**Part A : 1**

**Problem definition:**

Write a C program to schedule given set of processes using shortest remaining time first algorithm, given ( pid, arrival time , cpu burst time) for set of processes. Calculate average waiting time and average turnaround time .

**Description: Shortest remaining time first (SRTN)**

This is a preemptive version of shortest job first, in which the scheduler always dispatches that ready process which has the shortest expected remaining time to completion. The dispatching decision is always made when a new process is submitted: the currently running process has obviously a smaller remaining time than all other ready ones, otherwise it wouldn't have been dispatched. But a newly submitted job may have an even shorter one, in which case the currently running one would be blocked and put in ready state.

This decreases the average turnaround time since short, newly arrived processes are given immediate attention, if their expected completion time happens to be less than the remaining time to completion of the currently running and ready processes. As with SJF, there's a risk of starvation for long processes, in presence of a constant supply of short ones.

**Input:**

Set of processes where each process information is (pid, arrival time, cpu burst time)

**Output:**

Table showing details of each process as –pid, arrival time, cpu burst time, completion time, turnaround time, waiting time.

Average turn around time= ∑turnaround time/ no. of processes

Average waiting time =∑waiting time/ no. of processes

**Pseudo code :**

1) For all processes in the list , on arrival of new process select the process with shortest remaining time .

2) On completion of each process record completion time and calculate turn around and

waiting time.

3) Display table.

4) Calculate and display Average turn around time and Average waiting time .

5) Stop.

**Part A : 2**

**Problem definition:**

Write a C program to implement worst fit algorithm given available space information and sequence of memory requests. If not possible to allocate space perform memory compaction and then allocate space if possible.

**Description : Worst fit allocation**

Worst fit allocation always allocates the largest block, whose size is greater than reqiredsize from list of free blocks hoping that the remainder of the block will be useful for servicing a future request.

**Input:**

1. Total space of the system.
2. Free space data for each block – (start address and size of block) in the increasing order of

start address.

1. Sequence **of memory requests .**

**Output**:

Available space information if allocation is made for each request. Otherwise failure message.

**Pseudo code:**

1. Accept total space , available space and sequence of memory requests.
2. For each request
3. Select the free block of largest size which is greater than request and create new free block of remaining size after allocation. Display free space table.
4. If not possible to allocate

Perform memory compaction.

Allocate space if possible.

Display free space table.

Otherwise

Display failure message.

1. Stop.

Part A : 3

**Problem definition:**

Given main memory size and page size in terms of word, sequence of program words referred , write a C program to implement optimal page replacement and find no. of page faults

**Description :Optimal page replacement**

On page fault if no free frame is available replace the page that will be referenced furthest in the future or not at all

**Input:**

1.Main memory space allocated for process in terms of words.

2. Page size, program size

3. Sequence of program words referred.

**Output:**

Reference string along with page faults ad total number of page faults.

**Pseudo code:**

1.Accept allocated main memory space ,page size, program size and sequence of program words

referred.

2.Calculate no. of frames allocated , no. of pages of program and reference string.

3.For each entry of reference string

If (page fault and no free frames available )

1. replace the page whose next reference is farthest in the future or not at all
2. increment no. of page faults by 1.

4.Display no. of page faults along with reference string.

5.Stop.

**Part A: 4**

**Problem definition:**

Given resource allocation information of a system, where each resource is having single instance, write a C program to check whether deadlock is existing in a system using wait- for –graph concept.Display deadlocked processes if deadlock exists.

**Description :**

**Input:**

Wait for graph data in the form of matrix.

**Output:**

If deadlock exists , display list of deadlocked processes.

Otherwise no deadlock message.

**Pseudo code:**

1. Accept matrix for wait- for-graph.
2. Detect cycle in the graph.
3. If cycle exists Display deadlocked processes on cycle

Otherwise

Display – no deadlock detected

1. Stop.

Part A: 5

**Problem definition:**

Given system status about resources, write a C program to check whether system is in safe state or not and given a resource request of a process to check whether resources can be granted immediately or not using bankers algorithm.

**Description : Banker’s Algorithm**

It is the method to avoid deadlock occurrence. It tentatively grant each resource request and analyze resulting system state to see if it is “safe”. If safe, it grants the request. If unsafe refuse the request (undo the tentative grant) and block the requesting process until it is safe to grant it.

**Input:**

1)n = number of processes,

m = number of resource types

2) Available: Vector of length m. If Available [j] = k, there are k instances of resource type Rj currently available

3) Max: n x m matrix. If Max [i,j] = k, then process Pi will request at most k instances of resource type Rj.

4) Alloc: n x m matrix. If Alloc[i,j] = k then Pi is currently allocated (i.e. holding) k instances of Rj.

**Output:**

If system is in safe state display safe sequence and new system state.

Otherwise display not a safe state and no allocation.

**Pseudo code:**

1. Accept System state- n, m, available, max, alloc .
2. Accept resource request of a process.
3. Calculate need matrix.
4. Call resource request algorithm.
5. Call safety algorithm.
6. If new state is safe state allocate requested resource and display safe sequence ,

new state of system.

Otherwise display unsafe state and not possible to allocate.

1. Stop.

**Safety Algorithm:**

1.Let Work and Finish be vectors of length m and n, respectively. Initialize:

Work := Available

Finish [i] == false for i = 1,2, …, n.

2. Find an i such that both:

Finish [i] == false

Need[i]<=Work

If no such i exists, go to step 4.

3.Work:= Work + Allocationi  
 (Resources freed when process completes!)

Finish[i] := true

go to step 2.

4. If Finish [i] = true for all i, then the system is in a safe state.

**Resource Request Algorithm:**

Request[i] = request vector for Pi .

Requesti [j] = k means process Pi wants k instances of resource

typeRj.

1. If Request[i]<=Need[i] go to step 2. Otherwise, error ( process exceeded its maximum claim).

2. If Request[i]<= Available, go to step 3. Otherwise Pi must wait, (resources not available).

3. “Allocate” requested resources to Pi as follows:

Available := Available – Request[i]

Alloc[i] := Alloc[i] + Request[i]

Need[i] := Need[i] – Request[i]

If safe then the resources are allocated to Pi.

If unsafe restore the old resource-allocation state and block Pi

Part A: 6

**Problem definition:**

Write a C program to book seats at two locations using Peterson’s algorithms.

**Description :Peterson’s Algorithm**

When multiple processes try to access the shared data concurrently , race condition occurs. Data access synchronization is used to access shared data in an mutually exclusive manner. It avoids race condition and safe guards consistency of shared data.

Data access synchronization is achieved by defining critical sections in processes. And Petersons algorithm is algorithmic method to achieve Data access synchronization.

**System calls used:**

NAME

shmget - allocates a shared memory segment

SYNOPSIS

#include <sys/ipc.h>

#include <sys/shm.h>

intshmget(key\_t key, size\_t size, intshmflg);

DESCRIPTION

shmget() returns the identifier of the shared memory segment associated with the value of the argument key. A new shared memory segment, with size equal to the value of size rounded up to a multiple of PAGE\_SIZE, is created if key has the value IPC\_PRIVATE or key isn’t IPC\_PRIVATE, no shared memory segment corresponding to key exists, and IPC\_CREAT is specified in shmflg.

If shmflg specifies both IPC\_CREAT and IPC\_EXCL and a shared memory segment already exists for key, then shmget() fails with errno set to EEXIST. (This is analogous to the effect of the combination O\_CREAT | O\_EXCL for open(2).)

shmop - shared memory operations

SYNOPSIS

#include <sys/types.h>

#include <sys/shm.h>

void \*shmat(intshmid, const void \*shmaddr, intshmflg);

intshmdt(const void \*shmaddr);

DESCRIPTION

shmat() attaches the shared memory segment identified by shmid to the address space of the calling process. The attaching address is specified by shmaddr with one of the following criteria:

If shmaddr is NULL, the system chooses a suitable (unused) address at which to attach the segment.

If shmaddr isn’t NULL and SHM\_RND is specified in shmflg, the attach occurs at the address equal to shmaddr rounded down to the nearest multiple of SHMLBA. Otherwise shmaddr must be a page-aligned address at which the attach occurs.

If SHM\_RDONLY is specified in shmflg, the segment is attached for reading and the process must have read permission for the segment. Otherwise the segment is attached for read and write and the process must have read and write permission for the segment. There is no notion of a write-only shared memory segment.

The (Linux-specific) SHM\_REMAP flag may be specified in shmflg to indicate that the mapping of the segment should replace any existing mapping in the range starting at shmaddr and continuing for the size of the segment. (Normally an EINVAL error would result if a mapping already exists in this address range.) In this case, shmaddr must not be NULL.

The brk(2) value of the calling process is not altered by the attach. The segment will automatically be detached at process exit. The same segment may be attached as a read and as a read-write one, and more than once, in the process’s address space.

**Input:--**

**Output:**

Display of seats booked by each process as time passes.

**Pseudo code:**

1. Get shared memory for

Boolean flag[2] = {false, false};

int turn, nextseatno;

1. In process P0

While(1)

flag[0] = true;

turn = 1;

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

// critical section

To allocate seats at location P0

...

// end of critical section

flag[0] = false;

whileend

terminate a process P0.

1. In process P1

While(1)

flag[1] = true;

turn = 0;

while (flag[0] && turn == 0); // critical section

To allocate seats at location P1 // end of critical section

flag[1] = false

whileend

terminate a process P1.

**Critical section for P0and P1:**

1. If seats are available book requested number of seats and update nextseatno
2. Display booked seats at that place.

Part A: 7

**Problem definition:**

Write a C program to implement Producer and consumer problem with buffer of finite size

where produced items must be consumed in FIFO manner, using semaphores.

**Description :Producer and Consumer problem**

Well-known problems in concurrent programming, finite-size buffer and two classes of Processes - producers and consumers. Producer puts an item into the buffer and Consumer takes an item out of the buffer .A producer must wait until the buffer has space before it can put something in, and a consumer must wait until something is in the buffer before it can take something out. Producer and consumer must access buffer in a mutually exclusive manner. It can be achieved using counting semaphores – full and empty and binary semaphore – mutex.

**System calls used:**

NAME

sem\_init - initialise an unnamed semaphore (REALTIME)

SYNOPSIS

#include <semaphore.h>

intsem\_init(sem\_t \*sem, intpshared, unsigned int value);

DESCRIPTION

The sem\_init() function is used to initialise the unnamed semaphore referred to by sem. The value of the initialised semaphore is value. Following a successful call to sem\_init(), the semaphore may be used in subsequent calls to sem\_wait(), sem\_trywait(), sem\_post(), and sem\_destroy(). This semaphore remains usable until the semaphore is destroyed.

If the pshared argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore sem can use sem for performing sem\_wait(), sem\_trywait(), sem\_post(), and sem\_destroy() operations.

Only sem itself may be used for performing synchronisation. The result of referring to copies of sem in calls to sem\_wait(), sem\_trywait(), sem\_post(), and sem\_destroy(), is undefined.

If the pshared argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use sem for performing sem\_wait(), sem\_trywait(), sem\_post(), and sem\_destroy() operations. The use of the semaphore by threads other than those created in the same process is undefined.

Attempting to initialise an already initialised semaphore results in undefined behaviour.

RETURN VALUE

Upon successful completion, the function initialises the semaphore in sem. Otherwise, it returns -1 and sets errno to indicate the error.

ERRORS

The sem\_init() function will fail if:

[EINVAL]

The value argument exceeds SEM\_VALUE\_MAX.

[ENOSPC]

A resource required to initialise the semaphore has been exhausted, or the limit on semaphores (SEM\_NSEMS\_MAX) has been reached.

[ENOSYS]

The function sem\_init() is not supported by this implementation.

[EPERM]

The process lacks the appropriate privileges to initialise the semaphore.

NAME

sem\_post - unlock a semaphore (REALTIME)

SYNOPSIS

#include <semaphore.h>

intsem\_post(sem\_t \*sem);

DESCRIPTION

The sem\_post() function unlocks the semaphore referenced by sem by performing a semaphore unlock operation on that semaphore.

If the semaphore value resulting from this operation is positive, then no threads were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

If the value of the semaphore resulting from this operation is zero, then one of the threads blocked waiting for the semaphore will be allowed to return successfully from its call to sem\_wait(). If the symbol \_POSIX\_PRIORITY\_SCHEDULING is defined, the thread to be unblocked will be chosen in a manner appropriate to the scheduling policies and parameters in effect for the blocked threads. In the case of the schedulers SCHED\_FIFO and SCHED\_RR, the highest priority waiting thread will be unblocked, and if there is more than one highest priority thread blocked waiting for the semaphore, then the highest priority thread that has been waiting the longest will be unblocked. If the symbol \_POSIX\_PRIORITY\_SCHEDULING is not defined, the choice of a thread to unblock is unspecified.

The sem\_post() interface is reentrant with respect to signals and may be invoked from a signal-catching function.

RETURN VALUE

If successful, the sem\_post() function returns zero; otherwise the function returns -1 and sets errno to indicate the error.

ERRORS

The sem\_post() function will fail if:

[EINVAL]

The sem does not refer to a valid semaphore.

[ENOSYS]

The function sem\_post() is not supported by this implementation.

NAME

sem\_wait, sem\_trywait - lock a semaphore (REALTIME)

SYNOPSIS

#include <semaphore.h>

intsem\_wait(sem\_t \*sem);

intsem\_trywait(sem\_t \*sem);

DESCRIPTION

The sem\_wait() function locks the semaphore referenced by sem by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread will not return from the call to sem\_wait() until it either locks the semaphore or the call is interrupted by a signal. The sem\_trywait() function locks the semaphore referenced by sem only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.

Upon successful return, the state of the semaphore is locked and remains locked until the sem\_post() function is executed and returns successfully.

The sem\_wait() function is interruptible by the delivery of a signal.

RETURN VALUE

The sem\_wait() and sem\_trywait() functions return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by sem. If the call was unsuccessful, the state of the semaphore is unchanged, and the function returns a value of -1 and sets errno to indicate the error.

ERRORS

The sem\_wait() and sem\_trywait() functions will fail if:

[EAGAIN]

The semaphore was already locked, so it cannot be immediately locked by the sem\_trywait() operation ( sem\_trywait only).

[EINVAL]

The sem argument does not refer to a valid semaphore.

[ENOSYS]

The functions sem\_wait() and sem\_trywait() are not supported by this implementation.

The sem\_wait() and sem\_trywait() functions may fail if:

[EDEADLK]

A deadlock condition was detected.

[EINTR]

A signal interrupted this function.

NAME

sem\_destroy - destroy an unnamed semaphore (REALTIME)

SYNOPSIS

#include <semaphore.h>

intsem\_destroy(sem\_t \*sem);

DESCRIPTION

The sem\_destroy() function is used to destroy the unnamed semaphore indicated by sem. Only a semaphore that was created using sem\_init() may be destroyed using sem\_destroy(); the effect of calling sem\_destroy() with a named semaphore is undefined. The effect of subsequent use of the semaphore sem is undefined until sem is re-initialised by another call to sem\_init().

It is safe to destroy an initialised semaphore upon which no threads are currently blocked. The effect of destroying a semaphore upon which other threads are currently blocked is undefined.

RETURN VALUE

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned and errno is set to indicate the error.

ERRORS

The sem\_destroy() function will fail if:

[EINVAL]

The sem argument is not a valid semaphore.

[ENOSYS]

The function sem\_destroy() is not supported by this implementation.

The sem\_destroy() function may fail if:

[EBUSY]

There are currently processes blocked on the semaphore.

**Input:** --

**Output:**

Display of item placed in the buffer by producer and item consumed by consumer for some period of time.

**Pseudo code:**

1. Create shared memory for buffer of finite size.
2. Create shared memory for semaphores full, empty , and mutex and initialize .
3. If (producer)

While (1)

Wait(empty)

Wait(mutex)

Produce item and place into buffer at next position

Signal(mutex)

Signal( full)

Whileend

1. If (consumer)

While (1)

Wait(full)

Wait(mutex)

Consume an item from buffer at next position

1. Signal(mutex)

Signal(mutex)

Signal(empty)

1. Whileend
2. Stop.

Part A: 8

**Problem definition:**

Write a C program to sort given set of numbers using parent and child processes. Let each process sort half of the list. When child process signals about completion of sorting, parent process goes for merging both half lists and when parent signals child process about completion of merging , child process display the complete sorted list.

**Description :Control Synchronization**

Control synchronization is required if process should perform some action ai only when other process has performed action aj. This can be achieved using signalling using binary semaphores.

**Input:**

List size and list of numbers.

**Output:**

Sorted second half list by child process

Sorted firsthalf list by parent process

Sorted complete list by parent process

Sorted complete list by child process

**Pseudo code:**

1.Get shared memory for list and semaphores.

2.Read list size and list of numbers.

3.Initialize semaphores.

4.Create child process.

5.If (child )

1. Sort second half of the list. Display second half sorted list.
2. Send signal to parent about completion of sorting.
3. Wait for signal from parent about completion of merging
4. Display complete sorted list
5. Terminate.

6.If ( parent )

1. Sort first half of the list. Display first half sorted list.
2. wait for completion of sorting by child.

iii) merge both half lists.

1. Send signalto child about completion of merging
2. Display complete sorted list
3. Terminate.

Part B: 1

**Problem definition:**

Write a C program to create a child process and execute new

program in child process using exec() .

**Description : Multiple processes:**

Multiple processes can be created in an application using fork( ) system call. Child is duplicate of parent and executes same code as that of parent. New program can be executed using exec( ) system call in a process. Processes can display their pids and parent pids using getpid( ) and getppid( ) system calls respectively.

**System calls used:**

1.fork - create a child process

SYNOPSIS

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

DESCRIPTION

fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.

Under Linux, fork() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent’s page tables, and to create a unique task structure for the child.

RETURN VALUE

On success, the PID of the child process is returned in the parent’s thread of execution, and a 0 is returned in the child’s thread of execution. On failure, a -1 will be returned in the parent’s context, no child process will be created, and errno will be set appropriately.

ERRORS

EAGAIN : fork() cannot allocate sufficient memory to copy the parent’s page tables and allocate a task structure for the child.

EAGAIN : It was not possible to create a new process because the caller’s RLIMIT\_NPROC resource limit was encountered. To exceed this limit, the process must have either the CAP\_SYS\_ADMIN or the CAP\_SYS\_RESOURCE capability.

ENOMEM: fork() failed to allocate the necessary kernel structures because memory is tight.

2) getpid, getppid - get process identification

SYNOPSIS

#include <sys/types.h>

#include <unistd.h>

pid\_tgetpid(void);

pid\_tgetppid(void);

DESCRIPTION

getpid() returns the process ID of the current process. (This is often used by routines that generate unique temporary filenames.)

getppid() returns the process ID of the parent of the current process.

3) execl, execlp, execle, execv, execvp, execvpe - execute a file

Synopsis

#include <unistd.h>

extern char \*\*environ;

intexecl(const char \*path, const char \*arg, ...);

intexeclp(const char \*file, const char \*arg, ...);

intexecle(const char \*path, const char \*arg,

..., char \* constenvp[]);

intexecv(const char \*path, char \*constargv[]);

intexecvp(const char \*file, char \*constargv[]);

intexecvpe(const char \*file, char \*constargv[],

char \*constenvp[]);

Description

The exec() family of functions replaces the current process image with a new process image. The functions described in this manual page are front-ends for execve(2). (See the manual page for execve(2) for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed.

The const char \*arg and subsequent ellipses in the execl(), execlp(), and execle() functions can be thought of as arg0, arg1, ..., argn. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments must be terminated by a NULL pointer, and, since these are variadic functions, this pointer must be cast (char \*) NULL.

The execv(), execvp(), and execvpe() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers must be terminated by a NULL pointer.

The execle() and execvpe() functions allow the caller to specify the environment of the executed program via the argument envp. The envp argument is an array of pointers to null-terminated strings and must be terminated by a NULL pointer. The other functions take the environment for the new process image from the external variable environ in the calling process.

Special semantics for execlp() and execvp()

The execlp(), execvp(), and execvpe() functions duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character. The file is sought in the colon-separated list of directory pathnames specified in the PATH environment variable. If this variable isn't defined, the path list defaults to the current directory followed by the list of directories returned by confstr(\_CS\_PATH). (This confstr(3) call typically returns the value "/bin:/usr/bin".)

If the specified filename includes a slash character, then PATH is ignored, and the file at the specified pathname is executed.

In addition, certain errors are treated specially.

If permission is denied for a file (the attempted execve(2) failed with the error EACCES), these functions will continue searching the rest of the search path. If no other file is found, however, they will return with errno set to EACCES.

If the header of a file isn't recognized (the attempted execve(2) failed with the error ENOEXEC), these functions will execute the shell (/bin/sh) with the path of the file as its first argument. (If this attempt fails, no further searching is done.)

Return Value

The exec() functions only return if an error has occurred. The return value is -1, and errno is set to indicate the error.

Errors

All of these functions may fail and set errno.

**Input:--**

**Output**:

Display of parent process id and its child pid.

And output generated by the program executed in child process.

**Pseudo code:**

1. Create child process using fork () .
2. Display parent and child process ids.
3. If child process

Execute new program.

1. Terminate.

Part B: 2

**Problem definition:**

Write a C program to display the content of a given directory and count number of regular and directory files .

**Description :**

Linux defines different types of files. Directory is also file consisting of list of structures named directory entry –( name , inode number ).With file name as an argument attributes of the file can be read in structure of type struct stat. By checking file attribute for type of file different types of files in a directory can be counted.

**System calls used:**

**1)opendir, fdopendir - open a directory**

Synopsis

#include <sys/types.h>

#include <dirent.h>

DIR \*opendir(const char \*name);

DIR \*fdopendir(intfd);

Description

The opendir() function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

The fdopendir() function is like opendir(), but returns a directory stream for the directory referred to by the open file descriptor fd. After a successful call to fdopendir(), fd is used internally by the implementation, and should not otherwise be used by the application.

Return Value

The opendir() and fdopendir() functions return a pointer to the directory stream. On error, NULL is returned, and errno is set appropriately.

2)stat, fstat, lstat - get file status

SYNOPSIS

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

int stat(const char \*path, struct stat \*buf);

intfstat(intfiledes, struct stat \*buf);

intlstat(const char \*path, struct stat \*buf);

DESCRIPTION

These functions return information about a file. No permissions are required on the file itself, but — in the case of stat() and lstat() — execute (search) permission is required on all of the directories in path that lead to the file.

stat() stats the file pointed to by path and fills in buf. lstat() is identical to stat(), except that if path is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat() is identical to stat(), except that the file to be stat-ed is specified by the file descriptor filedes.

All of these system calls return a stat structure, which contains the following fields:

struct stat {

dev\_tst\_dev; /\* ID of device containing file \*/

ino\_tst\_ino; /\* inode number \*/

mode\_tst\_mode; /\* protection \*/

nlink\_tst\_nlink; /\* number of hard links \*/

uid\_tst\_uid; /\* user ID of owner \*/

gid\_tst\_gid; /\* group ID of owner \*/

dev\_tst\_rdev; /\* device ID (if special file) \*/

off\_tst\_size; /\* total size, in bytes \*/

blksize\_tst\_blksize; /\* blocksize for filesystem I/O \*/

blkcnt\_tst\_blocks; /\* number of blocks allocated \*/

time\_tst\_atime; /\* time of last access \*/

time\_tst\_mtime; /\* time of last modification \*/

time\_tst\_ctime; /\* time of last status change \*/

};

The following POSIX macros are defined to check the file type using the st\_mode field:

Tag Description

S\_ISREG(m) is it a regular file?

S\_ISDIR(m) directory?

S\_ISCHR(m) character device?

S\_ISBLK(m) block device?

S\_ISFIFO(m) FIFO (named pipe)?

S\_ISLNK(m) symbolic link? (Not in POSIX.1-1996.)

S\_ISSOCK(m) socket? (Not in POSIX.1-1996.)

**Input:**

Path name of dir

**Output**:

Display all files in the given dir. Display no. of dir. Files and regular files in a given dir.

**Pseudo code:**

1. Accept path name of dir.
2. Open directory using opendir( ).
3. For each entry of dir Display filename.

If reg.file increment regcount by 1.

If dir.file increment dircount by 1.

4) Display no.of reg. files and dir. Files.

5) Stop.

Part B: 3

**Problem definition:**

Write a C program to search student information where child process sends roll number to parent process and parent searches and sends student information to child using pipe.

**Description :Inter process communication through pipe.**

In Unix based operating systems the primary means for inter-process communication is the pipe concept. A pipe is a unidirectional data connection between two processes. A process uses file descriptors to read from or write to a pipe.A pipe is designed as a queue offering buffered communication. Therefore the operating system will receive messages written to the pipe and make them available to entities reading from the pipe. The pipe is referenced using file descriptors for both end points. Each endpoint can have more than one file descriptor associated with it. If two-way communication is desired, two pipes have to be used

**System calls used:**

**1)pipe - create pipe**

SYNOPSIS

#include <unistd.h>

int pipe(intfiledes[2]);

DESCRIPTION

pipe() creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

2)**read - read from a file descriptor**

SYNOPSIS

#include <unistd.h>

ssize\_t read(intfd, void \*buf, size\_t count);

DESCRIPTION

read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

If count is zero, read() returns zero and has no other results. If count is greater than SSIZE\_MAX, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

3)**write - write to a file descriptor**

SYNOPSIS

#include <unistd.h>

ssize\_t write(intfd, const void \*buf, size\_t count);

DESCRIPTION

write() writes up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. POSIX requires that a read() which can be proved to occur after a write() has returned returns the new data. Note that not all file systems are POSIX conforming.

RETURN VALUE

On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately.

If count is zero and the file descriptor refers to a regular file, 0 may be returned, or an error could be detected. For a special file, the results are not portable.

**Input:**

1. Set of student records.
2. Roll number of a student whose details are required.

**Output**:

If student is existing in the list display that student details in child.

**Pseudo code:**

1. Create 2 pipes , one for sending data and another to receive data.
2. Create child process.
3. If (child)

Read rollno.

Write rollno. Into pipe.

Sleep for some time.

Read from pipe to get student details sent from parent.

Display details

Terminate.

1. If ( parent )

Sleep for some time .

Read roll number from pipe.

Search for student details.

If found in the list

Write details into pipe

Otherwise

Send failure message to child through pipe.

Sleep for some time.

Terminate.

1. Stop.

Part B: 4

**Problem definition:**

Write a C program to send file from one process to another using FIFO .

**Description :Inter process communication using FIFO**

One of the major disadvantage of pipes is that the they can not be accesed using their names by any other process other than child and the parent as they do not get listed in the directory tree.

This problemcan be overcome using named pipe - FIFO, which stands for First in First out, meaning the data that is written into the pipe first will be read out first always.

The fifos get listed in the directory tree and any process can access it using its name by providing the approproiate path.

fifo are created using the function mkfifo() which takes as arguments

1. The name of the fifo that has to be created

2. The permissions for the file.

Once the file is created, it needs to be opened using the system call open() and the data can be read and written from the file using read() and write system calls.

**System calls used:**

mkfifo - make a FIFO special file (a named pipe)

Synopsis

#include <sys/types.h>#include <sys/stat.h>

intmkfifo(const char \*pathname, mode\_t mode);

Description

mkfifo() makes a FIFO special file with name pathname. mode specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system by calling mkfifo().

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa. See fifo(7) for nonblocking handling of FIFO special files.

Return Value

On success mkfifo() returns 0. In the case of an error, -1 is returned (in which case, errno is set appropriately).

**Input:**

File name

**Output**:

Display received file content through FIFO in child.

**Pseudo code:**

1. Create FIFO.
2. Create child process.
3. If ( parent )

Read file name.

Open file.

While not eof of file

Read file

Write into FIFO

Sleep for some time.

Terminate.

1. If (child )

Read from FIFO

Display on std.device

Terminate.

1. Stop.

Part B: 5

**Problem definition:**

Write a C program to display access permissions of a given files and to change permissions. Accept file names as command line arguments.

**Description :Basic File Permissions**.

**Permission Groups**

Each file and directory has three user based permission groups:

**owner** - The Owner permissions apply only the owner of the file or directory, they will not impact the actions of other users.

**group** - The Group permissions apply only to the group that has been assigned to the file or directory, they will not effect the actions of other users.

**all users** - The All Users permissions apply to all other users on the system, this is the permission group that you want to watch the most.

Permission Types

Each file or directory has three basic permission types:

**read** - The Read permission refers to a user's capability to read the contents of the file.

**write** - The Write permissions refer to a user's capability to write or modify a file or directory.

**execute** - The Execute permission affects a user's capability to execute a file or view the contents of a directory

Using file name as parameter attributes of the file can be read into struct stat using stat ( ) function. By checking st\_mode field of stat structure access permissions can be displayed.

By using chmod( ), permissions can be changed.

**System calls used:**

NAME

chmod, fchmod - change permissions of a file

SYNOPSIS

#include <sys/types.h>

#include <sys/stat.h>

intchmod(const char \*path, mode\_t mode);

intfchmod(intfildes, mode\_t mode);

DESCRIPTION

The mode of the file given by path or referenced by fildes is changed.

Modes are specified by or’ing the following:

Tag Description

S\_ISUID 04000 set user ID on execution

S\_ISGID 02000 set group ID on execution

S\_ISVTX 01000 sticky bit

S\_IRUSR 00400 read by owner

S\_IWUSR 00200 write by owner

S\_IXUSR 00100 execute/search by owner

S\_IRGRP 00040 read by group

S\_IWGRP 00020 write by group

S\_IXGRP 00010 execute/search by group

S\_IROTH 00004 read by others

S\_IWOTH 00002 write by others

S\_IXOTH 00001 execute/search by others

The effective UID of the calling process must match the owner of the file, or the process must be privileged (Linux: it must have the CAP\_FOWNER capability).

If the calling process is not privileged (Linux: does not have the CAP\_FSETID capability), and the group of the file does not match the effective group ID of the process or one of its supplementary group IDs, the S\_ISGID bit will be turned off, but this will not cause an error to be returned.

As a security measure, depending on the file system, the set-user-ID and set-group-ID execution bits may be turned off if a file is written. (On Linux this occurs if the writing process does not have the CAP\_FSETID capability.) On some file systems, only the superuser can set the sticky bit, which may have a special meaning. For the sticky bit, and for set-user-ID and set-group-ID bits on directories, see stat(2).

On NFS file systems, restricting the permissions will immediately influence already open files, because the access control is done on the server, but open files are maintained by the client. Widening the permissions may be delayed for other clients if attribute caching is enabled on them.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

**Input:**

Absolute or relative path names of the files as command line arguments.

New access permissions of file as 3 digit octal number.

**Output**:

Access permissions of each command line argument.

Changed access permissions.

**Pseudo code:**

1. For each command line parameter stat ( ) file attributes of the file.
2. By checking st\_mode field of stat struct display permissions.
3. By accepting new permissions as octal number using chmod ( ) set new permissions.
4. Display changed permissions.
5. Stop.

Part B: 6

**Problem definition:**

Write a C program to map a given file to memory and modify the memory content. Display modified memory content and file content.

**Description :Mapping Files into Memory**

As an alternative to standard file I/O, the kernel provides an interface that allows an application to map a file into memory, meaning that there is a one-to-one correspondence between a memory address and a word in the file. The programmer can then access the file directly through memory, identically to any other chunk of memory-resident data—it is even possible to allow writes to the memory region to transparently map back to the file on disk.

**System calls used:**

mmap, munmap - map or unmap files or devices into memory

SYNOPSIS

#include <sys/mman.h>

void \*mmap(void \*start, size\_t length, intprot, int flags,

intfd, off\_t offset);

intmunmap(void \*start, size\_t length);

DESCRIPTION

The mmap() function asks to map length bytes starting at offset offset from the file (or other object) specified by the file descriptor fd into memory, preferably at address start. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by mmap().

The prot argument describes the desired memory protection (and must not conflict with the open mode of the file). It is either PROT\_NONE or is the bitwise OR of one or more of the other PROT\_\* flags.

Tag Description

PROT\_EXEC Pages may be executed.

PROT\_READ Pages may be read.

PROT\_WRITE Pages may be written.

PROT\_NONE Pages may not be accessed.

The flags parameter specifies the type of the mapped object, mapping options and whether modifications made to the mapped copy of the page are private to the process or are to be shared with other references. It has bits

Tag Description

MAP\_FIXED Do not select a different address than the one specified. If the memory region specified by start and len overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded. If the specified address cannot be used, mmap() will fail. If MAP\_FIXED is specified, start must be a multiple of the page size. Use of this option is discouraged.

MAP\_SHARED Share this mapping with all other processes that map this object. Storing to the region is equivalent to writing to the file. The file may not actually be updated until msync(2) or munmap(2) are called.

MAP\_PRIVATE Create a private copy-on-write mapping. Stores to the region do not affect the original file. It is unspecified whether changes made to the file after the mmap() call are visible in the mapped region.

**Input:**

Absolute or relative path name of a file

**Output**:

Content of file from memory .

Content of file after updation from memory. Content of file from file system.

**Pseudo code:**

1. Open file in required mode to get file descriptor using open ( ).
2. Using file descriptor map required region of file to memory using mmap ( ).
3. Display memory content of a file.
4. Update memory content of a file.
5. Display updated memory content of file and also file content from file system.
6. Stop.

Part B: 7

**Problem definition:**

Write a C program to create thread and sort given set of numbers using parent thread and child thread.

**Description : Thread**

A thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, ( and a thread ID. )

Traditional ( heavyweight ) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.Multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input ( keystrokes ), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.

**System calls used:**

Name

**pthread\_create -** create a new thread

Synopsis

#include <pthread.h>

intpthread\_create(pthread\_t \*thread, constpthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*), void \*arg);

Compile and link with -pthread.

Description

The pthread\_create() function starts a new thread in the calling process. The new thread starts execution by invoking start\_routine(); arg is passed as the sole argument of start\_routine().

The new thread terminates in one of the following ways:

1)it calls pthread\_exit(3), specifying an exit status value that is available to another thread in the same process that calls pthread\_join(3).

2)It returns from start\_routine(). This is equivalent to calling pthread\_exit(3) with the value supplied in the return statement.

3)It is canceled

Any of the threads in the process calls exit(3), or the main thread performs a return from main(). This causes the termination of all threads in the process.

The attr argument points to a pthread\_attr\_t structure whose contents are used at thread creation time to determine attributes for the new thread; this structure is initialized using pthread\_attr\_init(3) and related functions. If attr is NULL, then the thread is created with default attributes.

Before returning, a successful call to pthread\_create() stores the ID of the new thread in the buffer pointed to by thread; this identifier is used to refer to the thread in subsequent calls to other pthreads functions.

The new thread inherits a copy of the creating thread's signal mask (pthread\_sigmask(3)). The set of pending signals for the new thread is empty (sigpending(2)). The new thread does not inherit the creating thread's alternate signal stack (sigaltstack(2)).

Return Value

On success, pthread\_create() returns 0; on error, it returns an error number, and the contents of \*thread are undefined.

Name

pthread\_attr\_init, pthread\_attr\_destroy - initialize and destroy thread attributes object

Synopsis

#include <pthread.h>

**intpthread\_attr\_init(pthread\_attr\_t \*attr);**

**intpthread\_attr\_destroy(pthread\_attr\_t \*attr);**

Compile and link with -pthread.

Description

The pthread\_attr\_init() function initializes the thread attributes object pointed to by attr with default attribute values. After this call, individual attributes of the object can be set using various related functions (listed under SEE ALSO), and then the object can be used in one or more pthread\_create(3) calls that create threads.

Calling pthread\_attr\_init() on a thread attributes object that has already been initialized results in undefined behavior.

When a thread attributes object is no longer required, it should be destroyed using the pthread\_attr\_destroy() function. Destroying a thread attributes object has no effect on threads that were created using that object.

Once a thread attributes object has been destroyed, it can be reinitialized using pthread\_attr\_init(). Any other use of a destroyed thread attributes object has undefined results.

Return Value

On success, these functions return 0; on error, they return a nonzero error number.

Name

pthread\_attr\_setdetachstate, pthread\_attr\_getdetachstate - set/get detach state attribute in thread attributes object

Synopsis

#include <pthread.h>

intpthread\_attr\_setdetachstate(pthread\_attr\_t \*attrint " detachstate );

intpthread\_attr\_getdetachstate(pthread\_attr\_t \*attrint \*" detachstate);

Compile and link with -pthread.

Description

The pthread\_attr\_setdetachstate() function sets the detach state attribute of the thread attributes object referred to by attr to the value specified in detachstate. The detach state attribute determines whether a thread created using the thread attributes object attr will be created in a joinable or a detached state.

The following values may be specified in detachstate:

PTHREAD\_CREATE\_DETACHED

Threads that are created using attr will be created in a detached state.

PTHREAD\_CREATE\_JOINABLE

Threads that are created using attr will be created in a joinable state.

The default setting of the detach state attribute in a newly initialized thread attributes object is PTHREAD\_CREATE\_JOINABLE.

The pthread\_attr\_getdetachstate() returns the detach state attribute of the thread attributes object attr in the buffer pointed to by detachstate.

Name

pthread\_join - join with a terminated thread

Synopsis

#include <pthread.h>intpthread\_join(pthread\_t thread, void \*\*retval);

Compile and link with -pthread.

Description

The pthread\_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread\_join() returns immediately. The thread specified by thread must be joinable.

If retval is not NULL, then pthread\_join() copies the exit status of the target thread (i.e., the value that the target thread supplied to pthread\_exit(3)) into the location pointed to by \*retval. If the target thread was canceled, then PTHREAD\_CANCELED is placed in \*retval.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling pthread\_join() is canceled, then the target thread will remain joinable (i.e., it will not be detached).

Return Value

On success, pthread\_join() returns 0; on error, it returns an error number.

**Input:**

List of numbers to be sorted.

**Output**:

Half List of numbers sorted by thread 0.

Half list of numbers sorted by thread 1.

Display of complete list of numbers.

**Pseudo code:**

1) Read set of numbers to be sorted.

2) Create thread using pthread\_create( ) function by specifying work function.

3) Sort numbers in thread 0.

4) Display sorted list by thread 0 and thread 1.

5) Merge the numbers.

6) Display complete list of sorted numbers.

7) stop.

Part B: 8

**Problem definition:**

Write a C program to illustrate user defined signal handler for SIGFPE.

**Description : Signals**

In Linux, every signal has a name that begins with characters SIG. For example :

* A SIGINT signal that is generated when a user presses ctrl+c. This is the way to terminate programs from terminal.
* A SIGALRM  is generated when the timer set by alarm function goes off.
* A SIGABRT signal is generated when a process calls the abort function.
* SIGFPE is generated because of floating point error

When the signal occurs, the process has to tell the kernel what to do with it.  There can be three options through which a signal can be disposed :

1. The signal can be ignored. By ignoring we mean that nothing will be done when signal occurs. Most of the signals can be ignored but signals generated by hardware exceptions like divide by zero, if ignored can have weird consequences. Also, a couple of signals like SIGKILL and SIGSTOP cannot be ignored.
2. The signal can be caught. When this option is chosen, then the process registers a function with kernel. This function is called by kernel when that signal occurs. If the signal is non fatal for the process then in that function the process can handle the signal properly or otherwise it can chose to terminate gracefully.
3. Let the default action apply. Every signal has a default action. This could be process terminate, ignore etc.

**System calls used:**

signal - ANSI C signal handling

SYNOPSIS

#include <signal.h>

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(intsignum, sighandler\_t handler);

DESCRIPTION

The signal() system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either SIG\_IGN or SIG\_DFL.

Upon arrival of a signal with number signum the following happens. If the corresponding handler is set to SIG\_IGN, then the signal is ignored. If the handler is set to SIG\_DFL, then the default action associated with the signal (see signal(7)) occurs. Finally, if the handler is set to a function sighandler then first either the handler is reset to SIG\_DFL or an implementation-dependent blocking of the signal is performed and next sighandler is called with argument signum.

Using a signal handler function for a signal is called "catching the signal". The signals SIGKILL and SIGSTOP cannot be caught or ignored.

RETURN VALUE

The signal() function returns the previous value of the signal handler, or SIG\_ERR on error.

**Input:--**

**Output**:

When signal SIGFPE is caught error message is displayed by user defined signal

Handler.

**Pseudo code:**

1) Register signal handler for SIGFPE using signal ( ).

2) perform arithmetic operation which leads to generation of SIGFPE.

3) stop.