Concurrency: Honors
Welcome to cs378h

Chris Rossbach
Outline for Today

• Questions?
• Administrivia
• Course Overview
• Course Details and Logistics
• Concurrency & Parallelism Basics

Acknowledgments: some materials in this lecture borrowed from:

• Emmett Witchel, who borrowed them from: Kathryn McKinley, Ron Rockhold, Tom Anderson, John Carter, Mike Dahlin, Jim Kurose, Hank Levy, Harrick Vin, Thomas Narten, and Emery Berger
• Mark Silberstein, who borrowed them from: Blaise Barney, Kunle Olukoton, Gupta
## Course Details

<table>
<thead>
<tr>
<th>Course Name:</th>
<th>CS378H – Concurrency: Honors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique Number:</td>
<td>50710</td>
</tr>
<tr>
<td>Lectures:</td>
<td>T-Th 9:30-11:00AM WAG 420</td>
</tr>
<tr>
<td>Instructor:</td>
<td>Chris Rossbach</td>
</tr>
<tr>
<td>TA:</td>
<td>Ariel Szkeley</td>
</tr>
</tbody>
</table>

*Please read the syllabus!*

...*More on this shortly...*
Why you should take this course

• Concurrency is super-cool, and super-important
• You’ll learn important concepts and background
• Have *fun* programming cool systems
  • GPUs! FGPAs!
  • Modern Programming languages: Go! Rust!
  • Interesting synchronization primitives (not just boring old locks)
  • Programming tools people use to program *super-computers* (ooh...)

Two perspectives:
• The “just eat your kale and quinoa” argument
• The “it’s going to be fun” argument
My first computer

Wires +
gobble-dy-gook
(sp?)

CPU

screen

Storage

Tape drive!
(Also good for playing heavy metal music)
My current computer

Too boring...
Another of my current computers

A lot has changed but... the common theme is...??

CPU

CPU

GPU

Image DSP

Crypto

...
Modern Technology Stack
Concurrency and Parallelism are Everywhere

Wait!
• What’s concurrency?
• What’s parallelism?
Concurrency and Parallelism are Everywhere
Concurrency and Parallelism are everywhere

How much parallel and concurrent programming have you learned so far?
• Concurrency/parallelism can’t be avoided anymore (want a job?)
• A program or two playing with locks and threads isn’t enough
• I’ve worked in industry a lot—I know

Course goal is to expose you to lots of ways of programming systems like these

...So "you should take this course because it’s good for you" (eat your #$(*& kale!)
**Goal**: Make Concurrency Your Close Friend

**Method**: Use Many Different Approaches to Concurrency

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locks and Shared Memory Synchronization</td>
<td>Prefix Sum with pthreads</td>
</tr>
</tbody>
</table>
| Language Support | Go lab: condition variables, channels, go routines  
Rust lab: 2PC |
| Parallel Architectures | GPU Programming Lab  
*FPGA Programming Lab* |
| HPC | Optional MPI lab |
| Distributed Computing / Big Data | Rust 2PC / MPI labs |
| Modern/Advanced Topics | • Specialized Runtimes / Programming Models  
• Auto-parallelization  
• Race Detection |
| Whatever Interests YOU | Project |
Logistics Reprise

<table>
<thead>
<tr>
<th>Course Name:</th>
<th>CS378H – Concurrency: Honors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique Number:</td>
<td>50710</td>
</tr>
<tr>
<td>Lectures:</td>
<td>TTh 9:30-11:00AM WAG 420</td>
</tr>
<tr>
<td>Instructor:</td>
<td>Chris Rossbach</td>
</tr>
<tr>
<td>TA:</td>
<td>Ariel Szkeley</td>
</tr>
</tbody>
</table>

Seriously, read the syllabus!
Also, start Lab 1!
Serial vs. Parallel Program

Key concerns:
- Programming model
- Execution Model
- Performance/Efficiency
- Exposing parallelism
Free lunch...

35 YEARS OF MICROPROCESSOR TREND DATA

Free lunch – is over 😞

35 YEARS OF MICROPROCESSOR TREND DATA

- Transistor number grows (Moore’s law)
- Sequential performance no longer improves
- Cores number grows

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>SISD</th>
<th>SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
Execution Models: Flynn’s Taxonomy

Normal Serial program

Uncommon architecture:
- Fault – tolerance
- Pipeline parallelism

Our main focus
SIMD

- Example: vector operations (e.g., Intel SSE/AVX, GPU)
MIMD

• Example: multi-core CPU
Problem Partitioning

• Decomposition: Domain v. Functional

• Domain Decomposition
  • SPMD
  • Input domain
  • Output Domain
  • Both

• Functional Decomposition
  • MPMD
  • Independent Tasks
  • Pipelining
Game of Life

- Given a 2D Grid:
- \( v_t(i,j) = F(v_{t-1}(\text{of all its neighbors})) \)
Domain decomposition

• Each CPU gets part of the input

How could we do a functional decomposition?

Issues?
• Accessing Data
  • Can we access v(i+1, j) from CPU 0
    • ...as in a “normal” serial program?
    • Shared memory? Distributed?
  • Time to access v(i+1,j) == Time to access v(i-1,j) ?
  • Scalability vs Latency
• Control
  • Can we assign one vertex per CPU?
  • Can we assign one vertex per process/logical task?
  • Task Management Overhead
• Load Balance
• Correctness
  • order of reads and writes is non-deterministic
  • synchronization is required to enforce the order
  • locks, semaphores, barriers, conditionals....
Load Balancing

• Slowest task determines performance
Granularity

\[ G = \frac{\text{Computation}}{\text{Communication}} \]

- Fine-grain parallelism
  - G is small
  - Good load balancing
  - Potentially high overhead
  - Hard to get correct

- Coarse-grain parallelism
  - G is large
  - Load balancing is tough
  - Low overhead
  - Easier to get correct
Performance: Amdahl’s law

- Speedup is bounded by serial component
- Speedup

\[
\text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}}
\]

\[
\text{Speedup}(\#\text{CPUs}) = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{1}{A + \frac{1}{\#\text{CPUs}}(1 - A)}
\]
Amdahl’s law

What makes something “serial” vs. parallelizable?
Amdahl’s law

End to end time: \( \frac{X}{2} + \frac{X}{4} = \frac{3}{4}X \) seconds

What is the “speedup” in this case?

\[
\text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)} = \frac{1}{\frac{.5}{2 \text{ cpus}} + (1-.5)} = 1.333
\]
Speedup exercise

What is the “speedup” in this case?

\[
Speedup = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)} = \frac{1}{\frac{.75}{8} + (1-.75)} = 2.91x
\]
Amdahl Action Zone

**50% PARALLEL**

![Graph showing the Amdahl Action Zone with the x-axis representing the number of CPUs and the y-axis representing speedup. The line indicates a speedup of 2 when 50% of the work is parallel.]
Amdahl Action Zone

The graph shows the speedup for different numbers of CPUs, with two lines representing 50% and 75% efficiency. As the number of CPUs increases, the speedup initially increases but levels off at higher numbers of CPUs, indicating the diminishing returns of adding more processors.
Amdahl Action Zone

SPEEDUP

NUMBER OF CPUS

50%  75%  90%  95%  99%
Strong Scaling vs Weak Scaling

• Amdahl vs. Gustafson

• $N = \#CPUs$, $S = \text{serial portion} = 1 - A$

• Amdahl's law: $\text{Speedup}(N) = \frac{1}{\frac{A}{N} + S}$
  - **Strong scaling**: $\text{Speedup}(N)$ calculated given total amount of work is fixed
  - Solve same problems faster when problem size is fixed and #CPU grows
  - Assuming parallel portion is fixed, speedup soon seizes to increase

• Gustafson's law: $\text{Speedup}(N) = N + (N-1) \cdot S$
  - **Weak scaling**: $\text{Speedup}(N)$ calculated given amount of work per CPU is fixed
  - Keep the amount of work per CPU when adding more CPUs to keep the granularity fixed
  - Problem size grows: solve larger problems
  - **Consequence**: speedup upper bound much higher
Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: *best* serial algorithm
- Example:
  - Efficient **bubble sort** takes:
    - Parallel 40s
    - Serial 150s
    - \( \text{Speedup} = \frac{150}{40} = 3.75 \)
  - NO!
    - Serial quicksort runs in 30s
    - \( \Rightarrow \text{Speedup} = 0.75 \)
If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

Thread 1

```c
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Thread 2

```c
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Answer:
A. 0
B. 1
C. 2
D. More than 2
Schedules/Interleavings

Model of concurrent execution

• Interleave statements from each thread into a single thread
• If **any** interleaving yields incorrect results, synchronization is needed

```plaintext
Thread 1
tmp1 = X;
tmp1 = tmp1 + 1;
X = tmp1;

Thread 2
tmp1 = X;
tmp2 = X;
tmp2 = tmp2 + 1;
tmp1 = tmp1 + 1;
X = tmp1;
X = tmp2;
```

If X==0 initially, X == 1 at the end. **WRONG** result!
Locks fix this with Mutual Exclusion

```
void increment() {
    lock.acquire();
    int temp = X;
    temp = temp + 1;
    X = temp;
    lock.release();
}
```

Mutual exclusion ensures only safe interleavings

- But it limits concurrency, and hence scalability/performance

Is mutual exclusion a good abstraction?
Why Locks are Hard

- Coarse-grain locks
  - Simple to develop
  - Easy to avoid deadlock
  - Few data races
  - Limited concurrency

- Fine-grain locks
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
    - Not composable
  - Potential data races
    - Which lock to lock?

// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}

DEADLOCK!

Thread 0
move(a, b, key1);
move(b, a, key2);

Thread 1
The correctness conditions

• Safety
  • Only one thread in the critical region

• Liveness
  • Some thread that enters the entry section eventually enters the critical region
  • Even if other thread takes forever in non-critical region

• Bounded waiting
  • A thread that enters the entry section enters the critical section within some bounded number of operations.

• Failure atomicity
  • It is OK for a thread to die in the critical region
  • Many techniques do not provide failure atomicity

```c
while(1) { 
  Entry section
  Critical section
  Exit section
  Non-critical section
}
```
Read-Modify-Write (RMW)

- Implement locks using read-modify-write instructions
  - As an atomic and isolated action
    1. read a memory location into a register, **AND**
    2. write a new value to the location
  - Implementing RMW is tricky in multi-processors
    - Requires cache coherence hardware. Caches snoop the memory bus.

- Examples:
  - Test&set instructions (most architectures)
    - Reads a value from memory
    - Write “1” back to memory location
  - Compare & swap (68000)
    - Test the value against some constant
    - If the test returns true, set value in memory to different value
    - Report the result of the test in a flag
    - if [addr] == r1 then [addr] = r2;
  - Exchange, locked increment, locked decrement (x86)
  - Load linked/store conditional (PowerPC, Alpha, MIPS)
Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;

Lock::Acquire() {
    while (test&set(lock) == 1) //spin
}

Lock::Release() {
    *lock = 0;
}
```

What is the problem with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work
Test & Set with Memory Hierarchies

Initially, lock already held by some other CPU—A, B busy-waiting
What happens to lock variable’s cache line when different cpu’s contend?

Load can stall

CPU A
while(test&set(lock));
// in critical region

CPU B
while(test&set(lock));

L1
lock: 1
...

L2
lock: 1
...

Main Memory

0xF0 lock: 1
0xF4 ...

L1

L2

- With bus-locking, lock prefix blocks *everyone*
- With CAS, LL-SC, cache line cache line “ping pongs” amongst contenders
- More on this next time...
Programming and Machines: a mental model

```
struct machine_state{
    uint64 pc;
    uint64 Registers[16];
    uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
    ...
} machine;
while(1) {
    fetch_instruction(machine.pc);
    decode_instruction(machine.pc);
    execute_instruction(machine.pc);
}
void execute_instruction(i) {
    switch(opcode) {
    case add_rr:
        machine.Registers[i.dst] += machine.Registers[i.src];
        break;
    }
```
Parallel Machines: a mental model

```c
struct machine_state{
  uint64 pc;
  uint64 Registers[16];
  uint64 cr[6]; // control registers cr0-cr4 and EFER on AMD
...
} machine;
while(1) {
  fetch_instruction(machine.pc);
  decode_instruction(machine.pc);
  execute_instruction(machine.pc);
}
void execute_instruction(i) {
  switch(opcode) {
  case add_rr:
    machine.Registers[i.dst] += machine.Registers[i.src];
    break;
  }
}
```
Processes and Threads

• Abstractions
• Containers
• State
  • Where is shared state?
  • How is it accessed?
  • Is it mutable?
Processes & Virtual Memory

• Virtual Memory: Goals...what are they again?
• Abstraction: contiguous, isolated memory
  • Remember overlays?
• Prevent illegal operations
  • Access to others/OS memory
  • Fail fast (e.g. segv on *(NULL))
  • Prevent exploits that try to execute program data
• Sharing mechanism/IPC substrate
Process Address Space

Access requires kernel mode

Access possible in user mode

0

3 GB

1 GB

C0000000

C0400000

FFFFFFFFF

P1

kcode

kdata

kbss

kheap

udata (1)

ucode (1)

user (1)

user (2)

user (2)

user (1)

user (1)

user (1)

kernel

kernel

kernel

kernel

used

free
Processes
The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant
# Implementation of Processes

## Fields of a process table entry

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Threads
The Thread Model (1)

(a) Three processes each with one thread
(b) One process with three threads
The Thread Model

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
- Items private to each thread
The Thread Model

Each thread has its own stack
Using threads

Ex. How might we use threads in a word processor program?
Thread Usage

A multithreaded Web server

(a) Dispatcher thread
(b) Worker thread
# Thread Usage

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls, interrupts</td>
</tr>
</tbody>
</table>

Three ways to construct a server
Implementing Threads in User Space

A user-level threads package
Implementing Threads in the Kernel

A threads package managed by the kernel
Pthreads

• POSIX standard thread model,
• Specifies the API and call semantics.
• Popular – most thread libraries are Pthreads-compatible
Preliminaries

• Include `pthread.h` in the main file

• Compile program with `-lpthread`
  • `gcc -o test test.c -lpthread`
  • may not report compilation errors otherwise but calls will fail

• Good idea to check return values on common functions
Thread creation

• Types: `pthread_t` – type of a thread

• Some calls:

```c
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void * (*start_routine)(void *),
                  void *arg);

int pthread_join(pthread_t thread, void **status);
int pthread_detach();
void pthread_exit();
```

• No explicit parent/child model, except main thread holds process info
• Call `pthread_exit` in main, don’t just fall through;
• Most likely you wouldn’t need `pthread_join`
  • `status` = exit value returned by joinable thread
• Detached threads are those which cannot be joined (can also set this at creation)
Creating multiple threads

```c
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
    printf("Hello Thread\n");
}

int main() {
    pthread_t tid[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], NULL, hello, NULL);

    for (int i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
```
Can you find the bug here?

What is printed for myNum?

```c
void *threadFunc(void *pArg) {
    int* p = (int*)pArg;
    int myNum = *p;
    printf( "Thread number %d\n", myNum);
}
...
// from main():
for (int i = 0; i < numThreads; i++) {
    pthread_create(&tid[i], NULL, threadFunc, &i);
}
Pthread Mutexes

• **Type:** `pthread_mutex_t`

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

• **Attributes:** for shared mutexes/condition vars among processes, for priority inheritance, etc.
  • use defaults

• **Important:** Mutex scope must be visible to all threads!
Spinlock vs Mutex
Lab #1

• Basic synchronization
• http://www.cs.utexas.edu/~rossbach/cs378/lab/lab0.html

• *Start early***!!
Questions?