COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/

Processes

- A process is an instance of a running program.
- Programs do not have to have a GUI!
- When you issue a command in the UNIX shell, the shell starts a new process
- `ps -x` will list all of your current processes.

Executing other Programs

- `int system(const char * string)`
- `int execvp(const char* file, const char* argv[])`
- `pid_t fork(void)`
- `pid_t wait(int * status)`
- `pid_t waitpid(pid_t pid, int* status, int options)`

Fork

- Creates a second process, identical to the current one except for the return value of `fork`
- A pipe created before `fork` can then be used for communication between the two processes.
- `execvp` will (if successful) replace the current process one running the target binary's `main` function
- File descriptors without the close-on-exec flag will survive `exec`

Fork Bombs

```
int main(int argc, char ** argv) {
  while (1) fork();
  return 0;
}
```

Fork: Same Code, New Address Space

```
int main(int argc, char ** argv) {
  const char * c1;
  const char * c2;
  c1 = "Hello ";
  if (fork())
    c2 = "World!";
  else
    c2 = "Child!";
  printf("%s %s\n", c1, c2);
  return 0;
}
```

Pipe, Fork, Dup2 and Exec

```
int main(int argc, char ** argv) {
  int p[2]; pipe(p);
  if (fork() == 0) {
    close(p[1]);
    close(0);
    dup2(p[0], 0);
    execlp("/bin/cat", "", "-", NULL);
  }
  close(p[0]);
  while (1) write(p[1], ".", 1);
  return 0;
}
```

Example: Pipes

- $ls|less$
- Pipes – The pipe (|) is a junction that tells the shell to connect the standard output of one program with the standard input of another.

Example: Input-output redirection

- $<$ tells the shell to use a file as `stdin`
- $>$ tells the shell to create a file for `stdout`
- $>>$ tells the shell to append to a file for `stdout`
- $2>$ tells the shell to create a file for `stderr`

Example: Filters

```
cat -n < test.c > test.txt
```

- A filter is a program that accepts input, transforms it, and outputs the transformed data.
- Pipes and IO redirection can be used to combine filters to create powerful shell programs

All of these examples are implemented by the shell using `pipe`, `dup2`, `fork` and `exec`.

Zombies!

- Exit status of process must be communicated to parent
- Parent may not acknowledge status immediately

$\Rightarrow$ Zombie process is left

You cannot kill zombies, but you can kill their parents (useful if they fail to acknowledge the death of their child)!

Zombie Attack

```
int main(int argc, char ** argv) {
  while (! fork());
  return 0;
}
```

Homework

Watch "Shawn of the Dead".

Init

- Process 1
- Parent of orphans
- Reads (and discards) exit status

$\Rightarrow$ Orphaned zombies die immediately

Introspection

- `pid_t getpid(void)`
- `pid_t getppid(void)`

Be nice!

- `int nice(int incr)`
- Only root can use a negative priority

Process Termination

- `return` from main method
- `void exit(int status)`
- `void abort(void)`
- `void _exit(int status)`
- `int atexit(void (*function)(void))`

Inter-Process Communication (IPC)

- FIFOs
- Semaphores
- Shared memory
- Messages

The command "`ipcs -a`" lists all active IPC resources managed by the kernel (except for FIFOs).

Named Pipes or FIFOs

- FIFOs work just like regular pipes
- However, they have a filename on disk

$\Rightarrow$ FIFOs can be used for communciations between arbitrary processes

- `int mkfifo(const char * pathname, mode_t mode)`

Semaphores

- A semaphore is created with a given non-negative number describing available resources
- Acquiring a semaphore atomically decrements the number of available resources (down)
- Releasing a semaphore atomically increments the number of available resources (up)

Example

- A computer has two monitors
- Applications want to be displayed full-screen on a monitor
- Semaphore is created with initial value of 2
- Application going full-screen tries to acquire (decrement by one); if more than two applications are already full-screen, acquisition either blocks or fails
- Full-screen applications that exit release the semaphore (increment by one)

IPC Semaphore Operations

- `sem_t *sem_open(const char *name, int oflag, mode_t mode unsigned int value)`
- `int sem_close(sem_t *sem)`
- `int sem_wait(sem_t *sem)` (down)
- `int sem_trywait(sem_t *sem)`
- `int sem_post(sem_t *sem)` (up)

Homework

Read up on the “Sleeping barber problem”.

Shared Memory

- `int shmget(key_t key, size_t size, int shmflg)`
- `void *shmat(int shmid, const void *shmaddr, int shmflg)`
- `int shmdt(const void *shmaddr)`
- `int shmctl(int shmid, int cmd, struct shmid_ds *buf)`

Messages

- `int msgget(key_t key, int msgflg)`
- `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)`
- `ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg)`

Questions

?

Question!

These days, TCP/IP sockets are virtually always used instead of IPC messages. Why?

Process User Identifiers

- Each process is associated with multiple user IDs: real, effective, saved and possibly others
- Real UID is the UID of the process that created this process. Can only be changed if effective UID is root (0).
- Effective UID is used for permission checks; EUID can be changed to real UID or to saved UID. If EUID is 0, anything goes.
- New files are created using the effective UID

SUID, SGID

- If permissions of executable file are set to SUID, SUID of executed process will be set to UID of the file's owner.
- This allows the program to switch to those permissions using `seteuid(SUID)`
- Processes also have multiple group IDs, the same rules apply.
- Binaries with SUID and SGID can be used to elevate permissions

The API

- `uid_t getuid(void)`
- `uid_t geteuid(void)`
- `gid_t getgid(void)`
- `gid_t getegid(void)`
- `int setuid(uid_t uid)`
- `int setgid(gid_t gid)`

The API

- `struct passwd * getpwnam(const char * name)`
- `struct passwd * getpwuid(uid_t uid)`

struct passwd

```
struct passwd {
  char   *pw_name;
  char   *pw_passwd;
  uid_t   pw_uid;
  gid_t   pw_gid;
  char   *pw_gecos;
  char   *pw_dir;
  char   *pw_shell;
};
```

clone

`clone` is how `fork` is implemented on Linux:

```
int clone(int (*fn)(void *),
          void *child_stack,
          int flags, void *arg, ...)
```

Again, the `clone` call is Linux-specific.

$\Rightarrow$ More about this in Lecture 19.

ipc

`ipc` is how `semctl`, `semget`, `semop`, `shmget` and so on are implemented on Linux:

```
int ipc(unsigned int call,
        int first,
        int second,
        int third,
        void *ptr,
        long fifth)
```

Again, the `ipc` call is Linux-specific.

Questions

?

Homework

Can you use `getpwnam` to obtain the password of another user in the GNU/Linux lab? How about obtaining your own password? Find out!