Process Control

Advanced Programming in the UNIX Environment

Chun-Ying Huang <chuang@cs.nctu.edu.tw>

Outline

Overview

Process creation

Process termination

Program execution

Process Identifiers

Every process has a unique process ID

- A non-negative integer
- Process ID can be reused after a process has terminated

The init program (/sbin/init)

- Bring up the system - /etc/inittab, /etc/rc*, or /etc/events.d
- The init process never dies
- The parent process of all orphaned process

List of Running Processes – ps

The `ps` command

Something like "Task Manager" in Windows

An example: "`ps au`" output

- List user-oriented processes with terminal attached

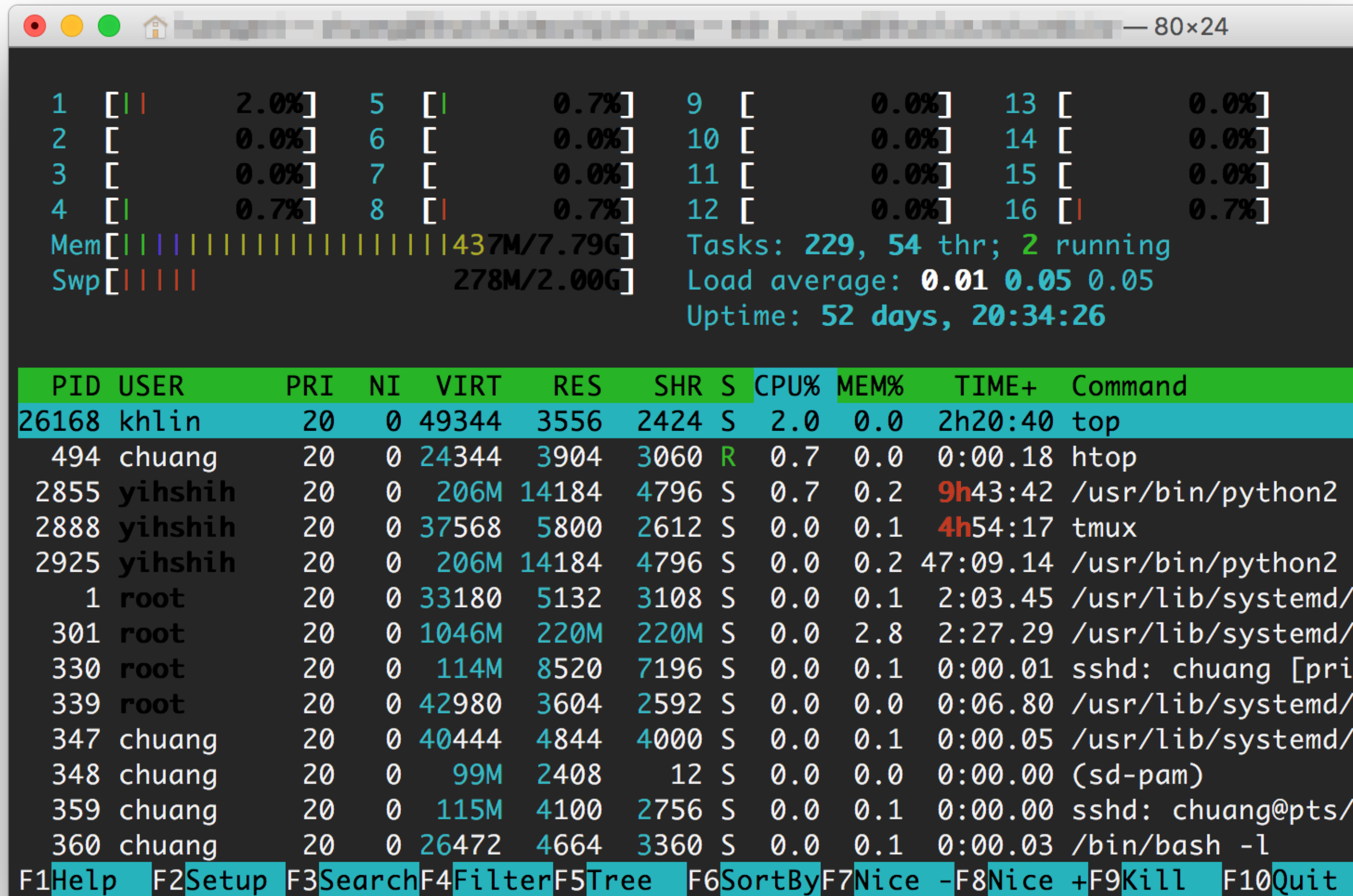
```
$ ps au
USER       PID %CPU %MEM   VSZ   RSS TTY      STAT START   TIME COMMAND
root      4608  0.0  0.1  1780   532 tty4     Ss+  11:02   0:00 /sbin/getty 38400 tty4
root      4609  0.0  0.1  1780   540 tty5     Ss+  11:02   0:00 /sbin/getty 38400 tty5
root      4616  0.0  0.1  1780   540 tty2     Ss+  11:02   0:00 /sbin/getty 38400 tty2
root      4619  0.0  0.1  1780   540 tty3     Ss+  11:02   0:00 /sbin/getty 38400 tty3
root      4622  0.0  0.1  1780   540 tty6     Ss+  11:02   0:00 /sbin/getty 38400 tty6
root      6104  0.0  0.1  1780   536 tty1     Ss+  11:08   0:00 /sbin/getty 38400 tty1
root      7237  0.8  2.3 20128 12168 tty7     Ss+  11:15   2:39 /usr/X11R6/bin/X :0 -br -audit
huangant  7478  0.0  0.5  5676  3076 pts/0    Ss   11:16   0:01 bash
huangant  9273  0.0  0.6  5756  3156 pts/1    Ss+  13:07   0:00 bash
huangant 11906  0.0  0.1  2744  1016 pts/0    R+   16:39   0:00 ps au
```

List of Running Processes – top

```
top - 13:01:55 up 5 days, 19:57,  2 users,  load average: 0.05, 0.07, 0.05
Tasks: 217 total,   1 running, 216 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.5 us,  0.5 sy,  0.0 ni, 98.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  24628860 total,  4576008 used, 20052852 free,  1443900 buffers
KiB Swap: 23435256 total,        0 used, 23435256 free.  2096572 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
10000 www-data  20   0 1322848  69912  12084 S  14.0  0.3 838:09.39 imageserver
    7 root      20   0       0      0      0 S   0.3  0.0   7:37.43 rcu_sched
 2916 gdm       20   0   39236   2640   2180 S   0.3  0.0   2:07.54 dbus-daemon
 2935 gdm       20   0  700840  23776  18016 S   0.3  0.1  13:07.56 gnome-sett+
    1 root      20   0   34184   4680   2696 S   0.0  0.0   0:01.10 init
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.01 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:02.32 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:+
    8 root      20   0       0      0      0 S   0.0  0.0   2:44.74 rcuos/0
    9 root      20   0       0      0      0 S   0.0  0.0   2:45.73 rcuos/1
   10 root      20   0       0      0      0 S   0.0  0.0   2:19.19 rcuos/2
   11 root      20   0       0      0      0 S   0.0  0.0   2:06.06 rcuos/3
   12 root      20   0       0      0      0 S   0.0  0.0   0:05.88 rcuos/4
   13 root      20   0       0      0      0 S   0.0  0.0   0:21.18 rcuos/5
   14 root      20   0       0      0      0 S   0.0  0.0   0:07.18 rcuos/6
   15 root      20   0       0      0      0 S   0.0  0.0   0:04.43 rcuos/7
   16 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
```

List of Running Processes – htop



Process Relationships

Tree structure

The `pstree` command

The `init` process

- The 1st process in most Linux systems
- Usually has a PID of 1

Some systems uses 'systemd' to replace the old 'init'

```
init-+-NetworkManager
     |-acpid
     |-atd
     |-cron
     |-cupsd
     |-2*[dbus-daemon]
     |-dbus-launch
     |-6*[getty]
     |-gnome-settings----{gnome-settings-}
     |-gnome-terminal-+-bash---pstree
     |                |-bash
     |                |-gnome-pty-helpe
     |                `-{gnome-terminal}
     |-hald---hald-runner-+-hald-addon-acpi
     |                    |-hald-addon-inpu
     |                    `-hald-addon-stor
     |-klogd
     |-syslogd
     |-system-tools-ba
     |-udevd
     `-vmware-guestd
```

Retrieve Process Identifiers

Synopsis

- pid_t getpid(void);
- pid_t getppid(void);
- uid_t getuid(void);
- uid_t geteuid(void);
- gid_t getgid(void);
- gid_t getegid(void);

None of these functions has an error return

Process Creation

The fork Function

Create a new (child) process, synopsis

- pid_t fork(void);
- Returns: 0 in child, process ID of child in parent, -1 on error

Both the child and the parent continue executing with the instruction that follows the call to fork

The child is a copy of the parent

- The child gets a copy of the parent's data space, heap, and stack
- The parent and the child do not share these portions of memory, but they share the text segment
- Since a fork is often followed by an exec, a technique called copy-on-write (COW) is used to

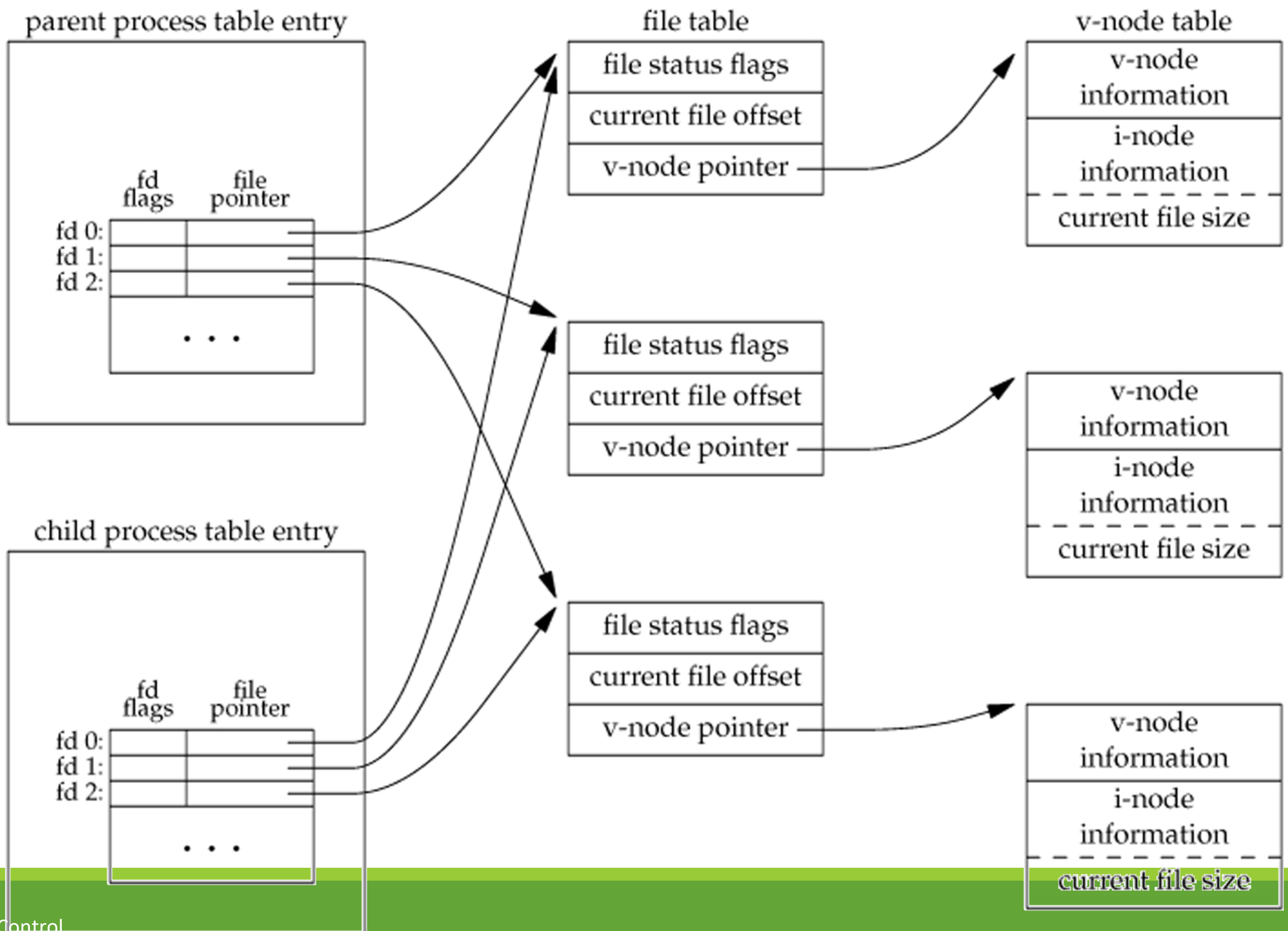
A fork Example

```
#include "apue.h“
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(STDOUT_FILENO, buf, sizeof(buf)-1)!=sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");        /* we don't flush stdout */
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {          /* child */
        glob++;                     /* modify variables */
        var++;
    } else {
        sleep(2);                   /* parent */
    }
    printf("pid=%d, glob=%d, var=%d\n", getpid(), glob, var);
    exit(0);
}
```

A fork Example (Cont’d)

$ ./fig8.1-fork1	*terminal devices are line buffered*
a write to stdout	
before fork	
pid = 430, glob = 7, var = 89	*child's variables were changed*
pid = 429, glob = 6, var = 88	*parent's copy was not changed*
$ ./fig8.1-fork1 > temp.out	*non-terminal devices are fully buffered*
$ cat temp.out	
a write to stdout	
before fork	
pid = 432, glob = 7, var = 89	
before fork	
pid = 431, glob = 6, var = 88	

fork and File Sharing



Handling File Descriptors after fork

The parent waits for the child to complete

- The parent does not need to do anything with its descriptors
- Any of the shared descriptors that the child reads from or writes to have their file offsets updated accordingly

Both the parent and the child go their own ways

- After the fork, the parent closes the descriptors that it doesn't need
- The child does the same thing
- This scenario is often the case with network servers

Other Properties Inherited by the Child

Real user ID, real group ID, effective user ID, effective group ID

Supplementary group IDs

Controlling terminal

The set-user-ID and set-group-ID flags

Current working directory

File mode creation mask

Signal mask and dispositions

The close-on-exec flag for any open file descriptors

Environment variables

…

Uses of fork

When a process wants to duplicate itself

- The parent and child can each execute different sections of code at the same time
- This is common for network servers
  - The parent waits for a service request from a client
  - When the request arrives, the parent calls fork and lets the child handle the request
  - The parent goes back to waiting for the next service request to arrive

When a process wants to execute a different program

- This is common for shells
  - the child does an exec right after it returns from the fork

Variants of fork

vfork

- Creates a child process of the calling process without copying the address space of the parent into the child
- Usually used when the child simply calls exec (or exit) right after the vfork
- While the child is running and until it calls either exec or exit, the child runs in the address space of the parent
- More efficient than use fork – no copy is better than some copies

clone

- Linux system calls for implementing fork and vfork
- A generalized form of fork that allows the caller to control what is shared between parent and child

Process Termination

Child Process Termination

Zombie process

- When a child process terminates, its exit status is expected to be read by its parent process
- If the parent process does not read the exit status, the child process becomes a zombie
  - Resources occupied by the child process are freed
  - But the PID and termination state are kept in the kernel

Guarantee the existence of parent processes

- If a parent process is terminated before its child processes
- The init process becomes the parent process of any process whose parent terminates
  - The parent process ID of the surviving process is changed to be 1

Child Process Termination (Cont’d)

When a child process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent

The termination of a child is an asynchronous event as it can happen at any time while the parent is running

This signal is the asynchronous notification from the kernel to the parent

The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs

- The signal handler function

The wait and waitpid Function

A parent process is able to call wait and waitpid functions to receive child process termination status

The two functions may …

- Block, if all of its children are still running
- Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
- Return immediately with an error, if it doesn't have any child processes

If the process calls wait on receipt of the SIGCHLD signal

- We expect wait to return immediately
- But if we call it at any random point of time, it might be blocked

The wait and waitpid Function (Cont'd)

Synopsis

- pid_t wait(int *status);
- pid_t waitpid(pid_t pid, int *status, int options);

The differences between these two functions

- Block or not block
  - The wait function always block the caller until a child process terminates
  - The waitpid function has an option that prevents it from being blocked
- Process termination order
  - The waitpid function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.

Macros to Interpret Exit Status

Macro	Description
WIFEXITED(status)	True if status was returned for a child that terminated normally. In this case, we can execute ***WEXITSTATUS(status)*** to fetch the low-order 8 bits of the argument that the child passed to exit, _exit,or _Exit.
WIFSIGNALED (status)	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute ***WTERMSIG(status)*** to fetch the signal number that caused the termination. Additionally, some implementations define the macro ***WCOREDUMP(status)*** that returns true if a core file of the terminated process was generated.
WIFSTOPPED (status)	True if status was returned for a child that is currently stopped. In this case, we can execute ***WSTOPSIG(status)*** to fetch the signal number that caused the child to stop.
WIFCONTINUED (status)	True if status was returned for a child that has been continued after a job control stop

wait and waitpid – an Example (1/3)

Print exit status

```
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),
            WCOREDUMP(status) ? " (core file generated)" : "");
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number=%d\n",
            WSTOPSIG(status));
}
```

wait and waitpid – an Example (2/3)

```
$ ./fig8.6-wait1
normal termination, exit status = 7
abnormal termination, signal number = 6
abnormal termination, signal number = 8
```

```
int main(void) {
    pid_t pid;int status;
    if ((pid = fork()) < 0)          err_sys("fork error");
    else if (pid == 0) /* child */   exit(7);
    if (wait(&status) != pid)        err_sys("wait error");
    pr_exit(status);                 /* and print its status */
    if ((pid = fork()) < 0)          err_sys("fork error");
    else if (pid == 0) /* child */   abort(); /* generates SIGABRT */
    if (wait(&status) != pid)        err_sys("wait error");
    pr_exit(status);                 /* and print its status */
    if ((pid = fork()) < 0)          err_sys("fork error");
    else if (pid == 0) /* child */   status /= 0;
                                     /* divide by 0 generates SIGFPE */
    if (wait(&status) != pid)        err_sys("wait error");
    pr_exit(status);                 /* and print its status */
    exit(0);
}
```

The waitpid Function

The wait function waits for any of the children

if we want to wait for a specific process to terminate, use waitpid instead

Synopsis, again

- pid_t waitpid(pid_t pid, int *status, int options);

The meaning of the argument ‘pid’

pid	Interpretation
< -1	Waits for any child whose process group ID equals the absolute value of pid.
== -1	Waits for any child process. In this respect, waitpid is equivalent to wait.
== 0	Waits for any child whose process group ID equals that of the calling process.
> 0	Waits for the child whose process ID equals pid.

The waitpid Function (Cont'd)

waitpid options

Constant	Description
WNOHANG	The waitpid function will not block if a child specified by pid is not immediately available. In this case, the return value is 0
WUNTRACED	If the implementation supports job control, the status of any child specified by pid that has stopped, and whose status has not been reported since it has stopped, is returned. The ***WIFSTOPPED*** macro determines whether the return value corresponds to a stopped child process
WCONTINUED	If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned

Avoid Zombies by Calling fork Twice

```
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 second fork==first child */
        /* We're the second child; our parent becomes init as soon as our real parent calls
         * exit() in the statement above. Here's where we'd continue executing, knowing that
         * when we're done, init will reap our status. */
        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }
    if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
        err_sys("waitpid error");
    /* We're the parent (the original process); we continue executing, knowing that we're
     * not the parent of the second child. */
    exit(0);
}
```

Race Conditions

Recall that the fork function create a process, but it does not guarantee which process, the parent or the child, runs first

An example (Figure 8.12)

- You cannot predict the parent or the child runs first

```
int main(void) {
    pid_t   pid;
    if ((pid = fork()) < 0)      { err_sys("fork error"); }
    else if (pid == 0)           { charatatime("output from child\n"); }
    else                         { charatatime("output from parent\n"); }
    exit(0);
}
```

Race Conditions – Solution #1

If the parent waits until a child terminates

- Use wait or waitpid to block the parent process
- Make sure that the child runs first

If a child waits until its parent terminates

- When its parent terminates, *init* will be the new parent, which has a PID of 1
- Use getppid function to check the value of *ppid* periodically

```
while (getppid() != 1)
    sleep(1);
```

The problem

- Either the parent or the child has to terminate
- Polling is not efficient

Race Conditions – Solution #2

Communication via interprocess communications (IPC)

An example of implementing using signals

- TELL_WAIT(): Initialize
- WAIT_PARENT(): blocks execution and waits for its parent
- TELL_CHILD(pid): tell a child that it has finished
- WAIT_CHILD(): blocks execution and waits for its child
- TELL_PARENT(ppid): tell its parent that it has finished

Race Conditions – Solution #2 (Cont'd)

Modifications to Figure 8.12 example

```
    int main(void) {
        pid_t   pid;
+       TELL_WAIT();
        if ((pid = fork()) < 0)    {
            err_sys("fork error");
        } else if (pid == 0) {
+           WAIT_PARENT();              /* parent goes first */
            charatatime("output from child\n");
        } else {
            charatatime("output from parent\n");
+           TELL_CHILD(pid);
        }
        exit(0);
    }
```

Process Execution

The exec Functions

Replace the calling process with a new program

The new program starts executing at its main function

The process ID does not change across an exec, because a new process is not created

Synopsis

- extern char **environ;
- int execl(const char *path, const char *arg, ...);
- int execlp(const char *file, const char *arg, ...);
- int execle(const char *path, const char *arg, ..., char * const envp[]);
- int execv(const char *path, char *const argv[]);
- int execvp(const char *file, char *const argv[]);
- int execve(const char *path, char *const argv[], char *const envp[]);

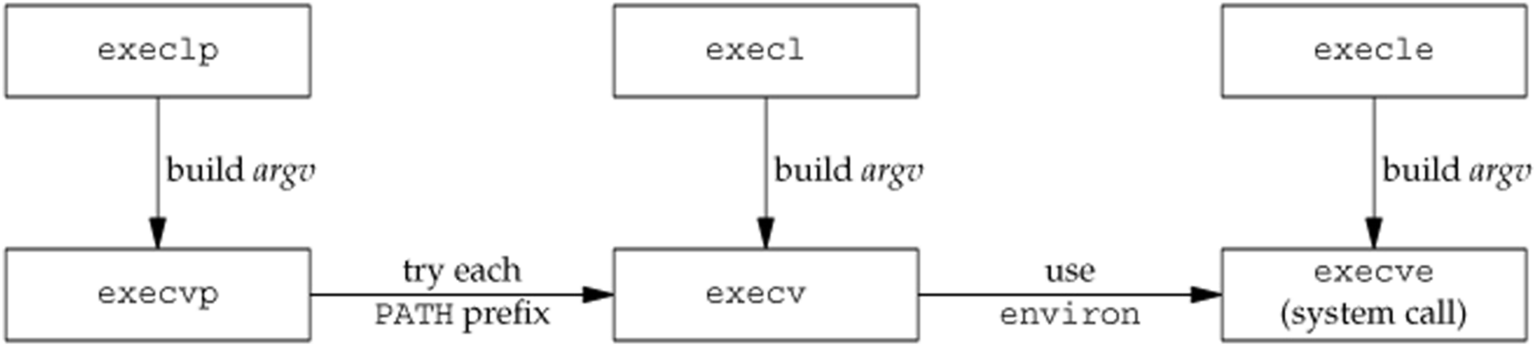
Differences Among the Six exec Functions

pathname – must be absolute or relative paths

filename – does not contain a slash (/), filename will be searched in directories listed in the PATH variable

Function	pathname	filename	arg list	argv[]	environ	envp[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e

Relationship of the Six exec Functions



How UNIX Recognizes Binaries?

It is done by checking file content

ELF binary

```
$ hexdump -C some-ELF-binary | head
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  30 07 40 00 00 00 00 00  |..>.....0.@.....|
00000020  40 00 00 00 00 00 00 00  48 15 00 00 00 00 00 00  |@.......H.......|
```

Interpreter files

```
$ hexdump -C some-interpreter-file | head
00000000  23 21 2e 2f 65 63 68 6f  62 69 6e 20 66 6f 6f 0a  |#!./echobin foo.|
00000010
```

Support More Binaries (Linux)

The binfmt_misc file system (on Linux)

```
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,noexec,nosuid,nodev)
```

Mount it manually

- You need a privileged docker runtime to do that!

```
# mount -t binfmt_misc none /proc/sys/fs/binfmt_misc
```

Support More Binaries (Linux) (Cont'd)

Add new binary format by writing to **/proc/sys/fs/binfmt_misc/register**

- Basic format: **:name:type:offset:magic:mask:interpreter:flags**
- You may have a look at the document https://www.kernel.org/doc/Documentation/binfmt_misc.txt
- Example: (as root)

```
# echo ":DOSWin:M::MZ::/usr/bin/wine:" > /proc/sys/fs/binfmt_misc/register
# cat /proc/sys/fs/binfmt_misc/DOSWin
enabled
interpreter /usr/bin/wine
flags:
offset: 0
magic: 4d5a
```

More binfmt_misc Examples (1/4)

Binary support from emulator

```
# apt install qemu-user qemu-user-static
```

Ubuntu: binfmt setup files can be found in /usr/share/binfmts/*

Sample formats listed in binfmt_misc file system

```
$ ls /proc/sys/fs/binfmt_misc/
jar            qemu-arm         qemu-mips        qemu-s390x        qemu-sparc64
python2.7      qemu-armeb       qemu-mipsel      qemu-sh4          register
python3.4      qemu-cris        qemu-ppc         qemu-sh4eb        status
qemu-aarch64   qemu-m68k        qemu-ppc64       qemu-sparc
qemu-alpha     qemu-microblaze  qemu-ppc64abi32  qemu-sparc32plus
```

More binfmt_misc Examples (2/4)

Jar

```
$ cat /proc/sys/fs/binfmt_misc/jar
enabled
interpreter /usr/bin/jexec
flags:
offset 0
magic 504b0304
```

ARM executable

```
$ cat /proc/sys/fs/binfmt_misc/qemu-armeb
enabled
interpreter /usr/bin/qemu-armeb-static
flags: OC
offset 0
magic 7f454c460102010000000000000000000000020028
mask  ffffffffffffff00fffffffffffffffffffeffff
```

More binfmt_misc Examples (3/4)

MIPS64

```
$ cat /proc/sys/fs/binfmt_misc/qemu-mips64
enabled
interpreter /usr/bin/qemu-mips64-static
flags: OCF
offset 0
magic 7f454c460202010000000000000000000000020008
mask ffffffffffffff00fefffffffffffffffffeffff
```

Sample: Cross-compile MIPS64 binary

```
# apt instll gcc-mips64-linux-gnuabi64
$ mips64-linux-gnuabi64-gcc hello.c –o hello -static
$ ./hello          ## with binfmt_misc support, -- or --
$ qemu-mips64 ./hello
```

More binfmt_misc Examples (4/4)

ARM64/AARCH64

```
$ cat /proc/sys/fs/binfmt_misc/qemu-aarch64
enabled
interpreter /usr/bin/qemu-aarch64-static
flags: OCF
offset 0
magic 7f454c460201010000000000000000000200b700
mask ffffffffffffff00fffffffffffffffffeffffff
```

Sample: Cross-compile ARM64 binary

```
# apt instll gcc-aarch64-linux-gnu
$ aarch64-linux-gnu-gcc hello.c –o hello -static
$ ./hello          ## with binfmt_misc support, -- or --
$ qemu-aarch64 ./hello
```

Create a "Foreign" Runtime

Run binaries in a "Foreign" runtime environment (cross-platform)

binfmt_misc must be configured properly first

Install a minimal root filesystem

```
# apt install debootstrap
# mkdir /arm64
# debootstrap --foreign --arch arm64 buster /arm64 http://ftp.debian.org/debian/
# mount --bind /proc /arm64/proc
# mount --bind /dev  /arm64/dev
# mount --bind /sys  /arm64/sys
# cp /usr/bin/qemu-aarch64-static /arm64/usr/bin/
# chroot /arm64
<IN the target platform>
# /debootstrap/debootstrap --second-stage     # optional – not really necessary
```

An exec Example

Suppose we have a program ***echoall*** that dumps argv[*] and environ[*]

- Note: echoall must be placed in one directory listed in $PATH

```
char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void) {
    pid_t pid;
    if ((pid = fork()) < 0)             { err_sys("fork error"); }
    else if (pid == 0) {                /* specify pathname, specify environment */
        if (execle("./fig8.17-echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }
    if (waitpid(pid, NULL, 0) < 0)      { err_sys("wait error"); }
    if ((pid = fork()) < 0)             { err_sys("fork error"); }
    else if (pid == 0) {                /* specify filename, inherit environment */
        if (execlp("fig8.17-echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }
    exit(0);
}
```

An exec Example (Cont'd)

```
$ PATH=$PATH:. ./fig8.16-exec1
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
argv[0]: echoall
argv[1]: only 1 arg
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:.
TERM=xterm
SHELL=/bin/bash
```

41 more lines that aren't shown

```
DISPLAY=localhost:10.0
LESSCLOSE=/usr/bin/lesspipe %s %s
_=./fig8.16-exec1
```

exec of Interpreter Files

All contemporary UNIX systems support interpreter files

These files are text files that begin with a line of the form

- #! pathname [optional-argument]
- For example, the shell scripts begins with the line #!/bin/sh

Interpreter files can be also executed by exec functions

exec of Interpreter Files, an Example

Suppose we have a program ***echoarg*** that prints all arguments

Suppose we have an interpreter file ***testinterp*** contains

#!/path/to/echoarg foo

```
int main(void) {
        pid_t pid;
        if ((pid = fork()) < 0)          { err_sys("fork error"); }
        else if (pid == 0) {             /* child */
                if (execl("/path/to/testinterp", "testinterp",
                        "myarg1", "MY ARG2", (char *)0) < 0)
                        err_sys("execl error");
        }
        if (waitpid(pid, NULL, 0) < 0)   /* parent */
                err_sys("waitpid error");
        exit(0);
}
```

exec of Interpreter Files, an Example (Cont'd)

```
$ cat /path/to/testinterp
#!/path/to/echoarg foo
$ ./fig8.20-exec2
argv[0]: /path/to/echoarg
argv[1]: foo
argv[2]: /path/to/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

The output of the previous example is shown above

The kernel actually executes the interpreter (pathname and argument after the #! symbol)

The exec executable name and its arguments are passed as additional arguments to the interpreter

More on exec of Interpreter Files

Usage of most of the shells, for example ***bash***

- *bash [options] [command] [arguments]*
- If a shell script *sample.sh* begins with ***#!/bin/bash***
- Execution of the shell script with a command "*./sample.sh 1 2 3*" is equivalent to run "***/bin/bash*** *./sample.sh 1 2 3*"

Another example, usage of the ***gawk*** utility

- *gawk [options] -f program-file [--] [files …]*
- A gawk script *sample.awk* ***must*** begin with ***#!/bin/gawk -f***
- Execution of the gawk script with a command "*./sample.awk test*" is equivalent to run "***/bin/gawk -f*** *./sample.awk test*"

The system Function

Execute shell commands in the program

Synopsis

- int system(const char *cmdstring);

An example

- system("date > file");
- Execute the ***date*** command and redirect its output to ***file***

It’s much more convenient

The system Function

It is implemented by calling fork(), exec(), and waitpid()

If either fork() fails or waitpid() returns an error other than EINTR, system() returns -1 with *errno* set to indicate the error

If exec() fails, it implies that the shell cannot be executed, the return value is as if the shell had executed exit(127).

If all the three functions (fork, exec, and waitpid) succeed, the return value from system() is the termination status of the shell, in the same format to that of waitpid().

The system Function – A Simple Implementation

```
int system(const char *cmdstring)    /*  version without signal handling  */ {
    pid_t pid;
    int status;
    if (cmdstring == NULL)
        return(-1);                  /*  always a command processor with UNIX  */
    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 EINTR from waitpid() */
                break;
    } } }
    return(status);
}
```

system and suid/sgid Programs

It might be a security problem if a suid/sgid program use the system function

If a suid/sgid program use the system function to execute a command

- The executed command has the same euid/egid as the calling process

If a suid/sgid program needs to execute a program

- Use exec functions instead
- Change euid/egid before calling exec
- seteuid and setegid

User Identification

Any process can find out its real and effective user ID and group ID

- `struct passwd *getpwuid(uid_t uid);`
- `getpwuid(getuid())`

It may not work for a single user that has multiple login names, and all have the same UID

An alternative

- `#include <unistd.h>`
- `char *getlogin(void);`
- `int getlogin_r(char *buf, size_t bufsize);`

With a login name, the correspond password entry can be obtained using `getpwnam()`

Process Times

The times(2) function

Count the current process user/system CPU time

Count the user/system CPU time for all waited processes

- A child's CPU times are counted after its termination status has been read by using wait() functions

- `#include <sys/times.h>`
- `clock_t times(struct tms *buf);`

```
struct tms {
        clock_t tms_utime;  /* user time */
        clock_t tms_stime;  /* system time */
        clock_t tms_cutime; /* user time of children */
        clock_t tms_cstime; /* system time of children */
};
```

Q & A