Process Control

System Programming

2019 여름 계절학기

한양대학교 공과대학 컴퓨터소프트웨어학부
홍석준

- **Process control – process creation, program execution, and process termination**
- **Process properties**
  - real, effective, and saved; user and group IDs
- **Interpreter files and the `system` function**
- **Process accounting**

I. Process Identifiers

```c
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

- Process ID: a unique, non-negative integer

I. Process Identifiers

❑ Process ID 0

- The scheduler process, a.k.a. `swapper`

❑ Process ID 1

- The `init` process invoked by the kernel at the end of the bootstrap procedure
- `/sbin/init`
- It reads the system-dependent initialization files (i.e., `/etc/rc`*) and brings a Unix system to a certain state.
  - ((`/etc/inittab` and `/etc/init.d/`) or `/etc/rc`*)

❑ Process ID 2

- `pagedaemon` responsible for supporting the paging of the virtual memory system.

I. fork Function

```
#include <unistd.h>
pid_t fork(void);
```

- **This function is called once, but returns twice.**
  - returns 0 in the child,
  - returns the process ID of the new child in the parent.
- **The child is a copy of the parent (data space, heap, and stack). Often, text segment is shared.**
- ***Copy-on-write* (COW)**

I. fork Function

```
#include "apue.h"
int glob = 6; /* external variable in initialized data */
char buf[] = "a write to stdout\n";
int main(void)
{
   int var; /* automatic variable on the stack */
   pid_t pid;
   var = 88;
   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) {
      glob++; /* modify variables */
      var++;
   } else {
      sleep(2);
   }
   printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
   exit(0);
}
```

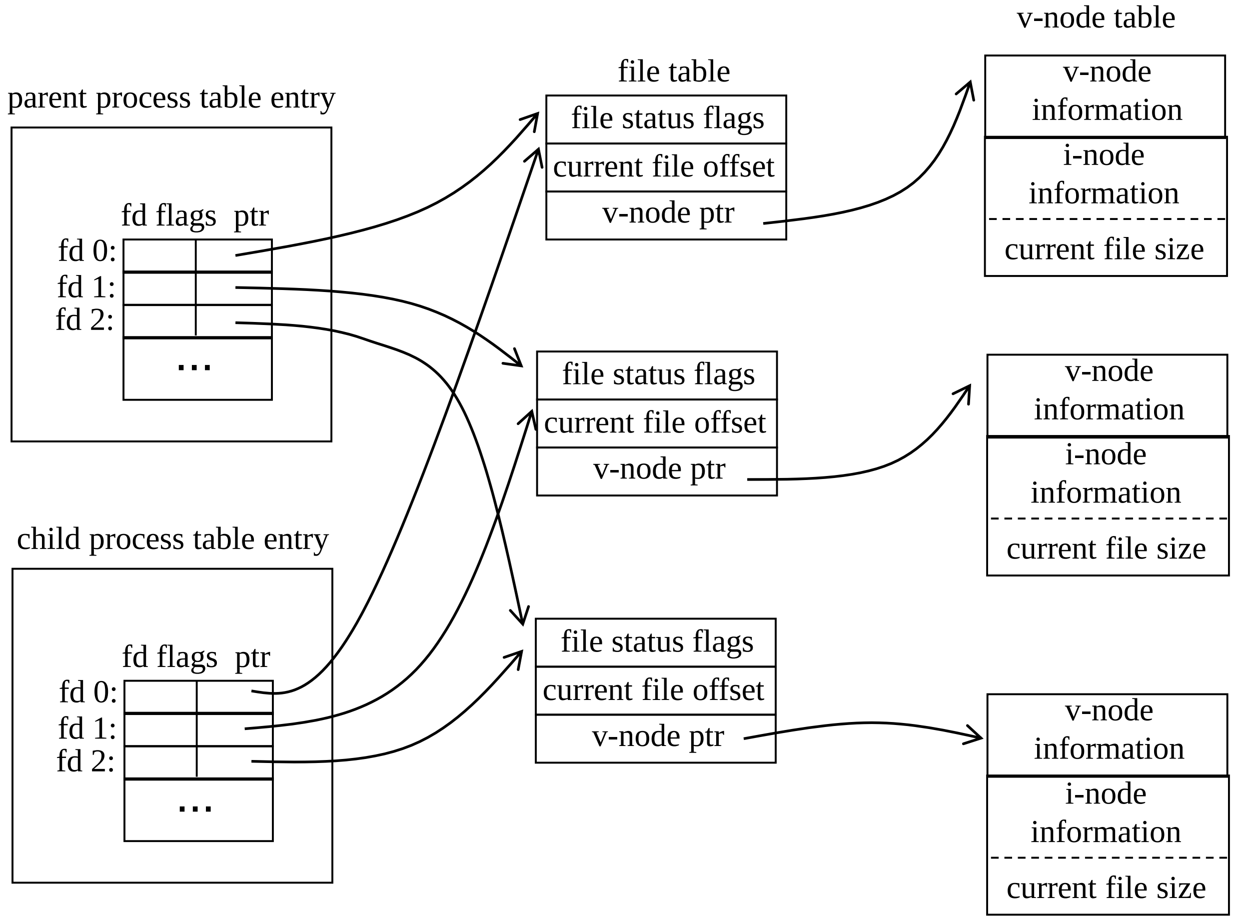
Figure 8.1

Ⅰ. fork Function

- Figure 8.1

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89
pid = 429, glob = 6, var = 88
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

Ⅰ. fork Function



- All descriptors that are open in the parent are **duplicated** in the child.

❑ Two normal cases for handling descriptors after a `fork`

- The parent waits for the child to complete. When the child terminates, any of shared descriptors read/written by the child will have their file offsets updated accordingly.
- The parent and child each go their own way. After `fork`, they close the descriptors that they don't need.

❑Properties inherited by the child

- real UID/GID, effective UID/GID, supplementary GIDs
- process group ID
- session ID
- controlling terminal
- set-user-ID flag and set-group-ID flag
- 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
- memory mapping
- resource limits

Differences between the parent and child

- the **return value** from `fork`
- the process IDs, parent process IDs
- the child's values for `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` are set to 0
- **file locks** set by the parent are not inherited by the child
- **pending alarms** are cleared for the child
- the set of **pending signals** for the child is set to the empty set

❑ Two reasons for `fork` to fail

- Too many processes in the system
- CHILD_MAX: the total number of processes per real user ID

❑ Two uses for `fork`

- The parent and child execute different sections of code at the same time, e.g. network servers.
- The parent and child execute a different program, e.g. shells (the child does an `exec` right after returning from the `fork`.)

- `vfork` does not fully copy the address space of the parent into the child
  (since the child won't reference that address space – the child just calls `exec` or `exit`.)
  The child runs in the parent address space.
- `vfork` guarantees that the child runs first, until the child calls `exec` or `exit`.
- Figure 8.3
  `_exit` vs. `exit` (flushing and closing `stdout`)

```
$ a.out
before vfork
pid = 29039, glob = 7, var = 89
```

I. vfork Function

```
#include "apue.h"
int glob = 6; /* external variable in initialized data */
int main(void)
{
   int var; /* automatic variable on the stack */
   pid_t pid;
   var = 88;
   printf("before vfork\n"); /* we don't flush stdio */
   if ((pid = vfork()) < 0) {
      err_sys("vfork error");
   } else if (pid == 0) { /* child */
      glob++; /* modify parent's variables */
      var++;
      _exit(0); /* child terminates */
   }
/* Parent continues here.*/
   printf("pid = %d, glob = %d, var = %d\n",
            getpid(), glob, var);
   exit(0);
}
```

Figure 8.3

❑ Five ways to terminate

- Normal termination
  - `return` from the `main` function
  - `exit` function that calls all exit handlers and closes all standard I/O streams.
  - `_exit` or `_Exit` function (`_Exit` to terminate a process without running exit handlers and signal handlers)
- Abnormal termination
  - `abort` function (SIGABRT signal)
  - When the process receives certain signals.

`I.exit` Functions

- **An *exit status* as the argument to `exit`, `_exit`, and `_Exit`**
- **A *termination status* generated by the kernel to indicate the reason for the termination.**
- **In any case, the parent can obtain the termination status from `wait/waitpid`. (the exit status is converted to a termination status by the kernel when `_exit` is called.)**
- **An orphan process is inherited by `init`.**
- **What if the child terminates before the parent?**
  - The kernel keeps a certain amount of information (pid, termination status, and used CPU time), so that the information is available for the parent's call to `wait/waitpid`.
  - A *zombie* is a process that has terminated, but whose parent has not yet waited for it.

I. wait and waitpid Function

- When a process terminates, the parent is notified by the kernel via the SIGCHLD signal.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

- `wait` can block the caller until a child terminates, while `waitpid` has an option that prevents it from blocking.
- If a child is a zombie, `wait` immediately returns that child's process ID with its termination status. Otherwise, it blocks the caller until a child terminates.
- `waitpid` can wait for a specific process.

❑ Macros to examine termination status

- WIFEXITED(*status*): normal termination
  - WEXITSTATUS(*status*)
- WIFSIGNALED(*status*) : terminated by a signal
  - WTERMSIG(*status*)
  - WCOREDUMP(*status*)
- WIFSTOPPED(*status*): currently stopped
  - WSTOPSIG(*status*)
- WIFCONTINUED(*status*): continued after a job control stop

❑ Figure 8.5 and Figure 8.6

I. wait and waitpid Function

```
#include "apue.h"
#include <sys/wait.h>
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));
}
```

Figure 8.5

I. wait and waitpid Function

```
#include "apue.h"
#include <sys/wait.h>
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) /* wait for child */
      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) /* wait for child */
      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) /* wait for child */
      err_sys("wait error");
   pr_exit(status); /* and print its status */
   exit(0);
}
```

Figure 8.6

- **`pid_t waitpid(pid_t` *pid*, `int` **statloc*, `int` *options*);**
  - *pid*
    - *pid* == -1 waits for any child process.
    - *pid* > 0 waits for the child whose process ID equals *pid*.
    - *pid* == 0 waits for any child whose process group ID equals that of the calling process.
    - *pid* < -1 waits for any child whose process group ID equals the absolute value of *pid*.
  - *options*
    - WCONTINUED – the status of any child continued is returned.
    - WNOHANG – *waitpid* will not block (returns 0).
    - WUNTRACED – the status of any child stopped is returned.

- **Figure 8.8**

I. wait and waitpid Function

```
#include "apue.h"
#include <sys/wait.h>
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);
}
```

Figure 8.8

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *statloc, int options,
            struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options,
            struct rusage *ruage);
```

- **It returns a summary of the resources used by the terminated process and all its child processes.**
  - User/system CPU time, number of page faults, number of signals received, and the like

```
#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[]);
```

- `exec` merely replaces the current process (its text, data, heap, and stack segments) with a brand new program from disk.
- *l* – list of arguments, *v* – argv[] vector, *e* – an envp[] array, and *p* – a filename argument.

❑ Filename argument (`execlp/execvp`)

- If filename contains a slash, it is taken as a pathname.
- Otherwise, the executable is searched for in PATH environment variable directories.
- If not a machine executable, it invokes `/bin/sh` with the *filename* as input to the shell.

❑ Argument passing

- `execl/execlp/execle` require separate command-line arguments with the end of the arguments marked with a null pointer.

❑ Environment list passing

- `execle/execve` passing `const *char envp[]` instead of using `extern char **environ`

❑ Properties inherited from the calling process

- pid, ppid, real UID/GID, supplementary GIDs
- process group ID, session ID
- controlling terminal
- time left until alarm clock
- current working directory, root directory
- file mode creation mask, file locks
- process signal mask, pending signals
- resource limits
- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`

❑ Handling of open files

- the *close-on-exec* flag of every open descriptor: if set, the descriptor is closed across an exec.
  - FD_CLOEXEC flag
- Effective UID/GID can change, depending on the status of the set-user-ID and the set-group-ID bits for the program file.

Ⅰ. exec Functions

```
#include "apue.h"
#include <sys/wait.h>
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("/home/sar/bin/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("echoall", "echoall", "only 1 arg",
                   (char *)0) < 0)
           err_sys("execlp error");
   }
   exit(0);
}
```

Figure 8.16

```
#include "apue.h"
int main(int argc, char *argv[])
{
   int i;
   char **ptr;
   extern char **environ;
   for (i = 0; i < argc; i++) /* echo all command-line args */
      printf("argv[%d]: %s\n", i, argv[i]);
   for (ptr = environ; *ptr != 0; ptr++) /* and all env strings */
      printf("%s\n", *ptr);
   exit(0);
}
```

Figure 8.17

- Figure 8.16 & Figure 8.17

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
. . .
```

Thank you for your attention !!

Q and A