Process Creation with fork()

Overview

- **fork()** creates a new child process that is an almost identical copy of the parent.
- The return value:
    - **Child:** `fork()` returns 0.
    - **Parent:** `fork()` returns the child's process ID (PID).
    - **Error:** If fork fails, it returns a negative value.

Code Example

c

Copy

```
#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 occurred
        perror("fork failed");
        exit(1);
    }

    if (pid == 0) {       // Child process block
        printf("Hello from child process, PID = %d\n", getpid());
        exit(0);
    } else {              // Parent process block
        printf("Hello from parent process, PID = %d, child's PID =
%d\n", getpid(), pid);
        wait(NULL);       // Wait for the child to finish
    }
    return 0;
}
```

---

2. File Descriptor Duplication with dup() and dup2()

dup()

- **dup()** duplicates an open file descriptor.
- The new descriptor is the lowest-numbered available descriptor.

Code Example

c

Copy

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>   // For open()

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

    int newfd = dup(fd);      // Duplicate fd
    if (newfd < 0) {
        perror("dup failed");
        exit(1);
    }

    write(fd, "Hello using fd\n", 16);          // Write using
original fd
    write(newfd, "Hello using newfd\n", 19);      // Write using new
descriptor

    close(fd);
    close(newfd);

    return 0;
}
```

dup2()

- **dup2(oldfd, newfd)** duplicates `oldfd` into a specific descriptor (`newfd`). If `newfd` is open, it is closed first.

Code Example

c

Copy

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.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 (STDOUT_FILENO, which is 1) to
output.txt
    if(dup2(fd, STDOUT_FILENO) < 0) {
        perror("dup2 failed");
        exit(1);
    }

    printf("This will go to output.txt file instead of the
terminal\n");
    close(fd);

    return 0;
}
```

---

3. Interprocess Communication with pipe()

Overview

- **pipe()** creates a unidirectional data channel.
- It returns two file descriptors:
    - One for reading, and one for writing.
- Crucial: Always close unused ends in parent/child processes.

Code Example

c

Copy

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

int main() {
    int pipefd[2];    // pipefd[0] is read end, pipefd[1] is write
end.
    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 child";
        if(write(pipefd[1], message, strlen(message) + 1) == -1) {
            perror("write failed");
            exit(1);
        }
        close(pipefd[1]);        // Close write end.
        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]);        // Close read end.
        wait(NULL);
    }

    return 0;
}
```

---

4. open() Flags for File Descriptors

When opening a file with `open()`, you choose flags that determine read/write behavior:

Access Mode Flags (choose one):

Flag	Description
`O_RDONLY`	Read-only
`O_WRONLY`	Write-only
`O_RDWR`	Read and write

Additional Flags (can be combined):

Flag	Description
`O_CREAT`	Create file if it does not exist (requires mode argument).

O_EXCL	Fails if file already exists (with O_CREAT).
O_TRUNC	Truncate file to zero length if it exists.
O_APPEND	All writes will be appended to the end of file.
O_NONBLOCK	Open in non-blocking mode.
O_SYNC	Writes are synchronized to disk.
O_CLOEXEC	Set close-on-exec flag; descriptor is closed on exec.

Example Usage:

c
Copy

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

---

5. Standard File Descriptors

Standard file descriptors (FDs) defined in `<unistd.h>`:

FD Number	Macro	Description
0	STDIN_FILENO	Standard Input
1	STDOUT_FILENO	Standard Output
2	STDERR_FILENO	Standard Error Output

---

6. Printing a String in C

Using `printf()`

c

Copy

```c
#include <stdio.h>
int main() {
    char name[] = "Alice";
    printf("Hello, %s!\n", name);
    return 0;
}
```

Using `puts()`

c

Copy

```c
#include <stdio.h>
int main() {
    puts("This is a simple string printed with puts");
    return 0;
}
```

Using `write()` (for low-level file descriptor printing)

c

Copy

```c
#include <unistd.h>
int main() {
    write(STDOUT_FILENO, "Hello via write!\n", 18);
    return 0;
}
```

---

7. exec Family of Functions

The exec functions replace the current process image with a new program. They do not return unless an error occurs.

Common Variants

a. execl

c

Copy

```
#include <unistd.h>
#include <stdio.h>
int main() {
    if (execl("/bin/ls", "ls", "-l", (char *)NULL) == -1) {
        perror("execl failed");
    }
    return 0;
}
```

b. execv

c

Copy

```
#include <unistd.h>
#include <stdio.h>
int main() {
    char *argv[] = {"ls", "-l", NULL};
    if (execv("/bin/ls", argv) == -1) {
        perror("execv failed");
    }
    return 0;
}
```

c. execlp / execvp

- **execlp** and **execvp** search for the executable in the system's PATH.

c

Copy

```
#include <unistd.h>
#include <stdio.h>
int main() {
    char *argv[] = {"ls", "-l", NULL};
    if (execvp("ls", argv) == -1) {
        perror("execvp failed");
    }
    return 0;
}
```

d. execve

Allows passing a custom environment.

c

Copy

```
#include <unistd.h>
#include <stdio.h>
int main() {
    char *argv[] = {"ls", "-l", NULL};
    char *envp[] = { "PATH=/bin:/usr/bin", NULL };
    if (execve("/bin/ls", argv, envp) == -1) {
        perror("execve failed");
    }
    return 0;
}
```

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();   // fork a child process

    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    else if (pid == 0) {  // Child process
        char *argv[] = {"ls", "-l", NULL};
        if (execvp("ls", argv) == -1) {
            perror("execvp failed");
            exit(1);
        }
    }
    else {              // Parent process
        int status;
        wait(&status);   // Wait for child
        printf("Child process finished\n");
    }
    return 0;
}
```

---

8. Puzzle-Solving with fork(), pipe(), and State Updates

Overview

- The puzzle array is defined as:
  `int a[] = {3, 6, 4, 1, 3, 4, 2, 5, 3, 0};`
- The goal is to use fork() to explore different paths (left and right moves) until the walker reaches the goal index (9, where value is 0).
- A pipe is used to collect and print out solution strings from the child processes.
- **Key Steps in Each Branch:**
    - **Record state:** Save the current index in the path array (`b`).
    - **Increment moves:** Increase the move counter.
    - **Update current index:**
        - For right move: `cur = cur + a[cur];`
        - For left move: `cur = cur - a[cur];`

Code Snippet for the Move Branches

c

Copy

```
// For the right move branch in the child:
if(a[cur] == 0) {
    b[moves - 1] = cur;
    write_solution(b, moves, pd[1]);
    close(pd[1]);
    return 0;
} else if(cur + a[cur] >= 0 && cur + a[cur] < n) {
    b[moves] = cur;      // Record current index.
    moves++;             // Increment move counter.
    cur = cur + a[cur];  // Move right.
}

// For the left move branch in the parent:
if(a[cur] == 0) {
    b[moves - 1] = cur;
    write_solution(b, moves, pd[1]);
    close(pd[1]);
    return 0;
} else if(cur - a[cur] >= 0 && cur - a[cur] < n) {
    b[moves] = cur;      // Record current index.
    moves++;             // Increment move counter.
    cur = cur - a[cur];  // Move left.
}
```

Additional Key Points

- **Pipe Closure:**
  Close all unused ends of the pipe in both child and parent processes to allow EOF to be detected.
- **Return Handling:**
  When a solution is written to the pipe, close the write end and exit the process so that the parent can stop waiting.
- **Separate Process State:**
  Remember that fork creates separate copies; therefore, updating variables in one process does not affect others.

Summary

This note sheet combined our discussions of:

- Creating processes with **fork()**.

- Duplicating file descriptors with **dup()** and **dup2()**.
- Setting up interprocess communication using **pipe()**.
- Using various **open()** flags for file I/O.
- The standard file descriptors (**STDIN, STDOUT, STDERR**).
- Ways to print strings in C (**printf, puts, write**).
- The exec family of functions (**execl**, **execv**, **execlp**, **execvp**, **execve**) and their typical usage with **fork()**.
- How to structure a puzzle solver that uses forks and pipes to track and report solutions.

Each section includes code examples with explanations that you can reference as a cheat sheet when writing your own code. Happy coding!