Department of Computer Science and Engineering
Indian Institute Of Technology Bombay
CS 347M/CS219 Autumn 2025 www.cse.iitb.ac.in/~puru/courses/autumn2025/

CS 347m/CS219 Operating Systems
Autumn 2025
Lab 2: Stay calm and trust the process

Instructions

- Login with username **labuser** on software lab (SL1/SL2/SL3) machines for this lab.
- This file is part of the **lab2.tar.gz** archive which contains multiple directories with programs associated with the exercise questions listed below.
- Most questions of this lab are tutorial style and are aimed to provide an introduction to tools and files related to system and process information and control.

Part 1: Forking processes in C ?

`Process Management`

In operating systems, processes are the fundamental units of execution. A process can create another process, known as a child process, using the **fork()** system call in UNIX-like systems. The **fork()** call creates a new process by duplicating the calling process. The new process (child) is an exact copy of the calling process (parent), except for a few attributes like the process ID (PID). The parent can wait for the child to complete using the **wait()** system call and retrieve its exit status, which is useful for understanding how the child process terminated. This exercise explores creating and managing processes using **fork()**, **getpid()**, **getppid()**, **wait()**, and **WEXITSTATUS()** to handle process IDs and exit statuses effectively.

Task A1.
Write a C program **task_a1.c** that creates a child process using **fork()**. The parent process should print its own process ID and the child's process ID, then wait for the child to terminate and print the child's exit status. The child process should print its own process ID and its parent's process ID, then exit with a specific status code (e.g., 42). Ensure the program handles fork errors appropriately.

Requirements

- Use the fork(), getpid(), getppid(), wait(), and WEXITSTATUS() system calls/macros.
- The child process must exit with status code 42.
- The parent must wait for the child to terminate before printing the child's exit status.
- Handle fork errors by printing an error message and returning 1.

Sample Output

```
Parent: My process ID is: 10856

Parent: The child process ID is: 10857

Child: My process ID is: 10857

Child: My parent process ID is: 10856

Parent: Child process exited with status: 42
```

System calls of interest ---

- fork(): Creates a new process.
- getpid(): Returns the process ID of the calling process.
- getppid(): Returns the process ID of the parent process.
- wait(): Makes the parent wait for the child to terminate.
- WEXITSTATUS(): Extracts the exit status of the terminated child.

Task A2.

Write a C program **task_a2.c** that prints a prompt "Enter program: " and reads the name of an executable program (located in the same folder) from the terminal/console. The program should fork a child process, and the child should use a variant of the **exec** family to execute the specified program. The parent process should wait for the child to terminate, then print a completion message indicating the child's PID and the executed program's name, before displaying the prompt again to accept another program name. This process should repeat until interrupted (e.g., by Ctrl+C). Handle errors for invalid program names and fork failures appropriately.

Requirements

- Use **fork()**, a variant of **exec** (e.g., execlp), wait(), and standard I/O functions (printf, scanf).
- Print the prompt "Enter program: " before reading input.
- The child process executes the specified program using an exec variant.
- The parent waits for the child to terminate, then prints a message like "Child [PID] finished executing [program]".
- Handle fork errors by printing an error message and exiting with status 1.
- Handle **exec** errors in the child by printing an error message and exiting with a non-zero status.
- The program loops indefinitely until interrupted (e.g., Ctrl+C).
- Assume the executable programs are in the same folder

Sample Output

```
Enter program: ./helloworld
```

```
This is hello world program

Child 7631 finished executing ./helloworld

Enter program: ./animeworld

This is anime world program

Child 7659 finished executing ./animeworld

Enter program: ^C
```

System calls of interest ---

- fork(): Creates a new process.
- execlp(): Executes a program, searching the PATH or current directory.
- wait(): Makes the parent wait for the child to terminate.
- getpid(): Returns the process ID of the calling process (optional, for printing in the completion message).

Task A3.

Write a C program **task_a3.c** that creates a parent process (Process A). Process A should prompt the user to input a positive integer and then use the **fork()** system call to spawn a child process (Process B). Process B should prompt the user to input another positive integer and multiply it with the integer received from Process A via a **pipe**. The parent process (Process A) should then read the result from the **pipe()** and print it. Use the **pipe()** system call for inter-process communication (IPC) between Process A and Process B. Handle errors for **pipe** creation, **fork**, and invalid inputs (non-positive integers).

Requirements

- Use **pipe(), fork(), write(), read(), getpid()**, and standard I/O functions (printf, scanf).
- Process A should set up the pipe later used for IPC.
- Process A reads a positive integer from the user and sends it to Process B via the pipe.
- Process B reads another positive integer from the user, multiplies it with the integer received from the pipe, and sends the product back to Process A via the pipe.
- Process A prints the product with a message including its PID.
- Validate that both inputs are positive integers (greater than 0); if not, print an error message and exit with status 1.
- Handle errors for pipe() and fork() by printing appropriate error messages and exiting with status 1.
- Close unused pipe ends in both processes to prevent resource leaks.

Sample Output

```
Process A (PID: 11272): Enter a positive
integer: 5
```

```
Process B (PID: 11273): Enter a positive
integer: 5

Process A (PID: 11272): Product is 25
```

System calls of interest ---

- pipe(): Creates a pipe, initializing an array of two file descriptors (fd[0] for reading, fd[1] for writing).
- fork(): Creates a new process.
- write(): Writes data to the write end of the pipe.
- read(): Reads data from the read end of the pipe.
- getpid(): Returns the process ID of the calling process.
- close(): Closes unused pipe file descriptors.

Task A4.

Write a C program **task_a4.c** that demonstrates the concept of an **orphan** process. The program should **fork** a child process. The parent process should print its process ID (PID) and the child's PID, while the child process should print its PID and its parent's PID. The child process should then **sleep** for 8 seconds, during which the parent process should exit. After waking up, the child process should print its PID and its new parent PID (which should be different, typically the PID of the init process or another system process that adopts orphans). Include a brief status message in the child to indicate it is now an orphan. Handle fork errors appropriately.

Requirements

- Use **fork()**, **getpid()**, **getppid(), sleep()**, and standard I/O functions (printf).
- The parent process prints its PID and the child's PID.
- The child process prints its PID and its parent's PID before sleeping.
- The child sleeps for 8 seconds, and the parent exits during this time.
- After waking, the child prints its PID, its new parent PID, and a message indicating it is an orphan.
- Handle fork errors by printing an error message and exiting with status 1.

Sample Output

```
Parent (PID: 11406): Child PID is 11407

Child (PID: 11407): Parent PID is 11406

cse@iitb:~/$ Child (PID: 11407): I am now an
orphan, new Parent PID is 1361
```

System calls of interest

- fork(): Creates a new process.

- getpid(): Returns the process ID of the calling process.
- getppid(): Returns the process ID of the parent process.
- sleep(): Suspends execution for a specified number of seconds.

Task A5.

Write a C program **task_a5.c** that takes a positive integer **n** as a command-line argument and creates n child processes ***recursively*** — the parent creates the first child, the first child creates the second child, and so on, up to n children. Each process should print a creation message with its process ID (PID) in the order of creation. The child processes should exit in the reverse order of creation (i.e., the last child exits first, then the second-to-last, and so on, up to the parent). Upon exiting, each process should print an exit message including its PID and its parent's PID. Use **wait()** to ensure proper ordering of children and parent termination. Handle errors for invalid command-line arguments and fork failures.

Requirements

- Use **fork()**, **getpid()**, **getppid()**, **sleep()**, **atoi()**, and standard I/O functions (printf).
- The program takes a single command-line argument n (a positive integer).
- Create n child processes recursively (parent → child 1 → child 2 → ... → child n).
- Each process prints a creation message in the format: Process [PID] created.
- Processes exit in reverse order (child n exits first, then child n-1, ..., then parent).
- Each process prints an exit message in the format: Process [PID] exiting, parent PID is [PPID].
- Use sleep(1) in the creation phase to ensure creation messages appear in order.
- Use sleep(2) in the exit phase to ensure exit messages appear in reverse order.
- Use wait in required places to ensure ordered exit of parent and child processes.
- Validate that n is a positive integer; if not, print an error message and exit with status 1.
- Handle fork errors by printing an error message and exiting with status 1.

Sample Command

```
.\taska5 5
```

Sample Output

```
Process 2592 created, parent pid is 1734
Process 2604 created, parent pid is 2592
Process 2615 created, parent pid is 2604
Process 2623 created, parent pid is 2615
Process 2624 created, parent pid is 2623
Process 2624 exiting, parent PID is 2623
Process 2623 exiting, parent PID is 2615
Process 2615 exiting, parent PID is 2604
Process 2604 exiting, parent PID is 2592

Process 2592 exiting, parent PID is 1734
```

Hint: As a first step, you want to write an initial version of the program that creates **n** child processes from the same parent process and waits till off them finish exit, before exiting itself.

System calls of interest

- fork(): Creates a new process.
- getpid(): Returns the process ID of the calling process.
- getppid(): Returns the process ID of the parent process.
- sleep(): Suspends execution for a specified number of seconds.
- wait(): Makes the parent wait for the child to terminate.
- atoi(): Converts a string to an integer for parsing the command-line argument.

Part 2: File Content Copying in C

System call based file operations are essential for precise control over file input and output. This exercise involves creating a C program that uses system calls such as **open()**, **read()**, **write()**, **lseek()**, and **close()** to manipulate file contents.

Task B1.

Write a C program called **task_b1.c** that demonstrates the management of file descriptors using system calls. The program should perform the following steps:

1. Open two files named **test1.txt** and **test2.txt** in read-only mode using the **open** system call.

2. Print the **file descriptors (fds)** of both opened files to standard output using the

write system call (do not use printf).

3. Close the first file descriptor (for test1.txt).

4. Open a third file named **test3.txt** in read-only mode and print its file descriptor. Observe if the descriptor is reused.

5. Use **fork()** to create a child process.

6. In the child process, use one of the inherited file descriptors (e.g., for test2.txt) to read some content from the file and print it to standard output using **write**.

7. In the parent process, wait for the child to finish using **wait()**.

8. Handle all possible errors for system calls like **open, close, read, fork, and wait**. Print error messages to standard error using write if any errors occur.

9. Ensure all file descriptors are properly closed in both parent and child processes to avoid resource leaks.

10. File descriptors are small integers assigned by the OS (starting from 3 usually, after 0=stdin, 1=stdout, 2=stderr).

11. When closing a fd (e.g., fd1), the next open may reuse that number (e.g., fd3 might equal the old fd1).

12. After fork, the child inherits copies of the parent's open fds, allowing the child to read/write to them independently.

13. Always check return values of system calls for errors (<0) and close fds to prevent leaks.

Sample Output

```
test1.txt fd: 3

test2.txt fd: 4

Closed test1.txt

test3.txt fd: 3

Child read from test2.txt: This is test2 file
content.

Parent: Child process finished
```

Task B2.

Write a C program **task_b2.c** that takes three filenames and an integer offset as command-line arguments. The parent process should copy the content of the **second file to the third file**, and then copy the content of the **first file to the third file starting**

at the specified offset. Use only filesystem system calls (**open, read, write, lseek, close**) for file operations, not C library functions from stdio.h (e.g., fopen, fread, fwrite, printf, etc.). Handle errors for invalid arguments, file operations, and ensure proper file descriptor management.

Requirements

- Use **open(), read(), write(), lseek(), close(), and atoi()** for processing.
- The program takes four command-line arguments: <program> <file1> <file2> <file3> <offset>.
- Copy the entire content of file2 to file3 starting at offset 0.
- Copy the entire content of file1 to file3 starting at the specified offset.
- The offset must be a non-negative integer; validate it and exit with status 1 if invalid.
- Handle errors for:
    - Incorrect number of command-line arguments.
    - Failure to open files (e.g., non-existent files, permission issues).
    - Failure in read(), write(), or lseek() operations.
- Use a buffer of size 4096 bytes for reading and writing.
- Close all file descriptors after use to prevent resource leaks.

Sample Command

```
./task_b2 file1 file2 file3 4
```

Sample Output

Before program execution

```
File1 Content

Bombay

File2 Content
IIT

File3 Content //Empty File
```

After program execution

```
File1 Content

Bombay

File2 Content
IIT

File3 Content

IIT Bombay
```

System calls of interest ---

- open(): Opens a file and returns a file descriptor.
- read(): Reads data from a file descriptor into a buffer.
- write(): Writes data from a buffer to a file descriptor.
- lseek(): Sets the file offset for reading or writing.
- close(): Closes a file descriptor.
- atoi(): Converts a string to an integer for parsing the offset.

Task B3.

Write a C program **task_b3.c** that takes **three filenames as command-line arguments**. The program must use system calls (**open, read, write, close**) for all file operations, not C library functions from stdio.h (e.g., fopen, fread, fwrite, printf, etc.). The program must:

1. Create a child process using **fork()**.

2. The parent process opens up three files, obtaining three file descriptors.

3. The parent process should copy the content of the first file to the second file.

4. The child process should copy the content of the first file to the third file, without creating any additional file streams than the ones already created.

5. Handle errors for invalid command-line arguments, file operations, and ensure proper file descriptor management.

Sample Command

```
./task_b3 file1 file2 file3
```

Sample Output
Before program execution

```
File1 Content
IIT Bombay

File2 Content //Empty


File3 Content //Empty
```

After program execution

```
File1 Content
IIT Bombay

File2 Content
IIT Bombay

File3 Content
IIT Bombay
```

System calls of interest ---

- **open()**: Opens a file and returns a file descriptor. Used to open the source file for reading and the two destination files for writing.
- **read()**: Reads data from a file descriptor into a buffer. Used to read content from the source file in both parent and child processes.
- **write()**: Writes data from a buffer to a file descriptor. Used to write content to the second destination file (parent) and the third destination file (child).
- **lseek():** Sets the file offset for reading or writing. Used in the child process to reset the source file's offset to the beginning before copying.
- **close()**: Closes a file descriptor. Used to close all open file descriptors in both parent and child processes after operations are complete.
- **fork()**: Creates a child process by duplicating the calling process. Used to create a child process that handles copying to the third file.
- **wait()**: Suspends execution of the parent process until the child process terminates. Used to ensure the parent waits for the child to complete its task.
- **exit()**: Terminates the calling process. Used to exit the program with an appropriate status code after errors or successful completion in the child process.

Submission Guidelines

- All submissions via moodle. Name your submissions as: <rollno_lab2>.tar.gz
- For example; if your roll number is 123456789 then the submission will be 123456789.tar.gz and if your roll number is 12D345678 then the submission will be 12d345678.tar.gz
- The tar should contain the following files in the following directory structure:

```
<rollnumber_lab2>/
|__part1/
      |____task_a1.c
      |____task_a2.c
      |____task_a3.c
      |____task_a4.c
      |____task_a5.c
|__part2/
      |____task_b1.c
      |____task_b2.c
      |____task_b3.c
```

- **Deadline: 20th August 2025, 5 pm.**