULL, for defaults)
start_routine:	function to start running
arg:	arguments for start_routine (NULL if no args)

Thread completion

```
int
pthread_join(pthread_t thread,
                void **value_ptr);
```

thread:	a pthread_t structure (not a pointer to one!)
value_ptr:	pointer to the expected return value

Race conditions and mutual exclusion

A **critical section** is a piece of code that accesses a shared resource, such as a shared variable.

When multiple threads are running, and the output depends on the timing of their execution, then the code has a **race condition**.

What we want is that at most one thread at a time can be in the critical section – this is **mutual exclusion**.

Question

Two processes may share memory if they each have multiple threads

a) true

b) false

This is generally false, but there is an exception: processes can share memory if parts of their virtual memory spaces are mapped to the same are of physical memory.

We said this can happen with segmented memory, when two processes sharing running the same program can have their code regions mapped to the same physical memory.

Question

Processes can have zero threads

a) true

b) false

False. Every process has at least one thread.

Lock operations

- A lock can be in one of two states: busy or free
- A lock is initially in the free state
- Operation **lock** waits until the lock is free, and automatically make the lock busy
- Operation **unlock** makes the lock free.

‘lock’ is sometimes called ‘acquire’
‘unlock’ is sometimes called ‘free’

‘lock’ is a blocking operation

(some text here from *Operating Systems: Principle and Practice*, Anderson and Dahlin)

Using locks for mutual exclusion

A **lock** is an object that can be "held" by only one thread at a time.

A **lock** can be acquired by a thread.

A **lock** can be released by a thread.

```
void *mythread(void *arg) {
   int i;
   for (i = 0; i < 1e7; i++) {

      counter = counter + 1;

   }
   return NULL;
}
```

acquire lock here

release lock here

critical section

Question

What would happen if we acquired and released the lock at these locations? (assume two threads)

```
void *mythread(void *arg) {
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }

    return NULL;
}
```

acquire lock here

release lock here

critical section

One thread would do all of its counter increments, then the other thread would do all of its counter increments.

Pthreads locks

```
pthread_mutex_t lock;

void *mythread(void *arg) {
   int i;
   for (i = 0; i < 1e7; i++) {
      Pthread_mutex_lock(&lock);
      counter = counter + 1;
      Pthread_mutex_unlock(&lock);
   }
   return NULL;
}

int main(int argc, char *argv[]) {
   Pthread_mutex_init(&lock);
   pthread_t p1, p2;
   Pthread_create(&p1, NULL, mythread, "A");
   Pthread_create(&p2, NULL, mythread, "B");

   // join waits for the threads to finish
   Pthread_join(p1, NULL);
   Pthread_join(p2, NULL);
   printf("main: done with both (counter = %d)\n", counter);
   return 0;
}
```

declare lock

acquire lock

release lock

initialize lock

Counter code with pthread locks

```
static volatile int counter = 0;
pthread_mutex_t lock;

void *mythread(void *arg) {
   int i;
   for (i = 0; i < 1e7; i++) {
      Pthread_mutex_lock(&lock);
      counter = counter + 1;
      Pthread_mutex_unlock(&lock);
   }
   return NULL;
}

int main(int argc, char *argv[]) {
   Pthread_mutex_init(&lock, NULL);

   pthread_t p1, p2;
   Pthread_create(&p1, NULL, mythread, "A");
   Pthread_create(&p2, NULL, mythread, "B");

   Pthread_join(p1, NULL);
   Pthread_join(p2, NULL);
   printf("main: counter = %d\n", counter);
   return 0;
}
```

This is ugly.

The “application” code is aware of the lock, and has to use it correctly.

It would be better to have a “thread-safe” counter.

(code from Operating Systems: Three Easy Pieces, Arpaci-Dusseau et al)

A counter with locks

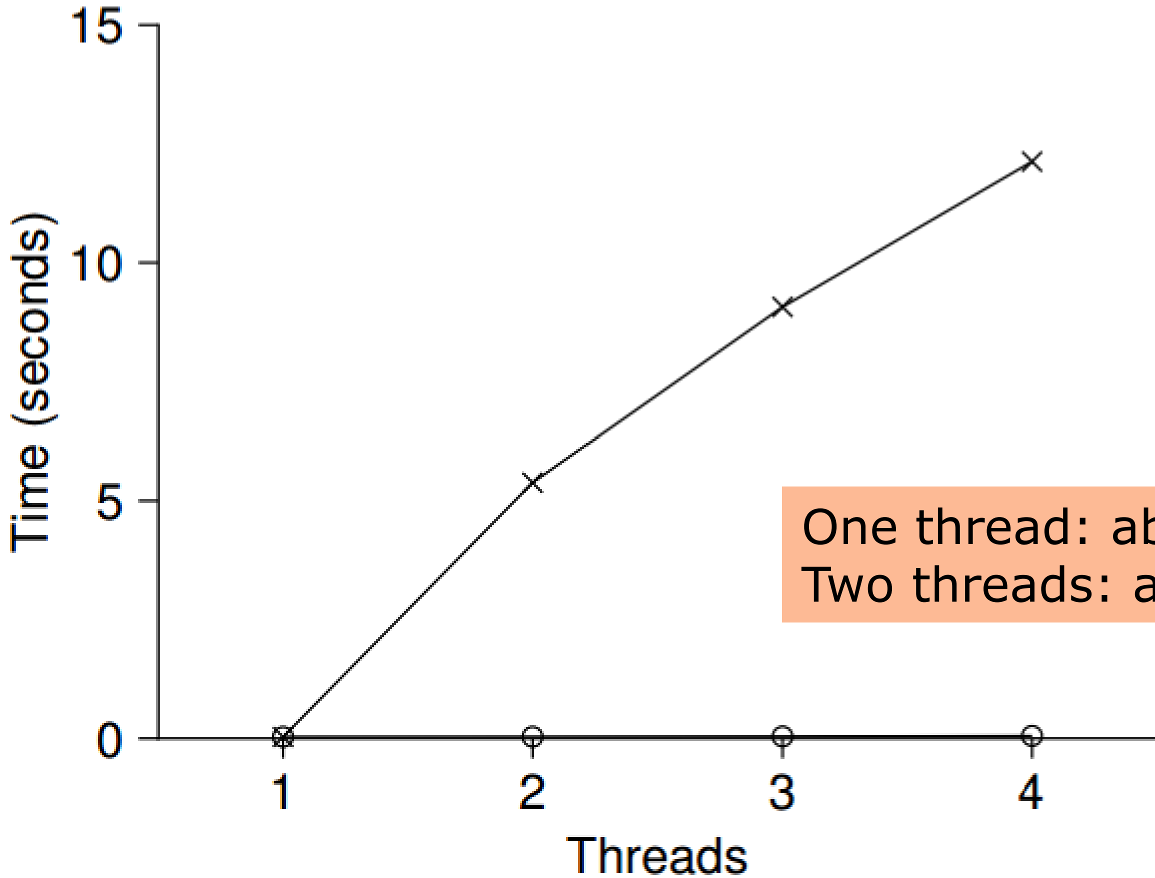
```
typedef struct __counter_t {
  int value;
  pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
  c->value = 0;
  Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
  c->value++;
  Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
  c->value--;
  Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
  int rc = c->value;
  Pthread_mutex_unlock(&c->lock);
  return rc;
}
```

Now we have a counter object that includes locks.

Operations are init, increment, decrement, and get.

Users aren't aware of the locks, but it works correctly with threads.

Counter performance



One thread: about 0.03 seconds
Two threads: about 5 seconds

(plot from Operating Systems: Three Easy Pieces, Arpaci-Dusseau et al)

Question

All the threads of a process run the same code

a) true

b) false

False. Take for example the readers/writers problem. The readers and writers are threads in the same process.

Question

The function 'pthread_create' creates a clone of the currently running thread (like 'fork' of the process API).

a) true

b) false

False. One of the parameters of pthread_create specifies the function that should be run by the newly-created thread.

Condition variables

A condition variable is a synchronization object that lets threads wait efficiently

`pthread_cond_wait(cond, lock)`

- lock is released, calling thread is suspended and put on the condition variable's waiting list
- lock is re-acquired before wait returns

`pthread_cond_signal(cond)`

- takes a thread off the waiting list and marks it as "ready"
- if no thread on the waiting list, does nothing

`pthread_cond_broadcast(cond)`

- like signal, but takes all threads off the waiting list

Question

When a lock is initialized, it is in the 'free' state.

a) true

b) false

True.

Question

Condition variables can be implemented with locks

a) true

b) false

False. The only state of a lock is whether it is busy or free. Condition variables have a queue to keep track of waiting threads.

Question

Which condition variable operations are blocking?

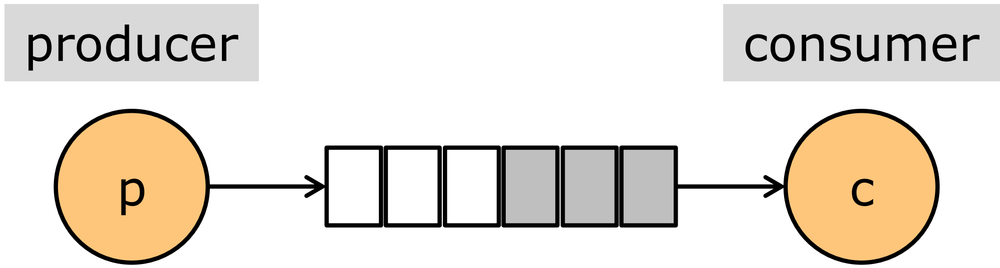
Only wait. A thread will never wait on signal.

Classic concurrency problems

- Bounded buffer
- Readers/writers lock
- Synchronization barrier

The bounded buffer problem

- One thread waits for buffer to be non-full before writing
- Another thread waits for buffer to be non-empty before reading
- Linux pipes use a bounded buffer



Dahlin Method, part 1: class design

- Identify objects
- Define interfaces, and identify state variables
- Implement methods

This is just how you would do class design with an OO language

Part 2: multi-threaded case

- add a lock
- add code to acquire and release the lock
- add zero or more condition variables
- add wait calls within loops
- add signal and broadcast calls

Simple bounded buffer: class design

```
typedef struct {
  // state variables
  int cnt;      // 0 or 1, depending on whether buffer empty or not
  int val;      // value of item in buffer
} SBUF;

// create a new synchronized buffer
SBUF *sbuf_create();

// write to the buffer
void sbuf_write(SBUF *sbuf, int a);

// read from the buffer
int sbuf_read(SBUF *sbuf);
```

This illustrates how to do OO-style code (without inheritance) in C

Implement methods

```
typedef struct {
  int cnt;
  int val;
} SBUF;

// create a new synchronized buffer
SBUF *sbuf_create() {
  SBUF *sbuf = (SBUF *)malloc(sizeof(SBUF));
  sbuf->cnt = 0;
  sbuf->val = 0;
}

// write to the buffer
void sbuf_write(SBUF *sbuf, int a) {
  sbuf->val = a;
  sbuf->cnt = 1;
}

// read from the buffer
int sbuf_read(SBUF *sbuf) {
  int a = sbuf->val;
  sbuf->cnt = 0;
  return(a);
}
```

Add lock, and code to use it

```
typedef struct {
  // state variables
  int cnt;
  int val;

  // sync. variables
  pthread_mutex_t lock;
} SBUF;
```

```
void sbuf_write(SBUF *sbuf, int a) {
  pthread_mutex_lock(&sbuf->lock);
  sbuf->val = a;
  sbuf->cnt = 1;
  pthread_mutex_unlock(&sbuf->lock);
}

int sbuf_read(SBUF *sbuf) {
  pthread_mutex_lock(&sbuf->lock);
  int a = sbuf->val;
  sbuf->cnt = 0;
  pthread_mutex_unlock(&sbuf->lock);
  return(a);
}
```

- normally a shared object will have exactly one lock
- each method begins and end with locking/unlocking

Add condition variables

```
typedef struct {
  // state variables
  int cnt;
  int val;
  // sync. variables
  pthread_mutex_t lock;
  pthread_cond_t read_go;
  pthread_cont_t write_go;
} SBUF;
```

- think about situations in which methods will have to wait
- if method never need to wait, no condition vars needed
- in this step the designer has a lot of freedom

Add wait calls inside loops

```
void sbuf_write(SBUF *sbuf, int a) {
  pthread_mutex_lock(&sbuf->lock);

  while ("buffer full") {
    pthread_cond_wait(&sbuf->write_go, &sbuf->lock);
  }
  sbuf->val = a;
  sbuf->cnt = 1;

  pthread_mutex_unlock(&sbuf->lock);
}
```

```
int sbuf_read(SBUF *sbuf) {
  pthread_mutex_lock(&sbuf->lock);

  while ("buffer empty") {
    pthread_cond_wait(&sbuf->read_go, &sbuf->lock);
  }
  int a = sbuf->val;
  sbuf->cnt = 0;

  pthread_mutex_unlock(&sbuf->lock);
  return(a);
}
```

Why are wait calls within loops?

Add signal and broadcast calls

```
void sbuf_write(SBUF *sbuf, int a) {
  pthread_mutex_lock(&sbuf->lock);

  while (“buffer full”) {
    pthread_cond_wait(&sbuf->write_go, &sbuf->lock);
  }
  sbuf->val = a;
  sbuf->cnt = 1;

  pthread_mutex_unlock(&sbuf->lock);
}
```

```
int sbuf_read(SBUF *sbuf) {
  pthread_mutex_lock(&sbuf->lock);

  while (“buffer empty”) {
    pthread_cond_wait(&sbuf->read_go, &sbuf->lock);
  }
  int a = sbuf->val;
  sbuf->cnt = 0;

  pthread_mutex_unlock(&sbuf->lock);
  return(a);
}
```

- what happens if a thread signals “too much”

- related to idea of “spurious wakeup”

Question

The wait() call on a condition variable requires a parameter for a condition variable and a parameter for a lock.

a) true

b) false

True.

Question

When a thread calls wait(), the lock it passes must be in which state?

a) acquired

b) free

acquired. The thread must be holding the lock when it calls wait().

Question

If your code is designed following the Anderson/Dahlin guidelines, then your code will never break if a 'spurious wakeup' occurs

a) true

b) false

True. A spurious wakeup is when a call to wait() returns even if a signal has not occurred.

Question

In the Anderson/Dahlin style of designing shared objects, each object has one lock

a) true

b) false

True. That lock is used to ensure mutually exclusive access to shared variables.

Semaphores

A semaphore is like a variable, but with three differences:

1. when you create a semaphore, you can initialize it to any value, but afterward there are only two operations you can perform
   - increment
   - decrement
2. when a thread decrements the semaphore, if the result is negative, the thread blocks itself and can't continue until another thread increments the semaphore
3. when a thread increments the semaphore, if there are threads waiting, one of them gets unblocked

Semaphores were invented by Edsgar Dijkstra

2-thread rendezvous with semaphores

```
sem1 = Semaphore(0)
sem2 = Semaphore(0)
```

thread A

```
a1
sem1.signal()
sem2.wait()
a2
```

thread B

```
b1
sem2.signal()
sem1.wait()
b2
```

Mutual exclusion with semaphores

```
sem = Semaphore(1)
```

thread A

```
sem.wait()
   // critical section
   count = count + 1
sem.signal()
```

thread B

```
sem.wait()
   // critical section
   count = count + 1
sem.signal()
```

A “symmetric” solution

Question

initialization:

```
sem = Semaphore(0)
```

thread A

```
sem.signal()
print 'A'
```

thread B

```
sem.wait()
print 'B'
```

What do we know about the output of this program:

a) 'A' must be printed before 'B'

b) 'B' must be printed before 'A'

c) 'A' and 'B' are both printed, in either order

c. The signal() call does not mean that B starts running immediately.

Make sure you understand:

concept of “atomic execution”

use of synchronization variables in Dahlin method

your solutions to concurrency homeworks

how to do address translation with paging

how caching affects memory access time

how to write awk scripts, esp. ones that use associative arrays

how to use find

how to write Make files