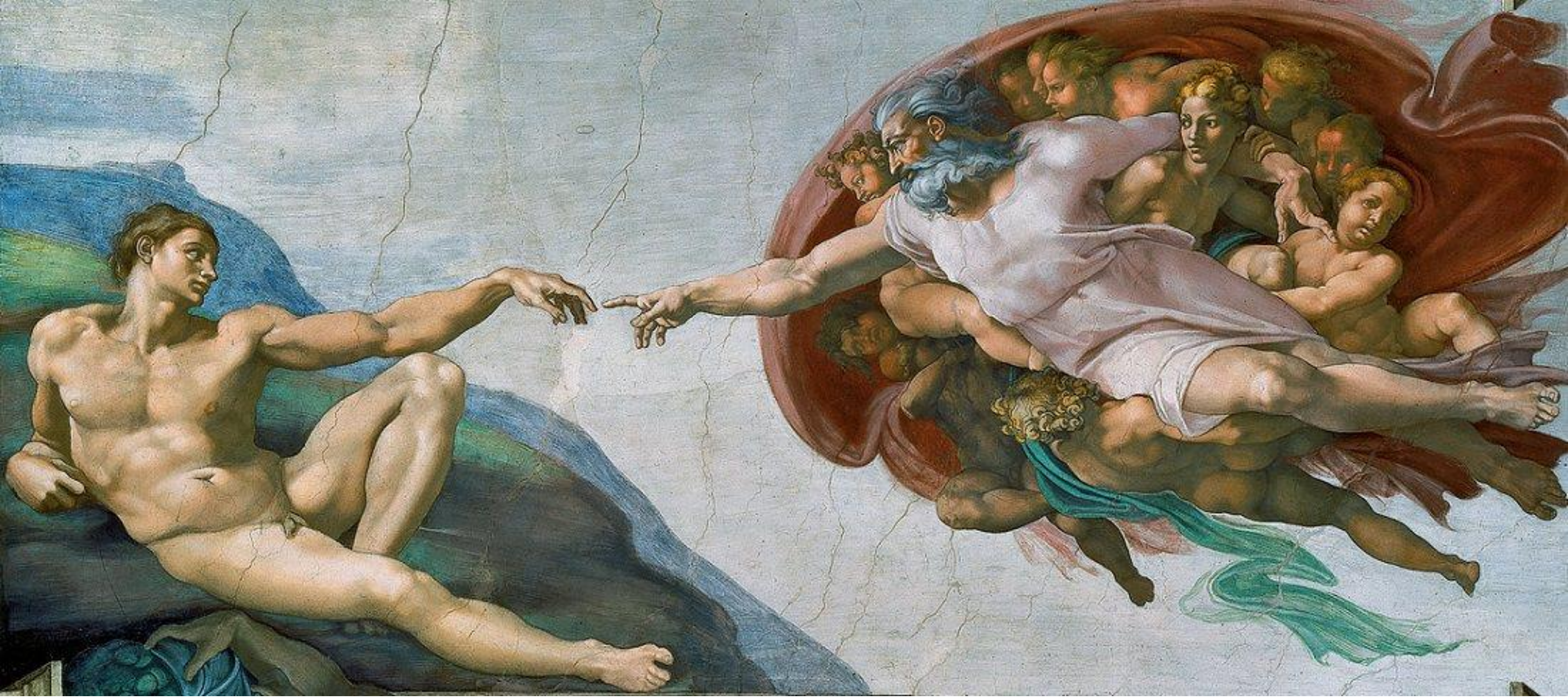


Processes

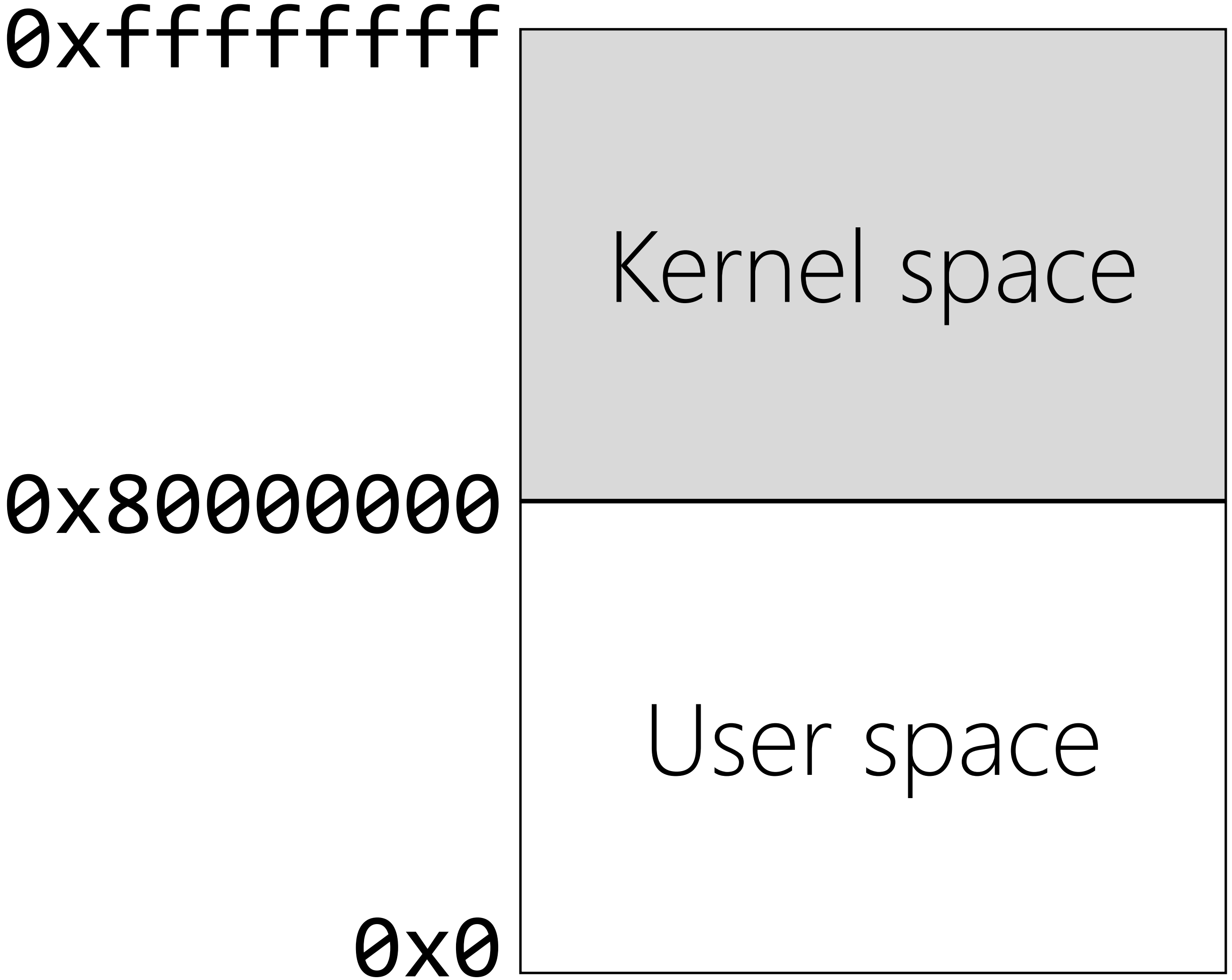
CS 161: Lecture 2

1/31/17

Processes

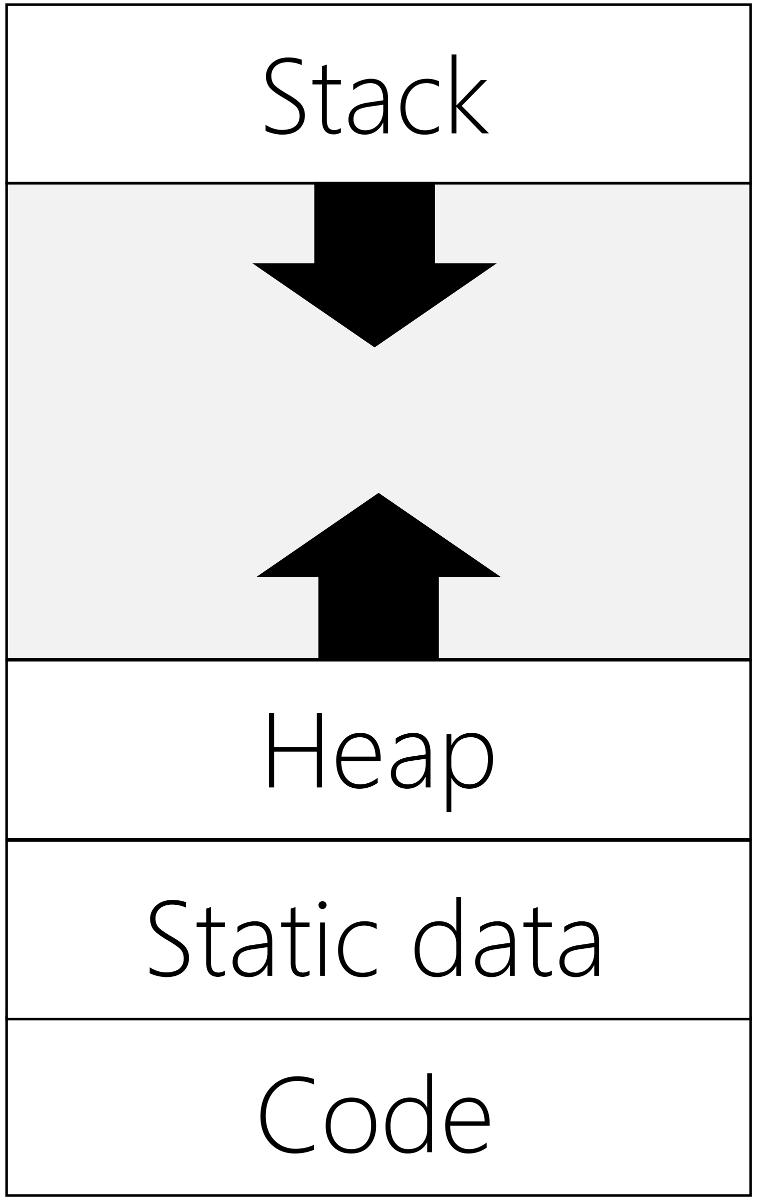
- A process is a collection of resources
  - An address space, which contains:
    - Code (i.e., executable instructions)
    - Data (i.e., static data like constant strings, dynamic data like the heap)
    - Stack (used to support function calls)
  - Bookkeeping stuff like . . .
    - A process id (PID)
    - Open file descriptors (e.g., network sockets, pipes, open disk files)
    - A current working directory
  - One or more threads of execution
    - A thread represents a computation which shares the code, data, and bookkeeping stuff inside the process
    - Each thread has a separate stack, and a separate set of registers
- A single "application" consists of one or more processes

Inside a 32-bit Address Space on MIPS



- Address space: the set of virtual addresses that a process's code can access
  - A large array of bytes starting at 0 and extending to 2^{32}-1
  - Kernel code can access any offset
  - User-level code can only access offsets in [0x0, 0x80000000]
- Physical RAM may be larger or smaller than a 2^{32} bytes
  - OS must handle the translation between virtual addresses and physical addresses (i.e., the addresses that are sent to the memory hardware)
  - We'll discuss this translation in detail in a few weeks!

Inside User Space



Stack

Heap

Static data

Code

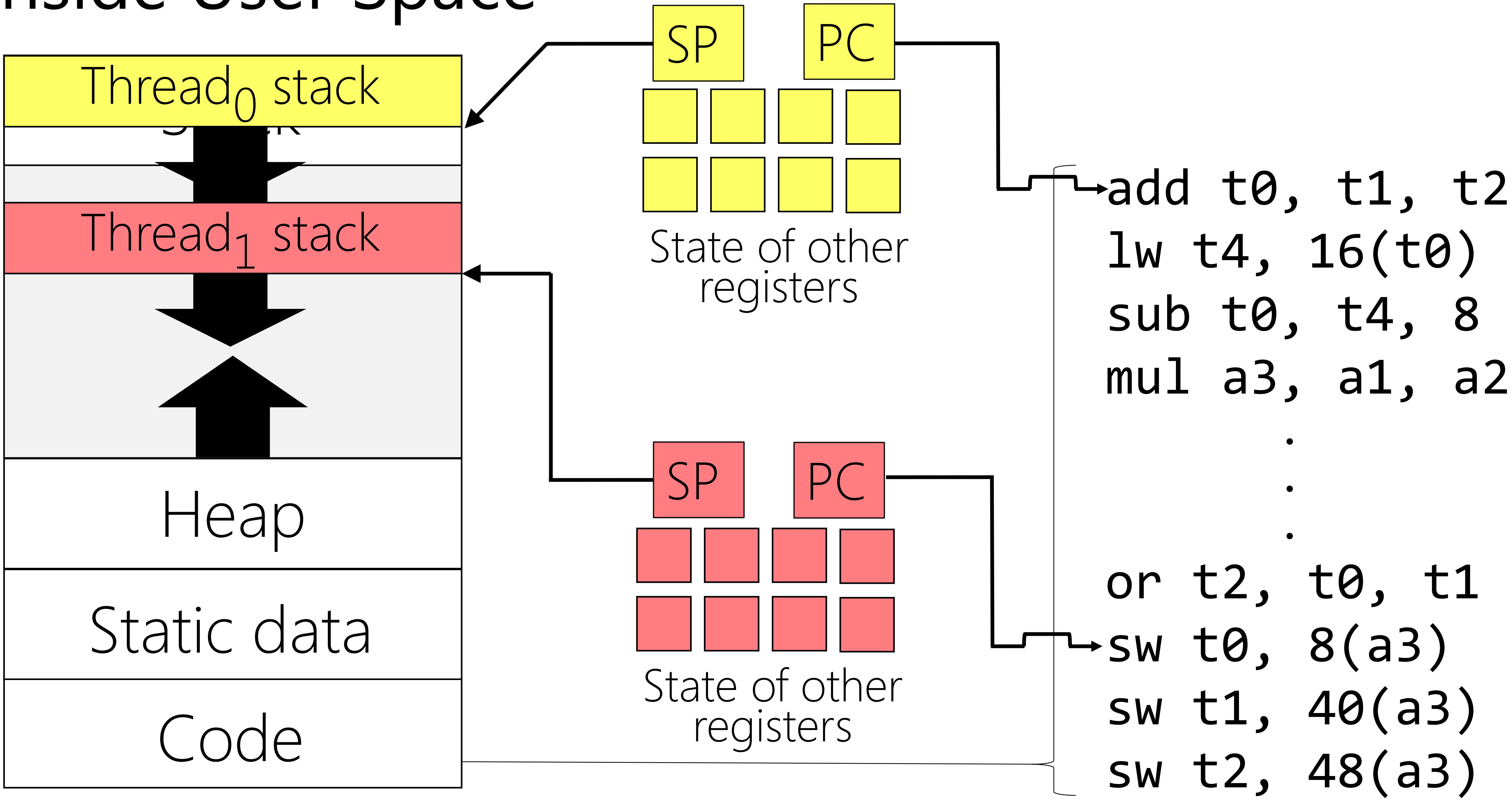
```
void h(){}
void g(){h();}
void f(){g();}
f(); //Stack records pushed,
     //popped as functions are
     //called and return
```

```
char *ptr = malloc(4096);
printf("%p\n", (void *)&ptr);
            //“0x7ffd90590168”
```

```
//At top of .c file
int foo = 42;
```

```
add t0, t1, t2
lw t4, 16(t0)
sub t0, t4, 8
```

Inside User Space



Thread_0 stack

Thread_1 stack

Heap

Static data

Code

SP

PC

State of other registers

SP

PC

State of other registers

```
add t0, t1, t2
lw t4, 16(t0)
sub t0, t4, 8
mul a3, a1, a2
        .
        .
        .
or t2, t0, t1
sw t0, 8(a3)
sw t1, 40(a3)
sw t2, 48(a3)
```

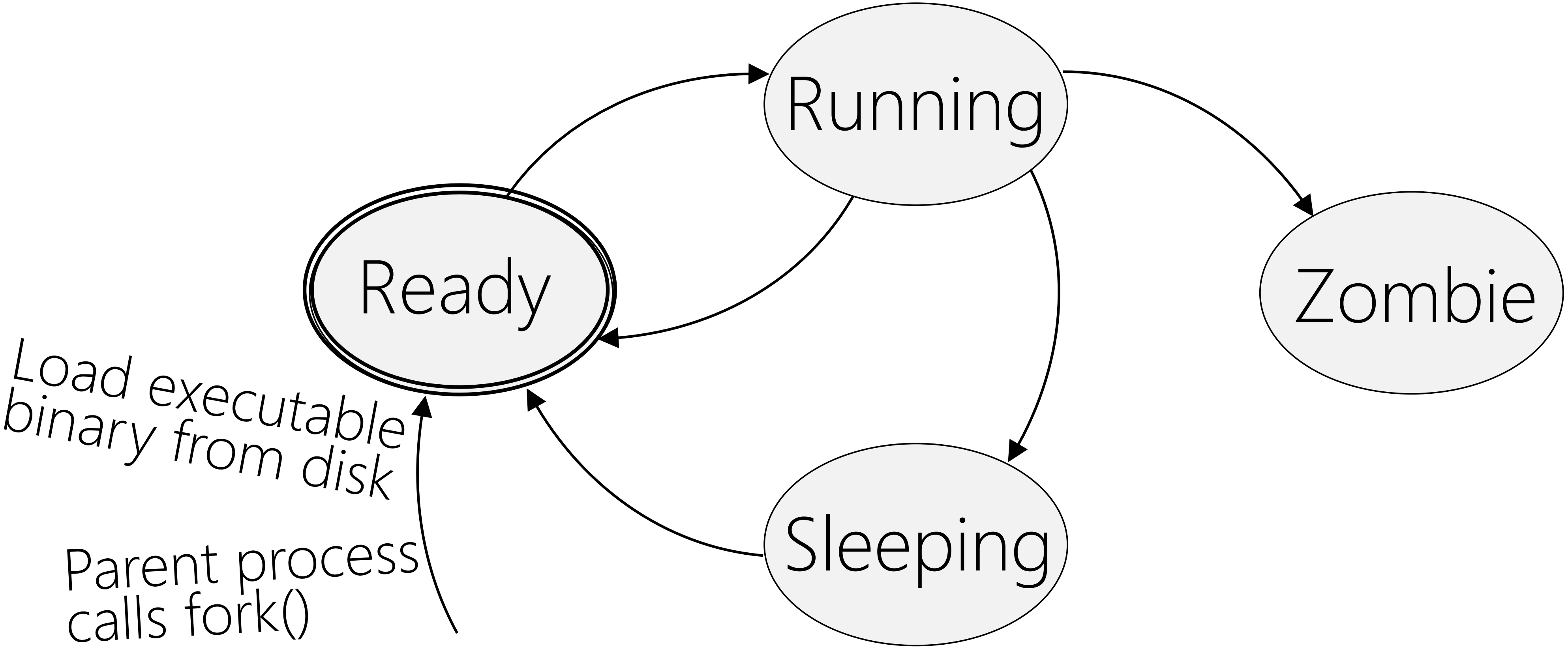
What Processes Are Running Now?

On Linux, try "ps -ef | less":

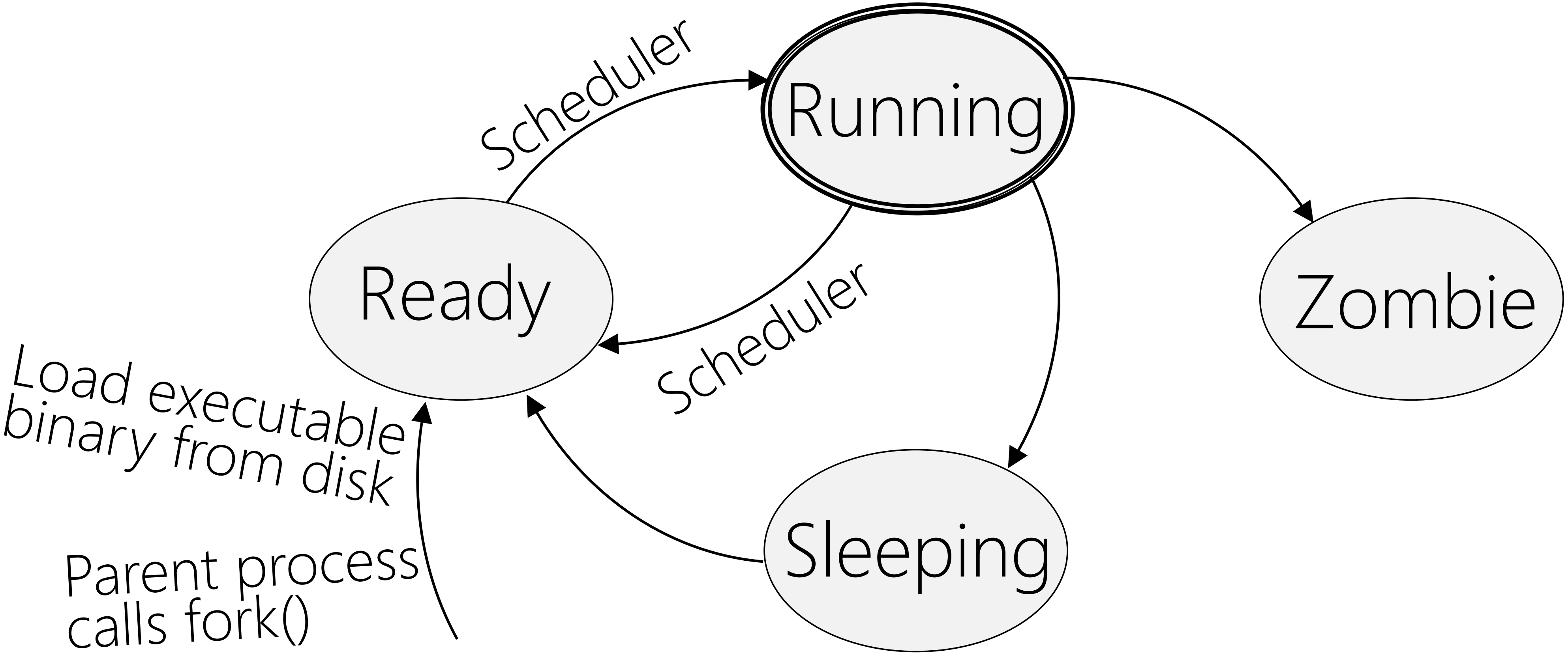
```
UID          PID   PPID  C STIME TTY          TIME CMD
root           1      0  0  2015 ?        00:00:02 init [3]
root           2      1  0  2015 ?        00:00:00 [migration/0]
root           3      1  0  2015 ?        00:00:00 [ksoftirqd/0]
                      .
                      . //Many other processes!
                      .
cs161      21085  20995  0 23:43 pts/1    00:00:00 ps -ef
cs161      21086  20995  0 23:43 pts/1    00:00:00 less
```

We created these processes!

Process States



Process States



Running

Ready

Zombie

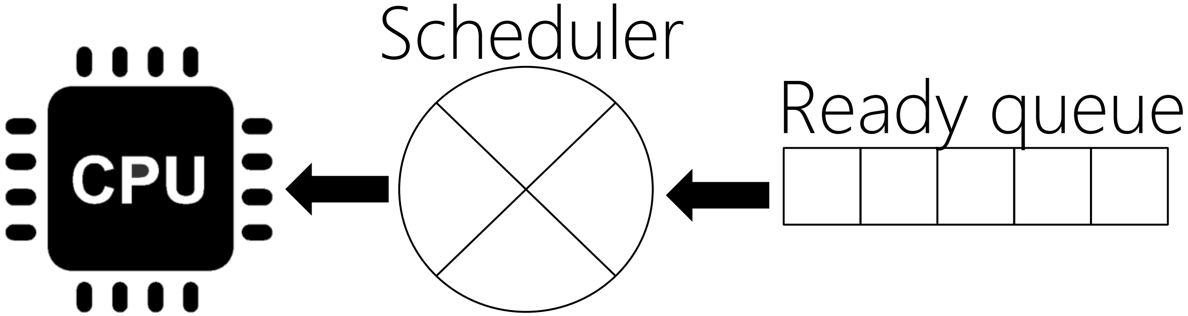
Sleeping

Scheduler

Scheduler

Load executable binary from disk

Parent process calls fork()



Scheduler

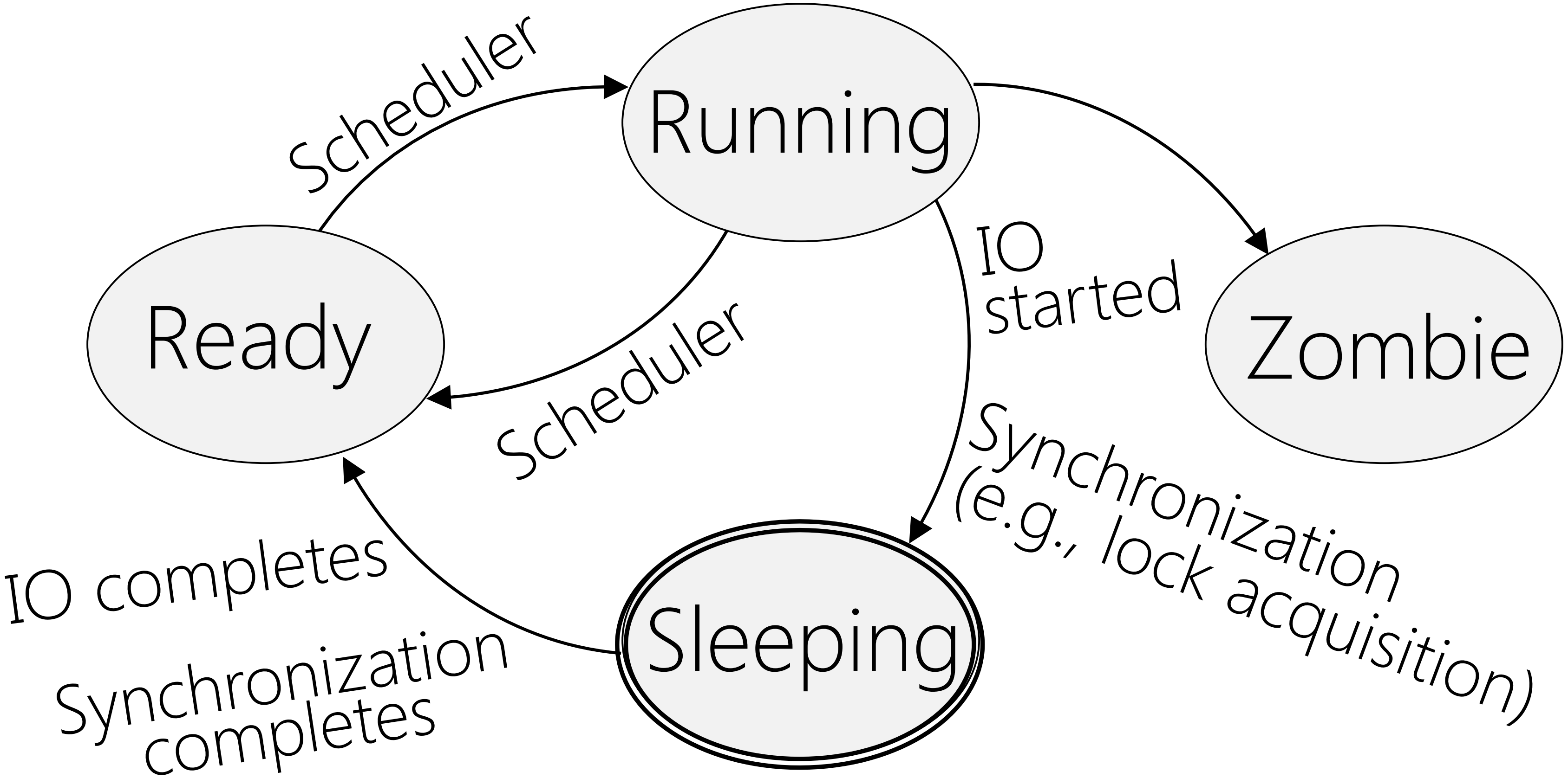
Ready queue

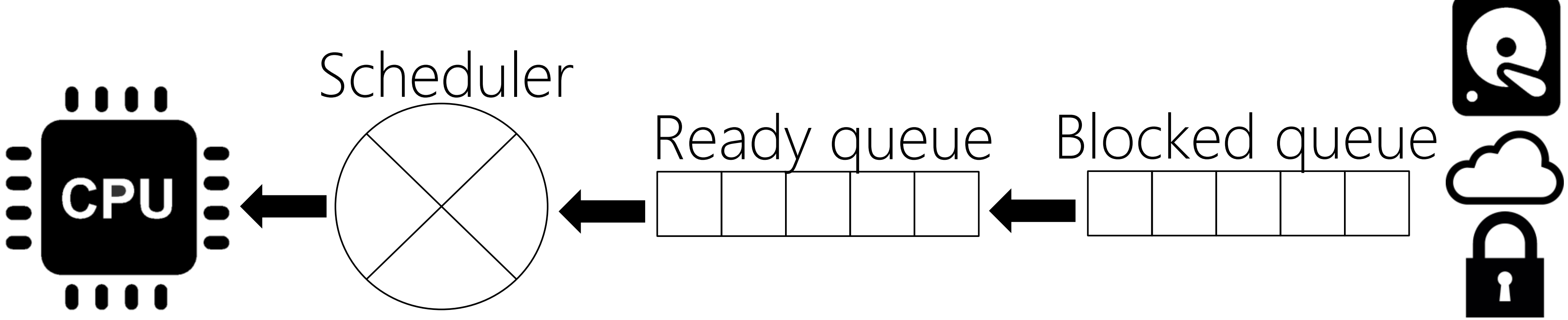
Scheduler Algorithms

FIFO?

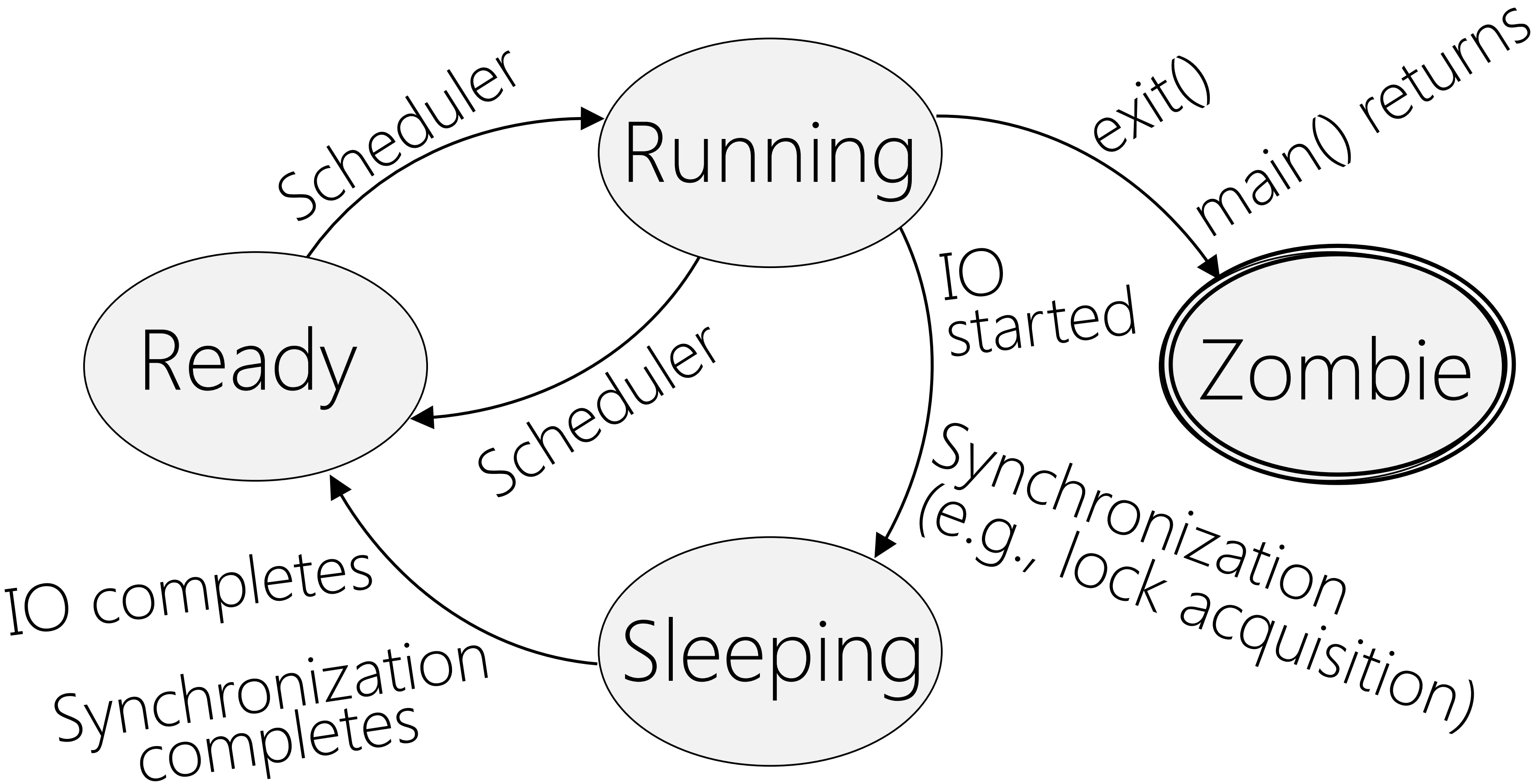
Priorities?

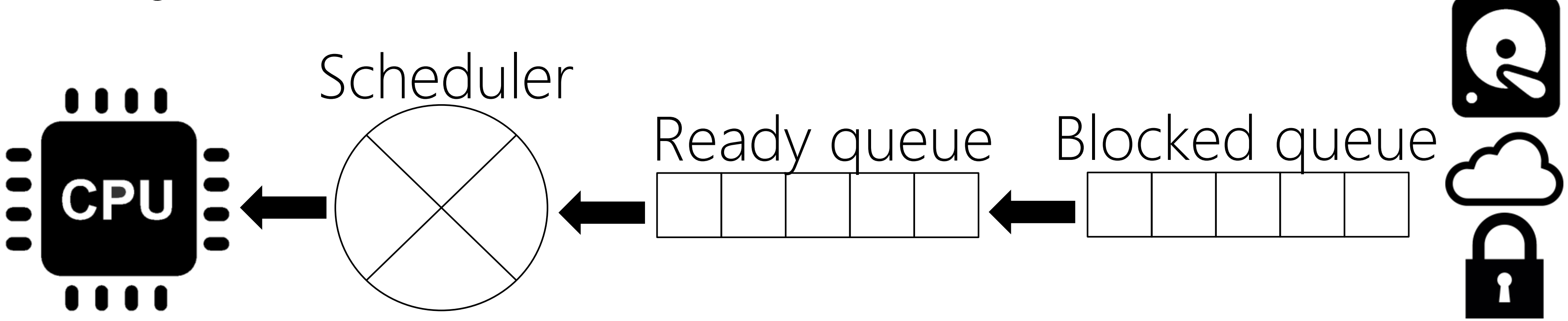
Process States



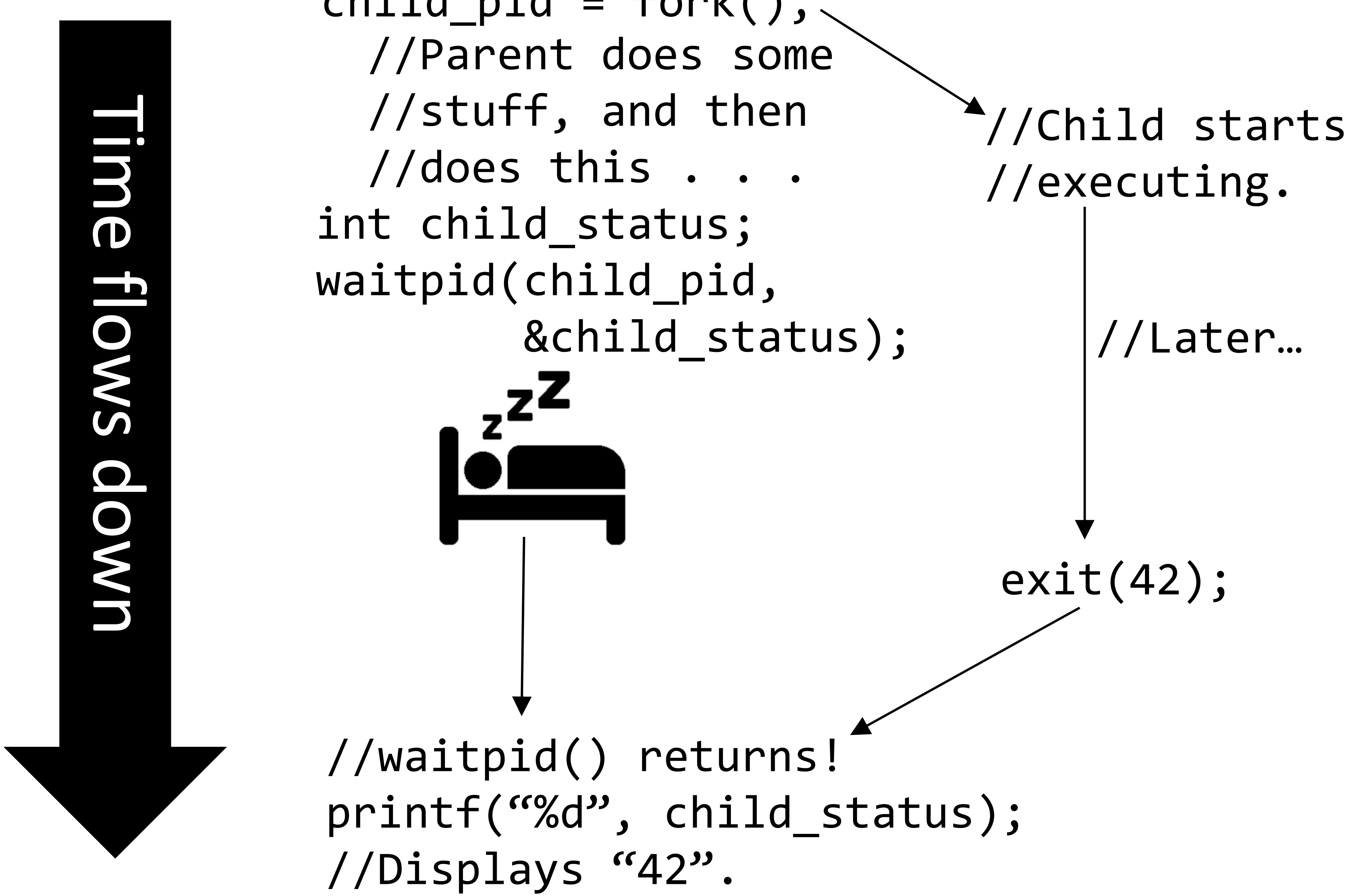


Process States





Process Termination



Parent

```
child_pid = fork();
  //Parent does some
  //stuff, and then
  //does this . . .
int child_status;
waitpid(child_pid,
        &child_status);
```

```
//waitpid() returns!
printf(“%d”, child_status);
//Displays “42”.
```

Child

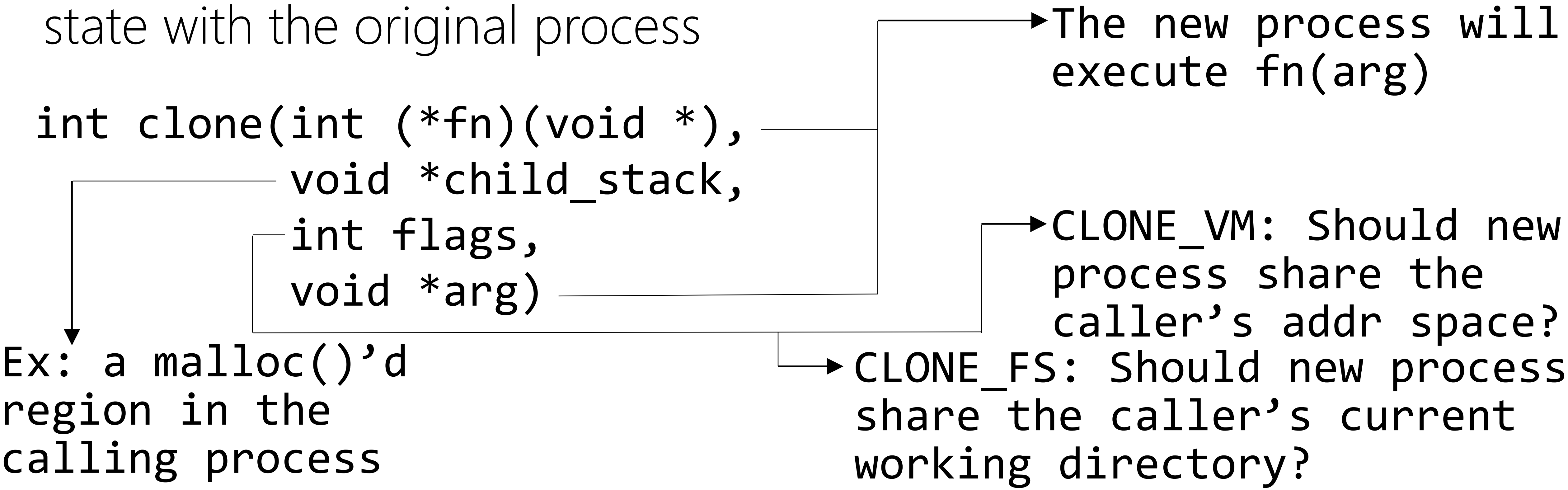
```
//Child starts
//executing.
```

```
//Later…
```

```
exit(42);
```

Making Threads On Linux Via clone()

- fork() creates a new child process from a parent process
  - The child has a copy of the memory space of the parent, and copies of a bunch of other state (e.g., open file descriptors, signal handlers, current working directory, etc.)
- clone() creates a new process that might only share *some* state with the original process



```
int clone(int (*fn)(void *),
          void *child_stack,
          int flags,
          void *arg)
```

`The new process will execute fn(arg)`

`Ex: a malloc()’d region in the calling process`

`CLONE_VM: Should new process share the caller’s addr space?`

`CLONE_FS: Should new process share the caller’s current working directory?`

Scheduling Threads

- If the kernel is thread-aware, then the kernel can schedule threads
    - Kernel picks a different thread to run when a timer interrupt fires, or when a thread makes a blocking IO call, etc.
    - Linux pthread API uses clone(), so the kernel is aware of pthread threads
- Threading can also be implemented purely at the user-level!
    - A single process can manually create separate stack regions, and explicitly switch between different execution contexts (`man swapcontext`)
    - Thread switches might occur when:
        - A thread tries to make a system call that might block, e.g., read(); the thread manager can use system calls like select() to determine in a non-blocking way which file descriptors are ready for IO
        - The compiler can also sprinkle code with calls to a thread_yield() function; this function diverts control to the thread manager

Kernel threads

Advantages

- Multiple threads from the same process can be run simultaneously on different cores
- A thread in a process can sleep without forcing the entire process to sleep

Disadvantages

- Thread creation, destruction, scheduling require a context switch into and out of the kernel (saving registers, polluting L1/L2/L3 caches, etc.—pure overhead!)

User-level threads

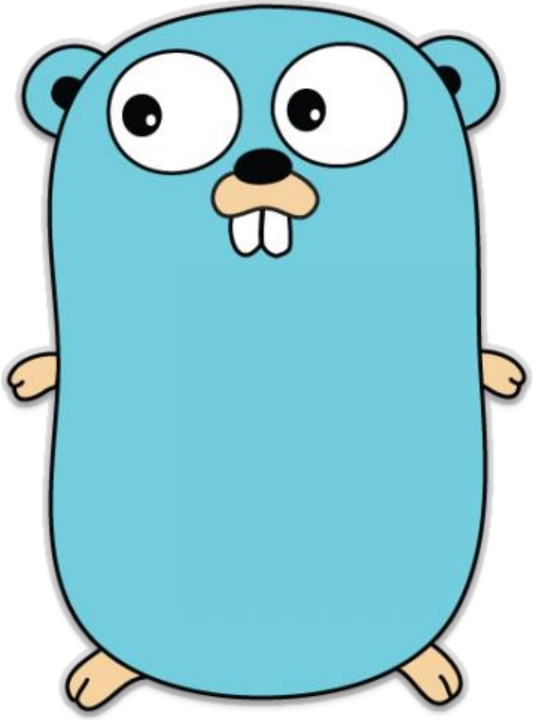
Advantages

- A process can implement application-specific scheduling algorithms
- Thread creation, destruction, scheduling don’t require context switches . . .

Disadvantages

- . . . but polling for ready file descriptors (select(), etc.) does
- Can’t leverage multiple cores, since OS only knows how to schedule processes

Hybrid Threading



The Go bison or whatever

- A single application can use multiple kernel threads, and place several user-level threads inside each kernel thread
- Example: the goroutines in a single Go program
  - GOMAXPROCS environment variable sets the number of kernel threads to use for a single Go program
  - Calls to Go runtime allow goroutine scheduler to run
  - Each goroutine gets a 2 KB stack at first
  - Each function preamble checks whether there’s enough stack space to execute the function; if not, runtime doubles the size of the stack, copies old stack into new space, updates stack pointer

kern/include/proc.h

```
/*
 * Process structure.
 * Note that we only count the number of threads in each process.
 * (And, unless you implement multithreaded user processes, this
 * number will not exceed 1 except in kproc.)
 */
struct proc {
    struct spinlock p_lock;  /* Lock for this structure */
    unsigned p_numthreads;   /* Number of threads in this process */
    struct addrspace *p_addrspace;  /* virtual address space */

    /* ...other stuff... */
};

/* This is the process structure for the kernel and for
 * kernel-only threads. */
extern struct proc *kproc;
```

kern/include/thread.h

```
/* Size of kernel stacks; must be power of 2 */
#define STACK_SIZE 4096

/* States a thread can be in. */
typedef enum {
	S_RUN,		/* running */
	S_READY,	 /* ready to run */
	S_SLEEP,	/* sleeping */
	S_ZOMBIE,	  /* zombie; exited but not yet deleted */
} threadstate_t;
```

kern/include/thread.h

```
struct thread {
        threadstate_t t_state;    /* State this thread is in */
        void *t_stack;            /* Kernel-level stack: Used for
                                   * kernel function calls, and
                                   * also to store user-level
                                   * execution context */
        struct switchframe *t_context;  /* Saved kernel-level
                                         * execution context */
        /* ...other stuff... */
}
```

Kernel Structure: Concurrency and Isolation

- When a thread makes a system call, control flow diverts to the kernel
  - Kernel code executes to handle the system call (e.g., to initiate an IO operation, to retrieve the PID of the thread, etc.)
  - Kernel code may need to sleep (e.g., because IO device is slow) . . .
  - . . . but we don’t want to busy-wait for wake condition: we want the kernel to be able to do other things on that core!
- The kernel needs a protected memory region for code, data, stack, and heap
  - Ex: Malicious/buggy user-level code should not be able to overwrite the kernel’s scheduling queues
  - Ex: Malicious/buggy user-level code should not be able to directly jump to kernel functions and skip security checks

Kernel Structure: Isolation via Hardware Privilege Modes

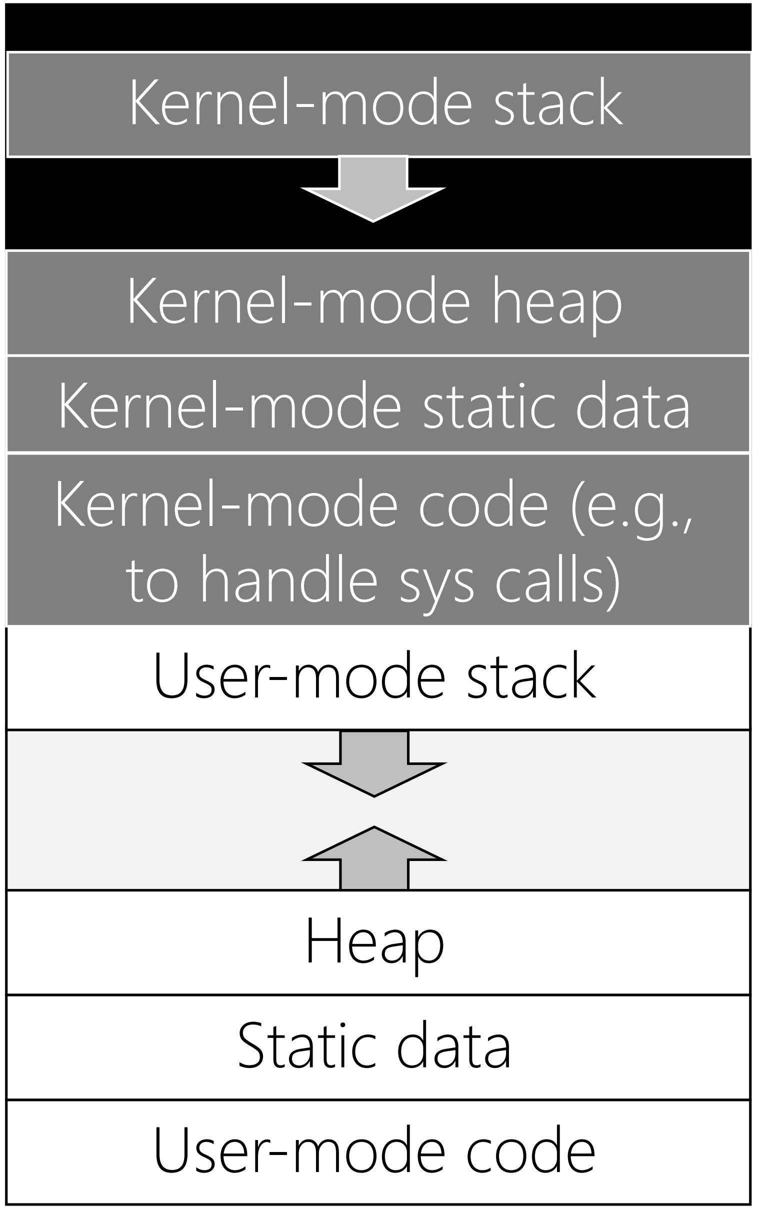
- An ISA defines privilege modes that determine:
  - which instructions are legal to execute
  - which virtual addresses are legal to access
  - how virtual addresses (i.e., the addresses that programs generate) are translated to physical addresses (i.e., the addresses that the processor gives to the memory hardware)
- Most ISAs (like MIPS) define two privilege levels
  - When a core runs in user-mode, code cannot use sensitive instructions (e.g., to directly access IO devices or memory-mapping hardware); cannot access privileged registers or privileged areas of virtual memory
  - In kernel-mode, there are no restrictions

Kernel Structure: Isolation via Hardware Privilege Modes

x86 defines four privilege levels (Ring 0—3)

Changing Privilege Levels

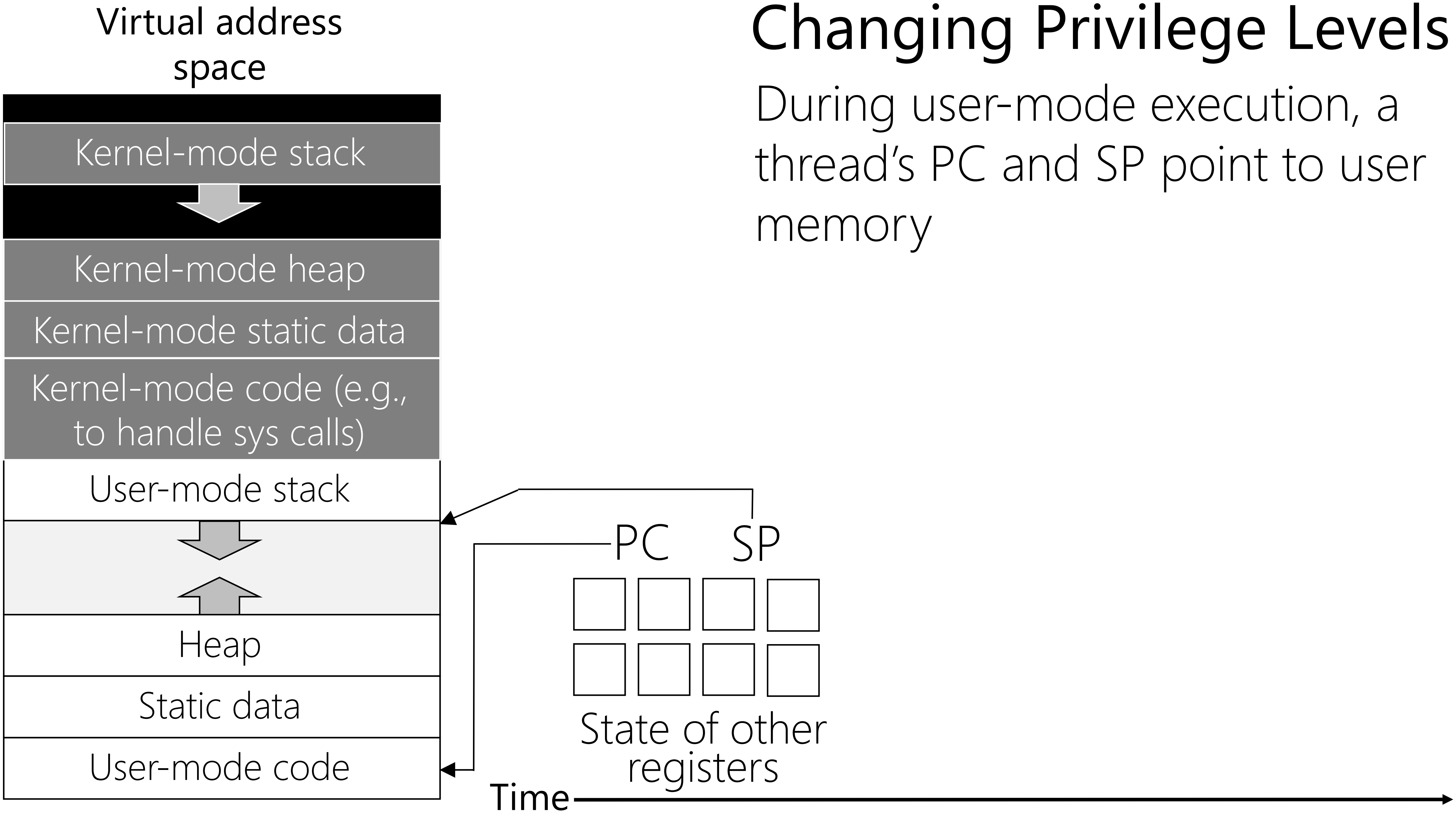
Virtual address space



- Privilege mode changes during traps and return-from-traps
- In OS161 (and many other OSes):
  - User-mode execution keeps call state on a per-thread user-level stack
  - Kernel-mode execution keeps call state on a per-thread kernel-level stack
- In OS161, a thread’s kernel stack is defined by **`struct thread::void *t_stack`**

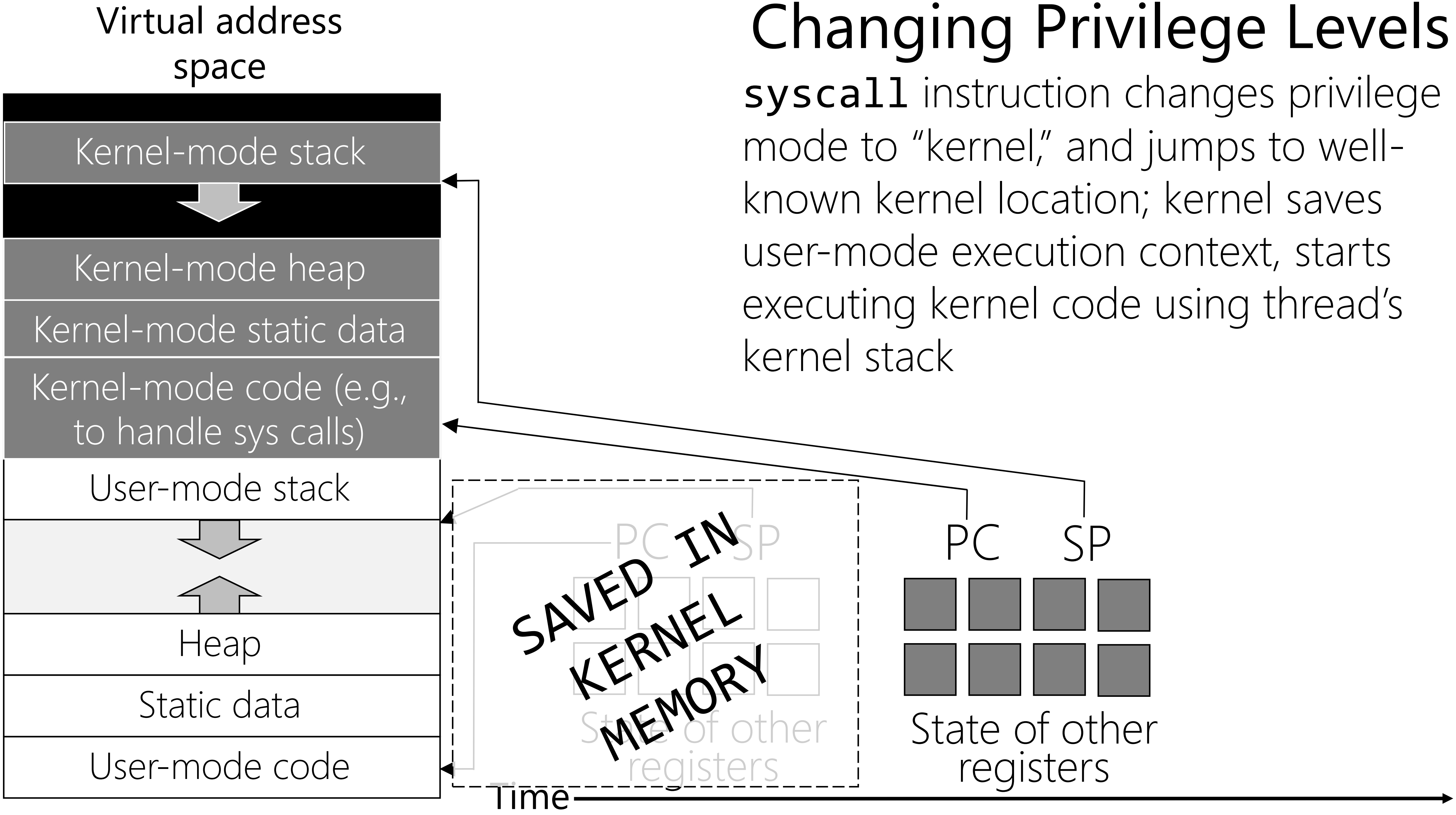
Changing Privilege Levels

During user-mode execution, a thread's PC and SP point to user memory



Changing Privilege Levels

`syscall` instruction changes privilege mode to “kernel,” and jumps to well-known kernel location; kernel saves user-mode execution context, starts executing kernel code using thread’s kernel stack



Virtual address space

Kernel-mode stack

Kernel-mode heap

Kernel-mode static data

Kernel-mode code (e.g., to handle sys calls)

User-mode stack

Heap

Static data

User-mode code

SAVED IN KERNEL MEMORY

PC SP

State of other registers

PC SP

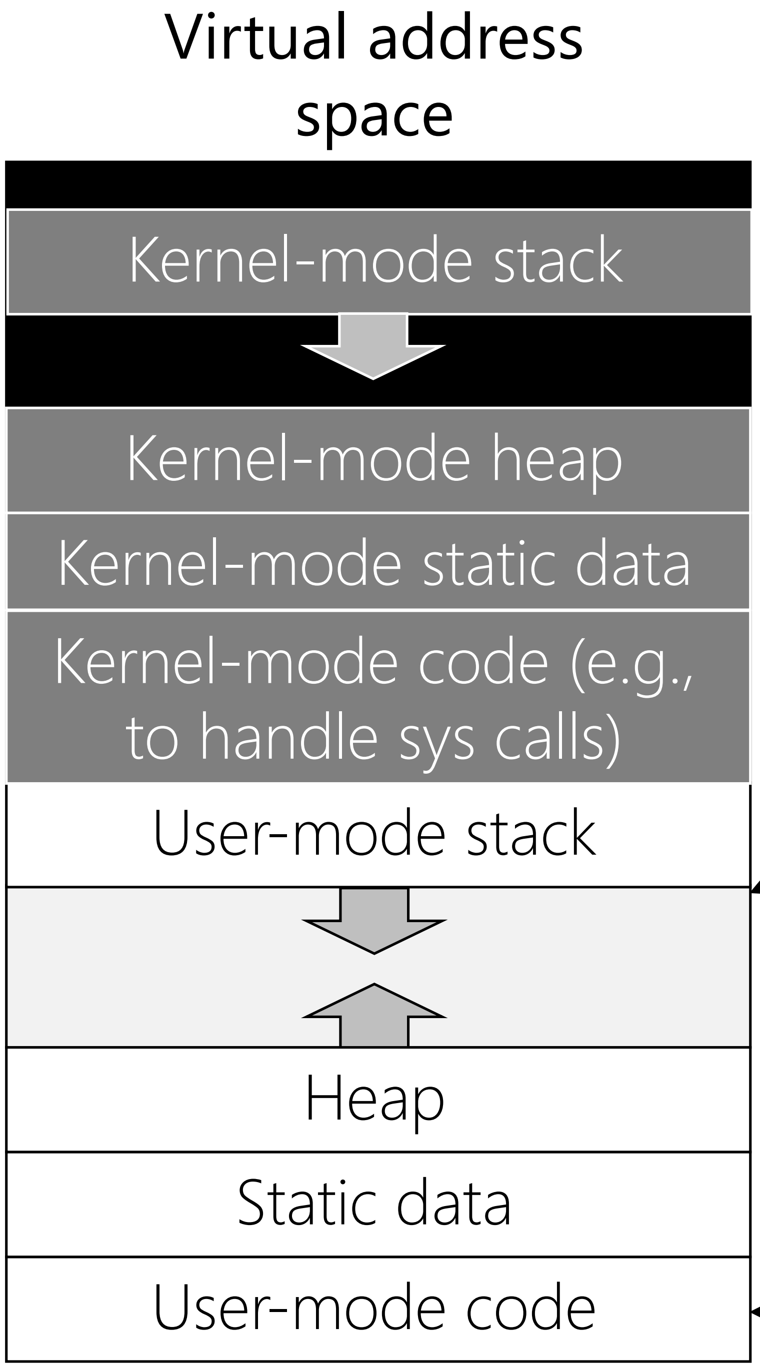
State of other registers

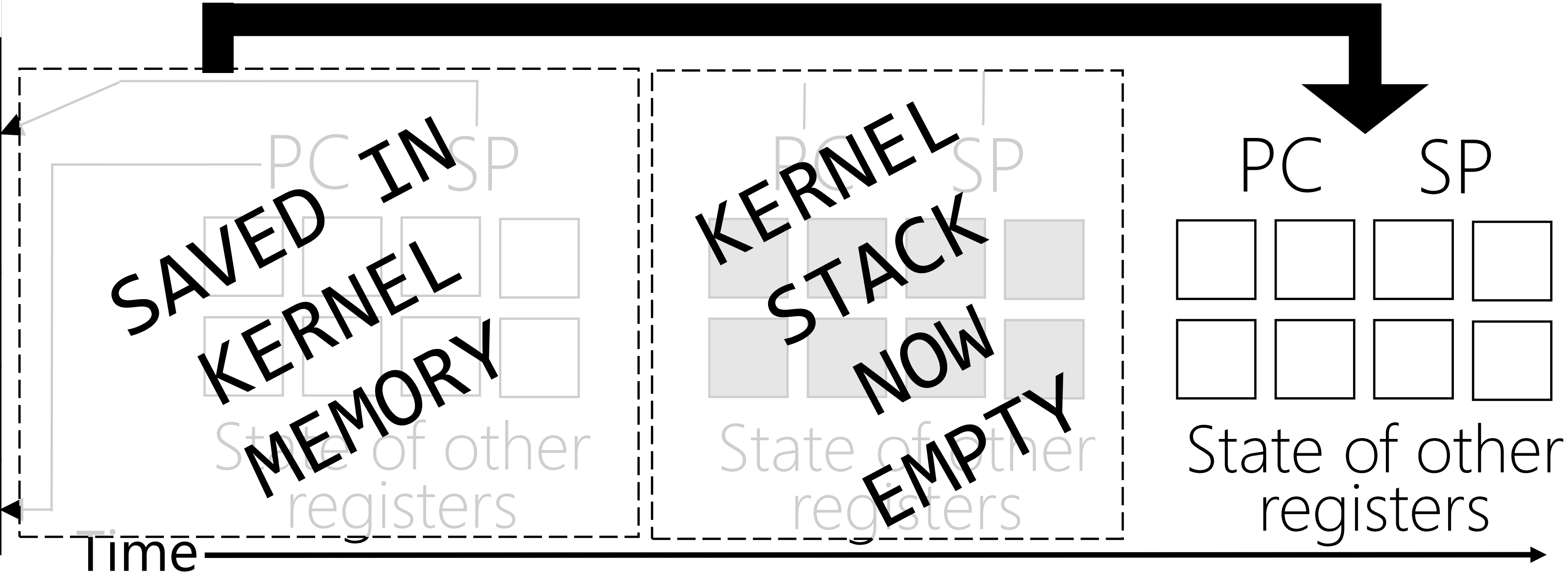
Time

Changing Privilege Levels

To return from the system call, the kernel:

- restores the user-level execution context (except PC, which is placed in k1)
- executes **`rfe`** (return from exception) to restore old privilege mode
- Executes **`jr k1`** to jump to the next user-level instruction to execute





What If A Thread Needs To Wait?

- Previous example assumed a system call that returns immediately (e.g., **`getpid()`**) . . .
- . . . but sometimes, a thread must wait in the kernel for something to happen
  - Ex: A blocking **`read()`** on an IO device like a disk
  - Ex: A call to **`lock_acquire(lock)`** if **`lock`** is already owned by another thread
- In these cases, the kernel must mark the thread as “sleeping,” and add the thread to a wait queue
  - The kernel pulls a new thread from the ready queue to run
  - Later, when the waited-upon condition becomes true, the kernel moves the original thread from the wait queue to the ready queue
  - At some point, the kernel pulls the original thread from the ready queue and actually schedules it on a core

OS161 wchans (“Wait Channels”)

```
/* Wait channel. A wchan is protected by
 * an associated, passed-in spinlock. */
struct wchan {
    const char *wc_name; /* name for this channel */
    struct threadlist wc_threads;  /* waiting threads */
};
```

```
/*
 * Yield the cpu to another process, and go to sleep,
 * on the specified wait channel WC, whose associated
 * spinlock is LK. Calling wakeup on the channel will
 * make the thread runnable again. The spinlock must
 * be locked. The call to thread_switch unlocks it; we
 * relock it before returning.
 */
void
wchan_sleep(struct wchan *wc, struct spinlock *lk){
    /* may not sleep in an interrupt handler */
    KASSERT(!curthread->t_in_interrupt);

    /* must hold the spinlock */
    KASSERT(spinlock_do_i_hold(lk));

    /* must not hold other spinlocks */
    KASSERT(curcpu->c_spinlocks == 1);

    thread_switch(S_SLEEP, wc, lk); //Adds this thread
                                    //to wc->wc_threads
    spinlock_acquire(lk);
}
```

```
/*
 * Wake up one thread sleeping on a wait channel.
 */
void
wchan_wakeone(struct wchan *wc, struct spinlock *lk){
    struct thread *target;

    KASSERT(spinlock_do_i_hold(lk));
    target = threadlist_remhead(&wc->wc_threads);
    if (target == NULL) { /* Nobody was sleeping. */
        return;
    }
    thread_make_runnable(target, false); /* Adds to ready
                                          * queue! */
}
```