Introduction to Parallel Programming

- What is Parallel Programming?
- Why Parallel?
- Processes and Threads
- Creating a thread (Java, C)
- Races and Locks
What is Parallel Programming?

- **Parallel Programming** is:
  - Writing multiple programs that are designed to run at the same time

- **Examples:**
  - Multiple processes in your OS
  - Distributed algorithms
  - **Multithreading** (this slide deck)
What is Multithreading?

- Multithreading is:
  - Multiple “threads” (programs)
  - Running in the same memory (globals, heap)
  - At the same time

- But with different stacks (local variables)
- And different program counters
- And different registers
Any number of CPUs can be connected by a “bus” to a single memory.

The same address accesses the same word, from any different CPU.
Sharing a Memory

• Since they share a memory, each thread can see changes made by other threads (to main memory)

CPU 0
load x
inc
store x

CPU 1
load x
print

Memory
x=0
Sharing a Memory

- Since they share a memory, each thread can see changes made by other threads (to main memory)

CPU 0

load x
inc
store x
----
x=0

CPU 1

load x
print

Memory

x=0
Sharing a Memory

- Since they share a memory, each thread can see changes made by other threads (to main memory)

CPU 0

load x
inc
store x
----
x=1

CPU 1

load x
print

Memory

x=0
Sharing a Memory

- Since they share a memory, each thread can see changes made by other threads (to main memory)

```plaintext
CPU 0
load x
inc
store x
----
x=1

CPU 1
load x
print
```

Memory

x=1
Sharing a Memory

- Since they share a memory, each thread can see changes made by other threads (to main memory)

CPU 0
load x
inc
store x
----
x=1

CPU 1
load x
print
----
x=1

Memory
x=1
Sharing a Memory

- Since they share a memory, each thread can see changes made by other threads (to main memory)

CPU 0
load x
inc
store x
----
x=1

Output:
----
1

Memory
x=1

CPU 1
load x
print
----
x=1
Why Parallel?

- Key Uses for Parallelism:
  - Separate different tasks
    - Blocking semantics
  - Improve performance
    - Multiple CPUs mean multiple ALUs
    - Do useful work while one thread stalls
  - Isolation (multiprocessing)
Reason 1: Blocking Semantics

• Many function calls have **blocking semantics**
  - Do a task
  - Wait for an event
  - Don't return until complete

• Examples:
  - `Scanner.next()`
  - `fopen()`
  - `printf()`
  - `isPrime()`
Reason 1: Blocking Semantics

• Many function calls have **blocking semantics**
  - Do a task
  - Wait for an event
  - Don't return until complete

• Examples:
  - `Scanner.next()`  \(\rightarrow\) Wait on user
  - `fopen()`  \(\rightarrow\) Wait on I/O
  - `printf()`  \(\rightarrow\) Consume CPU
  - `isPrime()`
Reason 1: Blocking Semantics

- Single threaded programs cannot “do something else” while a long call is running.
  - How would you implement the following program?

```java
findNextPrime();
while searching for primes,
    listen for keyboard, mouse
draw progress bar
check email
play YouTube video
```
Reason 2: Improving Performance

• A single CPU has an upper bound on its performance
  – Max instructions per second
  – Max ALU ops per second

• Multiple CPUs can spread the work out
  – The Dream: Linear speedup
  – The Reality: Never perfect
Reason 2: Improving Performance

- Some problems are easy to work on in parallel.
- Many are not.

```
for i=0 to 999
doWork(i)
```

```
for i=1000 to 1999
doWork(i)
```
Reason 2: Improving Performance

- Occasionally, two CPUs share an ALU. When one stalls, the other still does useful work.
Reason 3: Isolation (Multiprocessing)

- Each program in your computer has its own thread (or several)

- If the OS works well, one program should not be able to affect the other ones
  - Can't monopolize CPU
  - Can't corrupt memory

How to protect memory? Look up “virtual memory” on Google and Wikipedia.
Processes and Threads

• A **process** is the state of a currently running program.
  - One memory context
  - One set of open files
  - etc.

• A **thread** is one chain of execution in a process
  - One program counter
  - One stack
  - One set of registers
Processes and Threads

- **process** Factorio
  - thread GUI
  - thread network
  - thread sound
  - thread simulation1
  - thread simulation2
  - thread simulation3
  - thread simulation4

- **process** Mozilla
  - thread window1
  - thread window2
  - thread window3
  - thread download
  - thread javascript
  - thread vidPlayer

- **process** bash
  - thread main

- **process** bash
  - thread main

- **process** gmail
  - thread GUI
  - thread checkMail
Processes and Threads

• Every time you run a program, a new process is created
  – Every process starts with 1 thread

• **Inside** the program, sometimes a process asks for more threads
  – Users usually have no control over this
Q: How to write a thread?
A: You already have done it!

```c
int main(int argc, char **argv)
{
    ... this is a thread ...
}
```
Creating a New Thread

- Most (imperative) languages represent each thread with a function
  - When the thread starts, the function runs
    - One parameter, usually
  - When the function returns, the thread dies
    - (Or, call `thread_exit()`.)
Creating a Thread in Java

- Create a class which implements the `Runnable` interface
  - Create one copy of this object!
- Create a `Thread` object
  - Pass your `Runnable` as constructor parameter
- `Thread.start()`
- `Java calls` `Runnable.run()` on your object
Creating a Thread in C (pthreads)

- **pthreads** is a famous threading library (for *NIX)

```c
void *worker(void *arg);

---

pthread_create(NULL, NULL,
    &worker, argValue);
```
main():

printf("foo\n");
pthread_create(...);
printf("bar\n");

worker():

printf("baz\n");

Current Running Threads

main()
----
foo
create()
bar

Output: ----
main():

printf("foo\n");
pthread_create(...);
printf("bar\n");

worker():

printf("baz\n");

Current Running Threads

main()
----
foo
create()
bar

Output:
----
foo
main():
printf("foo\n");
pthread_create(...);
printf("bar\n");

worker():
printf("baz\n");

Current Running Threads

main()
----
foo
create()
bar

worker()
----
baz

Output:
----
foo
main():
printf("foo\n");
pthread_create(...);
printf("bar\n");

worker():
printf("baz\n");

Current Running Threads

main()
----
foo
create()
bar

worker()
----
baz

Output:
----
foo

Q: Which is printed first, bar or baz?
Q: Which is printed first, bar or baz?
A: Undefined
Races and Locks

- The order in which threads do their work is **undefined**.
  - Each thread runs its code without any regard for other threads.

- Threads can run:
  - Arbitrarily fast
  - Arbitrarily slow
  - Slow, then fast!
Races

• A **race** is a condition where:
  - Two or more threads run at the same time
  - Trying to access the same variables
  - Different runtime orders would produce different results

• Often, races are **rare and hard to find.**
  - Program “normally” runs one way
  - Occasionally runs differently
Perhaps, in your experiments, bar gets printed first (because it takes a while for the worker to start).

But if `main()` pauses suddenly, baz might get printed first!

```
main()
----
foo
create()
bar
```

```
worker()
----
baz
```

Output:
```
    ----
foo
```
Causes of Races

• Why do threads get slow without warning?
  - Interrupts
  - Page faults
  - Multitasking (swapped out of the CPU)
  - Cache misses
  - Contention for hardware (ALUs, memory bus, etc.)

• You cannot predict when a thread might pause.
  - Pauses can take arbitrarily long.
Intra-Statement Races

• Races can happen in the middle of statements.

• What is the MIPS assembly for this statement? (Assume that counter is a word in memory.)

    counter++;
Group Exercise:

Assume that counter is 1 before either thread runs. What are the possible values that counter might have after both threads are done?
main()
----
la   $t0, counter
lw   $t1, 0($t0)
addi $t1, $t1, 1
sw   $t1, 0($t0)

worker()
----
la   $t0, counter
lw   $t1, 0($t0)
addi $t1, $t1, 1
sw   $t1, 0($t0)

Group Exercise:

Assume that `counter` is 1 before either thread runs. What are the possible values that `counter` might have after both threads are done?

Answer: 2 or 3

How is 2 possible???
Possible Sequence:
- **main()** reads counter (1)
- **worker()** reads counter (still 1!)
- Both threads increment their registers
- Both threads write back (both 2!)
Locks

- There are many ways to solve races. The simplest are **locks**.
- A lock indicates that the data is being written (or read) by a thread, and other threads should not interfere.
Locks

while lock is busy
    wait a while

grab lock
    do work
release lock
Oops, our lock didn't work

- Wouldn't a lock be susceptible to races, too?
  - Yes!

Group Exercise:

Look at the (broken) lock pseudocode below. Imagine that there are two threads running it at the same time. Describe (as exactly as you can) how this might fail.

```plaintext
while lock is busy
    wait a while
    grab the lock
```
How Spin Loops Fail

• Wouldn't a lock be susceptible to races, too?
  – Yes!

Another thread:
do work
release lock

Lock State:
busy

while lock is busy
wait a while
grab the lock

while lock is busy
wait a while
grab the lock
How Spin Loops Fail

- Wouldn't a lock be susceptible to races, too?
  - Yes!

```
while lock is busy
  wait a while
grab the lock
```

```
while lock is busy
  wait a while
grab the lock
```

Another thread:
```
do work
release lock
```

Lock State:
```
busy
```
Oops, our lock didn't work

- Wouldn't a lock be susceptible to races, too?
  - Yes!

Another thread:
- do work
- release lock

<table>
<thead>
<tr>
<th>Lock State:</th>
<th>free</th>
</tr>
</thead>
</table>

```
while lock is busy
  wait a while
grabs the lock
```

```
while lock is busy
  wait a while
grabs the lock
```
Oops, our lock didn't work

- Wouldn't a lock be susceptible to races, too?
  - Yes!

while lock is busy
  wait a while
  grab the lock

Another thread:
do work
release lock

while lock is busy
  wait a while
  grab the lock

Lock State:
  free
Oops, our lock didn't work

- Wouldn't a lock be susceptible to races, too?
  - Yes!

Another thread:
do work
release lock

Lock State:
free

while lock is busy
    wait a while
    grab the lock
Oops, our lock didn't work

• Wouldn't a lock be susceptible to races, too?
  – Yes!

Another thread:
do work
release lock

Lock State:
free
Oops, our lock didn't work

- Wouldn't a lock be susceptible to races, too?
  - Yes!

while lock is busy
  wait a while
  grab the lock

Another thread:
do work
release lock

while lock is busy
  wait a while
  grab the lock

Lock State:
busy
Oops, our lock didn't work

• Wouldn't a lock be susceptible to races, too?
  – Yes!

Another thread:
do work
release lock

while lock is busy
wait a while
grab the lock

while lock is busy
wait a while
grab the lock

Lock State: busy
Atomic Operations

- Locks are implemented using **atomic operations**.
  - One instruction reads **and** writes the same value
  - Cannot be interrupted
  - Nothing can happen in-between

**Group Exercise:**

There are many atomic instructions that have been invented over the years. All of them have read-modify-write as a **single atomic operation**.

Brainstorm some ideas of what sorts of instructions you might devise.
Atomic Operations

- Common atomic operations:
  
  ```c
  test_and_set(int *lock, int val)
  compare_and_swap(int *lock,
                   int compareVal,
                   int newVal)
  
  linked load/conditional store
  ```
while (test_and_set(&lock, 1) == 1)  
sleep();

// when we get here, lock == 1 (and we  
// made the change). We own the lock.

doWork();
lock = 0;
while (!compare_and_swap(&lock, 0, 1))
    sleep();

// when we get here, lock == 1 (and we made the change). We own the lock.

doWork();

lock = 0;
Atomic Operations

while (true)
{
    int old = linked_load(&lock);
    if (old == 0 &&
        conditional_store(&lock,1))
    {
        break;
    }
    sleep();
}