Assignment 6-Solution

1. Write a C program that implements a simple, signal-safe logging system to log messages when certain signals are received (e.g., SIGUSR1 and SIGUSR2). The logging function should be designed to handle signal interruptions, as functions like printf() and fprintf() are not safe to call from within a signal handler. Instead, use the low-level system call write () to safely log messages to a log file. Your program should log ”SIGUSR1 received!” and ”SIGUSR2 received!” when the corresponding signals are caught, and it should avoid deadlock or undefined behavior even when signals are received rapidly or during a critical section.

Code:

```c
#include <stdio.h>

#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <string.h>

#define LOG_FILE "signal_log.txt"

void safe_log_message(const char *message) {
    int fd = open(LOG_FILE, O_WRONLY | O_APPEND | O_CREAT, 0644);
    if (fd == -1) {
        _exit(1);
    }
    write(fd, message, strlen(message));
    close(fd);
}
void sigusr1_handler(int sig) {
    const char *msg = "SIGUSR1 received!\n";
    safe_log_message(msg);
}

void sigusr2_handler(int sig) {
    const char *msg = "SIGUSR2 received!\n";
    safe_log_message(msg);
}

int main() {
    struct sigaction sa_usr1, sa_usr2;

    sa_usr1.sa_handler = sigusr1_handler;
    sigemptyset(&sa_usr1.sa_mask);
    sa_usr1.sa_flags = SA_RESTART;
    sigaction(SIGUSR1, &sa_usr1, NULL);

    sa_usr2.sa_handler = sigusr2_handler;
    sigemptyset(&sa_usr2.sa_mask);
    sa_usr2.sa_flags = SA_RESTART;
    sigaction(SIGUSR2, &sa_usr2, NULL);
    printf("Logging system initialized. Send SIGUSR1 and SIGUSR2 to log 
messages.\n");
    printf("Process ID: %d\n", getpid());

    while (1) {
        pause();
    }
    return 0;
}
```

Input:

```
kill -SIGUSR1 <process_id>  # Log "SIGUSR1 received!"

kill -SIGUSR2 <process_id>  # Log "SIGUSR2 received!"
```

Output:

```
SIGUSR1 received!
SIGUSR2 received!
```

2. Signal-based Parallel Sorting System: Design a sorting system where multiple processes are responsible for sorting different parts of an array in parallel. The array will be divided into sub-arrays, and each process will handle sorting its assigned part. The processes will be triggered to start sorting by receiving specific signals, such as SIGUSR1 for one process and SIGUSR2 for another. After each process completes sorting its sub-array, it will notify the parent process using a signal. Once all processes have finished, the parent process will merge the sorted sub-arrays to produce the final sorted array. The system must ensure that the sorting operations happen in parallel and the results are combined correctly. Implement detailed step-by-step output to show the progress of sorting and merging operations.

Code:

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

int sub_array1[] = {15, 22, 35};
int sub_array2[] = {8, 12, 19};
int sorted1 = 0, sorted2 = 0;

void process1_sort_handler(int sig) {
    printf("Process 1 sorting sub-array [15, 22, 35]\n");

    sorted1 = 1;
    printf("Process 1 finished sorting sub-array: [15, 22, 35]\n");
    kill(getppid(), SIGUSR1);
}

void process2_sort_handler(int sig) {
    printf("Process 2 sorting sub-array [8, 12, 19]\n");

    sorted2 = 1;
    printf("Process 2 finished sorting sub-array: [8, 12, 19]\n");
    kill(getppid(), SIGUSR2);
}
void parent_merge_handler(int sig) {
    static int signals_received = 0;
    signals_received++;

    if (signals_received == 2) {
        printf("Merging sorted sub-arrays...\n");
        int merged_array[6];
        int i = 0, j = 0, k = 0;

        while (i < 3 && j < 3) {
            if (sub_array1[i] < sub_array2[j]) {
                merged_array[k++] = sub_array1[i++];
            } else {
                merged_array[k++] = sub_array2[j++];
            }
        }
        while (i < 3) merged_array[k++] = sub_array1[i++];
        while (j < 3) merged_array[k++] = sub_array2[j++];

        printf("Final sorted array: ");
        for (int i = 0; i < 6; i++) {
            printf("%d ", merged_array[i]);
        }
        printf("\n");
    }
}

int main() {
    // Set up signal handlers for parent process
    struct sigaction sa_parent;
    sa_parent.sa_handler = parent_merge_handler;
    sigemptyset(&sa_parent.sa_mask);
    sa_parent.sa_flags = 0;
    sigaction(SIGUSR1, &sa_parent, NULL);
    sigaction(SIGUSR2, &sa_parent, NULL);

    // Fork process 1
    pid_t pid1 = fork();
    if (pid1 == 0) {
        // Child process 1: Set up handler and wait for SIGUSR1
        struct sigaction sa1;
        sa1.sa_handler = process1_sort_handler;
        sigemptyset(&sa1.sa_mask);
        sa1.sa_flags = 0;
        sigaction(SIGUSR1, &sa1, NULL);
        while (!sorted1) pause();  // Wait until sorted
        exit(0);
    }

    // Fork process 2
    pid_t pid2 = fork();
    if (pid2 == 0) {
        // Child process 2: Set up handler and wait for SIGUSR2
        struct sigaction sa2;
        sa2.sa_handler = process2_sort_handler;
        sigemptyset(&sa2.sa_mask);
        sa2.sa_flags = 0;
        sigaction(SIGUSR2, &sa2, NULL);
        while (!sorted2) pause();  // Wait until sorted
        exit(0);
    }

    // Parent process waits for signals from both children
    printf("Parent process waiting for sorting processes to complete...\n");

    // Wait for both child processes to finish
    wait(NULL);
    wait(NULL);

    return 0;
}
```

Input:

```
kill -SIGUSR1 <process_id>  # Trigger first process to sort sub-array
kill -SIGUSR2 <process_id>  # Trigger second process to sort sub-array
```

Note: Should take input array from input.

Output:

```
Process 1 sorting sub-array [15, 22, 35]
Process 1 finished sorting sub-array: [15, 22, 35]
Process 2 sorting sub-array [8, 12, 19]
Process 2 finished sorting sub-array: [8, 12, 19]
Merging sorted sub-arrays...
Final sorted array: [8, 12, 15, 19, 22, 35]
```

3. Signal-Based Inter-Process Communication and Task Synchronization: In this problem, you will design a system that demonstrates inter-process communication and synchronization using signals. The system consists of two separate programs (processA.c and processB.c), where each process performs specific tasks in a coordinated manner. The processes will communicate using signals (SIGUSR1 and SIGUSR2) to synchronize their tasks. Process A will begin by generating a random number and storing it in shared memory (Task 1), after which it sends a signal to Process B to start its Task 1. Process B reads the number from shared memory, adds 10 to it, and updates the memory. Once Process B finishes, it signals Process A to begin Task 2, where it multiplies the updated number by 2 and stores the result. The final result is read and printed by Process B. You must ensure proper signal handling, synchronization, and shared memory usage to avoid race conditions and ensure orderly task execution between the processes.

Code:

Process A:

```c
#include <stdio.h>

#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>

int *shared_number;
int shmid;

void handle_sigusr2(int sig) {
    printf("Process A: Received signal from Process B to start Task 2\n");

    *shared_number *= 2;
    printf("Process A: Multiplied by 2. Result: %d\n", *shared_number);

    kill(getppid(), SIGUSR1);
    shmdt(shared_number);
}

int main() {
    signal(SIGUSR2, handle_sigusr2);

    key_t key = ftok("shmfile", 65);
    shmid = shmget(key, sizeof(int), 0666 | IPC_CREAT);
    shared_number = (int *)shmat(shmid, NULL, 0);
    srand(time(NULL));
    *shared_number = rand() % 100;
    printf("Process A: Generated random number: %d\n", *shared_number);

    kill(getppid(), SIGUSR1);

    while (1) pause();
    return 0;
}
```

Process B:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int *shared_number;
int shmid;

void handle_sigusr1(int sig) {
    printf("Process B: Received signal from Process A to start Task 1\n");

    *shared_number += 10;
    printf("Process B: Read number, added 10. Updated number: %d\n",
*shared_number);

    kill(getppid(), SIGUSR2);
}

int main() {
    signal(SIGUSR1, handle_sigusr1);

    key_t key = ftok("shmfile", 65);
    shmid = shmget(key, sizeof(int), 0666 | IPC_CREAT);
    shared_number = (int *)shmat(shmid, NULL, 0);

    while (1) pause();

    printf("Process B: Final result read: %d\n", *shared_number);

    shmdt(shared_number);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;}
```

Input:

```
 The random number is generated by Process A.
```

Output:

```
Process A: Generated random number: 7
Process B: Read number, added 10. Updated number: 17
Process A: Multiplied by 2. Result: 34
Process B: Final result read: 34
```

4. Design a signal-based file downloader that can be paused and resumed using user signals. The program simulates the download of a large file in chunks. When the SIGUSR1 signal is received, the download should pause, and it should not continue until the SIGUSR2 signal is received to resume. The download's progress should be saved during pauses so that it can continue from the last saved point when resumed. Additionally, the program must handle the SIGTERM signal for graceful termination, saving the current progress to allow resumption if the download is restarted. The output should display progress during download, the status when paused or resumed, and the final status when the download completes or the program is terminated.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <stdbool.h>

#define TOTAL_SIZE 100
#define CHUNK_SIZE 10

volatile sig_atomic_t download_paused = 0;
volatile sig_atomic_t download_progress = 0;
volatile sig_atomic_t terminate_signal = 0;

void pause_handler(int sig) {
    printf("Download paused.\n");
    download_paused = 1;
}

void resume_handler(int sig) {
    printf("Download resumed.\n");
    download_paused = 0;
}

void terminate_handler(int sig) {
    printf("\nReceived termination signal. Saving progress and exiting...\n");
    terminate_signal = 1;
}

void download_file() {
    while (download_progress < TOTAL_SIZE && !terminate_signal) {
        if (download_paused) {
        } else {
            printf("Downloading... %d%% complete\n", download_progress);
            download_progress += CHUNK_SIZE;
            if (download_progress >= TOTAL_SIZE) {
                printf("Download complete!\n");
                break;
            }
        }
    }

    if (terminate_signal) {
        printf("Download progress saved at %d%%\n", download_progress);
    }
}

int main() {
    struct sigaction sa_pause, sa_resume, sa_term;

    sa_pause.sa_handler = pause_handler;
    sigemptyset(&sa_pause.sa_mask);
    sa_pause.sa_flags = 0;
    sigaction(SIGUSR1, &sa_pause, NULL);

    sa_resume.sa_handler = resume_handler;
    sigemptyset(&sa_resume.sa_mask);
    sa_resume.sa_flags = 0;
    sigaction(SIGUSR2, &sa_resume, NULL);

    sa_term.sa_handler = terminate_handler;
    sigemptyset(&sa_term.sa_mask);
    sa_term.sa_flags = 0;
    sigaction(SIGTERM, &sa_term, NULL);

    printf("File download started. Use SIGUSR1 to pause, SIGUSR2 to resume, and SIGTERM to terminate.\n");

    download_file();

    return 0;
}
```

Input :

Signals are sent to control the download process:

- `SIGUSR1`: Pauses the download.
- `SIGUSR2`: Resumes the download.
- `SIGTERM`: Terminates the program safely, saving progress

```
•   kill -SIGUSR1 <process_id>  # Pause the download
•   kill -SIGUSR2 <process_id>  # Resume the download
•   kill -SIGTERM <process_id>  # Terminate the download
```

Output:

```
File download started. Use SIGUSR1 to pause, SIGUSR2 to resume, and SIGTERM to
terminate.
Downloading... 0% complete
Downloading... 10% complete
Downloading... 20% complete
Download paused.
Download resumed.
Downloading... 30% complete
Downloading... 40% complete
Download complete!
```