BT17CSE030_042 OS LAB 3 Assignment

FORK1:

output->

```
hello world (pid:21062)

hello, I am parent of 21063 (pid:21062)

hello, I am child (pid:21063)
```

Explanation->

here we have use fork() system call which create its child process(pid=21063);

and first the parent process(pid=21063) get executed than child process get executed.

getpid is used to get the pid of that process where it is used.

FORK2:

output->

```
hello world (pid:21408)

hello, I am child (pid:21409)

hello, I am parent of 21409 (rc_wait:21409) (pid:21408)
```

Explanation->

here we have use fork() system call as well as wait system call due to which child process(pid=21409) is created ;

and first the child process(pid=21409) get executed than parent process(pid=21408) get executed because of wait system call in parent process.

FORK3:

output->

```
hello world (pid:21739)

hello, I am child (pid:21740)
```

```
 28 123 814 sort.c
hello, I am parent of 21740 (rc_wait:21740) (pid:21739)
```

Explanation->

here execvp system call is used in child proces which count word char line in file provide.

child is excuted before parent because we have used wait system in parent process.

if execvp system call is not executed properly than this will be output->

```
hello world (pid:21760)
hello, I am child (pid:21761)
this shouldn’t print outhello, I am parent of 21761 (rc_wait:21761) (pid:21760)
```

FORK4:

output-> p4.output cointains->26 109 718 sort.c

Explanation->

here execvp system call is used in child proces which count word char line in file provide.

child is excuted before parent because we have used wait system in parent process.

and we save the output in file named as p4.output.

by using open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

and here:

O_WRONLY-->

Open for writing only.

O_CREAT--->If the file exists, this flag has no effect except as noted under O_EXCL below. Otherwise, the file is created;

the user ID of the file is set to the effective user ID of the process; the group ID of the file is set to the group ID

of the file's parent directory or to the effective group ID of the process;

and the access permission bits (see <sys/stat.h>) of the file mode are set to the value of the third argument taken as type mode_t modified as follows:

a bitwise-AND is performed on the file-mode bits and the corresponding bits in the complement of the process' file mode creation mask.

Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared. When bits other than the file permission bits are set, the effect is unspecified.

The third argument does not affect whether the file is open for reading, writing or for both.

O_TRUNC-->

If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length is truncated

to 0 and the mode and owner are unchanged.

It will have no effect on FIFO special files or terminal device files.

Its effect on other file types is implementation-dependent.

The result of using O_TRUNC with O_RDONLY is undefined.

5) If the parent process assigns a variable some

value before fork (), that value remains same in

the child process. If the value of the variable is

changed in child process, it does not reflect in the

parent process.

E.g. code:

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<time.h>

int main()

{

int x = 100;
int rc = fork();
if(rc < 0)
{
fprintf(stderr,"Fork failed\n");
}
else if(rc == 0 )
{
x=x+1;
}
printf("x = %d ",x);
return 0;
}
```

Output:

```
x = 100 x =101
```

6) The system call waitpid () is different from wait() because waitpid () takes the pid of the child process as argument and waits only for that child to change state, while wait () suspends calling process till all its children terminate.

E.g. code:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<time.h>
int main()
```

```
{

int rc1 = fork();

int rc2 = fork();

if(rc1<0)

{

fprintf(stderr, "fork1 failed\n");

exit(1);

}

else if(rc1==0)

{

printf("This is child process of id = %d ",(int)getpid());

}

if(rc2<0)

{

fprintf(stderr, "fork2 failed\n");

exit(1);

}

else if(rc2==0)

{

printf("This is child process of id = %d ",(int)getpid());

}

if(rc1!=0&&rc2!=0)

{

int rc_wait = waitpid(rc1,NULL,0);

printf("Parent process of %d & %d (rc_wait = %d) with pid = %d
```

",rc1,rc2,rc_wait,(int)getpid());

}

return 0;

}

Output:

This is child process of id = 2312 This is child process of id = 1688 This is child process of id

= 1688 This is child process of id = 6680 Parent process of 6680 & 2312 (rc_wait = 6680)

with pid = 1680

Here both children complete executing before

parent even though parent is waiting for only one.

This is because while parent is waiting for first

child to complete, the second gets completed as

well, as both children take almost equal time for

execution.