CSE2431 – Lecture Topic 2
Process (part 1)

Process (Part 1)

Instructor: Luan Duong, Ph.D.

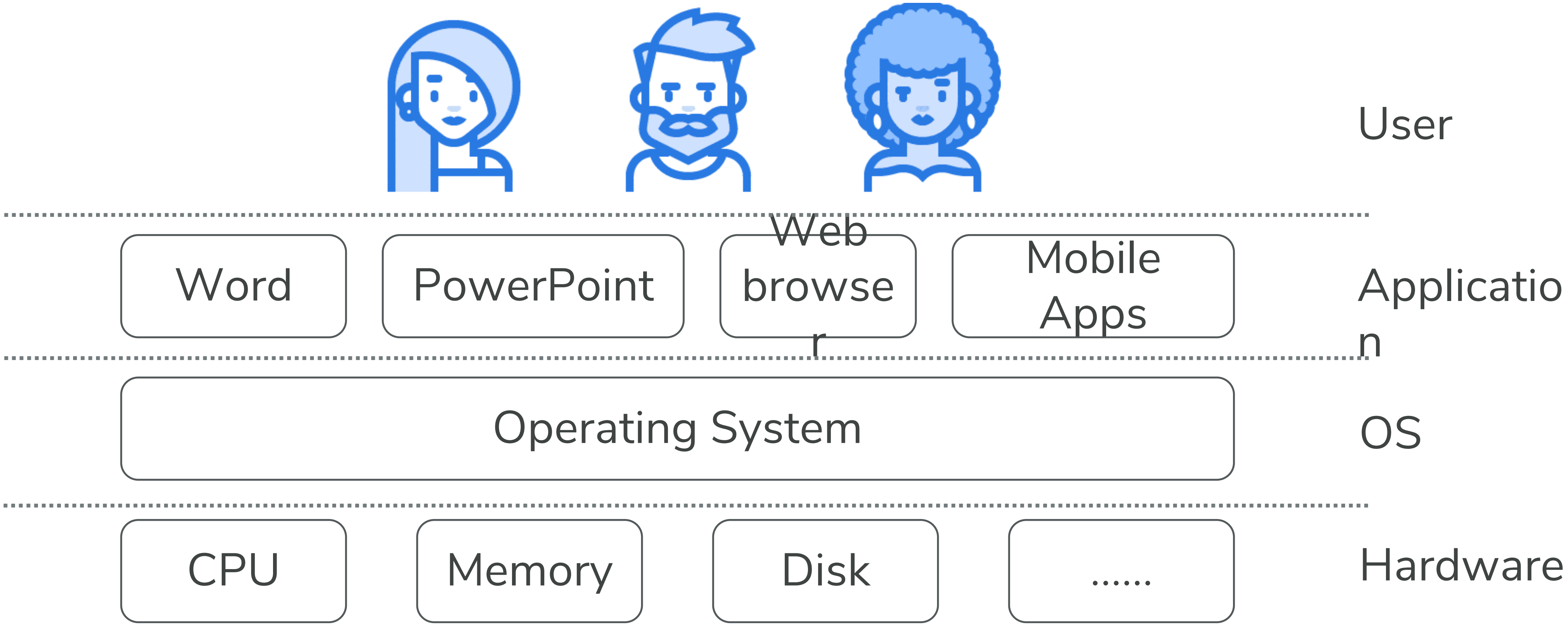
CSE 2431: Introduction to Operating Systems

Reading: Chapter 4-5 in required textbook

Lecture slides and materials adopted and referred from previously taught course by Dr. Yang Wang and Dr. Adam C. Champion

Last lecture

- Computer Architecture:



Last lecture

- Computer Architecture:

Operating System

Provide services

- Abstraction
- Convenience
- Standardization

Manage resources

- Allocation (CPUs, Memory, I/O devices)
- Reclamation (Voluntary at runtime, preemptive…)
- Protection (Protect from unauthorized access)
- Virtualization (Virtual memory, timeshared CPU)

Outline: Process

- What is a process?
- Process States; Process Control Block (PCB)
- Process Creation; *fork* command
- Process Memory Layout
- Process Scheduling
- Context Switch
- Process Operations
- Inter-Process Communication
- Client-Server Communication

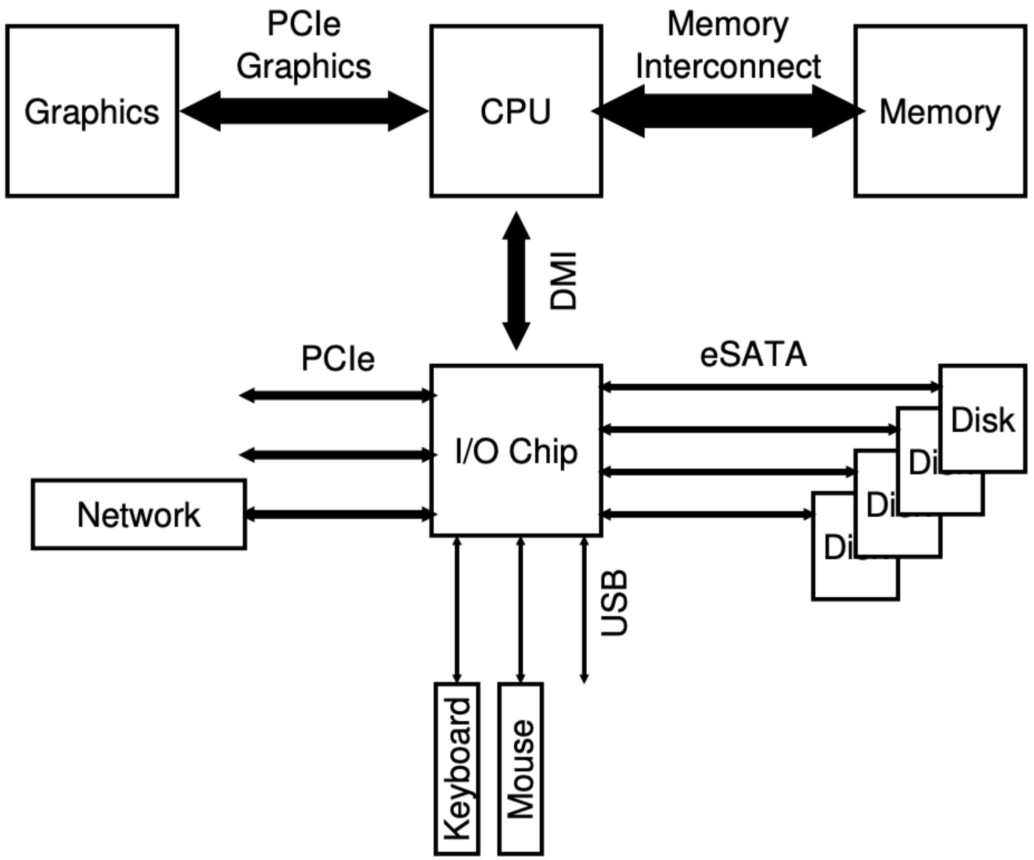
What is a process?

- Back to a question from previous lecture:
  - We only have one CPU, how could we run multiple programs or commands at the same time?
  - How to provide the illusion of many CPUs?

Virtualization

  - So how can we call/abstract a thing provided by the OS through virtualization?

A Process



What is a process?

- Users launch programs
  - Many users can launch the same programs
  - One user can launch many instances of the same programs
- Processes: an executing ‘piece’ of program
- Do you think of any real-life analogies? → Class discussion

What is a process?

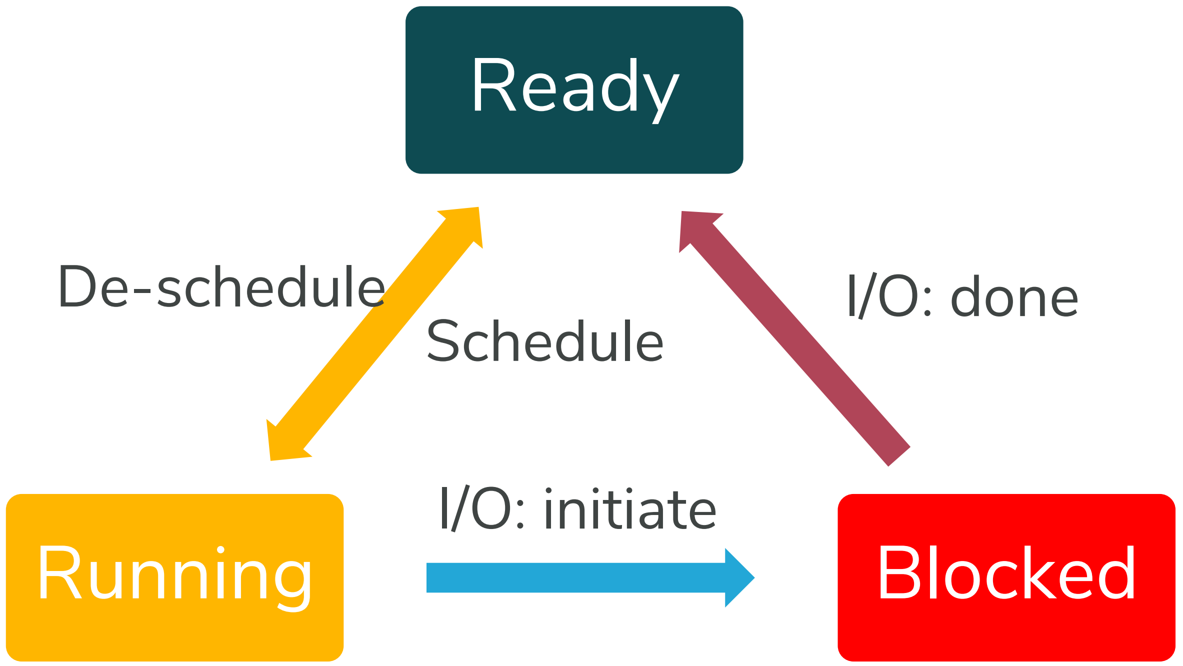
- A process is simply a running program; at any instant in time. It is also separated from other instances.
  - On batch system, refer to *jobs*
  - On interactive system (today OS), refer to *processes*
- Process <u>can launch other processes</u> or <u>can be launched</u> by others.
- Process ≠ Program:
  - A program is ***static***, while a process is ***dynamic***.
- A process includes:
  - A program counter
  - Stack
  - Data section

How can OS perform virtualization?

- Time sharing (and Space sharing)
    - “Allowing the resource to be used for a little while by one entity, and then a little while by another, and so forth” [Operating Systems: Three Easy Pieces]
    - Each process becomes slower, but users usually cannot tell, because computers are much faster than human beings!
    - Counterpart of time sharing is space sharing, where a resource (CPU) is divided (in space/memory) among those who wish to use it.
- Process operation
    - Create
    - Destroy
    - Wait
    - Miscellaneous Control
    - Status (i.e. Process state)

Process status

- Running: a process is being executed by a CPU
- Ready: a process is ready to run but is not running
- Blocked: a process is waiting on some event to take place



Process status: Examples

- Running: a process is being executed by a CPU
- Ready: a process is ready to run but is not running
- Blocked: a process is waiting on some event to take place

Time	Process[0]	Process[1]	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process[0] now done
4	[Destroyed]	Running	
5 …		Blocked	I/O initiate (keyboard interrupt → run other processes)
11		Ready	I/O done
12		Running	
13		[Destroyed]	Process[1] now done

Process status: Examples

- Running: a process is being executed by a CPU
- Ready: a process is ready to run but is not running
- Blocked: a process is waiting on some event to take place

Time	Process[0]	Process[1]	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process[0] initiates I/O
4	Blocked	Running	Process[0] is blocked, so Process[1] runs
5	Blocked	Running	
6	Ready	Running	I/O done
7	Ready	Running	Process[1] now done
8	Running	[Destroyed]	Process[0] now done

Process status: PCB

- How to maintain a lot of processes' status? → Process Control Block (PCB)
  - Process state
  - Process identification (PID)
  - Program counter
  - CPU registers
  - CPU scheduling info
  - Memory-management info
  - Accounting info
  - I/O status info
  - PID of parent process

Process status: PCB

- How to maintain a lot of processes' status? → Process Control Block (PCB)

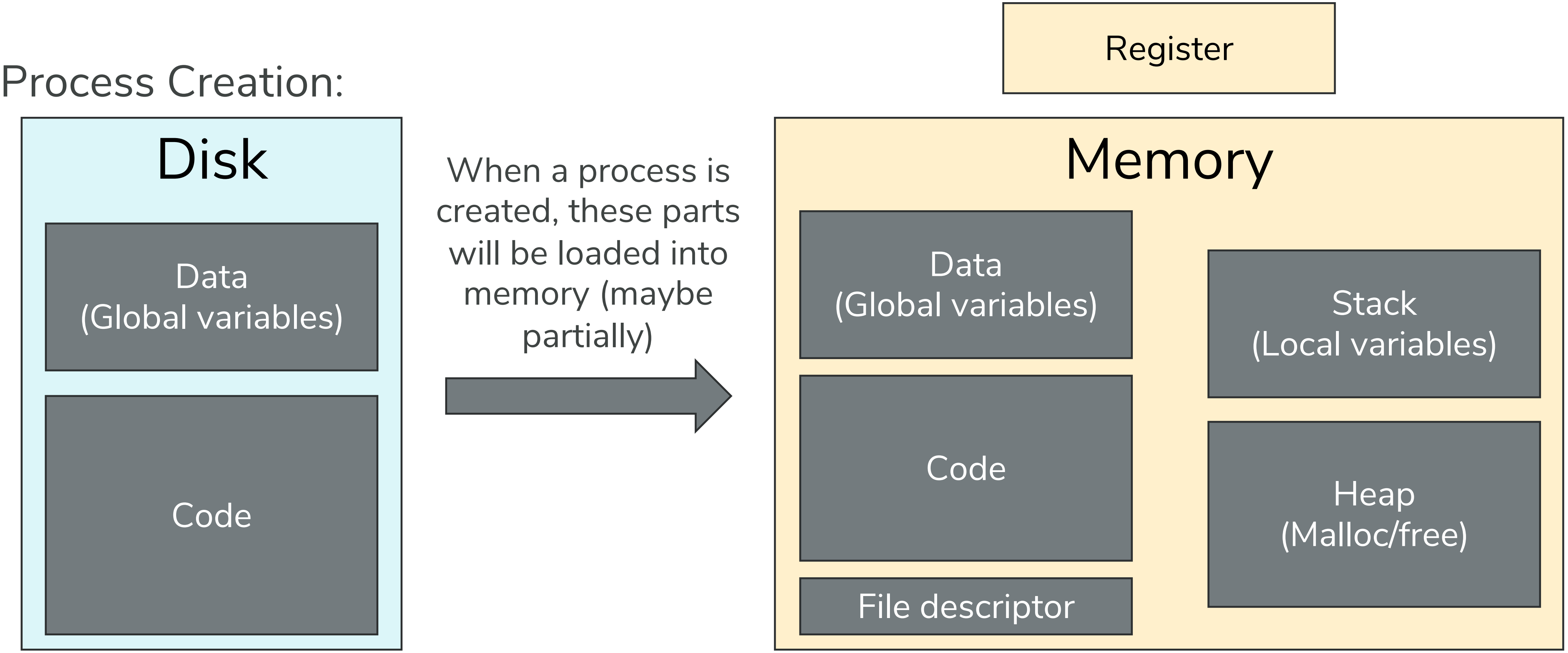
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };
```

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If !zero, sleeping on chan
  int killed;                 // If !zero, has been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Example of a PCB: The xv6 Process Structure [Operating Systems: Three Easy Pieces

Process Creation

- Process Creation:



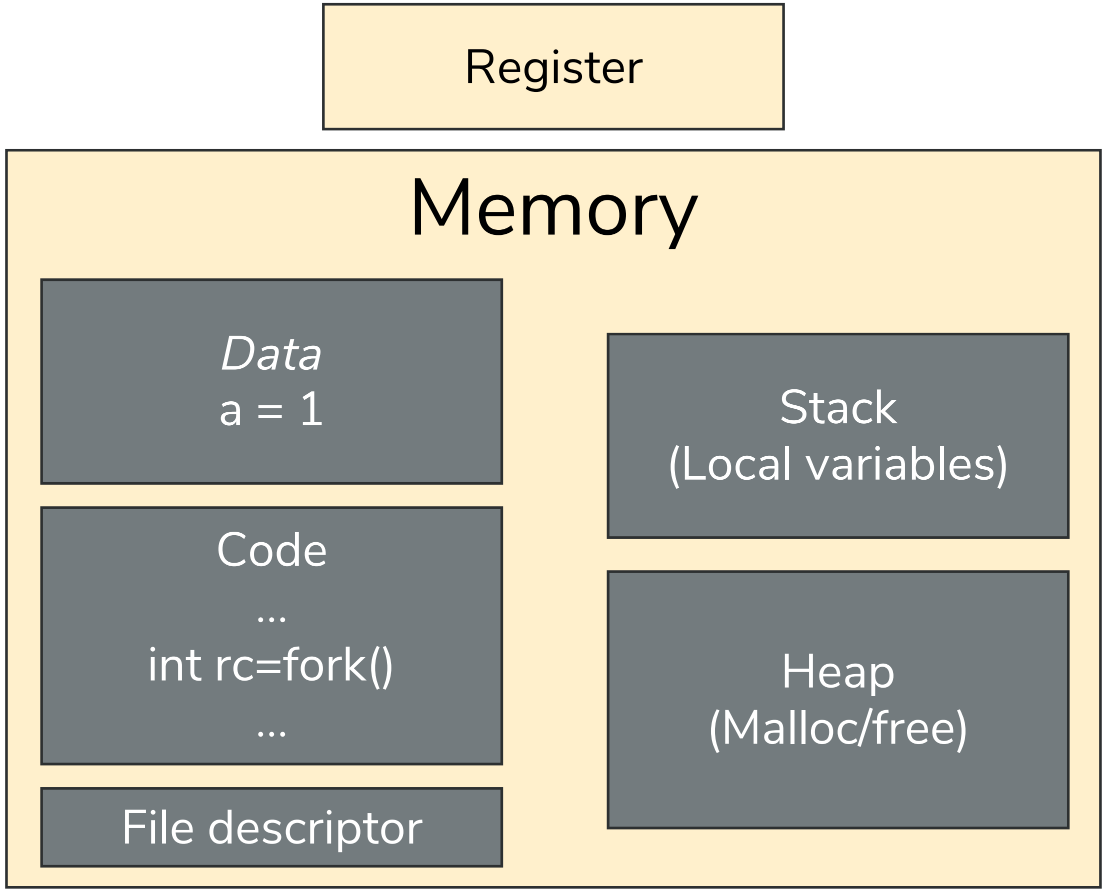
After your program is compiled, it usually consists of two parts

OS creates additional data structures for a process

Process Creation through System calls

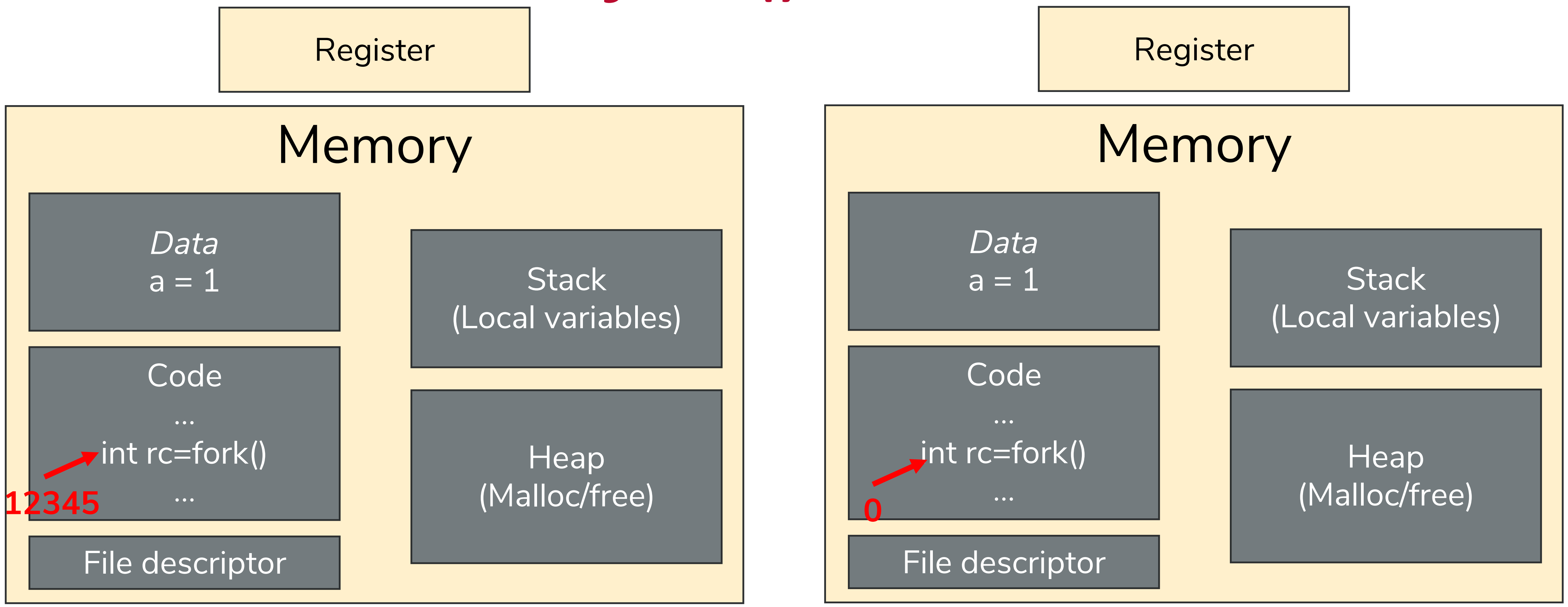
- ***fork***: duplicate the process that is calling fork
- ***exec***: run a new program (the new process will replace the current process)
- *fork* and *exec* are often used in combination to create a new process
- Other useful system calls: wait, pid, …

Process Creation: *fork()*



Parent process --- the process that is calling fork

Process Creation: *fork()*



Parent process --- the process that is calling fork

Child process

Parent and child processes are almost identical except one thing: return value of fork. For parent, fork returns the process ID of the child. For child, fork returns 0. Why?

Process Creation: *fork()*

- First: A parent process sometimes needs to control its child processes
  - For example: wait for a child process to terminate; kill a child process; …
  - So a parent process needs the process ID of its child

- Second: we need a way to differentiate parent and child processes
  - This means the return value to the child should not be a valid process ID: that is why we give it 0 (negative return value usually indicates error in C)

- Solution: return 0 to child; return child process id to parent

Process Creation: *fork()* example

```
int rc = fork();
if (rc < 0) {
  // fork failed
  fprintf(stderr, "fork failed\n");
  exit(1);
} else if (rc == 0) {
  // child (new process)
  printf("child (pid:%d)\n", (int) getpid());
} else {
  // parent goes down this path (main)
  printf("parent of %d (pid:%d)\n",
          rc, (int) getpid());
}
```

Process Creation: *fork()* – Important facts

- Parent and child processes **do NOT share memory**
- Parent's execution after folk **will NOT** affect child, and vice versa (which means parent and child can either execute concurrently or parent waits until child terminates it depends)
- Parent and child are executed in parallel and can be executed **in any order**.

- In Unix system:
  - All resources shared (i.e. child is a clone)
  - Execve() system call used to replace process' memory with a new program.

Process Creation: *fork()* – Some exercise

- What is the output of this code?
- Naturally, you will think it will output this:

```
prompt> ./p1
hello (pid:29146)
child (pid:29147)
parent of 29147 (pid:29146)
prompt>
```

- But another case can happen. Why?

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

Let’s do an exercise

```
int num = 3;

int main(int argc, char**argv) {
	printf("start num is %d\n", num); num++;
	int rc = fork();
	if(rc>0)
	{
		num+=2;
		printf("Parent's num is %d\n", num);
	}
	else if(rc==0)
	{
		num+=3;
		printf("Child's num is %d\n", num);
	}
	num+=4;
	printf("end num is %d\n", num);
}
```

What is the output of this program?

Let’s do an exercise

```
int num = 3;

int main(int argc, char**argv) {
    printf("start num is %d\n", num);
    num++;
    int rc = fork();
    if(rc>0)
    {
        num+=2;
        printf("Parent's num is %d\n", num);
    }
    else if(rc==0)
    {
        num+=3;
        printf("Child's num is %d\n", num);
    }
    num+=4;
    printf("end num is %d\n", num);
}
```

One process (num = 3):
Output “start num is 3”

Let’s do an exercise

```
int num = 3;

int main(int argc, char**argv) {
	printf("start num is %d\n", num);
	num++;
	int rc = fork();
	if(rc>0)
	{
		num+=2;
		printf("Parent's num is %d\n", num);
	}
	else if(rc==0)
	{
		num+=3;
		printf("Child's num is %d\n", num);
	}
	num+=4;
	printf("end num is %d\n", num);
}
```

One process (num = 3):
Output “start num is 3”

Parent process (num = 4)   Child process (num = 4)

Let’s do an exercise

```
int num = 3;

int main(int argc, char**argv) {
	printf("start num is %d\n", num);
	num++;
	int rc = fork();
	if(rc>0)
	{
		num+=2;
		printf("Parent's num is %d\n", num);
	}
	else if(rc==0)
	{
		num+=3;
		printf("Child's num is %d\n", num);
	}
	num+=4;
	printf("end num is %d\n", num);
}
```

One process (num = 3):
Output “start num is 3”

Parent process (num = 4)

Child process (num = 4)

Parent process (num =6)

Let’s do an exercise

```
int num = 3;

int main(int argc, char**argv) {
        printf("start num is %d\n", num);
        num++;
        int rc = fork();
        if(rc>0)
        {
                num+=2;
                printf("Parent's num is %d\n", num);
        }
        else if(rc==0)
        {
                num+=3;
                printf("Child's num is %d\n", num);
        }
        num+=4;
        printf("end num is %d\n", num);
}
```

One process (num = 3):
Output “start num is 3”

Parent process (num = 4)

Child process (num = 4)

Parent process (num =6)

Parent process output
“Parent’s num is 6”

Let’s do an exercise

```
int num = 3;

int main(int argc, char**argv) {
        printf("start num is %d\n", num);
        num++;
        int rc = fork();
        if(rc>0)
        {
                num+=2;
                printf("Parent's num is %d\n", num);
        }
        else if(rc==0)
        {
                num+=3;
                printf("Child's num is %d\n", num);
        }
        num+=4;
        printf("end num is %d\n", num);
}
```

One process (num = 3):
Output “start num is 3”

Parent process (num = 4)

Child process (num = 4)

Parent process (num =6)

Parent process output
“Parent’s num is 6”

Child process (num = 7)

Let’s do an exercise

```
int num = 3;

int main(int argc, char**argv) {
        printf("start num is %d\n", num);
        num++;
        int rc = fork();
        if(rc>0)
        {
                num+=2;
                printf("Parent's num is %d\n", num);
        }
        else if(rc==0)
        {
                num+=3;
                printf("Child's num is %d\n", num);
        }
        num+=4;
        printf("end num is %d\n", num);
}
```

One process (num = 3):
Output “start num is 3”

Parent process (num = 4)

Child process (num = 4)

Parent process (num =6)

Parent process output “Parent’s num is 6”

Child process (num = 7)

Child process output “Child’s num is 7”

Let’s do an exercise

```
int num = 3;

int main(int argc, char**argv) {
        printf("start num is %d\n", num);
        num++;
        int rc = fork();
        if(rc>0)
        {
                num+=2;
                printf("Parent's num is %d\n", num);
        }
        else if(rc==0)
        {
                num+=3;
                printf("Child's num is %d\n", num);
        }
        num+=4;
        printf("end num is %d\n", num);
}
```

One process (num = 3):
Output “start num is 3”

Parent process (num = 4)

Child process (num = 4)

Parent process (num =6)

Parent process output
“Parent’s num is 6”

Child process (num = 7)

Child process output
“Child’s num is 7”

Parent process (num = 10)

Child process (num = 11)

Let’s do an exercise

```
int num = 3;

int main(int argc, char**argv) {
        printf("start num is %d\n", num);
        num++;
        int rc = fork();
        if(rc>0)
        {
                num+=2;
                printf("Parent's num is %d\n", num);
        }
        else if(rc==0)
        {
                num+=3;
                printf("Child's num is %d\n", num);
        }
        num+=4;
        printf("end num is %d\n", num);
}
```

One process (num = 3):
Output “start num is 3”

Parent process (num = 4)

Child process (num = 4)

Parent process (num =6)

Parent process output
“Parent’s num is 6”

Child process (num = 7)

Child process output
“Child’s num is 7”

Parent process (num = 10)

Child process (num = 11)

Parent process output
“end num is 10”

Child process output
“end num is 11”

Parent and child may execute in any order!

Let’s do an exercise

Parent’s execution (red)

```
int num = 3;

int main(int argc, char**argv) {
	printf("start num is %d\n", num);
	num++;
	int rc = fork();
	if(rc>0)
	{
		num+=2;
		printf("Parent's num is %d\n", num);
	}
	else if(rc==0)
	{
		num+=3;
		printf("Child's num is %d\n", num);
	}
	num+=4;
	printf("end num is %d\n", num);
}
```

Child’s execution (blue)

```
int num = 3;

int main(int argc, char**argv) {
	printf("start num is %d\n", num);
	num++;
	int rc = fork();
	if(rc>0)
	{
		num+=2;
		printf("Parent's num is %d\n", num);
	}
	else if(rc==0)
	{
		num+=3;
		printf("Child's num is %d\n", num);
	}
	num+=4;
	printf("end num is %d\n", num);
}
```

Process Creation: *fork()*, and *wait()*

- Do you notice any difference? Yes, there is a wait() system call
- So what is the result?

```
prompt> ./p2
hello (pid:29266)
child (pid:29267)
parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

```
int main(int argc, char *argv[]) {
  printf("hello (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("child (pid:%d)\n", (int) getpid());
  } else {               // parent goes down this path
    int rc_wait = wait(NULL);
    printf("parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

- Variation: waitpid, waitid, …

Process Creation: *fork()*, *wait()*, and *exec()*

- A final and important piece is the *exec()* system call.

→Used when you want to run a program that is different from the calling program. (Unlike *fork()*, *exec()* will load a different program)

- The new process will replace the current process that is calling exec().
  - This means statements **after** *exec()* call **will NOT be executed**
  - So don't put any statements after *exec()*.
- For example: calling *fork()* in previous example is only useful if you want to keep the running copies of the same program. However, often you want to run a different program --> *exec()* will do that.

Process Creation: *fork()*, *wait()*, and *exec()*

- In this example, the child process calls *execvp()* in order to run the program **wc**, which is the word counting program. In fact, it runs wc on the source file , thus telling us how many lines, words, and bytes are

```
prompt> ./p3
hello (pid:29383)
child (pid:29384)
      29     107    1030 p3.c
parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

```
int main(int argc, char *argv[]) {
  printf("hello (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("child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc"
    myargs[1] = strdup("p3.c"); // arg: input file
    myargs[2] = NULL;            // mark end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn’t print out");
  } else {                // parent goes down this path
    int rc_wait = wait(NULL);
    printf("parent of %d (rc_wait:%d) (pid:%d)\n",
            rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Process Creation: *fork()*, *wait()*, and *exec()*

- It is confusing, is it? Why is it designed in this way? Why not use a single call to do the job?

- We need to duplicate the process and then let another process replace it. Is it a waste of time to copy paste?

Process Creation: *fork()*, *wait()*, and *exec()*

- Why separating *fork()* and *exec()*?
- It has benefits:
  - *Fork()* can be used independently: If we need multiple processes from the same program, then fork can do the job. (i.e. a web server can create a new process for each client)
  - Separating *fork()* and *exec()* allows additional control: Redirect output to a file
    - wc p4.c > p4.output

```
} else if (rc == 0) { // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRU

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");    // program: "wc" (word count)
    myargs[1] = strdup("p4.c");  // argument: file to count
    myargs[2] = NULL;            // marks end of array
    execvp(myargs[0], myargs);   // runs word count
} else {                         // parent goes down this path (main)
    int wc = wait(NULL);
}
```

Executing these two lines in child before exec allows us to redirect output.

Reduce copy overhead by copy-on-write

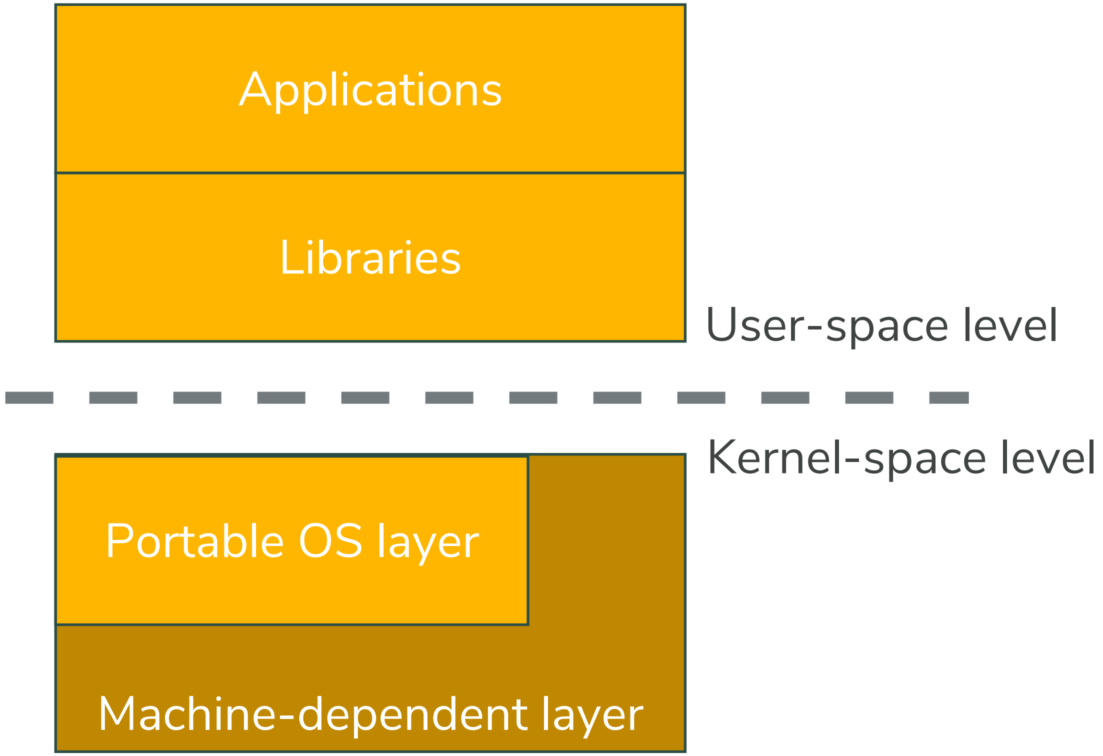
- Basic idea:
  - Don’t copy memory when a process is forked
  - Only copy memory when either process is modifying memory
  - And only copy the part that is modified

- If calling exec() right after a fork(), almost nothing will be copied.

- We will learn more details later in memory management.

Process Memory Layout

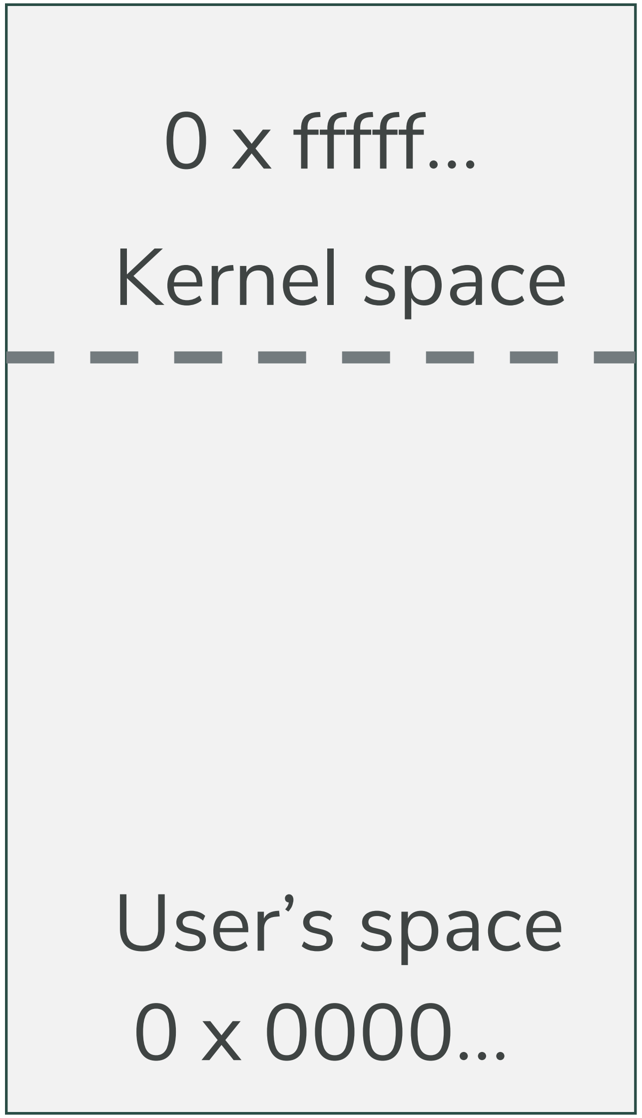
- Remember this UNIX structure?
  - We need memory to store “user-space” and “kernel-space”
  - How can we arrange the storing?



Address Space

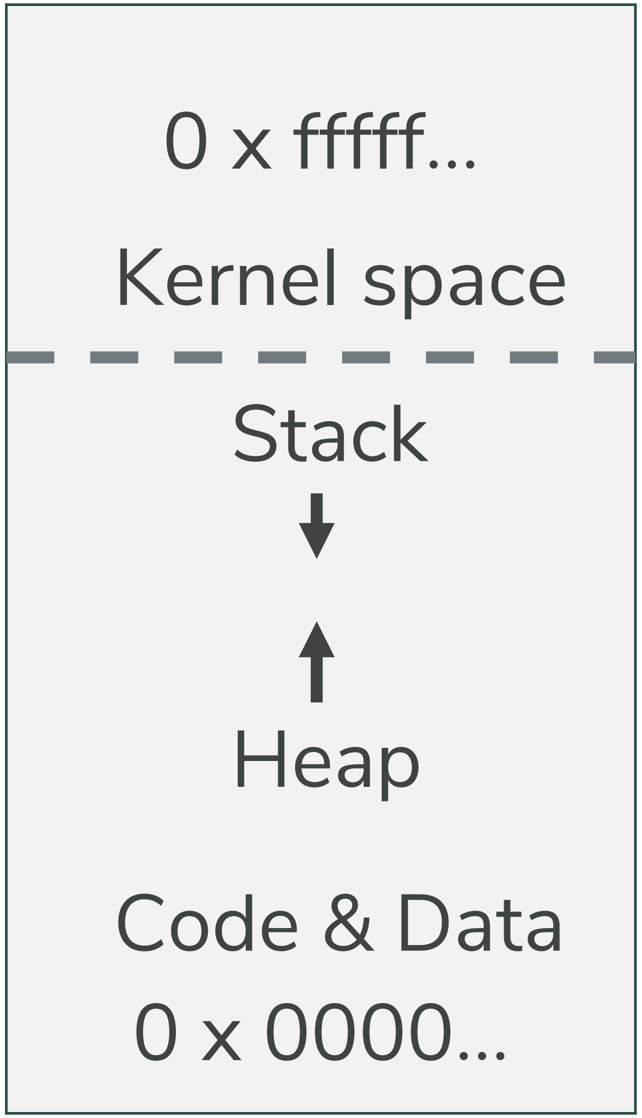
- One (common) approach:
  - Kernel is high memory
  - User is low memory

- Program segments:
  - Stack
  - Data
  - Text
  - Heap…



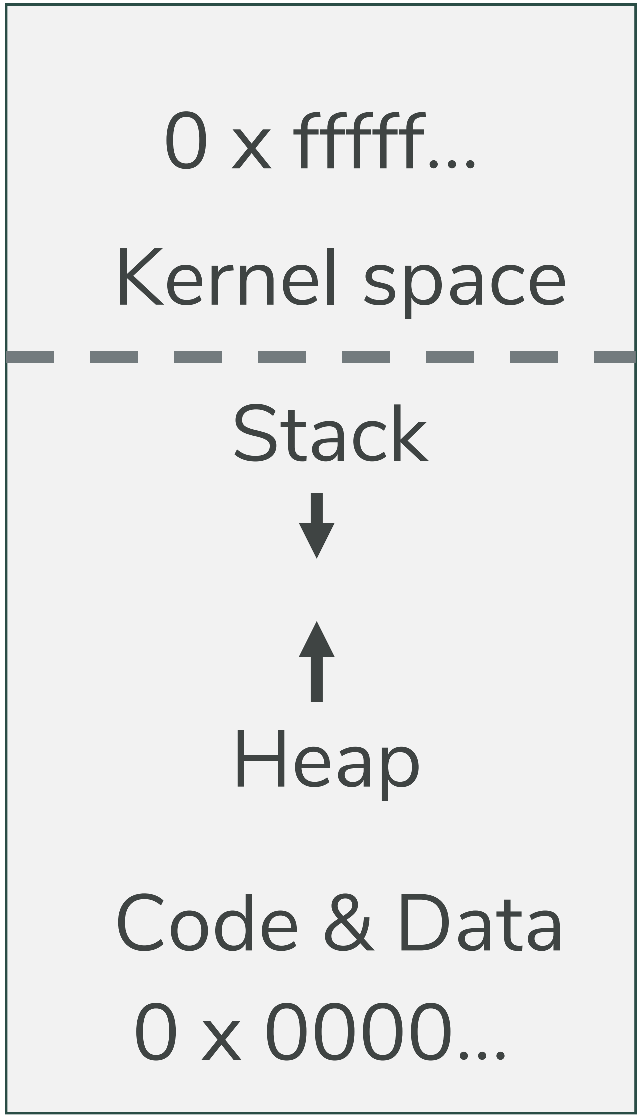
Address Space

- One (common) approach:
  - Kernel is high memory
  - User is low memory
- Program segments:
  - Stack
  - Data
  - Text
  - Heap…



Address Space

- Lots of flexibility
  - Allows stack growth
  - Allows heap growth
  - No pre-determined division.
  - Heap…



Cross-process communication (optional)

- Option 1: Share memory
- By default, process don’t share memory
- System calls like mmap can create shared memory space among processes
- But be careful about all the correctness issues that can be caused by memory sharing: more details in Concurrency Chapter

Cross-process communication (optional)

- Option 2: Named Pipe

- What is a pipe? It’s a UNIX mechanism that allows one process to write data to another process

- Named pipe: It has a well-known name (much like a filename) so that different processes can access it

Cross-process communication (optional)

- Option 3: Socket
- Key idea: Let processes communicate through networking
  - Socket is standard UNIX API to perform network communication
- Benefit: Processes can communicate even when they are not on the same machine

Cross-process communication (optional)

- Option 4: File

- How to know a file has been changed?
  - Periodic scanning is fine, but usually is not efficient.
  - In Linux, it provides system calls like inotify to monitor file changes.

Summary: Process (part 1)

- Process definition
- Process States; PCB
- Process Creation
- Process Memory layout
- Inter-process communication

- Next: Process Scheduling, Context Switch, and more on Inter-Process Communication