Threads & Concurrency
Why Use Threads?

Utilize multiple CPU’s concurrently

Low cost communication via shared memory

Overlap computation and blocking on a single CPU
- Blocking due to I/O
- Computation and communication

Handle asynchronous events
Typical Thread Usage

A word processor with three threads
Processes vs Threads

GET / HTTP/1.0

HTTPD

disk
Processes vs Threads

Why is this not a good web server design?
Processes vs Threads

GET / HTTP/1.0
Processes vs Threads

GET / HTTP/1.0

HTTPD

GET / HTTP/1.0

disk
Processes vs Threads
Common Thread Strategies

Manager/worker
- Manager thread handles I/O
- Magaer assigns work to worker threads
- Worker threads created dynamically
- ... or allocated from a thread-pool

Pipeline
- Each thread handles a different stage of an assembly line
- Threads hand work off to each other in a producer-consumer relationship
Pthreads: A Typical Thread API

Pthreads: POSIX standard threads
First thread exists in main(), creates the others

`pthread_create (thread, attr, start_routine, arg)`
- Returns new thread ID in “thread”
- Executes routine specified by “start_routine” with argument specified by “arg”
- Exits on return from routine or when told explicitly
Pthreads (continued)

`pthread_exit (status)`
- Terminates the thread and returns “status” to any joining thread

`pthread_join (threadid,status)`
- Blocks the calling thread until thread specified by “threadid” terminates
- Return status from `pthread_exit` is passed in “status”
- One way of synchronizing between threads

`pthread_yield ()`
- Thread gives up the CPU and enters the run queue
Using Create, Join and Exit
An Example Pthreads Program

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    printf("%d: Hello World!\n", threadid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

For more examples see: http://www.llnl.gov/computing/tutorials/pthreads

Program Output

Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
3: Hello World!
Creating thread 4
4: Hello World!
Pros & Cons of Threads

Pros:
- Overlap I/O with computation!
- Cheaper context switches
- Better mapping to multiprocessors

Cons:
- Potential thread interactions
- Complexity of debugging
- Complexity of multi-threaded programming
- Backwards compatibility with existing code
Concurrency
Sequential Programming

Sequential programming with processes

- Private memory
  - a program
  - data
  - stack
  - heap

- CPU context
  - program counter
  - stack pointer
  - registers
Sequential Programming Example

```c
int i = 0
i = i + 1
print i
```

What output do you expect?
Why?
Concurrent Programming

Concurrent programming with threads
- Shared memory
  - a program
  - data
  - heap
- Private stack for each thread
- Private CPU context for each thread
  - program counter
  - stack pointer
  - registers
Concurrent Threads Example

int i = 0

Thread 1:  i = i + 1
        print i

What output do you expect with 1 thread? Why?
Concurrent Threads Example

```java
int i = 0

Thread 1:   i = i + 1   Thread 2:   i = i + 1
            print i        print i
```

What output do you expect with 2 threads? Why?
Race Conditions

How is $i = i + 1$ implemented?

- load $i$ to register
- increment register
- store register value to $i$

Registers are part of each thread’s *private* CPU context
Race Conditions

Thread 1

load i to regn
inc regn
store regn to i

Thread 2

load i to regn
inc regn
store regn to i
Critical Sections

What is dangerous in the previous example?
How should we reason about this kind of code?
What property did we have in sequential programming, which we lost in concurrent programming?
Why was that property important?
Memory Invariance

Sequential programs have the property that memory values do not change unless the control flow changes them.

Hence, we can reason about the effects of a control flow.

How can we regain this memory invariance property in concurrent programs?
Mutual Exclusion

Diagram showing the sequence of events for processes A and B:

- **Process A**
  - Enters critical region at time $T_1$
  - Leaves critical region at time $T_2$

- **Process B**
  - Attempts to enter critical region at time $T_2$
  - Enters critical region at time $T_3$
  - Leaves critical region at time $T_4$

Note: During $T_2$, B is blocked from entering the critical region.
Mutual Exclusion

But how can we implement it?