Thread Synchronization

11/17/16
Threading: core ideas

• Threads allow more efficient use of resources.
  • Multiple cores
  • Down time while waiting for I/O

• Threads are better than processes for parallelism.
  • Cheaper to create and context switch
  • Easier to share information

• Threading makes programming harder.
  • Need to think about how to split a problem up
  • Need to think about how threads interact
Create and Join

• Each process starts with a single thread.
• Any thread can spawn new threads with `create`.
  • Starts a new call stack for the thread.
  • `create` specifies what function the thread starts with.
    • Processes always start with `main`.
    • Different threads can start with different functions.
  • Returns the ID of the new thread.

• `join` causes one thread to block until another thread completes.
  • `join` must specify the ID of the thread to wait for.
  • `join` gives access to the thread function’s return value.
Create and Join example

```c
main(){
    double x = 1, y = -1;
    tid t1, t2;
    double res;
    t1 = create(worker, x);
    t2 = create(worker, y);
    res = join(t1);
    res += join(t2);
    printf("%d\n",res);
}

worker(double d){
    do_work(&d);
    return d;
}
```

**IMPORTANT:** this is not correct C code. We will talk about the pthreads library next week.
Create and Join illustrated

main thread

create()
create()

join(t1)
main thread waits for thread 1 to terminate

join(t1) returns
join(t2)
join(t2) returns

printf() exit()
terminates main thread and any peer threads

do_work(&d)
return d;
do_work(&d)
return d;

(peer threads terminate)
Thread Ordering
(Why threads require care. Reasoning about this is hard.)

• 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 T₀
Credit (int a) {
    int b;

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

    PrintReceipt (b);
}

Thread T₁
Debit (int a) {
    int b;

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

    PrintReceipt (b);
}
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);
}

Say $T_0$ runs first
Read $\$1000$ into $b$
Credit/Debit Problem: Race Condition

Thread T₀

Credit (int a) {
    int b;
    b = ReadBalance ();
    b = b + a;
    WriteBalance (b);
    PrintReceipt (b);
}

Say T₀ runs first
Read $1000 into b
Switch to T₁
Read $1000 into b
Debit by $100
Write $900

Thread T₁

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 $900$
PrintReceipt ($b$);

Say $T_0$ runs first
Read $1000$ into $b$
Switch to $T_1$
Read $1000$ into $b$
Debit by $100$
Write $900$
PrintReceipt ($b$);

Switch back to $T_0$
Read $1000$ into $b$
Credit $100$
Write $1100$
PrintReceipt ($b$);

Race Condition: outcome depends on scheduling order of concurrent threads.

Bank gave you $100$!

What went wrong?

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);
}

Say $T_0$ runs first
Read $1000$ into $b$
Switch to $T_1$
Read $1000$ into $b$
Debit by $100$
Write $900$
PrintReceipt ($b$);

Switch back to $T_0$
Read $1000$ into $b$
Credit $100$
Write $1100$
PrintReceipt ($b$);

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 ($b$);
}

Thread $T_1$

Debit (int $a$) {
    int $b$;
    $b$ = ReadBalance ();
    $b$ = $b$ - $a$;
    WriteBalance ($b$);
    PrintReceipt ($b$);
}

Bank gave you $100$!

What went wrong?

Badness if context switch here!
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

```c
void thread_main()
{
    int a, b;
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    a += 1
    return a;
}
```

Thread B

```c
void thread_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

```c
thread_main ()
{
    int a, b;
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    return a;
}
```

Thread B

```c
thread_main ()
{
    int a, b;
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    return a;
}
```

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

Thread A

```c
main ()
{
    int a,b;
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    return a;
}
```

Thread B

```c
main ()
{
    int a,b;
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    return a;
}
```
B runs after A Completes

Thread A

```c
main ()
{
    int a,b;
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    return a;
}
```

Thread B

```c
main ()
{
    int a,b;
    a = getShared();
    b = 20;
    a = a - b;
    saveShared(a);
    return a;
}
```
What about interleaving?

Thread A

```c
main ()
{ int a,b;
    a = getShared();
    b = 10;
    a = a + b;
    saveShared(a);
    return a;
}
```

Thread B

```c
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 \texttt{count} is a global variable, multiple threads increment it:
\texttt{count++;}

A. Yes, there’s a race condition (\texttt{count++} is a critical section).
B. No, there’s no race condition (\texttt{count++} is not a critical section).
C. Cannot be determined.

How about if compiler implements it as:

\begin{verbatim}
movl (%edx), %eax  // read count value
addl $1, %eax     // modify value
movl %eax, (%edx) // write count
\end{verbatim}

How about if compiler implements it as:

\begin{verbatim}
incl (%edx)        // increment value
\end{verbatim}
Mutex Locks

The OS provides the following atomic operations:

• Acquire/lock a mutex.
  • If no other thread has locked the mutex, claim it.
  • If another thread holds the mutex, block.
  • Threads unblocked in FIFO order.
• Release/unlock a mutex.

To enforce a critical section:

• Before the critical section, lock the mutex.
• After the critical section unlock the mutex.
Using Locks

Thread A

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

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

    return a;
}
```

Thread B

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

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

    return a;
}
```

shared memory

s = 40;
Using Locks

Thread A

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

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

    return a;
}
```

Thread B

```c
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

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

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

    return a;
}
```

Thread B

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

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

    return a;
}
```

s = 40;
Lock l;

Held by: Thread A
Using Locks

Thread A

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

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

    return a;
}
```

Thread B

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

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

    return a;
}
```

```markdown
s = 40;
Lock l;
```

Held by: Thread A
Using Locks

Thread A

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

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

  return a;
}
```

Thread B

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

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

  return a;
}
```

Locked memory

- Lock 1

Held by: Thread A

Lock already owned. Must Wait!
Using Locks

Thread A

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

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

    return a;
}
```

Thread B

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

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

    return a;
}
```

```
s = 50;
Lock l;
```

Held by: Nobody
Using Locks

Thread A

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

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

    return a;
}
```

Thread B

```c
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

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

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

    return a;
}
```

Thread B

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

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

    return a;
}
```

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

Sometimes we want all threads to catch up to a specific point before we continue.

• Think about parallelizing the polygons simulator.
  • We could split up regions of the world across threads.
  • We don’t want one thread to start round 2 before another has finished round 1.

Solution: barriers

• A thread that calls barrier_wait will block until all other threads have also called barrier_wait.
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round();
        barrier_wait(&b)
    }
}

Barrier (0 waiting)
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Threads make progress computing current round at different rates.

Barrier (0 waiting)
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round();
        barrier_wait(&b)
    }
}

Threads that make it to barrier must wait for all others to get there.
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Barrier allows threads to pass when N threads reach it.
Barrier Example, N Threads

shared barrier b;

init_barrier(&b, N);

create_threads(N, func);

void *func(void *arg) {
    while (...) {
        compute_sim_round()
        barrier_wait(&b)
    }
}

Threads compute next round, wait on barrier again, repeat...

T0 T1 T2 T3 T4
Thread operations

• create
  • Starts a new thread, calling a specified function.
  • Returns the thread’s ID.

• join
  • Block until a specified thread terminates.
  • Gives access to the thread function’s return value.

• mutex_lock
  • Block until the mutex is available, then claim it.

• mutex_unlock
  • Release a mutex.

• barrier_wait
  • Block until a specified number of threads reach the barrier.
Devise a parallel algorithm for max

Write pseudocode for main and a thread function that uses (some of) create, join, mutex_lock, mutex_unlock, and barrier_wait.

• Array size M
• N threads

• Version 1: each thread returns its local max
• Version 2: each thread updates a global max
• Version 3: the thread that found the max prints