7. Processes

Terminology

- Program - Programs are the executable instructions of a program; either source code or compiled code.
- Process - Processes are the running instance of a program. A process includes the machine code and information about the current state of the process such as current values of variables.

Memory Structure

- Program Code
- Globals - Holds the values of global variables.
- Heap - Holds the values of `malloc`'d variables .
- Stack - Holds the values of current local variables of the function(s) being executed.
- Operating System
    - Process Control Block - A data structure that stores additional information about the state of a process
        - PID - Process ID (a unique identifier of the process)
        - PPID - Parent PID (PID of the parent process)
        - PGID - Process Group ID - Unique ID of the process group
        - SID - Session ID
        - Program Counter - Identifies the next instruction to be executed
        - Stack Pointer - Identifies the top of the stack
        - Open File Table
        - Signal Table - Pointer to code to handle a signal
        - Other things that the OS manages

Process States

The number of CPUs determines how many processes can execute instructions at a time. The scheduler determines which process should be run and when.

- Running State - Processes that are executing instructions simultaneously.
- Ready State - Processes that are ready and would be executing instructions if a CPU were available.
- Blocked State - Processes that are waiting for an event to occur. They are in a blocked or sleeping state.

Creating Processes

Processes can be created using the `fork` system call. This is done by duplicating the process calling `fork`. The operating system duplicates a process by copying its address space, data, and process control block. The child process is almost identical to its parent, differing only in its PID, its Parent PID, and the return value of `fork`.

Upon successful completion, `fork` returns a value of `0` to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of `-1` is returned to the parent process, no child process is created, and the global variable `errno` is set to indicate the error.

The child process has the same value for the program counter as the original process, so when the child process runs, it starts executing just after fork returns. However, when the operating system is finished creating the new process, control can be returned to either the parent *or* the child.

The parent and child processes are now completely separate processes and don't share memory.

Refer to `man fork` for more details.

Parent-Child Relationship and Termination

A parent process can terminate before its child process(es).

The `wait` system call can be used to suspend the execution of the parent process until one of its child processes has terminated. It has one parameter (`int *stat_loc`), a pointer to an `int`. On return from a successful `wait` call, the parameter contains termination information about the process that exited.

The termination information consists of two parts

- The lowest 8 bits indicate whether the child process terminated normally, or whether it terminated because it received a signal. If it terminated due to a signal, the lower 8 bits indicate which signal.
- The next `8` bits contain the exit status.

Macros defined in `sys/wait.h` can be used to extract this information

- `WIFEXITED(status)` - True if the process terminated normally by a call to `exit`.
- `WEXITSTATUS(status)` - If `WIFEXITED(status)` is true, evaluates to the low-order 8 bits of the argument passed to `exit` by the child.
- `WIFSIGNALED(status)` - True if the process terminated due to receipt of a signal.
- `WTERMSIG(status)` - If `WIFSIGNALED(status)` is true, evaluates to the number of the signal that caused the termination of the process.

If `wait` returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

The `wait` system call will fail and return immediately if the calling process has no existing unwaited-for child processes and `errno` will be set to `ECHILD`.

Refer to `man 2 wait` for more details.

Zombie and Orphan Processes

A zombie process is a process that has terminated but still has an entry in the process table. The operating system can't delete the process control block of a terminated process until it knows it is safe to clean it up. Hence, it is only deleted once `wait` is called and its termination status is collected.

An orphan process is a process that is still executing but whose parent process has terminated. The parent PID of an orphan process is `1`, which is the PID of the `init` process. This is because orphan processes are adopted by the `init` process. The `init` process calls `wait` in a loop, hence terminating the processes it adopts.

If the parent process of a child process terminates without calling `wait` and collecting the child's termination status, the child process becomes an orphan. It is then adopted by the `init` process, which calls `wait` and terminates it.

Executing Programs - The `execl` and `execv` family

Explanation

The `execv` and `execl` family of functions replaces the current process image with a new process image. The functions are front-ends for the function `execve`.

The new process is constructed from an ordinary file, whose name is pointed to by `path`, called the new process file. This file is either an executable object file, or a file of data for an interpreter. The system tells whether a file contains a program by looking at its first two bytes or so. If these contain a special value called the magic number, then the system treats the file as a program.

The PID and the parent PID of the process remain the same.

Usage

The `v` stands for vector and `l` for list since these functions accept the arguments as variadic array and an array of strings respectively.

The prototype for `execl` is `int execl(const char *path, const char *arg0, ..., /*, NULL */);`

The prototype for `execv` is `int execv(const char *path, char *const argv[]);`

The initial argument for these functions is the pathname of a file which is to be executed.

The `const char *arg0` and subsequent ellipses in the `execl` can be thought of as `arg0`, `arg1`, ..., `argn`. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments must be terminated by a `NULL` pointer.

The `execv` provides an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the file name associated with the file being executed. The array of pointers must be terminated by a `NULL` pointer.

If any of the `exec` functions returns, an error will have occurred. The return value is `-1`, and the global variable errno will be set to indicate the error.

Commonalities between the caller and the new process

File descriptors open in the calling process image remain open in the new process image, except for those for which the close-on-exec flag is set. Signals set to be ignored in the calling process are set to be ignored in the new process. Signals which are set to be caught in the calling process image are set to default action in the new process image. Blocked signals remain blocked regardless of changes to the signal action. The real user ID, real group ID and other group IDs of the new process image remain the same as the calling process image.

The new process also inherits the following attributes from the calling process - process ID, parent process ID, process group ID, working directory, root directory, control terminal, signal mask

Refer to `man 3 exec` and `man execve` for more details

Terminating Processes

`exit` is used to terminate a process, although a process will also stop when it runs out of program by reaching the end of the main function, or when main executes a return statement.

The single, integer argument to exit is called the process' exit status. It is normally used to indicate the success or failure of the task performed by the process. By convention a process returns zero on normal termination, some non-zero value if something has gone wrong.

A call to `exit` flushes all open output streams closes all open file descriptors. `exit` will also call any programmer-defined exit handling routines and perform what are generally described as clean-up actions. A programmer can also set at least 32 exit handling routines with the `atexit` function. Each of the exit handling functions recorded by `atexit` will be called on exit in the reverse order to which they were set.

`exit` does nothing to prevent bottomless recursion should a function registered using `atexit` itself call `exit`. Such functions must call `_exit` instead.

Files

Every process expects three files to be open, those on file descriptors `0`, `1` and `2`. By default, these correspond to the standard input, standard output, and standard error.

We can redirect where processes take their input from or output to by opening files on file descriptors `0` and `1` respectively.

A successful `open` always gives us the lowest available file descriptor number. If file descriptors `0`, `1`, and `2` are open, and we call `open` successfully, it will give us file descriptor `3`. However, if we close file descriptor `1` before calling `open`, so that the open file descriptors are now only `0` and `2`, then a successful `open` will return `1`, because that's the lowest available file descriptor number. So, the file will be open on file descriptor `1`.

Files kept open across a call to fork remain intimately connected in child and parent. This is because the read-write pointer for each file is shared between the child and the parent. This is possible because the read-write pointer is maintained by the system; it is not embedded explicitly within the process itself. Consequently, when a child process moves forward in a file, the parent process will also find itself at the new position.

Refer to [8. Pipes and Unbuffered IO]

Sample Code

`fork` and `wait`

```
int main() {
        int result;
        int i, j;

        printf("[%d] Original Process\n", getpid());

        for(i = 0; i < 5; i++){
                if((result = fork()) == -1) {  // Fork failed
                        perror("wait");
                        exit(0);  // Terminate parent process
                } else if(result == 0) {  // Child process
                        for(j = 0; j < 5; j++) {
                                printf("[%d] Child %d, %d\n",
getpid(), i, j);
                                usleep(100);  // Suspend thread
execution
                        }
                        exit(0);  // Terminate child process
                }
        }

        for(i = 0; i < 5; i++) {
                pid_t pid;
                int status;
                if( (pid = wait(&status)) == -1 ) {
                        perror("wait");
                } else {
                        printf("Child %d terminated with %d\n", pid,
status);
                }
        }

        printf("[%d] Parent about to terminate\n", getpid());
        return(0);
}
```

`execl`

```
int main()
{
    execl("/bin/cat", "cat", "a wacky file name illustrating that
each of these is an argument without reparsing", (char *)NULL);
    perror("/bin/cat");
    return(1);
}
```

`execv`

```
int main()
{
    char *x[3];

    x[0] = "cat";
    x[1] = "a wacky file name illustrating that each of these is an
argument without parsing";
    x[2] = NULL;
    execve("/bin/cat", x);
```

```
        perror("/bin/cat");
        return(1);
    }
```

`fork`, `exec`, and `wait`

```
int main()
{
    int x = fork();
    if (x == -1) {
        perror("fork");
        return(1);
    } else if (x == 0) {
        /* child */
        execl("/bin/ls", "ls", (char *)NULL);
        perror("/bin/ls");
        return(1);
    } else {
        /* parent */
        int status, pid;
        pid = wait(&status);
        printf("pid %d exit status %d\n", pid, WEXITSTATUS(status));
        return(0);
    }
}
```

Approximating `system`

```
int docommand(char *command) {
        pid_t pid;
        if((pid = fork()) < 0) {
                perror("fork");
                return(-1);
        } else if (pid == 0) {
                execl("/bin/sh", "sh", "-c", command, (char *) 0);
                perror("execl");
                exit(1);
        } else {
                wait((int *) 0);
```

```c
                return(0);
        }
}

int main() {
        char command[3] = "ls\0";
        return(docommand(command));
}
```
