Enhancements to xv6: Adding and Modifying System Calls

Contributors:
CS22B1030 K Manvitha
CS22B1078 P Keerthi
CS22B1081 K Neha
CS22B2030 V Manaswini

Contents

1. Abstract

This project involves the addition and modification of system calls in the `xv6` operating system. It aims to enhance functionality by introducing features such as process creation with direct program execution, process management, thread-level sleep, and signal-based inter-process communication. These additions demonstrate fundamental OS concepts, including scheduling, signals, and process control.

2. Introduction

2.1. Background

The `xv6` operating system is a simple Unix-like teaching operating system used to understand core operating system concepts. While it provides basic functionality, it lacks many advanced features such as signal handling, advanced process management, and thread functionalities.

2.2. Problem Statement

The original `xv6` system lacks support for critical process and thread management features such as:

- Process creation with direct execution of a program.
- Thread-level sleep and synchronization mechanisms.
- Signal-based inter-process communication.

2.3. Objective

The project enhances the `xv6` operating system by implementing six system calls: `forex()`, `getppid()`, `usleep()`, `waitpid()`, `sigstop()`, and `sigcont()`. These system calls aim to address the limitations of the original system.

3. System Call Descriptions

3.1. `forex()`

Type: Modification of `fork()`
Purpose: Combines `fork()` and `exec()` functionality, allowing direct execution of a program in a new process.
Prototype:

```
int forex(const char *filename);
```

Behavior:

- If `filename` is provided, creates a new process and executes the specified program.
- If `filename` is `NULL`, behaves like the traditional `fork()`.

Example Usage:

```
if (forex("program_name") == 0) {
    // Child process
} else {
    // Parent process
}
```

3.2. getppid()

Type: New System Call
Purpose: Returns the parent process ID of the calling process.
Prototype:

```
int getppid();
```

Example Usage:

```
int ppid = getppid();
printf("Parent PID: %d\n", ppid);
```

3.3. usleep()

Type: New System Call
Purpose: Suspends the calling thread for a specified number of microseconds.
Prototype:

```
int usleep(unsigned int usec);
```

Example Usage:

```
usleep(1000000);
```

3.4. waitpid()

Type: New System Call
Purpose: Waits for a specific child process to terminate.
Prototype:

```
int waitpid(int pid, int *status);
```

Example Usage:

```
waitpid(pid, &status);
```

3.5. sigstop()

Type: New System Call
Purpose: Sends a signal to a process to stop its execution.
Prototype:

```
int sigstop(int pid);
```

Example Usage:

```
sigstop(pid);
```

3.6. `sigcont()`

Type: New System Call
Purpose: Resumes the execution of a stopped process.
Prototype:

```
int sigcont(int pid);
```

Example Usage:

```
sigcont(pid);
```

4. Implementation Details

File	Purpose
`syscall.h`	Add a unique identifier (system call number) for the new system call.
`syscall.c`	Add the system call to the system call table, linking it to the kernel function.
`defs.h`	Declare the kernel function for the new system call.
`sysproc.c` or `proc.c`	Implement the kernel logic for the system call.
`user.h`	Declare the user-facing function prototype for the system call.
`usys.pl`	Add an entry to generate assembly stubs for the system call automatically.
`Makefile`	Include any new kernel files required for the system call in the build process.
User programs	Create test programs to verify the functionality and correctness of the new system call.

Table 1: Files Modified for Adding a New System Call in `xv6`

5. User Programs

This section contains the user programs that demonstrate the usage of the added or modified system calls. Each subsection provides the relevant program's source code in the form of an image and a brief description.

5.1. `forex()`

Description: This program demonstrates the usage of the `forex()` system call, which combines the functionality of `fork()` and `exec()` by creating a new process and directly executing a specified program without duplicating the parent process.

```c
#include "../kernel/types.h"
#include "user/user.h"

int main(int argc, char *argv[]) {
    int pid;



    // Test forex() with arguments (direct exec behavior)
    printf("Testing forex() with arguments:\n");

    pid = forex("check");
    if (pid < 0) {
        printf("forex failed with exec argument\n");
        exit(1);
    }

    if (pid == 0) {  // Child process
        printf("This line will not execute if exec succeeds.\n");
        //exit(0);
    } else {
        wait(0);  // Parent process waits for the child
        printf("Parent process waited for exec child (PID: %d)\n", pid);
    }

    exit(0);
}
```

Figure 1: User Program demonstrating `forex()`.

5.2. `getppid()`

Description: This program illustrates the usage of the `getppid()` system call to retrieve the parent process ID of the calling process.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main() {

    printf("In the initial process, Parent PID: %d\n", getppid());
    int pid = fork();  // Create a child process

    if (pid < 0) {
        // Error during fork
        printf("Fork failed!\n");
        exit(1);
    }

    if (pid == 0) {
        int child_pid = fork();
        if (child_pid < 0) {
            // Error during fork
            printf("Child fork failed!\n");
            exit(1);
        }

        if (child_pid == 0) {
            // In the grandchild process
            printf("In the grandchild process (PID: %d), Parent PID: %d\n", getpid(), getppid());
            exit(0);  // Grandchild exits
        }
        wait(0);
        printf("In the child process (PID: %d), Parent PID: %d\n", getpid(), getppid());
        exit(0);  // Child exits
    }

    // Parent process
    wait(0);  // Wait for child processes to exit

    printf("In the parent process (PID: %d), Parent PID: %d\n", getpid(), getppid());

    exit(0);  // Parent exits
}
```

Figure 2: User Program demonstrating `getppid()`.

5.3. `usleep()`

Description: This program uses the `usleep()` system call to put the calling process to sleep for a specific duration, specified in microseconds.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main() {
    printf("Testing usleep...\n");

    for (int i = 0; i < 5; i++) {
        printf("Sleeping for 500 microseconds (%d)\n", i + 1);
        usleep(500); // Sleep for 500 microseconds
    }

    printf("Testing completed!\n");
    exit(0);
}
```

Figure 3: User Program demonstrating `usleep()`.

5.4. `sigstop()` and `sigcont()`

Description: This program demonstrates the usage of `sigstop()` to suspend a process and `sigcont()` to resume its execution.

```c
#include "kernel/types.h"
#include "user/user.h"
#include "kernel/syscall.h"

int main() {
    int pid = fork();

    if (pid == 0) {
        // Child process
        for (int i = 1; i <= 5; i++) {
            printf("Child running iteration %d...\n", i);
            sleep(3);  // Simulate some work
        }
        printf("Child completed its task\n");
        exit(0);  // Exit gracefully
    } else if (pid > 0) {
        // Parent process
        sleep(8);  // Let the child run for a while
        printf("Parent sending SIGSTOP to child %d\n", pid);
        sigstop(pid);  // Stop the child process

        sleep(5);  // Wait for a while before continuing the child
        printf("Parent sending SIGCONT to child %d\n", pid);
        sigcont(pid);  // Continue the child process

        wait(0);  // Wait for child process to exit
        printf("Parent: Child process %d finished\n", pid);
    } else {
        // Error in fork
        printf("Fork failed\n");
    }

    exit(0);
}
```

Figure 4: User Program demonstrating `sigstop()` and `sigcont()`.

5.5. `waitpid()`

Description: This program showcases the usage of the `waitpid()` system call, which allows a process to wait for a specific child process to terminate and retrieves its exit status.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main() {
    int pid = fork();  // Create a child process

    if (pid < 0) {
        printf("Fork failed\n");
        exit(1);
    }
    if(pid > 0)
    {
        // This is the parent process
        int status = 0;
        int ret = waitpid(pid, &status);  // Wait for the specific child process
        printf("Parent finished waiting for child PID: %d, Parent executing now------\n", pid);
        if (ret >= 0) {
            printf("Parent: Child %d exited with status %d\n", ret, status);
        } else {
            printf("Parent: waitpid failed\n");
        }
        printf("Statement after child is completed.\n");
    }
    else if (pid == 0) {
        // This is the child process
        printf("\n\nChild process (PID: %d) running\n", getpid());
        sleep(10);  // Simulate work
        exit(42);   // Exit with status 42
    }
    printf("Successful termination\n\n\n");
    exit(0);
}
```

Figure 5: User Program demonstrating `waitpid()`.

6. Testing and Results

This section demonstrates the results obtained after testing the implemented or modified system calls. Each subsection provides a description of the test scenario, followed by an output image.

6.1. `forex()`

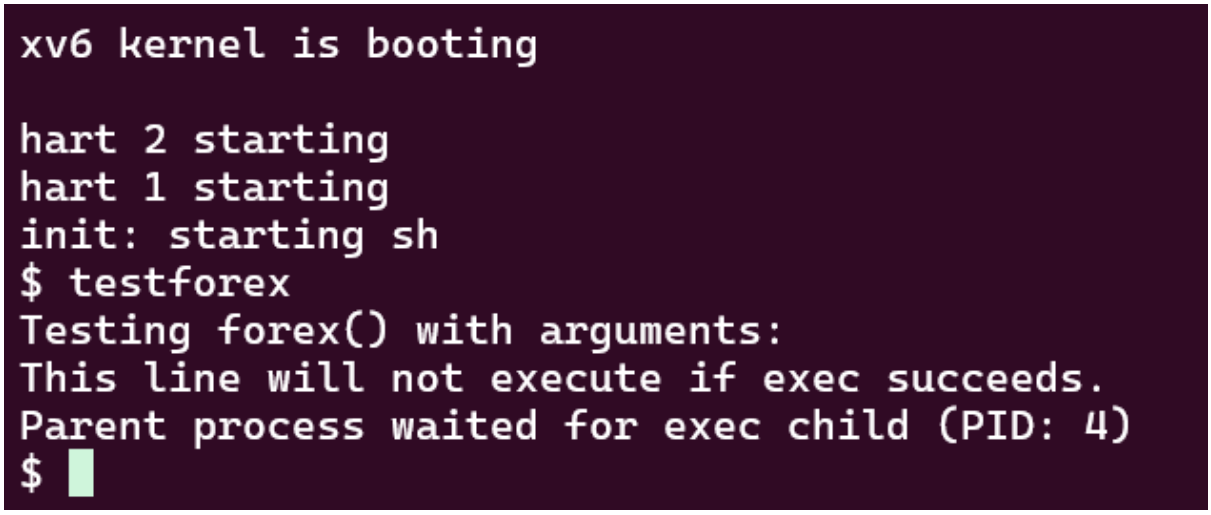

Figure 6: Output of testing the `forex()` system call.

6.2. `getppid()`

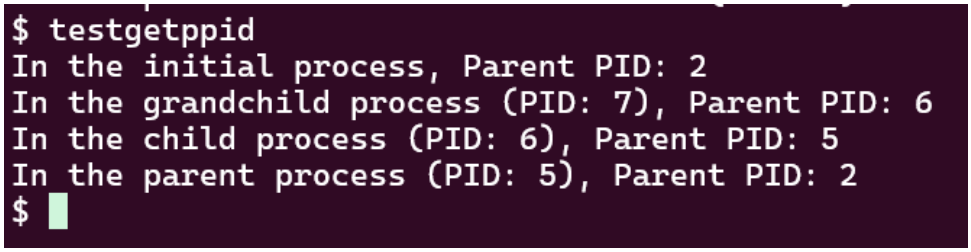

Figure 7: Output of testing the `getppid()` system call.

6.3. `usleep()`

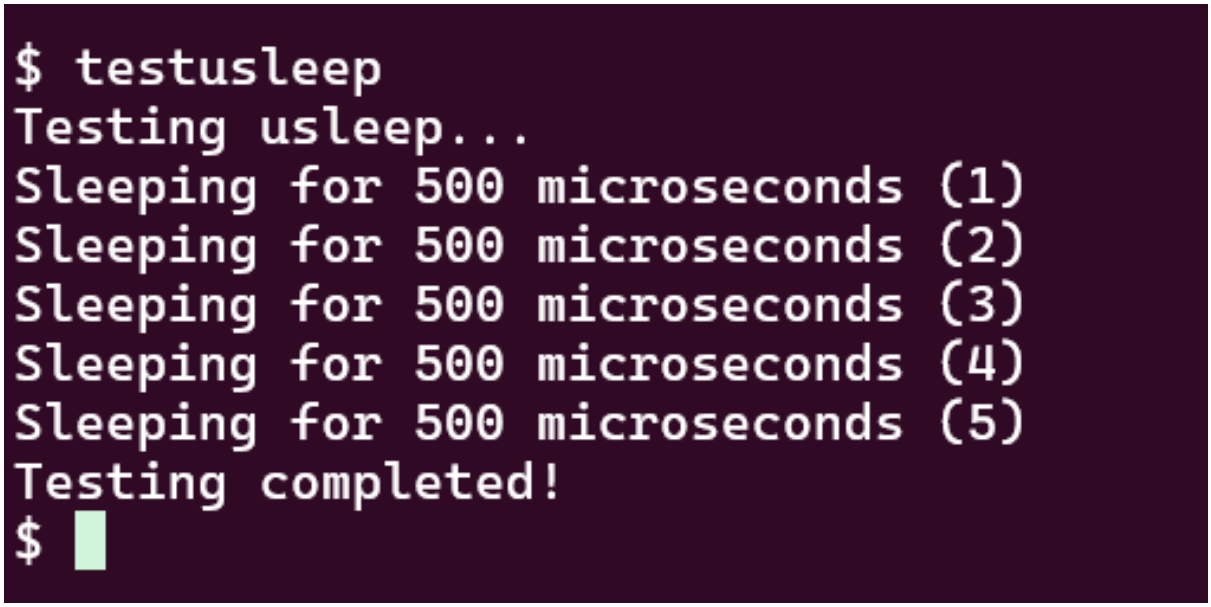

Figure 8: Output of testing the `usleep()` system call.

6.4. `sigstop()` and `sigcont()`

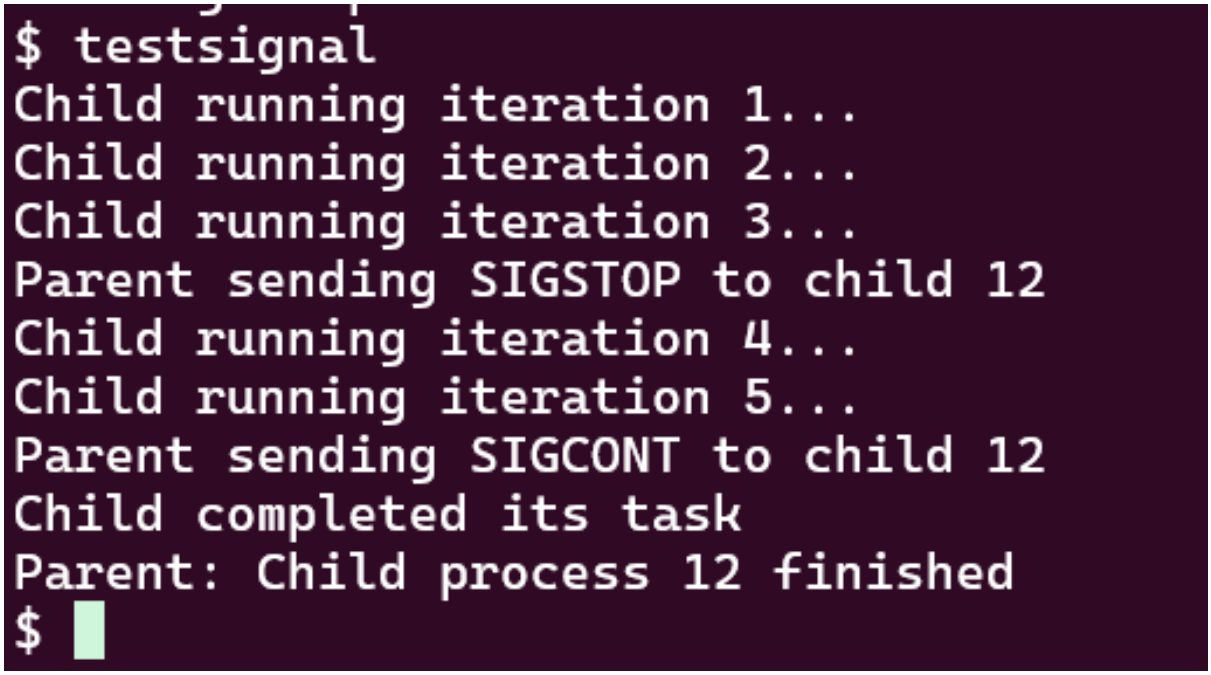

Figure 9: Output of testing the `sigstop()` and `sigcont()` system calls.

6.5. `waitpid()`

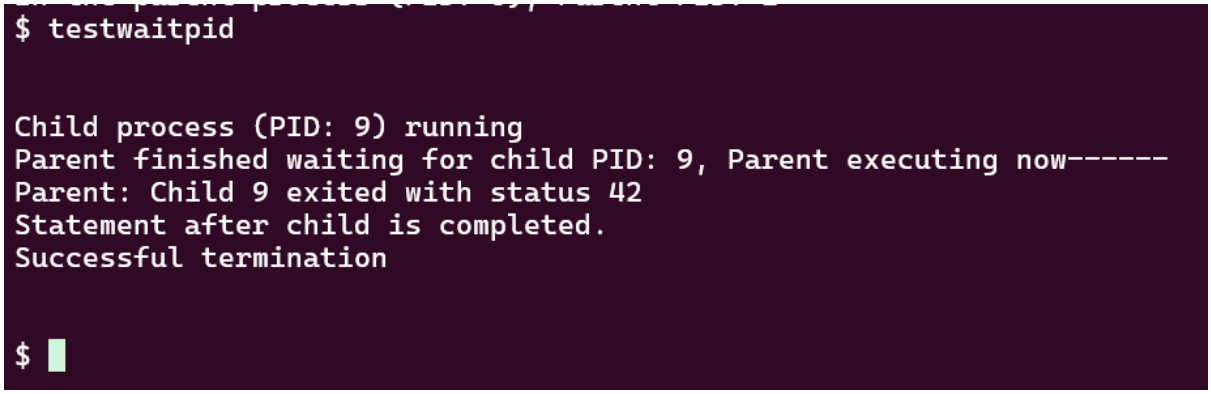

Figure 10: Output of testing the `waitpid()` system call.