Section 8: Processes

- ~~What is a process~~
- ~~Creating processes~~
- Fork-Exec

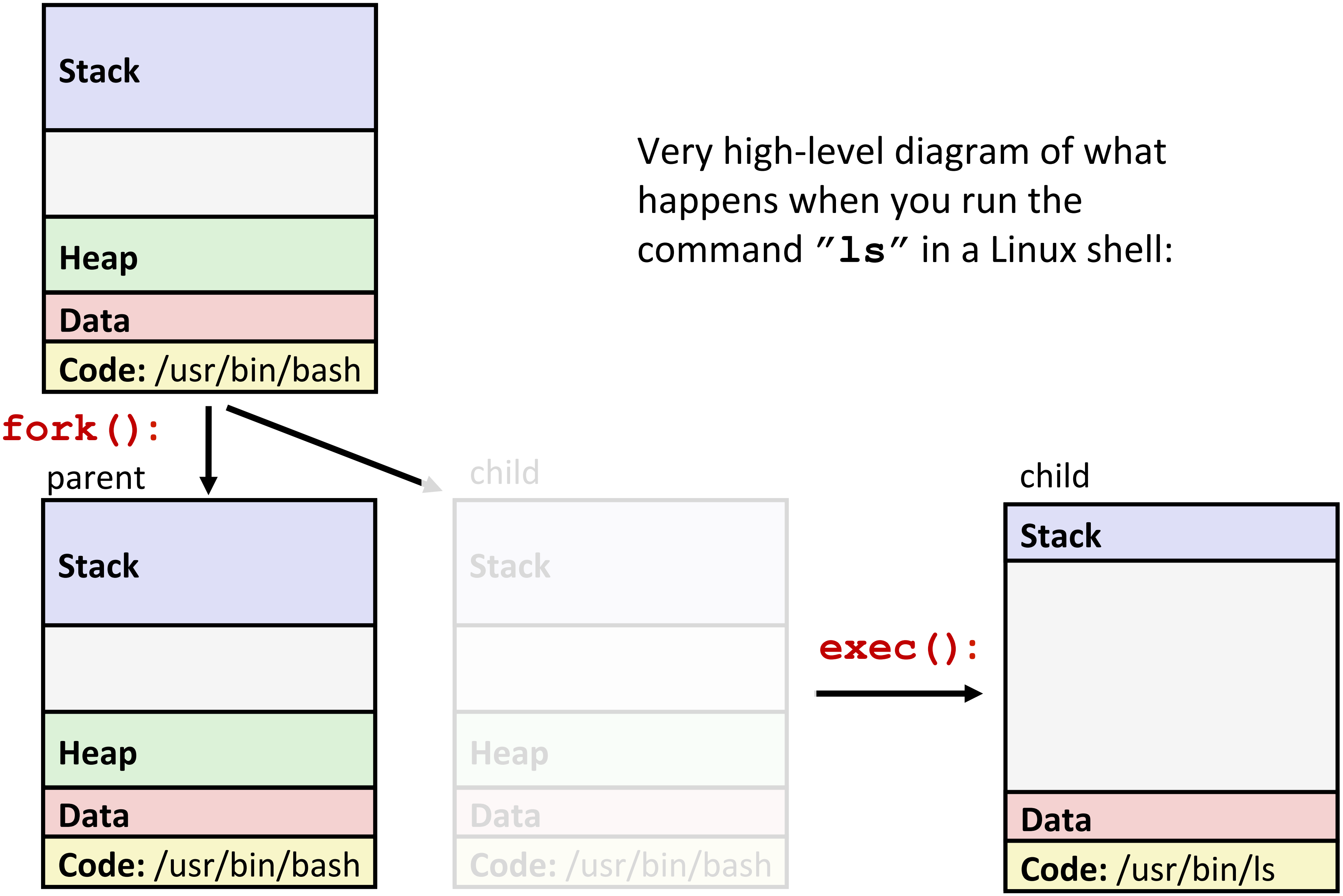
Fork-Exec

- **fork-exec model:**
  - **`fork()`** creates a copy of the current process
  - **`execve()`** replaces the current process’ code & address space with the code for a different program
    - There is a whole family of **`exec`** calls – see **exec(3)** and **execve(2)**

```
// Example arguments: path="/usr/bin/ls”,
//     argv[0]="/usr/bin/ls”, argv[1]="-ahl", argv[2]=NULL
void fork_exec(char *path, char *argv[])
{
    pid_t pid = fork();
    if (pid != 0) {
        printf("Parent: created a child %d\n”, pid);
    } else {
        printf("Child: exec-ing new program now\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
```

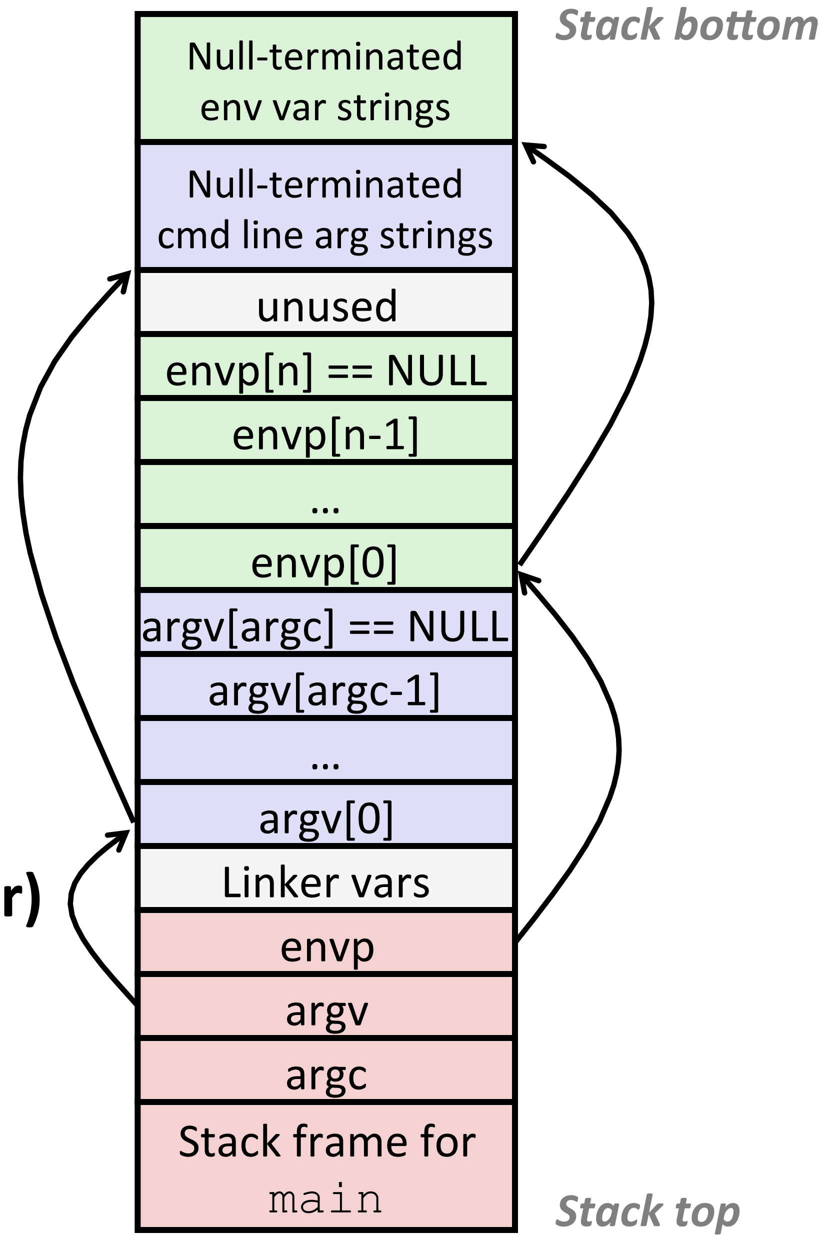
Exec-ing a new program

Very high-level diagram of what happens when you run the command "**`ls`**" in a Linux shell:



`execve`: Loading and Running Programs

-
  ```
  int execve(
     char *filename,
     char *argv[],
     char *envp[]
  )
  ```
- **Loads and runs in current process:**
  - Executable **`filename`**
  - With argument list **`argv`**
  - And environment variable list **`envp`**
    - Env. vars: "name=value" strings (e.g. "`PWD=/homes/iws/pjh`")
- **`execve` *does not return* (unless error)**
- **Overwrites code, data, and stack**
  - Keeps pid, open files, a few other items



Stack bottom

Null-terminated env var strings

Null-terminated cmd line arg strings

unused

envp[n] == NULL

envp[n-1]

…

envp[0]

argv[argc] == NULL

argv[argc-1]

…

argv[0]

Linker vars

envp

argv

argc

Stack frame for `main`

Stack top

`exit`: Ending a process

- `void exit(int status)`
  - Exits a process
    - Status code: 0 is used for a normal exit, nonzero for abnormal exit
  - `atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

- **Idea**
  - When process terminates, it still consumes system resources
    - Various tables maintained by OS
  - Called a “zombie”
    - A living corpse, half alive and half dead
- **Reaping**
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process
- **What if parent doesn’t reap?**
  - If any parent terminates without reaping a child, then child will be reaped by `init` process (pid == 1)
  - But in long-running processes we need *explicit* reaping
    - e.g., shells and servers

`wait`: Synchronizing with Children

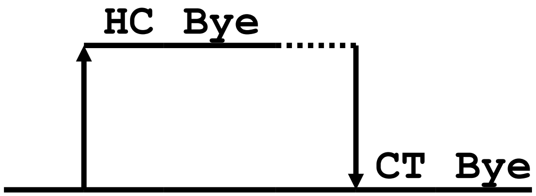
- `int wait(int *child_status)`
  - Suspends current process (i.e. the parent) until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
    - On successful return, the child process is reaped
  - If **`child_status != NULL`**, then the int that it points to will be set to a status indicating why the child process terminated
    - There are special macros for interpreting this status – see **wait(2)**

- If parent process has multiple children, **`wait()`** will return when *any* of the children terminates
  - **`waitpid()`** can be used to wait on a specific child process

wait Example

```
void fork_wait() {
   int child_status;
   pid_t child_pid;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   } else {
      child_pid = wait(&child_status);
      printf("CT: child %d has terminated\n”,
          child_pid);
   }
   printf("Bye\n");
   exit(0);
}
```



Process management summary

- **`fork` gets us two copies of the same process (but `fork()` returns different values to the two processes)**
- **`execve` has a new process substitute itself for the one that called it**
  - Two-process program:
    - First **`fork()`**
    - if (pid == 0) { //child code } else { //parent code }
  - Two different programs:
    - First **`fork()`**
    - if (pid == 0) { **`execve()`** } else { //parent code }
    - Now running two completely different programs
- **`wait` / `waitpid` used to synchronize parent/child execution and to reap child process**

Summary

- **Processes**
  - At any given time, system has multiple active processes
  - Only one can execute at a time, but each process appears to have total control of the processor
  - OS periodically “context switches” between active processes
    - Implemented using *exceptional control flow*
- **Process management**
  - fork-exec model