Signal Handling, Mutexes, Semaphores and System Calls

Computing Lab
CS 69201
Department of Computer Science and Engineering
IIT Kharagpur

Preamble

- ➢ We delved deep into the basics of System Programming.
- ➢ We discussed about Processes, Concurrency, Parallelism, Event Loops, threads, pthreads, IPC, Synchronization, Mutex locks etc.
- ➢ We shall ponder upon other aspects - Signal Handling , System calls, Shared Memory.

Signal Handling

- While writing codes, we make some mistakes like accessing unknown memory locations, divide something by 0. This may lead to spurious behaviors during execution. **How to understand?**
- Also, due to some reasons, we would terminate our codes abruptly. That is, we interrupt the system by pressing keys, switching off PC etc. **How can we convey such information?**
- In IPC, we need to communicate with other processes. **How?**

All I Have To Do Is Signal!

Signals

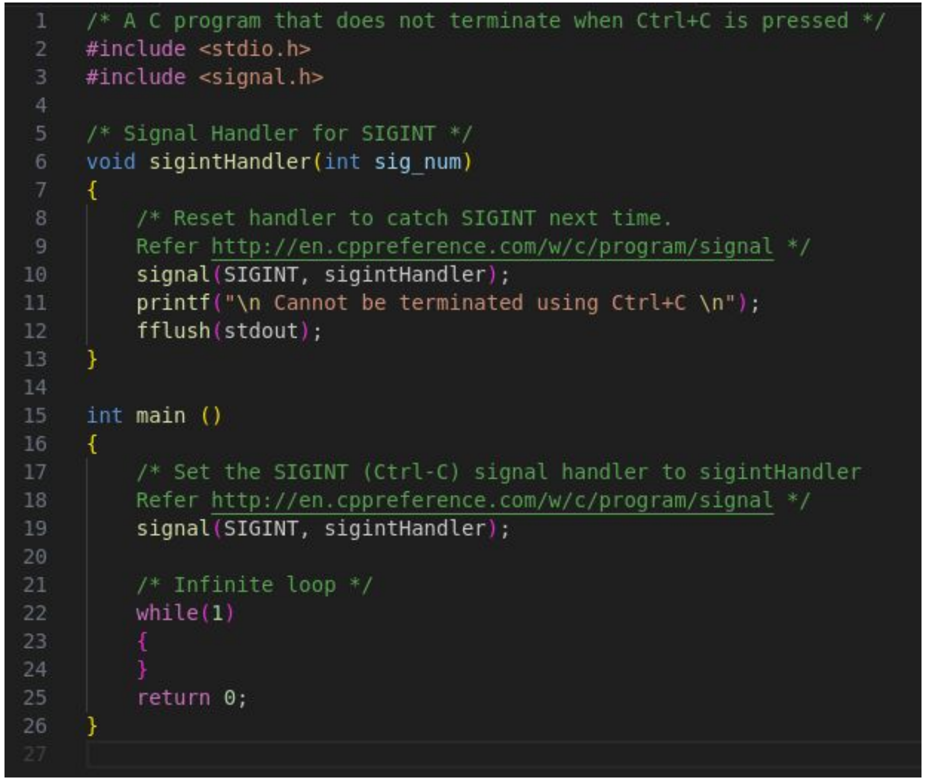
- A signal is a software generated interrupt that is sent to a process by the OS.
- A signal can report some exceptional behavior within the program (such as division by zero), or a signal can report some asynchronous event outside the program (such as someone striking an interactive attention key on a keyboard).
- Some examples include :
  - `SIGFPE - Floating Point Exception`
  - `SIGSEGV - The infamous segmentation fault.`
  - `SIGINT - Interrupt call Ctrl + C`
  - `SIGILL - Illegal Instruction`
  - `SIGSTP - Signal Trap Ctrl + Z.(Doing so may lead to process in zombie state - consumes resources for no reason).`

These signals are contained in <signal.h>

Signal Handler

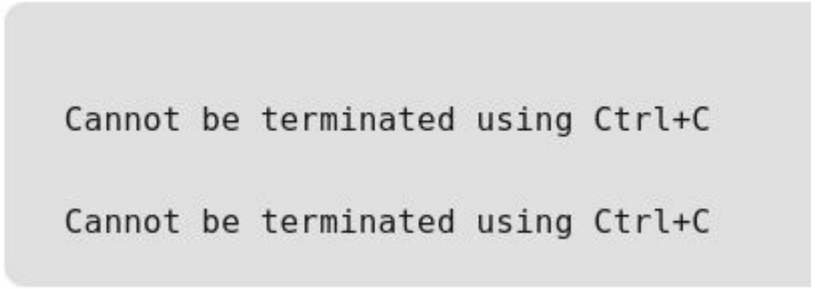
- ➢ A signal handler is a function which is called by the target environment when the corresponding signal occurs.
- ➢ **signal()** is a function which handles signals.
- ➢ The signal() call takes two parameters: the signal in question, and an action to take when that signal is raised.

Signal Handler



```c
/* A C program that does not terminate when Ctrl+C is pressed */
#include <stdio.h>
#include <signal.h>

/* Signal Handler for SIGINT */
void sigintHandler(int sig_num)
{
    /* Reset handler to catch SIGINT next time.
    Refer http://en.cppreference.com/w/c/program/signal */
    signal(SIGINT, sigintHandler);
    printf("\n Cannot be terminated using Ctrl+C \n");
    fflush(stdout);
}

int main ()
{
    /* Set the SIGINT (Ctrl-C) signal handler to sigintHandler
    Refer http://en.cppreference.com/w/c/program/signal */
    signal(SIGINT, sigintHandler);

    /* Infinite loop */
    while(1)
    {
    }
    return 0;
}
```

Output: When Ctrl+C was pressed two times



```
 Cannot be terminated using Ctrl+C

 Cannot be terminated using Ctrl+C
```

pThreads

- For UNIX systems, implementations of threads that adhere to the IEEE POSIX 1003.1c standard are Pthreads.
- The primary motivation behind Pthreads is improving program performance
- Can be created with much less OS overhead & Needs fewer system resources to run.

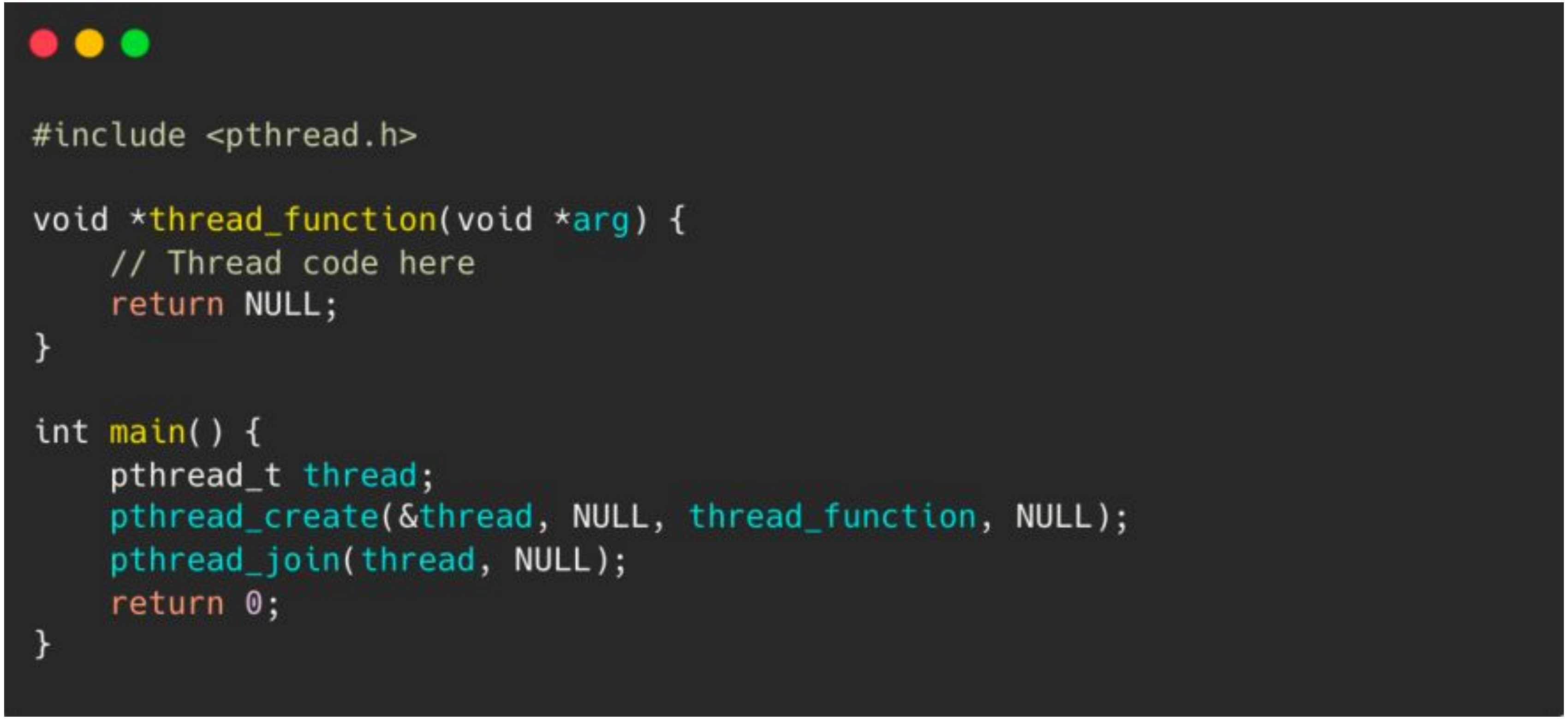
pThreads Library

- Programs must include the file pthread.h
- Programs must be linked with the pthread library
  - gcc -pthread main.c -o main
- Some functions :
  - int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg): Creates a new thread
    - thread: pointer to an unsigned integer value that returns the thread id of the thread created.
    - attr: pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
    - start_routine: pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
    - arg: pointer to void that contains the arguments to the function defined in the earlier argument

pThreads Library

- Some functions :
  - void pthread_exit(void *retval): Terminates the calling thread
    - This method accepts a mandatory parameter retval which is the pointer to an integer that stores the return status of the thread terminated.
    - The scope of this variable must be global so that any thread waiting to join this thread may read the return status.
  - pthread_join(): Causes the calling thread to wait for another thread to terminate

pThreads Library



```c
#include <pthread.h>

void *thread_function(void *arg) {
    // Thread code here
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, thread_function, NULL);
    pthread_join(thread, NULL);
    return 0;
}
```

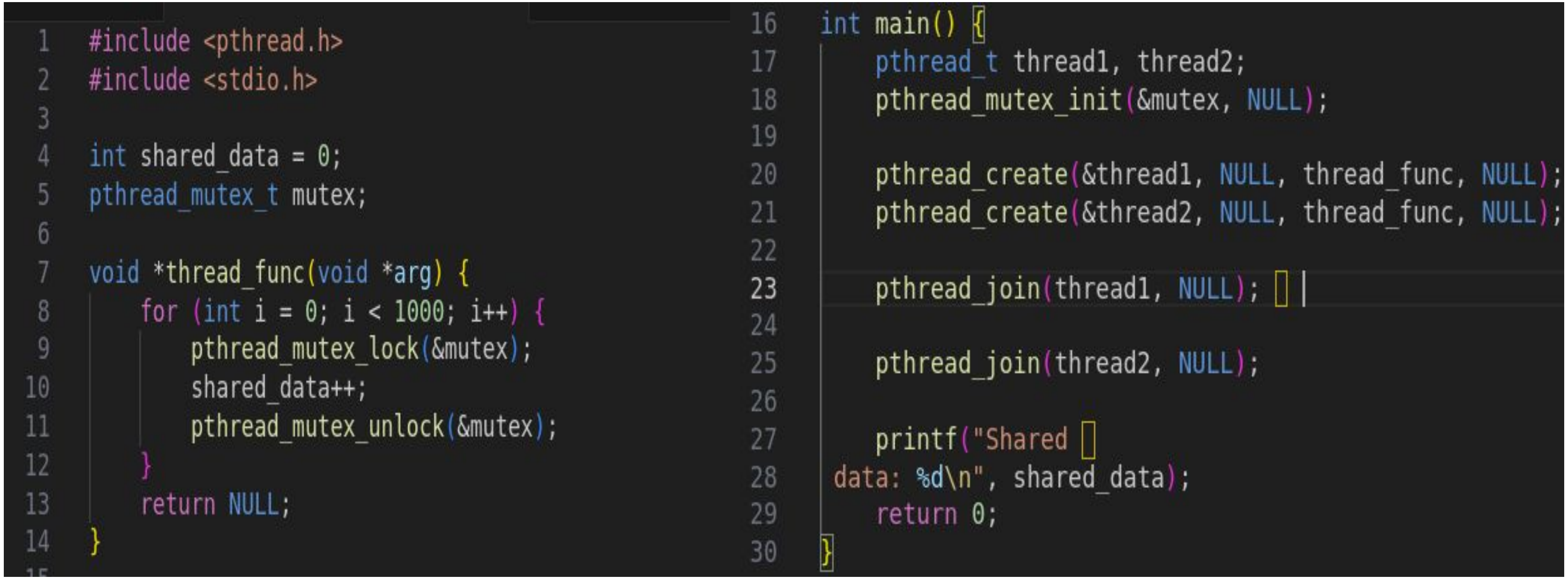
Mutexes

- A mutex (mutual exclusion) is a synchronization object that is used to protect shared data from concurrent access by multiple threads.
- A mutex is a binary semaphore that can be in one of two states: locked or unlocked.
- Only one thread can hold a mutex at a time.

Mutexes

- ➢ To acquire a mutex, you call the pthread_mutex_lock() function.
  - ○ **int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr) :** Creates a mutex, referenced by mutex, with attributes specified by attr.
  - ○ Returns 0 if successful else -1
- ➢ To release a mutex, you call the pthread_mutex_unlock() function.
  - ○ **int pthread_mutex_lock(pthread_mutex_t *mutex) :** Locks a mutex object, which identifies a mutex. If the mutex is already locked by another thread, the thread waits for the mutex to become available.
  - ○ Returns 0 if successful else -1.
- ➢ If a thread tries to acquire a mutex that is already locked, it will block until the mutex is released.

Mutex



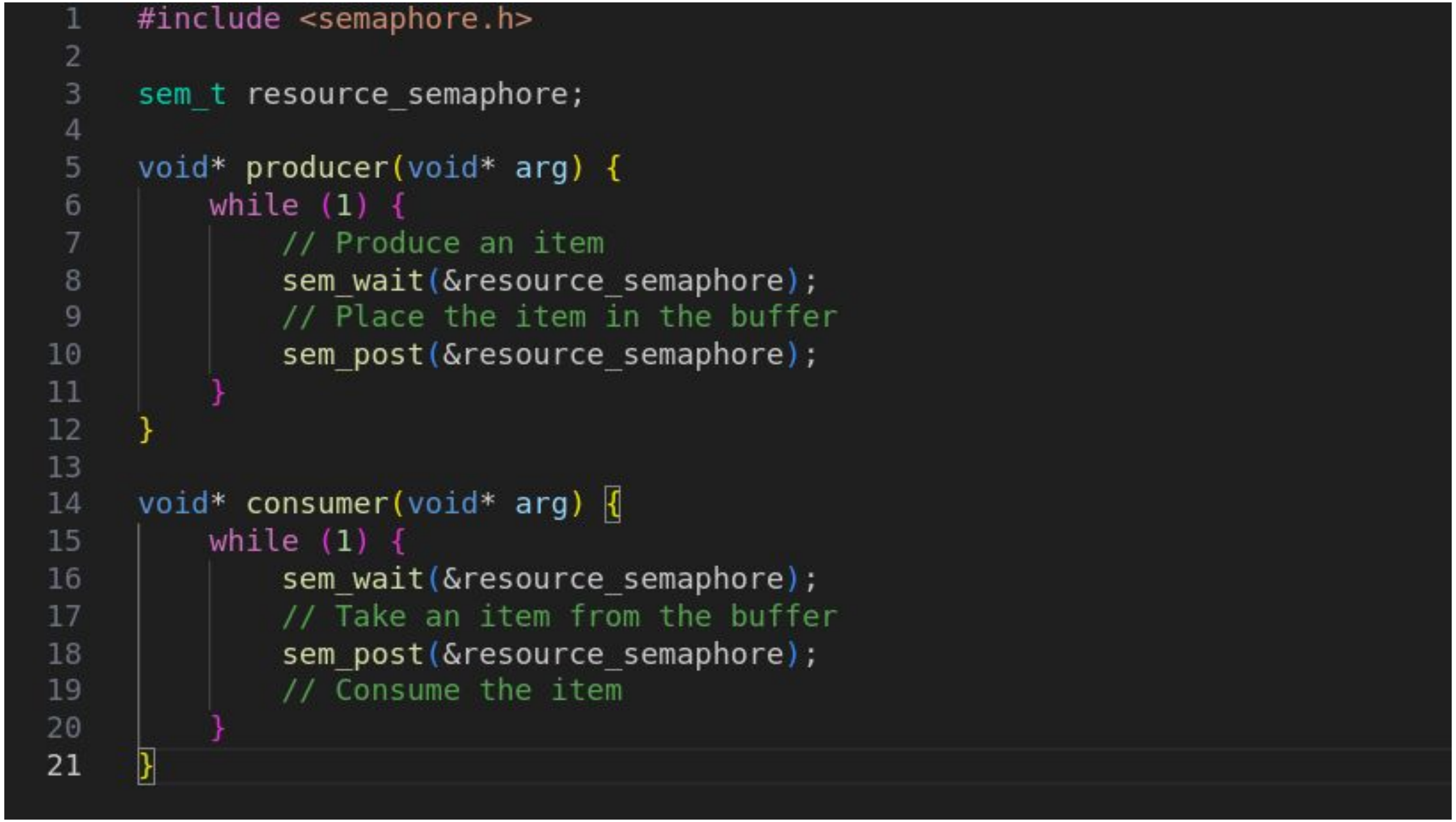
```c
#include <pthread.h>
#include <stdio.h>

int shared_data = 0;
pthread_mutex_t mutex;

void *thread_func(void *arg) {
    for (int i = 0; i < 1000; i++) {
        pthread_mutex_lock(&mutex);
        shared_data++;
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```c
int main() {
    pthread_t thread1, thread2;
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&thread1, NULL, thread_func, NULL);
    pthread_create(&thread2, NULL, thread_func, NULL);

    pthread_join(thread1, NULL);

    pthread_join(thread2, NULL);

    printf("Shared 
 data: %d\n", shared_data);
    return 0;
}
```

Semaphores

- ➢ A synchronization mechanism used to control access to shared resources.
- ➢ A non-negative integer.
- ➢ Operations:
  - **wait(sem)**: Decrements the semaphore value. If the value becomes negative, the process is blocked.
  - **signal(sem)**: Increments the semaphore value. If there are any blocked processes, one of them is unblocked.
- ➢ Types of Semaphores
  - **Counting Semaphore**: Can have any non-negative integer value. Used to control access to a resource with a limited number of available instances.
  - **Binary Semaphore:** Can only have the values 0 and 1. Used to protect critical sections of code.

Semaphores

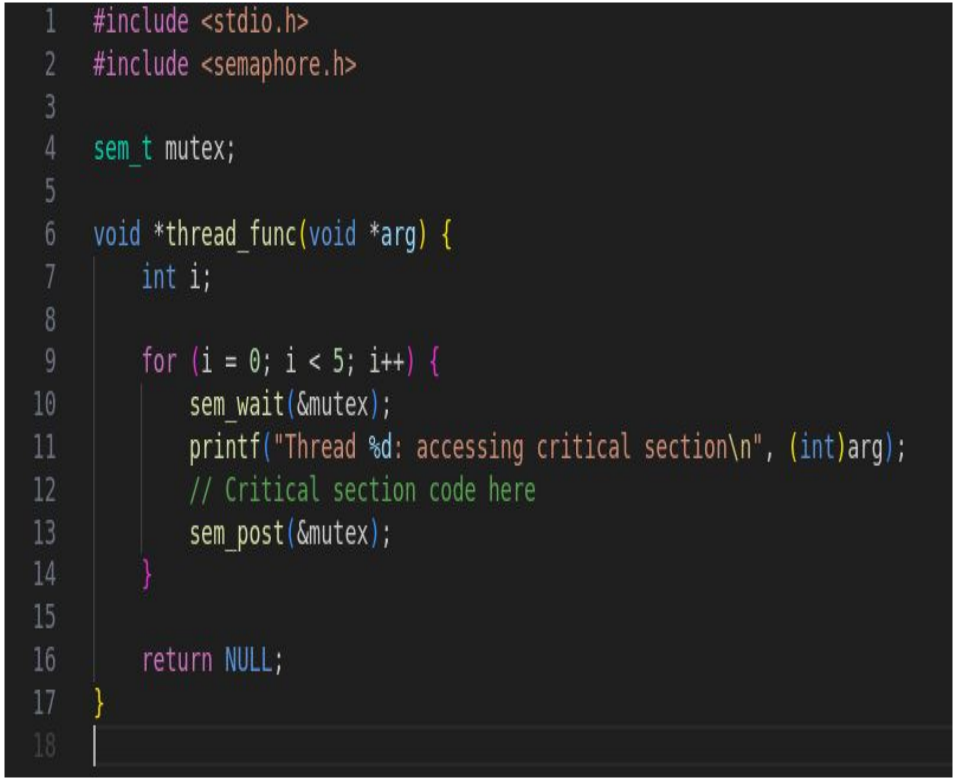


```c
#include <semaphore.h>

sem_t resource_semaphore;

void* producer(void* arg) {
    while (1) {
        // Produce an item
        sem_wait(&resource_semaphore);
        // Place the item in the buffer
        sem_post(&resource_semaphore);
    }
}

void* consumer(void* arg) {
    while (1) {
        sem_wait(&resource_semaphore);
        // Take an item from the buffer
        sem_post(&resource_semaphore);
        // Consume the item
    }
}
```

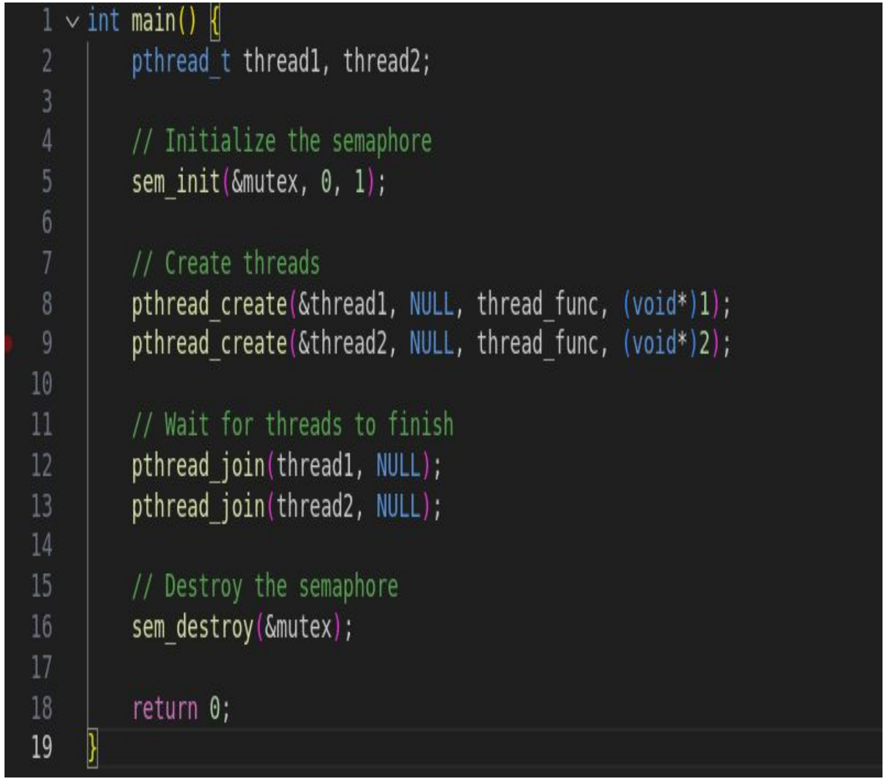
Semaphores

- Operations (include “#include<semaphore.h>”):
  - void sem_init(sem_t *sem, int pshared, unsigned int value):
    - sem : Specifies the semaphore to be initialized.
    - pshared : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
    - value : Specifies the value to assign to the newly initialized semaphore
  - int sem_wait(sem_t *sem): Decrements the semaphore value. If the value becomes negative, the process is blocked. sem : Specifies the semaphore to be initialized.
  - int sem_post(sem_t *sem): Increments the semaphore value. If there are any blocked processes, one of them is unblocked.
  - sem_destroy(sem_t *mutex); Destroys semaphore
- Types of Semaphores
  - Counting Semaphore: Can have any non-negative integer value. Used to control access to a resource with a limited number of available instances.
  - Binary Semaphore: Can only have the values 0 and 1. Used to protect critical sections of code.

Binary Semaphores

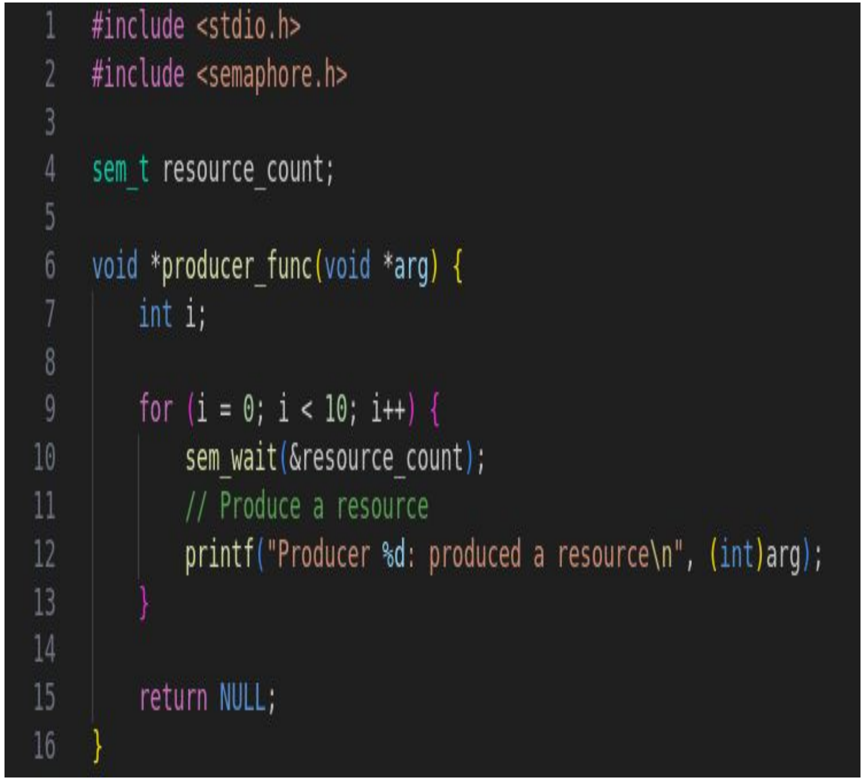


```c
#include <stdio.h>
#include <semaphore.h>

sem_t mutex;

void *thread_func(void *arg) {
    int i;

    for (i = 0; i < 5; i++) {
        sem_wait(&mutex);
        printf("Thread %d: accessing critical section\n", (int)arg);
        // Critical section code here
        sem_post(&mutex);
    }

    return NULL;
}

```

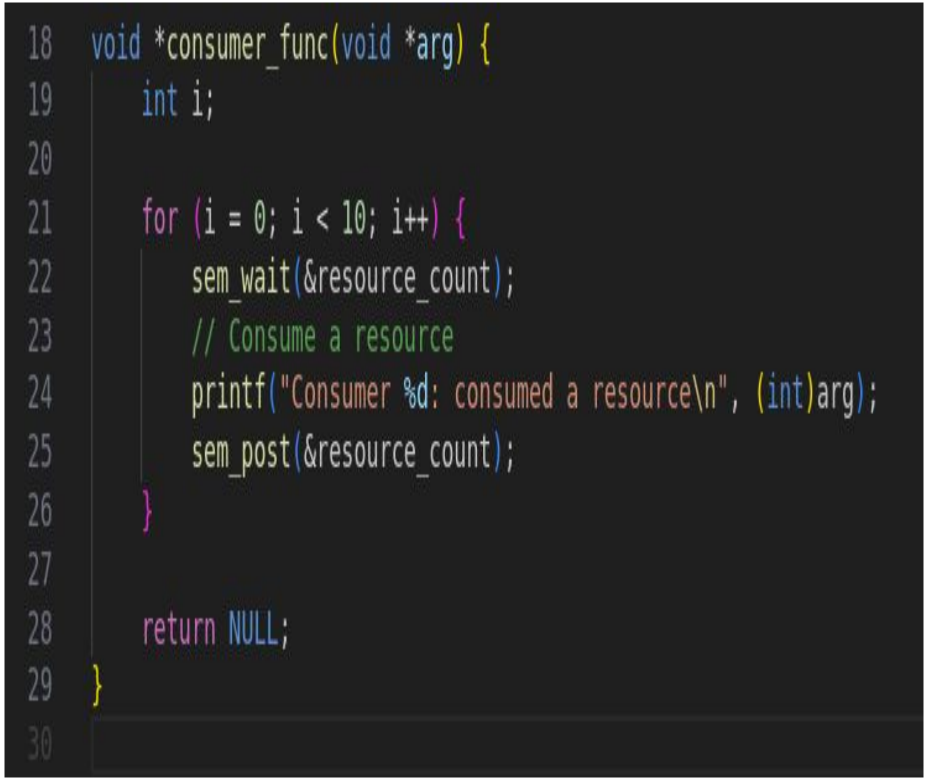


```c
int main() {
    pthread_t thread1, thread2;

    // Initialize the semaphore
    sem_init(&mutex, 0, 1);

    // Create threads
    pthread_create(&thread1, NULL, thread_func, (void*)1);
    pthread_create(&thread2, NULL, thread_func, (void*)2);

    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Destroy the semaphore
    sem_destroy(&mutex);

    return 0;
}
```

Counting Semaphores

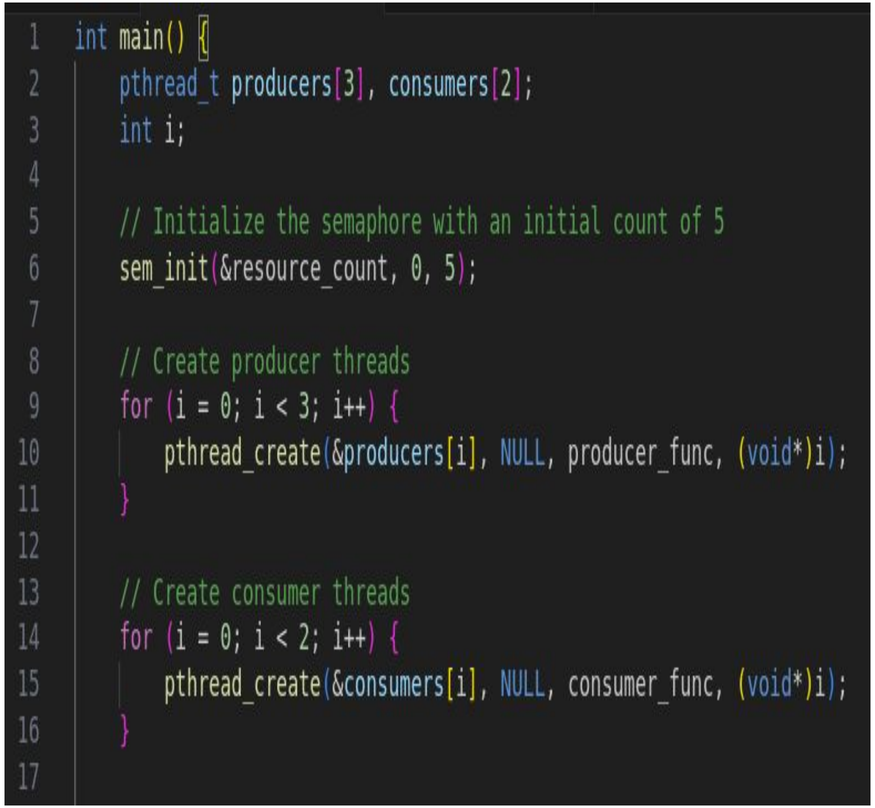


```c
#include <stdio.h>
#include <semaphore.h>

sem_t resource_count;

void *producer_func(void *arg) {
    int i;

    for (i = 0; i < 10; i++) {
        sem_wait(&resource_count);
        // Produce a resource
        printf("Producer %d: produced a resource\n", (int)arg);
    }

    return NULL;
}
```

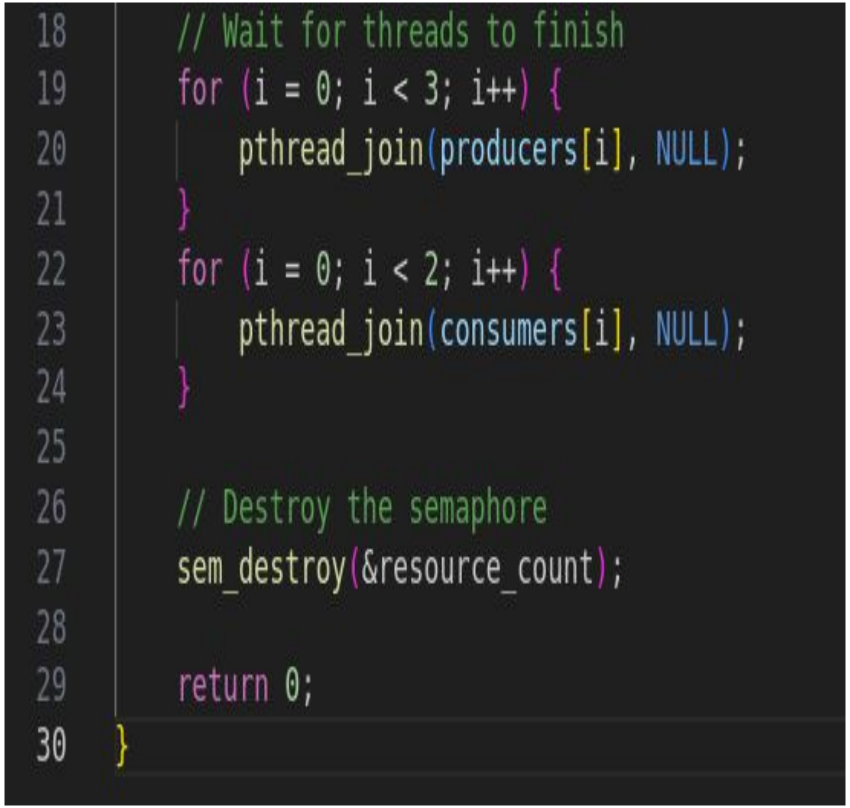


```c
void *consumer_func(void *arg) {
    int i;

    for (i = 0; i < 10; i++) {
        sem_wait(&resource_count);
        // Consume a resource
        printf("Consumer %d: consumed a resource\n", (int)arg);
        sem_post(&resource_count);
    }

    return NULL;
}

```

Counting Semaphores



```c
int main() {
    pthread_t producers[3], consumers[2];
    int i;

    // Initialize the semaphore with an initial count of 5
    sem_init(&resource_count, 0, 5);

    // Create producer threads
    for (i = 0; i < 3; i++) {
        pthread_create(&producers[i], NULL, producer_func, (void*)i);
    }

    // Create consumer threads
    for (i = 0; i < 2; i++) {
        pthread_create(&consumers[i], NULL, consumer_func, (void*)i);
    }

```

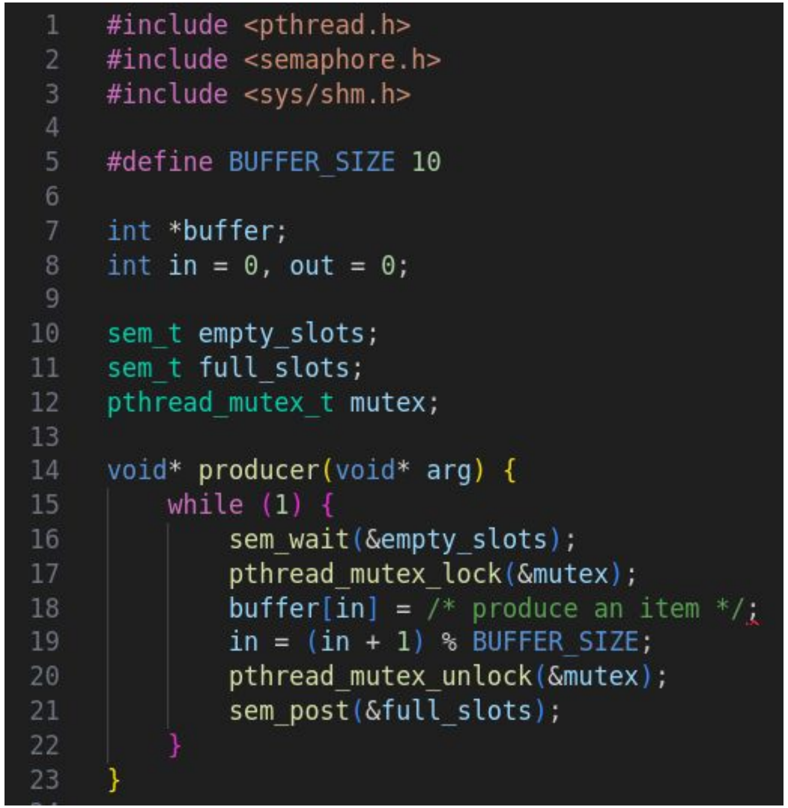


```c
    // Wait for threads to finish
    for (i = 0; i < 3; i++) {
        pthread_join(producers[i], NULL);
    }
    for (i = 0; i < 2; i++) {
        pthread_join(consumers[i], NULL);
    }

    // Destroy the semaphore
    sem_destroy(&resource_count);

    return 0;
}
```

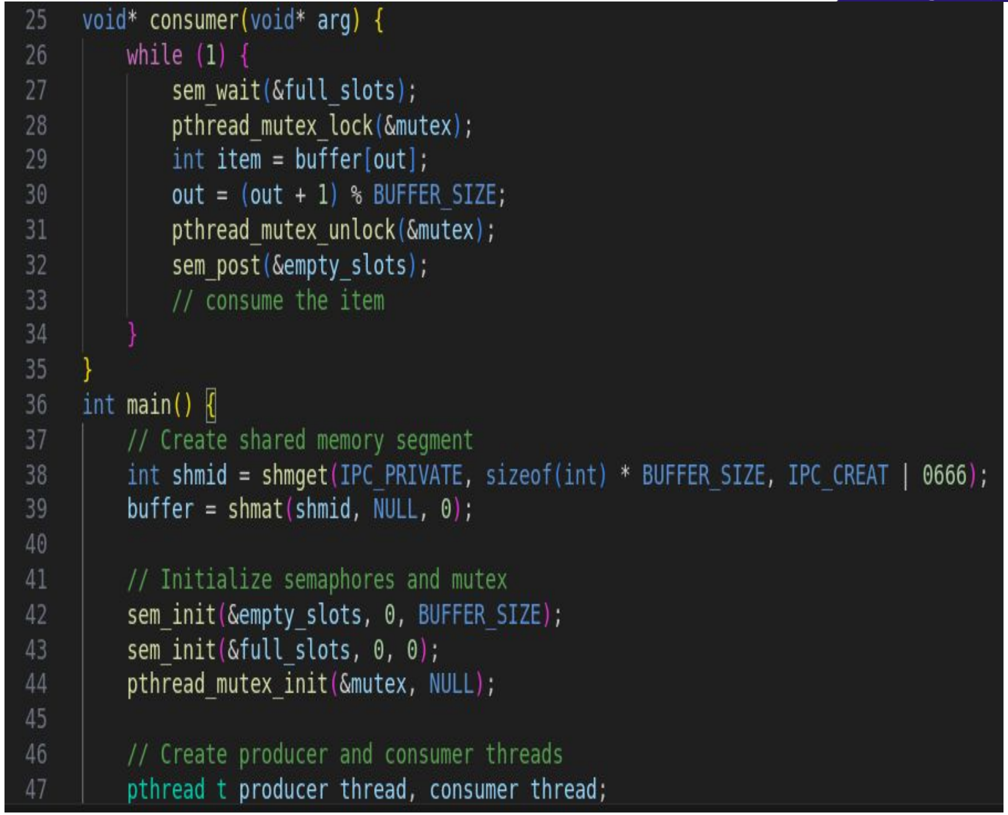
Shared Memory

- A technique where multiple processes can access the same memory region. This allows for efficient communication and data sharing between processes.
- Operations (include the line “#include<sys/shm.h>”:
  - shmget(IPC_PRIVATE, sizeof(int) * BUFFER_SIZE, IPC_CREAT | 0666) :
    - Creates a shared memory segment with a unique identifier (IPC_PRIVATE).
    - Allocates memory for the buffer with a size of sizeof(int) * BUFFER_SIZE.
    - Sets the permissions for the shared memory segment to 0666 (read and write permissions for owner, group, and others).
  - buffer = shmat(shmid, NULL, 0)
    - Attaches the shared memory segment to the process's address space.
    - Returns a pointer to the attached memory, which is stored in the buffer variable.
  - shmdt(buffer): Detach the shared memory segment from the process's address space.
  - shmctl(shmid, IPC_RMID, NULL): Remove the shared memory segment from the system.

Shared Memory - Producer Consumer

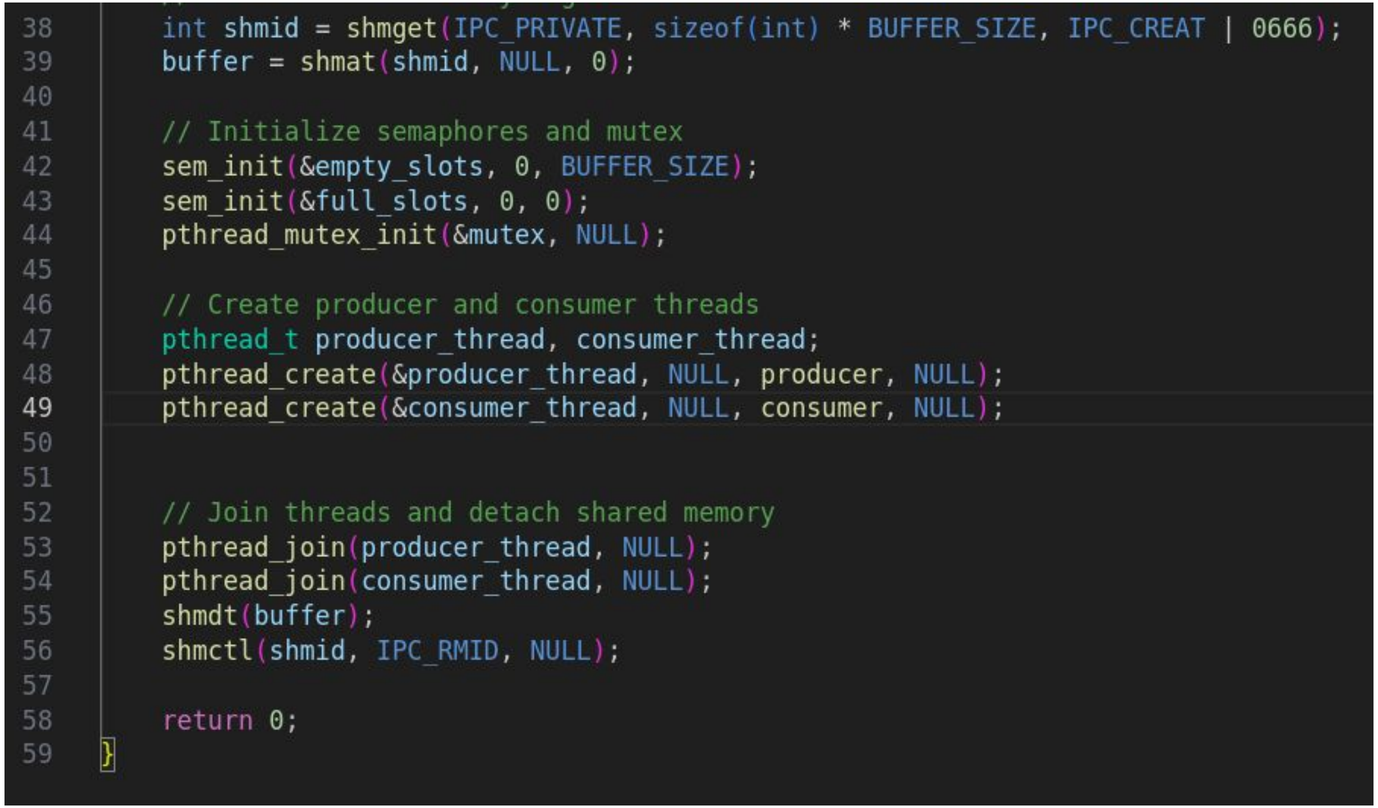


```c
#include <pthread.h>
#include <semaphore.h>
#include <sys/shm.h>

#define BUFFER_SIZE 10

int *buffer;
int in = 0, out = 0;

sem_t empty_slots;
sem_t full_slots;
pthread_mutex_t mutex;

void* producer(void* arg) {
    while (1) {
        sem_wait(&empty_slots);
        pthread_mutex_lock(&mutex);
        buffer[in] = /* produce an item */;
        in = (in + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&full_slots);
    }
}
```



```c
void* consumer(void* arg) {
    while (1) {
        sem_wait(&full_slots);
        pthread_mutex_lock(&mutex);
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty_slots);
        // consume the item
    }
}
int main() {
    // Create shared memory segment
    int shmid = shmget(IPC_PRIVATE, sizeof(int) * BUFFER_SIZE, IPC_CREAT | 0666);
    buffer = shmat(shmid, NULL, 0);

    // Initialize semaphores and mutex
    sem_init(&empty_slots, 0, BUFFER_SIZE);
    sem_init(&full_slots, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    // Create producer and consumer threads
    pthread_t producer_thread, consumer_thread;
```

Shared Memory - Producer Consumer



```
    int shmid = shmget(IPC_PRIVATE, sizeof(int) * BUFFER_SIZE, IPC_CREAT | 0666);
    buffer = shmat(shmid, NULL, 0);

    // Initialize semaphores and mutex
    sem_init(&empty_slots, 0, BUFFER_SIZE);
    sem_init(&full_slots, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    // Create producer and consumer threads
    pthread_t producer_thread, consumer_thread;
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);


    // Join threads and detach shared memory
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);
    shmdt(buffer);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

System Calls

- ➢ A system call is a programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
- ➢ A system call is a way for programs to interact with the operating system.
- ➢ A computer program makes a system call when it requests the operating system's kernel.
- ➢ Features of System Calls :
  - Provide a well-defined interface between user programs and the operating system.
  - Access privileged operations that are not available to normal user programs.
  - Switch to Kernel Mode
  - Context Switch
  - Error Handling
  - Synchronization

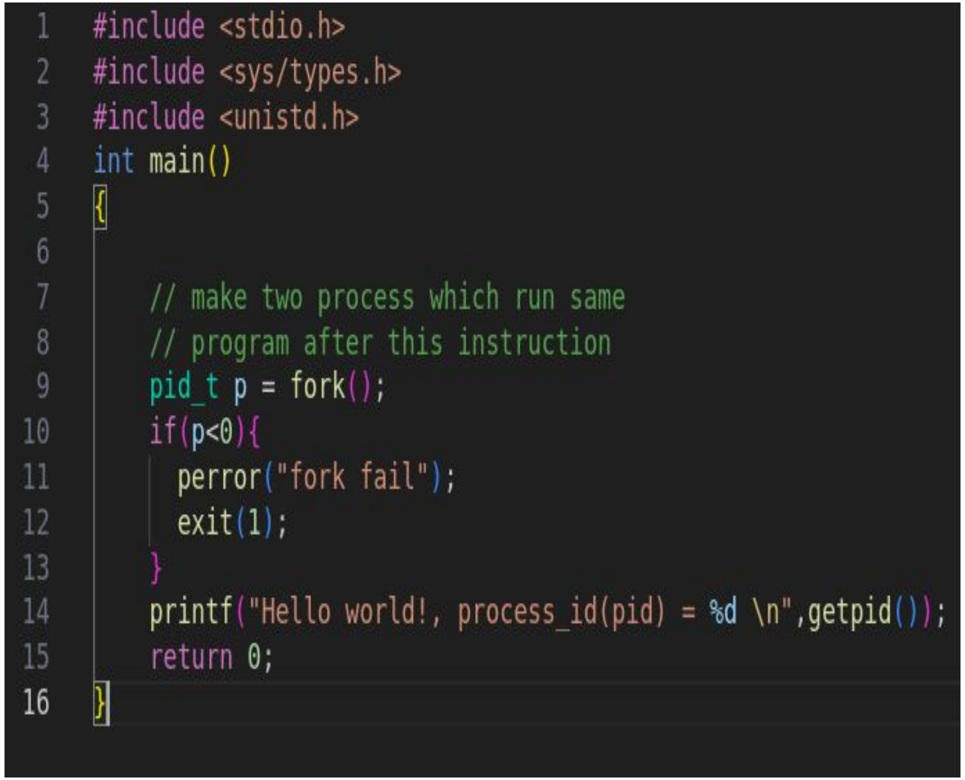
System Calls

- Procedure of System Call Invocation :
  - Users need special resources
  - The program makes a system call request
  - Operating system sees the system call
  - The operating system performs the operations
  - Operating system give control back to the program
- Examples : fork(), pipe()

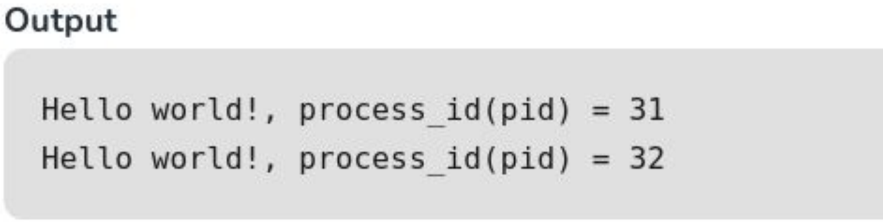
fork()

- Creates a new process as a child process of the calling process (parent)
- Both have similar code segments
- The child gets a copy of the parents data segment at the time of forking
- Returns the following :
  - Negative Value: The creation of a child process was unsuccessful.
  - Zero: Returned to the newly created child process.
  - Positive value: Returned to parent or caller. The value contains the process ID of the newly created child process.

fork()

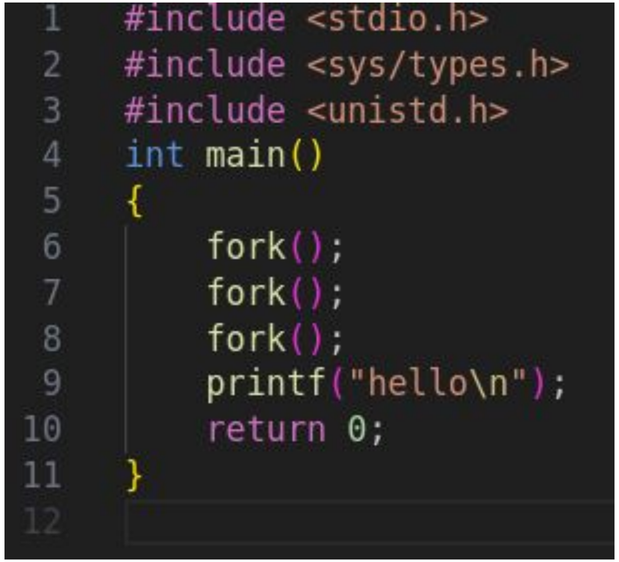


```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    pid_t p = fork();
    if(p<0){
      perror("fork fail");
      exit(1);
    }
    printf("Hello world!, process_id(pid) = %d \n",getpid());
    return 0;
}
```

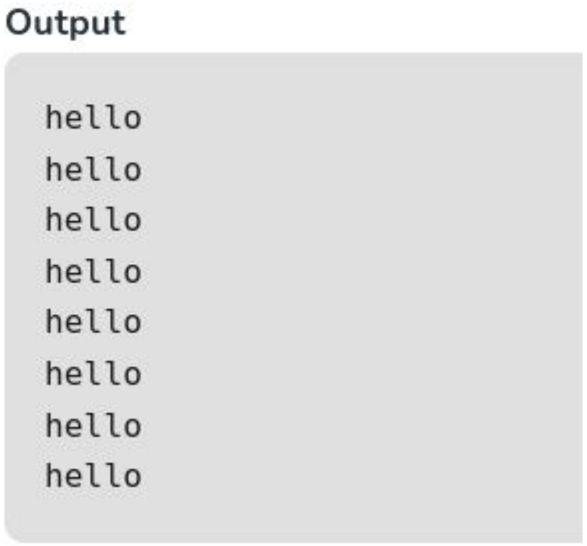


```
Hello world!, process_id(pid) = 31
Hello world!, process_id(pid) = 32
```

fork()



```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```



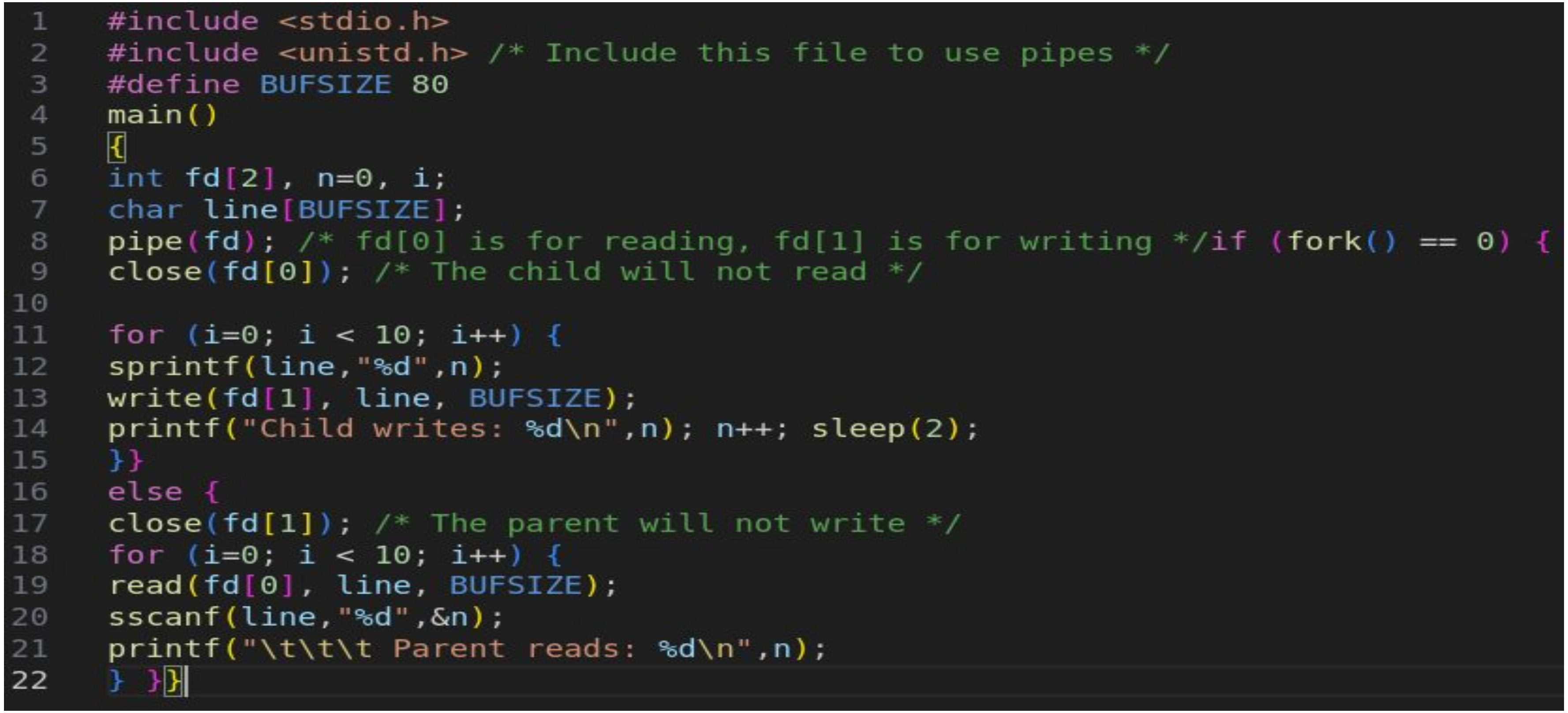
Output

```
hello
hello
hello
hello
hello
hello
hello
hello
```

pipe()

- ➢ The pipe( ) system call creates a pipe that can be shared between processes
- ➢ It returns two file descriptors, one for reading from the pipe, the other for writing into the pipe.
- ➢ File descriptors (fd) is an integer that uniquely identifies an open file of the process.
- ➢ 0 for stdin, 1 for stdout, 2 for stderr.

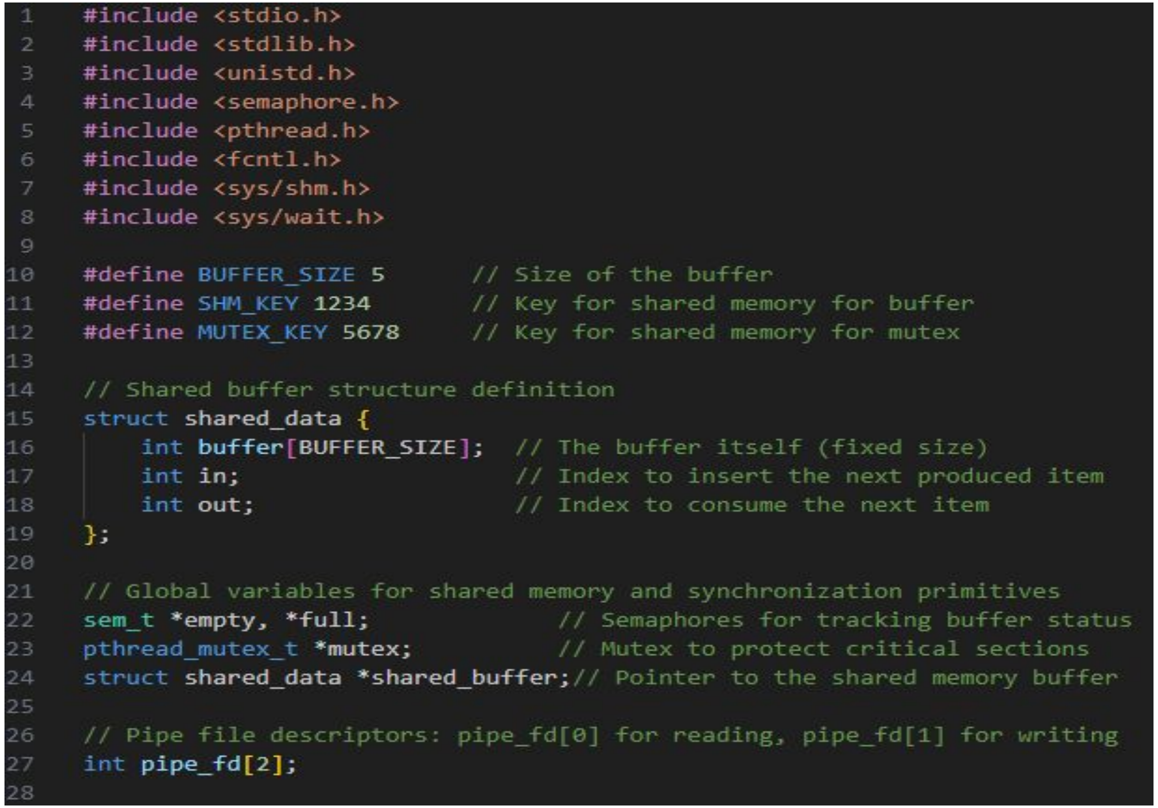
pipe()



```c
#include <stdio.h>
#include <unistd.h> /* Include this file to use pipes */
#define BUFSIZE 80
main()
{
int fd[2], n=0, i;
char line[BUFSIZE];
pipe(fd); /* fd[0] is for reading, fd[1] is for writing */if (fork() == 0) {
close(fd[0]); /* The child will not read */

for (i=0; i < 10; i++) {
sprintf(line,"%d",n);
write(fd[1], line, BUFSIZE);
printf("Child writes: %d\n",n); n++; sleep(2);
}}
else {
close(fd[1]); /* The parent will not write */
for (i=0; i < 10; i++) {
read(fd[0], line, BUFSIZE);
sscanf(line,"%d",&n);
printf("\t\t\t Parent reads: %d\n",n);
} }}
```

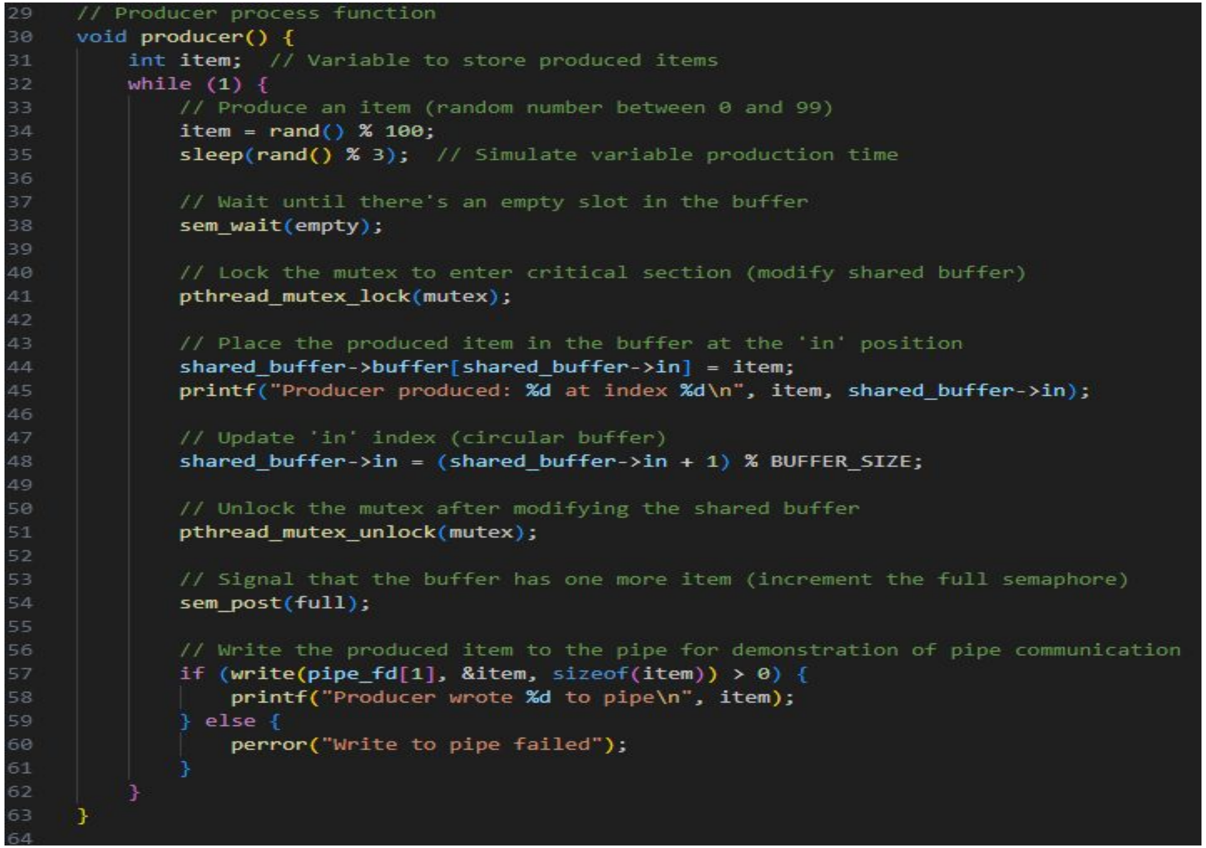
Producer Consumer Using Processes, Threads, Semaphores, Shared Processes

Producer Consumer - Processes



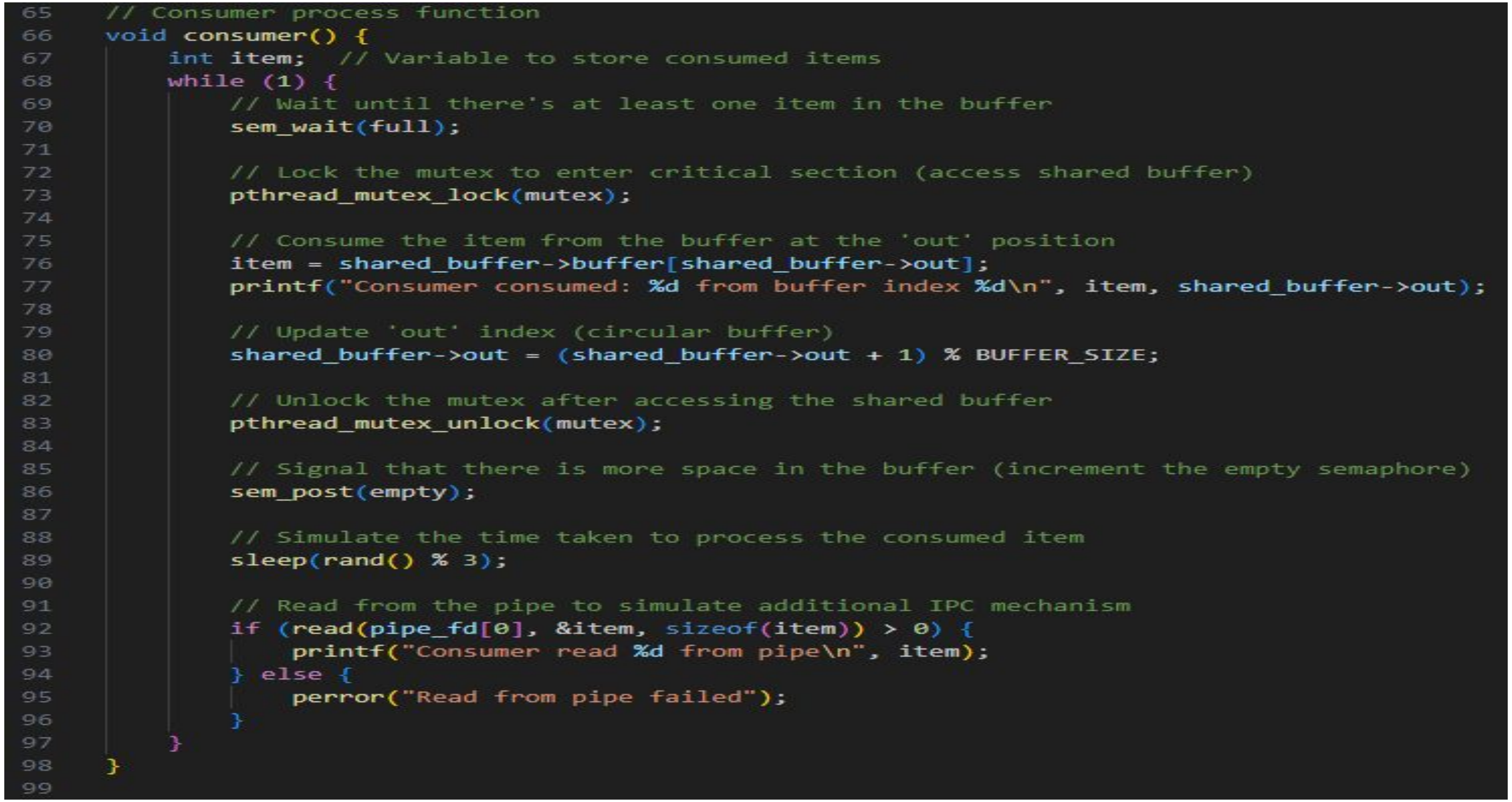
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <semaphore.h>
#include <pthread.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/wait.h>

#define BUFFER_SIZE 5      // Size of the buffer
#define SHM_KEY 1234       // Key for shared memory for buffer
#define MUTEX_KEY 5678     // Key for shared memory for mutex

// Shared buffer structure definition
struct shared_data {
    int buffer[BUFFER_SIZE];  // The buffer itself (fixed size)
    int in;                   // Index to insert the next produced item
    int out;                  // Index to consume the next item
};

// Global variables for shared memory and synchronization primitives
sem_t *empty, *full;             // Semaphores for tracking buffer status
pthread_mutex_t *mutex;          // Mutex to protect critical sections
struct shared_data *shared_buffer;// Pointer to the shared memory buffer

// Pipe file descriptors: pipe_fd[0] for reading, pipe_fd[1] for writing
int pipe_fd[2];

```

Producer Consumer - Processes



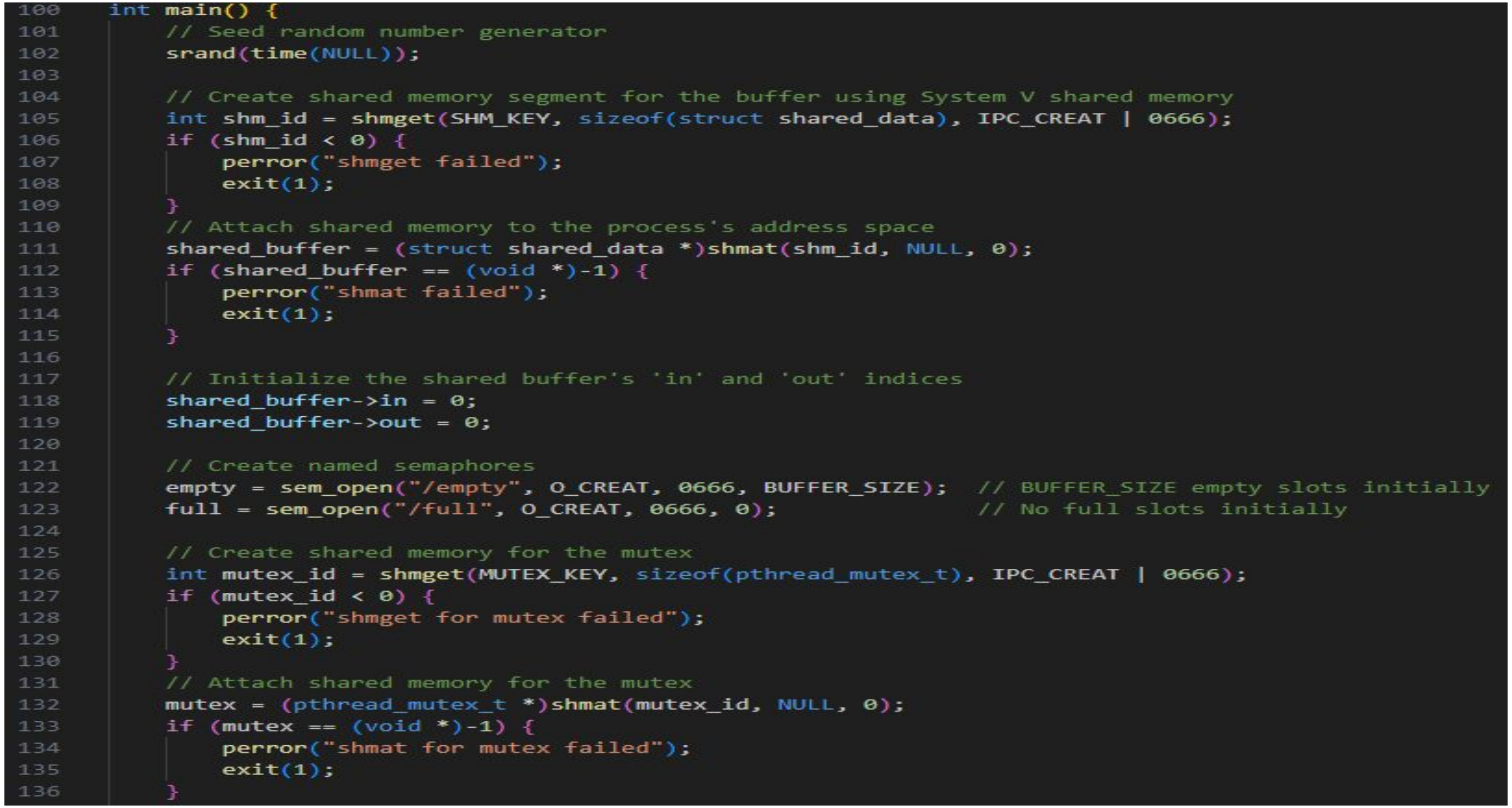
```c
// Producer process function
void producer() {
    int item;  // Variable to store produced items
    while (1) {
        // Produce an item (random number between 0 and 99)
        item = rand() % 100;
        sleep(rand() % 3);  // Simulate variable production time

        // Wait until there's an empty slot in the buffer
        sem_wait(empty);

        // Lock the mutex to enter critical section (modify shared buffer)
        pthread_mutex_lock(mutex);

        // Place the produced item in the buffer at the 'in' position
        shared_buffer->buffer[shared_buffer->in] = item;
        printf("Producer produced: %d at index %d\n", item, shared_buffer->in);

        // Update 'in' index (circular buffer)
        shared_buffer->in = (shared_buffer->in + 1) % BUFFER_SIZE;

        // Unlock the mutex after modifying the shared buffer
        pthread_mutex_unlock(mutex);

        // Signal that the buffer has one more item (increment the full semaphore)
        sem_post(full);

        // Write the produced item to the pipe for demonstration of pipe communication
        if (write(pipe_fd[1], &item, sizeof(item)) > 0) {
            printf("Producer wrote %d to pipe\n", item);
        } else {
            perror("Write to pipe failed");
        }
    }
}
```

Producer Consumer - Processes



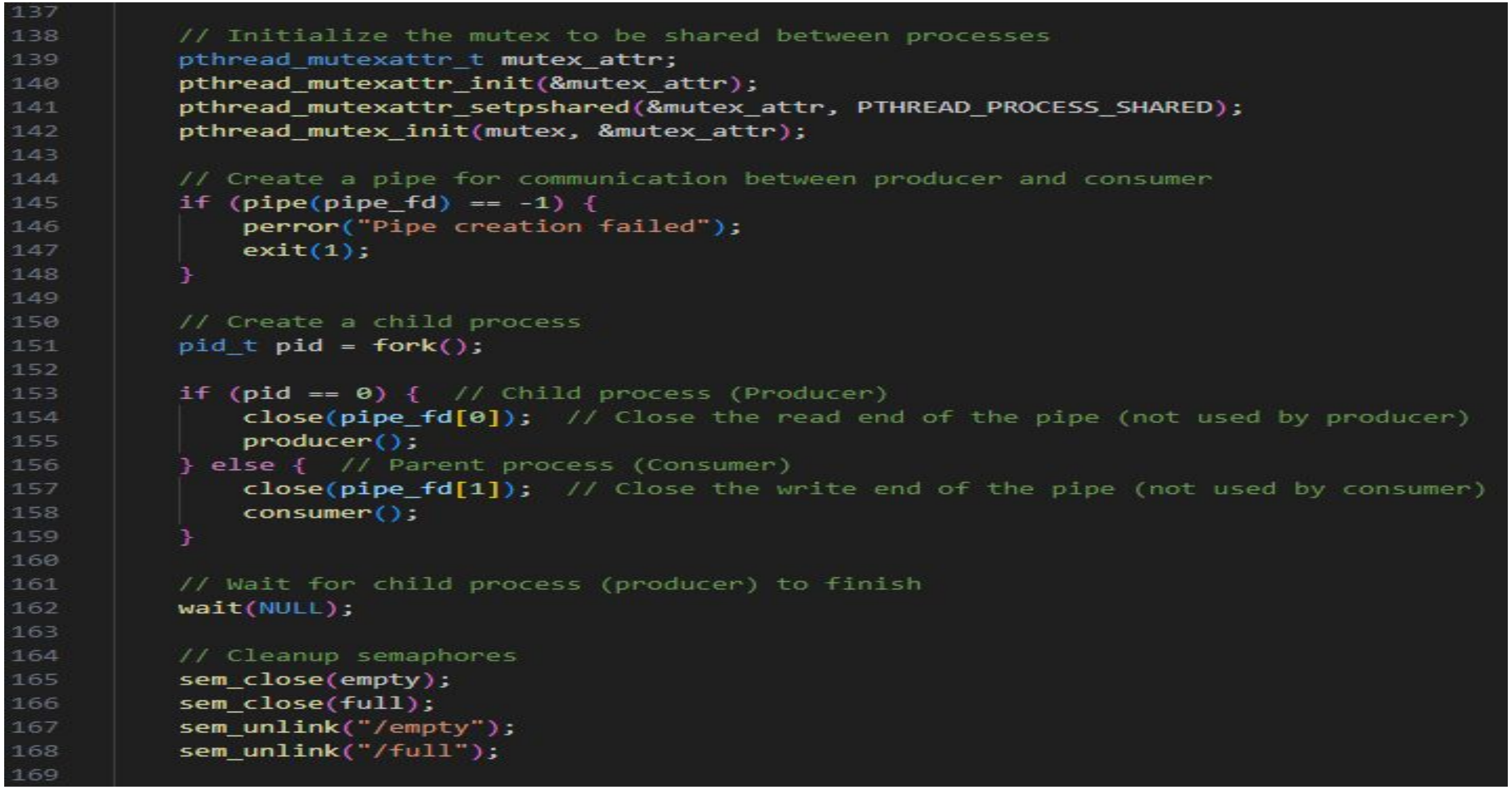
```c
// Consumer process function
void consumer() {
    int item;  // Variable to store consumed items
    while (1) {
        // Wait until there's at least one item in the buffer
        sem_wait(full);

        // Lock the mutex to enter critical section (access shared buffer)
        pthread_mutex_lock(mutex);

        // Consume the item from the buffer at the 'out' position
        item = shared_buffer->buffer[shared_buffer->out];
        printf("Consumer consumed: %d from buffer index %d\n", item, shared_buffer->out);

        // Update 'out' index (circular buffer)
        shared_buffer->out = (shared_buffer->out + 1) % BUFFER_SIZE;

        // Unlock the mutex after accessing the shared buffer
        pthread_mutex_unlock(mutex);

        // Signal that there is more space in the buffer (increment the empty semaphore)
        sem_post(empty);

        // Simulate the time taken to process the consumed item
        sleep(rand() % 3);

        // Read from the pipe to simulate additional IPC mechanism
        if (read(pipe_fd[0], &item, sizeof(item)) > 0) {
            printf("Consumer read %d from pipe\n", item);
        } else {
            perror("Read from pipe failed");
        }
    }
}

```

Producer Consumer - Processes



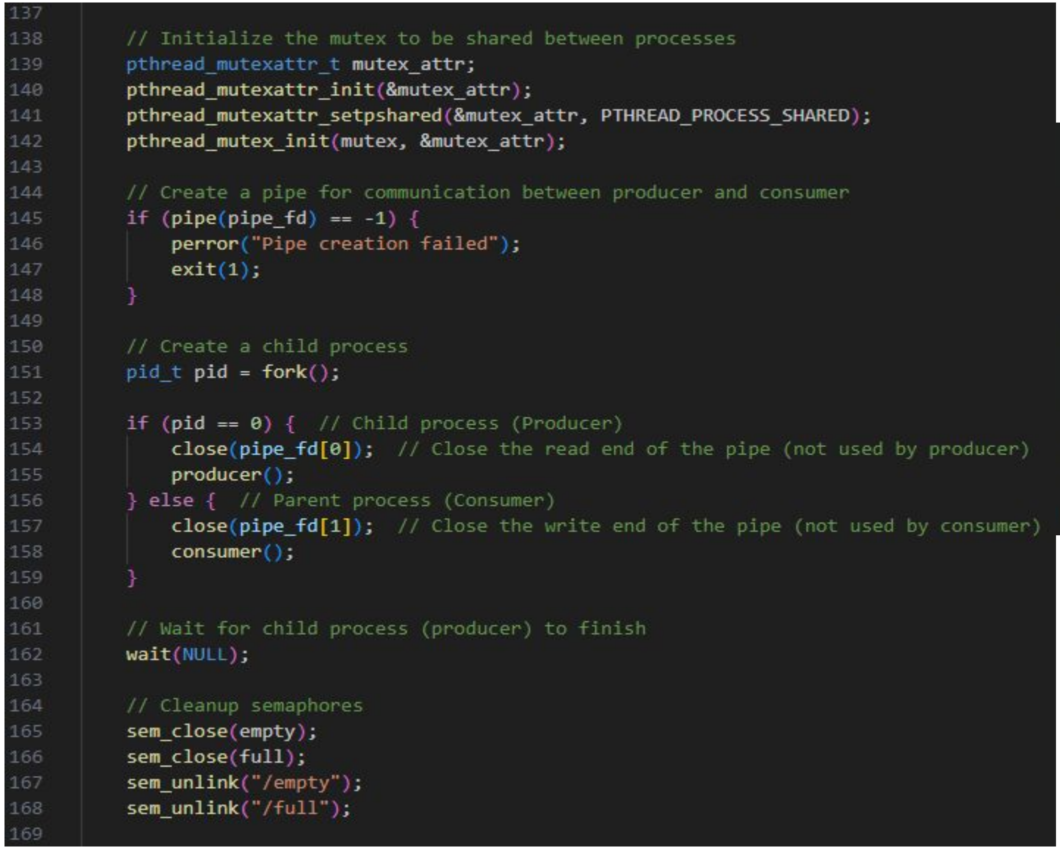
```c
int main() {
    // Seed random number generator
    srand(time(NULL));

    // Create shared memory segment for the buffer using System V shared memory
    int shm_id = shmget(SHM_KEY, sizeof(struct shared_data), IPC_CREAT | 0666);
    if (shm_id < 0) {
        perror("shmget failed");
        exit(1);
    }
    // Attach shared memory to the process's address space
    shared_buffer = (struct shared_data *)shmat(shm_id, NULL, 0);
    if (shared_buffer == (void *)-1) {
        perror("shmat failed");
        exit(1);
    }

    // Initialize the shared buffer's 'in' and 'out' indices
    shared_buffer->in = 0;
    shared_buffer->out = 0;

    // Create named semaphores
    empty = sem_open("/empty", O_CREAT, 0666, BUFFER_SIZE);  // BUFFER_SIZE empty slots initially
    full = sem_open("/full", O_CREAT, 0666, 0);              // No full slots initially

    // Create shared memory for the mutex
    int mutex_id = shmget(MUTEX_KEY, sizeof(pthread_mutex_t), IPC_CREAT | 0666);
    if (mutex_id < 0) {
        perror("shmget for mutex failed");
        exit(1);
    }
    // Attach shared memory for the mutex
    mutex = (pthread_mutex_t *)shmat(mutex_id, NULL, 0);
    if (mutex == (void *)-1) {
        perror("shmat for mutex failed");
        exit(1);
    }
```

Producer Consumer - Processes

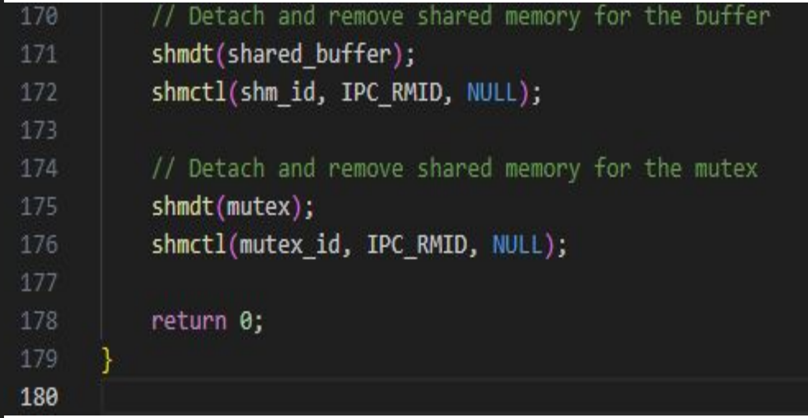


```c
    // Initialize the mutex to be shared between processes
    pthread_mutexattr_t mutex_attr;
    pthread_mutexattr_init(&mutex_attr);
    pthread_mutexattr_setpshared(&mutex_attr, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(mutex, &mutex_attr);

    // Create a pipe for communication between producer and consumer
    if (pipe(pipe_fd) == -1) {
        perror("Pipe creation failed");
        exit(1);
    }

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

    if (pid == 0) {  // Child process (Producer)
        close(pipe_fd[0]);  // Close the read end of the pipe (not used by producer)
        producer();
    } else {  // Parent process (Consumer)
        close(pipe_fd[1]);  // Close the write end of the pipe (not used by consumer)
        consumer();
    }

    // Wait for child process (producer) to finish
    wait(NULL);

    // Cleanup semaphores
    sem_close(empty);
    sem_close(full);
    sem_unlink("/empty");
    sem_unlink("/full");

```

Producer Consumer - Processes



```c

    // Initialize the mutex to be shared between processes
    pthread_mutexattr_t mutex_attr;
    pthread_mutexattr_init(&mutex_attr);
    pthread_mutexattr_setpshared(&mutex_attr, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(mutex, &mutex_attr);

    // Create a pipe for communication between producer and consumer
    if (pipe(pipe_fd) == -1) {
        perror("Pipe creation failed");
        exit(1);
    }

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

    if (pid == 0) {  // Child process (Producer)
        close(pipe_fd[0]);  // Close the read end of the pipe (not used by producer)
        producer();
    } else {  // Parent process (Consumer)
        close(pipe_fd[1]);  // Close the write end of the pipe (not used by consumer)
        consumer();
    }

    // Wait for child process (producer) to finish
    wait(NULL);

    // Cleanup semaphores
    sem_close(empty);
    sem_close(full);
    sem_unlink("/empty");
    sem_unlink("/full");

```



```c
    // Detach and remove shared memory for the buffer
    shmdt(shared_buffer);
    shmctl(shm_id, IPC_RMID, NULL);

    // Detach and remove shared memory for the mutex
    shmdt(mutex);
    shmctl(mutex_id, IPC_RMID, NULL);

    return 0;
}

```

Thank You