RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	19CSC304A
Course Name	Operating Systems
Programme	B. Tech.
Department	Computer Science and Engineering
Faculty	CSE

Name of the Student	Subhendu Maji
Reg. No	18ETCS002121
Semester/Year	5th / 2018
Course Leader/s	Ms. Naveeta

Declaration Sheet					
Student Name	Subhendu Maji				
Reg. No	18ETCS002121				
Programme	B. Tech			Semester/Year	5th / 2018
Course Code	19CSC304A				
Course Title	Operating Systems				
Course Date		to			
Course Leader	Ms. Naveeta				

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

Faculty of Engineering and Technology			
Ramaiah University of Applied Sciences			
Department	Computer Science and Engineering	Programme	B. Tech. in CSE
Semester/Batch	5th / 2018		
Course Code	19CSC304A	Course Title	Operating Systems
Course Leader(s)	Ms. Jishmi Jos Choondal/Ms. Naveeta		

Assignment			
Register No.	18ETCS002121	Name of Student	SUBHENDU MAJI

Sections		**Marking Scheme**	**Max Marks**	**First Examiner Marks**	**Moderator Marks**
Question 1	Q1.1	Introduction to 16-bit, 32- bit or 64-bit operating systems	01		
	Q1.2	Reasons for the transition from 16-bit to 32- bit and 32-bit to 64-bit operating systems	01		
	Q1.3	Reasons for the transition from 64-bit to 128-bit operating systems	02		
	A1.4	Stance with justification	01		
		Part A	**05**		
estion 2	Q2.1	Introduction to NRU, FIFO, LRU and second chance algorithms	02		
	Q2.2	Compute the page replaced on a page fault	08		
		B2 Max Marks	**10**		
estion3	Q3.1	Problem solving approach for spooler	02		
	Q3.2	Design and implementation of spooler	06		
	Q3.3	Results and analysis of spooler	02		
		B2 Max Marks	**10**		

	Total Assignment Marks		25		

Course Marks Tabulation				
Component- 1(B) Assignment	**First Examiner**	**Remarks**	**Moderator**	**Remarks**
Q1				
Q2				
Q3				
Marks (out of 25)				

Signature of First Examiner   Signature of Second Examiner

Question No. 1

Solution to Question No. 1:

In computer architecture, 128-bit integers, memory addresses, or other data units are those that are 128 bits (16 octets) wide. Also, 128-bit CPU and ALU architectures are those that are based on registers, address buses, or data buses of that size.

1.1 Introduction to 16-bit, 32- bit or 64-bit operating systems

The 16-bit Microprocessor means that it has 16 address lines.
In computer architecture, 16-bit integers, memory addresses, or other data units are those that are 16 bits (2 octets) wide. Also, 16-bit CPU and ALU architectures are those that are based on registers, address buses, or data buses of that size. 16-bit microcomputers are computers in which 16-bit microprocessors were the norm. A 16-bit microprocessor is having 16bit register set. It has 16 address and data lines to transfer address and data both. Hence it is 16 address lines. The maximum addresses are 2^16 means 65536.
The term is often applied to the following: microprocessor: indicates the width of the registers. A 16-bit microprocessor can process data and memory addresses that are represented by 16 bits. bus: indicates the number of wires in the bus. A 16-bit bus transmits 16 bits in parallel. Some example of 16-bit operating system is free cell, Dialer, DriveSpace.

The 32-bit processor was the primary processor used in all computers until the early 1990s. Intel Pentium processors and early AMD processors were 32-bit, which means the operating system and software work with data units that are 32 bits wide. Most computers made in the 1990s and early 2000s were 32-bit machines. The CPU register stores memory addresses, which is how the processor accesses data from RAM. One bit in the register can reference an individual byte in memory, so a 32-bit system can address a maximum of 4 GB (4,294,967,296 bytes) of RAM. The actual limit is often less around 3.5 GB since part of the register is used to store other temporary values besides memory addresses. Most computers released over the past two decades were built on a 32-bit architecture, hence most operating systems were designed to run on a 32-bit processor. Windows 95, 98, and XP are all 32-bit operating systems.
A 32-bit system can access 2^{32} memory addresses, i.e., 4 GB of RAM or physical memory ideally, it can access more than 4 GB of RAM also.

An operating system that is designed to work in a computer that processes 64 bits at a time. Most personal computers today are 64-bit, and mobile devices are expected to begin using 64-bit CPUs in the 2014–2015-

time frame. A 64-bit operating system will not work in a 32-bit computer, but a 32-bit operating system will run in a 64-bit computer. A 64-bit register can theoretically reference 18,446,744,073,709,551,616 bytes, or 17,179,869,184 GB (16 exabytes) of memory. This is several million times more than an average workstation would need to access. What's important is that a 64-bit computer (which means it has a 64-bit processor) can access more than 4 GB of RAM. If a computer has 8 GB of RAM, it better has a 64-bit processor. Otherwise, at least 4 GB of the memory will be inaccessible by the CPU.

A 64-bit system can access 2^{64} memory addresses, i.e actually 18-Quintillion bytes of RAM. In short, any amount of memory greater than 4 GB can be easily handled by it.

1.2 Reasons for the transition from 16-bit to 32- bit and 32-bit to 64-bit operating systems

Transition from 16-bit to 32-bit.

Because of their internal data paths are narrower, so they use fewer transistors and cost less to manufacture. They only need 16-bit paths to external memory, so they have half as many memory bus data pins as 32-bit processors.

Transition from 32 bit to 64 bits.

- A major difference between 32-bit processors and 64-bit processors is the number of calculations per second they can perform, which affects the speed at which they can complete tasks. 64-bit processors can come in dual-core, quad-core, six-core, and eight-core versions for home computing. Multiple cores allow for an increased number of calculations per second that can be performed, which can increase the processing power and help make a computer run faster. Software programs that require many calculations to function smoothly can operate faster and more efficiently on the multi-core 64-bit processors, for the most part.
- A computer with a 64-bit processor can have a 64-bit or 32-bit version of an operating system installed. However, with a 32-bit operating system, the 64-bit processor would not run at its full capability.
- On a computer with a 64-bit processor, we can't run a 16-bit legacy program. Many 32-bit programs will work with a 64-bit processor and operating system, but some older 32-bit programs may not function properly, or at all, due to limited or no compatibility.
- Unlike the migration from 16 to 32 bit (which made perfect sense at the time), the switch from 32 to 64 bit has been proven substantially less beneficial. The benefits of the wider registers are limited to applications that require heavy math computations such as encryption software and the like. And while the larger address space (up to 2^64 = 16 Exabyte!) is useful for applications that benefit from more than 4GB, this is also at the expense of data locality (the processor cache didn't grow with the address space) and ability to scale to a wider range of devices.

1.3 Reasons for the transition from 64-bit to 128-bit operating systems

Because of the 128 bits is still overkill these days. For most purpose 32 or 64 bits is still more than enough. For simpler things even 8 or 16 bits are still practical.

While there are currently no mainstream general-purpose processors built to operate on 128-bit integers or addresses, a number of processors do have specialized ways to operate on 128-bit chunks of data. The IBM System/370 could be considered the first simple 128-bit computer, as it used 128-bit floating-point registers. Most modern CPUs feature single-instruction multiple-data (SIMD) instruction sets (Streaming SIMD Extensions, AltiVec etc.) where 128-bit vector registers are used to store several smaller numbers, such as four 32-bit floating-point numbers. A single instruction can then operate on all these values in parallel. However, these processors do not operate on individual numbers that are 128 binary digits in length; only their registers have the size of 128 bits.

Since the increase to 64-bit has already pushed the limits of practical processors, a further increase to 128 bit which will require introducing yet another WoW compatibility layer makes even less sense. That's especially true for desktop applications, but also goes for the majority of server applications, which are designed for scale-out architectures and deployed on commodity hardware.

1.4 Stance with justification

There aren’t many applications that are actually better off in 64 bits space.

We have to pay for the wide ALU whether we use it, or not, or only a small fraction of its capabilities. To justify a 128-bit ALU, we have to find a problem, significant enough, that can benefit from manipulating 128 bit words in single instructions. We can probably come up a few examples, however these are already served with special purpose instruction sets (like SSE) or highly targeted processors for that purpose (e.g., DSP), and there aren't enough of such problems that make manufacturers feel they will get a return on the investment required to produce such a chip. For instance, we wouldn't put one of those chips in a fridge, a car, etc. In terms of cost, consider the wide ALU doesn't only make it more expensive, but other parts of the chip as well. A 128-bit wide ALU also means there have to be 128-bit wide data paths: that takes a lot of silicon area. Also, that data has to come from somewhere and go somewhere, so there would need to be similarly large registers, cache, other memory, etc., for the wide ALU to be used effectively.

My final point is that we can already perform arbitrary width arithmetic on any processor. We can add 32-bit memory words on a PIC 18 in 8 instructions, whereas we could do it on the same architecture scaled to 32 bits in only 2 instructions. We can perform arithmetic on arbitrarily large numbers using software libraries which can be compiled on current processors. A narrow ALU doesn't keep us from performing big computations, it only limits us in certain aspects.

Solution to Question No. 2:

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

2.1 Introduction to NRU, FIFO, LRU and second chance algorithms

NRU- Not Recently Used Page Replacement Algorithm

This algorithm removes a page at random from the lowest numbered non-empty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick than a clean page that is in heavy use.
When page is modified, a modified bit is set. When a page needs to be replaced, the Operating System divides pages into 4 classes.

	R	**M**
CLASS 0	Not Referenced	Not Modified
CLASS 1	Not Referenced	Modified
CLASS 2	Referenced	Not Modified
CLASS 3	Referenced	Modified

Out of above 4 categories, NRU will replace a Not Referenced, Not Modified page, if such page exists. This algorithm implies that a Modified but Not Referenced is less important than a Not Modified and Referenced.

FIFO- First In First Out Page Replacement Algorithm

In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

LRU- Least Recently Used Page Replacement Algorithm

In this algorithm page will be replaced which is least recently used. In other words, LRU replaces the page that has not been used for the longest period of time

Second Chance Page Replacement Algorithm

In the Second Chance page replacement policy, the candidate pages for removal are considered in a round robin matter, and a page that has been accessed between consecutive considerations will not be replaced. The page replaced is the one that, when considered in a round robin matter, has not been accessed since its last consideration.

It can be implemented by adding a “second chance” bit to each memory frame-every time the frame is considered (due to a reference made to the page inside it), this bit is set to 1, which gives the page a second chance, as when we consider the candidate page for replacement, we replace the first one with this bit set to 0 (while zeroing out bits of the other pages we see in the process). Thus, a page with the “second chance” bit set to 1 is never replaced during the first consideration and will only be replaced if all the other pages deserve a second chance too.

2.2 Compute the page replaced on a page fault

We are given,

Page	Loaded Time (in clock ticks)	Time of Last Reference (in clock ticks)	R	M
0	250	280	0	1
1	120	285	1	0
2	265	282	0	0
3	110	295	1	0
4	185	289	1	1
5	135	283	0	0
6	275	291	1	1
7	115	279	1	0

Table 1 Given time of loading, time of last access, and the R and M bits for each page

a. NRU makes an approximation to replace the page based on referenced and modified bits. Based upon R and M bits, there are 4 classes:

	R	**M**
CLASS 0	Not Referenced	Not Modified
CLASS 1	Not Referenced	Modified
CLASS 2	Referenced	Not Modified
CLASS 3	Referenced	Modified

Table 2 Classes based upon referenced and modified bits

According to the Table 2,

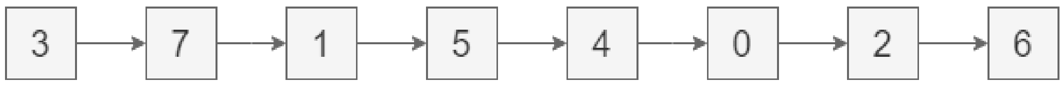
Page	Belongs to
0	Class 1
1	Class 2
2	Class 0
3	Class 2
4	Class 3
5	Class 0
6	Class 3
7	Class 2

NRU algorithm removes class 0 pages first. Hence, **Page 2** and **Page 5** will be replaced by NRU.

b. FIFO replacement algorithm works in a way that it will replace the page that arrived the earliest.

According to Table 1,

FIFO queue will be as follows:

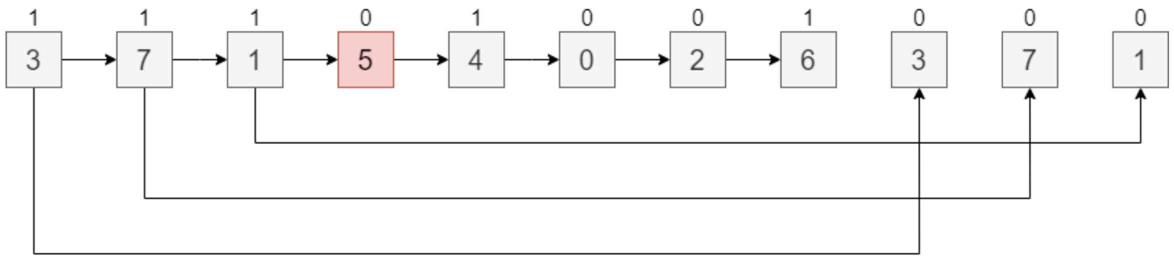


Page 3 got the least value of Loaded Time, which means it arrived the earliest.
Hence, **Page 3** will be replaced by FIFO.

c. LRU is a page replacement algorithm that replaces the page that is least recently used. Thus, LRU takes decision on the basis of the Last Reference and page having least Last Reference is being replaced.

According to Table 1, Page 7 got the least Time of last Reference.
Hence, **Page 7** will be replaced by LRU.

d. Second chance algorithm works by adding a 'second chance' bit to each memory frame whenever page is referenced associated to that frame, this bit is set and when replacement is needed, this bit is reset (set to 0).



Hence, **Page 5** will be replaced by Second Chance Algorithm.

Question No. 3

Solution to Question No. 3:

3.1 Problem solving approach for spooler

This problem is a classic Producer-Consumer Problem.
The producer and consumer share a fixed-size buffer used as a queue. The producer's job is to generate data and put this in the buffer. The consumer's job is to consume the data from this buffer, one at a time.

In a computer system peripheral equipment, such as printers and punch card readers etc (batch processing), are very slow relative to the performance of the rest of the system. Getting input and output from the system was quickly seen to be a bottleneck.

Spooling works like a typical request queue where data, instructions and processes from multiple sources are accumulated for execution later on. It is a kind of buffering mechanism or a process in which data is temporarily held to be used and executed by a device, program or the system.

The printer spooler problem, is similar to the producer consumer problem. The producer and consumer share a fixed-size buffer used as a queue. The producer's job is to generate data and put this in the buffer. The consumer's job is to consume the data from this buffer, one at a time.

The problem occurs when to producers try to put data into the buffer at the same time, the outcome (here which file prints first), depends on the particular sequence in which usage of the shared buffer (spooler) occurs, this is called a race condition.
This is solved by achieving mutual exclusion, here I'm using mutex to get mutual exclusion. Mutex works like a lock, if one process is using the shared buffer, it locks the access to the buffer for all the other processes, it unlocks the mutex after it is done. The same is repeated by all the processes that need to access the data

Another problem occurs when producer tries to put data into the buffer when it is full, it wastes CPU cycles. The same is true for consumer it tries to consumer from an empty buffer.

This can be solved by using two semaphores, full and empty: full represents the number of full slots in the spooler, and the empty represents the number of empty slots. A process waits if there are no full slots, similarly the printer waits if there are no empty slots.

3.2 Design and implementation of spooler

We can use Semaphore and Mutex to solve our problem.
Data structures we are going to use:

- Header file `semaphore.h`
  We will use semaphore of type `sem_t`

commands	purpose
`sem_init`	Initializes the semaphore to some initial value
`sem_wait`	Blocks the calling process until one of its child processes exits or a signal is received.
`sem_post`	Generates interrupt that is sent to a process
`sem_destroy`	Destroys the semaphore to avoid memory leak

- Header file `pthread.h`
  We will use mutex of type `pthread_mutex_t`

commands	purpose
`pthread_mutex_init`	Initializes the mutex
`pthread_mutex_lock()`	Blocks the calling process until one of its child processes exits or a signal is received.
`pthread_mutex_unlock()`	Generates interrupt that is sent to a process
`pthread_mutex_destroy()`	Destroys the mutex to avoid memory leak

Pseudocode solution using semaphore and mutex:
Initialization

```
Mutex mutex; // Used to provide mutual exclusion for critical section
Semaphore empty = N; // Number of empty slots in buffer
Semaphore full = 0 // Number of slots filled
int in = 0; //index at which producer will put the next data
int out = 0; // index from which the consumer will consume next data
int buffer[N];
```

Producer Code

```
while(True) {
   // produce an item
   wait(empty); // wait/sleep when there are no empty slots
   wait(mutex);
   buffer[in] = item
   in = (in+1)%buffersize;
   signal(mutex);
   signal(full); // Signal/wake to consumer that buffer has some   data and
they can consume now
}
```

Consumer Code

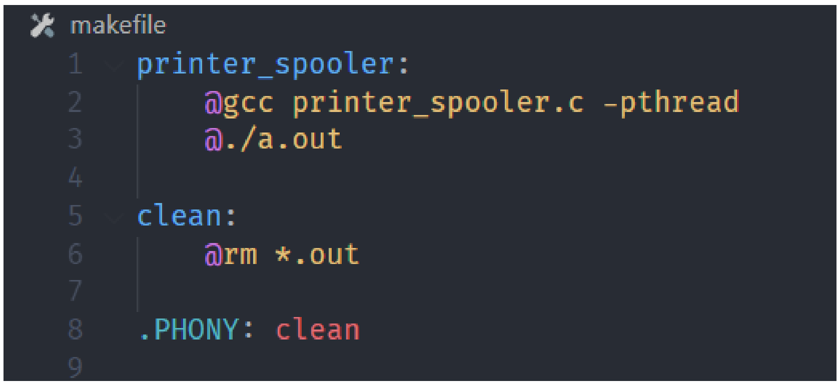
```
while(True) {
   wait(full); // wait/sleep when there are no full slots
   wait(mutex);
   item = buffer[out];
   out = (out+1)%buffersize;
   signal(mutex);
   signal(empty); // Signal/wake the producer that buffer slots are emptied 
and they can produce more
   //consumer the item
}
```

Source Code of Printer Spooler Program

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

#define MaxItems 2// Maximum items a producer can produce or a consumer can consume
#define SpoolerSize 5 // Size of the buffer

sem_t empty;
sem_t full;
int in = 0;
int out = 0;
int spooler[SpoolerSize];
pthread_mutex_t mutex;

void *producer(void *pno)
{
    int item;
    for (int i = 0; i < MaxItems; i++)
    {
        item = rand(); // Produce an random item
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        spooler[in] = item;
        printf("Process %d: Insert document %d at index %d\n", *((int *)pno), spool
er[in], in);
        in = (in + 1) % SpoolerSize;
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
}
void *consumer(void *cno)
{
    for (int i = 0; i < MaxItems; i++)
    {
```

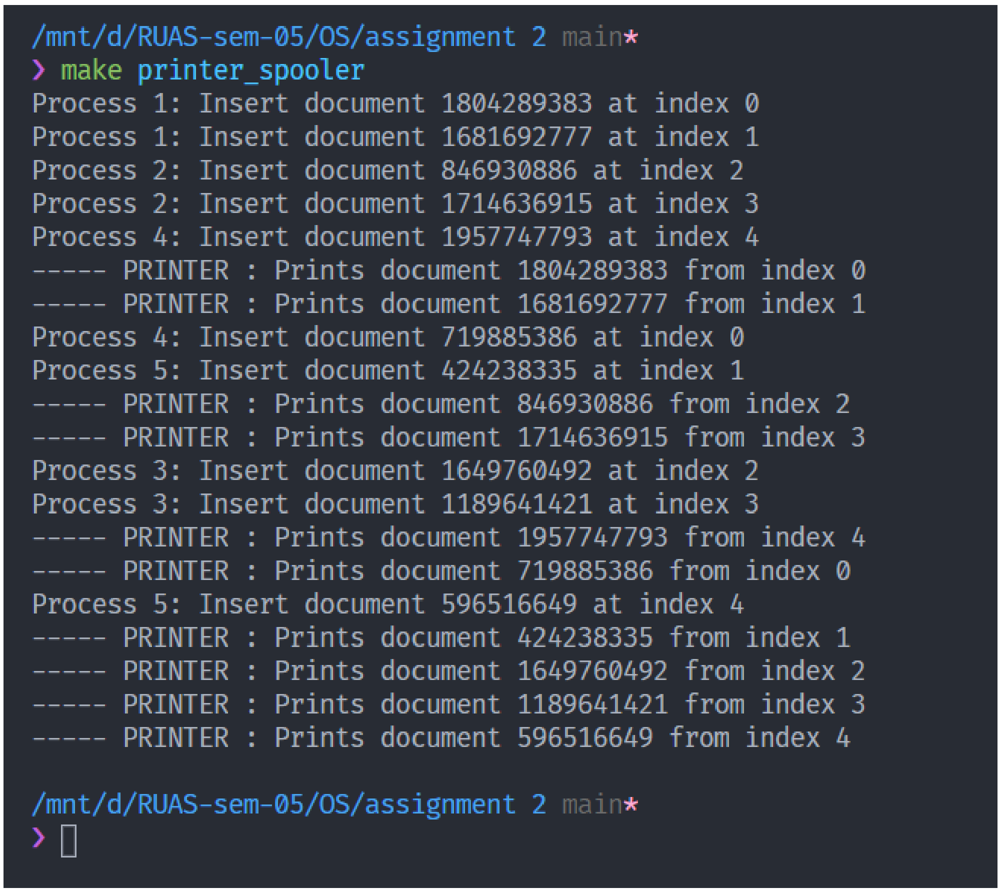
```c
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = spooler[out];
        printf("----- ");
        printf("PRINTER : Prints document %d from index %d\n", item, out);

        out = (out + 1) % SpoolerSize;
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
}

int main()
{

    pthread_t pro[5], con;
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty, 0, SpoolerSize);
    sem_init(&full, 0, 0);

    int a[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++)
    {
        pthread_create(&pro[i], NULL, (void *)producer, (void *)&a[i]);
    }

    for (int i = 0; i < 5; i++)
    {
        pthread_create(&con, NULL, (void *)consumer, (void *)&a[i]);
        pthread_join(con, NULL);
    }

    for (int i = 0; i < 5; i++)
    {
        pthread_join(pro[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

3.3 Results and analysis of spooler



```
makefile
printer_spooler:
	@gcc printer_spooler.c -pthread
	@./a.out

clean:
	@rm *.out

.PHONY: clean

```

Figure 1 Makefile



```
/mnt/d/RUAS-sem-05/OS/assignment 2 main*
❯ make printer_spooler
Process 1: Insert document 1804289383 at index 0
Process 1: Insert document 1681692777 at index 1
Process 2: Insert document 846930886 at index 2
Process 2: Insert document 1714636915 at index 3
Process 4: Insert document 1957747793 at index 4
----- PRINTER : Prints document 1804289383 from index 0
----- PRINTER : Prints document 1681692777 from index 1
Process 4: Insert document 719885386 at index 0
Process 5: Insert document 424238335 at index 1
----- PRINTER : Prints document 846930886 from index 2
----- PRINTER : Prints document 1714636915 from index 3
Process 3: Insert document 1649760492 at index 2
Process 3: Insert document 1189641421 at index 3
----- PRINTER : Prints document 1957747793 from index 4
----- PRINTER : Prints document 719885386 from index 0
Process 5: Insert document 596516649 at index 4
----- PRINTER : Prints document 424238335 from index 1
----- PRINTER : Prints document 1649760492 from index 2
----- PRINTER : Prints document 1189641421 from index 3
----- PRINTER : Prints document 596516649 from index 4

/mnt/d/RUAS-sem-05/OS/assignment 2 main*
❯ 
```

Figure 2 Execution of Printer Spooler

Hence, we can see in Fig 2 that there are 5 threads each sending 2 documents for printing. Each document names are generated randomly. We can see that all the documents sent by 5 processes are getting printed successfully.

1. https://shivammitra.com/c/producer-consumer-problem-in-c/#
2. https://hackmd.io/@25077667/os-hw4#fn1
3. https://en.wikipedia.org/wiki/128-bit_computing
4. https://www.embedded.com/migrating-from-8-16-bit-to-32-bit-lessons-learned-the-hard-way/
5. https://www.quora.com/Why-isn%E2%80%99t-there-any-128-bit-Windows