Scheduler Activations

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

Systems Software &
Architecture Lab.

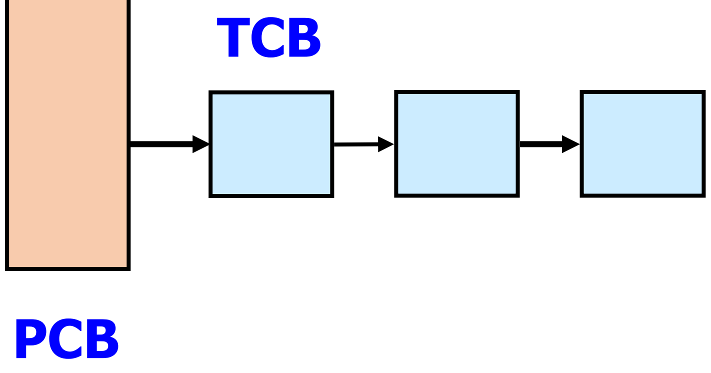
Seoul National University

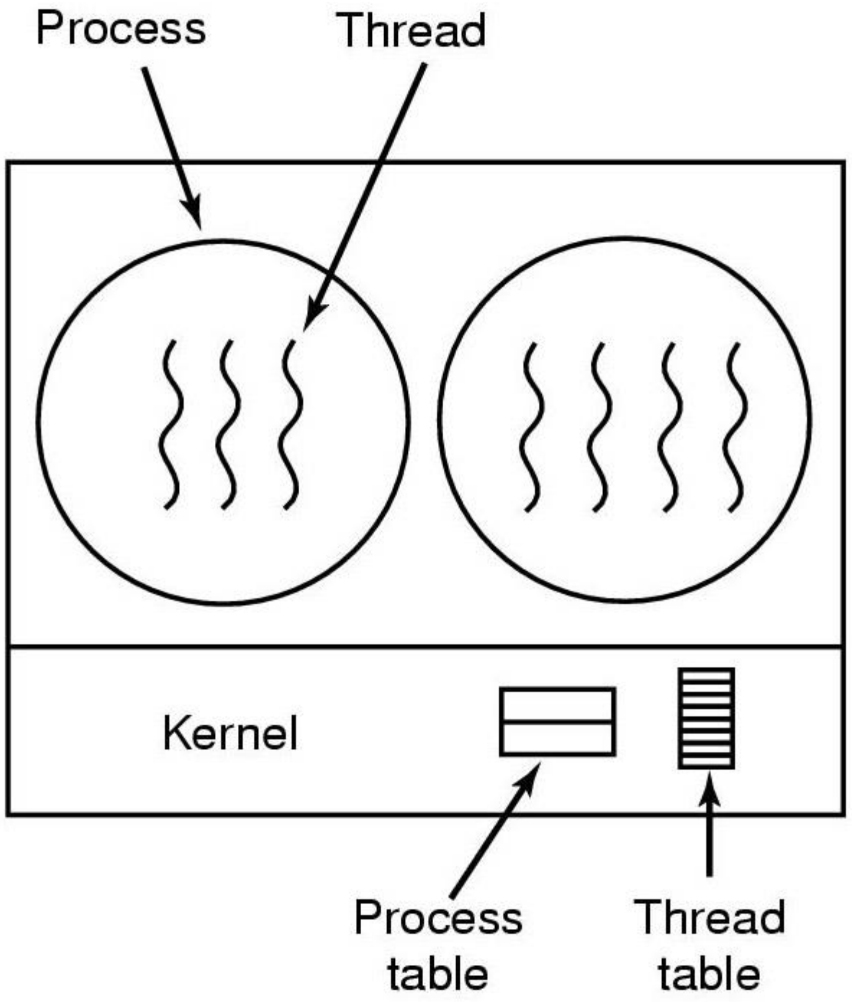
Fall 2020

(Thomas Anderson et al., TOCS '92)

Kernel-Level Threads: Implementation

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





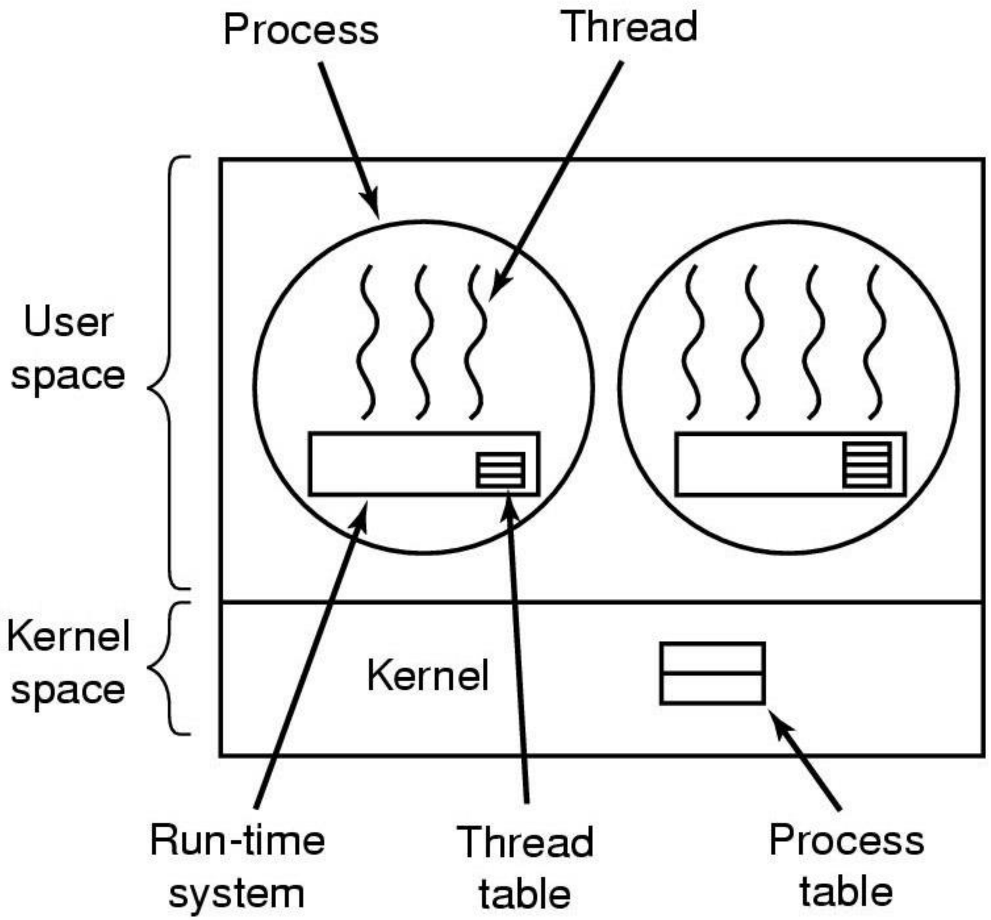
Kernel-Level Threads

- Pros
  - Cheaper than processes
  - Scheduling/management done by the kernel
    - Possible to overlap I/O with the computation
    - Multiple CPUs can be exploited

- Cons
  - Still too expensive (compared to user-level threads)
  - Thread state in the kernel
  - Need to be general to support the needs of all programmers, languages, runtimes, etc.

User-Level Threads: Implementation

- Managed by runtime library
- Views each process as a "virtual processor"
- 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

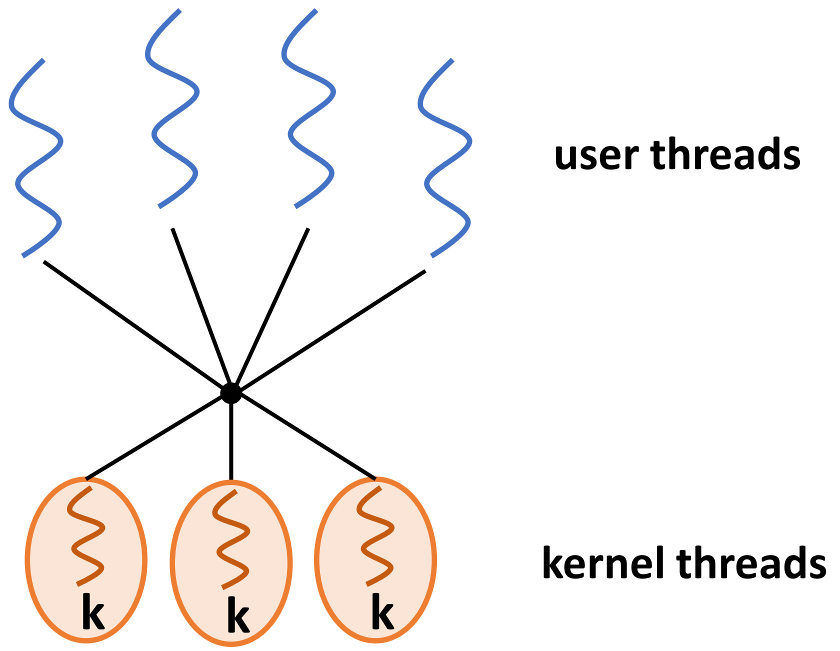
Goals

- 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
- Problems:
  - Preempting lock holder?
  - Scheduling an idle thread?
  - Preempting high-priority thread?
  - Running out of kernel threads?



Observations

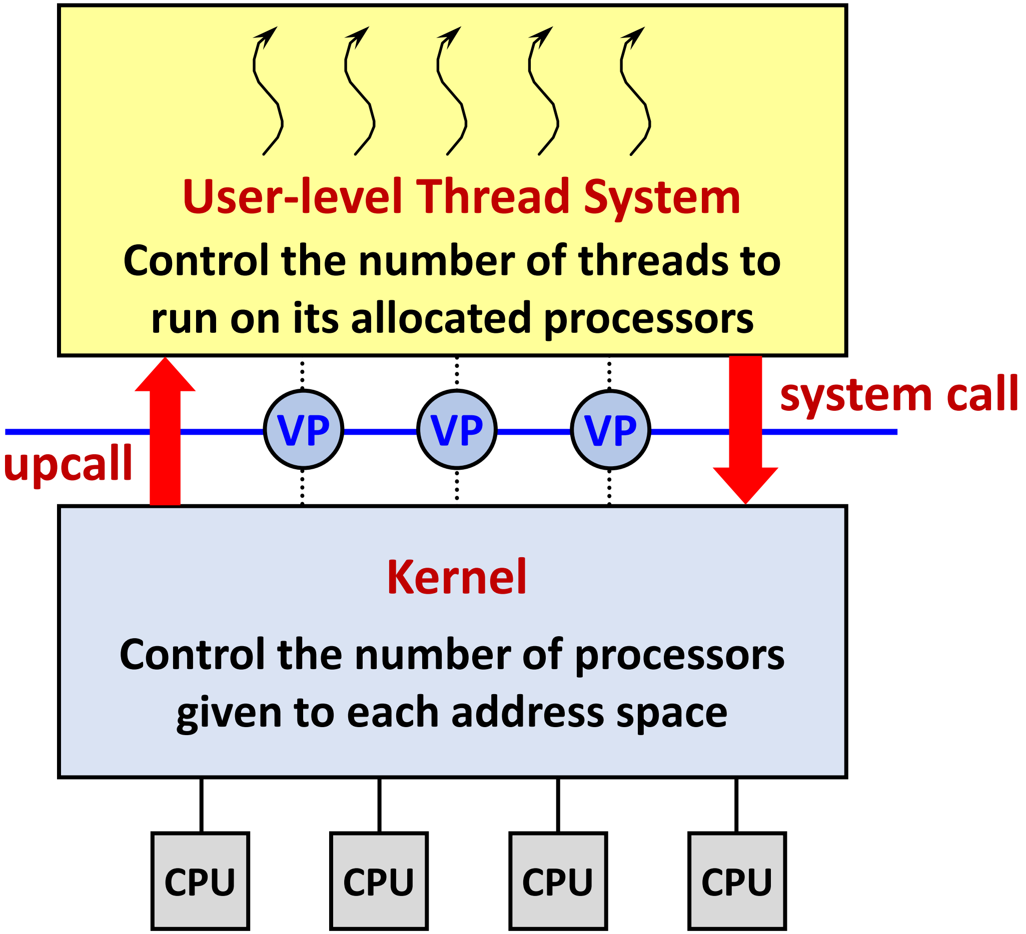
- Kernel threads are the wrong abstraction for supporting user-level thread management
- The kernel needs access to user-level scheduling information
- The user-level thread system needs to be aware of kernel events

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 upcalls
- Requires two stacks:
  - A kernel-level stack: used during system calls
  - A user-level stack: used during upcalls
  - 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

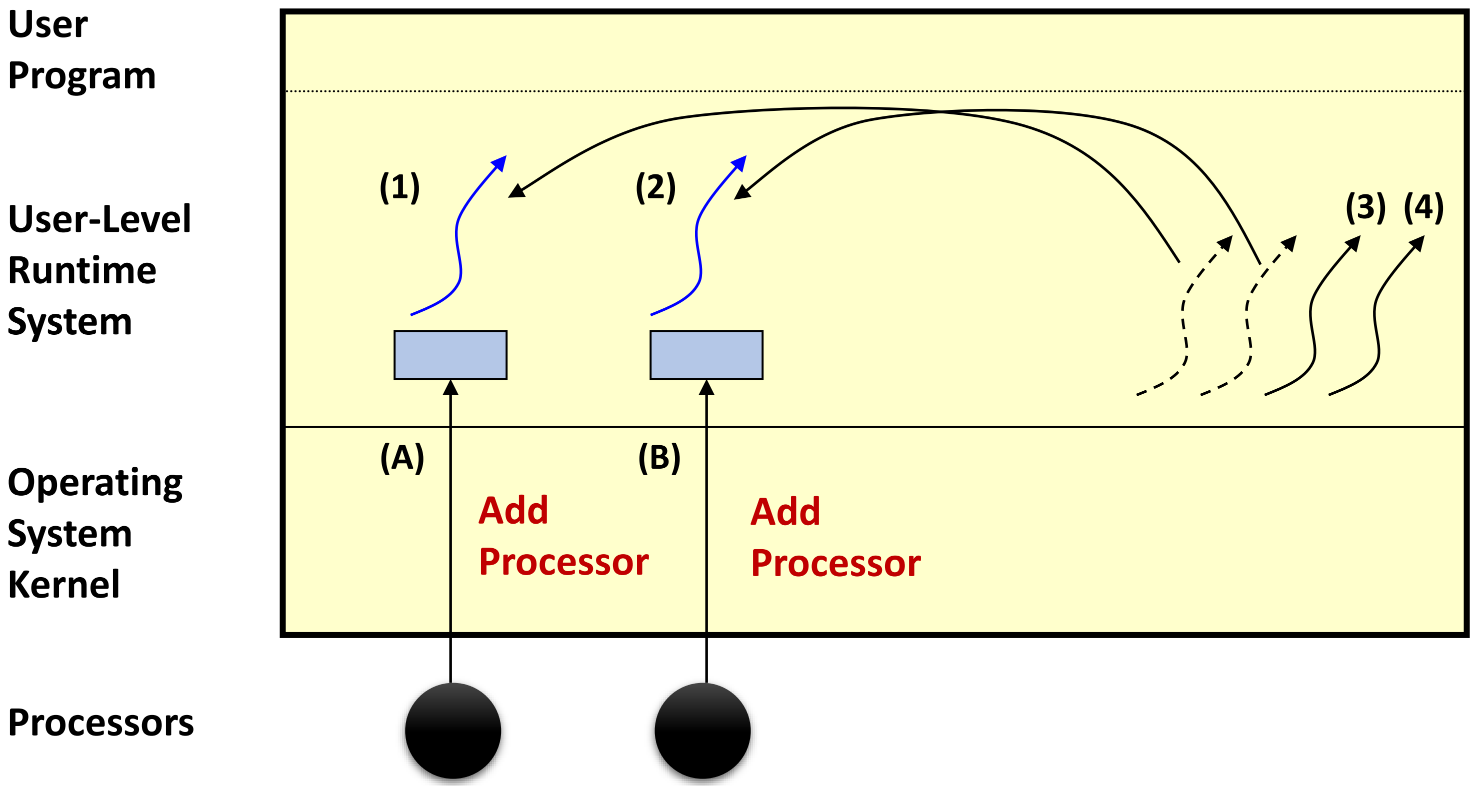
Notifies the user-level thread system whenever the kernel changes the number of processors assigned to it



Notifies the kernel when the application needs more or fewer processors

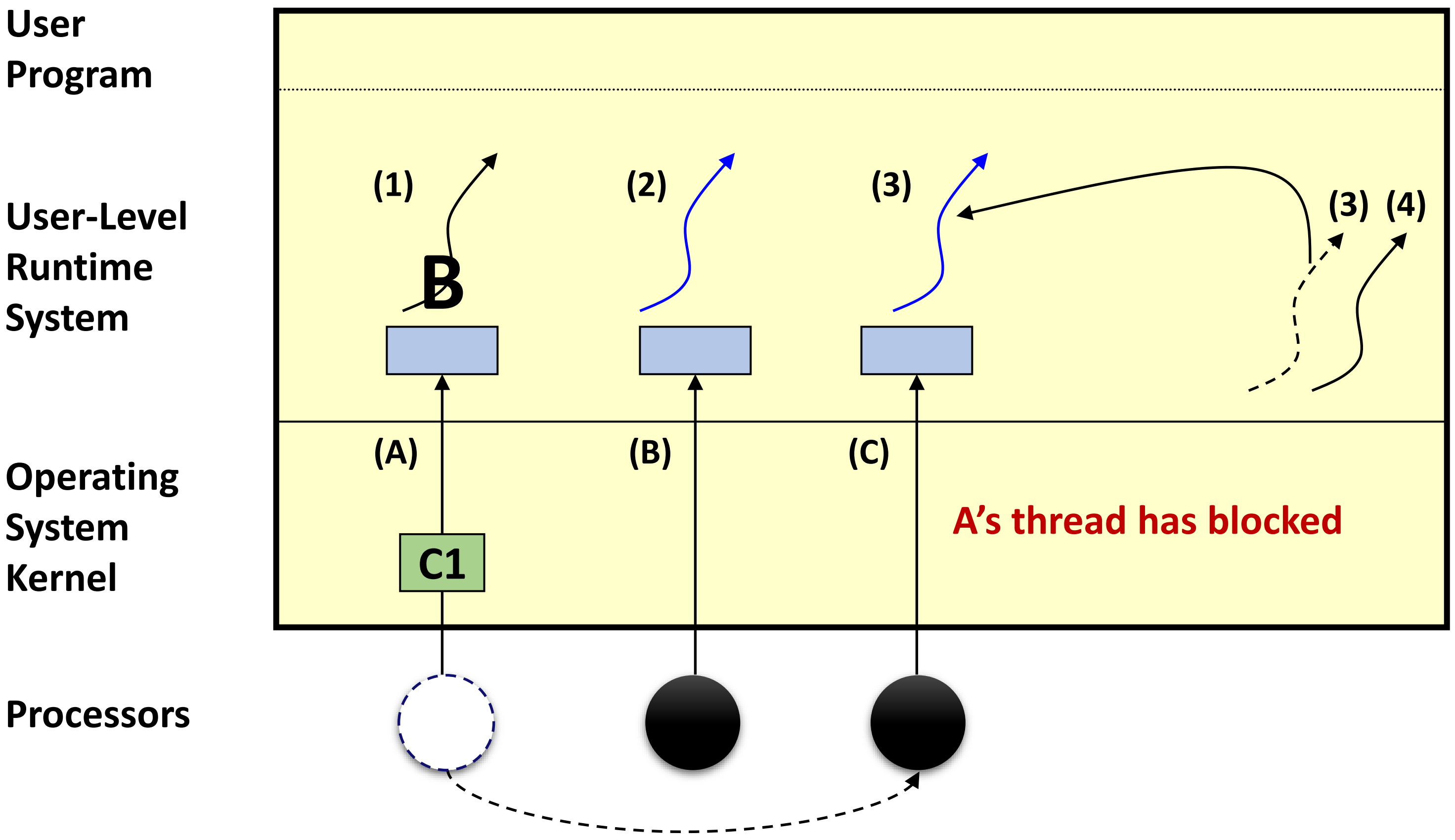
Example:

- T1: The kernel allocates two processors



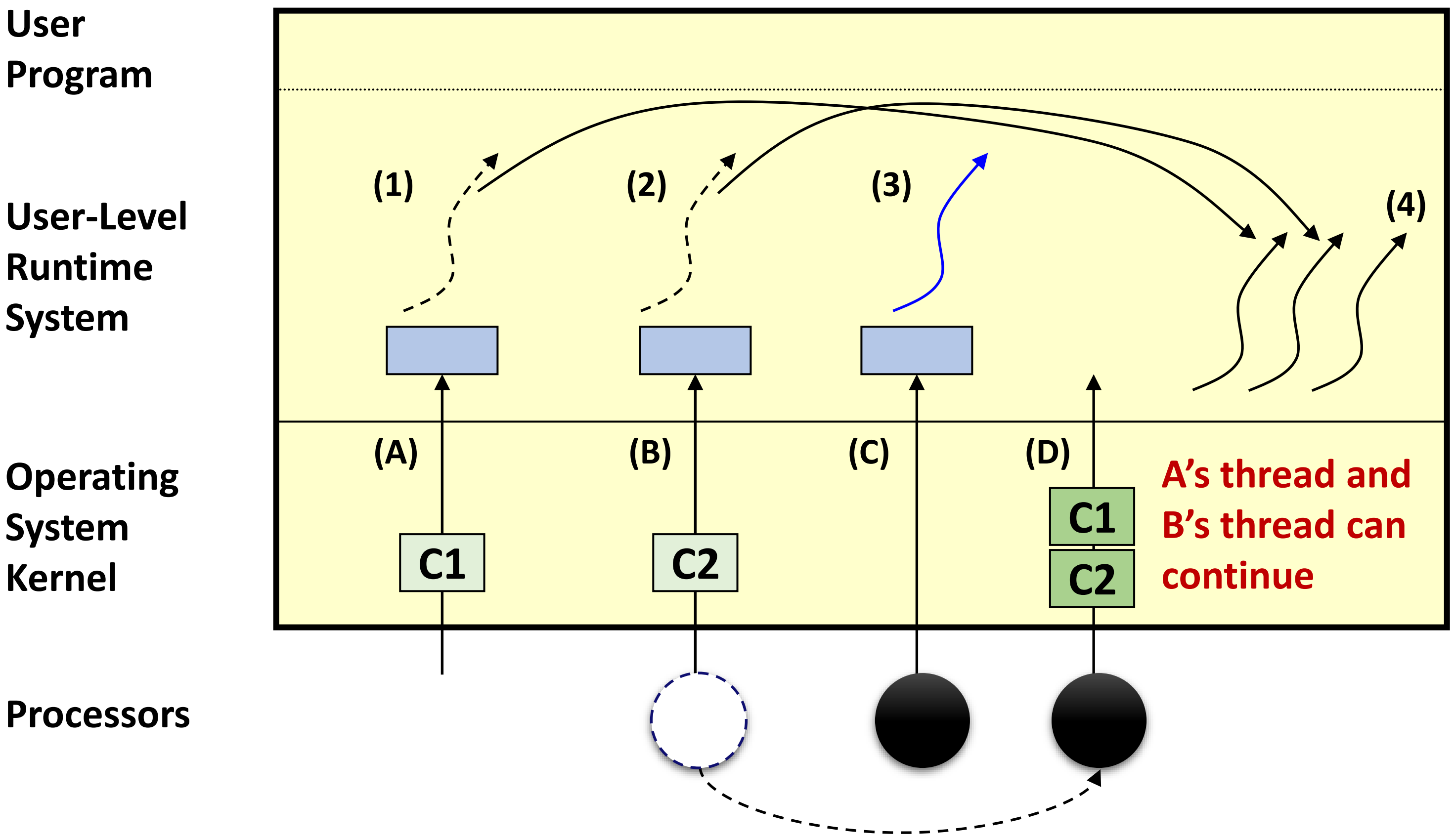
Example:

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



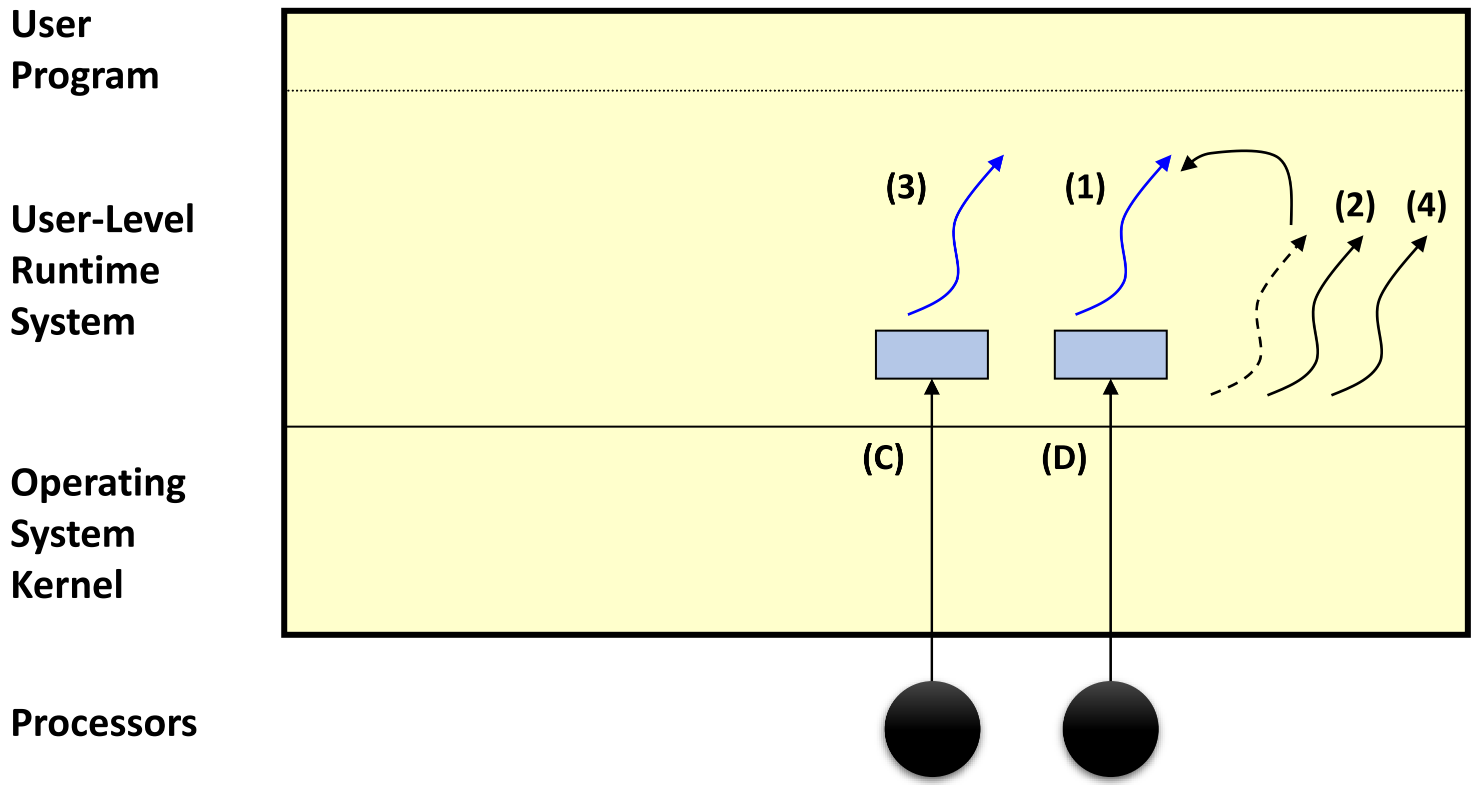
Example:

- T3: Thread 1 completes the I/O



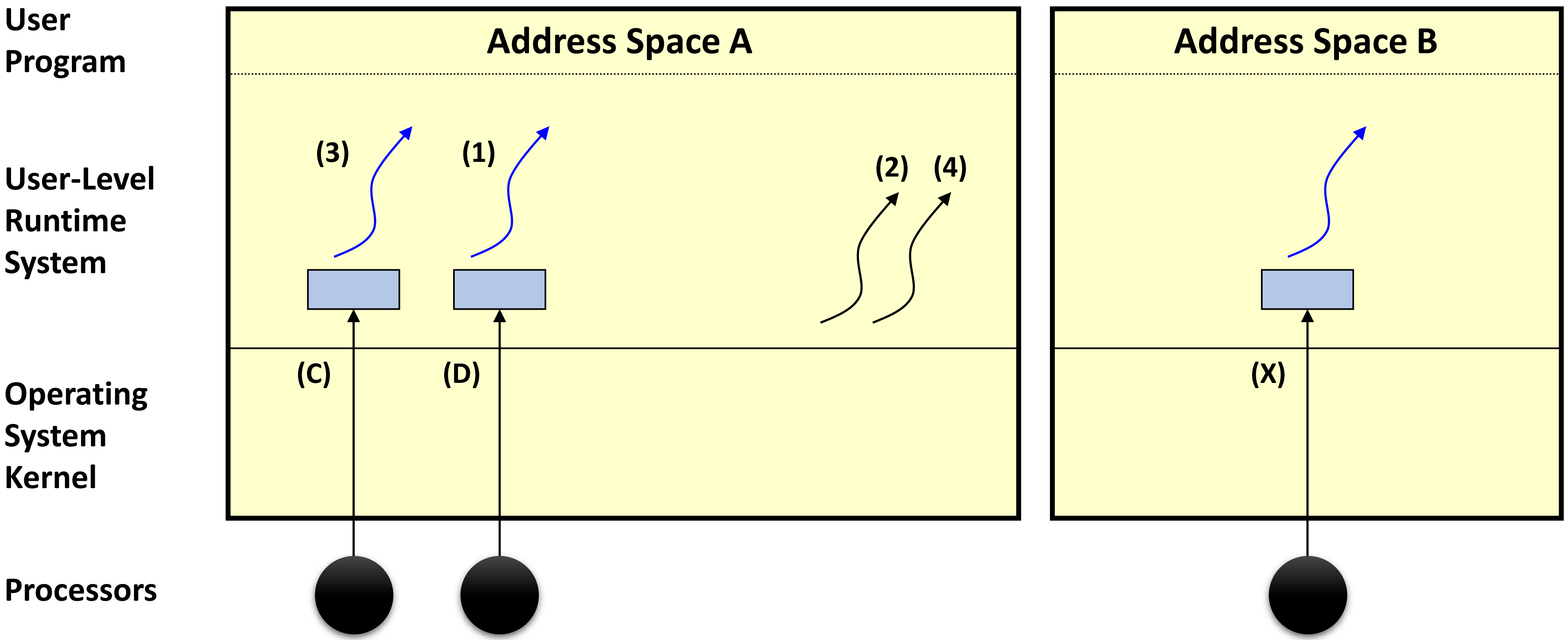
Example:

- T4: Thread 1 resumes



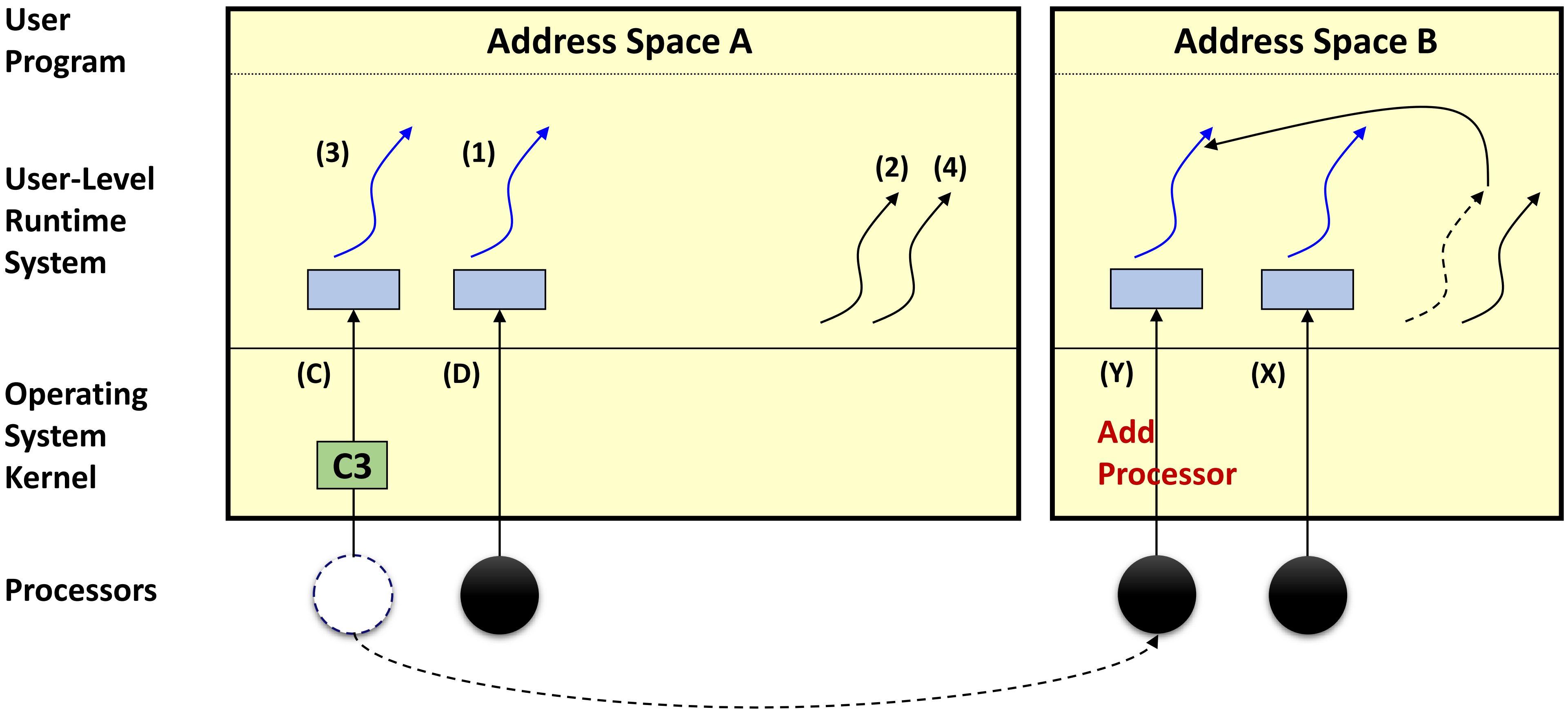
Example:

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



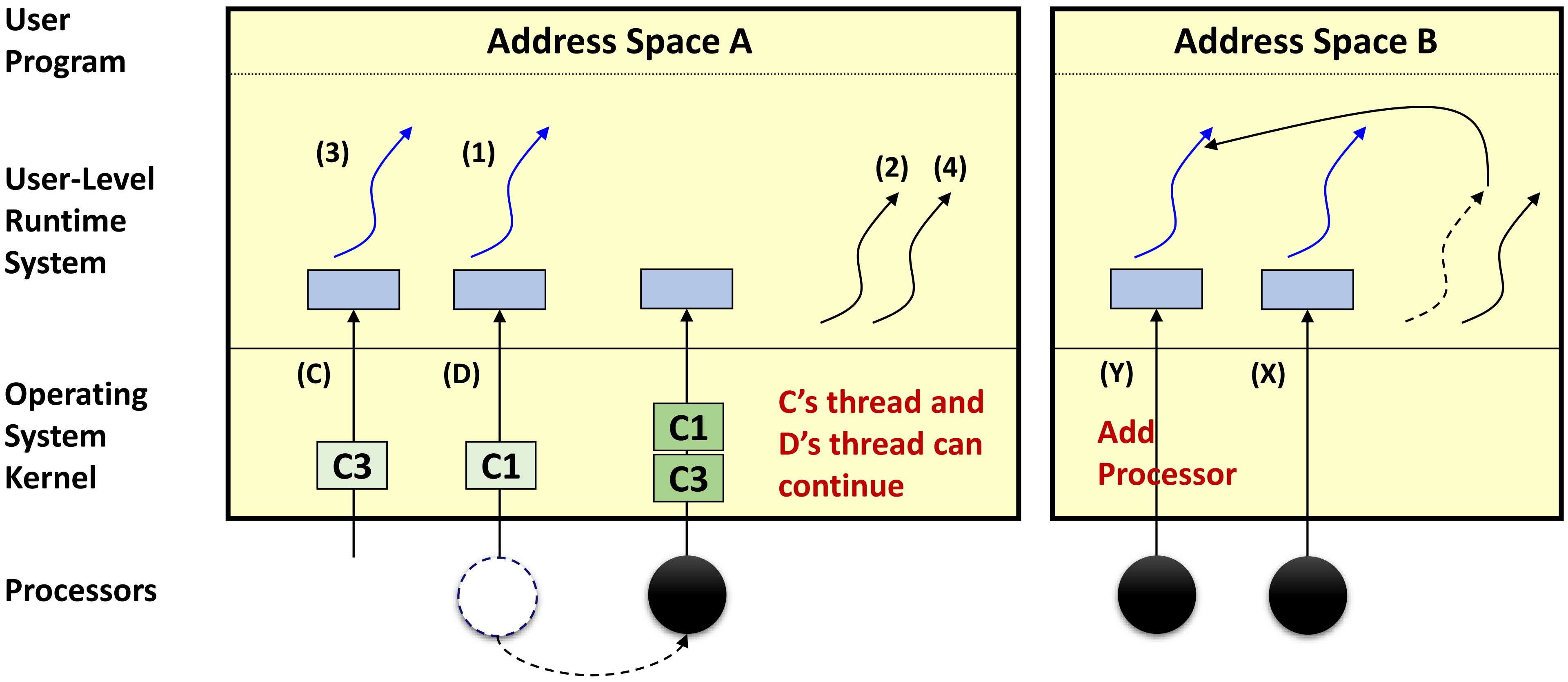
Example:

- 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

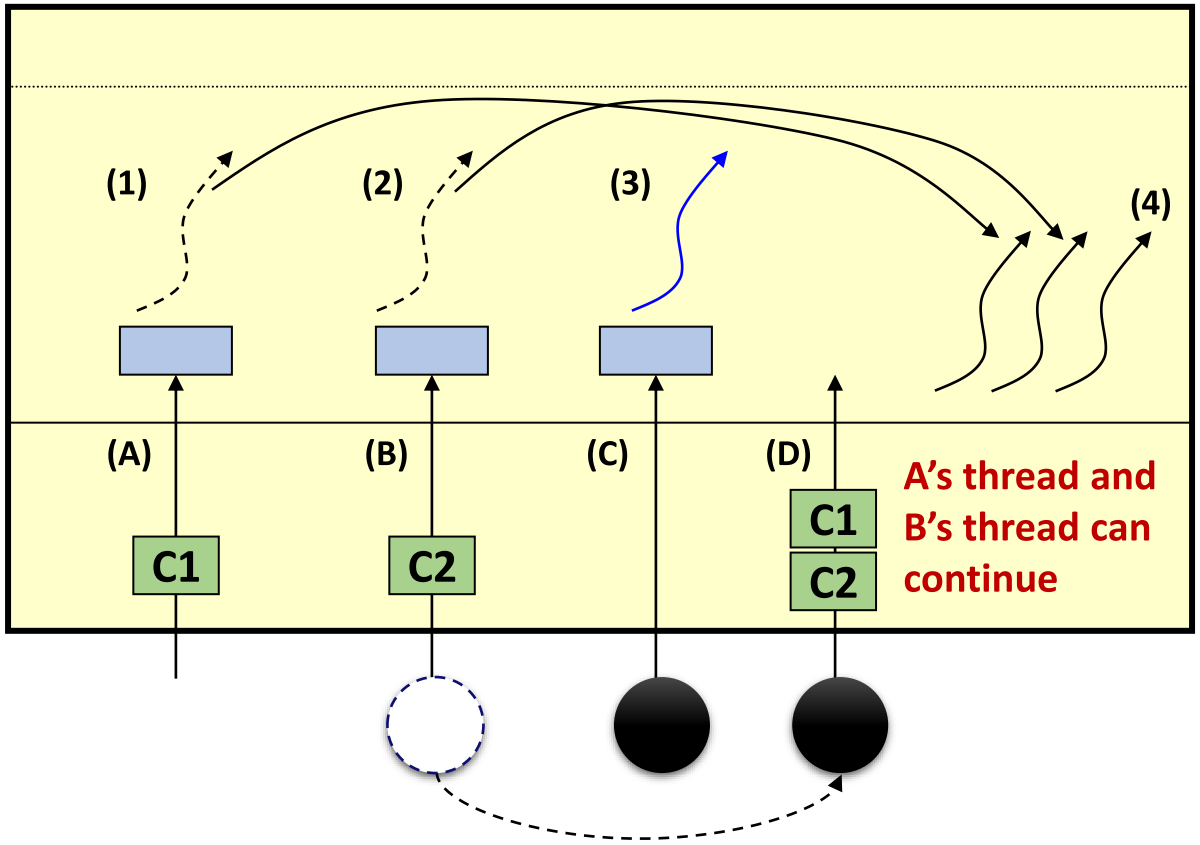
- This processor is idle ()
  - Preempt this processor if another address space needs it

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

Preemptive Priority Scheduling?

- Assume Thread 3 has a lower priority than Thread 1 and 2
- Can the user-level thread manager preempt Thread 3?

- Why or Why not?



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!

Application Transparency

- The application is free to build any other concurrency model on top of scheduler activations
- The kernel needs no knowledge of the data structures used to represent parallelism at the user level

- Scheduler activations provide a "mechanism", not a "policy"

Basic Performance

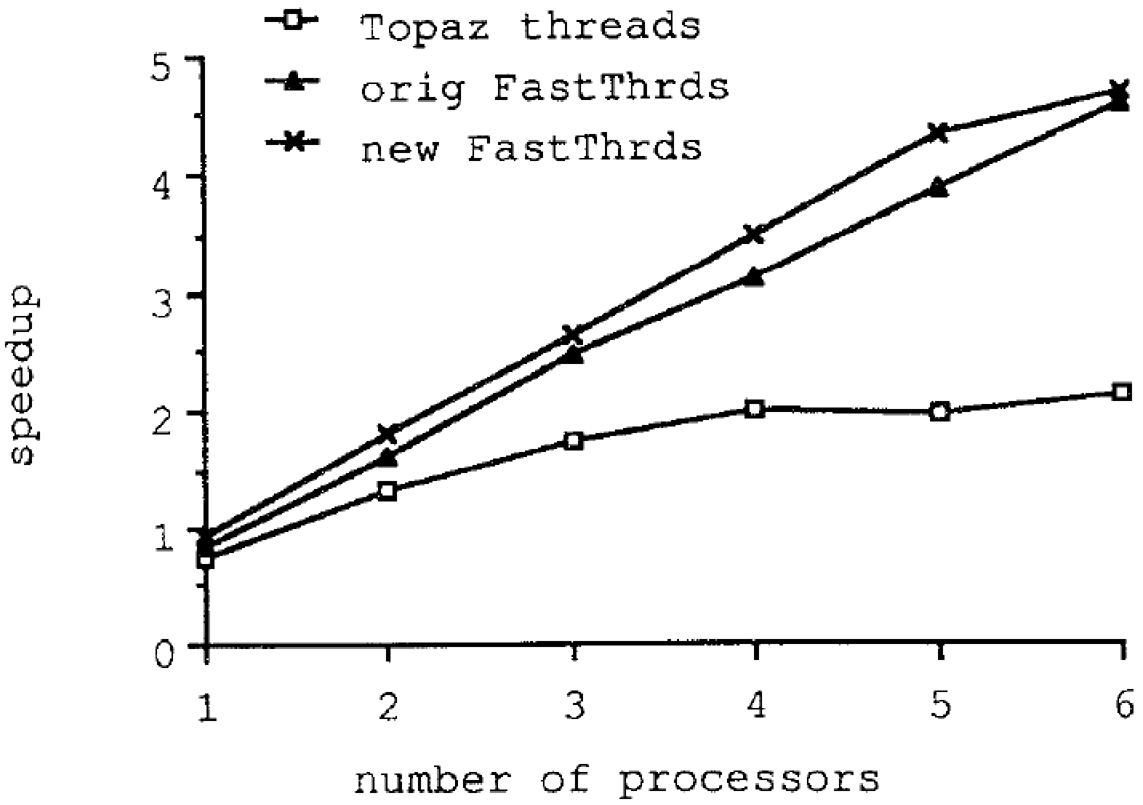
- Thread operation latencies

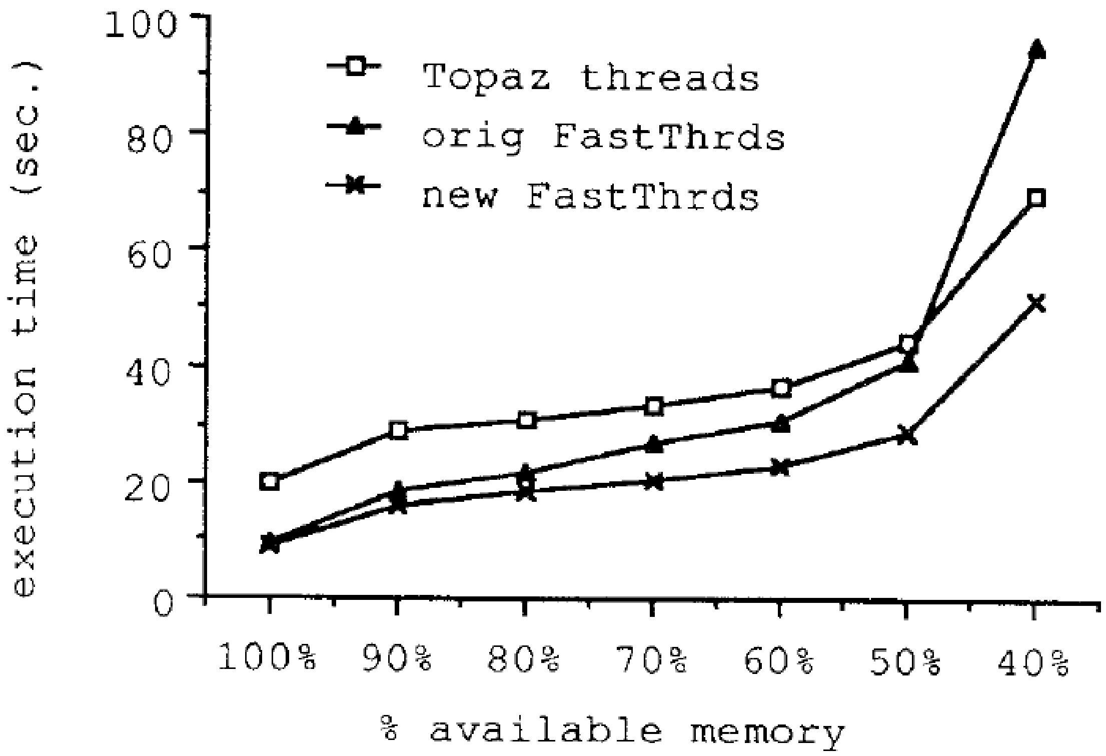
Operation	FastThreads on Topaz threads	FastThreads on Scheduler Activations	Topaz threads (Kernel-level)	Ultrix processes
Null Fork	34µs	37µs	948µs	11300µs
Signal-Wait	37µs	42µs	441µs	1840µs

- 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





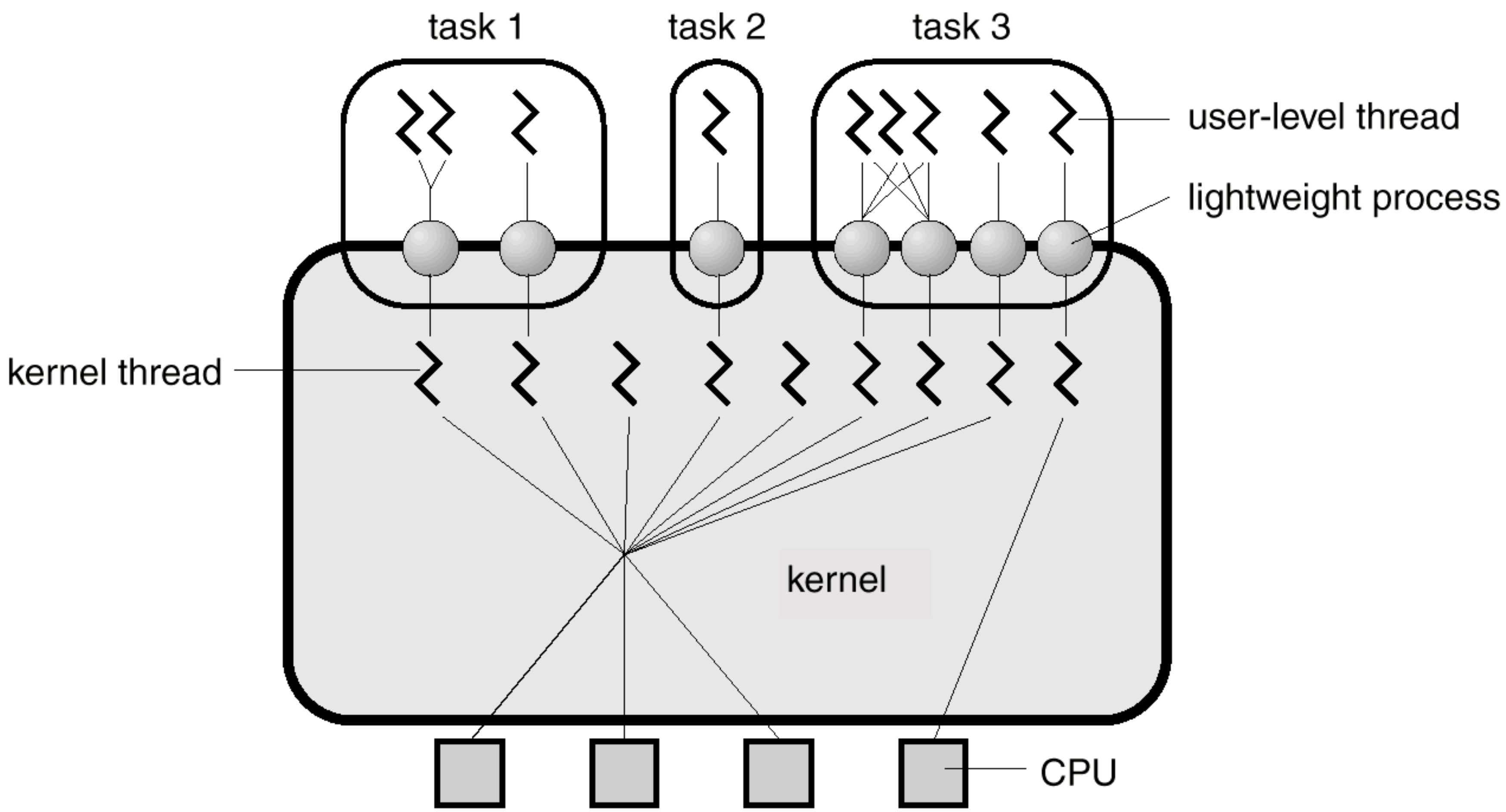
Conclusion

- 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