Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

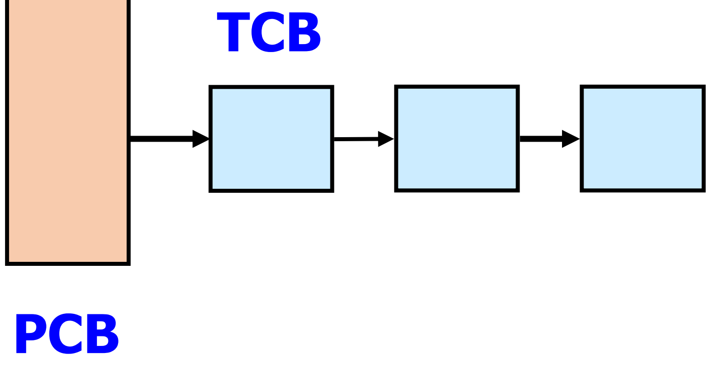
Fall 2021

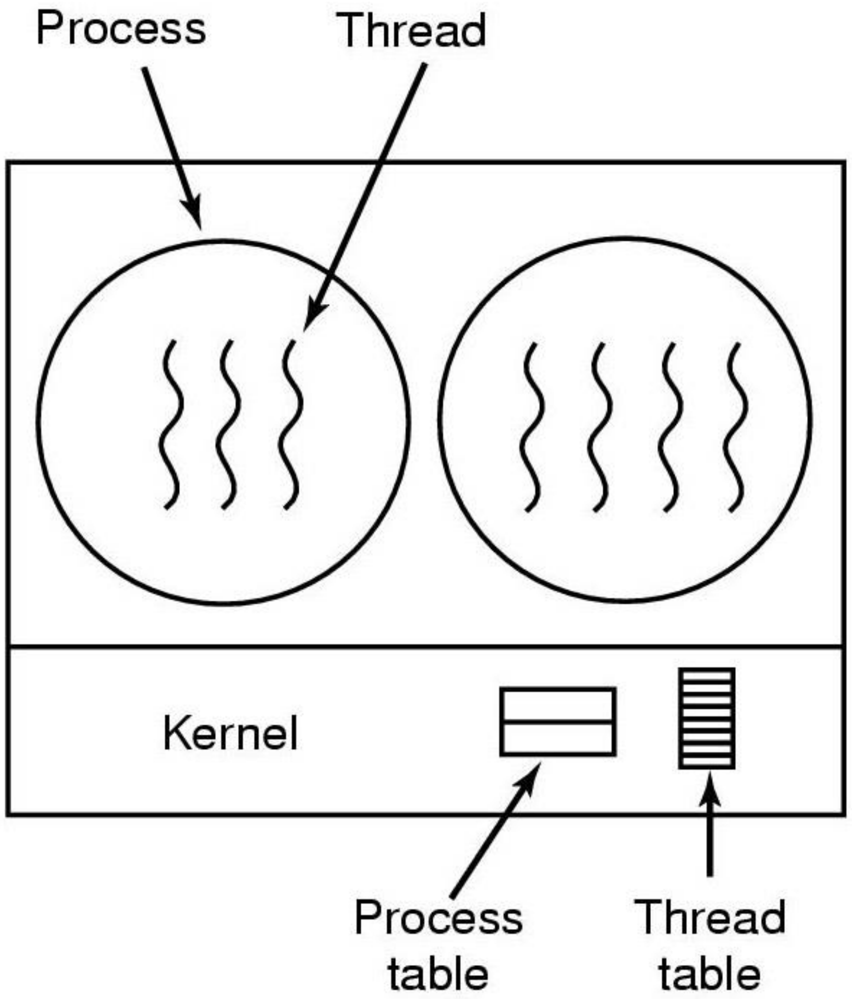
Scheduler Activations

(Thomas Anderson et al., TOCS '92)

Kernel-Level Threads: Implementation

- Every thread operations are system calls
- 1:1 model



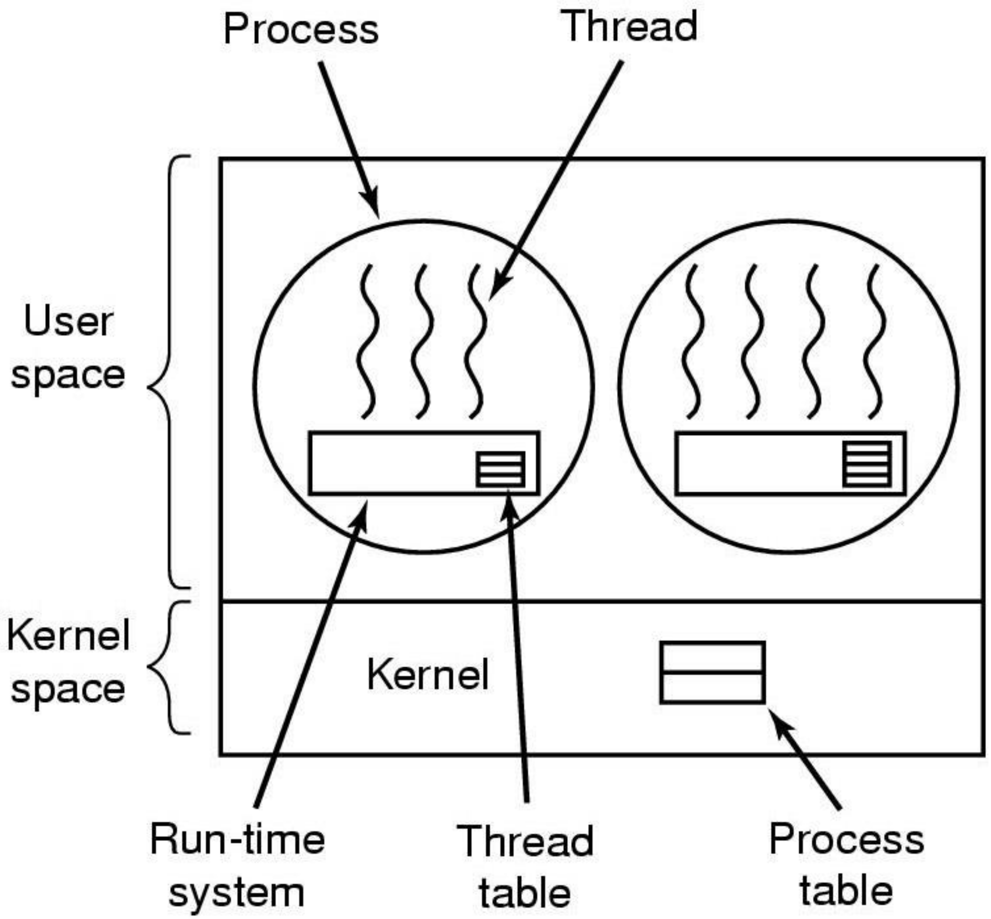


Kernel-Level Threads

- Pros
  - Cheaper than ___________
  - Scheduling/management done by the __________
    - Possible to overlap I/O with the computation
    - Multiple CPUs can be exploited
- Cons
  - Still too expensive (compared to ______________)
  - Thread state in the _________
  - Need to be general to support the needs of all programmers, languages, runtimes, etc.

User-Level Threads: Implementation

- Managed by ______________
- Views each ___________ as a "virtual processor"
- Thread management operations require no kernel intervention
- N:1 model



User-Level Threads

- Pros
  - Fast
  - Portable
  - Flexible

Operation	FastThreads (User-level)	Topaz threads (Kernel-level)	Ultrix processes
Null Fork	34µs	948µs	11300µs
Signal-Wait	37µs	441µs	1840µs

- Cons
  - Invisible to OS; OS can make poor decisions
  - Cannot exploit multiple CPUs

Goals

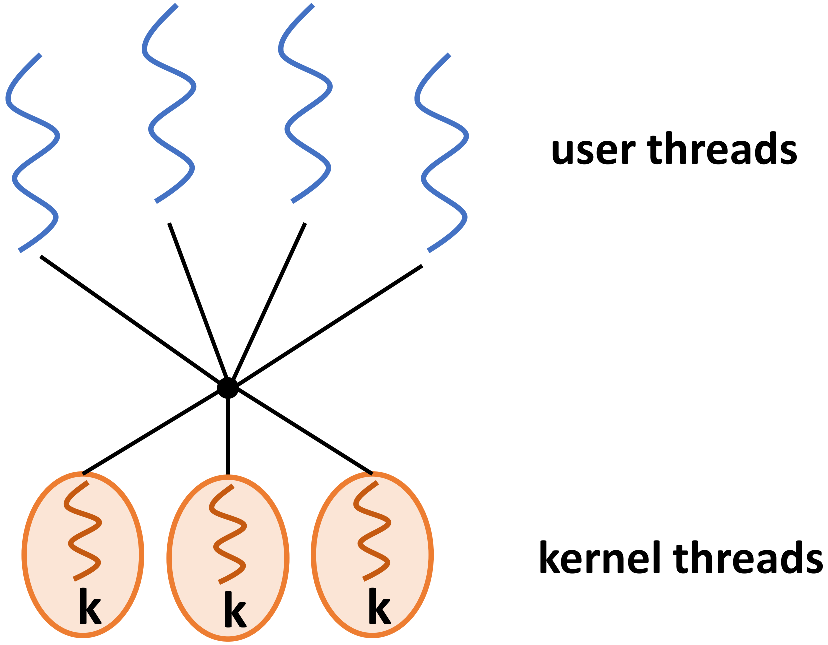
- The performance and flexibility of user-level threads
  - The performance of user-level thread systems in the common case
  - Simplify application-specific customization:
    e.g., scheduling policy, concurrency models, etc.

- The functionality of kernel threads
  - No processor idles in the presence of ready threads
  - No high-priority thread waits for a processor while a low-priority thread runs
  - When a thread traps to the kernel to block, the processor can be used to run another thread from the same or from a different address space

A Simple Solution

- How about to provide multiple kernel threads to a user-level thread system?
  - M:N model

- ***What's bad?***



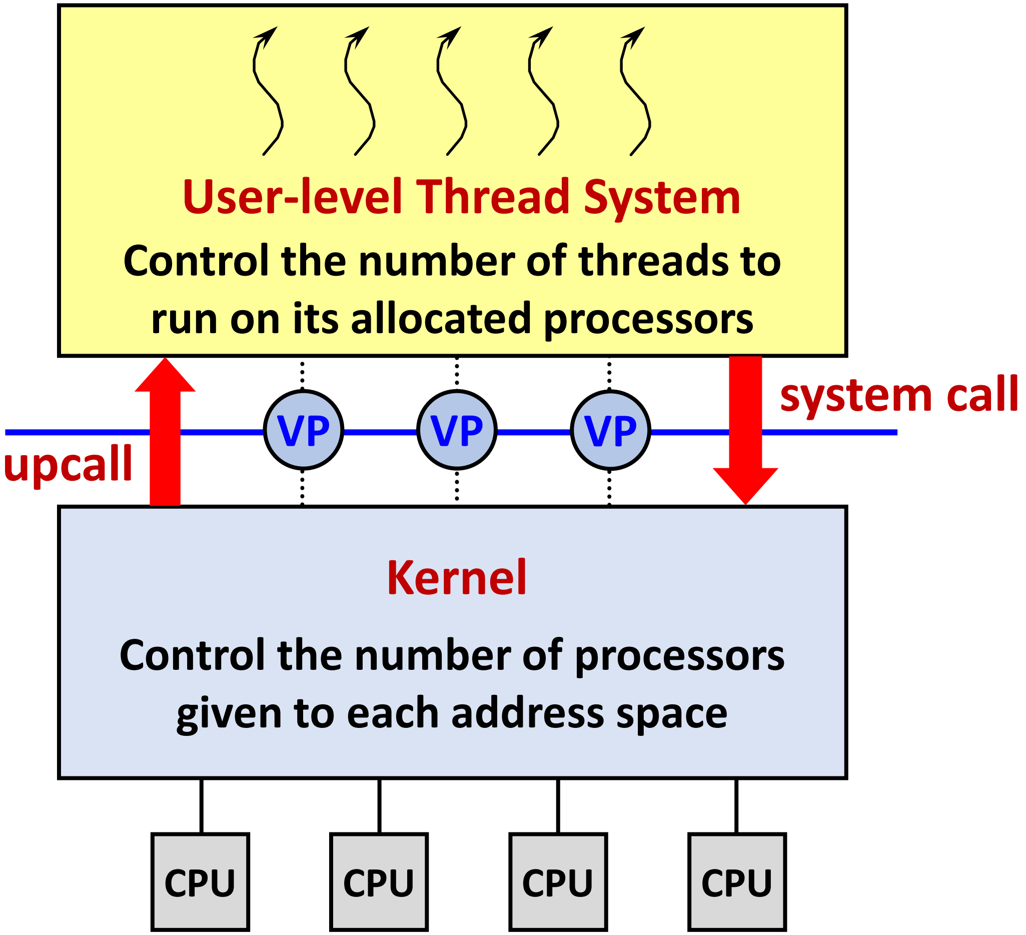
Observations

- Kernel threads are the wrong abstraction for supporting user-level thread management
- The kernel needs access to user-level scheduling information
- The user-level thread system needs to be aware of kernel events

Scheduler Activation

- Serves as a vessel, or execution context, for running user-level threads (an extension of a kernel thread)
- Notifies the user-level thread system of a kernel event via ________
- Requires two stacks:
  - A kernel-level stack: used during ___________
  - A user-level stack: used during __________
  - Note: Each user-level thread has its own stack
- Activation control block
  - For saving the processor context of the activation's current user-level thread, when the thread is stopped by the kernel

Scheduler Activation: Overview



User-level Thread System

Control the number of threads to run on its allocated processors

Kernel

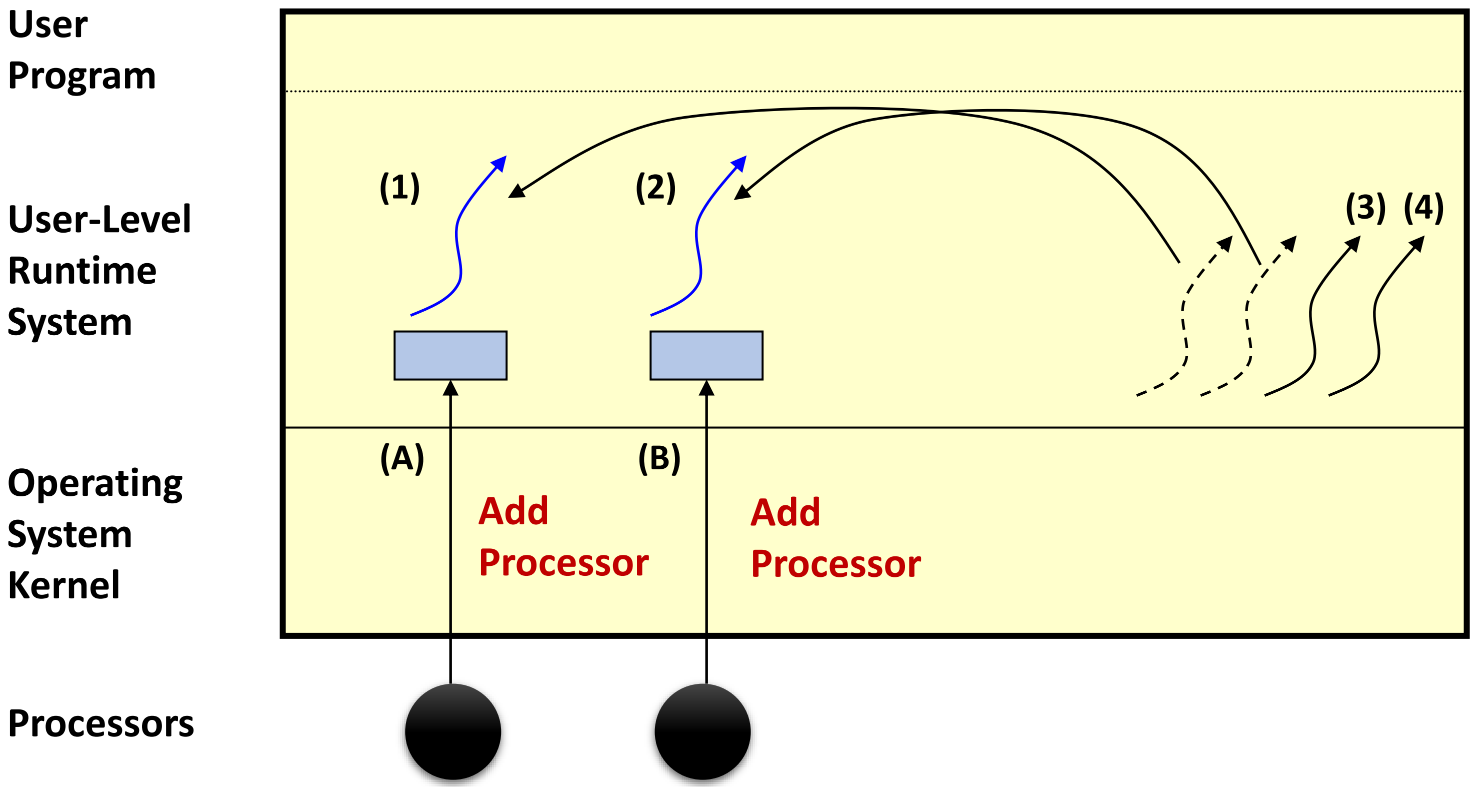
Control the number of processors given to each address space

Notifies the user-level thread system whenever the kernel changes the number of processors assigned to it

Notifies the kernel when the application needs more or fewer processors

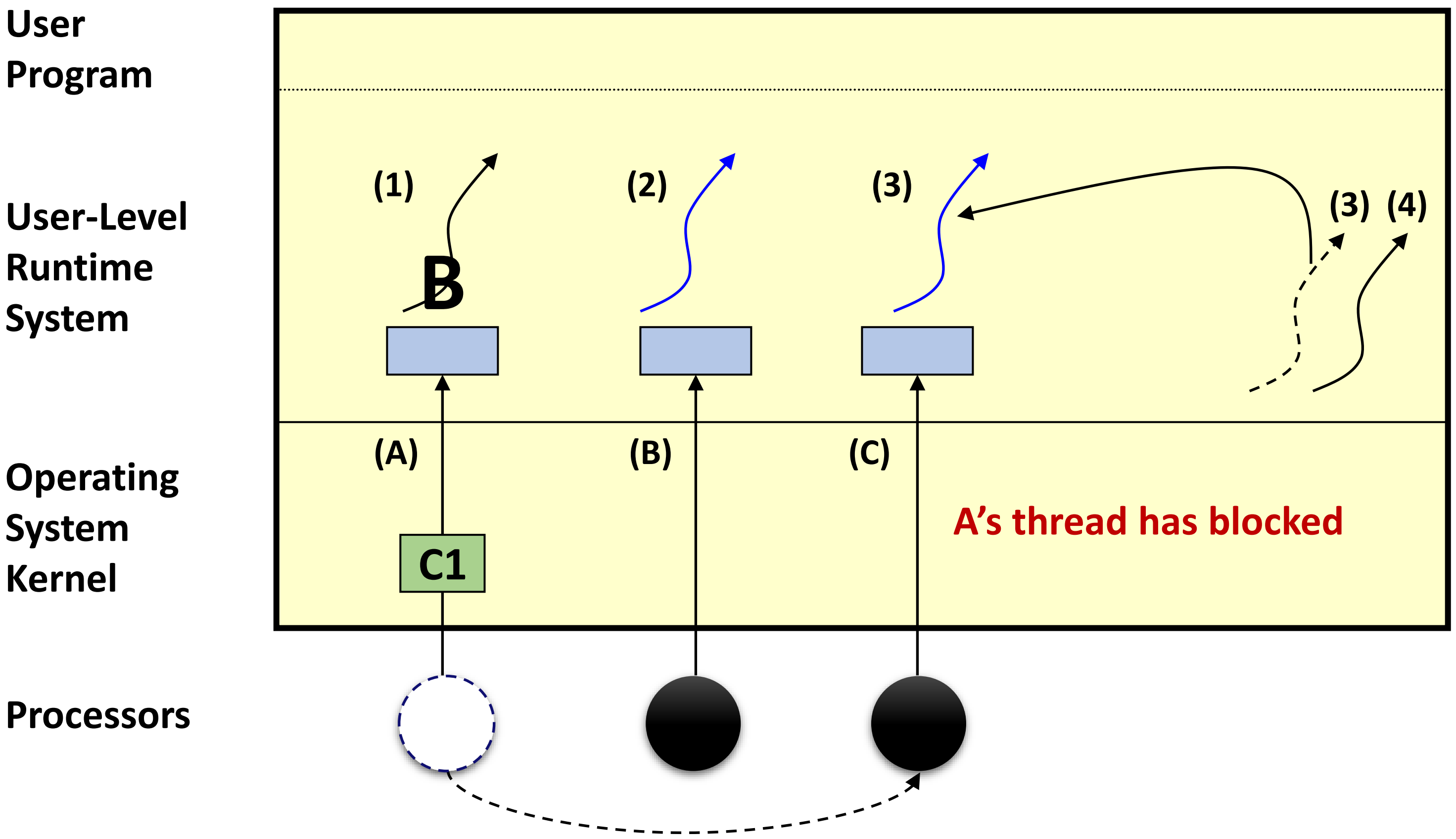
Example:

- T1: The kernel allocates two processors



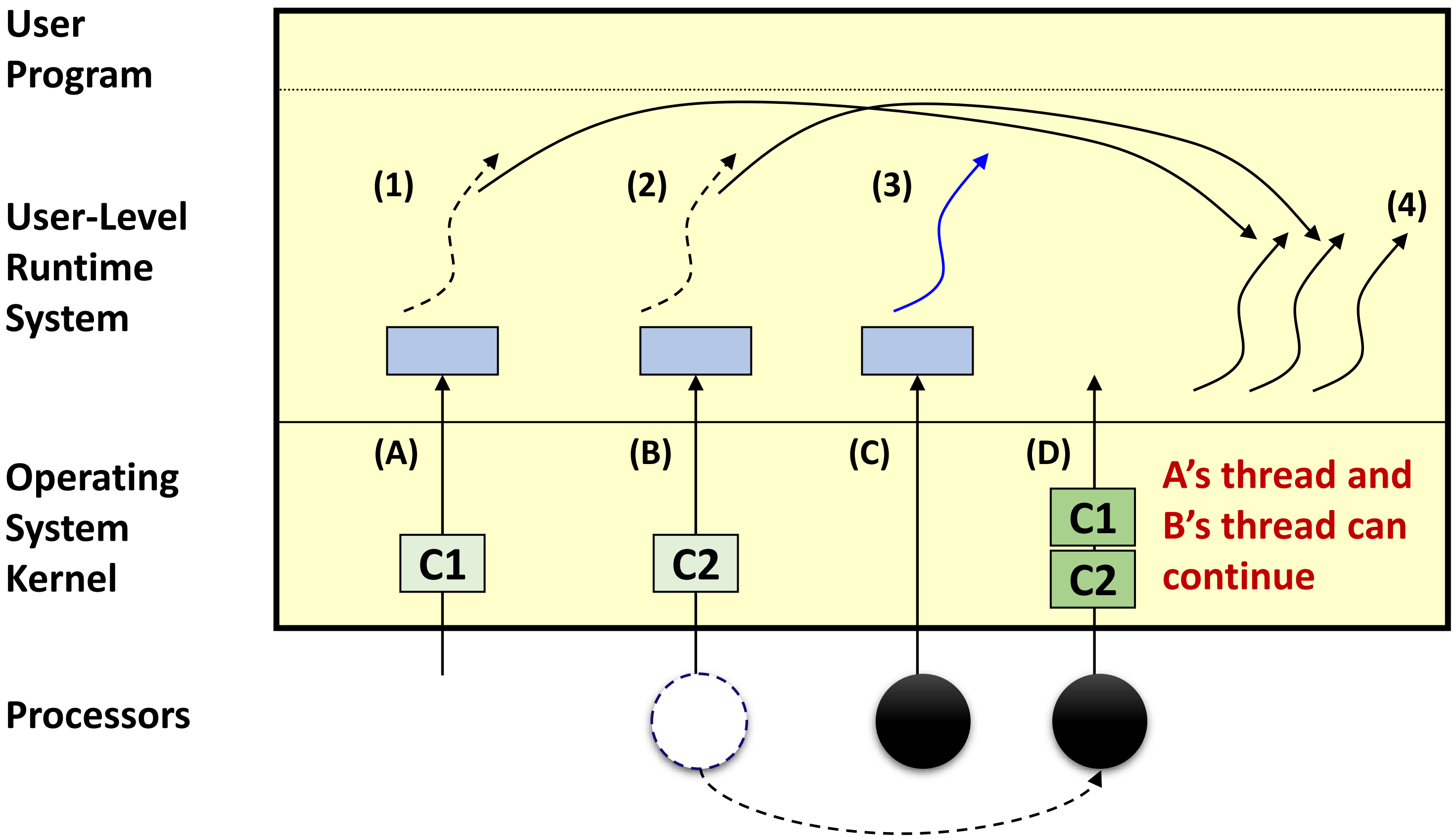
Example:

- T2: Thread 1 blocks in the kernel for I/O



Example:

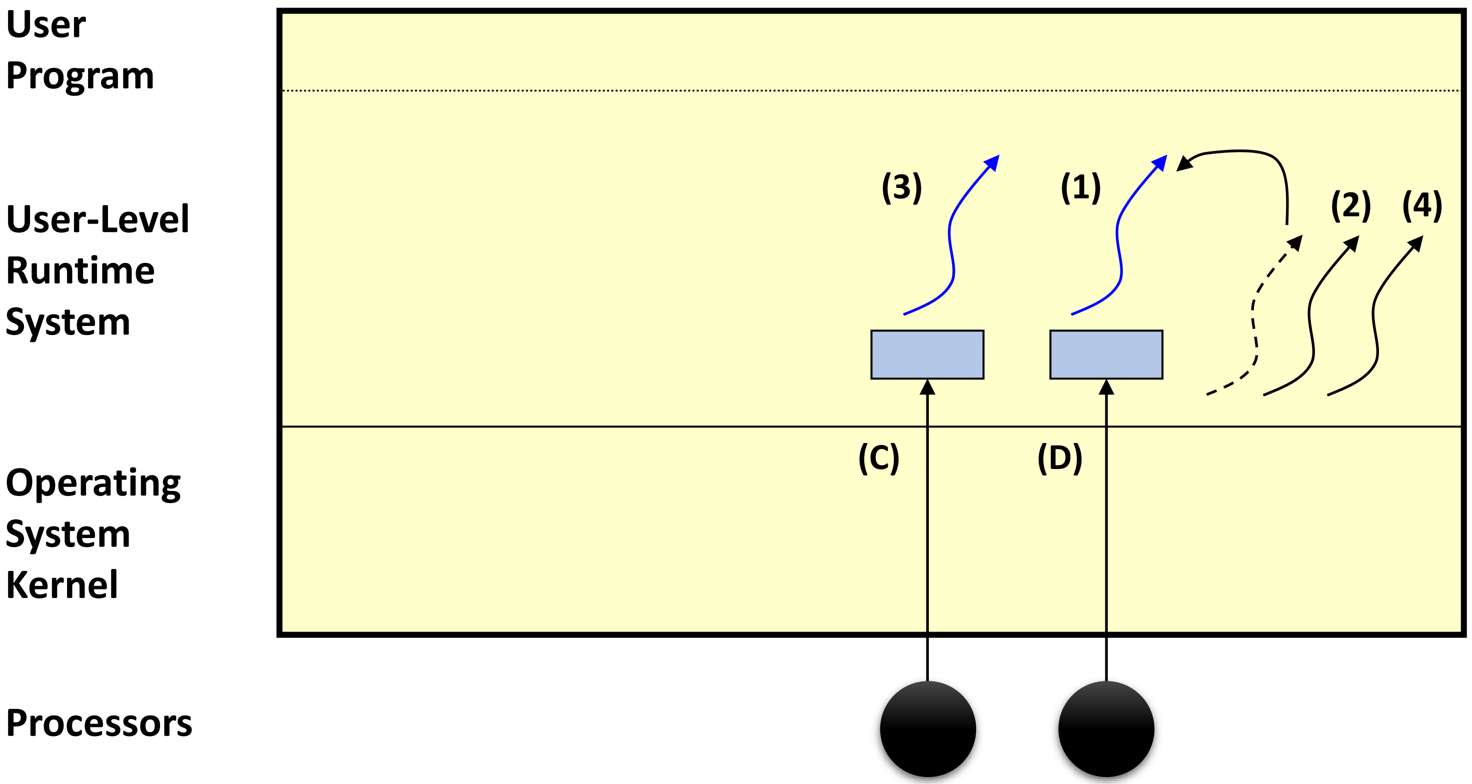
- T3: Thread 1 completes the I/O



A's thread and B's thread can continue

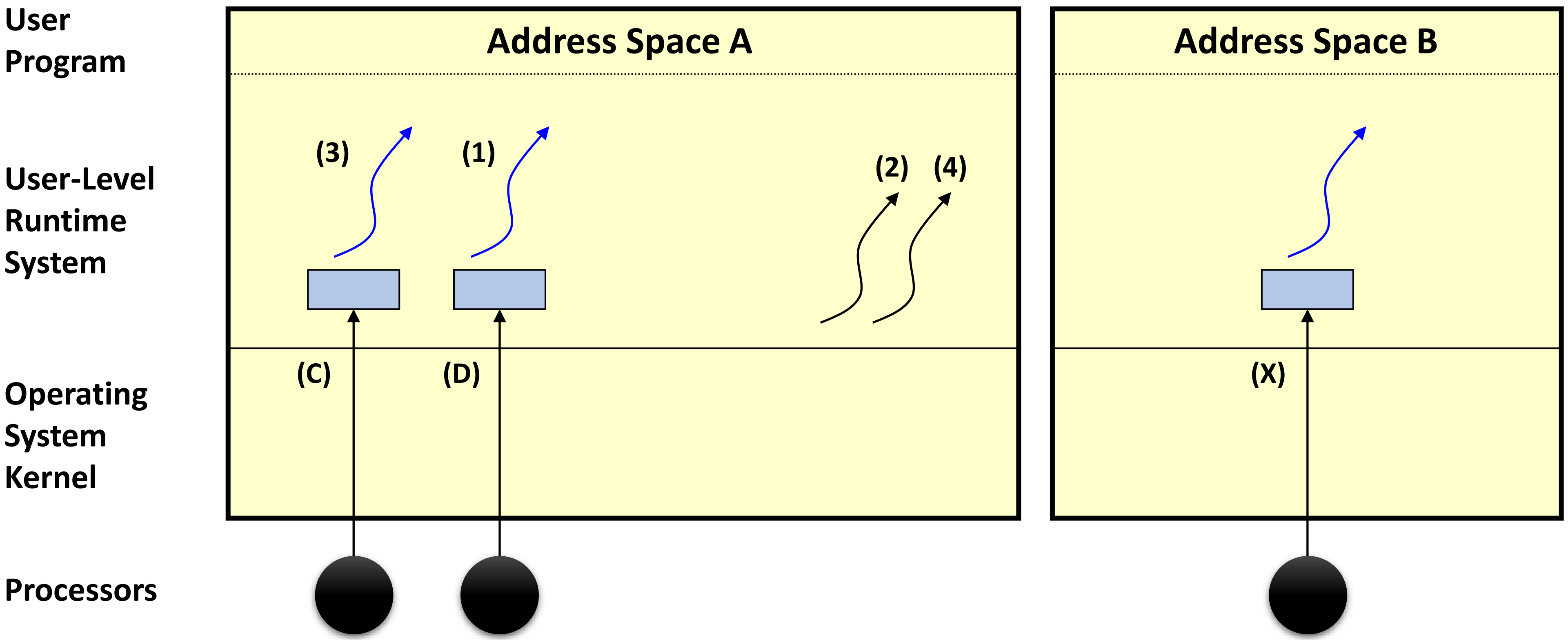
Example:

- T4: Thread 1 resumes



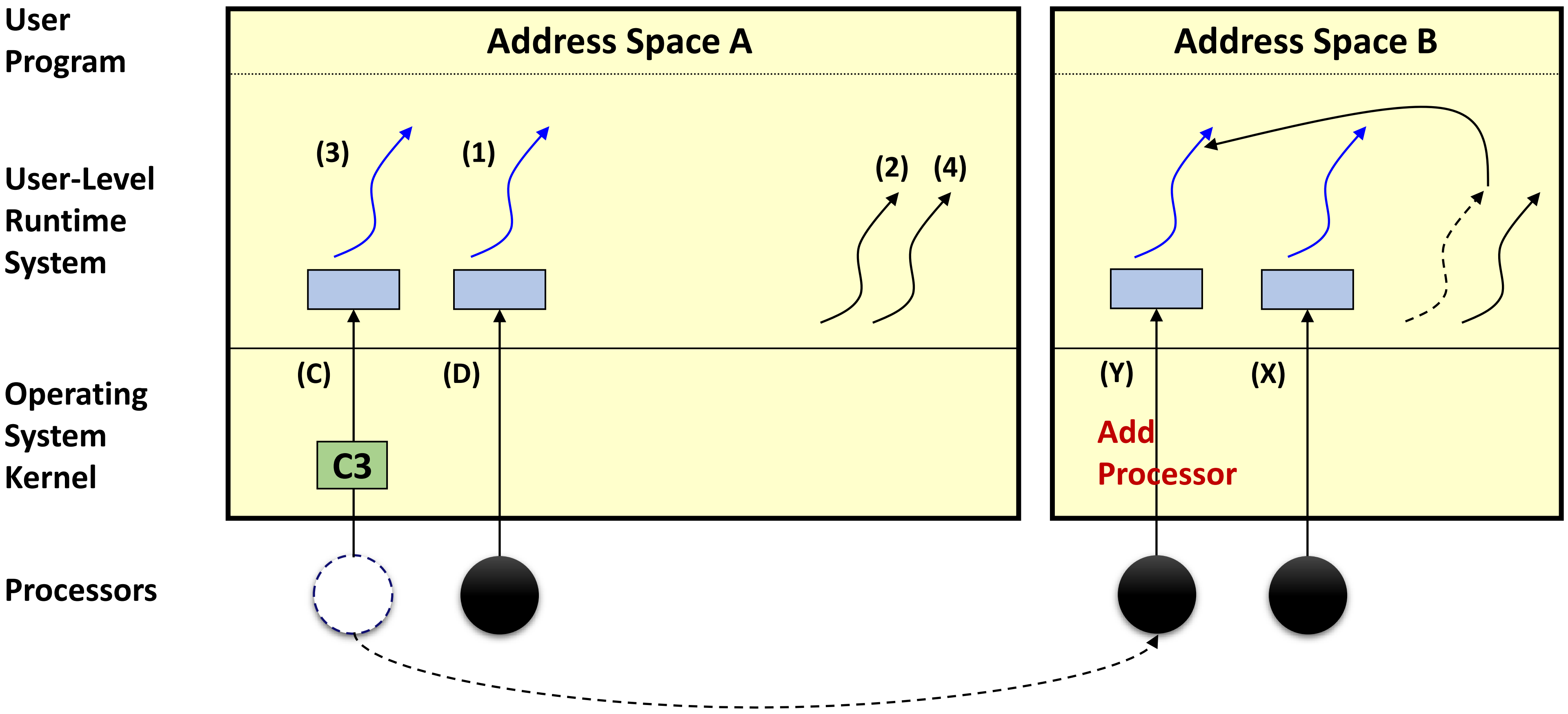
Example:

- T5: Kernel wants to take a processor away from address space A



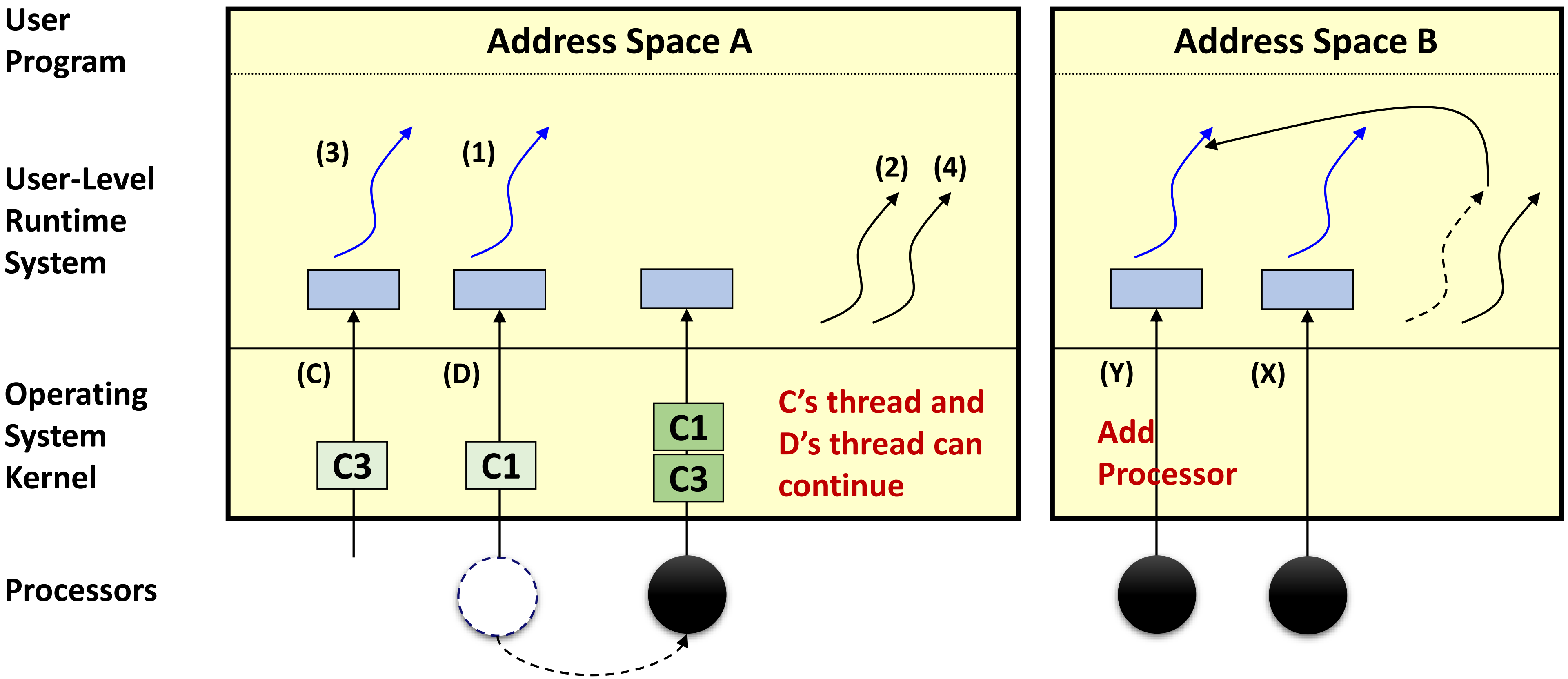
Example:

- T6: Thread 3 is preempted and the processor is allocated to B



Example:

- T7: Thread 1 is preempted and kernel notifies to A



Upcall Points: Kernel → User

- Add this processor (processor #)
  - Execute a runnable user-level thread
- Processor has been preempted (preempted SA# and its machine state)
  - Return to the ready list the user-level thread that was executing in the context of the preempted SA
- Scheduler activation has blocked (blocked SA#)
  - The blocked SA is no longer using its processor
- Scheduler activation has unblocked (unblocked SA# and its machine state)
  - Return to the ready list the user-level thread that was executing in the context of the blocked SA

Processor Allocation/Release

- An address space gives hints
  - It has more unable threads than processors, or
  - It has more processors than runnable thread
  - Only hints: processor allocation is not guaranteed

- Idle processors may be left in the address space to avoid the overhead of processor reallocation

- Dishonest or misbehaved programs?

System Call Points: User → Kernel

- Add more processors (additional # of processors)
  - Allocate more processors to this address space and start them running SAs

- This processor is idle ()
  - Preempt this processor if another address space needs it

- The user-level thread system need not tell the kernel about every thread operation

Critical Sections

- What if the preempted or blocked thread is in the critical section?
  - Poor performance or deadlock
- Solution based on "recovery":
  - Check whether the preempted thread was in the critical section (How?)
  - If so, it is continued temporarily via a user-level context switch
- Performance enhancements
  - Make a copy of each critical section
  - Runtime checks using the section begin/end addresses
  - Normal execution uses the original version
  - The copy returns to the scheduler at the end of the critical section
  - Imposes no overhead in the common case!

Basic Performance

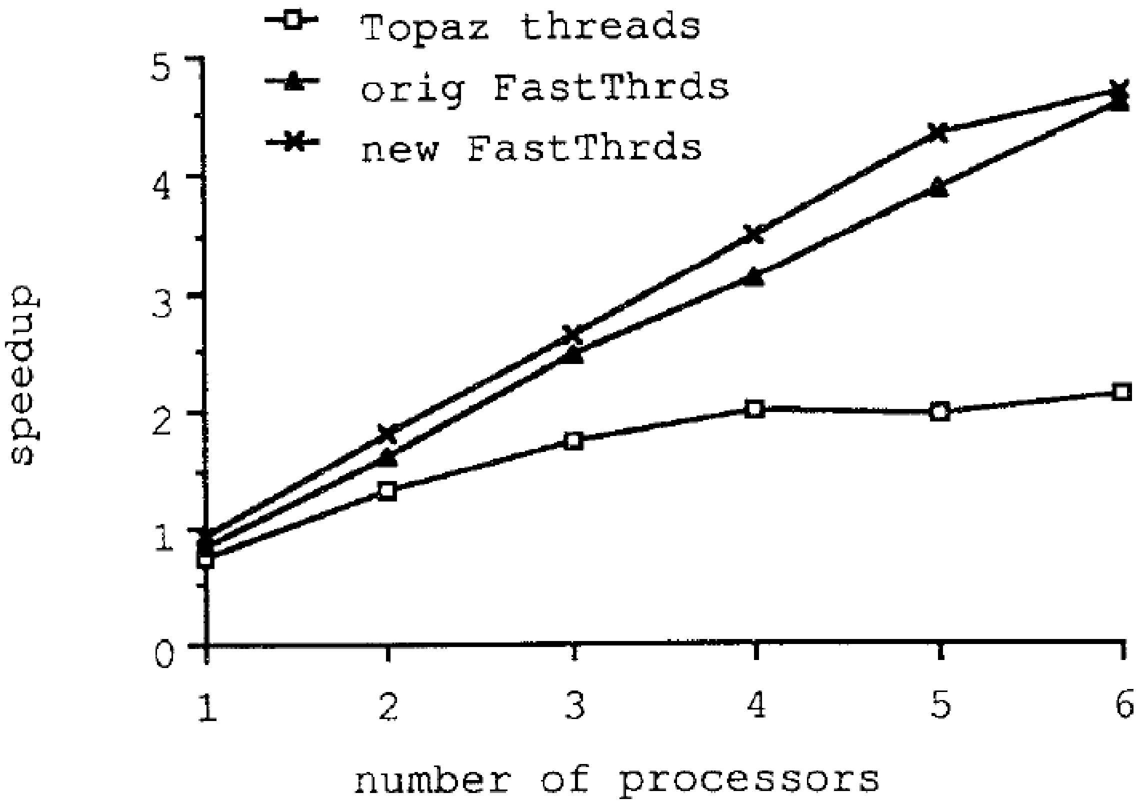
- Thread operation latencies

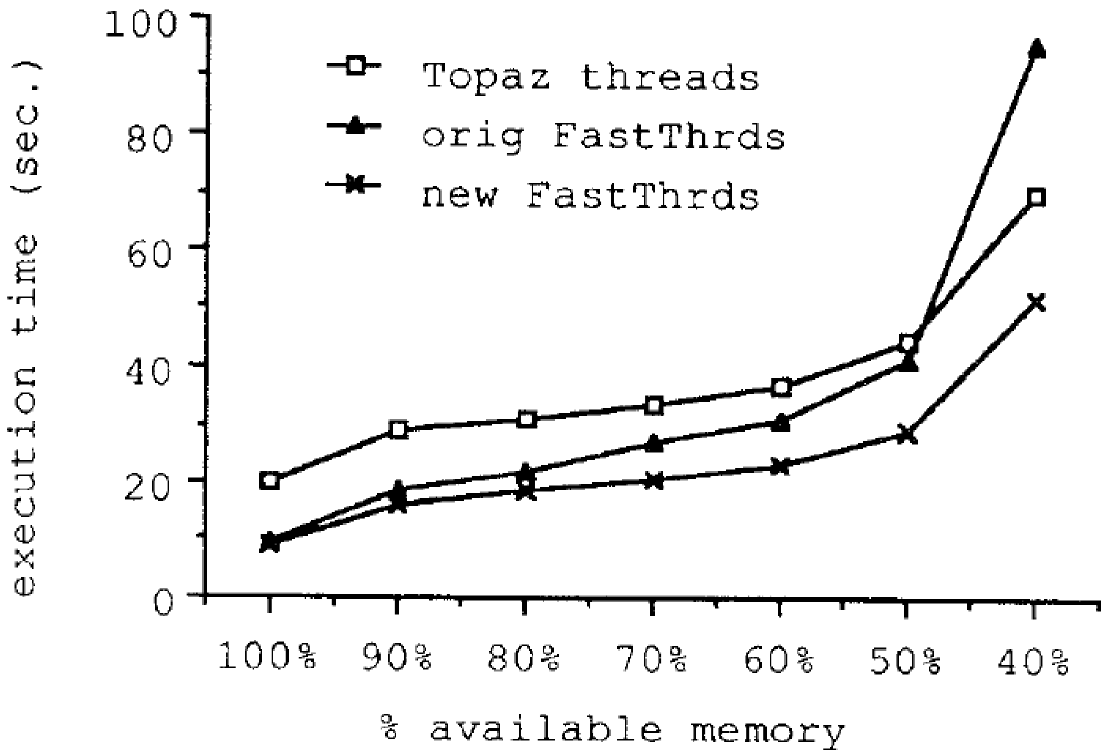
Operation	FastThreads on Topaz threads	FastThreads on Scheduler Activations	Topaz threads (Kernel-level)	Ultrix processes
Null Fork	34µs	37µs	948µs	11300µs
Signal-Wait	37µs	42µs	441µs	1840µs

- Upcall performance: Signal-Wait through the kernel
  - 5x slower than Topaz threads!
  - Quick modification
  - Modular-2+ vs. assembly

Application Performance

- N-Body (memory-intensive) on 6-processor CVAX Firefly





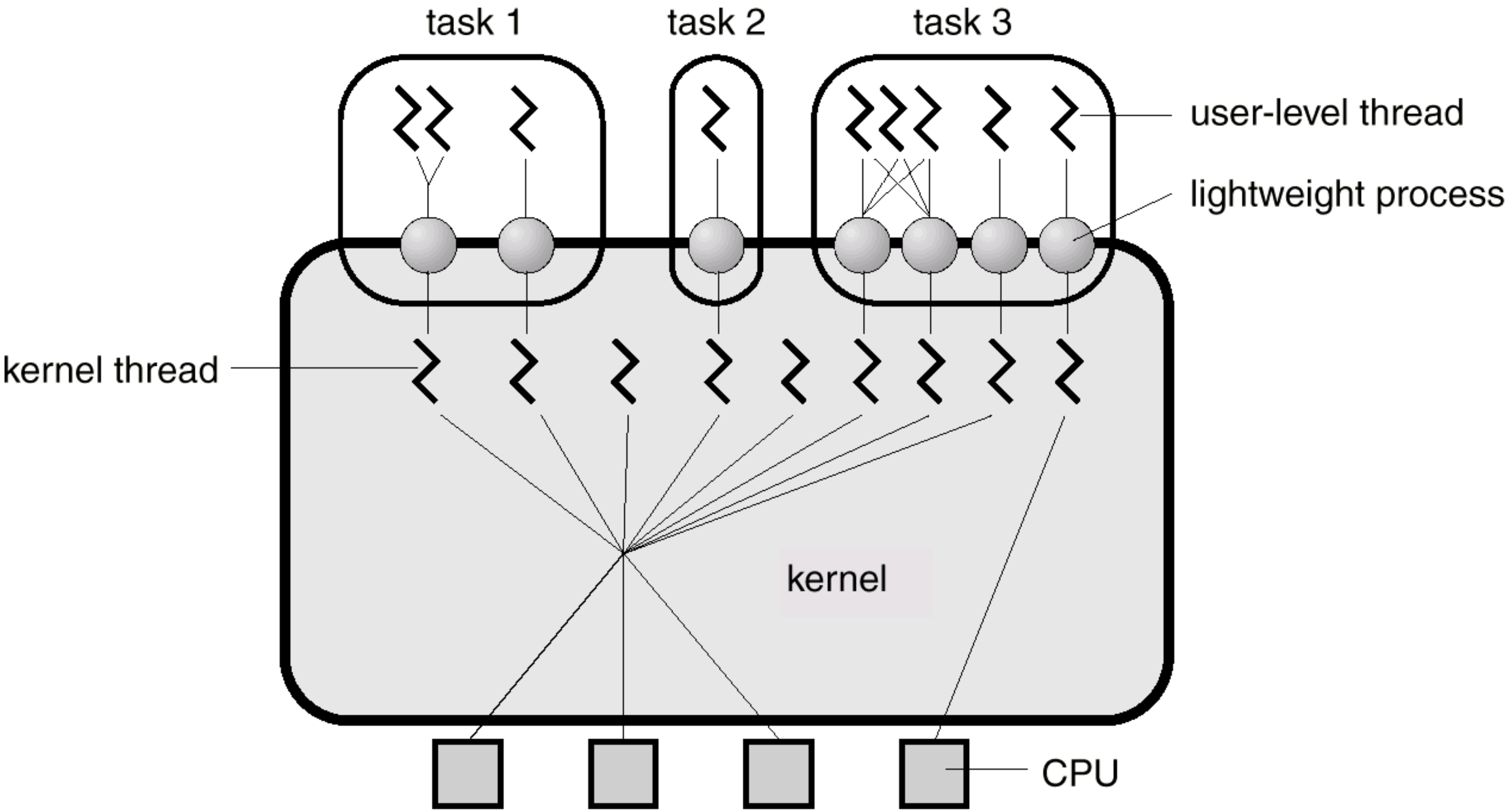
Conclusion

- Implementations
  - Topaz (original implementation)
  - Taos, Mach 3.0, BSD/OS, NetBSD [Usenix '02]
  - Digital Unix (Compaq Tru64 Unix), Solaris

- Lessons
  - Make the common case fast
  - Separating policy from mechanisms
  - Export your functionality out of the kernel for improved performance and flexibility

Solaris 2 Threads Architecture

- Scheduler activations implemented since Solaris 2.6 (prior to version 9)



Solaris 2 Threads

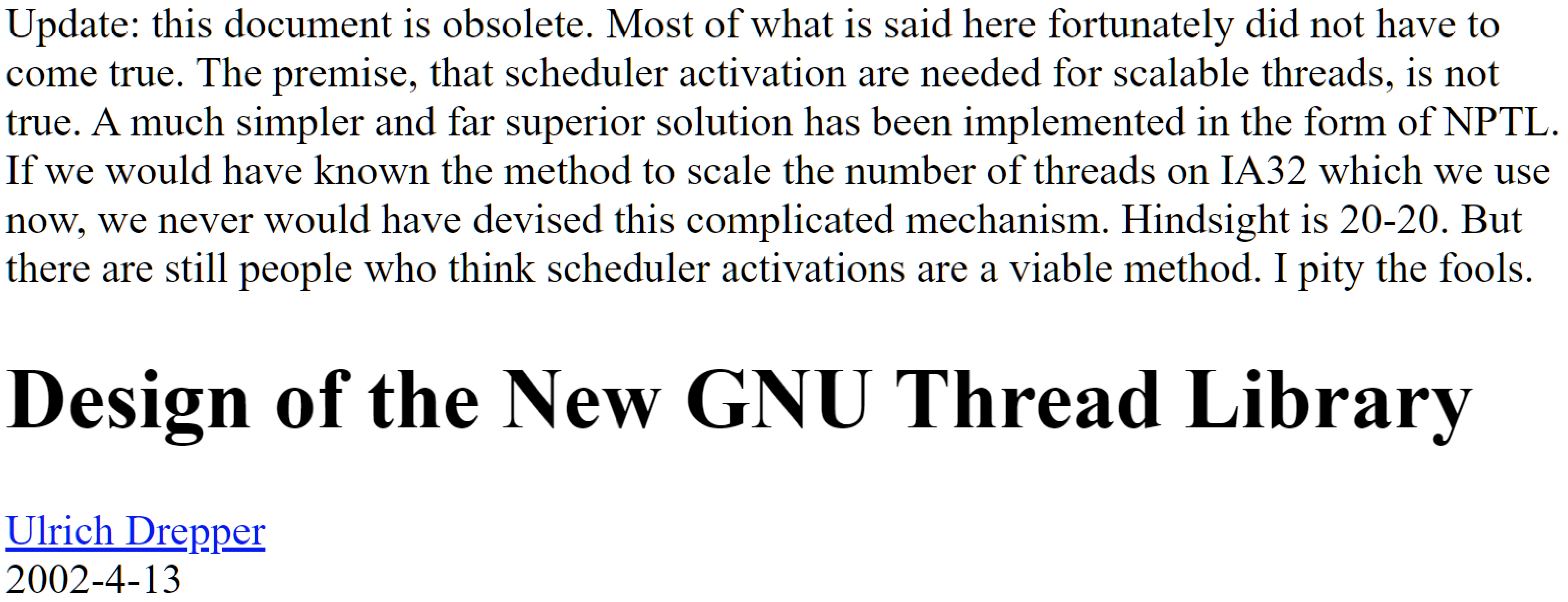
- LWPs (Lightweight Processes) sit in between the user-level and kernel-level threads.
  - User-level threads may be scheduled and switched among kernel supported LWPs without kernel intervention (no context switching)
- There is a one-to-one mapping between kernel-level threads and LWPs.
  - Operations within the kernel is maintained by kernel-level threads.
  - Kernel-level threads are scheduled by the CPUs.
- If a kernel thread blocks, it blocks the LWP and using the chain the user thread also blocks.

Solaris 9 Threads

- Things change: Going back to one-to-one model
- M:N model is too complex
  - Signal handling
  - Automatic concurrency management
  - Poor scalability due to an internal lock in user-level thread scheduler
  - Advances in kernel thread scalability
- The quality of an implementation is often more important.
  - Code paths were generally more efficient than those of the old implementation
  - More robust and intuitive
  - Simpler to develop and easier to maintain
- Binary compatibility is preserved

Linux

- LinuxThreads
  - A library implementing the POSIX 1003.1c standard for threads (introduced in 1996)
  - Standard thread library in Linux distributions from 1998 to 2004
- NGPT (Next Generation POSIX Threading) by IBM
  - M:N model based on scheduler activations
  - Extends GNU Pth library (M:1) by using multiple Linux tasks
  - https://akkadia.org/drepper/glibcthreads.html
- NPTL (Native POSIX Threading Library) by RedHat
  - 1:1 model
  - Adopted for Linux kernel 2.6
  - https://akkadia.org/drepper/nptl-design.pdf



Update: this document is obsolete. Most of what is said here fortunately did not have to come true. The premise, that scheduler activation are needed for scalable threads, is not true. A much simpler and far superior solution has been implemented in the form of NPTL. If we would have known the method to scale the number of threads on IA32 which we use now, we never would have devised this complicated mechanism. Hindsight is 20-20. But there are still people who think scheduler activations are a viable method. I pity the fools.

Design of the New GNU Thread Library

Ulrich Drepper
2002-4-13