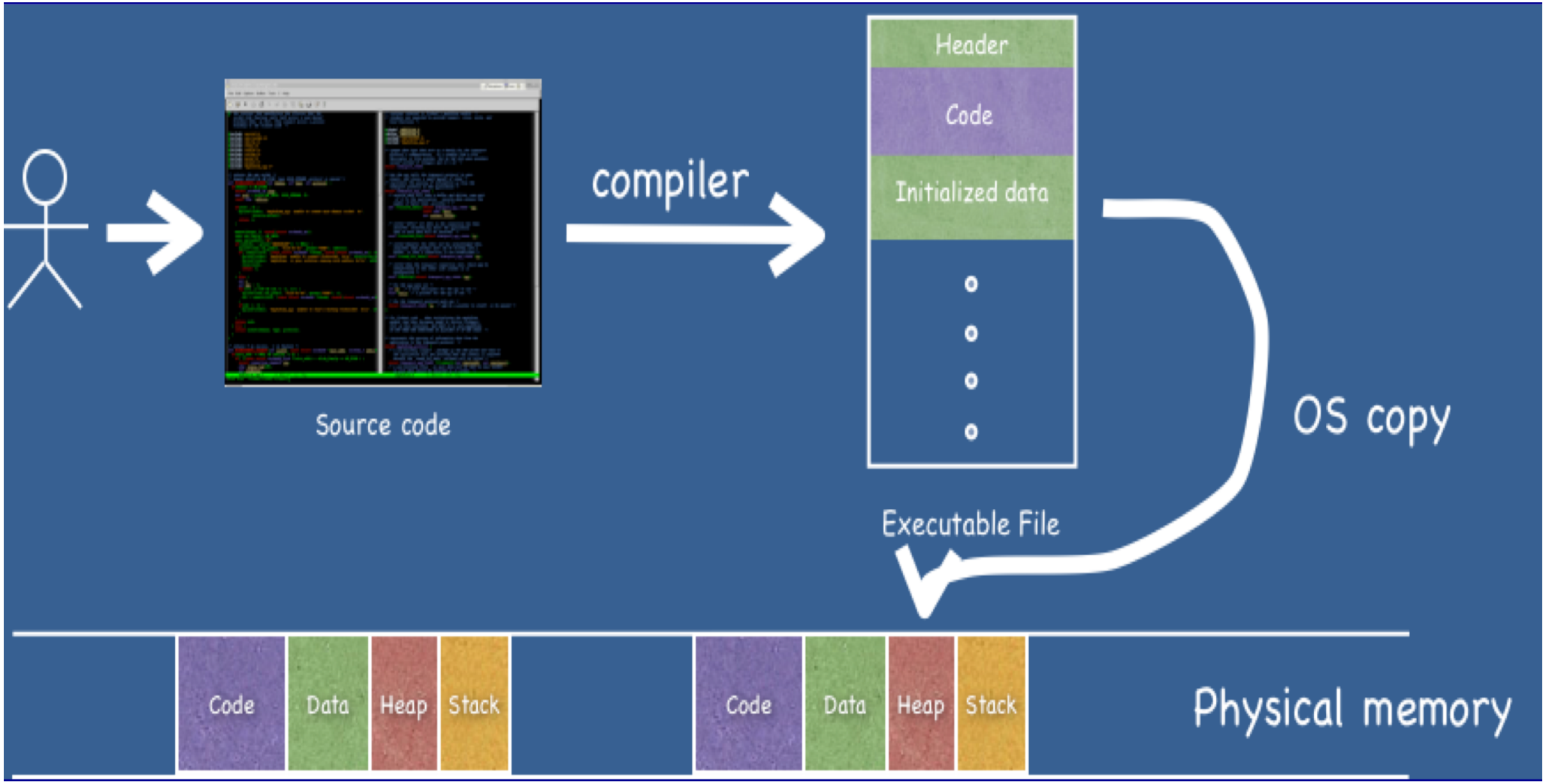
Processes

Bringing It All Together

- Operating system tasks can be divided into three main roles: the referee, the illusionist, and the glue
- Operating system complexity increased over time in response to economic and technological changes
  - The three roles did not show up all at once and fully formed!

What is a Process?

- A process is a program during execution.
  - Program = static file (image)
  - Process = executing program = program + execution state
- A process is the basic unit of execution in an OS
- Different processes may run different instances of the same program
  - e.g., my gcc and your gcc process both run the GNU C compiler
- At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution


Header
Code
Initialized data
compiler
Source code
OS copy
Executable File
Code
Data
Heap
Stack
Code
Data
Heap
Stack
Physical memory

Process State

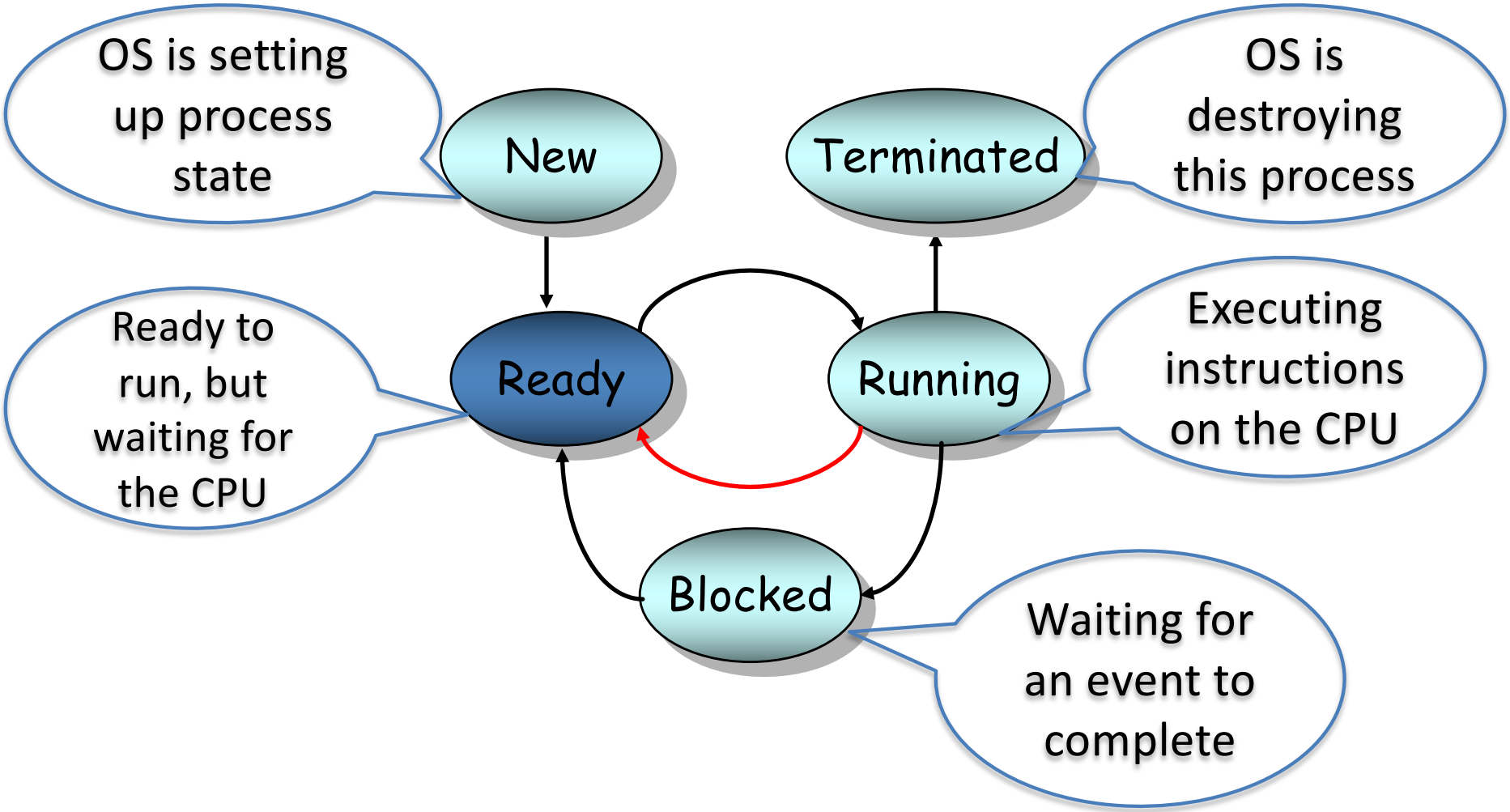
Process state consists of at least:

- The contents of the address space, including:
  - The code for running the program
  - The static data for running the program
  - An execution stack with the program's call chain (the stack)
  - The heap, where dynamic data is allocated
- Registers and their contents, including:
  - The heap pointer (HP)
  - The Program Counter (PC) indicating the next instruction
  - The stack pointer (SP)
- The set of OS resources in use (e.g., open files)
- Process identifier (PID)
- Process execution state (ready, running, etc.)

Process Life Cycle

Process Life Cycle

Processes are always either *running*, *ready to run* or *blocked waiting for an event* to occur



How does the OS track this data?

The OS uses a *Process Control Block* (PCB)

- Dynamic kernel data structure kept in memory
- Represents the execution state and location of each process when it is not executing

The PCB contains:

- Process identification number, program counter, stack pointer, contents of general purpose registers, memory management information (HP, etc), username of owner, list of open files… (basically any process execution state that is not stored in the address space)

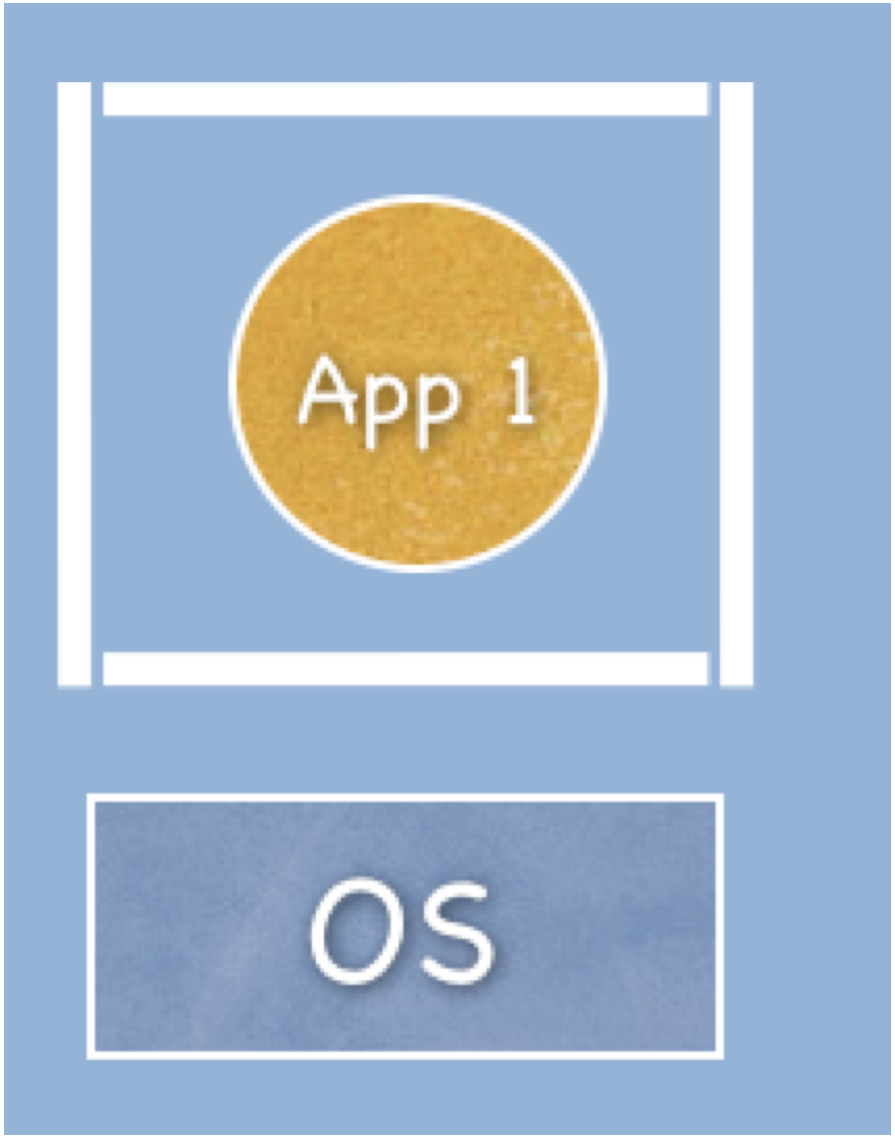
PCBs are initialized when a process is created and deleted when a process terminates

iClicker Question

When a process is waiting for I/O, what is its scheduling state?

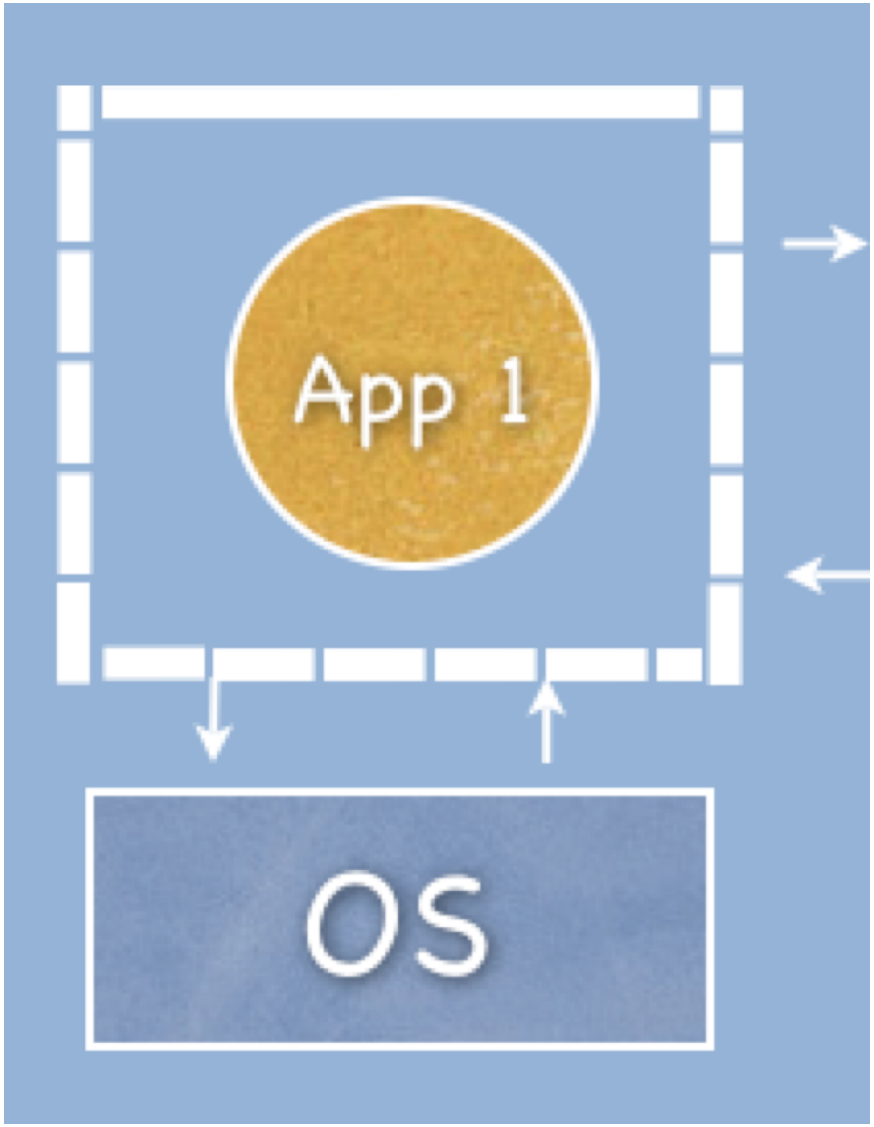
A. Ready
B. Running
C. Blocked
D. Zombie
E. Exited

The Process: Boxes in the Application



- An abstraction for protection
  - Represents an application program executing with restricted rights
- Restricting rights must not hinder functionality
  - Must still allow efficient use of hardware

The Process: Boxes in the Application



- An abstraction for protection
  - Represents an application program executing with restricted rights
- Restricting rights must not hinder functionality
  - Must still allow efficient use of hardware
  - Must still allow safe communication

If Applications Had Free Rein…

Buggy or malicious applications could:

- crash other applications
- violate privacy of other applications
- hog all the resources
- change the OS
- crash the OS

We would be trusting every software developer!

Dual Mode Execution

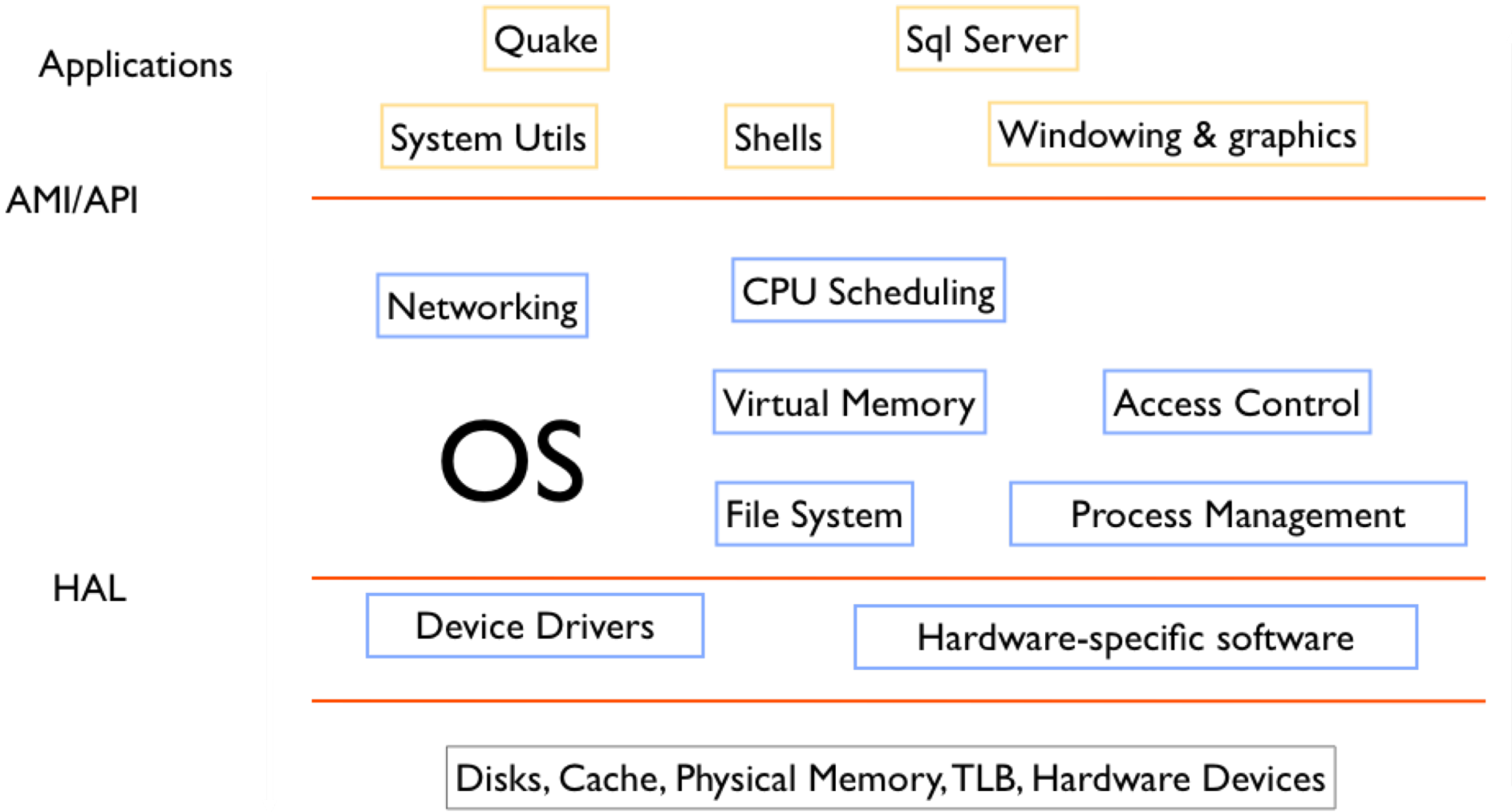
OS Interfaces

We know that the OS has three interfaces

- Abstract Machine Interface (AMI)
  - between OS and apps: API + memory access model + legally executable instructions
- Application Programming Interface (API)
  - function calls provided to apps
- Hardware Abstraction Layer (HAL)
  - abstracts hardware *internally to the OS*

Why?

Logical OS Structure



How can the OS enforce restricted rights?

- Easy: OS interprets each instruction!
  - Good solution?
    - No! Slow
  - Most instructions are safe: can we just run them in hardware?
- *Dual Mode Execution*
  - User mode: access is restricted
  - Kernel mode: access is unrestricted
  - Supported by the hardware
    - Mode is indicated by a bit in the process status register

Kernel vs. User Mode: Privileged Instructions

User processes may not:

- address I/O directly
- use instructions that manipulate OS memory (e.g., page tables)
- set the mode bits that determine user or kernel mode
- disable and enable interrupts
- halt the machine

But in kernel mode, the OS does all these things.

Executing a privileged instruction while in user mode causes a processor exception…

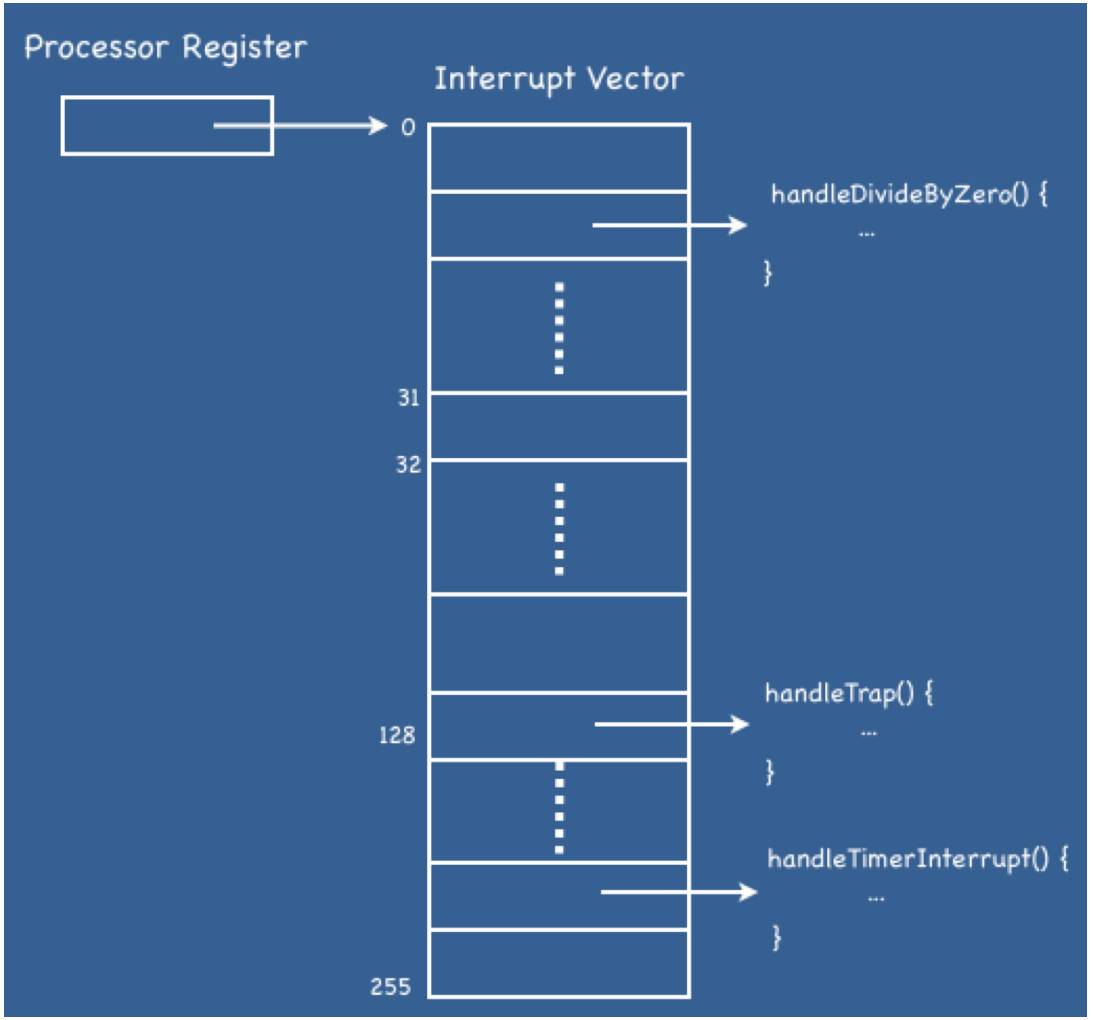
...which passes control to the kernel

Transitioning from User Mode to Kernel Mode…

- Often called *entering the kernel*
- Three methods:
  - Exceptions
    - user program acts silly (e.g. division by zero)
    - or attempts to perform a privileged instruction
      - sometimes on purpose! (breakpoints)
    - synchronous (related to instruction that just executed)
  - Interrupts (asynchronous exceptions)
    - something interrupts the currently executing process
      - timer, HW device requires OS service, …
    - asynchronous (not related to instruction that just executed)
  - System calls/Traps
    - user program requests OS service
    - looks like a function call
    - synchronous

User Mode to Kernel Mode: Details

- OS saves state of user program
- Hardware identifies why boundary is crossed
  - system call?
  - interrupt? then which hardware device?
  - which exception?
- Hardware selects entry from interrupt vector
- Appropriate handler is invoked



Saving the State of the Interrupted Process

- Privileged hw register points to exception stack
  - on switch, hw pushes some of interrupted process registers (SP, PC, etc) on exception stack before handler runs. Why?
  - then handler pushes the rest
  - On return, do the reverse
  - One exception stack per process/thread
- Why not use user-level stack?
  - reliability: even if user's stack points to invalid address, handlers continue to work
  - security: kernel state should not be stored in user space (or could be read/written)

Switching Back!

- From an interrupt, just reverse all steps!
  - asynchronous, so not related to executing instruction
- From exception and system call, increment PC on return
  - synchronous, so you want to execute the *next* instruction, not the same one again!
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by the hardware

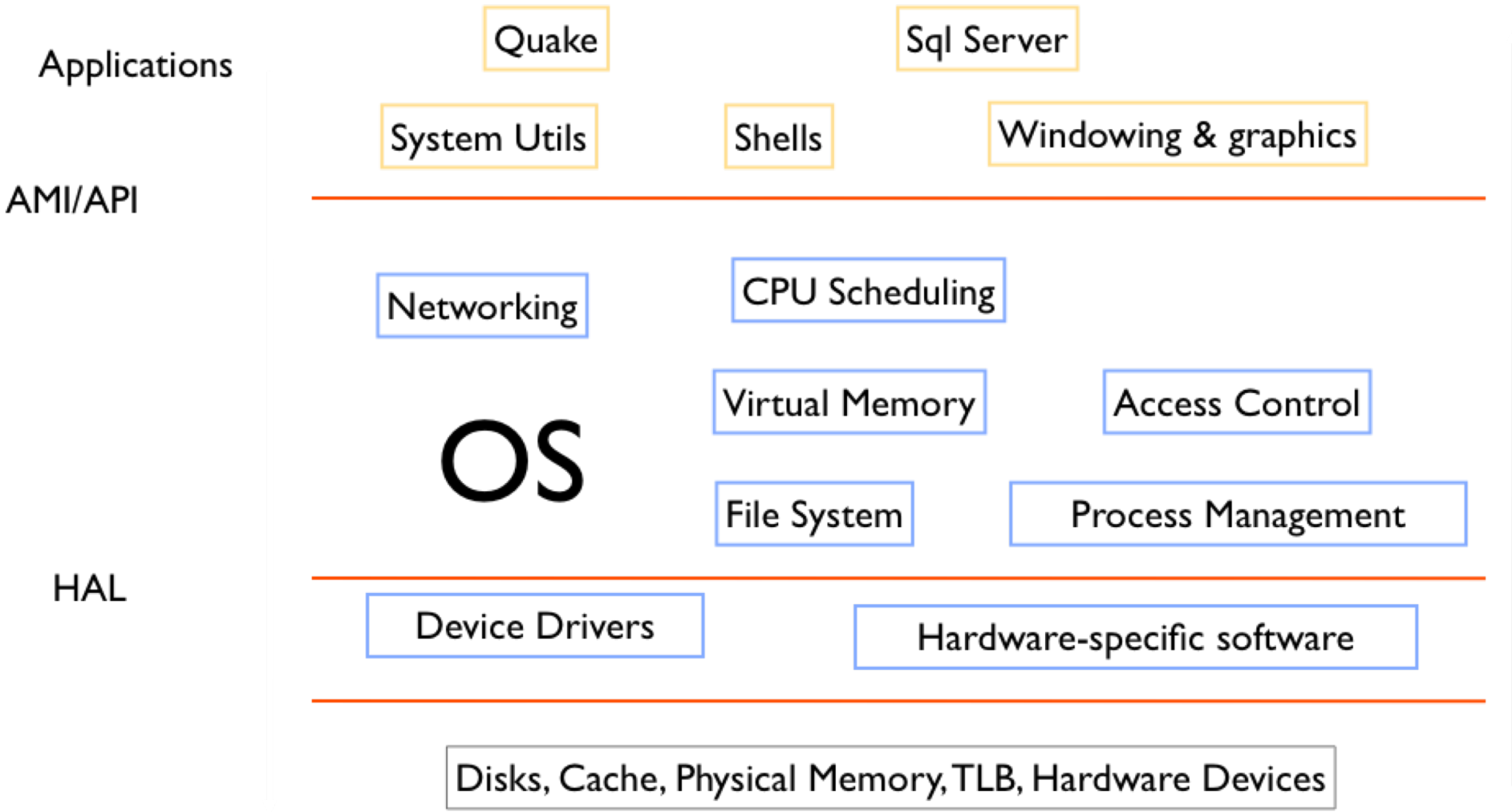
Dual Mode Execution: One Piece of the Protection Pie

For efficient protection, the hardware must support at least three features:

- Privileged instructions
  - Instructions only available in kernel mode
  - In user mode, no way to execute potentially unsafe instructions
  - Prevents user processes from, for instance, halting the machine
  - Implementation: mode status bit in the process status register
- Timer interrupts
  - Kernel must be able to periodically regain control from running process
  - Prevents process from gaining control of the CPU and never releasing it
  - Implementation: hardware timer can be set to expire after a delay and pass control back to the kernel
- Memory protection
  - In user mode, memory accesses outside a process' memory region are prohibited
  - Prevents unauthorized access of data
  - Implementation: We'll return to this later in the course

Accessing System Resources

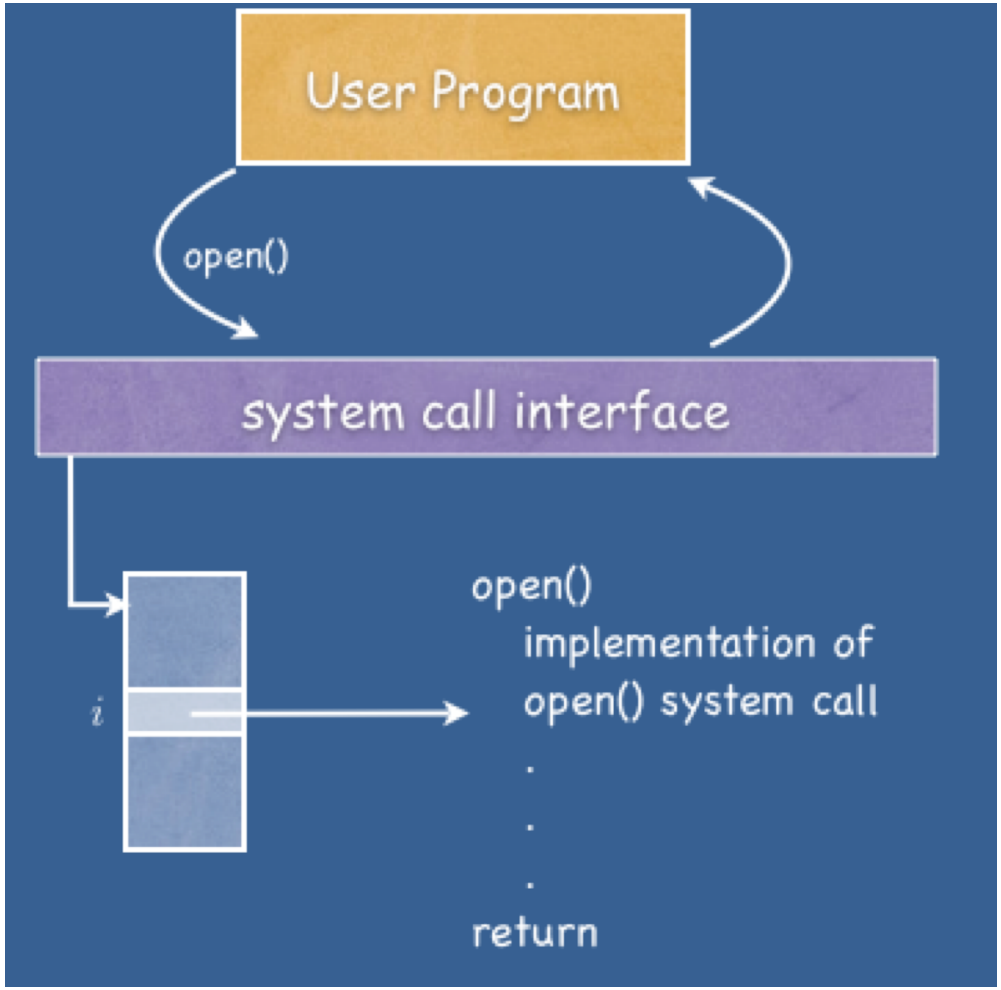
Logical OS Structure



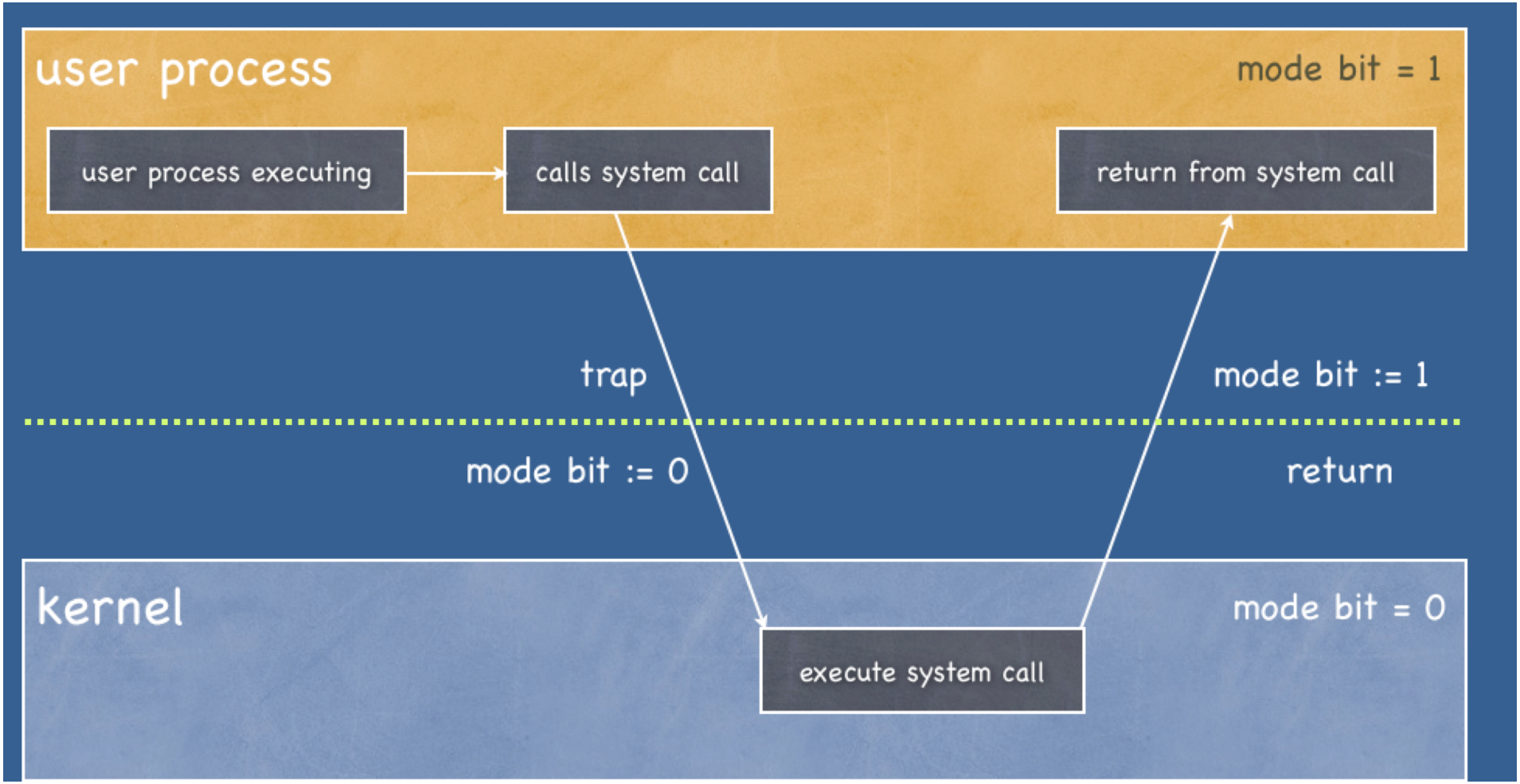
System Calls

System Calls

- A request by a user-level process to call a function in the kernel is a *system call*
  - Examples: read(), write(), exit()
- The interface between the application and the operating system (API)
  - Mostly accessed through *system-level libraries*
- Parameters passed according to calling convention
  - registers, stack, etc



System Calls: A Closer Look



System Calls: A Closer Look

- User process executes a trap instruction
- Hardware calls the OS at the system-call handler, a pre-specified location
- OS then:
  - identifies the required service and parameters (e.g. open(filename, O_RDONLY))
  - executes the required service
  - sets a register to contain the result of call
  - Executes an RTI instruction to return to the user program
- User program receives the result and continues

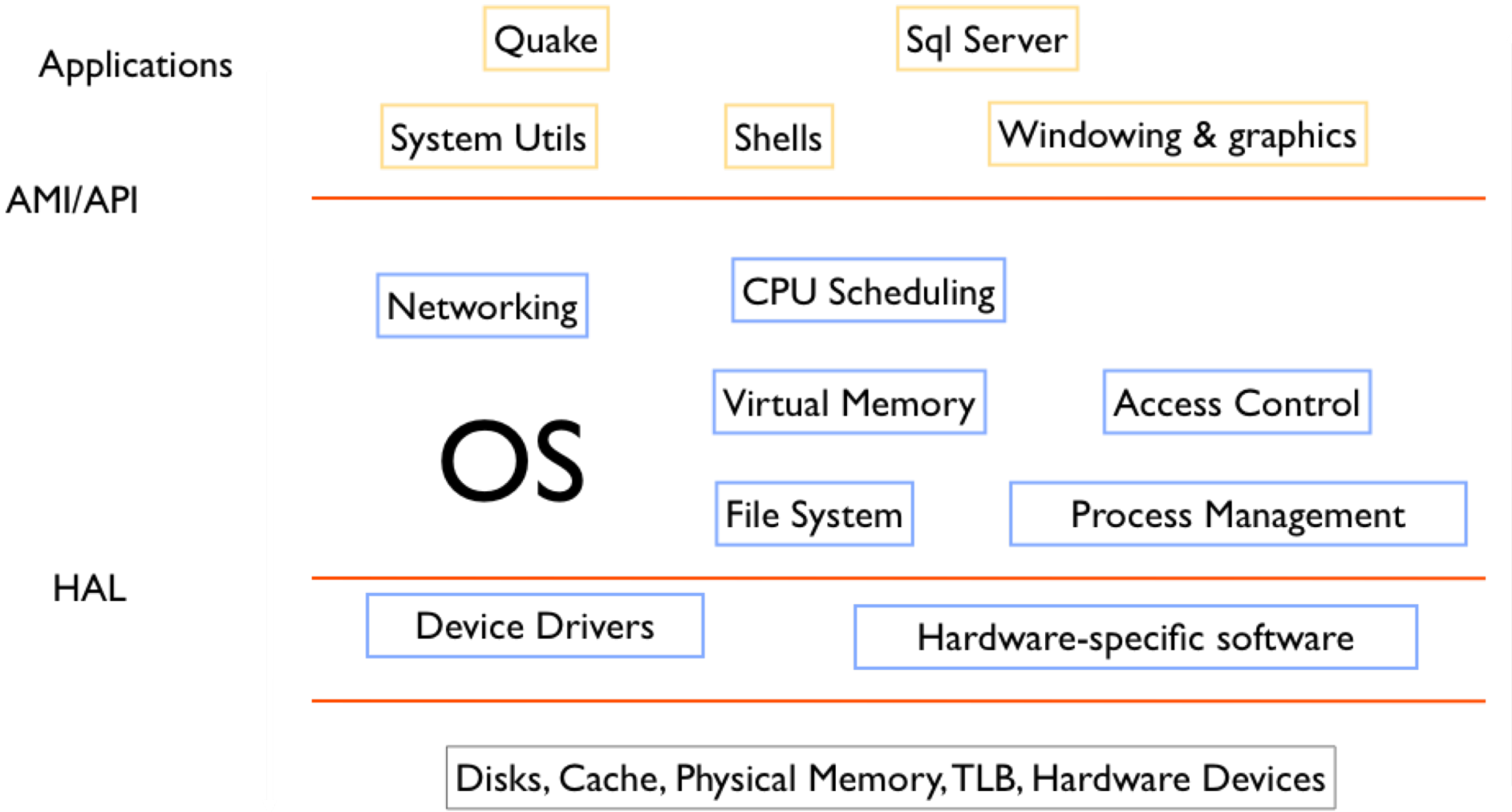
iClicker Question

The interrupt vector is used to determine the action taken by the OS when:

A. An exception occurs
B. An interrupt occurs
C. A system call is executed
D. All of the above
E. None of the above

The Shell

Logical OS Structure



So… The Unix Shell

- When you log in to a machine running Unix, the OS creates a shell process for you to use
- Almost every command you type into the shell will cause the shell to create a new process to execute that command
  - For example, if you type “calc”, the OS forks a new process and then execs calc
  - Other commands may be “built-in” to the shell---that is, they are implemented directly in the shell code base
- If you type an & after your command, Unix will run the process concurrently with your shell, otherwise your next shell command must wait until the first one completes.

More Shell

The shell also:

- Receives your <CTRL-C> command and send a SIGINT signal to the currently executing foreground process
- Receives your <CTRL-Z> command and send a SIGSTP signal to the currently executing foreground process
- Allows input-output redirections, pipes, and a lot of other stuff that we will see later

Process Creation

How to Create a Process

- One process can create other processes
  - The created processes are the *child* processes
  - The creator is the *parent* process
- In some systems, the parent defines (or donates) resources and privileges to its children
- The parent can either wait for the child to complete or continue in parallel

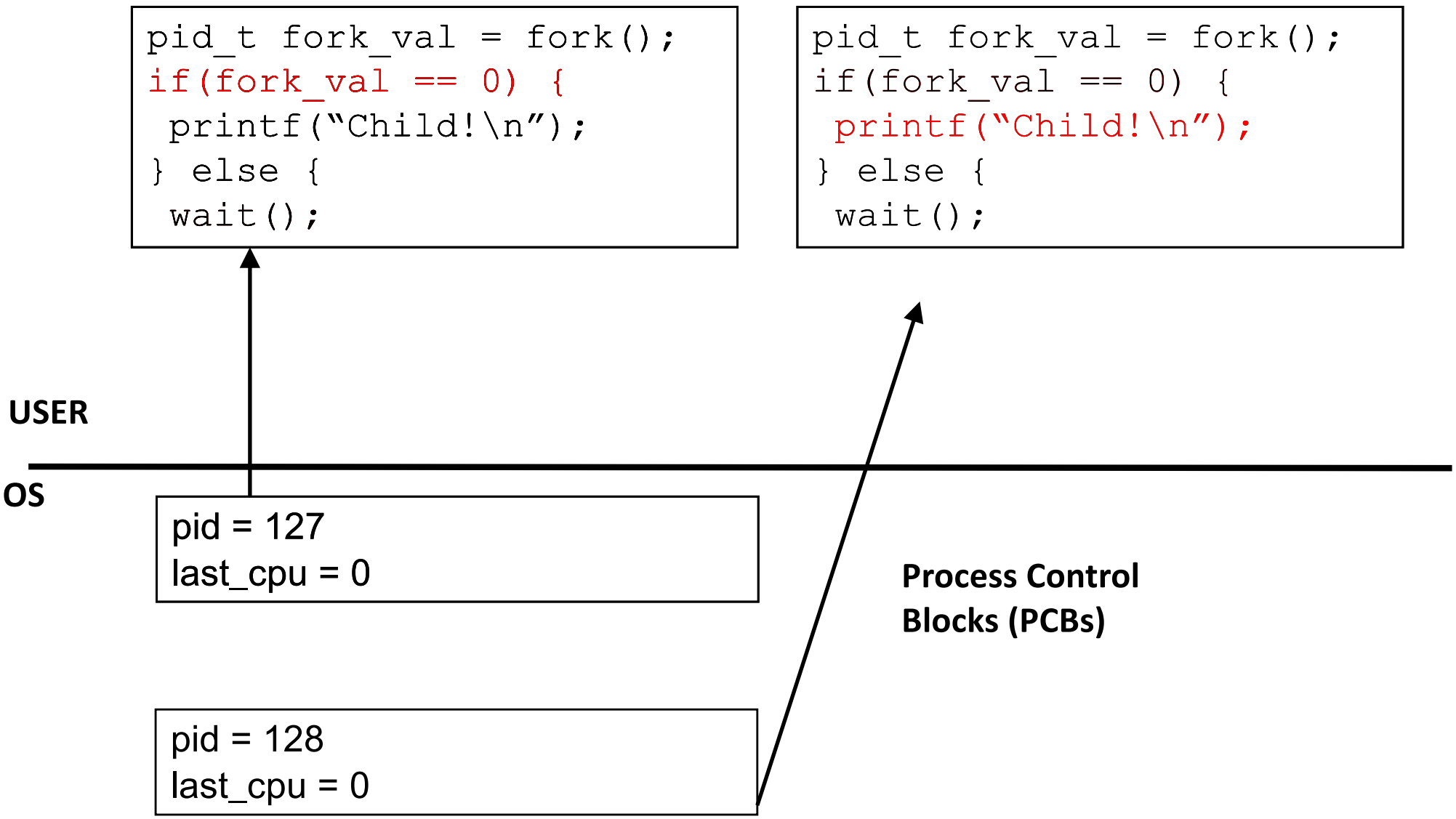
`fork()`

- In Unix, processes are created by `fork()`
- `fork()`: copies a process into an (identical) process
  - Copies variable values and program counter from parent to child
  - Returns twice: once to the parent and once to the child
  - Return value is different in the parent and child (This is the only difference!)
    - In parent, it is child process id
    - In child, it is 0
  - Both processes begin execution from the same point
    - Immediately following the call to `fork()`
  - Each process has its own memory and its own copy of each variable
    - Changes to variables in one process are not reflected in the other!

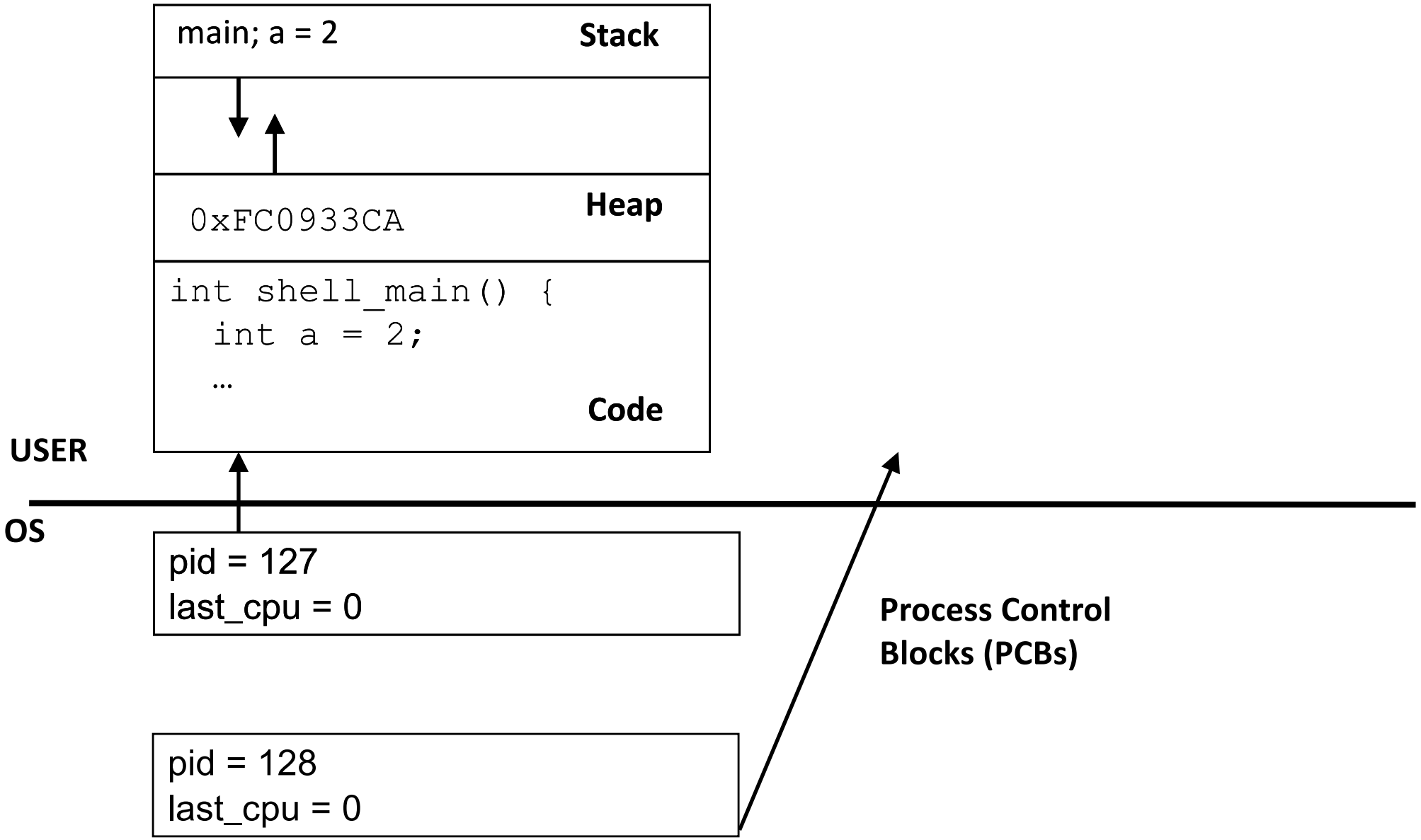
fork(): Pseudocode

```
pid_t fork_val = fork();                    //create a child
if((fork_val == FORKERR)                    //FORKERR is #define-d to -1
   printf("Fork failed!\n");
   return EXIT_FAILURE;
else if(fork_val == 0)                      //fork_val != child's PID
  printf("I am the child!");                //so child continues here
  return EXIT_SUCCESS;
else
  pid_t child_pid = fork_val                //parent continues here
  printf("I'm the parent.");
  int status;
  pid_t fin_pid = wait(&status);            //wait for child to finish
```

Example: `fork()`



Example: `fork()`



Ummm, okay, but….

Why do I want two copies of the same process?

What if I want to start a different process?

How do I do that?

`exec()`

- Overlays a process with a new program
  - PID does not change
  - Arguments to new program may be specified
  - Code, stack, and heap are overwritten
    - Sometimes memory-mapped files are preserved
- Child processes often call `exec()` to start a new and different program
  - New program will begin at `main()`
- If call is successful, it is the *same* process, but it is running a *different* program!

`fork()` and `exec()`: Pseudocode

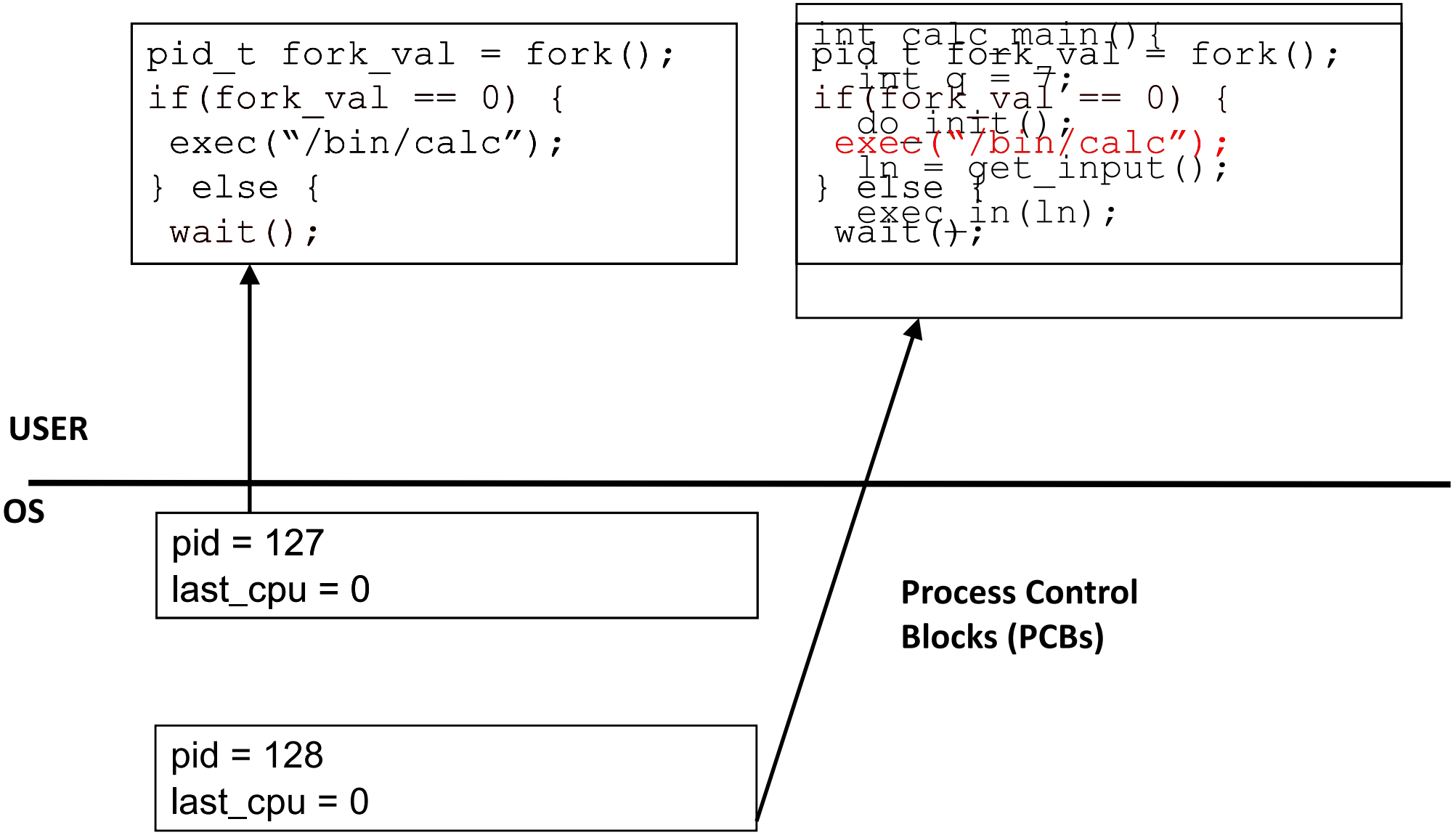
```
pid_t fork_val = fork();                          //create a child
if(fork_val  == FORKERR)
   printf("Fork failed!\n");
   return EXIT_FAILURE;
else if(fork_val == 0)                            //child continues here
  exec_status = exec("calc", argc, argv0, argv1, …);
  printf("Why would I execute?");                 //should NOT execute
  return EXIT_FAILURE;
else
  pid_t child_pid = fork_val                      //parent continues here
  printf("I'm the parent.");
  int status;
  pid_t fin_pid = wait(&status);                  //wait for child to finish
```

iClicker Question

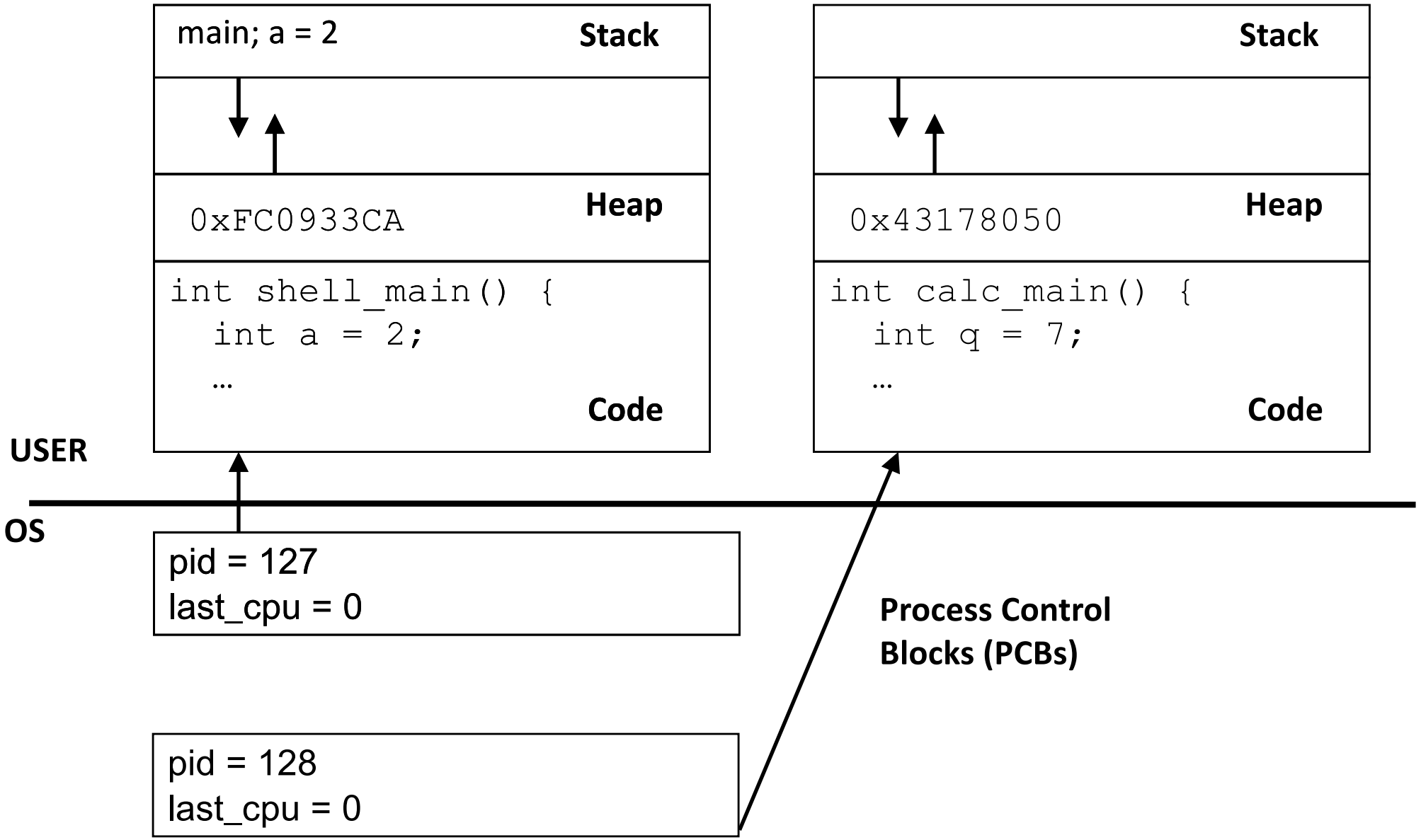
What creates a process?

A. `fork()`
B. `exec()`
C. both

Example: `fork()` and `exec()`



Example: `fork()` and `exec()`



Process Control

`fork()` and `exec()`: Pseudocode

```
pid_t fork_val = fork();                          //create a child
if(fork_val  == FORKERR)
   printf("Fork failed!\n");
   return EXIT_FAILURE;
else if(fork_val == 0)                            //child continues here
  exec_status = exec("calc", argc, argv0, argv1, …);
  printf("Why would I execute?");                 //should NOT execute
  return EXIT_FAILURE;
else
  pid_t child_pid = fork_val                      //parent continues here
  printf("I'm the parent.");
  int status;
  pid_t fin_pid = wait(&status);                  //wait for child to finish
```

The `wait()` System Call

`wait()` system call causes the parent process to wait for the child process to terminate

- Allows parent process to get return value from the child
- It blocks the parent until the child completes
- When a child calls `exit()`, the OS unblocks the parent and returns the value passed by `exit()` as a result of the wait call (along with the pid of the child)
- If there are no children alive, `wait()` returns immediately
- Also, if there are zombies waiting for their parents, wait() returns one of the values immediately (and deallocates the zombie)

Zombie Processes

- Process has terminated
- Parent process has not collected its status
- Dead, but not gone…

Orphaned Processes

- Parent terminates before the child:
  - In some instances, the child becomes an *orphan process*
    - In UNIX, parent automatically becomes the init process
  - In other instances, all children are killed (depending on the shell)
    - Bash kills all child processes when it receives a SIGHUP
- Child can orphan itself to keep running in the background
  - nohup command (also prevents it from being killed when SIGHUP is sent)

Terminating a process: `exit()`

- After the program finishes execution, it calls `exit()`
- This system call:
  - Takes the result (return value) of the process as an argument
  - Closes all open files, connections, etc.
  - Deallocates memory
  - Deallocates most of the OS structures supporting the process
  - *Checks if parent is alive:*
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the *zombie/defunct* state
    - If not, it deallocates all data structures, the process is dead
  - Cleans up all waiting zombies
- Process termination is the ultimate garbage collection (resource reclamation).

Terminating a process: `kill()`

- A parent can terminate a child using `kill()`
  - `kill()` is also used for interprocess communication
- This system call:
  - Sends a *signal* to a specified process (identified by its PID)
    - SIGHUP, SIGKILL, SIGCHLD, SIGUSR1, SIGUSR2
  - The receiving process can define *signal handlers* to handle signals in a particular way
  - If a handler for that signal does not exist, the default action is taken

Practical Usage: `ps` and `kill`

If you have a process running that you need to kill:

- From the command line, type:

  ```
  ps –au <login_name>
  ```

- Find the process you would like to terminate (the name is in the CMD column) and then determine its PID. You can do this visually or use `grep`:

  ```
  ps –au <login_name> | grep <program_name>
  ```

- From the command line, type:

  ```
  kill -9 <PID>
  ```

Other Process Control

- Priority manipulation:
  - `nice()`, which specifies base process priority (initial priority)
  - In UNIX, process priority decays as the process consumes CPU
- Debugging support:
  - `ptrace()` allows a process to be put under control of another process
  - The other process can set breakpoints, examine registers, etc.
- Alarms and time:
  - Sleep puts a process on a timer queue waiting for some number of seconds, supporting an alarm functionality