From Processes to Threads

Processes and Threads

- The process abstraction combines two concepts
  - Concurrency: each process is a sequential execution stream of instructions
  - Protection: Each process defines an address space that identifies what can be touched by the program
- Threads
  - Key idea: decouple 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

From Processes to Threads

The Case for Threads

- Consider the following code segment:

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

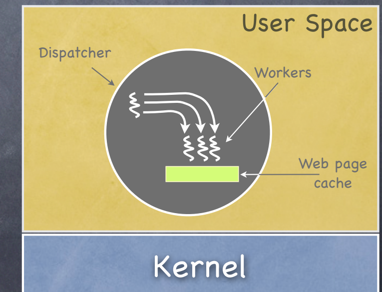
- Is there a missed opportunity here?

The Case for Threads

- Consider a Web server:
  - get network message from client
  - get URL data from disk
  - compose response
  - send response

The Case for Threads

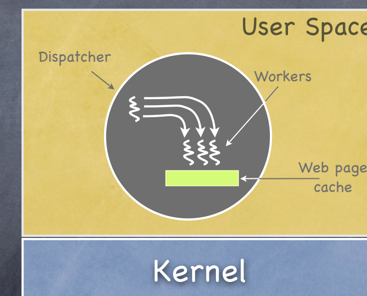
- Consider a Web server:
  - get network message from client
  - get URL data from disk
  - compose response
  - send response



A Third Way

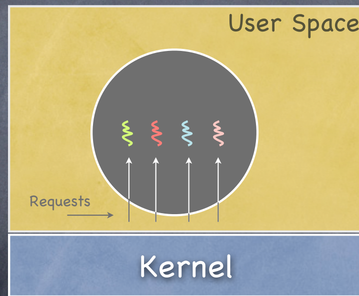
- Run the server as a single finite state machine
  - a large event queue
  - a single thread
  - no blocking system calls: on I/O, save current state in a table and fetch next event
    - if new request, then start on it
    - if I/O completion, then fetch state from table and continue
- Harder to program against

Multithreaded Processing Paradigms



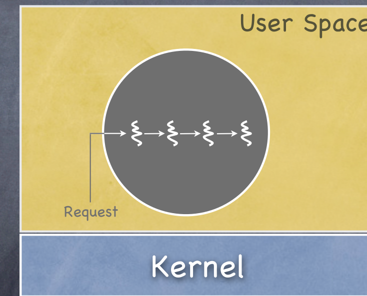
Dispatcher/Workers

Multithreaded Processing Paradigms



Specialists

Multithreaded Processing Paradigms



Pipelining

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
  - Much faster context switching!
- Examples:
  - OS-supported: Sun's LWP, POSIX's threads
  - Language-supported: Modula-3, Java

Per Process	Per Thread
Address space	Program counter
Global variables	Registers
Open Files	Stack
Child processes	State
Pending alarms	
Signals and their handlers	
Accounting info	

Programmer's View

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

fn1(int arg0, int arg1, ...)
    some code
```

After CreateThread is called, execution in parent thread continues in main function, and, in parallel, execution in child thread starts at fn1

How Can it Help?

- Consider again 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];
```

Threads vs. Processes

Threads	Processes
No data segment or heap	Have data/code/heap and other segments
Multiple can coexist in a process	Include at least one thread
Share code, data, heap and I/O	Have own address space, isolated from other processes'
Have own stack and registers, but no isolation from other threads in the same process	Expensive to create
Inexpensive to create	Expensive context switching
Inexpensive context switching	

Implementing Threads

```
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();
}
```

Threads Life Cycle

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

User-level Threads

- No need of OS support!
- Threads execute on top of a run time system
  - User-level library implementations for
    - CreateThread(), DestroyThread(), Yield(), ...
- If thread may be suspended (sys call, synch operation, etc)
  - call runtime system
  - check if thread must be suspended; if so
    - save current context
    - find unblocked thread
    - load that thread's context

Pros and Cons of User-level Threads

Pros

- Very low context switch cost
  - ~2 orders of magnitude
- Flexible scheduling
  - Application specific
  - Process specific
  - Threads voluntarily give up CPU–easy to reason about!

Cons

- OS is unaware of user-level threads
  - Blocking sys calls invoked by a thread block entire process
  - page fault: same thing!
  - Round-robin, anyone?
  - OS schedules processes independent of number of threads within a process

What to do???

- Change all blocking calls to non-blocking calls
- Rewrite sys call library to <u>detect</u>, if possible, whether a call will block
  - Use SELECT before READ
- Have the runtime system request periodic interrupts (say, every second)
- Implement threads in the kernel!

Kernel-level Threads

- All calls to run time system are now sys calls!
  - higher cost in managing thread operations
    - 1/10 of process; 10x of user-level threads
  - the "cost of generality": a service in the kernel should fit all applications!

Scheduler Activations - I

- Goal: combine functionality and ease of use of kernel threads with performance of user-level threads (ult)
- Abstraction: virtual multiprocessor for each ult system
  - kernel allocates processors to address spaces
  - each address space has complete control on which threads run on allocated processor
  - Kernel notifies thread package on:
    - change in # of processors
    - thread blocking or waking up on I/O events
  - Thread package notifies kernel on:
    - adding or returning processors–other thread management operations are performed in user space

Scheduler Activations - II

- Scheduler activations let kernel processor allocator communicate with user-level threads
  - data structure similar to a kernel thread
  - multiple execution stacks (one mapped to kernel, one for each thread in user space)
  - kernel keeps activation control block
  - when kernel knows that a the state of thread has changed, it activates ult package with an upcall
- On interrupt:
  - switch to kernel
  - if interrupt does not apply to virtual processor, continue
  - otherwise, suspend interrupted thread and give control to run time system of user level thread package

Any negatives?

What happens if interrupt occurs when a thread holds a lock?

Concurrency 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 and b at the end of execution?

...but can be problematic

```
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?

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;

This is because ...

- Order of process/thread execution is non-deterministic
  - A system may contain multiple processors and cooperating threads/processes can execute simultaneously
  - Thread/process execution can be interleaved because of time-slicing
- Operations are often not atomic
  - An atomic operation is one that executes to completion without any interruption or failure---it is "all or nothing"
  - x := x+1 is not atomic
    - read x from memory into a register
    - increment register
    - store register back into memory
- Goal: Ensure correctness under ALL possible interleaving

We have a problem...

- Enumerating all cases is impractical
- We need to
  - define constructs to help with synchronization and coordination
  - develop a programming style that eases the construction of concurrent programs
  - more fundamentally, we need to know what we are talking about we we mention "synchronization" or "coordination"...