	B.M.S COLLEGE OF ENGINEERING, BANGALORE-19 (Autonomous Institute, Affiliated to VTU) **Computer Science & Engineering**

INTERNALS-3		
Course Code : 20CS5PCUSP	**Course Title : Unix Shell and System Programming**	
Semester : 5 A/B/C/D	**Maximum Marks: 40**	**Date: 18-01-2022**
Faculty Handling the Course:	Dr. Kayarvizhy N, Dr. Nandhini Vineeth , Dr.Manjunath D R	
Instructions: ***Internal choice is provided in Part C.***		

PART-A

1. **5 Marks**

Solution:

When a main() function returns the value, the process is terminated.

- When you call an exit() function, which is available in the stdlib.h library, to terminate a process.
- When you call the _Exit() or _exit() functions available in stdlib.h and unistd.h, respectively, to terminate a process.
- When you call pthread_exit to terminate the process.
- When you call an abort() function to abnormally terminate the process.
- When the programmer raises a signal, the process is terminated abnormally if the custom handler or built-in signal handler is not available. But you can handle the signals with a custom/built-in signal handler.
- Thread cancellation requests are also responsible for process termination. A thread cancellation request is the termination of a thread before its job is done in the process.
- Any I/O failure/interrupt leads to process termination
- In some situations, a child process is terminated because of a parent process request.
- A process is terminated when it is trying to access unallocated or unauthorized resources. For example, when a process tries to execute a program that doesn't have execution permissions, it leads to process termination. When a program tries to access memory that it does not own, it leads to process termination.

Explanation of each normal and abnormal exit status = 5 x 1 -- 5 Marks

PART-B

2.a. Solution

You can still lseek and read anywhere in the file, but a write automatically resets the file offset to the end of file before the data is written. This makes it impossible to write anywhere other than at the end of file.

Justification = 2 Marks.
Program = 3 Marks
Total – 5 Marks

2.b. Identify errors Total = 5 Marks
Solution:

```
/* lockit -- demonstration of fcntl locking */
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
int main ()
{
int fd;
struct flock my_lock.
/* set the parameters for the write lock */
my_lock.l_type = F_WRLCK;
my_lock.l_whence - SEEK_SET;
my_lock.l_start = 0;
my_lock.l_len = 10;
/* open file */
fd = open ("locktest", O_RDWR);
/* lock first ten bytes */
if( fcntl(fd, F_SETLKW, &my_lock) == -1){
perror("parent: locking"); exit(1);
}
```

```
printf("parent: locked record\n");
switch(fork())
(
case -1:          /* error *I
perror("fork");
exit(1);
case 0: /* child *I
my_lock.l_len = 5;
if( fcntl(fd, F_SETLKW, &my_lock) == -1)
{
perror("child: locking"); exit(1);
}
printf("child: locked\n"); printf("child: exiting\n"); exit(0);
sleep(5);
/• now exit, which releases lock */
 printf("parent: exiting\n");
exit(0);
```

The actual output produced by lockit will look something like:

```
parent: locked record
parent: exiting
child: locked
child: exiting
```

2.c.

I) Solution: 3 * 1 = 3 Marks

a) `if((fd = open("/tmp/fifo", O_RDWR|O_NONBLOCK)) == -1)`
`perror("open on fifo");`

File is opened with read and write access only and non blocking mode for file is set that means read requests on file can return immediately with failure status if there is no input available instead of blocking and same applies to write requests.
The file description will be returned if no error is there.

b) All the environment variables are printed on screen which is present in env array.

c) Here wait is made non-blocking with WNOHANG option and parent waits for child with given pid until its terminated or stopped and its in a loop "still waiting" gets printed.
As soon as child process terminates, the respective pid is returned and it exits from loop.

II) Solution: 2 Marks

Anything written by the child to standard error appears wherever the parent's standard error would appear. To send standard error back to the parent, include the shell redirection 2>&1 in the cmdstring.

3.a. 10 Marks

```
*
 * This trivial program illustrates how to open files.  We present the
 * following use cases:
 *
 * - create a non-existent file
 * - "create" an existing file
 * - fail to create an existing file
 * - open an existing file
 * - fail to open a non-existing file
 * - truncate an existing file
 */
```

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

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#ifndef SLEEP
#define SLEEP 10
#endif

void
createFile() {
        int fd;

        printf("Checking if './newfile' exists...\n");
        system("ls -l ./newfile");
        printf("Trying to create './newfile' with O_RDONLY |
O_CREAT...\n");

        if ((fd = open("./newfile", O_RDONLY | O_CREAT,
                                    S_IRUSR | S_IWUSR)) == -1) {
                fprintf(stderr, "Unable to create './newfile': %s\n",
                                strerror(errno));
                exit(EXIT_FAILURE);
        }

        printf("'./newfile' created. File descriptor is: %d\n", fd);

        /* We are leaking a file descriptor here to illustrate that they
         * are increasing as we open more files.  Normally, we would call
         * close(2) whenever we're done using the decriptor. */
}

void
failExclFileCreation() {
        int fd;

        printf("Checking if './newfile' exists...\n");
        system("ls -l ./newfile");
        printf("Trying to create './newfile' with O_RDONLY | O_CREAT |
O_EXCL...\n");

        if ((fd = open("./newfile", O_RDONLY | O_CREAT | O_EXCL,
                                    S_IRUSR | S_IWUSR)) == -1) {
                fprintf(stderr, "Unable to create './newfile': %s\n",
                                strerror(errno));
        }

        /* We expect this to fail! */
        if (close(fd) == -1) {
                fprintf(stderr, "Closing failed: %s\n", strerror(errno));
        }
}


void
failOpenNonexistingFile() {
        int fd;
```

```
        printf("Trying to open (non-existant) './nosuchfile' with
O_RDONLY...\n");

        if ((fd = open("./nosuchfile", O_RDONLY)) == -1) {
                fprintf(stderr, "Unable to open './nosuchfile': %s\n",
                                strerror(errno));
        }

        /* We know this is going to fail, but no need to complain. */
        (void)close(fd);
}

void
openFile() {
        int fd;

        printf("Trying to open './openex.c' with O_RDONLY...\n");

        if ((fd = open("./openex.c", O_RDONLY)) == -1) {
                fprintf(stderr, "Unable to open './openex.c': %s\n",
                                strerror(errno));
                exit(EXIT_FAILURE);
        }

        printf("'./openex.c' opened. File descriptor is: %d\n", fd);

        if (close(fd) == 0) {
                printf("'./openex.c' closed again\n");
        }
}

void
truncateFile() {
        int fd;

        system("cp openex.c newfile");
        printf("Copied 'openex.c' to 'newfile'.\n");
        system("ls -l newfile");

        printf("Trying to open './newfile' with O_RDONLY |
O_TRUNC...\n");

        if ((fd = open("./newfile", O_RDONLY | O_TRUNC)) == -1) {
                fprintf(stderr, "Unable to open './newfile': %s\n",
                                strerror(errno));
                exit(EXIT_FAILURE);
        }

        printf("'./newfile' opened. File descriptor is: %d\n", fd);
        printf("'./newfile' truncated -- see 'ls -l newfile'\n");
        system("ls -l newfile");

        (void)close(fd);
}



int
main() {
        createFile();
```

```
        system("ls -l newfile");
        printf("\n");
        sleep(SLEEP);

        createFile();
        system("ls -l newfile");
        printf("\n");
        sleep(SLEEP);

        failExclFileCreation();
        printf("\n");
        sleep(SLEEP);

        openFile();
        printf("\n");
        sleep(SLEEP);

        failOpenNonexistingFile();
        printf("\n");
        sleep(SLEEP);

        truncateFile();

        return 0;
}
```

3b. 10 Marks

Solution :

```
/* This simple program illustrates the use of the chmod(2) system call,
 * and how it can be used to set explicit/absolute modes or to
selectively
 * enable individual modes.
 *
 * Set your umask to 077, create file and file1, then run this command.
 */

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

#include <stdio.h>
#include <stdlib.h>

int
main() {
        struct stat sbuf;

        umask (S_IRWXGRP | S_IRWXOTH /* == 0077 */);


        file_descriptor = creat(file, O_RDWR | O_CREAT);
        file_descriptor = creat(file1, O_RDWR | O_CREAT);
```

```
        if (stat("file", &sbuf) == -1) {
                perror("can't stat file");
                exit(EXIT_FAILURE);
        }

        /* turn off owner read permissions and turn on setgid */
        if (chmod("file", (sbuf.st_mode & ~S_IRUSR) | S_ISGID) == -1) {
                perror("can't chmod file");
                exit(EXIT_FAILURE);
        }

        /* set absolute mode to rw-r--r-- */
        if (chmod("file1", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) == -1)
{
                perror("can't chmod file1");
                exit(EXIT_FAILURE);
        }

        exit(EXIT_SUCCESS);
}
```

4.a. **Solution: 10 Marks**

```
/* A simple illustration of exit handlers.  Note that exit
handlers are
 * pushed onto a stack and thus execute in reverse order.
 *
 * Illustrate exiting at different times by invoking
 * this program as
 * ./a.out            exit handlers invoked after return from main
 * ./a.out 1          exit handlers invoked from within func
 * ./a.out 1 2        no exit handlers invoked
 * ./a.out 1 2 3      we call abort(3), no exit handlers invoked
 *
 *
 * */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
my_exit1(void) {
        (void)printf("first exit handler\n");
}

void
my_exit2(void) {
        (void)printf("second exit handler\n");
}

void
func(int argc) {
```

```
        (void)printf("In func.\n");
        if (argc == 2) {
                exit(EXIT_SUCCESS);
        } else if (argc == 3) {
                _exit(EXIT_SUCCESS);
        } else if (argc == 4) {
                abort();
        }
}


int
main(int argc, char **argv) {
        (void)argv;
        if (atexit(my_exit2) != 0) {
                perror("can't register my_exit2\n");
                exit(EXIT_FAILURE);
        }

        if (atexit(my_exit1) != 0) {
                perror("can't register my_exit1");
                exit(EXIT_FAILURE);
        }

        if (atexit(my_exit1) != 0) {
                perror("can't register my_exit1");
                exit(EXIT_FAILURE);
        }

        func(argc);

        (void)printf("main is done\n");

        return EXIT_SUCCESS;
}
```

4b) Solution: Total 10 Marks

```
/* This program illustrates that after fork(2), the
 * child has a copy of the file descriptors from the
 * parent pointing to the same file table entries,
 * meaning operations on the fd in one affect the
 * other. */

#include <sys/wait.h>

#include <err.h>
#include <fcntl.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>

#define NUM 32

void
readData(int fd) {
        int n;
        char buf[NUM];
        off_t offset;

        if ((offset = lseek(fd, 0, SEEK_CUR)) < 0) {
                err(EXIT_FAILURE, "%d lseek error", getpid());
                /* NOTREACHED */
        }

        (void)printf("%d offset is now: %ld\n", getpid(),
offset);
        if ((n = read(fd, buf, NUM)) < 0) {
                err(EXIT_FAILURE, "PID %d: read error",
getpid());
                /* NOTREACHED */
        }
        /* We don't do anything with the data; we just
         * wanted to illustrate that reading will
         * advance the offset. */
}


int
main(int argc, char **argv) {
        int fd;
        pid_t pid;

        if (argc != 2) {
                fprintf(stderr, "Usage: %s file\n", argv[0]);
                exit(EXIT_FAILURE);
        }

        if ((fd = open(argv[1], O_RDONLY)) < 0) {
                err(EXIT_FAILURE, "open error");
                /* NOTREACHED */
        }

        (void)printf("Starting pid is: %d\n", getpid());

        readData(fd);

        if ((pid = fork()) < 0) {
                err(EXIT_FAILURE, "fork error");
                /* NOTREACHED */
        } else if (pid == 0) {
                if (lseek(fd, NUM, SEEK_CUR) < 0) {
                        err(EXIT_FAILURE, "child lseek error");
                        /* NOTREACHED */
                }
                (void)printf("child %d done seeking\n",
getpid());
                /* give the parent a chance to read */
                sleep(2);
                readData(fd);
```

```
        } else {
                /* give the child a chance to lseek */
                sleep(1);
                readData(fd);
        }

        (void)wait(NULL);
        (void)close(fd);
        return EXIT_SUCCESS;
}
```