1 Prerequisite

1.1 Introduction to Unix System Programming

This assignment introduces fundamental concepts in Unix/Linux system programming, essential for understanding how processes are created, managed, and how they interact. These concepts form the bedrock for more advanced topics like inter-process communication, synchronization, and memory management, which you will explore in later assignments.

1.1.1 The Shell Environment

Before diving into programming, it's crucial to be comfortable with the Linux command-line interface.

- **Basic Commands Review:** `ls`, `cd`, `pwd`, `mkdir`, `rm`, `cp`, `mv`.
- **File Permissions:** Understanding `chmod`, `chown`, and `chgrp`.
- **Process Management Commands:** `ps`, `top`, `kill`, `pkill`.
- **Input/Output Redirection:** `>`, `>>`, `<`, `|`.
- **Environment Variables:** `echo $PATH`, `export`, `set`.

1.2 Process Creation and Management

Understanding how new processes are created and managed is fundamental to multi-process programming.

1.2.1 The `fork()` System Call

The `fork()` system call is used to create a new process, known as the child process, which is a near-identical copy of the calling process (parent process).

- **Basic `fork()` Operation:**
  - Creates a child process under the parent process.
  - The child process begins execution immediately after the `fork()` call.
  - If there are n `fork()` calls, 2^n processes will be created.
  - Both parent and child processes are independent, each with its own Process ID (PID), and can exist concurrently.
- **Return Values of `fork()`:**
  - To the parent process, `fork()` returns the PID of the child.
  - To the child process, `fork()` returns 0.
  - If `fork()` fails (e.g., due to memory problems), it returns -1.
- **`getpid()` and `getppid()`:**
  - `getpid()` returns the PID of the process from which it is called.
  - `getppid()` returns the PID of the parent of the process from which it is called.
- **Process Memory and Variables with `fork()`:**
  - `fork()` creates two identical processes, but they are completely independent.
  - Both processes share the same variables, but each has its own *copy* of these variables. Modifications to variables in one process do not affect the other.

1.2.2 Orphans and Zombies

These are special states that processes can enter due to the asynchronous nature of parent-child relationships.

- **Orphan Processes:**
  - An orphan process occurs when the parent process terminates *before* its child process.
  - The orphaned child is immediately adopted by the `init` process (process dispatcher), which typically has PID 1.
  - In Unix, `ps -l` might show 'O' in the second column for an orphan process, though Linux generally does not identify orphan processes this way.
- **Zombie Processes:**
  - A zombie process is a process that has terminated but remains in the process table because its parent has not yet read its exit status.
  - In Unix, `ps -l` shows 'Z' in the second column for a zombie process.
  - Zombie processes consume minimal system resources (only a process table entry), but too many can be problematic. The parent process must call `wait()` or `waitpid()` to clear the zombie.

1.2.3 The `exec()` System Call Family

The `exec()` family of system calls replaces the current process image with a new process image specified by a program file.

- **Purpose:** `exec()` loads a new program into the current process's memory space and begins its execution.
- **Key Difference from `fork()`:** Unlike `fork()`, `exec()` does *not* create a new process; it transforms the existing one. The PID of the process remains the same.
- **`execl()` and `execv()`:**
  - `execl()` takes the path to the new program, followed by a list of arguments (including the program name itself), terminated by a `NULL` pointer.
  - `execv()` takes the path to the new program and an array of strings for arguments, where the array is terminated by a `NULL` pointer. This provides more flexibility for command-line parameters.
- **Program Flow After `exec()`:** Any code in the calling program after a successful `exec()` call will not be executed, as the current process image is entirely overwritten.

1.3 Interprocess Communication (IPC) - Signals

Signals are a basic form of interprocess communication, often used for event notification.

1.3.1 Introduction to Signals

- Processes need to communicate with each other.
- In Unix, processes can send and receive signals to communicate.
- Signals are asynchronous notifications sent to a process to indicate that an event has occurred.

- Signals can be generated by the kernel (e.g., when a key is pressed, or an illegal operation occurs) or by other user processes.

1.3.2 Signal Handling with `signal()`

The `signal()` system call allows a process to specify how it will respond to certain signals.

- `signal(SIGNAL, HANDLER)()`:
  - The first parameter is the `SIGNAL` to trap (e.g., `SIGINT` for Ctrl+C, `SIGILL` for illegal instruction). These are defined in `<signal.h>`.
  - The second argument is the `HANDLER` function to invoke when the signal is generated.
- **Default Action (`SIG_DFL`):** If no custom handler is specified, the kernel performs a default action (e.g., program termination for `SIGINT`).
- **Ignoring Signals (`SIG_IGN`):** A process can be instructed to ignore a specific signal by passing `SIG_IGN` as the handler.
- **Re-registering Handlers:** After a signal handler is executed, it is often cleared from memory. To ensure subsequent signal occurrences are still handled, the `signal()` call must be re-registered within the handler function itself (recursive call).
- **`SIGINT` vs. DEL key:** On Linux, `SIGINT` is typically generated by `Ctrl-C`, not the `DEL` key.
- **Common Signals for Error Handling:** `SIGILL` (illegal instruction), `SIGFPE` (floating-point exception).

1.3.3 Process Termination and `SIGCLD`

- When a process terminates, it sends a `SIGCLD` signal to its parent.
- If the parent is the shell, upon receiving `SIGCLD`, the shell deletes the corresponding process entry from the process table.
- Programs can trap `SIGCLD` to perform actions when a child process terminates.

1.3.4 Sending Signals Between Processes with `kill()`

- Signals can be sent between two user processes using the `kill()` system call.
- `kill(PID, SIGNAL)()`:
  - The first argument is the Process ID (`PID`) of the target process.
  - The second argument is the `SIGNAL` to send.
- **User-Defined Signals:** `SIGUSR1` and `SIGUSR2` are two user-programmable signals not mapped to any specific keys, making them ideal for custom inter-process communication.

1.4 Interprocess Communication (IPC) - Message Queues

Message queues allow processes to exchange messages (like text conversations).

1.4.1 Introduction to Message Queues

- Unlike signals (which have predefined values and are event-driven), messages are user-defined and are exchanged for conversation between processes.
- A message queue acts as a mailbox where messages are queued for delivery until retrieved by the receiving process.

1.4.2 Creating and Managing Message Queues with `msgget()` and `msgctl()`

- **`msgget()` (Create/Access):**
  - Creates a new message queue or gets the ID of an existing one.
  - Takes two arguments: a `key_t` value (the queue's name/key) and a `flag`.
  - **Flags:**
    * `IPC_CREAT`: Creates the queue if it doesn't exist. Ignored if it does.
    * `IPC_EXCL`: Used with `IPC_CREAT` to force an error if the queue already exists (exclusive mode).
    * Permissions (e.g., `0644` for rw-r–r–): Specified in `ugo` notation and ORed with other flags.
  - Returns the message queue ID (`msqid`) on success, or -1 on failure.
  - **Permission Note:** Once a queue is created with specific permissions, it should be accessed with the same or a subset of those permissions. ORing `IPC_CREAT` with `0` can avoid errors when accessing an existing queue with unknown permissions.
- **`ipcs -q` (Inspect Queues):** Command-line tool to view existing message queues and their properties.
- **`msgctl()` (Control/Delete):**
  - Used to perform operations on a message queue, including deletion.
  - `msgctl(msqid, IPC_RMID, 0)`: Deletes the message queue identified by `msqid`. The last argument must be `0` for deletion.

1.4.3 Sending Messages with `msgsnd()`

The `msgsnd()` system call sends a message to a message queue.

- **`msgsnd(msqid, &message_struct, message_length, flags)():`**
  - `msqid`: The message queue identifier.
  - `&message_struct`: Starting address of the message data structure. This structure typically includes a `long mtype` (message priority) and `char mtext[]` (the message content).
  - `message_length`: Length of the actual message string (`mtext`).
  - `flags`:
    * `0`: Process waits if the queue is full until space becomes available.
    * `IPC_NOWAIT`: Process terminates immediately if the queue is full.

1.4.4 Receiving Messages with `msgrcv()`

The `msgrcv()` system call receives a message from a message queue.

- **`msgrcv(msqid, &buffer_struct, max_message_length, message_type, flags)():`**

- `msqid`: The message queue identifier.
- `&buffer_struct`: Address of the buffer structure to store the received message.
- `max_message_length`: The maximum length of the message to receive. This should be equal to or greater than the actual message length to avoid errors.
- `message_type`: The priority of the message to receive. Only messages with a matching priority are accepted.
- `flags`:
  - `0`: Process waits if the queue is empty until a message of the desired type and permission arrives.
  - `IPC_NOWAIT`: Process does not wait if the queue is empty.
  - `MSG_NOERROR`: ORed with other flags, it prevents an error if the received message length exceeds `max_message_length`; instead, it truncates the message.

1.5 Interprocess Communication (IPC) - Semaphores

Semaphores are data structures used for process synchronization, especially when accessing shared resources.

1.5.1 Introduction to Semaphores

- Semaphores help synchronize several processes accessing a common resource.
- They can be created and managed similarly to message queues.
- Semaphores are created in sets, with a maximum of 10 sets, each containing up to 25 semaphores.
- `ipcs -s` **(Inspect Semaphores):** Command-line tool to list available semaphores and their properties.

1.5.2 Creating and Managing Semaphores with `semget()` and `semctl()`

- `semget()` **(Create/Access):**
  - Creates a semaphore set or gets the ID of an existing one.
  - `semget(key, num_semaphores, flags)():`
    - `key`: The key or name of the semaphore set. All semaphores in a set have the same key value.
    - `num_semaphores`: The number of semaphores to create in that set.
    - `flags`:
      - `IPC_CREAT`: Creates the semaphore set if it doesn't exist.
      - `IPC_EXCL`: Used with `IPC_CREAT` to create the semaphore in exclusive mode, returning an error if it already exists.
      - Permissions (e.g., `0666`): ORed with other constants.
  - **Permission Note:** Any subsequent attempts to access or create the same semaphore set must use a permission mode that is the same as or a subset of the original creation permission.
  - **Number of Semaphores Note:** If a set is created with a certain number of semaphores, subsequent `semget()` calls for the same set will only succeed if they specify a number of semaphores less than or equal to the original number.
- `semctl()` **(Control/Delete/Set/Retrieve):**

  - A versatile function used for various semaphore operations.
  - **Deleting a Semaphore Set:** `semctl(semid, 0, IPC_RMID, 0)`.
    * `semid`: The ID of the semaphore set.
    * The second and fourth arguments must be `0` for deletion.
  - **Setting a Semaphore Value:** `semctl(semid, sem_index, SETVAL, value)`.
    * `semid`: The ID of the semaphore set.
    * `sem_index`: The index of the individual semaphore within the set whose value you want to set.
    * `SETVAL`: The operation to perform.
    * `value`: The integer value to set for the semaphore.
  - **Retrieving a Semaphore Value:** `semctl(semid, sem_index, GETVAL, 0)`.
    * `semid`: The ID of the semaphore set.
    * `sem_index`: The index of the individual semaphore within the set.
    * `GETVAL`: The operation to perform.
    * The last argument must be `0`.
  - **Getting PID of Last Setter:** `semctl(semid, sem_index, GETPID, 0)`.
    * Returns the PID of the process that last performed a `semop()` (semaphore operation) on the specific semaphore within the set.

1.6 Additional Necessary Topics & Future Considerations

To ensure a strong foundation for future assignments, students should also be familiar with:

- **Makefiles:** Essential for managing compilation of multiple source files in C projects.
- **Error Handling:** Proper use of `perror()` and checking return values of system calls for robust programs.
- **`wait()` and `waitpid()`:** Crucial for parent processes to wait for child processes to terminate and to reap zombie processes.
- **Process Exit Status:** Understanding how child processes communicate their exit status to parents.
- **Introduction to `gdb`:** A basic debugger is invaluable for tracing program execution and identifying issues in C programs.

1.7 Prerequisite Assignment Tasks

Students should write C programs for each of the following scenarios to solidify their understanding. Encourage them to compile and run these programs on their Linux systems and observe the behavior using commands like `ps`, `ipcs`.

1. **Fork Bomb (Cautionary Example):** Write a program that uses an infinite loop with `fork()` to demonstrate the resource consumption and potential system instability caused by uncontrolled process creation. (Emphasize stopping it with `killall` or `pkill` quickly).
2. **Orphan Process Demonstration:** Implement Example 2 from the provided text, demonstrating how a child process becomes an orphan and is adopted by `init` (PID 1). Use `getppid()` before and after the parent terminates.

3. **Zombie Process Demonstration:** Implement Example 3 from the provided text, demonstrating how a child process becomes a zombie until the parent terminates (or calls `wait()`). Use `ps -l` to observe the 'Z' status.
4. **`fork()` vs. `exec()` (PID Comparison):** Write a C program that:
   - Calls `fork()` to create a child.
   - In the parent, print its PID and the child's PID.
   - In the child, print its PID and its parent's PID.
   - Then, in the child process, use `execlp()` (a variant of `exec()` that searches PATH) to execute a simple command like `ls -l`. Observe that the child's PID remains the same after `exec()`, but its program changes.
5. **Signal Trapping (`SIGINT`):** Implement the modified Example 1 from subsection A of Chapter 2 (recursive `signal()` call) to trap `SIGINT` (Ctrl+C) and print a custom message, ensuring the program doesn't terminate on the first `Ctrl-C`.
6. **Signal Trapping (`SIGFPE`):** Implement Example 2 from subsection A of Chapter 2 to trap `SIGFPE` (floating-point exception, e.g., division by zero) and call a custom error handler.
7. **Parent-Child Signal Communication (`SIGUSR1`):** Write a program where a parent forks a child. The child goes to sleep for a short period. The parent sends `SIGUSR1` to the child. The child wakes up upon receiving the signal and prints a message indicating it received the signal, then exits. The parent waits for the child.
8. **Message Queue Communication (Sender & Receiver):** Implement the "Assignment" task on Page 18:
   - **Sender Program:** Accepts a string input from the console, sends it as a message to a message queue.
   - **Receiver Program:** Receives the message, displays it, then sends an acknowledgment message back to the sender.
   - **Sender Program (continued):** Upon receiving the acknowledgment, displays "Acknowledgment received from receiver" and then waits for the next user input.
   - *Hint:* This will require two message queues or careful use of message types to differentiate between messages and acknowledgments.
9. **Semaphore Creation and Value Management:**
   - Write a program that creates a semaphore set with a specific key and a single semaphore within that set.
   - Set the semaphore's initial value.
   - Retrieve and print the semaphore's value.
   - Retrieve and print the PID of the process that last set the semaphore's value using `GETPID`.

Advanced Topics to Explore (Optional)

For students who grasp these concepts quickly or wish to delve deeper, these topics offer additional challenges and preparation for future labs.

1.7.1 Pipes (Unnamed and Named)

- **Unnamed Pipes (`pipe()`):** Used for one-way communication between related processes (typically parent-child). Explore how to set up a pipe and use `read()` and `write()`.
- **Named Pipes (FIFOs - `mkfifo()`):** Allow communication between unrelated processes. Understand their creation and usage.

1.7.2 Shared Memory (System V IPC)

- **Introduction:** A faster IPC mechanism where processes map a region of memory into their address space, allowing direct read/write access.
- **System Calls:** `shmget()` (create/access), `shmat()` (attach), `shmdt()` (detach), `shmctl()` (control/delete).
- **Synchronization:** Emphasize the need for synchronization mechanisms (like semaphores) when using shared memory to prevent race conditions.

1.7.3 Threads (pthreads)

- **Introduction:** Understand the difference between processes and threads (shared memory space, lighter weight).
- **Basic pthreads:** `pthread_create()`, `pthread_join()`, `pthread_exit()`.
- **Thread Synchronization:** Brief introduction to mutexes (`pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`) and condition variables.

1.7.4 Kernel Modules (Basic)

- **Introduction:** Understanding what kernel modules are and why they are used (extending kernel functionality without recompiling).
- **Basic Module Structure:** `module_init`, `module_exit`.
- **Simple "Hello World" Module:** A basic module that prints a message to the kernel log (`dmesg`) when loaded and unloaded.