Cs362 Operating Systems Lecture 3

Concurrency: Processes, Threads, and Address Spaces

Goals for Today

- How do we provide multiprogramming?
- What are Processes?
- How are they related to Threads and Address Spaces?

Concurrency

- "Thread" of execution
  - Independent Fetch/Decode/Execute loop
  - Operating in some Address space
- Uniprogramming: one thread at a time
  - MS/DOS, early Macintosh, Batch processing
  - Easier for operating system builder
  - Get rid concurrency by defining it away
  - Does this make sense for personal computers?
- Multiprogramming: more than one thread at a time
  - MULTIX, UNIX, OS/2, Windows NT/2000/XP
  - Often called "multitasking", but multitasking has other meanings (talk about this later)

The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: users think they have exclusive access to machine
- OS Has to coordinate all activity
  - Multiple users, I/O interrupts, ...
  - How can keep this straight?
- Basic Idea: Use Virtual Machine abstraction
  - Decompose hard problem into simpler ones
  - Abstract the notion of an executing program
  - Then, worry about multiplexing these abstract machines
What happens during execution?

Execution sequence:
- Fetch Instruction at PC
- Decode
- Execute (possibly using registers)
- Write Results to registers
- PC = Next Instruction (PC)
- Repeat

Properties of this simple multiprogramming technique
- All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same
- Consequence of sharing:
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- This (unprotected) model common in:
  - Embedded applications
  - Windows 3.1/Macintosh (switch only with yield)
  - Windows 95/ME? (switch with both yield and timer)

How can we give the illusion of multiple processors?
- How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual “CPU” needs a structure to hold:
  - Program Counter (PC)
  - Registers (Integer, Floating point, others...?)
- How switch from one CPU to the next?
  - Save PC and registers in current state block
  - Load PC and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

Modern Technique: SMT/Hyperthreading
- Hardware technique
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- Can schedule each thread as if were separate CPU
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first
- Original technique called “Simultaneous Multithreading”
  - Alpha, SPARC, Pentium 4 (“Hyperthreading”), Power 5
How to protect threads from one another?

- Need three important things:
  - Protection of memory
    » Every task does not have access to all memory
  - Protection of I/O devices
    » Every task does not have access to every device
  - Preemptive switching from task to task
    » Use of timer
    » Must not be possible to disable timer from user code

Recall: Program’s Address Space

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are $2^{32} = 4$ billion addresses
- What happens when you read or write to an address?
  - Perhaps Nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    » (Memory-mapped I/O)
  - Perhaps causes exception (fault)

Providing Illusion of Separate Address Space: Load new Translation Map on Switch

Traditional UNIX Process

- Process: Operating system abstraction to represent what is needed to run a single program
  - Often called a “HeavyWeight Process”
  - Formally: a sequential stream of execution in its own address space
- Two parts:
  - Sequential Program Execution Stream
    » Code executed as a single, sequential stream of execution
    » Includes State of CPU registers
  - Protected Resources:
    » Main Memory State (contents of Address Space)
    » I/O state (i.e. file descriptors)
- Important: There is no concurrency in a heavyweight process
How do we multiplex processes?

- The current state of process held in a process control block (PCB):
  - This is a “snapshot” of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process “running” at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    » Memory Mapping: Give each process their own address space
    » Kernel/User duality: Arbitrary multiplexing of I/O through system calls

CPU Switch From Process to Process

- This is also called a “context switch”
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but... contention for resources instead

Diagram of Process State

- As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - ready: The process is waiting to run
  - terminated: The process has finished execution

Process Scheduling

- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)
What does it take to create a process?

- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space
  - More expensive
- Copy data from parent process? (Unix `fork()`)
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally very expensive
  - Much less expensive with “copy on write”
- Copy I/O state (file handles, etc)
  - Medium expense

Process =? Program

- More to a process than just a program:
  - Program is just part of the process state
  - I run emacs on lectures.txt, you run it on homework.java – Same program, different processes
- Less to a process than a program:
  - A program can invoke more than one process
  - `cc` starts up `cpp`, `cc1`, `cc2`, `ld`, etc

Multiple Processes to Contribute on Task

- High Creation/memory Overhead
- (Relatively) High Context-Switch Overhead
- Need Communication mechanism:
  - Separate Address Spaces Isolates Processes
  - Shared-Memory Mapping
    - Accomplished by mapping addresses to common DRAM
    - Read and Write through memory
  - Message Passing
    - `send()` and `receive()` messages
    - Works across network

Shared Memory Communication

- Communication occurs by “simply” reading/writing to shared address page
  - Really low overhead communication
  - Introduces complex synchronization problems
Inter-process Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message) - message size fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Modern “Lightweight” Process with Threads

- Thread: a sequential execution stream within process
  (Sometimes called a “Lightweight process”)
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: a single program made up of a number of different concurrent activities
  - Sometimes called multitasking, as in Ada...
- Why separate the concept of a thread from that of a process?
  - Discuss the “thread” part of a process (concurrency)
  - Separate from the “address space” (Protection)
  - Heavyweight Process = Process with one thread

Single and Multithreaded Processes

- Threads encapsulate concurrency
  - “Active” component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - “Passive” component of a process

Examples of multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Most modern OS kernel
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done
Examples of multithreaded programs (con't)

- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems

- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing

- Some multiprocessors are actually uniprogrammed:
  - Multiple threads in one address space but one program at a time

Thread State

- State shared by all threads in process/addr space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc)

- State “private” to each thread
  - Kept in TCB = Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack - what is this?

Execution Stack

- Parameters, Temporary variables
  - return PCs are kept while called procedures are executing

Classification

<table>
<thead>
<tr>
<th># threads Per AS:</th>
<th># of addr spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
</tr>
</tbody>
</table>

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space

- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs
Example: Implementation Java OS

- Many threads, one Address Space
- Why another OS?
  - Recommended Minimum memory sizes:
    » UNIX + X Windows: 32MB
    » Windows 98: 16-32MB
    » Windows NT: 32-64MB
    » Windows 2000/XP: 64-128MB
- What if want a cheap network
  point-of-sale computer?
  » Say need 1000 terminals
  » Want < 8MB
- What language to write this OS in?
  - Java/Lisp? Not quite sufficient – need direct access to HW/memory management

Summary

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (yield(), I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- Book talks about processes
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process