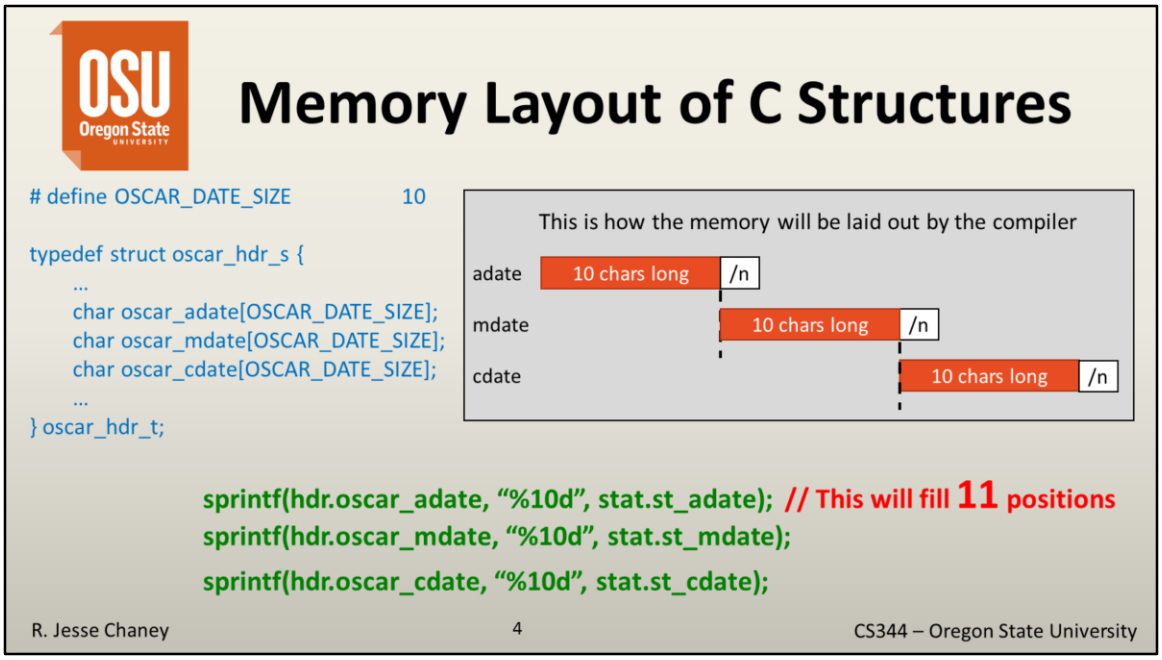
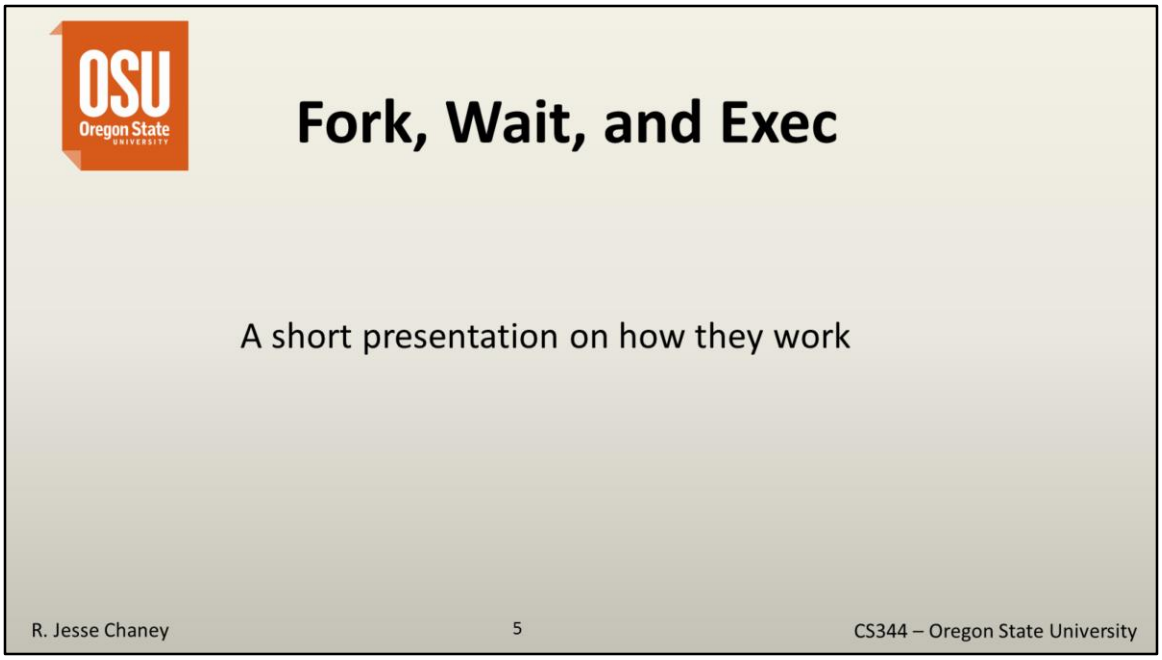


These are character arrays, **NOT** strings. Strings are NULL terminated. These are **NOT** NULL terminated.

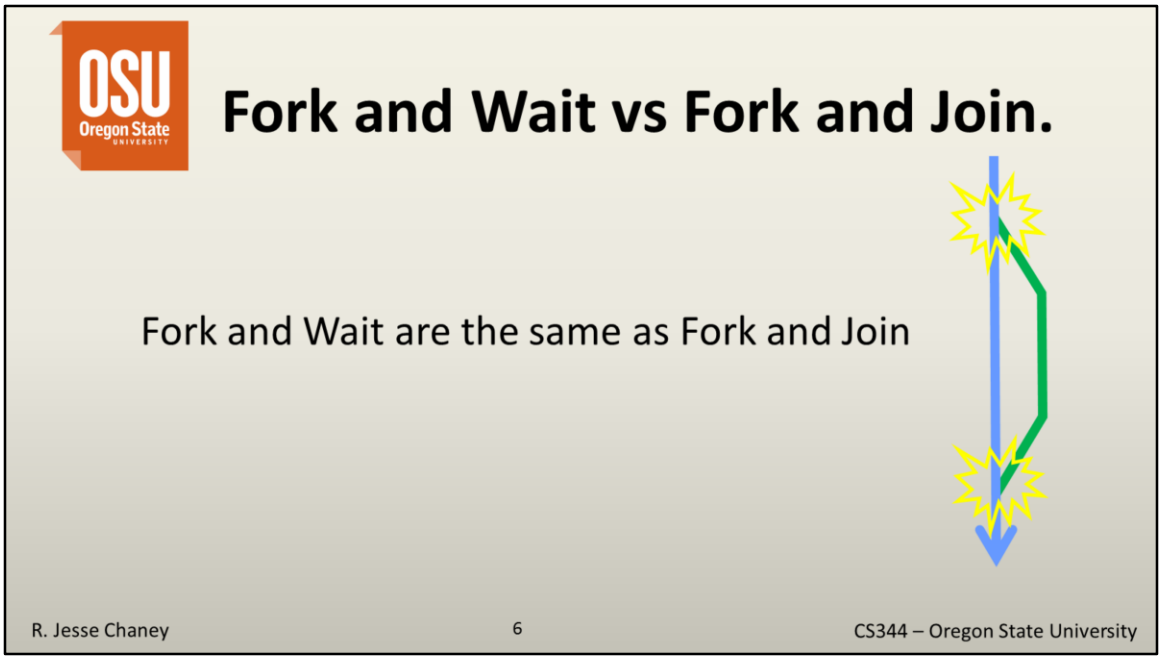
But, what happens if we do a sprintf() into the oscar_adate field in the oscar_hdr_t type (using a field width of 10 characters)? It will put NULL into the 11th position of the field. Of course, there is no 11th position in the field. There are only 10. However, given the way C lays out the memory of the fields in a structure, the 11th position in the oscar_adate field is in fact the initial position in the oscar_mdate field. The same goes for the 11th position in the mdate field.



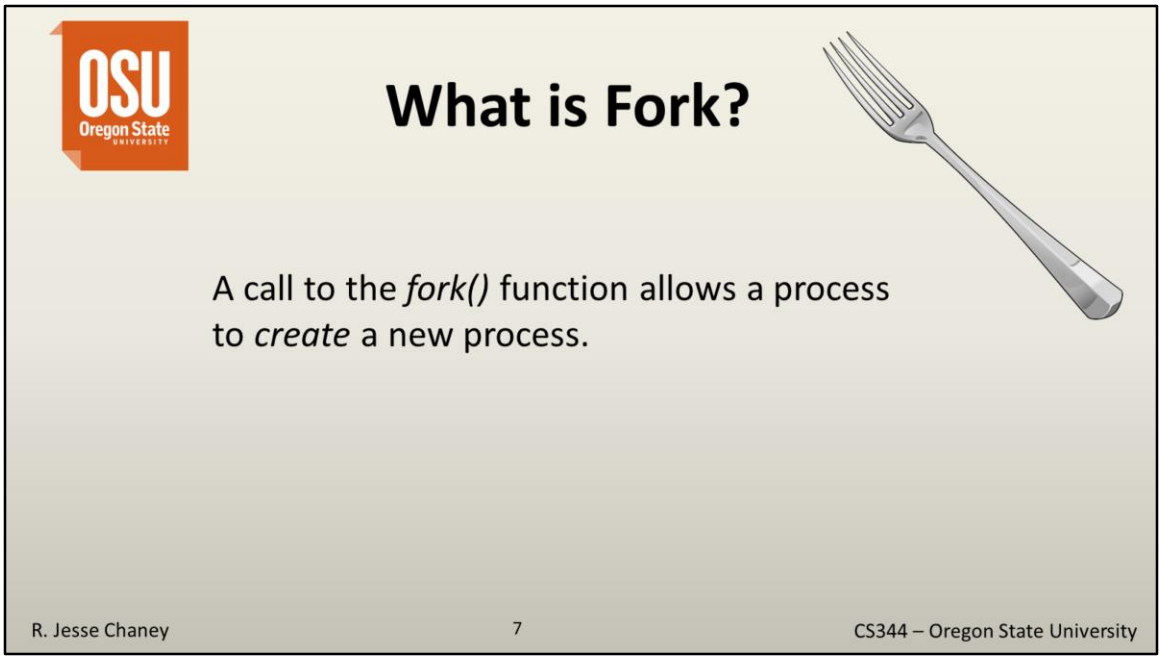
This is a presentation on fork, wait, and exec for CS344.

The fork, wait, and exec system calls are very important in this class, so I wanted to give an additional presentation demonstrating how they work. Most of this will be about fork and wait, but exec likes to tag along.

I'll be going over material from chapter 24 in the TLPI book. It will look a lot like figure 24-1 on page 515. The figure I'll draw here will look a lot like figure 24-1 in the book, but not exactly the same.

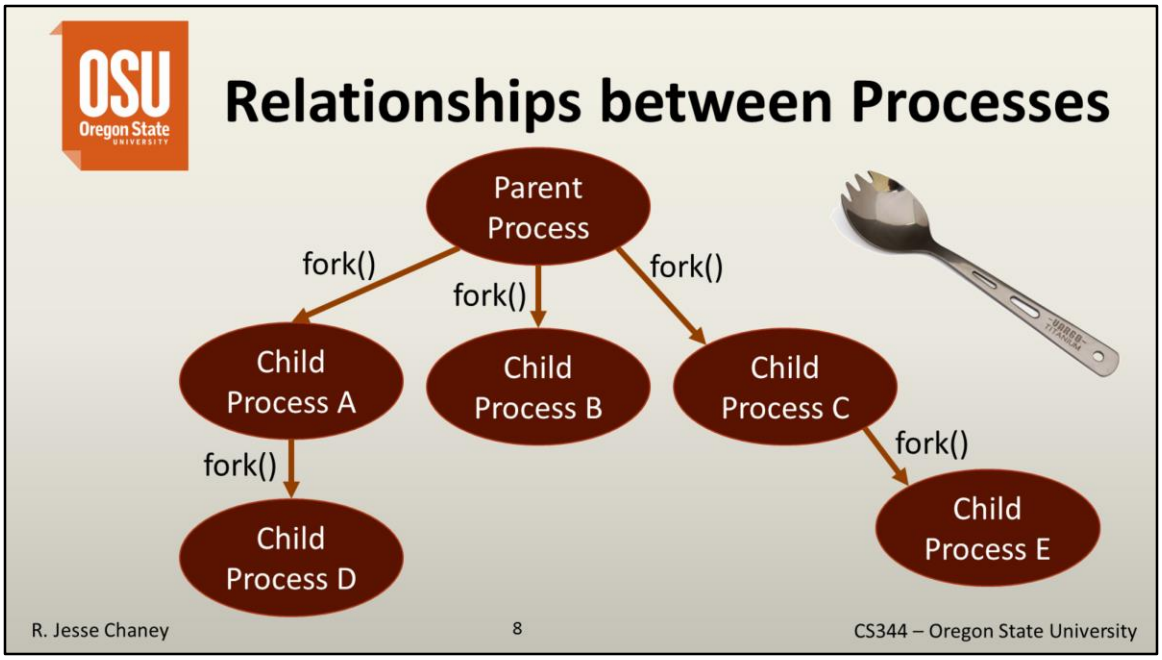


You'll also commonly hear about *fork and join* concepts. These are the same as the fork and wait, just a bit more generically labeled. If you think about a path of execution, the fork is where there become 2 paths. Then the join is there the paths become one again. The join in Linux is accomplished with a wait() call, or one of the wait variants (such as waitpid()).



Processes, processes, processes. We've spent quit a bit of time talking about processes. I'm sure you've been spent sleepless nights wondering how did we get so many processes and how do we create more processes. Well, the answer is *fork().*

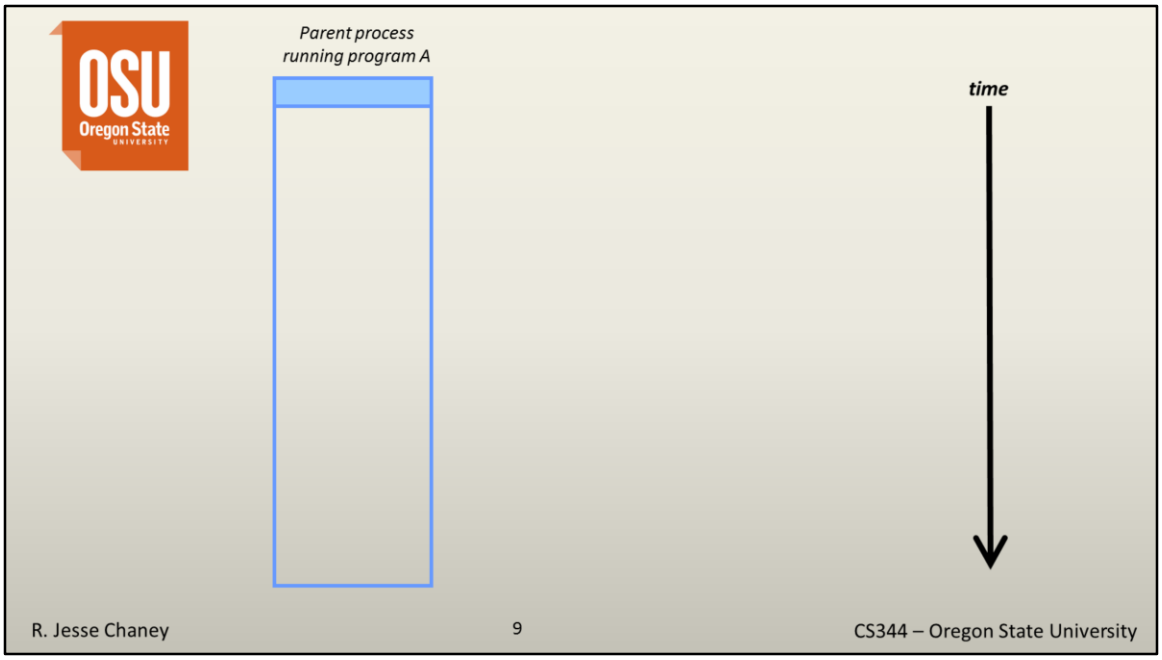
A call to *fork()* creates a new process. The process that calls fork() is called the parent process. The newly created process is called a child process. Specifically, the child process is called a child of the creating process.



The parent/child relationship between processes is an important feature that we'll make use of. You can think of the parent child relationship as looking like a tree. And yes, it is certainly possible that a child process create additional child processes. Looking at the parent child hierarchy above, we can say that process D is related to process A. And, we can say that process E is related to process A. We don't say that process D is a child of process A, but we can say it is a grandchild of A.

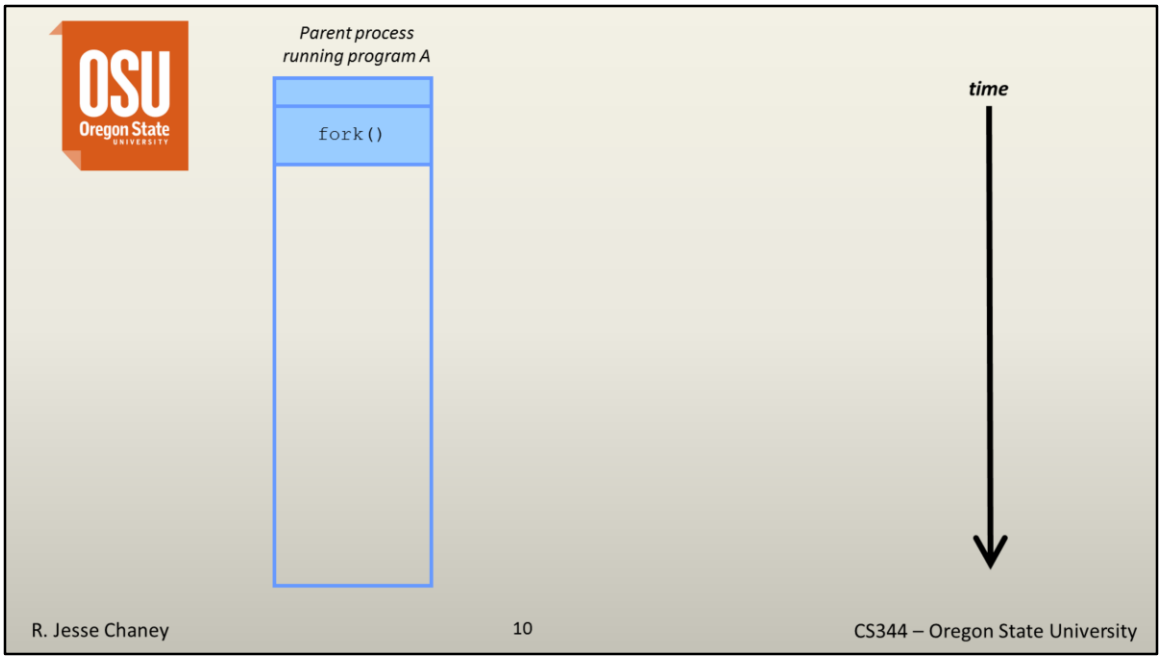
Implicitly, you are all about to become mommas and poppas of new bouncing baby processes.

Just to let you know, you do need to take some care when you go off creating your new offspring. Initial calls to fork() can sometimes get a little over zealous and wind up creating hundreds or thousands of new little processes. You don't want to do that.

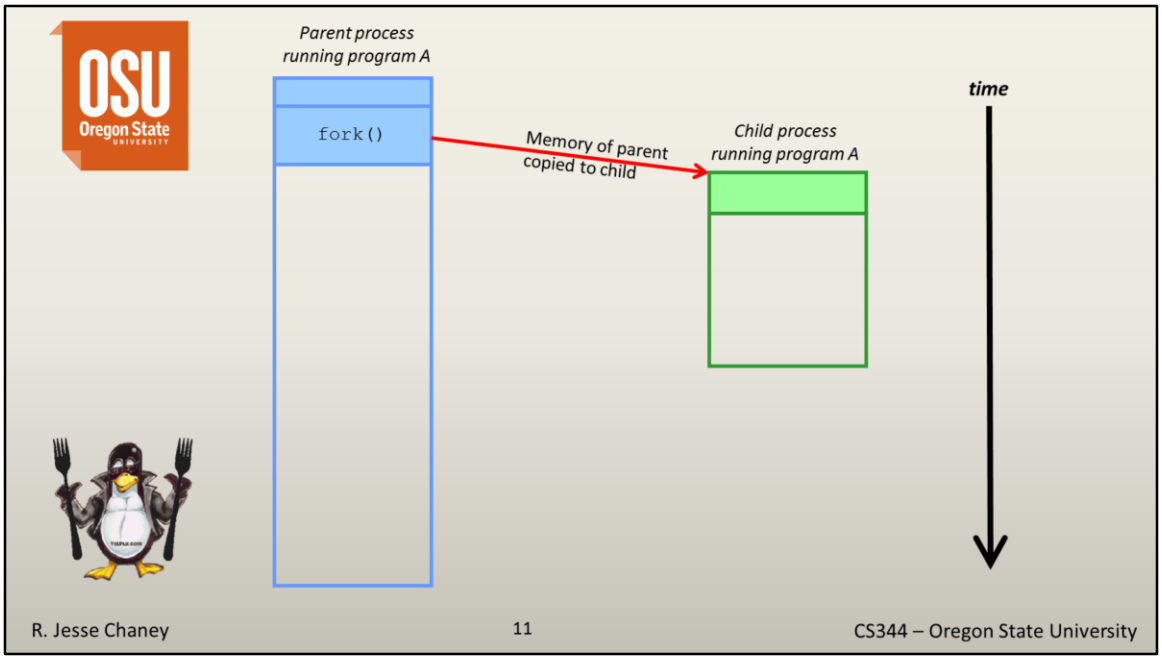


In this presentation, time moves from the top of the slide to the bottom.

We start with a single process, the parent process. The parent process is running program A.



The parent process makes a call to fork(). The fork() call is covered in detail in section 24.2 in your book.



When the parent makes a call to fork(), a new process is created. Once the call to fork() has completed, there are 2 processes, the parent A process and the child A process. At this point, both are running the same program, program A. In both processes, execution continues from the point where fork() returns.

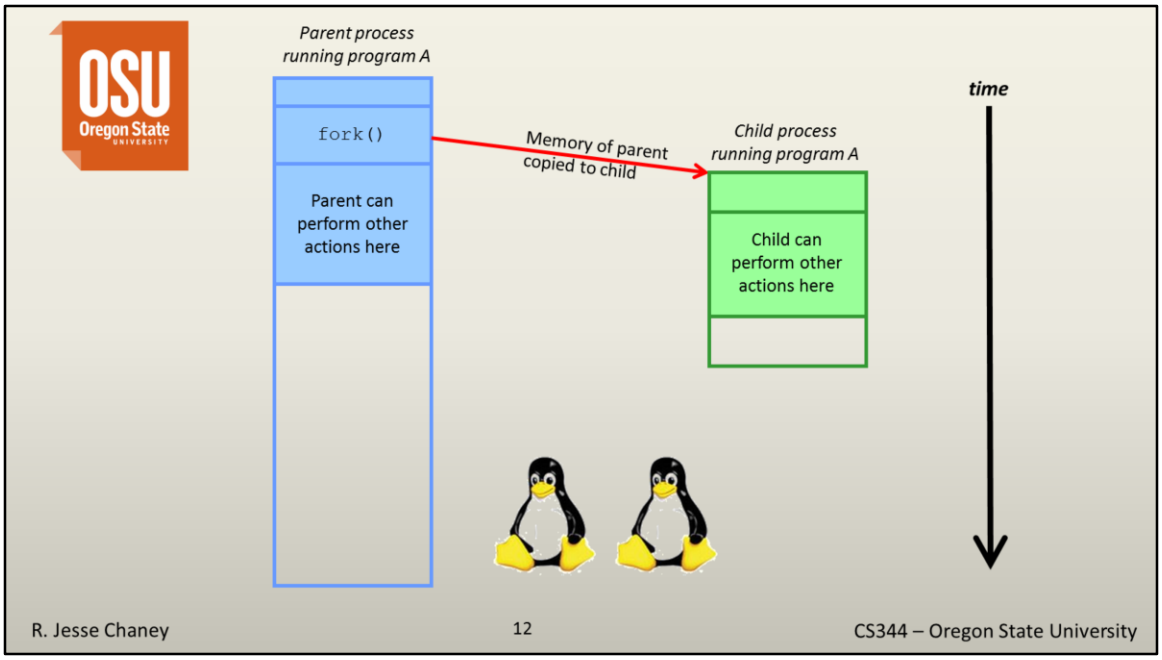
The two processes are sharing the same program text (the instructions), but they have separate **copies** of the stack, data, and heap. Initially, the child's stack, data, and heap segments are exact **duplicates** of the parent's memory segments. After the fork() returns, each process can modify the variables in its own stack, data, and heap segments without affecting the other process.

We don't know which process may run first when the fork() returns. Any reliance on which one runs first either needs to take special care to guarantee it works on multiple kernel releases or is asking for trouble (such as race conditions).

It is important to realize here that there are 2 processes (parent and child), but there is only 1 program A. The program A has 2 instances, the parent and the child.

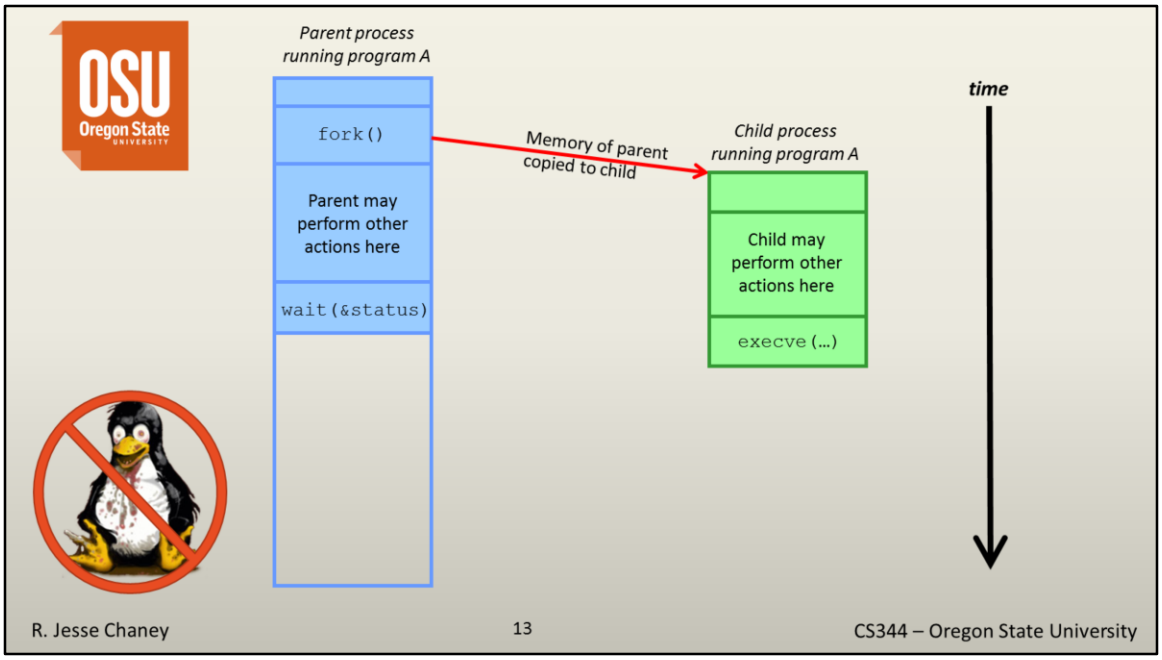
The actual precise memory semantics of what is copied from parent to child and

when, is more complex than I want to describe here. If you want more information on that, go to section 24.2.2 in your book.



Once the call to fork() has returned, each copy of the program A (parent and child processes) runs on its own. Each will be independently scheduled by the CPU. The parent and child process may run at the same time on multi-processor systems.

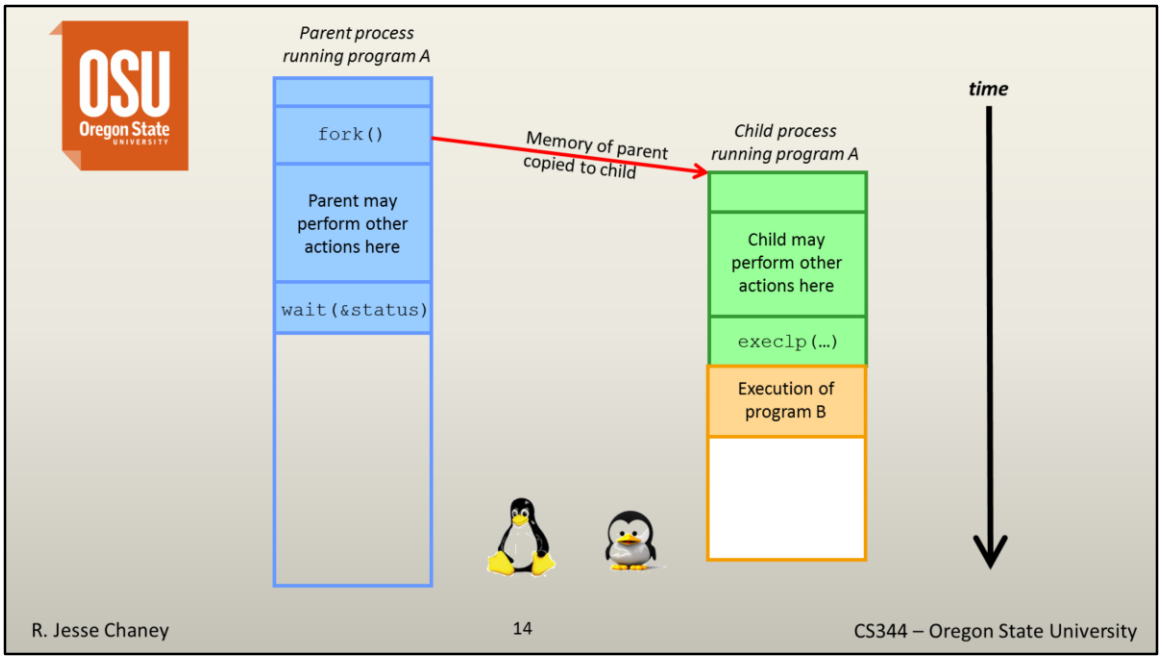
Within the code for program A, the parent and child processes can be distinguished by differing process ids (the PID). The parent and child processes can each take a different path of execution through the code for the program.



For this example, we are going to have the parent and child process do 2 very different things at this point. The parent process calls wait() so that it will wait on the process status for the child. Because the parent uses the wait() call, the parent will block until the call to wait() completes. For our example, this occurs when the child process completes or is terminated. Were the parent to call waitpid(), it could use a non-blocking option.

The child process makes a call to one of the exec functions, execve(). The execve() call loads a new program into the memory allocated for the child process. At the end of the call to execve(), the parent and child are no longer sharing the same text segments (instructions). In the child process, once the execve() call is complete, program A in the **child** no longer exists. Program A still exists for the parent process, but not the child process.

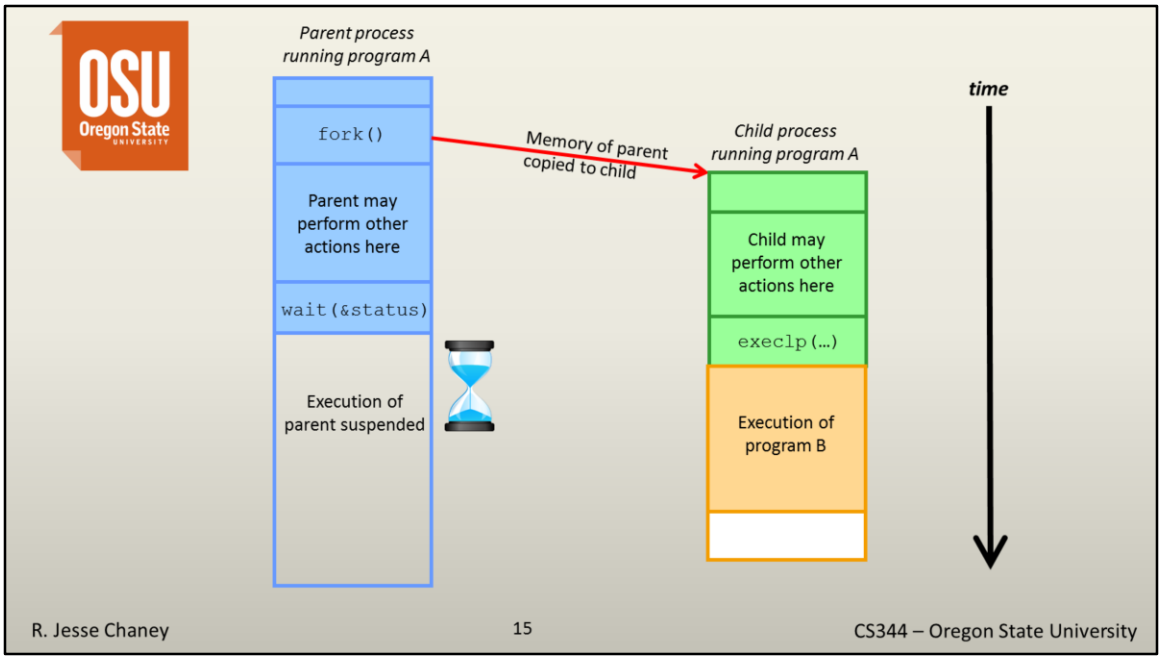
Of course one of the reasons the parent process calls the wait() function is to prevent zombies.



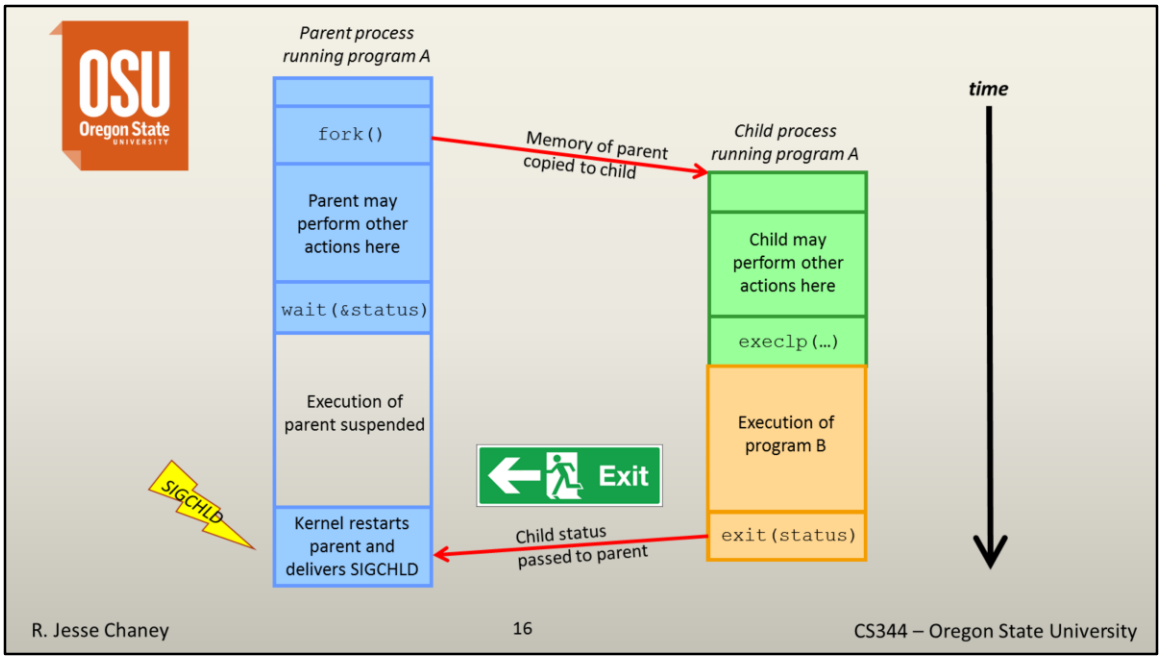
The child process has called execlp() for program B. The PID for the child process remains the same even though the program that is executing as that PID has completely changed, from program A to program B.

The parent process continues to wait on the child PID, even though the program running as that PID has changed from program A to program B.

I'm not going to cover a lot about the execlp() call or any of the other exec calls here. There are a number of them. When you need to know more about them, go to chapter 27 in your book.



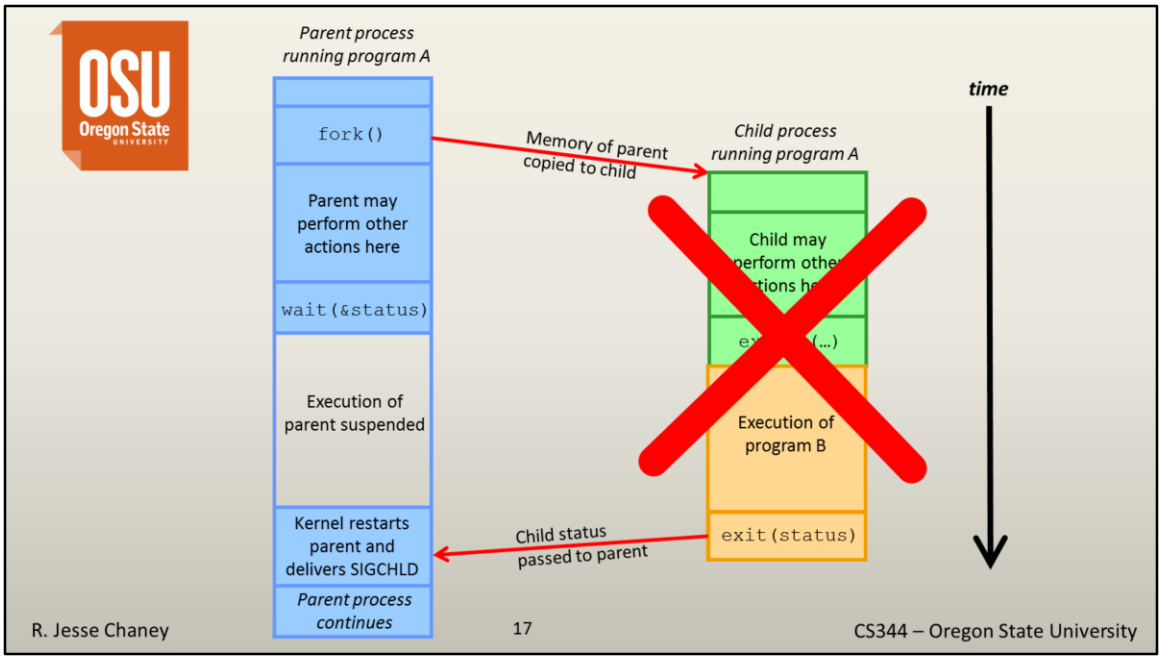
The child process, program B, goes on with its task. The parent process continues to wait. So long as the child PID is running, the parent process will wait, patiently.



The child process, program B, finally completes. When it calls exit(), its status value is passed back to the parent process. The parent process has been patiently waiting on the completion of the child process. The parent process will also receive a SIGCHLD signal (which may be handled or ignored).

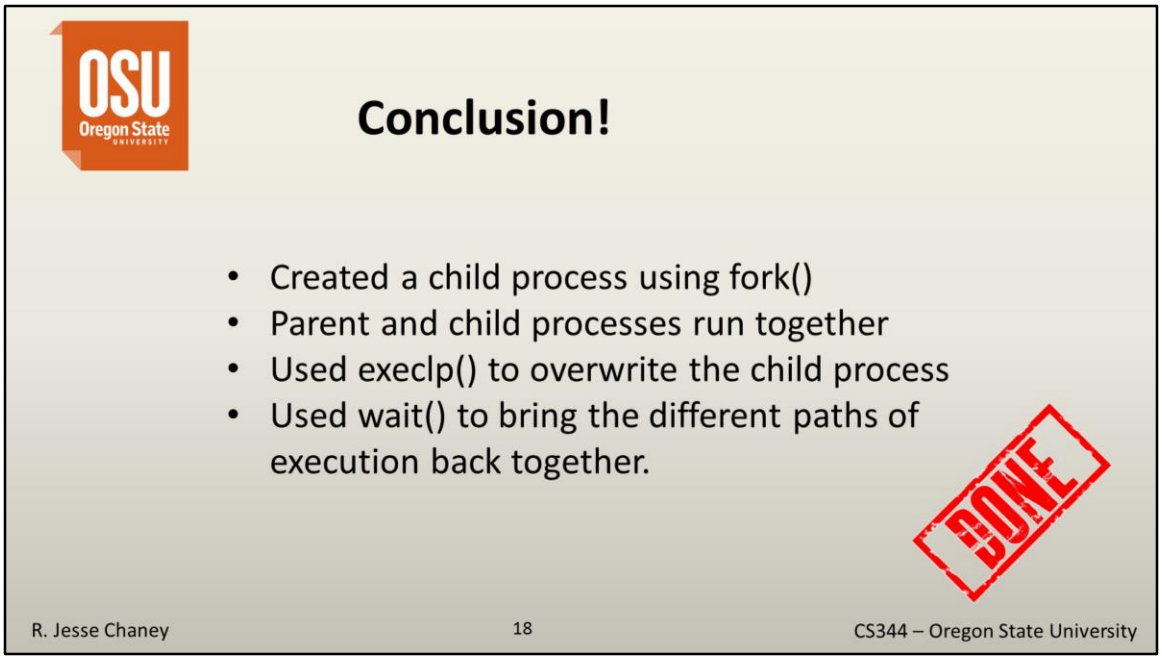
The parent process does not necessarily know that the child process was running program A or program B, just that the PID on which it was waiting completed.

There are some other status changes that can occur for a process that are not termination. Specifically, if a process has been paused using a ctrl-Z or if it is running in a debuger. Chapter 26 goes into those details.



At this point in time, there is no more child process. The parent process continues on its way.

It is possible that the PID that represented the child process will be reused (in fact it's likely), but it will not be the *same* process.



The semantics of how fork, wait, and exec work may seem a little awkward or scary at first, but they are actually pretty simple.

- In this presentation, we've seen how you create a new child process using the fork() call.
- We've seen how the parent and child process can be executing the same program, but with different paths of execution.
- We've seen how you can call one of the exec functions to overwrite one process with the code for a different program.
- We've see how you can use the wait call to have a parent process pause until a child process completes.

Thank you and have a good day.