CU CS 4995-03: Internet Systems Programming

Process Control

Overview

- process identifiers
- signals
- signal recovery: `setjmp` and `longjmp`

Process Identifiers

```
pid_t getpid(void);    /* process identifier */
pid_t getppid(void);   /* parent pid */
uit_t getuid(void);    /* real user id */
uid_t geteuid(void);   /* effective user id */
gid_t getgid(void);    /* real group id */
gid_t getegid(void);   /* effective group id */
```

Create a process: `fork`

```
int v = 42;
if ((pid = fork()) < 0) {
  perror("fork");
  exit(1);
}
else if (pid == 0) {
  printf("child %d of parent %d\n", getpid(), getppid());
  v++;
  glob++;
}
else sleep(10);
```

copy-on-write

Inheritance

- user and group ids
- process group id
- controlling terminal
- setuid flag
- current working directory
- root directory (chroot)
- file mode creation mask
- signal masks

- close-on-exec flag
- environment
- shared memory
- resource limits

Differences Parent – Child

- return value of `fork`
- process IDs and parent process IDs
- accounting information
- file locks
- pending alarms

File Sharing between Processes

- share *same* file table entry
- thus, `seek()` and `read()` affects both parent and child process(es)
- motivation?

Waiting for a Child to Terminate

```
if (wait(&status) != pid) {
}
```

- status: core file, signals, exit status

waitpid

```
pid_t waitpid(pid_t pid, int *statloc, int options);
```

-1	any child process
> 0	specific process
0	same process group
< -1	process group = $\|pid\|$

options: avoid blocking

Race conditions

- Race =
- can't predict if parent or child will run first after `fork()`!
- parent may sometimes exit, sometimes not ...
- how can child wait for *parent* to terminate?

`exec`

- replace current process by new program
- `vfork`

```
int execl(char *path, char *arg, ...);
int execv(char *path, char *argv[]);
int execvp(char *file, char *argv[]);
```

`PATH` (execvp) or absolute path name

Changing User and Group

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

	setuid off	on	setuid(u) root	other
real	same	same	u	same
effective	same	uid(file)	u	u if =saved, real
saved	e	e	u	same

Examples of setuid

- remote shell, tip, . . .
- suexec for cgi scripts
- web server
-

Interpreter files

- text files beginning with `#!` *pathname arg*
- needs absolute path
- invokes with *arg intfile original args*
- could let the shell do it
- interpreters must use # as comment character!

```
#! /bin/sh
# The next line is executed by /bin/sh, but not Tcl \
exec tclsh $0 ${1+"$@"}
```

`system`

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

- handled by shell (/usr/bin/ksh)
- *never* call from setuid programs

Process Groups

```
pid_t getpgrp(void);
```

- process group leader: pid == process group id
- signals
- `waitpid`

Signals

- software interrupts for *asynchronous* events
- no information beyond signal name (integer)
  - control keys on terminal
  - hardware exceptions: divide by 0, invalid memory reference (SIGSEGV), unaligned access (SIGBUS)
  - `kill()` or `kill`
  - `raise(sig) = kill(getpid(), sig)` send to self
  - software conditions: `SIGURG`, `SIGPIPE`, `SIGALRM`, `SIGCHLD`

Signal Handling

- ignore signals (except `SIGKILL` or `SIGSTOP`)
- catch the signal with a *signal handler*
- default action

`signal()`

```
void (*signal(int signo, void (*func)(int)))(int);
```

- returns previous signal disposition
- function: `SIG_IGN`, `SIG_DFL`, or handler
- handler receives signal number and returns nothing
- either returns to location or `longjmp`
- only call *reentrant* functions (not malloc, I/O, etc.)

Non-Local Jumps

- `break`, `continue`, `return`
- `goto`: *within* same routine
- across routines: `setjmp`, `longjmp`:

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

`longjmp` returns to `setjmp`; setjmp returns *val*

Signal Example

```
if (signal(SIGUSR1, sigusr1) == SIG_ERR) {
  perror("signal");
}
if (setjmp(jmpbuffer) != 0) {
  printf("we are done!\n");
  exit(1);
}
while (1) {
  printf("looping...\n");
}
void sigusr1(int sig)
{
  longjmp(jmpbuffer, 1);
}
```

`longjmp`, `siglongjmp`

- careful: return from the wild
- example is not quite right
- `setjmp` saves stack frame, `sigsetjmp` saves registers, too
- declare variables as `volatile`
- can also save signal mask, priority

Reliable Signals

- old: action set to default when signal occurred
- need to “block” (not ignore) signals
- signal: generated – pending – delivered
- multiple occurrences before handler is completed

Signal Sets

```
sigemptyset(&newmask);
sigaddset(&newmask, SIGQUIT);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);
sigpending(&pendmask)
sigismember(&pendmask, SIGQUIT);
```

alarm

```
unsigned int alarm(unsigned int s);
```

- generates `SIGALRM` after s seconds; returns time to next alarm
- only *one* pending alarm, but can be replaced
- $s = 0$ cancels alarm
- `pause()` waits until signal

Timed I/O

- common problem: wait n s for user input, network packet, ...

```
static void sig_alrm(int signo)
{
  longjmp(env_alrm, 1);
}
...
if (signal(SIGALRM, sig_alrm) == SIG_ERR) {
  perror("signal");
}
setjmp(env_alrm);
alarm(10); /* wait ten seconds */
n = read(STDIN_FILENO, line, MAXLINE);
alarm(0);
```

```
write(STDOUT_FILENO, line, n);
```