4.1 Aim: To implement a **producer-consumer problem** (also known as the **bounded-buffer problem)**.

Software Requirements: GNU C Compiler(GNU Compiler Collection), <stdio.h> <pthread.h> <sys/types.h> <sys/stat.h> <fcntl.h> <unistd.h>

Tested on hardware: -

Computer-

Processor	: Intel(R) Pentium(R) 4 CPU 2.40GHz
Memory	: 501MB (257MB used)
Operating System	: Linux Mint 9 Isadora
User Name	: jazz (Jai Shree Krishna)
OpenGL Renderer	: Mesa DRI Intel(R) 845G GEM 20091221 2009Q4 x86/MMX/SSE2

Description: producer-consumer problem (also known as the **bounded-buffer problem**) is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Source Code:

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


#define MAX 1000

long count=0,in=0,out=0;
long array[MAX];
int continue_check=1;

void producer()
{
        while(continue_check)
        {
                if(count<MAX)
                {
```

```
                array[in]=in;
                printf("Producer : Item %ld produced\n",in);
                count++;
                in = (in+1)%MAX;
        }
    }
}
void consumer()
{
    while(continue_check)
    {
        while(count==0);

        if((array[out]!=out) && array[out]>-1)
        {
            continue_check=-1;
            printf("Race condition at %ld\n",out);
            return;
        }
        else
        {
            printf("Consumer : Item %ld consumed at index %ld\n",array[out],out);
            count--;
            out=(out+1)%MAX;
        }
    }
}
int main()
{
    pthread_t producer_thread;
    pthread_t consumer_thread;
    int i=0;
    long index=0;

    for(index=0;index<MAX;index++)
        array[index]=-1;

    pthread_create(&producer_thread,NULL,(void *)producer,NULL);
    pthread_create(&consumer_thread,NULL,(void *)consumer,NULL);

    pthread_join(producer_thread,NULL);
    pthread_join(consumer_thread,NULL);
return 0;
}
```

4.2 Aim: To implement multiple **producers-consumers** bounded on to the same buffer and to observe the Race-condition**.**

Software Requirements: GNU C Compiler(GNU Compiler Collection), <stdio.h> <pthread.h> <sys/types.h> <sys/stat.h> <fcntl.h> <unistd.h>

Tested on hardware: -

Computer-

Processor	: Intel(R) Pentium(R) 4 CPU 2.40GHz
Memory	: 501MB (257MB used)
Operating System	: Linux Mint 9 Isadora
User Name	: jazz (Jai Shree Krishna)
OpenGL Renderer	: Mesa DRI Intel(R) 845G GEM 20091221 2009Q4 x86/MMX/SSE2

Description:

Consider the example:

```
int itemCount

procedure producer() {
    while (true) {
        item = produceItem()

        if (itemCount == BUFFER_SIZE) {
            sleep()
        }

        putItemIntoBuffer(item)
        itemCount = itemCount + 1

        if (itemCount == 1) {
            wakeup(consumer)
        }
    }
}

procedure consumer() {
    while (true) {

        if (itemCount == 0) {
            sleep()
        }

        item = removeItemFromBuffer()
        itemCount = itemCount - 1

        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer)
```

```
        }

        consumeItem(item)
    }
}
```

Race Condition that can lead into a deadlock. Consider the following scenario:

1. The consumer has just read the variable `itemCount`, noticed it's zero and is just about to move inside the if-block.
2. Just before calling sleep, the consumer is interrupted and the producer is resumed.
3. The producer creates an item, puts it into the buffer, and increases `itemCount`.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when `itemCount` is equal to 1.
6. The producer will loop until the buffer is full, after which it will also go to sleep.

Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

An alternative analysis is that if the programming language does not define the semantics of concurrent accesses to shared variables (in this case `itemCount`) without use of synchronization, then the solution is unsatisfactory for that reason, without needing to explicitly demonstrate a race condition.

Source Code:

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


#define MAX 1000

long count=0,in=0,out=0;
long array[MAX];
int continue_check=1;

void producer()
{
        while(continue_check)
```

```
        {
                if(count<MAX)
                {
                        array[in]=in;
                        printf("Producer : Item %ld produced\n",in);
                        count++;
                        in = (in+1)%MAX;
                }
        }
}

void consumer()
{
        while(continue_check)
        {
                while(count==0);

                if((array[out]!=out) && array[out]>-1)
                {
                        continue_check=-1;
                        printf("Race condition at %ld : in thread %u\n",out,pthread_self());
                        return;
                }
                else
                {
                        printf("Consumer : Item %ld consumed at index %ld\n",array[out],out);
                        count--;
                        out=(out+1)%MAX;
                }
        }
}

int main()
{
        pthread_t producer_thread[20];
        pthread_t consumer_thread[20];
        int i=0;
        long index=0;

        for(index=0;index<MAX;index++)
                array[index]=-1;

        for(index=0;index<20;index++)
        {
```

```
                pthread_create(&producer_thread[index],NULL,(void *)producer,NULL);
                pthread_create(&consumer_thread[index],NULL,(void *)consumer,NULL);
        }

        for(i=0;i<20;i++)
        {
                pthread_join(producer_thread[i],NULL);
                pthread_join(consumer_thread[i],NULL);
        }


        return 0;
}
```

4.3 Aim: To implement a **producer-consumer problem** by avoiding the Race-condition using pthread_mutex.

Software Requirements: GNU C Compiler(GNU Compiler Collection), <stdio.h> <pthread.h> <sys/types.h> <sys/stat.h> <fcntl.h> <unistd.h>

Tested on hardware: -

Processor	: Intel(R) Pentium(R) 4 CPU 2.40GHz
Memory	: 501MB (257MB used)
Operating System	: Linux Mint 9 Isadora
User Name	: jazz (Jai Shree Krishna)
OpenGL Renderer	: Mesa DRI Intel(R) 845G GEM 20091221 2009Q4 x86/MMX/SSE2

Description:

In concurrent programming a **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore.

By carefully controlling which variables are modified inside and outside the critical section (usually, by accessing important state only from within), concurrent access to that state is prevented. A critical section is typically used when a multithreaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure a shared resource, for example a printer, can only be accessed by one process at a time.

Source Code:

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


#define MAX 1000

long count=0,in=0,out=0;
long array[MAX];
```

```
int continue_check=1;

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;


void producer()
{
        while(continue_check)
        {
                if(count<MAX)
                {
                        pthread_mutex_lock(&m);
                        array[in]=in;
                        printf("Producer : Item %ld produced\n",in);
                        count++;
                        in = (in+1)%MAX;
                        pthread_mutex_unlock(&m);
                }
        }
}

void consumer()
{
        while(continue_check)
        {
                while(count==0);
                pthread_mutex_lock(&m);
                if((array[out]!=out) && array[out]>-1)
                {
                        continue_check=-1;
                        printf("Race condition at %ld : in thread %u\n",out,pthread_self());
                        return;
                }
                else
                {
                        printf("Consumer : Item %ld consumed at index %ld\n",array[out],out);
                        count--;
                        out=(out+1)%MAX;
                }
                pthread_mutex_unlock(&m);
        }
}
```

```
int main()
{
	pthread_t producer_thread[20];
	pthread_t consumer_thread[20];
	int i=0;
	long index=0;

	for(index=0;index<MAX;index++)
		array[index]=-1;


	for(index=0;index<20;index++)
	{
		pthread_create(&producer_thread[index],NULL,(void *)producer,NULL);
		pthread_create(&consumer_thread[index],NULL,(void *)consumer,NULL);
	}

	for(i=0;i<20;i++)
	{
		pthread_join(producer_thread[i],NULL);
		pthread_join(consumer_thread[i],NULL);
	}
	return 0;
}
```

5.1 Aim: To demonstrate the use of **execl** system call.

Software Requirements: GNU C Compiler(GNU Compiler Collection), <stdio.h> <strings.h> <unistd.h>

Tested on hardware:

Processor	: Intel(R) Pentium(R) 4 CPU 2.40GHz
Memory	: 501MB (257MB used)
Operating System	: Linux Mint 9 Isadora
User Name	: jazz (Jai Shree Krishna)
OpenGL Renderer	: Mesa DRI Intel(R) 845G GEM 20091221 2009Q4 x86/MMX/SSE2

Description:

The function call "`execl()`" initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. `/bin/ls`) and arguments are passed to the function. Note that "arg0" is the command/file name to execute.

```
int execl(const char *path, const char *arg0, const char *arg1, const char
*arg2, ... const char *argn, (char *) 0);
```

eg. execl("/bin/ls","ls","-l",(char *)NULL);

Source Code:

```
#include<stdio.h>
#include<strings.h>
#include<unistd.h>

int main()
{
        int id;

        id=fork();
        if(id>0)
        {
                printf("In Parent Process\n");
        }
        else
        {
                printf("In Child Process now running 'ls' command\n");
                execl("/bin/ls","ls","-l",(char *)NULL);
        }
```

```
        return 0;
}
```

Input Output:

```
jazz@linuxmint ~/Desktop/MTech/osdi/Pipe $ ./execl.out
In Parent Process
jazz@linuxmint ~/Desktop/MTech/osdi/Pipe $ In Child Process now running 'ls' command
total 48
-rw-r--r-- 1 jazz jazz  269 2010-10-29 02:35 execl.c
-rwxr-xr-x 1 jazz jazz 7212 2010-10-29 02:35 execl.out
-rw-r--r-- 1 jazz jazz  628 2010-10-29 02:49 lsandsort.c
-rwxr-xr-x 1 jazz jazz 7325 2010-10-29 02:49 lsandsort.out
-rw-r--r-- 1 jazz jazz  163 2010-10-29 02:37 pipe1.c
-rwxr-xr-x 1 jazz jazz 7177 2010-10-29 02:38 pipe1.out
-rw-r--r-- 1 jazz jazz  361 2010-10-29 02:42 pipe2.c
-rwxr-xr-x 1 jazz jazz 7370 2010-10-29 02:42 pipe2.out
```

5.2 Aim: To demonstrate the use of PIPE system call.

Software Requirements: GNU C Compiler(GNU Compiler Collection), <stdio.h>

Tested on hardware:

Processor	: Intel(R) Pentium(R) 4 CPU 2.40GHz
Memory	: 501MB (257MB used)
Operating System	: Linux Mint 9 Isadora
User Name	: jazz (Jai Shree Krishna)
OpenGL Renderer	: Mesa DRI Intel(R) 845G GEM 20091221 2009Q4 x86/MMX/SSE2

Description:

The function call "`execl()`" initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. `/bin/ls`) and arguments are passed to the function. Note that "arg0" is the command/file name to execute.

```
int execl(const char *path, const char *arg0, const char *arg1, const char
*arg2, ... const char *argn, (char *) 0);
```

eg. execl("/bin/ls","ls","-l",(char *)NULL);

Source Code:

```
#include<stdio.h>

int main()
{
        int pfd[2];
        pipe(pfd);
        printf("Pipe read end desc is %d\n",pfd[0]);
        printf("Pipe write end desc is %d\n",pfd[1]);
        return 0;
}
```

Input Output:

jazz@linuxmint ~/Desktop/MTech/osdi/Pipe $ ./pipe1.out
Pipe read end desc is 3
Pipe write end desc is 4
jazz@linuxmint ~/Desktop/MTech/osdi/Pipe $

5.3 Aim: To demonstrate the use of **pipe** system call.

Software Requirements: GNU C Compiler(GNU Compiler Collection), <stdio.h> <strings.h>

Tested on hardware:
Processor : Intel(R) Pentium(R) 4 CPU 2.40GHz
Memory : 501MB (257MB used)
Operating System : Linux Mint 9 Isadora
User Name : jazz (Jai Shree Krishna)
OpenGL Renderer : Mesa DRI Intel(R) 845G GEM 20091221 2009Q4 x86/MMX/SSE2

Description:

In Unix-like computer operating systems (and, to some extent, Windows), a **pipeline** is the original *software pipeline*: a set of processes chained by their standard streams, so that the output of each process (*stdout*) feeds directly as input (*stdin*) to the next one. Each connection is implemented by an anonymous pipe.

eg. ls -l | sort

In this example, `ls` is the Unix directory lister, and `sort` is sort system call which takes input and sorts it in ascending or descending order. This command gives output of ls -l in sorted ordered.

Source Code:

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

int main()
{
	int pfd[2];
	char buff[20];
	int n=0,pid=-1;
	pipe(pfd);
	pid=fork();
	if(pid>0)
	{
		printf("In Parent Process\n");
		write(pfd[1],"Hello",sizeof("Hello"));
	}
	else
	{
		n=read(pfd[0],buff,sizeof(buff));
		buff[n]='\0';
		printf("In Child Process : received the message : %s\n",buff);
	}
	return 0;
}
```

Input Output:

jazz@linuxmint ~/Desktop/MTech/osdi/Pipe $ ./pipe2.out
In Parent Process
jazz@linuxmint ~/Desktop/MTech/osdi/Pipe $ In Child Process : received the message : Hello

5.4 Aim: An Example using pipe.

Software Requirements: GNU C Compiler(GNU Compiler Collection), <stdio.h> <strings.h> <unistd.h> <sys/types.h> <sys/wait.h>

Tested on hardware:

Processor	: Intel(R) Pentium(R) 4 CPU 2.40GHz
Memory	: 501MB (257MB used)
Operating System	: Linux Mint 9 Isadora
User Name	: jazz (Jai Shree Krishna)
OpenGL Renderer	: Mesa DRI Intel(R) 845G GEM 20091221 2009Q4 x86/MMX/SSE2

Description:

The function call "`execl()`" initiates a new program in the same environment in which it is operating. An executable (with fully qualified path. i.e. `/bin/ls`) and arguments are passed to the function. Note that "arg0" is the command/file name to execute.

```
int execl(const char *path, const char *arg0, const char *arg1, const char
*arg2, ... const char *argn, (char *) 0);
```

eg. execl("/bin/ls","ls","-l",(char *)NULL);

Source Code:

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

int main()
{
        int pfd[2];
        int pid[2];

        pipe(pfd);
        pid[0]=fork();
        if(pid[0]>0)
        {
                printf("In Parent Process\n");
                pid[1]=fork();
                if(pid[1]>0)
```

```
                {
                }
                else
                {
                        printf("In Child Process : running 'sort' command\n");
                        close(pfd[1]);
                        dup2(pfd[0],0);
                        close(pfd[0]);
                        execl("/bin/sort","sort",(char *)NULL);
                }
        }
        else
        {
                printf("In Child Process: running 'ls' command\n");
                close(pfd[0]);
                dup2(pfd[1],1);
                close(pfd[1]);
                execl("/bin/ls","ls","-l",(char *)NULL);
        }
return 0;
}
```

5.4 Aim: Example two processes communicating via shared memory.

Software Requirements: GNU C Compiler(GNU Compiler Collection), <sys/shm.h> <sys/ipc.h> <sys/types.h> <unistd.h> <string.h> <stdio.h>

Tested on hardware:

Processor	: Intel(R) Pentium(R) 4 CPU 2.40GHz
Memory	: 501MB (257MB used)
Operating System	: Linux Mint 9 Isadora
User Name	: jazz (Jai Shree Krishna)
OpenGL Renderer	: Mesa DRI Intel(R) 845G GEM 20091221 2009Q4 x86/MMX/SSE2

Description:

Shared Memory is an efficeint means of passing data between programs. One program will create a memory portion which other processes (if permitted) can access.

A process creates a shared memory segment using `shmget()|`. The original owner of a shared memory segment can assign ownership to another user with `shmctl()`. It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl()`. Once created, a shared segment can be attached to a process address space using `shmat()`. It can be detached using `shmdt()` (see `shmop()`). The attaching process must have the appropriate permissions for `shmat()`. Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototypews can be found in <`sys/shm.h`>.

Source Code:

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

int main()
{
        pid_t pida[2];
        int pid;
        char buffer[1024];
```

```
        pid = fork();




        if(pid != 0)
        {
                printf("from write end:%s\n",strcpy((shmat(shmget((key_t)2,
(size_t)1024,IPC_CREAT|0666),NULL,0)),"hello world\n"));
        }
        else
        {
                sleep(1);
                printf("from read end:%s\n",strcpy(buffer,shmat(shmget((key_t)2,
(size_t)1024,0666),NULL,0)));
        }
}
```

Input Output:

jazz@linuxmint ~/Desktop/MTech/osdi/shm $ ./shm1.out
from write end:hello world

jazz@linuxmint ~/Desktop/MTech/osdi/shm $ from read end:hello world