The Process

CS 450: Operating Systems

Michael Lee <lee@iit.edu>

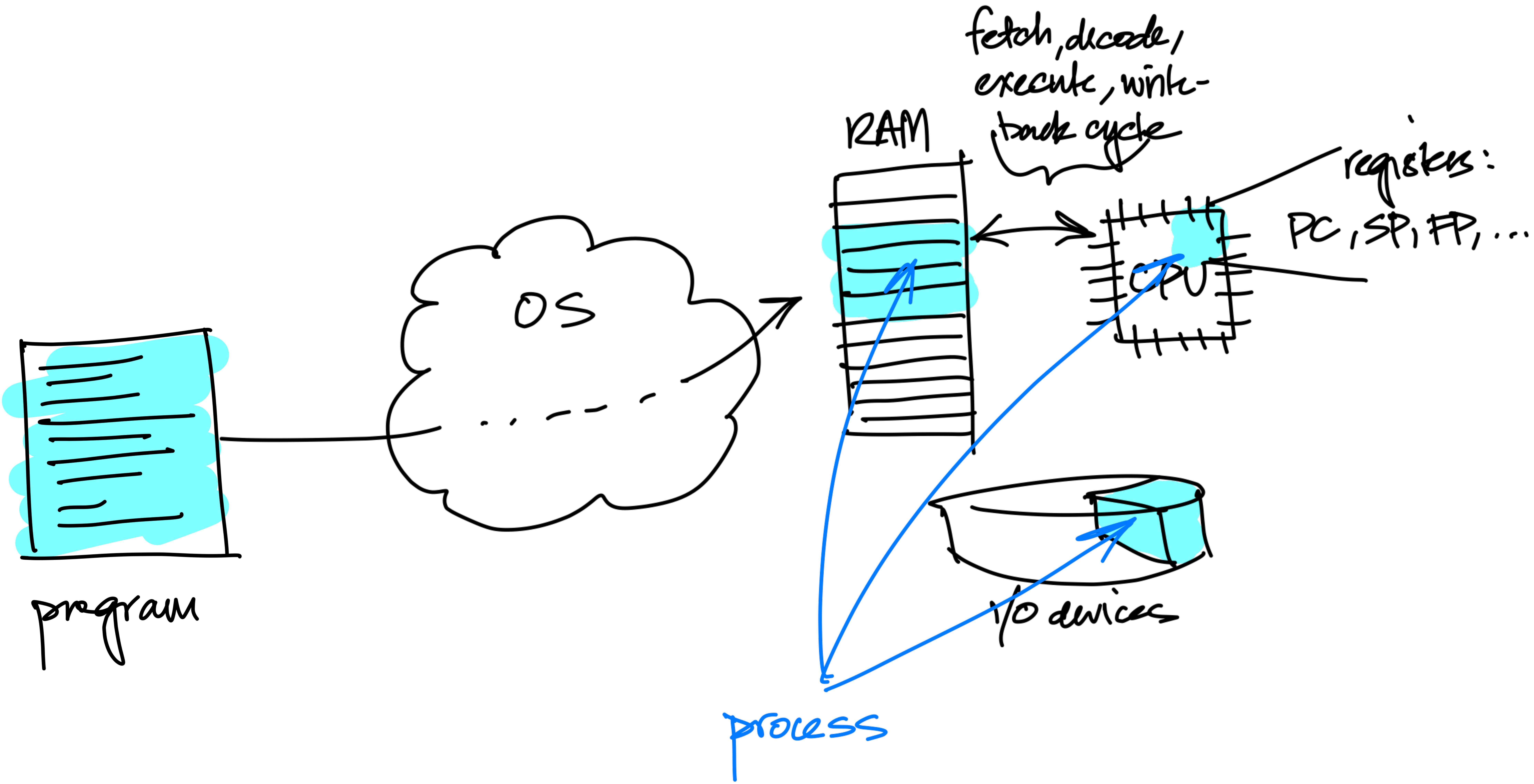
ILLINOIS TECH | College of Computing

Agenda

- The Process: what is it and what's in it?
- Forms of Multitasking
- Tracking processes in the OS
- Context switches and Scheduling
- Process API

a **process** is a *program in execution*

- its behavior is largely defined by the program being executed
- but a process is much more than just a program!

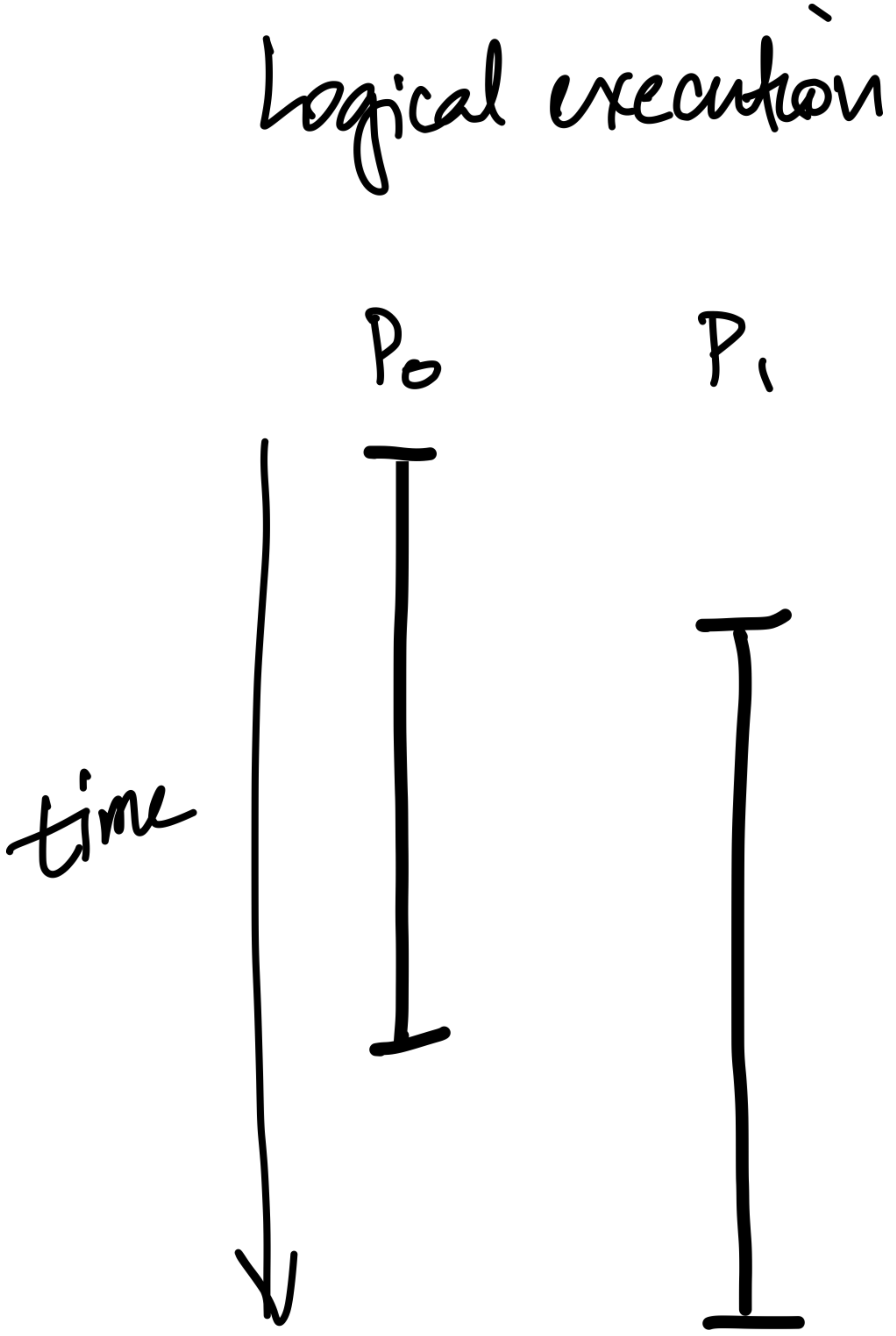

fetch, decode, execute, write-back cycle
RAM
registers:
PC, SP, FP, ...
CPU
OS
program
I/O devices
process

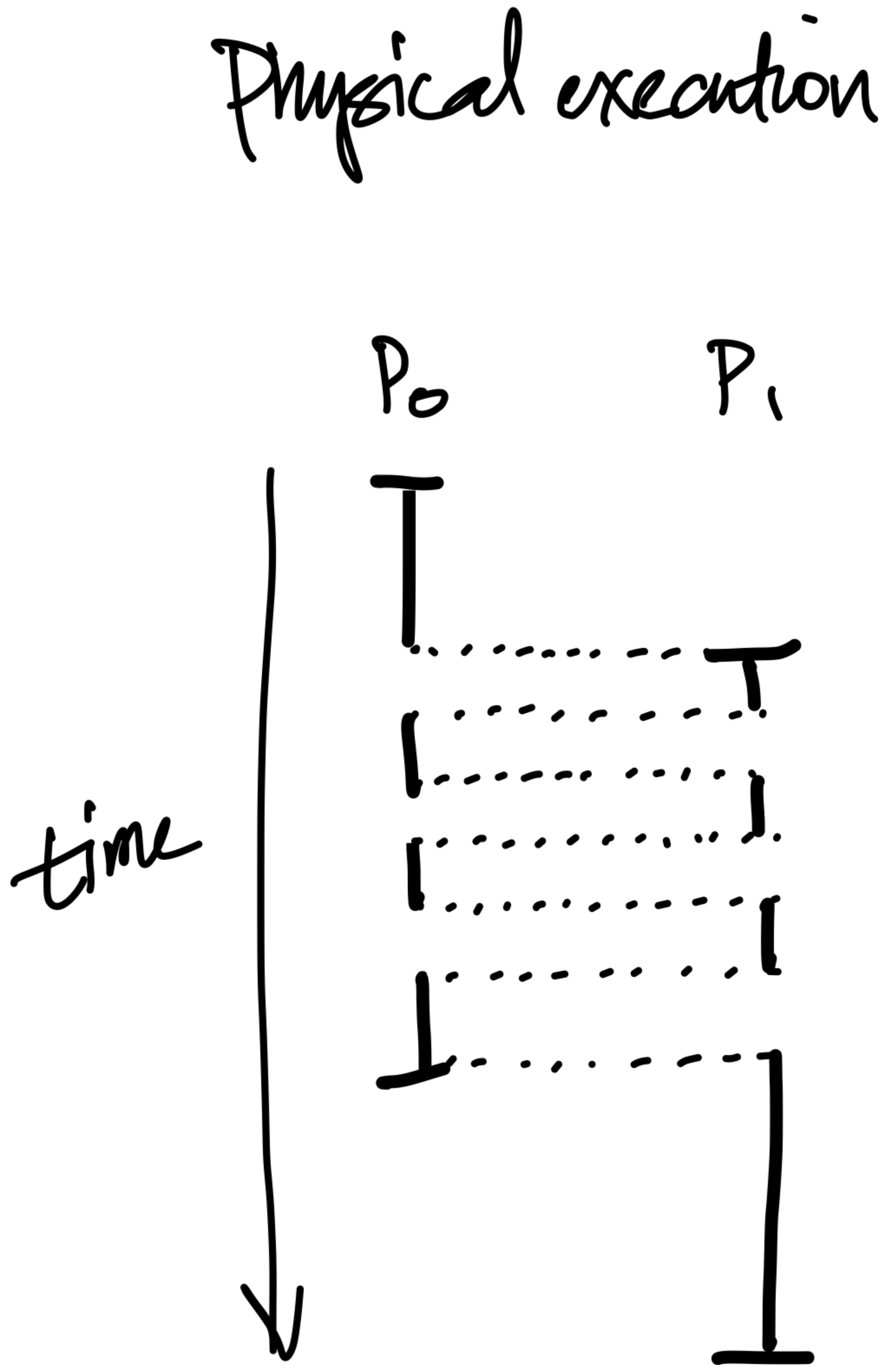
Multitasking

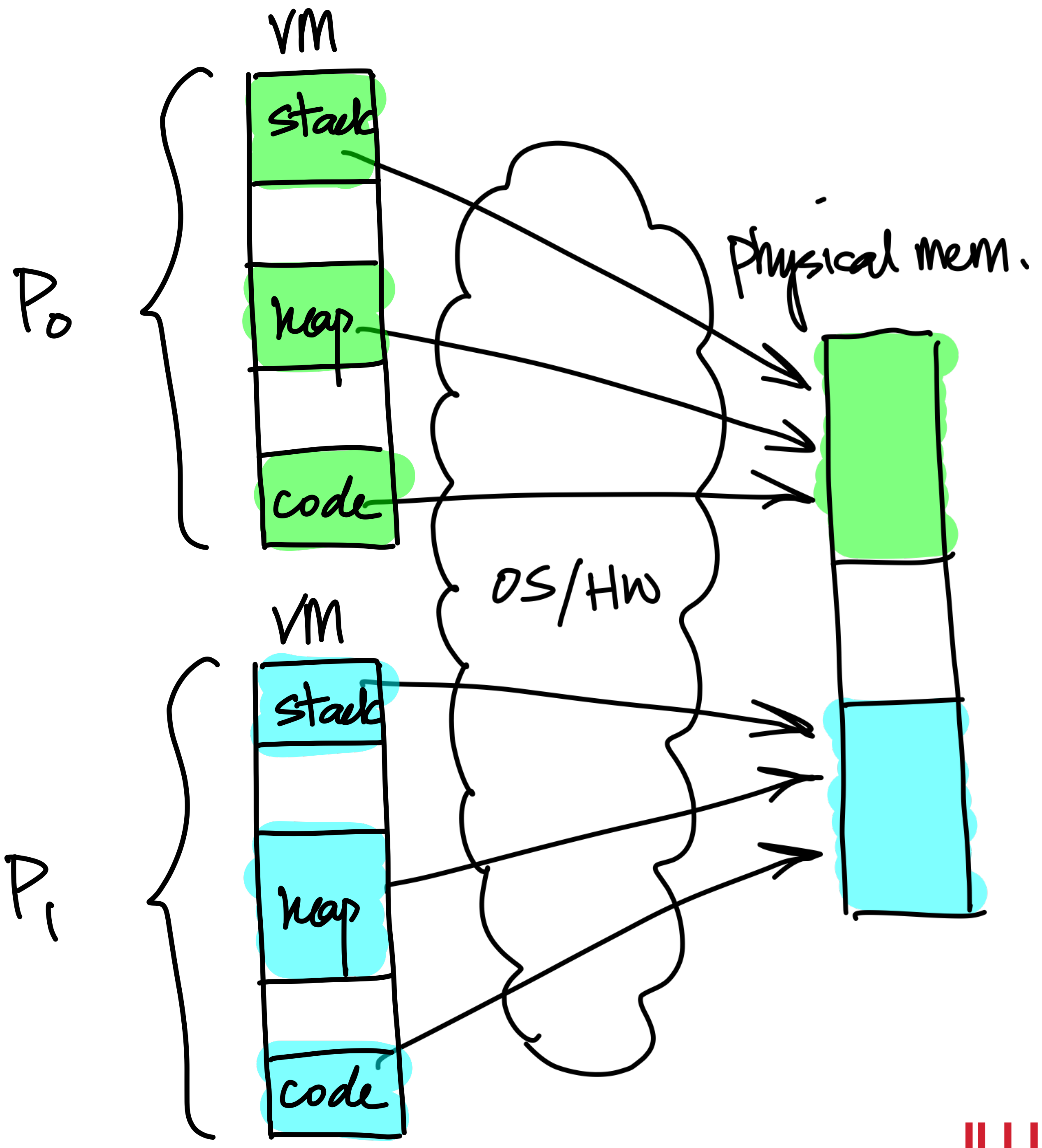
- Modern general-purpose OSes typically run dozens to hundreds of processes simultaneously
  - May collectively exceed capacity of hardware
  - Recall: *virtualization* allows each process to ignore physical hardware limitations and let OS take care of details

CPU/Memory Virtualization

- *Time-slicing* of CPU(s) is performed to simulate concurrency
- Memory is partitioned and shared amongst processes
  - But per-process view is of a *uniform address space*
  - *Lazy/On-demand loading* of processes lowers total burden

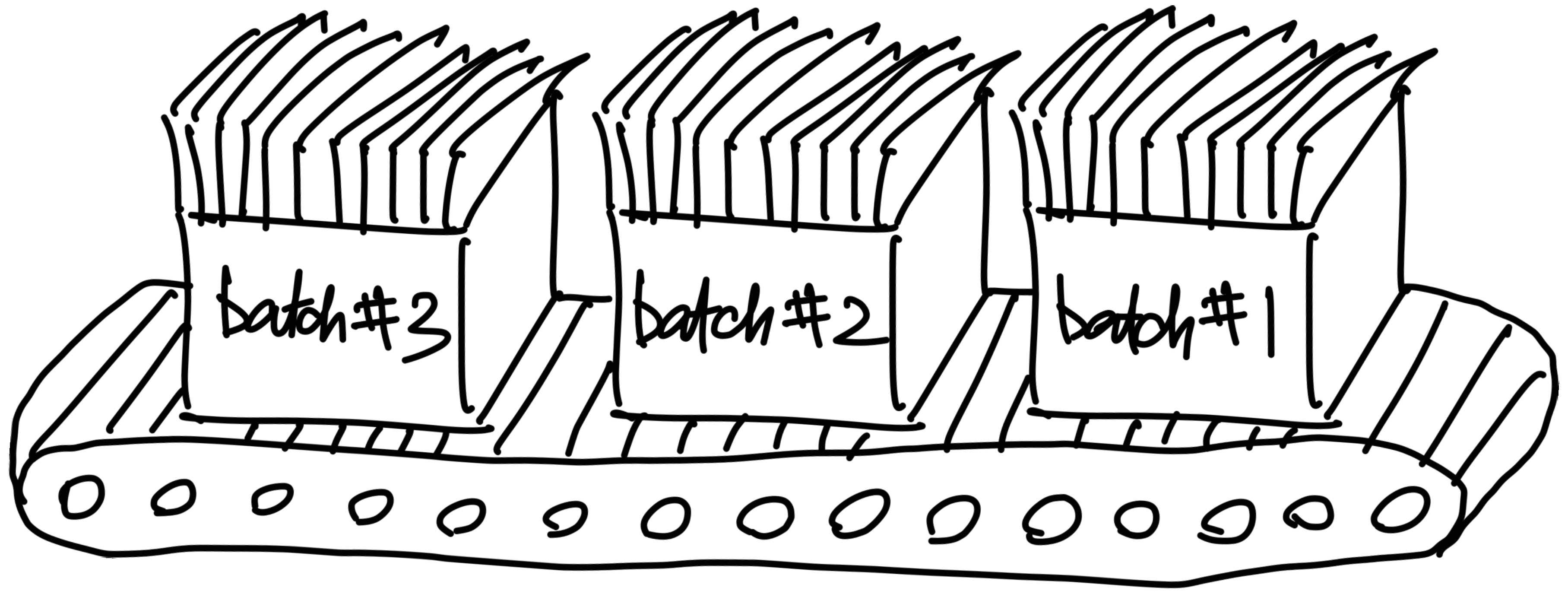

Logical execution
P_0
P_1
time


Physical execution
P_0
P_1
time


VM
stack
heap
code
P_0
VM
stack
heap
code
P_1
OS/HW
physical mem.

vs. “Batch” processing

- Without multitasking, each program is run from start to finish without interruption from other processes
  - Including any I/O operations (which may be lengthy!)
  - Ensures minimal overhead (but at what cost?)
  - Is virtualization still necessary?



Pros/Cons of Multitasking

- Pro: may improve resource *utilization* if we can run some processes while others are *blocking*
- Pro: makes process *interaction* possible
- Con: virtualization introduces *overhead* (examples?)
- Con: possibly reduced overall *throughput*

Forms of Multitasking

- *Cooperative* multitasking: processes voluntarily cede control
- *Preemptive* multitasking: OS polices transitions (how?)
- *Real-time* systems: hard, fixed time constraints (late is wrong!)

What’s in a process?

- Program (“text”) and data
  - Static/Stack/Heap memory contents
- Registers (e.g., PC, SP, FP)
- Open files and devices (e.g., network)
- What else?

Data vs. Metadata

- User-maintained data vs. Kernel-maintained data
- Metadata examples:
  - PID, GID, UID
  - Allotted CPU time
  - Virtual → Physical memory mapping
  - Pending I/O operations

OS Data Structures

- Critical function of OS is to maintain data structures for keeping track of and managing all current processes
- Layout of many structures are dictated by hardware
  - e.g., VM structures, interrupt stack frame

PCB

- Aggregate per-process data entry is referred to as the *Process Control Block* (PCB)
  - Implementation likely consists of many disparate structures

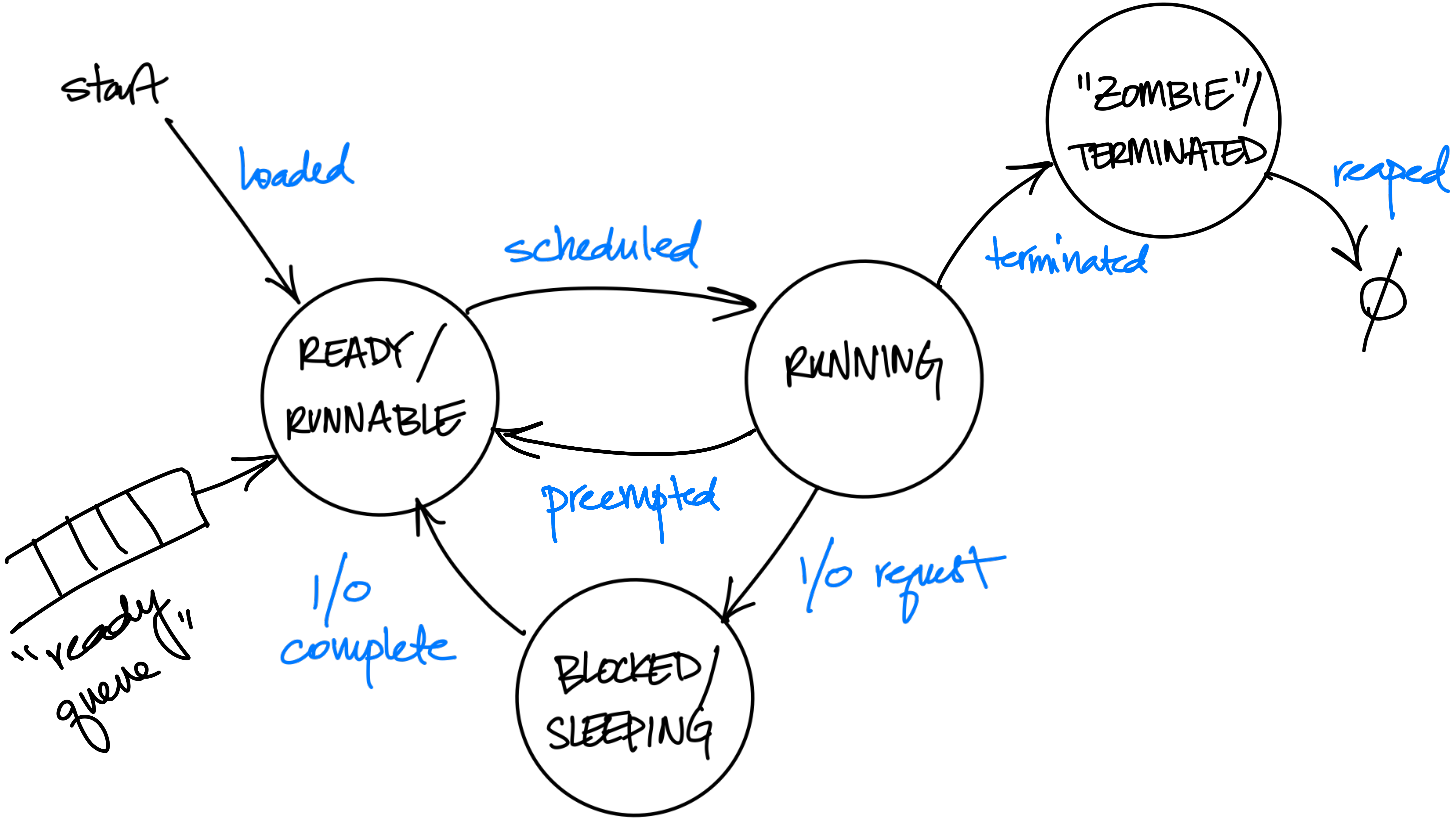
```c
// xv6 PCB components (not comprehensive!)

struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

struct proc {
  uint sz;                        // Size of process memory (bytes)
  pde_t* pgdir;                   // Page table
  char *kstack;                   // Bottom of kernel stack for this process
  enum procstate state;           // Process state
  int pid;                        // Process ID
  struct proc *parent;            // Parent process
  struct trapframe *tf;           // Trap frame for current syscall
  struct context *context;        // swtch() here to run process
  void *chan;                     // If non-zero, sleeping on chan
  int killed;                     // If non-zero, have been killed
  struct file *ofile[NOFILE];     // Open files
  struct inode *cwd;              // Current directory
  char name[16];                  // Process name (debugging)
};
```

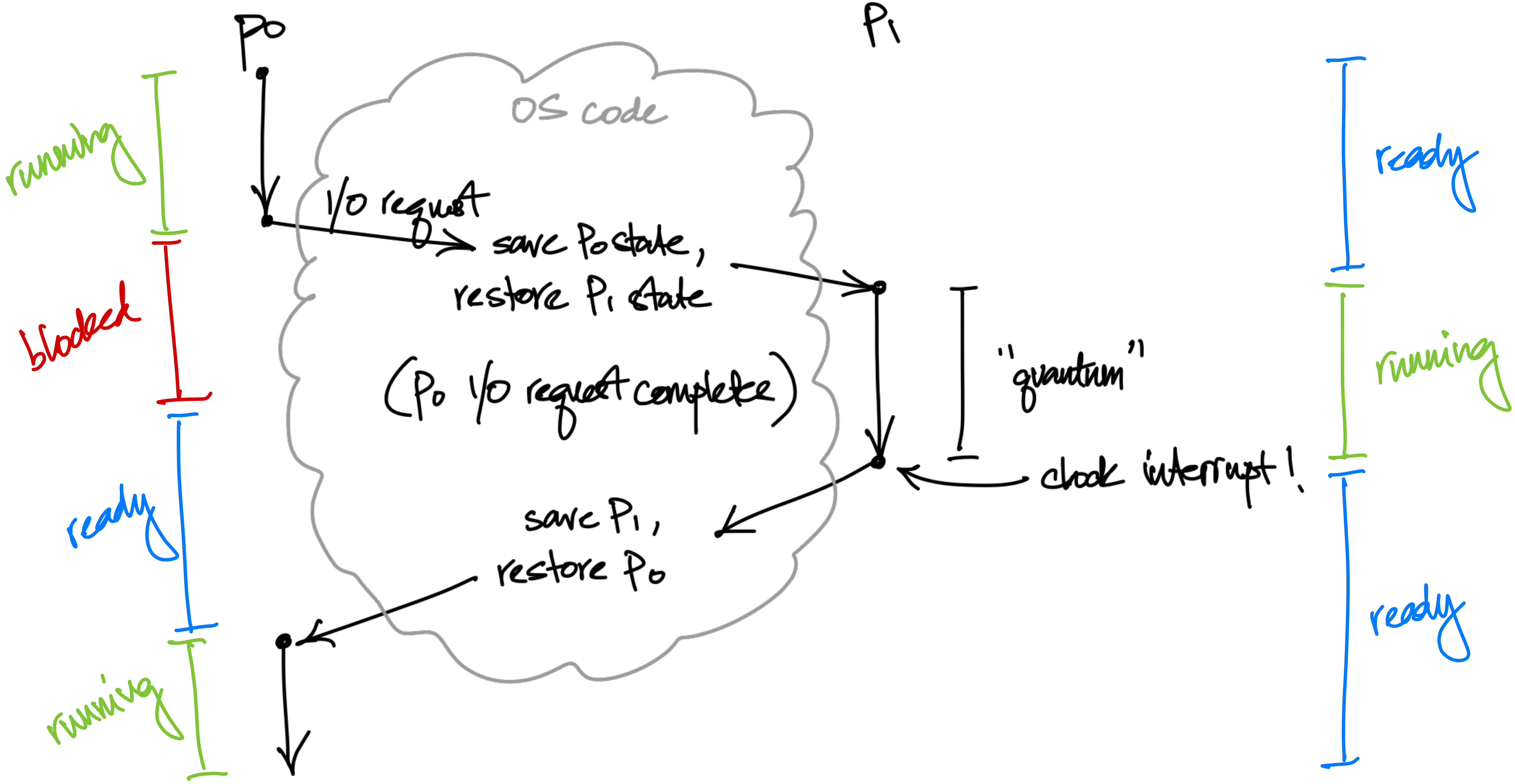
Context Switches

- Multitasking via virtualization relies on seamlessly *switching contexts* between processes on hardware
  - Requires frequently saving/loading state to/from PCB
- At any point may have multiple processes *ready* to run
  - How to pick the incoming process?

Scheduler

- Combination of *policies* & *mechanisms* used to select which process is allocated resources
- Can express operations in a state transition diagram


start
loaded
READY / RUNNABLE
scheduled
RUNNING
preempted
"ZOMBIE"/ TERMINATED
terminated
reaped
∅
I/O request
BLOCKED/ SLEEPING
I/O complete
"ready queue"



P_0
P_1
OS code
running
I/O request
save P_0 state,
restore P_1 state
ready
blocked
"quantum"
running
(P0 I/O request complete)
clock interrupt!
ready
save P_1,
restore P_0
running
ready

Policy vs. Mechanism

- Recurring theme in OS (and general software) implementation
- Ideally: keep policy separate from mechanism (why?)
  - Cross-cutting issues may be difficult to isolate, resulting in a high degree of *coupling* between modules
- API vs. Implementation is an example of policy vs. mechanism

Unix Process API

- Set of flexible, *orthogonal* process APIs that enable:
  - Creation & Program execution
  - Management (e.g., suspension, destruction, synchronization)
  - Metadata access (e.g., status, termination conditions)
  - Interoperation

Unix Process API (partial)

- Creation: `fork`
- Program execution: `exec`
- Synchronization: `wait`
- Termination: `exit`

```c
/* Simple forking example */

if (fork() == 0) {
    /* in child */
    printf("Hello from child!\n");
} else {
    /* in parent */
    printf("Hello from parent!\n");
}
```

```c
/* Primitive Unix shell: OS “interface" */

/* Read-Eval Loop */
while (1) {
    printf("$ ");  /* print prompt */

    /* read command and build argv */
    fgets(buf, MAXLINE, stdin);

    /* fork child process */
    if (fork() == 0) {
        /* parse command line into arguments */
        parsecmd(buf, argv);

        /* execute argument program in child */
        if (execvp(argv[0], argv) < 0) {
            printf("Command not found\n");
            exit(0); // terminate
        }
    }

    /* wait for child completion in parent */
    wait(&status);
}
```

API vs. Kernel Implementation

- Unix API has stood the test of time — large parts unchanged from earliest versions
  - “Those who don't understand Unix are condemned to reinvent it, poorly.” (Henry Spencer)
- But this doesn’t mean we can’t re-engineer things under the hood!