CS241 Systems Programming

Discussion Section Week 2
1/29/07 – 2/02/07
Stephen Kloder

Outline

- Processes
- `fork`
- `wait`
- `exec`

Processes

A process is an instance of a running program

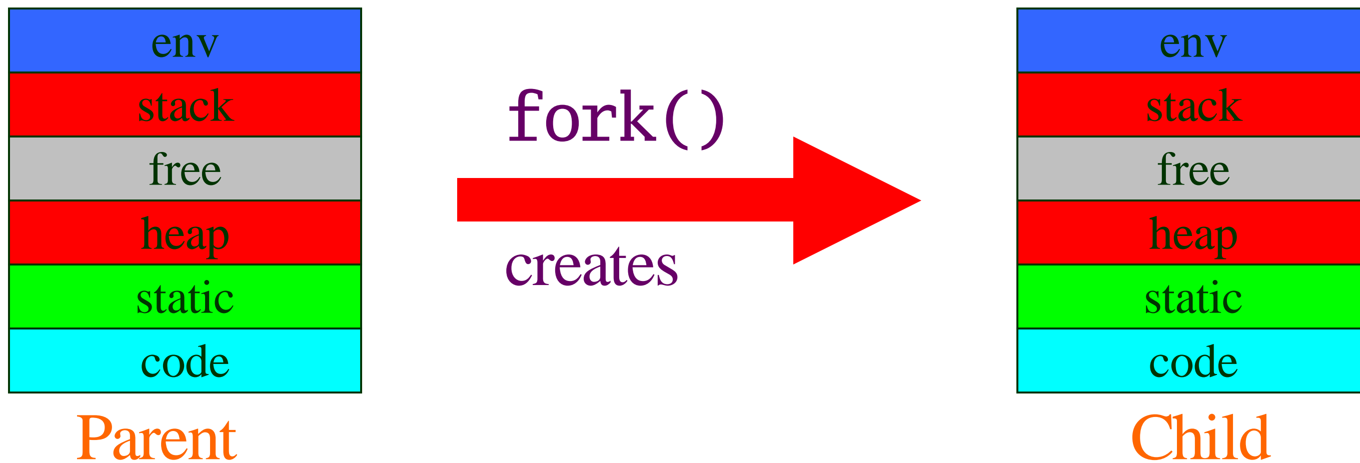
A process contains:

- Instructions (i.e. the program)
- Resources (variables, buffers, links, etc.)
- State (identity, ready/running/locked, etc.)

Processes can create other processes

Process Creation with fork()

`fork()` creates a new process:



- The new (child) process is identical to the old (parent) process, except…

Differences between parent and child

- Process ID (`getpid()`)
- Parent ID (`getppid()`)
- Return value of `fork()`
  - In parent, `fork()` returns child pid
  - In child, `fork()` returns 0

`fork()` Example 1

What does this do?

```
fprintf(stdout,“%d\n”,fork());
```

Try it!

fork() example 2
(p1-fork.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char** argv) {
   pid_t child_pid = fork();
   if (child_pid < 0) {          // error code
      perror(“Fork Failed”);
      return -1;
   }
   fprintf(stdout, “I am process %d\n”,getpid());
   if (child_pid == 0) {         // child code
      fprintf(stdout, “I’m the child process.\n”);
   } else {                      // parent code
      fprintf(stdout, “I’m the parent of child process %d.\n”,
      child_pid);
   }
   return 0;
}
```

Example 2 cont'd

This exits too quickly; let's slow it down:

```
if (child_pid == 0) {        // child code
    sleep(15);"
    fprintf(stdout, "I'm the child process.\n");
} else {                     // parent code
    sleep(20);
    fprintf(stdout, "I'm the parent of child process %d.\n",
            child_pid);
}
```

Example 2 cont'd

- In a second window, run `ps -a`, and look for the pids from the program output.
- Periodically run `ps -a` again, as the program in the first window executes.
  - What happens when the program runs?
  - What happens when the child finishes?
  - What happens when the parent finishes?
  - What happens when you switch the parent and child sleep statements?

Orphans and Zombies

When a process finishes, it becomes a *zombie* until its parent cleans up after it.

If its parent finishes first, the process becomes an *orphan*, and the `init` process (id 1) adopts it.

How can a parent know when its children are done?

Solution: `wait(…)`

- `wait()` allows a parent to wait for its child process, and save its return value

```
pid = wait(&status, options);
pid = waitpid(pid, &status, options);
```

- `wait()` waits for any child; `waitpid()` waits for a specific child.

`wait` cont'd

- `wait()` blocks until child finishes
  - `wait()` does not block if the option `WNOHANG` is included. When would we want to use this?
- The child's return value is stored in `*status`

`wait(...)` macros

`WIFEXITED(status)` is true iff the process terminated normally.

`WEXITSTATUS(status)` gives the last 8 bits of the process's return value (assuming normal exit)

Example 3: wait

```
#include <sys/wait.h>
  …
  …

    // add this to parent code
    if (waitpid(child_pid, &result, 0) == -1) {
       perror("Wait failed");
       return -1;
    }
    if( WIFEXITED(result)) {
    fprintf(stdout, "child %d returned %d\n",
          child_pid, WEXITSTATUS(result));
```

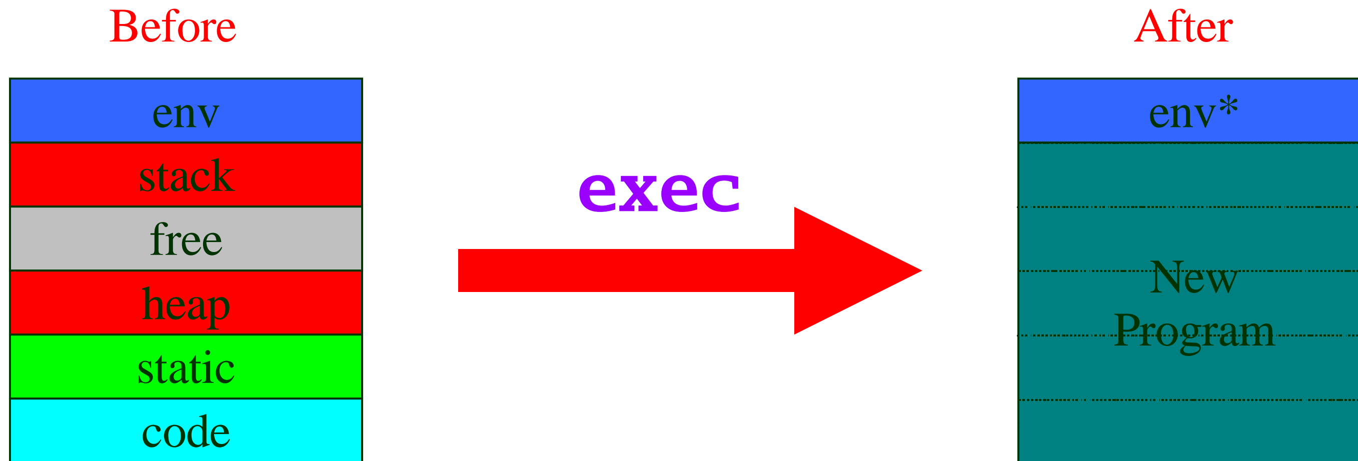
Problem 4

Recall Example 2. How can we use `wait` to clean up the zombie process?

exec

exec

exec replaces the current process image(code, variables, etc.) with that of a new program:



* The program may choose to change the environment

`exec` variations

There are 6 different ways of calling `exec`. Which one to use depends on three conditions:

1. How arguments are passed
2. How the path is specified
3. Whether a new environment is used

`exec` variations: passing parameters

`exec` can have parameters passed to it two different ways:

- List of parameters:

```
execl("/bin/ls", "ls", "-l", NULL);
```

- Argument Vector (like `argv`):

```
execv(argv[1],argv+1);
```

Q: When would you use `execl`?

When would you use `execv`?

`exec` variations: command path

Adding a "`p`" to an `exec` call tells the system to look for the command in the environment's path.

Compare:

- `execl("/bin/ls", "ls", "-l", NULL);`
- `execl`**`p`**`("ls", "ls", "-l", NULL);`

The difference is similar for `execv` and `execvp`.

`exec` variations (cont'd)

By default, the new program inherits the old program's environment. Adding an "`e`" to the `exec` call allows the new program to run with a new environment `**environ`

`execve` and `execle` allow the user to specify the environment. The others inherit the old environment.

`fork` + `exec`

- If `exec` succeeds, it does not return, as it overwrites the program.
  - No checking return values, executing multiple commands, monitoring results, etc.
- Solution: fork a new process, and have the child run `exec`

Example 5: exec

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

int main(int argc, char** argv) {
   child_pid = fork();
   if (child_pid < 0) {
      perror(“Fork failed”);
      return -1;
   } else if (child_pid == 0) {
      if (execvp(argv[1], argv+1) < 0) {
         fprintf(stderr, “Failed to execute %s!\n”, argv[1]);
         perror(“child exec:”);
         return -1;
      }
   }
   // What belongs here?
   ...
   return 0;
}
```

Problem 6: `p6-exec.c`

Compile and run this program. It seems to use `fork` and `exec` to run a list of argument-free commands (`ls`, `cal`, etc.) from a file `commands`.

- What's wrong with this program? What happens when you run it?
- How can you fix this?