Date_____ Page No._____

EXPERIMENT - 01

Objective :

To implement the fork() system call.

FORK Function :

An existing process can create a new one by calling the fork function. The new process created by the fork is called child process. This function is called once but returns twice. The only difference in the return is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. Both the child and the parent continue executing with the instruction that follows the call to fork.

System Call Used :

fork();

System call fork() is used to create process. It takes no arg. and returns a process ID. After the new child process is created both process will execute the next instruction following the fork() system call.

- If fork() returns negative value, the creation of process was unsuccessful.
- If fork() returns zero, then child has created
- If fork() return positive value, which is process ID of child to parent.

Teacher's Signature_____

Mohan

EXPERIMENT - 02

Objective :

Parent process computes sum of odd and child process computes sum of even number using fork.

System Call used :

fork() :

System call fork is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller.

After the new child process is created, both process will execute the next instruction following the fork() system call. This can be done by testing the returned value of fork().

exit() :

ISOC defines exit to provide a way for a process to terminate without returning exit handler or signal handlers. Whether or not standard I/O are flushed depends on the implementation.

Date_____ Page No._____

EXPERIMENT - 03

Objective :

Demonstration of wait() system call.

System Call Used :

fork() :

System call fork() is used to create processes. It takes no arguments and return a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller.

Wait() :

The wait function can block the caller until a child process terminates, If a child process has already terminated and is a zombie, wait returns immediately with that child's status. Otherwise, it blocks the caller until a child terminates. If a caller has multiple children, wait returns when one terminates.

Teacher's Signature_____

EXPERIMENT - 04

Objective :

Implementation of Orphan process and zombie process.

Orphan Process :

An orphan process in unix and unix like operating system, is a computer process whose parent process has finished or terminated, through itself remains running. A process become orphaned during remote invocation when the client process crashes after making a request to a server.
The orphaned process will be immediately adopted by the special 'init' process as its parent.

System Calls Used :

fork() :

System call fork() is used to create processes. It takes no arguments and return a process ID. The purpose of fork() is to create a new process, which become the child process of the caller.

getpid() :

getpid() return the process ID of the calling process. The ID is guarenteed to be unique and is useful for constructing

temporary file name.

getppid () :

getppid() returns the process ID of the calling process

sleep() :

A typical system call takes a time value as a parameter, specifying the minimum amount of time that process is sleep before execution. The parameter typically specifies seconds, although some OS provides finer resolution such as miliseconds or microseconds.

Date_____ Page No._____

Zombie Process : A zombie process is a process that has completed execution but still has an entry in the process table. The entry is still needed to allow the process started the process to read its exit status. The parent can read the childs exit status by executing the wait system call, at which stage the zombie is removed.

System Calls Used

fork() : System call fork() is used to create new processes. It takes no argument and return a process ID. The purpose of fork() is to create a new process, which becomes child process of the caller.

sleep() : A typical sleep call takes a time value as a parameter, specifying the minimum amount of time that process is sleep before execution.

exit() : ISOC defines exit to provide a way for a process to terminate without running exit handler or signal handlers. Whether or not standard I/O stream are flushed depends on the implementations.

EXPERIMENT - 05

Objective :

Implementation of PIPE

PIPE :

Pipes are the oldest form of UNIX System IPC and are provided by UNIX systems. PIPE is created by calling PIPE function. Two file descriptors are returned through the fd argument : fd[0] is open for reading, and fd[1] is open for writing. The output of fd[1] is the input for fd[0].

System Calls used

Pipe (): Pipe is created by calling pipe function. Two file descriptors are returned through file descriptors argument : fd[0] is open for reading and fd[1] is open for writing. The o/p of fd[1] is i/p for fd[0].

fork(): System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller.

- If fork() return a negative value, the creation was unsuccessful.
- If fork() return zero, the child is created.
- If fork() returns positive value, the process ID of the child process, to the parent.

<u>exit()</u>: ISOC defines exit to provide a way for a process to terminate without running exit handler or signal handler. Whether or not standard I/O streams are flushed depends on the implementation.

EXPERIMENT - 06

Objective :

To demonstrate the Implementation of FIFO.

FIFO : FIFO are sometimes called named pipes. Pipes can be used only between related process but with FIFO unrelated data process can exchange data. We can create a FIFO using

- mkfifo function
- mknod function

The specification of the mode argument for the mkfifo function is the same for the open function. Once we have used mkfifo to create a FIFO, we open it using open. Indeed, the normal file I/O functions (close, read, write, unlink, etc). all work with FIFO.

System Calls Used

mkfifo() or mknod() :

mkfifo() or mknod() is use to create the fifo file. Once we have used mkfifo to create fifo, we open it using open. Mknod() takes three argument file name, S_IFIFO| permision set and flag.

```
mknod("myfifo", S_IFIFO|0666, 0);
```

EXPERIMENT - 07

Objective :

To Implementation of MESSAGE QUEUE.

Message Queue :

A message queue is a linked list of messages stored within the kernal and identified by a message queue identifier. A new queue is created or an existing queue opened by msgget(). New messages are added to the end of a queue by msgsnd(). Every message has a positive long integer type field, a non-negative length, and the actual data bytes, all of which are specified to msgsnd() when the message is added to a queue. Messages are fetched from a queue by msgrcv(). We don't have to fetch the message in a first-in, first out order. Instead, we can fetch messages based on their type field.

System Calls Used

ftok() : ftok() is used to generate a unique key.

msgget() : msgget() either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with

Teacher's Signature

the same key value.

msgsnd(): Data placed on to a message queue by calling msgsnd(). The first argument to msgsnd() is queue identifier, returned by previous call to msgget() The pointer argument points to a longer integer followed by a data buffer for the actual message data.

msgrcv(): Using msgrcv() message are retrieved from a queue.

msgctl(): It performs various operation on a queue. Generally it is use to destroy message queue.

EXPERIMENT - 08

Objective :
Implementation of shared memory.

Shared Memory : The problem with pipes, fifo and message queue - is that for two process to exchange information. The information has to go through the kernal.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel, again requiring the data to be copied from kernal's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So shared memory provides a way by letting two or more process share a memory segment.

System Calls Used

ftok() : ftok() is used to generate a unique key.

shmget() : int shmget (key_t, size_t size, int shmflag);

upon successful completion, shmget() returns an identifier fo the shared memory segment.

Shmat () : Before you can use a shared memory segment, you have to attach yourself to it using shmat().

void *shmat (int shmid, void * shmaddr, int shmflag);

shmid is shared memory id, shmaddr specifies specific address to use but we should set it to zero and OS will automatically choose the address.

Shmdt() : When you are done with the shared memory segment, your program should detach itself from it using shmdt().

int shmdt(void *shmaddr);

Shmctl (): When you detach from shared memory, it is not destroyed. So, to destroy shmctl() is used.

shmctl(int shmid, IPC_RMID, NULL);

EXPERIMENT - 09

Objective :

To implement first come first serve scheduling algorithm.

First Come First Serve (FCFS) :

It is the simplest scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue.

The FCFS scheduling algorithm is non - preemtive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting i/o.

EXPERIMENT -10

Objective :
To implement the shortest - job- first scheduling algorithm

Shortest job first scheduling (SJFS)

A different approach to CPU Scheduling is the shortest-job first scheduling (SJFS) algorithm. This algorithm associate with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the A smallest next CPU burst. If the next burst of two processes are the same, FCFS is used to break the tie.

The SJF scheduling algorithm is probably optimal, that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases. the waiting time of the long process. Consequently, the average waiting time decrease.