Lecture 17: Processes

CSE 29: Systems Programming and Software Tools

Aaron Schulman (Shalev)



Process: A running program on an OS



- When you run a program on an OS (e.g., start in shell), it starts a “process”
- A process is an abstraction provided by the OS of a single isolated running program
- A process includes all of the **context** of the program
  - Memory: Stack/Heap and Code
  - Registers: Current state of execution (e.g.. what instruction in the code is running)
  - Peripherals: Even disk access!
- Processes are completely isolated from each other:
  - You can not overwrite another processes’ stack/heap or code!

Process State: What does an OS maintain?



- Process ID: PID – A unique number for a process
- The address space (range) in RAM for the process
- The execution state of the process (i.e., Registers)
- The hardware resources in use (e.g., open files on disk)

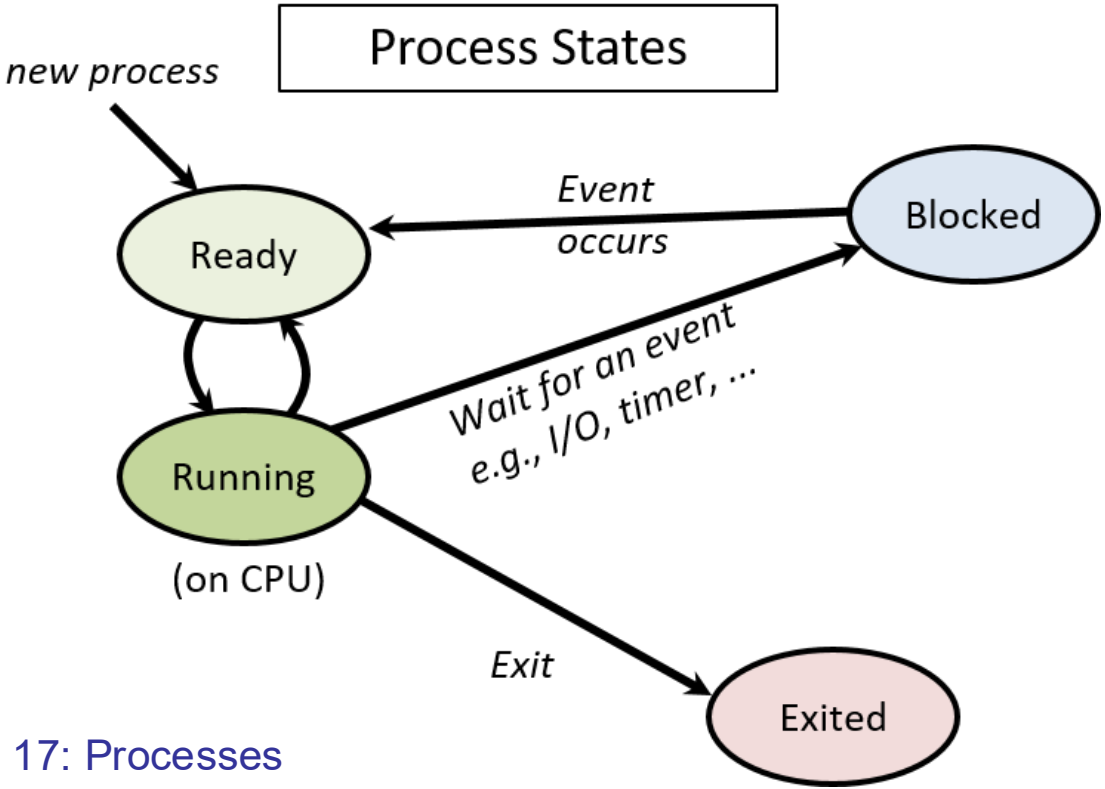
Context Switching: Changing processes



- The Operating System decides when to switch between processes
  - This is known as "**Context Switching**"
  - The this is one of the big benefits of the process abstraction

- When an OS decides that a process had its time to execute it performs the context switch
  1. The OS saves all context of the running process by copying all Registers to RAM
  2. The OS then copies the saved context of another process from RAM to the Registers
  3. The new context switched process begins!

State diagram of a process in the OS



Q1: Is a running process always running?

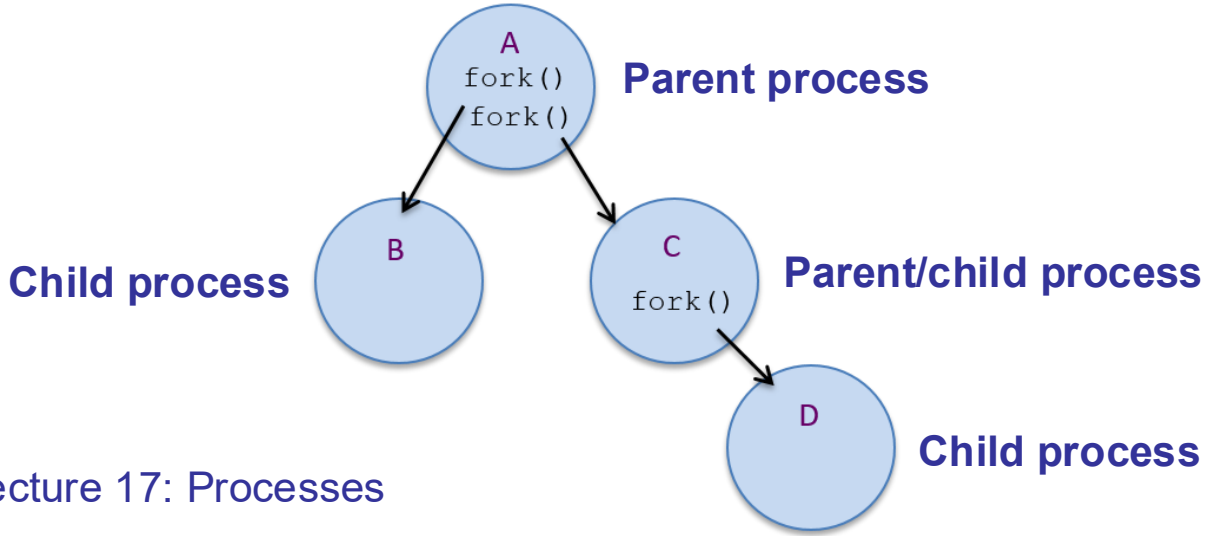


- Does a running process always run instructions or does it ever stop? Why would it stop?

Creating and destroying processes

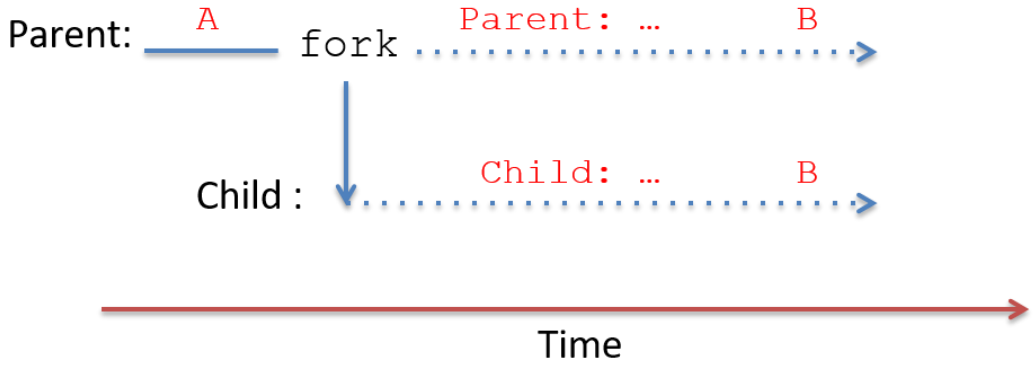
int fork(); Ask OS to create a new process running the current code

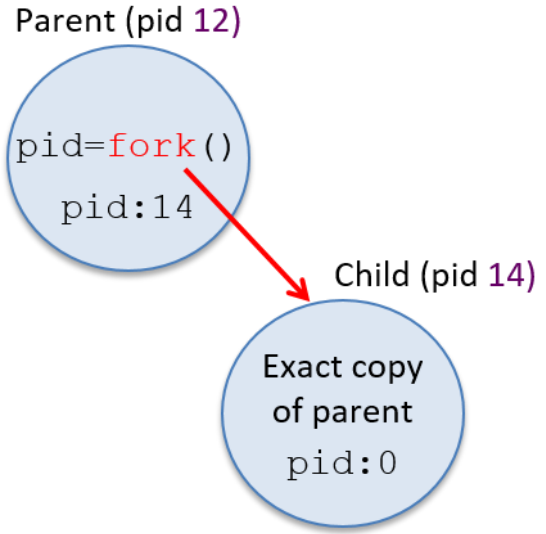
- Creates a new process (new state for address space, PID, registers, hardware)
- Starts with a copy of the same memory of the running process
  - » Process starts with a *copy* of the stack, heap, and code



Fork example

```
pid_t pid;
pid = fork(); /* create a new child process
print("pid = %d\n", pid); /* both parent and child execute this */
```





Q2: What software creates a child process?



- What code runs that will copy the running process into a new running process?

Waiting for a process to finish

int waitpid(pid_t pid, int* wstatus, int options);

Wait for a child process with **pid** to end

- Blocks the parent process until the child has an event (e.g., terminates)
- This is called only by a parent process
- NULL will wait for all children to finish
- Gets the return code from the child process (in wstatus bit field)
  - **WIFEXITED(wstatus) – Check if excited**
  - **WEXITSTATUS(wstatus) – Get return status integer**

What if a parent does not wait for a process to finish?



- Every child process runs in parallel with the parent process
  - Both processes are running at the same time
- Does a parent process have to wait on a child process?
  - No, if the parent does not wait, both processes continue to run in parallel

Q3: What happens child terminates before parent waits?



- If a parent has not yet wait()'ed for their child process but the child terminates, should the parent still be able to wait for it?

  Why or why not?

Zombie Processes



- **Normal Lifecycle:** When a child process finishes, it sends an exit status to its parent process which is wait()'ing for it.
- **Parent's Job:** The parent process must read this status using a system call (like wait() or waitpid()).
- **The Breakdown:** If the parent process is busy, or it does not ever wait for children, it fails to read the status, leaving the dead child behind as a zombie.

Exec – Load new code in a process

```
int execvp(char *filename, char *argv[]);
```

- Loads new code from a binary file (filename) into the running process
- Starts running the main() function in that code
- This function never returns to the calling processes' codebase

