Week 4: Synchronization & Concurrency

TABLE OF CONTENTS

Agenda

1. Review; Where are we now?
2. Finish discussing scheduling from last week
3. Introduce Concurrency and Synchronization
   - This week will be conceptual: the motivation, protential problems
   - Next week will be implementation: how to apply in practice

Weekly Summary and Where are we?

Topics

- **Last Week:** Introduction to systems in general
- **This Week:** Introduction to the Process abstraction and how to use it in C
- **Next Week:** How the OS schedules processes (and other things)

Assignments

- **Lab 2** (building `ls`) is out; due FRIDAY.
- **Lab 3** (multithreading) is out; due Feb 13.
- **HW 3** is due on Friday
- Grades for Lab 0, Lab 1, HW1 are out; regrade requests must be received before THIS FRIDAY.

Reading Summary

- OSTEP Chapter 25: Dialogue
- OSTEP Chapter 26: Concurrency and Threads
- OSTEP Chapter 27: Thread API
- OSTEP Chapter 28: Locks

Week 4

- Lab 2
- Lab 3: Multi-threading
- HW3
- Grades
  - HW1
  - L0
  - L1

OSTEP:

- Virtualization
- Concurrency
- Persistence
- Security

Kernel

- Reliability
- Security
- Privacy
- Fair resource allocation

Back to Scheduling

5 things:

- FIFO
- RR → shortest response
- MLFQ
- SJF
- STCF → Shortest turnaround
- Priority
- Lottery

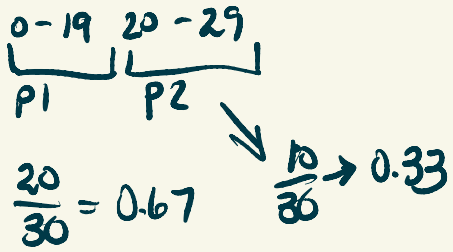
- Turnaround
- Response
- Throughput

Lottery ⇒ Stride scheduling

t = ~~100~~ 30

p1: 20 tickets
p2: 10 tickets

scheduler: 25



- Encapsulates priority
- "Expected" running time
- Non-deterministic
- Random
  - Mersenne Twister

Concurrency:

- Doing multiple things at once.

Synchronization

- force determinism into non-deterministic processes.

Threads:

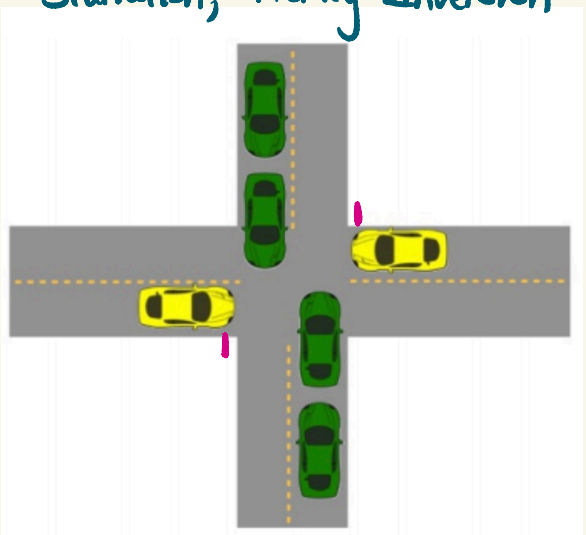
Me:	Ben
• Eat Breakfast	• Eat Breakfast
• Work	• Wait for phone call
• Lunch	• Lunch
• Call Ben	

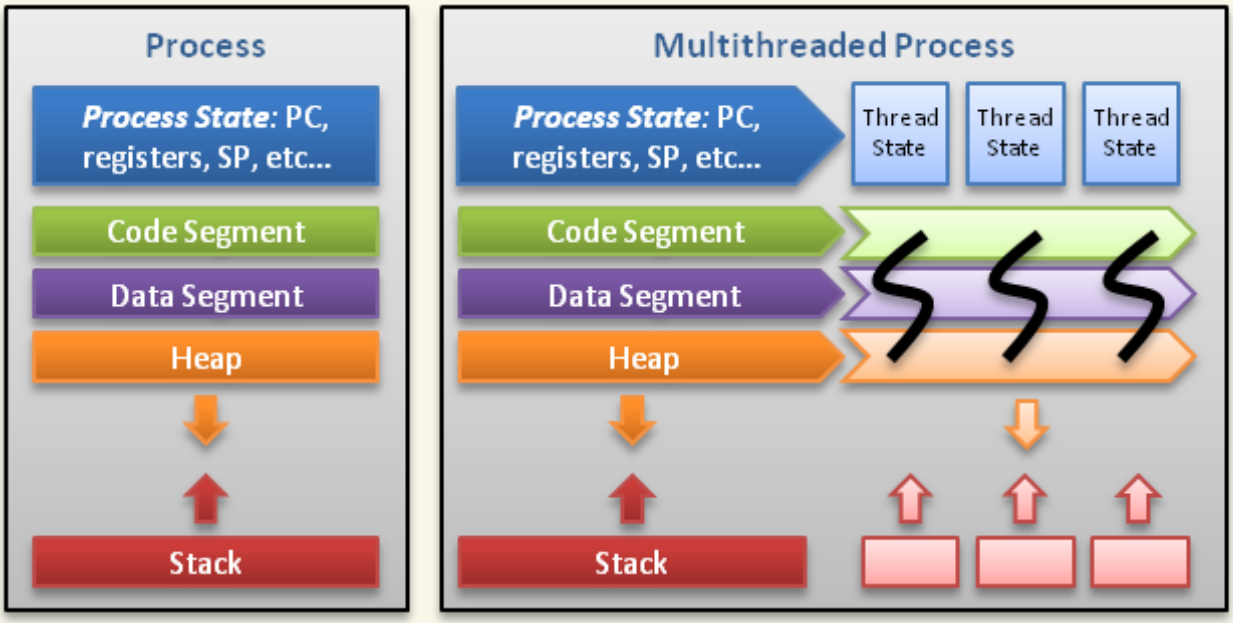
- Ben + I ate lunch "sequentially"
- Ben + I ate breakfast "concurrently"

Deadlock

Race

Starvation, Priority Inversion





Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

A little context/reminder/the big picture

- A *process* is the execution of a program with restricted rights
  - The process is the "abstraction for protected execution provided by the operating system kernel"
- The ***OS kernel*** is *trusted* code
  - It's the lowest level of software running on a computer/system
  - It has full access to all of the hardware
  - It's the "brain" of the operating system, but it's not the entire operating system, and it's not all the software that is part of an OS
- The *kernel* is responsible for:
  - **Reliability**: Prevent bugs in one program from causing system crashes or problems in other programs
  - **Security**: Ensure users or applications can't do malicious things to the system, such as deleting random files
  - **Privacy**: Ensure each user only has access to data they are permitted to access.
  - **Fair resource allocation**: Ensure that all processes/users use the resources they need without imposing on resources used by other processes/users.

Our book, OSTEP: Operating Systems, Three Easy Pieces

The three easy pieces:

- Virtualization
- Concurrency
- Persistence
- Security (a fourth piece :))

Back to Scheduling

Pick up from last week; see notes where we left off at Lottery and Stride Scheduling.

First, list 5 things!

- 1)
- 2)
- 3)
- 4)
- 5)

Intro to Concurrency, Threading, and Synchronization

- *Concurrency* is doing multiple things at the same time.
- *Synchronization* is how we can force some determinism into inherently non-deterministic concurrent processes.
- *Threads* is one way to implement concurrency within a program.

Intro to concurrency

Concurrency happens everywhere; we'll talk about it in terms of processes and specifically threads, but we see the same problem in many different contexts (pretty much any shared resource!)

- What is concurrency?
  - Stuff happening at the same time
- Sources of concurrency
  - on a single CPU, processes/threads can have their instructions interleaved (helpful to regard the instructions in multiple threads as "happening at the same time")
  - computers have multiple CPUs and common memory, so instructions in multiple threads can happen at the same time!
  - interrupts (CPU was doing one thing; now it's doing another)
- Reasons to use concurrency

    - Program structure: expressing logically concurrent tasks
    - Responsiveness: allowing work to happen in the background
    - Performance: utilize multiple processors
    - Performance: managing I/O
- Why is concurrency hard?
    - Hard to reason about all possible interleavings
    - (deeper reason: human brain is "single-threaded")

Real-World Examples

- You have a friend Ben.
- One day, you start to wonder: who ate lunch first today? You, or Ben? How do you find out?
- Call him and ask!
    - What if you had lunch starting at 11:59, and he had lunch starting at 12:01?
- Unless you know that both of you have accurate clocks, you can't be sure who ate first.

Goal: Guarantee you eat before Ben. How do you do this?

Your instructions:

```
Eat breakfast
Work
Eat lunch
Call Ben
```

Ben's Instructions:

```
Eat breakfast
Wait for a call
Eat lunch
```

- You and Ben ate lunch ***sequentially***: we know the order of events
- You and Ben ate breakfast ***concurrently***, because we don't know the order.
- **Non-deterministic**: It's not possible to tell what will happen when a program executes simply by looking at it.
- **Deterministic**: We know exactly how a program will behave when executed.
- **Concurrency**: Two things are ***concurrent*** if we cannot tell what will happen first just by looking at the code.
- **Synchronization**: Using tricks and tools to enforce order.

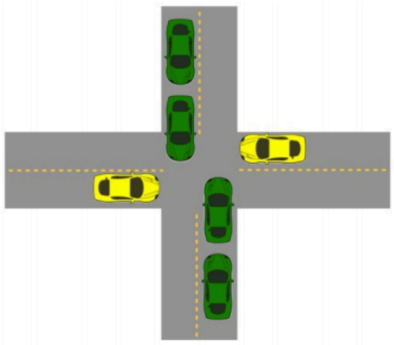
What could go wrong?

A race condition:

So you're out of milk...:

Deadlock:

Priority inversion/starvation:



NOTE: These problems all happened because resources were being shared!

The Thread Interface

Interface to POSIX threads (how to use threads in C/Unix):

- `tid thread_create (void (*fn) (void *), void *);`
  - Create a new thread, run `fn` with arg
- `void thread_exit ();`
- `void thread_join (tid thr);`
  - Wait for thread with tid `thr` to exit

Assume for now that threads are:

- an abstraction created by OS
- preemptively scheduled

THREAD VS. PROCESS: A FREQUENTLY ASKED QUESTION

The term *process* has changed over time in this field.

It was originally meant as what we consider a *thread* now: a logical sequence of instructions that executes either OS or application code. There was little concern about protecting the overall system between application programs.

As parallelization became more common, we needed a distinction between a process that ran a program with multiple instruction sequences. These were referred to as *lightweight processes* for a while.

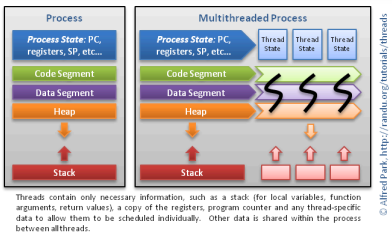
Current convention: A process executes a program, consisting of one or more threads running inside a protection boundary.

- Threads share memory, but they have their own *execution context* (registers and stack).
- Similar to PCB, there is TCB (thread control block)
  - **Thread Identifier**: Unique id (tid) is assigned to every new thread
  - **Stack pointer**: Points to thread's stack in the process
  - **Program counter**: Points to the current program instruction of the thread
  - State of the thread (running, ready, waiting, start, done)
  - Thread's register values
  - Pointer to the Process control block (PCB) of the process that the thread lives on

NOTE

Google uses user-level threads for performance.

See talk "User-level threads..... with threads." by Paul Turner from Google



Code Example 1: Concurrent Writes

Reference the following code section that illustrates interleaving.

- Say thread A executes `f()` and thread B executes `g()`.

```c
int x;

int main(int argc, char** argv) {

    tid tid1 = thread_create(f, NULL);
    tid tid2 = thread_create(g, NULL);

    thread_join(tid1);
    thread_join(tid2);

    printf("%d\n", x);

}

void f() {
    x = 1;
    thread_exit();
}

void g() {
    x = 2;
    thread_exit();
}
```

$x = 2$

$x = 1$

QUESTION

What are the possible values of `x` after A has executed `f()` and B has executed `g()`? (That is, what are the possible outputs of the program above?)

Answer

- **Global Variables**: declared outside a function. Can be accessed by any thread.
- **Local automatic variables**: Declared inside a function without the static attribute. Each thread's stack contains its own instance.
- **Local static variables**: Declared inside a function with static attribute. Every thread reads and writes the same instance.

Code Example 2: One thread reads, another writes

Okay, now let's say that `f()` and `g()` are defined as below:

```
int x;
int y = 12;

f() { x = y + 1; }
g() { y = y * 2; }
```

What are the possible values of `x`?

Answer

Code Example 3: Concurrent Updates

One more time, with `f()` and `g()` defined as below:

```
int x = 0;

f() { x = x + 1; }
g() { x = x + 2; }
```

What are the possible values of `x`?

Answer

Explanation:

Let's say `x` is stored at mem location `0x5000`.

The statement (for `f()`) `x = x+1;` will compile to something like:

```
movq 0x5000, %rbx    # load from address 0x5000 into register
addq $1, %rbx        # add 1 to the register's value
movq %rbx, 0x5000    # store back
```

And for `g()` we have `x = x+2;`, which might compile to:

```
movq 0x5000, %rbx    # load from address 0x5000 into register
addq $2, %rbx        # add 2 to the register's value
movq %rbx, 0x5000    # store back
```

Concurrency is hard when we share resources

The problem arises when we try to share resources.

One common way for threads to share data is to share variables that live outside the threads

- One thread reads a variable that another thread has written to
- Two (or more) threads write to a single variable
- Two (or more) threads read and write a single variable (update)
- Two (or more) threads reading the same variable rarely causes problems

Case 1: One thread writes; another reads

- If the threads are unsynchronized, we can't tell whether the reader will see the original value or the written value
- One constraint we can enforce is: Reader should not read until after the writer writes.
  - This is like lunch with Ben: He's not going to eat until I tell him I ate.
- That is, synchronization by policy or convention

Case 2: Concurrent Writes

- We care about 2 things here:
  - What gets printed?
  - What's the final value of x?

Thread A

```
x = 5
print x
```

Thread B

```
x = 7
```

- **Execution path**: Order of execution
- What are the possible execution paths that:
  - yield output 5 and final value 5? `b1 < a1 < a2`
  - yield output 7 and final value 7? `a1 < b1 < a2`
  - yield output 5 and final value 7? `a1 < a2 < b1`

Case 3: Concurrent Updates

Update: Read the value of a variable, compute a new value based on the old value, and write the new value to the variable.

Thread A

```
temp = x
x = temp + 1
```

Thread B

```
temp = x
x = temp + 1
```

What happens with this order of execution: `a1 < b1 < b2 < a2`

- Both threads read the same value, so they both write the same value...
- Would the same thing happen if we had written `x++` ?
  - Maybe... depends on the computer.
- **atomic**: An operation that cannot be interrupted.

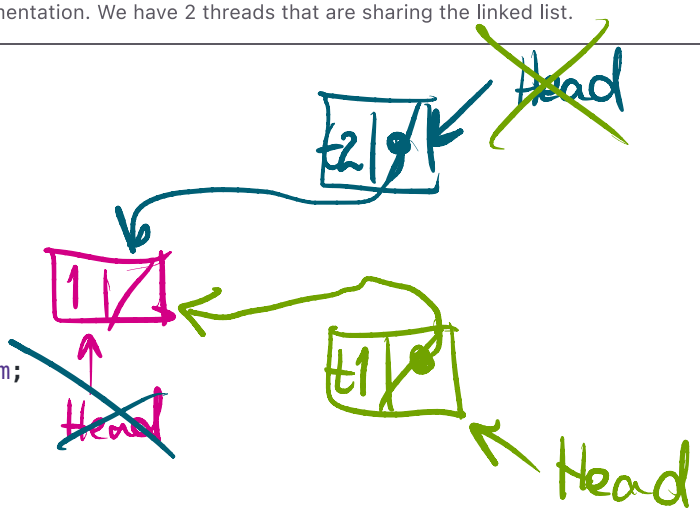
Example: Threads and a Linked List

Let's say we have the following Linked List implementation. We have 2 threads that are sharing the linked list.

```
struct List_elem {
    int data;
    struct List_elem* next;
};

List_elem* head = 0;

insert(int data) {
    List_elem* l = new List_elem;
    l->data = data;
    l->next = head;
    head = l;
}
```



What happens if two threads execute `insert()` at once and we get the following interleaving?

```
thread 1: l->next = head
thread 2: l->next = head
thread 2: head = l;
thread 1: head = l;
```

- both threads will insert new `List_elem` as head
- one `List_elem` will be lost (later threads cannot access it)

–> This will result in an incorrect (unexpected) linked list!

Example: Using a shared buffer (producer/consumer)

Incorrect count in buffer

```
        producer --+                +--> consumer
                   |                |
                   +---> [buffer] ---+
                      (count, in, out)
```

"buffer" of course is not a real ring; it is an array with a wrap around pointer.

Question: why might go wrong with this incorrect count [answer: the producer may overwrite items; the consumer may take out NULL.]

Takeaways

- all of these are called **race conditions**; not all of them are errors, though
- all of these can happen on one-CPU machine (multi-core case can be worse; will see soon)
- revisit: an example you've seen, HW2

```c
int main()
{
  fork();
  fork();

  printf("\nhello world");
}
```

- expected outputs: [many possibilities; should see four newlines ("\n") and four "hello world", but their order is uncertain.]
- What are you going to do when debugging concurrency code?
  - Concurrent code is hard to debug!!
    - It's nondeterministic
  - Race conditions are hard because a program may work fine most of the time and only occasionally show problems.
    - Why? (because the instructions of the various threads or processes or whatevever get interleaved in a non-deterministic order.)
  - And it's worse than that because inserting debugging code may change the timing so that the bug doesn't show up

Solving Concurrent Problems: Synchronization

- Hardware makes the problem even harder

[From S.V. Adve and K. Gharachorloo, IEEE Computer, December 1996, 66–76. http://sadve.cs.illinois.edu/Publications/computer96.pdf]

a. Can both "critical sections" run?

```c
int flag1 = 0, flag2 = 0;

int main () {
    tid id = thread_create (p1, NULL);
    p2 ();
    thread_join (id);
}

void p1 (void *ignored) {
    flag1 = 1;
    if (!flag2) {
        critical_section_1 ();
    }
}

void p2 (void *ignored) {
    flag2 = 1;
    if (!flag1) {
        critical_section_2 ();
    }
}
```

Can both critical sections run?

b. Can `use()` be called with value `0`, if `p2` and `p1` run concurrently?

```c
int data = 0, ready = 0;

void p1 () {
    data = 2000;
    ready = 1;
}

int p2 () {
    while (!ready) {}
    use(data);
}
```

use(0) ?

c. Can `use()` be called with value `0`?

```c
int a = 0, b = 0;

void p1 (void *ignored) { a = 1; }

void p2 (void *ignored) {
    if (a == 1)
        b = 1;
}

void p3 (void *ignored) {
    if (b == 1)
        use (a);
}
```

Nope!

- ***Sequential consistency*** is not always in effect

- Sequential consistency means:
  - (1) maintain a single sequential order among operations
    - namely: all memory operations can form a sequential order where the read operations read from the most recent writes.
  - (2) maintain program order on individual processors
    - namely: all operations from the same thread follow their issue order in the program
- A concurrent execution is **equivalent** to a sequential execution when

1. Both must have the same operations
2. (1) All (corresponding) reads return the same values
3. (2) The final state are the same

- If a memory has *sequential consistency*, it means that:
  - A concurrent program running on multiple CPUs looks as if the same program (with the same number of threads) running on a single CPU.
    - OR: There always *exists* a sequential order of memory operations that is *equivalent* to what the concurrent program produces running on multiple CPUs.
- On multiple CPUs, we can get "interleavings" *that are impossible on single-CPU machines*.
  - In other words, the number of interleavings that you have to consider is *worse* than simply considering all possible interleavings of sequential code.
- explain why sequential consistency is not held:
  - CPUs -> Memory (too slow)
  - CPUs -> Caches -> Memory (Write stall: invalidating cache requires a long time)
  - CPUs -> StoreBuffers -> Caches -> Memory
  - more...
  - out-of-order execution
  - speculative execution

[thanks to David Mazieres, Mike Dahlin, and Mike Walfish for content in portions of this lecture.]