WEEK 8

1. **Write a program using sigaction system call which calls a signal handler on SIGINT signal and then reset the default action of the SIGINT signal**
2. **Write a C program such that it initializes itself as a daemon Process.**
3. **Write a C program to simulate system function.**

1. Write a program using sigaction system call which calls a signal handler on SIGINT signal and then reset the default action of the SIGINT signal

```
#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>


// Signal handler function

void sigint_handler(int signum) {

  printf("Caught SIGINT (signal number %d). Now resetting to default action.\n", signum);


  // Set up the sigaction structure to reset SIGINT to default action

  struct sigaction sa;

  sa.sa_handler = SIG_DFL; // Default signal handler

  sa.sa_flags = 0;         // No flags

  sigemptyset(&sa.sa_mask); // Clear all signals from the signal set


  // Reset the signal action for SIGINT to the default action

  if (sigaction(SIGINT, &sa, NULL) == -1) {

    perror("sigaction");

    exit(EXIT_FAILURE);

  }

}
```

```
int main() {
    // Set up the sigaction structure for the custom signal handler
    struct sigaction sa;
    sa.sa_handler = sigint_handler; // Our custom signal handler
    sa.sa_flags = 0;              // No flags
    sigemptyset(&sa.sa_mask);      // Clear all signals from the signal set

    // Set the signal action for SIGINT
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    printf("Press Ctrl+C to trigger SIGINT. After the first Ctrl+C, the default action will be restored.\n");

    // Loop indefinitely to keep the program running
    while (1) {
        pause(); // Wait for signals
    }

    return 0;
}
```

Explanation:

1. **sigint_handler function**: This is the custom signal handler for SIGINT. When SIGINT is caught, it prints a message and then resets the signal action for SIGINT to its default action (**SIG_DFL**).
2. **main function**:
    - A **sigaction** structure **sa** is created and configured with the custom signal handler **sigint_handler**.
    - The signal mask (**sa.sa_mask**) is cleared, meaning no signals are blocked while the handler runs.

- The **sigaction** system call is used to set the action for SIGINT.
- The program then prints a message and enters an infinite loop, using **pause()** to wait for signals.

How it works:

- When the user presses Ctrl+C, SIGINT is sent to the program.
- The custom signal handler (**sigint_handler**) is executed, which prints a message and resets SIGINT to its default action.
- The next time the user presses Ctrl+C, the default action for SIGINT (which typically terminates the program) will be executed.

To compile and run this program:

```
gcc -o sigint_handler_example sigint_handler_example.c

./sigint_handler_example
```

After compiling and running the program, pressing Ctrl+C once will trigger the custom handler, and pressing Ctrl+C again will terminate the program with the default action.

2. Write a C program such that it initializes itself as a daemon Process.

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <syslog.h>


void create_daemon() {
```

```
pid_t pid;

// Fork the parent process
pid = fork();

// If the fork failed, exit
if (pid < 0) {
    exit(EXIT_FAILURE);
}

// If we got a good PID, let the parent terminate
if (pid > 0) {
    exit(EXIT_SUCCESS);
}

// Create a new SID for the child process
if (setsid() < 0) {
    exit(EXIT_FAILURE);
}

// Change the file mode mask
umask(0);

// Change the current working directory
if ((chdir("/")) < 0) {
    exit(EXIT_FAILURE);
}

// Close out the standard file descriptors
close(STDIN_FILENO);
close(STDOUT_FILENO);
```

```
    close(STDERR_FILENO);

    // Open new file descriptors to /dev/null
    open("/dev/null", O_RDONLY); // stdin
    open("/dev/null", O_WRONLY); // stdout
    open("/dev/null", O_WRONLY); // stderr
}

int main() {
    // Create the daemon process
    create_daemon();

    // Open a log for writing
    openlog("daemon_example", LOG_PID, LOG_DAEMON);

    // Daemon loop
    while (1) {
        // Daemon-specific code goes here
        syslog(LOG_NOTICE, "Daemon is running...");
        sleep(30); // Sleep for 30 seconds
    }

    // Close the log
    closelog();

    return EXIT_SUCCESS;
}
```

Explanation

Creating a daemon process in C involves several steps to ensure the process runs in the background, detaches from the terminal, and runs independently of any user interaction.

1. **Fork the Parent Process**: The **fork()** system call creates a new process. If **pid** is less than 0, the fork failed. If **pid** is greater than 0, we exit the parent process to ensure the child continues as the daemon.
2. **Change the File Mode Mask**: **umask(0)** sets the file mode creation mask to 0, ensuring the daemon has the permissions it needs to create files and directories.

   ensures the daemon has full access to files it creates.
3. **Open Logs**: The **openlog()** function initializes the logging system.
4. **Create a New SID**: **setsid()** creates a new session and sets the process group ID. This detaches the daemon from the terminal.
5. **Change the Working Directory**: **chdir("/")** changes the working directory to the root directory to avoid using a mounted filesystem.
6. **Close Standard File Descriptors**: The standard input, output, and error file descriptors are closed.
7. **Redirect Standard File Descriptors**: These are redirected to **/dev/null** to ensure the daemon doesn't interact with the terminal.
8. **Daemon Process Code**: The daemon process can now run its specific tasks. In this example, it logs a message every 30 seconds.

Compilation and Execution

To compile and run the daemon:

```
gcc -o daemon_example daemon_example.c

./daemon_example
```

The daemon will start, detach from the terminal, and run in the background, logging messages to the syslog every 30 seconds. You can check the syslog (usually **/var/log/syslog** or **/var/log/messages** depending on your system configuration) to see the daemon's output.

3. Write a C program to simulate system function.

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>
```

```
int my_system(const char *command) {
    if (command == NULL) {
        return -1;
    }

    pid_t pid = fork();
    if (pid < 0) {
        // Fork failed
        return -1;
    } else if (pid == 0) {
        // Child process
        execl("/bin/sh", "sh", "-c", command, (char *)NULL);
        // If execl returns, there was an error
        perror("execl");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            return -1;
        }

        if (WIFEXITED(status)) {
            return WEXITSTATUS(status);
        } else {
            return -1;
        }
    }
}

int main() {
    // Example usage of my_system function
    printf("Running 'ls -l':\n");
    int result = my_system("ls -l");
    if (result == -1) {
        perror("my_system");
    } else {
        printf("'ls -l' exited with status %d\n", result);
    }

    return 0;
}
```

Explanation:

The **system** function in C is used to execute shell commands from within a C program. To simulate the **system** function, we need to:

1. Fork a new process.
2. In the child process, replace the current process image with a new process image using **execl** or similar functions to execute the shell.
3. In the parent process, wait for the child process to complete and capture its exit status.

1. **Forking a New Process**:
    - **pid_t pid = fork();** creates a new process.
    - If **pid** is less than 0, the fork failed, and we return -1.
    - If **pid** is 0, we are in the child process.
2. **Executing the Command in the Child Process**:
    - **execl("/bin/sh", "sh", "-c", command, (char *)NULL);** runs the command using the **/bin/sh** shell.
    - If **execl** returns, it means there was an error, so we print the error message and exit with **EXIT_FAILURE**.
3. **Waiting for the Child Process in the Parent Process**:
    - **waitpid(pid, &status, 0);** waits for the child process to complete.
    - We check if the child process terminated normally using **WIFEXITED(status)**.

    - If it did, we return the exit status of the child process using **WEXITSTATUS(status)**.
    - If it did not terminate normally, we return -1.

4. **Example Usage**:
    - In the **main** function, we demonstrate how to use **my_system** by running the **ls -l** command and printing its exit status.

This implementation captures the core functionality of the **system** function, allowing you to execute shell commands from within your C program.