

From Processes to Threads

1

Processes and Threads

- ◆ Process abstraction combines two concepts
 - Concurrency
 - ◊ Each process is a sequential execution stream of instructions
 - Protection
 - ◊ Each process defines an address space
 - ◊ Address space identifies all addresses that can be touched by the program
- ◆ Threads
 - Key idea: separate the concepts of concurrency from protection
 - A thread represents a sequential execution stream of instructions
 - A process defines the address space that may be shared by multiple threads

2

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 Uni-processor?
On a Multi-processor?

3

The Case for Threads

Consider a Web server

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

How well does this web server perform?

4

Introducing Threads

- ◆ A thread represents an abstract entity that executes a sequence of instructions
 - > It has its own set of CPU registers
 - > It 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

5

Programmer's View

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

```
fn1(int 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*

6

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];
```
- ◆ Rewrite this code fragment as:

```
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?

7

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?

8

Threads vs. Processes

Threads

- ◆ 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

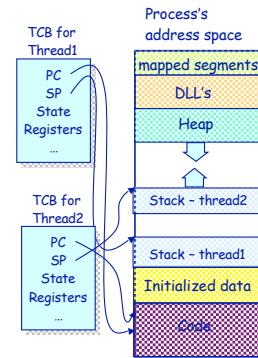
Processes

- ◆ 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

9

Implementing Threads

- ◆ Processes define an address space; threads share the address space
- ◆ Process Control Block (PCB) contains process-specific information
 - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- ◆ Thread Control Block (TCB) contain thread-specific information
 - Stack pointer, PC, thread state (running, ...), register



10

Implementing Threads (Cont'd.)

```

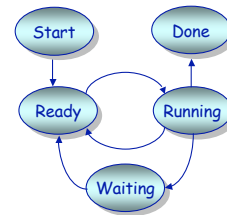
CreateThread(pointer_to_procedure, arg0, ...) {
    // allocate a new TCB and stack
    TCB tcb = new TCB();
    Stack stack = new Stack();
    // initialize TCB and stack with initial register values and address
    // of first instruction
    tcb.pc = stub;
    tcb.stack = stack;
    tcb.arg0reg = pointer_to_procedure;
    tcb.arg1reg = arg0;
    ...
    // Tell the dispatcher about the newly created thread
    ReadyQ.add(tcb);
}

Stub(proc, arg0, arg1, ...) {
    (*proc)(arg0, arg1, ...);
    DeleteCurrentThread();
}
    
```

11

Threads' Life Cycle

- ◆ Threads (just like processes) go through a sequence of *start*, *ready*, *running*, *waiting*, and *done* states



12

Dispatching Threads

◆ Basic operation

```
Thread is running
Switch to kernel
Save thread state (TCB)
Choose new thread to run
Load state of the chosen thread (from TCB)
...
```

◆ Switch to kernel initiated by

- System call (e.g., IO, yield CPU, etc.)
- Exception
- Interrupt (e.g., timer interrupt)

The threads that the kernel is aware of and schedules are called **kernel threads**.

13

User-level Threads

◆ Motivation

- Threads are a useful programming abstraction
- Implement thread creation/manipulation using procedure calls to a user-level library rather than system calls

◆ User-level threads

- User-level library implementations for
 - ◆ CreateThread()
 - ◆ DestroyThread()
 - ◆ Yield()
 - ◆ ...
- User-level library performs the same set of actions as the corresponding system calls
- Main difference: thread management is under the control of user-level library

14

Kernel Threads vs. User-level Threads

◆ Kernel threads:

- A kernel thread, aka *lightweight process*, is a thread that the OS knows about
- Switching between kernel threads of the same process is inexpensive
 - ◆ The values of registers, PC, and stack pointers are changed
 - ◆ Memory management information does not change
- Kernel uses process scheduling algorithms to manage threads

◆ User-level threads:

- OS does not know about user-level threads
- OS is only aware of the process containing threads
- OS schedules processes, not threads
- Programmer uses a threads library to manage threads (create, delete, synchronize and schedule)

15

User-level Threads (Cont'd.)

◆ Benefits:

- No context for switching between threads of a process
- Thread scheduling is more flexible
 - ◆ Can use application-specific scheduling policy *
 - ◆ Each process can use a different scheduling algorithm
 - ◆ Threads voluntarily give up CPU

◆ Drawbacks:

- OS is unaware of the existence of user-level threads
 - ◆ Poor scheduling decisions
 - ◆ If a user-level thread waits for I/O - entire process waits
- OS schedules processes independent of number of threads within a process

* How will you implement a time-slice based application-specific scheduling policy?

16

Sharing among threads is great ...

```
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?

17

... But it can lead to problems!!

```
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?

18

Some More Examples

- ◆ What are the possible values of x in these cases?

Thread1: x = 1; Thread2: x = 2;

Initially y = 10;

Thread1: x = y + 1; Thread2: y = y * 2;

Initially x = 0;

Thread1: x = x + 1; Thread2: x = x + 2;

19

This is because ...

- ◆ Order of thread execution is non-deterministic
 - Multiprocessing
 - ◆ A system may contain multiple processors → cooperating threads/processes can execute simultaneously
 - Multi-programming
 - ◆ Thread/process execution can be interleaved because of time-slicing
- ◆ Operations are often not "atomic"
 - Example: x = x + 1 is not atomic!
- ◆ Goal:
 - Ensure that your concurrent program works under ALL possible interleaving
- ◆ Challenge
 - Enumerating all cases is not possible!
 - Need to define synchronization constructs and programming style for developing concurrent programs

20