CS533 Concepts of Operating Systems

Jonathan Walpole
Scheduler Activations
How to Implement Threads?

Kernel threads
- built on real CPUs
- managed by the OS
- thread-related operations are system calls
- blocking is well supported

User-level threads
- built on kernel threads
- managed by user-level library code
- thread-related operations are function calls
- blocking is problematic
How Do User Threads Behave?

Blocking I/O?
- underlying kernel thread blocks
- user level thread scheduler has no chance to switch threads
- what if we wrap calls?
- page faults can’t be wrapped

CPU preemption and priority inversion

There is a mismatch: who’s fault is it?
How Should Threads Behave?

Blocking I/O?

Page faults?

CPU preemption?

These are all significant events for the thread scheduler
- it needs to get control!
Kernel-User Communication

How does an OS kernel talk to the hardware?
How does the hardware talk to the OS kernel?
How does a user-level scheduler talk to the kernel?
How does the kernel talk to a user-level scheduler?
Scheduler Activations

Kernel-level abstraction for control flow
Repository for holding state while blocked
Sounds like a kernel thread, but ...

- it can upcall into (interrupt) the user-level thread scheduler
- it brings along its own CPU
- it brings along the state of what was running before
- enables thread scheduler to switch threads
Processors

Operating System

User-Level Runtime system

User program

1 creates

Scheduler Activation

3 Upcall

Upcall

6 more

User-Level thread mgmmt system

5 creates

4 runs

7 creates

2 Assigns

2 Assigns

3 Upcall

Upcall

Scheduler Activation

Scheduler Activation

Scheduler Activation

Scheduler Activation

User-Level thread mgmmt system

User-Level thread mgmmt system

User-Level thread mgmmt system

User-Level thread mgmmt system

User-Level thread mgmmt system

User-Level thread mgmmt system

User-Level thread mgmmt system
Scheduler Activation System Calls

**Add more processors** (additional # of processors needed)

*Allocate more processors to this address space and start them running scheduler activations.*

**This processor is idle ()**

*Preempt this processor if another address space needs it.*
Scheduler Activation Upcalls

Add this processor (processor #)

Execute a runnable user-level thread.

Processor has been preempted (preempted activation # and its machine state)

Return to the ready list the user-level thread that was executing in the context of the preempted scheduler activation.

Scheduler activation has blocked (blocked activation #)

The blocked scheduler activation is no longer using its processor.

Scheduler activation has unblocked (unblocked activation # and its machine state)

Return to the ready list the user-level thread that was executing in the context of the blocked scheduler activation.
Example

Fig. 1. Example: I/O request/completion.
Tricky Problems

What if a scheduler activation interrupts a thread in a critical section?

What if thread library code is interrupted by a scheduler activation?

What if thread library code causes a scheduler activation?
- i.e. has a page fault?
- what if the scheduler activation handler has a page fault?
Performance Without I/O

Fig. 2. Speedup of N-Body application versus number of processors, 100% of memory available.
Performance With I/O

Fig. 3. Execution time of N-Body application versus amount of available memory, 6 processors.
Conclusions & Discussion

Scheduler Activations seem to have fixed the problem

Are scheduler activations kernel threads done right?