Chapter 8: Process Control

CMPS 105: Systems Programming
Prof. Scott Brandt
T Th 2-3:45
Soc Sci 2, Rm. 167

Process Identifiers

- Guaranteed to be unique for each currently executing process on a single computer
- Usually sequentially allocated
- Some systems services have PIDs as well
  - 0: scheduler/swapper
  - 1: init
  - 2: pagedaemon

Related Functions

- #include <sys/types.h>
- #include <unistd.h>
- pid_t getpid(void);
  - Returns PID of calling process
- pid_t getppid(void);
  - Returns PID of parent of calling process
- uid_t getuid(void);
- uid_t geteuid(void);
- gid_t getgid(void);
- gid_t getegid(void)
  - Return real or effective UID or GID for calling process

Fork

- #include <sys/types.h>
- #include <unistd.h>
- pid_t fork(void);
- Fork creates a new process
  - The new process is an exact clone of the calling process
- This is the only way to create a process in Unix
- Fork returns
  - The PID of the newly created process to the *parent* process
  - 0 to the newly created *child* process

Fork details

- The child is a clone of the parent
- It has a *copy* of the parent's
  - Address space (heap, stack, variables, stdio bufs)
  - File descriptors
  - Code (may be shared, since it is read-only)
- After the fork() call, each process executes as though it was the one that called fork()
  - The only difference is the return value from fork()
  - Usually, different code paths are taken based on a test of the PID returned

File Sharing between Parent and Child

- Each process has its own file descriptors
- The underlying kernel structures for managing the files are shared
- Specifically, the offsets are shared
- This means that shared output to the same file will work correctly
- Important if stdout has been redirected to a file

Normal cases

- Input and output isn’t redirected, so it doesn’t matter
- Parent waits for child to finish
  - Parent gets updated file pointers when it resumes executing
- Child redirects it’s input/output so no shared file pointers

Other Shared Info

- Real & effective user and group IDs
- Supplementary group IDs
- Session ID
- Controlling terminal
- set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Resource limits

Things that are Not Inherited by the Child

- The return value from fork()
- The process IDs
- The process ID of the parent
- The accumulated CPU time
- File locks
- Pending alarms
- Pending signals

Exit

- Three ways to terminate normally and two ways to terminate abnormally
- Normal Termination
  - Return from main()
  - Call exit() (C library function)
    - Cleans up standard I/O then calls _exit()
  - Call _exit() (underlying system call)
- Abnormal Termination
  - Receive certain signals from parent or kernel
  - Call abort (sends SIGABRT to self)
- Termination status: exit parameter or other kernel-generated status

Process Termination Details

- When a process terminates, the parent receives SIGCHLD
- wait() allows a parent process to wait for a child process to terminate
- When a process terminates, the kernel maintains a small amount of info until the parent calls wait()
  - Such a process is a *zombie* until the parent calls wait()
- If the parent terminates first, child is inherited by init

When a Process Terminates

- The parent process receives a SIGCHLD
- Parent can
  - Ignore the signal (the default action), or
  - Set up a signal handler to be called when the signal arrives
- Use wait() to wait for the child to finish
  - Blocks parent
  - Returns when a child process terminates
    - Returns immediately if any child is a zombie
  - Returns child's PID

wait() and waitpid()

- #include <sys/types.h>
- #include <sys/wait.h>
- pid_t wait(int *statloc);
  - Wait for any child process to terminate
- pid_t waitpid(pid_t *pid*, int **statloc*, int *options*);
  - Wait for a specific child process to terminate
- Statloc contains the child's termination status (the child's parameter to exit(), possibly with extra information – see the man page (section 2) for wait())

Race Conditions

- A race condition is any situation where two (or more) processes access shared data, AND
- The outcome of the processing depends upon the order in which the processes execute
- Example: two processes do x=x+1, where x is a shared variable
- Need some form of *synchronization*
  - Signals
  - File locks
  - Semaphores
  - ...

Running a Different Program

- fork() allows us to clone a process
  - The clone is a duplicate of the parent
  - It runs the same program as the parent
- We want to be able to run different programs, not just clones
- exec() allows us to run a different program *in the current process*
  - Often closely follows a call to fork()
- fork() creates a new process, and exec() makes it run a new program
  - Same PID, new text, data, BSS, stack, heap

exec()

- #include <unistd.h>
- int execl(const char *_pathname_, const char *_arg0_, ... /* (char *)0 */);
- int execv(const char *_pathname_, char const *_argv_[]);
- int execle(const char *_pathname_, const char *_arg0_, .../*(char *)0 */, char *const _envp_[]);
- int execve(const char *_pathname_, char const *_argv_[], char *const _envp_[]);
- int execlp(const char *_filename_, const char *_arg0_, ... /* (char *) 0 */);
- int execvp(const char *_filename_, char *const _argv_[]);

Variations of exec()

- l versions use a list of parameters
- v versions use an argv[] parameter
- e versions include an environment parameter
- p versions search PATH for executable
  - Probably the one you want for assignment 4

Changing User and Group IDs

- #include <sys/types.h>
- #include <unistd.h>
- int setuid(uid_t *uid*);
- int setgid(gid_t *gid*);
- If the process has superuser privileges
  - setuid() sets the real user ID, effective user ID, and saved set-user-ID to *uid*
- If the process does not have superuser privileges, but *uid* = the real user ID or the save set-user-ID
  - setuid sets the effective user ID to uid
- If neither is true, errno is set to EPERM and an error is returned

Facts about the three user IDs

- Only a superuser process can change the real user ID
- The effective user ID is set by the exec functions, only if the setuid bit is set for the program file
  - Can call setuid any time to set the effective user ID to the real user ID or the saved set-user-ID
- The saved set-user-ID is copied from the effective user ID by exec

Other functions

- int setreuid(uid_t *ruid*, uid_t *euid*);
- int setregid(gid_t *rgid*, gid_t *egid*);
- Swap real and effective user/group IDs
- int seteuid(uid_t *uid*);
- int setegid(gid_t *gid*);
- Set effective user/group ID

Interpreter Files

- Text files with: #! pathname [args] on the first line
- Recognized by the kernel
- Kernel starts the interpreter specified by pathname, and
- Redirects the rest of the file to the interpreter’s stdin

System()

- #include <stdlib.h>
- int system(char **cmdstring*);
- Does fork(), exec(), and waitpid() to execute cmdstring
- Waits for any old child to finish
- Don’t call system in a set-user-ID program

Process Times

- #include <sys/times.h>
- clock_t times(struct tms *buf);
- Fills in the tms struct and returns the current clock time (in seconds)
- struct tms {
  - clock_t tms_utime; // user CPU time
  - clock_t tms_stime; // system CPU time
  - clock_t tms_cutime; // user CPU time of terminated child processes
  - clock_t tms_cstime; // system CPU time of terminated child processes
- }

/proc

- See man –s4 proc
- Provides access to the state of each process and light-weight process in the system
- The name of the entry for a process is /proc/pid, where pid is the PID of the process
- Actual process state is contained in files in that directory
- The owner of the files is determined by the user ID of the process it describes

Accessing /proc

- Standard system calls are used to access /proc: open(), close(), read(), and write()
- Most files can only be opened for reading
- ctl and lwpctl (control) files can only be opened for writing
- as (address space) files contain the image of the running process and can be opened for reading and writing
  - Data can be transferred to and from the address space using read and write
- Files can be opened exclusively with O_EXCL
  - Advisory, i.e. only works if everyone does it

Information and Control Operations

- #include <procfs.h>
  - Contains definitions of data structures and message formats used with these files
- Every process contains at least one LWP
  - Each LWP represents a flow of execution that is independently scheduled by the OS
  - All LWPs in a process share its address space and many other attributes
  - We should stop and discuss threads here

/proc Directory Structure

- as (R/W): address space image, can seek
- ctl (W): messages can be written to control process state or behavior
- status (R): state information
- psinfo (R): miscellaneous information
- cred (R): description of the credentials
- sigact (R): array of sigaction structures
- map (R): virtual address map
- fd (R): directory containing references to open files
- usage (R): usage info (times, faults, blocks, msgs, sigs, syscalls, context switches)

- typedef struct pstatus {
- int pr_flags; /* flags (see below) */
- int pr_nlwp; /* number of lwps in the process */
- pid_tpr_pid; /* process id */
- pid_tpr_ppid; /* parent process id */
- pid_tpr_pgid; /* process group id */
- pid_tpr_sid; /* session id */
- id_t pr_aslwpid; /* lwp-id of the aslwp, if any */
- id_t pr_agentid; /* lwp-id of the agent lwp, if any */
- sigset_t pr_sigpend; /* set of process pending signals */
- uintptr_t pr_brkbase; /* virtual address of the process heap */
- size_t pr_brksize; /* size of the process heap, in bytes */
- uintptr_t pr_stkbase; /* virtual address of the process stack */
- size_tpr_stksize; /* size of the process stack, in bytes */
- timestruc_t pr_utime; /* process user cpu time */
- timestruc_t pr_stime; /* process system cpu time */
- timestruc_t pr_cutime; /* sum of children's user times */
- timestruc_t r_cstime; /* sum of children's system times */
- sigset_t pr_sigtrace; /* set of traced signals */
- fltset_t pr_flttrace; /* set of traced faults */
- sysset_t pr_sysentry; /* set of system calls traced on entry */
- sysset_t pr_sysexit; /* set of system calls traced on exit */
- char pr_dmodel; /* data model of the process */
- taskid_t pr_taskid; /* task id */
- projid_t pr_projid; /* project id */
- lwpstatus_t pr_lwp; /* status of the representative lwp */
- } pstatus_t;

- typedef struct psinfo {
- int pr_flag; /* process flags */
- int pr_nlwp; /* number of lwps in the process */
- pid_t pr_pid; /* process id */
- pid_t pr_ppid; /* process id of parent */
- pid_t pr_pgid; /* process id of process group leader */
- pid_t pr_sid; /* session id */
- uid_t pr_uid; /* real user id */
- uid_t pr_euid; /* effective user id */
- gid_t pr_gid; /* real group id */
- gid_t pr_egid; /* effective group id */
- uintptr_t pr_addr; /* address of process */
- size_t pr_size; /* size of process image in Kbytes */
- size_t pr_rssize; /* resident set size in Kbytes */
- dev_t pr_ttydev; /* controlling tty device (or PRNODEV) */
- ushort_t pr_pctcpu; /* % of recent cpu time used by all lwps */
- ushort_t pr_pctmem; /* % of system memory used by process */
- timestruc_t pr_start; /* process start time, from the epoch */
- timestruc_t pr_time; /* cpu time for this process */
- timestruc_t pr_ctime; /* cpu time for reaped children */
- taskid_t pr_taskid; /* task id */
- projid_t pr_projid; /* project id */
- char pr_fname[PRFNSZ]; /* name of exec'ed file */
- char pr_psargs[PRARGSZ]; /* initial characters of arg list */
- int pr_wstat; /* if zombie, the wait() status */
- int pr_argc; /* initial argument count */
- uintptr_t pr_argv; /* address of initial argument vector */
- uintptr_t pr_envp; /* address of initial environment vector */
- char pr_dmodel; /* data model of the process */
- lwpsinfo_t pr_lwp; /* information for representative lwp */
- } psinfo_t;