Review

- Finish scheduling
- Process Manipulation

Outline

- The Joys and Pains of Threads and Multithreading

Threads & Multithreading

The Case for Threads

Consider the following program fragment:

```
main()
    read_data()             // read input data
    for(all data)
        compute();          // do some computation
        write_data();       // output results to disk
    endfor
```

What is wasteful about a program of this sort?

The Case for Threads

Consider the following code fragment

```
for(k = 0; k < n; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
```

- Is there a missed opportunity here? On a Uniprocessor? Multi-processor?

The Case for Threads

Consider a Web server

    get network message from client
    get URL data from disk
    compose response
    send response

What happens to you if you use this Web server?

Lightweight Threads

- A thread is an abstraction of an entity that executes instructions within a processor:
  - it has its own set of CPU registers that are loaded when the thread runs
  - has its own stack
- Threads are lightweight:
  - there is no thread-specific heap or data segment (unlike process)
  - therefore, context switching is much cheaper than for a process
- Examples:
  - OS-supported: Sun's LWP, POSIX's threads
  - Language-supported: Modula-3, Java

Programmer's View

```
main()
    some code
    tid = CreateThread(fn1, arg0, arg1, …);
    some more code

fn1(in arg0, int arg1, …)
    some code
```

At the point CreateThread is called, execution continues in parent thread in main function, and execution starts at fn1 in the child thread, *both in parallel*

How Can it Help?

main()	main()
read_data()	read_data()
for(all data)	for(all data)
compute();	compute();
write_data();	CreateThread(write_data());
endfor	endfor

What did we gain?

How Can it Help?

Consider the following code fragment

```
for(k = 0; k < n; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
```

```
CreateThread(fn, 0, n/2);
CreateThread(fn, n/2, n);
fn(l, m)
    for(k = l; k < m; k++)
        a[k] = b[k] * c[k] + d[k] * e[k];
```

- What did we gain?

How Can it Help?

Consider a Web server

create a number of threads, and for each thread do

- get network message from client
- get URL data from disk
- compose response
- send response

What did we gain?

Threads vs. Processes

- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- *There can be more than one thread in a process, the first thread calls main & has the process's stack*
- Inexpensive creation
- Inexpensive context switching
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- There must be at least one thread in a process
- *Threads within a process share code/data/heap, share I/O, but each has its own stack & registers*
- Expensive creation
- Expensive context switching
- If a process dies, its resources are reclaimed & all threads die

How Does it Work?

```
int a = 1, b = 2;
main() {
    CreateThread(fn1, 4);
    CreateThread(fn2, 5);
}
fn1(int arg1) {
    if(a) b++;
}
fn2(int arg1) {
    a = arg1;
}
```

What are the value of a & b at the end of execution?
Lesson 1: Sharing

How Does it Work?

```
int a = 1, b = 2;
main() {
    CreateThread(fn1, 4);
    CreateThread(fn2, 5);
}
fn1(int arg1) {
    if(a) b++;
}
fn2(int arg1) {
    a = 0;
}
```

What are the values of a & b at the end of execution?
Lesson 2: Interleaving

How Does it Work?

```
int a = 1, b = 2;
main() {
    CreateThread(fn, 4);
    CreateThread(fn, 5);
}
fn(int arg1) {
    int v = a + arg1;
    static x;
    if(v) {
        x= b++;
        v = 0;
    }
}
```

What are the values of a, b, x & v at the end of execution?
Lesson 3: Privacy & sharing

Thread Hazards

```
int a = 1, b = 2;
main() {
    CreateThread(fn, 4);
    CreateThread(fn, 5);
    CreateThread(fn, 6);
    do some computation
    return 1;
}
fn() {
   a = b;
}
```

What happens here?
Lesson 4: Falling off the cliff

Thread Hazards

```
int a = 1, b = 2, w = 1;
main() {
    CreateThread(fn, 4);
    char * p = bogus value;
    *p = 4;
    while(w);
}
fn() {
  int v = a + b;
  w = 0;
}
```

What happens here?
Lesson 5: Protection

Thread Hazards

```
int a = 1, b = 2, w = 1;
main() {
    CreateThread(fn, 4);
    while(w)  ;
}
fn() {
  int v = a + b;
  // should have set w = 0 here
}
```

What happens here?
Lesson 6: Synchronization

Thread Hazards

```
int a = 1, b = 2, w = 1;
main() {
    CreateThread(fn, 4);
    while(w)  ;
}
fn() {
  int v = a + b;
  // should have set w = 0 here
}
```

What happens here?
Lesson 6: Synchronization

Thread Hazards

```
int a = 1, b = 2, w = 2;
main() {
    CreateThread(fn, 4);
    CreateThread(fn, 4);
    while(w)  ;
}
fn() {
  int v = a + b;
  w--;
}
```

What happens here?
Lesson 7: Concurrency

Other Hazards?

- If there is a signal, who should get it?
  - One?
  - All?
  - Some designated thread?
- If there is an exception:
  - Kill only offending thread?
  - Kill process?
- Hidden concurrency problems:
  - Sharing through files