Threads Introduction

Notebook Work

- What is a core?
- Where is a core?
- What is cache memory?
- Does each core have its own cache memory?
- How many cores on your laptop?
- What is the relationship between an OS and cores?

Parallel Programming

- Clusters of computers, multicore CPUS, AI style - Google TPU, NVIDIA - low level matrix multiply for computing the weights on neural nets
- Parallel Programming Languages - Haskell
- Using threads and a thread API
- OS provides synchronization primitives
- OS uses hardware support - Atomic instructions such as Test and Set
- Moore's Law - chips will double in capacity every year or so
- Dennard's Scaling - as chips double, they will consume the same power
- Ahmdal's Law - Thread speed up of parallel computing is limited by the amount of time that is sequential.
    - From Patterson and Hennessy Turing Lecture
    - Assume a parallel algorithm with 64 cores.
    - Assume that 1% of the time spent on the algorithm is sequential.
    - The speedup for a 64 core setup is about 35 - not 64
    - The power is proportional to the 64 cores so approximately 45% of the energy is wasted
    - Real algorithms probably have more than 1% of their time in sequential

Thread vs. Process

- Processes - separate entities. Separate registers, address space, code, static data, heap, and stack
- Threads - shared address space, code, static data, heap, open file descriptors, current working directory
- Each thread has its own
  - Thread ID (TID), registers, PC, SP, and stack
  - Set of registers, including Program counter and Stack pointer
  - Thread context switches
- Thread API - Variety of thread systems exist
  - POSIX Pthreads
  - Common thread operations - Create, Exit, Join (like wait() for processes), and synchronization API (locks, semaphores, condition variables)

Multithreaded Applications

- Web Browsers - handles each tab with a thread
- Web Servers - handles each request with a new thread.
- Computer Games - Objects like cars, humans, birds which are implemented as separate threads.
- Editors - When you are typing in an editor, spell-checking, formatting of text and saving the text are done concurrently by multiple threads.
- IDE - IDEs provide suggestions on the completion of a command which is a separate thread.
- Missile Launch - threads process computation/movement of data to/from Guidance computer, other computers, and command/control computer

Posix Thread Code - Brain Image

```
#include <stdio.h>
#include "mythreads.h"

int balance = 0; int loops = 0;
void* worker(void* arg) {
    for (int i = 0; i < loops; i++)
        balance++; // unprotected access
    return NULL;
}
int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    for (int i = 0; i < loops; i++)
        balance++; // unprotected access
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

```
$ ./a.out
```

How many processes?
How many threads?
Are they running concurrently?

Threads Incrementing a Shared Static Variable

```c
#include <stdio.h>
#include <pthread.h>
int balance = 0; int loops = 0;

void* worker(void* arg) {
    for (int i = 0; i < loops; i++)
        balance++; // ← Race Condition
    return NULL;
}
int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Two workers running

main is running

```
lla a5,balance
lw  a5,0(a5)
addiw   a5,a5,1
sext.w  a4,a5
lla a5,balance
sw  a4,0(a5)
```

Critical Region

RISC-V Assembly for balance++

Scenario 1 - Two threads executing balance++

Initial condition: balance is 20

Thread 1

```
lla a5,balance
lw  a5,0(a5)
addiw   a5,a5,1
sext.w  a4,a5
lla a5,balance
sw  a4,0(a5)
```

Thread 2

```
lla a5,balance
lw  a5,0(a5)
addiw   a5,a5,1
sext.w  a4,a5
lla a5,balance
sw  a4,0(a5)
```

Two threads inc balance.
Result is balance == 22

Thread 1 completes entire box.
balance is now 21
Context switch to Thread 2

Thread 2 completes entire box.
balance is now 22

Scenario 2 - Two threads executing balance++

Initial condition: balance is 20

Thread 1

```
lla a5,balance
lw  a5,0(a5)
addiw   a5,a5,1
sext.w  a4,a5
lla a5,balance
sw  a4,0(a5)
```

Thread 2

```
lla a5,balance
lw  a5,0(a5)
addiw   a5,a5,1
sext.w  a4,a5
lla a5,balance
sw  a4,0(a5)
```

Two threads inc balance.
Result is balance == 21

Context switch at red arrow
a5 has 20
balance is still 20

Thread 2 completes entire box.
balance is now 21
Context switch to Thread 1

Thread 1 finishes box
balance is now 21

Race Condition and Critical Region

- Race Condition and Critical Region - previous slides demonstrate a race condition. A race condition occurs in a critical region.
    - Thread code that performs concurrent operations and the outcome is dependent upon the order.
- Lock, Mutex, Mutex Lock - used to fix Race Condition

Concurrency and Nondeterminism

- Nondeterministic - different results with same inputs
- Bug depends on CPU scheduling
- For this example, we must create a way such that the assembly instructions cannot be interrupted in the middle

Allow one thread at a time in critical region

```
lla a5,balance
lw  a5,0(a5)
addiw   a5,a5,1
sext.w  a4,a5
lla a5,balance
sw  a4,0(a5)
```

← critical region

pThread Code

```
pthread_mutex_t m;
volatile int balance = 0;
int loops = 2000;
void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&m);
        balance++;
        pthread_mutex_unlock(&m);
    }
    pthread_exit(NULL);
}
```

Only one thread at a time

Two workers in critical region

```
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("Initial value : %d\n", balance);
    pthread_mutex_init(&m, NULL);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", balance);
    return 0;
}
```

Synchronization of Threads

- Use hardware support to create synchronization primitives for threads
- Synchronization Primitives
  - Locks, Condition Variables, Semaphores
- Hardware Support
  - Test and Set, Loads, Stores, Disable Interrupts

Locks - Provide Mutual Exclusion (mutex)

- Allocate and Initialize
  - `Pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;`
- Acquire - acquire exclusive access to lock
  - Wait if lock is not available (some other process in critical section)
  - Spin or block (relinquish CPU) while waiting
  - `Pthread_mutex_lock(&mylock);`
- Release -exclusive access to lock; let another process enter critical section
  - `Pthread_mutex_unlock(&mylock);`

Deadlock

- Deadlock – two threads T1, T2 and two locks L1, L2, where T1 has L1 and attempts to get L2 at the same time T2 has L2 and attempts to get L1. T1 cannot get L2 because T2 has it. T2 cannot get L1 because T1 has it.
  - Bank transactions - transfer(a, b) at the same time as transfer(b, a)

```
Thread 1: withdraw(amount)     Thread 2: withdraw(amount)
int b = getBalance();

                               int b = getBalance();
                               if (amount > b)
                                   bad balance;
                               setBalance(b-amount);
if (amount > b)
    bad balance;
setBalance(b - amount);
```

```
Thread 1: transfer(x,y)       Thread 2: transferTo(y,x)
acquire lock from x
withdraw 1 from x

                              acquire lock from y
                              withdraw 1 from y
                              block on lock for x
block on lock for y
```