Lab 3: Process & Thread Management in Linux

1. Process Creation & Termination

Process Creation (`fork()`, `exec()`)

- **Purpose:** Creates a new process (child) from an existing one (parent).
- **Key Concepts:**
  - `fork()`: Creates a child process (duplicate of parent).
  - `exec()`: Replaces the current process with a new program.

Sample C Code (`process_demo.c`)

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

int main() {
    printf("Parent starting (PID: %d)\n", getpid());

    pid_t pid = fork();  // Create child process

    if (pid == 0) {
        // Child process execution
        printf("Child running (PID: %d, Parent PID: %d)\n",
               getpid(), getppid());

        // Replace child process with ls command
        execlp("/bin/ls", "ls", "-l", NULL);

        // Only reached if execlp fails
        perror("execlp failed");
        _exit(1);  // Use _exit() in child after exec fails
    }
    else if (pid > 0) {
        // Parent process execution
        printf("Parent waiting for child (PID: %d)\n", pid);

        int status;
        waitpid(pid, &status, 0);  // Wait for specific child

        if (WIFEXITED(status)) {
            printf("Child exited with status %d\n", WEXITSTATUS(status));
        }
    }
    else {
        perror("fork failed");
        return 1;
    }

    printf("Parent process ending\n");
```

```
    return 0;
}
```

Compile & Run:

```
$ gcc process_demo.c -o process_demo
$ ./process_demo
```

Output Explanation:

1. Parent prints its PID (e.g., 1234)
2. fork() creates identical child process
3. Child prints its PID (1235) and parent's PID (1234)
4. Child executes `ls -l` command (replaces itself)
5. Parent waits specifically for this child
6. After child completes, parent reports exit status
7. Parent prints ending message

Expected Output:

```
Parent starting (PID: 1234)
Child running (PID: 1235, Parent PID: 1234)
total 24
-rw-r--r-- 1 user user  220 Jun 25 09:30 file.txt
drwxr-xr-x 2 user user 4096 Jun 25 10:00 Documents
Child exited with status 0
Parent process ending
```

Process Termination (`exit()`, `kill`)

- **Purpose:** Ends a process.
- **Methods:**
  - `exit(0)`: Normal termination (flushes buffers, runs atexit handlers)
  - `_exit(0)`: Immediate termination (no cleanup)
  - `kill -9 PID`: Forceful termination from command line

Example:

```
# Find process ID
$ ps aux | grep "process_demo"

# Graceful termination (SIGTERM)
$ kill 1235

# Forceful termination (SIGKILL)
$ kill -9 1235
```

Output:

```
[1]   Terminated              ./process_demo  # For SIGTERM
[1]   Killed                  ./process_demo  # For SIGKILL
```

---

2. Thread Creation & Termination

Thread Creation (`pthread_create()`)

- **Purpose:** Creates lightweight execution threads within a process.
- **Key Concepts:**
  - Threads share global variables and heap memory
  - Each thread has its own stack and registers

Sample C Code (`thread_demo.c`)

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 3

// Thread function
void *print_message(void *thread_id) {
    long tid = (long)thread_id;
    printf("Thread %ld started (TID: %lu)\n",
          tid, (unsigned long)pthread_self());

    // Simulate work
    sleep(1 + tid);

    printf("Thread %ld completing\n", tid);
    pthread_exit((void *)(tid * 100));
}

int main() {
    pthread_t threads[NUM_THREADS];
    int rc;

    for (long t = 0; t < NUM_THREADS; t++) {
        printf("Creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL,
                            print_message, (void *)t);
        if (rc) {
            printf("Error creating thread %ld\n", t);
            return -1;
        }
    }

    // Wait for threads to complete
    void *status;
    for (long t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], &status);
        printf("Thread %ld returned %ld\n", t, (long)status);
    }

    printf("All threads completed\n");
    pthread_exit(NULL);
}
```

Compile & Run:

```
$ gcc thread_demo.c -o thread_demo -lpthread
$ ./thread_demo
```

Output Explanation:

1. Main thread creates 3 worker threads
2. Each thread:
    - Prints its ID and system TID
    - Sleeps for 1-3 seconds (simulating work)
    - Returns a computed value (tid * 100)
3. Main thread waits for each thread using pthread_join()
4. Return values from threads are printed
5. Main thread exits after all threads complete

Expected Output:

```
Creating thread 0
Creating thread 1
Creating thread 2
Thread 0 started (TID: 139716318422784)
Thread 1 started (TID: 139716310030080)
Thread 2 started (TID: 139716301637376)
Thread 0 completing
Thread 0 returned 0
Thread 1 completing
Thread 1 returned 100
Thread 2 completing
Thread 2 returned 200
All threads completed
```

Thread Termination (`pthread_exit()`, `pthread_cancel()`)

- **Purpose:** Controlled thread termination
- **Methods:**
    - `pthread_exit()`: Clean thread exit (returns value to joiner)
    - `pthread_cancel()`: Forces thread cancellation

Example:

```
// In main(), before pthread_join()
pthread_cancel(threads[1]);  // Cancel second thread
```

Output Effect:

```
Thread 1 might not print completion message
Thread 1 returned -1  // PTHREAD_CANCELED
```

---

3. Comparison: Process vs Thread

Feature	Process	Thread

Creation	`fork()` (expensive)	`pthread_create()` (cheap)
Memory	Separate address space	Shared address space
Comm	IPC (pipes, shared mem)	Direct memory access
Sync	Not needed	Mutexes, condition vars
Fault	One crashes, others survive	Thread crash kills all
Use Case	Isolation needed	Parallel tasks, performance

---

4. Exam-Oriented Questions

Process Management

1. **What happens to open files during fork()?**
    - Child gets copies of file descriptors pointing to same files
    - Both processes share file offsets unless O_CLOEXEC is set
2. **Why use waitpid() instead of wait()?**
    - waitpid() can target specific children and provides more options

Thread Management

3. **What is a thread's stack size?**
    - Default ~8MB on Linux (adjustable via pthread_attr_setstacksize())
4. **How to pass data safely to threads?**
    - Either pass by value or allocate dynamic memory
    - Never pass stack variables that may go out of scope

---

5. Key Commands for Process/Thread Control

Command	Purpose	Example
`ps -Lf`	Show threads	`ps -Lf <PID>`
`pstree -p`	Process tree	`pstree -p 1234`
`gdb attach`	Debug process	`gdb -p 1234`
`strace -f`	Trace process+threads	`strace -f ./program`

---

6. Summary

- **Processes**: Heavyweight isolation, fork/exec model, independent
- **Threads**: Lightweight sharing, pthreads, require synchronization
- **Key Tools**: ps, top, gdb, strace for debugging