Recitation 9: Processes, Signals, TSHLab

Instructor: TAs

Outline

- Cachelab Style
- Process Lifecycle
- Signal Handling

Style Grading

- **Cachelab grades will soon be available on Autolab**
  - Click ‘view source’ on your latest submission to see our feedback
- **Common mistakes**
  - Descriptions at the top of your file and functions.
  - NULL checking for malloc/calloc and fopen.
    - ERROR CHECKING IS KEY IN TSHLAB!
  - Writing everything in main function without helpers.
  - Lack of comments in general.
- **The labs are hard, don’t lose points after your hard work.**

Shell Lab

- Due next week, Tuesday, October 31st.
- Simulate a Linux-like shell with I/O redirection.

- PLEASE review the lecture slides and read the writeup before starting, and also while mid way through the assignment.
- A lot of important points mentioned in the writeup.

- Lot of style points can be saved solely by reading the writeup and following the instructions.
- Pay attention to race conditions!

Process "Lifecycle"

We will review each of these phases today

- **Fork() – Create a duplicate, a "child", of the process**
- **Execve() – Replace the running program**
- **Exit() – End the running program**
- **Waitpid() – Wait for a child**

Notes on Examples

- **Full source code of all programs is available**
  - TAs may demo specific programs
- **In the following examples, exit() is called**
  - We do this to be explicit about the program's behavior
  - Exit should generally be reserved for terminating on error
- **Unless otherwise noted, all syscalls succeed**
  - Error checking code is omitted.

Processes are separate

- How many lines are printed?
- If pid is at address `0x7fff2bcc264c`, what is printed?

```
int main(int argc, char** argv)
{
    pid_t pid;
    pid = fork();
    printf("%p - %d\n", &pid, pid);
    exit(0);
}
```

Processes Change

- What does this program print?

```
int main(int argc, char** argv)
{
    char* args[3];
    args[0] = "/bin/echo";
    args[1] = "Hi 18213!";
    args[2] = NULL;
    execv(args[0], args);
    printf("Hi 15213!\n");
    exit(0);
}
```

On Error

- How should we handle malloc failing?

```
const size_t HUGE = 1 * 1024 * 1024 * 1024;

int main(int argc, char** argv)
{
    char* buf = malloc(HUGE * HUGE);
    if (buf == NULL)
    {
        fprintf(stderr, "Failure at %u\n", __LINE__);
        exit(1);
    }
    printf("Buf at %p\n", buf);
    free(buf);
    exit(0);
}
```

Exit values can convey information

- Two values are printed, describe their relation.

```
int main(int argc, char** argv)
{
    pid_t pid = fork();
    if (pid == 0) { exit(getpid());}
    else
    {
        int status = 0;
        waitpid(pid, &status, 0);
        printf("0x%x exited with 0x%x\n", pid,
               WEXITSTATUS(status));
    }
    exit(0);
}
```

Processes have ancestry

- Find the errors in this code, assume fork() and exit() are successful

```
int main(int argc, char** argv)
{
    int status = 0, ret = 0;
    pid_t pid = fork();
    if (pid == 0)
    {
        pid = fork();
        exit(getpid());
    }
    ret = waitpid(-1, &status, 0);
    printf("Process %d exited with %d\n", ret, status);
    ret = waitpid(-1, &status, 0);
    printf("Process %d exited with %d\n", ret, status);
    exit(0);
}
```

Process Graphs

- How many different sequences can be printed?

```
int main(int argc, char** argv)
{
    int status;
    pid_t pid;
    if (fork() == 0)
    {
        pid = fork();
        printf("Child: %d\n", getpid());
        if (pid == 0) {exit(0);}
    }
    pid = wait(&status);
    printf("Parent: %d\n", pid);
    exit(0);
}
```

Process Diagram

Process Graphs

- How many different sequences can be printed?

```
int main(int argc, char** argv)
{
    pid_t pid;
    char* tgt = "child";
    pid = fork();
    if (pid == 0) {
        pid = getppid(); // Get parent pid
        tgt = "parent";
    }
    kill(pid, SIGKILL);
    printf("Sent SIGKILL to %s:%d\n", tgt, pid);
    exit(0);
}
```

Signals and Handling

- **Signals can happen at any time**
  - Control when through blocking signals
- **Signals also communicate that events have occurred**
  - What event(s) correspond to each signal?
- **Write separate routines for receiving (i.e., signals)**

Blocking Signals

- What value(s) does this code print?

```
int counter = 0;
void handler(int sig) {counter++;}

int main(int argc, char** argv)
{
    sigset_t mask, prev;
    int i;

    sigfillset(&mask);
    sigprocmask(SIG_BLOCK, &mask, &prev);
    signal(SIGCHLD, handler);
    for (i = 0; i < 10; i++)
    {
        if (fork() == 0) {exit(0);}
    }
    sigprocmask(SIG_SETMASK, &prev, NULL);
    printf("%d\n", counter);
    return 0;
}
```

Proper signal handling

- **For the previous code, how to handle the signals?**
  - We want to count child exits.
  - We don't want to count exits until all 10 children are created.

- **Discuss**

If you get stuck

- **Read the writeup!**
- **Do manual unit testing before `runtrace` and `sdriver`!**

- **Read the writeup!**
- **Post private questions on piazza!**

- **Read the man pages on the syscalls.**
  - Especially the error conditions
  - What errors should terminate the shell?
  - What errors should be reported?