UNIX PROGRAMMING ASSIGNMENT

MCA - 372

Submitted by :- Riya Gurbani

Roll Number : 1947255

Describe the following from Process Control of Unix Environment

1) Fork, and Vfork with file sharing functions

<u>Fork</u>

When we create a new process through an existing one is by calling the fork function.
When we redirect the standard output of the parent from the program, the child's standard output is also redirected. All file descriptors that are open in the parent are duplicated in the child. Because its as if the dup function had been called for each descriptor. The parent & the child share a file table entry for every open descriptor. Consider a process that has three different files opened for standard-input, output & error. On return from fork, we have the arrangement.
It is important that the parent & the child share the same file offset. Consider a process that forks a child, then waits for the child to complete. Assume that both processes write to standard output as part of their normal processing.
If the parent has its standard output redirected it is essential that the parent's file offset be updated by the child when the child writes to

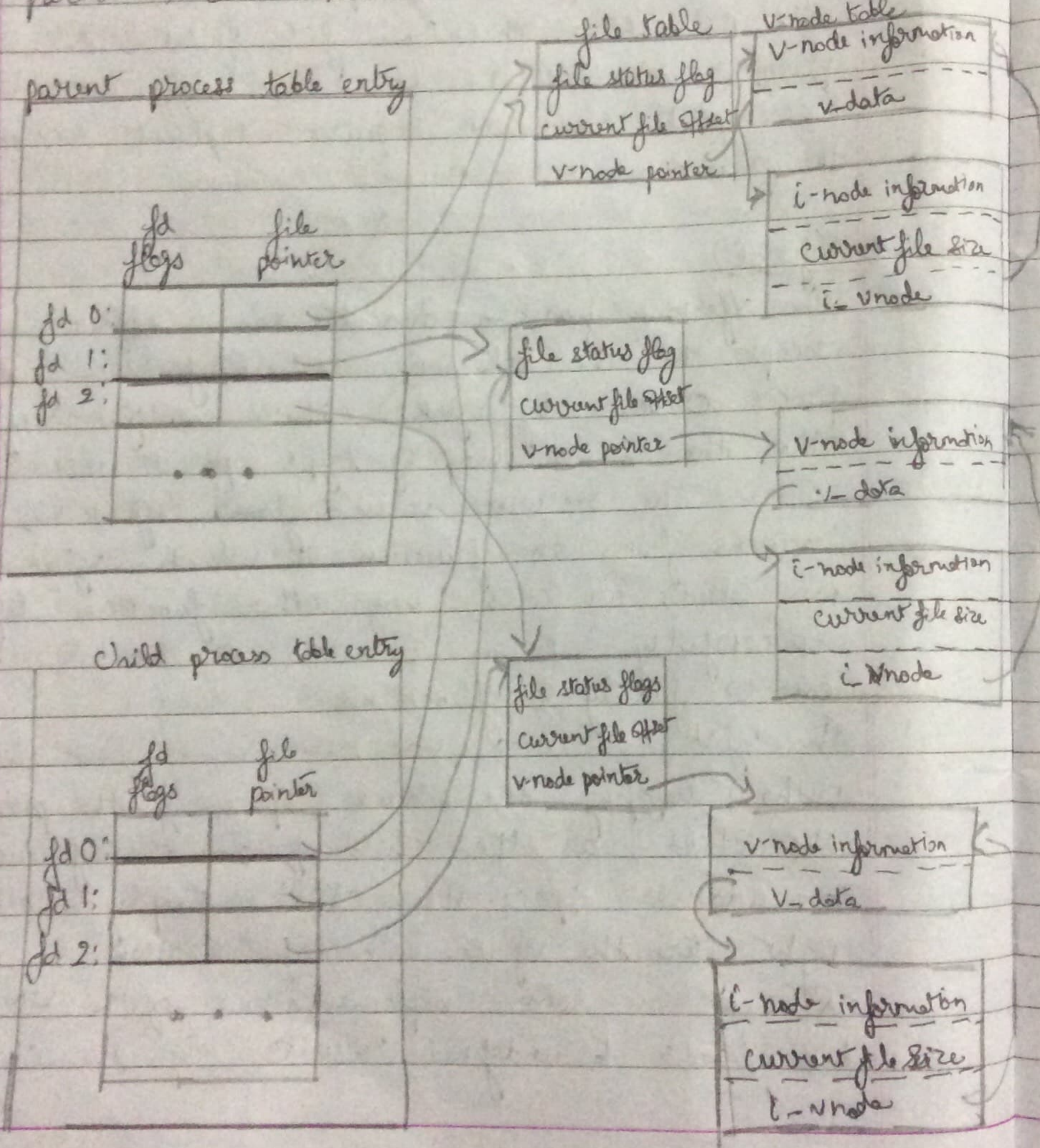
standard output. In this case, the child can write to standard output while the parent is waiting for it. On completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote. If parent & the child did not share the same file offset, this type of interaction would be more difficult to accomplish & would require explicit actions by the parent.

Vfork()

The vfork() function has the same effect as fork(), except that the behaviour is undefined if the process created by vfork() either modifies any data other than a variable of type pid_t used to store the return value from vfork(), or returns from the function in which vfork() was called, or calls any other function before successfully calling _exit() or one of the exec() family of functions.

The vfork() creates the new process just like fork, without copying the address space of the parent into the child, as the child ~~f~~ won't reference that ~~the~~ address space, the child simply calls exec right after the vfork. The optimization is more efficient on some implementations of the UNIX system, but leads to undefined results if the child

modifies any data, makes function calls, or return without calling exec or exit.
Vfork guarantees that the child runs first, until the child calls exec or exit. When the child calls either of those functions, the parent resumes.



Sharing of open files between parent & child after fork.

2. Wait and Wait ID functions.

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event, it can happen at any time while the parent is running - this signal is the asynchronous notification from the kernel of the parent. The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler. The default action for this signal is to be ignored.

→ Block, if all of its children are still running

→ Return immediately with the termination status of a child, if a child has ~~too~~ terminated & is waiting for its termination status to be fetched.

→ Return immediately with an error, if it doesn't have any child processes.

If the process is calling wait because it received the SIGHLD signal, we expect wait to return immediately. But if we call it any random point in time, it can block.

```
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

The difference b/w these two functions are:-

→ The wait function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.

→ The waitpid () doesn't wait for the child that terminates first, it has a number of options that control which process it waits for.

If a child has already terminated & is a zombie, wait returns immediately with that child's status. Otherwise it blocks the caller until a child terminates. If the caller blocks & has multiple children, wait returns when one terminates. We can always tell which child terminated, because the process ID is returned by the function.

Waitid function

The Single UNIX Specification includes an additional function to retrieve the exit status of a process. the waitid () is similar to waitpid but with extra flexibility.

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t
*infop, int options);
```

Return 0 if ok -1 on error

Like waitpid, waitid allows a process to specify which ~~et~~ children to wait for. Instead of encoding this information in a single argument

combined with the process ID or process group ID, two seperate arguments are used. The id parameter is interpreted based on the value of idtype.
The types supported are :-

Constant	Description
P_PID	wait for a particular process ; id contains the process ID of the child to wait for.
P_PGID	wait for any child process in a particular process group ; id contains the process group ID of the children to wait for.
P_ALL	wait for any child process ; id is ignored.

infop A pointer to a siginfo_t structure, where the function can store the current state of the child.

3 Exec function :-
When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created ; exec merely replaces the current process - its text, data, heap, & stack segments - with a brand new program from disk. There are seven different exec functions, round out the UNIX System process control primitives. With fork, we can create a new processes & with exec functions, we can initiate new programs.

The exit function & the wait functions handle termination & waiting for termination.

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0,
    ... /* (char *) 0 */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ---
int execve(const char *pathname, char *const argv[]
char *const envp[]);
```

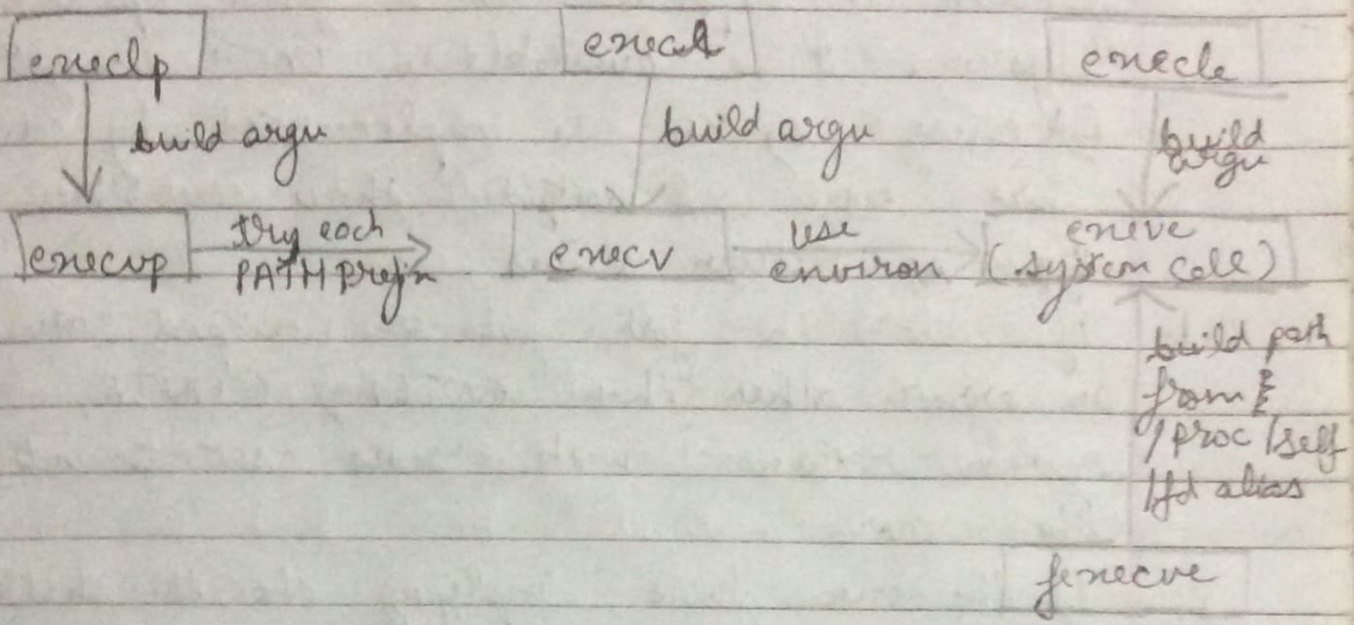
In all the above example the first four take a pathname argument, the next two take a filename argument, and the last one takes a file descriptor argument. When a filename argument is specified.

- If filename contains a slash, it is taken as a pathname.
- Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.

Second is the argument passing (l stands for list & v stands for vector).

Since the arguments for these exec functions are difficult to remember. The letters in the function names help somehow. The letter p means that the function takes a filename argument & uses the Path env. variable to find the executable file.

The letter l means it takes a list of arguments & is mutually exclusive with the letter v.
The letter e means that the function takes an envp[] array instead of using the current environment.



Relationship of the seven exec functions.

4. System Function

It is convenient to execute a command string from within a program. For eg. if we want to display time & date stamp into a certain file we can use the function system ("date > file");
Call time to get current calendar time, then call localtime to convert it to a broken-down time, then call strftime to format the result & finally write the result to the file.

```
#include <stdlib.h>
int system(const char *cmdstring);
```

If cmdstring is a null pointer, system returns nonzero only if a command processor is available. It determines whether the system function is supported on a given operating system. It is available in UNIX always. Because system is implemented by calling fork, exec, & waitpid, there are three types of return values.

1. If either the fork fails or waitpid returns an error other than ~~EINTR~~ EINTR, system returns -1 with errno set to indicate the error.
2. If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127).
3. Otherwise all three functions - fork, exec, & waitpid - succeed, & the return value from system is the termination status of the shell, in the format specified for waitpid.

Set-User-ID Programs.

It creates a security hole & should never be attempted.

5. Process Scheduling

The scheduling policy and priority were determined by the kernel. A process could choose to run with lower priority by adjusting its nice value. Only a privileged process was allowed to increase its scheduling priority. In the Single Unix specification, nice values range from 0 to (2 * NZERO) -1, lower nice values have higher scheduling priority. NZERO is the default nice value of the system.

A process can retrieve & change its nice value with the nice function. With this function, a process can affect only its own nice value, it can't affect the nice value of any other process.

```
#include <unistd.h>
int nice (int incr);
```

incr arg. is added to the nice value of the calling process. If incr is too large, the system silently reduces it to the maximum legal value. If its too small the system silently increases it to the minimum legal value. Because -1 is a legal successful return value, we need to clear errno before calling nice & check its value if nice returns -1. If the call to nice succeeds & the return value is -1, then errno will still be zero. If errno is nonzero

it means that the call to nice failed.
The getpriority() can be used to get the nice value for a process, just like the nice function. However, getpriority can also get the nice value for a group of related processes.

#include <sys/resource.h>
int getpriority (int which, id_t who)
Returns: nice value b/w -NZERO & NZERO-1 if OK, -1 on ~~error~~, error.

The which arg can take on one of 3 values: PRIO_PROCESS to indicate process, PRIO_PGRP to indicate a process group & PRIO_USER to indicate a user ID. The which arg controls how the who arg is interpreted & the who arg selects the process or processes of interest.

The setpriority() can be used to set the priority of a process, a process group, or all the processes belonging to a particular user ID.

#include <sys/resource.h>
int setpriority (int which, id_t who, int value)
Returns: 0 if ok, -1 on error

The which & who arg are the same as in the getpriority(). The value is added to NZERO

& this becomes the new nice value.
The Single UNIX specification leaves it up to the implementation whether the nice value is inherited by a child process after a fork. However, XSI-compliant systems are required to preserve the nice value across a call to exec.