Based on slides by Harsha V. Madhyastha

EECS 482 Introduction to Operating Systems

Spring/Summer 2020

Lecture 3: Synchronization

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Announcements

Get started on Project 1.

- Due in two weeks, May 27
- Read the project spec carefully!
- Get familiar with the tools, git, gdb, VS Code, etc.
- (Valgrind will not work with P1.)

Attempt pre-lab questions for Friday's lab section.

Today

Ensuring that you have enough but not too much milk at home

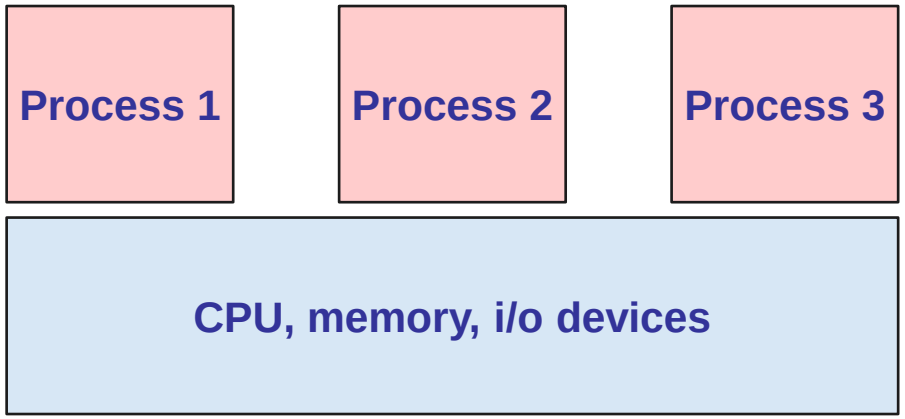
Managing threads

What do these have to do with each other?

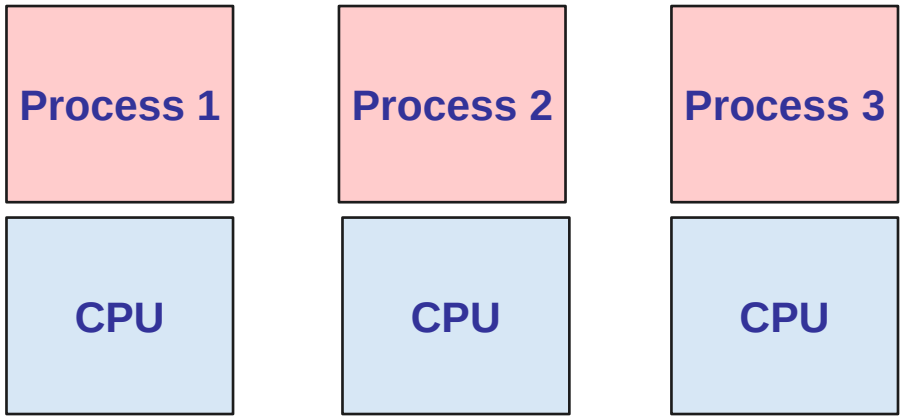
The Process

The process is the OS abstraction for execution.

The reality



The abstraction



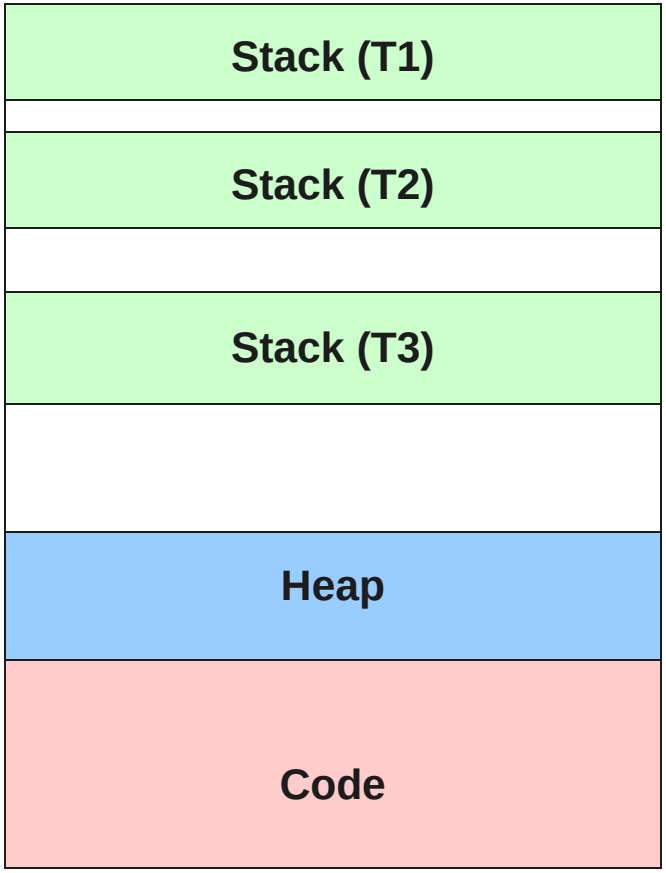
Recap: Threads

Benefits:

- Simplify concurrent programming.
- Useful when there is a slow resource.

Challenge:

- Share parts of address space.
- Prevent undesired outcomes?



Non-deterministic ordering → Non-deterministic results

Arithmetic example (y is initially 10)

What's being shared between these threads?

y

Possible results?

If A runs first: x = 11 and y = 20

If B runs first: x = 21 and y = 20

Thread A	Thread B
x = y + 1	y = y * 2

Non-deterministic ordering → Non-deterministic results

Another example

Possible results?

x = 1 or -1

Impossible results?

x = 0

Thread A	Thread B
x = 0	x = 0
x = 1	x = -1

Non-deterministic ordering → Non-deterministic results

A final example

Possible results?

x = 0, 1 or -1

Impossible results?

x = 2, -2

Thread A	*Thread B*
x = 0	x = 0
x++	x--

Atomic operations

Before we can reason at all about cooperating threads, we must know that some operation is *atomic.*

1. It's indivisible. It happens in its entirety or not at all.
2. No events from other threads can occur in between when it starts and when it finishes.

Atomic operations

On most computers:

1. Memory load and store are atomic.
2. Many other instructions, e.g., double precision floating point, are not atomic.

Need an atomic operation to build bigger atomic operations.

Example

Thread 1	Thread 2
Print ABC	Print 123

What if each print statement were atomic?

What if printing a single character were not atomic?

Example

Assume i is a global shared variable.

Thread A

```
i = 0;
while ( i < 10 )
   i++;
print "A finished";
```

Thread B

```
i = 0;
while ( i > -10 )
    i--;
print "B finished";
```

Which thread will exit its while loop first?

Is the thread that exits first guaranteed to print first?

Is it guaranteed that anything will print?

Debugging Multi-Threaded Programs

Challenging due to non-deterministic interleaving.

Heisenbug: a bug that occurs non-deterministically

Something for you to worry about?

YES!!!

Think Murphy's Law.

- Famous errors:

1. Northeast blackout of 2003
2. Over-radiation in Therac-25

- All possible interleavings must be correct.

Synchronization

Objective:

Constrain interleavings between threads such that all possible interleavings produce a correct result.

Trivial solution:

Run each until it finishes before starting the next but that defeats the purpose of threads.

Challenge:

Constrain thread executions as little as possible

Insight:

Some events are independent → order is irrelevant

Other events are dependent → order matters

Too much milk

Problem definition:

1. Obama family wants to always have one jug of milk.
2. No room for two jugs of milk.
3. Whoever sees the fridge empty goes to buy milk.

Solution 0, no synchronization.

Barack

```
if ( noMilk )
    buy milk;
```

Michelle

```
if ( noMilk )
    buy milk;
```

Problems?

Race condition!

First type of synchronization: Mutual exclusion

Ensure that only 1 thread is doing a certain thing at any moment in time.

"Only 1 person goes shopping at a time"

Constrains interleavings of threads

Does this remind you of any other concept we've talked about?

Critical section

A section of code that needs to be run atomically with respect to selected other pieces of code.

Critical sections must be atomic w.r.t each other because they access a shared resource.

In our example, the critical section is:

```
if ( nomilk )
   buy milk;
```

How do we make this critical section atomic?

Barack

```
if ( noMilk )
    buy milk;
```

Michelle

```
if ( noMilk )
    buy milk;
```

Solution 1

Leave a note that you're going to check on the milk, so the other person doesn't also buy.

Assume the only atomic operations are load and store.

Does this work?

Is it better than solution 0?

A little, there's a smaller window when both might go out, but they could still end up with too much milk.

Barack

```
if ( noNote )
   {
   leave note;
   if ( noMilk )
      buy milk;
   remove note;
   }
```

Michelle

```
if ( noNote )
   {
   leave note;
   if ( noMilk )
      buy milk;
   remove note;
   }
```

Solution 2

Label the notes and change the order of “leave note” and “check note”.

Problems?

Nobody buys milk.

Barack

```
leave noteBarack;
if ( noNoteMichelle )
   if ( noMilk )
      buy milk;
remove noteBarack;
```

Michelle

```
leave noteMichelle;
if ( noNoteBarack )
   if ( noMilk )
      buy milk;
remove noteMichelle;
```

Solution 3

Decide who buys milk when both leave notes at same time.

Barack hangs around to make sure job is done.

Barack's "while (noteMichelle)" prevents him from entering the critical section at the same time as Michelle.

Barack

```
leave noteBarack;
while ( noteMichelle )
   ;
if ( noMilk )
    buy milk;
remove noteBarack;
```

Michelle

```
leave noteMichelle;
if ( no noteBarack )
   if ( noMilk )
      buy milk;
remove noteMichelle;
```

Proof of correctness

if no noteM, it's safe to buy. (He's already left noteBarack, which Michelle will check.)

if noteM, Barack waits to see what Michelle does and decides whether to buy after Michelle exits.

if no noteB, then Barack hasn't started yet, so it's safe to buy. (Barack will wait for Michelle to be done before checking.)

if noteB, then Barack will eventually buy milk if needed. (Barack may be waiting for Michelle to exit.)

Barack

```
leave noteBarack;
while ( noteMichelle )
    ;
if ( noMilk )
     buy milk;
remove noteBarack;
```

Michelle

```
leave noteMichelle;
if ( no noteBarack )
   if ( noMilk )
      buy milk;
remove noteMichelle;
```

Analysis of solution 3

Good

1. It works!
2. Relies on simple atomic operations.

Bad

1. Complicated and not obviously correct.
2. Asymmetric.
3. Not obvious how to scale to three people.
4. Barack consumes CPU time while waiting, called *busy-waiting.*

Barack

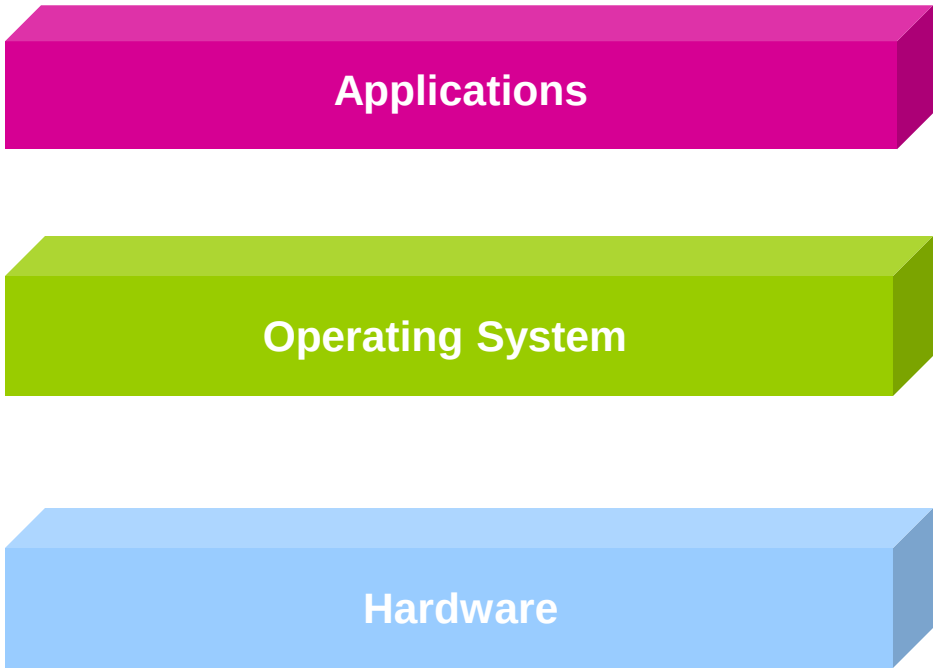
```
leave noteBarack;
while ( noteMichelle )
    ;
if ( noMilk )
     buy milk;
remove noteBarack
```

Michelle

```
leave noteMichelle;
if ( no noteBarack )
    if ( noMilk )
        buy milk;
remove noteMichelle;
```

Higher-level synchronization

Raise the level of abstraction to make life easier for programmers.



Concurrent programs

Higher-level synchronization primitives
(lock, monitor, semaphore)

Atomic operations
(load/store, interrupt enable/ disable, test&set)

Locks (mutexes)

A lock prevents another thread from entering a critical section

> *"Lock fridge while checking milk status and shopping"*

Two operations:

1. lock( ), wait until the lock is free, then acquire it.
2. unlock( ), release the lock.

Checking and acquiring must be atomic.

Why was the note solutions 1 and 2 not a good lock?

```
lock( )
   {
   while ( true )
      if ( lock is free )
      {
      acquire lock;
      break;
      }
   }

unlock( )
   {
   release lock;
   }
```

Atomic

Solution using locks

Lock usage:

1. Initialized to free.
2. Acquire lock before entering critical section.
3. Release lock when done with critical section.

All synchronization involves waiting.

Threads can be running or blocked.

Barack

```
milk.lock( );
if ( noMilk )
    buy milk;
milk.unlock( );
```

Michelle

```
milk.lock( );
if ( noMilk )
    buy milk;
milk.unlock( );
```

Efficiency

But this prevents Michelle from doing things while Barack is buying milk.

Can we minimize the time the lock is held?

Barack

```
milk.lock( );
if ( noMilk )
    buy milk;
milk.unlock( );
```

Michelle

```
milk.lock( );
if ( noMilk )
    buy milk;
milk.unlock( );
```

Efficiency

Use a lock to protect posting or viewing of any notes.

```
note.lock( );
if ( noNote )
   {
   leave note;
   note.unlock( );
   if ( noMilk )
      buy milk;
   note.lock( );
   remove note;
   }
note.unlock( );
```