Recitation 0

Welcome, everybody!

Recitation Logistics

- Turn in your worksheets! This is how we grade recitation participation
  - Due every Thursday at 11:59 pm.

- No need to submit the original file, - can just upload a copy of your answers on blank paper, especially since we make you draw out answers sometimes

Today's Topics

- Environment Setup
- Intro to GDB
- Intro to Valgrind
- Processes
- Fork & Exec
- Wait
- Alarm

Environment Setup - Any Questions?

- Docker Setup
  - Installation
  - Container
- Git Repo Setup
  - SSH Key
  - Clone the repo
- VSCode
  - Installation
  - Extension
  - Entering the project

GDB (Debugger)

What is GDB?

GNU (GNU's Not Unix) Project Debugger

- You can can examine the program as it is executing!
- Supports a variety of languages
  - (e.g. asm, C, C++, D, Fortran, Go, Rust, etc.)
- ***Your best friend for this course!***

"But can I use print statements?"

- Using print statement can be tedious.
- Requires the use of many to find where the bug is.
- Unable to examine the state of the program after it crashes.
- `printf("here\n");` can only take you so far.

Using GDB

Use `-g` or `-g3` flag in Makefile for compiling (the provided Makefile has this)

Type in your shell:

a) gdb pokemon_buggy

b) <enter>, then run

start	Start from beginning and stop there
run	Start and run program from beginning
continue	Run until program exits*
step	Run until next line*
bt	Shows call stack
b [fname:]function b [fname:]linenum	Sets breakpoint at beginning of function or at line
print var	Prints var

* = or until next breakpoint

Exercise 1: GDB

- Download files from website
- Run `make`, then debug via gdb

Valgrind

Valgrind

- Your handy debugging tool for memory mismanagement
- It runs your program, and looks for any memory errors during execution
- It will only catch errors it encounters in runtime! Pay attention to **code coverage** - ensure all* lines of code are run in a valgrind session

*or at the very least, the most critical lines

Valgrind Errors

- Memory leaks: memory that hasn't been freed by the time the program exits
- Invalid read/write: accessing unallocated (or deallocated) memory
- Uninitialized bytes: Using memory that was allocated but never had any values put into them

How to Run Valgrind

- Can run in terminal: **`valgrind ./program <program arguments>`**
- Useful valgrind arguments (put between `valgrind` and `./program`)
  - **`--trace-children=<yes|no>`** (default: no)
  - **`--track-origins=<yes|no>`** (default: no)
  - **`--leak-check=<no|summary|yes|full>`** (default: summary)
    - `full` option gives you the most info

Processes

Processes

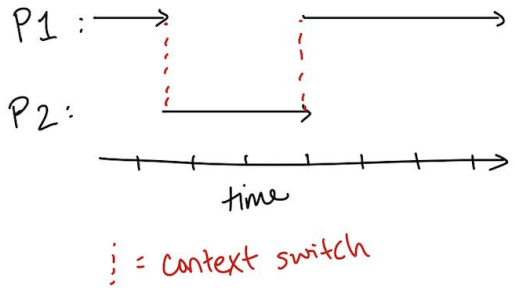
- Process: One instance of a running (or ready to run) program

- Two ways to visualize processes

Processes

- Process: One instance of a running (or ready to run) program

- Two ways to visualize processes

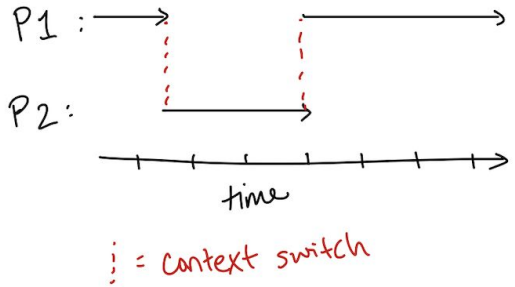


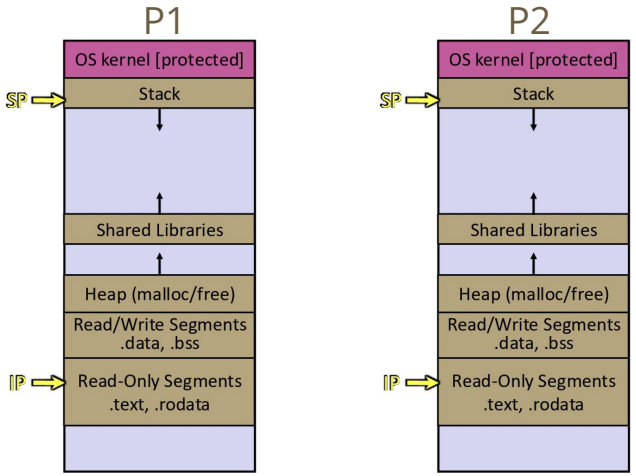
Processes as separate lines of execution

Processes

- Process: One instance of a running (or ready to run) program

- Two ways to visualize processes



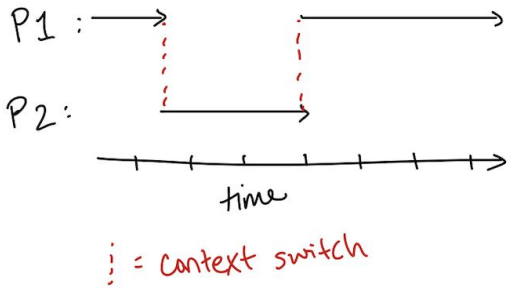


Processes as separate lines of execution OR Processes as separate memory environments

Processes

- Process: One instance of a running (or ready to run) program

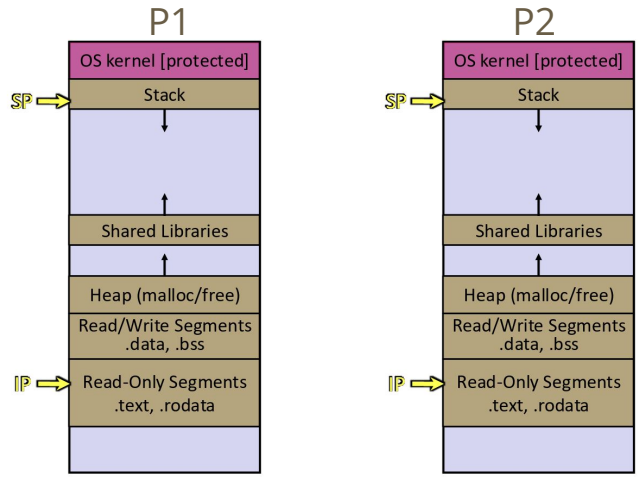
- Two ways to visualize processes



Processes as separate lines of execution

OR

this visual will be more useful later in the semester



Processes as separate memory environments

Exercise 2a: Processes

Which process(es) have access to `file.txt`?

A. Parent
B. Child
C. Both
D. Neither

```c
#include <fcntl.h>
#include <stdlib.h>

int main() {
  pid_t child = fork();
  int fd = open("file.txt", O_WRONLY);
  if (fd == -1) {
    exit(EXIT_FAILURE);
  }
  write(fd, "this is parent or child.", 25);
  close(fd);
  return 0;
}
```

Exercise 2b: Processes

If the parent closes the file, can the child still write to `file.txt`? **Explain you answer.**

```c
#include <fcntl.h>
#include <stdlib.h>

int main() {
  pid_t child = fork();
  int fd = open("file.txt", O_WRONLY);
  if (fd == -1) {
    exit(EXIT_FAILURE);
  }
  write(fd, "this is parent or child.", 25);
  close(fd);
  return 0;
}
```

Fork

Fork

- “The only function that returns twice”
- Generally invoked when we want to run a different program without terminating the current program
- Clones the process that called fork()
  - Memory environment: stack, heap, read-only memory, registers, etc.
  - File descriptor table
  - Signal handlers & mask
- Child starts running the line immediately following fork()

Exec

Exec(ve)

- Replaces the current process with another

  ```
  execve(char *pathname, char *argv[], char *envp[]);
  ```

  - `pathname` = string containing path to binary file to be executed
  - `argv` = array of strings containing arguments to run the next program
    - Argv[0] == pathname
  - `envp` = list of environment variables
    - Just set this parameter to `NULL`

- What's replaced?

- What's unchanged?

Exec(ve)

- Replaces the current process with another

```
execve(char *pathname, char *argv[], char *envp[]);
```

  - `pathname` = string containing path to binary file to be executed
  - `argv` = array of strings containing arguments to run the next program
    - Argv[0] == pathname
  - `envp` = list of environment variables
    - Just set this parameter to `NULL`

- What's replaced? Memory layout (stack, heap globals, loaded code, registers), **signal handlers**
- What's unchanged?

Exec(ve)

- Replaces the current process with another

```
execve(char *pathname, char *argv[], char *envp[]);
```

  - `pathname` = string containing path to binary file to be executed
  - `argv` = array of strings containing arguments to run the next program
    - Argv[0] == pathname
  - `envp` = list of environment variables
    - Just set this parameter to `NULL`

- What's replaced? Memory layout (stack, heap globals, loaded code, registers), **signal handlers**
- What's unchanged? List of open file descriptors, kernel, PID

Wait

Wait

- Parent waits for its child to finish - will block until it receives a signal indicating that the child finished running
  - Can also query how the child finished: was it natural, or was it from a signal?

- A process can only wait on its child (no sibling or grandchild waiting allowed!)

- wait_pid() is more expressive than wait()
  - Waitpid allows you to specify which child you're waiting for
  - Waitpid also allows you to indicate the "type of waiting" you want
    - Block wait
    - Nonblocking wait (with no hang)

Fork, Exec, Wait

- Commonly, the three work together!

- **Fork + Exec** = start a completely new task as a child of current process
  - i.e. if Google Chrome was a running process, then you open a new tab

- **Fork + Exec + Wait** = indicates the current process should not run until newly created task has completed
  - i.e. your shell!

Exercise 3a: The Process "Family Tree"

Here are two diagrams, where each labeled box represents a process. P0 is the "original process" that forks P1. Arrows show the parent-child relationship. The order of processes spawning from first to last is: P0, P1, P2, P3.

Using either C code, psuedocode, or a written description, describe how you would fork 3 processes to achieve diagram 1 and diagram 2.

Diagram 1

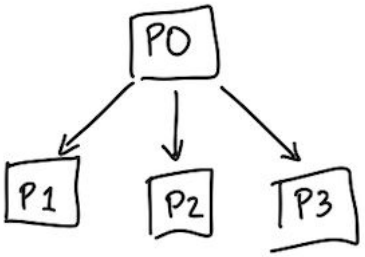
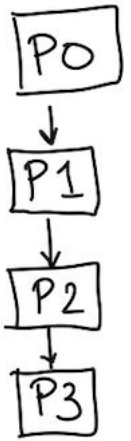


Diagram 2



* arrow shows: parent → child

Exercise 3b: Choose Your Own Fork

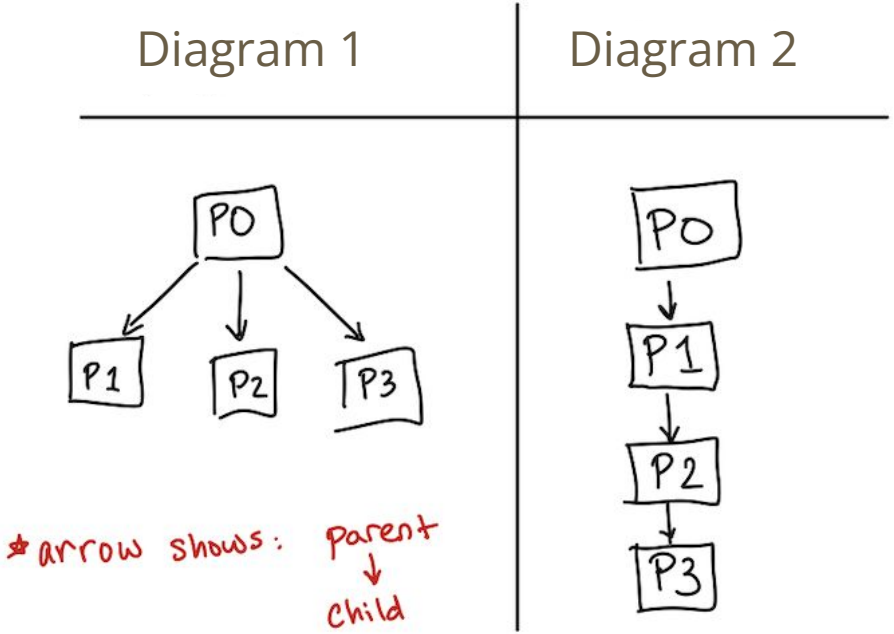
Let's say I have 3 independent tasks: T1, T2, and T3.

- P1 will exec T1
- P2 will exec T2
- P3 will exec T3

All 3 tasks require I/O calls to be made.

P0 must wait until T1, T2, and T3 have finished.

Which diagram will result in the faster runtime? **Explain your answer.**



Exercise 4: Waiting

1. Draw a diagram of all processes and clearly indicate all parent-child relationships.

2. Which of the following are possible outputs? Select all that apply:
   a. B0AC0D0
   b. D0CA0B0
   c. D0A0B0C
   d. CAD00B0
   e. ABCD000

```c
int main(void){
  int level_1 = fork();
  if (level_1 == 0) {
    int level_2a = fork();
    if (level_2a == 0) {
      printf("A");
    } else {
      wait(NULL);
      printf("B");
    }
  } else {
    int level_2b = fork();
    if (level_2b == 0) {
      printf("C");
      exit(0);
    }
    printf("D");
  }
  printf("0");
  return (0);
}
```

Alarm

Alarm

- Will send a SIGALRM signal after a set number of seconds unless cancelled

  Question: Which command will cancel an alarm?

  a. alarm(-1);
  b. alarm(0)

- SIGALRM default disposition: terminate process receiving the signal
  - But can change the default behavior using signal handlers, or block it with a mask