1. getpit

#include<stdio.h>

#include<unistd.h>//header file

void main()

{

    printf("hello my name is sumit and my id is %d",getpid());

    return 0;

}

1. getppid

#include <stdio.h>

#include <unistd.h>//header file

int main()

{

    printf("hello my pid is %d\n",getpid());

    printf("hello my ppid is %d\n",getppid());

    printf("hello my ppid is %d\n",getppid());

    return 0;

}

1. exit

#include <stdio.h>

#include <stdlib.h>//header file

int main(void)

{

    printf("Start of the program....\n");

    printf("Exiting the program....\n");

    //it clear the buffer

    exit(300);

    printf("End of the program....\n");

    return 0;

}

1. \_exit

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>//header file

int main()

{

    printf("hello i start the program \n");//remember about putting \n

    \_exit(2);

    printf("This is our last statement of the program \n");

}

1. fork

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>//header file

int main()

{

    //this fork make 1 child after calling of this fork control will go to the next line in main and child

    fork();

    //this fork make 1 more child after calling of this fork control will go to the next line in main and child

    fork();

    printf("hello world\n");//4times

}

1. multiple\_fork

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>//header file

int main()

{

    //this fork make 1 child after calling of this fork control will go to the next line in main and child

    fork();

    //this fork make child of 2 parents

    fork();

    //this hello world print 4 times

    printf("hello world\n");

    //this fork make child of 4 parents

    fork();

    printf("hello world hy\n");

}

1. use\_fork\_with\_its\_id

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>//header file

int main()

{

    int a=fork();

    int b=fork();

    if(a>0&&b>0)

    {

    printf("this is parent first node\n");

    }

    else if(a>0&&b==0)

    {

    printf("this is child of a and second node\n");

    }

    else if(b>0&&a==0)

    {

    printf("This is child of b and third node\n");

    }

    else if(a==0&&b==0)

    {

    printf("This is child of b and forth node\n");

    }

}

1. wait

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>//header file

int main()

{

    int a=fork();

    if(a>0)

    {

    wait(NULL);

    printf("i am comming after waiting\n");

    }

    else

    {

    printf("i am child of a\n");

    }

}

1. wait\_with\_passing\_pointer

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>//header file

#include<stdlib.h>

int main()

{

    int \*p=(int\*)malloc(2);

    int id=fork();

    if(id>0)//parent

    {

    wait(p);//pid\_t waitpid(int \*stat\_loc);

    //here it wait because id is >0

    printf("p terminated\n");

    //after waiting when it come here it print the exit status like

    //if exit status is 1-:0000001  0000000

    //if exit status is 2-:0000010  0000000

    //if exit status is -1-: 255

    //if exit status is -1-: 254

    printf("Exit status %d\n",\*p);

    printf("Exit status %d\n",\*p/256);

    }

    else

    {

    printf("c finished\n");

    exit(-2);

    }

}

1. waitpid

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/wait.h>//header file

int main()

{

    printf("here we start our waitpid process\n");

    int p=fork();

    int a=fork();

    int childid;

    int grandchild;

    int \*t=(int\*)malloc(2);

    if(a==0)

    {

        grandchild=getpid();

    }

    if(p==0)

    {

        childid=getpid();

    }

    if(p>0)

    {

        waitpid(childid,t,0);//pid\_t waitpid(pid\_t pid,int \*stat\_loc,int options);

        printf("I am comming after my chid is done\n");

    }

    else{

        printf("I am child\n");

    }

    if(p>0&&a>0)

    {

        printf("I am the parent");

    }

}

1. pause

#include<stdio.h>

#include<unistd.h>//header file

#include<stdlib.h>

#include<sys/wait.h>

#include<signal.h>

void signal\_handler(int sig)

{

}

int main()

{

    signal(SIGCONT,signal\_handler);

    int a=fork();

    if(a>0)

    {

        sleep(1);

        printf("parent process\n");

        kill(a,SIGCONT);

    }

    else{

        pause();//int pause(void);

        printf("child process\n");

    }

}

1. kill

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<signal.h>//header file

#include<sys/wait.h>

int main()

{

    int a=fork();

    if(a>0)

    {

        sleep(1);

        printf("This is parent\n");

        kill(a,SIGCONT);

        wait(NULL);

    }

    else

    {   kill(getpid(),SIGSTOP);

        printf("This is child");

    }

    return 0;

}

1. sleep

// #include<stdio.h>

// #include<unistd.h>//header file

// #include<signal.h>

// #include<stdlib.h>

// #include<sys/wait.h>

// int main()

// {

//     int \*p=(int\*)malloc(2);

//     int id=fork();

//     if(id>0)

//     {

//         kill(id,SIGKILL);

//         int status;

//         wait(&status);

//         printf("Parent hase been called\n");

//     }

//     else{

//         sleep(5);

//         printf("child process\n");//if you cant write \n than it cant print this line

//         kill(id,SIGKILL);

//         exit(3);

//     }

//     printf("p's value is %d\n",\*p/256);

//     free(p);

//     return 0;

// }

OR

#include<stdio.h>

#include<unistd.h>//header file

#include<signal.h>

#include<stdlib.h>

int main()

{

    int \*ptr=(int\*)malloc(2);

    int id=fork();

    if(id>0)

    {

        sleep(10);

        printf("Parent hase been called");

        printf("yes no");

    }

    else{

        sleep(5);

        printf("value of %d\n",id);

        printf("child process\n");//if you cant write \n than it cant print this line

        kill(id,SIGKILL);

        exit(3);

    }

}

1. abort

#include<stdio.h>

#include<stdlib.h>//header file

int main()

{

    printf("Before abourt()\n");

    //Terminate the program abnormally

    abort();

    //This line will not be executed

    printf("After abort\n");

    return 0;

}

1. pthread

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>//header file

#include<stdlib.h>

int count=0;

void\* inc()

{

    printf("hello\n");

    for(int i=1;i<10000;i++)

    {

        //printf("%d ",count);

        count++;

    }

    int\* exit\_value = (int\*)malloc(sizeof(int));

    \*exit\_value = 100; // Set the exit value

}

void\* sub()

{

    printf("hello\n");

    for(int i=1;i<10000;i++)

    {

        //printf("%d ",count);

        count--;

    }

    // Exit with a value

    int\* exit\_value = (int\*)malloc(sizeof(int));

    \*exit\_value = 200; // Set the exit value

}

int main()

{

    pthread\_t t1,t2;

    int\* exit\_status1;

    int\* exit\_status2;

    pthread\_create(&t1,NULL,&inc,NULL);

    pthread\_create(&t2,NULL,&sub,NULL);

    pthread\_join(t1,NULL);

    pthread\_join(t2,NULL);

    printf("%d",count);

    return 0;

}

1. pthread\_join

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>//header file

#include<wait.h>

//#include<cstdlib>

int count=0;

void\* inc()

{

    printf("hello");

    for(int i=1;i<10000;i++)

    {

        printf("%d ",count);

        count++;

    }

}

void\* sub()

{

    printf("hello");

    for(int i=1;i<10000;i++)

    {

        printf("%d ",count);

        count--;

    }

}

int main()

{

    // int \*p=(int\*)malloc(2);

    // int id=fork();

    pthread\_t t1,t2;

    pthread\_create(&t1,NULL,&inc,NULL);

    pthread\_create(&t2,NULL,&sub,NULL);

    pthread\_join(t1,NULL);

    pthread\_join(t2,NULL);

    printf("%d",count);

    return 0;

}

1. ptread\_exit

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>//header file

#include<stdlib.h>

int count=0;

void\* inc()

{

    printf("hello\n");

    for(int i=1;i<10000;i++)

    {

        //printf("%d ",count);

        count++;

    }

    int\* exit\_value=(int\*)malloc(sizeof(int));

    \*exit\_value=100; // Set the exit value

    pthread\_exit(exit\_value);

}

void\* sub()

{

    printf("hello\n");

    for(int i=1;i<10000;i++)

    {

        //printf("%d ",count);

        count--;

    }

    // Exit with a value

    int\* exit\_value=(int\*)malloc(sizeof(int));

    \*exit\_value = 200; // Set the exit value

    pthread\_exit(exit\_value);

}

int main()

{

    pthread\_t t1,t2;

    int\* exit\_status1;

    int\* exit\_status2;

    pthread\_create(&t1,NULL,&inc,NULL);

    pthread\_create(&t2,NULL,&sub,NULL);

    pthread\_join(t1,(void\*\*)&exit\_status1);

    pthread\_join(t2,(void\*\*)&exit\_status2);

    printf("%d",count);

    // Print exit values

    printf("\nExit status of thread 1: %d\n",\*exit\_status1);

    printf("Exit status of thread 2: %d\n",\*exit\_status2);

    // Free memory allocated for exit values

    free(exit\_status1);

    free(exit\_status2);

    return 0;

}

1. pthread\_self

#include<stdio.h>

#include<unistd.h>

#include<pthread.h>//header file

#include<string.h>

void \*fun1()

{

    pthread\_t pid=pthread\_self();

    pthread\_exit((void\*)pid);

}

void \*fun2()

{

    pthread\_t pid=pthread\_self();

    pthread\_exit((void\*)pid);

}

int main()

{

    void \*status;

    pthread\_t t1,t2;

    int\* exit\_status1;

    int\* exit\_status2;

    pthread\_create(&t1,NULL,&fun1,NULL);

    pthread\_create(&t2,NULL,&fun2,NULL);

    pthread\_join(t1,(void\*\*)&exit\_status1);

    pthread\_join(t2,(void\*\*)&exit\_status2);

    printf("\nId of 1st thread %lu \n",(unsigned long)exit\_status1);

    printf("\n Id of 2nd thread %lu \n",(unsigned long)exit\_status1);

}

1. pthread\_return\_status

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>//header file

#include<stdlib.h>

int count=0;

void\* inc()

{

    printf("hello\n");

    for(int i=1;i<10000;i++)

    {

        //printf("%d ",count);

        count++;

    }

    int \*status = malloc(sizeof(int));

    \*status = 123;

    return (void\*)status;

}

int main()

{

    pthread\_t t1,t2;

    int\* status;

    int\* status1;

    pthread\_create(&t1,NULL,&inc,NULL);

    pthread\_create(&t2,NULL,&inc,NULL);

    pthread\_join(t1,(void \*\*)&status);

    pthread\_join(t2,(void \*\*)&status1);

    printf("Thread 1 exit status: %d\n",\*status);

    printf("Thread 1 exit status: %d\n",\*status1);

    free(status);

    free(status1);

    return 0;

}

1. pthread\_mutex\_lock

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>//header file

int count=0;

pthread\_mutex\_t mute;

void\* inc()

{

    for(int i=1;i<10000;i++)

    {

        pthread\_mutex\_lock(&mute);

        count++;

        pthread\_mutex\_unlock(&mute);

    }

}

void\* sub()

{

    for(int i=1;i<10000;i++)

    {

        pthread\_mutex\_lock(&mute);

        count--;

        pthread\_mutex\_unlock(&mute);

    }

}

int main()

{

    pthread\_mutex\_init(&mute,NULL);

    pthread\_t t1,t2;

    pthread\_create(&t1,NULL,&inc,NULL);

    pthread\_create(&t2,NULL,&sub,NULL);

    pthread\_join(t1,NULL);

    pthread\_join(t2,NULL);

    pthread\_mutex\_destroy(&mute);

    printf("%d",count);

    return 0;

}

1. pthread\_mutex\_trylock

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>//header file

int count=0;

pthread\_mutex\_t mute;

void\* inc()

{

    for(int i=1;i<10000;i++)

    {

        pthread\_mutex\_lock(&mute);

        pthread\_mutex\_trylock(&mute);//if lock is not here or we remove lock previously so here we give trylock it try to lock if lock is not present it lock otherwise ignore

        count++;

        pthread\_mutex\_unlock(&mute);

    }

}

void\* sub()

{

    for(int i=1;i<10000;i++)

    {

        pthread\_mutex\_lock(&mute);

        pthread\_mutex\_trylock(&mute);

        count--;

        pthread\_mutex\_unlock(&mute);

    }

}

int main()

{

    pthread\_mutex\_init(&mute,NULL);

    pthread\_t t1,t2;

    pthread\_create(&t1,NULL,&inc,NULL);

    pthread\_create(&t2,NULL,&sub,NULL);

    pthread\_join(t1,NULL);

    pthread\_join(t2,NULL);

    pthread\_mutex\_destroy(&mute);

    printf("%d",count);

    return 0;

}

1. race\_condition\_using\_semaphore

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>

#include<wait.h>

#include<semaphore.h>//header file

int count=0;

//make a samaphore

sem\_t s1;

void\* inc()

{

    for(int i=1;i<10000;i++)

    {

        //decrement the value of samaphore

        sem\_wait(&s1);

        count++;

        //increment the value of samaphore

        sem\_post(&s1);

    }

}

void\* sub()

{

    for(int i=1;i<10000;i++)

    {

        //decrement the value of samaphore

        sem\_wait(&s1);

        count--;

        //increment the value of samaphore

        sem\_post(&s1);

    }

}

int main()

{

    //initalize the samaphore

    //first:-samaphore second:-if theread-0 if process-1 third:-value of samaphore

    sem\_init(&s1,0,1);

    pthread\_t t1,t2;

    pthread\_create(&t1,NULL,&inc,NULL);

    pthread\_create(&t2,NULL,&sub,NULL);

    pthread\_join(t1,NULL);

    pthread\_join(t2,NULL);

    //distroy the samaphore

    sem\_destroy(&s1);

    printf("%d",count);

    return 0;

}

1. race\_condition\_using\_condition\_variable

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>//header file

#include<semaphore.h>

int count=0;

pthread\_cond\_t cond=PTHREAD\_COND\_INITIALIZER; // Initialize condition variable

pthread\_mutex\_t mutex=PTHREAD\_MUTEX\_INITIALIZER; // Initialize mutex

void\* inc()

{

    for(int i=1;i<10000;i++) {

        pthread\_mutex\_lock(&mutex);

        count++;

        pthread\_cond\_broadcast(&cond); // Signal waiting threads

        pthread\_mutex\_unlock(&mutex);

    }

    return NULL;

}

void\* sub()

{

    for(int i=1;i<10000;i++) {

        pthread\_mutex\_lock(&mutex);

        count--;

        pthread\_cond\_broadcast(&cond); // Signal waiting threads

        pthread\_mutex\_unlock(&mutex);

    }

    return NULL;

}

int main()

{

    pthread\_t t1,t2;

    pthread\_create(&t1,NULL,inc,NULL);

    pthread\_create(&t2,NULL,sub,NULL);

    pthread\_join(t1,NULL);

    pthread\_join(t2,NULL);

    printf("%d\n", count);

    // Destroy the condition variable and mutex

    pthread\_cond\_destroy(&cond);

    pthread\_mutex\_destroy(&mutex);

    return 0;

}

24.race\_condition\_using\_rwlock

#include<stdio.h>

#include<pthread.h>//header file

#include<stdlib.h>

int count=0;

pthread\_rwlock\_t rwlock=PTHREAD\_RWLOCK\_INITIALIZER;

void\* inc()

{

    for(int i=1;i<10000;i++)

    {

        pthread\_rwlock\_wrlock(&rwlock);

        count++;

        pthread\_rwlock\_unlock(&rwlock);

    }

    return NULL;

}

void\* sub()

{

    for(int i=1;i<10000;i++)

    {

        pthread\_rwlock\_wrlock(&rwlock);

        count--;

        pthread\_rwlock\_unlock(&rwlock);

    }

    return NULL;

}

int main()

{

    pthread\_t t1,t2;

    pthread\_rwlock\_init(&rwlock,NULL);

    pthread\_create(&t1,NULL,inc,NULL);

    pthread\_create(&t2,NULL,sub,NULL);

    pthread\_join(t1,NULL);

    pthread\_join(t2,NULL);

    pthread\_rwlock\_destroy(&rwlock);

    printf("%d\n",count);

    return 0;

}

25.create\_a\_file

#include<fcntl.h>//header file

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main()

{

    // Define the file name and permissions

    const char \*filename="newfile.txt";

    // Read and write permissions for the owner

    mode\_t mode=S\_IRUSR|S\_IWUSR;

    // Create the file using the creat system call

    int file\_descriptor=creat(filename,mode);

    if(file\_descriptor<0)

    {

        printf("Error creating file");

        exit(EXIT\_FAILURE);

    }

    printf("File '%s' created successfully with file descriptor %d\n",filename,file\_descriptor);

    // Close the file descriptor

    if(close(file\_descriptor)<0)

    {

        perror("Error closing file");

        exit(EXIT\_FAILURE);

    }

    return 0;

}

26.write\_in\_file

#include<fcntl.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>//header file

#include<string.h>

int main() {

    // Define the file name and permissions

    const char \*filename="hello.txt";

    // Read and write permissions for the owner

    int mode=S\_IRUSR|S\_IWUSR;

    // Create the file using the open system call with O\_CREAT flag

    int file\_descriptor=open(filename,O\_CREAT|O\_WRONLY,mode);//header file fcntl

    if(file\_descriptor<0)

    {

        printf("Error creating file");

        exit(EXIT\_FAILURE);

    }

    // Define the content to be written to the file

    const char \*content="Hello, World!\n";

    // Write the content to the file

    ssize\_t bytes\_written=write(file\_descriptor,content,strlen(content));

    if(bytes\_written<0)

    {

        printf("Error writing to file");

        close(file\_descriptor);

        exit(EXIT\_FAILURE);

    }

    printf("Successfully wrote %zd bytes to the file '%s'\n",bytes\_written,filename);

    // Close the file descriptor

    if(close(file\_descriptor)<0)

    {

        printf("Error closing file");

        exit(EXIT\_FAILURE);

    }

    return 0;

}

27. transfer\_content\_of\_one\_file\_to\_another

#include<fcntl.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#define BUFFER\_SIZE 1024

int main()

{

    // Define the source and destination file names

    const char \*source\_filename="source.txt";

    const char \*destination\_filename="destination.txt";

    // Open the source file for reading

    int source\_fd=open(source\_filename,O\_RDONLY);

    if(source\_fd<0)

    {

        printf("Error opening source file");

        exit(EXIT\_FAILURE);

    }

    // Open the destination file for writing (create if it does not exist)

    int destination\_fd=open(destination\_filename,O\_CREAT|O\_WRONLY|O\_TRUNC,S\_IRUSR|S\_IWUSR);

    if(destination\_fd<0)

    {

        printf("Error opening destination file");

        close(source\_fd);

        exit(EXIT\_FAILURE);

    }

    // Buffer to hold data during transfer

    char buffer[BUFFER\_SIZE];

    ssize\_t bytes\_read,bytes\_written;

    // Transfer the content from the source file to the destination file

    while((bytes\_read=read(source\_fd,buffer,BUFFER\_SIZE))>0)//header file unistd.h

    {

        bytes\_written=write(destination\_fd,buffer,bytes\_read);

        if(bytes\_written!=bytes\_read)

        {

            printf("Error writing to destination file");

            close(source\_fd);

            close(destination\_fd);

            exit(EXIT\_FAILURE);

        }

    }

    if (bytes\_read < 0) {

        printf("Error reading from source file");

    }

    // Close both file descriptors

    if(close(source\_fd)<0)

    {

        printf("Error closing source file");

        exit(EXIT\_FAILURE);

    }

    if(close(destination\_fd)<0)

    {

        printf("Error closing destination file");

        exit(EXIT\_FAILURE);

    }

    printf("Content transferred from '%s' to '%s' successfully\n",source\_filename,destination\_filename);

    return 0;

}

28.fork\_with\_excel

#include<stdio.h>

#include<unistd.h>//header file

#include<sys/types.h>

#include<sys/wait.h>

int main()

{

    pid\_t pid=fork();

    if(pid>0)

    {

        wait(NULL);

        printf("\nhere parent lies\n");

    }

    else

    {

        // Child process

        //remember here give the executable file location

        execl("/media/sumit/New Volume/Study/Du\_mca/2nd\_semester/Operating\_system/Lab/1\_getpid","1\_getpid.c",NULL);

        // If execl() fails

        printf("\nexecl");

        return 1;

    }

    return 0;

}

29.producer\_consumer\_problem\_using\_semaphore

#include<pthread.h>

#include<semaphore.h>//header file

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#define BUFFER\_SIZE 10

int buffer[BUFFER\_SIZE];

int in=0;  // Index for the next write by the producer

int out=0; // Index for the next read by the consumer

sem\_t empty; // Semaphore to count empty slots in the buffer

sem\_t full;  // Semaphore to count full slots in the buffer

pthread\_mutex\_t mutex; // Mutex to protect buffer access

void \*producer(void \*arg)

{

    int item;

    while(1)

    {

        item=rand()%100; // Produce an item

        sem\_wait(&empty); // Wait for an empty slot

        pthread\_mutex\_lock(&mutex); // Lock the buffer

        // Critical section: add the item to the buffer

        buffer[in]=item;

        printf("Producer produced %d\n",item);

        in=(in+1)%BUFFER\_SIZE;

        pthread\_mutex\_unlock(&mutex); // Unlock the buffer

        sem\_post(&full); // Signal that a new item has been added

        sleep(1); // Simulate time taken to produce an item

    }

}

void \*consumer(void \*arg)

{

    int item;

    while(1)

    {

        sem\_wait(&full); // Wait for a full slot

        pthread\_mutex\_lock(&mutex); // Lock the buffer

        // Critical section: remove the item from the buffer

        item=buffer[out];

        printf("Consumer consumed %d\n",item);

        out=(out+1)%BUFFER\_SIZE;

        pthread\_mutex\_unlock(&mutex); // Unlock the buffer

        sem\_post(&empty); // Signal that a slot has been emptied

        sleep(1); // Simulate time taken to consume an item

    }

}

int main() {

    pthread\_t prod\_tid,cons\_tid;

    // Initialize the semaphores

    //semaphore ,0:-to not share samaphore with other or 1:-to share samaphore withr other,initial value of semaphore

    sem\_init(&empty,0,BUFFER\_SIZE); // All buffer slots are initially empty

    sem\_init(&full,0,0); // No slots are initially full

    pthread\_mutex\_init(&mutex,NULL); // Initialize the mutex

    // Create the producer and consumer threads

    pthread\_create(&prod\_tid,NULL,producer,NULL);

    pthread\_create(&cons\_tid,NULL,consumer,NULL);

    // Join the threads (this program will run indefinitely)

    pthread\_join(prod\_tid,NULL);

    pthread\_join(cons\_tid,NULL);

    // Destroy the semaphores and mutex

    sem\_destroy(&empty);

    sem\_destroy(&full);

    pthread\_mutex\_destroy(&mutex);

    return 0;

}

30. pipe\_with\_parent\_and\_child

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<unistd.h>//header file

#include<sys/wait.h>

#include<sys/types.h>

#include<errno.h>

int main()

{

    int fd[2];

    if(pipe(fd)==-1)

    {

        printf("An error occurred with opening the pipe\n");

        return 1;

    }

    int id=fork();

    if(id==-1)

    {

        printf("An error ocurred with fork\n");

        return 2;

    }

    if(id==0)

    {

        //child process

        close(fd[0]);

        int data;

        printf("Input a no: ");

        scanf("%d",&data);

        write(fd[1],&data,sizeof(int));

        close(fd[1]);

    }

    else

    {

        //parent process

        close(fd[1]);

        int buffer;

        read(fd[0],&buffer,sizeof(int))==-1;

        printf("Got from child process %d\n",buffer);

        printf("Result is %d\n",buffer);

        close(fd[0]);

    }

    return 0;

}

31. semaphore

#include<pthread.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#include<semaphore.h>//header file

#define THREAD\_NUM 4

sem\_t semaphore;

void\* routine(void\* args)

{

    sem\_wait(&semaphore);

    sleep(1);

    printf("Thread %d\n",\*(int\*)args);

    sem\_post(&semaphore);

    free(args);

    return NULL;

}

int main(int argc,char \*argv[])

{

    pthread\_t th[THREAD\_NUM];

    sem\_init(&semaphore,0,4);

    for(int i=0;i<THREAD\_NUM;i++)

    {

        int\* a=malloc(sizeof(int));

        \*a=i;

        if(pthread\_create(&th[i],NULL,&routine,a)!=0)

        {

            printf("Failed to create thread");

        }

    }

    for(int i=0;i<THREAD\_NUM;i++)

    {

        if(pthread\_join(th[i],NULL)!=0)

        {

            printf("Failed to join thread");

        }

    }

    sem\_destroy(&semaphore);

    return 0;

}

32. semaphore

#include<pthread.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<semaphore.h>//header file

sem\_t s;

void\* child(void \*arg)

{

    printf("Child\n");

    sem\_post(&s);

    return NULL;

}

int main(int argc,char \*argv[])

{

    sem\_init(&s,0,0);

    printf("parent: begin\n");

    pthread\_t c;

    pthread\_create(&c,NULL,child,NULL);

    sem\_wait(&s);

    printf("parent: end\n");

    sem\_destroy(&s);

    return 0;

}

33. semaphore\_producer\_consumer\_problem

#include<pthread.h>

#include<semaphore.h>

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#define BUFFER\_SIZE 10

int buffer[BUFFER\_SIZE];

int in=0;  // Index for the next write by the producer

int out=0; // Index for the next read by the consumer

sem\_t empty; // Semaphore to count empty slots in the buffer

sem\_t full;  // Semaphore to count full slots in the buffer

pthread\_mutex\_t mutex; // Mutex to protect buffer access

void \*producer(void \*arg)

{

    int item;

    while(1)

    {

        item=rand()%100; // Produce an item

        sem\_wait(&empty); // Wait for an empty slot

        pthread\_mutex\_lock(&mutex); // Lock the buffer

        // Critical section: add the item to the buffer

        buffer[in]=item;

        printf("Producer produced %d\n",item);

        in=(in+1)%BUFFER\_SIZE;

        pthread\_mutex\_unlock(&mutex); // Unlock the buffer

        sem\_post(&full); // Signal that a new item has been added

        sleep(1); // Simulate time taken to produce an item

    }

}

void \*consumer(void \*arg)

{

    int item;

    while(1)

    {

        sem\_wait(&full); // Wait for a full slot

        pthread\_mutex\_lock(&mutex); // Lock the buffer

        // Critical section: remove the item from the buffer

        item=buffer[out];

        printf("Consumer consumed %d\n",item);

        out=(out+1)%BUFFER\_SIZE;

        pthread\_mutex\_unlock(&mutex); // Unlock the buffer

        sem\_post(&empty); // Signal that a slot has been emptied

        sleep(1); // Simulate time taken to consume an item

    }

}

int main() {

    pthread\_t prod\_tid,cons\_tid;

    // Initialize the semaphores

    sem\_init(&empty,0,BUFFER\_SIZE); // All buffer slots are initially empty

    sem\_init(&full,0,0); // No slots are initially full

    pthread\_mutex\_init(&mutex,NULL); // Initialize the mutex

    // Create the producer and consumer threads

    pthread\_create(&prod\_tid,NULL,producer,NULL);

    pthread\_create(&cons\_tid,NULL,consumer,NULL);

    // Join the threads (this program will run indefinitely)

    pthread\_join(prod\_tid,NULL);

    pthread\_join(cons\_tid,NULL);

    // Destroy the semaphores and mutex

    sem\_destroy(&empty);

    sem\_destroy(&full);

    pthread\_mutex\_destroy(&mutex);

    return 0;

}

34. setsid

#include<unistd.h>//header file

#include<sys/types.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

    pid\_t pid;

    int p[2];

    char c='?';

    if(pipe(p)!=0)

    {

        printf("pipe() error");

    }

    if((pid)=fork()==0)

    {

        printf("child process group id is %d\n",(int)getpgrp());

        write(p[1],&c,1);

        //it retrun group id

        printf("Now we make a new group that child has group leader %d\n",setsid());

        printf("child process group id is now %d\n",(int)getpgrp());

        exit(0);

    }

    else

    {

        printf("parent process group id is %d\n",(int)getpgrp());

        read(p[0],&c,1);

        sleep(5);

    }

}

35. getpgrp

#include<unistd.h>//header file

#include<sys/wait.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

    int status;

    int pid=fork();

    if(pid==0)

    {

        int cid=fork();

        if(cid==0)

        {

            printf("grandchild pid is : %d and process group id is: %d\n",getpid(),getpgrp());

            exit(0);

        }

        printf("child pid is: %d and process group id is: %d\n",getpid(),getpgrp());

        wait(&status);

        exit(0);

    }

    printf("parent pid is: %d and process group id is: %d\n",getpid(),getpgrp());

    printf("the parent's parent pid is: %d",getppid());

    wait(&status);

}

36. getpgrp()==getpgid(0)

#include<unistd.h>//header file

#include<sys/wait.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

    int status;

    int pid=fork();

    if(pid==0)

    {

        printf("child process pid: %d\n",getpid());

        printf("process group id using getpgrp: %d\n",getpgrp());

        printf("process group id using getpgid(0): %d\n",getpgid(0));

        exit(0);

    }

    else

    {

        wait(NULL);

        printf("parent process pid: %d\n",getpid());

        printf("process group id using getpgrp(): %d\n",getpgrp());

        printf("process group id using getpgid(0): %d\n",getpgid(0));

    }

    return 0;

}

37. setpgrp

#include<unistd.h>//header file

#include<unistd.h>

#include<stdio.h>

#include <sys/wait.h>

int main()

{

    int status;

    int pid=fork();

    if (pid==0)

    {

        printf("\nChild process pid : %d\n",getpid());

        printf("Process group id using getpgrp() : %d\n",getpgrp());

        //here problem it print 0 in place of group ID

        printf("Process group id after setpgrp() : %d\n",setpgrp());

        printf("Process group id using getpgrp() after setpgrp() : %d\n",getpgrp());

    }

    else

    {

        wait(NULL);

        printf("\nParent process pid : %d\n",getpid());

        printf("Process group id using getpgrp() : %d\n",getpgrp());

    }

}

38. setpgrp()==setpgid(0,0)

#include<unistd.h>//header file

#include<sys/wait.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

    int status;

    int pid = fork();

    if (pid == 0)

    {

        printf("\nChild process pid : %d\n",getpid());

        printf("Process group id using getpgrp() : %d\n",getpgrp());

        printf("Process group id after setpgrp() : %d\n",setpgrp());

        printf("Process group id after setpgid(0,0) : %d\n",setpgid(0,0));

        printf("Process group id using getpgrp() after setpgid(0,0) : %d\n",getpgrp());

    }

    else

    {

        wait(NULL);

    }

}

39. setpgid

#include<unistd.h>//header file

#include<sys/types.h>

#include<stdlib.h>

#include<stdio.h>

int main()

{

    pid\_t pid;

    int p1[2],p2[2];

    char c='?';

    if(pipe(p1)!=0)

    {

        printf("pipe() #1 error");

    }

    else if(pipe(p2)!=0)

    {

        printf("pipe() #2 error");

    }

    else if((pid=fork())==0)

    {

        printf("child process group id %d\n",(int)getpgrp());

        write(p2[1],&c,1);

        printf("child is waiting for parent to complete task\n");

        read(p1[0],&c,1);

        printf("child process group id is now %d\n",(int)getpgrp());

        exit(0);

    }

    else

    {

        printf("parent process group id is %d\n",(int)getpgrp());

        read(p2[0],&c,1);

        printf("parent is performing setpgid() on pid %d\n",(int)pid);

        //pid,processgrp id

        if(setpgid(pid,0)!=0)

        {

            printf("setpgid() error");

        }

        write(p1[1],&c,1);

        printf("parent process group id is now %d\n",(int)getpgrp());

        sleep(5);

    }

}

40.pipe

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>//header file

#include<sys/wait.h>

int main()

{

    int fd[2];

    if(pipe(fd)==-1)

    {

        printf("An error ocurred with opening the pipe\n");

        return 1;

    }

    int id=fork();

    //child process

    if(id==0)

    {

        close(fd[0]);

        int data;

        printf("Input a no: ");

        scanf("%d",&data);

        write(fd[1],&data,sizeof(int));

        close(fd[1]);

    }

    //parent process

    else

    {

        close(fd[1]);

        int buffer;

        read(fd[0],&buffer,sizeof(int))==-1;

        printf("Got from child process %d\n",buffer);

        printf("Result is %d\n",buffer);

        close(fd[0]);

    }

    return 0;

}

41.pipesend\_data\_from\_parent\_to\_child\_over\_a\_pipe

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>//header file

#include<sys/wait.h>

#define MAXLINE 100

int main()

{

    int n;

    int fd[2];

    pid\_t pid;

    char line[MAXLINE];

    if(pipe(fd)<0)

    printf("pipe error");

    pid=fork();

    if(pid>0)

    {

        close(fd[0]);

        write(fd[1],"hello world \n",12);

    }

    else{

        close(fd[1]);

        n=read(fd[0],line,MAXLINE);

        //To print how much reader read

        write(STDOUT\_FILENO,line,n);

    }

    exit(0);

}

42.filo

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/types.h>//header file

#include <sys/stat.h>

#define FIFO\_NAME "myfifo"

#define BUFFER\_SIZE 256

void writer()

{

    int fd;

    char buffer[BUFFER\_SIZE];

    // Open the FIFO for writing

    fd = open(FIFO\_NAME, O\_WRONLY);

    if (fd == -1) {

        perror("open");

        exit(EXIT\_FAILURE);

    }

    // Write data to the FIFO

    strcpy(buffer,"This is the content of the file.\n");

    write(fd,buffer,strlen(buffer));

    // Close the FIFO

    close(fd);

}

void reader()

{

    int fd;

    char buffer[BUFFER\_SIZE];

    ssize\_t num\_bytes;

    // Open the FIFO for reading

    fd = open(FIFO\_NAME, O\_RDONLY);

    if(fd == -1)

    {

        perror("open");

        exit(EXIT\_FAILURE);

    }

    // Read data from the FIFO

    num\_bytes=read(fd, buffer, BUFFER\_SIZE);

    if (num\_bytes == -1)

    {

        perror("read");

        close(fd);

        exit(EXIT\_FAILURE);

    }

    // Null-terminate the string

    buffer[num\_bytes]='\0';

    // Print the content of the file

    printf("File content:\n%s", buffer);

    // Close the FIFO

    close(fd);

}

int main()

{

    // Create the FIFO if it does not exist

    if(mkfifo(FIFO\_NAME,0666)==-1)

    {

        perror("mkfifo");

        exit(EXIT\_FAILURE);

    }

    pid\_t pid=fork();

    if(pid==-1)

    {

        perror("fork");

        exit(EXIT\_FAILURE);

    }

    else if(pid==0)

    {

        // Child process: reader

        reader();

    }

    else

    {

        // Parent process: writer

        writer();

    }

    // Remove the FIFO

    unlink(FIFO\_NAME);

    return 0;

}

43.reader\_writer\_mutex

#include<unistd.h>

#include<stdio.h>

#include<pthread.h>//header file

#include<sys/types.h>

#include<stdlib.h>

pthread\_mutex\_t reader;

pthread\_mutex\_t writer;

int read\_count=0;

int count=0;

void \*wr()

{

  for(int i=0;i<10;i++)

  {

    pthread\_mutex\_lock(&writer);

    printf("Writer entered\n");

    count++;

    printf("Write content\n");

    sleep(1);

    pthread\_mutex\_unlock(&writer);

    printf("Writer exit\n");

  }

  return NULL;

}

void \*re()

{

  for(int i=0;i<10;i++)

  {

    pthread\_mutex\_lock(&reader);

    read\_count++;

    if(read\_count==1)

    {

      pthread\_mutex\_lock(&writer);

    }

    pthread\_mutex\_unlock(&reader);

    printf("Reader entered\n");

    sleep(1);

    printf("Read content\n");

    pthread\_mutex\_lock(&reader);

    read\_count--;

    if(read\_count==0)

    {

      pthread\_mutex\_unlock(&writer);

    }

    pthread\_mutex\_unlock(&reader);

    printf("Reader exit\n");

  }

  return NULL;

}

int main()

{

  pthread\_t th[5];

  // Creating a 5 thread 3 reader 2 writer

  pthread\_mutex\_init(&reader, NULL);

  pthread\_mutex\_init(&writer,NULL);

  for(int i=0;i<5;i++)

  {

    if(i<=1)

    {

      pthread\_create(&th[i],NULL,&wr,NULL);

    }

    else

    {

      pthread\_create(&th[i],NULL,&re,NULL);

    }

  }

  for (int i=0;i<5;i++)

  {

    pthread\_join(th[i],NULL);

  }

  pthread\_mutex\_destroy(&reader);

  pthread\_mutex\_destroy(&writer);

}

44.reader\_writer\_rwlock

#include<stdio.h>

#include<pthread.h>//header file

#include<sys/types.h>

#include<unistd.h>

pthread\_rwlock\_t reader1;

pthread\_rwlock\_t writer1;

int read\_count=0;

int count=0;

void\* writer()

{

    for(int i=0;i<10;i++)

    {

       pthread\_rwlock\_wrlock(&writer1);

       printf("Writer entered\n");

       count++;

       printf("Write content\n");

       sleep(1);

       pthread\_rwlock\_unlock(&writer1);

       printf("Writer exit\n");

    }

}

void\* reader()

{

    for(int i=0;i<10;i++)

  {

    pthread\_rwlock\_wrlock(&reader1);

    read\_count++;

    if(read\_count==1)

    {

      pthread\_rwlock\_wrlock(&writer1);

    }

    pthread\_rwlock\_unlock(&reader1);

    printf("Reader entered\n");

    sleep(1);

    printf("Read content\n");

    pthread\_rwlock\_wrlock(&reader1);

    read\_count--;

    if(read\_count==0)

    {

      pthread\_rwlock\_unlock(&writer1);

    }

    pthread\_rwlock\_unlock(&reader1);

    printf("Reader exit\n");

  }

  return NULL;

}

int main()

{

    pthread\_t th[5];

  //Creating a 5 thread 2 reader 3 writer

    pthread\_rwlock\_init(&reader1,NULL);

    pthread\_rwlock\_init(&writer1,NULL);

    for(int i=0;i<5;i++)

    {

        if(i<=1)

        {

          pthread\_create(&th[i],NULL,&writer,NULL);

        }

        else

        {

            pthread\_create(&th[i],NULL,&reader,NULL);

        }

    }

    for (int i=0;i<5;i++)

    {

        pthread\_join(th[i],NULL);

    }

    pthread\_rwlock\_destroy(&reader1);

    pthread\_rwlock\_destroy(&writer1);

}

45.reader\_writer\_semaphore

#include <stdio.h>

#include <pthread.h>

#include <sys/types.h>

#include<stdlib.h>

#include<semaphore.h>//header file

#include<unistd.h>

sem\_t read1;

sem\_t write1;

int read\_count=0;

int count = 0;

void \*writer()

{

  for (int i=0;i<10;i++)

  {

    sem\_wait(&write1);

    printf("Writer entered\n");

    count++;

    printf("Write content\n");

    sleep(1);

    sem\_post(&write1);

    printf("Writer exit\n");

  }

}

void \*reader()

{

  for(int i=0;i<10;i++)

  {

    sem\_wait(&read1);

    read\_count++;

    if(read\_count==1)

    {

      sem\_wait(&write1);

    }

    sem\_post(&read1);

    printf("Reader entered\n");

    sleep(1);

    printf("Read content\n");

    sem\_wait(&read1);

    read\_count--;

    if(read\_count==0)

    {

      sem\_post(&write1);

    }

    sem\_post(&read1);

    printf("Reader exit\n");

  }

}

int main()

{

  pthread\_t th[5];

  // Creating a 5 thread 2 reader 3 writer

  sem\_init(&read1,0,1);

  sem\_init(&write1,0,1);

  for (int i=0;i<5;i++)

  {

    if (i<=1)

    {

      pthread\_create(&th[i],NULL,&writer,NULL);

    }

    else

    {

      pthread\_create(&th[i],NULL,&reader,NULL);

    }

  }

  for (int i=0;i<5;i++)

  {

    pthread\_join(th[i],NULL);

  }

  sem\_destroy(&read1);

  sem\_destroy(&write1);

}