

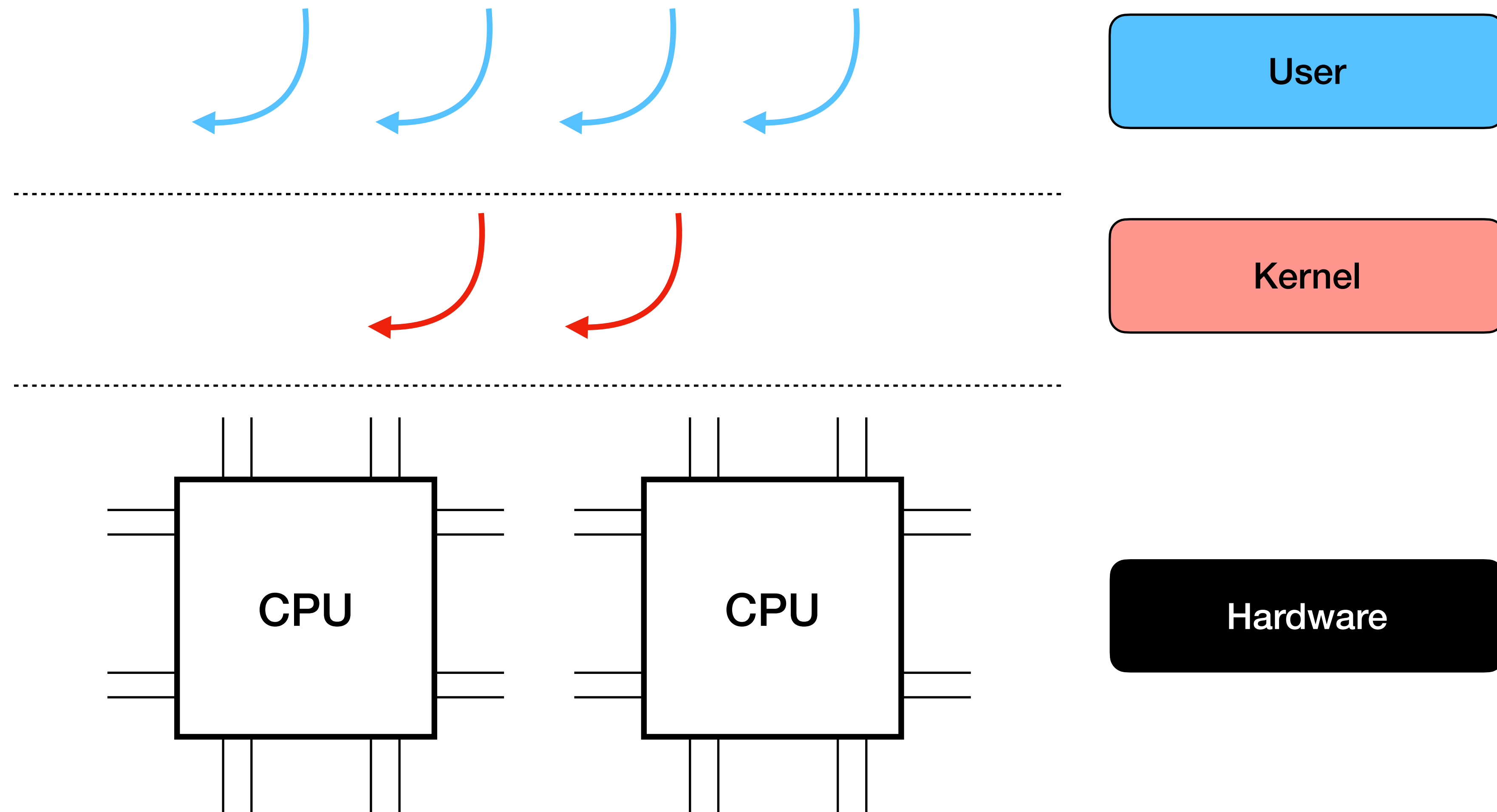
Process Subsystem, cont. (Linux Case)

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

Outline

- Comparing user-threading libraries
- Linux processes and threads
- Task life cycles
- Run queues, wait queues, and futexes

From Monday: Two-Level Threading (*M-to-N*)

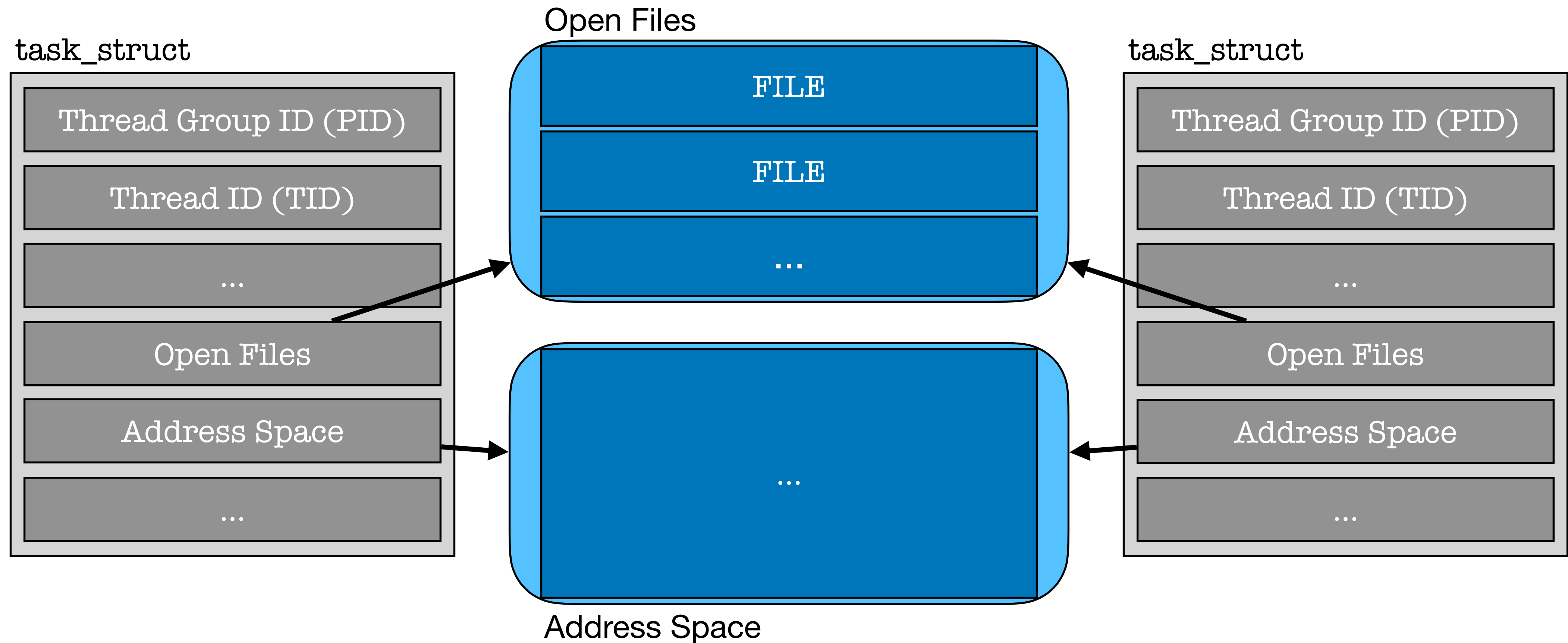


User Threads versus Kernel Threads

- Kernel threads are the operating system feature that implements the runnable context(s) associated with a process whereas user threads describe libraries in user space that may map to one or more kernel threads
 - Common examples of user threads are pthreads and/or thread pools (e.g., OpenMP library)
- User threads are implemented as part of the program binary rather than as part of the operating system kernel, meaning that the *thread implementation* is in user space (context switches do not require unloading/reloading address spaces, loss of locality)
- Different user threading libraries will have different methodologies for mapping user threads onto kernel threads ➡ in Linux, pthread creates threads using NPTL (Native POSIX Thread Library) in which user threads are created one-to-one with kernel threads

Demo!

Linux has neither “Processes” nor “Threads” but “Tasks”



Linux has neither “Processes” nor “Threads” but “Tasks”

```
execve("./hello-pthread", ["/hello-pthread"], 0x7ffd730ff0e0 /* 44 vars */) = 0
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffeff4b0a80) = -1 EINVAL (Invalid argument)
brk(NULL) = 0xfed000
brk(0xfedd00) = 0xfedd00
arch_prctl(ARCH_SET_FS, 0xfed3c0) = 0
uname({sysname="Linux", nodename="itbdcv-lnx04p.campus.pomona.edu", ...}) = 0
set_tid_address(0xfed690) = 239439
set_robust_list(0xfed6a0, 24) = 0
rt_sigaction(SIGRTMIN, {sa_handler=0x401000, sa_mask=[], sa_flags=SA_RESTORER|SA_SIGINFO, sa_restorer=0x40ca40}, NULL, 8) = 0
rt_sigaction(SIGRT_1, {sa_handler=0x401090, sa_mask=[], sa_flags=SA_RESTORER|SA_RESTART|SA_SIGI
NFO, sa_restorer=0x40ca40}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
readlink("/proc/self/exe", "/home/stum2025/cs134/demos/lec05"..., 4096) = 46
getrandom("\x49\x4a\x50\x45\x84\x81\x27\x72", 8, GRND_NONBLOCK) = 8
brk(0x100ed00) = 0x100ed00
brk(0x100f000) = 0x100f000
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0x7ff61a2a8000
mprotect(0x7ff61a2a9000, 8388608, PROT_READ|PROT_WRITE) = 0
clone(child_stack=0x7ff61aaa8330, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREA
D|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tid=[239440], tls
=0x7ff61aaa8700, child_tidptr=0x7ff61aaa89d0) = 239440
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0x7ff6137ff000
mprotect(0x7ff613800000, 8388608, PROT_READ|PROT_WRITE) = 0
clone(child_stack=0x7ff613fff330, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREA
D|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tid=[239441], tls
=0x7ff613fff700, child_tidptr=0x7ff613fff9d0) = 239441
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0x7ff619aa7000
mprotect(0x7ff619aa8000, 8388608, PROT_READ|PROT_WRITE) = 0
clone(child_stack=0x7ff61a2a7330, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREA
D|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tid=[239442], tls
=0x7ff61a2a7700, child_tidptr=0x7ff61a2a79d0) = 239442
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0x7ff6192a6000
```

CLONE_VM (since Linux 2.0)

If **CLONE_VM** is set, the calling process and the child process run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. Moreover, any memory mapping or unmapping performed with **mmap(2)** or **munmap(2)** by the child or calling process also affects the other process.

If **CLONE_VM** is not set, the child process runs in a separate copy of the memory space of the calling process at the time of **clone()**. Memory writes or file mappings/unmappings performed by one of the processes do not affect the other, as with **fork(2)**.

CLONE_FS (since Linux 2.0)

If **CLONE_FS** is set, the caller and the child process share the same filesystem information. This includes the root of the filesystem, the current working directory, and the umask. Any call to **chroot(2)**, **chdir(2)**, or **umask(2)** performed by the calling process or the child process also affects the other process.

If **CLONE_FS** is not set, the child process works on a copy of the filesystem information of the calling process at the time of the **clone()** call. Calls to **chroot(2)**, **chdir(2)**, or **umask(2)** performed later by one of the processes do not affect the other process.

CLONE_FILES (since Linux 2.0)

If **CLONE_FILES** is set, the calling process and the child process share the same file descriptor table. Any file descriptor created by the calling process or by the child process is also valid in the other process. Similarly, if one of the processes closes a file descriptor, or changes its associated flags (using the **fcntl(2)** **F_SETFD** operation), the other process is also affected. If a process sharing a file descriptor table calls **execve(2)**, its file descriptor table is duplicated (unshared).

If **CLONE_FILES** is not set, the child process inherits a copy of all file descriptors opened in the calling process at the time of **clone()**. Subsequent operations that open or close file descriptors, or change file descriptor flags, performed by either the calling process or the child process do not affect the other process. Note, however, that the duplicated file descriptors in the child refer to the same open file descriptions as the corresponding file descriptors in the calling process, and thus share file offsets and file status flags (see **open(2)**).

“Processes” and “Threads” in Linux

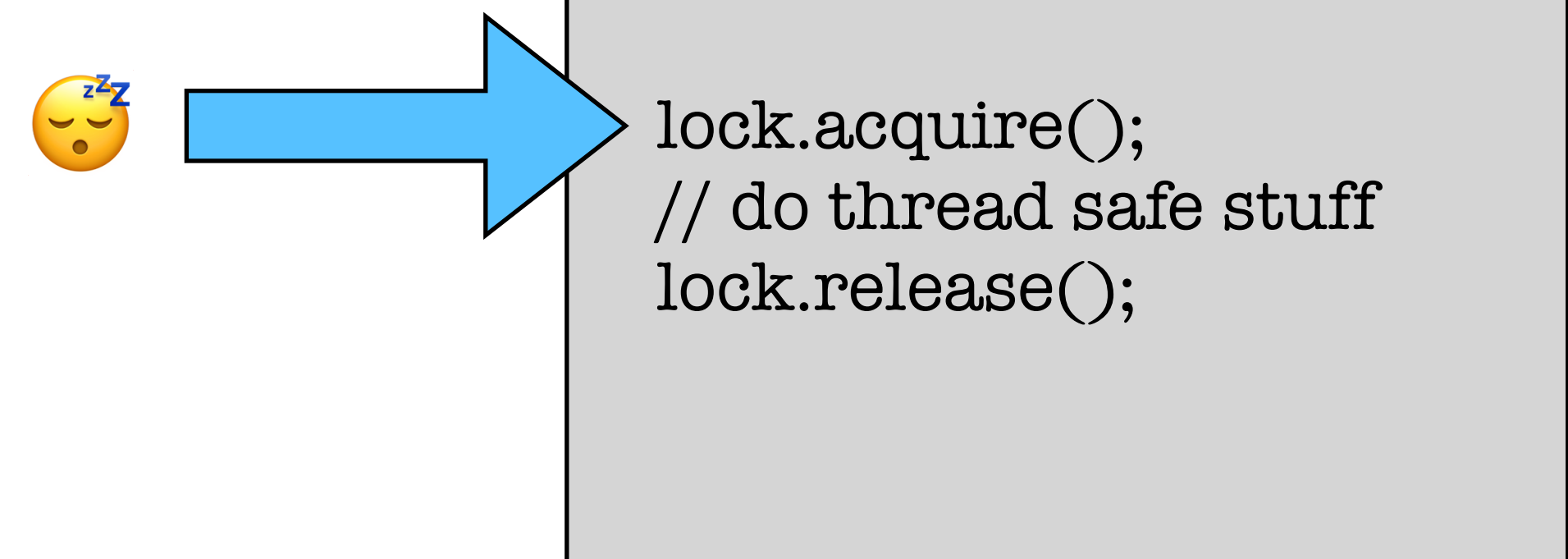
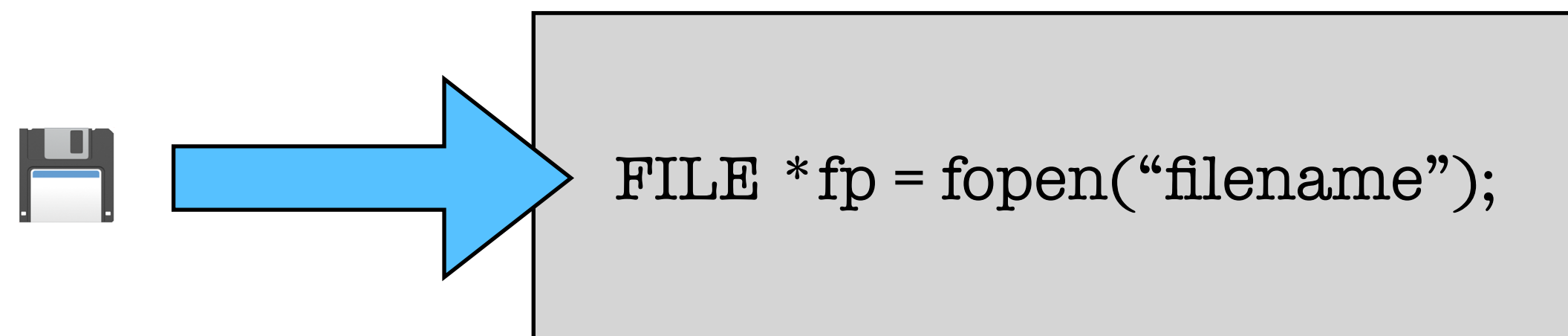
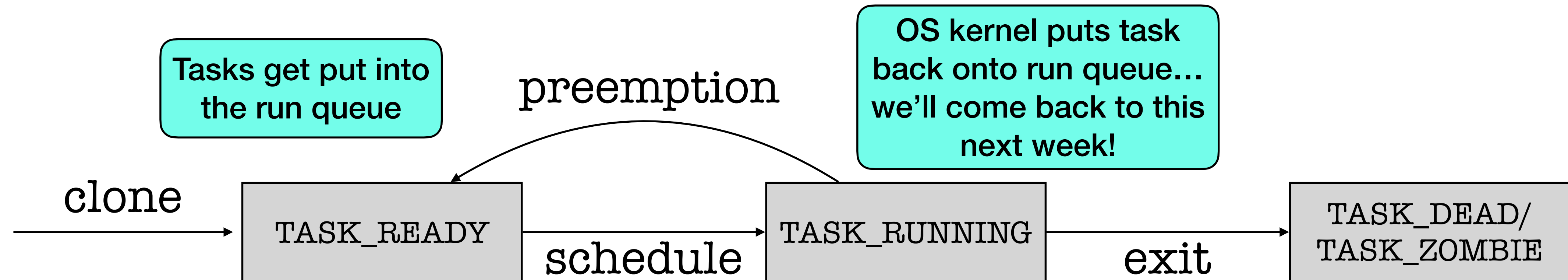
- Processes and threads are both expressed as “tasks” in Linux, which means that the same struct is used to describe both abstractions
- To create a new process (e.g., a call to `fork`), the clone system call is used with a particular set of flags to ensure that only the appropriate fields are copied over
 - A child process has the same state as its parent (CoW), but a executing a new process may have much less similar state to the parent (e.g., the terminal/shell process)
- Creating a new thread also implies copying over state  in this case, threads will share *all* resources with their parent “process” and will have their PID (thread group identifier) remain the same as the parent

Chat with your neighbor(s)!

**What is a benefit of defining the processes subsystem in terms of `proc_t` and `kthread_t` structs separately?
What is a benefit of having a uniform declaration of both types?**

Lifecycle of a Linux Task

Sometimes our processes/threads execute slow operations, like disk operations or be blocked on a mutex... we should have some way to switch between tasks!



Run Queue and Wait Queue

- The Linux OS kernel organizes tasks into three categories: running, ready, and waiting ➡ running describes the task pointed to by the “curproc” equivalent, ready describes tasks in the run queue, and waiting describes tasks in the wait queue
- When selecting the next thread to run, the scheduler chooses the next thread to execute from the *run queue* object ➡ these tasks are those that have been *preempted* by the OS kernel (had their time taken away against their will)
- If it is the case that a thread is about to do something slow, it can choose to put itself in the *wait queue* associated with its *parent process* in which it will only be woken up when a particular event occurs

Using the Wait Queue (implementing a futex)

```
void spinlock_lock(int *mutex) {  
    while (1) {  
        while (*mutex != 0) {  
            // the mutex is currently unlocked!  
            if (!CAS(mutex, 0, 1) {  
                // lock acquired!  
                return;  
            }  
        }  
    }  
}
```

Returns previously
pointed to value on
success (0)

```
void spinlock_unlock(int *mutex) {  
    assert(CAS(mutex, 1, 0) == 1);  
}
```

Returns previously
pointed to value on
success (1)

What happens if a thread is
preempted by the operating
system right after it wins a
lock?

Using the Wait Queue (implementing a futex)

```
void futex_wait(futex *futex, int value) {  
    if (futex->value == value) {  
        // puts current thread in wait queue  
        sleep();  
    }  
}
```

```
void futex_wake(futex *futex) {  
    // wake up one thread from the wait queue  
    // (if any are there)  
}
```

```
int atomic_inc(int *i) {  
    return (i++); // i's original value  
}
```

```
void futex_lock(futex_t *futex) {  
    int c;  
    while ((c = atomic_inc(&futex->val)) != 0) {  
        futex_wait(futex, c + 1);  
    }  
}
```

```
void futex_unlock(futex_t *futex) {  
    futex->val = 0;  
    futex_wake(futex);  
}
```

Chat with your neighbor(s)!

Given the previous implementation of a futex-based lock, what issues could potentially arise?

Fixing the Futex Lock

```
void futex_wait(futex *futex, int value) {  
    if (futex->value == value) {  
        // puts current thread in wait queue  
        sleep();  
    }  
}
```

```
void futex_wake(futex *futex) {  
    // wake up one thread from the wait queue  
    // (if any are there)  
}
```

```
int atomic_dec(int *i) {  
    return (i--); // i's original value  
}
```

Mutex
no wait

```
void futex_lock(futex_t *futex) {  
    int c;  
    if ((c = CAS(&futex->val, 0, 1)) != 0) {  
        do{  
            futex_wait(futex, 2);  
        } while ((c = CAS(&futex->val, 0, 2)) != 0);  
    }  
}
```

```
void futex_unlock(futex_t *futex) {  
    if (atomic_dec(&futex->val) != 1) {  
        futex->val = 0;  
        futex_wake(futex);  
    }  
}
```

Summary

- The definition of a “user thread” depends on the library ➡ pthreads build threads directly, OpenMP threads create thread pools, etc...
- Linux uses `tasks` as its primitive struct to implement *both* processes and threads ➡ on creation, certain flags are passed to the `clone` system call to implement the functionality
- Beyond the run queue, the process structure maintains a wait queue in which threads may be placed to implement primitives like futexes (and condition variables from 105!)