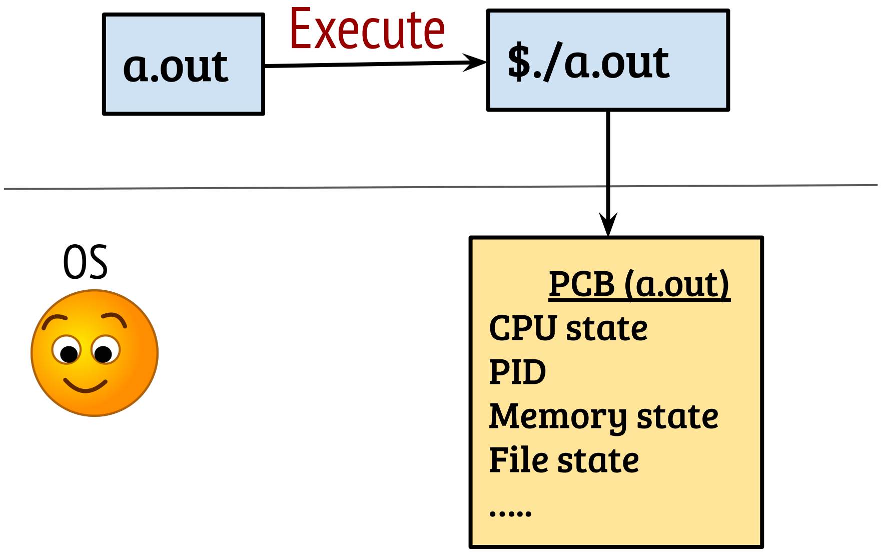
CS330: Operating Systems

Process API: System calls

Recap: The process abstraction

- The OS creates a process when we run an executable



a.out

Execute

$./a.out

OS

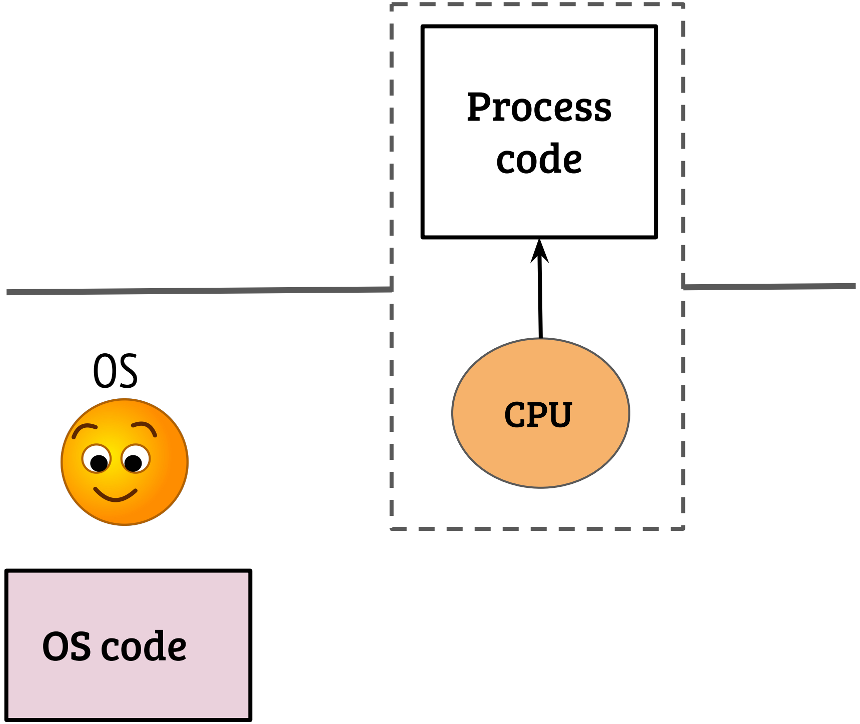
PCB (a.out)
CPU state
PID
Memory state
File state
.....

- When we execute "a.out" on a shell a process control block (PCB) is created
- Does it raise some questions related to the exact working?

Process creation: What and How?

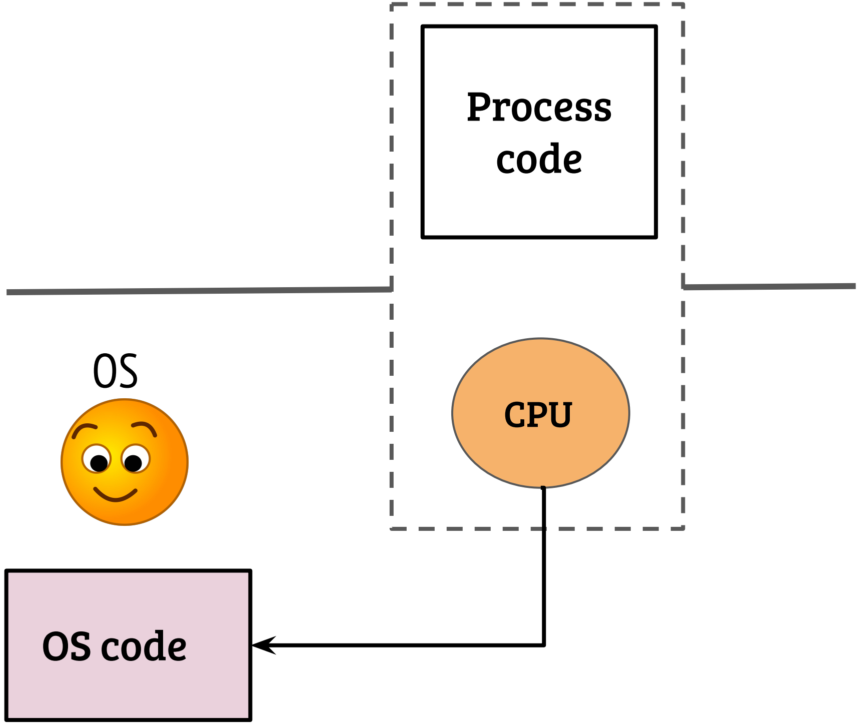
- How does OS come into action after typing "./a.out" in a shell?

System call



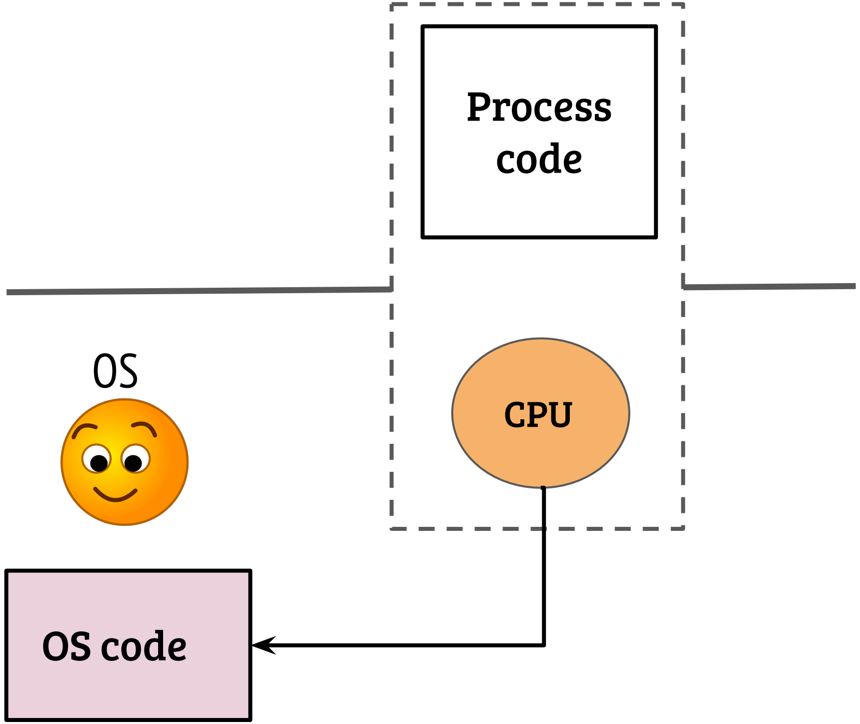
- CPU executing *user code* can invoke the *OS functions* using system calls

System call



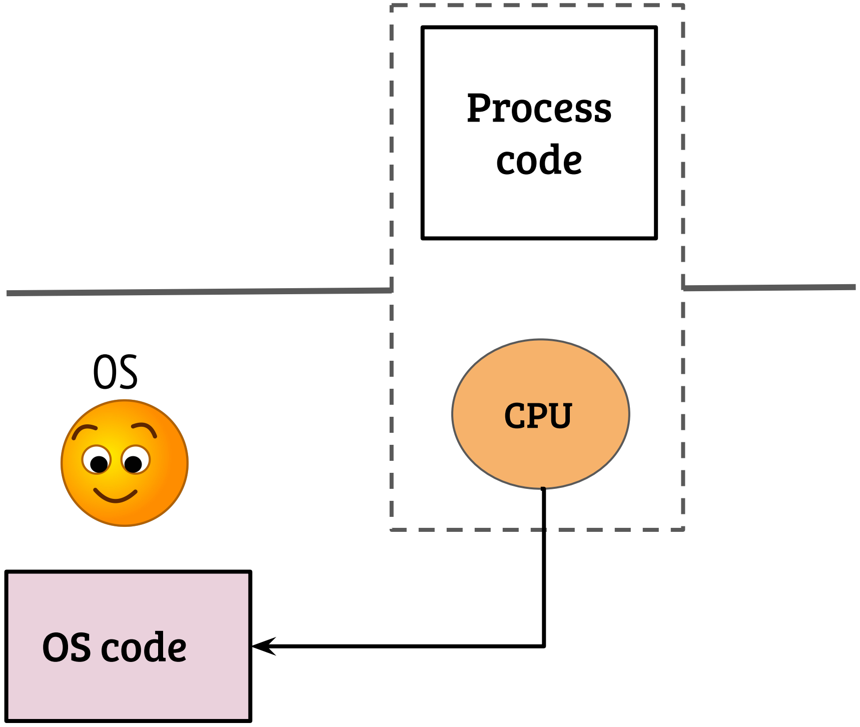
- CPU executing *user code* can invoke the *OS functions* using system calls
- The CPU executes the OS handler for the system call

System call



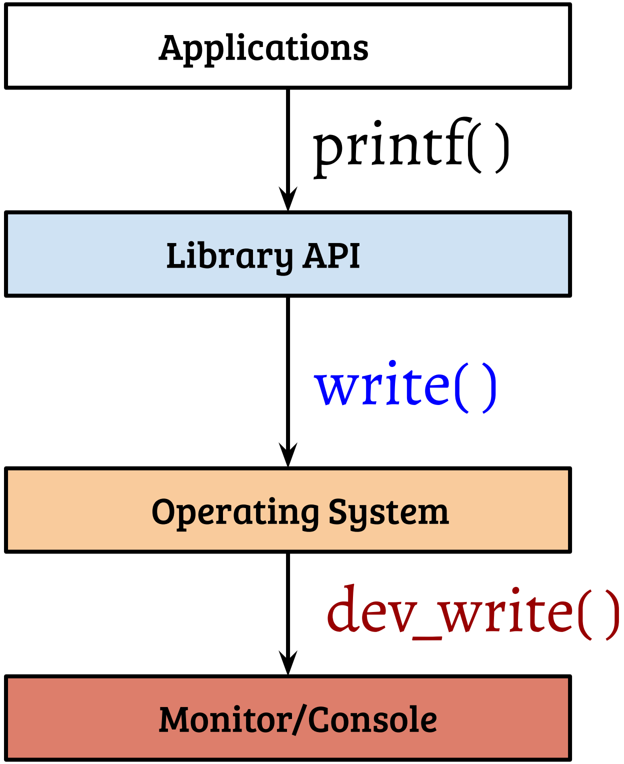
- CPU executing *user code* can invoke the *OS functions* using system calls
- The CPU executes the OS handler for the system call
- How system call is different from a function call?

System call



- CPU executing *user code* can invoke the *OS functions* using system calls
- The CPU executes the OS handler for the system call
- How is system call different from a function call?
- Can be thought as an invocation of privileged functions (will revisit)

System calls and user libraries



- Most system calls are invoked through wrapper library functions
- However, all system calls can be invoked directly
  - For example, in Linux systems, syscall( ) wrapper can be used (Refer: man syscall)

A simple system call: getpid( )

USER

```
main( )
{
  printf(“%d\n”, getpid( ));
}
```

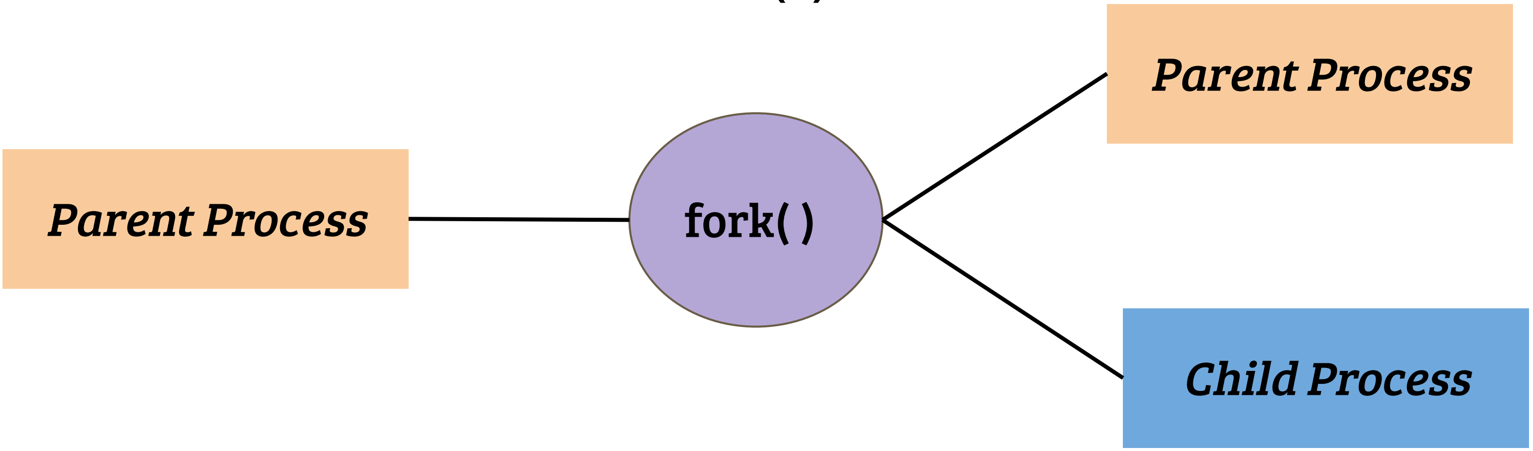
OS

```
pid_t getpid( )
{
     PCB *current = get_current_process( );
     return (current → pid);
}
```

Process creation: What and How?

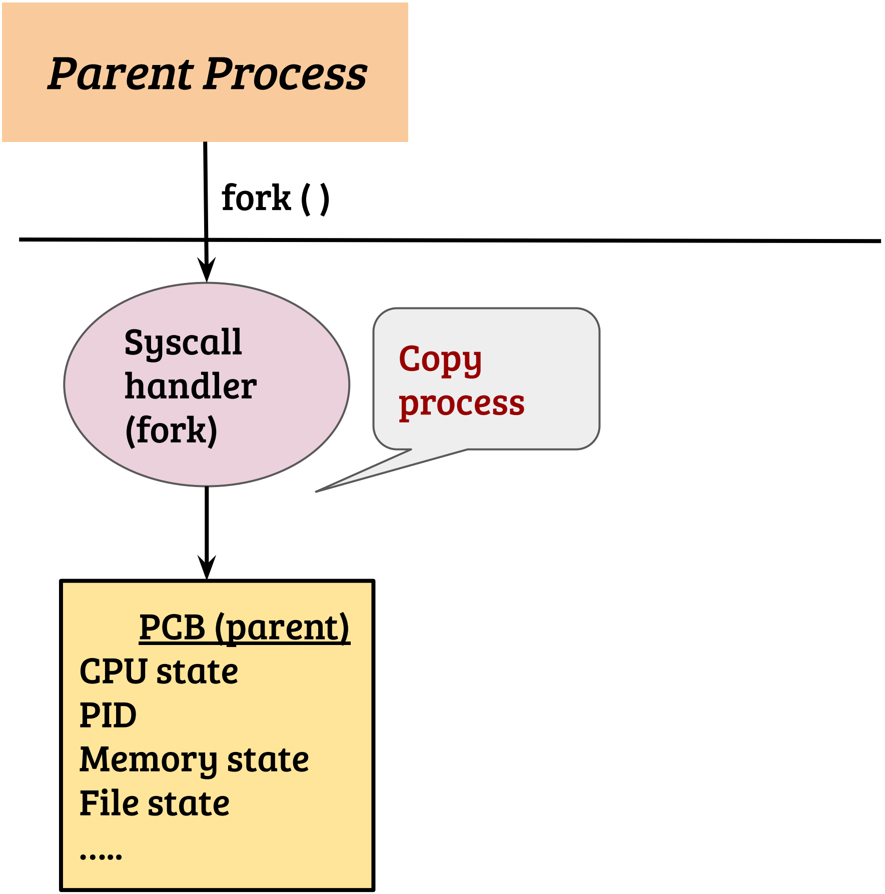
- How does OS come into action after typing “./a.out” in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?

Process creation - fork( )

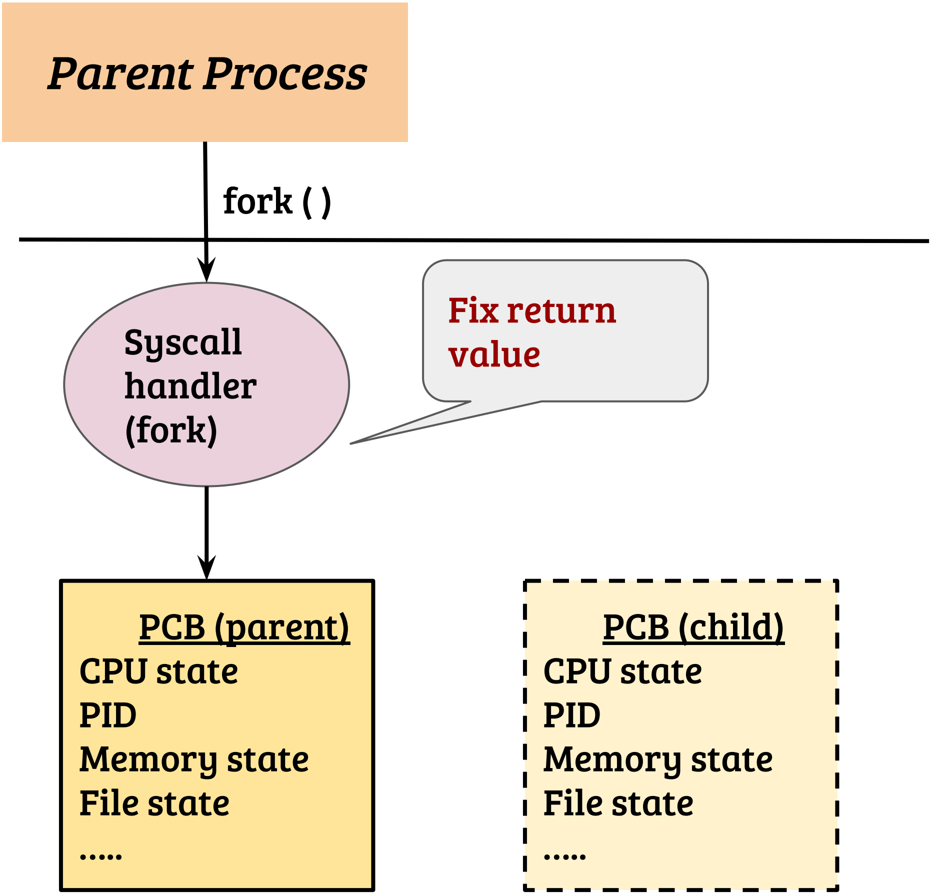


- fork( ) system call is weird; not a typical “privileged” function call
- fork( ) creates a new process; a *duplicate* of calling process
- On success, fork
  - Returns PID of child process to the caller (parent)
  - Returns 0 to the child

Typical implementation of fork

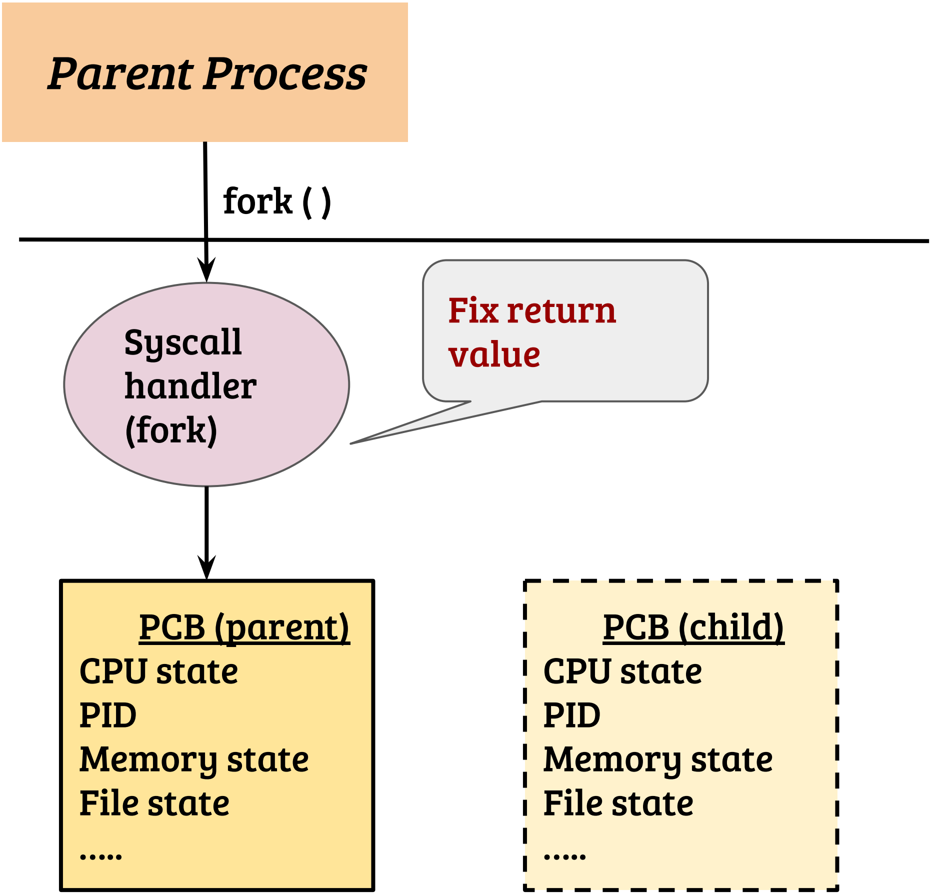


Typical implementation of fork



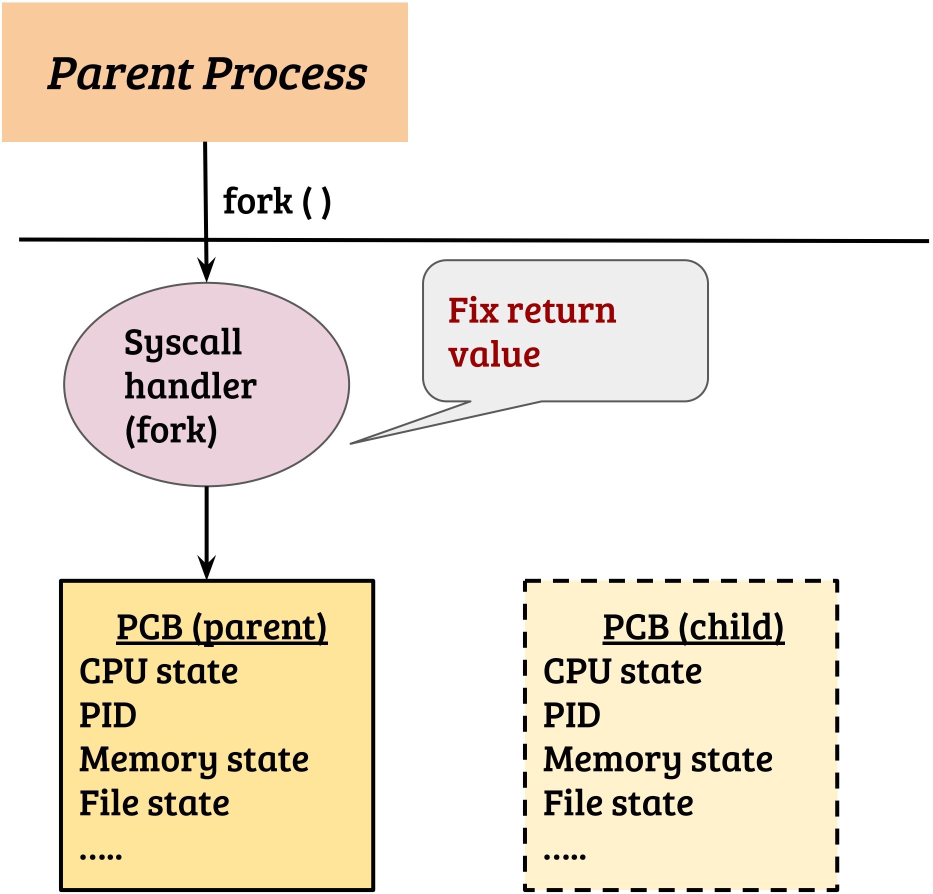
- Child should get ‘0’ and parent gets PID of child as return value. How?

Typical implementation of fork



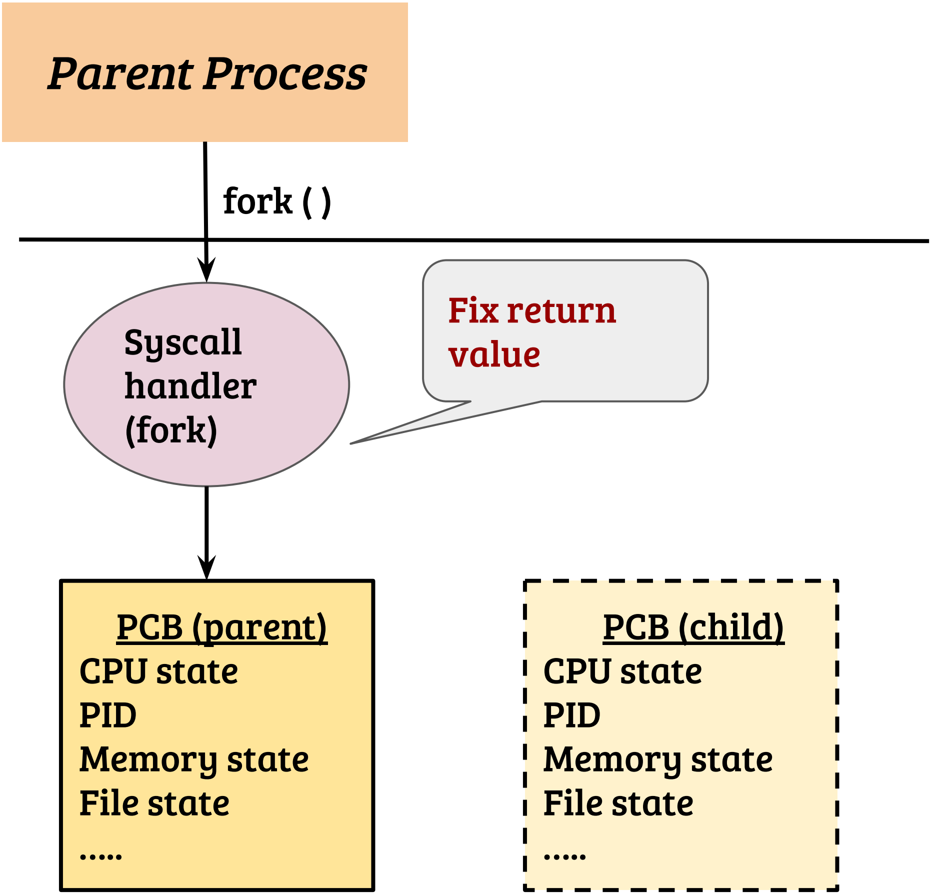
- Child should get ‘0’ and parent gets PID of child as return value. How?
- OS returns different values for parent and child

Typical implementation of fork



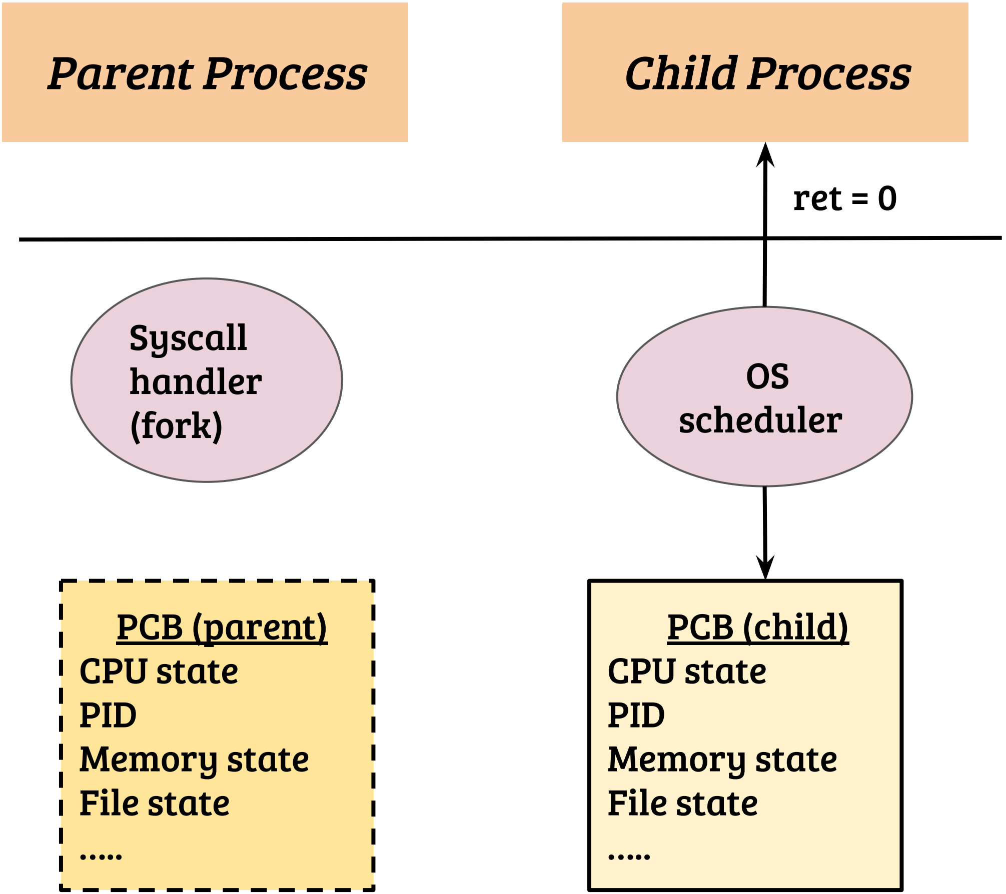
- Child should get ‘0’ and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?

Typical implementation of fork



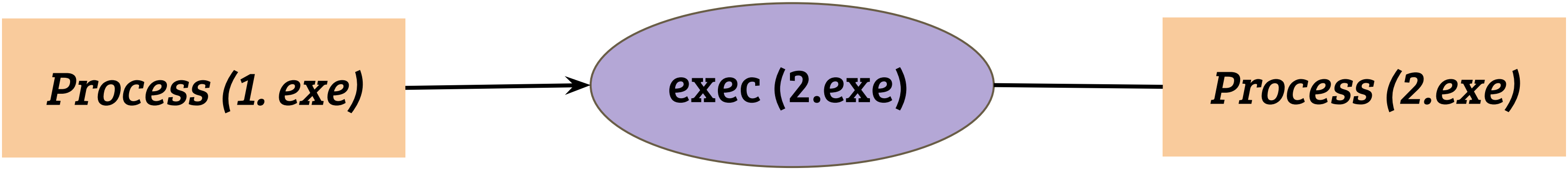
- Child should get '0' and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?
- When OS schedules the child process

Typical implementation of fork



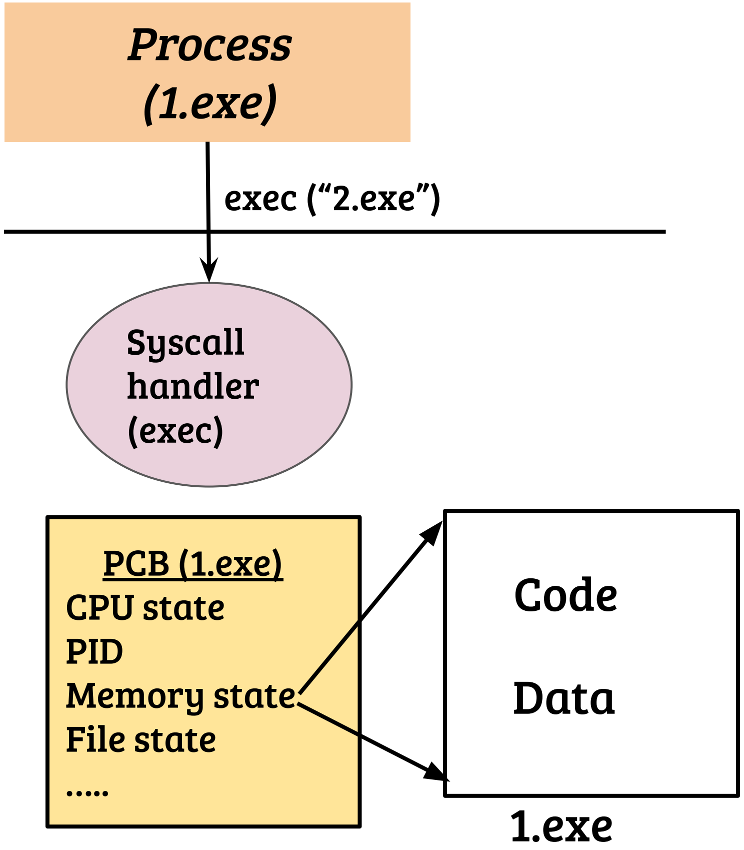
- PC is next instruction after fork( ) syscall, for both parent and child
- Child memory is an exact copy of parent
- Parent and child diverge from this point

Load a new binary - exec( )



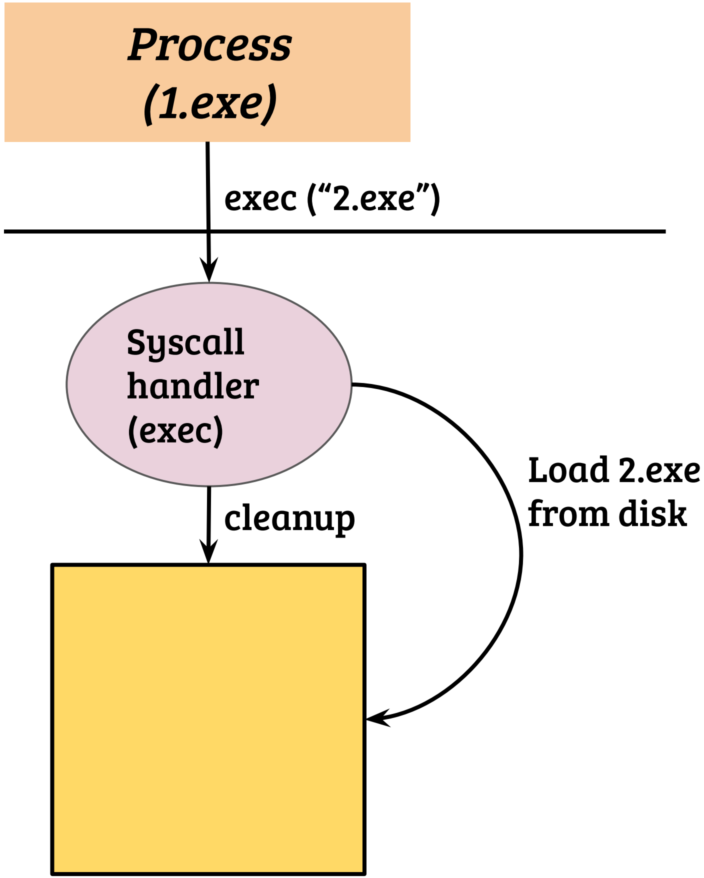
- Replace the calling process by a new executable
    - Code, data etc. are replaced by the new process
    - Usually, open files remain open

Typical implementation of exec



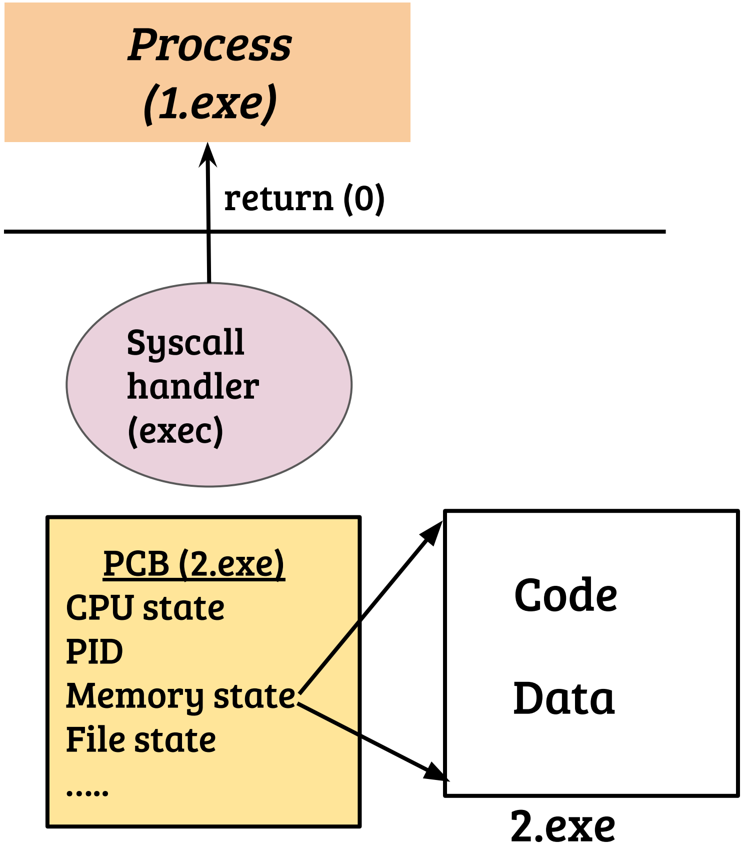
- The calling process commits self destruction! (almost)

Typical implementation of exec



- The calling process commits self destruction! (almost)
- The calling process is cleaned up and replaced by the new executable
- PID remains the same

Typical implementation of exec

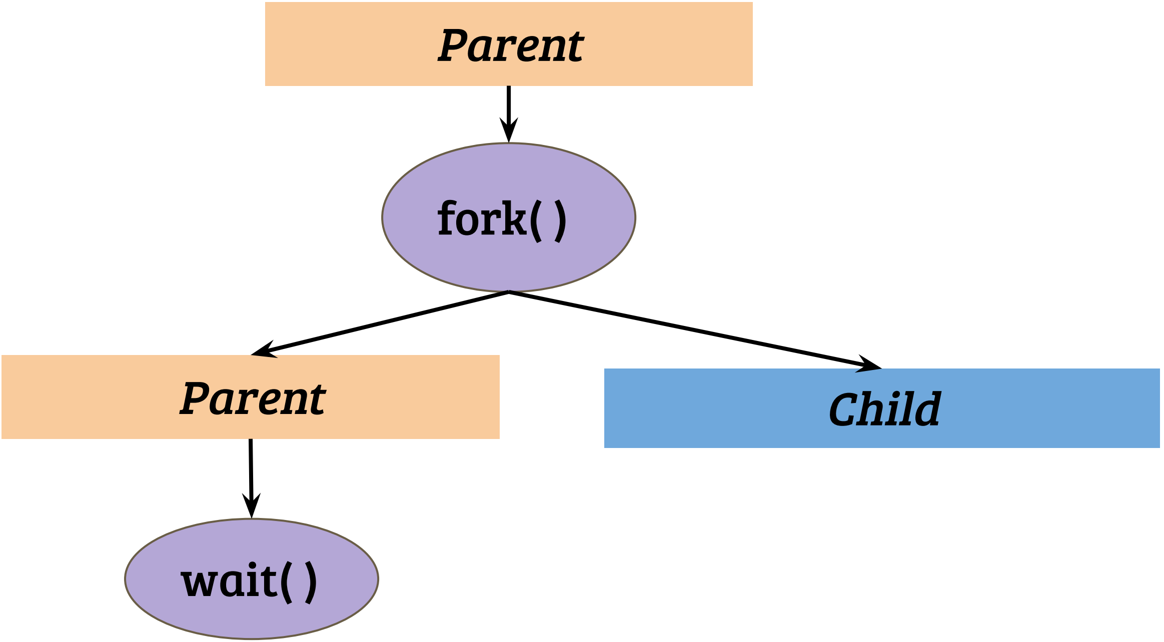


- The calling process commits self destruction! (almost)
- The calling process is cleaned up and replaced by the new executable
- PID remains the same
- On return, new executable starts execution
- PC is loaded with the starting address of the newly loaded binary

Process creation: What and How?

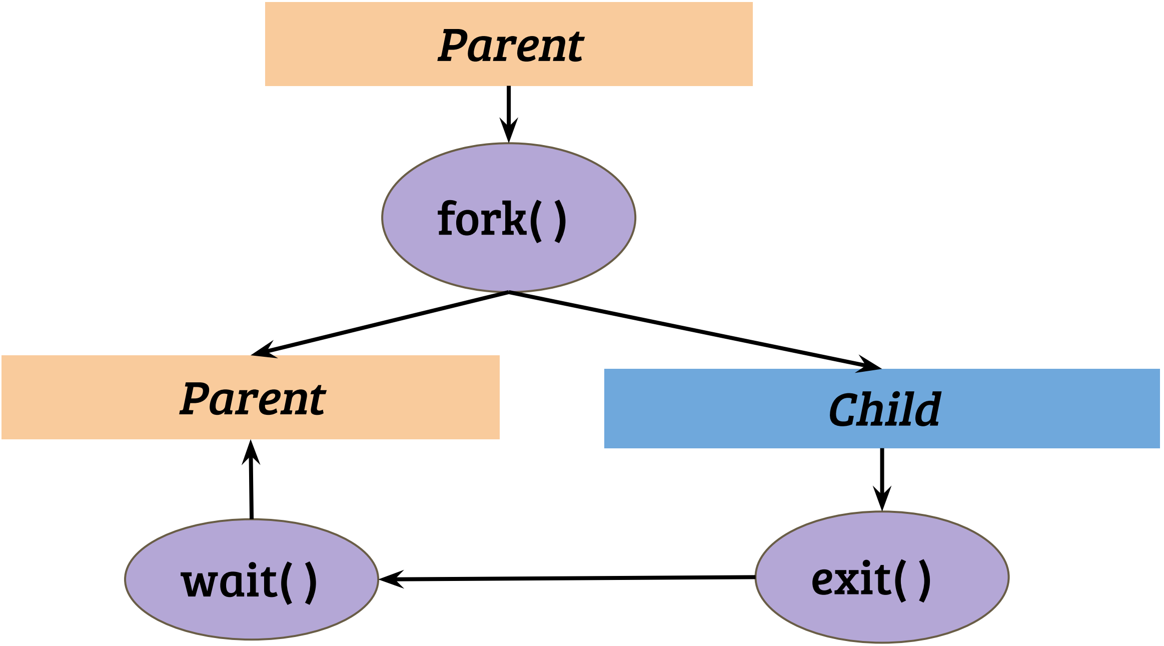
- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?
- fork( ), exec ( ), wait( ) and exit( )
- Who invokes the system calls? In what order?

wait( ) and exit( )



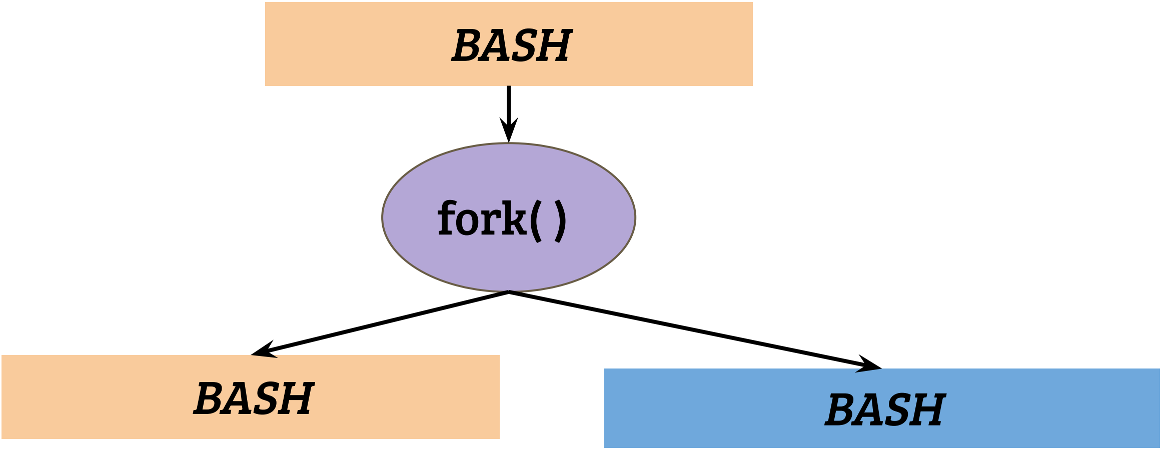
- The wait system call makes the parent wait for child process to exit

wait( ) and exit( )



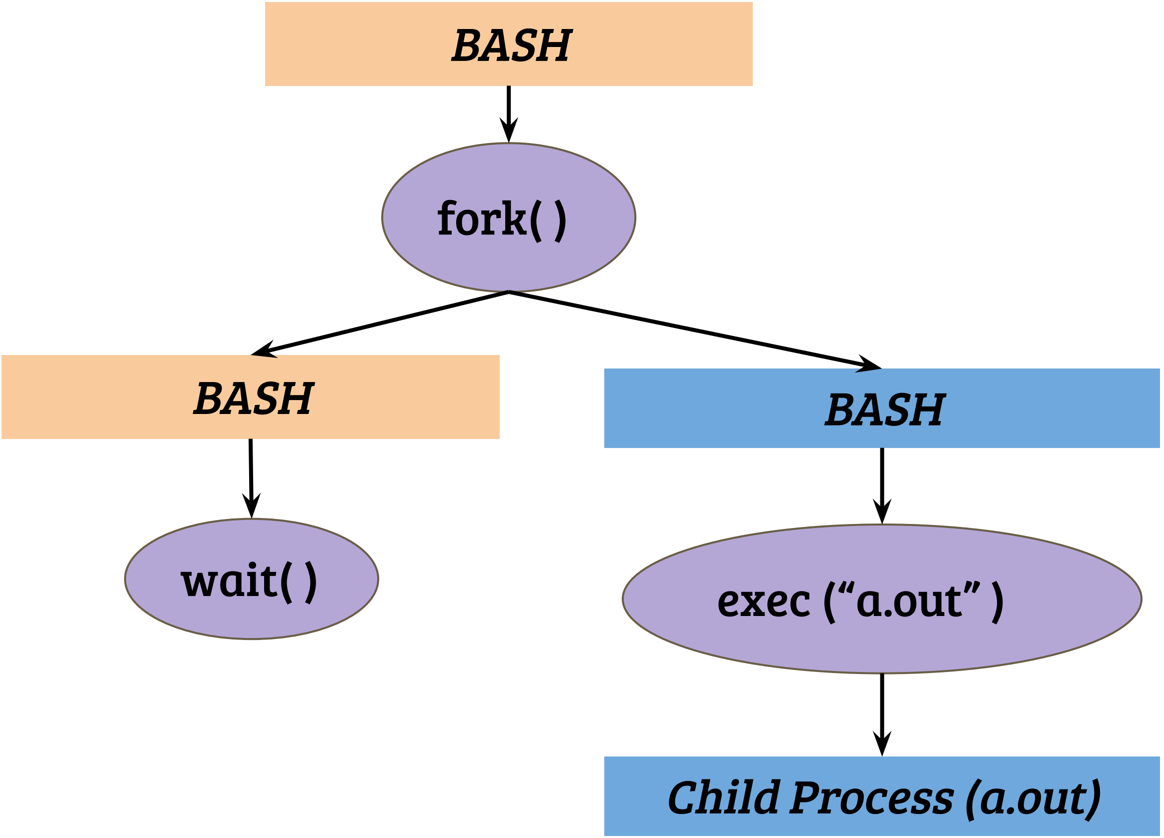
- The wait system call makes the parent wait for child process to exit
- On child exit( ), the wait() system call returns in parent

Shell command line: fork + exec + wait



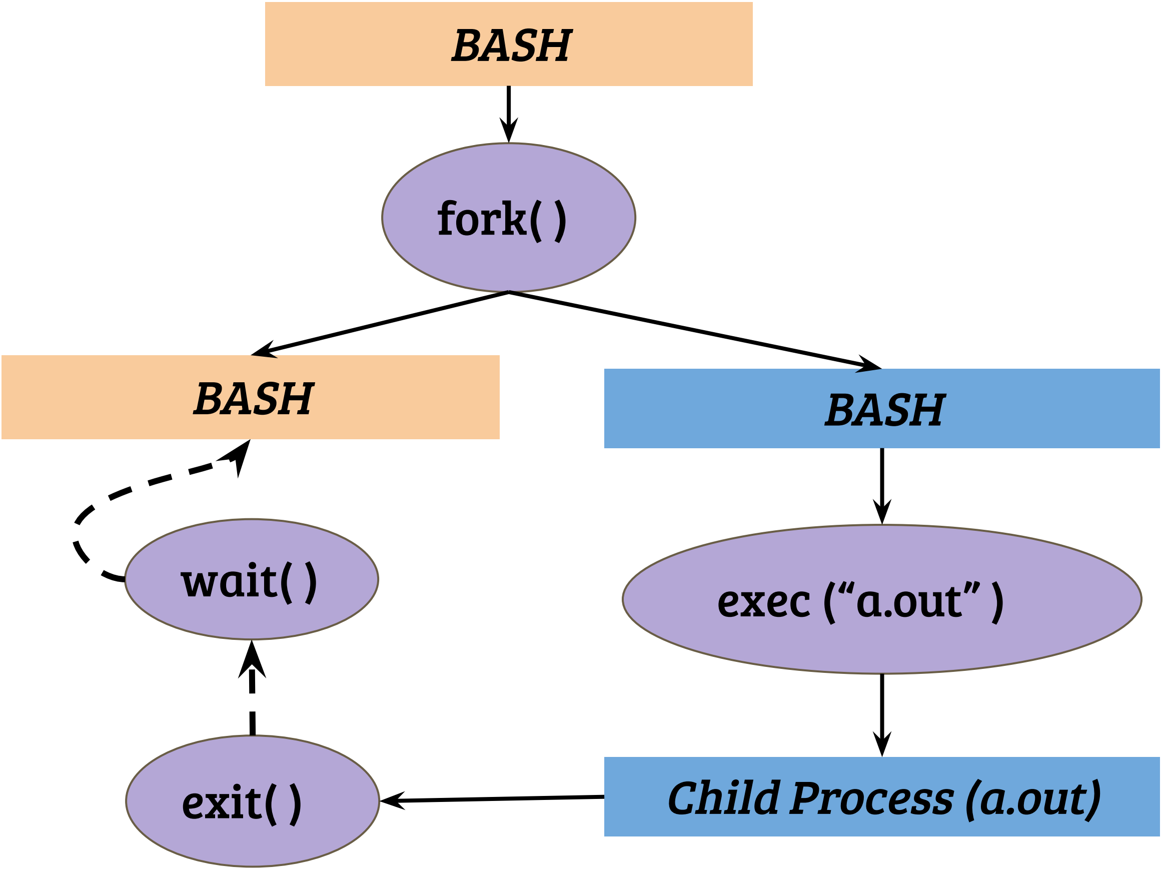
- The BASH process calls fork( )

Shell command line: fork + exec + wait



- Parent process calls wait( ) to wait for child to finish
- Child process invokes exec( )

Shell command line: fork + exec + wait

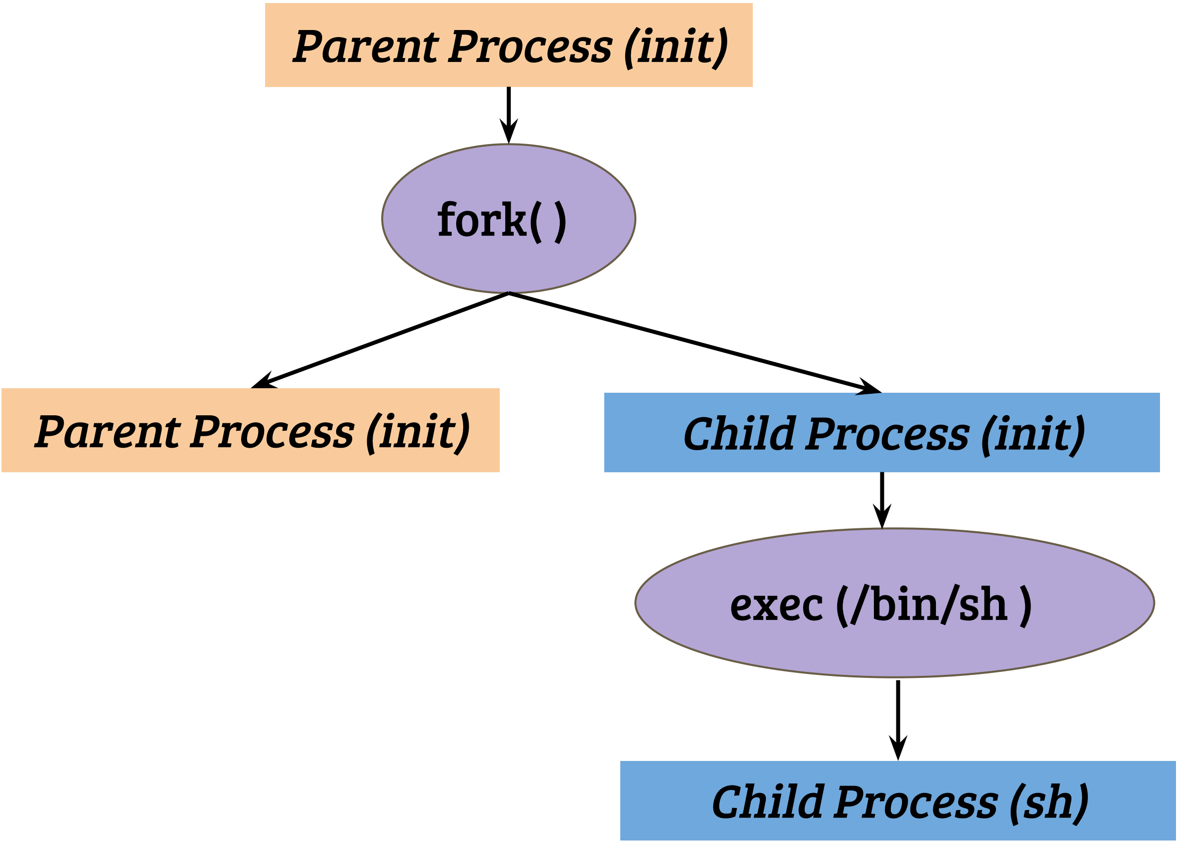


- When child exits, parent gets notified
- The BASH shell is ready for the next command at this point of time

Process creation: What and How?

- How does OS come into action after typing “./a.out” in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?
- fork( ), exec ( ), wait( ) and exit( )
- Who invokes the system calls? In what order?
- The shell process (bash process)
- What is the first user process?

Unix process family using fork + exec



- Fork and exec are used to create the process tree
- Commands: ps, pstree
- See the /proc directory in linux systems

Process creation: What and How?

- How does OS come into action after typing "./a.out" in a shell?
- System calls invoked to explicitly give control to the OS
- What exact system calls are invoked?
- fork( ), exec ( ), wait( ) and exit( )
- Who invokes the system calls?
- The shell process (bash process)
- What is the first user process?
- In Unix systems, it is called the *init* process
- Who creates and schedules the init process?