Processes and Forking

A. Overview of Processes

- **Definition:**
  A process is an instance of a program in execution. It has its own memory space, file descriptor table, and process control block (PCB) maintained by the operating system.
- **Key Concepts:**
  - **Concurrency:** Multiple processes can execute concurrently (or in parallel on multi-core systems).
  - **Process Table:** The operating system keeps a list (process table) that stores information like the process ID (PID), program counter, and resource usage.
  - **Copy-on-Write:** When a new process is created via fork(), the child initially shares the parent's memory pages until one process writes, at which point a copy is made.

B. The fork() System Call

- **Purpose:**
  `fork()` creates a new process by duplicating the calling process. Both parent and child continue execution from the statement after the fork.
- **Return Values:**
  - **Child Process:** Receives 0 from fork().
  - **Parent Process:** Receives the child's PID (a positive number).
  - **Error:** Returns -1 on failure.

C. Code Example: Basic Fork

c
Copy
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();   // Create a new process.

    if (pid < 0) {        // Error in forking.
        perror("fork failed");
        exit(1);
```

```
    }

    if (pid == 0) {       // Child process block.
        printf("Child: My PID is %d\n", getpid());
        // Child-specific work here.
        exit(0);        // Terminate child.
    } else {              // Parent process block.
        printf("Parent: My PID is %d and my child's PID is %d\n",
getpid(), pid);
        wait(NULL);     // Wait for the child process to finish.
    }

    return 0;             // Both processes eventually exit.
}
```

D. Additional Forking Concepts from the Lectures

- **Multiple Forks:**
  When you call `fork()` repeatedly, each process gets its own copy of the parent's memory—including variables and open file descriptors. (See Lecture 1 and Lecture 2 slides for diagrams.)
- **Process Scheduling:**
  The operating system scheduler decides which process runs when. Therefore, you cannot assume which process (parent or child) will complete first.
- **Zombie Processes:**
  A child that terminates but has not been reaped by its parent becomes a zombie (still holding an entry in the process table). Using `wait()` or `waitpid()` avoids zombie processes.

Visual Diagrams (Text-Based):

ruby

Copy

```
Before fork():
  [Process S]
  FD Table: {0:stdin, 1:stdout, 2:stderr}

After fork():
  Parent Process (S):
    FD Table: {0:stdin, 1:stdout, 2:stderr}
  Child Process (S'):
```

```
    FD Table: {0:stdin, 1:stdout, 2:stderr}  (identical copy, then
diverges on write)
```

-

---

2. Redirection and File Descriptors

A. Standard File Descriptors

- **FD 0:** `STDIN_FILENO` – standard input
- **FD 1:** `STDOUT_FILENO` – standard output
- **FD 2:** `STDERR_FILENO` – standard error

B. Redirection in the Shell and in C Programs

- **Shell Redirections:**
  - `< infile` – redirect standard input
  - `> outfile` – redirect standard output (overwrite)
  - `>> outfile` – append standard output
  - `2> outfile` – redirect standard error
  - `&>` – redirect both output and error
- **Redirection using C Functions:**
  Redirection in C is typically done by using the **dup2()** system call along with **open()**.

C. Code Example: Redirecting Output to a File Using dup2()

c
Copy

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

int main() {
    int fd = open("output.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open failed");
        exit(1);
    }
```

```
    // Redirect standard output (fd 1) to the file.
    if (dup2(fd, STDOUT_FILENO) < 0) {
        perror("dup2 failed");
        exit(1);
    }

    // From here, all output to stdout goes to "output.txt"
    printf("This will be written to output.txt\n");

    close(fd);  // Safe to close original descriptor; fd1 still refers
to output.txt.
    return 0;
}
```

D. Lecture Highlights on Redirection

- **Redirection and FD Tables:**
  The lectures (Lecture 3) explain that file descriptors are stored in a table. When you use dup() or dup2(), you are modifying the mapping in that table.
- **Duplication Example:**
  Using `dup()`, the OS returns the next available FD that points to the same open file table entry.
  Using `dup2(oldfd, newfd)`, the file descriptor newfd is closed (if open) and then made to refer to the same file as oldfd.
- **Use Cases:**
  Redirecting output from one program to a file before invoking an exec call, so that the new program's output is captured.

---

3. File Descriptor Duplication (dup() and dup2())

A. The dup() Function

- **Purpose:**
  Duplicates an existing file descriptor into the lowest numbered unused descriptor.
- **Code Example:**

c

Copy

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

int main() {
    int fd = open("dup.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open failed");
        exit(1);
    }
    int copy_fd = dup(fd);
    if (copy_fd < 0) {
        perror("dup failed");
        exit(1);
    }
    write(copy_fd, "This is written using dup()\n", 28);
    write(fd, "This is written using original fd\n", 34);
    close(fd);
    close(copy_fd);
    return 0;
}
```

B. The dup2() Function

- **Purpose:**
  Duplicates one file descriptor to a specific file descriptor number. If that target is already open, it is closed first.
- **Code Example (same as in redirection):**
  (See the redirection example above.)
- **When to Use:**
  It is most commonly used to redirect standard input, output, or error.

---

4. Interprocess Communication with Pipes

A. Overview of Pipes

- **Definition:**
  A pipe is a unidirectional communication channel that connects the output of one process to the input of another.
- **File Descriptors:**
  - `pipefd[0]` is the read end.
  - `pipefd[1]` is the write end.

B. Creating and Using a Pipe

Syntax:

c

Copy

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

-
  - On success, two file descriptors are provided.
- **Crucial Steps:**
  - **Close Unused Ends:** For processes that only read or only write, close the opposite end.
  - **EOF and Cleanup:** All write ends must be closed to signal EOF to the reader.

C. Code Example: Simple Pipe

c

Copy

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main() {
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        perror("pipe failed");
        exit(1);
    }

    pid_t pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
```

```
    }

    if (pid == 0) {  // Child process: writes to the pipe.
        close(pipefd[0]);  // Close read end.
        char *message = "Hello from pipe!";
        if (write(pipefd[1], message, strlen(message) + 1) == -1) {
            perror("write failed");
            exit(1);
        }
        close(pipefd[1]);  // Signal EOF.
        exit(0);
    } else {         // Parent process: reads from the pipe.
        close(pipefd[1]);  // Close write end.
        char buffer[100];
        if (read(pipefd[0], buffer, sizeof(buffer)) == -1) {
            perror("read failed");
            exit(1);
        }
        printf("Parent received: %s\n", buffer);
        close(pipefd[0]);
        wait(NULL);
    }
    return 0;
}
```

D. Advanced Pipe Setup (Pipelines)

- **Two-Stage Pipelines:**
  You can create a pipeline that connects the output of one process (child A) to the input of another process (child B). The typical strategy is:
  1. Create a pipe.
  2. Fork a child (A) and redirect its standard output (using dup2) to the write end of the pipe; then exec a program (e.g., `cat`).
  3. Fork another child (B) and redirect its standard input (using dup2) to the read end of the pipe; then exec a program (e.g., `tr` to translate text).
  4. The parent closes the unused ends and waits for both children.

Diagram (Text-Based):

css

Copy
```
Parent Process
   ├─ creates pipe -> pipefd[0] (read), pipefd[1] (write)
   ├─ fork Child A:
   |     Child A:
   |       dup2(pipefd[1], STDOUT_FILENO)  // stdout now goes to pipe
   |       close(pipefd[0]); close(pipefd[1]);
   |       execvp("cat", ["cat", "filename", NULL]);
   |
   └─ fork Child B:
         Child B:
           dup2(pipefd[0], STDIN_FILENO)  // stdin comes from pipe
           close(pipefd[0]); close(pipefd[1]);
           execvp("tr", ["tr", "a-z", "A-Z", NULL]);
```

-
- **Atomicity Note:**
  When writing to a pipe, writes are atomic if they are less than PIPE_BUF (typically 4096 bytes on Linux).

---

5. The exec Family of Functions

A. Overview

- **Purpose:**
  The exec functions replace the current process image with a new program. They are normally used after fork() in the child process.
- **Key Point:**
  On success, an exec call never returns. If it does, an error occurred.

B. Variants and Their Usage

1. execl

Prototype:

c

Copy
```c
int execl(const char *path, const char *arg0, ..., (char *)NULL);
```

-

- **Usage:**
  Pass a list of arguments, terminated by a NULL pointer.

Example:
c
Copy
```
if (execl("/bin/ls", "ls", "-l", (char *)NULL) == -1) {
    perror("execl failed");
}
```

-

2. execv
Prototype:
c
Copy
```
int execv(const char *path, char *const argv[]);
```

-
- **Usage:**
  Pass an array of argument strings (the last element must be NULL).

Example:
c
Copy
```
char *argv[] = {"ls", "-l", NULL};
if (execv("/bin/ls", argv) == -1) {
    perror("execv failed");
}
```

-

3. execlp and execvp

- **Key Difference:**
  These functions search for the executable in the directories listed in the PATH environment variable.

Example using execvp:
c
Copy
```
char *argv[] = {"ls", "-l", NULL};
if (execvp("ls", argv) == -1) {
    perror("execvp failed");
```

```
}
```

-

4. execve
Prototype:
c
Copy
```
int execve(const char *filename, char *const argv[], char *const
envp[]);
```

-
- **Usage:**
  This is the underlying system call that allows you to specify a custom environment.

Example:
c
Copy
```
char *argv[] = {"ls", "-l", NULL};
char *envp[] = {"PATH=/bin:/usr/bin", NULL};
if (execve("/bin/ls", argv, envp) == -1) {
    perror("execve failed");
}
```

-

C. Typical Pattern with fork() and exec

c
Copy
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();   // Create a child process.

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
```

```
    else if (pid == 0) {  // In the child process.
        char *argv[] = {"ls", "-l", NULL};
        // Replace the child process image with "ls -l"
        if (execvp("ls", argv) == -1) {
            perror("execvp failed");
            exit(1);
        }
    }
    else {              // In the parent process.
        int status;
        wait(&status);
        printf("Child process finished.\n");
    }

    return 0;
}
```

---

6. Combining Concepts: Processes, Redirection, and Pipes

A. High-Level Strategy for Pipelines

- **Goal:**
  Use multiple child processes where one process's output (stdout) becomes another's input (stdin) via a pipe.
- **Steps:**
  1. **Create a Pipe:**
     Call `pipe(pipefd)` to create the communication channel.
  2. **Fork Child A:**
     In Child A, use `dup2(pipefd[1], STDOUT_FILENO)` to redirect its standard output to the write end of the pipe. Then execute a program (e.g., `cat filename`).
  3. **Fork Child B:**
     In Child B, use `dup2(pipefd[0], STDIN_FILENO)` to redirect its standard input to the read end of the pipe. Then execute a program (e.g., `tr a-z A-Z`).

4. **Clean-Up:**
   Close unused file descriptors in both children and the parent so that EOF conditions occur properly.

B. Code Example: Simple Two-Stage Pipeline

c

Copy

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pipefd[2];
    if(pipe(pipefd) == -1) {
        perror("pipe failed");
        exit(1);
    }

    // Fork first child (Producer)
    pid_t pid1 = fork();
    if(pid1 < 0) {
        perror("fork failed");
        exit(1);
    }
    if(pid1 == 0) {
        // Child A: redirect stdout to pipe write end.
        close(pipefd[0]);             // Close unused read end.
        if(dup2(pipefd[1], STDOUT_FILENO) == -1) {
            perror("dup2 failed");
            exit(1);
        }
        close(pipefd[1]);             // Close original write end.
        execlp("cat", "cat", "input.txt", (char *)NULL);  // Example
executable.
        perror("execlp failed");
        exit(1);
    }
```

```c
    // Fork second child (Consumer)
    pid_t pid2 = fork();
    if(pid2 < 0) {
        perror("fork failed");
        exit(1);
    }
    if(pid2 == 0) {
        // Child B: redirect stdin from pipe read end.
        close(pipefd[1]);             // Close unused write end.
        if(dup2(pipefd[0], STDIN_FILENO) == -1) {
            perror("dup2 failed");
            exit(1);
        }
        close(pipefd[0]);             // Close original read end.
        execlp("tr", "tr", "a-z", "A-Z", (char *)NULL);  // Example:
convert text to uppercase.
        perror("execlp failed");
        exit(1);
    }

    // Parent process: close both ends of the pipe.
    close(pipefd[0]);
    close(pipefd[1]);
    wait(NULL);
    wait(NULL);

    return 0;
}
```

C. Lecture Insights on Pipeline Setups

- **Clean Up Is Critical:**
  As the lectures stress, every process that does not need a pipe end (read or write) must close it. Failure to close unused ends means the reader may block forever waiting for EOF.
- **Redirecting FDs for Exec:**
  Note that when a child process calls exec, its file descriptor table (including any redirections set by dup2) is inherited by the new executable. Many system programs rely on FDs 0, 1, and 2.

- **Multi-Stage Pipelines:**
  The same design pattern can be extended to more than two processes by chaining multiple pipes together.

---

7. Summary of Key Points

- **Processes:**
  - Created via fork().
  - Execute concurrently with independent memory (using copy-on-write).
  - Inherit file descriptors from the parent.
- **File Descriptors & Redirection:**
  - Standard FDs are 0 (stdin), 1 (stdout), 2 (stderr).
  - Use dup() / dup2() to duplicate or redirect FDs.
- **Pipes:**
  - Provide a unidirectional communication channel.
  - Must be set up carefully with proper closing of unused ends.
- **exec Family:**
  - Replace the current process image with a new executable.
  - Common variants: execl, execv, execlp, execvp, execve.
  - Typically used in the child after fork().
- **Combining All Concepts:**
  - A common use case is a shell: it forks processes, redirects FDs using dup2(), creates pipes for IPC, and then calls exec to run commands.
  - Understanding the file descriptor table and proper clean-up is key to preventing deadlocks and ensuring correct data flow between processes.

Additional Study References

- **Lecture Slides:**
  - *Part 2 Lecture 1: Introduction to Processes* – Basic concepts, fork(), process memory, and scheduling.
  - *Part 2 Lecture 2: More on Processes* – Detailed forking examples and multi-fork issues.
  - *Part 2 Lecture 3: Redirection* – How file descriptors work, redirection via dup/dup2, and shell I/O redirection concepts.
  - *Part 2 Lecture 4: Pipes* – Pipe creation, usage in interprocess communication, and setting up pipelines with multiple processes.
- **Manual Pages:**

- Use `man 2 fork`, `man 2 dup`, `man 2 dup2`, `man 2 pipe`, and `man 2 execvp` on Linux for more detailed system call documentation.

These notes—complete with code examples and conceptual diagrams—should serve as a comprehensive cheat sheet for understanding processes, redirection, file descriptor manipulation, and interprocess communication using pipes in Unix systems. Happy coding and studying!