Process Subsystem, cont. Threads

BOLO for deadline flexibility;
Colloquium on Friday (from me);
List of clarifications to be added to the course page shortly!

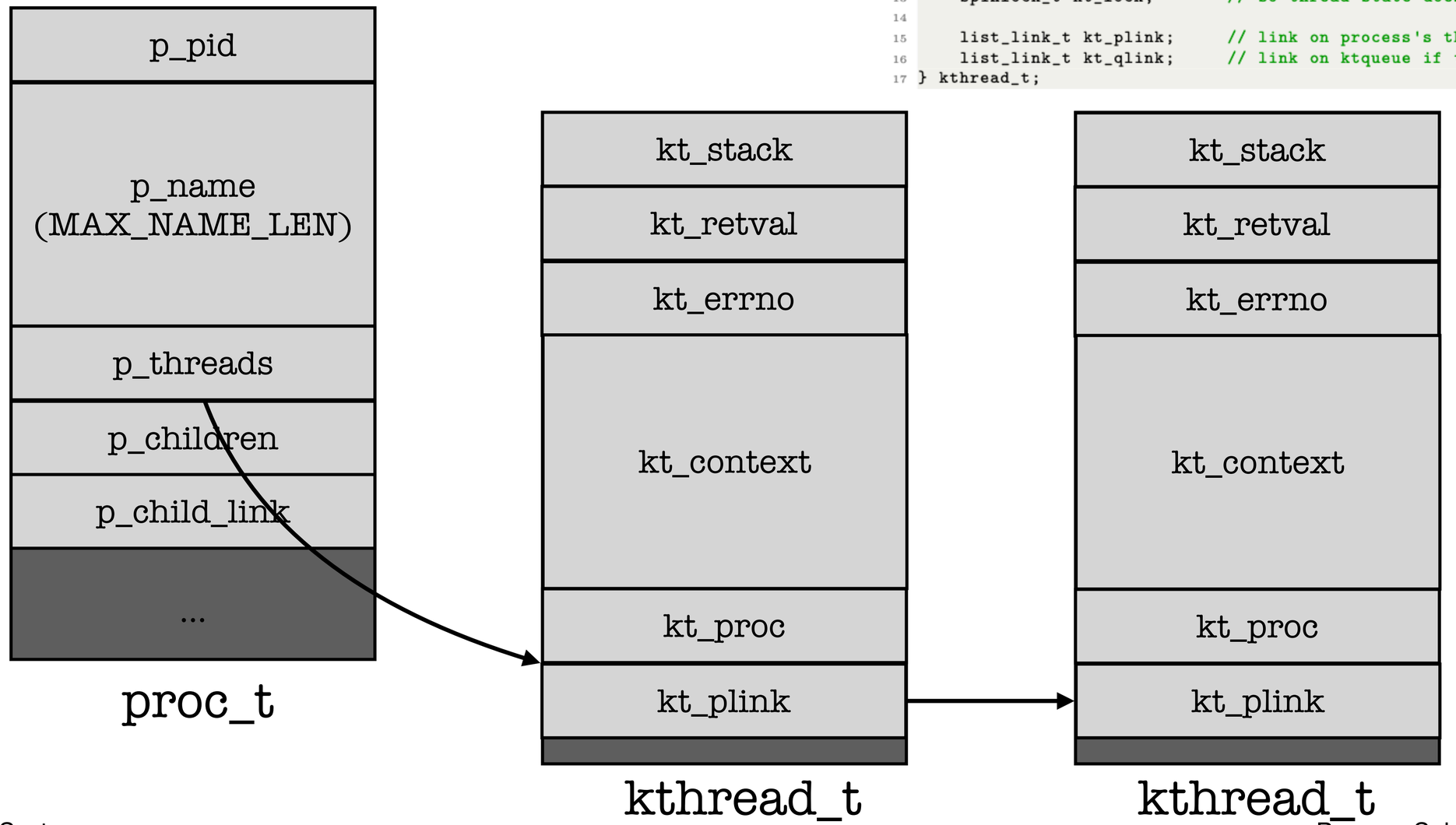
Homework 1 Reflection Form (same as from email)

Outline

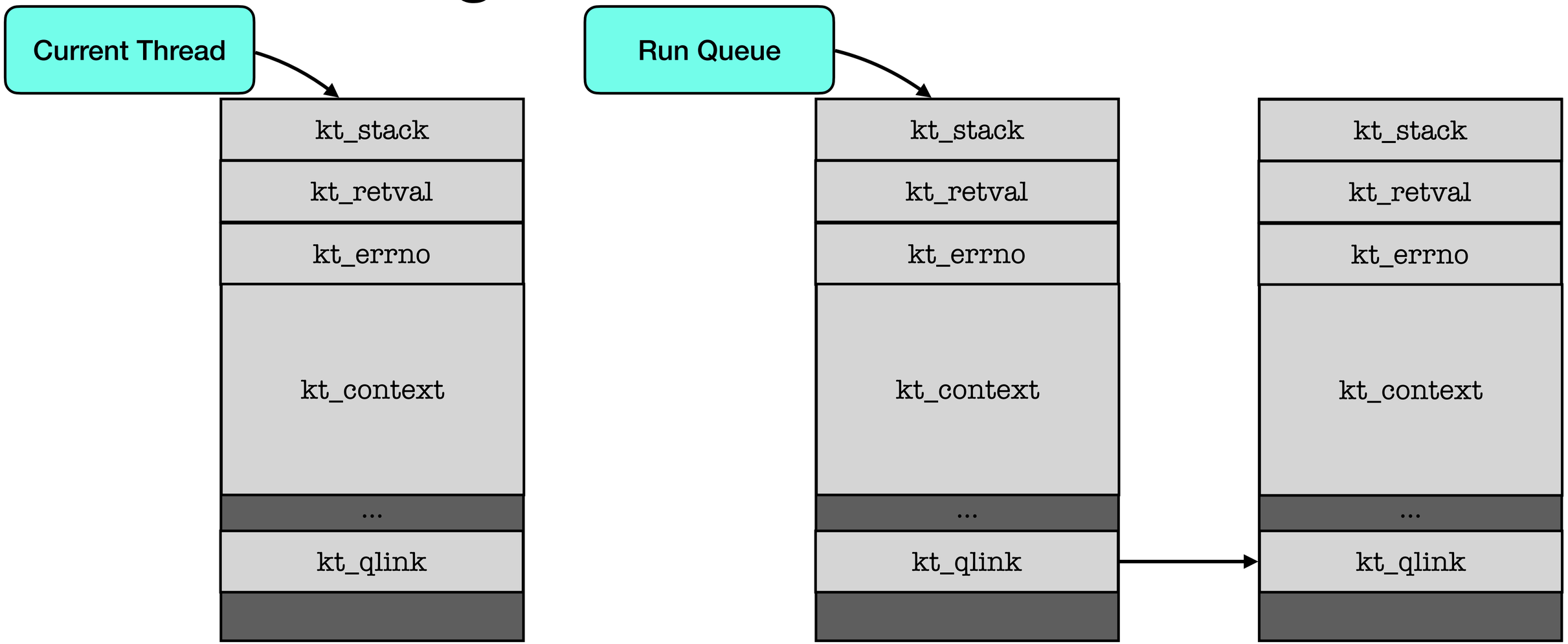
- Implementing context switches
- Two-Level Threading
- Thread Termination

From Friday: Threads

```c
/* Thread descriptor */
typedef struct kthread {
    char *kt_kstack;           // kernel stack pointer
    void *kt_retval;           // return value
    long kt_errno;             // errno of most recent syscall
    context_t kt_ctx;          // thread context

    struct proc *kt_proc;      // corresponding process

    long kt_cancelled;         // set if the thread has been cancelled
    kthread_state_t kt_state; // thread state

    spinlock_t kt_lock;        // so thread state doesn't change during clone

    list_link_t kt_plink;      // link on process's thread list, p_threads
    list_link_t kt_qlink;      // link on ktqueue if thread isn't running
} kthread_t;
```

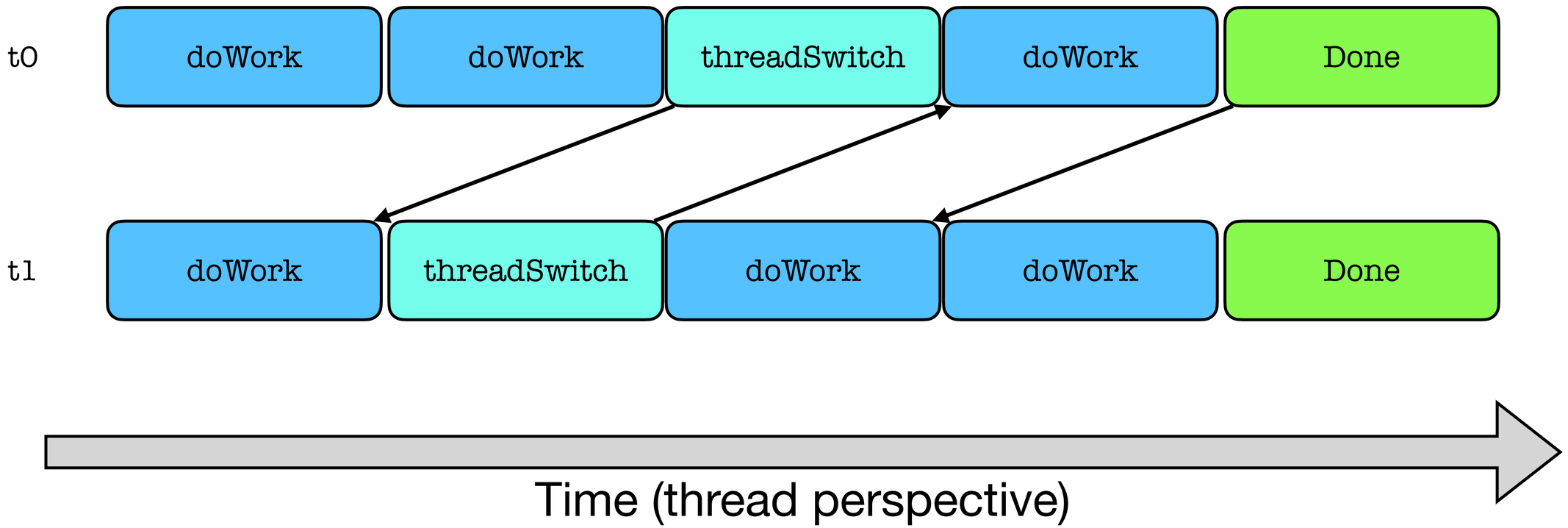


Threads Organization



Thread Organization

Two key challenges:
(1) How do we implement `threadSwitch`?
(2) How do we know when to call `threadSwitch`?



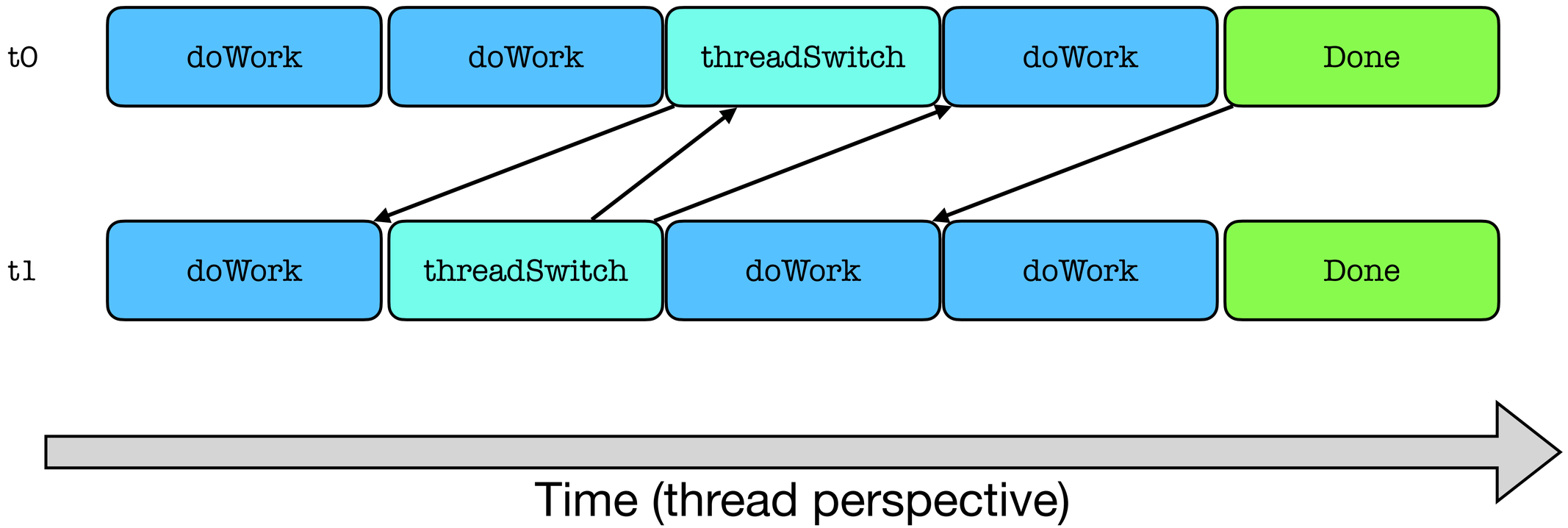
Switching Between Threads

```
void switchThread(thread_t *next_thread) {
    saveContext(&curthr->ctx);
    curthr = next_thread;
    setContext(&curthr->ctx);
}
```

Chat with your neighbor(s)!
Think about our previous example of two threads. What can go wrong in this implementation?

We save the instruction pointer right during `setContext`, so it will be restored to the line where we set `curthr` to `next_thread` again when it is restored!

Switching Between Threads (v0)



Chat with your neighbor(s)!

How can we fix the broken implementation to switch between threads? What are the issues that are leading to the first implementation being broken, and what are alternative mechanisms that could be used?

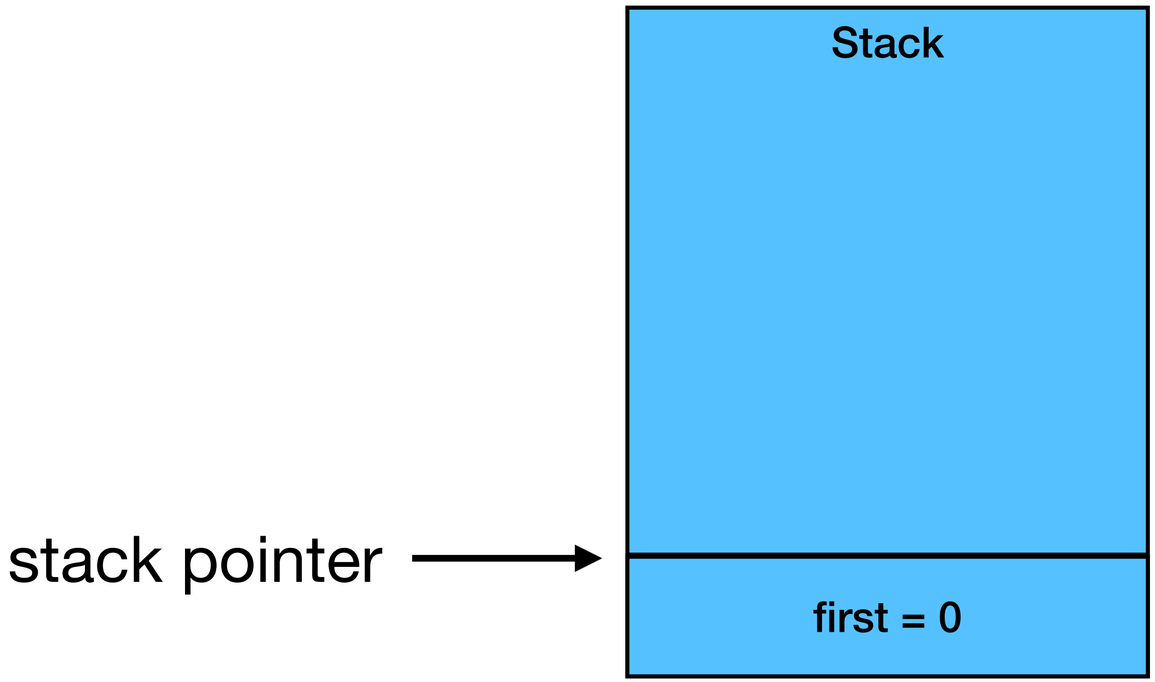
Switching Between Threads (v1)

```
thread_t *curthr; // a pointer to the current running thread
void saveContext(context_t *ctx); // saves the provided context to its TCB
void setContext(context_t *ctx); // set the provided context as active

void switchThread(thread_t *next_thread) {
    volatile int first = 1;
    saveContext(&curthr->ctx);
    if (first) {
        first = 0;
        curthr = next_thread;
        setContext(&curthr->ctx);
    }
}
```

Variable saved on the stack instead of in a register

Chat with your neighbor(s)!
Think about our previous example of two threads. Does this implementation fix the previous implementation?

Switching Between Threads (v1)




```
void switchThread(thread_t *next_thread) {
    volatile int first = 1;
    saveContext(&curthr->ctx);
    if (first) {
        first = 0;
        curthr = next_thread;
        setContext(&curthr->ctx);
    }
}
```

inst pointer

Because the stack pointer remains the same, when the thread is restored the data at the address of the stack pointer will have been affected by the first call

When the thread's context is restored, reading first from memory will mean that the data is up-to-date with the previous call from the execution

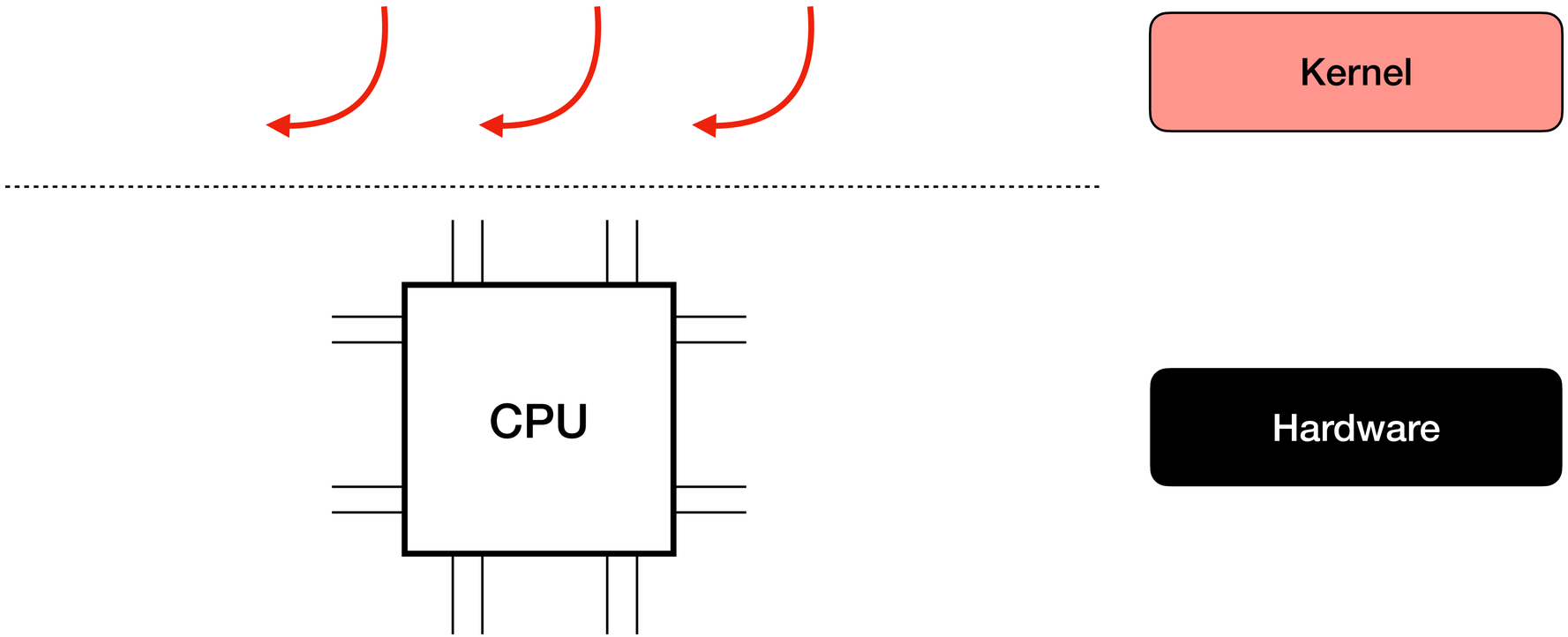
Switching Between Threads

- Threads rely on contexts as input to the real hardware that executes next lines of code across the executable units of the system
- If we are not careful with how we setup the switching between these executable units, we will affect the processor in such a way that we end up executing incorrect information
- We can use techniques (like hiding information on the stack) to work around the corruption of register values
- To get the next thread, we need to query the run queue to get the `next_thread` input to the `switchThread` function!

Chat with your neighbor(s)!

Think about the system that you are implementing in homework 2. In this system, how do we know when to switch threads? Once we know when to switch threads, how do we pick the next thread to execute?

One-Level Threading Structure



Two-Level Threading (Motivation)

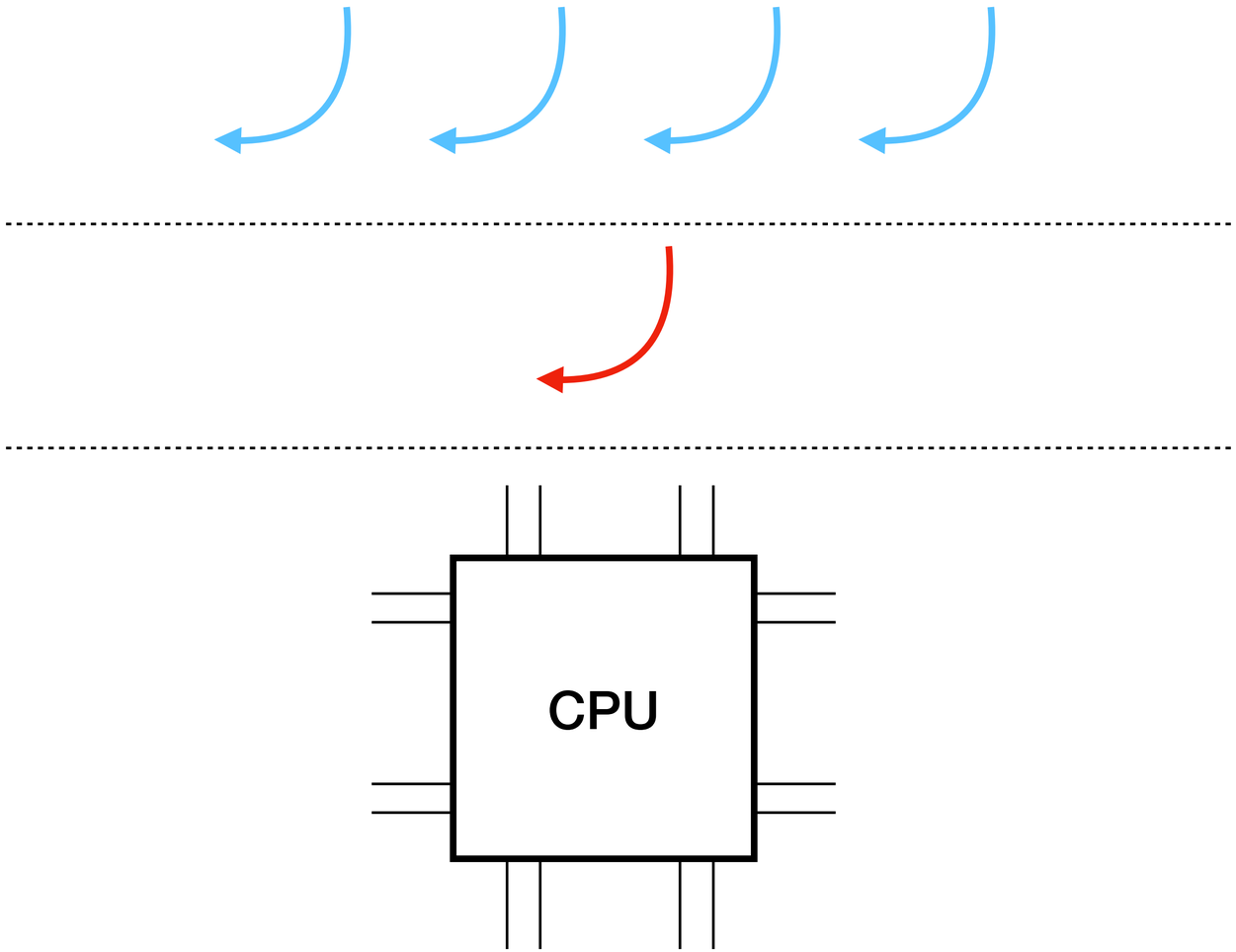
- You may have noticed that there is a discrepancy between the idea of a "`kthread_t`" in the assignment and the working example of a "`thread_t`" so far in this lecture…
- If it is the case that the only definition of a "thread" is in the operating system, then every single time we want to execute a thread action (e.g., `pthread_mutex_lock`) we would need to context switch into the operating system
- In practice, while there are many thread actions that should involve the OS in some way (e.g., `pthread_create`, `pthread_join`), there are a lot of synchronizing inter-thread actions that can be performed without any notion of privilege (e.g., lock acquisition)
- For this reason, most systems today tend to use a two-level threading model (with a user-level library playing a major role)

Two-Level Threading

No system calls needed to switch into the kernel context! 🥳

One kthread at a time… if one uthread makes a system call than all uthreads are interrupted

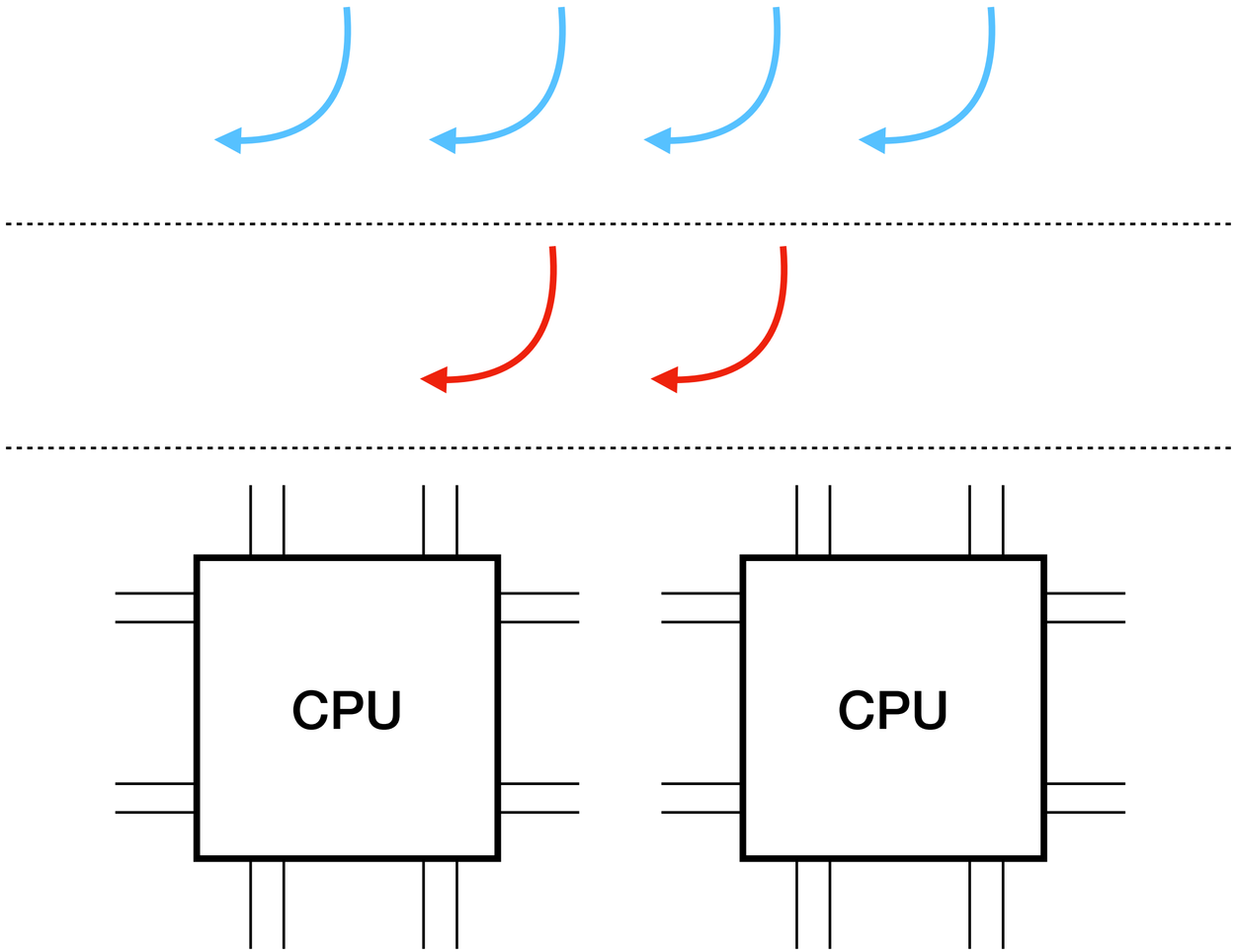
Real parallelism is limited by the number of kthreads running on cores… this is called *N*-to-1 threading



Two-Level Threading (*M*-to-*N*)

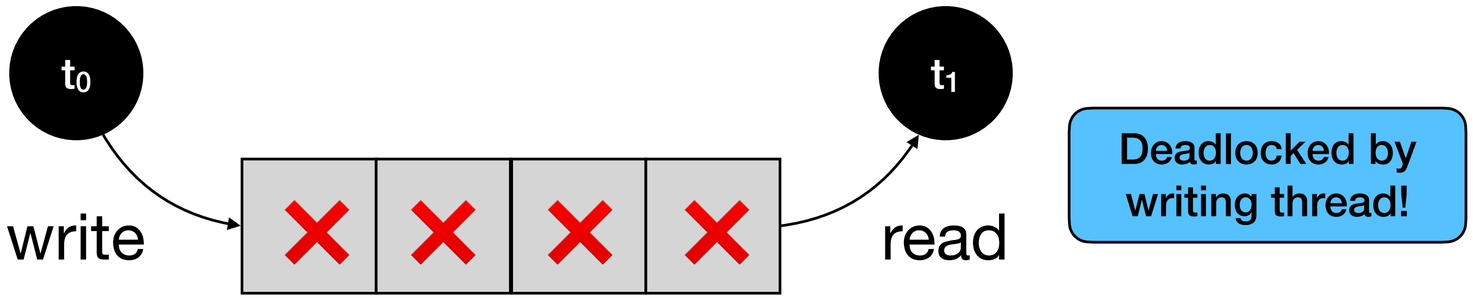
A system call in one thread does not necessarily interrupt other threads 🥳

Can get into still get into trouble with too few kernel threads…



Chat with your neighbor(s)!

Suppose two user threads are running in an *M*-to-*N* threading system with one kernel thread, where one thread is communicating to the other by adding elements to a file for the other to read (producer-consumer). If there is a maximum file size, what issues can arise?



Thread Termination

- On termination, the thread changes state to “zombie state”
- If the thread is joinable, then it notifies the waiting thread (which must be implemented in its thread structure)
- If the thread is detached, it disappears!
- To collect detached zombie threads, Linux and other operating systems implement a “reaper” thread who’s job it is to wait for zombie threads to appear and then delete the zombies ➡️ this is often implemented as a functionality of the thread associated with the init process

Summary

- We can organize the threads that can run in a system in terms of the current running thread and the remaining threads in the run queue
- To switch between threads, we need to be careful to avoid overwriting the processor state in such a way that makes it difficult to revert back to the thread being switched away from
- We can add a level of indirection on top of kernel threads to avoid expensive context switches for all normal case thread operations