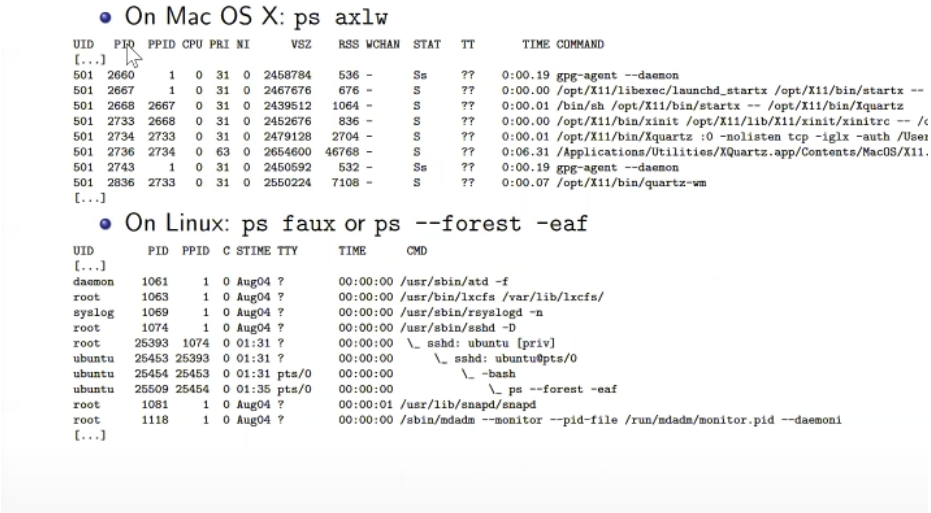
Week 4 Notes (Resume video at: --: 45: 19)

- Other Lifecycles:
    - Recall: one reason to study OSes is that their abstraction/ideas/designs/techniques are relevant to many software projects, i.e., Android Lifecycle
        - Each app on Android goes through the same cycle
- A process can create processes
- If process A creates process B, “A is the **parent** of B”, “B is the **child** of A”
    - A process can have at most one parent
    - A process can have many children
    - Event handlers are responsible for the creation of processes
    - At least one process needs to be running
- Each process has a **PID** (Process ID, integer)
    - Picked by OS and is increasing
    - i.e., if process created is PID 456, next process PID will be 457
    - If PID is 2042, 2042 processes have been created since system boot
    - C POSIX function “getpid()” returns PID
- Bottom line: Processes form a genealogy tree.
-



- Process Creation
    - After creating a child, the parent continues executing
        - Child process cannot run without its parent
    - But at any point, event right away, it can wait for the child’s completion
    - Child can be:
        - A complete clone of the parent, or rather an exact copy of the address space
        - Be an entirely new program
    - POSIX Standard:
        - UNIX (Linux)
        - Darwin (MacOS, iOS, tvOS, watchOS)
- Fork() system call

- A system call that creates a new process
  - A really thin layer over the more complicated “clone” system call
  - “fork”, however, is kept as a system call for backward compatibility reasons
  - “man 2 fork”
- Child is almost an exact copy of parent, except for:
  - PID (two process cannot have the same ID)
  - PPID (parent cannot be its grandparent)
  - Resource utilization (set to 0 as it has just started)
- After call, parent continues executing and the child begins executing
- **The confusing part: “fork()” returns an integer:**
  - **Fork() returns 0 to the child**
  - **Fork() returns the child’s PID to parent**
  - In the case of an error (process table is full), fork() returns -1

-

fork(): The basic example

The basic use of fork()

```
returnedValue = fork();
if (returnedValue < 0) {
    // Manage the error
    print("Error: Can't fork!");
} else if (returnedValue == 0) {
    // Child code
    print("I am the child and my pid is %d", getpid());
    while (1==1); // I just don't want it to terminate
} else {
    // Parent code
    print("I am the parent and the pid of my child is %d",
     returnedValue);
    while (1==1); // I just don't want it to terminate either
}
```

- Simplified version of fork_example1.c
- Note: Errors cases should always be handled (but perhaps not for printf?)

fork(): Second example

What does the following code print?

Second fork() example

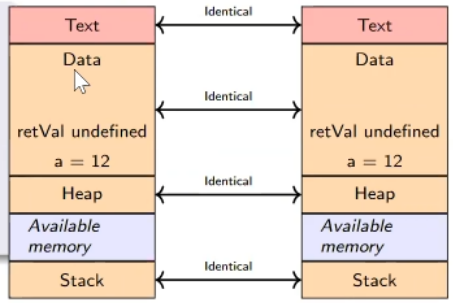
```
int a = 12;
retVal = fork();
if (retVal) {
    // The PARENT (or error)
    sleep(5); // Ask the OS to put me in the WAITING state for 5s
    printf("a = %d", a); // Display the value of a
    while (1); // Loop forever
} else {
    // The CHILD
    a += 3;
    while (1); // Loop forever
}
```

- In this case, you would see the value 12 or 15 on the screen

fork(): Second example

Right after fork() and before the assignment to retVal

```
int a = 12;
retVal = fork();
if (retVal) {
    sleep(5);
    printf("a = %d", a);
    while (1);
} else {
    a += 3;
    while (1);
}
```



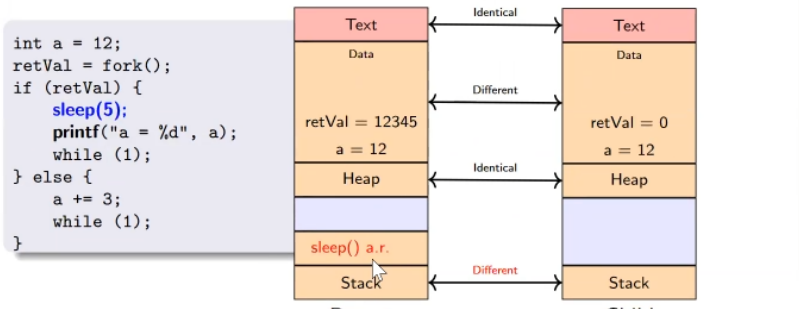
fork(): Second example

There are now two processes executing independently (assuming no error)

In the Parent

First an activation record is created because of the sleep call.

```
int a = 12;
retVal = fork();
if (retVal) {
    sleep(5);
    printf("a = %d", a);
    while (1);
} else {
    a += 3;
    while (1);
}
```



- Both processes coexist independently
  - The code is executed **independently** in the parent and child
  - The segment for the parent has **nothing to do** with the child
  - The stack of the parent has **nothing to do** with the stack of the child
  - The heap of the parent has **nothing to do** with the heap of the child

fork(): Second example tweaked

No infinite loop, print after the if statement

```
int a = 12;
retVal = fork();
if (retVal) {
    sleep(5);
} else {
    a += 3;
}
printf("%d", a);
```

What does this program print? Any ideas?...

- The program prints 1512
-

fork() can be confusing

How many times does this program print "Hello" (source code)?

```
fork();
printf("Hello");
fork();
printf("Hello");
```

Show of hands?

- This will print 6 times
-

fork(): "crazy example" (just for practice)

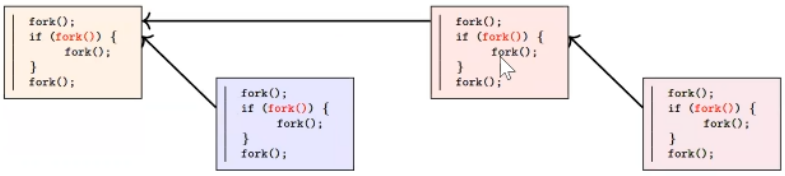
Question: How many processes does this C program create?

```
fork();
if (fork()) {
  fork();
}
fork();
```

- The above code is (likely) not useful
  - Note the typical C coding style: call `fork()` and execute the if clause if the return value is non-zero
- This is just a complex example that you can use to double-check that you truly understand `fork()`
- Let's go through the solution...

fork(): "crazy example"

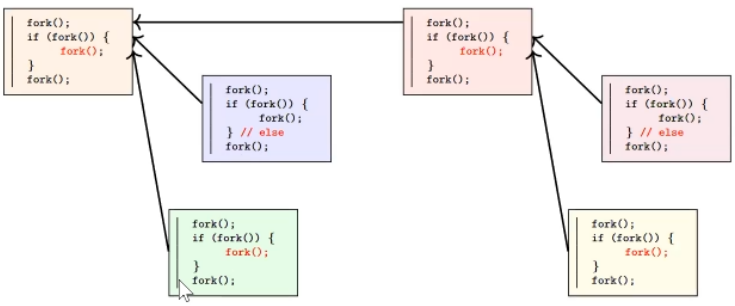
The second fork() (the one in the `if` condition) is executed



We now have **4** processes. Two of them return from the second fork() with non-zero, and two of them with zero

fork(): "crazy example"

The third fork() gets executed by only two processes, thus creating only two new processes:



We now have **6** processes. ALL of them call the last fork(), meaning that we have **12 processes** in total

- Now we can better understand the fork bomb that was run on the VM a week back
- Int main() {
    - While (1) {
        - Fork()
    - };
- }
- Best way to prevent forkbombs from killing system is to limit the number of processes a user can use via "ulimit -u <count>"
    - Calls to "fork()" will fail before process table is full
- The Exec*() family:
    - "man 3 exec:
    - Execl, execlp, execle, execv, execvp, execvpe – execute a file
    - These are all variations of the execve system call and typically one says "exec"
    - The system call **replaces the process image** (such as the process address space) **by that of a specific program** (stored on disk as an executable)
    - Essentially, you give "exec":
        - A path to an executable
        - Arguments to be passed to that executable
        - Possibly a set of environment variables
    - The call to exec() never returns
        - Unless there is an error such as the executable cannot be executed

The exec⋆() Family: Example 1

```
int main(int argc, char *argv[]) {
char* const arguments[] = {"ls", "-l", "/tmp", NULL};
execv("/bin/ls", arguments);
printf("This never gets executed...\n");
return 0;
}
```

The exec⋆() Family: Example 4

To spawn a new process: combine with `fork()`

```
if (fork() == 0) {
    // Child
    char* const arguments[] = {"ls", "-l", "/tmp", NULL};
    execv("bin/ls", arguments);
} else {
    // Parent
    while (1);
}
```

The exec⋆() Family: Examples 2 and 3

```
char* const arguments[] = {"ls", "-l", "/tmp", NULL};
if (execv("Let's try something fun", arguments) == -1) {
    fprintf(stderr, "Error: %s\n", strerror(errno));
}
```

exec_example2.c

```
char* const arguments[] = {"ls", "-l", "/tmp", NULL};
if (execv("/tmp", arguments) == -1) {
    fprintf(stderr, "Error: %s\n", strerror(errno));
}
```

exec_example3.c

They're dead... but alive!

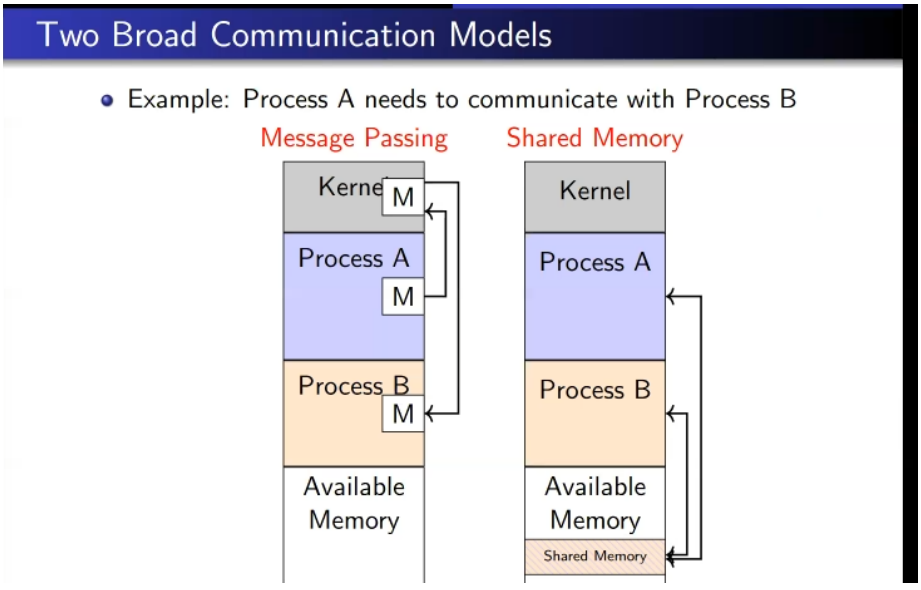
Let's have a look at the processes list while the last example is running...

```
% ps f
  PID   TTY      STAT   TIME   COMMAND
 9129   pts/18   Ss     0:00   bash
10418   pts/18   R+     0:03     \_ ./exec_example4
10419   pts/18   Z+     0:00         \_ [ls] <defunct>
```

Defunct: from Latin *defunctus*, i.e., dead

- When a child process terminates, it remains as a **zombie** in the **terminated** state
    - Recall that in the Process Lifecycle Diagram, we had a terminated state, which some of us might have though was useless
- Rationale: the parent process may want to know about the status of a child process:
    - i.e., Run this first program and when it is finished, run that one…
    - therefore, the parent may want to wait for its childs completion
- The OS keeps zombies around for this purpose

    - Zombies do not use hardware resources
    - But each zombie uses a slot in the process table!
    - The Process Table may thus eventually fill up due to Zombies and thus cause “fork()” to fail
- A zombie lingers on until:
    - **Its parent has acknowledged termination**
    - **Its parent dies**
- A process terminates itself with the “exit()” system call, which takes as argument an integer called the process **exit/return/error value/code**
- All resources of the process are then deallocated by the OS (i.e., memory, open files, I/O buffers, etc.)
- Note that the termination of a process can terminate another process
- This is done using **signals** and the “kill()” system call
- Communicating Processes?
    - So far we have seen **independent** processes
        - Each process runs code independently
        - Parents are aware of their children, and children are aware of their parents. They do not interact, however aside from the ability to wait for a process to terminate
    - We often need processes to **cooperate** however:
        - To share information such as access to common data
        - To speed up computation such as using multiple cores
        - For the sake of convenience such as some applications are naturally implemented as sets of interacting process



Message Passing	Shared Memory
☹ Requires data copies (user space → kernel space → user space)	☺ Fast because no memory copy is needed
☹ High overhead because one system call per operation (`send` and `receive`)	☺ Low overhead because system calls only to set up the shared memory region (after that it's just memory reads and writes)
☹ Cumbersome for developers: code will be sprinkled with send and receive everywhere	☺ Easy for developers: Just read/write to shared memory region
☺ Preserves memory protection / isolation across processes	☹ Violates the principle of memory protection between processes, which is dangerous / bug-prone (more on this in Synchronization module)

- **All OSes** provide system calls for Shared Memory
- **All OSes** provide system calls for Message passing
- There are many variations in APIs and implementations