COMP1521 25T2

Week 9 Lecture 1

Processes and Pipes

Adapted from Angela Finlayson, Abiram Nadarajah,
Hammond Pearce,
Andrew Taylor and John Shepherd's slides

Today's Lecture

- Processes
  - Recap: setenv, getenv
  - Recap: execve, fork, waitpid
  - system(...)
  - posix_spawn(...)

- Inter Process Communication
  - Pipes!

When a Linux Process stops responding

Recap: Environment variables

- When run, a program is passed a set of environment variables:
  - Array of strings of the form name=value, terminated with NULL.

```c
// Access via environ variable
extern char **environ;
for (int i = 0; environ[i] != NULL; i++)
    printf("%s\n", environ[i]);
```

```c
// Access via a third argument to main (sometimes)
int main(int argc, char *argv[], char *env[])
```

```c
// Preferred method is to use getenv/setenv
char *getenv(const char *name);
int setenv(const char *name, const char *value, int overwrite);
```

Recap: Processes

- A process is an instance of an executing program.
- Each process has an execution state, defined by…
  - Current values of CPU registers
  - Current contents of its memory
  - Information about open files (and other results of system calls)
- Each process has a unique process ID, or PID: a positive integer, type **pid_t**, defined in <unistd.h>.
  getpid() to get PID; getppid() to get parent PID
- Each process has a parent process
- A process may have child processes

Recap: processes in C

- exec() family

```
int execv(const char *file, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

  - Run another program in place of the current process
  - Most of the current process is re-initialized:
    - e.g. new address space is created - all variables lost
  - Open file descriptors survive (e.g. stdin/stdout unchanged)
  - PID unchanged

- fork() -- clone yourself

```
pid_t fork(void);
```

Recap: waitpid()

```
pid_t waitpid(pid_t pid, int *wstatus, int options)
```

- **`wstatus`** is set to hold info about pid.
  - e.g., exit status if pid terminated
  - macros allow precise determination of state change
    - (e.g. WIFEXITED(wstatus), WCOREDUMP(wstatus))
- **`options`** provide variations in waitpid() behaviour
  - default: wait for child process to terminate
  - WNOHANG: return immediately if no child has exited
  - WCONTINUED: return if a stopped child has been restarted
- For more information, **`man 2 waitpid`**.

Recap: fork() and exec() to run /bin/date

```c
pid_t pid = fork();
if (pid == -1) {
    perror("fork"); // print why fork failed
} else if (pid == 0) { // child
    char *date_argv[] = {"/bin/date", "--utc", NULL};
    execv("/bin/date", date_argv);
    perror("execvpe"); // print why exec failed
} else { // parent
    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        exit(1);
    }
    printf("/bin/date exit status was %d\n", exit_status);
}
```

Demo: fork_exec.c

system(): convenient but unsafe

```c
#include <stdlib.h>
int system(const char *command)
```

Creates another process and runs command via /bin/sh.
Waits for command to finish and returns exit status

```c
// run date --utc to print current UTC
int exit_status = system("/bin/date --utc");
printf("/bin/date exit status was %d\n", exit_status);
return 0;
```

system(): convenient but unsafe

- Convenient … but extremely dangerous —
  - very brittle; highly vulnerable to security exploits
  - https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=OS+Command+Injection
  - use for quick debugging and throw-away programs only

Demo: system.c

Making Processes

- Old-fashioned way **fork()** then **exec()**
  - **fork()** duplicates the current process (parent+child)
  - **exec()** “overwrites” the current process (run by child)

- New, standard way **posix_spawn()**

posix_spawn() — Run a new process

```
#include <spawn.h>
int posix_spawn(
    pid_t *pid, const char *path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *attrp,
    char *const argv[], char *const envp[]);
```

- pid: returns process id of new program
- path: path to the program to run
- file_actions: specifies file actions to be performed before running the program
    - can be used to redirect stdin, stdout to file or pipe

posix_spawn() — Run a new process

```c
#include <spawn.h>
int posix_spawn(
    pid_t *pid, const char *path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *attrp,
    char *const argv[], char *const envp[]);
```

- attrp: specifies attributes for new process (not covered in COMP1521)
- argv: arguments to pass to new program
- envp: environment to pass to new program
- can also use **posix_spawnp** which searches PATH

Example: posix_spawn() to run /bin/date

```c
pid_t pid;
extern char **environ;
char *date_argv[] = {"/bin/date", "--utc", NULL};
// spawn "/bin/date" as a separate process
int ret = posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv, environ);
if (ret != 0) {
    errno = ret; //posix_spawn returns error code, does not set errno
    perror("spawn"); exit(1);
}
// wait for spawned processes to finish
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); exit(1);
}
printf("/bin/date exit status was %d\n", exit_status);
```

Demo: spawn.c

Example: posix_spawn() versus system()

Running ls -ld via posix_spawn()

```
char *ls_argv[2] = {"/bin/ls", "-ld", NULL};
pid_t pid; int ret;
extern char **environ;
if((ret = posix_spawn(&pid, "/bin/ls", NULL,
    NULL, ls_argv, environ)) != 0)
{
    errno = ret; perror("spawn"); exit(1);
}
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); exit(1);
}
```

Running ls -ld via system()

```
system("ls -ld");
```

Demo: lsld_spawn.c
lsld_system.c

Setting environment var for child process

```
// set environment variable STATUS
setenv("STATUS", "great", 1);
char *getenv_argv[] = {"./get_status", NULL};
pid_t pid;
extern char **environ;
int ret = posix_spawn(&pid, "./get_status", NULL, NULL,
                          getenv_argv, environ);
if (ret != 0) {
    errno = ret; perror("spawn"); return 1;
}
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); exit(1);
```

Demo

Change behaviour with an environment var

```
pid_t pid;
char *date_argv[] = { "/bin/date", NULL };
char *date_environment[] = { "TZ=Australia/Perth", NULL };
// print time in Perth
int ret = posix_spawn(&pid, "/bin/date", NULL, NULL, date_argv,
date_environment);
if (ret != 0) {
    errno = ret; perror("spawn"); return 1;
}
int exit_status;
if (waitpid(pid, &exit_status, 0) == -1) {
    perror("waitpid"); return 1;
}
printf("/bin/date exit status was %d\n", exit_status);
```

Demo: spawn_environment.c

exit() — terminate yourself

```c
#include <stdlib.h>
void exit(int status);
```

- Triggers any functions registered as atexit()
- Flushes stdio buffers; closes open FILE *'s
- Terminates current process
- SIGCHLD signal is sent to parent
- Returns status to parent (via waitpid())
- Any child processes are inherited by init (pid 1)

_exit() — terminate yourself without …

```c
#include <stdlib.h>

void _exit(int status);
```

- terminates current process without triggering functions registered as atexit()
- stdio buffers not flushed
- sometimes used by children of fork() when exiting

Aside: Zombie Processes

Aside: Zombie Processes

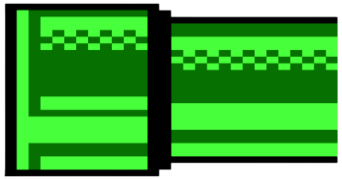
- When a process terminates, some of its details remain in the process table, and the process is called a **zombie**.
- A **zombie** remains until its parent calls wait() or waitpid() (reaps the process)
  - This can be a problem for long running processes that don't reap their children.
  - **zombie**s that hang around waste system resources.
- Orphan process = a process whose parent has exited
  - when parent exits, orphan assigned PID 1 (init) as its new parent
  - init always accepts notifications of child terminations

Inter Process Communication

pipe() — stream bytes between processes

Send output of one process as input to another

Process A
WRITES
to the
pipe

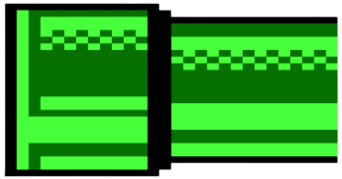


Process B
READS
from the
pipe

pipe() — stream bytes between processes

Demo: on the command line: `seq 1 10 | wc`

Process A
WRITES
to the
pipe



Process B
READS
from the
pipe

pipe() — stream bytes between processes

```c
#include <unistd.h>
int pipe(int pipefd[2]);
```

- Pipes: unidirectional byte streams provided by operating system
  - **pipefd[0]**: set to file descriptor of read end of pipe
  - **pipefd[1]**: set to file descriptor of write end of pipe
  - bytes written to **pipefd[1]** will be read from **pipefd[0]**
- Child processes (by default) inherit file descriptors including pipes

Closing pipes

- Parent can send/receive bytes (not both) to child via pipe
  - parent and child should both close unused pipe file descriptors
  - e.g if bytes being written (sent) parent to child
    - parent should close read end **pipefd[0]**
    - child should close write end **pipefd[1]**

- Pipe file descriptors can be used with stdio via:

```
FILE *fdopen(int fd, const char *mode);
```

popen() — convenient way to set up pipe

```c
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- runs command via /bin/sh
- if type is “w” pipe to stdin of command created
- if type is “r” pipe from stdout of command created
- FILE * stream returned - then use fgetc/fputc etc
  - NULL returned if error
- close stream with pclose (not fclose)
  - pclose waits for command and returns exit status

popen() — unsafe

```c
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- Convenient but brittle
- Vulnerable to command injection (same as system())
- Try to avoid use except in debugging and throw-away programs

Example: process output with popen()

```c
// popen passes string to a shell for evaluation
// brittle and highly-vulnerable to security exploits
// popen is suitable for quick debugging and throw-away programs only
FILE *p = popen("/bin/date --utc", "r");
if (p == NULL) {
   perror(""); return 1;
}
char line[256];
if (fgets(line, sizeof line, p) == NULL) {
   fprintf(stderr, "no output from date\n"); return 1;
}
printf("output captured from /bin/date was: '%s'\n", line);
pclose(p); // returns command exit status
```

Demo: read_popen.c

Example: input to a process with popen()

```c
int main(void) {
    // popen passes command to a shell for evaluation
    // brittle and highly-vulnerable to security exploits
    //
    // tr a-z A-Z - passes stdin to stdout converting lower case to upper case
    FILE *p = popen("tr a-z A-Z", "w");
    if (p == NULL) {
        perror("");
        return 1;
    }
    fprintf(p, "hello, i am a COMP1521 aficionado\n");
    pclose(p); // returns command exit status
    return 0;
}
```

Demo: write_popen.c

posix_spawn and pipes (advanced topic)

```
int posix_spawn_file_actions_destroy(
   posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_init(
   posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_addclose(
   posix_spawn_file_actions_t *file_actions, int fildes);
int posix_spawn_file_actions_adddup2(
   posix_spawn_file_actions_t *file_actions, int fildes, int newfildes);
```

- functions to combine file ops with posix_spawn process creation
- awkward to understand and use — but robust

Example: capturing output from a process: spawn_read_pipe.c
Example: sending input to a process: spawn_write_pipe.c

What we learnt Today

- Processes
  - posix_spawn
- Pipes
  - posix_spawn (advanced usage)

```
system("ls -l");
```

```
pid_t pid;
char *argv[] = {"/bin/ls", "-l", cmd, NULL};
posix_spawn(&pid, "/bin/ls", NULL, NULL, argv, NULL);
```

Next Lecture

- Concurrency and Parallelism
  - Threads
  - Mutexes
  - Atomics

Reach Out

Content Related Questions:
Forum

Admin related Questions email:
cs1521@cse.unsw.edu.au

Student Support | I Need Help With…

My Feelings and Mental Health
Managing Low Mood, Unusual Feelings & Depression

- **Mental Health Connect** — student.unsw.edu.au/**counselling** Telehealth
- **Mind HUB** — student.unsw.edu.au/**mind-hub** Online Self-Help Resources

In Australia Call Afterhours UNSW Mental Health Support Line — 1300 787 026 5pm-9am

Outside Australia Afterhours 24-hour Medibank Hotline — +61 (2) 8905 0307

Uni and Life Pressures
Stress, Financial, Visas, Accommodation & More

- **Student Support** — student.unsw.edu.au/**advisors**
- **Indigenous Student Support**

Reporting Sexual Assault/Harassment

- **Equity Diversity and Inclusion (EDI)** — edi.unsw.edu.au/**sexual-misconduct**

Educational Adjustments
To Manage my Studies and Disability / Health Condition

- **Equitable Learning Service (ELS)** — student.unsw.edu.au/**els**

Academic and Study Skills

- **Academic Language Skills** — student.unsw.edu.au/**skills**

Special Consideration
Because Life Impacts our Studies and Exams

- Special Consideration — student.unsw.edu.au/**special-consideration**