Operating Systems
(1DT020 & 1TT802)

Lecture 2
Processes, threads, process dispatching

March 25, 2009

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt09
What is an Operating System?

• No universally accepted definition
  – “Everything a vendor ships when you order an operating system” is good approximation
  – “The one program running at all times on the computer” is the kernel.

An OS is responsible of 2 main tasks:

– Provide a virtual machine abstraction
  » Turn hardware/software peculiarities into what programmers want/need
    Application program view: an OS extends the processor’s instruction set with new (complex) instructions accessible via system calls.

– Resources (Hardware and Software) management, sharing and protection
  » Optimize for convenience, utilization, security, reliability, etc.

The 2 tasks are not separate
Goals for Today

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

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)
Concurrency

• Stream ("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 of concurrency by defining it away
  – Does this make sense for personal computers?

• Multiprogramming: more than one thread at a time
  – Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X
  – Often called “multitasking”, but multitasking has other meanings (see 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 it keep all these things 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
- Dijkstra did this for the “THE system”
  - Few thousand lines vs 1 million lines in OS 360 (1K bugs)
Recall (Computer Architecture): What happens during execution?

Execution sequence:
- Fetch Instruction at PC
- Decode
- Execute (possibly using registers)
- Write results to registers or memory
- PC = Next Instruction(PC)
- Repeat
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 data structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How do we switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things
Properties of this simple multiprogramming technique

• All virtual CPUs share same non-CPU resources
  – I/O devices are the same
  – Memory is 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 to protect threads from one another?

• Need three important things:
  1. Protection of memory
     » Every task does not have access to all memory
  2. Protection of I/O devices
     » Every task does not have access to every device
  3. 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
  – Divided in user program address space and kernel address space

• 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)
SPIM user programs and kernel address spaces (MIPS simulator – pedagogical tool)

- **user program address space**

  - `0x7FFFFFFF` stack
  - `0x00000000` code
  - `0x10000000` static data (global vars, ...)
  - `0x00400000` reserved

- **kernel address space** : `0x80000000` → `0xFFFFFFFF`

3/25/09
INTEL/Linux address spaces

- Kernel memory (code, data, heap, stack)
- User stack (created at runtime)
- Memory-mapped region for Shared libraries
- Run-time heap (malloc, free, new)
- Read/write data segment (.data, .bss)
- Read-only data segment (.init, .text, .rodata)
- Unused

Memory invisible to user code
%esp (stack pointer)
brk
Loaded from the Executable file
Providing Illusion of Separate Address Space: Load new Translation Map on Switch

Translation Map 1

Translation Map 2

Physical Address Space

Translation Map 1

Translation Map 2

Program 1
Virtual Address Space 1

Program 2
Virtual Address Space 2

OS heap & Stacks

OS code

OS data
Traditional UNIX Process

• **Process**: *Operating system abstraction to represent what is needed to run a single program*
  – Often called a “Heavy Weight Process”
  – Formally: a single, 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 (CPU Scheduling or Process dispatching):
  – 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

<table>
<thead>
<tr>
<th>pointers</th>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td>process id</td>
<td></td>
</tr>
<tr>
<td>program counter</td>
<td></td>
</tr>
<tr>
<td>other registers</td>
<td></td>
</tr>
<tr>
<td>memory limits</td>
<td></td>
</tr>
<tr>
<td>list of open files</td>
<td></td>
</tr>
</tbody>
</table>
CPU Switch From Process to Process

• This is also called a “context switch”
• How long does it take to switch from one process to another?
• Code executed in kernel just for switching is overhead
  – Overhead sets minimum practical switching time
  – Less overhead with SMT/hyperthreading, but… contention for resources instead
• As a process executes, it changes state
  – new: The process is being created
  – ready: The process is waiting to run
  – running: Instructions are being executed
  – waiting: Process waiting for some event to occur
  – terminated: The process has finished execution
• PCBs move from queue to queue as they change state
  – Decisions about which order to remove from queues are **Scheduling** decisions
  – Many algorithms possible
What does it take to create a process?

• **Must construct new PCB**
  – Inexpensive

• **Must set up new translation map 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
Multiple Processes Collaborate on a Task

- (Relatively) High Context-Switch Overhead
- Separate address spaces isolates processes
- Need Inter-Process Communication mechanism (IPC):
  - 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
Message Passing Communication

• Mechanism for processes to communicate and to synchronize their actions
• Message system – processes communicate with each other without resorting to shared variables
• 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, system calls/traps)
  – logical (software)
Modern “Lightweight” Process with Threads

• Thread: *a sequential execution stream within a 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?
  – Deal with 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
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?
Examples of multithreaded programs

• Embedded systems
  – Elevators, Planes, Medical systems, Wristwatches
  – Single Program, concurrent operations

• Most modern OS kernels
  – 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, local variables, temporary storage
  – return PCs are kept while called procedures are executing
### Classification

<table>
<thead>
<tr>
<th># threads Per AS:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)</td>
<td>Mach, OS/2, Linux Windows 9x???, Win NT to XP, Solaris, HP-UX, OS X</td>
</tr>
</tbody>
</table>

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

- Windows 95/98/ME did not have real memory protection
  - Users could overwrite process tables/System DLLs
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