// Assignment 1

1. **Write a program to create a file with a hole in it.**

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define FILENAME "sparse_file.bin"
#define FILE_SIZE 10*10 // 1 MB

int main() {
    int fd = open(FILENAME, O_CREAT |
O_WRONLY | O_TRUNC, 0644);
    if (fd == -1) {
        perror("Error opening file");
        return EXIT_FAILURE;
    }

    // Write data at the beginning
    write(fd, "Hello, this is the start of the
file.", 37);

    // Create a hole (seek to a specific offset
without writing)
    lseek(fd, 10, SEEK_SET);

    // Write data at the end
    lseek(fd, FILE_SIZE - 30, SEEK_SET);
    write(fd, "This is the end of the file.", 30);

    close(fd);
    printf("Sparse file created: %s\n",
FILENAME);
    return EXIT_SUCCESS;
}
```

2. **Write a program to open a file and go to sleep for 15 seconds before terminating**.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // For sleep function
#define FILENAME "temp_file.txt"

int main() {
    printf("Opening the file: %s\n",
FILENAME);

    FILE *file = fopen(FILENAME, "w");
    if (file == NULL) {
        perror("Error opening file");
        return EXIT_FAILURE;
    }
    sleep(2);
    printf("Writing to the file...\n");
    fprintf(file, "The program is going to
sleep for 15 seconds.\n");
    fclose(file);
    sleep(2);

    printf("Sleeping for 15 seconds...\n");
    sleep(15);

    printf("Waking up and terminating the
program.\n");

    return EXIT_SUCCESS;
}
```

3. **Write a program to read the current directory and display the name of the files, size ofthe file, type of file and no of files in the current directory**.

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.h>


int main() {
    DIR *directory;
    struct dirent *entry;
    struct stat fileStat;
    int fileCount = 0;
    directory = opendir(".");

    if (directory == NULL) {
        perror("Error opening directory");
        return EXIT_FAILURE;

    }

    printf("Files in the current directory:\n");
    printf("%-25s %-10s %-10s\n", "File
Name", "Size (bytes)", "Type");
    while ((entry = readdir(directory)) !=
NULL) {
        if (strcmp(entry->d_name, ".") == 0 ||
strcmp(entry->d_name, "..") == 0) {
            continue;
        }

        if (stat(entry->d_name, &fileStat) == -
1) {
            perror("Error getting file stats");
            continue;
        }

        char *fileType;
        if (S_ISREG(fileStat.st_mode)) {
            fileType = "File";
        } else if (S_ISDIR(fileStat.st_mode)) {
            fileType = "Directory";
        } else {
            fileType = "Other";
        }

        printf("%-25s %-10ld %-10s\n", entry-
>d_name, fileStat.st_size, fileType);
        fileCount++;
    }

    closedir(directory);
    printf("\nTotal number of files: %d\n",
fileCount);
    return EXIT_SUCCESS;
}
```

// Assignment 2

1. **Write a C program that illustrates banking transactions using the 4th and 5th buffer allocation scenario. Consider three processes- EMI, withdraw and deposit. EMI and withdrawal processes will go into sleep mode due to insufficient balance. These two processes should be included in the race condition after completion of the deposit process. Display inter process communication between all these processes by implementing free list and buffer block list. (Use appropriate system calls too).**

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

// Semaphore operations
#define P(s) semop(s, &(struct sembuf){0, -1, 0}, 1)
#define V(s) semop(s, &(struct sembuf){0, 1, 0}, 1)

#define SHM_KEY 0x1234
#define SEM_KEY 0x5678

typedef struct {
    int balance;
} Data;

Data *shared_data;
int semid;

void deposit(Data *data, int amt) {
    P(semid);
    data->balance += amt;
    printf("Deposited %d. Balance: %d\n", amt, data->balance);
    V(semid);
}

void withdraw(Data *data, int amt) {
    while (1) {
        P(semid);
        if (data->balance >= amt) {
            data->balance -= amt;
            printf("Withdrew %d. Balance: %d\n", amt, data->balance);
            V(semid);
            break;
        } else {
            printf("Insufficient balance for withdrawal of %d. Sleeping...\n", amt);
            V(semid);
            sleep(10);
            kill(getppid(), SIGUSR1);
        }
    }
}

void emi(Data *data, int amt) {
    while (1) {
        P(semid);
        if (data->balance >= amt) {
            data->balance -= amt;
            printf("EMI paid %d. Balance: %d\n", amt, data->balance);
            V(semid);
            break;
        } else {
            printf("Insufficient balance for EMI of %d. Sleeping...\n", amt);
            V(semid);
            sleep(10);
            kill(getppid(), SIGUSR1);
        }
    }
}

void sig_handler(int sig) {
    if (sig == SIGUSR1) {
        int amt;
        printf("Enter amount to deposit: ");
        scanf("%d", &amt);
        deposit(shared_data, amt);
    }
}

int main() {
    int shmid = shmget(SHM_KEY, sizeof(Data), IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }
    shared_data = (Data *)shmat(shmid, NULL, 0);
    if (shared_data == (void *)-1) {
        perror("shmat");
        exit(1);
    }

    semid = semget(SEM_KEY, 1, IPC_CREAT | 0666);
    if (semid == -1) {
        perror("semget");
        exit(1);
    }
    semctl(semid, 0, SETVAL, 1);

    shared_data->balance = 1000;
    printf("Initial balance: %d\n", shared_data->balance);

    signal(SIGUSR1, sig_handler);

    int deposit_amt, withdraw_amt, emi_amt;

    printf("Enter amount to deposit: ");
    scanf("%d", &deposit_amt);

    printf("Enter amount to withdraw: ");
    scanf("%d", &withdraw_amt);

    printf("Enter amount for EMI: ");
    scanf("%d", &emi_amt);

    pid_t pid;

    if ((pid = fork()) == 0) {
        deposit(shared_data, deposit_amt);
        exit(0);
    }

    if ((pid = fork()) == 0) {
        emi(shared_data, emi_amt);
        exit(0);
    }

    if ((pid = fork()) == 0) {
        withdraw(shared_data, withdraw_amt);
        exit(0);
    }

    while (wait(NULL) > 0);

    shmdt(shared_data);
    shmctl(shmid, IPC_RMID, NULL);
    semctl(semid, 0, IPC_RMID);

    return 0;
}
```

2. **Write a C program that demonstrate buffer block allocation (use memory pool to fixed size buffer block. Store string in block and print it).**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BLOCK_SIZE 64  // Size of each buffer block
#define POOL_SIZE 5    // Number of blocks in the memory pool

typedef struct {
    char blocks[POOL_SIZE][BLOCK_SIZE];  // Array of fixed-size blocks
    int nextFree;                        // Index of the next free block
} MemoryPool;

MemoryPool* createMemoryPool() {
    MemoryPool *pool = (MemoryPool *)malloc(sizeof(MemoryPool));
    if (!pool) {
        perror("Failed to create memory pool");
        return NULL;
    }
    pool->nextFree = 0;  // Start with the first block
    return pool;
}
char* allocateBlock(MemoryPool *pool) {
    if (pool->nextFree < POOL_SIZE) {
        return pool->blocks[pool->nextFree++];
    } else {
        printf("Memory pool is full!\n");
        return NULL;  // No free block available
    }
}
void freeMemoryPool(MemoryPool *pool)
{
    free(pool);
}

int main() {
    MemoryPool *pool = createMemoryPool();
    if (!pool) {
        return EXIT_FAILURE;
    }

    const char *strings[] = {
        "Hello, World!",
        "Memory pools are efficient!",
        "This is a fixed-size buffer block allocation.",
        "C programming is fun.",
        "Learning about memory management."
    };
    for (int i = 0; i < POOL_SIZE; i++) {
        char *block = allocateBlock(pool);
        if (block) {
            strncpy(block, strings[i], BLOCK_SIZE - 1); // Copy string to block
            block[BLOCK_SIZE - 1] = '\0'; // Ensure null-termination
        }
    }
    printf("Stored strings in the memory pool:\n");
    for (int i = 0; i < pool->nextFree; i++) {
        printf("%s\n", pool->blocks[i]);
    }

    freeMemoryPool(pool);
    return EXIT_SUCCESS;
}
```

// Assignment 3

1. **Write a program to create ‘n’ children. When the children will terminate, display total cumulative time children spent in user and kernel mode.**

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

int main() {
   int n = 5; // Number of child processes
   pid_t pid;
   int i, status;
   struct rusage usage;
   // Create n child processes
   for (i = 0; i < n; i++) {
      pid = fork();

      if (pid < 0) {
         perror("fork");
         exit(EXIT_FAILURE);
      }

      if (pid == 0) {
         printf("Child %d (PID: %d)
started\n", i, getpid());
         sleep(1);
         printf("Child %d (PID: %d)
terminating\n", i, getpid());
         exit(0);
      }
   }
   // Parent process waits for all child
processes to terminate
   for (i = 0; i < n; i++) {
      waitpid(-1, &status, 0);  // Wait for any
child process
   }
   // Retrieve and print cumulative resource
usage for children
  getrusage(RUSAGE_CHILDREN,
&usage) ;
   long user_time_microseconds =
(long)usage.ru_utime.tv_sec * 1000000 +
usage.ru_utime.tv_usec;
   long system_time_microseconds =
(long)usage.ru_stime.tv_sec * 1000000 +
usage.ru_stime.tv_usec;

   printf("Total cumulative time spent by
children:\n");
   printf("User time: %ld microseconds\n",
user_time_microseconds);
   printf("System time: %ld
microseconds\n",
system_time_microseconds);
   return 0;
}
```

---

2. **Write a program to demonstrate the use of atexit() function.**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // for sleep()

void dummy1(void) {
   printf("dummy function 1 called.\n");
}
void dummy2(void) {
   printf("dummy function 2 called.\n");
}

int main() {
  atexit(dummy1);
  atexit(dummy2);
  printf("Main function running...\n");
  sleep(2);
  printf("Main function terminating...\n");
  return 0;
}
```

---

3. **Write a program to handle the two-way communication between parent and child using pipe.**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#define BUFFER_SIZE 100
int main() {
   int pipe1[2];  // Pipe for parent-to-child
communication
   int pipe2[2];  // Pipe for child-to-parent
communication
   pid_t pid;
   char buffer[BUFFER_SIZE];

   // Create pipes
   pipe(pipe1);
   pipe(pipe2);

   pid = fork();

   if (pid == 0) {  // Child process
      close(pipe1[1]);
      close(pipe2[0]);
      // Read message from the parent
      ssize_t bytesRead = read(pipe1[0],
buffer, sizeof(buffer) - 1);
      buffer[bytesRead] = '\0';
      printf("Child received: %s\n", buffer);

      // Get user input in child process
      printf("Child: Enter a message to send
to parent: ");
      fgets(buffer, BUFFER_SIZE, stdin);
      buffer[strcspn(buffer, "\n")] = '\0';  //
Remove newline

      // Send message to the parent
      write(pipe2[1], buffer, strlen(buffer));

      close(pipe1[0]);
      close(pipe2[1]);

      exit(EXIT_SUCCESS);
   } else {  // Parent process
      close(pipe1[0]);
      close(pipe2[1]);

      // Get user input in parent process
      printf("Parent: Enter a message to send
to child: ");
      fgets(buffer, BUFFER_SIZE, stdin);
      buffer[strcspn(buffer, "\n")] = '\0';  //
Remove newline

      // Send message to the child
      write(pipe1[1], buffer, strlen(buffer));

      // Read message from the child
      ssize_t bytesRead = read(pipe2[0],
buffer, sizeof(buffer) - 1);
      buffer[bytesRead] = '\0';
      printf("Parent received: %s\n", buffer);
      close(pipe1[1]);
      close(pipe2[0]);

      wait(NULL);
   }

   return 0;
}
```

---/

// Assignment 4

1. **Write a C program that illustrates inter process communication using shared memory**.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <wait.h>

#define SHM_KEY 1234
#define SHM_SIZE 1024

int main() {
   int shm_id;
   char *shm_ptr;
   pid_t pid;

   shm_id = shmget(SHM_KEY,
SHM_SIZE, IPC_CREAT | 0666);
   shm_ptr = (char *) shmat(shm_id, NULL,
0);
   pid = fork();

   if (pid == 0) {
      sleep(2);
      printf("Child process reads: %s\n",
shm_ptr);
            sleep(1);

      snprintf(shm_ptr, SHM_SIZE, "I'm
child");
            printf("Child process writes:
%s\n", shm_ptr);
      sleep(1);
      exit(0);
   } else {
      snprintf(shm_ptr, SHM_SIZE, "I'm
parent");
      printf("Parent process writes: %s\n",
shm_ptr);

      wait(NULL);

      printf("Parent process reads: %s\n",
shm_ptr);
   }
   return 0;
}
```

---

2. **Write a program that intentionally creates a memory leak and then modify it to fix the leak**

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int n = 500;
   int *arr = (int *)malloc(n * sizeof(int));

   for (int i = 0; i < n; i++) {
      arr[i] = i;
      printf("arr[%d] = %d\n", i, arr[i]);
   }
   printf("Memory leak created\n");

   free(arr);
   arr = NULL;

   if (arr == NULL) {
      printf("Memory Freed\n");
   } else {
      printf("Memory not Freed\n");
   }
   return 0;
}
```

// Assignment 5

1. **Write a C program which creates a child process which catches a signal sighup, sigint and sigquit. The Parent process send a sighup or sigint signal after every 3 seconds, at the end of 30 second parent send sigquit signal to child and child terminates by displaying message "Parent Process has killed child process!!!".**

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

void handle_hup(int sig) {
   printf("Child received SIGHUP\n");
   //fflush(stdout); // Ensure immediate
output
}

void handle_int(int sig) {
   printf("Child received SIGINT\n");
   //fflush(stdout); // Ensure immediate
output
}

void handle_quit(int sig) {
   printf("\n\nChild received SIGQUIT\n");
   printf("Parent Process has killed child
process!!!\n");
   //fflush(stdout); // Ensure immediate
output
   exit(0);
}

int main() {
   pid_t pid;
   int i;

   // Create a child process
   pid = fork();
   if (pid < 0) {
      perror("fork failed");
      exit(EXIT_FAILURE);
   }

   if (pid == 0) { // Child process
      // Set up signal handlers
      signal(SIGHUP, handle_hup);
      signal(SIGINT, handle_int);
      signal(SIGQUIT, handle_quit);

      // Wait for signals
      while (1) {
         pause(); // Wait for signals
      }
   } else { // Parent process
      sleep(1); // Allow child to set up

      // Send signals every 3 seconds for 30
seconds
      for (i = 0; i < 10; i++) {
         sleep(3); // Wait 3 seconds between
signals
         if (i % 2 == 0) {
            kill(pid, SIGHUP); // Send
SIGHUP
         } else {
            kill(pid, SIGINT); // Send
SIGINT
         }
      }
      // Send SIGQUIT after 30 seconds
      kill(pid, SIGQUIT); // Send SIGQUIT

      // Wait for child to terminate
      wait(NULL);
   }

   return 0;
}
```

2. **Write a C program that illustrates suspending and resuming processes using signals.**

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

void child_process() {
   while (1) {
      printf("\nChild process is
running...\n");
      fflush(stdout); // Ensure immediate
output
      sleep(1); // Simulate work by sleeping
for 1 second
   }
}

int main() {
   pid_t pid;
   char input;

   // Create a child process
   pid = fork();
   if (pid < 0) {
      perror("fork failed");
      exit(EXIT_FAILURE);
   }

   if (pid == 0) { // Child process
      child_process();
   } else { // Parent process
      while (1) {
         printf("Enter 's' to suspend, 'r' to
resume, or 'q' to quit: ");
         scanf(" %c", &input); // Read user
input

         if (input == 's') {
            printf("\nParent sending
SIGSTOP to suspend the child process...");
            kill(pid, SIGSTOP); // Suspend
the child process
         } else if (input == 'r') {
            printf("\nParent sending
SIGCONT to resume the child process...");
            kill(pid, SIGCONT); // Resume
the child process
         } else if (input == 'q') {
            printf("\nExiting the program...");
            kill(pid, SIGTERM); // Terminate
the child process
            wait(NULL); // Wait for the child
to terminate
            break;
         } else {
            printf("\nInvalid input! Please
enter 's', 'r', or 'q'.");
         }
      }
   }
   return 0;
}
```

3. **Write a C program that handling both SIGINT (triggered by pressing Ctrl+C) and SIGTERM (which can be sent to the process using the kill command) using signal handlers**

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

// Signal handler for SIGINT
void handle_sigint(int sig) {
   printf("\nCaught SIGINT (Ctrl+C).
Exiting gracefully...\n");
   exit(0);
}

// Signal handler for SIGTERM
void handle_sigterm(int sig) {
   printf("\nCaught SIGTERM. Exiting
gracefully...\n");
   exit(0);
}

int main() {
   // Register signal handlers
   signal(SIGINT, handle_sigint);
   signal(SIGTERM, handle_sigterm);

   printf("Process ID: %d\n", getpid());
   printf("Waiting for signals (SIGINT or
SIGTERM)... Press Ctrl+C to send
SIGINT.\n");

   // Infinite loop to keep the program
running
   while (1) {
      printf("Running...\n");
      sleep(15);
   }

   return 0;
}
```