



Making Processes

- Topics
 - Process creation from the user level:
 - Fork, execvp, wait, waitpid, pipe
- Learning Objectives:
 - Explain how processes are created
 - Create new processes, synchronize with them, and communicate exit codes back to the forking process.
 - Start building a simple shell.



Where do Processes Come From?

- There are two models of process creation:
 - Copy an existing process (UNIX `fork/exec` model).
 - Single system call to create a new process (Windows model).
- In UNIX-like systems we use the `fork/exec` model.



Fork

- **System call** that copies the calling process, creating a second process that is identical (in all but one regard) to the process that called `fork`.
- We refer to the calling process as the **parent** and the new process as the **child**.
- On return from successful `fork`:
 - **Parent: return value is the pid of the child process.**
 - **Child: return value is 0.**
- If the `fork` fails:
 - No child process created.
 - Parent gets return value of -1 (and `errno` is set).



Programming with fork

```
#include <unistd.h>
pid_t  ret_pid;

ret_pid = fork();
switch (ret_pid){
    case 0:
        /* I am the child. */
        break;
    case -1:
        /* Something bad happened. */
        break;
    default:
        /*
         * I am the parent and my child's
         * pid is ret_pid.
         */
        break;
}
```

Handwritten annotations:

- A checkmark is drawn next to the `ret_pid` variable declaration.
- A checkmark is drawn next to the `/* I am the child. */` comment.
- The word "Parent" is written in cursive and underlined, with a line pointing to the `default` case.



But what good are two identical processes?

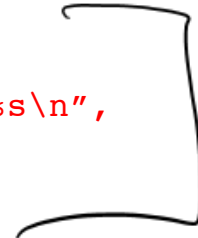
- So `fork` let us create a new process, but it's identical to the original. That might not be very handy.
- Enter `exec` (and friends): The `exec` family of functions **replaces the current process image with a new process image**.
- `exec` is the original/traditional API
- `execve` is the modern day (more efficient) implementation.
- There are a pile of functions, all of which ultimately invoke `execve`; we will focus on `execvp` (which is recommended for Assignment 4).
- If `execvp` returns, then it was unsuccessful and will return -1 and set `errno`.
- If successful, **`execvp` does not return**, because it is off and running as a **new process**.
- Arguments:
 - `file`: Name of a file to be executed
 - `args`: Null-terminated argument vector; the first entry of which is (by convention) the file name associated with the file being executed.



Programming with Exec

```
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
pid_t    ret_pid;

ret_pid = fork();
switch (ret_pid){
    case 0:
        /* I am the child. */
        if (execvp(path, argv) == -1)
            printf("Something bad happened: %s\n",
                strerror(errno));
        break;
    case -1:
        /* Something bad happened. */
        break;
    default:
        /*
         * I am the parent and my child's
         * pid is ret_pid.
         */
        break;
}
```





Coordinating with your child

- Sometimes it is useful for a parent to wait until a specific child, all children, or any child exits.

```
pid_t wait (int *stat_loc)
```

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

- **wait**: suspends execution of the parent until some child of the parent terminates or the parent receives a **signal**.
 - **Return value is the pid** of the terminating process
 - `stat_loc` is filled in with a status indicating how/why the child terminated.
- **waitpid**: suspends until a particular child terminates.



Programming with Fork, Exec, Wait

```
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
pid_t    ret_pid;

ret_pid = fork();
switch (ret_pid){
    case 0:
        /* I am the child. */
        if (execvp(path, argv) == -1)
            printf("Something bad happened: %s\n",
                strerror(errno));

        break;

    case -1:
        /* Something bad happened. */
        break;

    default:
        /*
         * I am the parent and my child's
         * pid is ret_pid.
         */
        if (waitpid(ret_pid, &exit_status, 0) != ret_pid)
            printf("Something bad happened!\n");

        break;
}
```




Communicating with child processes

- You've all used the `|` character to create pipes on the command line in the shell (I hope).
- What exactly does the pipe character do?
- The effect:
 - When you type:
`% foo | bar`
 - The `stdout` stream of `foo` is connected to the `stdin` stream of `bar`.
- You've probably used the `file pointers`, `stderr`, `stdout`, `stdin` in `fprintf` and `fscanf`.
- `STDIN_FILENO/STDOUT_FILENO`(and `STDERR_FILENO`) are the corresponding `file descriptors`.
- They are opened on behalf of every process.
 - By convention, `stdin` comes from the console
 - By convention, `stdout` goes to the display
- Allowing two processes to interact as shown above requires that `we connect foo's stdout to bar's stdin`



The pipe system call

- `pipe(int fildes[2])` creates a **pair of file descriptors** and places them in the array referenced by `fildes`.
 - `fildes[0]` is for reading
 - `fildes[1]` is for writing
- **When a parent forks children, the parent and child share file descriptors.**
- By combining, `fork`, `exec`, and `pipe`, parents can communicate with children and/or set up pipelines between children.





The dup2 system call

- `dup2(int fildes, int fildes2)`
 - duplicates the first file descriptor (`fildes`) into the second file descriptor (`fildes2`).
 - After the call, **both file descriptors refer to the same object**, so reading from/writing to one descriptor changes the file position in both descriptors.
 - If `fildes2` already refers to an open object, that object is closed.





Creating a Pipeline (foo | bar)

Note: Terrible error handling to save space!

```
pid_t child1, child2;
int pipedes[2], status;

assert (pipe(pipedes) == 0);           /* Create the pipe. */
child1 = fork();
if (child1 == 0) {
    /* child */
    close(pipedes[0]);                 /* Close read end */
    dup2(pipedes[1], STDOUT_FILENO); /* Make stdout the same as the pipe write fd */
    execvp("foo", argv);              /* Assume argv is set */
}
/* only parent gets here */
child2 = fork();
if (child2 == 0) {
    /* child */
    close (pipedes[1]);                /* Close writing end */
    dup2(pipedes[0], STDIN_FILENO);   /* Make stdin the same as the pipe read fd */
    execvp("bar", argv);
}
/* Parent once again */
close (pipedes[0]);                  /* Close pipe fDs in parent. */
close (pipedes[1]);
waitpid(child2, &status, 0);        /* Wait for second process to complete. */
```




Wrapping Up

- On most UNIX-like systems today, we use `fork/`
`exec` to create new processes.
- `wait` and `waitpid` allow parents and children to synchronize.
- The `dup2` and `pipe` calls provide for communication between parents and children.
- Next, we'll learn about signals, another mechanism used to coordinate processes.