Operating Systems
(1DT020 & 1TT802)

Lecture 3 (2)

More on threads

March 26, 2009

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt09
Goals

• Further Understanding Threads
• Thread Dispatching

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne.
Many slides generated from my lecture notes by Kubiatowicz.
Recall: 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?
Single-Threaded Example

• Imagine the following C program:

```c
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
}
```

• What is the behavior here?
  – Program would never print out class list
  – Why? ComputePI would never finish
Use of Threads

• Version of program with Threads:

```c
main() {
    CreateThread(ComputePI("pi.txt"));
    CreateThread(PrintClassList("clist.text"));
}
```

• What does “CreateThread” do?
  – Start independent thread running given procedure

• What is the behavior here?
  – Now, you would actually see the class list
  – This should behave as if there are two separate CPUs
Memory Footprint of Two-Thread Example

• If we stopped this program and examined it with a debugger, we would see
  – Two sets of CPU registers
  – Two Stacks

• Questions:
  – How do we position stacks relative to each other?
  – What maximum size should we choose for the stacks?
  – What happens if threads violate this?
  – How might you catch violations?
Per Thread State

• Each Thread has a *Thread Control Block* (TCB)
  – Execution State: CPU registers, program counter, pointer to stack
  – Scheduling info: State (more later), priority, CPU time
  – Accounting Info
  – Various Pointers (for implementing scheduling queues)
    – Pointer to enclosing process? (PCB)?
  – Etc (add stuff as you find a need)

• OS Keeps track of TCBs in protected memory
  – In Array, or Linked List, or …
Lifecycle of a Thread (or Process)

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

- “Active” threads are represented by their TCBs
  - TCBs organized into queues based on their state

![Lifecycle of a Thread Diagram]

I/O or event completion → scheduler dispatch → I/O or event wait

new → admitted → interrupt → exit → terminated

ready → running → waiting → ready
Ready queue and various I/O device queues

- Thread not running ⇒ TCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy

![Diagram showing linked structures for different devices and states]
Running a thread / switching to a new thread

• How do I run a thread?
  – Load its state (registers, PC, stack pointer) into CPU
  – Load environment (virtual memory space, etc)
  – Jump to the PC

• How does the dispatcher get control back?
  – Internal events: thread returns control voluntarily (using a system call instruction)
  – External events: thread gets preempted (interrupt)

• How does dispatcher switch to a new thread?
  – Save anything next thread may trash: PC, regs, stack
  – Maintain isolation for each thread private state
  – Also save current environment (virtual memory space, etc) and load new environment if switch to a thread in another process
Dispatch Loop

• Conceptually, the dispatching loop of the operating system looks as follows:

\[
\text{Loop} \{ \\
    \text{SaveStateOfCPU(curTCB);} \\
    \text{ChooseNextThread();} \\
    \text{LoadStateOfCPU(newTCB);} \\
\}
\]

• This is an *infinite* loop
  – One could argue that this is all that the OS does

• Should we ever exit this loop???
  – When would that be?
  – Is this really implemented as loop (with jumps ?)
Saving/Restoring state
(often called “Context Switch”)

SaveStateOfCPU(curTCB) { /* Unload old thread */
    TCB[tCur].regs.r7 = CPU.r7;

    ...

    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/
}

LoadStateOfCPU(newTCB) {
    /* Load and execute new thread */
    CPU.r7 = TCB[tNew].regs.r7;

    ...

    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    returnFromKernel; /* Return to CPU.retpc */
}
Switch Details

• How many registers need to be saved/restored?
  – MIPS 4k: 32 Int(32b), 32 Float(32b)
  – Pentium: 14 Int(32b), 8 Float(80b), 8 SSE(128b),…
  – Sparc(v7): 8 Regs(32b), 16 Int regs (32b) * 8 windows = 136 (32b)+32 Float (32b)
  – Itanium: 128 Int (64b), 128 Float (82b), 19 Other(64b)

• retpc is where the return should jump to.
  – In reality, this is implemented as a jump+switching to user mode

• Normally, switch is implemented in assembly language!
  – (Why ?)

• what if you make a mistake in implementing switch?
  – Suppose you forget to save/restore register 4
  – Get intermittent failures depending on when context switch occurred and whether new thread uses register 4
  – System will give wrong result without warning

• Can you devise an exhaustive test to test switch code?
  – No! Too many combinations and inter-leavings
Choosing a Thread to Run

• How does Dispatcher decide what to run?
  – Zero ready threads – dispatcher loops
    » Alternative is to create an “idle thread”
    » Can put machine into low-power mode
  – Exactly one ready thread – easy
  – More than one ready thread: use scheduling priorities

• Possible priorities:
  – LIFO (last in, first out):
    » put ready threads on front of list, remove from front
  – Pick one at random
  – FIFO (first in, first out):
    » Put ready threads on back of list, pull them from front
    » This is fair and is what Nachos does
  – Priority queue:
    » keep ready list sorted by TCB priority field
Kernel versus User-Mode threads

• We have been talking about Kernel threads
  – Native threads supported directly by the kernel
  – Every thread can run or block independently
  – One process may have several threads waiting on different things

• Downside of kernel threads: a bit expensive
  – Need to make a crossing into kernel mode to schedule

• Even lighter weight option: User Threads
  – User program provides scheduler and thread package
  – May have several user threads per kernel thread
  – User threads may be scheduled non-preemptively relative to each other (only switch on yield())
  – Cheap

• Downside of user threads:
  – When one thread blocks on I/O, all threads block
  – Kernel cannot adjust scheduling among all threads
  – Option: Scheduler Activations
    » Have kernel inform user level when thread blocks...
Threading models mentioned by book

Simple One-to-One Threading Model

Many-to-One

Many-to-Many
Correctness for systems with concurrent threads

• If dispatcher can schedule threads in any way, programs must work under all circumstances
  – Can you test for this?
  – How can you know if your program works?

• Independent Threads:
  – No state shared with other threads
  – Deterministic $\Rightarrow$ Input state determines results
  – Reproducible $\Rightarrow$ Can recreate Starting Conditions, I/O
  – Scheduling order doesn’t matter (if `switch()` works!!!)

• Cooperating Threads:
  – Shared State between multiple threads
  – Non-deterministic
  – Non-reproducible

• Non-deterministic and Non-reproducible means that bugs can be intermittent
  – Sometimes called “Heisenbugs”
Why allow cooperating threads?

• People cooperate; computers help/enhance people’s lives, so computers must cooperate
  – By analogy, the non-reproducibility/non-determinism of people is a notable problem for “carefully laid plans”

• Advantage 1: Share resources
  – One computer, many users
  – One bank balance, many ATMs
    » What if ATMs were only updated at night?
  – Embedded systems (robot control: coordinate arm & hand)

• Advantage 2: Speedup
  – Overlap I/O and computation
    » Many different file systems do read-ahead
  – Multiprocessors – chop up program into parallel pieces

• Advantage 3: Modularity
  – More important than you might think
  – Chop large problem up into simpler pieces
    » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    » Makes system easier to extend
Summary

• The state of a thread is contained in the TCB
  – Registers, PC, stack pointer
  – States: New, Ready, Running, Waiting, or Terminated

• Multithreading provides simple illusion of multiple CPUs
  – Switch registers and stack to dispatch new thread
  – Provide mechanism to ensure dispatcher regains control

• Switch routine
  – Can be very expensive if many registers
  – Must be very carefully constructed!

• Many scheduling options
  – Decision of which thread to run complex enough for complete lecture

• Threads may be at user-level or kernel level

• Cooperating threads have many potential advantages
  – But: introduces non-reproducibility and non-determinism
  – Need to have Atomic operations