www.oranelabs.com
ORANE LABS
SIIC IIT KANPUR

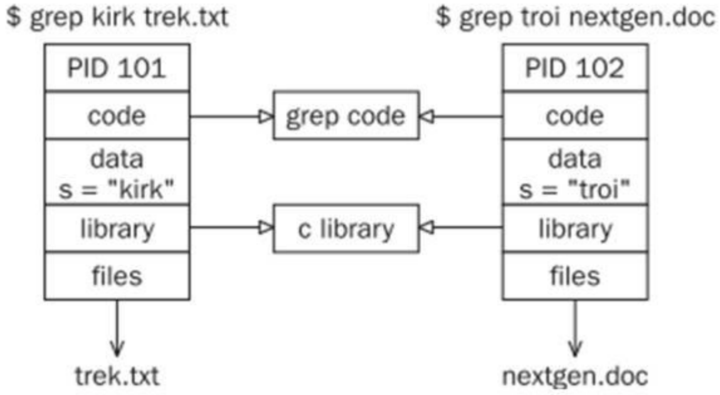
PROCESS MANAGEMENT

What is a Process

- Each instance of a running program constitutes a process.
- A typical Linux system allows multiple users to run many programs or even many instances of the same program.
- In such a case, multiple processes are running and being managed by the Operating system.

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Process Structure



Information inside a Program File

- **Binary format identification**: Each program file includes meta information describing the format of the executable file.
- **Machine-language instructions:** These encode the algorithm of the program.
- **Program entry-point address:** This identifies the location of the instruction at which execution of the program should commence.
- **Data:** The program file contains values used to initialize variables and also literal constants used by the program.
- **Symbol and relocation tables:** These describe the locations and names of functions and variables within the program.
- **Shared-library and dynamic-linking information**

www.oranelabs.com
ORANE LABS
SIIC IIT KANPUR

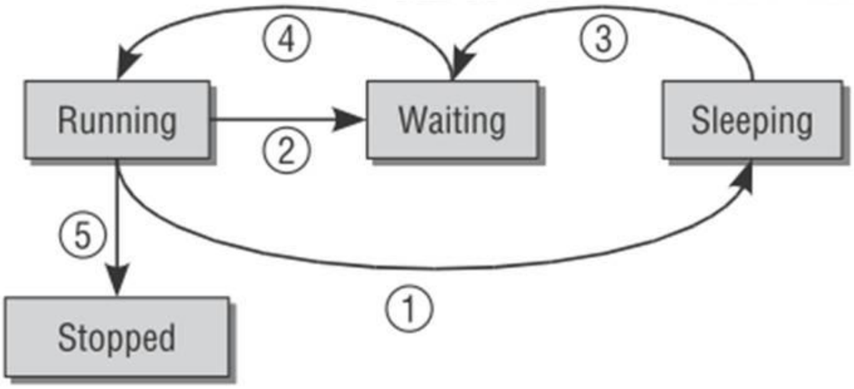
Process Priorities

- Distinction based on priorities:
  - Hard real-time processes are subject to strict time limits during which certain tasks must be completed.
    - Example: Flight control System.
    - Normal Linux Kernel does not support this type.
    - Modified Linux such as RTLinux, Xenomai or RATI.
  - Soft real-time processes are a softer form of hard real-time processes.
    - Although quick results are still required, it is not the end of the world if they are a little late in arriving.
  - Normal Processes.

Process Life Cycle

- A process may have one of the following states:
  - Running — The process is executing at the moment.
  - Waiting — The process is able to run but is not allowed to because the CPU is allocated to another process. The scheduler can select the process, if it wants to, at the next task switch.
  - Sleeping — The process is sleeping and cannot run because it is waiting for an external event. The scheduler cannot select the process at the next task switch.

www.oranelabs.com
ORANE LABS
SIIC IIT KANPUR



The Process ID

- Each process is represented by a unique identifier, the process ID .
- The pid is guaranteed to be unique at any single point in time.
- The idle process—the process that the kernel “runs” when there are no other runnable processes—has the pid 0.
- The first process that the kernel executes after booting the system, called the init process, has the pid 1.

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Process ID Allocation

- By default, the kernel imposes a maximum process ID value of 32768. This is for compatibility with older Unix systems, which used smaller 16-bit types for process IDs.
- System administrators can set the value higher via */proc/sys/kernel/pid_max*, trading a larger pid space for reduced compatibility.
- The kernel allocates process IDs to processes in a strictly linear fashion. If pid 17 is the highest number currently allocated, pid 18 will

Preemptive Multitasking

- The kernel uses this form for scheduling processes.
- Normally a kernel is in user mode in which it may access only its own data and cannot therefore interfere with other applications in the system.
- If a process wants to access system data or functions, it must switch to kernel mode via system calls.
- A second way of switching from user mode to

www.oranelabs.com
ORANE LABS
SIIC IIT KANPUR

The preemptive scheduling model of the kernel

- Normal processes may always be interrupted — even by other processes.
- If the system is in kernel mode and is processing a system call, no other process in the system is able to cause withdrawal of CPU time.
- Interrupts can suspend processes in user mode and in kernel mode. They have the highest priority because it is essential to handle them as soon as possible after they are

Process Representation

- All algorithms of the Linux kernel concerned with processes and programs are built around a data structure named task_struct and defined in include/sched.h.
- This is one of the central structures in the system.

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

What is represented in this structure

- State and execution information.
- Information on allocated virtual memory.
- Process credentials such as user and group ID, capabilities, and so on.
- Files used.
- Thread information, which records the CPU-specific runtime data of the process.
- Information on interprocess communication required when working with other applications.
- Signal handlers used by the process to respond to incoming signals.

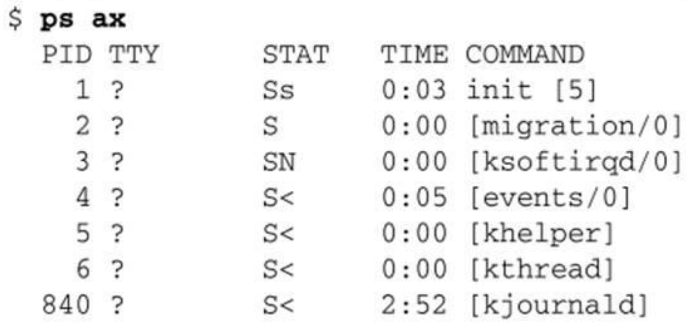
Process Types

- New processes are generated using the fork and exec system calls:
  - *fork* generates an identical copy of the current process; this copy is known as a child process.
    - All resources of the original process are copied in a suitable way so that after the system call there are two independent instances of the original process.
  - *exec* replaces a running process with another application loaded from an executable binary file.

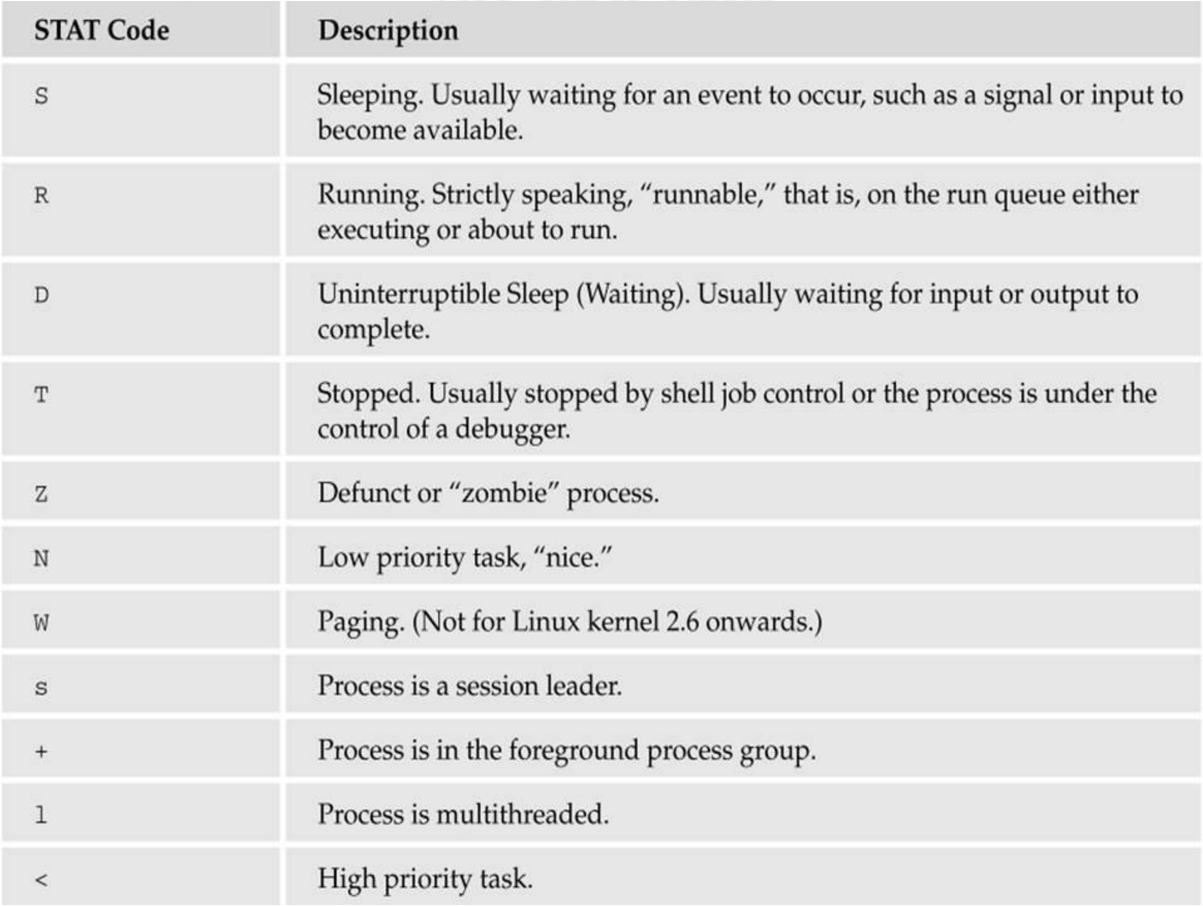
www.oranelabs.com
ORANE LABS
SIIC IIT KANPUR

Viewing Processes

- The ps command shows the processes you're running, the process another user is running, or all the processes on the system.
- For example the ps –ax command shows the system processes.



```
$ ps ax
  PID TTY          STAT   TIME COMMAND
    1 ?            Ss     0:03 init [5]
    2 ?            S      0:00 [migration/0]
    3 ?            SN     0:00 [ksoftirqd/0]
    4 ?            S<     0:05 [events/0]
    5 ?            S<     0:00 [khelper]
    6 ?            S<     0:00 [kthread]
  840 ?            S<     2:52 [kjournald]
```



STAT Code	Description
S	Sleeping. Usually waiting for an event to occur, such as a signal or input to become available.
R	Running. Strictly speaking, "runnable," that is, on the run queue either executing or about to run.
D	Uninterruptible Sleep (Waiting). Usually waiting for input or output to complete.
T	Stopped. Usually stopped by shell job control or the process is under the control of a debugger.
Z	Defunct or "zombie" process.
N	Low priority task, "nice."
W	Paging. (Not for Linux kernel 2.6 onwards.)
s	Process is a session leader.
+	Process is in the foreground process group.
l	Process is multithreaded.
<	High priority task.

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Starting New Processes

- We can cause a program to run from inside another program and thereby create a new process by using the system library function.

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

Example

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
printf("Running ps with system\n");
system("ps -ax");
printf("Done.\n");
exit(0);
}
```

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Running the Process in background

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
printf(“Running ps with system\n”);
system(“ps -ax  &”);
printf(“Done.\n”);
exit(0);
}
```

Runs in Background

pid_t

- Programmatically, the process ID is represented by the pid_t type, which is defined in the header file <sys/types.h>.
- On Linux, however, pid_t is generally a typedef to the C int type.

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Obtaining the Process ID and Parent Process ID

- The getpid() system call returns the process ID of the invoking process:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid (void);
```

- The getppid() system call returns the process ID of the invoking process's parent:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid (void);
```

Running a New Process

- In Unix, the act of loading into memory and executing a program image is separate from the act of creating a new process.
- One system call loads a binary program into memory, replacing the previous contents of the address space, and begins execution of the new program. The functionality is provided by *the exec family of calls*.
- The act of creating a new process is called forking, and this functionality is provided by

The Exec Family of Calls

- There is no single exec function; instead, there is a family of exec functions built on a single system call. Let’s first look at the simplest of these calls, execl():
- Syntax:

```
#include <unistd.h>
int execl (const char *path,
        const char *arg,
        ...);
```

- Example:

```
int ret;
ret = execl ("/bin/vi", "vi", NULL);
```

- As another example, if you wanted to edit the file /home/kidd/hooks.txt, you could execute the following code:

```
int ret;
ret = execl ("/bin/vi", "vi", "/home/kidd/hooks.txt",
  NULL);
```

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

The rest of the Family

- In addition to execl(), there are five other members of the exec family:

```
#include <unistd.h>
✓int execlp (const char *file,
        const char *arg,
        ...);
✓int execle (const char *path,
        const char *arg,
        ...,
        char * const envp[]);
✓int execv (const char *path, char *const argv[]);
✓int execvp (const char *file, char *const argv[]);
✓int execve (const char *filename,
        char *const argv[],
        char *const envp[]);
```

Code Fragments

- #include <unistd.h>
- /* Example of an argument list */
- /* Note that we need a program name for argv[0] */
  - char *const ps_argv[] =
  - {“ps”, “-ax”, 0};
- /* Example environment, not terribly useful */
  - char *const ps_envp[] =
  - {“PATH=/bin:/usr/bin”, “TERM=console”, 0};
- /* Possible calls to exec functions */
  - execl(“/bin/ps”, “ps”, “-ax”, 0);   /* assumes ps is in /bin */
  - execlp(“ps”, “ps”, “-ax”, 0);   /* assumes /bin is in PATH */
  - execle(“/bin/ps”, “ps”, “-ax”, 0, ps_envp); /* passes own environment */
  - execv(“/bin/ps”, ps_argv);
  - execvp(“ps”, ps_argv);
  - execve(“/bin/ps”, ps_argv, ps_envp);

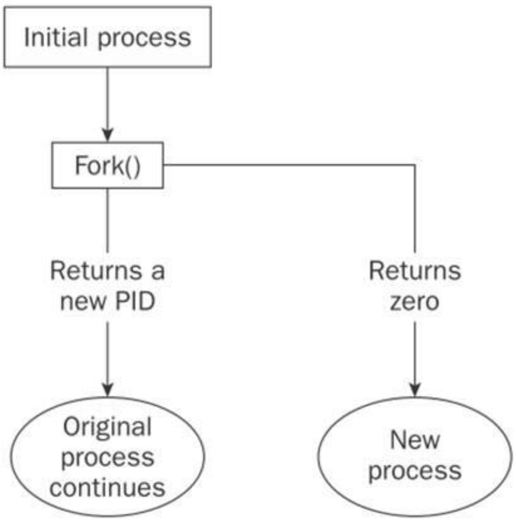
www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Another Example

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("Running ps with execlp\n");
execlp("ps", "ps", "ax", 0);
printf("Done.\n");
exit(0);
}
```

The fork() System Call

- A new process running the same image as the current one can be created via the fork() system call:
  - **#include <sys/types.h>**
  - **#include <unistd.h>**
  - **pid_t fork (void);**



www.oranelabs.com ORANE LABS SIIC IIT KANPUR

A typical code fragment using fork is

```
pid_t new_pid;
new_pid = fork();
switch(new_pid) {
case -1 :    /* Error */
break;
case 0 :     /* We are child */
break;
default :    /* We are parent */
break;
}
```

Example

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
pid_t pid;
char *message;
int n;
printf(“fork program starting\n”);
pid = fork();
switch(pid)
{
case -1:
perror(“fork failed”);
exit(1);
case 0:
message = “This is the child”;
n = 5;
break;
default:
message = “This is the parent”;
n = 3;
break;
}
for(; n > 0; n--) {
puts(message);
sleep(1);
}
exit(0);
}
```

Copy-on-write

- In early Unix systems, Upon invocation, the kernel created copies of all internal data structures, duplicated the process' page table entries, and then performed a page-by-page copy of the parent's address space into the child's new address space.
- This was very much time consuming.

Optimization

- Modern Unix systems behave more optimally.
- Instead of a wholesale copy of the parent's address space, modern Unix systems such as Linux employ copy-on-write (COW) pages.

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

The Premise

- If multiple consumers request read access to their own copies of a resource, duplicate copies of the resource need not be made.
- Instead, each consumer can be handed a pointer to the same resource.
- So long as no consumer attempts to modify its "copy" of the resource, the illusion of exclusive access to the resource remains, and the overhead of a copy is avoided.

Contd.

- If a consumer does attempt to modify its copy of the resource, at that point, the resource is transparently duplicated, and the copy is given to the modifying consumer.
- Hence the name: *the copy occurs only on write*.
- The primary benefit is that if a consumer never modifies its copy of the resource, a copy is never needed.

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

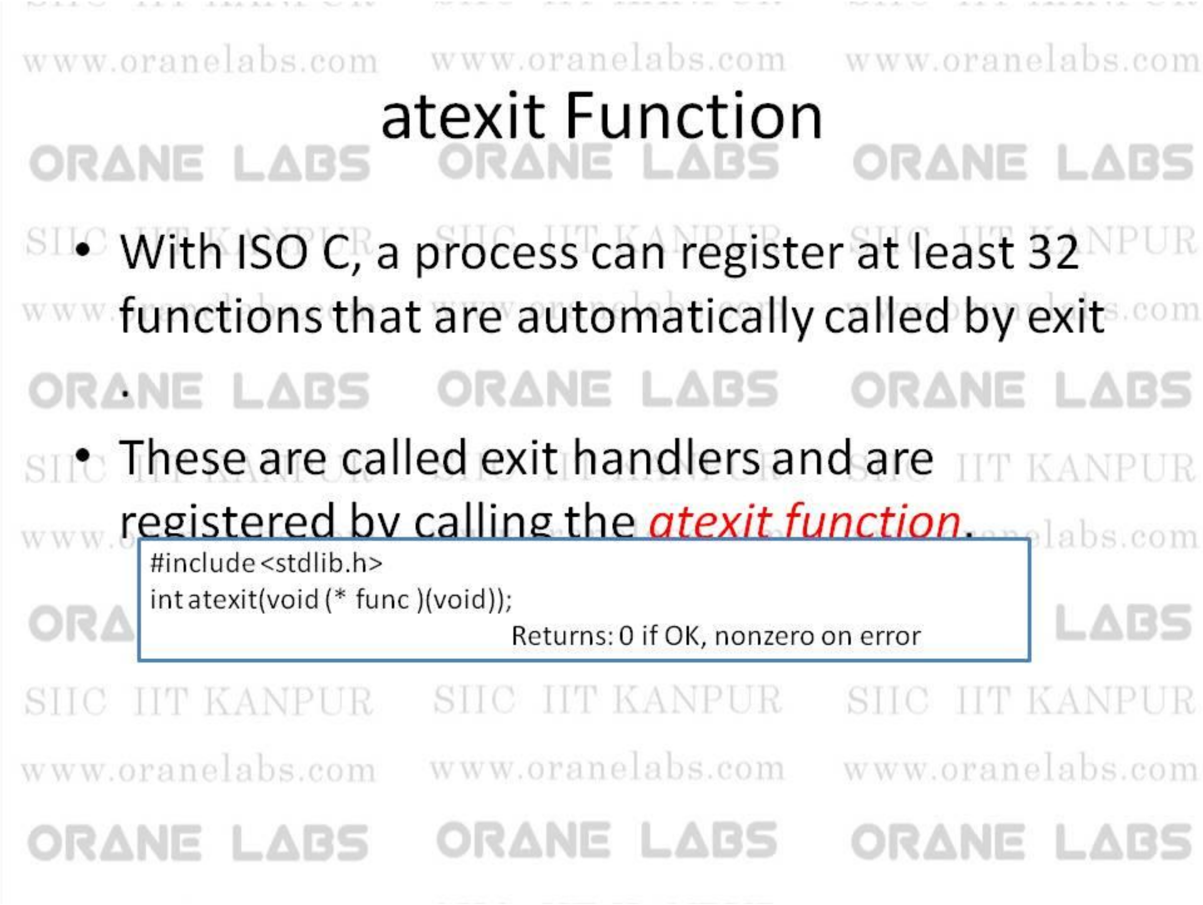
www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Terminating a Process

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

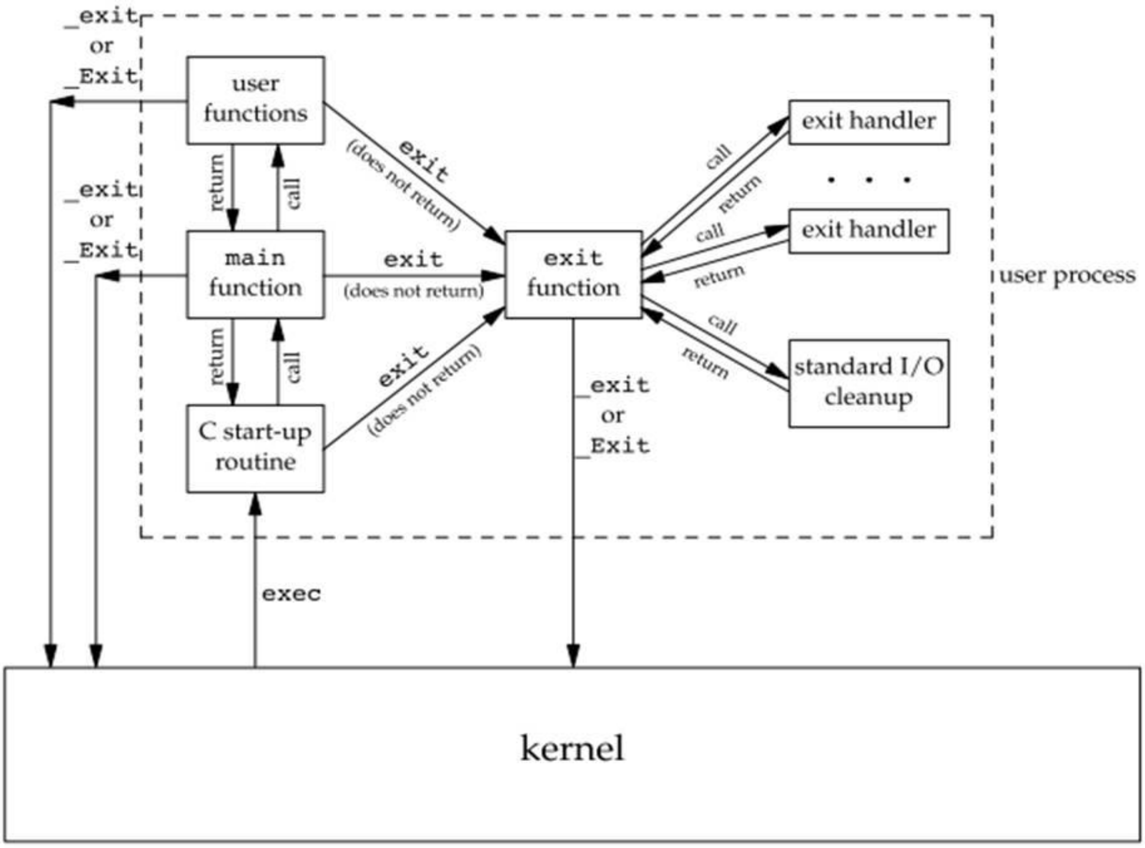


atexit Function

- With ISO C, a process can register at least 32 functions that are automatically called by exit .
- These are called exit handlers and are registered by calling the *atexit function*.

```
#include <stdlib.h>
int atexit(void (* func )(void));
                                   Returns: 0 if OK, nonzero on error
```

SIIC IIT KANPUR www.oranelabs.com ORANE LABS



www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Example

```
#include<stdlib.h>
void my_exit1(void);
void my_exit2(void);
int main(void)
{
if (atexit(my_exit2) != 0)
printf("can't register my_exit2");
if (atexit(my_exit1) != 0)
printf("can't register my_exit1");
if (atexit(my_exit1) != 0)
printf("can't register my_exit1");
printf("main is done\n");
return(0);
}
```

```
my_exit1(void)
{
printf("first exit handler\n");
}

my_exit2(void)
{
printf("second exit handler\n");
}
```

Waiting for a Process

- When you start a child process with fork, it takes on a life of its own and runs independently. Sometimes, you would like to find out when a child process has finished.
    - **#include <sys/types.h>**
    - **#include <sys/wait.h>**
    - **pid_t wait(int *stat_loc);**
- The wait system call causes a parent process to pause until one of its child processes is stopped.
- The call returns the PID of the child process.
- The status information allows the parent process

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Macro	Definition
WIFEXITED(stat_val)	Nonzero if the child is terminated normally.
WEXITSTATUS(stat_val)	If WIFEXITEDis nonzero, this returns child exit code.
WIFSIGNALED(stat_val)	Nonzero if the child is terminated on an uncaught signal.
WTERMSIG(stat_val)	If WIFSIGNALEDis nonzero, this returns a signal number.
WIFSTOPPED(stat_val)	Nonzero if the child has stopped.
WSTOPSIG(stat_val)	If WIFSTOPPEDis nonzero, this returns a signal number.

Example

```
#include <stdlib.h>
int main()
{
pid_t pid;
char *message;
int n;
int exit_code;
printf("fork program starting\n");
pid = fork();
switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
message = "This is the child";
n = 5;
exit_code = 37;
break;
default:
message = "This is the parent";
n = 3;
exit_code = 0;
break;
}
for(; n > 0; n--) {
puts(message);
sleep(1);
}
```

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Waiting for the child to finish

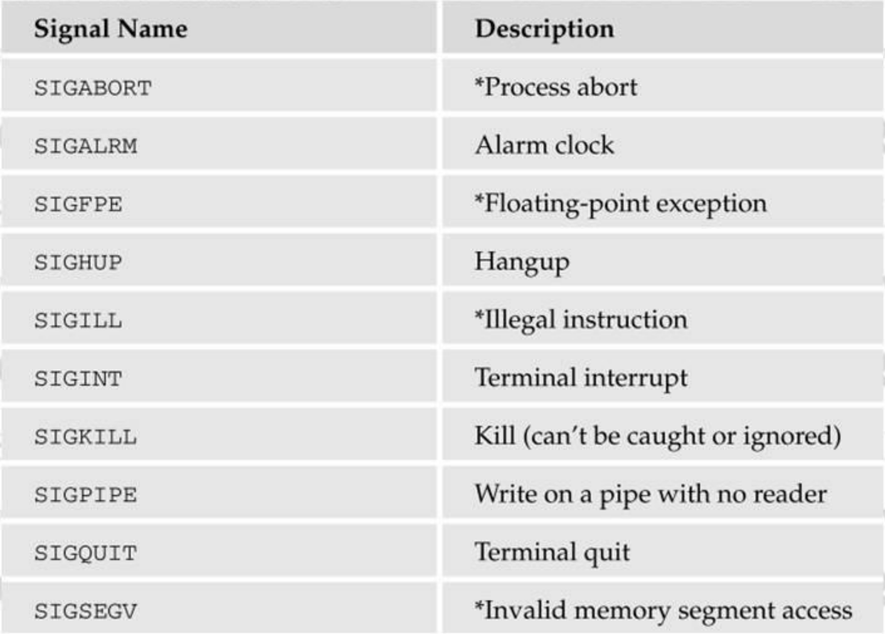
```
if (pid != 0) {
int stat_val;
pid_t child_pid;
child_pid = wait(&stat_val);
printf("Child has finished: PID = %d\n", child_pid);
if(WIFEXITED(stat_val))
printf("Child exited with code %d\n",
   WEXITSTATUS(stat_val));
else
printf("Child terminated abnormally\n");
}
exit(exit_code);
```

Signals

- A signal is an event generated by the UNIX and Linux systems in response to some condition, upon receipt of which a process may in turn take some action.
- Signals are raised by some error conditions, such as memory segment violations, floating-point processor errors, or illegal instructions.
- Signals can be raised, caught and acted upon, or (for some at least) ignored.

www.oranelabs.com ORANE LABS SIIC IIT KANPUR

Using the Header <signal.h>



Signal Name	Description
SIGABORT	*Process abort
SIGALRM	Alarm clock
SIGFPE	*Floating-point exception
SIGHUP	Hangup
SIGILL	*Illegal instruction
SIGINT	Terminal interrupt
SIGKILL	Kill (can't be caught or ignored)
SIGPIPE	Write on a pipe with no reader
SIGQUIT	Terminal quit
SIGSEGV	*Invalid memory segment access

Signal Handling Example

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig)
{
printf("OUCH! - I got signal %d\n", sig);
(void) signal(SIGINT, SIG_DFL);
}
```

```
int main()
{
(void) signal(SIGINT, ouch);
while(1) {
printf(“Hello World!\n”);
sleep(1);
}
}
```

Sending Signals

- A process may send a signal to another process, including itself, by calling kill.
- The call will fail if the program doesn’t have permission to send the signal, often because the target process is owned by another user.
    - **#include <sys/types.h>**
    - **#include <signal.h>**
    - **int kill(pid_t pid, int sig);**

ORANE LABS SIIC IIT KANPUR www.oranelabs.com