Lecture 31 (4-13-26)

I/O (System Calls)

Motivating Questions:

- What is a **system call?**
- What is a **process?**
  - What **attributes** does it have?
  - What **system calls** can you do with **processes?**
- What is a **signal?**
  - What does a **signal** do to a **process**?
  - What **system calls** can you do with **signals?**

Review from last lecture

- remember when you make a system call, your program is suspended until the system call is complete
- make sure you check if system calls fail

`copy.c`

```c
/* copy.c */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <fcntl.h>

void usage(const char *program) {
    fprintf(stderr, "Usage: %s source target\n", program);
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 3) {
         usage(argv[0]);
         return EXIT_FAILURE;
    }

    // Command line arguments
    const char *source_path = argv[1];
    const char *target_path = argv[2];


    // Open file for reading
    int source_fd = open(source_path, O_RDONLY);
    if (source_fd < 0){
         fprintf(stderr, "open(%s): %s\n", source_path,  strerror(errno));
         return EXIT_FAILURE
     }

     FILE *source_steam = fdopen(source_fd, "r");
     if (!source_stream){
          close(source_fd)
          return EXIT_FAILURE;
     }

    // Open file for writing
    FILE *target_stream = fopen(target_path, "w");
    if (!target_stream) {
         fclose(source_stram);
         return EXIT_FAILURE;
     }

    // Copy from source file to target file
    char buffer[BUFSIZ];
    size_t nread; //bc fread() sys calls returns number of bytes read
    while (nread = fread(buffer, 1, BUFSIZ, source_stream) > 0){
         fwrite(buffuer, 1, n_read, target_stream);
    }

    // Close files
    fclose(target_stream);
    fclose(source_stream)

    return EXIT_SUCCESS;
}
```

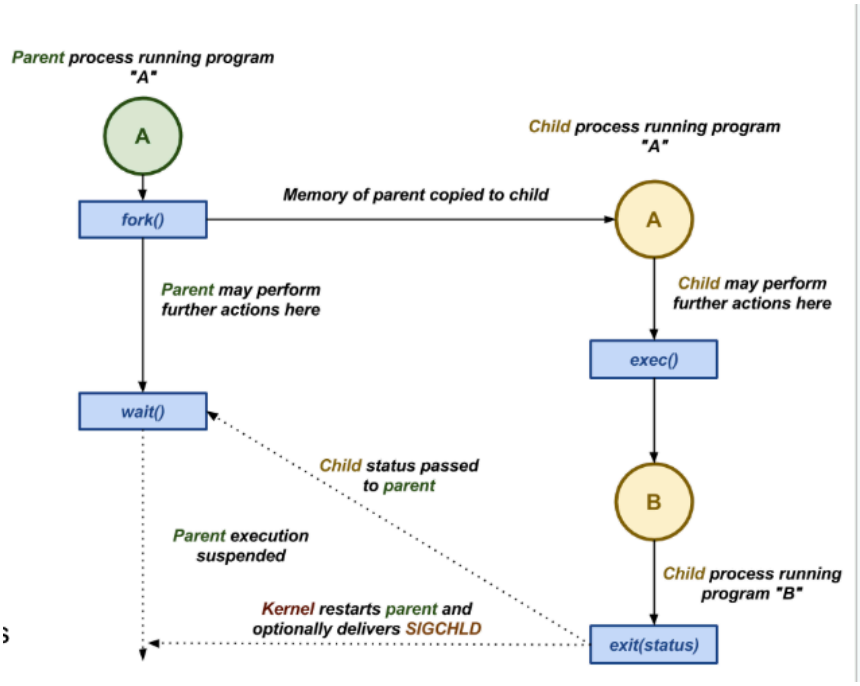
Processes: Overview

- A process is a unit of **allocation**
- **Address Space:**
  - Code, data, heap, stack
- **Kernel State:**
  - Permissions, file descriptors, etc.
- **Execution Context:**
  - Program counter, stack pointer, data registers

Process: Life Cycle

1. Parent **forks** to create a new process
2. **Child** performs actions, possibly **exec** to run another program
3. **Parent waits** for **child** process
4. **Child** exits
5. **Parent** receives **child's** exit status

- Diagram:



- **System calls that will defintelty be on the final:**
  - `fork()` → allocates a new process
  - `exec()` → Load code into address space
  - `exit()` → store exit status

- `wait()` → retrieves exit system call

`system.c`

```
/* system.c: An example of using system instead of manually forking,
execing,
 * and waiting. */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
     int status = system("ls -l");
     if (status != EXIT_SUCCESS){
          fprintf(stderrm "system: %s\n", sterror(errno));
          return EXIT_FAILURE;
     }

    return EXIT_SUCCESS;
}

/* vim: set sts=4 sw=4 ts=8 expandtab ft=c: */
```

- when you call `fork()` on Linux it internally calls `clone()`

`popen.c`

```
/* popen.c: An example of using os.popen instead of manually creating a
pipe,
 * forking, and execing. */

#include <errno.h>
#include <stdio.h>
```

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

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
     FILE *pstream = popen("ls -l", "r");
     if (!pstream){
          perror("popen")
          return EXIT_FAILURE;
     }
     pclose(pstream);
    return EXIT_SUCCESS;
}

/* vim: set sts=4 sw=4 ts=8 expandtab ft=c: */
```

- parents create a pipe
    - parents read from it
    - child writes to i