Processes, threads, and kernel architecture

This week: Processes and threads

You will be creating new processes:

- Set up a new CSpace for capabilities
- Set up a new VSpace (virtual address space)
- Load an ELF image into memory
- Map it into both address spaces
- Create a dispatcher for the new process
- Set it running

Note: no `fork()`!

KERNEL THREADING MODELS: MULTIPLEXING CPUS

Kernel Thread Models

- Key design choices for an OS:
  - Support for >1 execution context in the kernel?
  - Where is the stack for executing kernel code?
  - Can kernel code block?
  - If so, how?
- Result: the **Kernel Thread Model**

Kernel Thread Models

Two common alternatives:

1. Per-Thread Kernel Stack:
   – Every thread has a matching kernel stack
2. Single Kernel Stack:
   – Only one stack is used in the kernel (per core).

Per-Thread Kernel Stack

- Every user thread/process has its own kernel stack
- Thread's kernel state implicitly stored in its kernel stack
- If a kernel thread blocks, switch stacks
- To resume switch back to original stack
- Preemption is easy
- No great difference between kernel- and user-mode

```
example(arg1, arg2) {
        P1(arg1, arg2);
        if (need_to_block) {
                thread_block();
                P2(arg2);
        } else {
                P3();
        }
        /* return to user */
        return SUCCESS;
}
```

Single Kernel Stack

- Challenges:
  - How can a single kernel stack support many application processes/threads?
  - How to handle system calls that block?
- Two basic approaches:
  1. Continuations [Draves et al., 1991]
  2. Stateless Kernel [Ford et al., 1999]

Continuations

- Kernel state saved explicitly in TCB
  - Function pointer
  - Variables
- Stack can be reused
- To resume, discard the current stack and load the continuation

```
example(arg1, arg2)
{
    P1(arg1, arg2);
    if (need_to_block) {
        save_context_in_TCB;
        thread_block(example_continue);
        panic("thread_block returned");
    } else {
        P3();
    }
    thread_syscall_return(SUCCESS);
}

example_continue()
{
    recover_context_from_TCB;
    P2(recovered arg2);
    thread_syscall_return(SUCCESS);
}
```

Stateless Kernel

- System calls never block
- If a system call can't complete, it fails
  - User restarts when resources available
  - Kernel stack discarded
- Preemption is hard
  - Must (partially) roll back to a restart point
  - Or design very carefully e.g. **seL4**
- No page faults in kernel code
  - System call arguments in registers
  - Nested page faults are fatal

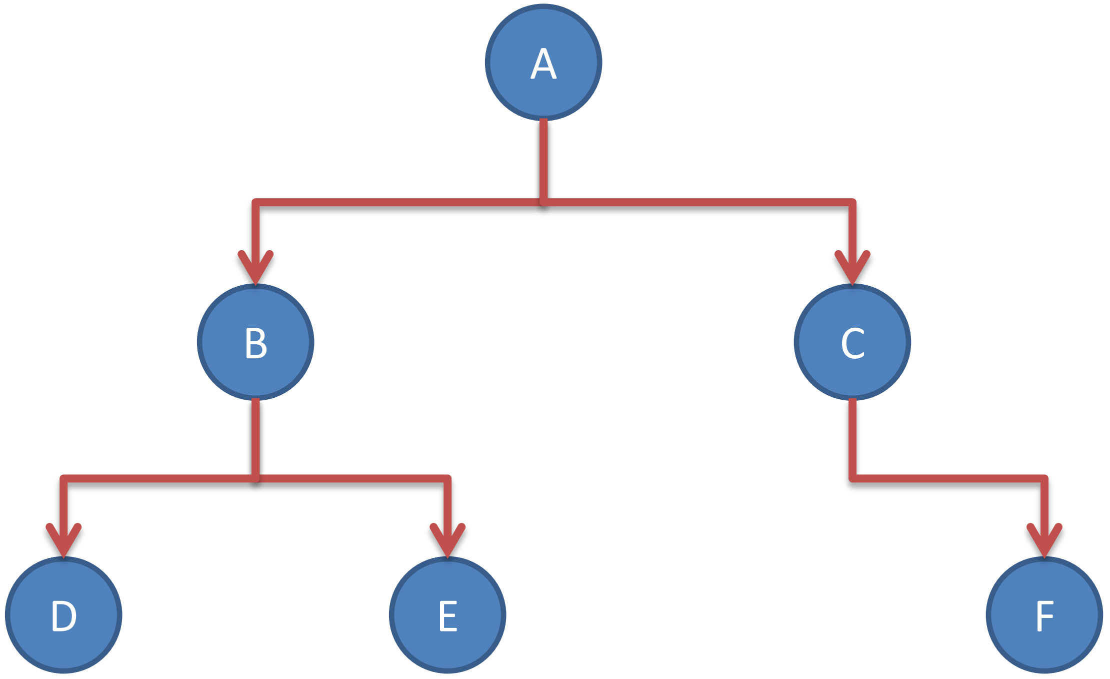
Example: preemption in seL4

- seL4 is a single-stack microkernel
  - No kernel threads
  - No continuations
- seL4 is a real-time system
  - The microkernel is preemptable
- seL4 has long-running operations
  - Recursive delete
  - Zeroing frames (to avoid information leaks)

Example: preemption in seL4

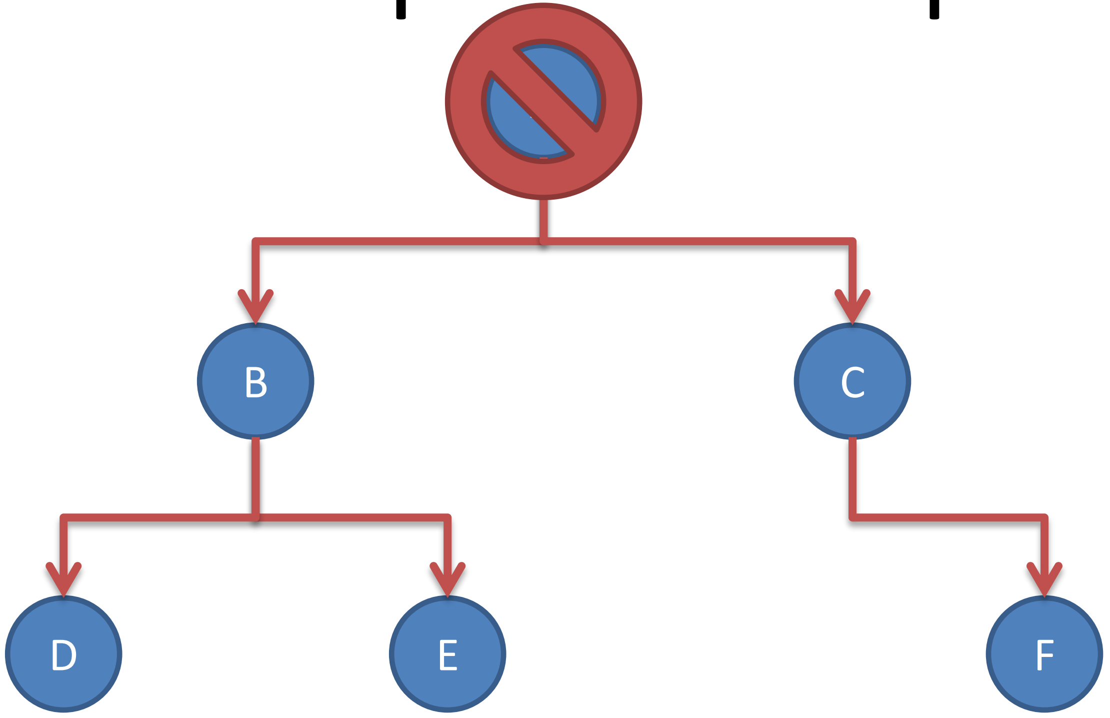
- Long operations have *preemption points*:
  - Where system invariants hold
  - Progress has been made
- Preempted operations *block* the calling thread
- If the thread is rescheduled, it continues
- If another thread touches the partially-deleted region, it restarts the blocked thread

Preemption Example



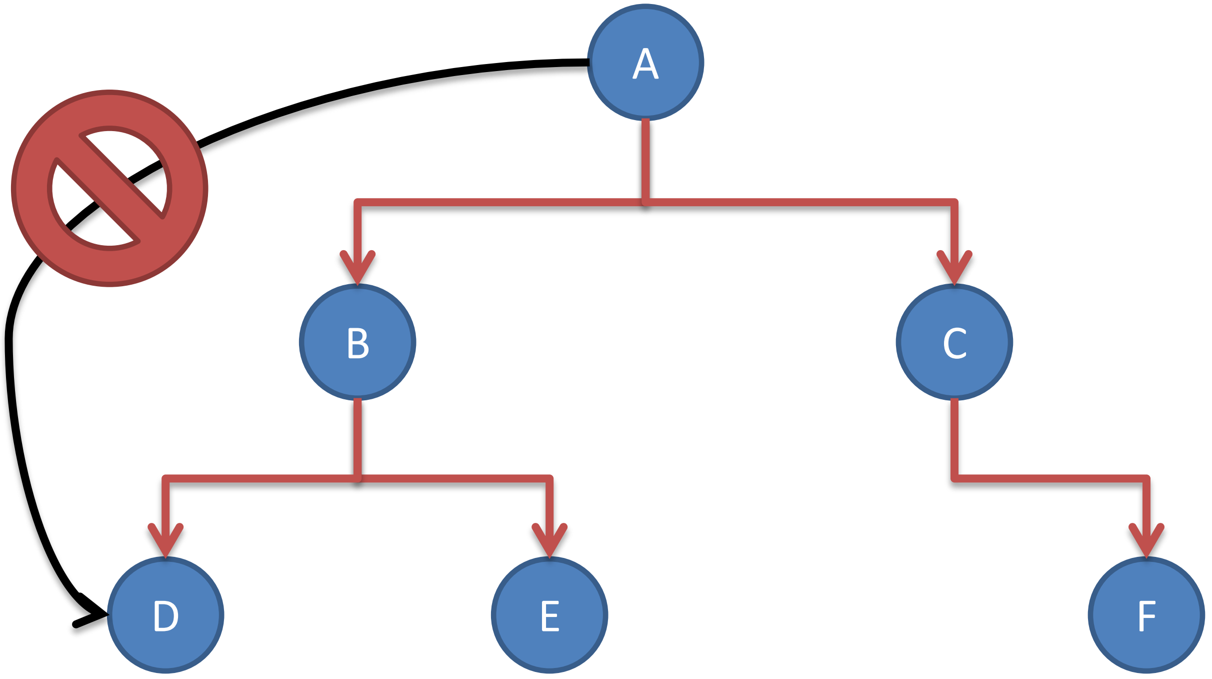
- Delete the whole tree
  - Must delete children before parents
  - Tree may be arbitrarily deep

Preemption Example



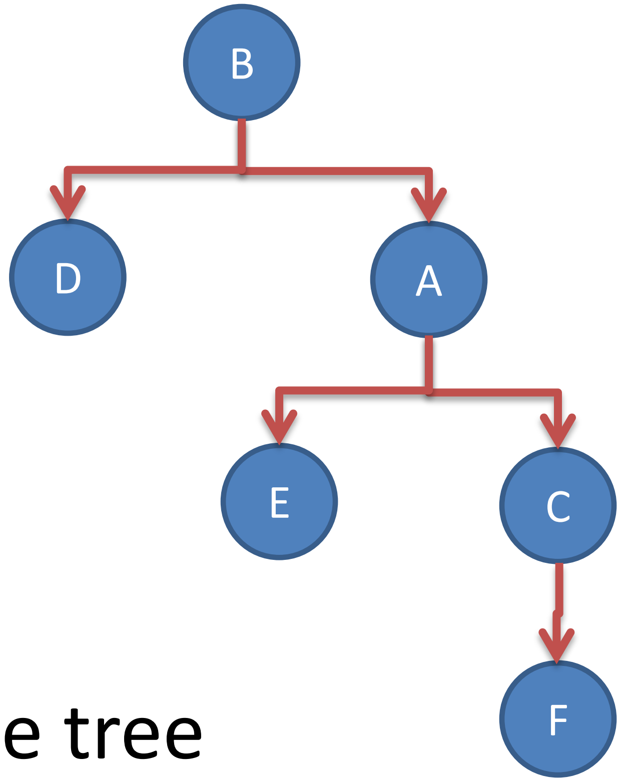
- Delete the whole tree
  - **Must delete children before parents**
  - Tree may be arbitrarily deep

Preemption Example



- Delete the whole tree
  – Must delete children before parents
  – **Tree may be arbitrarily deep**

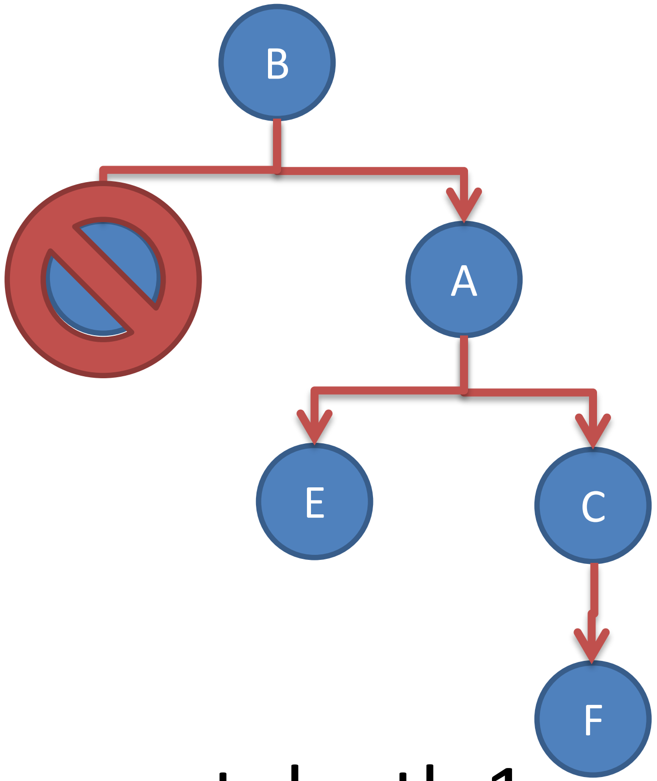
Preemption Example



$$\begin{tikzcd}
 & B \arrow[dl] \arrow[dr] & & \\
D & & A \arrow[dl] \arrow[dr] & \\
 & E & & C \arrow[d] \\
 & & & F
\end{tikzcd}$$

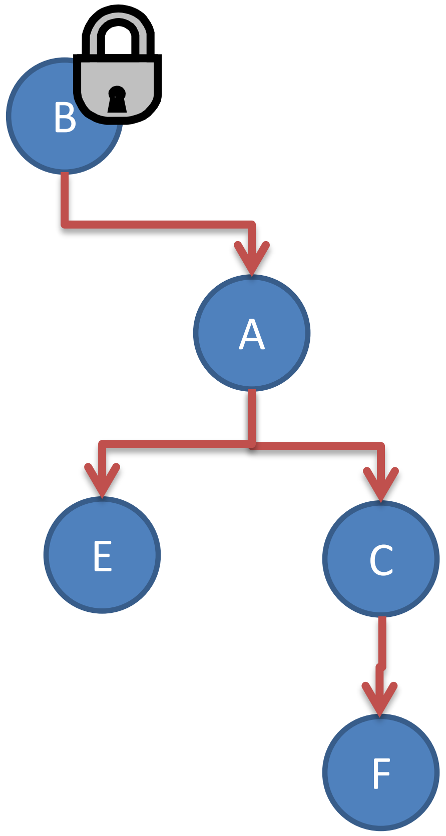
- Rotate the tree
  - Preserves the invariant: “No unrooted subtrees”
  - Only need to inspect up to depth 2

Preemption Example



- Delete leaves at depth 1
- It’s safe to preempt after any such step
  – What if another thread looks at B?

Preemption Example



- Mark the root
  - If any thread enters the tree, it restarts the delete
  - No thread sees a partly-deleted tree

Kernel Stack Model Summary

Per-Thread Kernel Stack

✓ Simple, flexible

- – Kernel can always use threads
- – No special technique to block
- – No real difference between kernel and user mode

✘ Larger cache and memory footprint

- Used by L4Ka::Pistachio, UNIX, Linux, etc.

Kernel Stack Model Summary

Single Kernel Stack

✓ Lower cache & memory footprint (one stack)

Continuations:

✗ Complex to program

✗ Must save state conservatively (whatever might be needed)

- Used by Mach, NICTA::Pistachio

Kernel Stack Model Summary

Single Kernel Stack

✓ Lower cache & memory footprint (one stack)

Stateless kernel:

✘ Even harder to program
  – Must request all resources prior to execution
  – Blocking system calls must be restartable

✘ Processor-provided stack mgmt. can get in the way
  – System calls need to be atomic

- Used by Fluke, Nemesis, Exokernel, seL4, and …

Why Go Stateless?

- Simplest model, if all invocations are:
  - Atomic
  - Non-blocking
  - Fast and **Bounded**
  - Non-preemptable
  - Guaranteed not to page fault
- Restrictive, but fits a uniprocessor µkernel.

Barrelfish

- Stateless kernel, in physical memory
  - No page faults in kernel
- All system calls run to completion
- Non-preemptable $\Rightarrow$ sequentially processes:
  - System calls
  - Interrupts
- Long-running ops: pushed to user space
  - See later…

USER THREAD MODELS

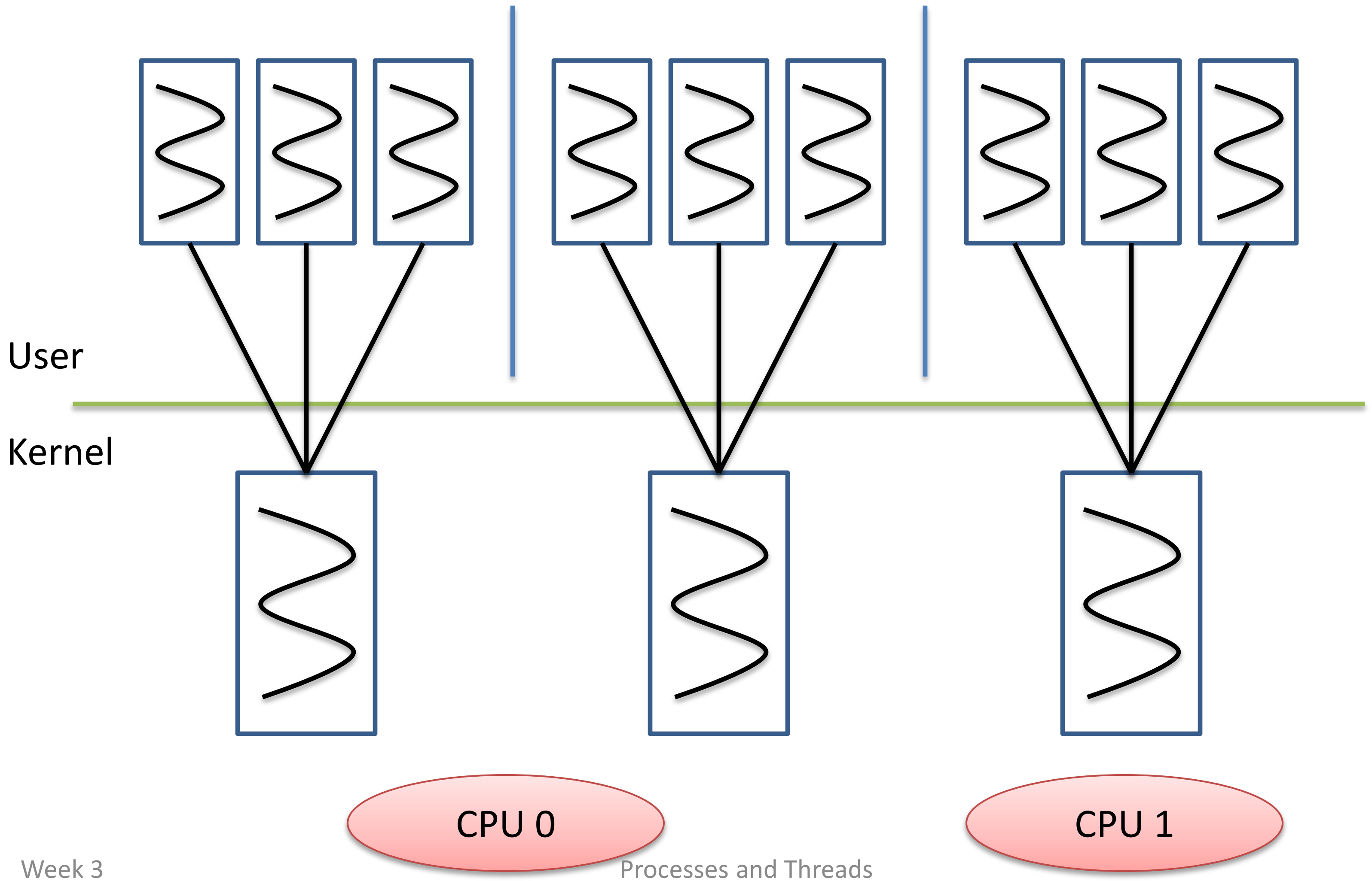
The Problem

- Threads are a programming language abstraction
  - Start lots (1000s) of blocking (e.g. IO) tasks in parallel
  - Must be lightweight
- Threads are **also** a kernel abstraction
  - *Virtual CPUs* – heavyweight abstractions
  - No real point having >real CPUs
- Threads need to communicate
  - Thread and IPC performance critical to applications

What Are The Options?

1. Multiple threads per vCPU (kernel thread)
   – Perhaps many VCPUs per process
2. One user-level thread per vCPU
   – Multiple kernel threads in a process
3. Some combination of the above
   – Multiplex user threads over kernel threads

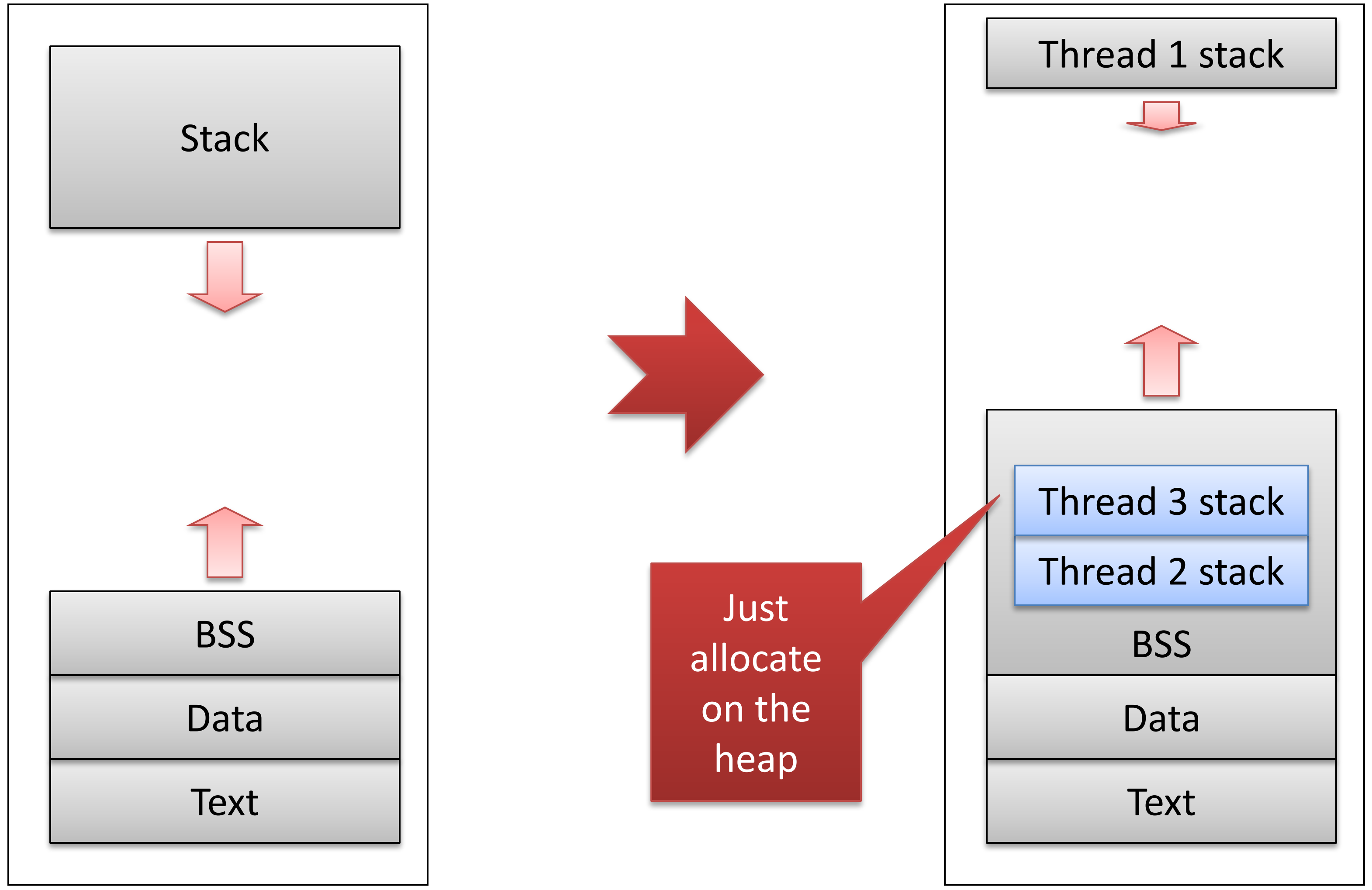
1. Many-to-One Threads



1. Many-to-One Threads

- Early thread libraries:
  - Green threads (original Java VM)
  - GNU Portable Threads
  - Standard student exercise: implement them!
- Sometimes called “pure user-level threads”
  - No kernel support required
  - Also (confusingly) “Lightweight Processes”

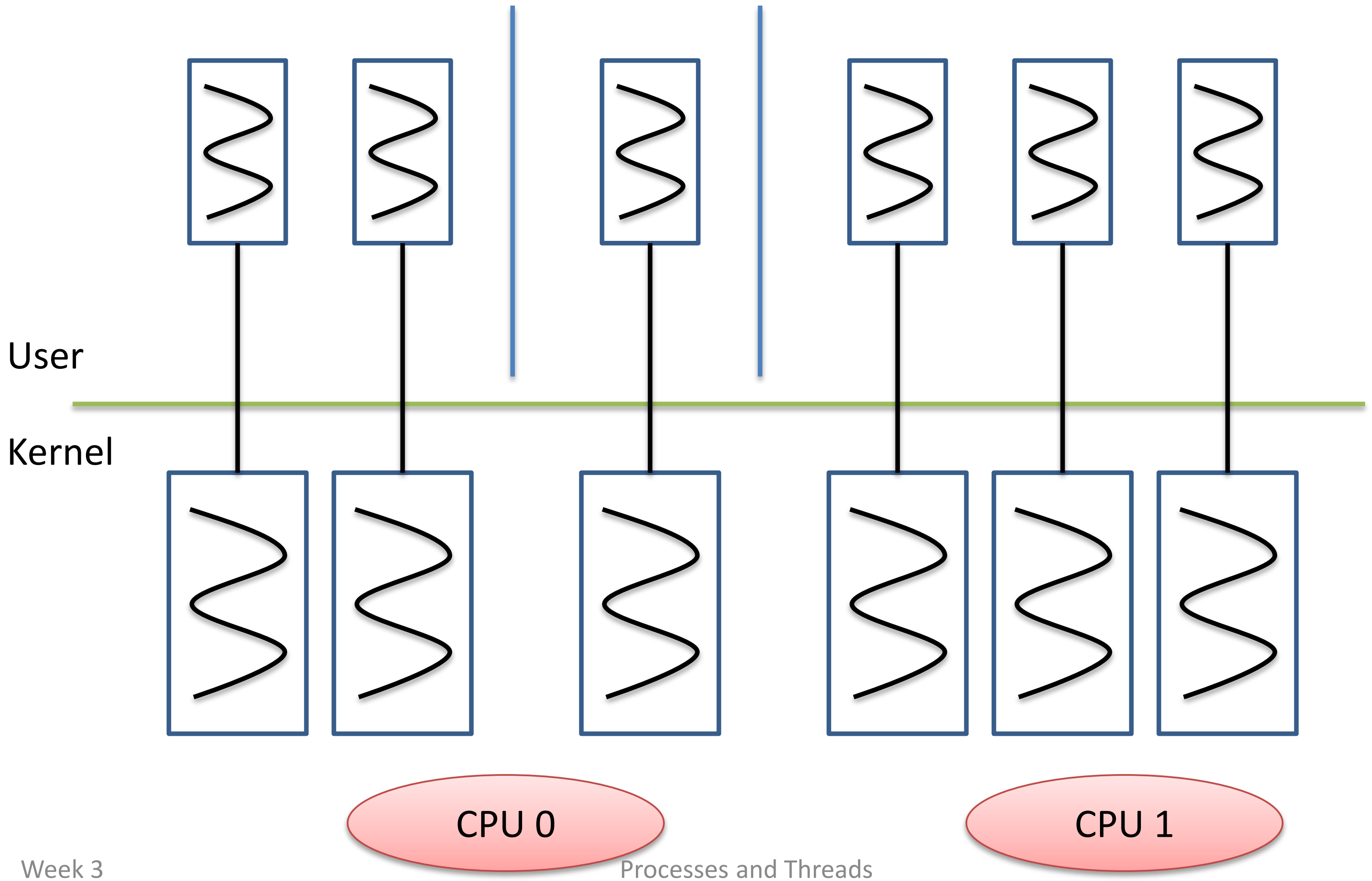
Address Space Layout for User-Level Threads



User-Level Threads
Old Unices, etc.

- High performance: 10x procedure call
- Scalable
- Flexible (application-specific)
- Treat a process as a virtual processor
- But it isn't:
  - Page faults
  - I/O
  - Multiprocessors

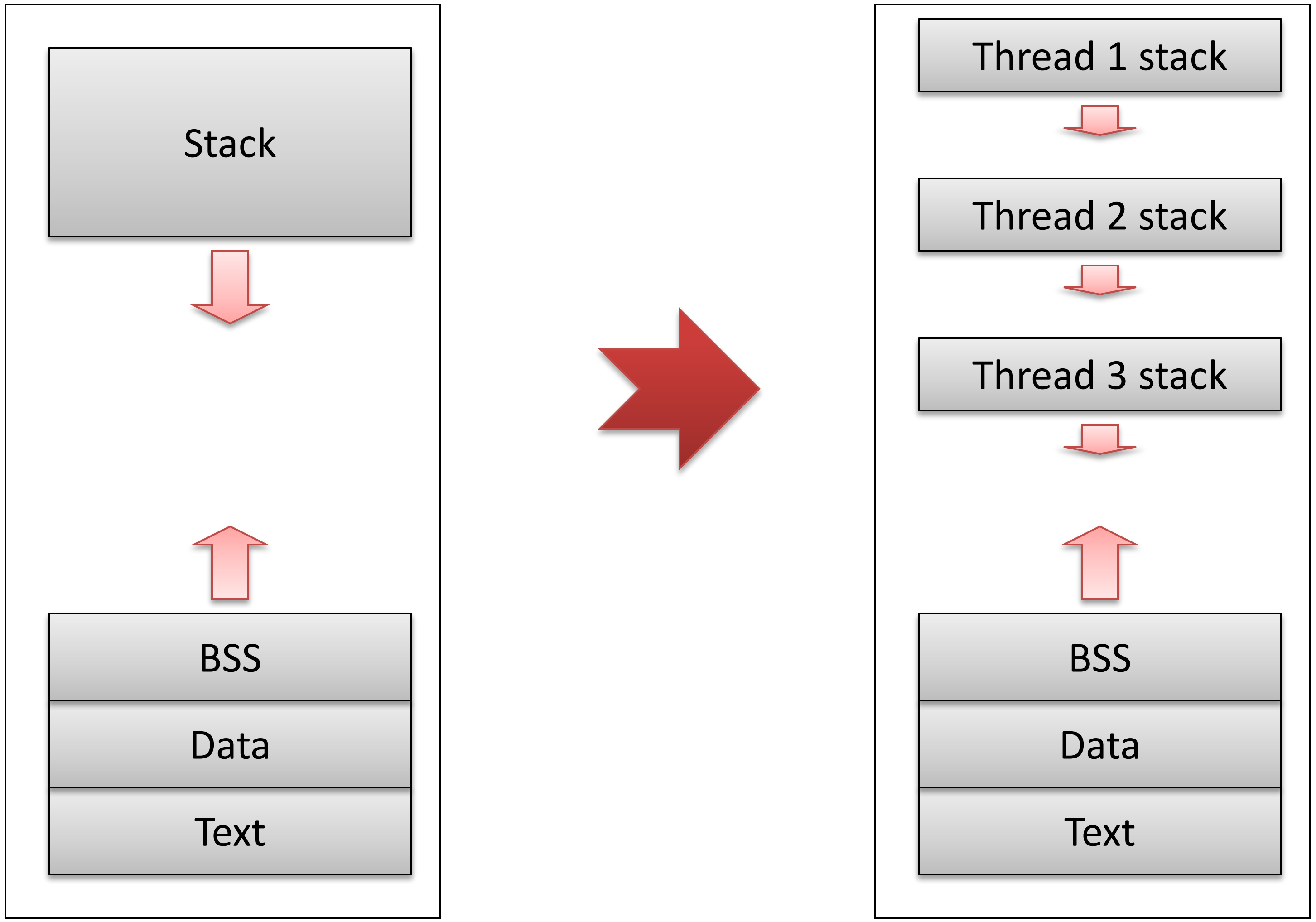
2. One-to-one threads



2. One-to-One User Threads

- Every user thread is/has a kernel thread.
- Equivalent to:
  - Multiple processes sharing an address space
  - Except that “process” is now a group of threads
- Most modern OS threads packages:
  - Linux, Solaris, Windows XP, OS X

One-to-One User Threads



Kernel Threads
Linux, Vista+, etc.

- Excellent integration with the OS
- Slow: similar to process switch time
- Inflexible
  - kernel policy
  - Evidence: people implemented user-level threads over kernel threads anyway

$\Rightarrow$ same old problems...

Comparison

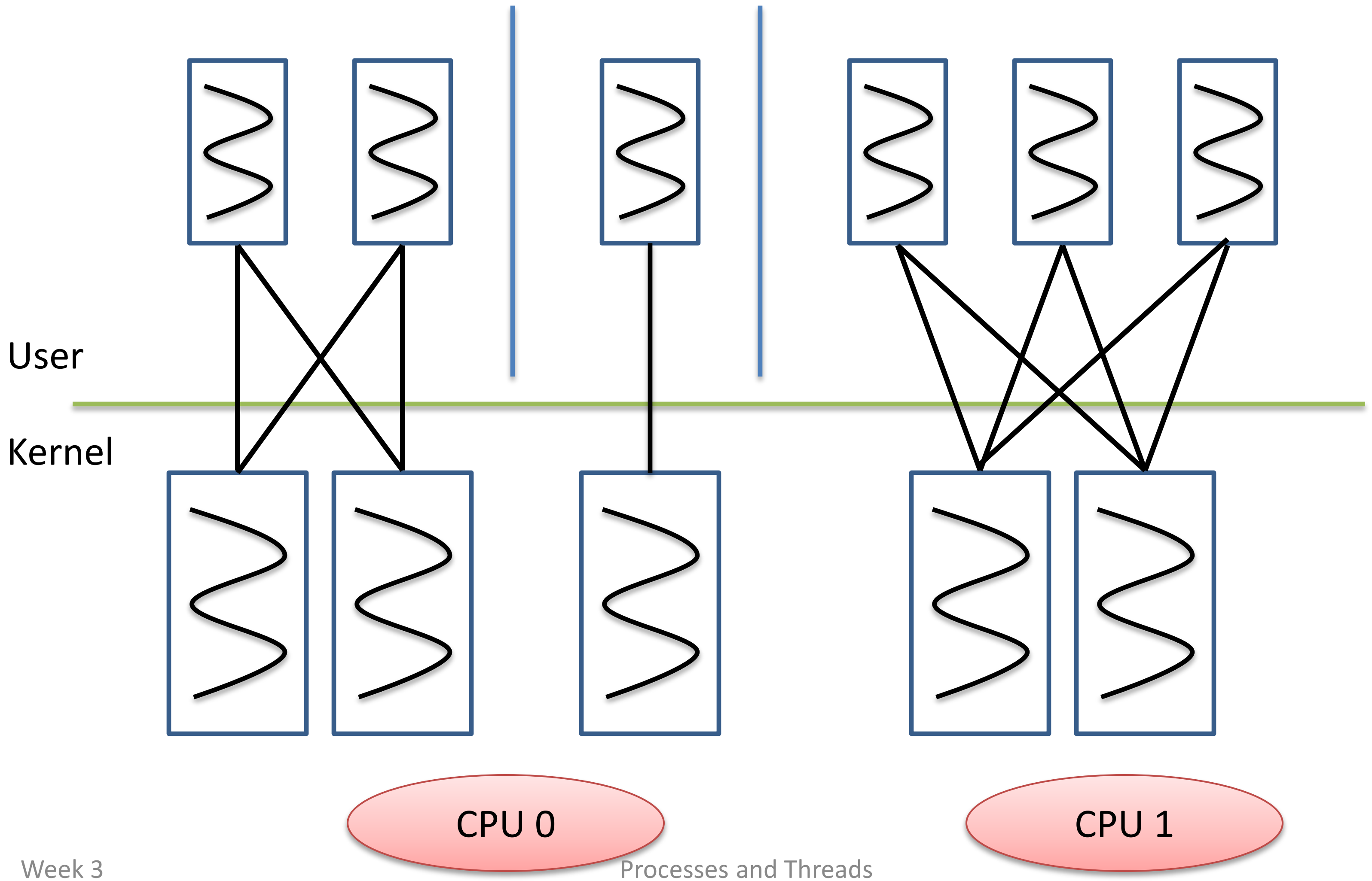
User-Level Threads

- ✓ Cheap to create and destroy
- ✓ Fast to context switch
- ✘ Block entire process
  - Not just on system calls
  - Page faults!

One-to-One Threads

- ✓ Easier to schedule
- ✓ Nicely handle blocking
- ✘ Memory (kernel stack)
- ✘ Slow to switch
  - Requires kernel crossing

3. Many-to-Many Threads



2. Many-to-Many Threads

- Multiplex user-level threads over several kernel-level threads
- Can pin a user thread to a kernel thread for performance or predictability
- Thread migration costs are “interesting”…

Not seen in Linux, but default in Windows, Solaris.

Issues with Modern Threads

- Hard for a runtime to tell:
  - How many user-level threads can run at a time (i.e. how many physical cores are allocated)
  - Which user-level threads can run (i.e. which physical cores are allocated)
- Severely limits of user-level scheduling
  - Can critically impact performance of parallel applications.

SCHEDULER ACTIVATIONS AND DISPATCH

Definitions

- Scheduling:
  - Deciding which task to run
- Dispatch:
  - How the chosen task starts (or resumes) execution

Scheduler Activations

- Mechanism: upcall to the ULS from the kernel
  - Context for this: a *scheduler activation*
- Structurally like a kernel thread but...
  - created on-demand in response to events (blocking, preemption, etc.)
- User level threads package built on top
- Hardware: DEC SRC Firefly workstation (7-processor VAX)

Scheduler Activations Speedup

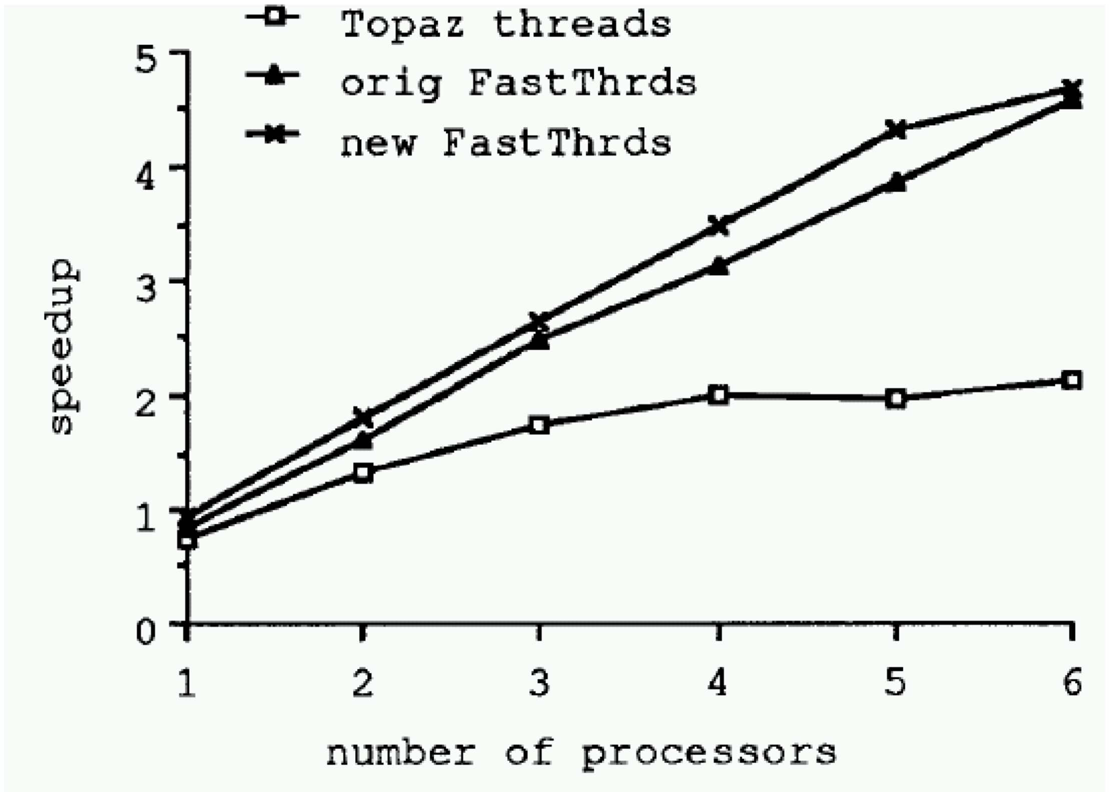


Figure 1: Speedup of N-Body Application vs. Number of Processors, 100% of Memory Available

Scheduler Activations Memory Footprint

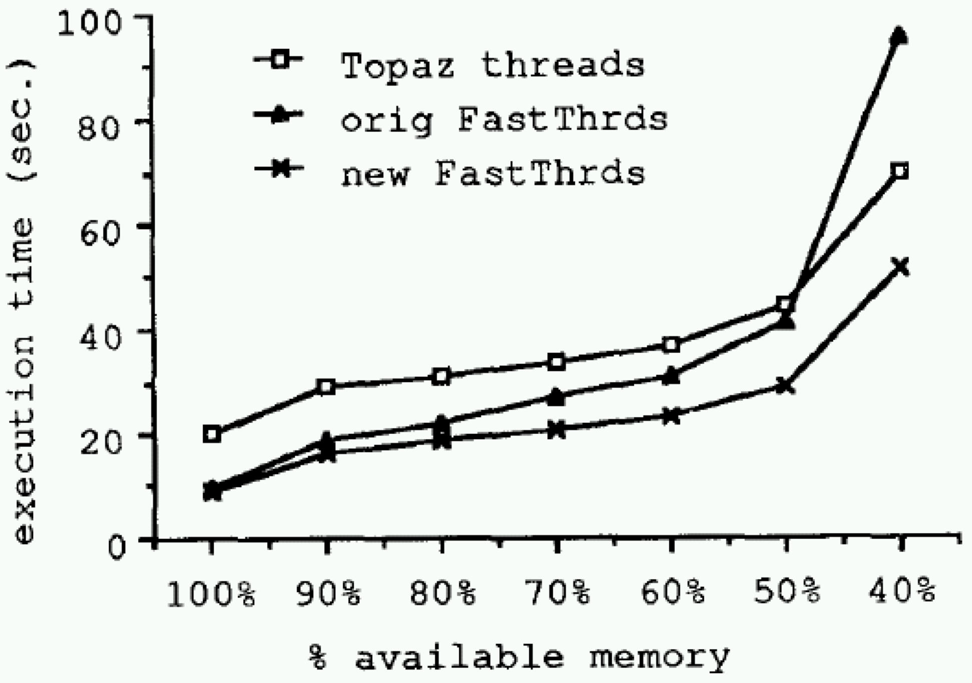
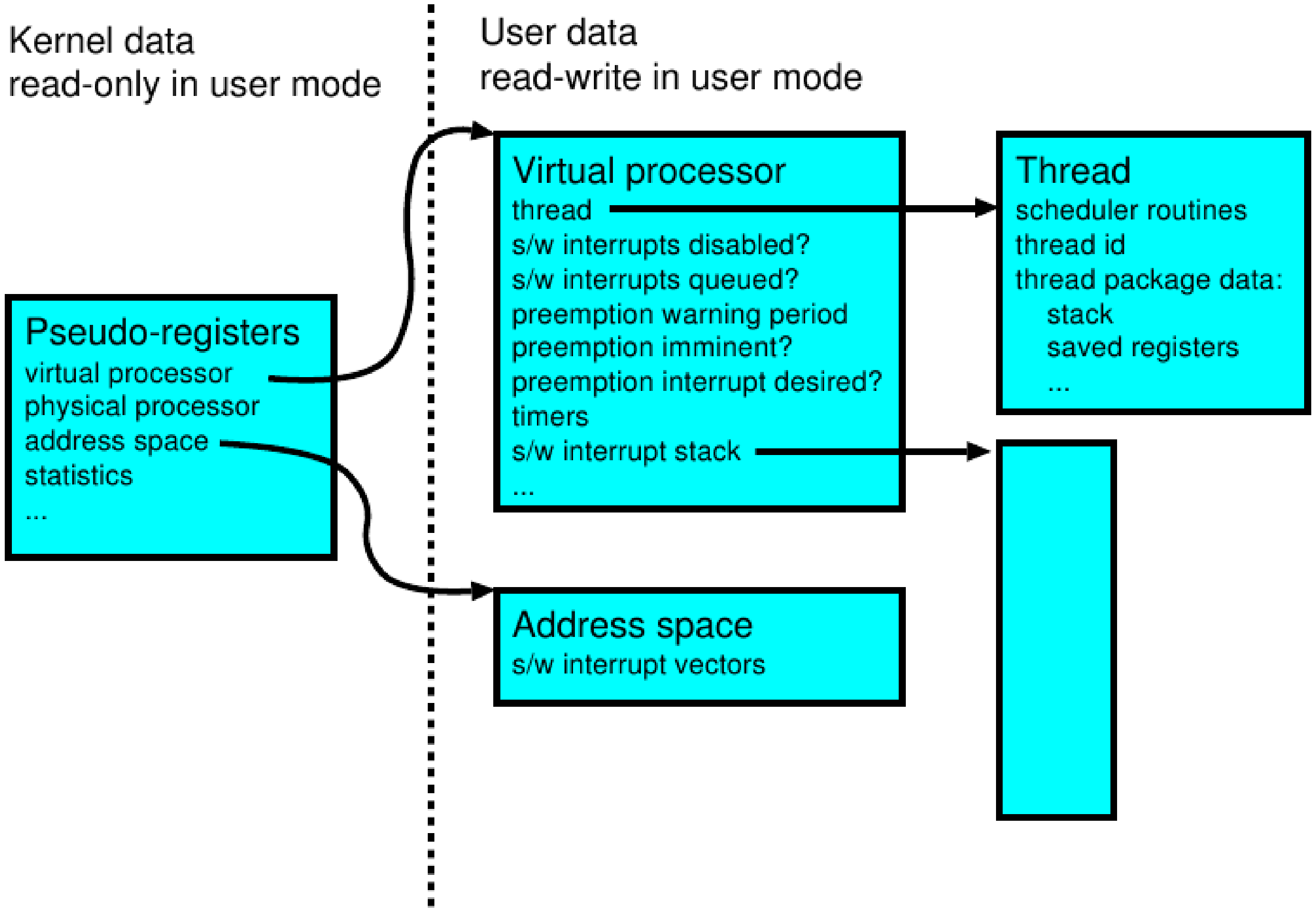


Figure 2: Execution Time of N-Body Application vs. Amount of Available Memory, 6 Processors

Psyche Threads

- Similar: Remove kernel from most thread scheduling decisions, reflect events to user space
- Kernel and ULS share data structures (RW, RO)
- Kernel upcalls ULS (“software interrupts”) in a virtual processor for:
  - Timer expiration
  - Imminent preemption (err...)
  - Start of blocking system call
  - Unblocking of a system call
- Shared data structure standardises interface for blocking/unblocking threads

Psyche Data Structures



Interesting Features of Psyche

- Threads given warning of imminent preemption
  - Is there a problem here?
- Upcalls can be nested (stack)
  - Likewise?
- Upcalls can be disabled or queued
- Lots of user space data structures to be pinned
- Unlike Scheduler Activations, doesn't handle (e.g.) page faults

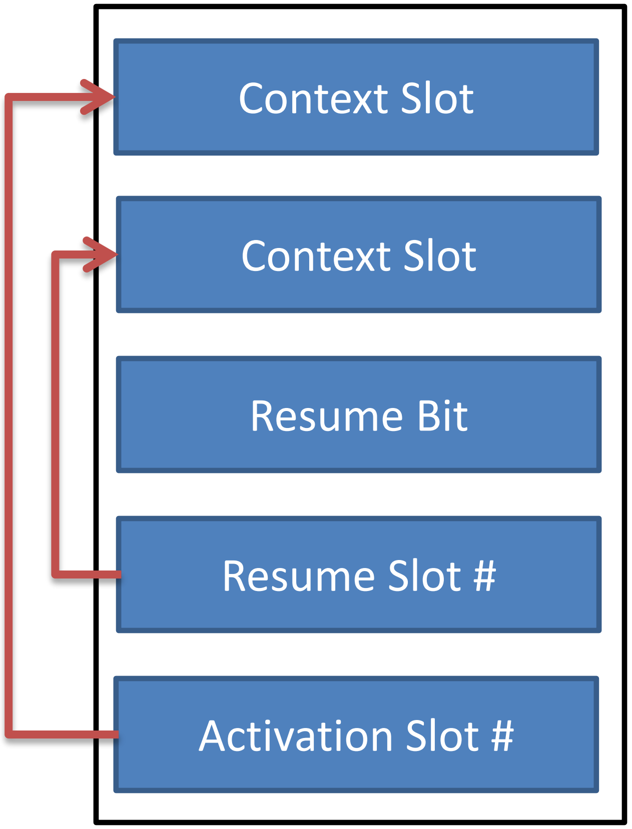
Summary: upcall-based dispatch

- Many ways to give processor to applications
- Upcalls:
  - Expose underlying scheduler decisions
  - Give full control to user-level thread scheduler
- Advantages:
  - Flexibility of policy on a single core (e.g. Nemesis)
  - Better performance across cores (e.g. Psyche)

DISPATCH IN BARRELFISH

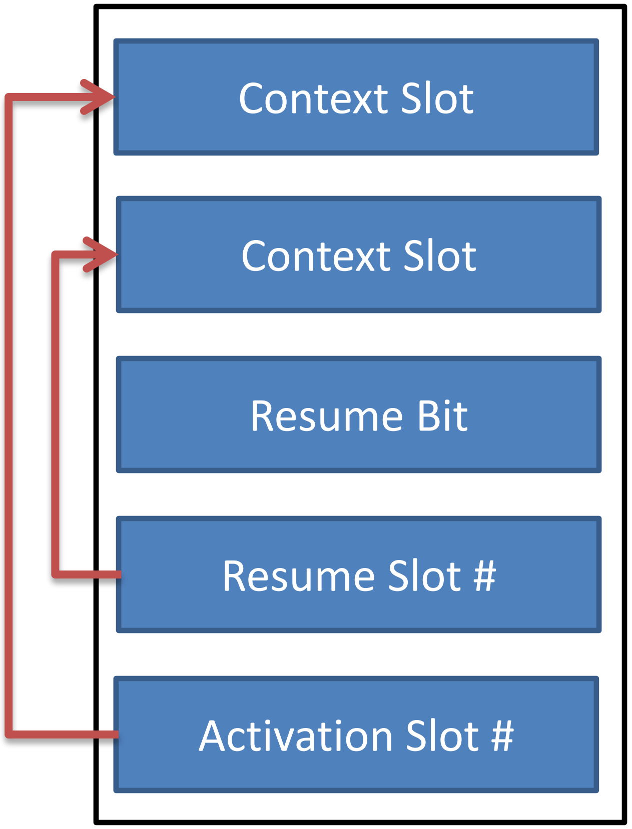
Barrelfish Dispatch
(and K42, Nemesis, …)

- No nested upcalls
  - Activation handler doesn't need to be reentrant
- Control of upcalls:
  - Per-domain data structures
  - User read/write
- Information from kernel:
  - Per-domain variables
  - User read-only



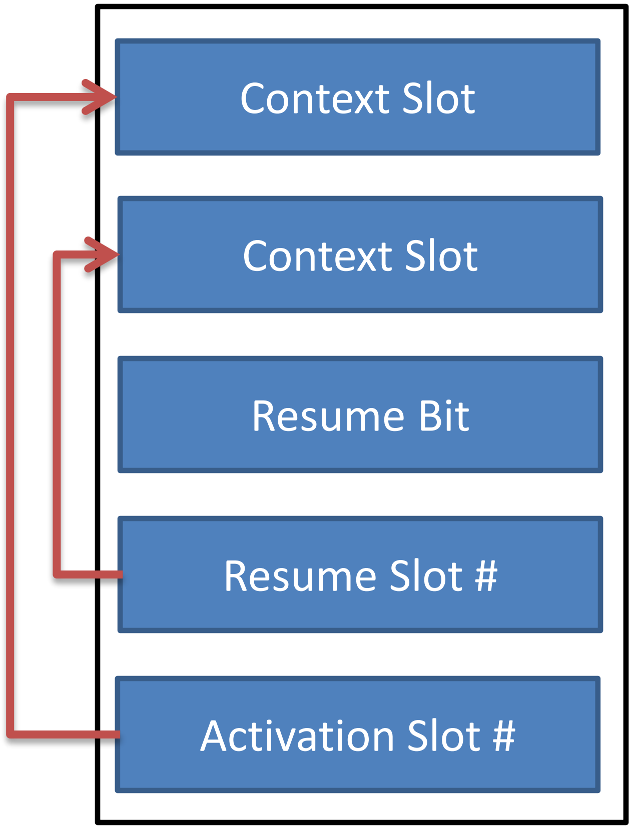
Deschedule / Preempt

```
if resume bit == 0:
  Processor state → activation slot;
else:
  Processor state → resume slot;
schedule()
```



Dispatch / Reschedule

```
if resume bit == 0:
  resume bit ← 1
  jump to activation address
else:
  processor state ← resume slot
```



Non Reentrant User-Level Schedulers

- Upcall handler gets activations on reschedule
- Resume always set on activation

  $\Rightarrow$ no need for reentrant ULS
- Picks a context slot to run from
  - Slots are a cache for thread contexts
- Clears resume bit and resumes context
- All implemented in user-level library

Challenging hack no. 1:

- How to clear resume flag and resume context
  - Race condition!

- Solution #1 (Nemesis):
  - Alpha PALmode call (2 pipeline drains)
  - Not available on ARM or x86 ☹
- Solution #2 (aka "Justin Cappos' hack"):
  - Enter known code sequence (start+end listed in DCB)
  - Clear resume bit
  - Resume context
  - Interrupt $\Rightarrow$ kernel examines flag *and* program counter

Challenging hack no. 2:

- Resuming context from user thread
  - – need to restore all registers and the PC.
- ARMv7:

```
clrex
mov     r2, #0
str     r2, [r0,#disabled_flag_offset]
ldr     r0, [r1], #4
msr     cpsr, r0
ldmia   r1, {r0-r15}
mov     r0, r0
```

On 64-bit x86:

```
mov        %[fs], %%ax
mov        %%ax, %%fs
mov        %[gs], %%ax
mov        %%ax, %%gs
movq        0*8(%[regs]), %%rax
movq        2*8(%[regs]), %%rcx
movq        3*8(%[regs]), %%rdx
movq        4*8(%[regs]), %%rsi
movq        5*8(%[regs]), %%rdi
movq        6*8(%[regs]), %%rbp
movq        8*8(%[regs]), %%r8
movq        9*8(%[regs]), %%r9
movq       10*8(%[regs]), %%r10
movq       11*8(%[regs]), %%r11
movq       12*8(%[regs]), %%r12
movq       13*8(%[regs]), %%r13
movq       14*8(%[regs]), %%r14
movq       15*8(%[regs]), %%r15
pushq      %[ss]
pushq       7*8(%[regs])
pushq      17*8(%[regs])
pushq      %[cs]
pushq      16*8(%[regs])
movq        1*8(%[regs]), %%rbx
iretq
```

Works because `iretq` is atomic and available in user space

RISC-V?

- No user-space:
  - Load multiple register instruction
  - Return from interrupt (yet…)
- So?
  - Find register which never changes (e.g. GP)
  - Do this:

```
        lw   gp, offset(tp)
        jalr gp, gp, 0
    gp: <global data>
```

Hack attributed to Andrew Waterman at UC Berkeley

64-bit ARM (v8)?

- Simple answer: you can’t
  - Requires a kernel trap, *we think*
  - Bonus points: prove us (and ARM Ltd.) wrong!

- Moral:
  - Processor architecture is often unconsciously influenced by OS orthodoxy
  - Not always in a good way.

FURTHER READING

Further reading

- Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. 1991. First-class user-level threads. In Proceedings of the thirteenth ACM symposium on Operating systems principles (SOSP '91). ACM, New York, NY, USA, 110-121. http://dx.doi.org/10.1145/121132.344329
- Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1992. Scheduler activations: effective kernel support for the user-level management of parallelism. ACM Trans. Comput. Syst. 10, 1 (February 1992), 53-79. http://dx.doi.org/10.1145/146941.146944
- I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. 1996. The design and implementation of an operating system to support distributed multimedia applications. IEEE J.Sel. A. Commun. 14, 7 (September 1996), 1280-1297. http://dx.doi.org/10.1109/49.536480
- Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. 1991. Using continuations to implement thread management and communication in operating systems. In Proceedings of the thirteenth ACM symposium on Operating systems principles (SOSP '91). ACM, New York, NY, USA, 122-136. http://dx.doi.org/10.1145/121132.121155
- Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. 1999. Interface and Execution models in the Fluke kernel. In Proceedings of the third symposium on Pperating systems design and implementation (OSDI '99). USENIX Association, Berkeley, CA, USA, 101-115. https://www.usenix.org/legacy/publications/library/proceedings/osdi99/lepreau.html