Duke Systems

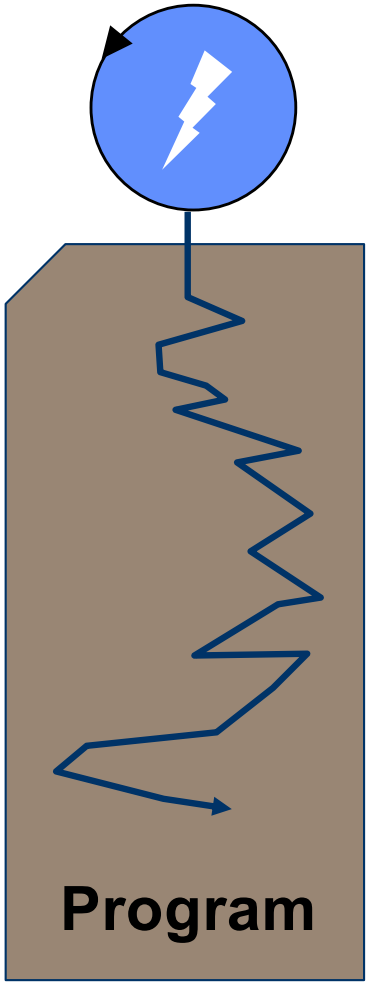
Scheduler Activations

Reconceptualizing the kernel interface for high-performance threads

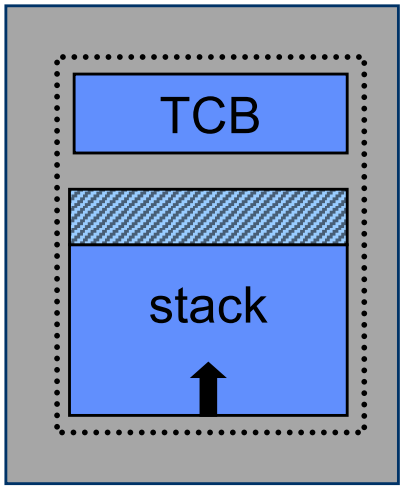
Jeff Chase

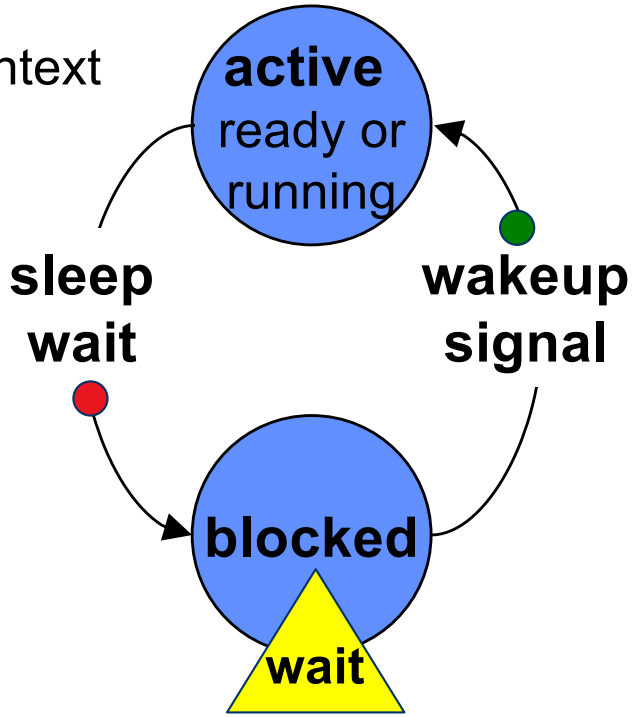
Duke University

A thread



Each thread has a thread control block (TCB) or context block, and its own stack.





When **blocked** its TCB is on a **sleep queue** of threads waiting for a specific wakeup event.

When **ready** its TCB is on the **ready queue**.

Threads: abstraction vs. implementation

Thread is an **abstraction** with multiple implementations.

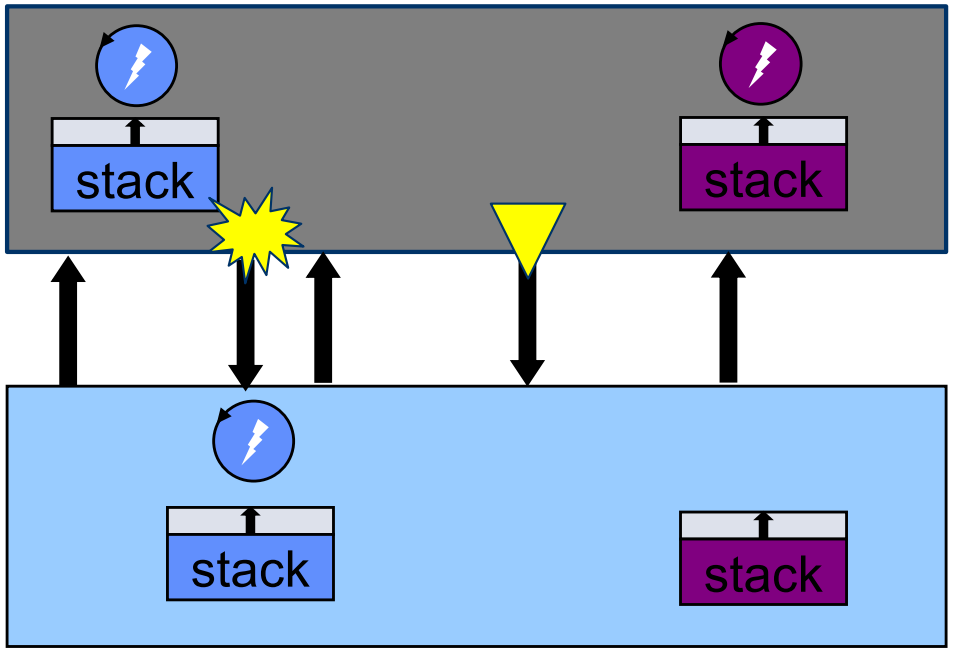
1. The OS lectures focus on **kernel-based** thread model.
   - OS kernel manages threads (by any name).
   - E.g., Linux, Windows, MacOS
2. The p1* labs use a **library** for **user-level** threads.
   - No specific kernel support
   - Works on classic Unix on uniprocessors

Which is “better”? Is it an either/or choice?

What’s the “right” way to implement threads?

Kernel-based threads

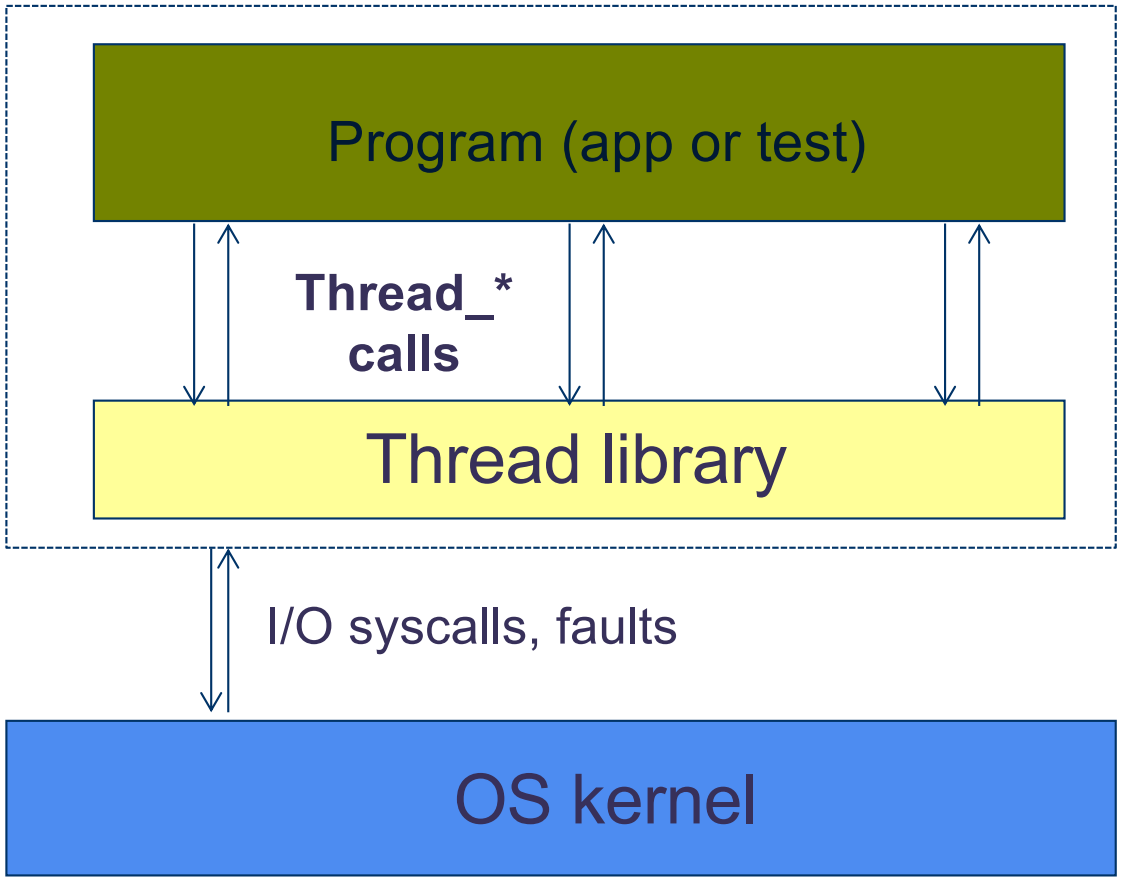
- Syscall to create thread, with second kernel stack in kernel space.
- Syscalls to sleep/wakeup to synchronize (e.g., futex).
- Threads can use all blocking syscalls, e.g., for I/O.
- Block only in kernel mode: kernel sched dispatches next ready.



- Kernel starts thread in its start procedure with an “upcall”.
- Thread enters kernel on trap or fault, runs on its kernel stack.
- Resumes user context on (optional) return from trap/fault.
- Kernel may modify context and resume wherever it wants.
- On timer interrupt, kernel preempts the running thread if its quantum expired.

Thread library

- Library routines to create/block/switch threads in user mode.
- Uses makecontext/swapcontext to switch between streams.



- Does not request threads from kernel.
- Kernel does not know or care.
- At most one thread at a time can run on process context.
- At most one in kernel for trap/fault at a time.
- Threads use the same kernel stack.

Kernel+library: better together

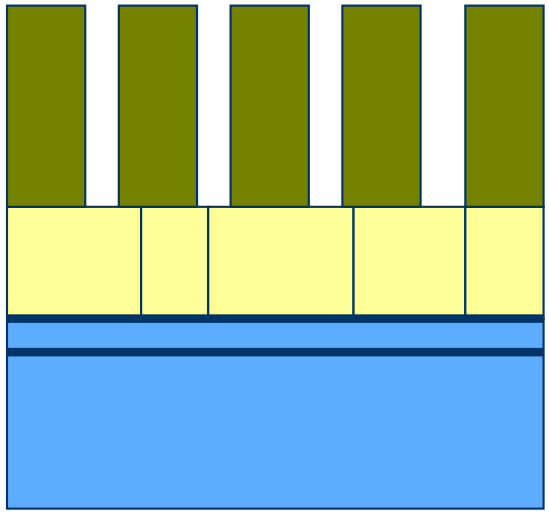
- In truth, all thread systems combine kernel + library.
  - Programmer's API is User-Level Thread System (ULTS) library.
  - The library uses the kernel interface and **hides** it from the app.

→ **The kernel syscall interface is not the API.**

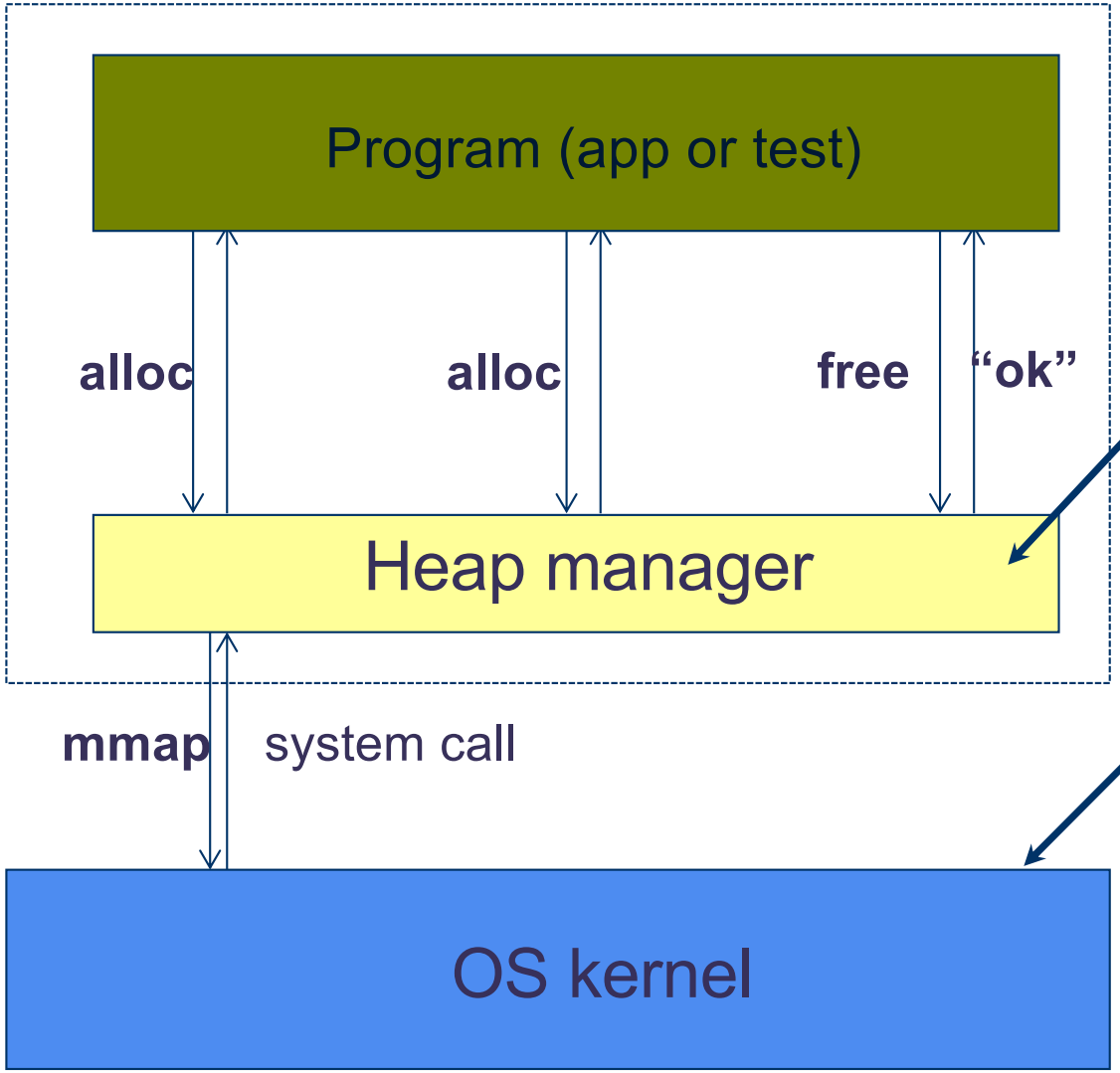
  - Design library API for ease of use.
  - Design syscalls for **power** and **speed**.

- **Example**: pthreads on Linux
  - NPTL library ("Posix threads")
  - Uses "hidden" syscalls clone(), futex.

- **Today:** what is the "right" division of function between lib/kernel code?



Library and kernel work together
Example: heap



Library manages its own resources its own way.

“Hidden” kernel syscall interface for “raw” resources: e.g., VM slabs.

OS mediates access to shared machine resources. Only as needed for trust, protection and isolation.

Library OS: philosophy

- **Principle**: keep it in a library whenever possible.
- Keep the kernel simple and small.
  - Minimal function to support library.
  - Kernel protects access to machine, nothing more.
- Benefits:
  - **Fast**: fewer syscall traps
  - **Secure**: minimal kernel has fewer bugs
  - **Flexible**: permits innovation in library without changing kernel. Users choose a library to suit their needs.
  - **Safe**: library bugs crash process, not the whole system
  - **Customizable**: Factor policy+programming model out of kernel.

Goal: keep the kernel out of it

All times were measured in microseconds 30 years ago:

Operation	FastThreads	Topaz Threads	Ultrix Processes
Null fork	34	948	11300
Signal-wait	37	441	1840

User-level thread system (ULTS)

OS kernel thread system

Classic Unix processes

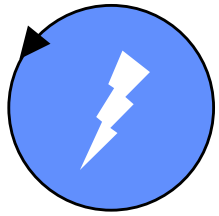
ULTS is order(s) of magnitude faster. **Why not just use it?**

Scheduler Activations. Anderson et. al. SOSP 1991

Thread library API

Threads

```
thread_create(func, arg);
thread_yield();
```



Mesa monitors

Locks/Mutexes

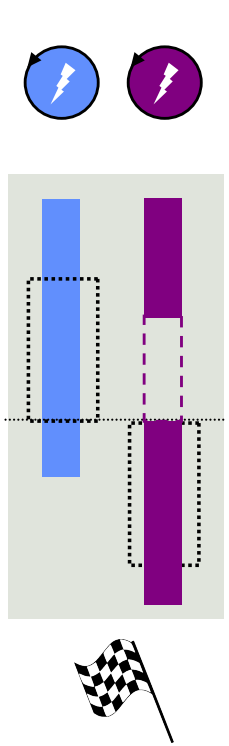
```
thread_lock(lockID);
thread_unlock(lockID);
```

Condition Variables

```
thread_wait(lockID, cvID);
thread_signal(lockID, cvID);
thread_broadcast(lockID, cvID);
```

All functions return an error code: 0 is success, else -1.

Thread library: the story so far



Context switch out on **sleep**, yield/**preempt**, or **exit**. Get next ready thread, dispatch it.

What about multiple cores?

- Our p1 thread library uses only one core.
- What if multiple cores?
  - ULTS can use a kernel-based thread as “vessel” for each core.
  - ULTS decides threads to run on each “vessel”, e.g., priority.
  - Physical concurrency→nice to have spinlocks!
  - What if no ready threads? Spin?
- What if the number of cores changes?
  - OS allocates CPU resources.
  - Grants cores, takes them away.
  - E.g., by timer-driven slicing
  - Or if a thread blocks for I/O…

Integration: kernel-based threads

With **kernel-based** threads, the kernel "knows" the status of all threads and all resources (cores).

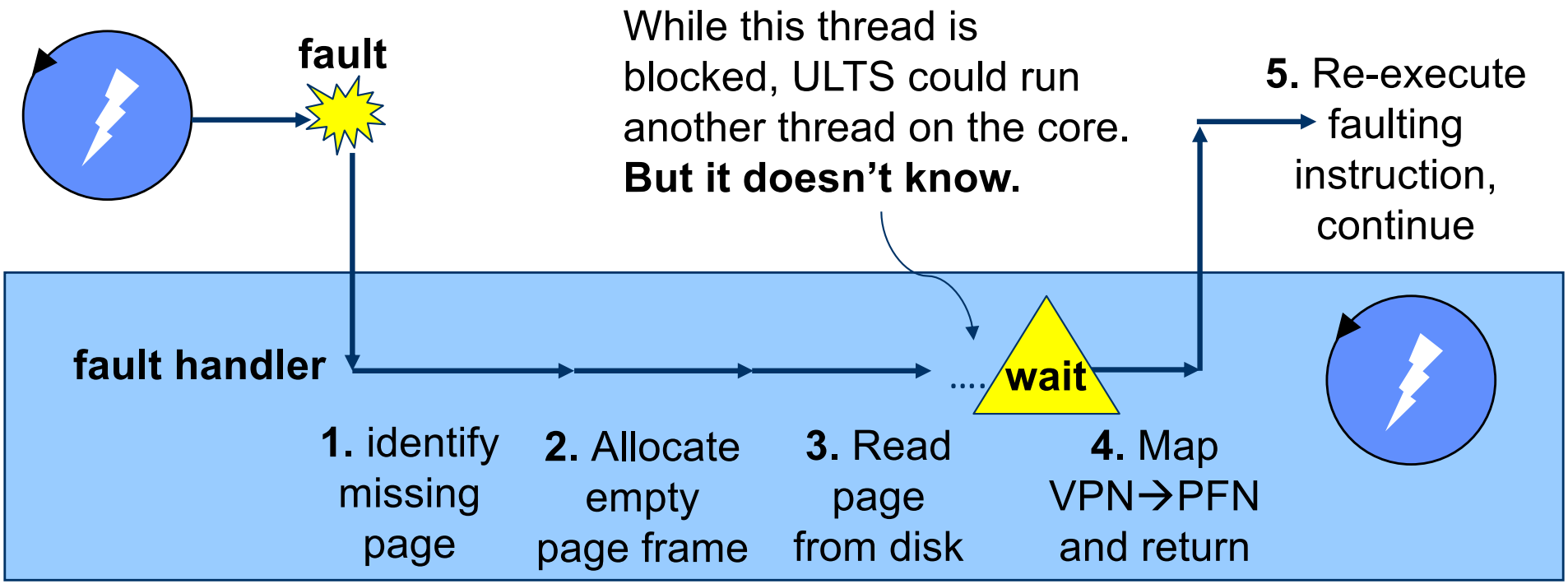
- Thread blocks for I/O → run another thread.
  - **No core idles when a thread is ready.**
- Put each core to its "highest and best use".
  - Timer-driven preemption with system-wide scheduling
  - **No thread runs when a higher-priority thread is ready.**
- But kernel does not "see" events in the user program.
  - Kernel does not "see" idle cores.
  - Does not know if a preempted thread holds a spinlock.
  - No spinlocks! Use only kernel-supported blocking locks: slow.

Integration: ULTS

Poor integration for ULTS over kernel-based threads:

1. App has idle core → core is wasted.
2. Thread blocks in kernel for I/O → app loses the core.
3. Kernel preempts a core (“vessel”) running thread T?
   - T holds a spinlock→wasted cycles. No fast spinlocks for ULTS!
   - T holds any lock→other threads wait longer for the lock.
   - T is a high-priority thread→priority inversion.

Page fault → app loses the core



When a thread **blocks**, it stops running, leaves the core for use by other threads, and **sleeps**: it wakes/resumes only after some specified event or condition occurs.

Threads block for page faults, and for many other reasons as well.
It happens all the time.

Scheduler Activations.
Anderson et. al. SOSP 1991

Scheduler Activations

- Scheduler Activations is a kernel abstraction.
  - FastThreads User-Level Thread System (**ULTS**) shows how to build threads efficiently in a library using scheduler activations.
- It's **not an API**! Designed for use by the ULTS.
  - **Kernel** manages only resource (processor) allocation and I/O.
  - ULTS library **runtime system** does the rest: schedule threads, implement thread+synchronization primitives, etc.
- Key idea: "Pass the cores"
  - Asynchronous control transfers across the kernel boundary
  - A transfer passes control of the core ULTS←→kernel.
  - Transfer contexts as data.
  - Expose knowledge and control over resources to the ULTS.

Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism

Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Threads are the vehicle for concurrency in many approaches to parallel programming. Threads separate the notion of a sequential execution stream from the other aspects of traditional UNIX-like processes, such as address spaces and I/O descriptors. The objective of this separation is to make the expression and control of parallelism sufficiently cheap that the programmer or compiler can exploit even fine-grained parallelism with acceptable overhead.

Threads can be supported either by the operating system kernel or by user-level library code in the application address space, but neither approach has been fully satisfactory. This paper addresses this dilemma. First, we argue that the performance of kernel threads is *inherently* worse than that

1 Introduction

The effectiveness of parallel computing depends to a great extent on the performance of the primitives that are used to express and control the parallelism within programs. Even a coarse-grained parallel program can exhibit poor performance if the cost of creating and managing parallelism is high. Even a fine-grained program can achieve good performance if the cost of creating and managing parallelism is low.

One way to construct a parallel program is to share memory between a collection of traditional UNIX-like processes, each consisting of a single address space and a single sequential execution stream within that address space. Unfortunately, because such processes were designed for multi-

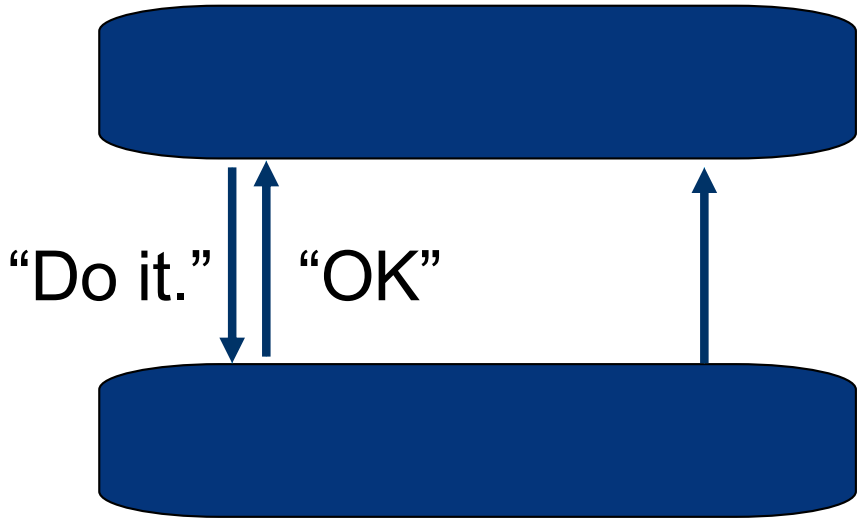
SOSP 1991

Reconceptualizing the kernel interface

- Kernel gives process a virtualized multiprocessor (MP).
- In-process ULTS manages its virtual MP.
- Kernel allocates cores to ULTS. (**How?**)
- In-process ULTS "knows" what/how many cores it has.
  - Kernel can "take it back" at any time, but it notifies ULTS.
- ULTS manages its own cores its own way.
- ULTS notifies kernel of its demand for cores.
  - Don't need the core? Return it to the kernel. (**How?**)
  - Want more? Request more cores from the kernel.
- **Needed**: some way to pass cores back and forth.

Upcall

Standard trap/return.



Upcall

"Something happened." E.g. I/O completion for non-blocking I/O API, Asynchronous event

Might not return.

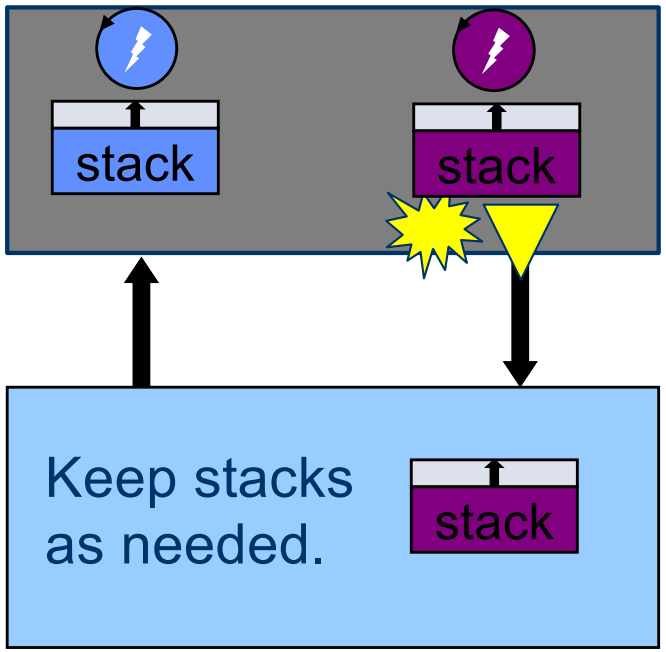
- **Upcall**: cause to invoke a procedure in a client module.
- E.g., kernel: point PC at handler; point SP at a stack; **go**.
- Notifies client of an asynchronous event or completion.
- "Like an interrupt." *See also*: Unix signal handler.

Pass the core
Asynchronous control transfers

- Traps/faults and “returns” are decoupled: every return is an upcall.
- Library and kernel manage/exchange contexts, switch among stacks as needed to run asynchronous streams with available cores.

“Here is a core; run it your way. See also the info at addr X.”

Kernel constructs trapframe to “return” to designated handler in user mode: **upcall**. Like launch to main() or caught signal.



“Here is a core back to execute a request for me—or—I just don’t need this core any longer.”

Transfer to kernel by ordinary trap or fault. Return is optional, e.g., like exit().

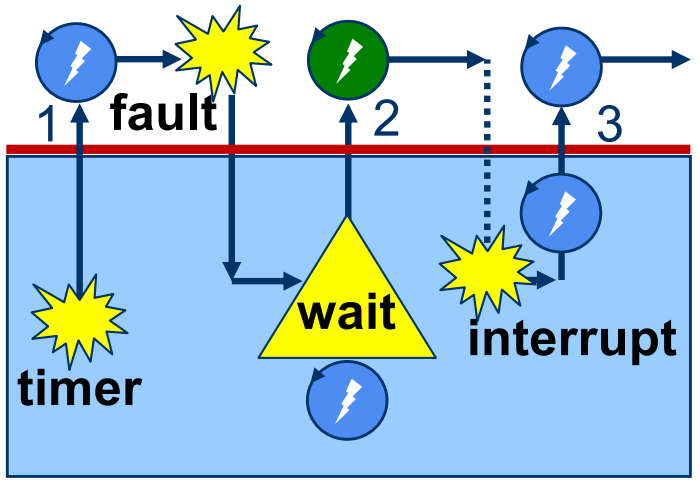
What is a scheduler activation?

An **activation** is a virtual core delivered to the ULTS.

- Kernel delivers core with an **upcall** to ULTS scheduler.
- The core invokes a ULTS entry point.
- ULTS dispatches next ready thread.

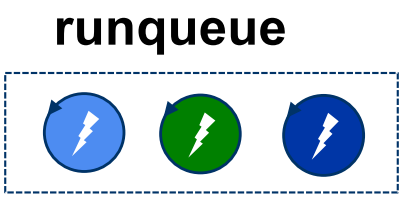
Uses:

1. Here is the new core you requested; use it as you choose.
2. Your thread has blocked; here is your core back.
3. A thread has unblocked: here is its context, and your current one.
4. I took a core away that was running this context (not shown).

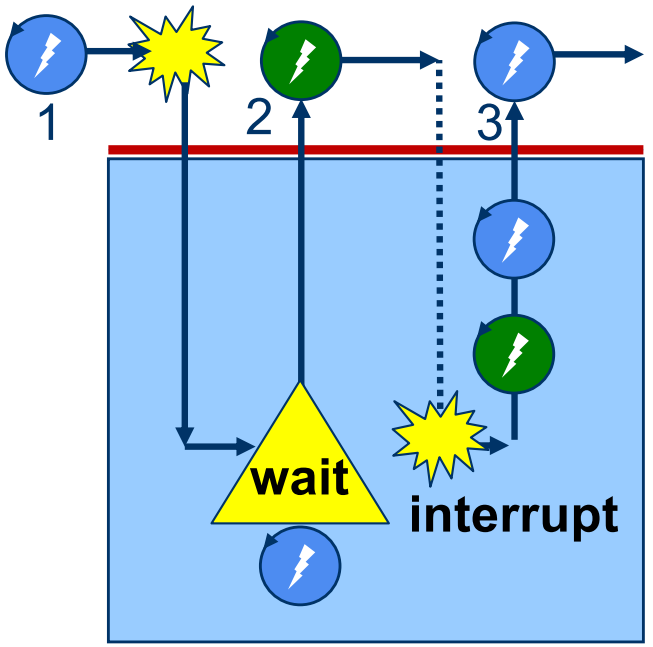


The tricky part: delivering contexts

- **What core to use to notify ULTS of an event?**
  - Unblock: deliver unblocked context
  - Loss of core: deliver preempted context
- Pick another of app's cores and preempt current context.
- Return context with the activation.
- **And** the context affected by event.
- So upcall delivers **two** contexts.

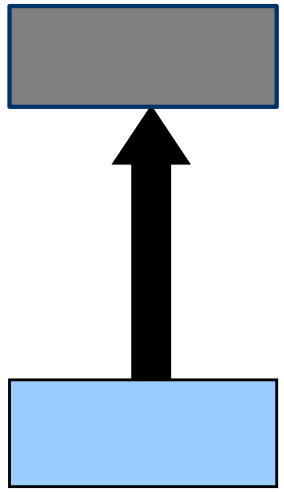


1. Dispatch blue
2. Dispatch green
3. Return green to queue, then dispatch blue.



Activation upcalls

1. Add new core
   - ULTS picks a thread/context and dispatches it.
2. Blocked context
   - ULTS picks another thread and dispatches it.
3. Unblocked context
   - Unblocked context to resume (A).
   - **And** context (B) from core used for upcall.
   - Put A or B back on runqueue, resume the other.
4. Preempted core!
   - Ready context (A) of preempted virtual core.
   - Deliver/handle as for #3.



Scheduler Activations.
Anderson et. al. SOSP 1991

Complications

- **If** a core is prempted in ULTS code (e.g., context switch)
- **or** in a spinlock-protected critical section
- **then** ULTS must resume the preempted context until it exits its critical section.
- **Note**: this approach enables **safe spinlocks in user mode**. If preempted, ULTS resumes thread immediately on another core.
- Page fault in ULTS code also requires special handling.

A classic systems paper

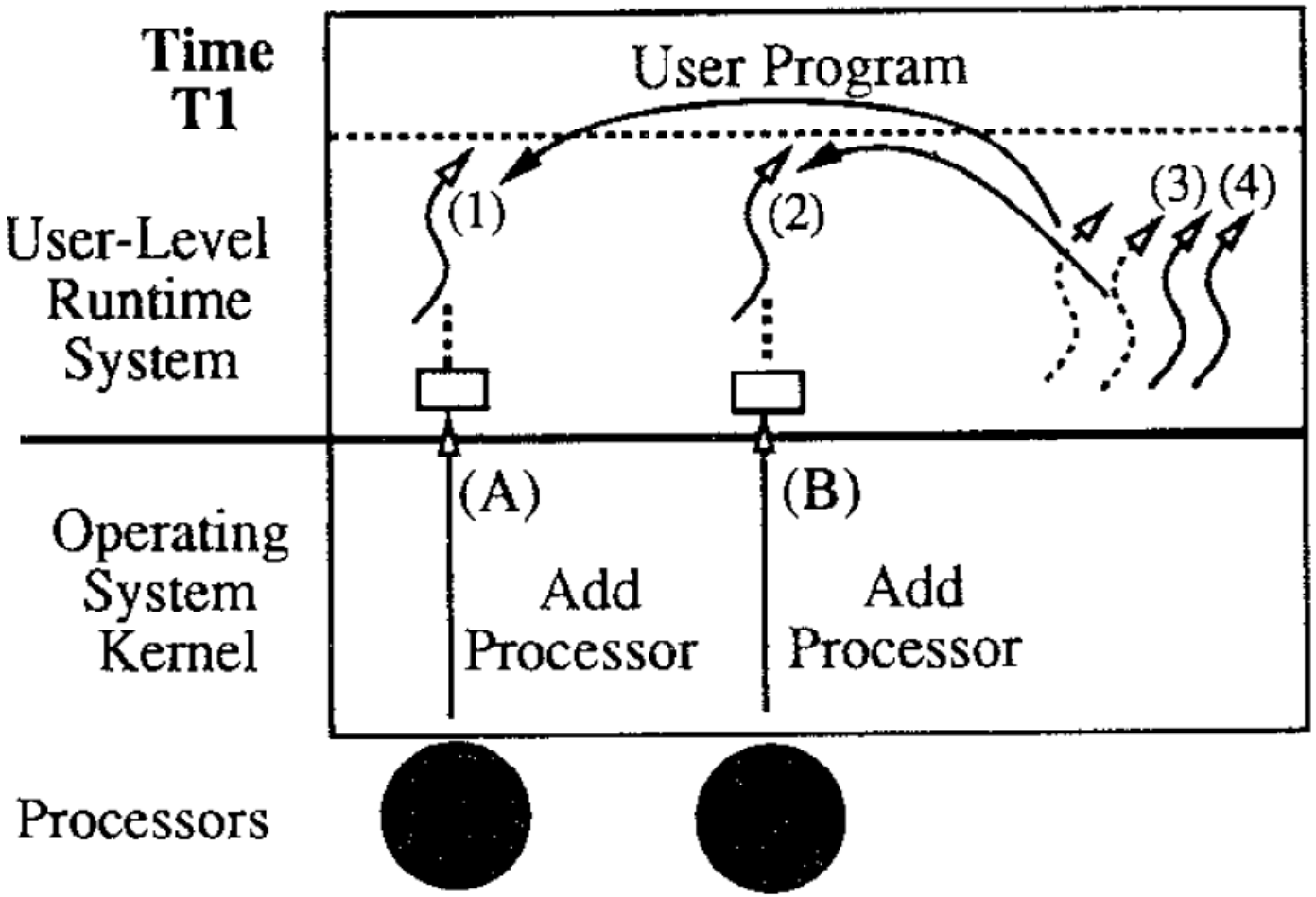
- **Kernel-level threads: well-integrated, but slow.**
  - Kernel has full view of threads+resources, manages everything.
  - The cost is **inherent**: protection, traps, dilemma of generality.
- **ULTS threads: fast, but poorly integrated.**
  - Can't react to changing number of cores.
  - Can't manage your cores.
  - Can't use spinlocks or priority.
  - Can't overlap compute and I/O.
- Scheduler Activations model yields the benefits of both and the drawbacks of neither.
- **Yes, you can have it all.**

A classic systems paper

- The solution is transparent to programmers.
  - Library implements familiar program abstraction: **threads**.
  - Kernel handles machine-defined objects: **cores** and **contexts**.
  - Support for the programming model is decoupled from kernel.
- Rewire your brain, rethink the foundations.
- This landmark paper launched direction of **library OS**.
  - Beyond microkernels
  - Exokernel (MIT 1995-): ”no abstractions” for kernel, just raw resources and protected control of the machine.
  - Trusted execution environments—enclaves. Haven, Scone.
  - Unikernels, kernels for high-speed server VMs. Dune, Arrakis.
  - Serverless cloud computing

Scheduler Activations.
Anderson et. al. SOSP 1991

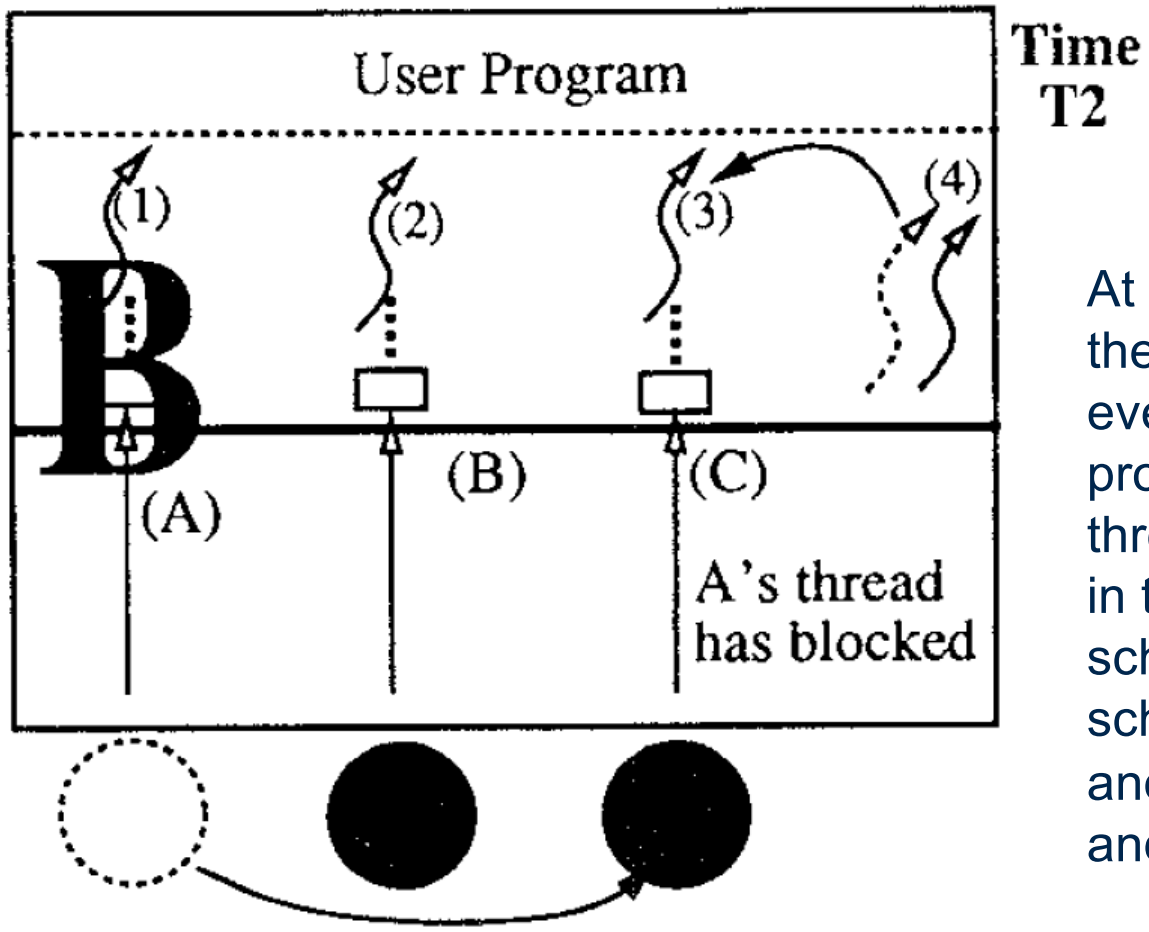
Example (T1)



At time TI, the kernel allocates the application two processors (cores). On each core, the kernel upcalls to ULTS code that picks a thread from the ready list and starts running it.

Scheduler Activations.
Anderson et. al. SOSP 1991

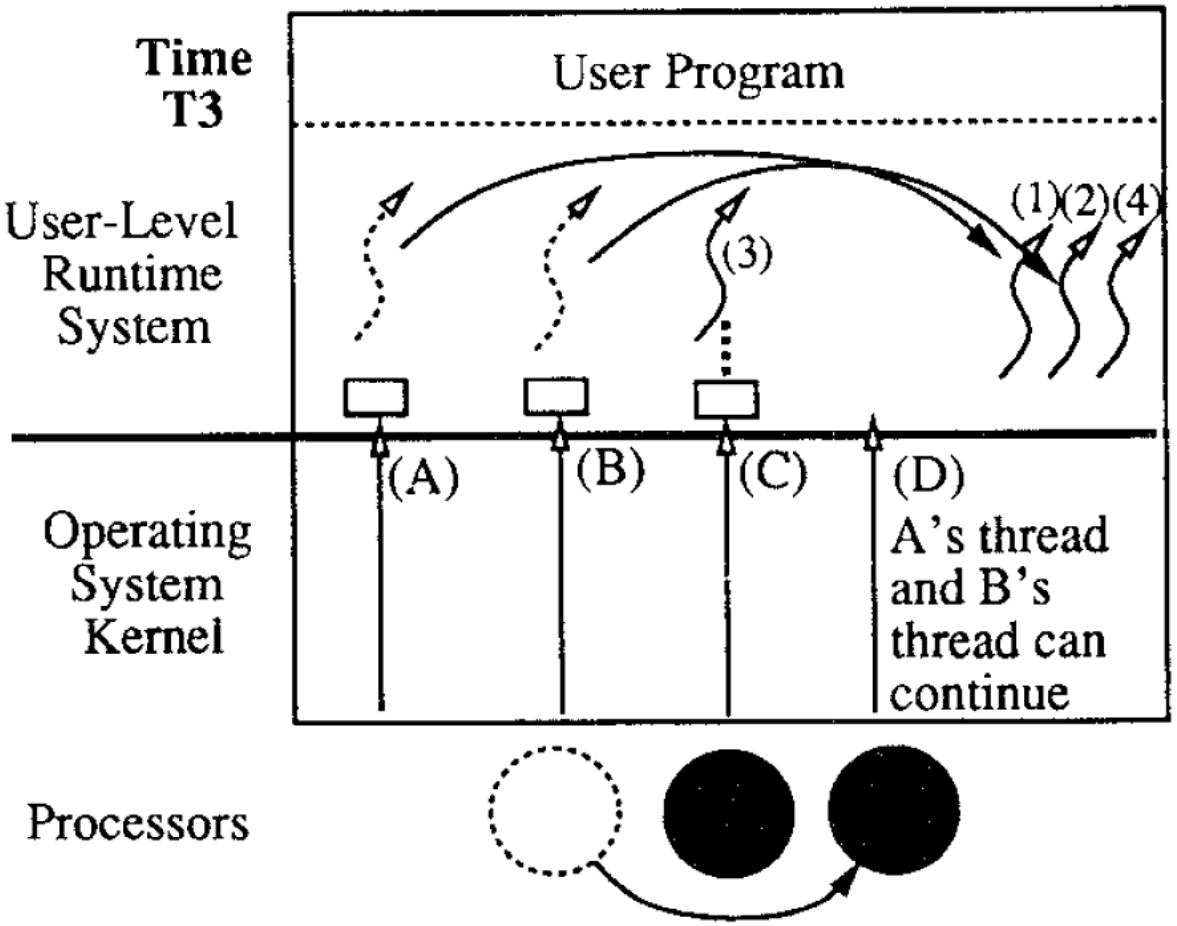
Example (T2)



At time T2, thread 1 **blocks** in the kernel. To notify ULTS of this event, the kernel takes the processor that had been running thread 1 and performs an upcall in the context of a fresh scheduler activation. The ULTS scheduler uses the core to take another thread off the ready list and start running it.

Scheduler Activations.
Anderson et. al. SOSP 1991

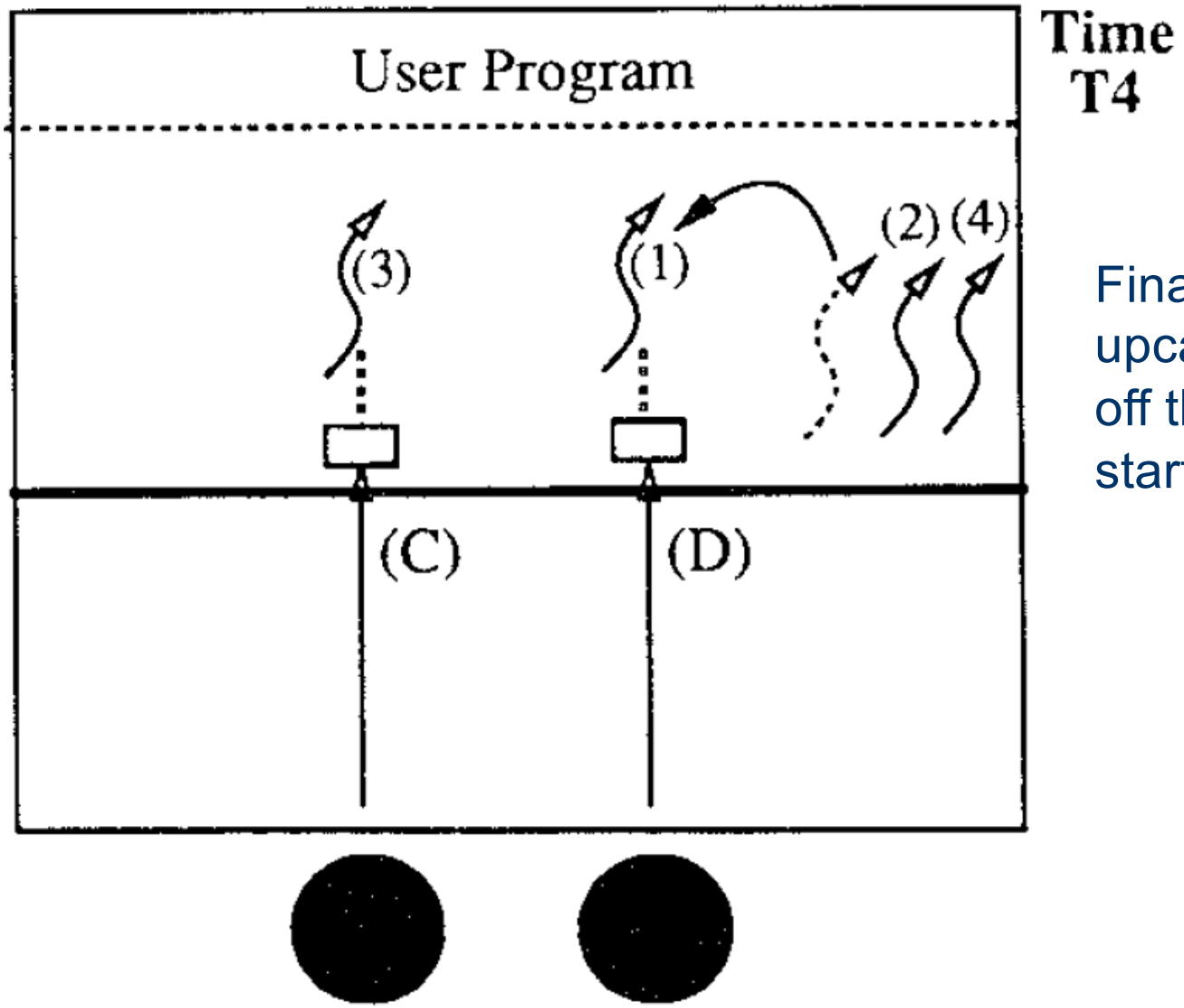
Example (T3)



At time T3, the **I/O completes**. Kernel notifies ULTS of the event, but this requires a core. The kernel preempts one of the cores running in the VAS and uses it to do the upcall. (If there are no cores assigned, the upcall must wait until the kernel allocates one). This upcall notifies ULTS of two things: the I/0 completion and the preemption. The upcall invokes code in the ULTS that (1) puts the thread that had been blocked on the ready list and (2) puts the thread that was preempted on the ready list.

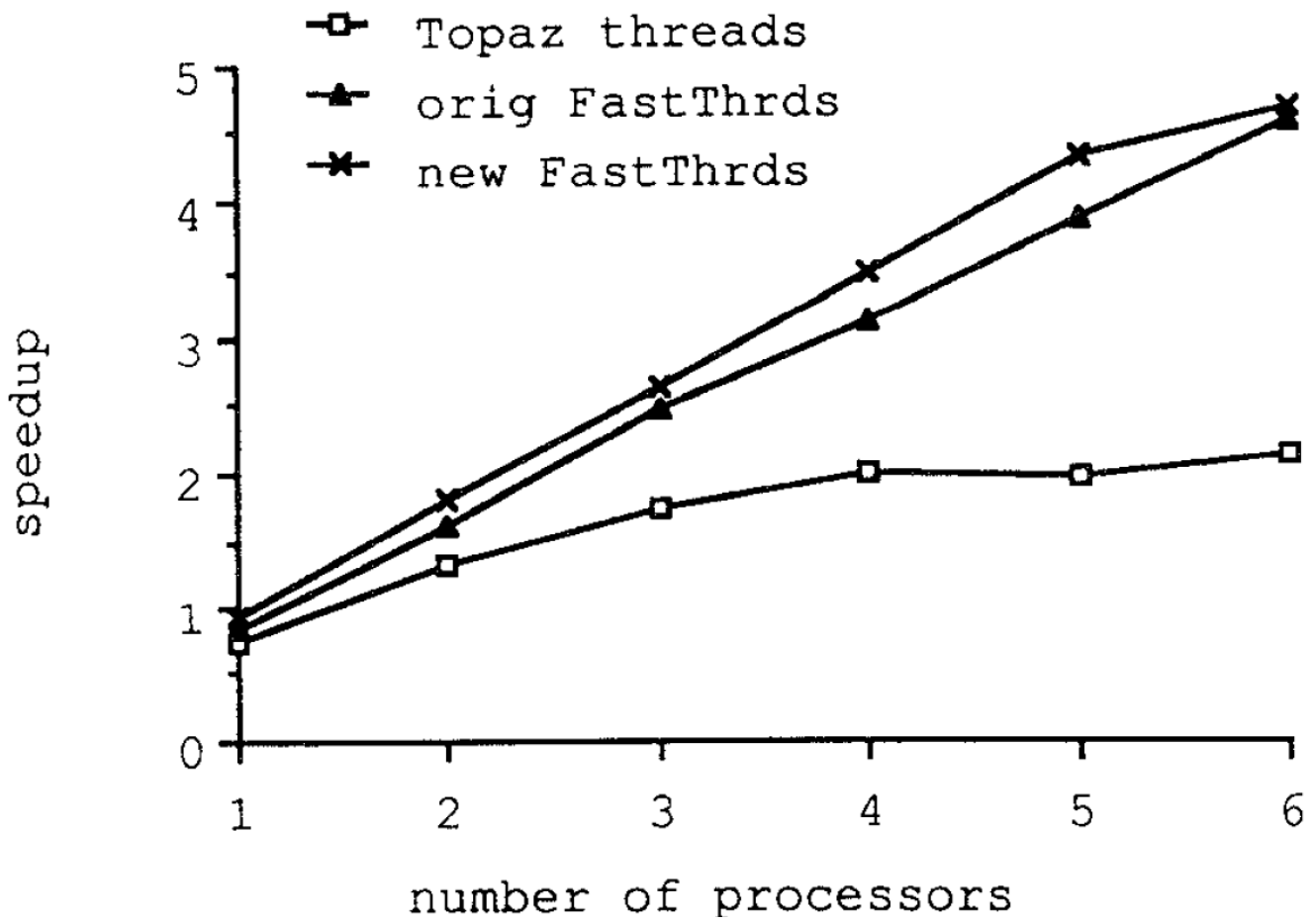
Scheduler Activations.
Anderson et. al. SOSP 1991

Example (T4)



Finally, at time T4, the upcall takes a thread off the ready list and starts running it.

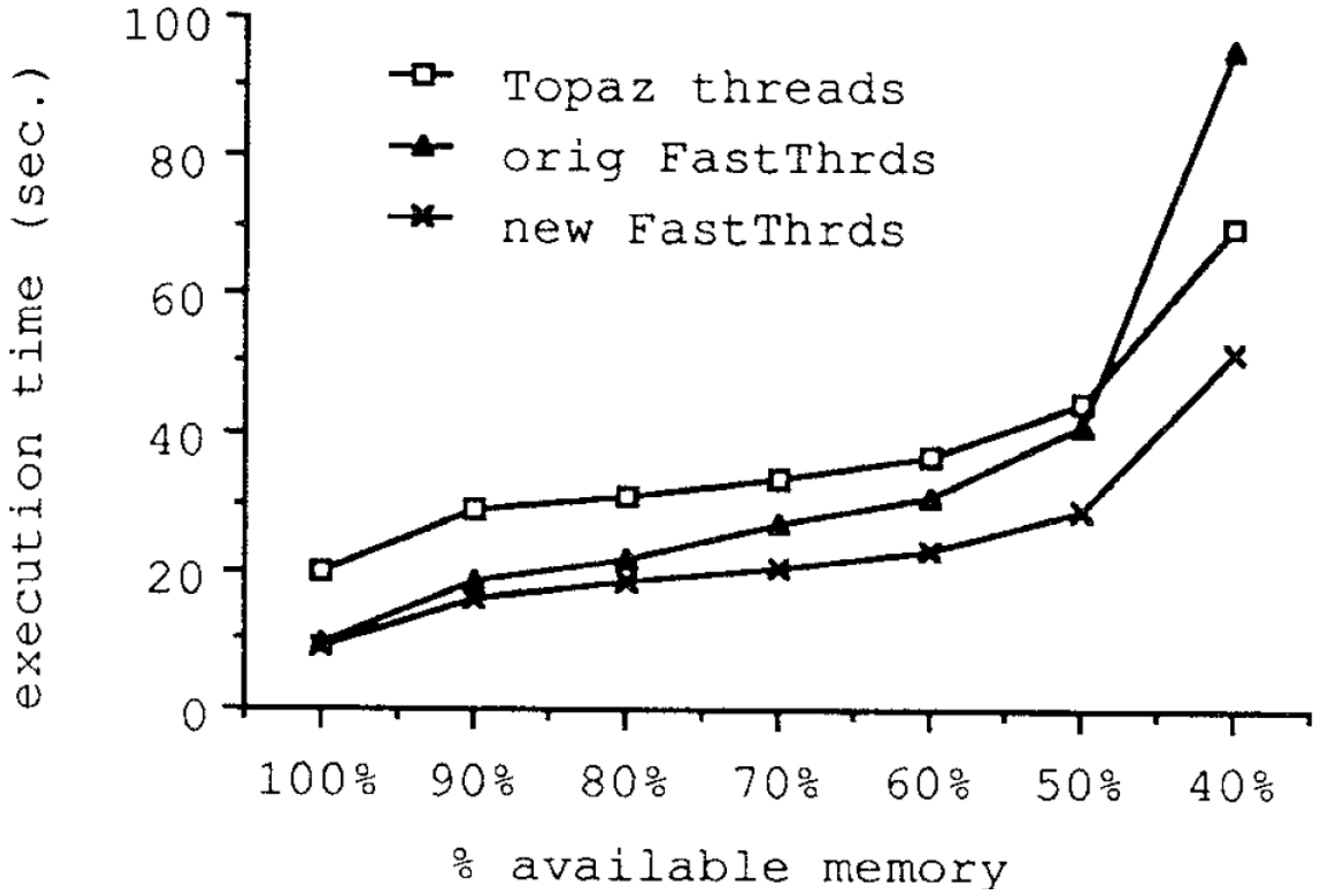
Speedup: compute-bound



No spin-waiting for Topaz so lots of overhead.

New FastThreads better for less than 6 cores due to poor scheduling of Topaz. Daemon thread causes preemption, even if processors avail.

Results: I/O bound



FastThreads degrades due to inability to reallocate core when thread blocks.

Topaz has poorer performance due to kernel involvement.

Multiprogramming

- Kernel must (re)allocate cores among multiple contending “virtual MPs” (processes).
- Two processes, N cores, each should speed up by N/2.
- Results in paper give a few small data points suggesting that unmanaged timeslicing → underperforms.
- And FastThreads on Scheduler Activations approaches the expected speedup.