University of Dhaka
Institute of Information Technology (IIT)
Bachelor of Science in Software Engineering (BSSE)

Course: **CSE 401: Operating Systems**
Instructors: **Dr. Naushin Nower, Professor, IIT, DU**
Toukir Ahammed, Lecturer, IIT, DU
Lab 03

Getting a process' process ID (PID)

Every process on the system has a unique process ID number, known as the *pid*. This is simply an integer. You can get the pid for a process via the *getpid* system call.

Example

This is a small program that simply prints its process ID number and exits.

```
/* getpid: print a process' process ID */
/* Paul Krzyzanowski */

#include <stdlib.h>    /* needed to define exit() */
#include <unistd.h>    /* needed to define getpid() */
#include <stdio.h>     /* needed for printf() */

int main() {

      printf("my process ID is %d\n", getpid());

      exit(0);
}
```

Save this file, compile it and run.

Source: https://people.cs.rutgers.edu/~pxk/416/notes/c-tutorials/index.html

Getting a parent process' process ID (PID)

Processes in Unix have a hierarchical relationship. When a process creates another process, that new process becomes a *child* of the process, which is known as its *parent*. If the parent process exits and the child is still running, the child gets inherited by the *init* process, the first process and everyone's ultimate parent.

You can get the parent's pid (process ID) for a process via the *getppid* system call.

Example

This is a small program that simply prints its process ID number and its parent's prcess ID number and then exits.

```
/* getppid: print a child's and its parent's process ID numbers */
/* Paul Krzyzanowski */

#include <stdlib.h>    /* needed to define exit() */
#include <unistd.h>    /* needed to define getpid() */
#include <stdio.h>     /* needed for printf() */

int main() {

      printf("my process ID is %d\n", getpid());
      printf("my parent's process ID is %d\n", getppid());
      exit(0);
}
```

Save this file, compile it and run.

Creating a process

A new process is created with the *fork* system call. When *fork* is called, the operating system creates a new process: it assigns a new process entry in the process table and clones the

information from the current one. All file descriptors that are open in the parent will be open in the the child. The executable memory image is copied as well. As soon as the *fork* call returns, both the parent and child are now running at the same point in the program. The only difference is that the child gets a different return value from fork. The parent gets the procsss ID of the child that was just created. The child gets a return of 0. If fork returns -1 then the operating system was unable to create the process.

The crucial thing to note with *fork* is that nothing is shared after the *fork*. Even though the child is running the same code and has the same files open, it maintains its own seek pointers (positions in the file) and it has its own copy of all memory. If a child changes a memory location, the parent won't see the change (and vice versa).

Example

This is a small program that clones itself. The parent prints a message stating its process ID and the child's process ID. It gets its process ID via the *getpid* system call and it gets its child's process ID from the return of *fork*. the child prints its process ID. The parent and child then each exit. Note that exit takes a parameter. This becomes the exit code of the program. The convention for Unix systems is to exit with a code of zero on success and a non-zero on failure. This helps in scripting programs.

```
/* fork: create a new process */
/* Paul Krzyzanowski */

#include <stdlib.h>    /* needed to define exit() */
#include <unistd.h>    /* needed for fork() and getpid() */
#include <stdio.h>     /* needed for printf() */

int main() {
      int pid;    /* process ID */

      switch (pid = fork()) {
      case 0:           /* a fork returns 0 to the child */
            printf("I am the child process: pid=%d\n", getpid());
            break;
```

Source: https://people.cs.rutgers.edu/~pxk/416/notes/c-tutorials/index.html

```
      default:    /* a fork returns a pid to the parent */
           printf("I  am  the  parent  process:  pid=%d,  child  pid=%d\n",
getpid(), pid);
           break;

      case -1:    /* something went wrong */
           perror("fork");
           exit(1);
      }
      exit(0);
}
```

Save this file, compile it and run.

Running a new process: exec

The *fork* system call creates a new process but that process contains, and is executing, exactly the same code that the parent process has. More often than not, we'd like to run a new program. The *execve* system call replaces the current process with a new program. The syntox of *execve* is:

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

This system call returns only if the named program could not run.

`argv` is the argument list that is passed to the *main* function of the program, which is the function that is called when any program starts. The last item on the list is 0 (to mark the end of the list). The first, `argv[0]`, is essentially ignored and expected to contain the command name. The program that is executed may use it to find out its name.

`filename` contains the fill pathname of the program to load and run; for example, `"/bin/ls"`. This is what the operating system tries to load.

Source: https://people.cs.rutgers.edu/~pxk/416/notes/c-tutorials/index.html

`envp` contains character strings representing the environment in the form *name=value*. `envp` is also stored in the global variable `char **environ` (*getenv* is a library function to search the environment for the value of a particular name).

Example

This is a small program that overrites itself with another program. The program first prints a message stating that it will run the *ls* program and then execute:

```
ls -aF /
```

As soon as *execve* is called, the process is overritten with the new program (*/bin/ls*) and the program's *main* function gets run.

```
/* execve: run a program */
/* Paul Krzyzanowski */

#include <stdlib.h>    /* needed to define exit() */
#include <unistd.h>    /* needed to define getpid() */
#include <stdio.h>     /* needed for printf() */

int main() {
      char *args[] = {"ls", "-aF", "/", 0};  /* each element represents a command line argument */
      char *env[] = { 0 };   /* leave the environment list null */

      printf("About to run /bin/ls\n");
      execve("/bin/ls", args, env);
      perror("execve");      /* if we get here, execve failed */
      exit(1);
}
```

Save this file, compile it and run.

Variations on exec

In addition to the *execve* syatem call, there are a number of library routines that are front-ends to execve. You will usually fine these more convenient than *execve*. They are described in the *execl* manual page. There are six of these functions. A couple of useful ones are *execlp* and *execvp*.

execlp

execlp, which allows you to specify all the arguments as parameters to the function. Note that the first parameter is the command. The second parameter is the first argument in the argument list that is passed to the program (`argv[0]`). These are often the same but don't have to be. The last parameter must be a null pointer.A *execlp* uses the search path set by the PATH environment variable to find the command. For example,

```
execlp("ls", "ls", "-aF", "/", (char *)0);
```

```
/* execlp: run a program using execlp */
/* Paul Krzyzanowski */

#include <stdlib.h>    /* needed to define exit() */
#include <unistd.h>    /* needed to define getpid() */
#include <stdio.h>     /* needed for printf() */

int
main() {

      printf("About to run ls\n");
      execlp("ls", "ls", "-aF", "/", (char*)0);
      perror("execlp");      /* if we get here, execlp failed */
      exit(1);
}
```

Save this file, compile it and run.

execvp

Source: https://people.cs.rutgers.edu/~pxk/416/notes/c-tutorials/index.html

The function *execvp* is almost like the system call *execve* except that the PATH environment variable is used as a search path for the command and the default environment is used, so you don't have to worry about passing the environment or searching for the command. Here's an example:

```
char **av[] = { "ls", "-al", "/usr/paul", (char *)0};

execvp("ls", av);
```

```
/* execvp: run a program using execvp */

/* Paul Krzyzanowski */

#include <stdlib.h>    /* needed to define exit() */
#include <unistd.h>    /* needed to define getpid() */
#include <stdio.h>     /* needed for printf() */

int main() {
      char *args[] = {"ls", "-aF", "/", 0};  /* each element represents a command line argument */

      printf("About to run ls\n");
      execvp("ls", args);
      perror("execvp");      /* if we get here, execvp failed */
      exit(1);
}
```

Save this file, compile it and run.

Spawning a new program with fork + exec

The fork system call creates a new process. The execve system call overwrites a process with a new program. These two system calls go together to spawn a new running program. A process forks itself and the child process execs a new program, which overlays the one in the current process.

Because the child process is initially running the same code that the parent was and has the same open files and the same data in memory, this scheme provides a way for the child to

tweak the environment, if necessary, before calling the *execve* system call. A common thing to do is for the child to change its standard input or standard output (file descriptors 0 and 1, respectively) and then exec the new program. That program will then read from and write to the new sources.

Example

This is a program that forks itself and has the chiled exec the *ls* program, running the "`ls -aF /`" command. Five seconds later, the parent prints a message saying, *I'm still here!*. We use the *sleep* library function to put the process to sleep for five seconds.

```
/* forkexec: create a new process. */
/*  The child runs "ls -aF /". The parent wakes up after 5 seconds */
/* Paul Krzyzanowski */

#include <stdlib.h>    /* needed to define exit() */
#include <unistd.h>    /* needed for fork() and getpid() */
#include <stdio.h>     /* needed for printf() */

int main() {

      int pid;    /* process ID */

      switch (pid = fork()) {
      case 0:           /* a fork returns 0 to the child */
            runit();
            break;

      default:    /* a fork returns a pid to the parent */
            sleep(5);   /* sleep for 5 seconds */
            printf("I'm still here!\n");
            break;

      case -1:    /* something went wrong */
            perror("fork");
            exit(1);
      }
      exit(0);
}
```

Source: https://people.cs.rutgers.edu/~pxk/416/notes/c-tutorials/index.html

```
void runit() {
      printf("About to run ls\n");
      execlp("ls", "ls", "-aF", "/", (char*)0);
      perror("execlp");      /* if we get here, execlp failed */
      exit(1);
}
```

Save this file, compile it and run.

Source: https://people.cs.rutgers.edu/~pxk/416/notes/c-tutorials/index.html