Operating Systems: Process - Thread

1 Introduction

In this lab, we will explore the multi-process and multi-threaded programs. We also understand various APIs used to create, synchronize and join these multi-programming mechanisms.

2 Background

In Linux, creating multiple processes allows a program to perform multiple tasks simultaneously, improving performance and responsiveness. Processes in Linux are independent units of execution that have their own memory space and system resources.

Important APIs

- Forking: The fork() system call is the primary method to create a new process. It creates a child process that is an exact copy of the parent process.
- Exec: The exec() family of functions replaces the current process image with a new program, allowing the execution of a different program.
- Process Control: The wait() system call allows a parent process to wait for a child process to finish, and kill() can terminate a process.
- Inter-Process Communication (IPC): Processes can communicate using mechanisms like pipes, message queues, shared memory, and signals.

In Linux, multithreading allows a single process to create multiple threads that run concurrently, sharing the same memory space and system resources. Threads are lightweight and more efficient than processes because they share code, data, and file descriptors, reducing the overhead of context switching.

Important APIs

- pthread library: The POSIX thread (pthread) library provides functions to create and manage threads in Linux.

- Thread Creation: The pthread_create() function creates a new thread within a process.
- Synchronization: Threads can be synchronized using mutexes, condition variables, and semaphores to prevent race conditions and ensure proper data access.
- Thread Termination: pthread_exit() allows a thread to exit, and pthread_join() lets one thread wait for another to finish.
- pthread_create() creates two threads.
- pthread_join() blocks the calling thread until the specified thread terminates, ensuring proper cleanup.

3 Programs

- 100.c
  Summary

  This program creates `NUM_THREADS` (32 in this case) threads, each running an infinite loop (`busy_loop` function), which keeps the CPU continuously busy. The main thread creates the worker threads and then waits for them to finish (though they never will due to the infinite loop). The program demonstrates how to create multiple threads that consume CPU resources without doing any useful work.

  Conclusion

  - The program creates threads that keep the CPU busy, without performing meaningful tasks.
  - This demonstrates a high CPU load scenario, but does not contribute to productive computation.

- bad_vfork.c
  Summary

  This program demonstrates the use of `vfork()`, which creates a child process that shares the parent's memory space. A variable `x` is initialized to `42` in the parent before calling `vfork()`. The child modifies `x` to `99`, which directly affects the parent's memory. The child process returns, but because `vfork()` shares the stack, the parent's stack might be corrupted. The parent prints `x`, which may show unexpected behavior due to memory modification by the child.

  Conclusion

  - Using `vfork()` can lead to **undefined behavior** if the child modifies variables, as both processes share memory.

- To avoid corruption, the child should use `_exit()` instead of `return` to terminate cleanly.

- benchmark.c
  Summary

  This program benchmarks and compares the performance of various process and thread creation mechanisms: `pthread_create()`, `fork()`, `vfork()`, and `clone()`.
  Each benchmark function creates `NUM_PROCESSES` threads or processes and measures the time taken for their creation and execution using `clock_gettime()` to record start and end times.
  The `main()` function prints the time taken by each method to create the processes or threads.

  Conclusion

  - `pthread_create()` is used to benchmark thread creation and join time.
  - `fork()` and `vfork()` are compared in terms of process creation time, with `vfork()` generally being more efficient in some use cases due to not duplicating the parent's memory.
  - `clone()` provides more control over process creation, but it requires manual management of memory (e.g., stack allocation).
  - The program demonstrates differences in performance between various process and thread creation methods.

- benchmark_clone_flags.c
  Summary

  This program benchmarks the performance of the `clone()` system call with different flag configurations. It measures the time taken to create and wait for 1000 child processes while varying levels of resource sharing. The `benchmark_clone()` function creates child processes using `clone()` and records execution time using `clock_gettime()`. Different `clone()` flags are used to test the effects of sharing memory, file descriptors, filesystem information, and other resources between processes. The results provide insights into how resource sharing impacts process creation overhead.

  Conclusion

  - The `clone()` system call allows fine-grained control over process creation and resource sharing.
  - The benchmark highlights how sharing memory and resources affects performance, with increased sharing typically reducing overhead.

- clone.c
  Summary

This program demonstrates the creation of a **lightweight process (LWP)** using the `clone()` system call. The parent process allocates a separate stack for the child thread and uses `clone()` to create a new LWP that executes the function `thread_function()`. The LWP prints a message and then terminates. The parent waits for the child to finish using `waitpid()`, ensuring proper cleanup. The allocated stack is freed after execution.

Conclusion

- The `clone()` system call provides a way to create LWPs, similar to threads, with separate stack memory.
- This method is useful in applications requiring fine-grained control over thread creation and execution.

• clone_full.c
Summary

This program demonstrates process creation using `clone()` with **Thread-Local Storage (TLS)**. A structure `tls_data` is used to store a range of numbers and the computed sum. The parent process initializes the TLS structure and creates a child process using `clone()` with flags for sharing memory and setting TLS. The child computes the sum from `start` to `end`, stores the result in TLS, and prints its PID. The parent reads the child's result from the TLS structure and prints it.

Conclusion

- The use of `CLONE_SETTLS` allows each thread to have its own Thread-Local Storage, useful for isolated computation.
- This approach demonstrates how `clone()` can be used for fine-grained process control while safely sharing TLS structures.

• cow.c
Summary

This program benchmarks the performance of **Copy-on-Write (COW)** memory in a `fork()` process. It first allocates and initializes a large 100 MB array in the parent process. After initialization, the parent performs two write passes over the array to measure memory access times. Then, the process forks, and the child performs two similar write operations, demonstrating the impact of COW.

Conclusion

- The first write in the child process is slower due to **Copy-on-Write (COW)**, which allocates new pages.
- The second write is faster, as the memory has already been copied, showing how COW optimizes memory efficiency in 'fork()' operations.

- drop_priv.c
  Summary

  This program demonstrates how a **lightweight process (LWP)** can drop root privileges using `setuid()` and `setgid()`. The parent process creates a new LWP using `clone()`, which initially runs with root privileges. Inside the LWP, the function `thread_function()` attempts to lower its privileges to a non-root user (UID 1000). If successful, the child prints its new UID and GID before exiting. The parent process waits for the child to complete execution.

  Conclusion

  - Dropping privileges in child processes enhances security by limiting the scope of potential exploits.
  - The use of `clone()` allows fine-grained control over process privileges, ensuring minimal privilege execution.

- execl.c
  Summary

  This program demonstrates the use of `execl()` to replace the current process image with a new program. The `execl()` function executes the `ls` command with the `-l` flag to list the contents of the `/home` directory in a detailed format. If the call to `execl()` succeeds, the current process is completely replaced by `ls`, and execution never returns to the original program. If it fails, an error message is printed.

  Conclusion

  - The `execl()` function replaces the calling process with a new program, making further execution of the original code impossible.
  - If `execl()` fails, error handling is necessary since the function does not return on success.

- execve.c
  Summary

  This program demonstrates the use of `execve()` to execute an external program (`./printer`) while passing command-line arguments and environment variables. The argument list (`argv`) includes the program name and two additional arguments (`"arg1"` and `"arg2"`). The environment variables (`envp`) define key-value pairs such as `VAR1=value1` and `VAR2=value2`. If `execve()` is successful, the calling process is entirely replaced by the new program; otherwise, an error message is printed.

  Conclusion

  - The `execve()` function enables fine-grained control over process execution by specifying both arguments and environment variables.

– Since `execve()` replaces the calling process, proper error handling is necessary to detect and respond to failures.

- execvpe.c
  Summary

  This program demonstrates the use of `execvpe()` to execute an external command while specifying custom arguments and environment variables. The argument list (`argv`) includes the program name (`ls`) and its options (`-l /home`). The environment list (`envp`) includes a custom variable (`MY_CUSTOM_ENV`) and a modified `PATH`. If `execvpe()` succeeds, the calling process is entirely replaced by `ls`; otherwise, an error message is printed.

  Conclusion

  – The `execvpe()` function allows setting a custom environment while executing a new program, making it useful for controlled execution.

  – Since `execvpe()` replaces the calling process, error handling is necessary to handle execution failures.

- fork_bomb.c
  Summary

  This program is an example of a **fork bomb**, which continuously creates new processes by calling `fork()` inside an infinite loop. Each new process spawns additional child processes, exponentially increasing the number of processes running on the system. This can quickly exhaust system resources, leading to a system crash or unresponsiveness.

  Conclusion

  – This program should never be executed, as it can cause a denial-of-service (DoS) attack by overwhelming system resources.

- fork.c
  Summary

  This program demonstrates process creation using the `fork()` system call. The parent process calls `fork()`, creating a new child process. If `fork()` succeeds, both the parent and child execute their respective code blocks. The child prints its own process ID (`PID`), while the parent prints the child's PID. If `fork()` fails, an error message is displayed.

  Conclusion

  – The `fork()` system call creates a new child process that runs independently from the parent.

  – Both parent and child execute separately, demonstrating parallel execution in process management.

- fork_files.c
  Summary

  This program demonstrates how file descriptors are shared between a parent and child process when using `fork()`. The parent process opens a file `shared_file.txt` before forking. Both the parent and child write to the same file descriptor. The child writes a message first, then exits. The parent waits for the child to finish before writing its own message to the file. Since file descriptors are inherited, both processes modify the same file.

  Conclusion

  - File descriptors are shared across `fork()`, meaning both parent and child can modify the same file.
  - Proper synchronization, such as waiting for the child using `wait()`, ensures orderly file access and prevents data corruption.

- fork_memory.c
  Summary

  This program demonstrates how memory is handled when a process forks. A global variable `variable` is initialized to 42. Before forking, the program prints its address and value. After the fork, both parent and child processes have separate copies of the variable. The child modifies `variable` to 99, but this change does not affect the parent's copy.

  Conclusion

  - Each process gets a separate copy of memory, so modifications in the child process do not affect the parent.
  - The addresses of variables appear the same, but they reside in separate memory spaces due to **Copy-on-Write (COW)** behavior.

- fork_stack_heap.c
  Summary

  A stack variable (`stack_var`) and a dynamically allocated heap variable (`heap_var`) are initialized before forking. The child process modifies both, but due to **Copy-on-Write (COW)**, the stack variable remains independent between parent and child. However, since the heap is shared, the parent sees an updated value of `heap_var` after the child exits.

  Conclusion

  - Stack variables are **copied** during `fork()`, so modifications in the child do not affect the parent.
  - Heap variables remain **shared** unless explicitly managed, allowing inter-process communication through dynamically allocated memory.

- main.c

- patient_zero.c
  Summary

  This program continuously creates child processes using `fork()` inside an infinite loop. Each child prints its process ID (PID) and parent PID before immediately exiting using `_exit(0)` to avoid a fork bomb. The parent prints a message confirming the creation of the child and then sleeps for 1 second before forking again.

  Conclusion

  - Unlike a fork bomb, it regulates process creation by ensuring each child exits immediately and introducing a delay using `sleep(1)`.

- printer.c
  Summary

  This program shows how to access and display **command-line arguments** and **environment variables** in a C program. The function parameters `argc`, `argv[]`, and `envp[]` allow retrieval of input arguments and environment settings.

- pthread.c
  Summary

  This program creates a thread using `pthread_create` and passes an integer (42) as an argument. The thread function correctly casts the `void*` argument to `int*`, but the second assignment `int num2 = arg` is incorrect and causes a warning due to a type mismatch. The main thread waits for the new thread to finish using `pthread_join` before exiting.

  Conclusion

  - Proper casting of `void*` to the expected type is essential to avoid type mismatches.
  - The program demonstrates the correct use of `pthread_create` and `pthread_join`, with a small mistake that leads to a warning.

- pthread_files.c
  Summary

  This program creates a child thread that writes to a shared file, while the parent thread also writes after the child finishes. The issue is that the child thread closes the file descriptor, which causes the parent to attempt writing to a closed file, potentially leading to an error.

  Conclusion

  - The shared file descriptor should not be closed by one thread before other threads are finished using it.
  - Proper thread synchronization is necessary to ensure resources (like file descriptors) remain available to all threads when needed.

- pthread_memory.c
  Summary

  This program demonstrates how multiple threads can access and modify a shared variable. The main thread creates a child thread that modifies the shared variable, and both threads print the variable's address and value. Since the main thread waits for the child thread to finish before accessing the shared variable, they are not accessing it simultaneously, so there is no race condition. The main thread prints the updated value after the child thread completes.

  Conclusion

  - Waiting for threads to finish using `pthread_join` prevents race conditions in shared resource access.
  - The program demonstrates correct synchronization, ensuring safe access to shared variables.

- pthread_stack_heap.c
  Summary

  This program demonstrates the difference between stack and heap memory across threads. The stack variable (`stack_var`) is local to each thread, meaning each thread has its own copy, and modifications in one thread don't affect the other. The heap variable (`heap_var`) is shared between the threads, meaning modifications made by one thread are visible to the other. The main thread and the child thread both access and modify the shared heap variable, while each thread operates on its own local stack variable.

  Conclusion

  - Stack variables are local to each thread and do not interfere with others.
  - Heap variables are shared across threads, so modifications by one thread are visible to others.

- real_fork.c
  Summary

  This program creates a process using a direct system call to `fork()` via `syscall(SYS_fork)` instead of the standard `fork()` wrapper. Before forking, the program prints a message. After the fork, the child and parent processes print their respective process IDs (PIDs) and parent-child relationships. Both processes execute the final print statement, showcasing parallel execution.

  Conclusion

  - The child and parent execute independently after forking.

- vfork.c
  Summary

  This program creates a child process that shares the parent's memory space until it calls `exec()` or exits. The child attempts to replace its execution image using `execlp()` to run the `ls -l` command otherwise it exits. The parent after forking, optionally waits for the child to complete.

  Conclusion

  - Unlike `fork()`, `vfork()` allows the child process to execute in the parent's memory space.

- wait.c
  Summary

  This program demonstrates the use of `fork()` to create a child process and `waitpid()` to ensure the parent process waits for the child's completion. The child process executes the `ls` command using `execlp()`, replacing itself with the command. If `execlp()` fails, an error message is printed.

  Conclusion

  - Using `waitpid()` allows the parent to wait for and retrieve the exit status of a specific child process, ensuring proper synchronization.
  - The exit status check helps determine whether the child process completed successfully or encountered an error.

4 Takeaway

In this way, multi-programming mechanisms help the user write programs that can take advantage of the concurrency offered by the hardware to achieve better performance. Students can look at the bonus programs to understand advanced topics.