Threads and Concurrency

Chapter 4 OSPP

Part I

Announcements

- HW #1 due (on Thursday)
- Questions?

- Today: through 4.6.2

Motivation

- Operating systems (and application programs) often need to be able to handle multiple things happening at the same time
  - Process execution, interrupts, background tasks, system maintenance
- Humans are not very good at keeping track of multiple things happening simultaneously
- Threads are an abstraction to help bridge this gap

Why Concurrency?

- Servers
  - Multiple connections handled simultaneously
- Parallel programs
  - To achieve better performance
- Programs with user interfaces
  - To achieve user responsiveness while doing computation
- Network and disk bound programs
  - To hide network/disk latency

Definitions

- A thread is a single execution sequence that represents a separately schedulable task
  - Single execution sequence: familiar programming model
  - Separately schedulable: OS can run or suspend a thread at any time
- Protection is an orthogonal concept
  - Can have one or many threads per protection domain

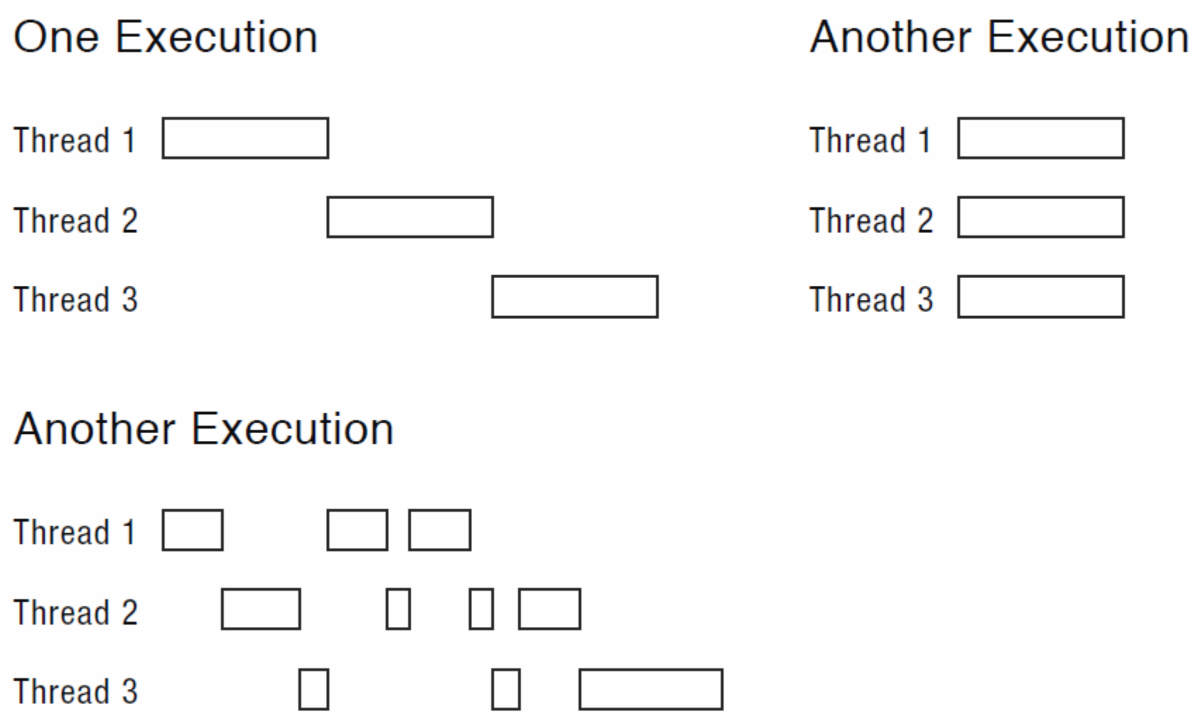
Hmmm: sounds familiar

- Is it a kind of interrupt handler?
- How is it different?
  - How execution is triggered.
  - Can be long running.

Threads in the Kernel and at User-Level

- Multi-threaded kernel
  - multiple threads, sharing kernel data structures, capable of using privileged instructions
- Multiprocessing kernel
  - Multiple single-threaded processes
  - System calls access shared kernel data structures
- Multiple multi-threaded user processes
  - Each with multiple threads, sharing same data structures, isolated from other user processes
  - Threads can be user-provided or kernel-provided

Possible Executions



Thread Operations

- `thread_create` (thread, func, args)
  - Create a new thread to run func(args)
- `thread_yield` ()
  - Relinquish processor voluntarily
- `thread_join` (thread)
  - In parent, wait for forked thread to exit, then return
- `thread_exit`
  - Quit thread and clean up, wake up joiner if any

Example: threadHello

```
#define NTHREADS 10
thread_t threads[NTHREADS];
main() {
    for (i = 0; i < NTHREADS; i++)
        thread_create(&threads[i], &go, i);
    for (i = 0; i < NTHREADS; i++) {
        exitValue = thread_join(threads[i]);
        printf("Thread %d returned with %ld\n", i,
               exitValue);
    }
    printf("Main thread done.\n");
}
void go (int n) {
    printf("Hello from thread %d\n", n);
    thread_exit(100 + n);
    // REACHED?
}
```

threadHello: Example Output

- Why must "thread returned" print in order?
  - What is maximum # of threads running when thread 5 prints hello?
  - Minimum?

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

Fork/Join Concurrency

- Threads can create children, and wait for their completion
- Data only shared before fork/after join
- Examples:
  - Web server: fork a new thread for every new connection
    - As long as the threads are completely independent
  - Merge sort
  - Parallel memory copy

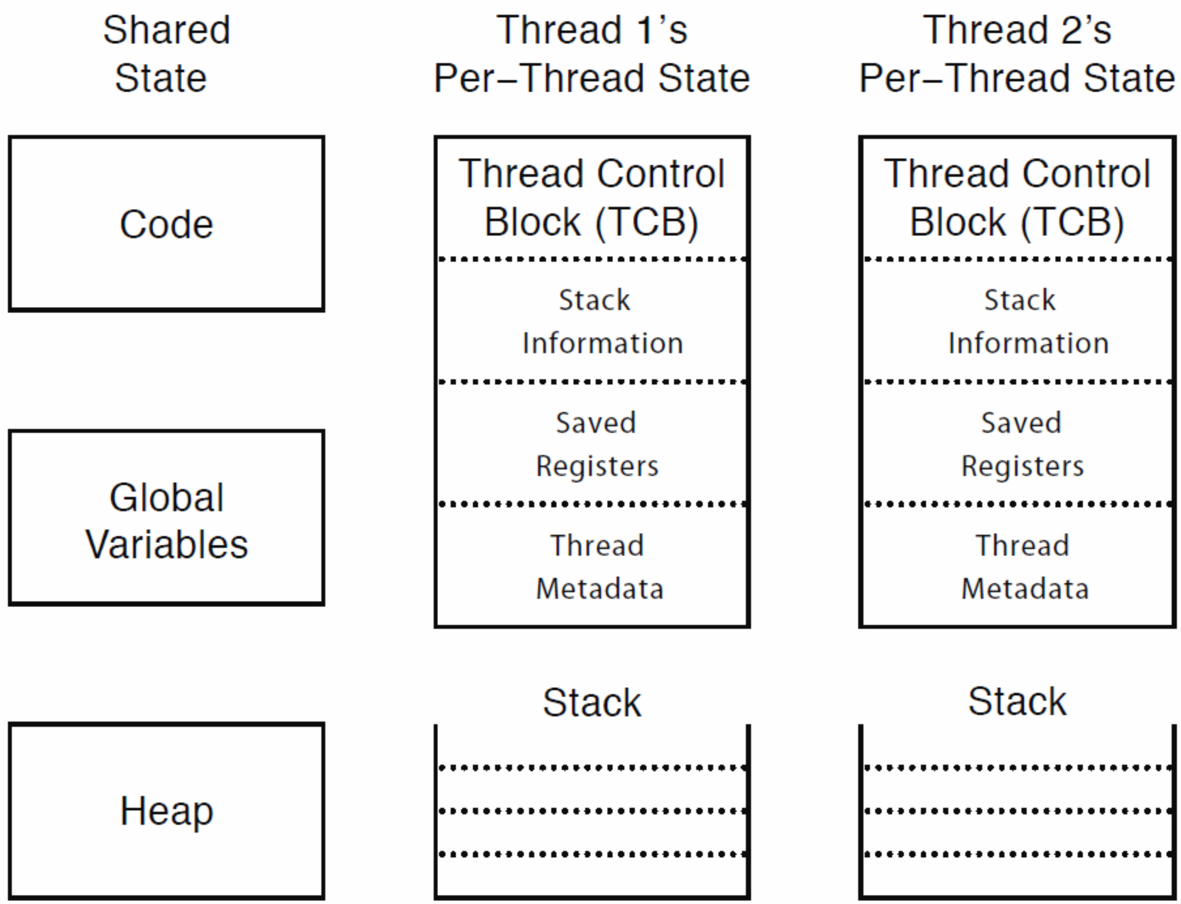
Example

- Zeroing memory of a process
- Why?
  - At process exit, prevent leakage

bzero with fork/join concurrency

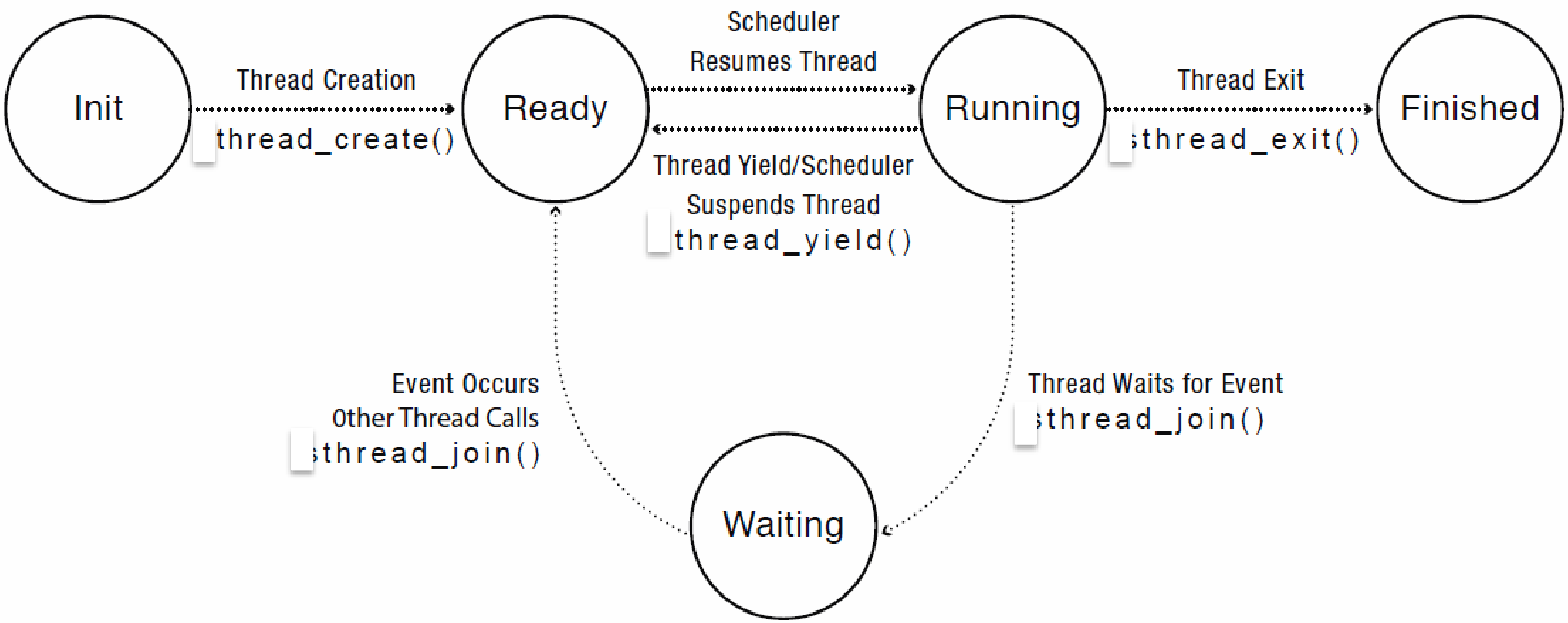
```
void blockzero (unsigned char *p, int length) {
    int i, j;
    thread_t threads[NTHREADS];
    struct bzeroparams params[NTHREADS];

// For simplicity, assumes length is divisible by NTHREADS.
for (i = 0, j = 0; i < NTHREADS; i++, j +=
     length/NTHREADS) {
        params[i].buffer = p + i * length/NTHREADS;
        params[i].length = length/NTHREADS;
        thread_create_p(&(threads[i]), &zero_go,
                                  &params[i]);
    }
    for (i = 0; i < NTHREADS; i++) {
        thread_join(threads[i]);
    }
}
```

Thread Data Structures



id, status, …

Thread Lifecycle



Thread Scheduling

- When a thread blocks or yields or is de-scheduled by the system, which one is picked to run next?
- Preemptive scheduling: preempt a running thread
- Non-preemptive: thread runs until it yields or blocks
- *Idle* thread runs until some thread is ready …
- Priorities? All threads may not be equal
  - e.g. can make bzero threads low priority (background)

Thread Scheduling (cont’d)

- Priority scheduling
  - threads have a priority
  - scheduler selects thread with highest priority to run
  - preemptive or non-preemptive

- Priority inversion
  - 3 threads, t1, t2, and t3 (priority order – low to high)
  - t1 is holding a resource (lock) that t3 needs
  - t3 is obviously blocked
  - t2 keeps on running!

- How did t1 get lock before t3?

How would you solve it?

- Think about it – will discuss next class

Threads and Concurrency

Chapter 4 OSPP

Part II

Thread Scheduling

- Priority scheduling
  - threads have a priority
  - scheduler selects thread with highest priority to run
  - preemptive or non-preemptive

- Priority inversion
  - 3 threads, t1, t2, and t3 (priority order – low to high)
  - t1 is holding a resource (lock) that t3 needs
  - t3 is obviously blocked
  - t2 keeps on running!

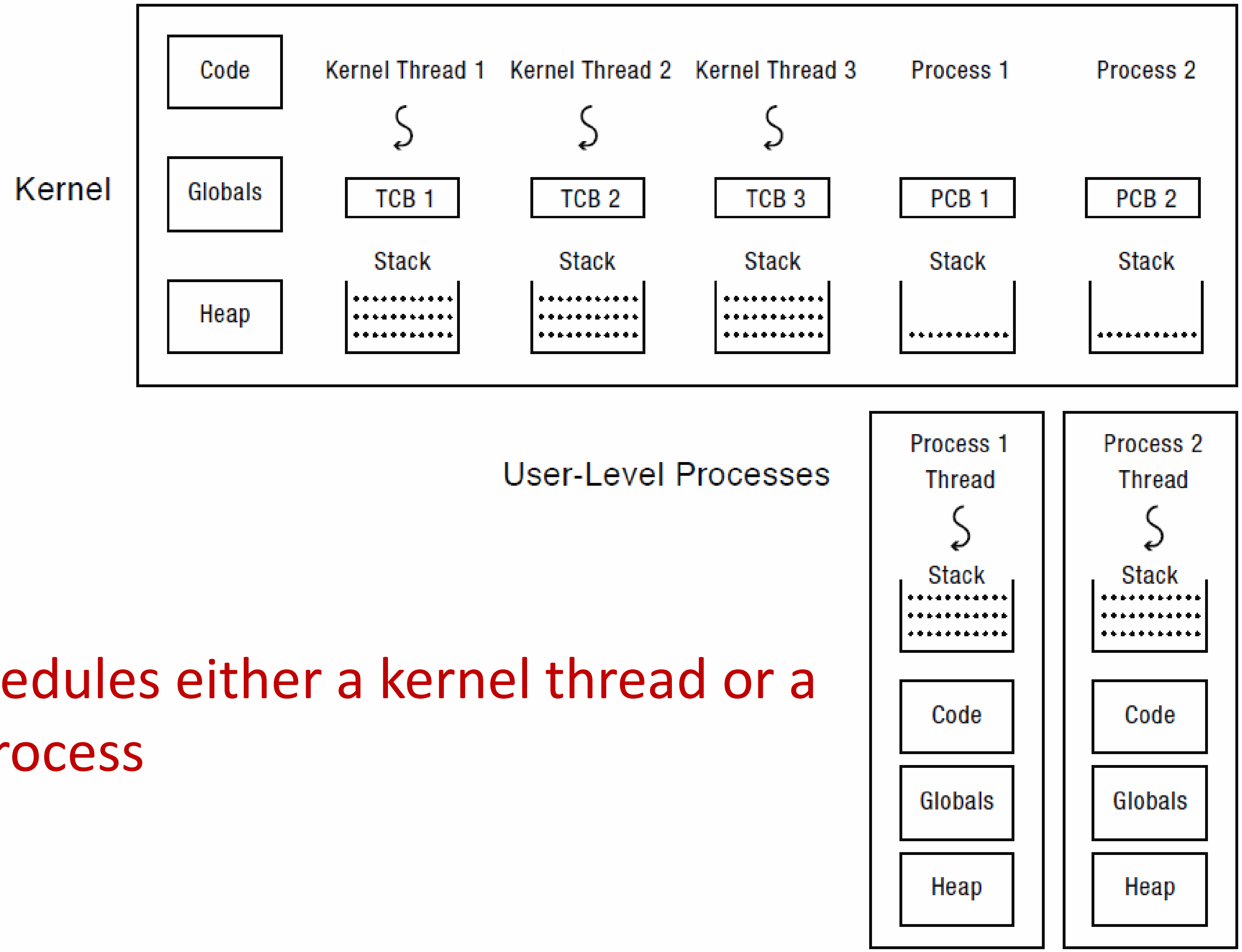
How would you solve it?

- Idea 1
  - One solution involves temporarily raising priority level of T1>T2 until it releases the lock L
  - Requires that the system be checking for this condition! Windows looks for starving threads (been on the ready list too long)
- Idea 2
  - Automatically elevate priority of T1 to priority of the resource L while it holds L
  - L will have a priority as high as the highest thread that acquires it

Implementing Threads: Roadmap

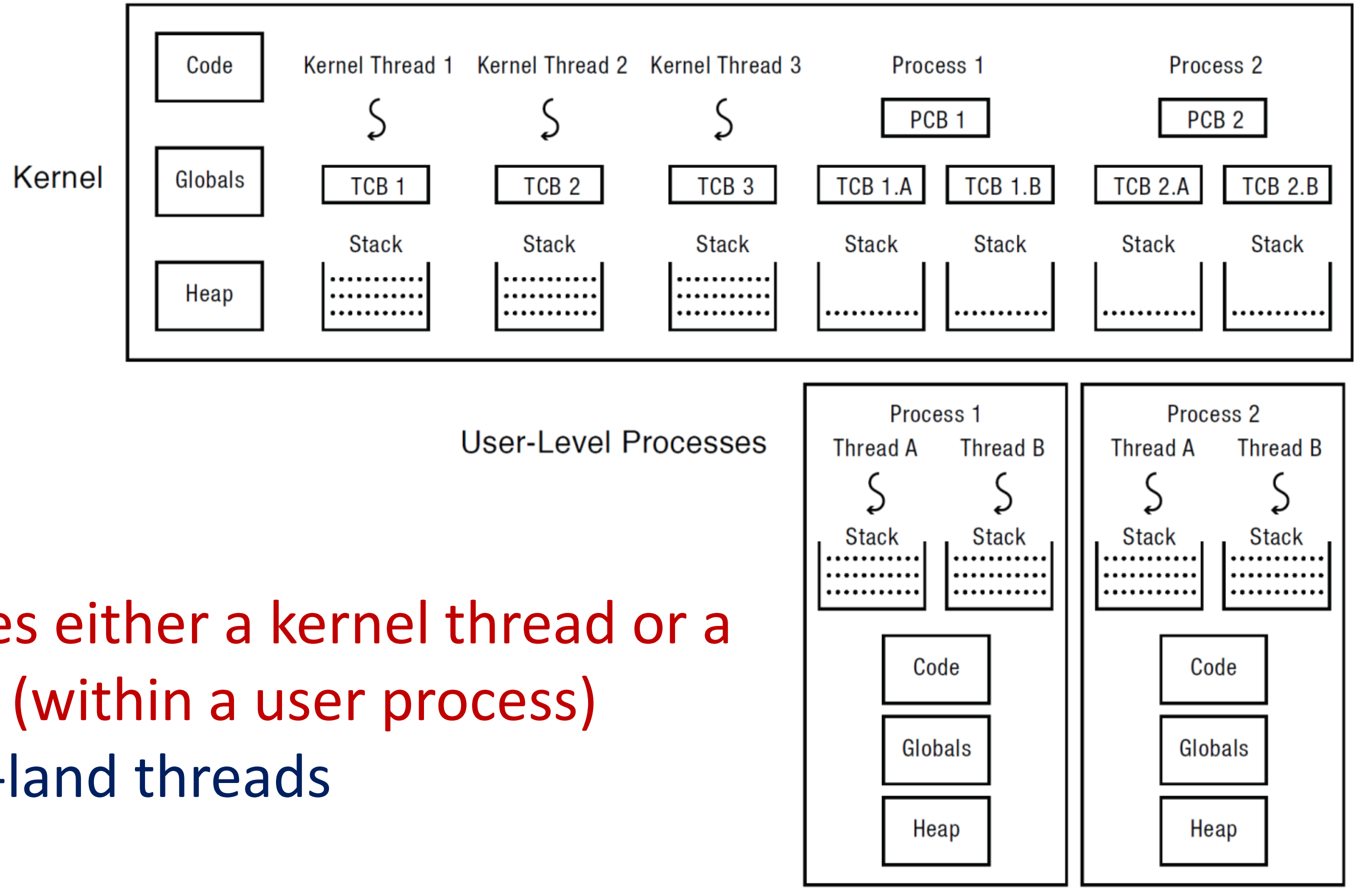
- Kernel threads + single threaded process
  - Thread abstraction only available to kernel
  - To the kernel, a kernel thread and a single threaded user process look quite similar
- Multithreaded processes using kernel threads
  - Linux, MacOS
  - Kernel thread operations available via syscall
- Multithreaded processes using user-level threads
  - Thread operations without system calls

Multithreaded OS Kernel; Single threaded process (i.e. no threads)



OS schedules either a kernel thread or a user process

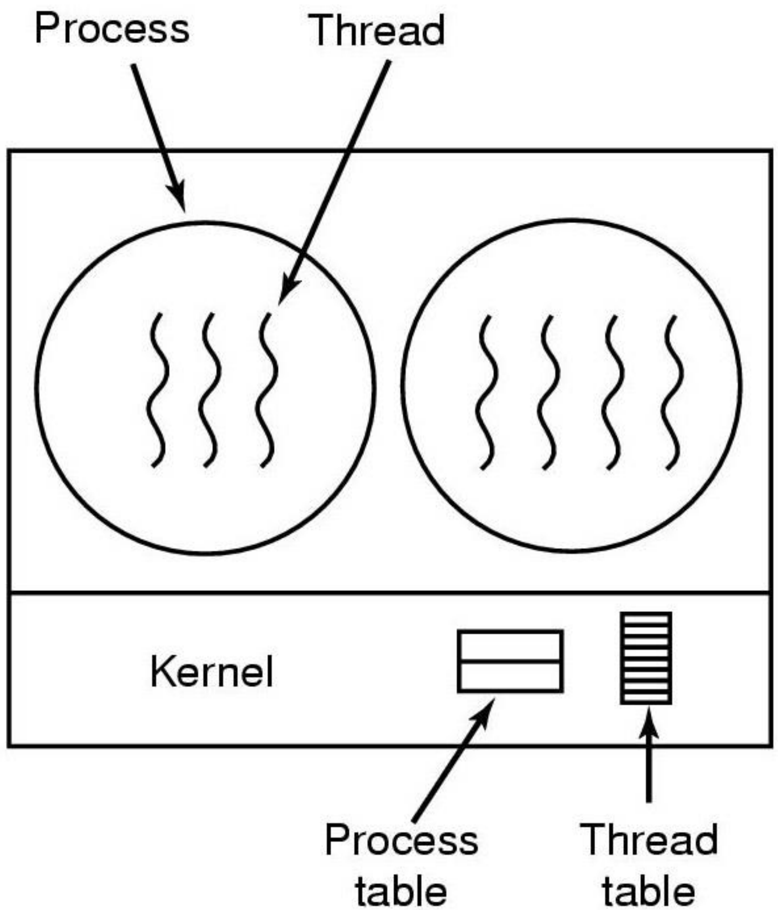
Multithreaded processes using kernel threads



OS schedules either a kernel thread or a user thread (within a user process)

no user-land threads

Implementing Threads in the Kernel

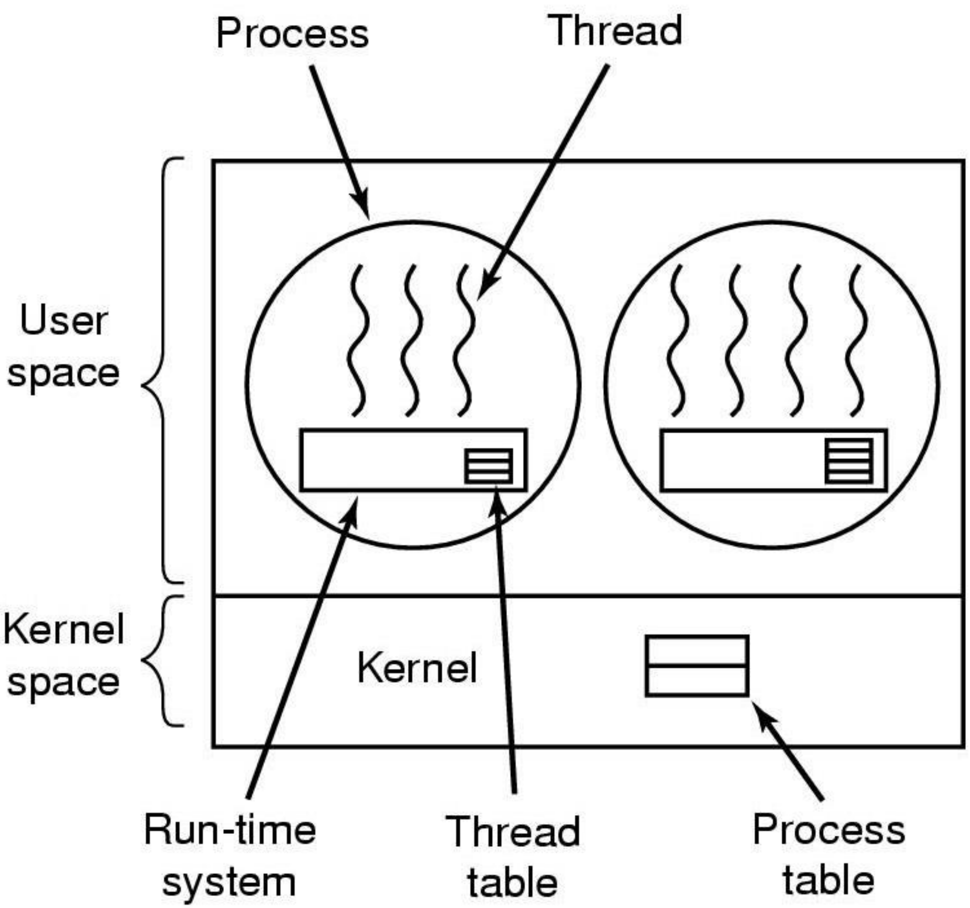


A threads package managed by the kernel

Implementing Threads Purely in User Space

OS schedules either a kernel thread or a user process (user library schedules threads)

user-land case



A user-level threads package

Kernel threads

- All thread management done in kernel
- Scheduling is usually preemptive

- Pros:
  - can block!
  - when a thread blocks or yields, kernel can select any thread from same process or another process to run

- Cons:
  - cost: better than processes, worse than procedure call
  - fundamental limit on how many – why
  - param checking of system calls vs. library call – why is this a problem?

User threads

- User
  - OS has no knowledge of threads
  - all thread management done by run-time library
- Pros:
  - more flexible scheduling
  - more portable
  - more efficient
  - custom stack/resources
- Cons:
  - blocking is a problem!
  - need special system calls!
  - poor sys integration: can’t exploit multiprocessor/multicore as easily

Implementing threads

- `thread_fork(func, args)` [create]
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)

- stub (func, args)
  - Call (*func)(args)
  - If return, call `thread_exit()`

- Thread create code

Implementing threads (cont’d)

- `thread_exit`
  - Remove thread from the ready list so that it will never run again
  - Free the per-thread state allocated for the thread

- Why can’t thread itself do the freeing?
  - deallocate stack: can’t resume execution after an interrupt
  - mark us finished and have another thread clean us up

Thread Stack

- What if a thread puts too many procedures or data on its stack?
  - User stack uses virt. memory: tempting to be greedy
  - Problem: many threads
  - Limit large objects on the stack (make static or put on the heap)
  - Limit number of threads
- Kernel threads use physical memory and they are *really* careful

Problems with Sharing: Per thread locals

- errno is a problem!
  - errno (thread_id) …
  - give each thread a copy of certain globals

- Heap
  - shared heap
  - local heap : allows concurrent allocation (nice on a multiprocessor)

Thread Context Switch

- Voluntary
  - `thread_yield`
  - `thread_join` (if child is not done yet)
- Involuntary
  - Interrupt or exception or blocking
  - Some other thread is higher priority

Voluntary thread context switch

- Save registers on old stack
- Switch to new stack, new thread
- Restore registers from new stack
- Return (pops return address off the stack, ie. sets PC)
- Exactly the same with kernel threads or user threads

x86 switch_threads

[Thread switch code](): high level

```
# Save caller's register state
#  NOTE: %eax, etc. are ephemeral
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# Get offsetof (struct thread, stack)
mov thread_stack_ofs, %edx
# Save current stack pointer to old
    thread's stack, if any.
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)
#esp saved into TCB
```

```
# Change stack pointer to new
    thread's stack
# this also changes currentThread
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp
#TCB esp moved to esp

# Restore caller's register state.
popl %edi
popl %esi
popl %ebp
popl %ebx
#tricky flow
ret
```

yield

- [Thread yield code]
- Why is state set to running and for whom?
- Who turns interrupts back on?
- Note: this function is reentrant!

Threads and Concurrency

Chapter 4 OSPP

Part II

Today

- Last time, `thread_yield`
- Today
  - wrap up threads (fast)
  - start synchronization

`thread_join`

- Block until children are finished
- System call into the kernel
  - May have to block
- Nice optimization:
  - If children are done, store their return values in user address space
  - Why is that useful?
  - Or spin a few υs before actually calling `join`

Multithreaded User Processes (Take 1)

- User thread = kernel thread (Linux, MacOS)
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switch
  - Simple, but a lot of transitions between user and kernel mode
  - + block, +multiprocessors

Multithreaded User Processes (Take 2)

- Green threads (early Java)
  - User-level library, within a single-threaded process
  - Blocking is tricky!
  - Library does thread context switch
  - Preemption via upcall/UNIX signal on timer interrupt
  - Use multiple processes for parallelism
    - Shared memory region mapped into each process

Multithreaded User Processes (Take 3)

- Scheduler activations (Windows 8)
  - Kernel allocates vprocessors to user-level library
  - User thread library implements context switch
  - User thread library decides what thread to run next

- Upcall whenever kernel needs a user-level scheduling decision
  - User process assigned a new vprocessor
  - vprocessor removed from process
  - System call blocks in kernel

Best of Both Worlds

- Scheduler Activations

Scheduler Activations

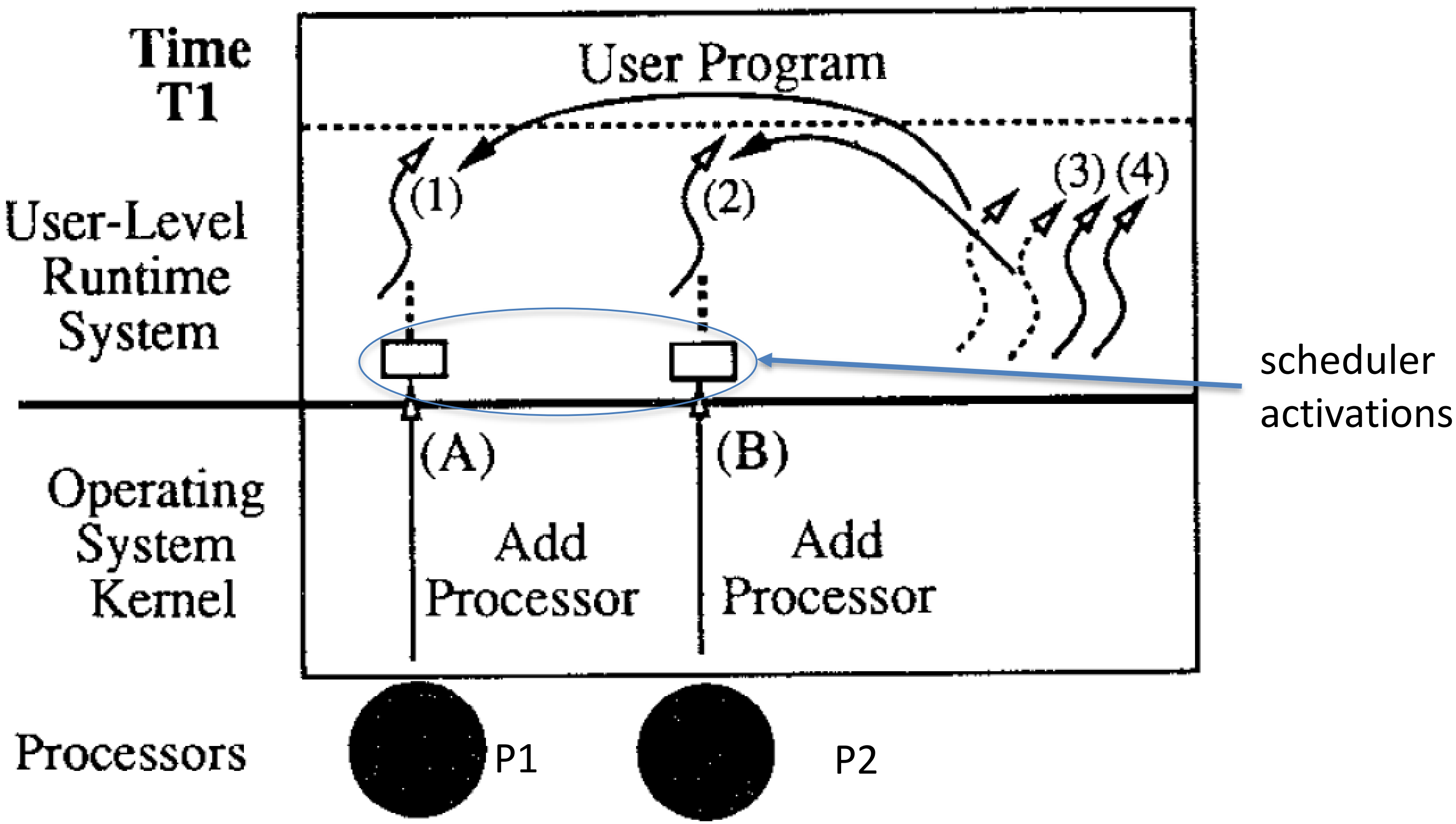
- Idea:
  - Create a structure that allows information to flow between:
  - user-space (thread library) and kernel

- One-way flow is common … system call
- Other way is uncommon …. upcall

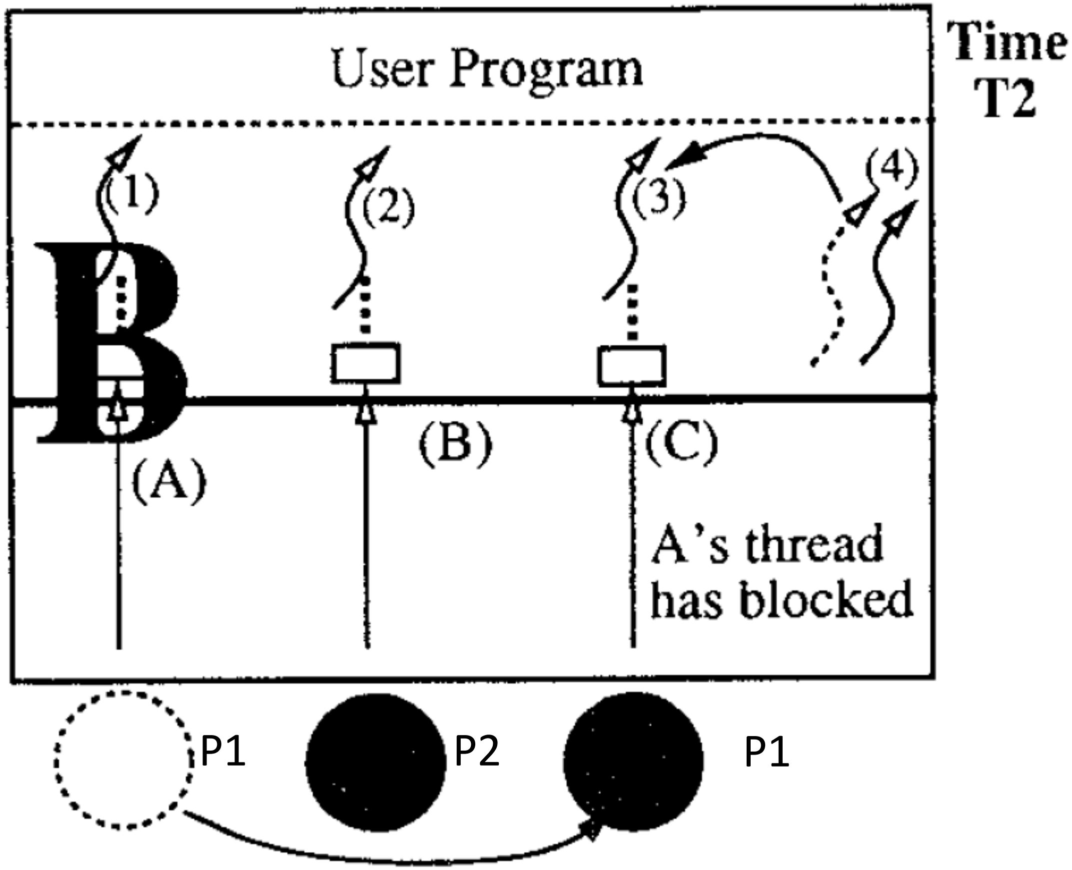
Scheduler Activations Cont'd

- Two new things:
- Activation: structure that allows information/events to flow (holds key information, e.g. stacks)
- Virtual processor: abstraction of a physical machine; gets "allocated" to an application
  - means any threads attached to it will run on that processor
  - want to run on multiple processors – ask OS for > 1 VP

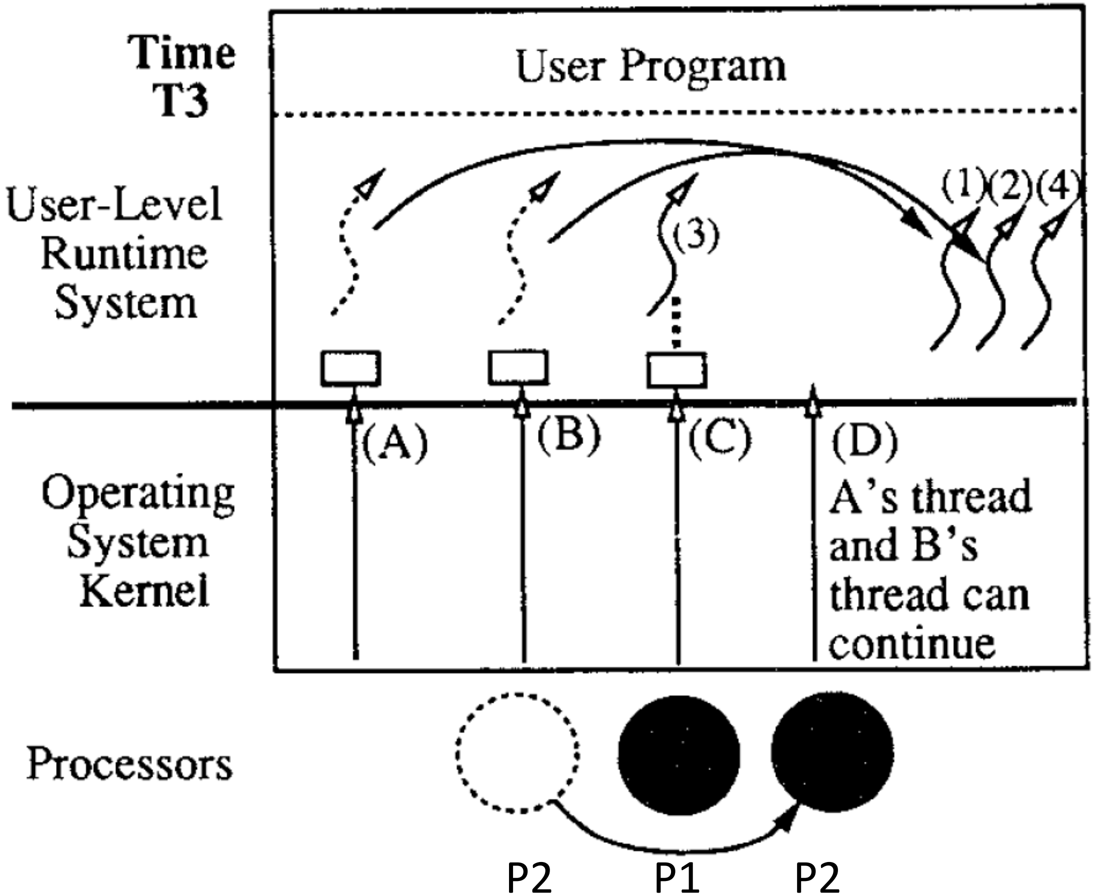
Example



- Kernel provides two processors to the application
  - upcall to scheduler: user library picks two threads to run ….
- Now, suppose T1 blocks ….

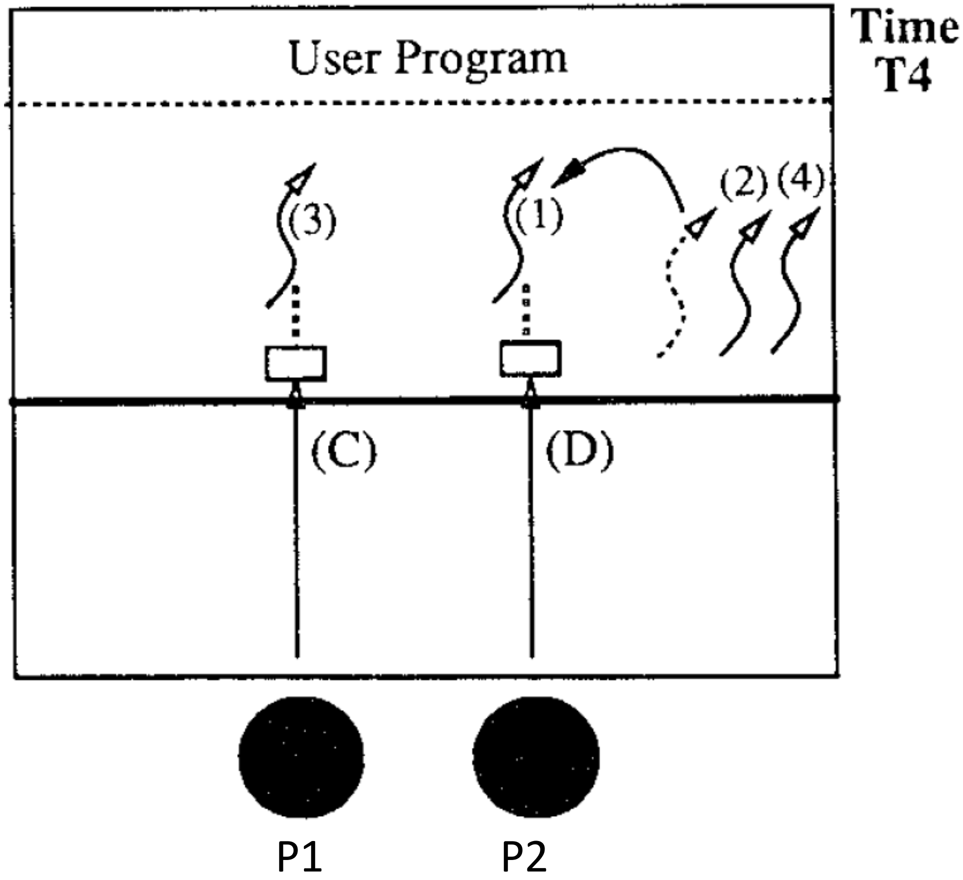


- T1 blocks in the kernel
  - kernel creates a SA; makes upcall on the processor running T1
  - user-level scheduler picks another thread (T3) to run on that processor
  - T1 put on blocked list



- I/O for (T1) completes
  - Notification requires a processor; kernel preempts one of them (P2 – T2), does upcall
  - Problem : suppose no processors! – must wait until kernel gives one
  - Two threads back on the ready list! (T1 and T2: why?)

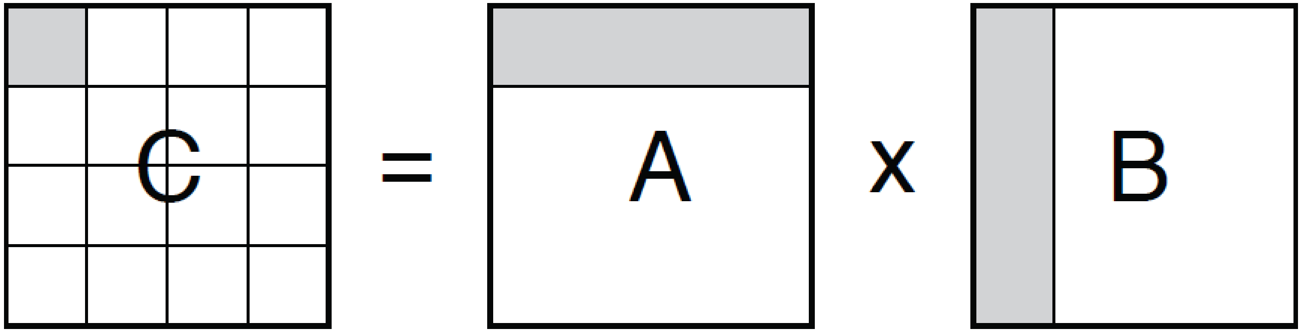
Example



- User library picks a thread to run (resume T1)

Alternative Abstractions

- Asynchronous I/O and event-driven programming
- Data parallel programming
  - All processors perform same instructions in parallel on a different part of the data



  - Have you seen this before?
    - bzero

Event-driven

- Poll or interrupts (Signals)
- Non-blocking I/O events get initiated
  - e.g. initiated by `aio_read`'s
- Check/wait for I/O event completion/arrival
  - e.g. can poll and/or block smartly: e.g. Unix `select`
  - e.g. can await a signal SIGIO
- Thread way
  - Just create threads and have them do blocking synchronous calls (e.g. `read`)

Performance Comparison

- Event-driven: explicit state management vs. automatic state savings in threads
- Responsiveness
  - Large tasks may have to be decomposed for event-driven programming to efficiently save state
- Performance: latency
  - thread could be slower due to stack allocation, but gap is closing particularly with user threads
- Performance: parallelism
  - events only work with a single core! but are great for servers that need to multiplex cores