The Abstraction: The process

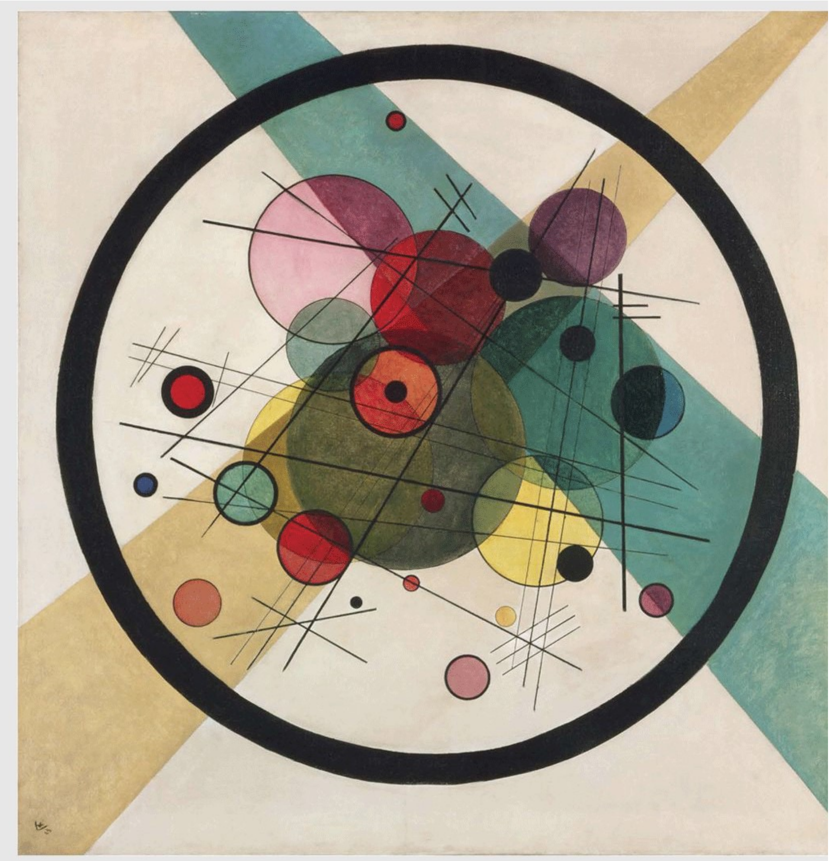
What is a process?

COM S 352
Iowa State University
Matthew Tan Creti

Program vs process

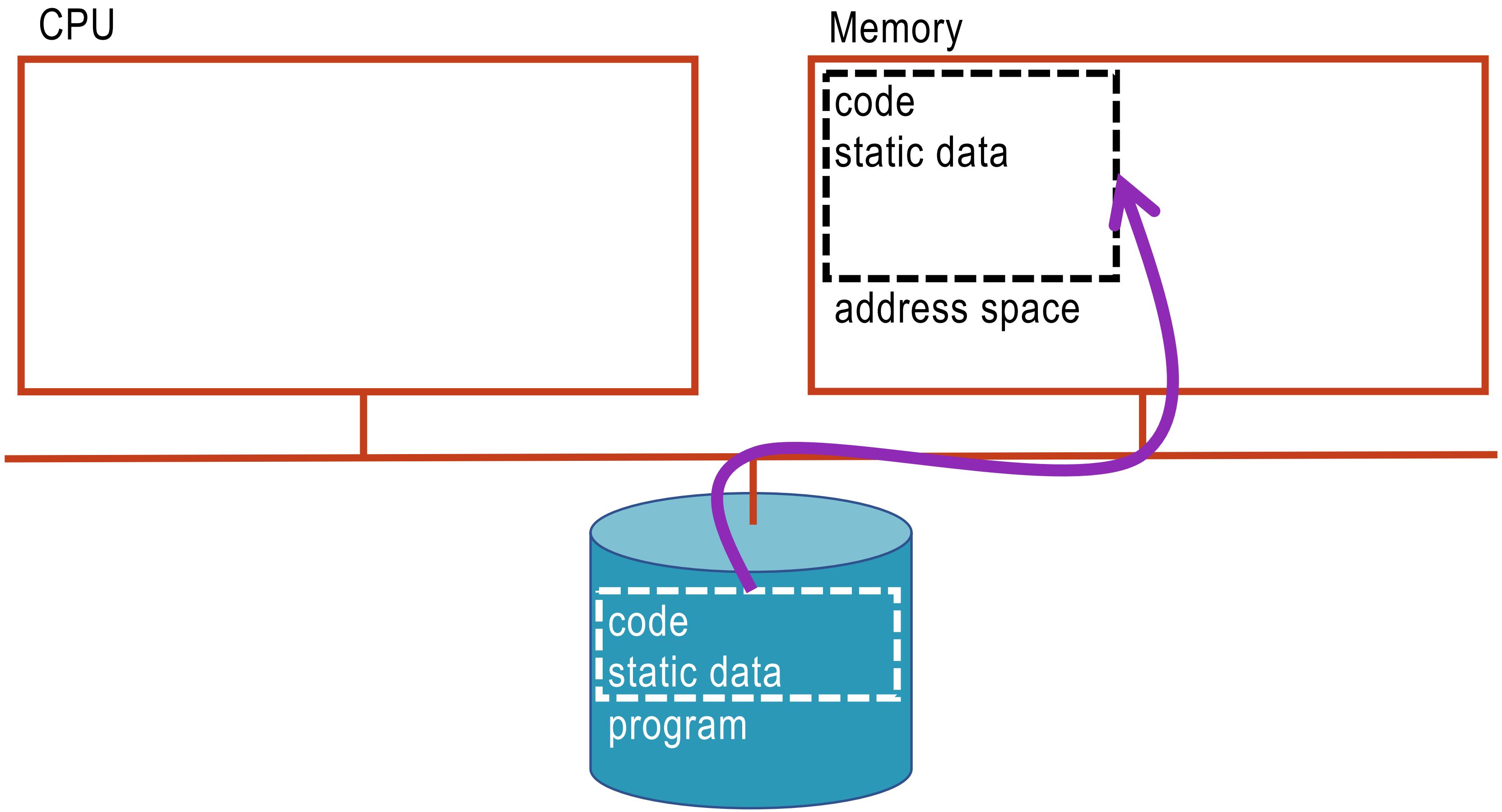
A program is instructions stored on disk – an executable file

A process is a running program

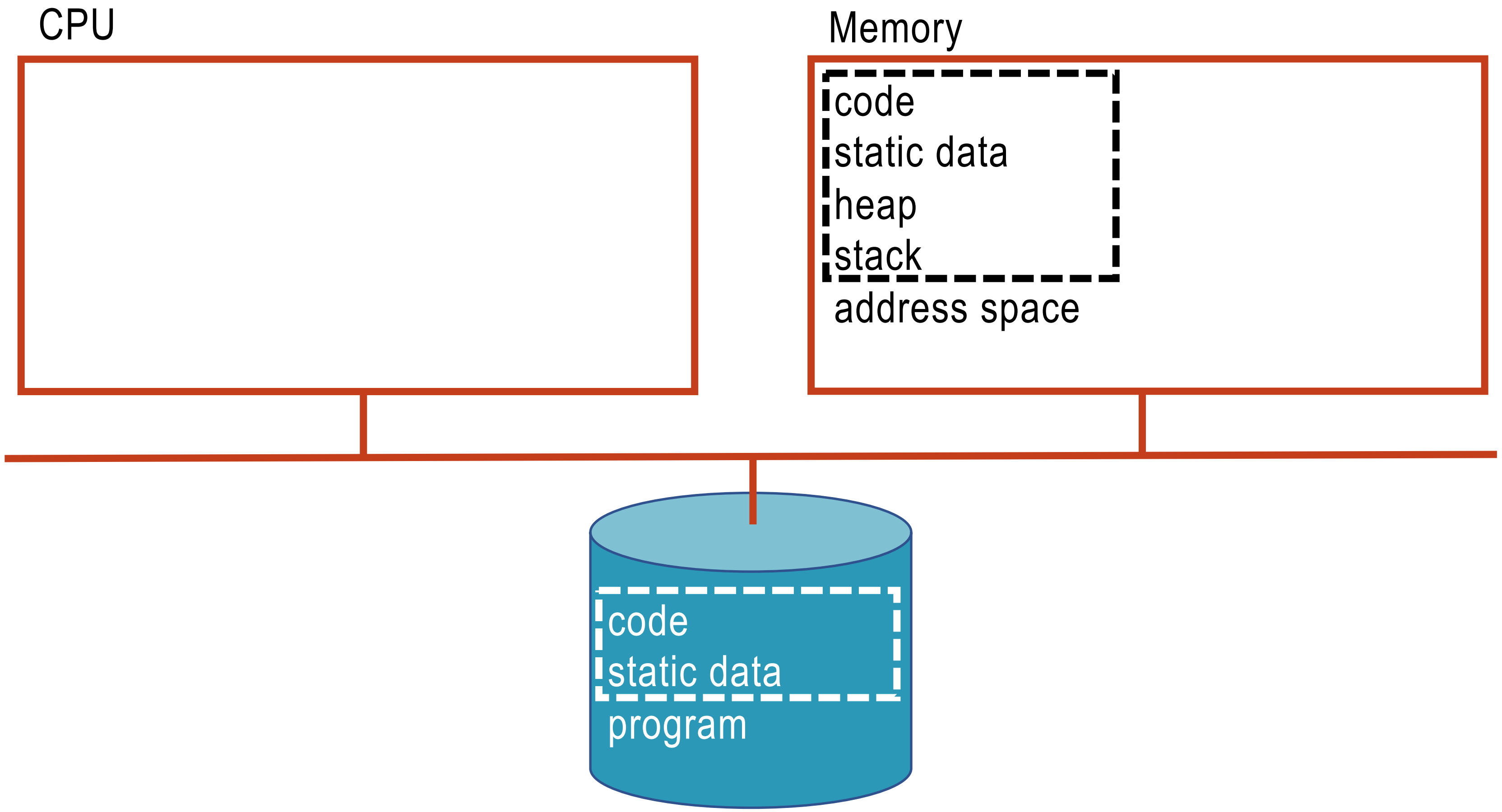


Circles in a Circle, Wassily Kandinsky, 1923 [source]

Process Creation



Address Space



The address space is where the programs data resides.

Process Information

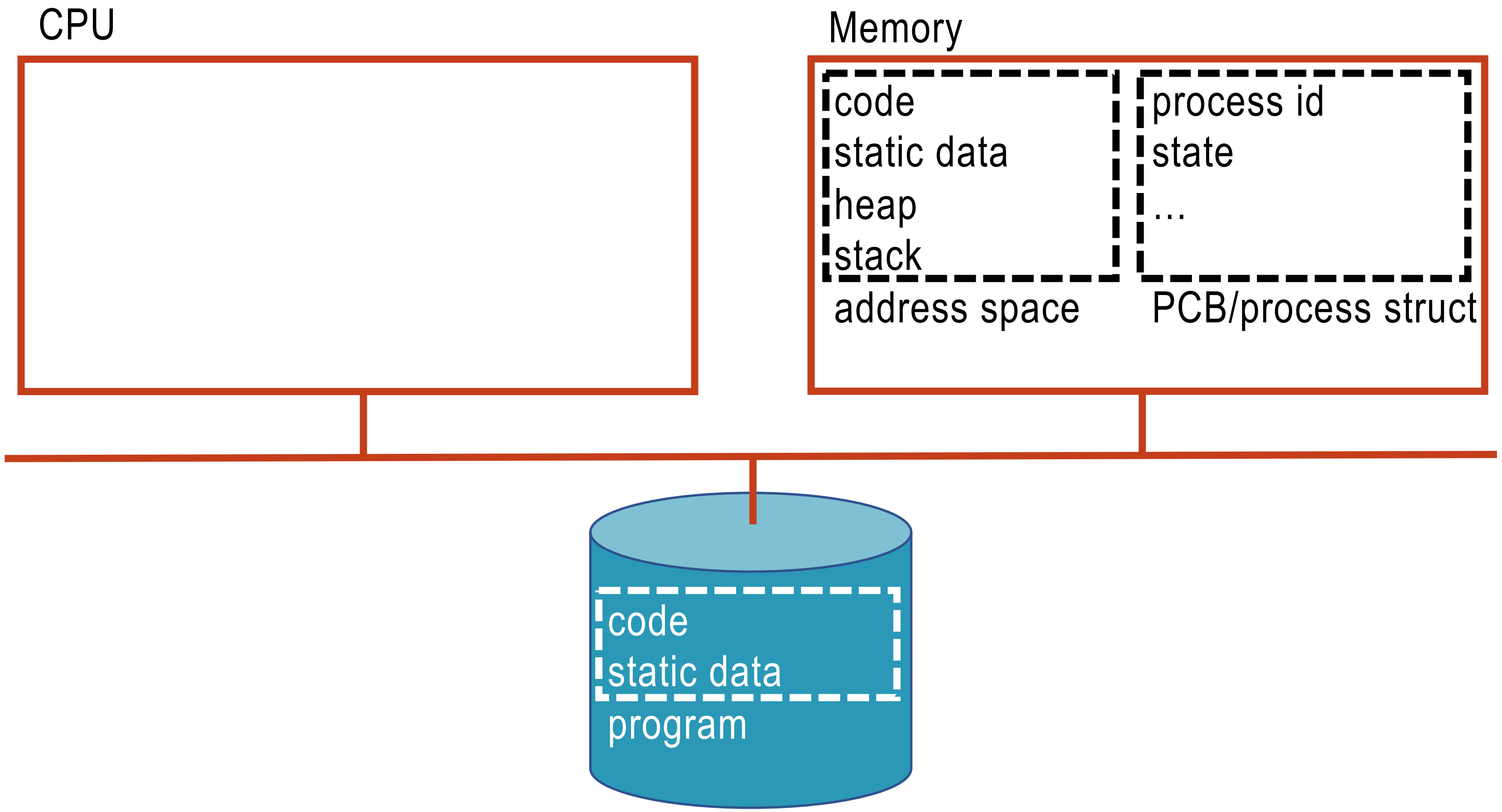
A process is more than a list of instructions, execution requires knowing…

- Which instruction to execute next? program counter
- What is the immediate data on which instructions operate? registers
- Where are the instructions and data stored in memory? address space
- How to find parameters of current function? stack and frame pointer
- And more that we will cover later in this class (e.g., open files, threads…)

Most of these terms should be familiar, if not,

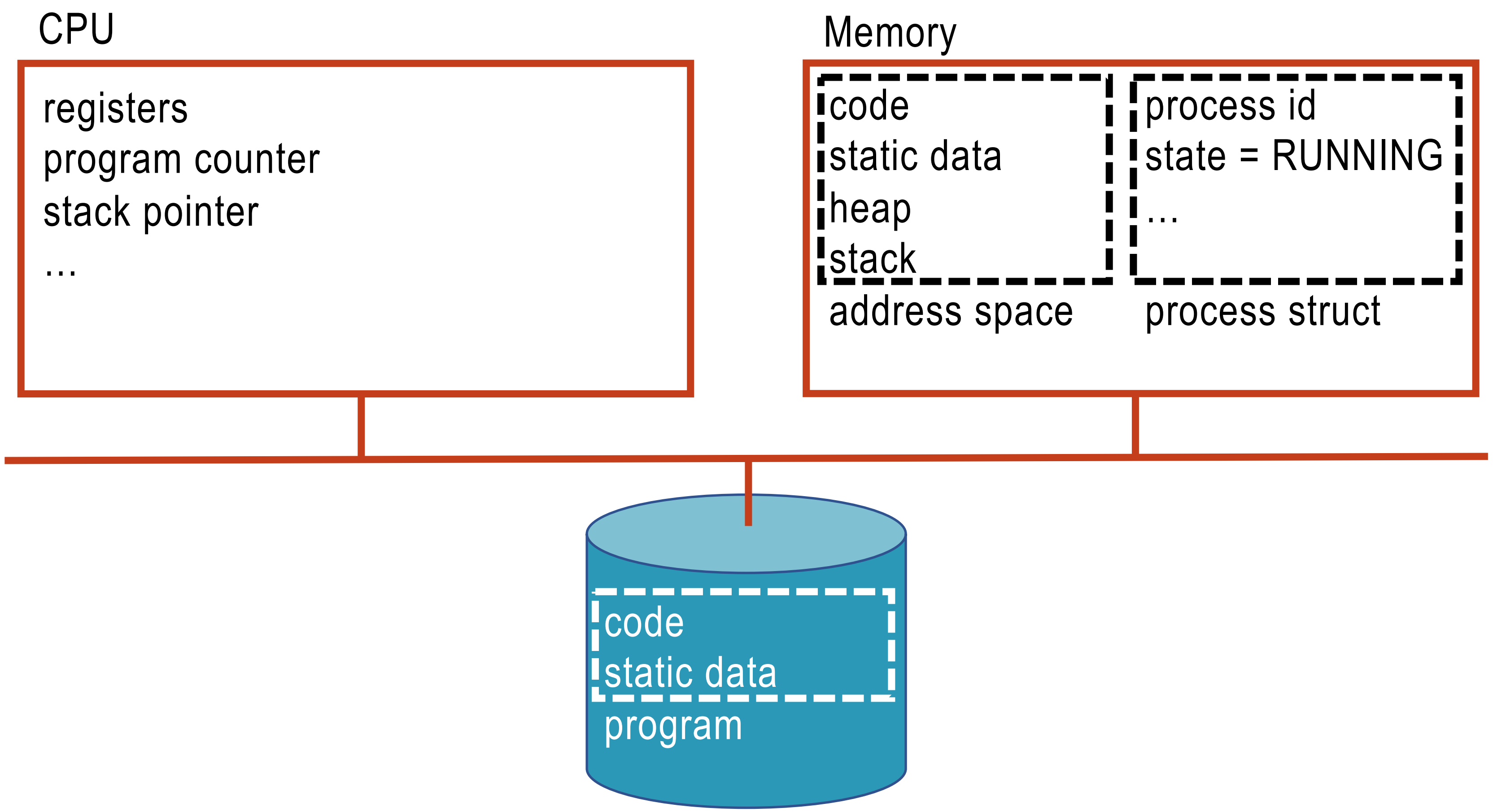
time to review your COM S 321/CPR E 228 notes.

Process Control Block



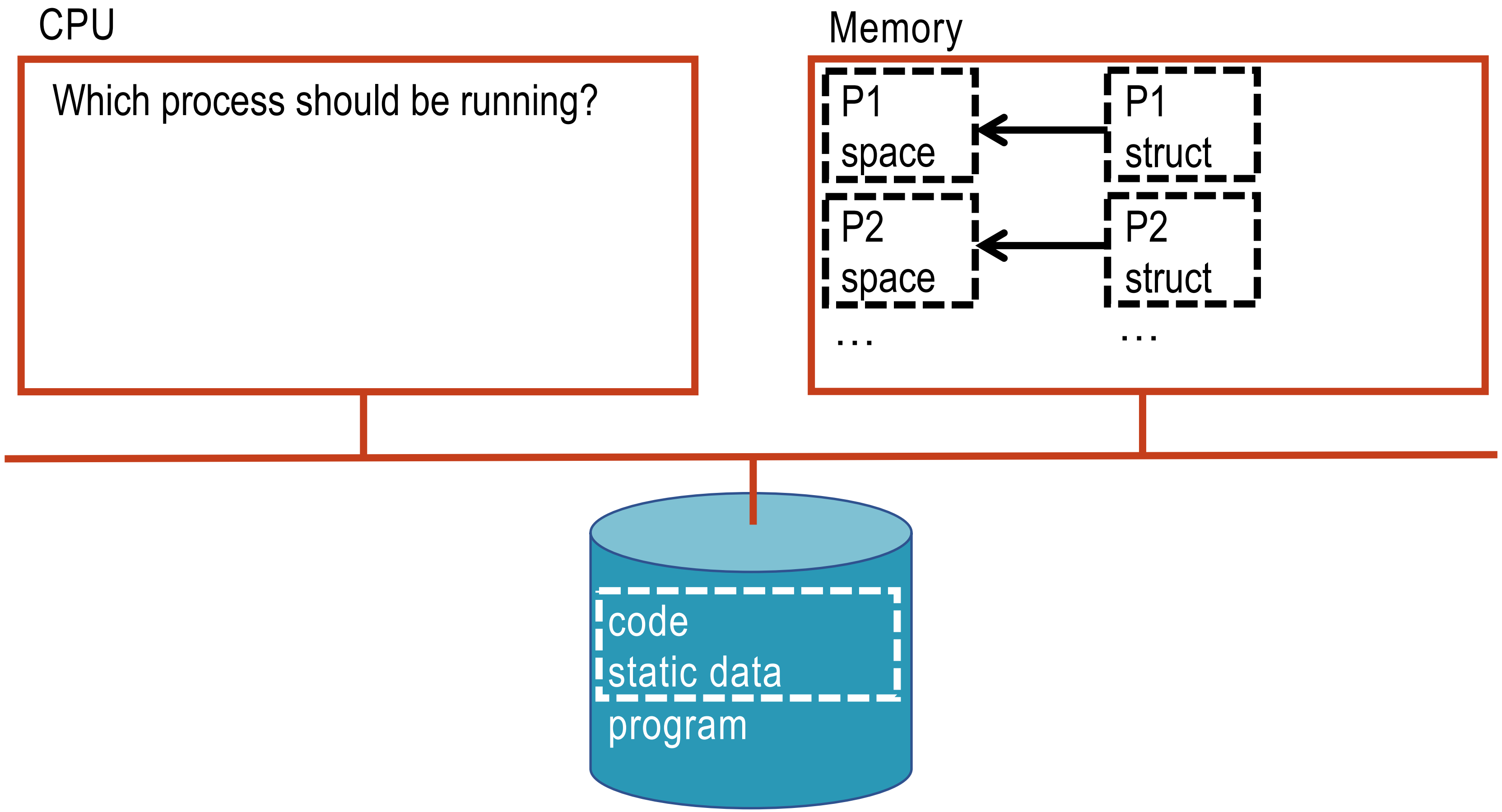
The Process Control Block (PCB) is a structure the OS uses to keep track of the process information.

Process In Execution



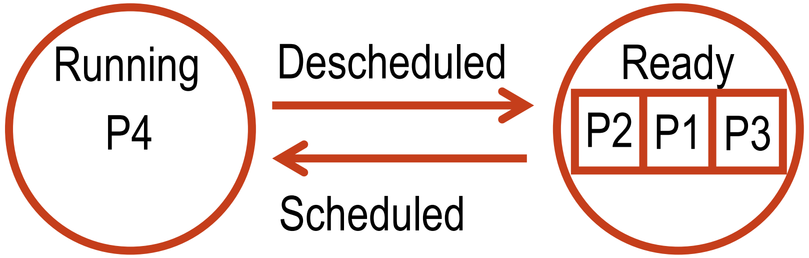
During execution, the state of the CPU is dedicated to the process.

Multiple Processes



We don’t want to be limited to one process, but a CPU can only execute one process at a time…

Scheduler: Running and Ready



The scheduler is part of the OS, its job is to manage the process' states
Only one process running at a time (single core assumption)
Other processes that are ready to execute go in a ready queue

Multiprogramming

Processes need to perform I/O, e.g., disk read/write

Consider a process the makes a random read from disk

Hard drive latency 10ms and SSD latency 1ms

CPU clock 0.3ns cycle and 3 instructions per cycle

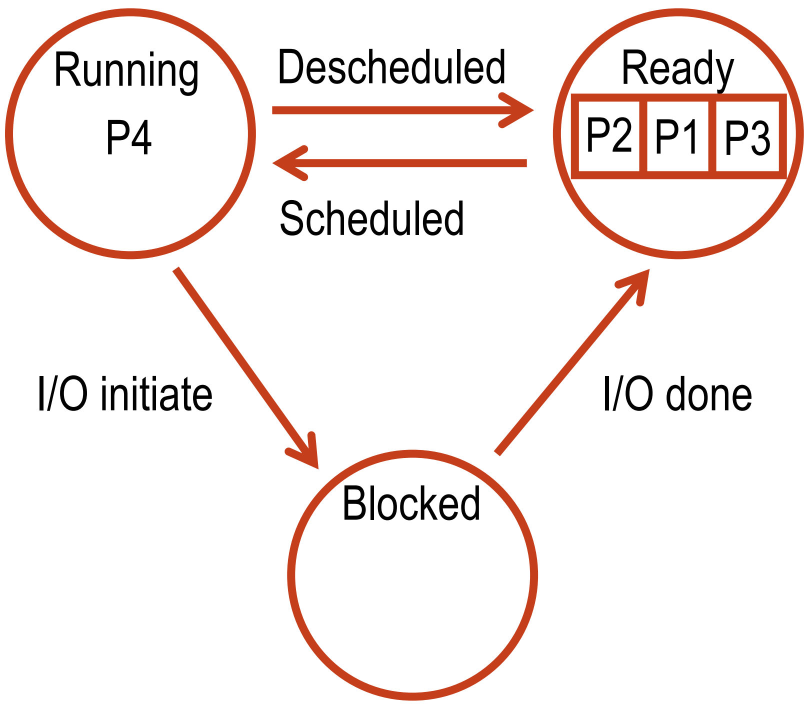
10 million instructions wasted waiting for SSD and 100 million for hard drive!!!!

Clearly not good use of the CPU

Multiprogramming is letting the user run multiple programs (processes) together

When one process needs to wait for I/O another can be scheduled on the CPU

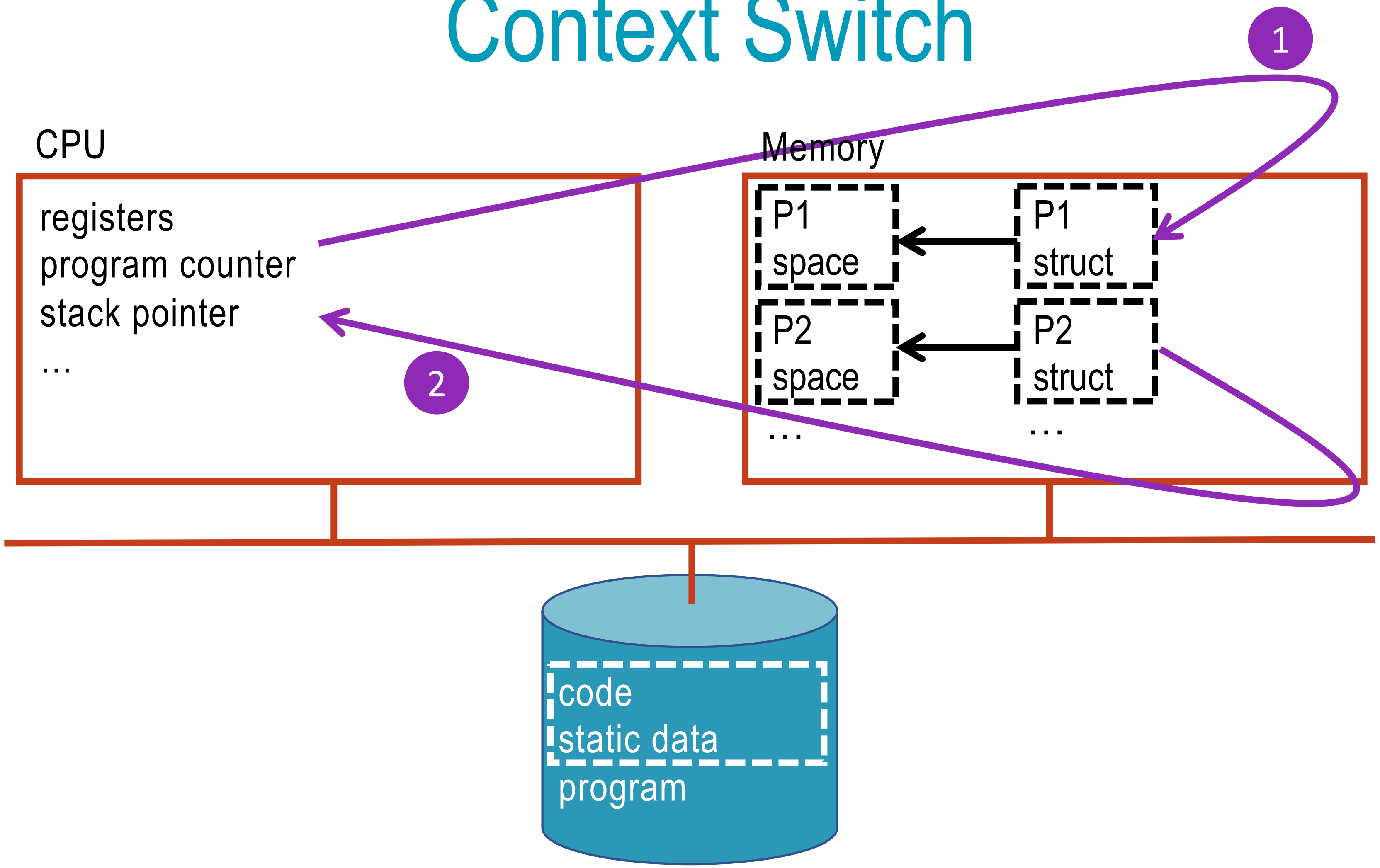
Scheduler with Blocked State



When a process is waiting for I/O it is placed in the blocked state

Next process in ready queue is set to running

Context Switch



Save CPU context of descheduled process
Load CPU context of scheduled process

Parent

Every process must have a parent

Initially the parent is the process that created it (the child process)

When a process is created it starts in an initial state and when it terminates it goes to a final (zombie) state

The final state is required because we can’t delete the PCB immediately, the parent may want information such as the exit code

Process Control Block (struct proc) in xv6

proc.h

```
// Saved registers for kernel context switches.
struct context {
  uint64 ra;
  uint64 sp;

  // callee-saved
  uint64 s0;
  uint64 s1;
  uint64 s2;
  uint64 s3;
  uint64 s4;
  uint64 s5;
  uint64 s6;
  uint64 s7;
  uint64 s8;
  uint64 s9;
  uint64 s10;
  uint64 s11;
};

enum procstate { UNUSED, USED, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };
```

```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // proc_tree_lock must be held when using this:
  struct proc *parent;         // Parent process

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Virtual address of kernel stack
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Process System Calls

POSIX (Portable Operating System Interface)

Standard programming interface provided by UNIX like systems

fork()	Create another process (child) that is a copy of the current process (parent)
exec()	Change the program of the currently executing process
wait()	Do nothing until a child process has terminated

Tuning fork [source]

fork( ) // Process Creation

```
pid_t fork(void);
```

Creates a new process by duplicating the calling process

Child process has a copy of parent's address space

On success:

- Both parent and child continue execution at the point of return from fork()
- Returns pid of the child process to the parent process; returns 0 to child process

On failure:

- Child is not created; returns –1 to parent fork.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int args, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

console

```
hello world (pid:19979)
hello, I am parent of 19980 (pid:19979)
hello, I am child (pid:19980)
```

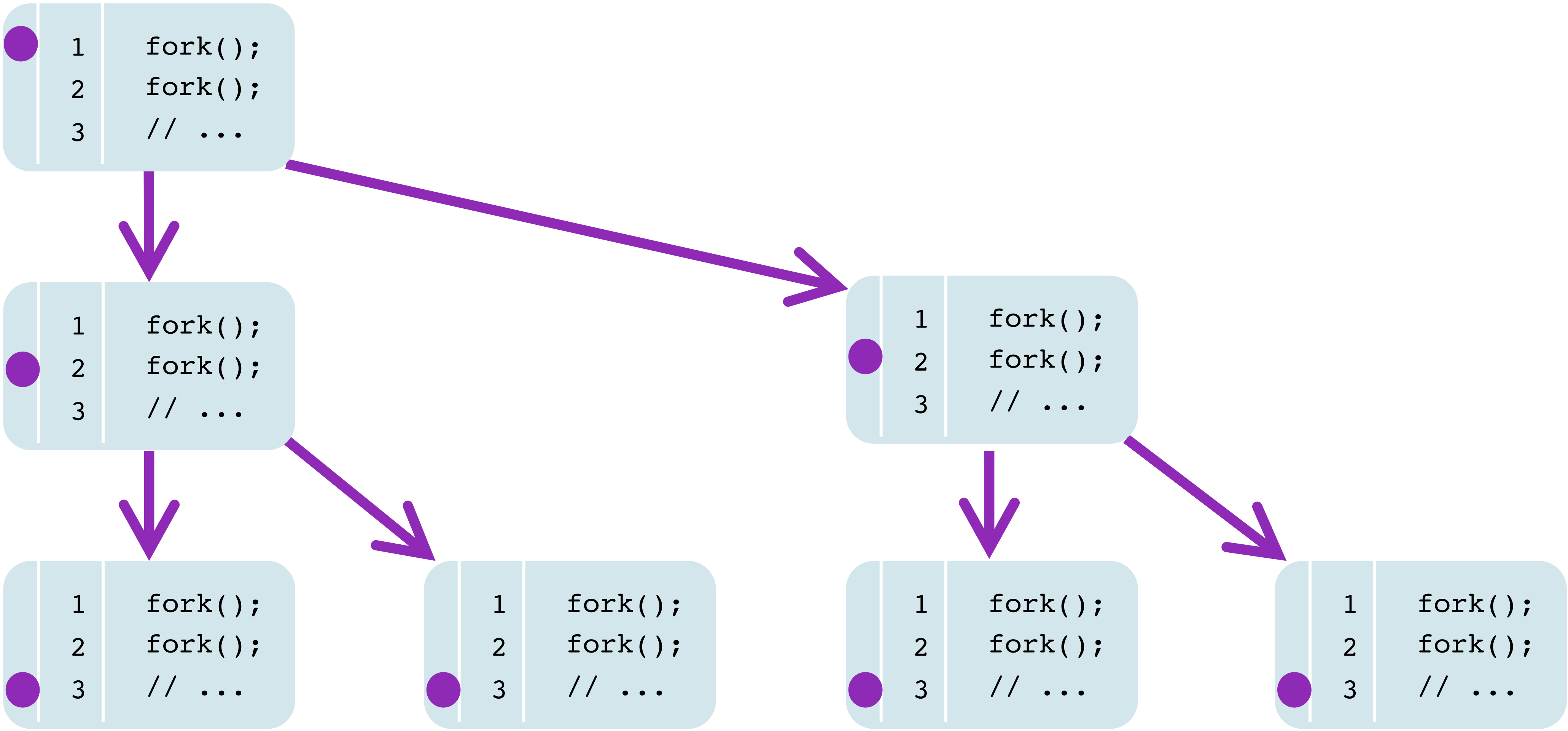
parent

```c
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

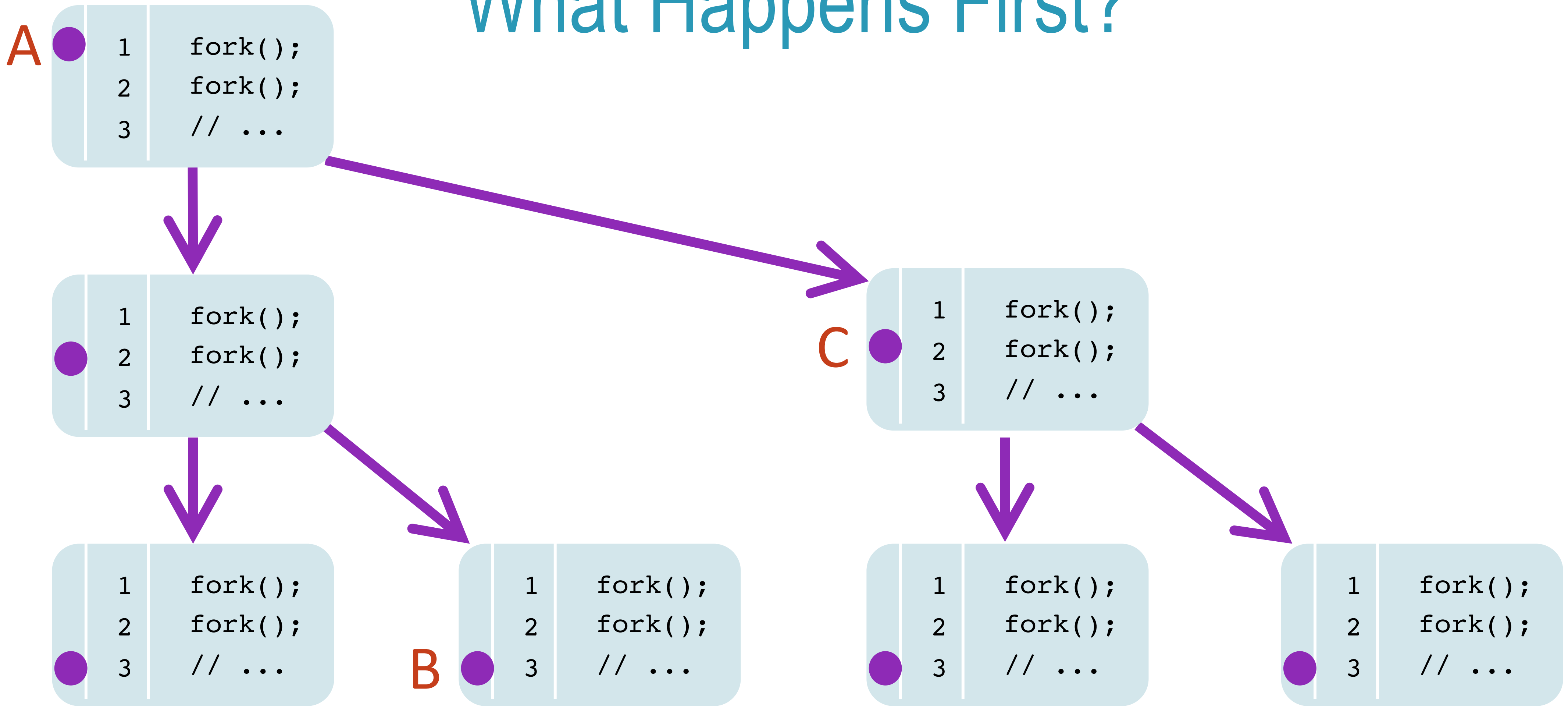
child

```c
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
```

Process Tree



What Happens First?



Question: What is the correct order for A, B and C?

wait( ) // Wait for Child

```
pid_t wait(int *wstatus);
```

Suspend execution of the parent until one of its children terminates

On success:

- Returns pid of the child process that terminated
- wstatus is populated with information about the way the child process terminated

If a process has terminated, but parent has not yet called wait(), the process becomes a zombie

If the parent terminated without calling wait(), the child process becomes an orphan, and a system process (systemd in Linux) becomes the parent

View Processes on Command Line

```
$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.1 186896 15884 ?        Ss    2020  13:22 /usr/lib/systemd/systemd --system --deserialize 39
root     2758580  0.1  0.1  41896  9916 ?        SNs  00:45   0:00 sshd: tancreti [priv]
tancreti 2758586  0.0  0.0  41772  5904 ?        RN   00:45   0:00 sshd: tancreti@pts/1
root     2758587  0.0  0.0      0     0 ?        I    00:45   0:00 [kworker/3:2]
tancreti 2758588  0.5  0.0  17352  5444 pts/1    SNs  00:45   0:00 -bash
root     2758598  0.0  0.0      0     0 ?        I    00:45   0:00 [kworker/4:2]
tancreti 2758634  0.0  0.0  17932  3616 pts/1    RN+  00:45   0:00 ps aux
```

systemd, Linux parent, PID=1

SSH processes managing my connection

bash shell process I am interacting with

ps command (it saw itself)

Exploring Process Tree on Command Line

```
$ pstree
systemd─┬─NetworkManager───2*[{NetworkManager}]
 ├─sshd─┬─sshd───sshd───bash───pstree
 │      └─sshd───sshd───bash
...
```

pstree command

bash shell

SSH child to manage connection A

SSH child to manage connection B

SSH server

systemd is Linux parent process

Question: How did bash create a child process that executes pstree?

exec( ) // Change the Program

```
int exec(const char *pathname, char *const argv[]);
```

Replaces the current program with a new one

Command line arguments are passed in argv

The process keeps it file descriptors

On success:

- The process is running a new program
- The function does not return (no where to return)

On failure:

- The function returns -1

Frist argument is always the name of the executable

Example usage:

```
char *args[]={"wc","README",0};
exec("wc", args);
```

Interprocess Communication

A pipe is a unidirectional data channel that can be used to communicate from one process to another

- Sender puts data to one end (the write-end of the pipe)
- Receiver gets data from the other end (the read-end of the pipe)

Common example is the shell, it creates two children and pipes standard out (e.g., printf) of one into standard in (e.g., read) of other

pipe( ) // Connect two processes

```
int pipe(int p[2]);
```

Creates communication channel

Typical usage is right before calling fork, each process must close the ends of the pipe it is not using

On success:

- p[0] is file descriptor of read side of pipe, p[1] is write side of pipe
- Returns 0

On failure:

- Returns –1

Example Usage:

- See sh.c

dup( ) // Duplicate file descriptor

```
int dup(int fd);
```

Returns new file descriptor that is the lowest numbered available descriptor

New file descriptor refers to the same source as fd did previously

For example, closing standard out (1) and then calling dup(fd) will cause all calls to printf to be directed to what fd pointed to

On success:

- New file descriptor points to source of provided file descriptor
- Returns new file descriptor

On failure:

- Returns –1

Example Usage:

- See sh.c