Operating Systems W4L1 - Processes & Threads: Part 2 (ctd)

Class	Operating Systems
Created	@Sep 21, 2020 12:47 PM
Materials	03 - Processes and Threads Part 2.pdf
Reviewed	☐
Type	Lecture

Fork

- A fork is an exact replica, but has its own `pid`, and it returns *twice* → `0` is returned to indicate the process is a child, otherwise it returns the parent id
- It is never reliable to know in what *order* a parent/child will run — There's no specific advantage to this, it's just that they become individual processes
    - Forking is two separate processes, there is nothing inherently sequential between them
- Forks will copy *all* code, but execution only begins from the line right after the fork (not from the start, per se)
    - Slides contain solid examples for forking
- You can use `fork()` in if statements so that only a child/parent (depending on what reutnrs) will execute the if block

Exit

- Exit returns nothing beacuse you won't be using anything from it

- Passing in a different number can be used to alert the parent process
- The `atexit` function runs by the time the program exists; so a function is executed, *then* the `exit()` part of a process is called

Zombies

▼ What is a zombie? Why have them?

A process that is finished, but doesn't do anything and consumes nothing. It seems the purpose of being repated a little later by its parent process

- **Reaping:** Removing a zombie from the process table, done by the parent process
  - The parent "marks" the child as complete, then teh OS does the dirty work of getting rid of it

▼ What happens if the parent process is killed before the child process is reaped?

If the parent has terminated, the `init` process- at the kernel level- will get rid of it

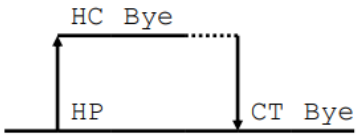
- A defunct process ↔ a zombie (stated in `ps` terminal command)
- Most parent processes will be reaped by the `init` process, and the most "senior" process is definitely reaped by the `init` process

Wait

- Blocks until child exits, return is `pid` of child
- For multiple children, assuming none of them use `wait` internally, the order will be arbitrarily determined by the OS

▼ HP and HC can print out of order, but CT will *always* print after due to the wait function

```
void fork8() {
   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(NULL);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit(0);
}
```



This is how child process is reaped by parent process.

- The `wait` function will only go one generation; parent and child
- After the `wait` (which is considered a way for a parent to reap a child) the child process is considered reaped

execve

- **Polymorphism Relevance:** Something can act differently based on environment and passed in arguments
- `execve` → Execute in virtual environment

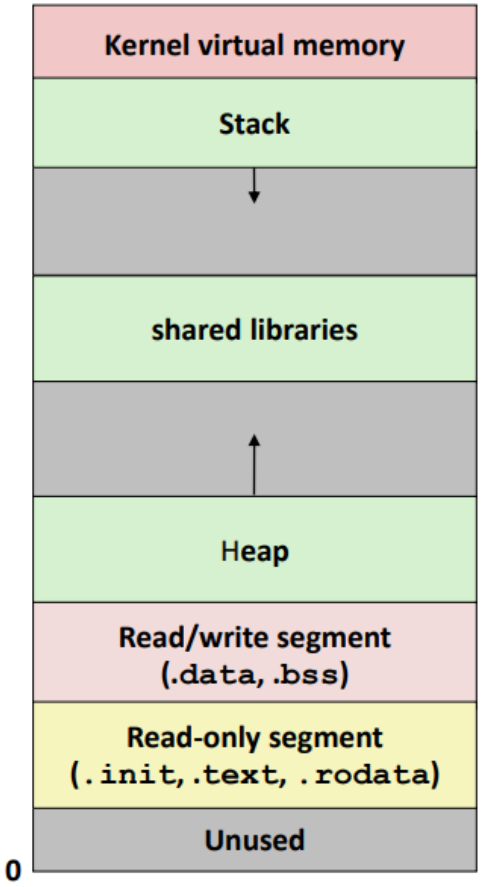
▼ Why does `execve` return a number?

A number is returned- in particular a negative number- if the file name passed into it to execute doesn't exist or something like that

`r < 0`

- The first argument is the name of the file to execute, the second argument (if any) is the arguments for that executable

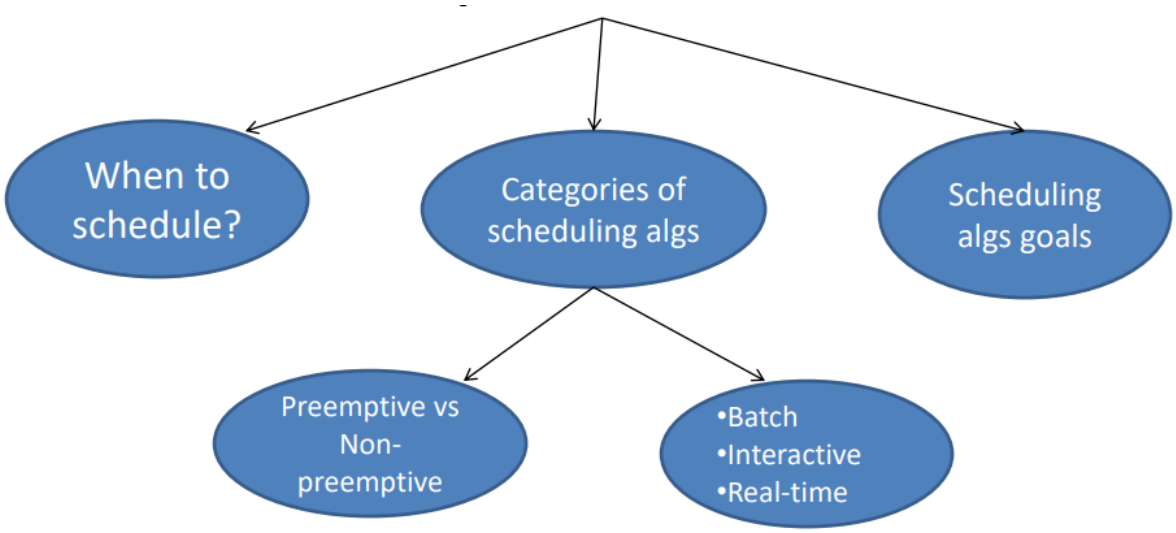
▼ The `execve` leads a new program image and causes OS to overwrite the old program/code and substitude the data with new data (but keeps `pid` and any open files)



Scheduling

- We often have more processes than cores, hence must *schedule*
  - Given a group of ready processes, which process do we choose to run next?
- Many algorithms to do this, but essentially we want to ensure the cores are kept busy and executing processes efficiently

▼ Taxonomy Overview



Definitions

▼ What is preemptive scheduling?

Preemptive scheduling is when a process that can be interrupted, and thus another process is scheduled to execute

▼ What is non-preemptive scheduling?

Non-preemptive scheduling is when the currently running process must finish/exit before another process is scheduled to execute

- Most often, things will be preemptive (especially user processes) — Think why

▼ What are four examples of times to schedule a process?

There are many more, these are just some examples

1. Process created
2. Process exits
3. Process blocks
4. I/O interrupt occurs

Categories

1. Batch — Mostly non-preemptive, no users impatiently waiting

2. Interactive — Premption is successful
3. Real-time — Deadlines