# Operating Systems

Linux Kernel / POSIX Threads & File I/O

**David O Neill**
**10/31/2009**

## Contents

## Example 1 - Implementing Kernel Threads on Linux

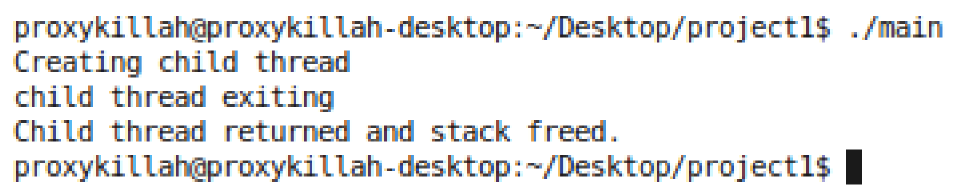
The main function runs is executed, **malloc()** allocates memory for the thread's stack. The main function then prints '**Creating child thread**' to the standard output.

The **clone()** system call creates a new thread who's entry point is the function signature '**threadFunction( void* argument )**'. The second argument is a pointer to a memory space to be used as the stack for the new thread. The third argument is the flags inherited by the parent process. The last argument is the argument passed to the threads function signature. On successful creation of the thread a **Process ID** is returned or -1 indicating failure.

The child thread then runs printing 'child thread exiting' to the standard output stream before returning. The parent thread during this time continues executing before the system call waitpid() is called. This suspends the parent process's execution until the child specific by a PID argument has changed state. The status of the thread can be stored in a pointer specified in the second argument. Options can be passed to the thread execution in the form of constants.
Finally after the child process (thread) has finished its memory is freed using **free()**. The parent process then prints '**Child thread returned and stack freed**' to the standard output.

If in the event of any errors, **perror()** is used to print errors the standard error stream.

*Fig. 1.1*



```
proxykillah@proxykillah-desktop:~/Desktop/project1$ ./main
Creating child thread
child thread exiting
Child thread returned and stack freed.
proxykillah@proxykillah-desktop:~/Desktop/project1$ 
```

## Example 1 - Extended

In the extended version we duplicate our work for **malloc()**, allocating two stacks (memory spaces), one for each thread. The parent prints '**Creating child threads**'.
Two child processes (**threads**) are then created. **Thread 1** executes incrementing the value of **x**. In this case it happens before **thread 2** has a chance to modify the data possibly causing a shared data inconsistency. This is because there is no mutual exclusion algorithm to prevent the simultaneous use of the shared resource x.

### Possible Problem

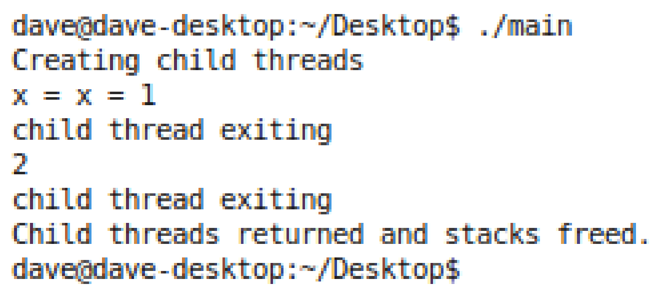
In the threads **critical section** namely the **atomic operation x++, this resource** can be read into a **CPU's register** and incremented in the **accumulator** while **Thread 2** is loading the value from the memory. Since the value in memory has not yet been updated by **Thread 1**, Thread **2** now has an old value of **X**. And the result of the **atomic operation** of **Thread 2 (x++)** will be same as **Thread 1's atomic operation** which has yet to be stored back into memory. Hence a shared data inconsistency.

### Test Case

In this cause however **thread 1** is not **pre-empted** by the **Kernel process scheduler dispatcher** and is allowed to perform its **atomic instruction**, saving the data back into memory before **Thread 2** begins its **atomic operation**. **Thread 2** performs its **atomic operation** incrementing **x** and stores **x** back into shared memory.

Once again we wait for these **2 threads** to complete using **waitpid()** before continuing the execution of the parent process. The two stacks are then freed using **free()**. The parent then prints '**Child threads returned and stacks freed.**' and exits.

*Fig. 1.2*

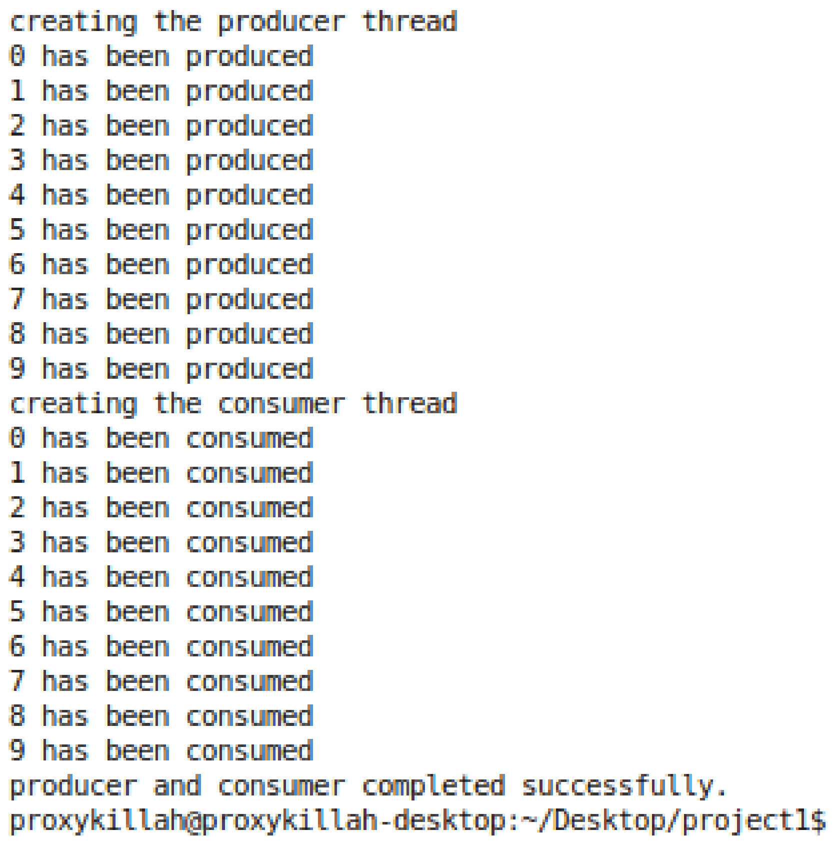


```
dave@dave-desktop:~/Desktop$ ./main
Creating child threads
x = x = 1
child thread exiting
2
child thread exiting
Child threads returned and stacks freed.
dave@dave-desktop:~/Desktop$
```

## Example 2 - Communicating via Shared Memory

In example 2 the producer produces 10 items before the consumer thread is allowed to begin. As waitpid() is called on producer before the consumer begins, the consumer does not consume until the producer has finished executing.

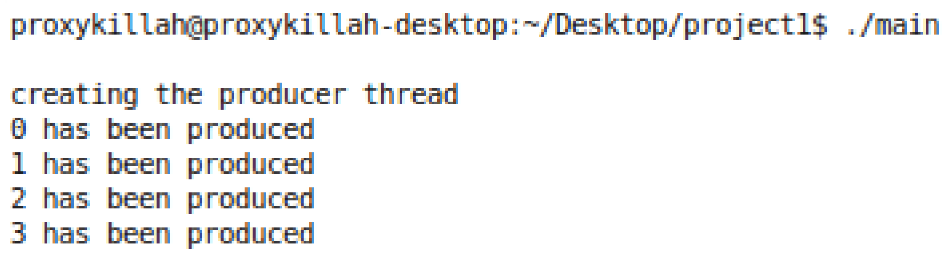
*Fig. 2.1*



```
creating the producer thread
0 has been produced
1 has been produced
2 has been produced
3 has been produced
4 has been produced
5 has been produced
6 has been produced
7 has been produced
8 has been produced
9 has been produced
creating the consumer thread
0 has been consumed
1 has been consumed
2 has been consumed
3 has been consumed
4 has been consumed
5 has been consumed
6 has been consumed
7 has been consumed
8 has been consumed
9 has been consumed
producer and consumer completed successfully.
proxykillah@proxykillah-desktop:~/Desktop/project1$
```

As the **buffer** array is **20** the **producer** does not reach a **deadlock situation**, where it is waiting for items to be consumed before it can produce more items.
Changing the buffer to size 5 illustrates this **deadlock situation.**

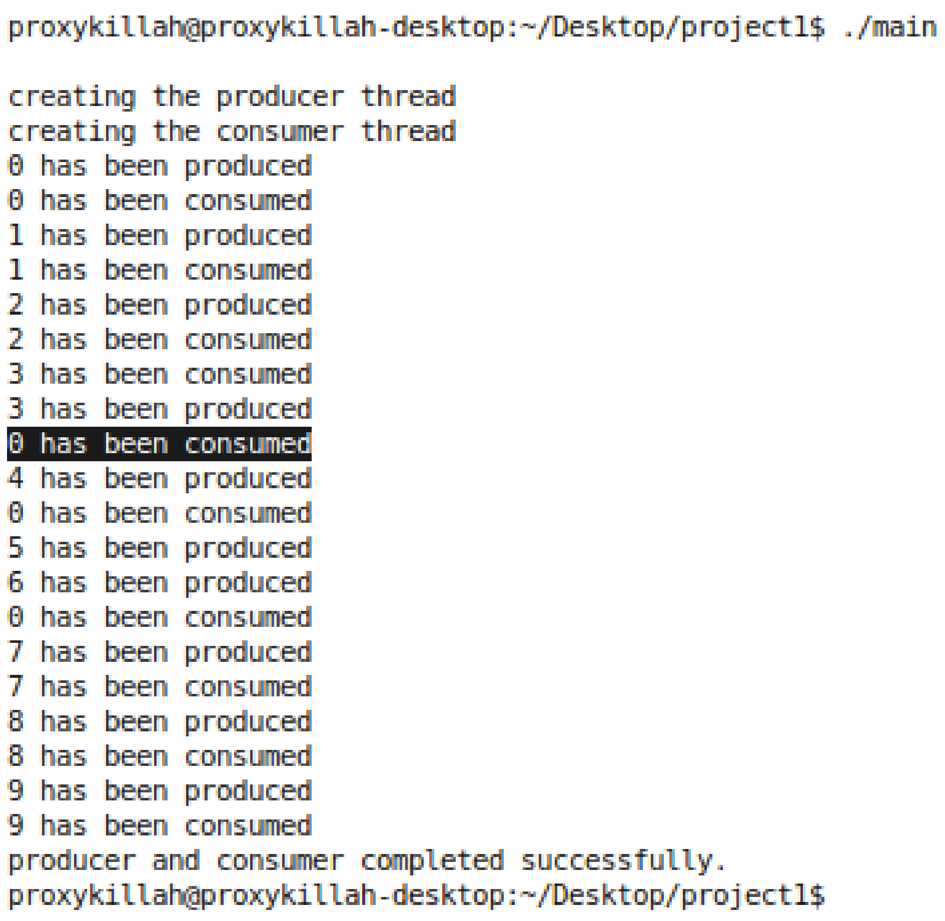
*Fig. 2.2*



```
proxykillah@proxykillah-desktop:~/Desktop/project1$ ./main

creating the producer thread
0 has been produced
1 has been produced
2 has been produced
3 has been produced
```

By removing the **waitpid( pid_producer, 0, 0 );** instruction after the thread creation and putting it after the **consumer thread** we now have multi-threaded **concurrent** execution of the **producer** and the **consumer**.

***Fig. 2.3***



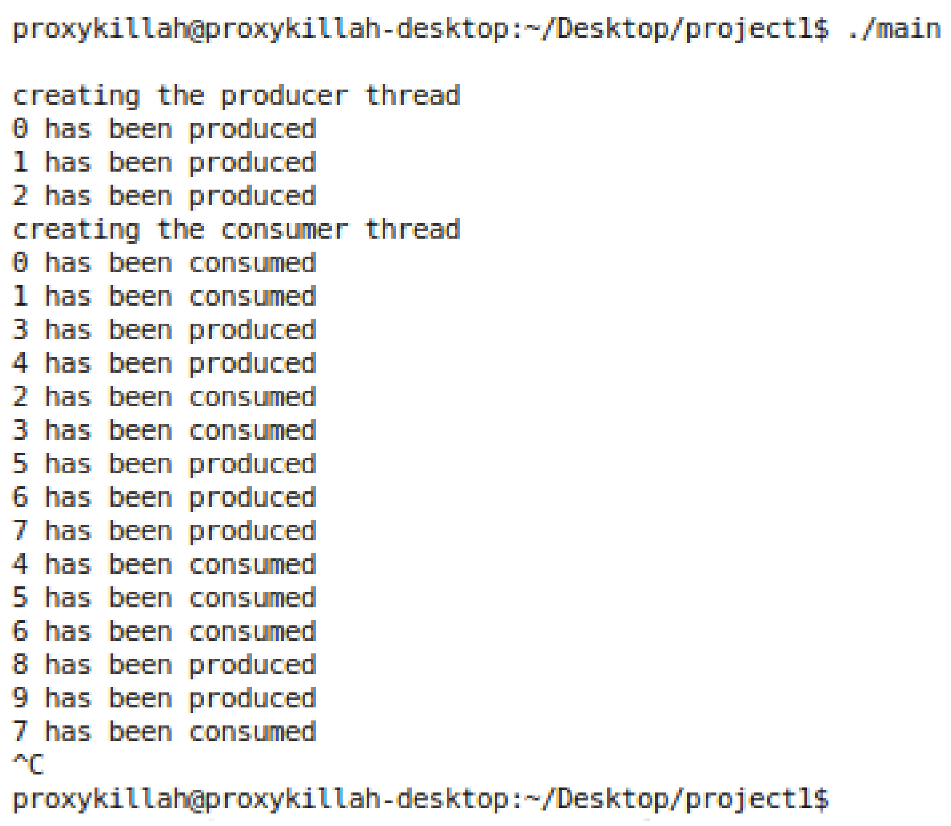
```
proxykillah@proxykillah-desktop:~/Desktop/project1$ ./main

creating the producer thread
creating the consumer thread
0 has been produced
0 has been consumed
1 has been produced
1 has been consumed
2 has been produced
2 has been consumed
3 has been consumed
3 has been produced
0 has been consumed
4 has been produced
0 has been consumed
5 has been produced
6 has been produced
0 has been consumed
7 has been produced
7 has been consumed
8 has been produced
8 has been consumed
9 has been produced
9 has been consumed
producer and consumer completed successfully.
proxykillah@proxykillah-desktop:~/Desktop/project1$
```

During the stages of the **consumer** and the **producer** lifetime, as **context switches** occur we can see non synchronization of the 2 threads in the producing and the consuming stages.
Because the buffer is of size twenty and the **producer** only producers 10 items we don't reach a **deadlock situation** where the **producer** is waiting for the **consumer** to consume an item based upon the shared variable **count (semaphore)**.
If we change the buffer size to **5** we can see this deadlock situation.

***See Fig. 2.2***

As **count (semaphore)** is **critical** to both the **consumer** and the **producer** and neither have **mutual exclusion** over this value. If the **producer / consumer** are **pre-empted** while working with **atomic operation count++ or count--** a shared data inconsistency can occur causing the **wait loops** in
**Producer :** while (count == BUFFER_SIZE); // wait
**Consumer :** while (count == 0); // wait
to cause a **deadlock situation**.
In this situation **the consumer** is in the **starvation** scenario (**indefinite blocking**) as the **semaphore count** is now zero and the **consumer** has not finished its consumption of 10 items.

*Fig. 2.4*



```
proxykillah@proxykillah-desktop:~/Desktop/project1$ ./main

creating the producer thread
0 has been produced
1 has been produced
2 has been produced
creating the consumer thread
0 has been consumed
1 has been consumed
3 has been produced
4 has been produced
2 has been consumed
3 has been consumed
5 has been produced
6 has been produced
7 has been produced
4 has been consumed
5 has been consumed
6 has been consumed
8 has been produced
9 has been produced
7 has been consumed
^C
proxykillah@proxykillah-desktop:~/Desktop/project1$
```

## Example 3 - Peterson's Algorithm

Running example 3 we can observe that again that there is no **mutual exclusion** over the **semaphore count.** This results in a **deadlock situation** again where the **consumer** is trying to consume but it is being blocked by '**if (count > 0)** '.

### The critical sections

The critical sections for the consumer and producer

*See Fig. 3.2 & Fig. 3.3 @ EOF*

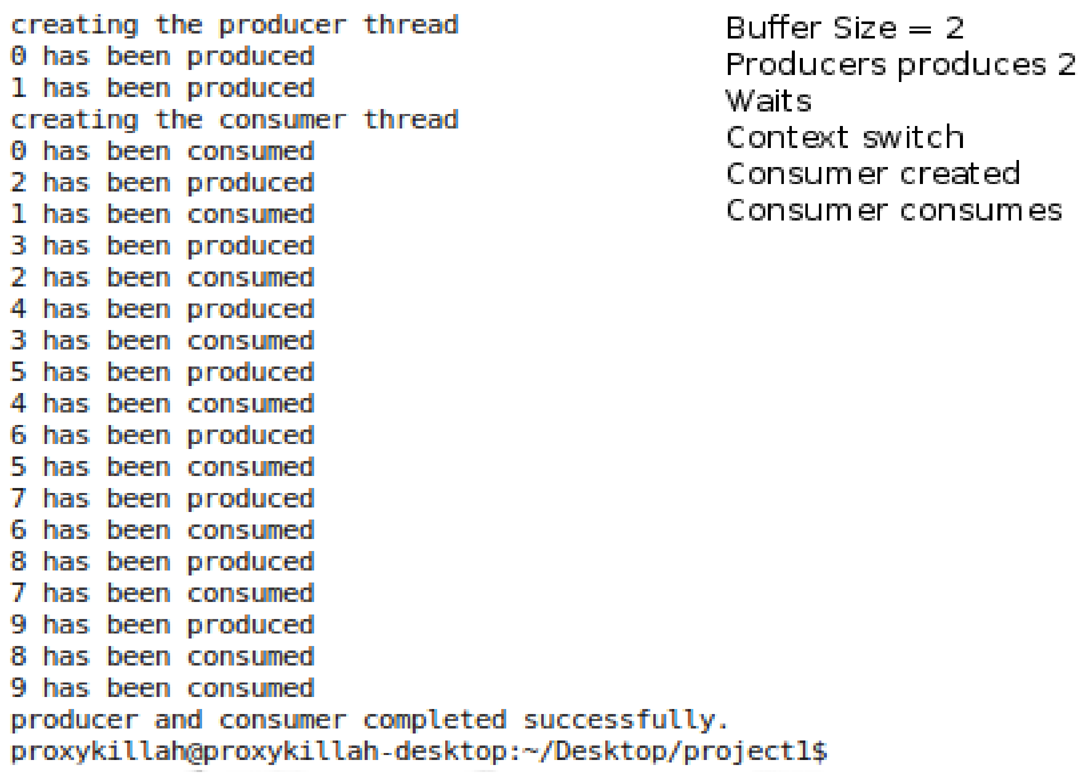
### Peterson's Algorithm

In order to implement Peterson's algorithm we need two shared variables flag and turn. We assume that the load and store for count++ and count-- are atomic instructions. Turn indicates whose turn it is to enter their critical section. The flag array is used to indicate if a process is ready to enter their critical section. Flag[P] = 1 implies a process is ready. Executing our modified code we can now see there is mutual exclusion over the semaphore count.

Before the Producer enters its critical section it sets the flag equal to 1 indicating it wants to enter its critical section. When it leaves its critical section it sets flag equal to 0 indicating it is finished with its critical section.
If the producer gets to the wait signal ' **while (flag[1] && turn==1);** ' it will wait here until the consumer is done with its critical section. The consumer sets its ready flag to 0 indicating it is done with its critical section. The producer will then break out of this waiting and continue with its critical section.

If a context switch then occurs during the producer's critical section the consumer will be waiting at its wait signal ' **while (flag[0] && turn==0);** ' until the producer exits its critical section setting its **flag = 0** indicating it is done.

*Fig. 3.1*



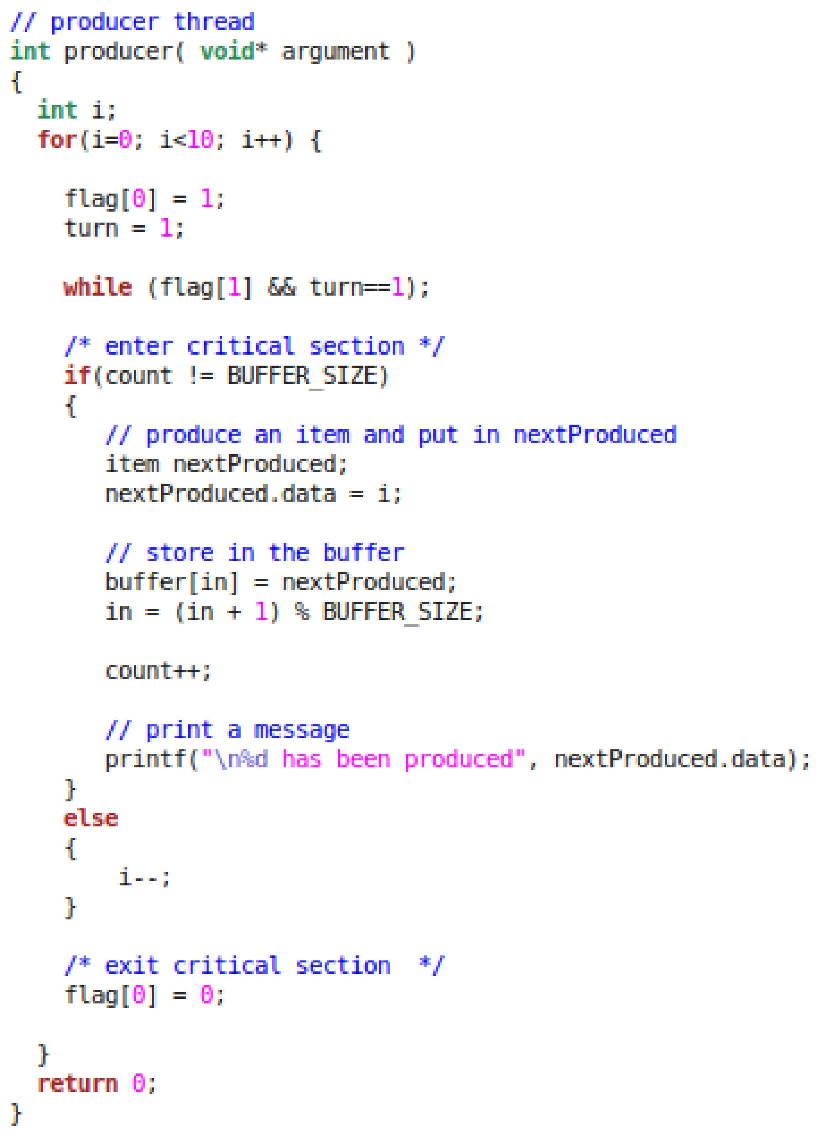
```
creating the producer thread
0 has been produced
1 has been produced
creating the consumer thread
0 has been consumed
2 has been produced
1 has been consumed
3 has been produced
2 has been consumed
4 has been produced
3 has been consumed
5 has been produced
4 has been consumed
6 has been produced
5 has been consumed
7 has been produced
6 has been consumed
8 has been produced
7 has been consumed
9 has been produced
8 has been consumed
9 has been consumed
producer and consumer completed successfully.
proxykillah@proxykillah-desktop:~/Desktop/project1$
```

Buffer Size = 2
Producers produces 2
Waits
Context switch
Consumer created
Consumer consumes

## Applying Peterson's Algorithm to Example 2

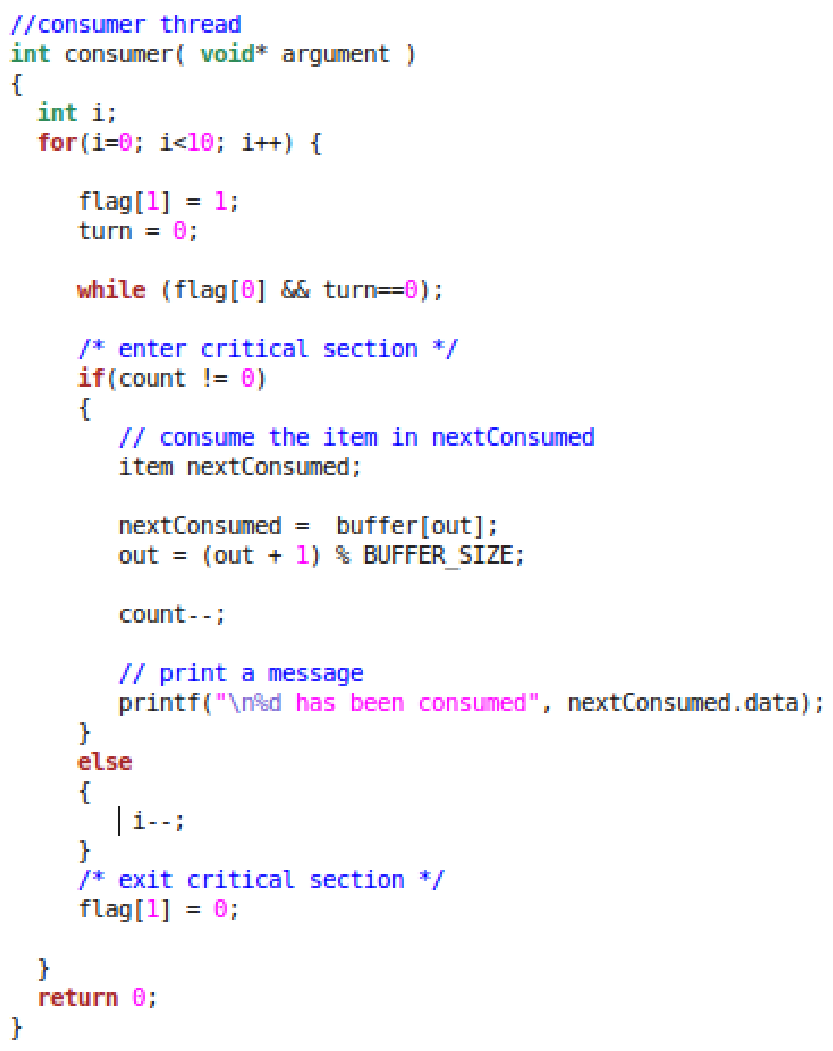
Firstly we identify the critical sections. Then apply the algorithm.

*Fig. 3.4 Producer*



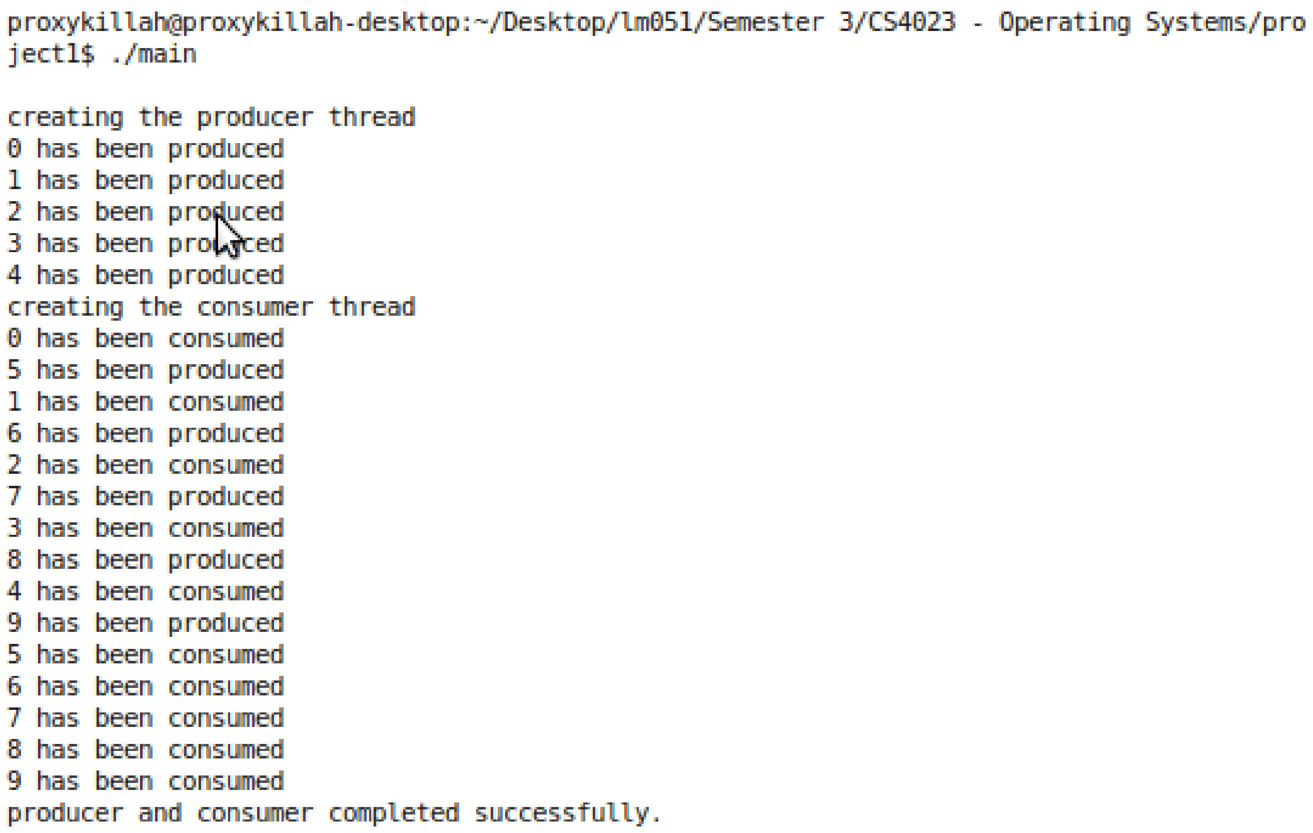
```
// producer thread
int producer( void* argument )
{
  int i;
  for(i=0; i<10; i++) {

    flag[0] = 1;
    turn = 1;

    while (flag[1] && turn==1);

    /* enter critical section */
    if(count != BUFFER_SIZE)
    {
       // produce an item and put in nextProduced
       item nextProduced;
       nextProduced.data = i;

       // store in the buffer
       buffer[in] = nextProduced;
       in = (in + 1) % BUFFER_SIZE;

       count++;

       // print a message
       printf("\n%d has been produced", nextProduced.data);
    }
    else
    {
        i--;
    }

    /* exit critical section  */
    flag[0] = 0;

  }
  return 0;
}
```

*Fig. 3.5 Consumer*



```
//consumer thread
int consumer( void* argument )
{
  int i;
  for(i=0; i<10; i++) {

     flag[1] = 1;
     turn = 0;

     while (flag[0] && turn==0);

     /* enter critical section */
     if(count != 0)
     {
        // consume the item in nextConsumed
        item nextConsumed;

        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        count--;

        // print a message
        printf("\n%d has been consumed", nextConsumed.data);
     }
     else
     {
         i--;
     }
     /* exit critical section */
     flag[1] = 0;

  }
  return 0;
}
```

*Fig. 3.6*

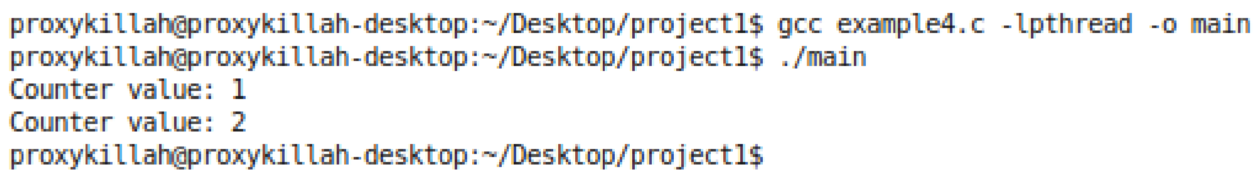


```
proxykillah@proxykillah-desktop:~/Desktop/lm051/Semester 3/CS4023 - Operating Systems/pro
ject1$ ./main

creating the producer thread
0 has been produced
1 has been produced
2 has been produced
3 has been produced
4 has been produced
creating the consumer thread
0 has been consumed
5 has been produced
1 has been consumed
6 has been produced
2 has been consumed
7 has been produced
3 has been consumed
8 has been produced
4 has been consumed
9 has been produced
5 has been consumed
6 has been consumed
7 has been consumed
8 has been consumed
9 has been consumed
producer and consumer completed successfully.
```

## Example 4 - POSIX Threads

Running example 4 we can observe that the 2 threads are created.  On entering the thread function each thread tried to obtain a mutual exclusion lock on the mutex mutex1.  Mutexes are used to prevent data inconsistencies due to race conditions and enabling successful thread synchronization.  Whichever thread gets the lock first will proceed into its critical section.  The other thread will wait till the mutex lock has been freed by the other.

*Fig. 4.1*



```
proxykillah@proxykillah-desktop:~/Desktop/project1$ gcc example4.c -lpthread -o main
proxykillah@proxykillah-desktop:~/Desktop/project1$ ./main
Counter value: 1
Counter value: 2
proxykillah@proxykillah-desktop:~/Desktop/project1$
```

### Critical section

The critical section of the thread block requests the shared value of counter and assigns it to a local variable newcounter.  This variable is then incremented, the thread is told to sleep for one second and the counter shared variable counter is then assigned the value of newcounter.  The thread prints the value of counter and then finally the mutual exclusion lock on the mutex mutex1 is freed.  Once the first thread has completed its critical section the mutex mutex1 is unlocked, the second thread will then be allowed to get a mutual exclusion lock on the mutex mutex1 and it will proceed into the critical section.

### pthread_create

The function signature pthread_create accepts four arguments.

**Thread** – Thread id or reference to (thread)

**Attributes** – constants which specify attributes of the threads execution states*

**Start routine** – a reference to the function to be executed as the threads entry point

**Args** – a pointer to an argument to send to the threads or a pointer to a data structure containing multiple arguments.

The return type of pthread_create is an integer.  On success, the identifier of the newly created thread is stored in the location pointed by the thread argument, and a 0 is returned. On error, a non-zero error code is returned.

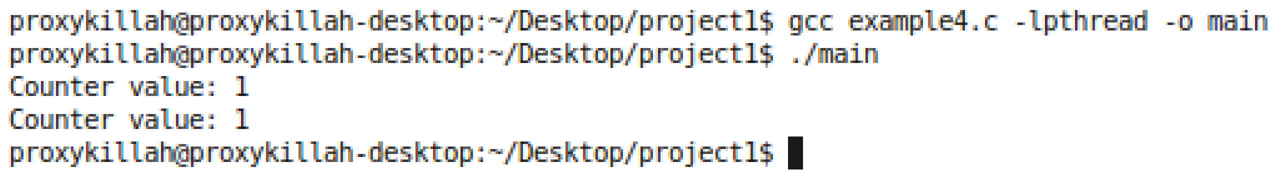
## pthread_join

Waits for a termination of another thread and accepts two arguments.  The thread to wait for and the return value of the thread.  If the thread returns 0 it means the threaded completed successfully.  On error a non zero value is returned.  In the test case a NULL value is specified in the second argument meaning that we don't care about its return value.  It suspends the calling thread until termination of the thread is completed whether through cancellation or termination.

## Removing the mutex

Removing the mutex mutex1 and pthread_mutex_lock() from the thread function entry point. We can observe a problem with the alteration after executing the code.  There is now no mutual exclusion on the shared variable counter.  Both threads will now race to get this counter followed by sleep(1) (sleep for one second),  As the Kernel **process scheduler dispatcher (context switch)** is supposed to have a response time of <1 second.  Thread one does not get the opportunity to store the new value of counter ( P' = P + 1), before the second thread loads the value of P.  So even if there was a delay on the creation of thread 2, we still could not guarantee that sleep(1) would finish waiting allowing P' to be stored back to memory.  As a context switch would most likely occur allowing thread 2 to load P it results in both threads storing the same value back to the shared memory.

*Fig. 4.2*



```
proxykillah@proxykillah-desktop:~/Desktop/project1$ gcc example4.c -lpthread -o main
proxykillah@proxykillah-desktop:~/Desktop/project1$ ./main
Counter value: 1
Counter value: 1
proxykillah@proxykillah-desktop:~/Desktop/project1$ 
```

# Example 5 - File Manipulation

## Open

// int open(FILE PATH,  ACCESS FLAGS, PERMISSION FLAGS );

The **open()** function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The **path** argument points to the file to open.  The **open()** function shall return a file descriptor for the named file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other process in the system.

The second argument  **status flags** and file access modes of the open file description shall be set according to the value of oflag.  Which specifies what functionality the open descriptor will have. Values in our exmaple are **O_RDONLY** – Read only.

The last argument of the open method is the mode, it sets the access permissions of the file  that permit reading and writing by the owner, and to permit reading only by group members and others.  In the case of opening a current  file however we are not setting the permissions.  So we specifiy 0 as the mode.  The operating system then when reading allows the opening of the file given the previously set permissions on the file.

In the case of creating a file using **open()**  we specify flags that tell the kernel to create a file.  For example. **O_WRONLY** for write only,  **O_CREAT** to create the file,  **O_TRUNC** truncate the file to 0 length and **O_EXCL** to prevent the over writing of an existing file.  Since were creating a file, we should specify access permissions, in this case we use **S_IRUSR** bit to allow read by the user, **S_IWUSR**  to allow writing to by the owner,  **S_IRGRP** to allow reading by the users group and **S_IROTH** to allow reading by others.

On error of opening a file **open()** will return a negative int.  We can then print the error using **perror()** which maps an error number to a predefined langauge independant error message.

## Read

// ssize_t read(FILE DESCRIPTOR, *BUF, N);

The **read()** function shall attempt to read **N** bytes from the file associated with the open file descriptor into the buffer pointed to by **buf** the second argument. The third argument **N** is the amount of bytes to read into the buffer. The return of read will be either the **Char**,** or 0 meaning End of File reached. Any other type of return in this case is considered an error. The exact error message is printed using **perror()** which maps an error number to a predefined language independent error message.

After a successful read, the next read position will be offset using the file descriptor. The offset will be incremented by a number of **N** bytes last read.

## Write

// ssize_t write(FILE DESCRIPTOR, *BUF, N);

The write() function shall attempt to write **N** bytes from the buffer pointed to by **buf** to the file associated with an open file descriptor with write mode specified.

If **N** is zero, the **write()** function may detect and return errors. In the absence of errors, or if error detection is not performed, the **write()** function shall return zero and have no other results.

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with the open file descriptor. Before successful return from **write()**, the file offset shall be incremented by the number of bytes actually written. On the file, if the file offset is greater than the length of the file, the length of the file shall be set to this file offset.

## Close

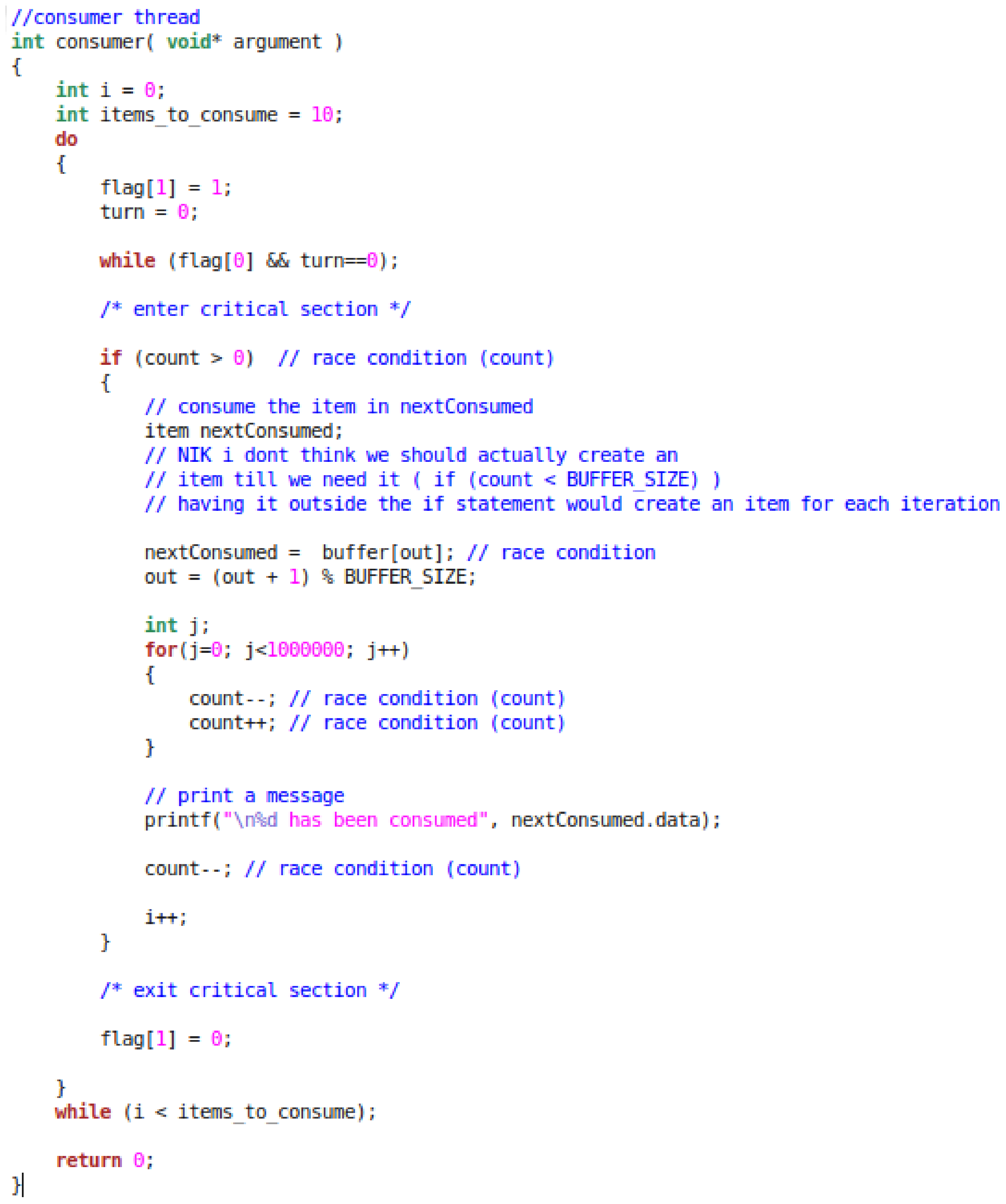
// int close(FILE DESCRIPTOR);

The **close()** function deallocates a file descriptor. All outstanding record locks owned by the process on the file associated with the file descriptor shall be removed (that is, unlocked).

If **close()** is interrupted by a signal that is to be caught, it shall return -1. The exact error message can then be printed using **perror()**. When all file descriptors associated with a file are closed, any data remaining data waiting to be read / written will be discarded.

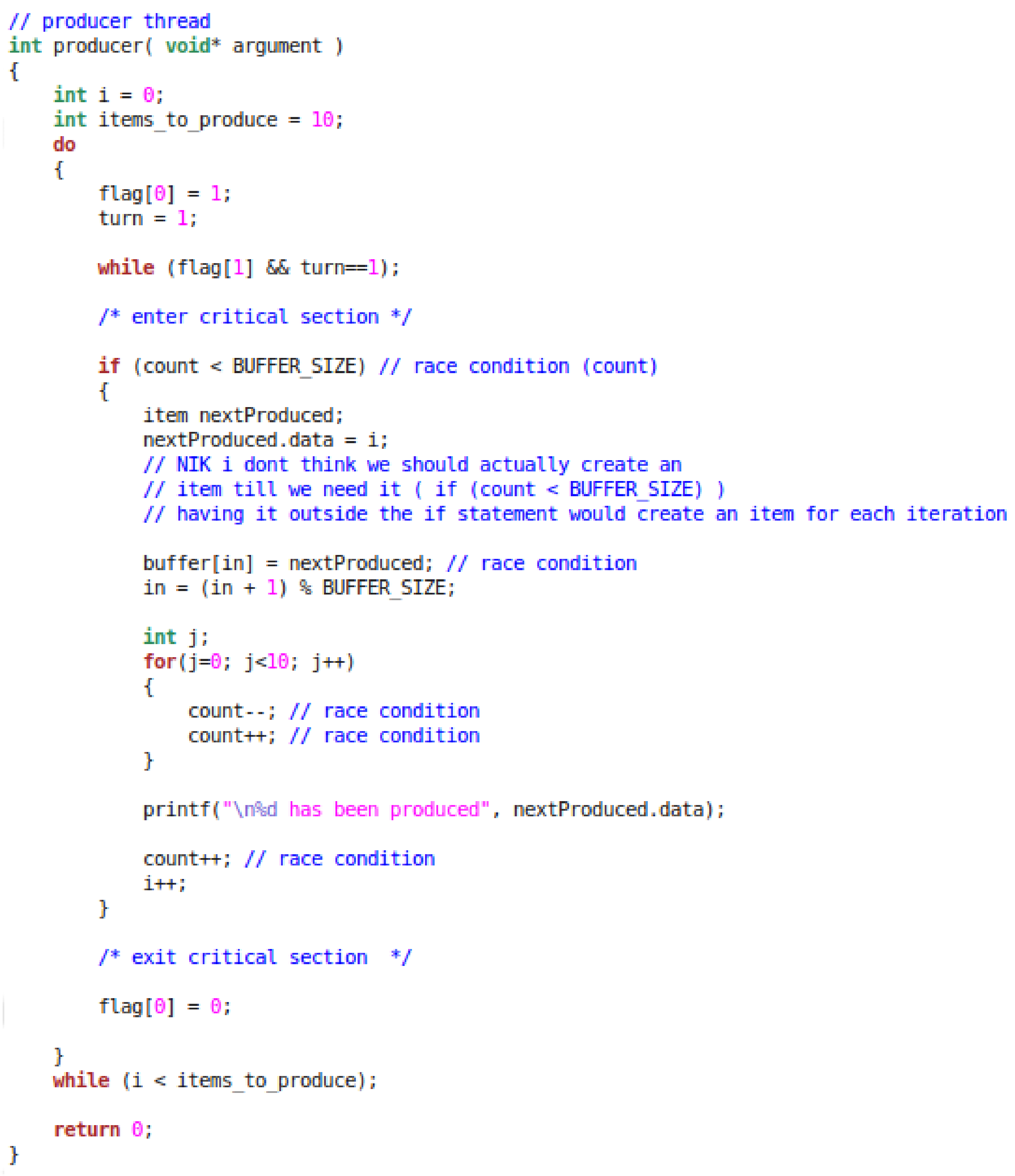
When all file descriptors associated with an open file description have been closed, the open file description shall be freed.

***See Fig. 5.1 & Fig. 5.2 @ EOF***



```
//consumer thread
int consumer( void* argument )
{
    int i = 0;
    int items_to_consume = 10;
    do
    {
        flag[1] = 1;
        turn = 0;

        while (flag[0] && turn==0);

        /* enter critical section */

        if (count > 0)  // race condition (count)
        {
            // consume the item in nextConsumed
            item nextConsumed;
            // NIK i dont think we should actually create an
            // item till we need it ( if (count < BUFFER_SIZE) )
            // having it outside the if statement would create an item for each iteration

            nextConsumed =  buffer[out]; // race condition
            out = (out + 1) % BUFFER_SIZE;

            int j;
            for(j=0; j<1000000; j++)
            {
                count--; // race condition (count)
                count++; // race condition (count)
            }

            // print a message
            printf("\n%d has been consumed", nextConsumed.data);

            count--; // race condition (count)

            i++;
        }

        /* exit critical section */

        flag[1] = 0;

    }
    while (i < items_to_consume);

    return 0;
}
```

**Figure 3.2**

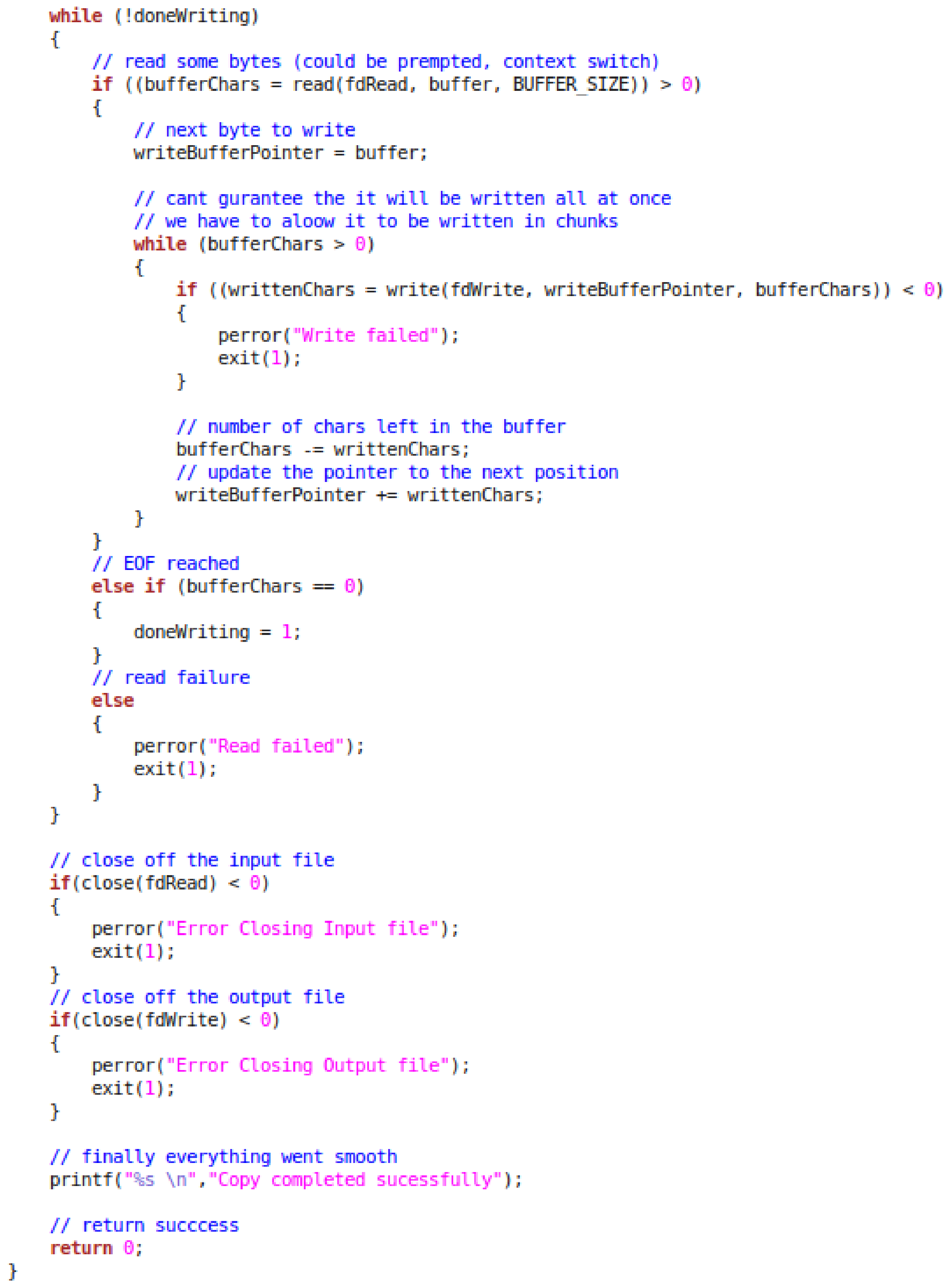


```
// producer thread
int producer( void* argument )
{
    int i = 0;
    int items_to_produce = 10;
    do
    {
        flag[0] = 1;
        turn = 1;

        while (flag[1] && turn==1);

        /* enter critical section */

        if (count < BUFFER_SIZE) // race condition (count)
        {
            item nextProduced;
            nextProduced.data = i;
            // NIK i dont think we should actually create an
            // item till we need it ( if (count < BUFFER_SIZE) )
            // having it outside the if statement would create an item for each iteration

            buffer[in] = nextProduced; // race condition
            in = (in + 1) % BUFFER_SIZE;

            int j;
            for(j=0; j<10; j++)
            {
                count--; // race condition
                count++; // race condition
            }

            printf("\n%d has been produced", nextProduced.data);

            count++; // race condition
            i++;
        }

        /* exit critical section  */

        flag[0] = 0;

    }
    while (i < items_to_produce);

    return 0;
}
```

**Figure 3.3**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

int main(int argc, char **argv)
{
    // file input descriptor
    int fdRead;
    // file output descriptor
    int fdWrite;
    // Read/Write buffer.
    char buffer[BUFFER_SIZE];
    // pointer to the write buffer
    char *writeBufferPointer;
    // number of bytes remaining to be written
    int bufferChars;
    // bytes written on last right
    int writtenChars;

    // allocate memory for the file input name
    char* fileInput = (char *) malloc(256);
    // allocate memory for the file output name
    char* fileOutput = (char *) malloc(256);

    if(argc!=3)
    {
        // arguments weren't specified correctly
        printf ("%s","Please specify the input file : ");
        scanf ("%s",fileInput);
        printf ("%s", "Please specify the output file name : ");
        scanf ("%s",fileOutput);
    }
    else
    {
        // use arguments
        fileInput = argv[1];
        fileOutput = argv[2];
    }

    // open file to be copied
    if ((fdRead = open(fileInput, O_RDONLY, 0)) < 0)
    {
        perror("Open source file failed");
        exit(1);
    }

    // open file to be copied to (check exists, apply permissions)
    if ((fdWrite = open(fileOutput, O_WRONLY | O_CREAT | O_TRUNC | O_EXCL,
                                    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) < 0)
    {
        perror("Open destination file failed");
        exit(1);
    }

    int doneWriting = 0;
```

**Fig 5.1**



```
    while (!doneWriting)
    {
        // read some bytes (could be prempted, context switch)
        if ((bufferChars = read(fdRead, buffer, BUFFER_SIZE)) > 0)
        {
            // next byte to write
            writeBufferPointer = buffer;

            // cant gurantee the it will be written all at once
            // we have to aloow it to be written in chunks
            while (bufferChars > 0)
            {
                if ((writtenChars = write(fdWrite, writeBufferPointer, bufferChars)) < 0)
                {
                    perror("Write failed");
                    exit(1);
                }

                // number of chars left in the buffer
                bufferChars -= writtenChars;
                // update the pointer to the next position
                writeBufferPointer += writtenChars;
            }
        }
        // EOF reached
        else if (bufferChars == 0)
        {
            doneWriting = 1;
        }
        // read failure
        else
        {
            perror("Read failed");
            exit(1);
        }
    }

    // close off the input file
    if(close(fdRead) < 0)
    {
        perror("Error Closing Input file");
        exit(1);
    }
    // close off the output file
    if(close(fdWrite) < 0)
    {
        perror("Error Closing Output file");
        exit(1);
    }

    // finally everything went smooth
    printf("%s \n","Copy completed sucessfully");

    // return succcess
    return 0;
}
```

**Fig 5.2**