COS 318: Operating Systems

Processes and Threads

Jaswinder Pal Singh and a Fabulous Course Staff
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)

Next Few Lectures

- Processing: Concurrency and Sharing
  - Processes and threads
  - Synchronization
  - CPU scheduling
  - Deadlock

Today's Topics

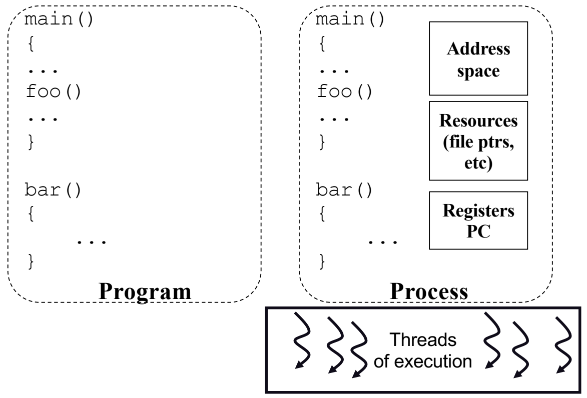
- Concurrency
- Processes
- Threads

Concurrency, Processes and Threads

- Concurrency
  - Many things going on in an operating system
    - Application process execution, interrupts, background tasks, maintenance
  - CPU is shared, as are I/O devices
  - Human beings are not very good at keep track of this and programming it monolithically
  - Processes (and threads) are abstractions to bridge this gap
- Concurrency via Processes
  - Decompose complex problems into simple ones
  - Make each simple one a process
  - Processes run 'concurrently' but each process feels like it has its own CPU
- Example: gcc (via "gcc –pipe –v") launches the following
  - /usr/libexec/cpp | /usr/libexec/cc1 | /usr/libexec/as | /usr/libexec/elf/ld
  - Each instance of cpp, cc1, as and ld running is a process

Process

- An instance of a program in execution
  - Program code, execution context, one or more threads



Process vs. Program

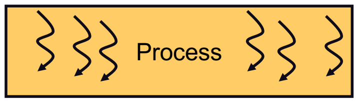
- Process > program
  - Program is just the code; just part of process state
  - Example: many users can run the same program
- Process < program
  - A program can invoke more than one process
  - Example: Fork off processes
  - Many processes can be running the same program

Simplest Process

- Sequential execution
  - One thread per process
  - No concurrency inside a process
  - Everything happens sequentially
  - Some coordination may be required
- Process state
  - Registers
  - Main memory
  - I/O devices
    - File system
    - Communication ports
  - …

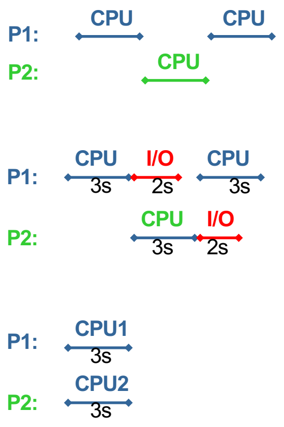
Threads

- A process has an address space and resources
- Thread: a locus of execution
  - A sequential execution stream within a process (sometimes called lightweight process)
  - Separately schedulable: OS/runtime can run/suspend
  - A process can have one or more threads
  - Threads in a process share the same address space
- Can have concurrency across processes, and/or across threads within a process
  - We will initially assume one thread per process



Process Concurrency

- Virtualization
  - Processes interleaved on CPU
- I/O concurrency
  - P1 doing I/O overlapped with P2 running on CPU
  - Each runs almost as fast as if it has its own computer
  - Reduce total completion time
- CPU parallelism
  - Multiple CPUs (such as SMP)
  - Processes running in parallel
  - Speedup



Parallelism

- Parallelism is common in real life
  - A single sales person sells $1M annually
  - Hire 100 sales people to generate $100M revenue
- Speedup
  - Ideal speedup is factor of N
  - Reality: bottlenecks + coordination overhead reduce speedup
- Questions
  - Can you speed up by working with a partner?
  - Can you speed up by working with 20 partners?
  - Can you get super-linear (more than a factor of N) speedup?

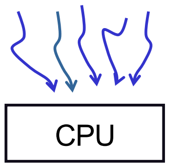
Concurrency in Computing

- Parallel programs
  - To achieve better performance
- Servers (expressing logically concurrent tasks)
  - Multiple connections handled simultaneously
- Programs with user interfaces
  - To achieve user responsiveness while doing computation
- Network and disk bound programs
  - To hide network/disk latency

The Processing Illusion

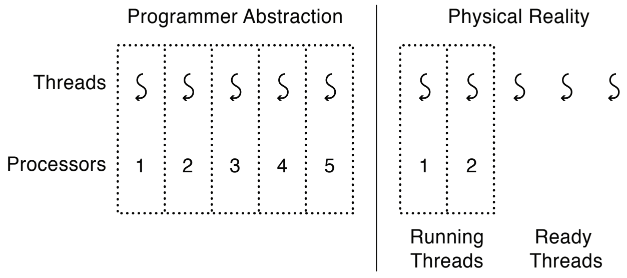
- Every process thinks it owns the CPU
  - Yet on a uniprocessor all processes share the same physical CPU
  - How does this work?
  - Processes are interleaved on the CPU
- Two key pieces:
  - PCB --- process control block, one per process, holds execution state
  - dispatching loop:

```
while(1)
  interrupt
  save state
  get next process
  load state, jump to it
```



The Abstraction

- Every process (thread) runs on a dedicated virtual processor, with unpredictable/variable speed
  - Programs must work with any schedule



Programmer vs. Processor View

Programmer's View

```
.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.
```

Possible Execution #1

```
.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.
```

Possible Execution #2

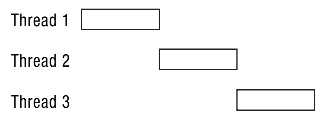
```
.
.
.
x = x + 1;
..............
Thread is suspended.
Other thread(s) run.
Thread is resumed.
...............
y = y + x;
z = x + 5y;
```

Possible Execution #3

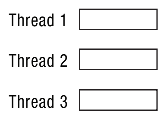
```
.
.
.
x = x + 1;
y = y + x;
...............
Thread is suspended.
Other thread(s) run.
Thread is resumed.
................
z = x + 5y;
```

Possible executions

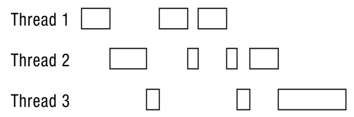
One Execution (1 core)



Another Execution (3 cores)



Another Execution (1 core)



Managing Execution: Process Control Block

PCB holds state and resource information associated with a process

- Process management info
  - Identification
  - State
    - Ready: ready to run.
    - Running: currently running.
    - Blocked: waiting for resources
  - Registers, EFLAGS, EIP, and other CPU state
  - Stack, code and data segment
  - Parents, etc
- Memory management info
  - Segments, page table, stats, etc
- I/O and file management
  - Communication ports, directories, file descriptors, etc.
- Resource allocation and accounting information

Process Control Block

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Possible fields of a PCB

API for Process Management

- Creation and termination
  - Exec, Fork, Wait, Kill
- Signals
  - Default action, Handler, Ways to send
- Operations
  - Block, Yield
- Synchronization
  - We will talk about this a lot more later

Create a Process

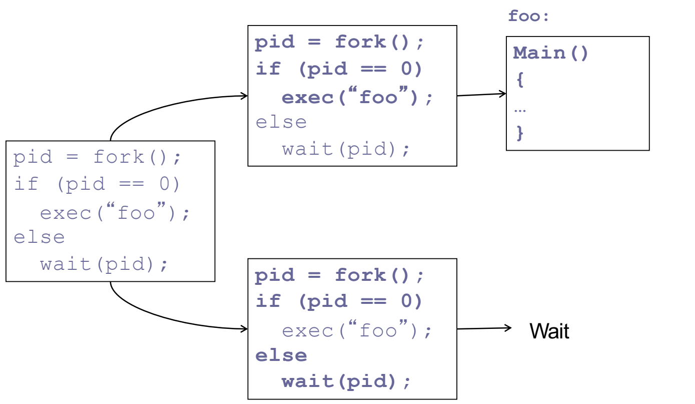
- Creation
  - Load code and data into memory
  - Create an empty call stack
  - Initialize state
  - Make the process ready to run
- Cloning a process
  - Save state of current process
  - Make copy of current code, data, stack and OS state
  - Make the process ready to run

Unix Example

- Methods to create and run processes:
  - fork clones a process
  - exec overlays the current process

```
pid = fork();
if (pid == 0)
  /* child process */
  exec("foo");  /* does not return */
Else
  /* parent */
  wait(pid);    /* wait for child to die */
```

Fork and Exec in Unix



```
pid = fork();
if (pid == 0)
  exec("foo");
else
  wait(pid);
```

```
pid = fork();
if (pid == 0)
  exec("foo");
else
  wait(pid);
```

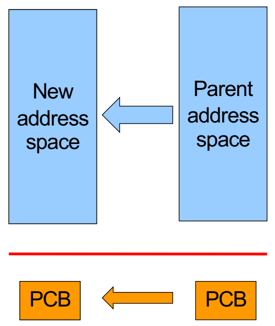
foo:

```
Main()
{
…
}
```

```
pid = fork();
if (pid == 0)
  exec("foo");
else
  wait(pid);
```

More on Fork

- Create and initialize PCB
- Create an address space
- Copy the content of the parent address space to the new address space
- Child inherits the execution context of the parent (e.g. open files)
- Inform scheduler that new process is ready

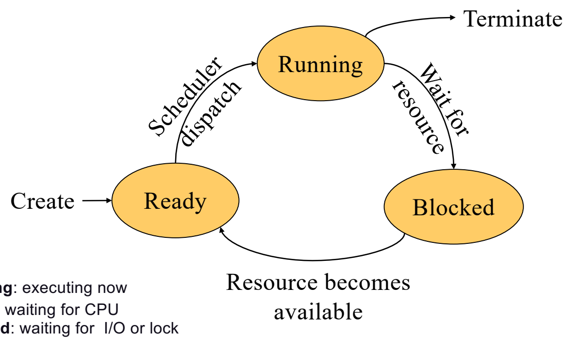


Process Context Switch

- Save a context (everything that a process may damage)
  - All registers (general purpose and floating point)
  - All co-processor state
  - Save all memory to disk?
  - What about cache and TLB?
- Start a context
  - Does the reverse
- Challenge
  - OS code must save state without changing any state
  - E.g. how should OS run without touching any registers?
    - CISC machines have a special instruction to save and restore all registers on stack
    - RISC: reserve registers for kernel or have way to carefully save one and then continue

Process State Transition

Non-preemptive case: e.g. no timer interrupts

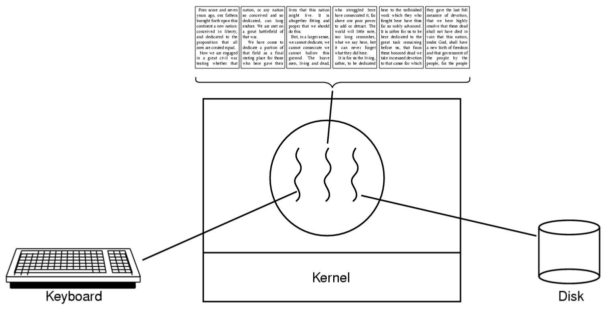


Running: executing now
Ready: waiting for CPU
Blocked: waiting for I/O or lock

Threads

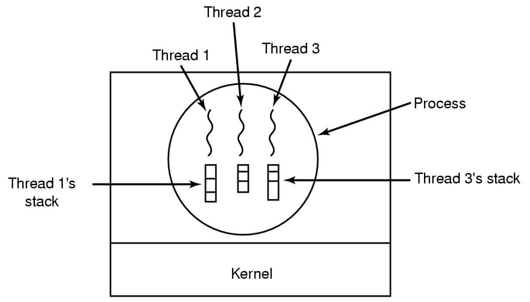
- Thread
  - A sequential execution stream within a process (also called lightweight process)
  - Separately schedulable: OS or runtime can run or suspend at any time
  - A process may have one or more threads (loci of execution)
  - Threads in a process share the same address space
- Thread concurrency
  - Easier to program overlapping I/O and CPU with threads than with signals
  - A server (e.g. file server) serves requests with different threads
  - Multiple CPUs sharing the same memory

Thread Usage Example



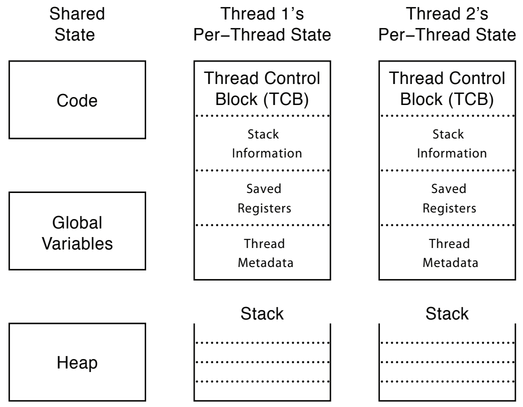
A word processor with three threads

Threads (cont’d)



Every thread has its own stack

Thread data structures



Thread Control Block (TCB)

- State
  - Ready: ready to run
  - Running: currently running
  - Blocked: waiting for resources
- Registers
- Status (EFLAGS)
- Program counter (EIP)
- Stack
- Code

Threads (cont'd)

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

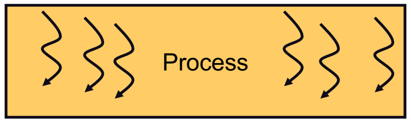
- Per process: Items shared by all threads in a process
- Per thread: Items private to each thread

Typical Thread API

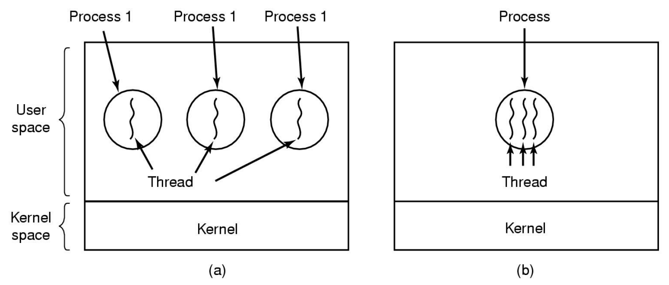
- Creation
  - Fork, Join
- Mutual exclusion
  - Acquire (lock), Release (unlock)
- Condition variables
  - Wait, Signal, Broadcast
- Alert
  - Alert, AlertWait, TestAlert

Revisit Process

- Process
  - Threads (simplest process has only one thread)
  - Address space
  - Environment for the threads to run on OS (resources in use like open files, etc)



Threads and Processes



(a) Three processes each with one thread

(b) One process with three threads

- Process = thread + address space + OS env (open files, etc.)
- Thread encapsulates concurrency; address space encapsulates protection

Thread Context Switch

- Save a context (everything that a thread may damage)
  - All registers (general purpose and floating point)
  - All co-processor state
  - Need to save stack?
  - What about cache and TLB?
- Start a context
  - Does the reverse
- May trigger a process context switch

Procedure Call

- Caller or callee save some context (same stack)
- Caller saved example:

```
save active caller registers
call foo
                                   foo() {
                                         do stuff
                                   }

restore caller regs
```

Threads vs. Procedures

- Threads may resume out of order
  - Cannot use LIFO stack to save state
  - Each thread has its own stack
- Threads switch less often
  - Each thread “has” its own CPU
- Threads can be asynchronous
  - Procedure call can use compiler to save state synchronously
  - Threads can run asynchronously
- Multiple threads
  - Multiple threads can run on multiple CPUs in parallel
  - Procedure calls are sequential

Process vs. Threads

- Address space
  - Processes do not usually share memory (address space)
  - Process context switch switches page table and other memory mechanisms
  - Threads in a process share the entire address space
- Privileges
  - Processes have their own privileges (e.g. file access)
  - Threads in a process share all privileges

40

Real Operating Systems

- One or many address spaces
- One or many threads per address space

	1 address space	Many address spaces
1 thread per address space	MSDOS Macintosh	Traditional Unix
Many threads per address space	Embedded OS, Pilot	VMS, Mach (OS-X), OS/2, Windows NT/XP/Vista/7, Solaris, HP-UX, Linux

41

Summary

- Concurrency
  - CPU and I/O
  - Among applications
  - Within an application
- Processes
  - Abstraction for application concurrency
- Threads
  - Abstraction for concurrency within an application

42