C/UNIX Seminar - January 2022

Day 4: Processes and IPC

Assistants Salles Machine 2024

Processes

Fork

- The fork(2) syscall duplicates a process
- The copy has a complete copy of the memory
- Execution resusmes at the same point after the fork
- From teh code perspective, only the current pid and the return value of the fork are different

fork(2)

```c
pid_t fork(void);
```

Returns:

- In the parent: the pid of the child
- In the child: 0
- In case of error: -1 and no child is created

Basic Example

```c
int main() {
  fork();
  printf("Process PID:\t%u, Parent PID:\t%u\n",
         getpid(), getppid());
  sleep(1); // wait before exiting
  return 0;
}
```

```
$ ./simple_fork
Process PID: 4948, Parent PID: 651
Process PID: 4949, Parent PID: 4948
```

Basic Example

```
int main() {
  printf("%u\t-\t%u\n", fork(), getpid());
  return 0;
}
```

```
$ ./simple_fork2
4972    -    4971
0       -    4972
```

From parent to child

- The original process is called the parent process
- The new process is the child process

Process Life

- All processes (but one) are created through fork(2)
- They all have a parent
- The (grand-)parent of all processes has PID 1
- 2 questions arise:
    - What is the parent of the process when its parent terminates ?
    - What happens to a process when it terminates ?

Orphan Process

Init (PID 1) usually adopts all orphan processes !

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  printf("Original process:\t%u\n", getpid());
  if (fork()) return 0; // Parent exits quickly
  sleep(1); // Give time for parent's exit
  printf("Child parent:\t%u\n", getppid());
  return 0;
}
```

```
> ./orphan
Original process: 16774
Child parent: 1
```

Dead Processes

- When dying, a process becomes a **zombie**!
- A zombie process is not *completely dead*
- A zombie's existence is tied with its parent's life
- Zombies are mostly usefull for debugging

Zombie

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main() {
  pid_t child;
  if (!(child = fork())) return 0;  // Child exits directly
  printf("(%u) PID of child: %u\n", getpid(), child);
  sleep(30); // give us time
  printf("(%u) Parent Process exits\n", getpid());
  return 0;
}
```

```
$ ./zombie_creator
(16264) PID of child: 16265
(16264) Parent Process exits
```

```
$ ps a | grep 16265
16265 pts/2  Z+ 0:00 [zombie_creator] <defunct>
16277 pts/1  S+ 0:00 grep 16265
```

wait(2) and waitpid(2)

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

- Block until process child terminates or gets interrupted
- Returns the PID of the child
- status: information about the child
- waitpid: wait for a specific child process with options

Eleminating Zombies

- Zombies usually die when their parent terminates
- You can also wait for child

Eliminating Zombies

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>
int main() {
  // Child exits directly
  if (!fork()) return 0;
  pid_t child = wait(NULL);
  printf("(%u) Child %u is dead\n",
         getpid(), child);
  sleep(30);
  return 0;
}
```

```
$ ./waiting
(16400) Child 16401 is dead
```

```
$ ps a | grep 16401
16277 pts/1 S+ 0:00 grep 16401
```

Cause of Termination

- Process termination belongs to two kinds:
  - Normally by using exit(3) or by returning from main()
  - Or by being terminated by a signal
- wait(2) can give the cause of the termination
- wait(2) can also give the exit code or the signal number

Cause of Termination

- status can be decomposed using macros
- For terminating processes:
  WIFEXITED, WEXITSTATUS ...
- For killed processes:
  WIFSIGNALED, WTERMSIG ...

Cause of Termination

```c
int main() {
  srandom(time(NULL));
  if (!fork()) {
    if (random()%2) return 42;
    else abort(); // send signal 6 to self
  }

  pid_t chld = wait(&status);
  int status;
  printf("(%u) Child %u died\n", getpid(), chld);
  if (WIFEXITED(status))
    printf("  exit %d\n", WEXITSTATUS(status));
  else
    printf("  killed %d\n", WTERMSIG(status));

  return 0;
}
```

Cause of Termination

```
$ ./waiting2
(1478214) Child 1478215 died:
  exited with 42
$ ./waiting2
(1478245) Child 1478246 died:
  killed by signal 6
```

Advanced versions

```
// non-standard
pid_t wait3(int *status, int options,
            struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options,
            struct rusage *rusage);

// standard, but must be used *with* wait(2) or
// waitpid(2)
int getrusage(int who, struct rusage *rusage);
```

- Add options
- rusage informations

Advanced versions

```c
struct rusage {
  struct timeval ru_utime; /* user CPU time used */
  struct timeval ru_stime; /* system CPU time used */
  long   ru_maxrss;        /* maximum resident set size */
  long   ru_ixrss;         /* integral shared memory size */
  long   ru_idrss;         /* integral unshared data size */
  long   ru_isrss;         /* integral unshared stack size */
  long   ru_minflt;        /* page reclaims (soft page faults) */
  long   ru_majflt;        /* page faults (hard page faults) */
  long   ru_nswap;         /* swaps */
  long   ru_inblock;       /* block input operations */
  long   ru_oublock;       /* block output operations */
  long   ru_msgsnd;        /* IPC messages sent */
  long   ru_msgrcv;        /* IPC messages received */
  long   ru_nsignals;      /* signals received */
  long   ru_nvcsw;         /* voluntary context switches */
  long   ru_nivcsw;        /* involuntary context switches */
};
```

Running Another Program

How it works:

- Load the new program binary
- Replace the current program memory
- Keep the same process
- Keep the same open resources
- Keep the same privileges (by default)

The exec family

- Syscall: execve(2)
- Wrappers: execl(3), execlp(3), execle(3), execv(3), execvp(3), execvpe(3)
- We'll focus on execvp(3)

execvp(3)

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

- file: the program file (using the path)
- argv: the arguments of the program
- Return value:
  - On error: returns -1 and sets errno
  - On success: **does not** return !

Using the path

file: may be a path or a simple file name

- Path: contains at least a "/"
  - at the beginning: absolute path
  - otherwise: relative path to the current directory
- Single file name (no "/")
  - First match search in the PATH environnement variable

Arguments

argv array:

- The one the program will get in its *main*
- argv[0] contains the program name
- Last element must be a NULL pointer

Behavior of the exec* functions

- If succesfull: they *don't* return
  The code after them only gets executed if they failed
- All FDs are kept open
  execpt those explicitely flagged with FD_CLOEXEC
- Signal handlers are reset to default

Example

```c
#include <err.h>
#include <unistd.h>

int main() {
  char *arg[3];
  arg[0] = "ls";
  arg[1] = "-l";
  arg[2] = NULL;
  execvp(arg[0], arg);

  // Should never be reached
  err(3, "couldn't exec ls");

  return 0;
}
```

Fork/Exec

To launch a new program:

1. Fork
   - In the parent: wait for the child
   - In the child: exec the wanted program
2. When wait returns: extract information on child termination if needed

Fork/Exe

```c
int main()
{
  fprintf(stderr, "(%u) launching command:\n", getpid());

  if (fork()) { // parent process
    wait(NULL);
  } else { // child process
    char *args[] = {"echo", "hello world !", NULL};
    execvp(args[0], args);
  }

  fprintf(stderr, "(%u) command done\n", getpid());
  return 0;
}
```

Fork/Exec

```
$ ./exec_echo
(11450) launching command:
hello world !
(11450) command done
$
```

Pipe

Pipe

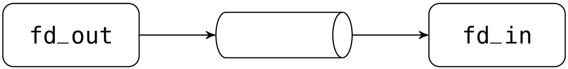
```
$ echo "I got full marks" | cat -e
```

Pipe

- Used for Inter-Process Communication (IPC)
- FIFO structure
- One 'Write' end and one 'Read' end $\rightarrow$ Unidirectional data channel
- Can be sharred between processes

Pipe usage

A process may use both end of the pipe but usually uses only one of the two



Pipe Usage

The manual will be your friend once again

```
$ man 7 pipe
$ man 2 pipe
```

pipe(2) syscall

```c
int pipe(int pipefd[2]);
```

- Modifies its parameter to store the write end in pipefd[1] and the read end in pipefd[0]
- Return -1 on failure, 0 otherwise

An incomplete Pipe

```c
int main() {
  // Error checking skipped for brevity.
  int fds[2];
  pipe(fds);

  if (fork())
    write(fds[ ? ], "I'm your father !\n", 18);
  else {
    char buf[256];
    int r = read(fds[ ? ], buf, 256);
    write(STDOUT_FILENO, "read> ", 6);

    // TODO : print the content of the buffer with write(2).

    write(STDOUT_FILENO, "Nooooooooo !!!!!!!\n", 19);
  }

  return 0;
}
```

An incomplete Pipe

```c
int main() {
  // Error checking skipped for brevity.
  int fds[2];
  pipe(fds);

  if (fork())
    write(fds[1], "I'm your father !\n", 18);
  else {
    char buf[256];
    int r = read(fds[0], buf, 256);
    write(STDOUT_FILENO, "read> ", 6);

    write(STDOUT_FILENO, buf, r);

    write(STDOUT_FILENO, "Noooooooooo !!!!!!!\n", 19);
  }

  return 0;
}
```

End of the Pipe

You may reach the end of a pipe:

- **when reading:**
  - The pipe is empty and no data can be read from it as nothing was written inside. (empty pipe blocks
  - fd[1] has been closed and nothing more will ever come, reading gets an end-of-file and read(2) returns 0.
- **when writing:**
  - Some implementations may block
  - When fd[0] is closed: receive SIGPIPE (which terminates) and write fails with EPIPE

End of the Pipe

```
int main() {
    int fd[2];
    pipe(fd);
    if (fork()) {
        int r;
        char buf[256];
        while ( (r = read(fd[0], buf, 256)) )
           write(STDOUT_FILENO, buf, r);
    } else {
        for (int i = 0; i <= 42; ++i) {
           char *buf;
           int len = asprintf(&buf, "%d\n", i);
           write(fd[1], buf, len);
           free(buf);
        }
    }
    return 0;
}
```

End of the Pipe

- The previous code is blocked on the read instruction
- fd[1] is still open in the parent
- read will never get the *end-of-file*
- to avoid such problems you need to use the syscall close(2)

Redirections

Concept

```
$ ls > foo
```

- How does the shell redirect the output os ls(1) ?
- It can't change the internal behavior of ls(1)

Redirecting File Descriptors

- FDs are just numbers referencing a resource
- One resource can correspond to several FD
- We can reassign a FD to another resource
- To do so we us the syscalls dup(2) and dup2(2)

dup(2)

```
int dup(int oldfd);
```

- newfd = dup(oldfd)
- finds the smallest available FD and retuns it
- binds newfd to the resource of oldfd

dup2(2)

```
int dup2(int oldfd, int newfd);
```

- dup2(oldfd, newfd)

- closes newfd if already binded
- rebinds newfd to the resource of oldfd
- if newfd = oldfd, dup2 does nothing

Example - code

```
int main(int argc, char *argv[]) {
  char *fname = "output";
  if (argc > 1)
    fname = argv[1];

  int fd = open(fname, O_WRONLY|O_CREAT|O_TRUNC, 0666);
  if (fd == -1)
    err(3,"error opening %s", fname);

  dup2(fd, STDOUT_FILENO);
  close(fd);
  printf("Normally this is in %s and not on the TTY !\n",
         fname);

  return 0;
}
```

Example - execution

```
$ ./redir
$ cat output
Normally this is in output and not on the TTY!
```

Shell pipes

```
$ cmd1 | cmd2
```

- Create a pipe
- First fork:
    - redirect STDIN_FILENO to the pipe
    - exec cmd2
- Second fork:
    - redirect STDOUT_FILENO to the pipe
    - exec cmd1
- Wait for both children

General recommandations

- All unused file descriptors should be closed
- Unclosed file desciptor can cause children to never stop
- More commands to execute in a row → add more pipes

When you need to have more than two commands, make sure to start the execution flow by the last command to ensure every command is successfully completed.

Named pipe

Concept

- Like a pipe but with an entry in the file system
- Usage:
  - Create the named pipe using mkfifo(3)
  - Open the corresponding file in both processes
  - use it
  - close it
  - delete the file using unlink(2)

Creator and writer

```
int main()
{
  printf("Creating named pipe: channel\n");
  if (mkfifo("channel", 0666) < 0)
    err(1, "failed while creating fifo 'channel'");

  printf("Opening fifo for writing\n");
  int fd = open("channel", O_WRONLY);
  if (fd < 0)
    err(1, "Can't open fifo for writing");

  printf("Sending some message\n");
  if (write(fd, "message in a fifo", 17) < 0)
    err(1, "Can't write to fifo");

  close(fd);
  unlink("channel");
  return 0;
}
```

Reader

```c
int main()
{
  printf("Opening fifo for reading\n");
  int fd = open("channel", O_RDONLY);
  if (fd < 0)
    err(1, "can't open fifo");

  ssize_t r;
  char buf[256];
  while ( (r = read(fd, buf, 256)) != 0) {
    if (r < 0)
      err(1, "can't read from fifo");
    write(STDOUT_FILENO, buf, r);
  }

  close(fd);
  return 0;
}
```

Running

```
$ ls
Makefile fifo_manager fifo_manager.c fifo_reader fifo_reader.c
$ ./fifo_manager
Creating named pipe: channel
Opening fifo for writing...
Sending some message...
```

```
$ ls
Makefile channel fifo_manager fifo_manager.c fifo_reader fifo_reader.c
$ ./fifo_reader
Opening fifo for reading
message in a fifo
$ ls
Makefile fifo_manager fifo_manager.c fifo_reader fifo_reader.c
```

0x5f3759df

asm 2024