Process Control

- Setting Effective IDs
- Interpreter Files
- Interpreter Files Continued
- Emulating Interpreter Files by `sh` Wrapping
- Process Accounting
- Get Login Name
- Process Times
- Time Measurements: Execution Log
- Time Measurements: Execution Log Continued
- Time Measurements Program
- Time Measurements Program Continued
- Text References

Overview

- Calls for creating processes.
- Calls for waiting for other processes to terminate.
- Calls for executing new programs.

Programs

- A program is a collection of instructions and data.
- Typically a program is written in a high-level language like C.
- C programs typically consist of a collection of `.c` and `.h` files.
- Typically (though not necessarily), only the `.c` files generate code. The `.h` files only contain declarations.
- Each `.c` file is compiled by a compiler into a object module (typical extension `.o`).
- A linker links all the object modules consituting the program into an *executable module*. It may also link static libraries into the executable module.
- The program loader loads the executable module into an *image* in memory. Any dynamically linked libraries may be loaded at this time too.
- The OS starts executing the program at `main()`. Runtime libraries may be linked as needed at runtime.

Programs Files

A program is an executable file (indicated via some kind of *magic* number). Contains:

- Executable code with start address.
- Initialized data.
- Size of uninitialized data.
- Symbol and relocation tables.
- Miscellaneous information.

There are formats which allow the same executable to run on different architectures (eg. on transition from classic Mac to OS/X).

Processes

- A *process* is an instance of a program which is executing.
- Each process has its own address space and executions state.
- Each process is uniquely identified by a process ID (PID), typically an integral type.
- The OS manages resource allocation requests by processes, including memory and I/O.
- A process has at least one *flow of control* called a *thread* of execution.
- Multiple processes may reside in memory simultaneously and execute concurrently (or simultaneously in MP systems).
- Processes can interact only by explicit communication using constructs like pipes, files, or network.

Process ID

- PID is a non-negative integer which uniquely identifies a process.
- PID has type `pid_t` which is an integral type (not necessarily `int`).
- Process 0 is the scheduler: a kernel process known as *swapper*.
- Process 1 is *init*: a user process which is the ancestor of all other processes. Runs with `root` privileges.
- Process 2 may be a kernel *pagedaemon* process.
- `getpid()`, `getppid()` return current process's PID and its parent's PID repectively.
- `getuid()`/`geteuid()` returns real/effective user ID of calling process.
- `getgid()`/`getegid()` returns real/effective group ID of calling process.

PID Example

Program pid.c:

```
int
main(void)
{
  printf("I am process %ld\n", (long)getpid());
  printf("My parent is %ld\n", (long)getppid());
  return 0;
}
```

```
$ ./programs/pid
I am process 18176
My parent is 23415
$
```

Memory Layout

```
high address +-------------------+
             |  Command-Line     |
             |  Arguments and    |   argv[argc] == NULL
             |  Envionment       |   (ANSI-C and POSIX).
             |                   |
             |-------------------|
             |                   |
             |  |  Stack         |
             |  |                |
             |--+----------------|
             |  |                |
             |  V                |
             |                   |
             |  ^                |
             |  |                |
             |--+----------------|
             |  |                |
             |  |  Heap          |
             |                   |
             |-------------------|
             | (uninitialized    | initialized
             |  data) bss        | to 0 by exec
             |                   |
             |-------------------|
             | initialized data  | read
             |                   | from
             |-------------------| program on
             |      text         | startup
             |                   |
 low address +-------------------+
```

Memory Layout Continued

- Most Unix systems provide `etext`, `edata`, `end` symbols giving location 1-beyond end of code area, end of of initialized data, end of uninitialized data. Declare using `extern char etext, edata, end;`. Get pointer using `&etext`.
- Stack contains frames for C run-time startup functions, followed by frame for `main()` followed by frames for currently active functions called from `main()`.
- Usually stack variables have high addresses, heap variables have lower addresses.

Starting/Stopping a Program

- `main()` is not the first function run. Instead an internal startup function is first run which calls `main()`.
- If `main()` returns, then the startup function calls `exit()` with return value.
- Program can also be terminated by a signal, or via `abort()`. This may create a `core` dump.

Environment List

- Environment list is simply a sequence of strings of the form *Name*=*Value*.
- Individual environment variables can be queried using:

```
char *getenv(const char *name);
```

- Individual environment variables can be changed using:

```
int putenv(const char *nameValue);
int setenv(const char *name,
           const char *val,
           int rewrite);
void unsetenv(const char *name);
```

- The environment is stored in the global variable `extern char **environ;`. This is useful for iterating over the entire environment.
- Some systems allow specifying environment using an optional 3rd argument to `main()`.

Process States

New
: Process is being created.

Running
: Instructions from process are being executed.

Blocked
: The process is waiting for an event such as I/O (or has voluntarily blocked using `sleep()`).

Ready
: The process is waiting to be assigned a processor.

Done
: The process has completed.

Context switch replaces a *running* process with another process from the *ready* state.

`ps` command displays process status. Non-portable options.

`fork()`

- Returns twice.
- Returns child's PID to parent.
- Returns 0 to child.
- A child can always discover it and its parent's PID using `getpid()` and `getppid()` respectively.
- Conceptually, the parent and child have totally separate copies of the program memory.
- Usually, parent and child share the text segment if it is read-only.
- If the system uses *Copy-on-Write*, then the data segment can also be shared, with pages being lazily copied in the child only when they are modified.

Example

```
int glob = 6;   /* external variable in initialized data */
char buf[] = "a write to stdout\n";

int
main(void)
{
  int var= 88;  /* automatic variable on the stack */
  pid_t pid;

  if (write(1, buf, sizeof(buf)-1) != sizeof(buf)-1) {
    perror("write error"); exit(1);
  }
  printf("before fork\n");  /* we don’t flush stdout */

  if ( (pid = fork()) < 0) {
    perror("fork error"); exit(1);
  }
  else if (pid == 0) { /* child */
    glob++;     /* modify variables */
    var++;
  }
  else {
    sleep(2);                           /* parent */
  }
  /* executed by both parent and child */
  printf("pid = %d, glob = %d, var = %d\n",
         getpid(), glob, var);
  exit(0);
}
```

Example Execution

```
$ ./fork1
a write to stdout
before fork
pid = 661, glob = 7, var = 89
pid = 660, glob = 6, var = 88
$ ./fork1 >t
$ cat t
a write to stdout
before fork
pid = 671, glob = 7, var = 89
before fork
pid = 670, glob = 6, var = 88
$
```

Process Chain

Program chain.c:

```
int
main(int argc, char *argv[])
{

  if (argc != 2) {
    fprintf(stderr, "usage: %s N_PROCESSES\n", argv[0]);
    exit(1);
  }
  else {
    int n = atoi(argv[1]);
    int i;
    pid_t pid;
    for (i = 0; i < n; i++) {
      pid = fork();
      if (pid < 0) {
        fprintf(stderr, "fork error: %s\n", strerror(errno));
        exit(1);
      }
      else if (pid > 0) {
        break;
      }
    }
    printf("i:%d; PID: %ld; PPID: %ld; child PID: %ld\n",
           i, (long)getpid(), (long)getppid(), (long)pid);
  }
  return 0;
}
```

Process Chain Log

```
$ ./programs/chain
usage: ./programs/chain N_PROCESSES
$ ./programs/chain 4
i:2; PID: 18299; PPID: 18298; child PID: 18300
i:1; PID: 18298; PPID: 18297; child PID: 18299
i:0; PID: 18297; PPID: 23415; child PID: 18298
$ i:4; PID: 18301; PPID: 18300; child PID: 0
i:3; PID: 18300; PPID: 1; child PID: 18301

$
```

Process Fan

Program fan.c:

```
int
main(int argc, char *argv[])
{

  if (argc != 2) {
    fprintf(stderr, "usage: %s N_PROCESSES\n", argv[0]);
    exit(1);
  }
  else {
    int n = atoi(argv[1]);
    int i;
    pid_t pid;
    for (i = 0; i < n; i++) {
      pid = fork();
      if (pid < 0) {
        fprintf(stderr, "fork error: %s\n", strerror(errno));
        exit(1);
      }
      else if (pid == 0) {
        break;
      }
    }
    printf("i:%d; PID: %ld; PPID: %ld; child PID: %ld\n",
           i, (long)getpid(), (long)getppid(), (long)pid);
  }
  return 0;
}
```

Process Fan Log

```
$ ./programs/fan 4
i:0; PID: 18309; PPID: 18308; child PID: 0
i:1; PID: 18310; PPID: 18308; child PID: 0
i:2; PID: 18311; PPID: 18308; child PID: 0
i:3; PID: 18312; PPID: 18308; child PID: 0
i:4; PID: 18308; PPID: 23415; child PID: 18312
$
```

Properties Inherited by Child

- Open files.
- Real/effective uid/gid, supplementary gids.
- Process group id.
- Session id.
- Flags for setuid/setgid.
- Working and root directories.
- File mode creation `umask`.
- Signal masks and dispositions.
- Environment.
- Attached shared memory segments and resource limits.

Parent-Child Differences

- `fork()` return value.
- PID's.
- Parent PID's.
- Child time statistics (`tms_utime`, `tms_stime`, `tms_cutime`, `tms_ustime`) set to 0.
- File locks in parent are turned off for child.
- Pending alarms cleared for the child.
- Set of pending signals are cleared for the child.

stdio buffers and `fork()`

Consider following program:

```
int
main(int argc, const char *argv[])
{
  printf("hello world");
  if (argc > 1) fflush(stdout);
  pid_t pid = fork();
  if (pid < 0) {
    perror("could not fork:");
  }
  return 0;
}
```

stdio buffers and `fork()` Log

```
$ ./fork-stdio-buffer
hello worldhello world$ ./fork-stdio-buffer 1
hello world$
```

Process Termination

- Normal Termination:
    1. `return` from `main()`.
    2. `exit()`. ANSI-C.
    3. `_exit`(). Unix specific.
- Abnormal Termination
    1. `abort()`. Generates `SIGABRT` and generates a `core` dump.
    2. Process receives a signal which was not caught.
- Termination status of child can be accessed by parent.

Child Termination

- If a child terminates before the parent, then the system must retain vestigial information (like its PID and termination status) so that the parent can access the termination status of the child.
- A child process which has terminated but has not had its termination status examined by its parent is known as a *zombie*.
- If a parent terminates before a child, then the child is adopted by `init` (with PID 1).
- When a orphaned child terminates, it does not become a zombie, because `init` is written so as to always access the termination status of its children (including adopted children).

Synchronizing Parent with Child Termination

- Parent is sent a SIGCHLD signal whenever anyone of its children terminates.
- Parent can also use `wait()` or `waitpid()` to synchronize with child termination.
- Above wait functions return immediately with a error if the process has no children.
- The wait functions can be used to access the termination status of a child.
- Return value is < 0 on error, 0 (explained later), or PID of terminated child.

`wait()` Function

```
pid_t wait(int *stat)
```

- Blocks until a child terminates.
- Returns when **any** child terminates or return immediately if a child has already terminated (zombie).
- Allow the parent to access the termination status of the terminated child in `*stat`.

`waitpid()` Function

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

- Much more flexible than `wait()`.
- Can specify waiting for a particular child (`pid > 0`), any child (`pid == -1`), any child with the same process group ID (`pid == 0`), or any child whose process group ID is `|pid|` (`pid < -1`).
- Allow the parent to access the termination status of the terminated child in `*stat`.
- Flags in `options` can specify nonblocking using `WNOHANG` (return value of 0), or `WUNTRACED` to get the status of a stopped process on a system which supports job control.

Decoding Child Termination Status

`WIFEXITED(status)`
: True if child terminated via `exit()` or `_exit`(). `WEXITSTATUS(status)` returns exit status.

`WIFSIGNALLED(status)`
: True if child was terminated by a uncaught signal. `WTERMSIG(status)` gives the signal number which caused termination. `WCOREDUMP(status)` is true if a `core` dump was produced.

`WIFSTOPPED(status)`
: True if child stopped. `WSTOPSIG(status)` retuns signal number.

Decoding Child Termination Status Routine

```
void
pr_exit(int status)
{
  if (WIFEXITED(status))
    printf("normal termination, "
           "exit status = %d\n",
           WEXITSTATUS(status));
  else if (WIFSIGNALED(status))
    printf("abnormal termination, "
           "signal number = %d%s\n",
           WTERMSIG(status),
#ifdef WCOREDUMP
           WCOREDUMP(status)
           ? " (core file generated)" : "");
#else
           "");
#endif
  else if (WIFSTOPPED(status))
     printf("child stopped, "
            "signal number = %d\n",
            WSTOPSIG(status));
}
```

Multiple Child Exits Example

```
static EndFnP fns[]= { exitFn, abortFn, div0Fn };

int main(void)
{
  int i;
  for (i= 0; i < sizeof(fns)/sizeof(fns[0]); i++) {
    int pid= fork();
    if (pid < 0) {
      perror("fork error"); exit(1);
    }
    else if (pid == 0) { /* child */
      (fns[i])(i+7);
    }
    else { /* parent */
      int status;
      if (wait(&status) != pid) {
        perror("wait error"); exit(1);
      }
      pr_exit(status);
    }

  }
}
```

Multiple Child Exits Execution

```
$ ./wait1
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGIOT       7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR
$
```

Avoiding Zombies

```
int
main(void)
{
  pid_t pid;

  if ( (pid = fork()) < 0)
    err_sys("fork error");
  else if (pid == 0) {          /* first child */
    if ( (pid = fork()) < 0)
      err_sys("fork error");
    else if (pid > 0)
      exit(0);  /* parent from 2nd fork == 1st child */

    /* second child. */
    sleep(2);
    printf("second child, parent pid = %d\n",
           getppid());
    exit(0);
  }

  /* wait for first child */
  if (waitpid(pid, NULL, 0) != pid)
    err_sys("waitpid error");

  /* original process */

  exit(0);
}
```

`exec()` System Calls

- Replaces memory image of currently executing process with image corresponding to specified program.
- If successful, then it does not return. Instead, it starts executing newly loaded program at `main()`.
- Only way to start new programs in Unix (except for booting of Unix kernel itself).
- Usually used in conjunction with `fork()`.

`exec()` System Call Variants

- No system call called `exec()`, as such. Instead, `exec()` is used loosely to refer to 6 system calls of the form `exec`*AB*`()`, for *A* in {`l`, `v`} (where `l` means *list* and `v` means *vector*), and *B* is empty or in {`p`, `e`} (where `p` means *path* and `e` means *environment*).
- 6 variants are `execl()`, `execv()`, `execlp()`, `execvp()`, `execle()`, `execve()`.

`exec()` List/Vector Variants

```
int execl(const char *path, const char *arg0, ...);
int execv(const char *path, char *const argv[]);
```

- For `execl()` (also `execlp()` and `execle()`), arguments to program are passed separately and are terminated by a `NULL` argument. The # of arguments must be known at compile-time.
- For `execv()` (also `execvp()` and `execve()`), arguments to program are passed in an array with last element `NULL`. The # of arguments need not be known at compile-time.

`exec()` Path Variants

```
int execlp(const char *file, const char *arg0, ...);
int execvp(const char *file, char *const argv[]);
```

- For `execlp()` and `execvp()`, if the `file` argument does not contain any `/` characters, then the `PATH` environmental variable is used to search for an executable `file`.
- If the file is found, but does not have the correct permissions, then errno is set to EACCES and the search continues.
- If the file is found but is not in executable format (`errno ENOEXEC`), then the shell attempts to execute the file. If that attempt fails, then the `exec()` fails.

`exec()` Environment Variants Continued

```
int execle(const char *path, const char *arg0, ...,
           char *const env[]);
int execve(const char *path, char *const argv[],
           char *const env[]);
```

- Additional argument of `NULL`-terminated `env` array.

Inheritance Across `exec()`

Most process attributes are inherited across `exec()`. However, if an attribute value entails the original memory image, then obviously the value cannot be inherited and is reset to some default value.

- Inherited attributes include PID, PPID, PGID, SID, TTY, UID, GID, TTY, cwd, root directory, priority, accumulated execution times, file descriptors (unless CLOEXEC flag set).
- Signal handlers reset to default (meaningless in new memory image).
- Effective UID/GID changed if SETUID/SETGID bit set on executable.
- `atexit()` handlers are unregistered.
- Shared memory segments are detached.

System Function

```
int
system(const char *cmdstring)   /* no signal handling */
{
  pid_t pid;
  int           status;

  if (cmdstring == NULL) return(1);

  if ( (pid = fork()) < 0) {
    status = -1; /* probably out of processes */
  } else if (pid == 0) { /* child */
    execl("/bin/sh", "sh", "-c",
          cmdstring, (char *) 0);
    _exit(127);  /* execl error */
  }
  else { /* parent */
    while (waitpid(pid, &status, 0) < 0) {
      if (errno != EINTR) {
        status = -1; /* error other than */
        break;       /* EINTR from waitpid() */
      }
    }
  }
  return(status);
}
```

Changing User and Group IDs

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

- 3 user/group IDs: *real*, *effective* and *saved-set*.
- If used by `root`, `setuid()`/`setgid()` sets all 3 ids.
- If used by non-`root`, `setuid(`uid`)` sets effective *uid* to `uid` if `uid` is real or saved-set uid; similarly for `setgid(`gid`)`.
- Else error with `errno` set to `EPERM`.
- Saved-set uid copied from effective uid by `exec()` **after** set-user-ID bit applied.
- Saved-set uid used by programs like `tip`, which run setuid `uucp`, lock files as `uucp`, but need to run with effective uid equal to real uid, and then need to revert to effective uid `uucp` to unlock files.

Changing Real/Effective IDs

```
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

- Can be used to swap real/effective user/group IDs.
- Can be used to allow a set-uid program to run with normal privileges and then swap back again for set-uid privileges.
- On many systems if an argument is specified as `-1`, then the corresponding id is not changed.

Setting Effective IDs

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

- Changes effective ID.
- Non-`root` process can only change effective uid/gid to saved-set uid/gid.
- On some systems, `seteuid(`uid`)` equivalent to `setreuid(-`1, uid`)`. Similarly for `setegid()`.

Interpreter Files

- Normally, if a file which is `exec()`'d is not in executable format, then it is fed as input to `/bin/sh`.
- On many systems, if the first line of the file starts with the characters `#!`, then that line specifies an interpreter which is used to interpret the file.

```
#!/bin/sh
#!/bin/awk -f
#!/usr/local/bin/perl -ws
```

- Interpreter path must be absolute.
- On many systems only a single argument can be specified.
- Many systems have a 32 character limit on the length of the first line (precludes long paths for the interpreter).

Interpreter Files Continued

- In general, subject to system limits, if an interpreter file `intfile` starts with the line `#!INTERPRETER_LINE`, then executing `intfile arg1 arg2` is equivalent to `INTERPRETER_LINE intfile arg1 arg2`.
- Allows hiding whether a file is an executable file or interpreter file.
- Kernel implements interpreter file feature.

Emulating Interpreter Files by `sh` Wrapping

If interpreter file feature is unavailable, then equivalent functionality can be obtained by wrapping file contents in a shell script.

```
$ cat t
/usr/bin/perl <<’END_PERL’ - $@

print "hello world\n";

print "my args are @ARGV\n";
END_PERL
$ ./t a b c d
hello world
my args are a b c d
$
```

Wrapping in a shell script is inefficient because of repeated `exec()`’s.

Process Accounting

- Typically `/var/adm/pacct` contains process accounting information, if process accounting is turned on by `accton(1)`.
- Contains information like termination status, real user/group ID, controlling terminal, starting calendar time, user/system/elapsed CPU time in clock ticks, average memory usage, # of bytes/blocks of I/O, command name.
- Accounting record is written when the process is terminated, not when it is started.
- Corresponds to processes, not programs. If `A` `exec()`'s `B` which `exec()`'s `C` which `exit()`'s, then accounting record is written for `C`, but contains sum of resources used by `A`, `B` and `C`.

Get Login Name

- If a program has a shell escape feature, then we may need the login name to decide which shell to run for user.
- Could use `getpwuid(getuid())->pw_name` to get login-name of the user running the program, but this may be wrong if there are multiple login-name’s for the same uid.
- Use `char *getlogin(void)` to get the login name the user is logged in under.
- Can fail if the process is not attached to a terminal.

Process Times

```
struct tms {
  clock_t tms_utime;    /* user */
  clock_t tms_stime;    /* system */
  clock_t tms_cutime;   /* child user */
  clock_t tms_cstime;   /* child system */
};

clock_t times(struct tms *buf);
```

- `times()` returns wall clock time.
- Returns time only for children waited for.

Time Measurements: Execution Log

```
$ ./times "sleep 5" date "echo ’Hello world’"
***Command: sleep 5
             elapsed:    5.27
                user:    0.00
              system:    0.00
          child user:    0.01
        child system:    0.01
***Command: date O
Tue Oct 21 22:50:27 EST 2008
             elapsed:    0.08
                user:    0.00
              system:    0.00
          child user:    0.02
        child system:    0.01
***Command: echo ’Hello world’
Hello world
             elapsed:    0.02
                user:    0.00
              system:    0.00
          child user:    0.00
        child system:    0.01
```

Time Measurements: Execution Log Continued

```
$ ./times "find ~zdu/devel -type f -exec grep xxx {} \; >/dev/null"
***Command: find ~zdu/devel -type f -exec grep xxx {} \; >/dev/null
             elapsed:   81.45
                user:    0.00
              system:    0.00
          child user:   11.31
        child system:   16.08
$
```

Time Measurements Program

```
#define SYS_ERR(m)      \
  do { perror(m); exit(1); } while (0)

static void
printTimes(clock_t elapsed,
           const struct tms *tmsBeginP,
           const struct tms *tmsEndP)
{
  printf("%20s: %7.2f\n", "elapsed",
         elapsed*1.0/CLK_TCK);
  printf("%20s: %7.2f\n", "user",
         (tmsEndP->tms_utime -
          tmsBeginP->tms_utime)*1.0/CLK_TCK);
  printf("%20s: %7.2f\n", "system",
         (tmsEndP->tms_stime -
          tmsBeginP->tms_stime)*1.0/CLK_TCK);
  printf("%20s: %7.2f\n", "child user",
         (tmsEndP->tms_cutime -
          tmsBeginP->tms_cutime)*1.0/CLK_TCK);
  printf("%20s: %7.2f\n", "child system",
         (tmsEndP->tms_cstime -
          tmsBeginP->tms_cstime)*1.0/CLK_TCK);
}
```

Time Measurements Program Continued

```
static void
doCmd(char *cmd)
{
  struct tms tmsBegin, tmsEnd;
  clock_t tBegin, tEnd;
  printf("***Command: %s\n", cmd);
  if ((tBegin= times(&tmsBegin)) < 0)
    SYS_ERR("times()");
  if (system(cmd) != 0) SYS_ERR("system()");
  if ((tEnd= times(&tmsEnd)) < 0)
    SYS_ERR("times()");
  printTimes(tEnd - tBegin, &tmsBegin, &tmsEnd);
}

int
main(int argc, char *argv[])
{
  int i;
  for (i= 1; i < argc; i++) doCmd(argv[i]);
}
```

Text References

Text, Chs 6, 24, 25, 26, 27.

APUE, Ch. 8.