Mini-Project

The Sandwich Problem

Course Title: Operating Systems

Course Code :CSE325

Section:01

Group No.: 06

Submitted to:-

Dr. Md. Nawab Yousuf Ali
Professor
Department of Computer Science Engineering

Submitted by:-

1.Sujana Islam Smrity
St. ID:2020-2-60-061
2.Mim Bin Hossain
St. ID: 2021-1-60-071
3.Rahma Mahbuba Adithi
St. ID:2020-2-60-
4.Sadia Islam
2020-1-60-167
Department of Computer Science Engineering

The Sandwich Problem

Project Description:

There are four students sitting in a room: three juniors and one senior. Each of the juniors will make a sandwich and eat it. To make a sandwich requires bread, cheese, and sausage. Each junior has one of the three items, that is, one has bread, another has cheese and the third has sausage. The senior has infinite supply of all three items. The senior places two of the three items on the table, and the junior that has the third item, makes the sandwich. However, the other two juniors cannot make a sandwich because they do not have the third required item. Thus, a junior can make a sandwich only when all the three items are available to him. Implement a synchronization method to solve the problem.

Project Analysis:

- Three juniors and one senior.
- Total Items 3. Bread, Cheese, and Sausage.
- Senior who has infinite items always places any two items in the table.
- Each of the juniors will make a sandwich who has third item.
- The other two juniors cannot make a sandwich if they do not have the third required item.

Abstruct:

In our following project, we will implement and test a solution for the IPC (Inter Process Communication) problem of T\the sandwich problem in which we will use semaphore and multi threads to execute our code in order to have a synchronization for sandwich making process. In this program, we will mainly focus on two thread functions (Producer(),Consumer()). These functions are related to each other with semaphore and created by threads. The following code will define some variables at the beginning of the program. But we can change it according to one's choice.

Introduction:

An operating system (OS) is software that manages computer hardware and software resources while also providing common functions to computer programs. Time-sharing operating systems plan tasks to make the most of the system's resources, and they may also contain accounting software for cost allocation of processor time, storage, printing, and other resources. Although application code is usually executed directly by the hardware and frequently makes system calls to an OS function or is interrupted by it, the operating system acts as an intermediary between programs and the computer hardware for hardware functions such as input and output and memory

allocation. From cellular phones and video game consoles to web servers and supercomputers, operating systems are found on many devices that incorporate a computer.

In our “The sandwich problem”, we will focus on the three main topics. Threads, process and semaphore.

Threads: Within a process, a thread is a path of execution. Multiple threads can exist in a process. The lightweight process is also known as a thread. By dividing a process into numerous threads, parallelism can be achieved. Multiple tabs in a browser, for example, can represent different threads. MS Word makes use of numerous threads: one to format the text, another to receive inputs, and so on. Below are some more advantages of multithreading.

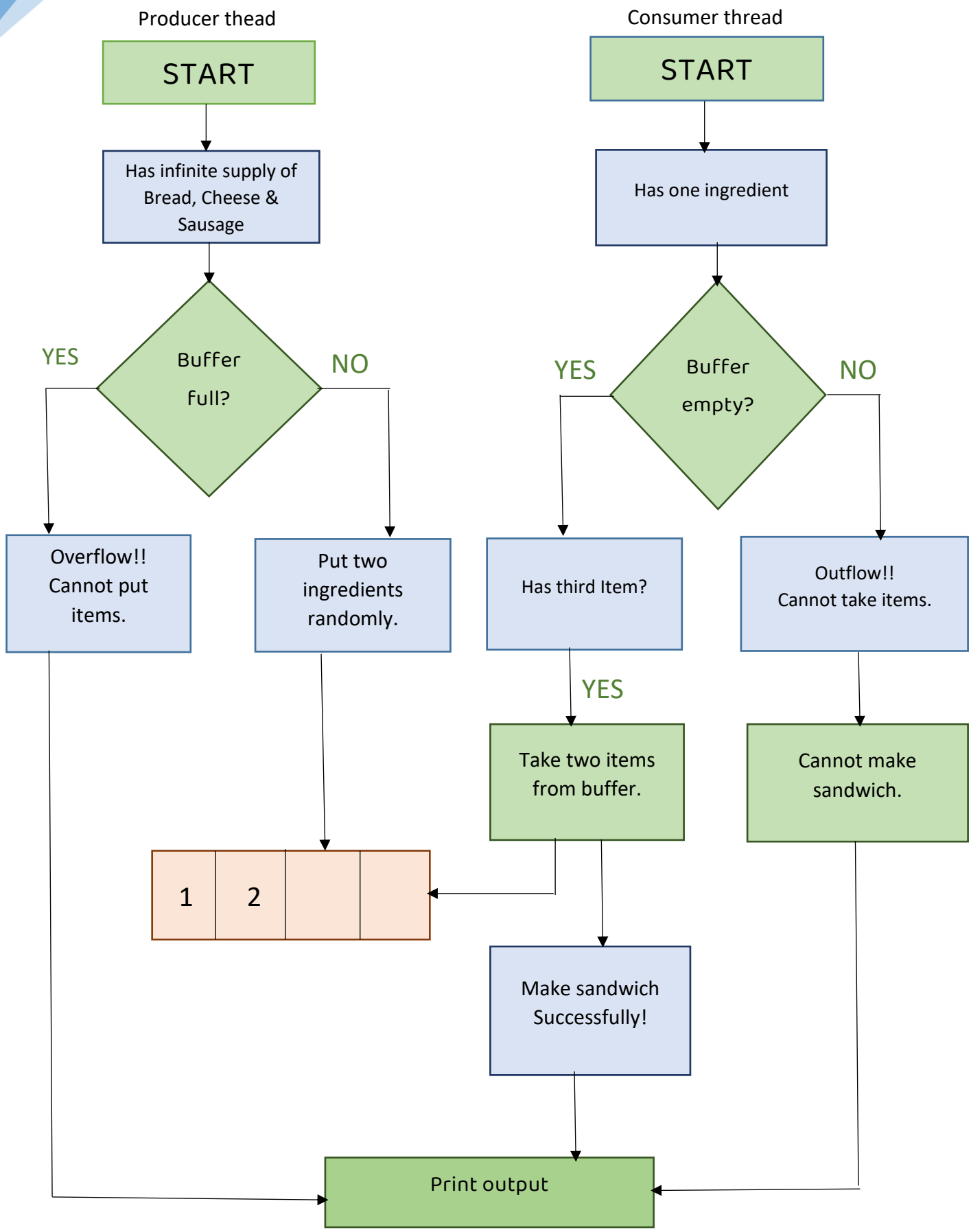
Process: A process is essential for running software. The execution of a process must be done in a specific order. To put it another way, we write our computer programs in a text file, and when we run them, they turn into a process that completes all of the duties specified in the program. A program can be separated into four components when it is put into memory and becomes a process: stack, heap, text, and data. The diagram below depicts a simplified structure of a process in main memory.

Semaphore: Dijkstra proposed the semaphore in 1965, which is a very important technique for managing concurrent activities using a basic integer value called a semaphore. A semaphore is just an integer variable shared by many threads. In a multiprocessing context, this variable is utilized to solve the critical section problem and establish process synchronization.

There are two types of semaphores:

1. **Binary Semaphore –**
   This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
2. **Counting Semaphore –**
   Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.
   .

FLOW CHART


Producer thead
START
Has infinite supply of Bread, Cheese & Sausage
Buffer full?
YES
NO
Overflow!! Cannot put items.
Put two ingredients randomly.
Consumer thread
START
Has one ingredient
Buffer empty?
YES
NO
Has third Item?
YES
Take two items from buffer.
Outflow!! Cannot take items.
Cannot make sandwich.
1
2
Make sandwich Successfully!
Print output

Main Code:

1.There are some important header files in this code for semaphore, threads, piping and exec functions. Here I define some integers value and semaphore and threads. For 3 items I define a **MaxItems = 3.** As senior randomly places two items I define a buffer: **BufferSize= 2** For thread synchronization I use semaphore where**:**

- **semEmpty** is Number of empty slots in buffer.
- **semFull** is Number of filled slots in buffer.
- **mutex** use for locking the thread before entering the critical section

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <semaphore.h>
#define MaxItems 3
#define BufferSize 2
sem_t semEmpty; //Number of empty slots in buffer
sem_t semFull; //Number of filled slots in buffer
sem_t mutex;

int buffer[BufferSize];
int in = 0;
int out =0 ;
int item ;
int i ,temp =-1 ;
int count=0;
int p1[2]; //pipe declare
char w1_number[30] ;
```

2.Here I use 2 functions for piping. Where Senior will write message and junior will read it.

```
void write_messege(){
close (p1[0]); //pipe() writing
printf ("\nSenior send messeage : Take two items!\n");
//fgets (w1_number,30,stdin);
write (p1[1],w1_number,strlen(w1_number));
close (p1[1]);
}
void read_messege(){
if (pipe(p1)!=-1){
close (p1[1]);
read (p1[0],w1_number,30);
printf ("\nJuniors recived message:Take two items!\n");
close (p1[0]);
}
```

3. Here I use 2 functions**: senior give item ()** and **junior take item ().** In senior give item () function senior will randomly produce 2 items among cheese, bread, and sausage. I also use rand () function for random number. Also give some conditions for produce 2 items**.** I use **count** variable here for junior take item () function. Count variable will add the 2 numbers according to the random items. And check the conditions which one will be the third item.

```
Void senior_give_item ()
{
printf (“\nSenior produced Two items :\n”);
srand(time(NULL));
for (i=0;i<2;i++)
{
item = rand() % MaxItems;
if (temp == item)
{
i--;
continue; }
```

```
if (item == 0 ){
printf (“Bread\n”);
}
else if (item ==1 ){
printf (“cheese\n”);
}
else if (item ==2){
printf (“Sausage\n”);
}
count =count+item ;
temp = item;
} }
void junior_take_item()
{
printf (“\nThird Item is :\n”);
if (count==1){
printf(“’Sausage’”);
}
else if (count==2){
printf(“’Cheese’”);
}
else if (count==3){
printf(“’Bread’”);
}
printf (“\n”);
}
```

4. In this part, there are two thread functions: **producer ()** , **consumer().**

```
Void* producer() //senior produced 2 items
{
sem_wait(&semEmpty);
sem_wait(&mutex); //semaphore value down (1=0)
//<critical section>
buffer[in] = item;
senior_give_item ();
write_messege();
in = (in+1)%BufferSize;
sem_post(&mutex); //semaphore value up (0=1)
sem_post(&semFull);
}
```

In **producer()** function –

- The semaphore **semEmpty** will decrease the buffer size from 2 to 1, 1 to 0. Because when senior produced items the empty slots will be filled up and decreased the empty space.
- The semaphore **mutex** will decrease the semaphore value from 1 to 0 and lock the threads and go to the critical section.
- In critical section we call those 2 functions: senior_give_item() and write_messege() because they were sharing global variable .
- The semaphore **semFull** will increased from 0 to 1 , 1 to 2 . Because when senior produced items the empty slots will be filled up and increased the filled space.
- Then again **mutex** will unlock the thread by increasing its value from 0 to 1.

```
void* consumer() //junior consumed 2 items
{
sem_wait(&semFull);
sem_wait(&mutex);
//<Critical Section>
int item = buffer[out];
read_messege();
junior_take_item();
out = (out+1)%BufferSize;
sem_post(&mutex);
sem_post(&semEmpty);
}
```

In **Consumer()** function-

- The semaphore **semFull** will deccreased from 2 to 0. Because when junior consumed 2 items the filled slots will be empty and decreased the filled space.
- The semaphore **mutex** will decrease the semaphore value from 1 to 0 and lock the threads and go to the critical section.
- In critical section we call those 2 functions: read_messege() and junior_take_item() because they were sharing global variable .
- The semaphore **semEmpty** will increased from 0 to 1, 1 to 2 . Because when junior consumed items the filled slots will be empty and increased the empty space.
- Then again **mutex** will unlock the thread by increasing its value from 0 to 1.

5. In main function we just initialized, joined (Main will waiting for threads) and destroy/exit the semaphores ,threads and also call the execv functions .

```
int main(int argc ,char *argv[])
{
pthread_t pro ,con ;
sem_init(&semEmpty, 0, 2);
sem_init(&semFull, 0, 0);
sem_init(&mutex , 0, 1);
pthread_create(&pro, NULL, &producer, NULL) ;
pthread_create(&con, NULL, &consumer, NULL) ;
pthread_join(pro, NULL);
pthread_join(con, NULL);
execv("./exec",argv);
sem_destroy(&semEmpty);
sem_destroy(&semFull);
sem_destroy(&mutex);
pthread_exit(NULL); //terminates the calling thread
return 0;
```

Exec file:

6. Here we use fork. In parent process we print “Sandwich make successfully!!!” which will print only when child process executes. Because of wait() function.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
int main()
{
pid_t a=fork();
if (a<0){
printf("fork failed");
}
else if(a==0){
printf("\nJunior who has third item start making sandwich\n");
}
else{
wait(NULL);
printf("\nSandwich make successfully!!!\n\n");
}
return 0 ;
}
```

Output :

```
sujana22@sujana22-virtual-machine:~/Desktop$ gcc Project6.c
sujana22@sujana22-virtual-machine:~/Desktop$ ./a.out

 Senior produced Two items :
cheese
Sausage
Senior send messeage : Take two items!

 Juniors recived message:Take two items!

Third Item is :
Bread

Junior who has third item start making sandwich

Sandwich make successfully!!!

sujana22@sujana22-virtual-machine:~/Desktop$ ./a.out

 Senior produced Two items :
Sausage
Bread
Senior send messeage : Take two items!

 Juniors recived message:Take two items!

Third Item is :
'Cheese'

Junior who has third item start making sandwich

Sandwich make successfully!!!

sujana22@sujana22-virtual-machine:~/Desktop$
```

```
sujana22@sujana22-virtual-machine:~/Desktop$ gcc Project6.c
sujana22@sujana22-virtual-machine:~/Desktop$ ./a.out

 Senior produced Two items :
cheese
Bread
Senior send messeage : Take two items!

 Juniors recived message:Take two items!

Third Item is :
'Sausage'

Junior who has third item start making sandwich

Sandwich make successfully!!!

sujana22@sujana22-virtual-machine:~/Desktop$ ./a.out

 Senior produced Two items :
Sausage
Bread
Senior send messeage : Take two items!

 Juniors recived message:Take two items!

Third Item is :
'Cheese'

Junior who has third item start making sandwich

Sandwich make successfully!!!

sujana22@sujana22-virtual-machine:~/Desktop$
```

Conclution:

Our project is about "The sandwich problem". We are four members in our group solve it using C language and ran our project in ubuntu terminal. Our honorable course instructor Md. Nawab Yousuf Ali sir has taught us the implementation of every topic theoretically and practically. So we use the knowledge and implemented a synchronisation method to solve the project. During the project solving, we found some issues multiple times and fixed them together. After this we got a perfectly done project. Overall the project is very efficient and usefull for users.