CMR
INSTITUTE OF
TECHNOLOGY

USN

Internal Assesment Test - II

Sub:	UNIX SYSTEM PROGRAMMING							Code:	10CS62
Date:	08 / 05 /2017	Duration:	90 mins	Max Marks:	50	Sem:	6(A,B,C)	Branch:	CSE

Answer any **FIVE** full questions

			Marks	OBE	
				CO	RBT
1	(a)	Explain various exec functions along with their prototype.	[3]	CO5	L4
	(b)	List out the three main difference between them	[3]	CO5	L1
	(c)	Write a C/C++ program to demonstrate execle and execlp APIs.	[4]	CO5	L3
2	(a)	List out similarities and differences between fork() and vfork() .	[4]	CO5	L1
	(b)	Write separate C/C++ program to demonstrate fork() and vfork().	[6]	CO5	L3
3	(a)	With related data structure, explain the UNIX kernel support for a process.	[6]	CO4	L4
	(b)	Explain with a neat diagram, the memory layout of a C program.	[4]	CO4	L4
4	(a)	What is job control? Summarize the job control features with the help of a neat diagram	[5]	CO5	L2
	(b)	Explain with a neat sketch, how a C program is started and terminated in UNIX.	[5]	CO4	L4
5	(a)	Explain getrlimit() and setrlimit() functions with prototype.	[4]	CO4	L4
	(b)	Mention the three rules to change the resource limits.	[3]	CO4	L1
	(c)	Give any six resource values along with their meaning.	[3]	CO4	L1
6	(a)	What is a Zombie process? Write a C/C++ program to avoid zombie process by calling fork() twice.	[6]	CO5	L3
	(b)	Explain the different ways in which a process can terminate.	[4]	CO4	L4
7	(a)	What is race condition? Write a C/C++ program to illustrate a race condition and to avoid the race condition.	[6]	CO5	L3
	(b)	Write a short note on controlling terminal with a neat sketch.	[4]	CO5	L1

DEPARTMENT : CSE/ISE

Scheme and Solution for IAT2-May 2017

Unix System Programming (10CS62)

1. a) 3Marks

```
#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,... /*(char *)0, char *const envp */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... * (char *)0 */ );
int execvp(const char *filename, char *const argv []);
```

All six return: -1 on error, no return on success.

b)Three Differences: 3 Marks

- The first difference in these functions is that the first four take a pathname argument, whereas the last two take a filename 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.
- The next difference concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl , execlp , and execle require each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (execv , execvp , and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.
- The final difference is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

c) 4 Marks

```
#include "apue.h"
#include <sys/wait.h>
char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void)
{
pid_t pid;
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid == 0) {
if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
"MY ARG2", (char *)0, env_init) < 0)
err_sys("execle error");
}
if (waitpid(pid, NULL, 0) < 0)
err_sys("wait error");
if ((pid = fork()) < 0) {
        err_sys("fork error");
        } else if (pid == 0) {
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
        err_sys("execlp error");
        }
        exit(0);
        }
```

2.a)
Similarities: 2 Marks

1. Both fork() and vfork() are used to create a neww process called child processes.
2. Calling sequences and return values are same for both fork() and vfork()

Differences 2Marks

1. The vfork function creates the new process, just like fork , without copying the address space of the parent into the child, as the child won't reference that address space.
2. Another difference between the two functions is that vfork guarantees that the child runs first, until the child calls exec or exit . When the child calls either of these functions, the parent resumes.

b)   4 Marks

<u>Example program that demonstrate fork()</u>

```
        #include <stdio.h>
        #include <sys/types.h>
        #include <unistd.h>


        int main()
        {

        int var = 5;
        int pid;
        printf(" parent process\n");
        if(pid = fork() == 0)
        {
                printf("child process\n");
                var++;
                printf("The value of var in child process = %d\n",var);
        }
        else
        {
                printf(" again  parent process\n");
                printf("The value of var in parent process = %d\n",var);
        }
        return 0;
        }
```

<u>Example program that demonstrate vfork()</u>

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int glob=5;
int var=8;
int main()
{

int pid;
printf("I am in parent process\n");
if(pid = vfork() == 0)
{
        printf("I am in child process\n");
glob++;
        var++;
        printf("glob = %d,var= %d\n",glob,var);

}
else
{
        //sleep(2);
        printf("I am again in parent process\n");
        printf("glob = %d,var= %d\n",glob,var);
}
return 0;
}
```

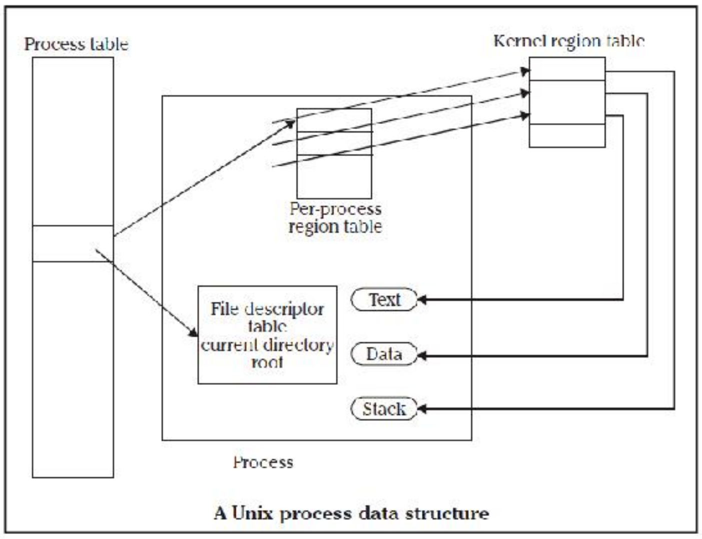
3 a) 2 Marks

A UNIX process consists minimally of a text segment, a data segment and a stack segment. A segment is an area of memory that is managed by the system as a unit.

- A text segment consists of the program text in machine executable instruction code format.
- The data segment contains static and global variables and their corresponding data.
- A stack segment contains runtime variables and the return addresses of all active functions for a process.

UNIX kernel has a process table that keeps track of all active process present in the system. Some of these processes belongs to the kernel and are called as “system process”. Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process. U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system imposed process limits.

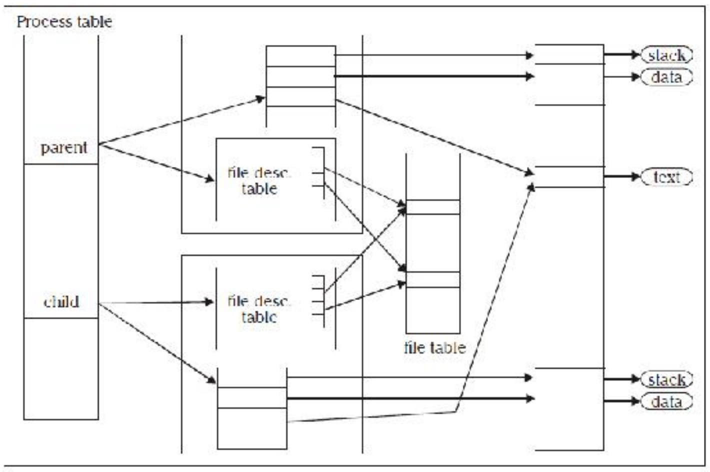
2 Marks



A Unix process data structure

All processes in UNIX system expect the process that is created by the system boot code, are created by the fork system call. After the fork system call, once the child process is created, both the parent and child processes resumes execution. When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below. Also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

2 Marks



The process will be assigned with attributes, which are either inherited from its parent or will be set by the kernel.

- A real user identification number (rUID)
- A real group identification number (rGID)
- An effective user identification number (eUID)
- An effective group identification number (eGID)
- Saved set-UID and saved set-GID
- Process group identification number (PGID) and session identification number (SID)
- Supplementary group identification numbers:
- Current directory
- Root directory
- Signal handling
- Signal mask
- Unmask
- Nice value
- Controlling terminal

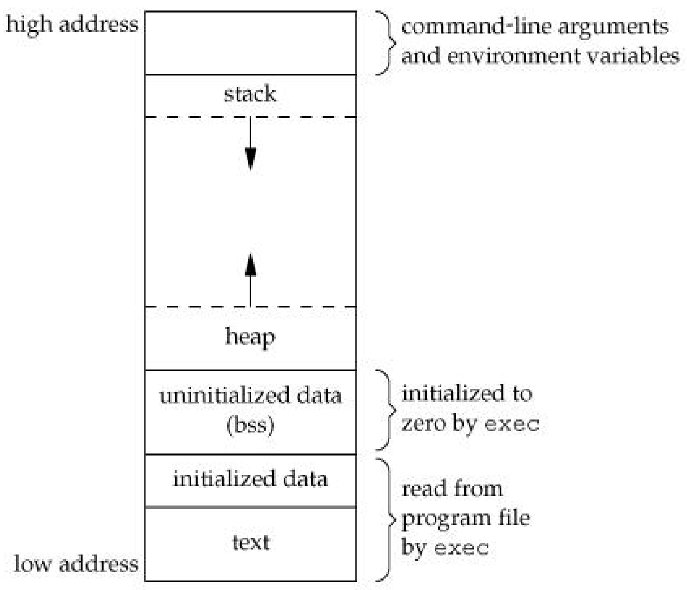
In addition to the above attributes, the following attributes are different between the parent and child processes:

- Process identification number (PID)
- Parent process identification number (PPID)
- Pending signals
- Alarm clock time
- File locks

b) 3 Marks

C program has been composed of the following pieces:
Text segment, the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.
Initialized data segment, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration
int long maxcount = 99;
appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.
Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration
sum[1000];
appearing outside any function causes this variable to be stored in the uninitialized data segment.
Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.
Heap, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.

2 Marks


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

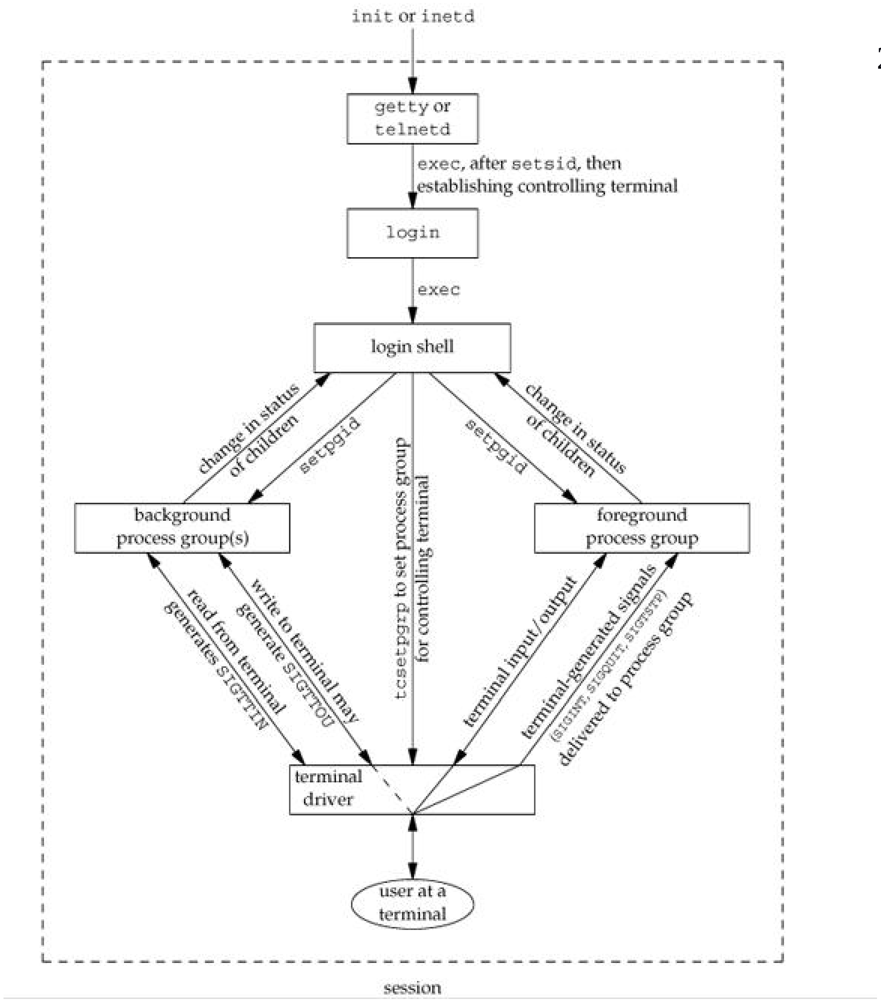
4 a) 3 Marks
This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are to run in the background.
Job control requires three forms of support:

- A shell that supports job control
- The terminal driver in the kernel must support job control
- The kernel must support certain job-control signals

The interaction with the terminal driver arises because a special terminal character affects the foreground job: the
suspend key (typically Control-Z). Entering this character causes the terminal driver to send the SIGTSTP signal to all processes in the foreground process group. The jobs in any background process groups aren't affected. The terminal driver looks for three special characters, which generate signals to the foreground process group.

- The interrupt character (typically DELETE or Control-C) generates SIGINT .
- The quit character (typically Control-backslash) generates SIGQUIT .
- The suspend character (typically Control-Z) generates SIGTSTP .

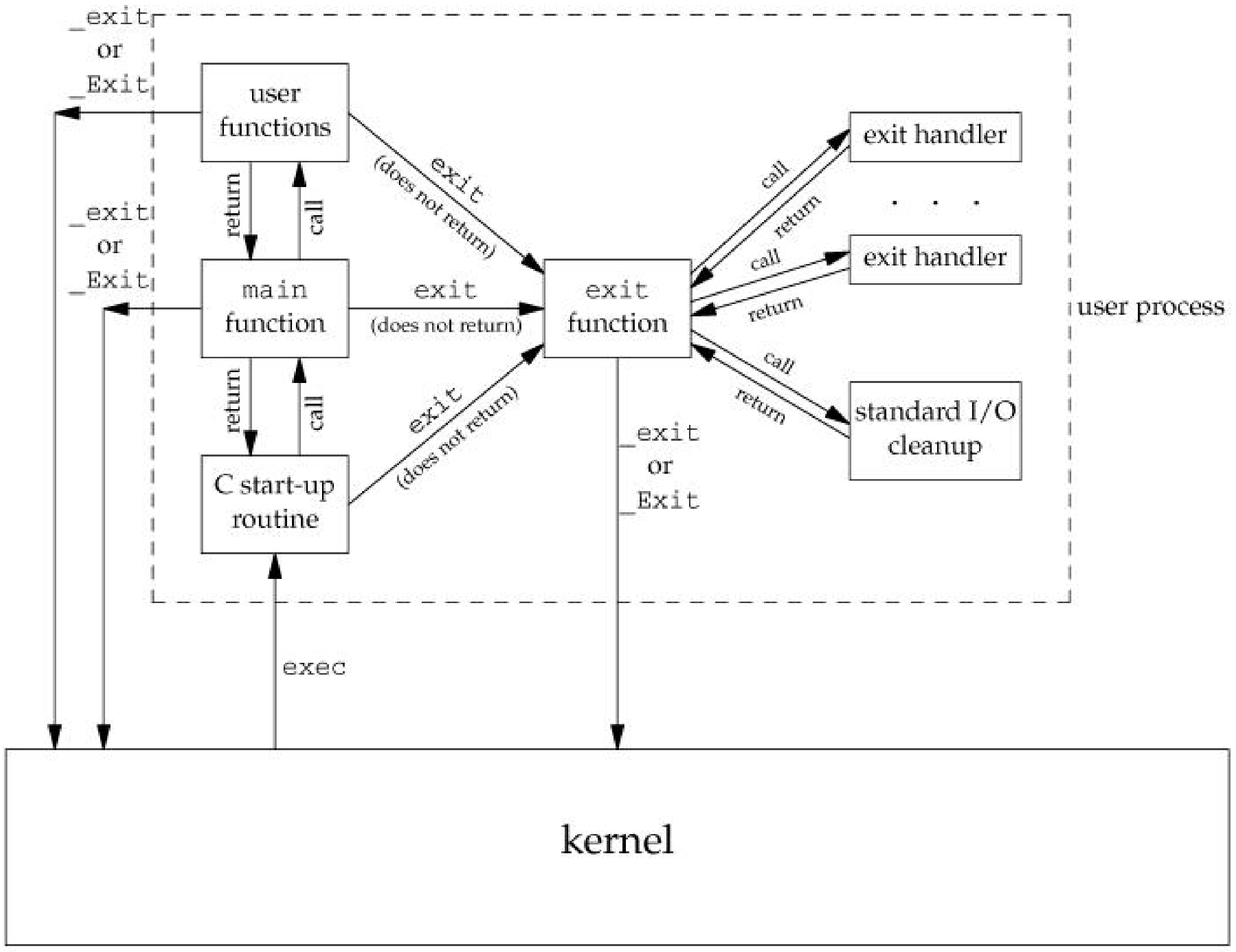
This signal normally stops the background job; by using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal.



init or inetd
getty or telnetd
exec, after setsid, then establishing controlling terminal
login
exec
login shell
change in status of children
setpgid
tcsetpgrp to set process group for controlling terminal
setpgid
change in status of children
background process group(s)
foreground process group
read from terminal generates SIGTTIN
write to terminal may generate SIGTTOU
terminal input / output
terminal-generated signals (SIGINT, SIGQUIT, SIGTSTP) delivered to process group
terminal driver
user at a terminal
session

2 Marks

b) 5 marks



5a) 4 marks

Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);
```

Both return: 0 if OK, nonzero on error

Each call to these two functions specifies a single resource and a pointer to the following structure:

```
struct rlimit
{
rlim_t rlim_cur;
rlim_t rlim_max;
};
```

b) 3 Marks

Three rules govern the changing of the resource limits.

- A process can change its soft limit to a value less than or equal to its hard limit.

- A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
- Only a superuser process can raise a hard limit.

c) 3 Marks

RLIMIT_AS The maximum size in bytes of a process's total available memory.
RLIMIT_CORE The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_CPU   The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.
RLIMIT_DATA
The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.
RLIMIT_FSIZE
The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZ signal.
RLIMIT_LOCKS
The maximum number of file locks a process can hold.

6a) 6marks

Program to Avoid zombie processes by calling fork twice

```
#include "apue.h"
#include <sys/wait.h>
Int main(void)
{
pid_t pid;
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid == 0) {

if ((pid = fork()) < 0)
err_sys("fork error");
else if (pid > 0)
exit(0);
sleep(2);
printf("second child, parent pid = %d\n", getppid());
exit(0);
}
if (waitpid(pid, NULL, 0) != pid)
err_sys("waitpid error");
exit(0);
}
```

Output:

```
$ ./a.out
$ second child, parent pid = 1
```

b) 4 marks

There are eight ways for a process to terminate. Normal termination occurs in five ways:

- Return from main
- Calling exit
- Calling _exit or _Exit
- Return of the last thread from its start routine
- Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

- Calling abort
- Receipt of a signal
- Response of the last thread to a cancellation request

Exit Functions

Three functions terminate a program normally: _exit and _Exit , which return to the kernel immediately, and exit , which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>
void exit(int status);
void _Exit(int status);
#include <unistd.h>
void _exit(int status);
```

All three exit functions expect a single integer argument, called the exit status. Returning an integer value from the main function is equivalent to calling exit with the same value. Thus exit(0); is the same as return(0); from the main function.

In the following situations the exit status of the process is undefined.

Any of these functions is called without an exit status. main does a return without a return value.

7 a) 2 Marks

A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run.

The program below outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

2 marks

```
#include "apue.h"
static void charatatime(char *);
int main(void)
{
pid_t pid;
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid == 0) {
charatatime("output from child\n");
} else {
charatatime("output from parent\n");
}
exit(0);
}
static void
```

```
charatatime(char *str)
{
char *ptr;
int c;
setbuf(stdout, NULL);
for (ptr = str; (c = *ptr++) != 0; )
putc(c, stdout);
}
```

Output:

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

program modification to avoid race condition

2 Marks

```
#include "apue.h"
static void charatatime(char *);
int main(void)
{
pid_t pid;

TELL_WAIT();
if ((pid = fork()) < 0) {
err_sys("fork error");
} else if (pid == 0) {
WAIT_PARENT();
charatatime("output from child\n");
} else {
charatatime("output from parent\n");
TELL_CHILD(pid);
}
exit(0);
}
static void charatatime(char *str)
{
char *ptr;
int c;
setbuf(stdout, NULL);
for (ptr = str; (c = *ptr++) != 0; )
putc(c, stdout);
}
```

When we run this program, the output is as we expect; there is no intermixing of output from the two processes.

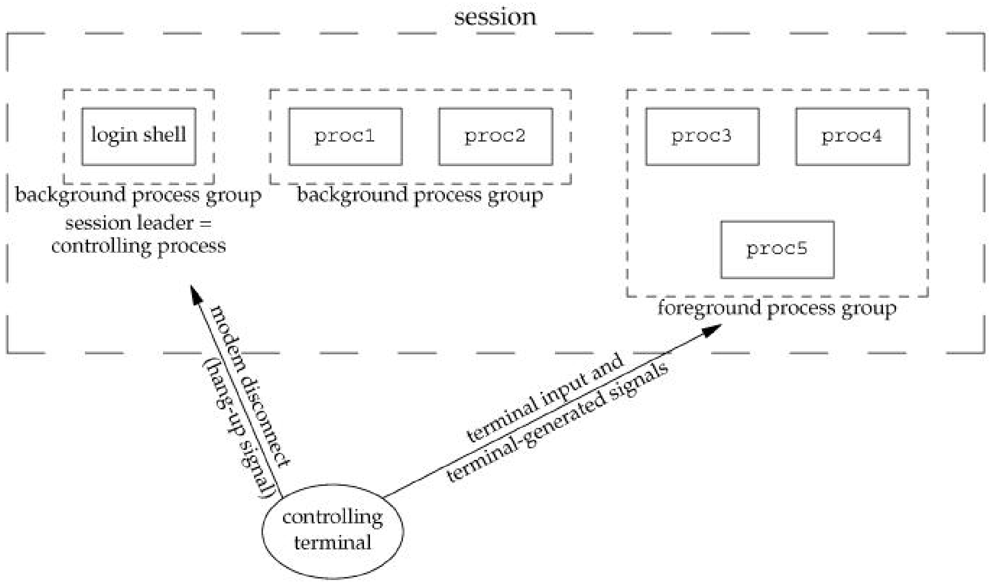
b) 2 Marks

Sessions and process groups have a few other characteristics.

- A session can have a single controlling terminal. This is usually the terminal device (in the case of a terminal login) or pseudo-terminal device (in the case of a network login) on which we log in.
- The session leader that establishes the connection to the controlling terminal is called the controlling process.
- The process groups within a session can be divided into a single foreground process group and one or more background process groups.
- If a session has a controlling terminal, it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type the terminal's interrupt key (often DELETE or Control-C), this causes the interrupt signal be sent to all processes in the foreground process group.
- Whenever we type the terminal's quit key (often Control-backslash), this causes the quit signal to be sent to all processes in the foreground process group.
- If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader).

2 Marks

These characteristics are shown in Figure


session
login shell
proc1
proc2
proc3
proc4
proc5
background process group
session leader =
controlling process
background process group
foreground process group
modem disconnect
(hang-up signal)
terminal input and
terminal-generated signals
controlling
terminal