Threads

Intro

- So far, we've seen a process as a combination of a THREAD, ADDRESS SPACE, and OPERATING SYSTEM STATE
  - every context switch involves transferring control to a new address space
- Modern systems now allow us to separate the concept of a thread from an address space
  - context switch into the same address space

Different kinds of OS

- One thread, one address space
  - real time systems, embedded systems, DOS
- Many address spaces, one thread per address space
  - old style time sharing systems (eg, UNIX, MULTICS)
- Many address spaces, many threads per adderss space
  - new systems (NT, most versions of UNIX)

What does it mean?

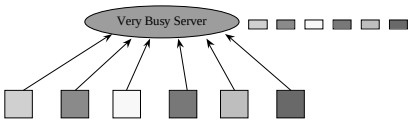
- One address space can have many threads in it.
- All threads share the same memory (unlike fork, which creates a new address space)
- All threads share the same OS resources
  - one thread can open a file; another can read it
- Leads to very rich, but complicated programs

Why Do It?

- Allows programs to overlap their own computation with I/O
- Simplifies the programming task when dealing with many different kinds of activities within the program
- Dierct map onto parallel processors

Problem Statement

- A node in a distributed system may be responsible for handling a large number of client machines
- How do we represent and manage the "contexts" of any and all outstanding service requests?



Threads

- A thread is an independent stream of control in a running program.
  - state of a computation "in action"
- One program can have many threads.
- Each is responsible for handling different activities.
- Threads are really a programming abstraction
  - under the covers, only one CPU

The Number Server

- We want to sell Unique Integers
- We expect a lot of customers.
- We will get sued if we sell the same integer twice.
- What's the best way to structure the solution?
  - Solution 1: single threaded server. write next-free-integer to disk.
  - Solution 2: single threaded server. check out integers in chunks.
  - Solution 3: multithreaded server. check out integers in chunks.

Why multithread the server?

- Simpler to program
  - Each server thread deals with only one request at a time.
- Better response time
  - A server thread can block (page fault, IO) without denying service to other users.
- Better performance
  - Server threads can share a pool of fresh integers directly in memory.
- Better control
  - can prioritize scheduler

The simplification argument

- The best way to solve a hard problem is to break it down into lots of little problems
  - structured programming, derivatives, cooking a tasty meal.
- Inside a distributed program or server, many activities outstanding
- Choice is to either write one big program (loop, really) to handle all, or to define each activity within its own execution context.
  - also a performance argument

The Operating System and Multithreadedness

```
for (;;)  {
      if (syscallOccurred()) handleSyscall();
      if (networkMessageIn()) getNetworkMessage();
      if (keyQueueReady()) getKeys();
      if (anyBlockReady()) getBlock();
      …
      …
}
```

It never finishes.
Each operation needs to execute quickly. (why?)
What do we do if some operations take a long time?
loop throughput is gated by slowest function.
want to overlap IO with computation

Threads and Structure

- With threads, we can put each activity on its own Virtual CPU.
  - can run independently (mostly) of the others
  - relative rates no longer matter

```
for(;;) {
        wait for keystroke
        getKeys();
}
```

```
for(;;) {
        wait for syscall
        handleSyscall();
}
```

```
for(;;) {
        wait for network msg
        getNetworkMessage();
}
```

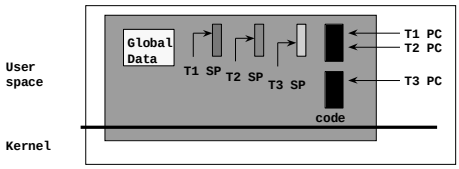
```
for(;;) {
        wait for diskBlocks
        getBlock();
}
```

Threads and Processor State

- A thread is defined by it's machine state (SP, PC)
  - The PC points to the next instruction to execute.
  - The SP points to the thread's private execution context
    - eg, local variables, in progress procedure calls, saved registers
- Threads run in the same address space
  - can share code and global data
  - should not share the stack.
  - share all the same operating system services and resources
    - network channels, open files, windows, user
- Context switching w/o changing address space

Processes, threads, and the program

- Threads can be implemented at user level, in the kernel, or in both places
  - Process as a virtual CPU!



Threads are "lighter weight" than processes

- To make a new thread, we only need a stack.
  - Can share all pre-existing resources.
  - Done at user-level without system calls
- To make a new process, we have to talk to the operating system
  - Make a new address space
  - Inherit resources from the original space
  - Compete for the same underlying global scheduler
  - Can easily run out, or thrash

Example: Web Serving

```
SocketServer()
{
    while (1) {
        int fd = GetNextWebRequest();
        if (fork() == 0)
            HandleRequest(fd);
    }
}

HandleRequest(int fd)
{
…
}
```

When we fork off a new process, we create a huge infrastructure for doing something simple

```
SocketServer()
{
    while (1) {
        int fd = GetNextWebRequest();
        if (fork() == 0)
            HandleRequest(fd);
    }

HandleRequest(int fd)
{
…
}
```

Operating System

Serving with Threads

```
SocketServer()
{
    while (1) {
       int fd = GetNextWebRequest();
        thread_fork(HandleRequest, fd);
    }
}

HandleRequest(int fd)
{
…
}
```

Operating System

Issues in Thread Systems

- The Interface
- Context switch
- Preemptive vs. Non Preemptive
- Scheduling policy
- Synchronization

The Interface

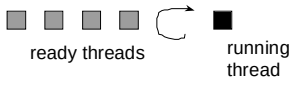
- thread_fork(initial context)
  - create a new thread of control
  - sometimes show as
    - thread_create(), thread_setstate(initial context)
- thread_stop()
  - stop the calling thread, sometimes called thread_block
- thread_start(t)
  - start the named thread
- thread_yield()
  - voluntarily give up the processor
- thread_exit()
  - terminate the calling thread, somtimes called thread_destroy.

The Scheduler

- The scheduler determines when a thread runs.
- It maintains some queues to keep track of what threads are doing.
  - the ready_queue: threads ready to run, but w/o a processor
  - the run_queue: threads running now (mostly only one)
  - is there a wait_queue??

```
// SIMPLIFIED!!!  Scheduler
queue_t readyQueue;
thread_t currentThread;
thread_t schedulerThread;

for (;;) {
        thread_t t = GetQueueHead(readyQueue);
        if (t) {
                swtch_context(schedulerThread, t);
        }
}
```

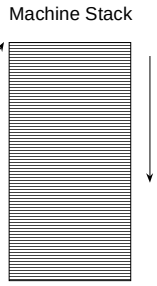


Three key data structures

- The ready queue
- The thread itself
- The machine dependent stack

```
typedef char *address_t;

typedef struct thread {
      queue_t next;
      address_t stack_cur;
} *thread_t;
```



Switching Contexts

- Switching contexts swtch_context(cur, next) involves four basic operations
  - save "cur"'s context of the currently running thread
    - push all machine state onto the stack
  - restore next's context
    - pop all machine state from the thread's stack
  - indicate that the currently running thread is "next"
  - return
    - where do you return to??
- This stuff is always done in assembly language
  - it works AT the level of the calling convention, so it can not use it.

Preemption

- Our scheduler so far is not very friendly
  - one long running thread will take over the machine forever.
  - only voluntary calls to thread_yield, thread_stop, or thread_exit get the scheduler going again.
    - consider how any one of these must work.
- We want to preemptively schedule the processor.
- Two issues
  - mechanism -- how are we going to yank away
  - policy -- when are we going to yank away
- This "mechanism/policy" perspective comes up again in system designs

Mechanism

- To preempt a thread, we need to regain control of the processor.
- This is done with a special timer interrupt that occurs once every *quantum*.
  - typically on the order of 10 to 100 milliseconds
- The interrupt forces control to an interrupt handler.
- The interrupt handler forces the current thread to "call" thread_yield.
  - lots of ways to do this. Why do any of them work?

Policy

- Goal is to optimize:
  - fairness, efficiency, response time, turnaround time, and throughput.
  - this is impossible (well, extremely hard).
- How short is the quantum?
  - too long, and you penalize short running jobs
  - too short, and you spen all your time context switching
  - want quantum to be slightly longer than is needed by most jobs
    - changes over time
- Who runs next?
  - Preemptive round-robin
    - everybody gets a turn;
  - Multiple queues (sometimes with feedback)

Summary

- The operating system is a big multithreaded program
  - each process executes as a thread within the OS
- Applications also use threads
- Threads and scheduling are a key service provided by the OS
- They also create some interesting problems in concurrency control