CS 31: Intro to Systems
Thread Synchronization

Kevin Webb
Swarthmore College
April 14, 2016

Reading Quiz

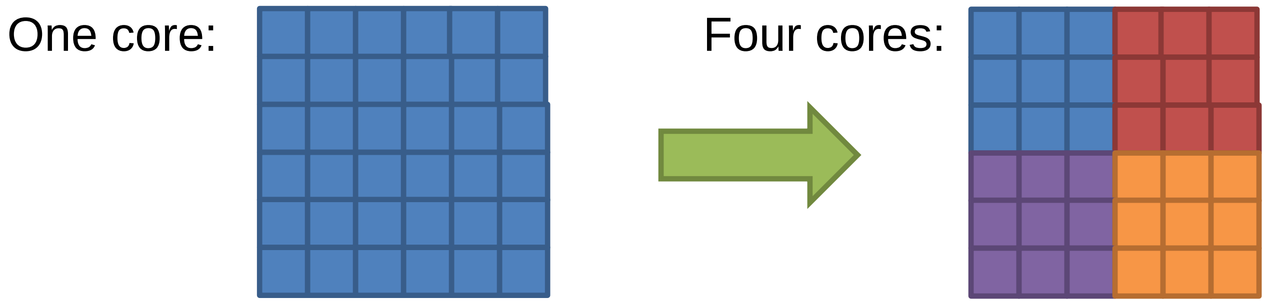
Recap

- To speed up a job, must divide it across multiple cores.
- Thread: abstraction for execution within process.
  - Threads share process memory.
  - Threads may need to communicate to achieve goal
- Thread communication:
  - To solve task (e.g., neighbor GOL cells)
  - To prevent bad interactions (synchronization)

Synchronization

- Synchronize: to (arrange events to) happen at same time (or ensure that they don't)
- Thread synchronization
  - When one thread has to wait for another
  - Events in threads that occur "at the same time"
- Uses of synchronization
  - Prevent race conditions
  - Wait for resources to become available

Synchronization Example



- Coordination required:
  - Threads in different regions must work together to compute new value for boundary cells.
  - Threads might not run at the same speed (depends on the OS scheduler). Can’t let one region get too far ahead.

Thread Ordering

(Why threads require care. Humans aren't good at reasoning about this.)

- As a programmer you have *no idea* when threads will run. The OS schedules them, and the schedule will vary across runs.
- It might decide to context switch from one thread to another *at any time*.
- Your code must be prepared for this!
  - Ask yourself: "Would something bad happen if we context switched here?"

Example: The Credit/Debit Problem

- Say you have $1000 in your bank account
  - You deposit $100
  - You also withdraw $100
- How much should be in your account?
- What if your deposit and withdrawal occur at the same time, at different ATMs?

Credit/Debit Problem: Race Condition

Thread T_0

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Thread T_1

```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Credit/Debit Problem: Race Condition

Say T_0 runs first
Read $1000 into b

Thread T_0

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Thread T_1

```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Credit/Debit Problem: Race Condition

Say T_0 runs first

Read $1000 into b

Switch to T_1

Read $1000 into b

Debit by $100

Write $900

Thread T_0

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Thread T_1

```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Credit/Debit Problem: Race Condition

Say T_0 runs first
Read $1000 into b

Switch to T_1
Read $1000 into b Debit by $100 Write $900

Thread T_0

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Thread T_1

```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Switch back to T_0
Read $1000 into b Credit $100 Write $1100

Bank gave you $100!

What went wrong?

“Critical Section”

Thread T_0

```
Credit (int a) {
  int b;

  b = ReadBalance ();
  b = b + a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Badness if context switch here!

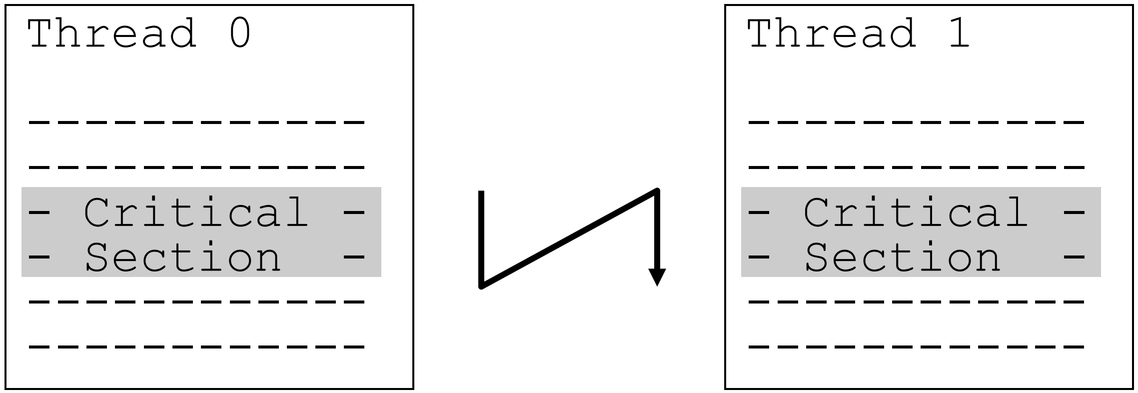
Thread T_1

```
Debit (int a) {
  int b;

  b = ReadBalance ();
  b = b - a;
  WriteBalance (b);

  PrintReceipt (b);
}
```

Bank gave you $100!

What went wrong?

To Avoid Race Conditions



1. Identify critical sections

2. Use synchronization to enforce mutual exclusion
   – Only one thread active in a critical section

What Are Critical Sections?

- Sections of code executed by multiple threads
  - Access shared variables, often making local copy
  - Places where order of execution or thread interleaving will affect the outcome

- Must run atomically with respect to each other
  - Atomicity: runs as an entire unit or not at all. Cannot be divided into smaller parts.

Which code region is a critical section?

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  a += 1

  return a;
}
```

A B C D E

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  a += 1

  return a;
}
```

shared memory

s = 40;

Which code region is a critical section?

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  a += 1

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  a += 1

  return a;
}
```

shared memory

s = 40;

Which values might the shared s variable hold after both threads finish?

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  return a;
}
```

shared memory

s = 40;

A. 30
B. 20 or 30
C. 20, 30, or 50
D. Another set of values

If A runs first

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  return a;
}
```

shared memory

s = 50;

B runs after A Completes

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  return a;
}
```

shared memory

s = 30;

What about interleaving?

Thread A

```
main ()
{ int a,b;

  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);

  return a;
}
```

shared memory

s = 40;

Is there a race condition?

Suppose `count` is a global variable, multiple threads increment it:

`count++;`

A. Yes, there's a race condition (`count++` is a critical section).
B. No, there's no race condition (`count++` is not a critical section).
C. Cannot be determined

How about if compiler implements it as:

```
movl  (%edx), %eax    // read count value
addl  $1, %eax        // modify value
movl  %eax, (%edx)    // write count
```

How about if compiler implements it as:

```
incl  (%edx)          // increment value
```

Four Rules for Mutual Exclusion

1. No two threads can be inside their critical sections at the same time.
2. No thread outside its critical section may prevent others from entering their critical sections.
3. No thread should have to wait forever to enter its critical section. (Starvation)
4. No assumptions can be made about speeds or number of CPU's.

How to Achieve Mutual Exclusion?

```
< entry code >
< critical section >
< exit code >
```

```
< entry code >
< critical section >
< exit code >
```

- Surround critical section with entry/exit code
- Entry code should act as a gate
  - If another thread is in critical section, block
  - Otherwise, allow thread to proceed
- Exit code should release other entry gates

Possible Solution: Spin Lock?

```
shared int lock = OPEN;
```

T_0

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

T_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

- Lock indicates whether any thread is in critical section.

Note: While loop has no body. Keeps checking the condition as quickly as possible until it becomes false. (It "spins")

Possible Solution: Spin Lock?

```
shared int lock = OPEN;
```

T_0

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

T_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

- Lock indicates whether any thread is in critical section.
- Is there a problem here?
  - A: Yes, this is broken.
  - B: No, this ought to work.

Possible Solution: Spin Lock?

```
shared int lock = OPEN;
```

T_0

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

T_1

```
while (lock == CLOSED);
lock = CLOSED;
< critical section >
lock = OPEN;
```

- What if a context switch occurs at this point?

Possible Solution: Take Turns?

```
shared int turn = T0;
```

T_0

```
while (turn != T0);
< critical section >
turn = T1;
```

T_1

```
while (turn != T1);
< critical section >
turn = T0;
```

- Alternate which thread can enter critical section
- Is there a problem?
  - A: Yes, this is broken.
  - B: No, this ought to work.

Possible Solution: Take Turns?

```
shared int turn = T0;
```

T_0

```
while (turn != T0);
< critical section >
turn = T1;
```

T_1

```
while (turn != T1);
< critical section >
turn = T0;
```

- Rule #2: No thread outside its critical section may prevent others from entering their critical sections.

Possible Solution: State Intention?

```
shared boolean flag[2] = {FALSE, FALSE};
```

T0

```
flag[T0] = TRUE;
while (flag[T1]);
< critical section >
flag[T0] = FALSE;
```

T1

```
flag[T1] = TRUE;
while (flag[T0]);
< critical section >
flag[T1] = FALSE;
```

- Each thread states it wants to enter critical section
- Is there a problem?
  – A: Yes, this is broken.
  – B: No, this ought to work.

Possible Solution: State Intention?

```
shared boolean flag[2] = {FALSE, FALSE};
```

T_0

```
flag[T0] = TRUE;
while (flag[T1]);
< critical section >
flag[T0] = FALSE;
```

T_1

```
flag[T1] = TRUE;
while (flag[T0]);
< critical section >
flag[T1] = FALSE;
```

- What if threads context switch between these two lines?
- Rule #3: No thread should have to wait forever to enter its critical section.

Peterson's Solution

```
shared int turn;
shared boolean flag[2] = {FALSE, FALSE};
```

T_0

```
flag[T0] = TRUE;
turn = T1;
while (flag[T1] && turn==T1);
< critical section >
flag[T0] = FALSE;
```

T_1

```
flag[T1] = TRUE;
turn = T0;
while (flag[T0] && turn==T0);
< critical section >
flag[T1] = FALSE;
```

- If there is competition, take turns; otherwise, enter
- Is there a problem?
  - A: Yes, this is broken.
  - B: No, this ought to work.

Spinlocks are Wasteful

- If a thread is spinning on a lock, it's using the CPU without making progress.
  - Single-core system, prevents lock holder from executing.
  - Multi-core system, waste core time when something else could be running.

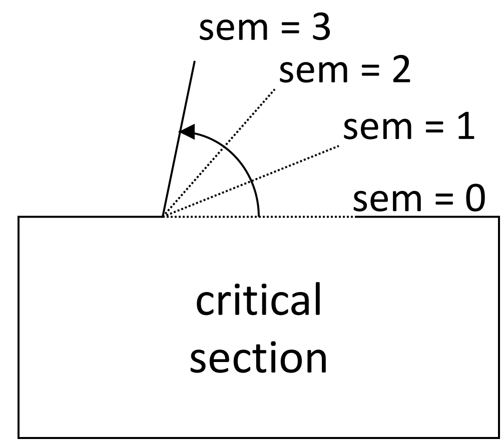
- Ideal: thread can't enter critical section? Schedule something else. Consider it *blocked*.

Atomicity

- How do we get away from having to know about all other interested threads?
- The implementation of acquiring/releasing critical section must be atomic.
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes “all or nothing”
- How do we make them atomic?
  - Atomic HW instructions (e.g., test-and-set)
  - Allows us to build “semaphore” abstraction

Semaphores

- Semaphore: synchronization variable
  - Has integer value
  - List of waiting threads
- Works like a gate
- If sem > 0, gate is open
  - Value equals number of threads that can enter
- Else, gate is closed
  - Possibly with waiting threads



Semaphores

- Associated with each semaphore is a queue of waiting threads
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread

Semaphore Operations

```
sem s = n;    // declare and initialize

wait (sem s)     // Executes atomically
    decrement s;
    if s < 0, block thread (and associate with s);

signal (sem s)  // Executes atomically
    increment s;
    if blocked threads, unblock (any) one of them;
```

Semaphore Operations

```
sem s = n;    // declare and initialize

wait (sem s)    // Executes atomically
    decrement s;
    if s < 0, block thread (and associate with s);

signal (sem s)  // Executes atomically
    increment s;
    if blocked threads, unblock (any) one of them;
```

Based on what you know about semaphores, should a process be able to check beforehand whether wait(s) will cause it to block?

A. Yes, it should be able to check.
B. No, it should not be able to check.

Semaphore Operations

```
sem s = n;    // declare and initialize

wait (sem s)
    decrement s;
    if s < 0, block thread (and associate with s);

signal (sem s)
    increment s;
    if blocked threads, unblock (any) one of them;
```

- No other operations allowed
- In particular, semaphore’s value can’t be tested!
  - No thread can tell the value of s

Mutual Exclusion with Semaphores

```
sem mutex = 1;
```

T_0

```
wait (mutex);
< critical section >
signal (mutex);
```

T_1

```
wait (mutex);
< critical section >
signal (mutex);
```

- Use a “mutex” semaphore initialized to 1
- Only one thread can enter critical section
- Simple, works for any number of threads
- Is there any busy-waiting?

Locking Abstraction

- One way to implement critical sections is to “lock the door” on the way in, and unlock it again on the way out
  - Typically exports “nicer” interface for semaphores in user space

- A lock is an object in memory providing two operations
  - acquire()/lock(): before entering the critical section
  - release()/unlock(): after leaving a critical section

- Threads pair calls to acquire() and release()
  - Between acquire()/release(), the thread holds the lock
  - acquire() does not return until any previous holder releases
  - What can happen if the calls are not paired?

Using Locks

Thread A

```
main ()
{ int a,b;



  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);



  return a;
}
```

shared memory

s = 40;

Thread B

```
main ()
{ int a,b;



  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);



  return a;
}
```

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

s = 40;

Lock l;

Held by: Nobody

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

s = 40;

Lock l;

Held by: Thread A

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

s = 40;

Lock l;

Held by: Thread A

Using Locks

Lock already owned.

Must Wait!

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

s = 40;

Lock l;

Held by: Thread A

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

s = 50;

Lock l;

Held by: Nobody

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

s = 30;

Lock l;

Held by: Thread B

Using Locks

Thread A

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 10;
  a = a + b;
  saveShared(a);
  release(l);

  return a;
}
```

Thread B

```
main ()
{ int a,b;

  acquire(l);
  a = getShared();
  b = 20;
  a = a - b;
  saveShared(a);
  release(l);

  return a;
}
```

shared memory

s = 30;

Lock l;

Held by: Nobody

- No matter how we order threads or when we context switch, result will always be 30, like we expected (and probably wanted).

Summary

- We have no idea when OS will schedule or context switch our threads.
  - Code must be prepared, tough to reason about.
- Threads often must synchronize
  - To safely communicate / transfer data, without races
- Synchronization primitives help programmers
  - Kernel-level semaphores: limit # of threads that can do something, provides atomicity
  - User-level locks: built upon semaphore, provides mutual exclusion (usually part of thread library)