

Threads

Philip Levis
CS240E

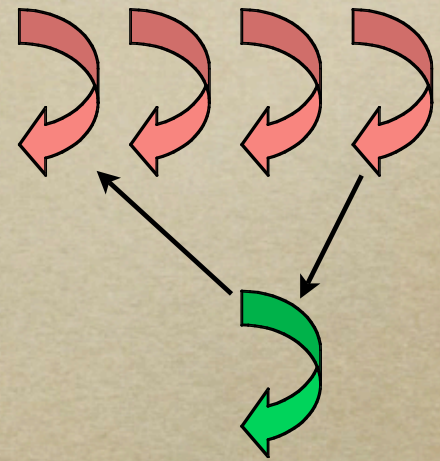
Elements of Consideration

- Processor: what actually runs a code stream
- Thread: an execution context for a processor
- Function: a particular code stream

- “Schedule a thread:” decide which thread runs on a processor
- “Spawn a thread to X:” allocate a new thread to execute function X, and submit it to the OS to schedule

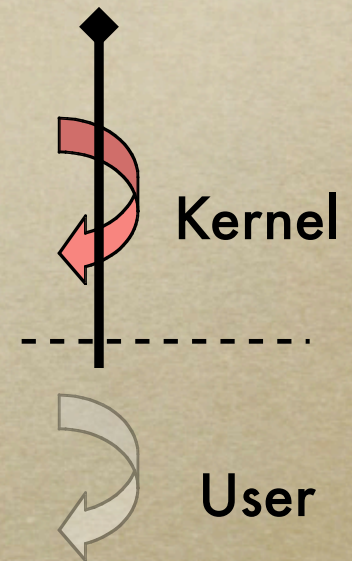
A Thread Context

- Stores the execution state of a code stream
- Three basic parts:
 - * Stack: private data from execution
 - * Registers: saving and restoring the thread
 - * Thread control block: metadata (id, etc.)
- When the OS schedules a thread, it stores the registers of the current one and restores the registers of the new one
 - * New thread starts running just where it left off
 - * `schedule()`: the function that returns at some point in the future...



Blocking Calls

- Sometimes, a thread asks the kernel to do something that doesn't require executing instructions
 - * Read a block off disk
 - * Receive a packet from the network
 - * These calls "block" in the kernel
- The kernel starts the operation, puts the thread on a wait queue for when the operation completes, and schedules a new thread to run
 - * It's as if the call to `schedule()` only returns when the operation is complete: the stack and data registers are maintained



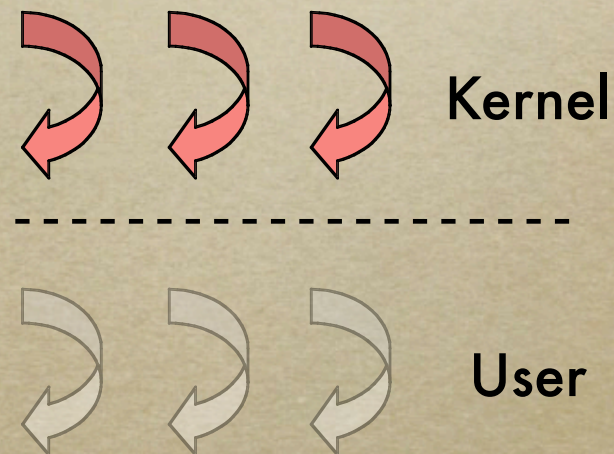
Asynchronous Calls

- Thread makes a call to start an operation
 - * Call returns, does not block
- Thread can later check if operation is complete
 - * Polling
 - * Block on wait()
- Allows a single kernel thread to enqueue outstanding operations without blocking



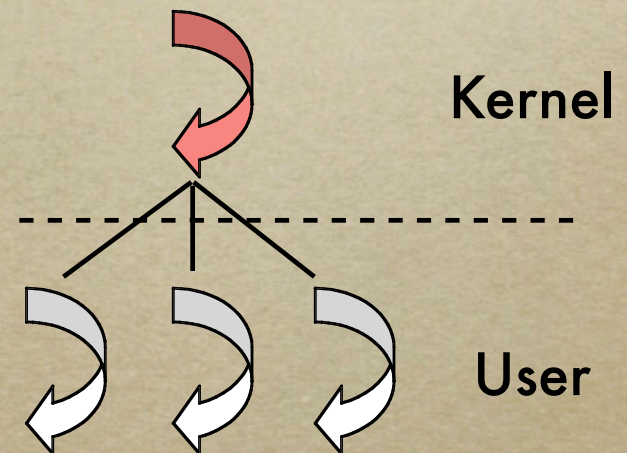
User vs. Kernel (1 to 1)

- In all thread models, the kernel schedules kernel-level threads
 - * The kernel decides which one to run, when to stop them, etc.
- In the simplest model, each thread visible to the user has an associated kernel thread (1 to 1)
- User code can allocate multiple threads, but do not control which one runs when
- Each thread has a single context, stored in the kernel



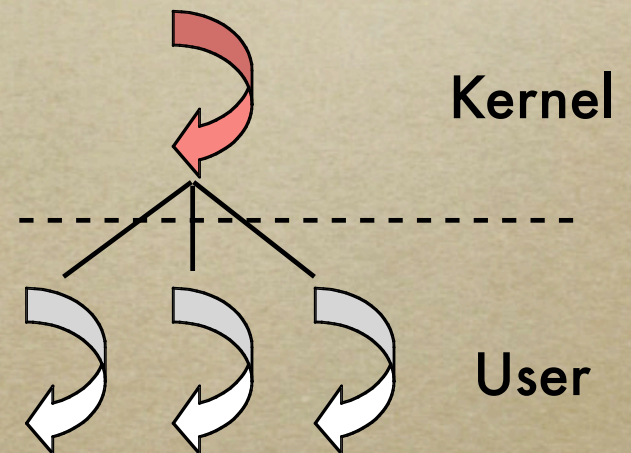
User vs. Kernel (n to 1)

- If your code makes blocking calls, then in order to highly utilize the processor you need multiple kernel threads
- But what if you want concurrency just for programming simplicity? E.g., spawn a thread each for functions A and B, rather than having to queue B for later execution.
- User-level threads store additional user-level thread contexts, which can be multiplexed on a single kernel thread (n to 1)



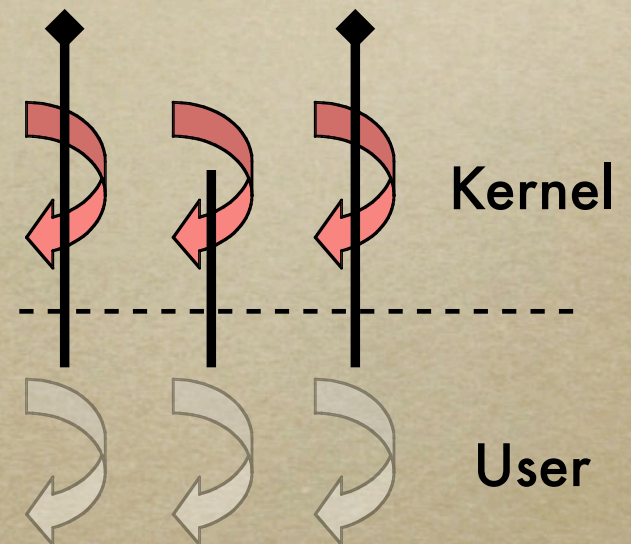
User vs. Kernel (n to 1)

- The kernel only sees one thread: two user threads can never run at the same time
- If any user thread issues a blocking call, all of them stop as the kernel puts the kernel thread to sleep
 - * Capriccio follows an n to 1 model and makes sure that user threads never issue blocking calls: only asynchronous I/O
- Why does this require storing N user contexts but 1-to-1 doesn't? Could you use N-1?



Kernel Concurrency

- When does the kernel consider a kernel thread valid to schedule?
 - * When executing user code?
 - * When executing kernel code?
- Preempting code in the kernel can leave data structures inconsistent
 - * Whole kernel lock
 - * Subsystem lock
 - * Individual data structure lock



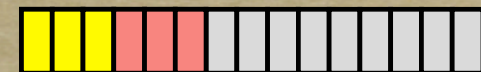
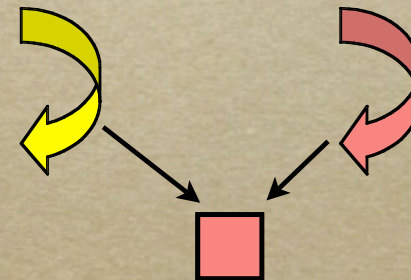
Locks

- Simplest abstraction: mutex (mutual exclusion lock)
- If data may be modified by more than one thread at once (e.g., a data structure), a thread must acquire a lock before accessing it
- If you request a lock and another thread holds it, the lock() call blocks until the other thread releases the lock

```
func(int count) {  
    lock(stack_lock);  
    addToStack(count);  
    unlock(stack_lock);  
}
```

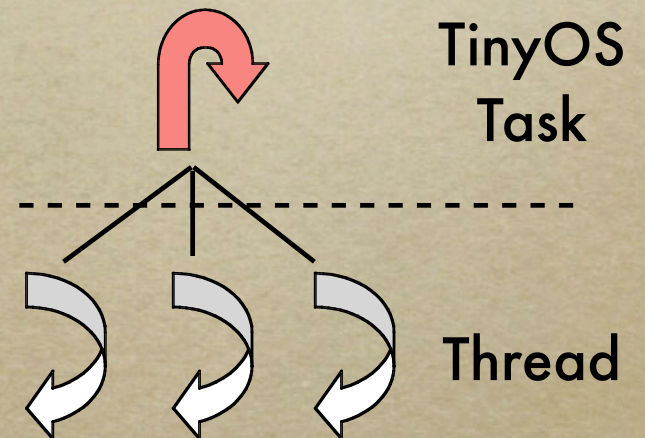
func(3);

func(3);



TinyThreads

- Schedules threads on top of TinyOS FIFO scheduler
 - * Assumes cooperative preemption points
 - * No forcible preemption
- Sort of like N-to-1 threads, except rather than a kernel thread there's a TinyOS task



TinyMOS

- Two versions
 - * Threads as long-running TinyOS computation: threads don't call TinyOS operations
 - * TinyOS as OS core for threaded programming: threads can call TinyOS operations
- Second model runs into concurrency problems
 - * Emulates TinyOS through MOS wrappers

