Lecture 32 (4-15-26)

Processes (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?**

- `wait()` is the system call deallocates a process
- with `exit()` the exit status is stored in the kernel state
- so a parent `fork()` to create a process
- a parents `wait()` to deallocate a process

`fork_wait.c`

```c
/* fork_wait.c: An example of forking 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 = EXIT_SUCCESS;

    pid_t pid = fork();

    switch (pid){
        case 0: //Child
            printf("[Child POV] Child pid: %d, Parent pid: %d\n", getpid(),
getppid());
            break;
        case -1: // Parent - Failure
            fprintf(stderr, "fork(): %s\n", sterror(errno));
            break;
        default: // Parent - Success
            printf("[Parent POV] Child pid: %d, Parent pid: %d\n", pid,
getpid());
            break;
    }
    return status;
}

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

creating an orphan

- Notice the parent dies before the child is done.
    - In the case PID = 1 adopts the orphan

```
/* fork_wait.c: An example of forking 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 = EXIT_SUCCESS;

    pid_t pid = fork();
```

```
    switch (pid){
         case 0: //Child
              sleep(1); //THIS IS THE LINE THAT CREATES THE ORPHAN)
              printf("[Child POV] Child pid: %d, Parent pid: %d\n", getpid(),
getppid());
              break;
         case -1: // Parent - Failure
              fprintf(stderr, "fork(): %s\n", sterror(errno));
              break;
         default: // Parent - Success
              printf("[Parent POV] Child pid: %d, Parent pid: %d\n", pid,
getpid());
              break;
    }
    return status;
}

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

fixing it by having parent `wait()`

```
/* fork_wait.c: An example of forking 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 = EXIT_SUCCESS;

    pid_t pid = fork();

    switch (pid){
         case 0: //Child
              sleep(1);
              printf("[Child POV] Child pid: %d, Parent pid: %d\n", getpid(),
getppid());
              break;
```

```
        case -1: // Parent - Failure
            fprintf(stderr, "fork(): %s\n", sterror(errno));
            break;
        default: // Parent - Success
            printf("[Parent POV] Child pid: %d, Parent pid: %d\n", pid,
getpid());
            pid = wait(&status);
            break;
    }
    return status;
}

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

- exit status you get back from `wait()` is a bitset
  - to get the real number call the macro `WEXITSTATUS` on it

`fork_exec.c`

```
/* fork_exec.c: An example of forking and execing */

#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 = EXIT_SUCCESS;
    pid_t pid = fork();

    if (pid < 0) { //Parent - Failure
        return EXIT_FAILURE
    }

    if (pid == 0){
        printf("[Child POV] Child pid: %d, Parent pid: %d\n", getpid(),
getppid());
```

```
        execlp("ls", "iDontCare", "-l", NULL); //execlp requires explicit
NULL at the end
        // the iDontCare is is argv[0] , but so we dont care
        // however, by convention it is the name of the program

    } else { // Parent - Success
        printf("[Child POV] Child pid: %d, Parent pid: %d\n", getpid(),
getppid());
        pid = wait(&status);
        printf("[Parent POV] Child pid: %d, Parent pid: %d\n", pid,
WEXITSTATUS(status));
    }
    return status;
}

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

- different approach:

```
/* fork_exec.c: An example of forking and execing */

#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 = EXIT_SUCCESS;
    pid_t pid = fork();

    if (pid < 0) { //Parent - Failure
        return EXIT_FAILURE
    }

    if (pid == 0){
        printf("[Child POV] Child pid: %d, Parent pid: %d\n", getpid(),
getppid());
```

```
        char *arguments[] = {
            "ls",
            "-l",
            NULL,
        };
        execlp(arguments[0], arguments); //execlp requires explicit NULL at
the end
        // the iDontCare is is argv[0] , but so we dont care
        // however, by convention it is the name of the program
        exit(EXIT_FAILURE) // because this line only runs if exec fails

    } else { // Parent - Success
        printf("[Child POV] Child pid: %d, Parent pid: %d\n", getpid(),
getppid());
        pid = wait(&status);
        printf("[Parent POV] Child pid: %d, Parent pid: %d\n", pid,
WEXITSTATUS(status));
    }
    return status;
}

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

`zombies.c

```
/* zombies.c:  A demonstration of what happens if parents don't wait for
their children. */

#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 nchildren = atoi(argv[1]);
```

```
    for int(i = 0; i < nchildren; i++;){
        pid_t pid = fork();

        if (pid < 0){ //Parent -Failure
            fprintf(stderr, "fork(): %s\n", sterror(errno))
        }
        if (pid == 0){ //Child
            printf("hi from %d\n", i);
            exit(i);
        } else {

        }
    }

    //all children spawn and then you wait
    // this enables parallel computing / concurrency
    for int(i = 0; i < nchildren; i++;){
        wait(NULL);
    }

    return EXIT_SUCCESS;
}

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

- to know limits on a process run `ulimit -a`