5. Process Control

Process Creation:

- ✓ **The only way for a user to create a new process in the UNIX operating system is to invoke the fork system call.**
- ✓ **The process that invokes fork is called the parent process, and the newly created process is called the child process.**
- ✓ **The syntax for the fork system call is**

```
pid = fork();
```

PROCESS CREATION

Process Creation:

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 a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

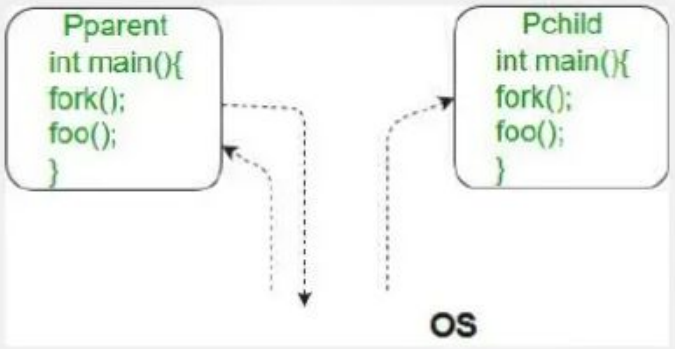
Negative Value: creation of a child process was unsuccessful.

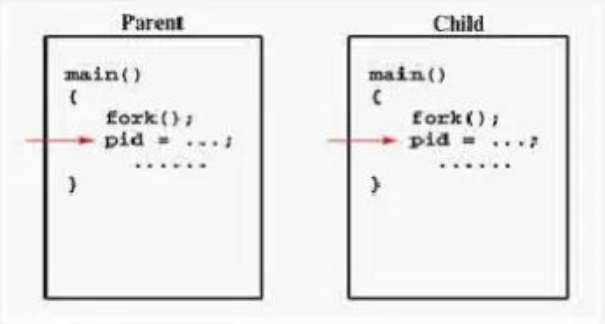
Zero: Returned to the newly created child process.

Positive value: returns a the ***process ID*** of the child process, to the parent.

PROCESS CREATION

Process Creation:





PROCESS CREATION

Process Creation:

Sequence of Operations for fork:

1. It allocates a slot in process table for the new process
2. It assigns a unique ID number to the child process
3. It makes a logical copy of the context of the parent process. Since certain portion of process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory
4. It increments file and inode table counters for files associated with the process.
5. It returns the ID number of child process to the parent process, and a 0 value to the child process.

PROCESS CREATION

Algorithm for fork

```
input : none
output : to parent process, child PID number
         to child process, 0
{
    check for available kernel resources;
    get free process table slot, unique PID number;
    check that user not running too many process;
    mark child state "being created";
    copy data from parent process table to new child slot;
    increment counts on current directory inode and changed root(if applicable);
    increment open file counts in file table;
    make copy of parent context(u area, text, data, stack) in memory;
    push dummy system level context layer onto child system level context;
            dummy context contains data allowing child process
            to recognize itself, and start from here
            when scheduled;
```

PROCESS CREATION

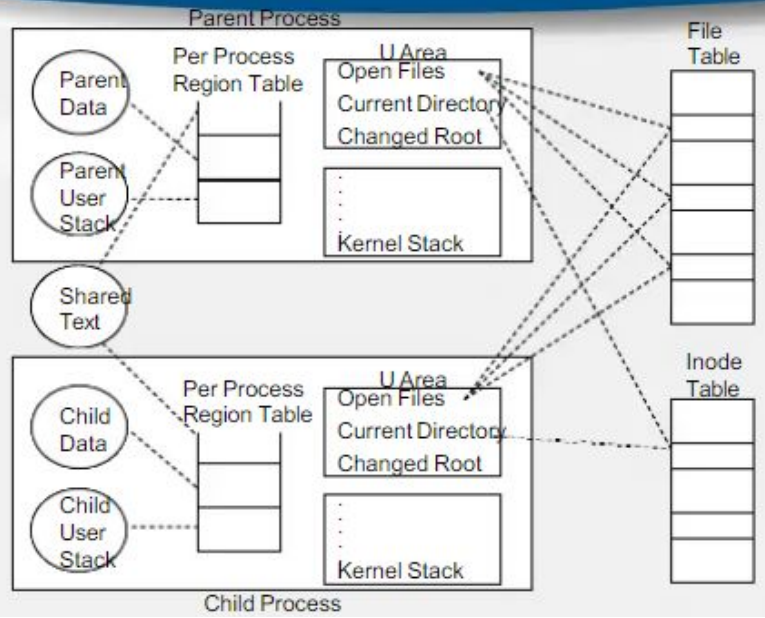
Process Creation:

Algorithm for fork

```
if ( executing process is parent )
{
    change child state to "ready to run";
    return(child ID);          /* from system to user */
}
else /* executing process is the child process */
{
    initialize u area timing fields;
    return(0);          /* to user */
}
}
```

PROCESS CREATION

Fork Creating a New Process Context



PROCESS CREATION

Process Creation:

Example-1:

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main() {
    // make two process which run same
   // program after this instruction
   fork();
    printf("Hello world!\n");
   return 0;
}
```

```
prsm@prsn-VirtualBox:~/Desktop/oslab/week-3$ cat fork4.c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main() {

    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
prsm@prsn-VirtualBox:~/Desktop/oslab/week-3$ gcc fork4.c
prsm@prsn-VirtualBox:~/Desktop/oslab/week-3$ ./a.out
Hello world!
prsm@prsn-VirtualBox:~/Desktop/oslab/week-3$ Hello world!
```

PROCESS CREATION

Process Creation:

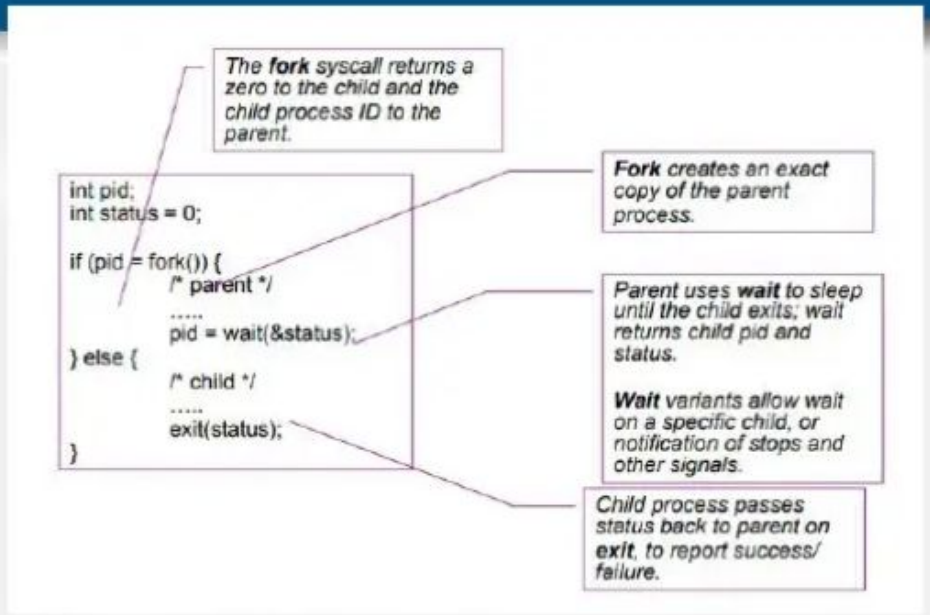
Example-2:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
   fork();
   fork();
   fork();
   printf("hello\n");
   return 0;
}
```

```
prsn@prsn-VirtualBox:~/Desktop/oslab/week-3$ cat fork3.c
#include  <stdio.h>
#include  <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
prsn@prsn-VirtualBox:~/Desktop/oslab/week-3$ gcc fork3.c
fork3.c: In function 'main':
fork3.c:5:5: warning: implicit declaration of function 'fork' [-Wimplicit-functi
on-declaration]
     fork();
     ^
prsn@prsn-VirtualBox:~/Desktop/oslab/week-3$ ./a.out
hello
prsn@prsn-VirtualBox:~/Desktop/oslab/week-3$ hello
hello
hello
hello
hello
hello
hello
```

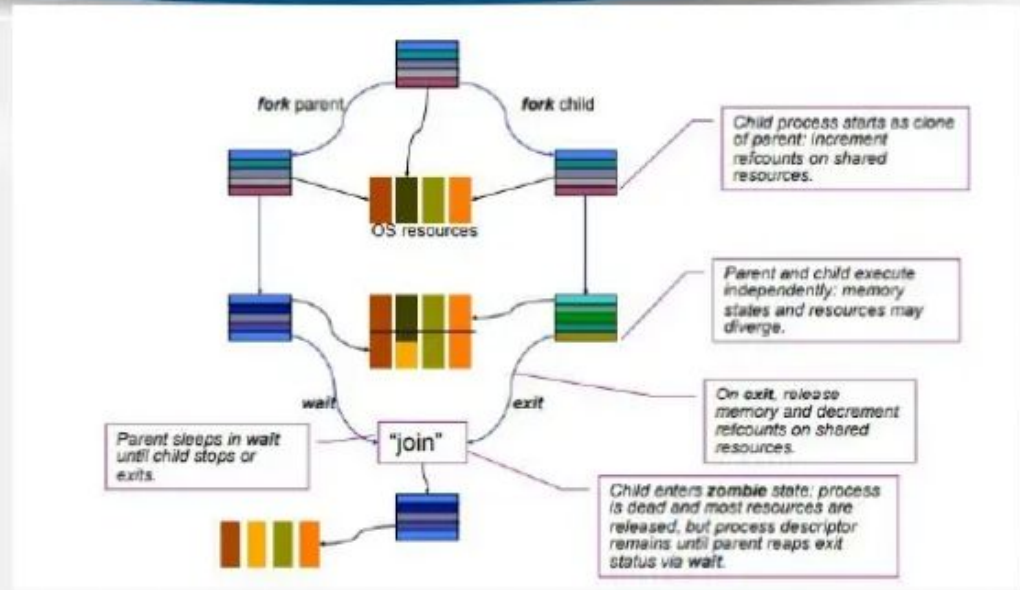
NOTE: For Examples please refer lab manual of week-3

PROCESS CREATION



PROCESS CONTROL

Fork/exit/wait example:



SIGNALS

Signals inform processes of the occurrence of asynchronous events.

A signal is a small message that notifies a process that an event of some types has occurred in the system.

- ✓ **Kernel Abstraction for exceptions and interrupts**
- ✓ **Sent from Kernel (sometimes at the request of another process) to a process.**
- ✓ **Different signals are identified by small integers.**
- ✓ **The only information in a signal is its ID and the fact that it arrived.**

Processes may send each other signal with the kill system call, or the kernel may send signals internally.

There are 19 signals in UNIX system

SIGNALS

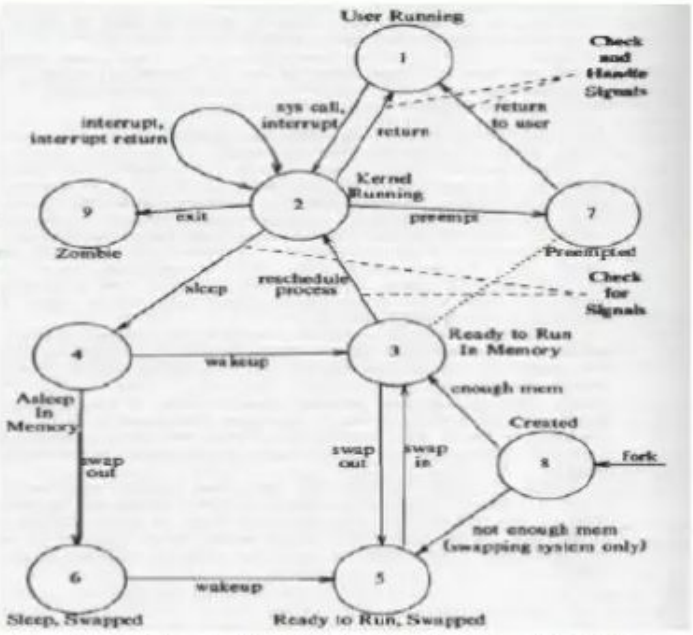


Figure 6.3: Checking and Handling Signals in the Process State Diagram

SIGNALS

Algorithm for recognizing signals

```
algorithm issig                    /* test for receipt of signals */
input:  none
output: true, if process received signals that it does not ignore
        false otherwise
{
    while (received signal field in process table entry not 0)
    {
        find a signal number sent to the process;
        if (signal is death of child)
        {
            if (ignoring death of child signals)
                free process table entries of zombie children;
            else if (catching death of child signals)
                return(true);
        }
        else if (not ignoring signal)
            return(true);
        turn off signal bit in received signal field in process table;
    }
    return(false);
}
```

SIGNALS

- ✓ A UNIX signal corresponds to an event
  - ✓ It is *raised* by one process (or hardware) to call another process's attention to an event
- ✓ UNIX has a fixed set of signals (Linux has 32 of them)
- ✓ signal.h defines the signals in the OS
- ✓ Appl. programs can use SIGUSR1 & SIGUSR2 for arbitrary signalling
- ✓ Raise a signal with kill(pid, signal)
- ✓ 3 ways to handle a *signal*
  - ❖ Ignore it: signal(SIG#, SIG_IGN)
  - ❖ Run the default handler: signal(SIG#, SIG_DFL)
  - ❖ Run the user handler: signal(SIG#, myHandler)

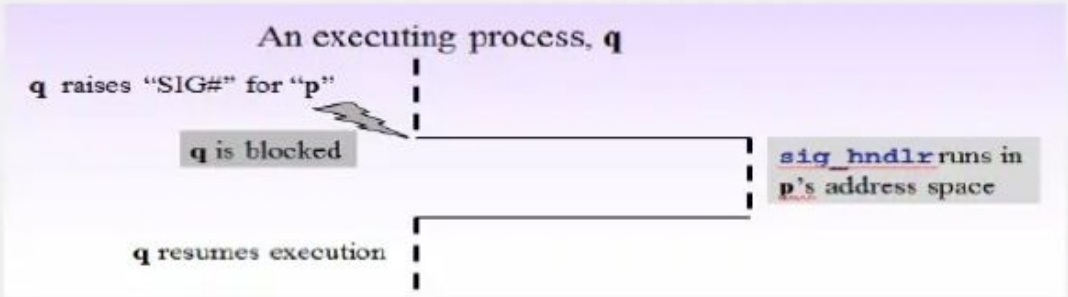
SIGNALS

SIGNAL HANDLING

```
/* code for process p */
. . .
signal(SIG#, sig_hndlr);
. . .
/* ARBITRARY CODE */
```

```
void sig_hndlr(...){
...
   /* handler code */

}
```



SIGNALS

Macro	#	Default action	Description
SIGHUP	1	quit	Hangup or death of controlling process.
SIGINT	2	quit	Keyboard interrupt.
SIGQUIT	3	core	Quit.
SIGILL	4	core	Illegal instruction.
SIGABRT	6	core	Abort.
SIGFPE	8	core	Arithmetic exception.
SIGKILL	9	quit	Kill (cannot be caught, blocked, or ignored).
SIGUSR1	10	quit	User-defined signal.
SIGSEGV	11	core	Segmentation violation (out of range address).
SIGUSR2	12	quit	User-defined signal.
SIGPIPE	13	quit	Write on a pipe or other socket with no one to read it.
SIGALRM	14	quit	Alarm clock.
SIGTERM	15	quit	Software termination signal (default signal sent by *kill*).
SIGCHLD	17	ignore	Status of child process has changed.
SIGCONT	18	none	Continue if stopped.
SIGSTOP	19	stop	Stop (suspend) the process.
SIGTSTP	20	stop	Stop from the keyboard.
SIGTTIN	21	stop	Background read from tty device.
SIGTTOU	22	stop	Background write to tty device.

SIGNALS

Signals:

Some of the important signals as follows:

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctl-c)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

SIGNALS

Signals:

HANDLING SIGNALS:

```
algorithm psig    /* handle signals after recognizing their existence */
input:   none
output: none
{
      get signal number set in process table entry;
      clear signal number in process table entry;
      if (user had called signal sys call to ignore this signal)
            return;              /* done */
      if (user specified function to handle the signal)
      {
            get user virtual address of signal catcher stored in u area;
            /* the next statement has undesirable side-effects */
            clear u area entry that stored address of signal catcher;
            modify user level context:
                        artificially create user stack frame to mimic
                        call to signal catcher function;
            modify system level context:
                        write address of signal catcher into program
                        counter field of user saved register context;
            return;
      }
      if (signal is type that system should dump core image of process)
      {
            create file named "core" in current directory;
            write contents of user level context to file "core";
      }
      invoke exit algorithm immediately;
}
```

Figure 6.5: Algorithm for handling signals

SIGNALS

HANDLING SIGNALS:

1. The kernel accesses the user saved register context, finding the program counter and stack pointer that it had saved for return to the user process.
2. It clears the signal handler field in the u area, setting it to the default state.
3. The kernel creates a new stack frame on the user stack, writing in the values of the program counter and stack pointer it had retrieved from the user saved register context and allocating new space, if necessary.
   a. The user stack looks as if the process had called a user-level function (the signal catcher) at the point where it had made the system call or where the kernel had interrupted it (before recognition of the signal).
4. The kernel changes the user saved register context: It resets the value for the program counter to the address of the signal catcher function and sets the value for the stack pointer to account for the growth of the user stack.

SIGNALS

HANDLING SIGNALS:

```
signal(SIGINT, INThandler);
signal(SIGQUIT, QUIThandler);
void INThandler(int sig)
{
    // SIGINT handler code
}
void QUIThandler(int sig)
{
    // SIGQUIT handler code
}
```

SIGNALS

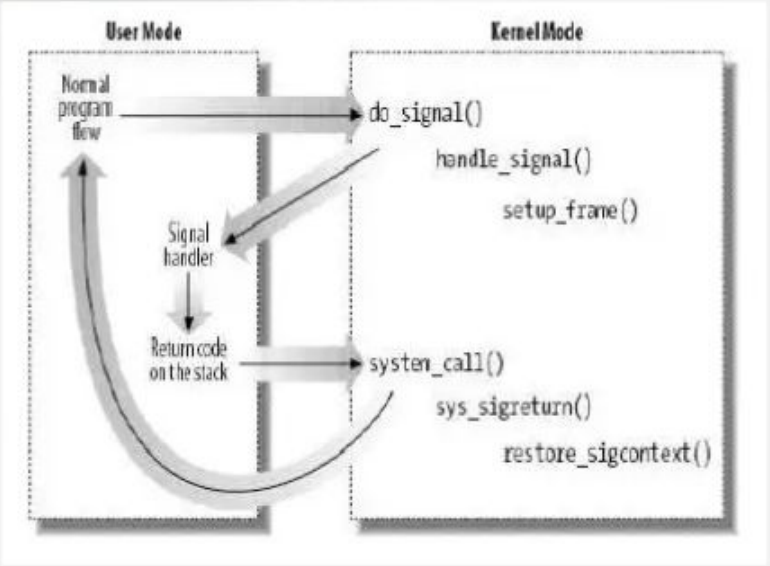
Signals:

How to catch Interrupt Signals?

If a handler has been established for the signal, the do_signal( ) function must enforce its execution.

It does this by invoking handle_signal( ):

handle_signal(signr, &info, &ka, oldset, regs);



SIGNALS

How to catch Interrupt Signals?

The setup_frame( ) function receives four parameters, which have the following meanings:

sig
Signal number

ka
Address of the k_sigaction table associated with the signal

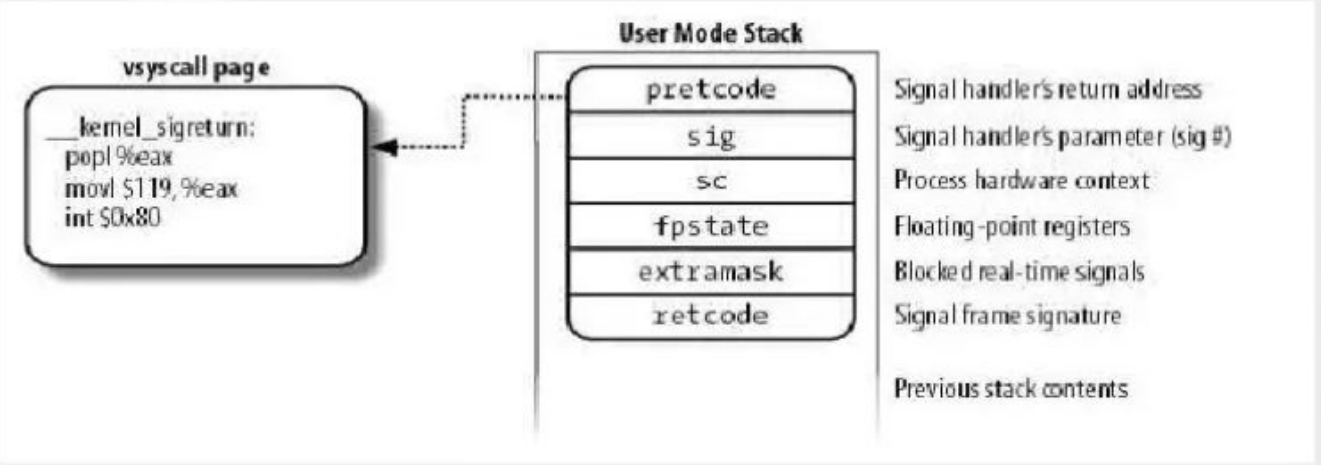
oldset
Address of a bit mask array of blocked signals

regs
Address in the Kernel Mode stack area where the User Mode register contents are saved

SIGNALS

How to catch Interrupt Signals?

A frame is a sigframe table that includes the following fields



SIGNALS

PROCESS GROUPS:

- ✓ A process group is a collection of one or more processes (usually associated with the same job) that can receive signals from the same terminal.
- ✓ Each process group has a unique process group ID. Process group IDs are similar to process IDs: they are positive integers and can be stored in a pid_t data type.
- ✓ The function getpgrp returns the process group ID of the calling process. The getpgid function took a *pid* argument and returned the process group for that process.

```
#include <unistd.h>
pid_t getpgrp(void);
/* Returns: process group ID of calling process */
pid_t getpgid(pid_t pid);
/* Returns: process group ID if OK, −1 on error */
```

SIGNALS

PROCESS GROUPS:

For getpgid, if pid is 0, the process group ID of the calling process is returned. Thus,

```
getpgid(0);
```

is equivalent to:

```
getpgrp();
```

Each process group can have a process group leader, whose process group ID equals to its process ID.

SIGNALS

PROCESS GROUPS:

Process group lifetime:

- ✓ **The process group life time is the period of time that begins when the group is created and ends when the last remaining process leaves the group.**
- ✓ **It is possible for a process group leader to create a process group, create processes in the group, and then terminate.**
- ✓ **The process group still exists, as long as at least one process is in the group, regardless of whether the group leader terminates.**
- ✓ **The last remaining process in the process group can either terminate or enter some other process group.**

SIGNALS

PROCESS GROUPS:

setpgid function

A process can join an existing process group or creates a new process group by calling setpgid.

#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);

The setpgid function sets the process group ID of the process whose process ID equals pid to pgid.

Arguments:

- ✓ **If pid == pgid, the process specified by pid becomes a process group leader.**
- ✓ **If pid == 0, the process ID of the caller is used.**
- ✓ **If pgid == 0, then the specified pid is used as the process group ID.**

SIGNALS

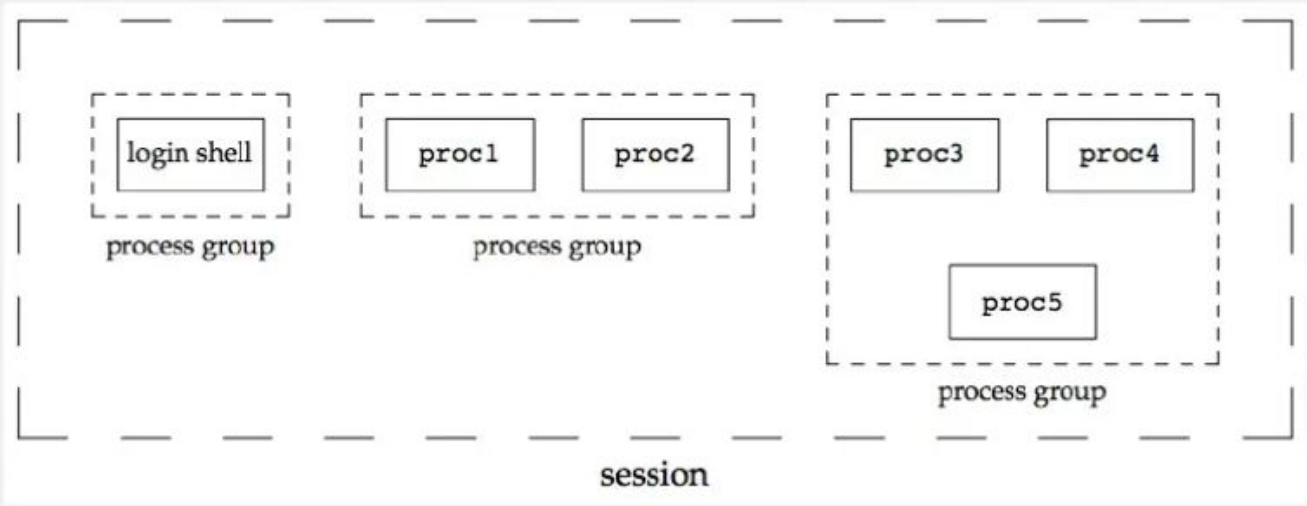
PROCESS GROUPS:

```
#include <signal.h>
main()
{
        register int i;

        setpgrp();
        for (i = 0;  i < 10;  i++)
        {
                if (fork() == 0)
                {
                        /* child proc */
                        if (i & 1)
                                setpgrp();
                        printf("pid = %d pgrp = %d\n", getpid(), getpgrp());
                        pause();        /* sys call to suspend execution */
                }
        }
        kill(0, SIGINT);
}
```

SIGNALS

PROCESS GROUPS:



SIGNALS

To Send Signal to a Process:

Use Unix system call kill() to send a signal to another process:

```
int kill(pid_t pid, int sig);
```

The kill command can also be used to send a signal to a process:

```
kill –l /* list all signals */
kill –XXX pid1 pid …… pid
```

- ✓ In the above XXX is the signal name without the initial letters SIG.
- ✓ kill –KILL 1357 2468 kills process 1357 and 2468.
- ✓ kill –INT 6421 sends a SIGINT to process 6421.
- ✓ A kill without a signal name is equivalent to SIGTERM.
- ✓ -9 is equal to –SIGKILL.

SIGNALS

To Send Signal to a Process:

- ✓ If **pid is a positive integer**, the kernel sends the signal to the process with process ID pid.
- ✓ If **pid is 0**, the kernel sends the signal to all processes in the sender's process group.
- ✓ If **pid is -1**, the kernel sends the signal to all processes whose real user lD equals the effective user ID of the sender. If the sending process has effective user ID of super user, the kernel sends the signal to all processes except processes 0 and 1.
- ✓ If **pid is a negative integer but not - 1** , the kernel sends the signal to all processes in the process group equal to the absolute value of pid.
- ✓ In all cases, if the sending process does not have effective user ID of super user, or its real or effective user ID do not match the real or effective user ID of the receiving process, kill fails.

PROCESS TERMINATION

Processes on a UNIX system terminate by executing the exit system call. An exiting process enters the zombie state, relinquishes its resources, and dismantles its context except for its slot in the process table. The syntax for the call is

```
exit (status);
```

Where the value of status is returned to the parent process for its examination.