Last Update: 1-26-23

Process Management

# Environment Variables

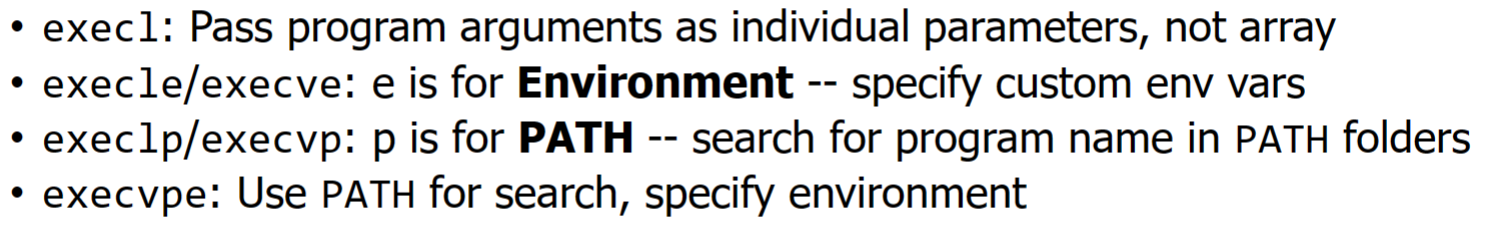
* OS maintains a list of configuration options called **environment variables**
* Environment variables have both names and values
* Each process has its own environment
* A process inherits the environment of its *parent*
* The PATH environment variable
  + Tells us where to look for “built in” programs
  + The built in programs implement the commands we know and love: ls, make, ..
  + If you invoke a command without specifying a directory path, OS searched through the folders in PATH to find program with matching name
* Changing Your Environment
  + Use library functions to set/get variables of type char \*\*environ
    - setenv(), getenv(), unsetenv()

# New Processes, Running Commands, Ending Processes

* Don’t do what your parents do, become an instance of any program you want to be

Int execv(char \* path, char \*argv[])

* + Path: executable file to load from
  + Argv: command-line arguments this process would see, with NULL sentinel
* 6 variants of exec, each with different arguments



* Running a command

execl(“bin/ls”, “ls”, “-l”, NULL)

* + - By convention, first argument is name of program
    - NULL comes at end as a sentinel value (like strings in C)
  + Exec overwrites process memory image inherited from parent
  + Process starts again from scratch with clean slate:
    - New code, program counter, stack
    - New arguments, (possibly) new environment
* Ending a Process
  + One way you have seen: return from main()
  + Can achieve the same effect from any function by calling exit(int status)
    - 0 means success, any other value indicates an error
  + Lots of cleanup happens on an exit: close any open files, tear down virtual address space
  + In bash shell, type echo $? To get exit code of last command
  + If you want to bail out ASAP, can call abort() function
    - Exit with nonzero- something undesirable happened but we can get a clean stopping point
    - Abort- just give up now
* Waiting for a Process
  + An exit code is like a very limited return capability
  + Allows process to communicate a number indicating what happened
  + How do we obtain this code and react to it?

Pid\_t wait(int \*status)

* + Called by a parent to wait for a child
  + The process ID of the child is returned
  + Child’s exit code is placed in location pointed to by status
  + Great example of a blocking operation
    - Process calls wait, cannot make progress until child terminates
  + Waiting for a specific process
    - Wait will brock until any child exits
    - An alternative allows you to wait for a specific child:

Pid\_t waitpid(pid\_t pid, int \*status, int opt)

* + - Wait for pid and store exit code a status
      * Also some special values you can pass in, see the man page for more info
    - Opt allows for some interesting customization
      * WNOHANG: don’t block. Check if child is done, but if not return 0 and continue on to the next line of code
* In-class exercise:

main(){

char cmd[512];

while(1){

printf(“> “);

fscanf(stdin, “%s”, cmd);

if(strcmp(“exit”, cmd) == 0){

Break;

}

pid\_t child\_pid= fork();

if(child\_pid == 0) {

if(execlp(cmd, cmd, NULL) == -1){

perror(“exec”);

return 1;

}   
}

else{

Int exit\_code;

wait(&exit\_code);

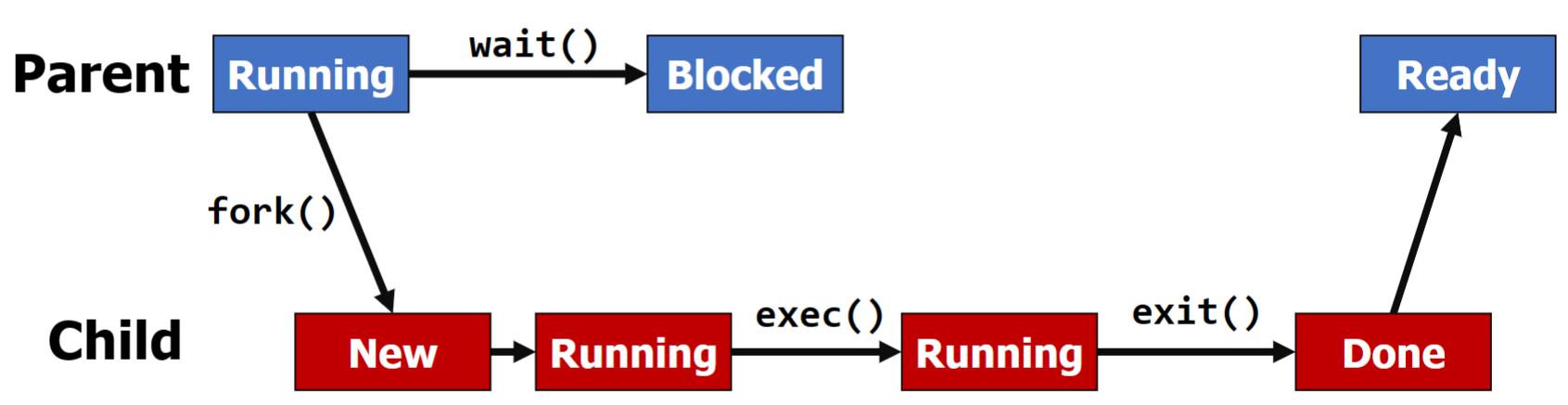
printf(“child exit code: %d\n”, exit\_code);

}

}

}

* Waiting for a child:



# The Shell

* When you are using “the terminal”, you are interacting with a shell process
* On Ubuntu and many other places, this is bash by default
* Read input, launch program, repeat

# Weird OS Processes Names

* Processes can fall into one of the following categories in certain circumstances
  + Daemon
    - A daemon is a process that is always running in the background to provide a service
    - Parent forks child, never waits for it
  + Orphans
    - If a parent terminates while it child is still running, that child becomes an orphan
    - Implies: parent never waited on the child
    - When a process becomes an orphan, it is adopted by init
      * Init: Hero or Villain?
        + Root of the process tree: ancestor of all other procs
        + Generously adopts all those orphans, then acts as a “reaper” when those orphans die
  + Zombie
    - Parent forks a child
    - Child exits before parent waits
    - Child becomes a zombie
    - Parent waits for child
    - Zombie eliminated
    - Why? Need to keep child around for its exit code

# fork() without an exec()

* Case 1: Make use of parallelism- multiple streams of work at the same time
  + Multi-process web browsers (tabs)
    - Each process gets its own address space => better isolation between different browser tabs
      * One tab can’t read/write another tab’s data
      * Fault in one tab cannot crash entire browser
  + Threads and Processes
    - Both are ways to enable multiple streams of execution at once
    - Threads in the same process share an address space
      * Easy cooperation, better efficiency, but less isolation
      * Browsers originally used threads for tabs
    - Each process gets its own address space
      * Better isolation, but less efficient
      * This is why Chrome consumes so much memory