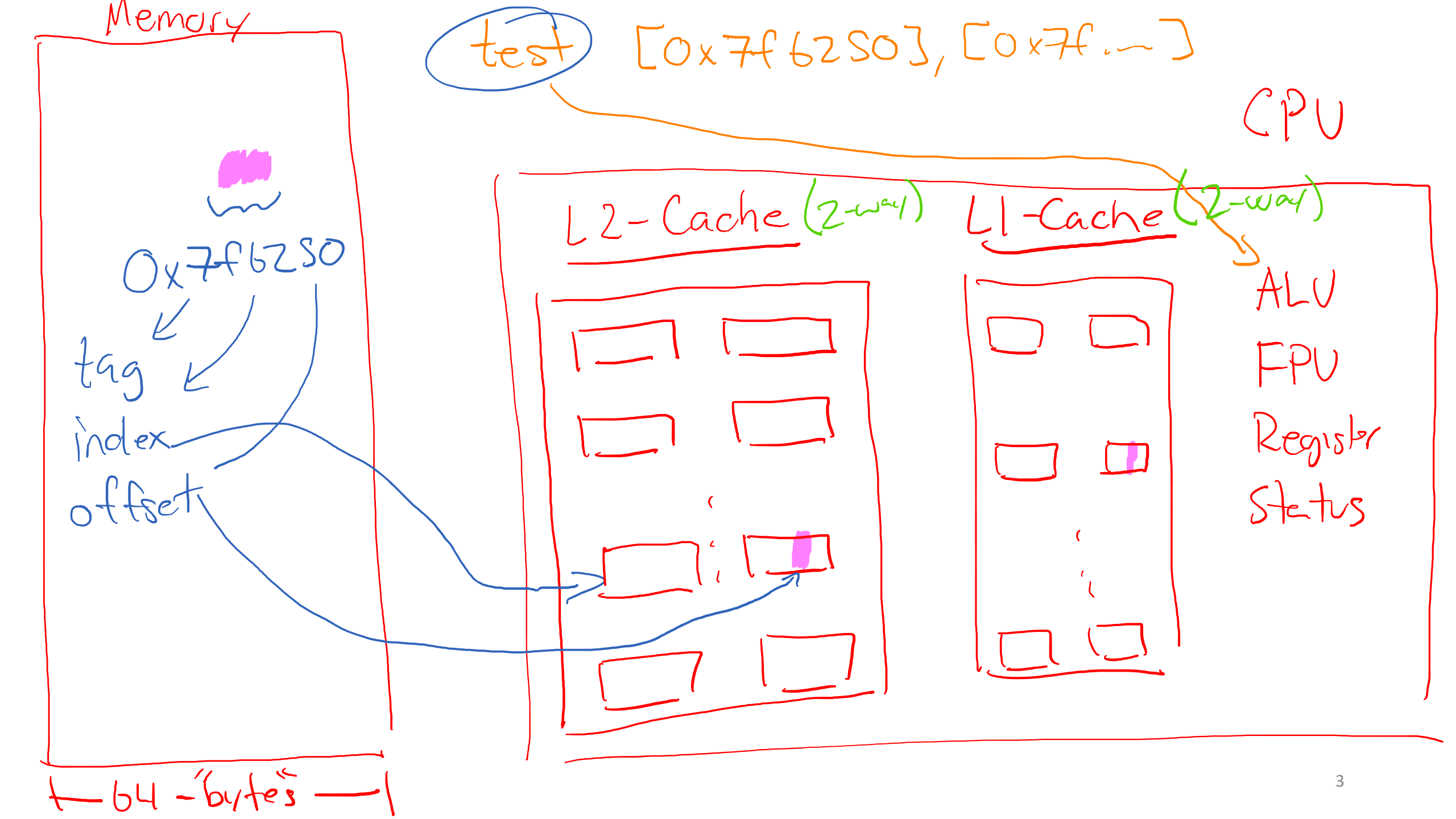
Operating System Processes

Drawing: Cache

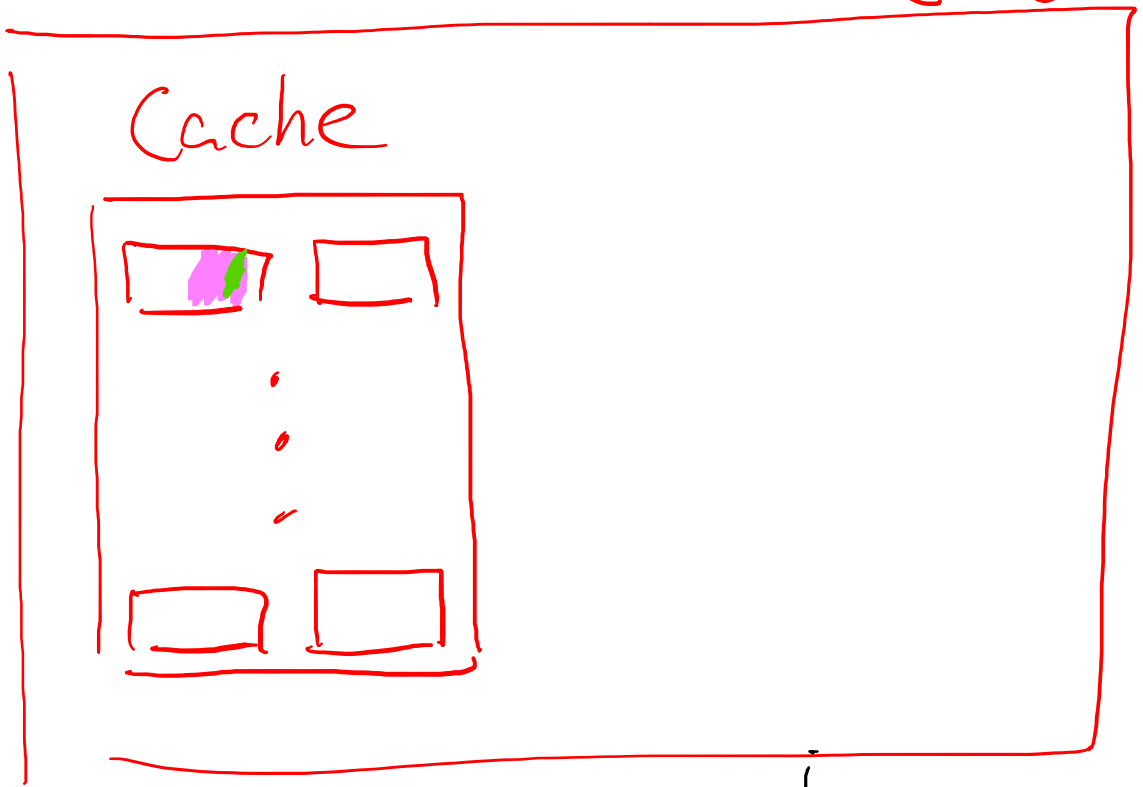
- Take three minutes to draw “cache”

- Some reminders
  - Multiple levels
  - Different sizes and speeds per level
  - Tag/Index/Offset
  - Lines
  - Sets
  - Data blocks
  - Valid bits

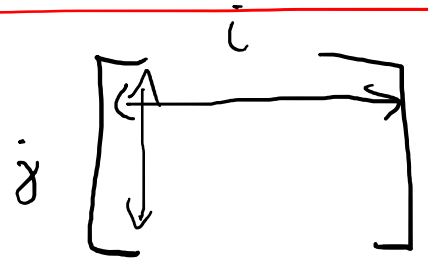

Memory
test
[0x7f6250], [0x7f...]
CPU
L2-Cache (2-way)
L1-Cache (2-way)
ALU
FPU
Register
Status
0x7f6250
tag
index
offset
64 "bytes"

C = malloc(n · n · sizeof(float))

CPU



C[i + j · n]



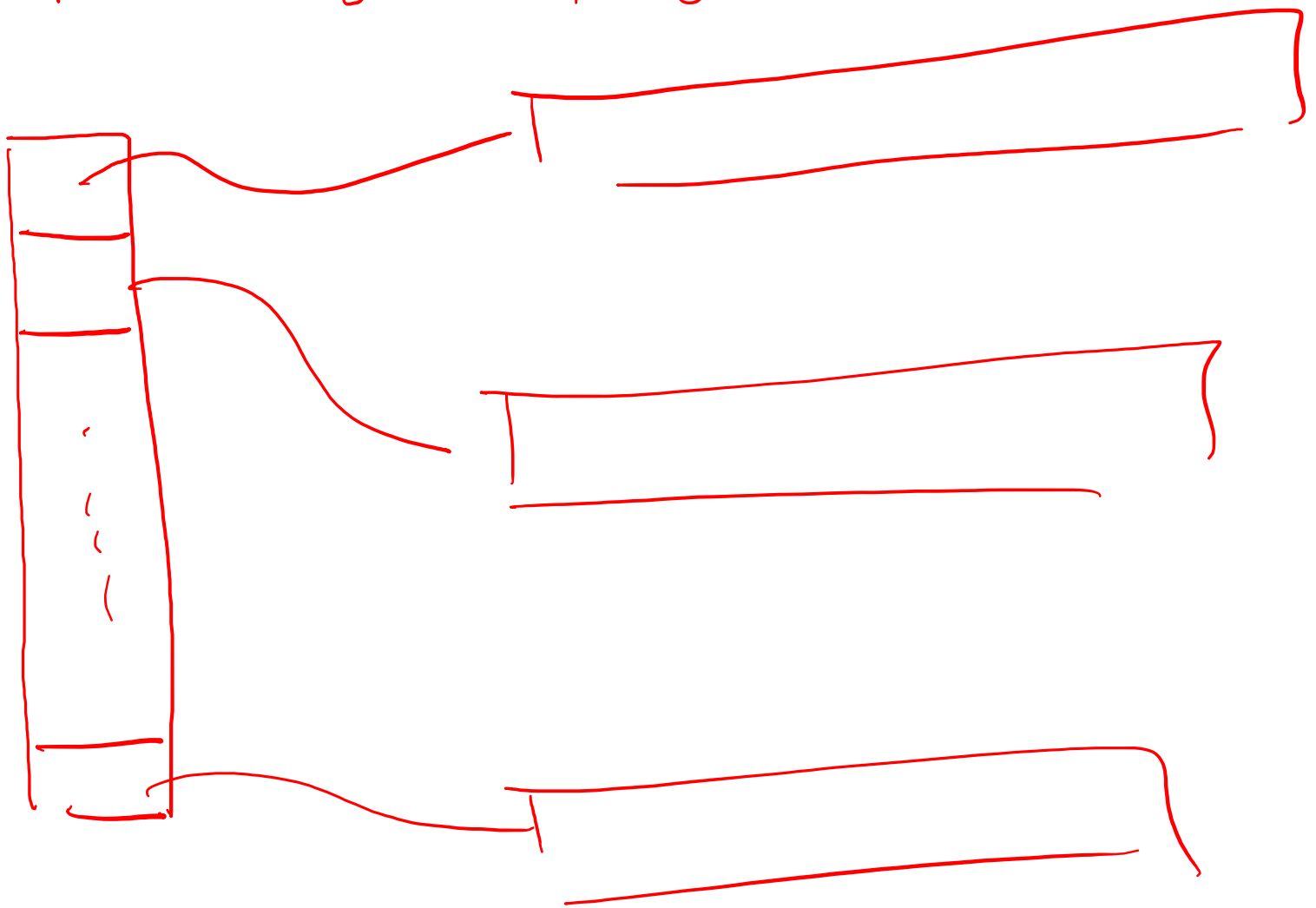
C = malloc(n * sizeof (float *))

for (int j = 0 ; j < n ; j++) C[j] = malloc(n

*

sizeof(

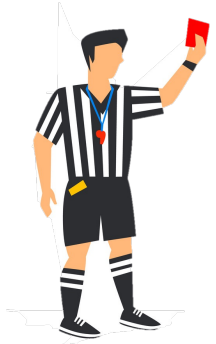
float))



(C[j])[i]

Introduction to Operating Systems

- An operating system (OS) manages a computer's resources
  - Examples: OSX, Windows, Ubuntu, iOS, Android, Chrome OS
- Core OS functionality is implemented by the OS kernel



- resource allocation
- isolation
- communication
- access control

- multiprocessing
- virtual memory
- reliable networking
- virtual machines

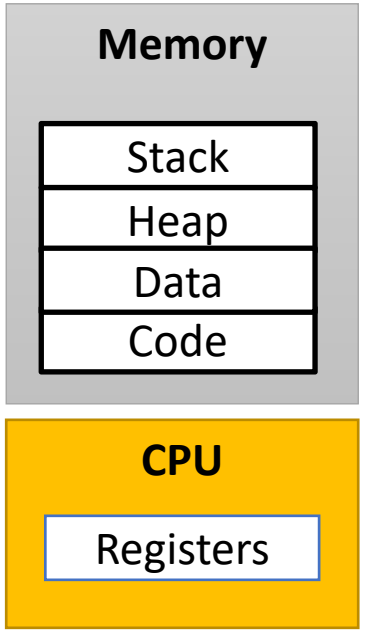
- user interface
- file I/O
- device management
- process control

Operating System Goals

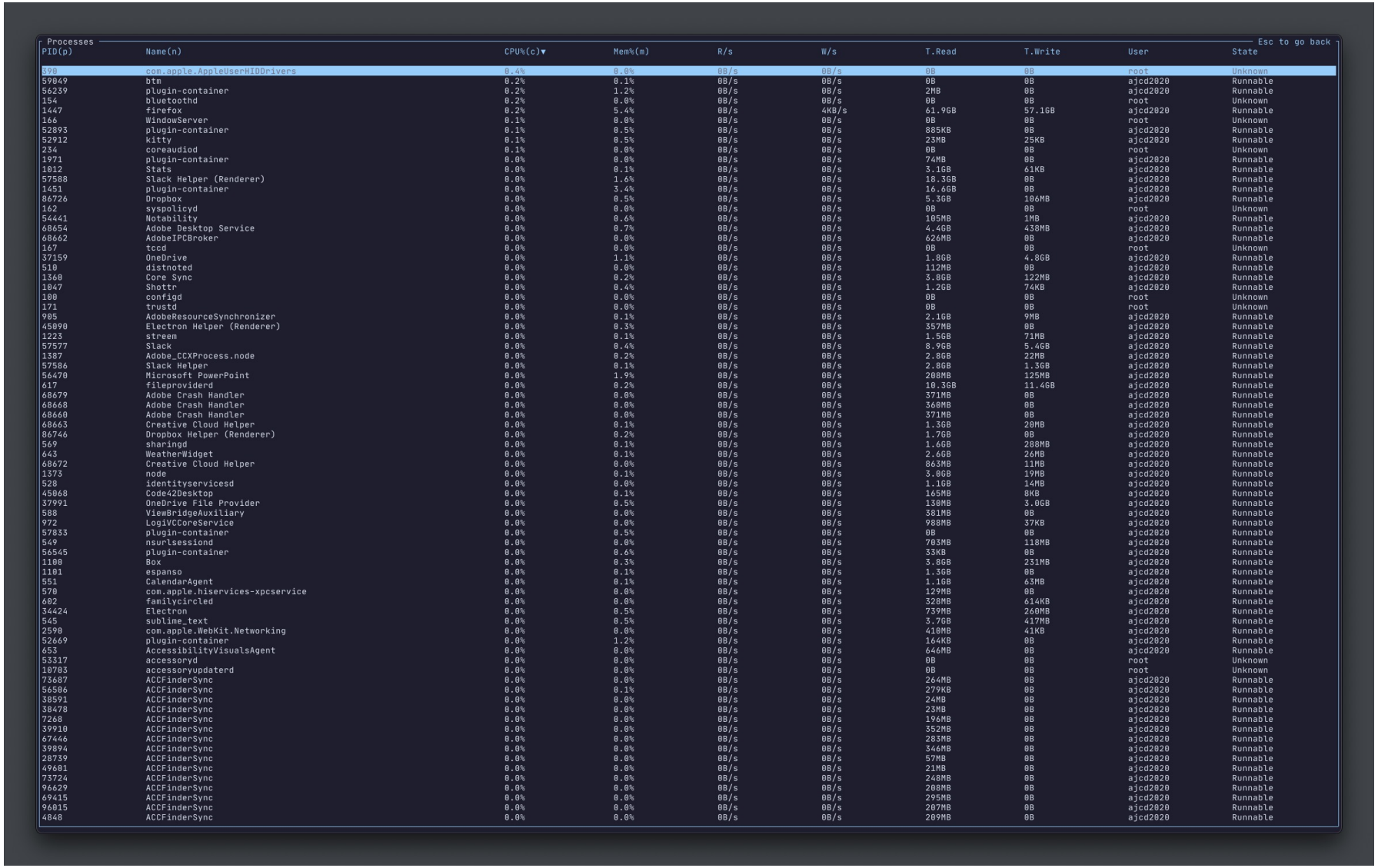
- Reliability: they OS should do what you want
- Availability: the OS should respond to user input
- Security: the OS should not be (easily) corrupted by an attacker
- Portability: the OS should be easy to move to new hardware platforms
- Performance: the OS should impose minimal overhead and be responsive

Processes

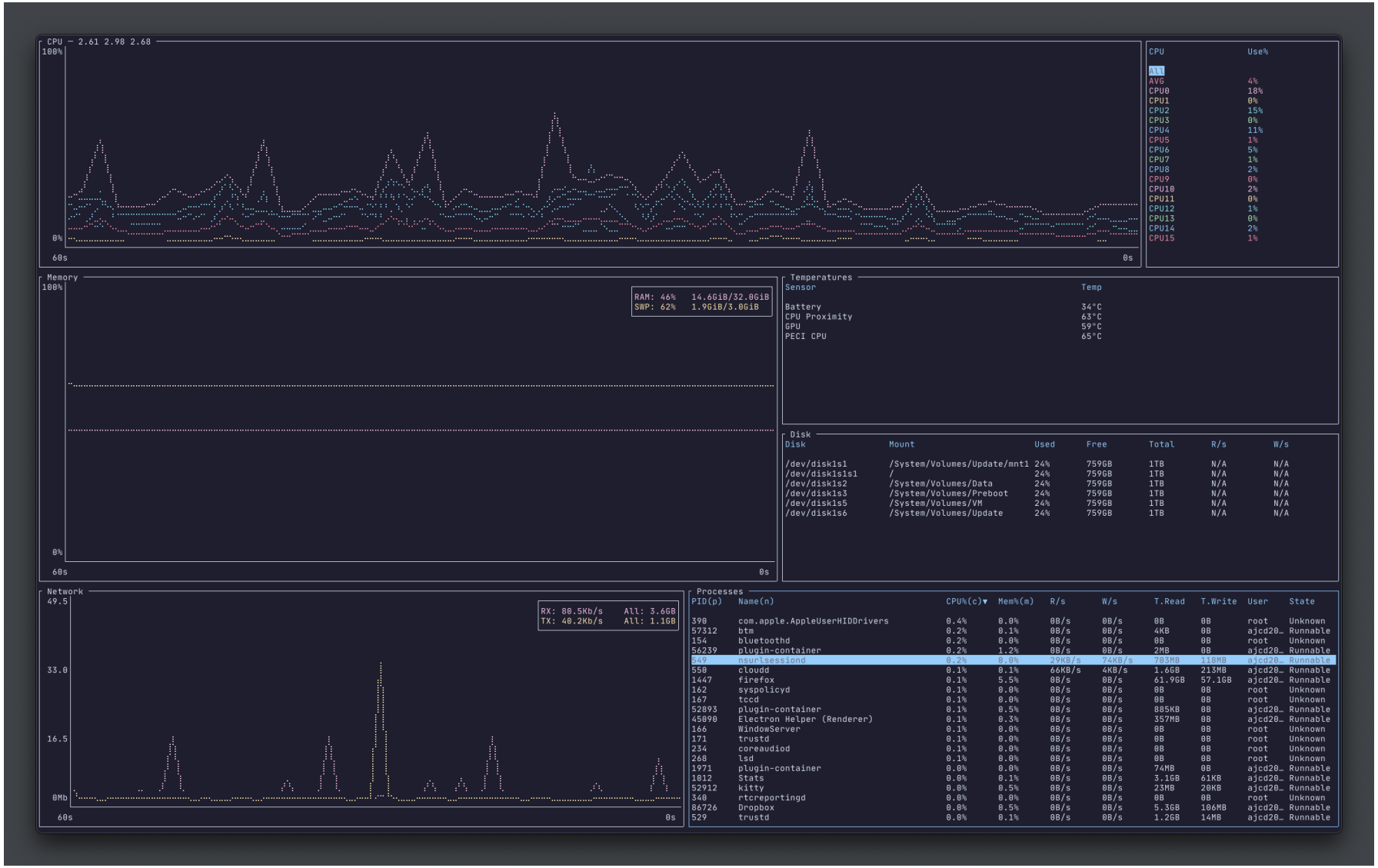
- A program (executable, binary, etc.) is a file containing code + data
  - For example, in the ELF format on Linux
- A process is an instance of a running program
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”
- Why would we ever have two instances of a single program?



Multiprocessing (running a monitor)



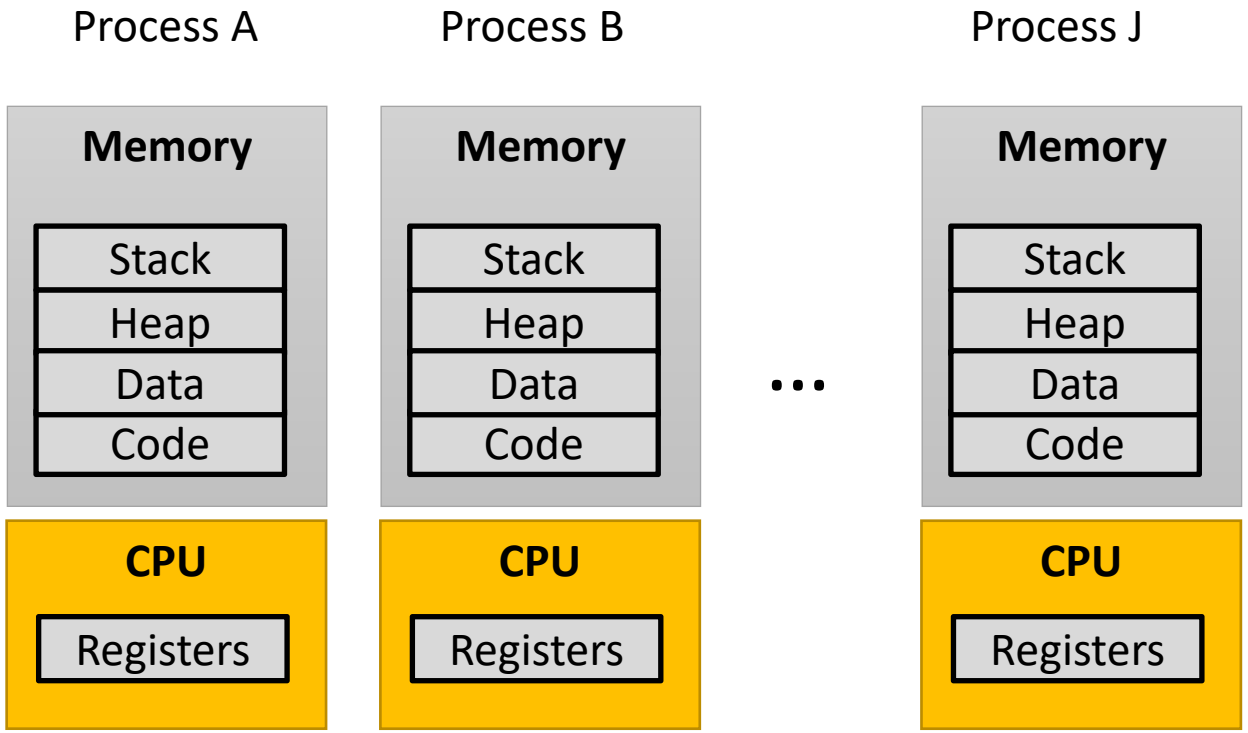
Multiprocessing (running a monitor)



Multiprocessing: The Illusion

Each process has its own:

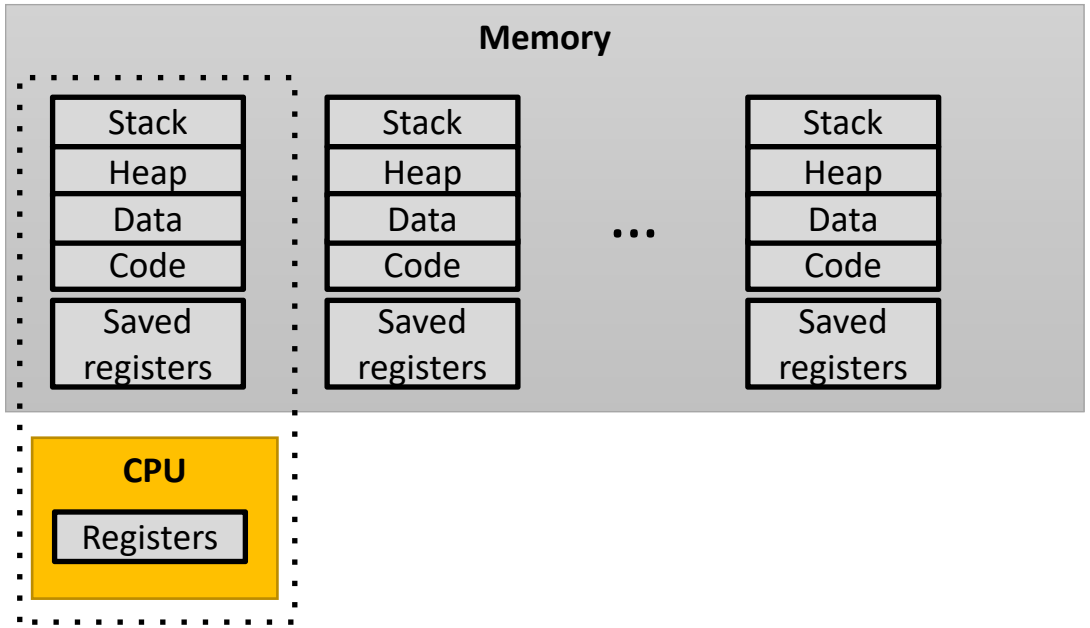
- Logical control flow
  - Each program seems to have exclusive use of the CPU
  - Provided by kernel mechanism called context switching
- Private address space
  - Each program seems to have exclusive use of main memory
  - Provided by kernel mechanism called virtual memory



Multiprocessing: The (Traditional) Reality

A single processor (CPU) executes multiple processes concurrently

- Process executions interleaved (multitasking)
- Register values for nonexecuting processes saved in memory
- Address spaces managed by virtual memory system



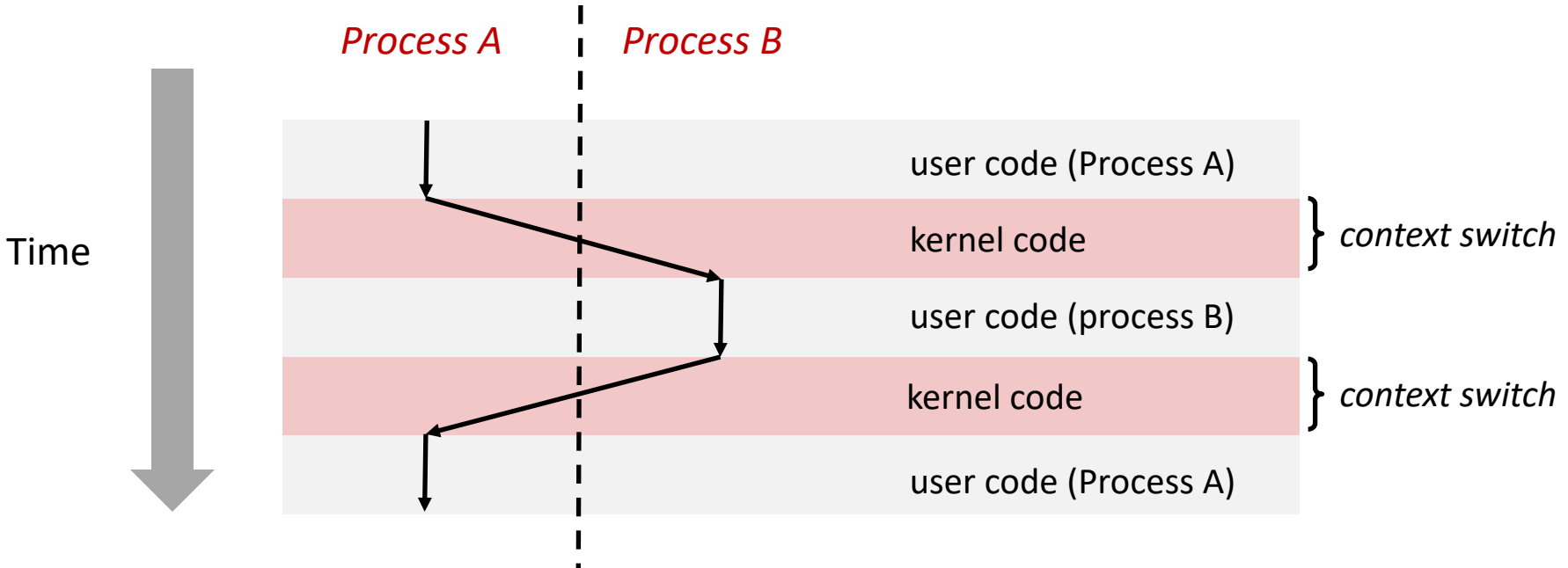
Process Control Block (PCB)

To switch from one process to another (a “context switch”), the OS maintains a PCB for each process containing:

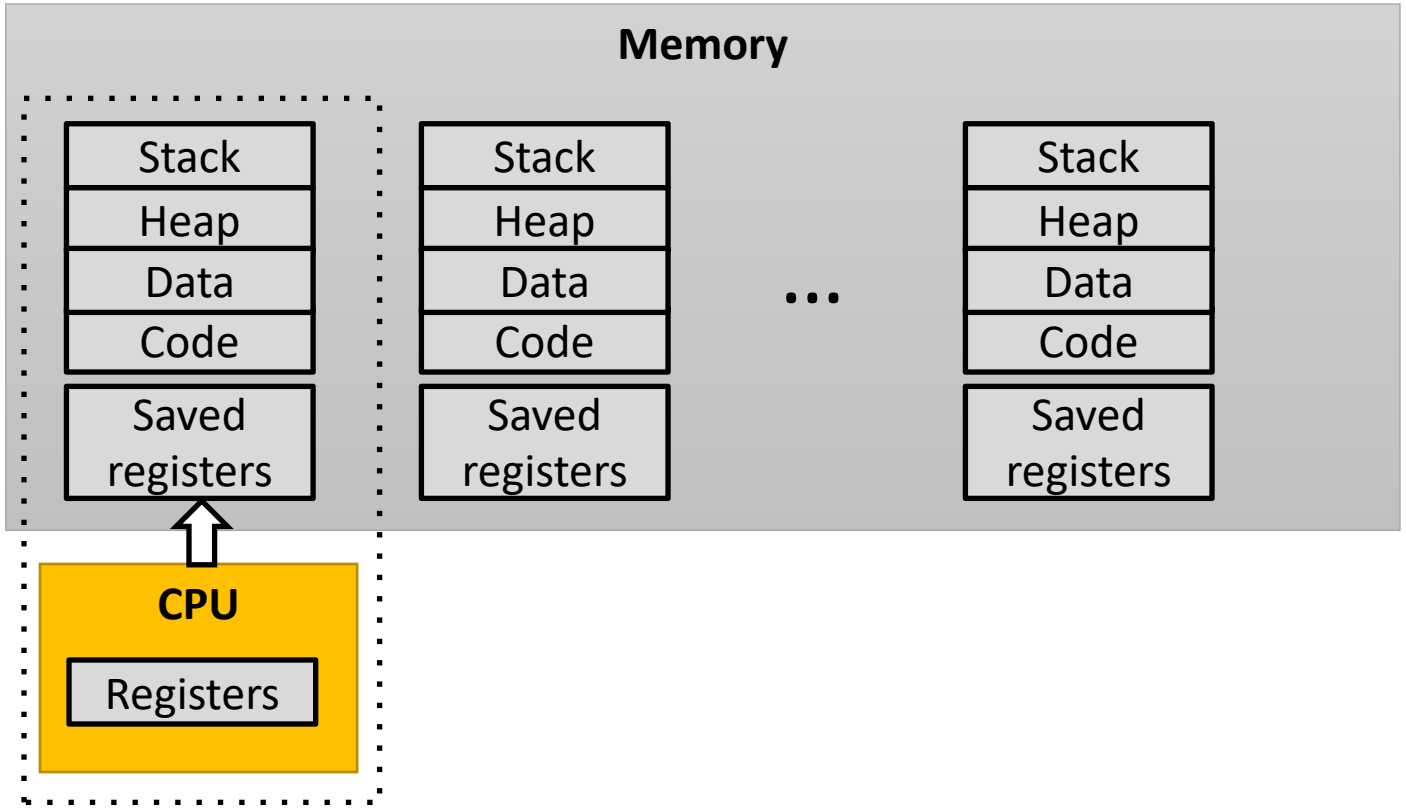
- process table (id, user, privilege level, arguments, status, etc.)
- location of executable in storage (e.g., SSD)
- file table (a list of all open “files”)
- register values (general-purpose registers, float registers, pc, eflags…)
- memory state (stack, heap, data, etc.)
- scheduling information (number of cycles, last run time, etc.)
- ... and more!

Context Switching

- Processes are managed by the (memory-resident) kernel code
  - Important: the kernel code is not a separate process, but rather code and data structures that the OS uses to manage all processes
- Control flow passes from one process to another via a ***context switch***

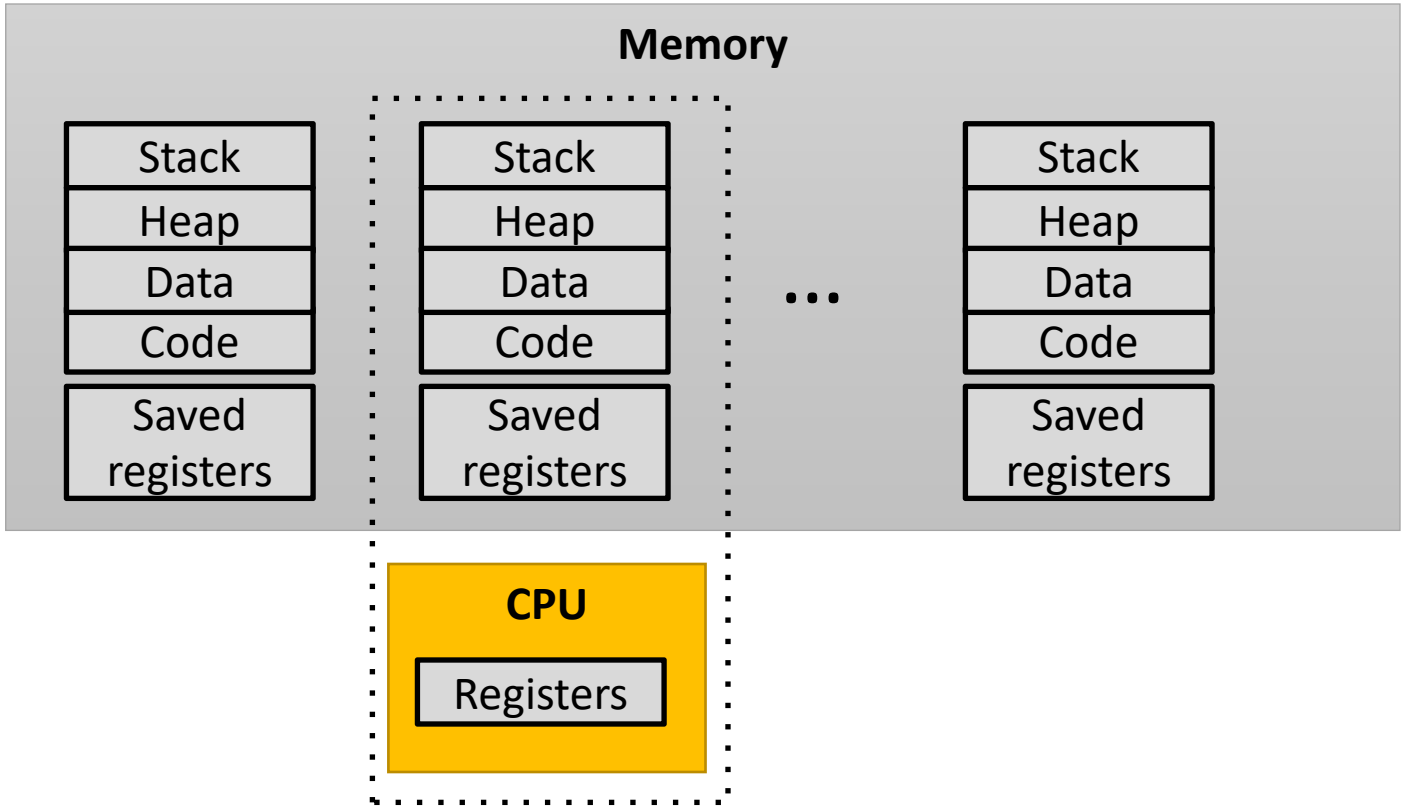


Multiprocessing: The (Traditional) Reality



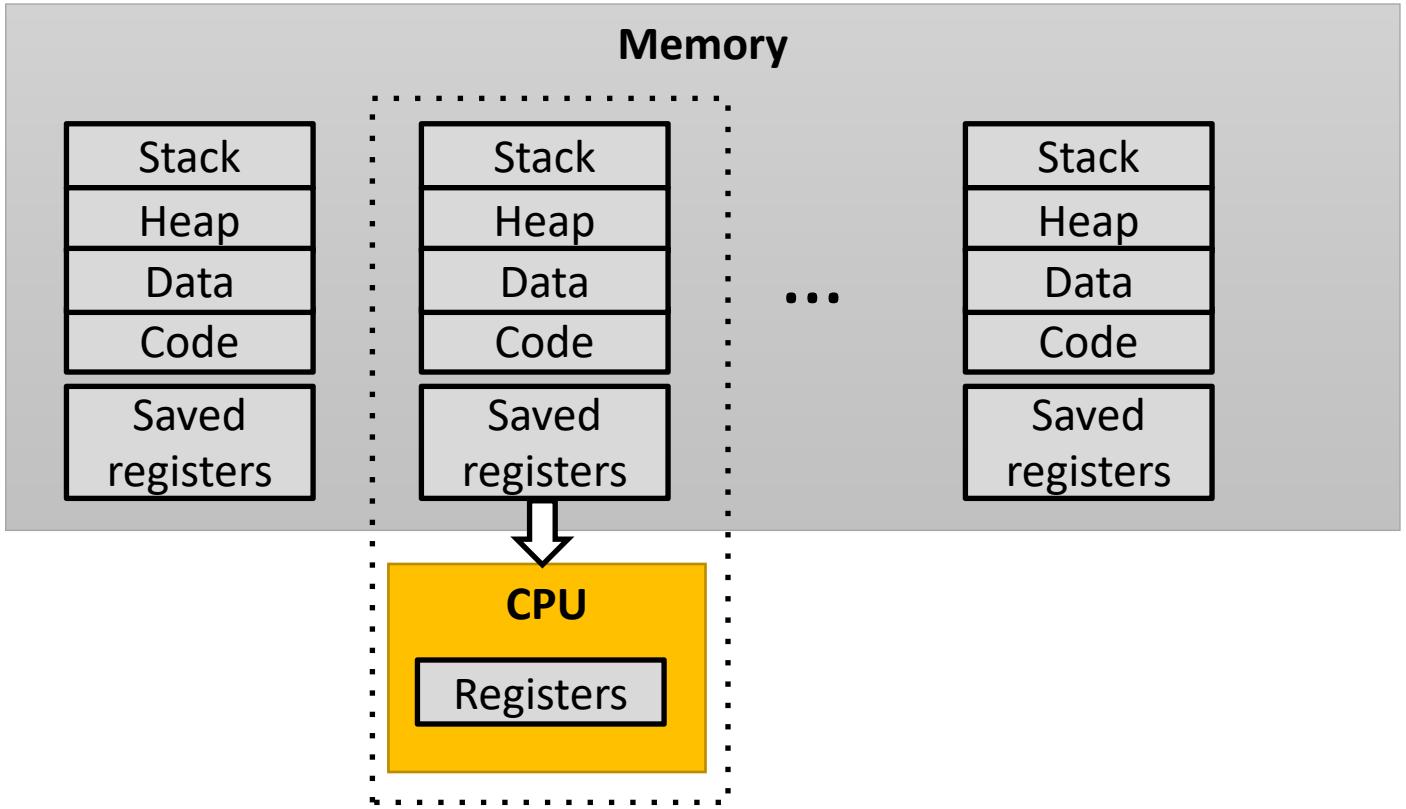
1. Save current registers to memory (in PCB)

Multiprocessing: The (Traditional) Reality



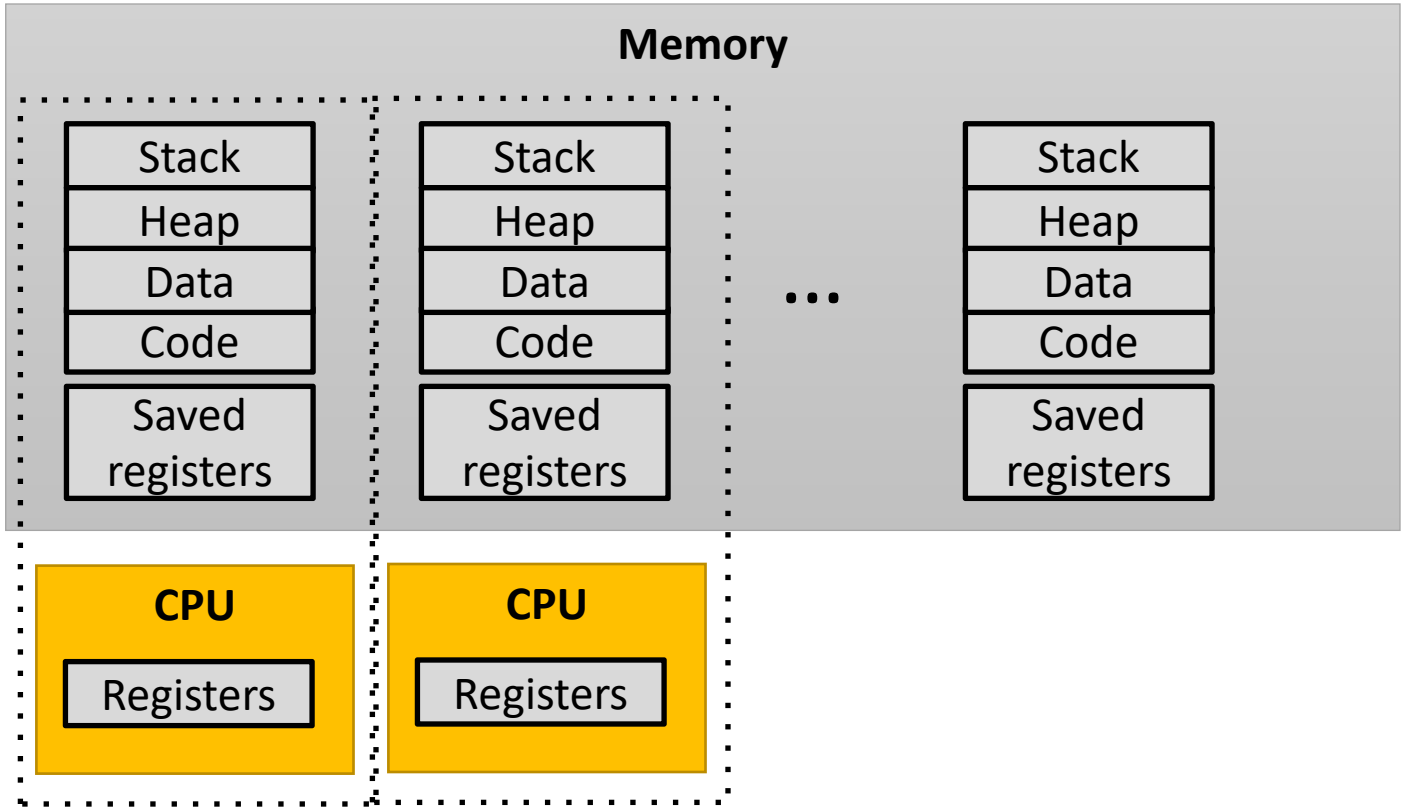
1. Save current registers to memory (in PCB)
2. Schedule next process for execution

Multiprocessing: The (Traditional) Reality



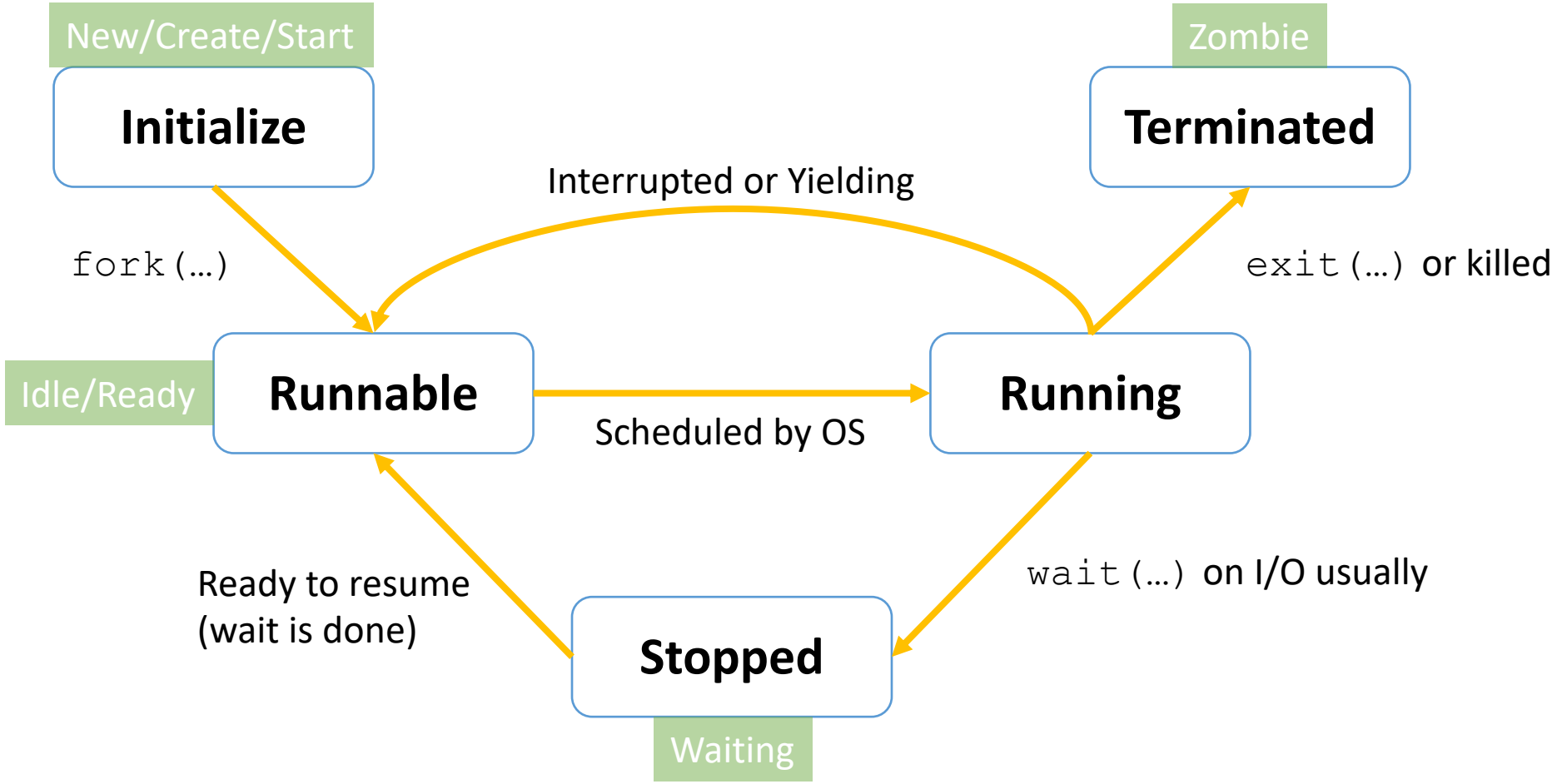
1. Save current registers to memory (in PCB)
2. Schedule next process for execution
3. Load saved registers and switch address space

Multiprocessing: The (Modern) Reality

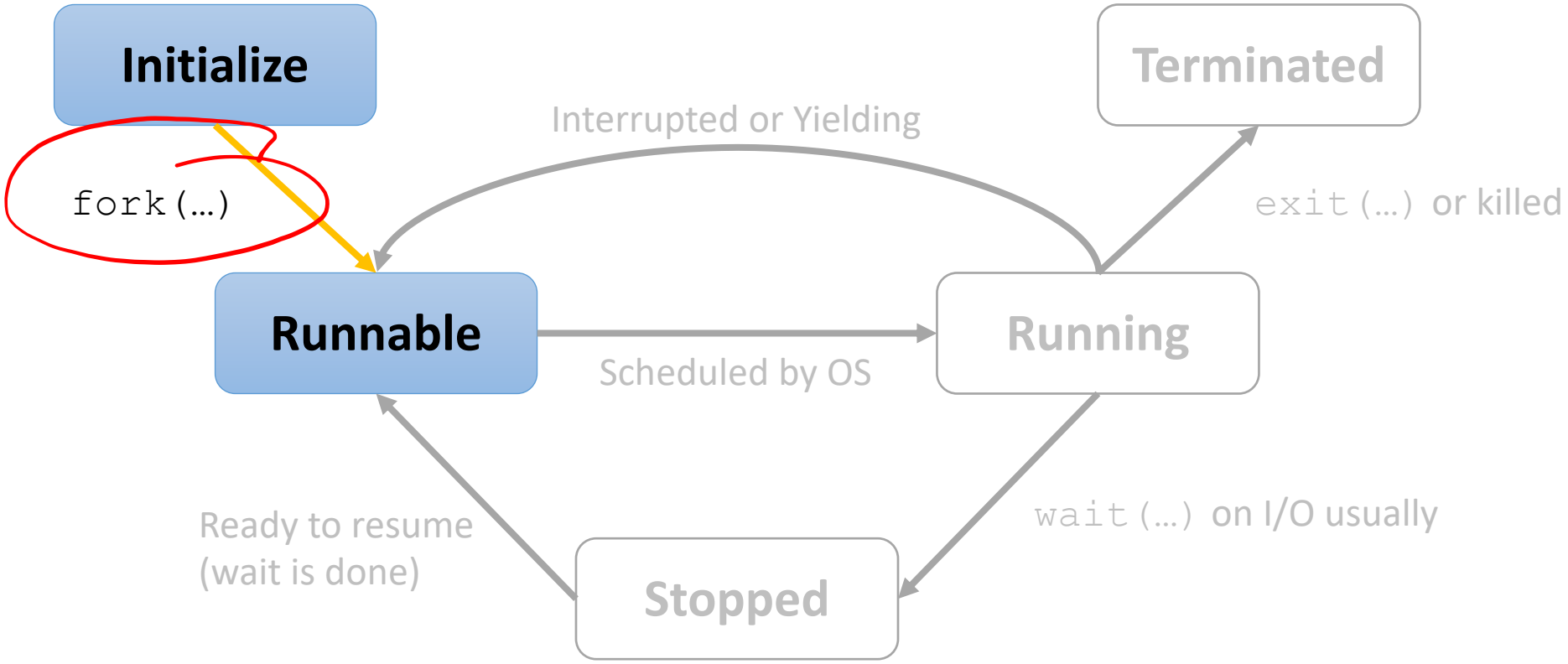


- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

Process Life Cycle (Linux)



Process Life Cycle (Linux)



Creating Processes

- Parent process creates a new running child process by calling `fork`

- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child gets an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

- `fork` is interesting (and often confusing) because it is called *once* but returns *twice*

`fork` Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

fork.c

What are the possible outputs?

What if we want to fork a new/separate program?

- Call once, return twice
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files
  - `stdout` is the same in both parent and child

Modeling `fork` with Process Graphs

- A process graph is a useful tool for capturing the partial ordering of statements in a concurrent program:
  - Each vertex is the execution of a statement
  - a -> b means a happens before b
  - Edges can be labeled with current value of variables
  - printf vertices can be labeled with output
  - Each graph begins with a vertex with no inedges

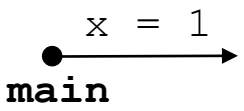
- Any topological sort of the graph corresponds to a feasible total ordering.
  - Total ordering of vertices where all edges point from left to right

Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

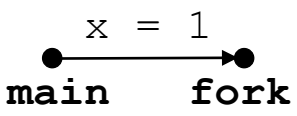
Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```



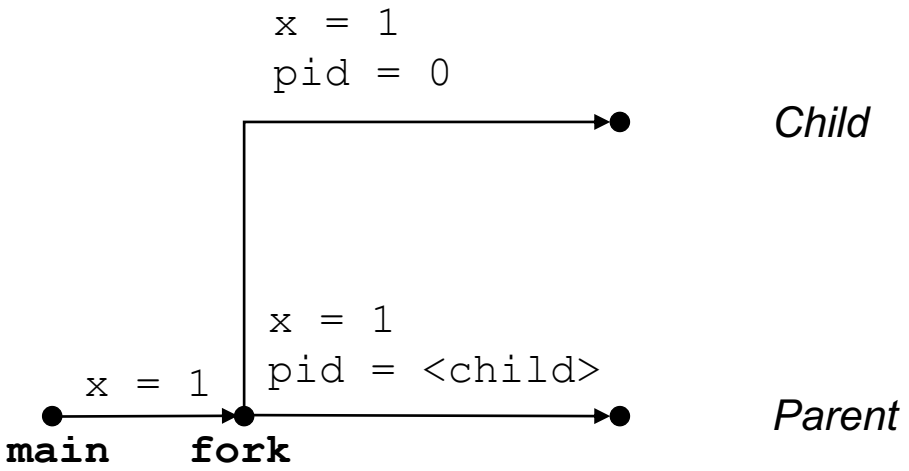
Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```



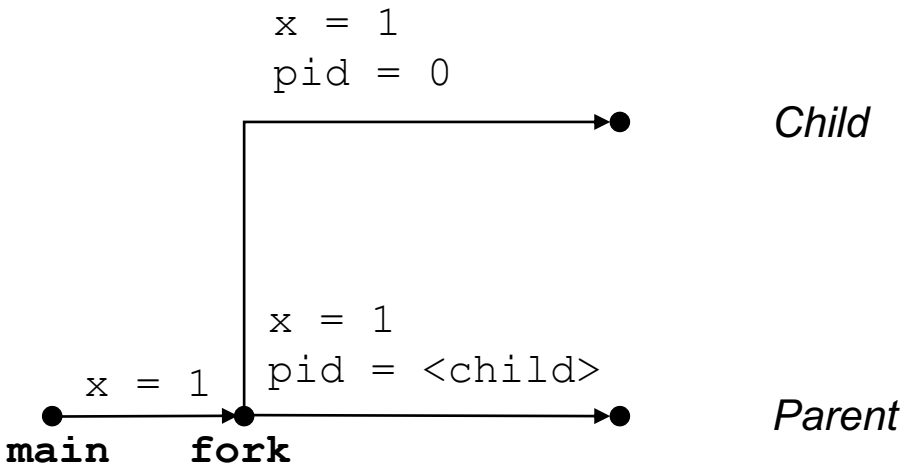
Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```



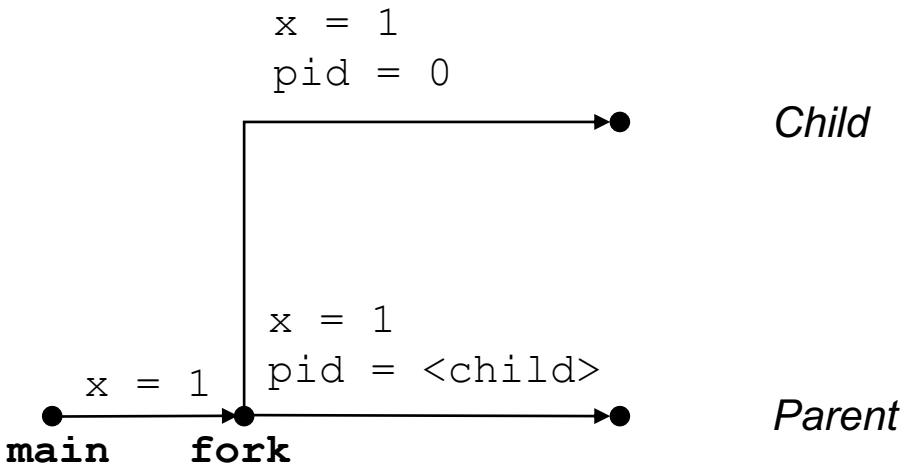
Process Graph Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```



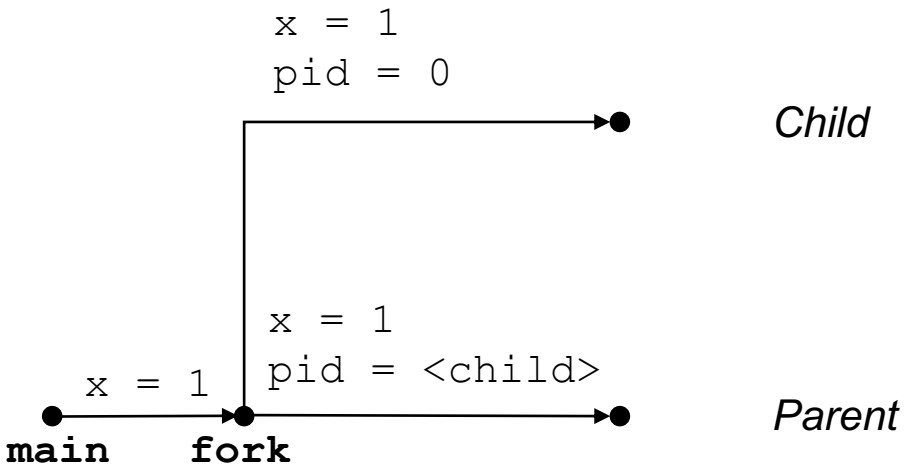
Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```



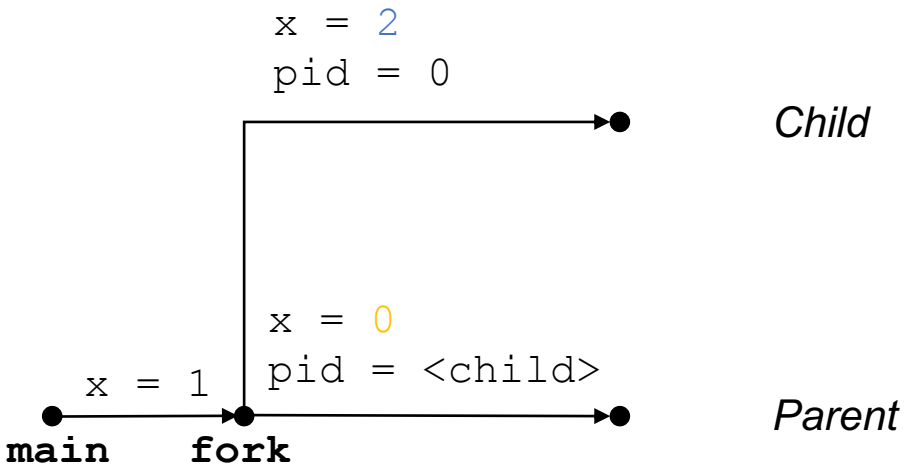
Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```



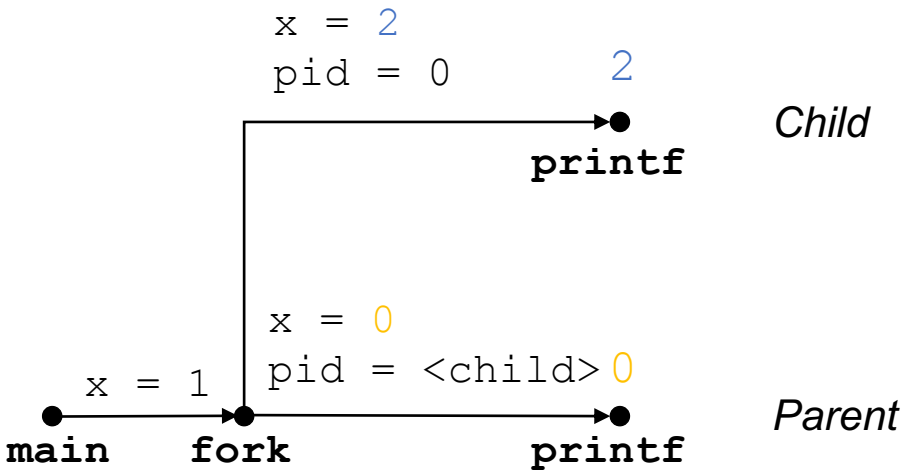
Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```



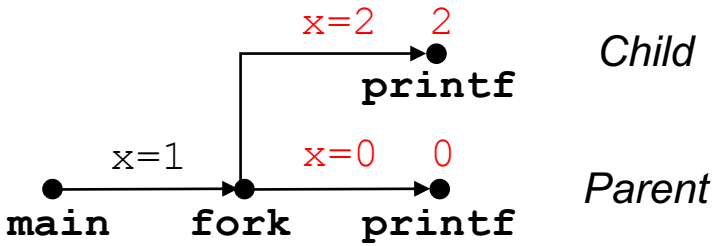
Process Graph Example

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

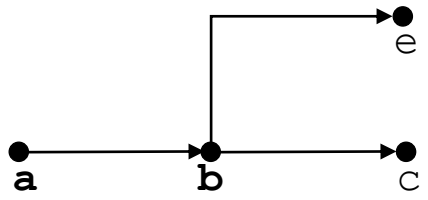


Interpreting Process Graphs

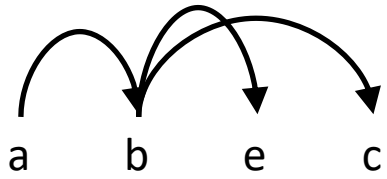
- Original graph:



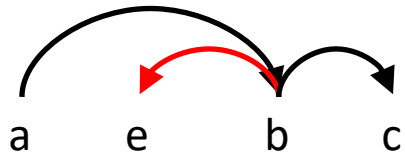
- Relabeled graph:



Feasible total ordering:

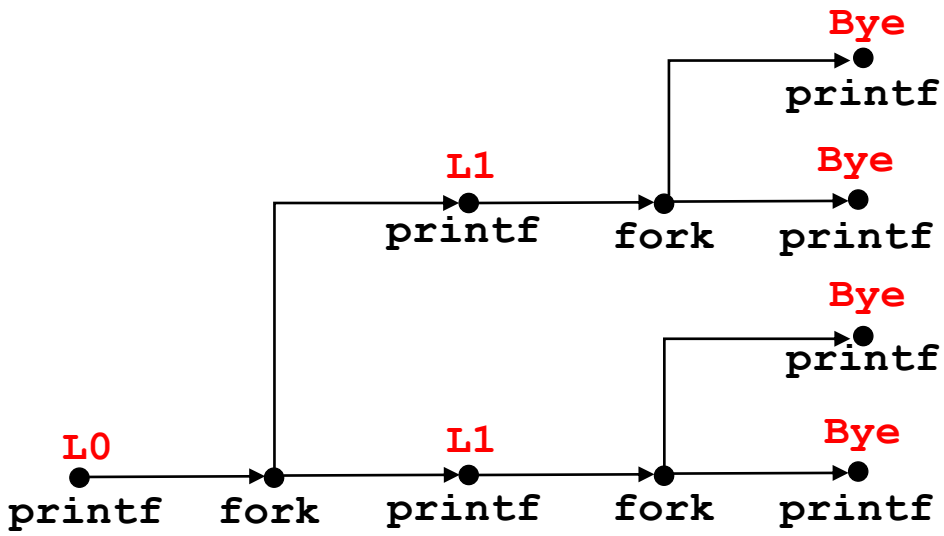


Infeasible total ordering:



Practice with `fork`

```
void fork1()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Which of these outputs are feasible?

L0	L0
L1	Bye
Bye	L1
Bye	Bye
L1	L1
Bye	Bye
Bye	Bye

More practice with `fork`

- For each of the following programs, draw the process graph and then determine which of the possible outputs are feasible

```
void fork2(){
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

L0
L1
Bye
Bye
L2
Bye

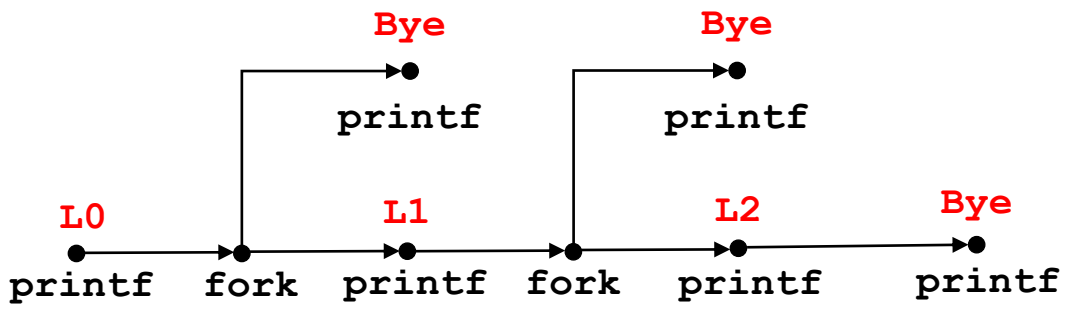
L0
Bye
L1
Bye
Bye
L2

```
void fork3(){
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

L0
Bye
L1
L2
Bye
Bye

L0
Bye
L1
Bye
Bye
L2

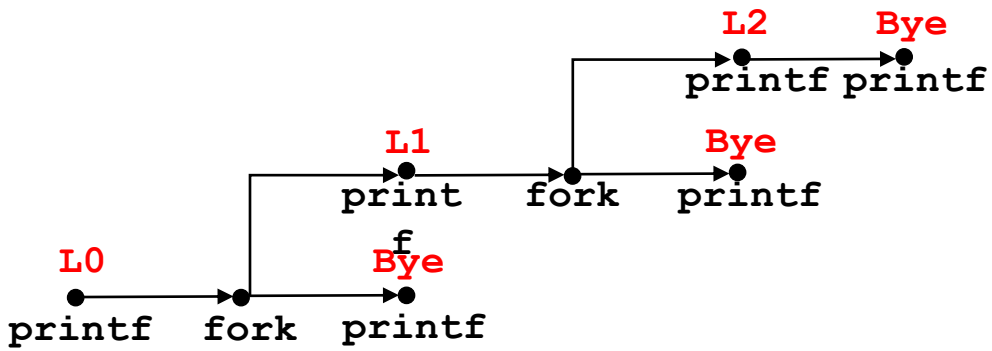
```c
void fork2()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```



Which of these outputs are feasible?

L0	L0
L1	Bye
Bye	L1
Bye	Bye
L2	Bye
Bye	L2

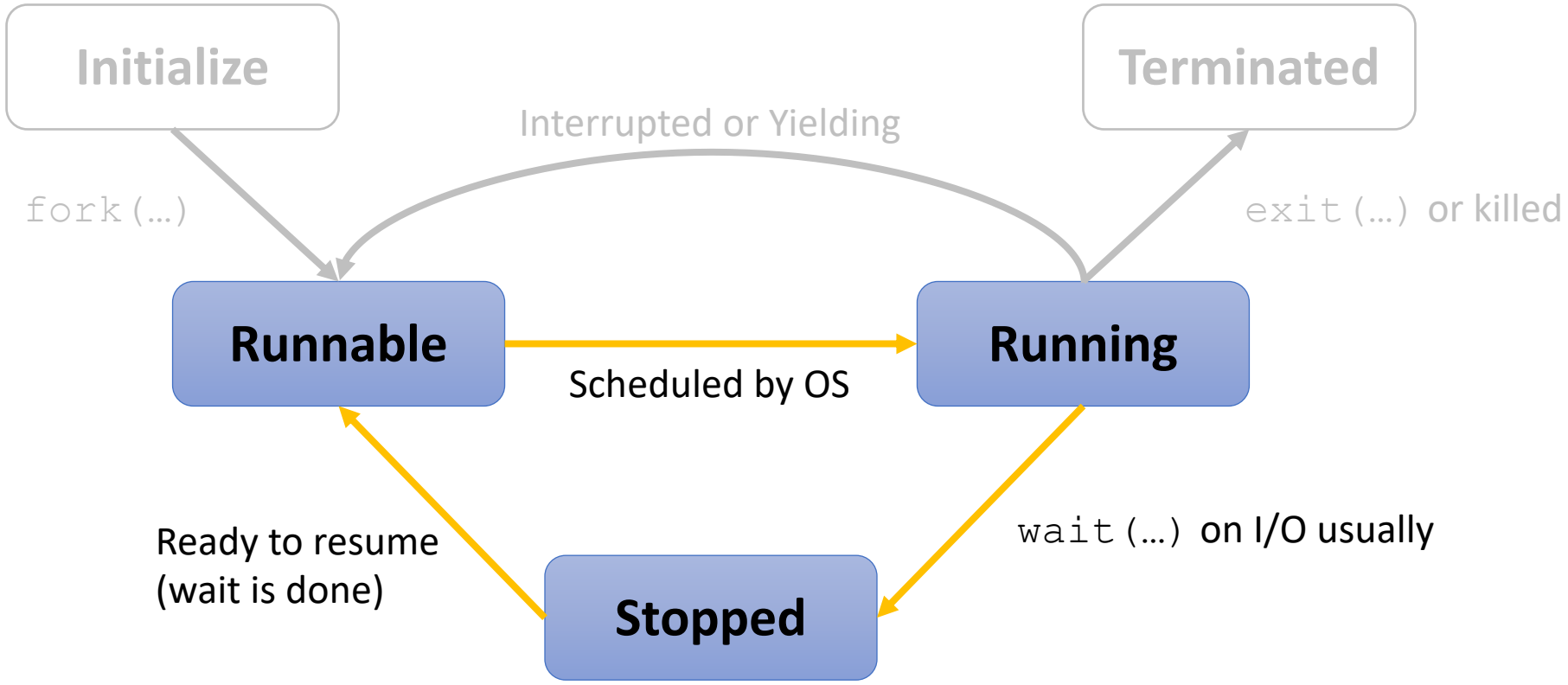
```
void fork3()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```



Which of these outputs are feasible?

L0	L0
Bye	Bye
L1	L1
L2	Bye
Bye	Bye
Bye	L2

Process Life Cycle (Linux)



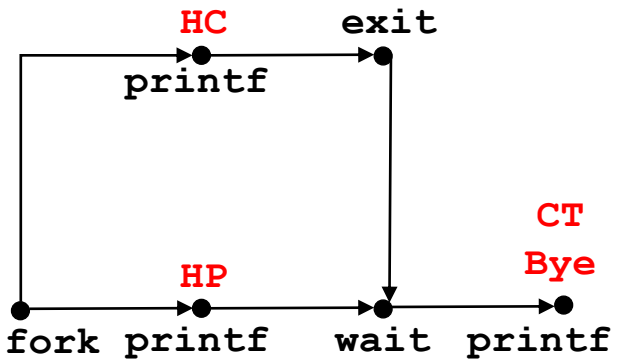
Reaping Children

- Reaping
  - Performed by parent on terminated child (using wait or waitpid)
  - Parent is given exit status information
  - Kernel then deletes zombie child process

- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is the pid of the child process that terminated
  - If child_status != NULL, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in wait.h
      - WIFEXITED, WEXITSTATIS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED
      - See textbook for details

`wait` Example

```
void fork6() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    }
    else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```



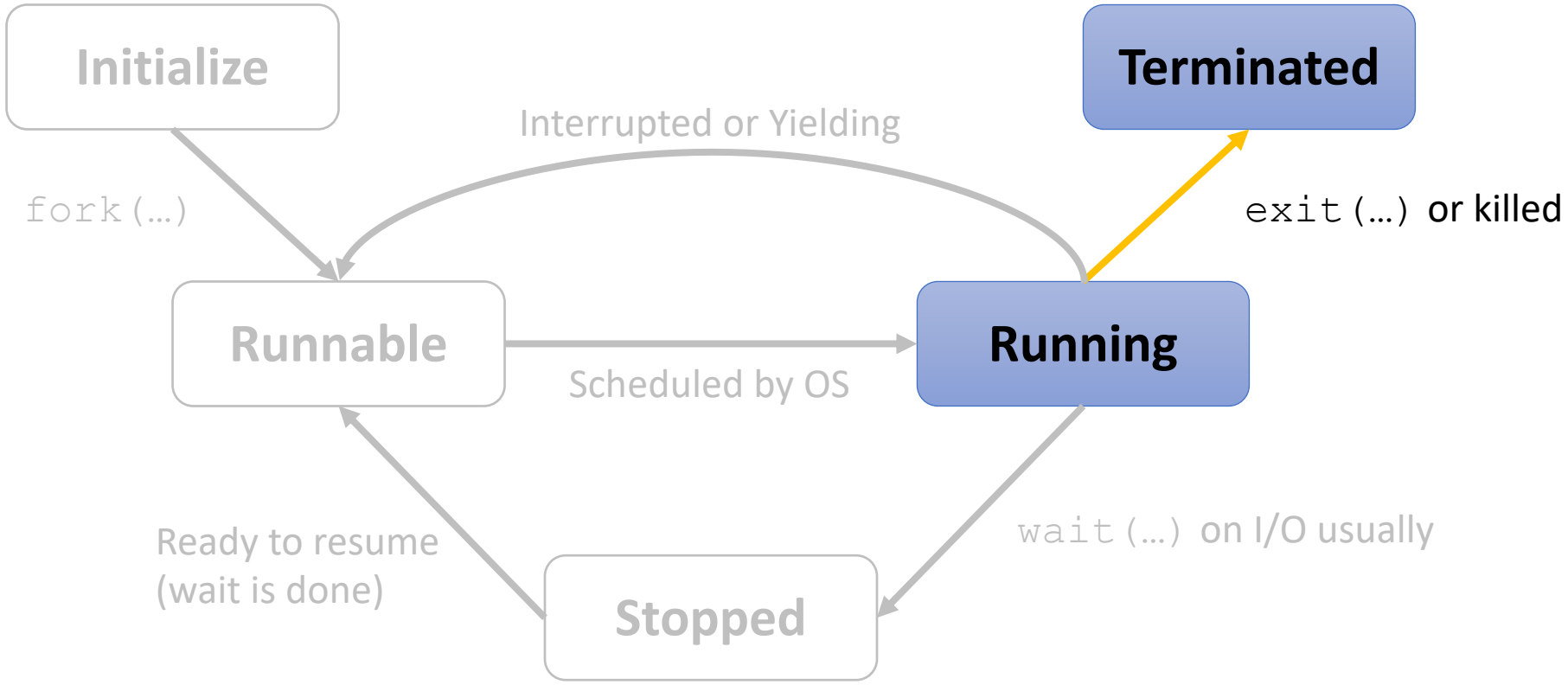
Feasible output:
HC
HP
CT
Bye

Infeasible output:
HP
CT
Bye
HC

Reaping Children

- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by init process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Process Life Cycle (Linux)



Terminating Processes

- Process becomes terminated for one of three reasons:
  - Returning from the `main` routine
  - Receiving a signal whose default action is to terminate
  - Calling the `exit` function

- `void exit(int status)`
  - Terminates with an **exit status** of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

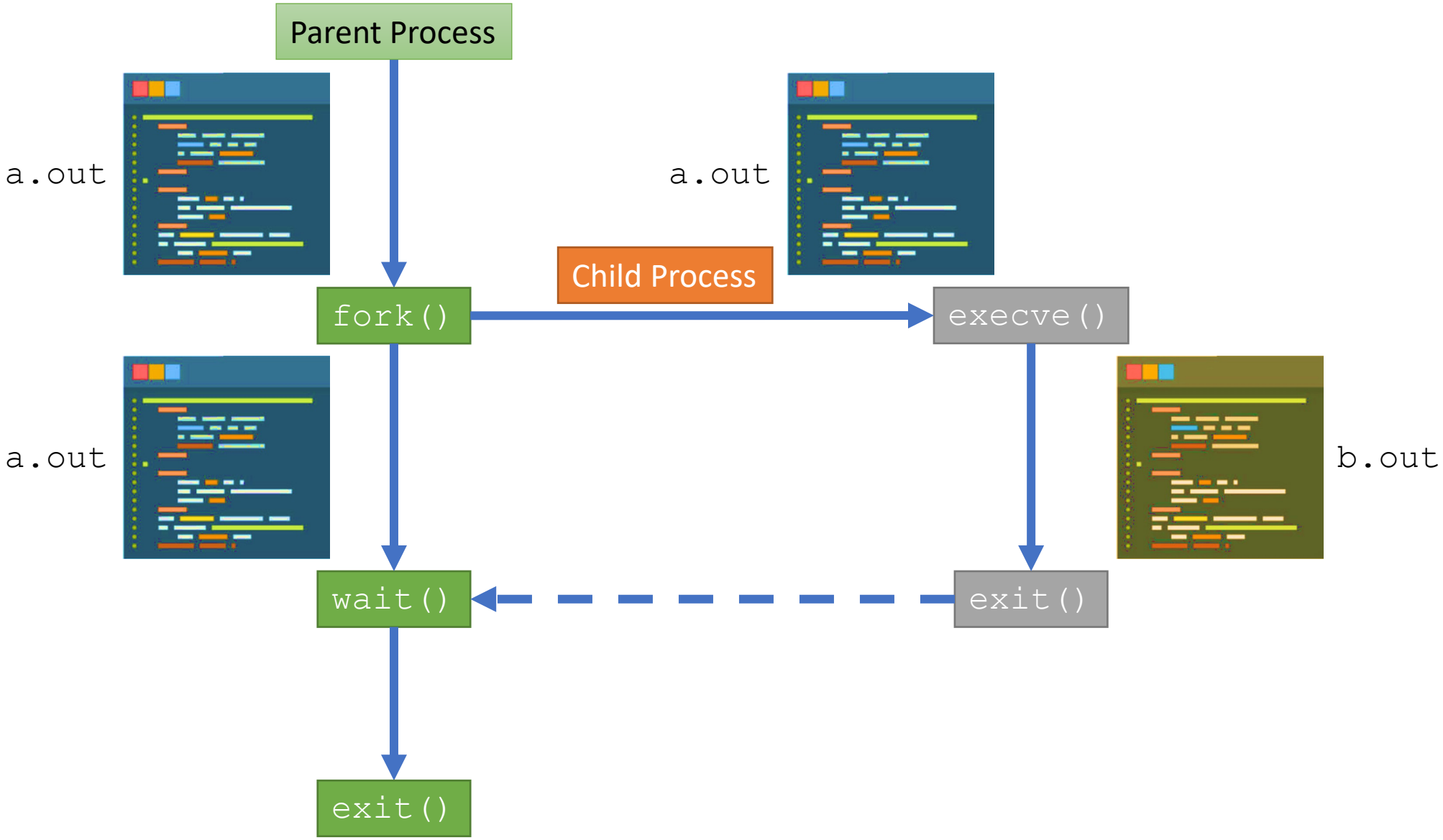
- `exit` is called once but never returns.

Loading and Running Programs: `execve`

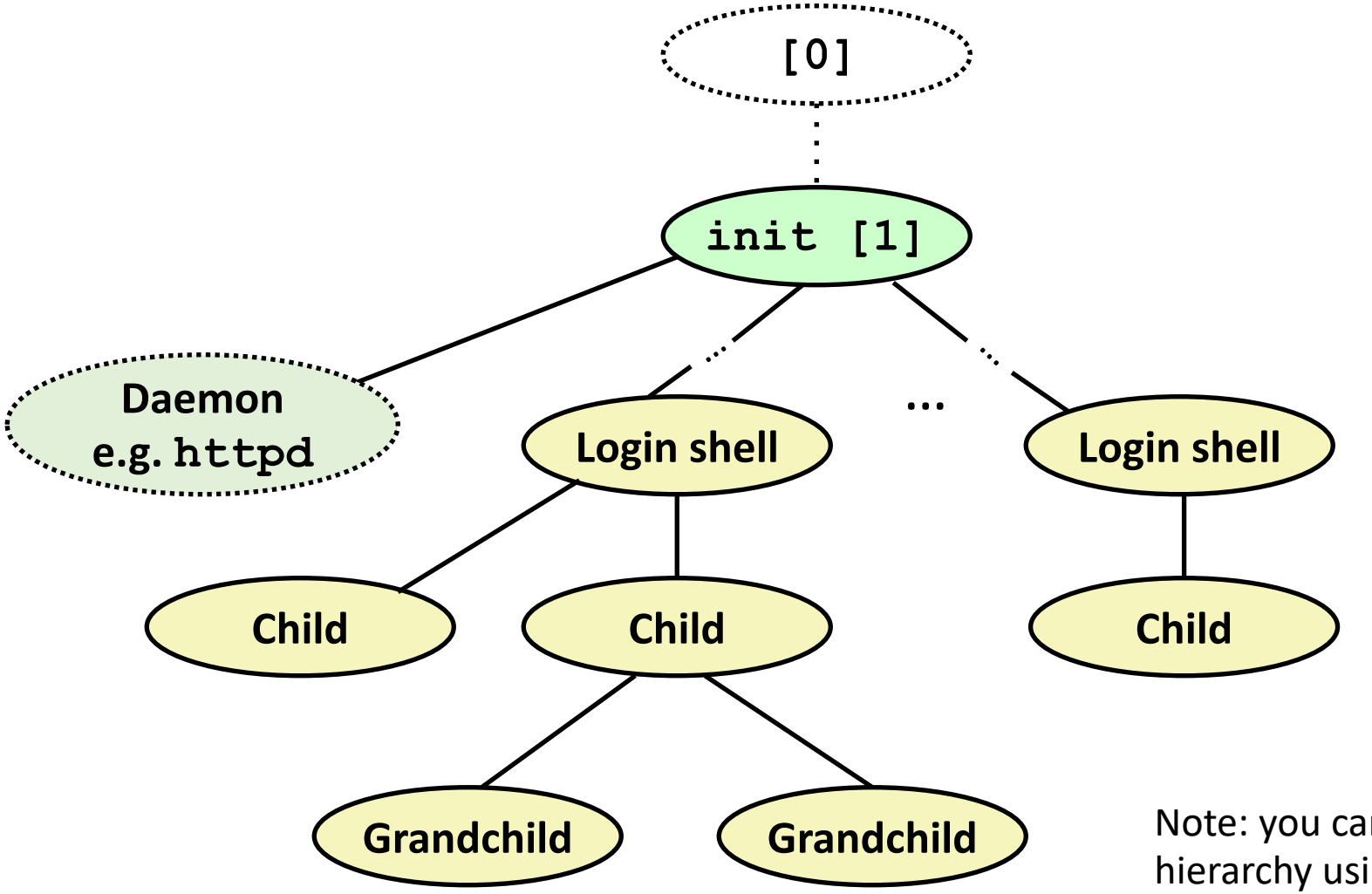
- What if we want to run a brand-new program?

```
int execve(char *filename, char *argv[], char *envp[])
```

- Loads and runs in the current process:
  - Executable file filename: can be object file or script file (e.g., `#!/bin/bash`)
  - …with argument list `argv`: by convention `argv[0]==filename`
  - …and environment variable list `envp`: "name=value" strings (e.g., `USER=droh`)
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called once and never returns (unless there is an error)


Parent Process
a.out
a.out
Child Process
fork()
execve()
a.out
b.out
wait()
exit()
exit()

Linux Process Hierarchy



Note: you can view the hierarchy using the Linux `pstree` command

pstree

```
[ajcd2020@itbdcv-lnx04p ~]$ pstree
systemd─┬─NetworkManager───2*[{NetworkManager}]
        ├─VGAuthService
        ├─agetty
        ├─atd
        ├─auditd───{auditd}
        ├─bomblab-reportd
        ├─bomblab-request
        ├─bomblab-resultd
        ├─bomblab.pl
        ├─chronyd
        ├─clamd───{clamd}
        ├─crond
        ├─dbus-daemon───{dbus-daemon}
        ├─firewalld───{firewalld}
        ├─freshclam
        ├─irqbalance───{irqbalance}
        ├─lsmd
        ├─mcelog
        ├─oddjobd
        ├─polkitd───11*[{polkitd}]
        ├─rhsmcertd
        ├─rsyslogd───2*[{rsyslogd}]
        ├─salt-minion───salt-minion─┬─salt-minion
        │                           └─3*[{salt-minion}]
        ├─2*[sh───node─┬─node───10*[{node}]]
        │              └─10*[{node}]]
        ├─smartd
        ├─sshd───sshd───sshd───bash───pstree
        ├─sssd─┬─2*[sssd_be]
        │      ├─sssd_nss
```

pstree

```
[ajcd2020@itbdcv-lnx04p ~]$ pstree ajcd2020
bomblab-reportd

bomblab-request

bomblab-resultd

bomblab.pl

sshd───bash───pstree

systemd───(sd-pam)
```