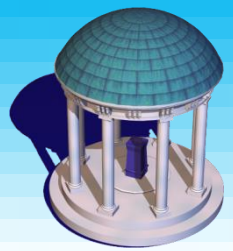


Advanced LINUX Programming

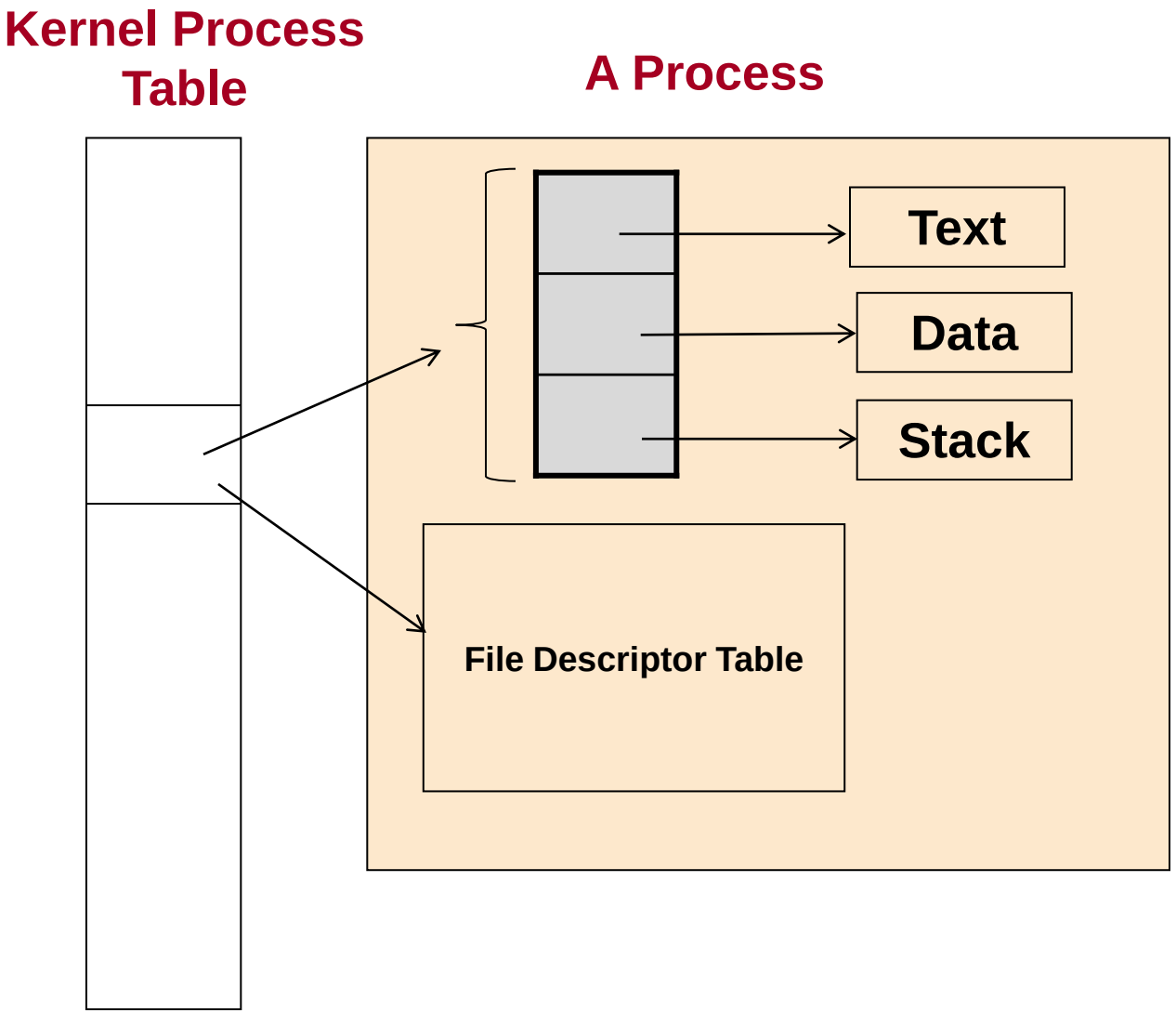
Unit - III

Unix Process and Signals

Process and its Structure

- A process consists of **text** (machine code), **data** and **stack.**
- Every process has a unique process ID, a non-negative integer.
- Process IDs can be reused. As processes terminate, their IDs become candidates for reuse.
- Process ID 0 is usually the **scheduler process** and is often known as the swapper /dispatcher.
- Process ID 1 is usually the **init process** and is invoked by the kernel at the end of the bootstrap procedure. The init process never dies.

Data Structure for Processes



Process Control

Process Control includes

- Creation of New Processes
- Process Execution
- Process Termination

Process Creation

- An existing process can create a new one by calling the **fork function**.

```
#include <unistd.h>

pid_t fork(void);
```

returns: 0 in child, process ID of child in parent, -1 on error.

- The new process created by fork is called the child process.
- This function is called once but returns twice.
- The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.

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

Parent

```
int pid ;
pid =  Process id
of child
```

fork

Child

```
int pid ;
pid = 0
```

Child

```
int pid ;
pid = fork();
if (pid == 0)
  {
     printf("Child");
  }
else
       printf("Parent") ;
}
```

Parent

```
int pid ;
pid = fork();
if (pid == 0)
  {
     printf("Child");
  }
else
    printf("Parent");
}
```

Parent

```
int pid ;
pid = fork();
if (pid == 0)
 {
   printf("Child");
 }
else
   printf("Parent");
}
```

Child

```
int pid ;
pid = fork();
if (pid == 0)
 {
   printf("Child");
 }
else
      printf("Parent") ;
}
```

```
#include<unistd.h>

main (int argc, char *argv[])
 {
    int pid ;
   pid =  fork() ;

   if (pid ==0)
         printf("Child");    // Executed by child

  else
        printf("Parent") ; // Executed by parent
  }
```

```
#include <unistd.h>

pid_t getpid (void);
```

returns: process ID of calling process.

```
pid_t getppid (void);
```

returns: parent process ID of calling process.

The pid is a 32bit unsigned integer, which usually ranges from 0 to 32767

gid_t getgid (void);

returns: real group ID of calling process.

gid_t getegid (void);

returns: effective group ID of calling process.

- **Write a program to show the implementation of fork**

```
#include<stdio.h>
#include<unistd.h>
int g=6;
main()
{
 int pid, var=88;  pid=fork();
if(pid==0) {
 printf("child process %d\n ", getpid());
 printf("Its parent process %d\n",
                getppid());
 g++;   var++;   }
else  {
  sleep(2);
 printf("parent process
%d\n", getpid());
 printf("its parent process
%d\n", getppid());
}
printf("pid=%d, g=%d, var
=%d\n", pid, g, var);
}
```

Uses of fork:

- Process can duplicate itself.
- Parent and child execute different sections of code at the same time.

vfork :

- The function vfork has the same calling sequence and same return values as fork.
- It creates the new process, just like fork but without copying the address space of the parent into the child.
- vfork guarantees that the child runs first, when child calls exec or _exit system call the parent resumes.

Properties inherited by child

- real user, group ID and effective user, group ID
- session ID
- controlling terminal
- current working directory
- root directory
- file mode permission
- Signal mask and dispositions
- environment
- shared memory
- resources

Difference between parent & child

- return value of fork
- process ID are different
- different parent ID's
- file locks are not inherited
- pending alarms are cleared

	fork()	vfork()
Basic	Child process and parent process has separate address spaces.	Child process and parent process shares the same address space.
Execution	Parent and child process execute simultaneously.	Parent process remains suspended till child process completes its execution.
Modification	If the child process alters any page in the address space, it is invisible to the parent process as the address space are separate.	If child process alters any page in the address space, it is visible to the parent process as they share the same address space.
Copy-on-write	fork() uses copy-on-write as an alternative where the parent and child shares same pages until any one of them modifies the shared page.	vfork() does not use copy-on-write.

Process termination

1. **Normal termination**
   - Executing return from main
   - Calling exit function
2. **Abnormal termination**
   - Calling abort
   - Signal

Orphan process

- Existence of a child process in the absence of parent creates a process called orphan process.
- Solution : kernel makes **init process** as parent of all orphan processes.

Zombie process

- Existence of a parent process in the absence of child creates a process called zombie process.
- Solution : child signals to the parent that it is leaving using a signal called SIGCHLD. Parent uses wait function to check the signal.

- **Write a program to show the implementation of orphan process**

```
#include<stdio.h>
#include<unistd.h>
main() {
 int pid;  pid=fork();
if(pid==0)  {
 printf("child process %d\n ", getpid());
 printf("its parent process %d\n",getppid());
 sleep(10);
 printf("child process %d\n",getpid());
 printf("its parents process %d\n",getppid());
}
else  {
 printf("parent process
%d\n",getpid());
 printf("its parent process
%d\n",getppid());
}   }
```

wait and waitpid function

Syntax

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

pid_t wait(int *status);

pid_t waitpid(pid_t  pid, int *status, int op);

        return process ID if OK, -1 error
```

- wait will block the parent until the child terminates.
- waitpid will block the parent until all children terminate.

Options of waitpid( )

- **WNOHANG** waitpid will not block if a child specified by pid is not immediately available.
- **WUNTRACED** child process which are stopped and whose status are not reported are returned
- if pid==-1 wait for any child process.
- if pid > 0 waits for any child whose process ID equals pid
- if pid==0 waits for any child whose process group ID equals that of calling process
- if pid < -1 waits for any child whose process group ID equals absolute pid.

macros used to tell how the process terminates are

- **WIFEXITED(status)** true if status was returned from a child which terminated normally
- **WEXITSTATUS(status)** returns the exit value
- **WIFSIGNALED(status)** true if status was returned from a child which terminated abnormally
- **WTERMSIG(status)** returns the value
- **WIFSTOPPED(status)** true if status was returned from a child which was stopped
- **WSIGSTOP(status)** returns the value

- **Write a program to display the status of terminating child**

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
main()  {
pid_t pid, status;
pid=fork();
if(pid==0)    exit(0);
if(wait(&status)!=pid)
    printf("pid error");
exitStatus(status);  }
```

```
void exitStatus(int st)  {
if(WIFEXITED(st))
 printf("normal termination , exit
status %d\n", WEXITSTATUS(st));
else if(WIFSIGNALED(st))
 printf("abnormal termination, signal
no =%d \n",    WTERMSIG(st));
else if(WIFSTOPPED(st))
 printf("child stopped, signal number
%d\n", WSTOPSIG(st));
}
```

exec functions

- fork is used to create a child which executes another program using exec function.
- The exec() functions all replace the current program running within the process with another program.
- There are two families of exec() functions
  - "l" family (list), and
  - "v" family (vector)
- Each exec() call can choose different ways of finding the executable and whether the environment is delivered in the form of a list or an array (vector)

Six functions in exec

- execl() execv()
- execlp() execvp()
- execle() execve()
- The base of each function is exec, followed by one or more letters:
  - **e** - An array of pointers to environment arguments is explicitly passed to the child process.
  - **l** - Command line arguments are passed individually to the function.
  - **p** - Uses the PATH argument variable to find the file to be executed.
  - **v** - Command line arguments are passed to the function as an array of pointers.

execl... functions

- **int execl(const char * path, const char * arg0, ...);**
  - executes the command at path, passing it the environment as a *list:* arg0 ... arg*n*
  - execl family breaks down argv into its individual constituents, and then passes them as a list to the execl function
- **int execlp(const char * path, const char * arg0, ...);**
  - same as execl, but uses $PATH resolution for locating the program in path, thus an absolute pathname is not necessary
- **int execle(const char * path, const char * arg0, ... char * envp[]);**
  - allows you to specifically set the new program's environment, which *replaces* the default current program's environment

Examples

- execl()

```
#include <unistd.h>
 execl ("/bin/ls", "ls", "-l", (char *)0);
```

- execle()

```
 char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
        ...
 execle ("/bin/ls", "ls", "-l", (char *)0, env);
```

- execlp()

```
  execlp ("ls", "ls", "-l", (char *)0);
```

execv... functions

- **int execv(const char * path, char *const argv[]);**
  - executes the command at path, passing it the environment contained in a single argv[] vector.
- **int execvp(const char * path, char *const argv[]);**
  - same as execv, but uses $PATH resolution for locating the program in path
- **int execve(const char * path, char *const argv[], char * const envp[]);**

- execv()

```
char *cmd[] = { "ls", "-l", (char *)0 };
execv ("/bin/ls", cmd);
```

- execvp()

```
char *cmd[] = { "ls", "-l", (char *)0 };
execvp ("ls", cmd);
```

- execve()

```
char *cmd[] = { "ls", "-l", (char *)0 };
char *env[] = { "HOME=/usr/home",  "LOGNAME=home", (char *)0 };
execve ("/bin/ls", cmd, env);
```

- **Write a program for displaying the details of files present in the directory using execl**

```
#include<stdio.h>
#include<unistd.h>
main()
{
 printf("Files in Directory are:  \n");
 execl("/bin/ls","ls", "-l",(char*)0);
}
```

system Function

Allows us to execute a command string from within a program.

Ex:

```
system("pwd");
```

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

Signal handling

- Signals are software interrupts
- Kernel sets a flag in process table when signal occurs
- Signals represent a primitive way of performing inter process communication
- It is a mechanism used by kernel to communicate the occurrence of an event to a process. The process often responds by terminating itself, but there are other actions the process can take – ignore it or invoke a user-defined function for instance.
- The action taken by a process on receipt of a signal is known as **disposition**.

- Signals are represented by integer numbers and its symbolic name.

  ex : SIGINT signal number=2

- No signal has 0 signo.
- The event that generates a signal can take place in hardware, OS or in user area.
- Signal can be asynchronous or synchronous. Asynchronous signal is one over which there is no control, it is because of events not directly connected with execution of program instructions.

ex : ^c to generate SIGINT by user

- SIGINT -- Terminate
- SIGSTOP -- Stop process
- SIGABRT -- Terminate / write Core
- SIGKILL -- Terminate
- SIGIO -- Terminate
- SIGCHLD -- Ignore
- SIGTERM -- Terminate
- SIGPIPE -- Terminate
- SIGALRM -- Terminate
- SIGQUIT -- Terminate
- SIGBUS -- Terminate / write Core

- SIGSEGV -- Terminate / write Core
- SIGSYS -- Terminate / write Core
- SIGUSR1 -- Terminate
- SIGUSR2 -- Terminate
- SIGPWR -- Terminate

Signal disposition

- When process receives a signal it can do 3 things
  - Ignore signal – process runs as if nothing happened

    ex : SIGCHLD sent to parent when child dies
  - Default action – Terminating a process, this the default disposition of most signals.

    ex : SIGINT and SIGHUP terminate process
  - Catch Signals – Provide a signal handler for the received signal

- **System calls used with signaling mechanism are**
  - signal
  - kill
  - alarm
  - pause
  - raise
  - sleep
  - abort
- To override the default action, **signal** system call is used.
- It doesn't generate a signal but only specifies its disposition by invoking the signal handling function.
- This function is invoked automatically when process receives a signal.

Signal function

sighandler_t signal(int signo, sighandler_t handler);

arguments :

- signal number
- pointer to a signal handling function or **SIG_IGN, SIG_DFL**

ex : signal(SIGALRM, alrm_handler);

signal(SIGINT, SIG_IGN); changes default to ignore

signal(SIGINT, SIG_DFL); restore default

- signal handlers are executed in user mode

- **WAP for using alarm call to set up a timer which times out in 5 sec. The program prompts the user for a filename which is displayed if the user input's it in 5 sec. If user is late in responding SIGALRM is generated, which invokes a signal handler to set the default filename to f1**

```
#include<signal.h>
void alrm_handler(int signo);
char buf[100]="f1";
main()
{
  int n;
```

```
if(signal(SIGALRM, alrm_handler)==SIG_ERR)
 printf("error");
    printf("enter filename  :");
    alarm(5);
    n=read(STDIN_FILENO, buf, 100);
    if(n>1)
      printf("filename  %s ", buf);
       exit(0);
}
```

```
void alrm_handler(int signo)
{
    printf("signal  %d  received, default
        filename %s",  signo, buf);
     exit(1);
}
```

signals are unreliable

- Unreliable signals mean that signals will be lost, a signal could occur and the process would never know about it.
- Each time when the signal occurs its action was reset to its default

Signal are reliable

- A pending state is introduced in between generation state and delivery state.
  - Generation state : When signal is created and generated.
  - Pending state : The state between the both 2 states where the signal is blocked.
  - Delivery state : When the process performs the action needed for signal the state of process is delivery state.
- The signal in pending state will not set to default state so the signals are reliable.

Write a program to catch SIGUSR1 and SIGUSR2 signals and display appropriate message

Execution scenario

$**a.out &** -- start process in background

[1]372 -- job control prints job number and process id

$ **kill –USR1 372** -- send to process SIGUSR1

received SIGUSR1

$ **kill –USR2 372** -- send to process SIGUSR2

received SIGUSR2

$ **kill 372** -- send SIGTERM

[1] + Terminated -- a.out &

Process is terminated because it does not catch the signal and the default action for this signal is termination

```
main()
{
  static void sigusr(int);
  if(signal(SIGUSR1,sigusr)==SIG_ERR)
     printf("error");
  else if(signal(SIGUSR2,sigusr)==SIG_ERR)
       printf("error");
   for(;;) pause();
}
```

```
static void sigusr(int signo)
{
  if(signo==SIGUSR1)
     printf("received SIGUSR1");
  else if(signo==SIGUSR2)
       printf("received SIGUSR2");
}
```

kill, raise, sleep functions

- **raise**

  Process sends signal to itself

  int raise(int signo);

  return 0 if OK -1 on error

- **sleep**

  unsigned int sleep(unsigned int seconds);

- **kill**

```
#include <sys/types.h>
#include <signal.h>
```

int kill(pid_t pid, int signo);

alarm , pause, abort functions

- **alarm**

  sets a timer that expires at a specified time. SIGALRM is generated when timer expires.

  #include<unistd.h>

  unsigned int alarm(unsigned int seconds);

- **pause**

  Pause suspends calling process until signal is caught

  #include<unistd.h>

  int pause(void);

- **abort**

  #include<stdlib.h>

  void abort(void);

End of Unit -III