Process Management

CST 357/457 – Systems Programming
Michael Ruth, Ph.D.
Associate Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Discuss process fundamentals including process concepts, process components, APIs, and executing new processes
- Explain the process termination concepts
- Discuss managing users and groups of processes including queries and change
- Explain the daemon concept and its core implementation in a Linux & C system

Processes

- Processes are object code in execution: active, running programs
- A process is also associated with various resources which are managed by kernel
  - Kernel manages these resources for the processes using a process descriptor
- A process is a virtualization abstraction
  - The kernel provides each process with a virtualized processor and memory
  - The kernel handles the multiple processes, but each process believes it's on its own

Programs, Processes, & Threads

- A ***binary*** is compiled, executable code that is dormant…
  - We colloquially use the term ***program***
- A ***process*** is a running program
  - Includes the binary, instance of virtualized memory, kernel resources, security context, and one or more threads
- A ***thread*** is the unit of activity inside a process
  - Has its own virtualized processor which includes a stack, processor state such as registers and an IP

The Process ID (**pid**)

- Each process is represented by a unique identifier (**pid**)
  - Guaranteed to be unique at any single point in time
    - Kernel allocates PIDs in strictly linear fashion until it reaches its max (max = 32768)
      - Will not reuse PIDs until it wraps around
  - The idle process which is the process that the kernel runs when there are no other runnable process has pid 0
    - first process started is called ***init*** and has pid 1
      - Boots the system & launches login programs

Process Hierarchy

- The process that spawns a new process is known as the ***parent***
  - New process is called the ***child***
- Every process is spawned from another process
  - So every process has a parent (**ppid**)
- Each process is ***owned by a user & group***
  - Controls access rights for the process
  - Each child process inherits their user/group
- Each process is also part of a ***process group***
  - Also inherited
  - Makes it easy to send signals to a group of processes

Get the PID/PPID

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```


Running a New Process

- There are two ways to do this:
  - **exec**
    - Loads the binary program which replaces the parent
    - Called ***executing*** a program
    - There is no single function, there are many
  - **fork**
    - Creates a near-duplicate of the parent (initially)
    - Called ***forking***

execl

```
#include <unistd.h>

int execl (const char *path,
const char *arg, …);
```

- The … is variadic!
  - more arguments can follow, but should be NULL terminated

execl EX

```
int ret;

ret =
execl("/bin/vi","vi","/home/stud
ent/homework.txt", NULL)
```


What's Different/Same?

- Difference?
  - Any pending signals are lost
  - Any signals the process is catching are returned to their default behavior
  - Any memory locks are dropped
  - Most thread attributes are returned to defaults
  - Anything related to process's memory address space is cleared
  - Anything that exists in user space is cleared
- Same?
  - PID, PPID, priority, and owning user/group

The rest of the **exec** family

```
#include <unistd.h>

int execlp(const char *file, const char *arg, …);

int execle(const char *path, const char *arg, …,
char * const envp[]);

int execv(const char *path, const char *argv[]);

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

int execve(const char *file, const char *argv[],
const char *envp[]);
```

The `fork()` System Call

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void)
```


What's Different/Same? (2)

- Difference?
  - The pid of the child is newly allocated and different from that of the parent
  - The child's parent pid is set to the pid of its parent process
  - Resource statistics are reset to zero in the child
  - Any pending signals are cleared and are not inherited by the child
  - Any acquired file locks are not inherited by the child

Common Usage of Fork

```
pid = fork();

If (!pid) {
    ret =
    execl("/bin/vi","vi","/ho
    me/student/homework.txt",
    NULL)
```

Copy-On-Write (COW)

- Early systems duplicated everything upon the fork which is time-consuming
  - Lazy optimization strategy (COW) with a simple premise:
    - If we're all reading, no copy needs to be made
    - If we're writing, make the copy

Terminating a Process

```
#include <stdlib.h>

void exit(int status)
```


Other Ways to Terminate

- The classic way to terminate a program is simply to let it finish
  - "falling off the end"
  - implicit exit() call
- Good Practice:
  - Either use exit() or have main return
- A process can also terminate:
  - Process is sent the signal SIGTERM/SIGKILL
  - Incurring the wrath of the kernel (faults)

atexit()

- You can register a function to execute upon termination
  - Think cleanup type of operation
  - Functions execute in the reverse order they are registered (LIFO)
  - Functions cannot call exit()
  - **#include <stdlib.h>**
  - **int atexit(void (*function)(void));**
    - Prototype:
      - **void function(void);**

Waiting for Terminated Children

- When a process terminates, the kernel sends SIGCHLD to the parent
  - We'll learn about signals later
- When a child terminates before its parent, it becomes a ***zombie*** process
  - Waits for parent to inquire about its status
  - Once parent inquires, the child ceases to exist

Wait() & Macros

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)
```

- Returns the PID of the terminated child or -1 on error
- If not NULL, the status pointer contains additional information (uses MACROS)
  - **int WIFEXITED(status) -NORMAL**
  - **int WIFSIGNALED(status)-KILLED BY SIGNAL**
  - **int WIFSTOPPED(status) -STOPPED**
  - **int WIFCONTINUED(status) -CONTINUED**
  - **int WEXITSTATUS(status)**
  - **int WTERMSIG(status)**
  - **int WSTOPSIG(status)**
  - **int WCOREDUMP(status)**

Launching & Waiting

```
#define _XOPEN_SOURCE
#include <stdlib.h>

int system(const char *cmd);
```


Users & Groups

- Best practices in software development encourage the use of ***least-privilege***
  - Processes should execute with the minimum level of rights possible
  - IE, don't run everything as root!
- Sometime, though, we need higher privileges, so we need to change users/groups, etc

Real, Effective, & Saved UID/GIDs

- There are 4 user IDs and 4 group IDs associated with each process:
  - Real ID:
    - UID of the user who ran the process
    - Set to parent's Real UID and doesn't change
  - **Effective ID:**
    - UID that the process is currently wielding
    - ***Permission checks happen against this value***
    - Initially the same as Real ID
    - By executing a setuid binary (suid), the process can change its effective ID
      - More exactly, it is changed to the owner of the executable file (**filesys ID**)
  - Saved ID:
    - Process's original effective UID
    - Upon an exec call, the effective UID is set to Saved ID

Changing Real/Saved UID/GID

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t uid);
int setgid(gid_t gid);
```

- Sets the effective UID/GID
  - if current effective UID/GID is root, then the saved and real are also set
- NOTE:
  - Root can use any value (valid UID/GID)
  - NonRoot must use either real or saved UID/GID

Changing Effective UID/GID

```
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t uid);
int setegid(gid_t gid);
```

- Sets the effective UID/GID
- NOTE:
  - Root can use any value (valid UID/GID)
  - NonRoot must use either real or saved UID/GID

Obtaining UID/GIDs

```
#include <sys/types.h>
#include <unistd.h>
/* real */
uid_t getuid(void);
gid_t getgid(void);
/* effective */
uid_t geteuid(void);
gid_t getegid(void);
```

Sessions & Process Groups

- Each process is a member of a ***process group***, which is one or more processes associated with another for job control
  - Primary attribute: send signals to entire group at once
  - each group has a ***process group leader***
- A ***session*** is a collection of one or more process groups (usually tied to a shell)
  - Exist to consolidate logins around terminals
  - Process groups in a session are divided:
    - One foreground process group
    - Zero or more background groups
  - Each session has a ***session leader***

Daemons

- A daemon is a process that runs in the background
  - Normally started at boot and are run as a special user (root, apache, etc)
  - Handle system-level tasks
  - As a convention, name usually ends in a d
  - In general, two requirements:
    - Runs as child of init
    - Must NOT be connected to a terminal

Steps to be a Daemon

- Call Fork()
- The parent calls exit()
  - Cleans up parent & ensures parent is init
- Call setsid()
  - Daemons want nothing to do with process groups or sessions, so it's necessary to create your own
- Changing working directory via chdir()
  - Don't wish to lock anything by accident
- Close all file descriptors
  - Don't want any inheritance
- Open file descripts 0, 1, and 2 and redirect them to /dev/null

Most UNIX Systems

```
#include <unistd.h>

int daemon(int nochdir, int
noclose)
```


Summary

- Discussed process fundamentals including process concepts, process components, APIs, and executing new processes
- Explained the process termination concepts
- Discussed managing users and groups of processes including queries and change
- Explained the daemon concept and its core implementation in a Linux & C system

Questions?