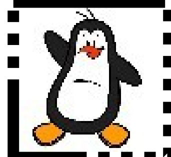


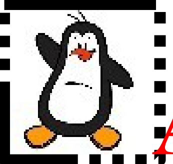
Lecture # 4.2
Process Management
Part - II

Course: Advanced Operating System

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Source Code files available at: **`https://bitbucket.org/arifpucit/spvl-repo/src`**
Lecture Slides available at: **`http://arifbutt.me`**



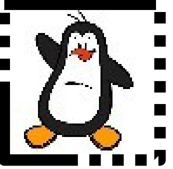
Agenda

- Process Creation using `vfork()`
- Copy-on-Write semantics
- Orphan processes
- Zombie processes
- Monitoring child processes using `wait()`
- Deciphering status argument of wait
  - Using bit-wise operators
  - Using macros
- Limitations of `wait()` system call
- Monitoring child processes using `waitpid()`
- Monitoring child processes using `wait3()`
- Monitoring child processes using `wait4()`

Process Creation using `vfork()`

```
pid_t vfork();
```

- In bad old days a **`fork()`** would require making a complete copy of the parent data space. This was an overhead because, since immediately after a fork the child calls **`exec()`** most of the times. So for greater efficiency BSD introduced `vfork()` system call. Also supported by POSIX.1
- **`vfork()`** is intended to create a new process when the purpose of the new process is to **`exec`** a new program, and it do so without fully copying the parent address space into the child
- Features that make **`vfork()`** more efficient than **`fork()`** are:
  - ✓ No duplication of virtual memory pages is done for child process. Child shares the parent's address space until it either performs **`exec()`** or call **`exit()`**
  - ✓ Execution of parent process is suspended until the child has performed an **`exec()`** or an **`exit()`**



Process Creation using `vfork()`
Proof of concept

`vfork1.c` & `vfork2.c`

Copy-On-Write Semantics

- Today most OSs implement **`fork()`** using copy-on-write pages so the only penalty incurred by **`fork()`** is the time & memory required:
  - To duplicate the parent's page table
  - To create a unique task structure for the child
- Parent forks a child process. Child gets a copy of the parent's page table. Pages which may change are marked "copy-on-write" ; i.e. the pages are not copied for the child rather the child starts sharing the pages and the writable pages are marked "copy-on-write"
- **What happens when the child reads the page.** Just accesses same memory as parent

Parent | Parent's page tbl | Child's page tbl | Child

(Reserved for OS) / Stack / Heap / Uninitialized vars / Initialized vars / Code

They point to the same physical frames!!!

Copy-On-Write Semantics

What happens when the child/parent writes the page?

- If either process (child/parent) tries to modify a shared page, a page fault occurs and the page is copied and inserted in the page table for that particular process
- The other process (who later faults on write) discovers it is the only owner; so no copying takes place

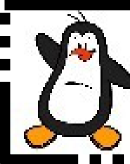
Copy-On-Write Semantics

Page table before and after modification of a shared copy-on-write page

SUSv3 marks **`vfork()`** as obsolete, and has removed the specification of **`vfork()`**

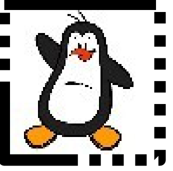
Orphan Processes

- If a parent has terminated before reaping its child, and the child process is still running, then that child is called **orphan**
- In UNIX all orphan processes are adopted by `init` or `systemd` which do the reaping
- Let us see this concept on a Linux terminal

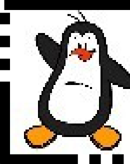


Zombie Processes

- Processes which have terminated but their parent(s) have not collected their `exit` status and has not reaped them are called **zombies** or **defunct**. So a parent must reap its children
- When a process terminates but still is holding system resources like PCB and various tables maintained by OS. It is half-alive & half-dead because it is holding resources like memory but it is never scheduled on the CPU
- Zombies can't be killed by a signal, not even with the silver bullet (SIGKILL). The only way to remove them from the system is to kill their parent, at which time they become orphan and adopted by `init` or `systemd`



Monitoring Child Process



wait() System call

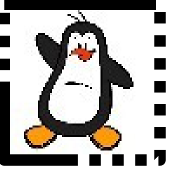
```
pid_t wait(int *status)
```

- The process that calls the `wait()` system call gets blocked till any one of its child terminates
- The child process returns its termination status using the `exit()` call and that integer value is received by the parent inside the `status` argument (used for reaping and cleaning zombies from system). On the shell, we can check this value in the `$?` environment variable
- On success, the `wait()` system call returns PID of the terminated child and in case of error returns a -1
- If a process wants to wait for termination of all its children, then

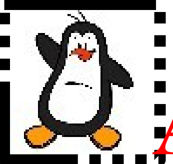
```
while(wait(null) > 0);
```

Two purposes of `wait()` system call:

- Notify parent that a child process finished running
- Tell the parent how a child process finished



Monitoring Child Processes
Proof of concept
`wait1.c`



A Process can Terminate in 4 Ways

- **Normal Termination:** On successful completion of the task, programs call `exit(0)` or `return 0` from `main()` function. In case of failure, programs call `exit()` with a non-zero value. The programmer need to document these error values in the manual page

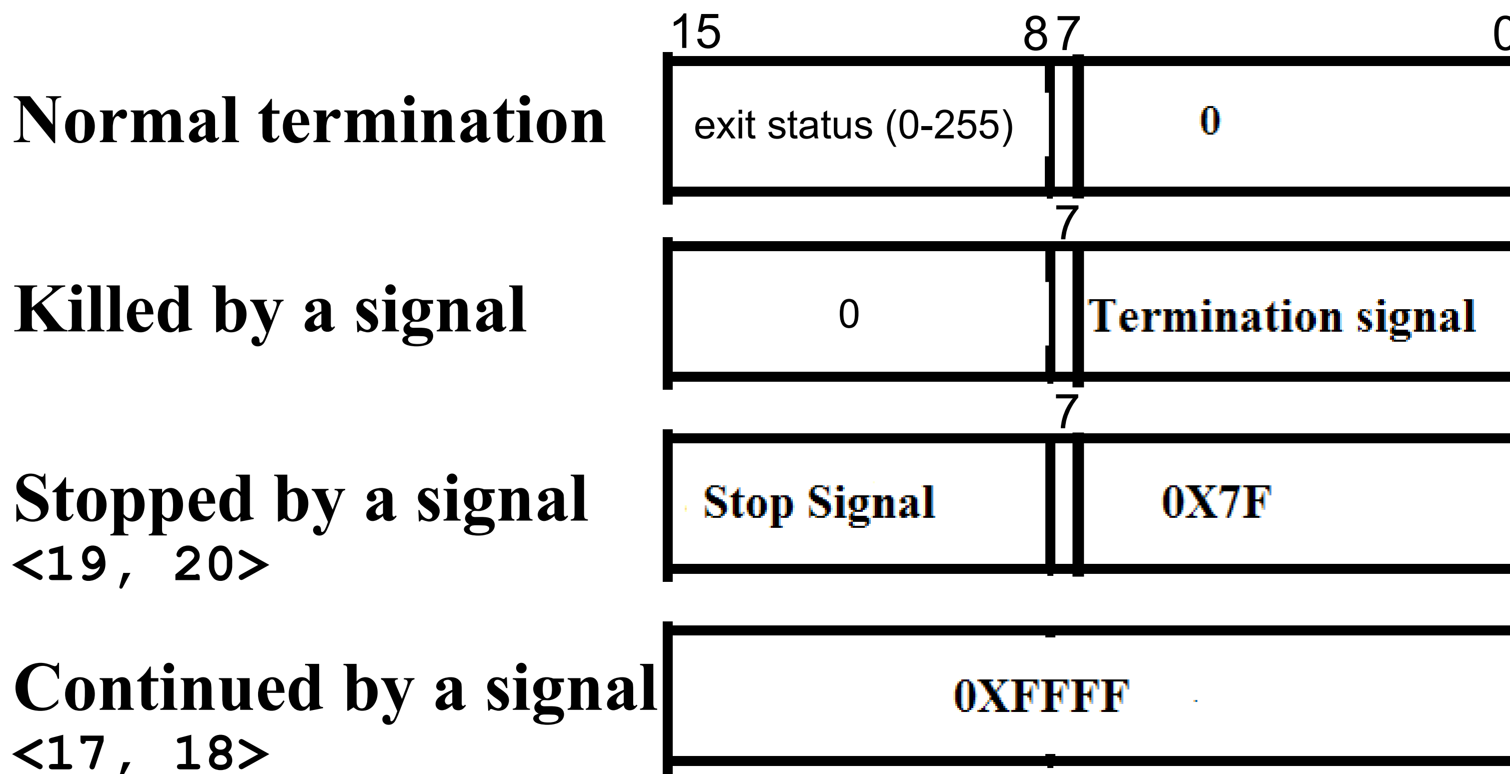
- **Killed by a Signal:** A process might get killed by a signal generated from the keyboard, form an interval timer, the kernel, or from another process

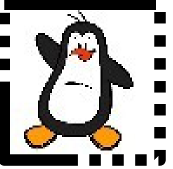
- **Stopped by a Signal:** A process might get `SIGSTOP(19)` or `SIGTSTP(20)` signal and temporary suspend its execution

- **Continued by a Signal:** A process might get `SIGCHILD(17)` or `SIGCONT(18)` signal and continue its execution

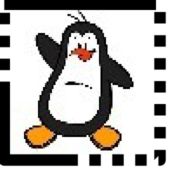
`wait()` Status Argument

All this information is encoded in the `status` argument of the `wait()` system call. A programmer can decipher this information using bit operators or using available macros

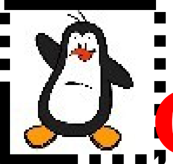




Monitoring Child Processes
Proof of concept
Retrieving the exit status on normal termination
`wait2.c`



Overview of Signals
On the Shell

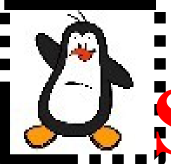


Overview of Signals

- Suppose a program is running in a **while(1)** loop and you press **Ctrl+C** key. The program dies. How does this happens?
  - User presses **Ctrl+C**
  - The **tty** driver receives character, which matches **intr**
  - The **tty** driver calls signal system
  - The signal system sends **SIGINT(2)** to the process
  - Process receives **SIGINT(2)**
  - Process dies
- Actually by pressing **<ctrl+c>**, you ask the kernel to send **SIGINT** to the currently running foreground process. To change the key combination you can use **stty(1)** or **tcsetattr(2)** to replace the current **intr** control character with some other key combination

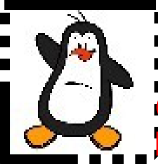
Overview of Signals (cont...)

- Signal is a software interrupt delivered to a process by OS because:
  - The process did something (**SIGFPE (8), SIGSEGV (11), SIGILL (4)**)
  - The user did something (**SIGINT (2), SIGQUIT (3), SIGTSTP (20)**)
  - One process wants to tell another process something (**SIGCHILD (17)**)
- **Signals are usually used by OS to notify processes that some event has occurred, without these processes needing to poll for the event**
- Whenever a process receives a signal, it is interrupted from whatever it is doing and forced to execute a piece of code called signal handler. When the signal handler function returns, the process continues execution as if this interruption has never occurred
- A **signal handler** is a function that gets called when a process receives a signal. Every signal may have a specific handler associated with it. A signal handler is called in *asynchronous mode*. Failing to handle various signals, would likely cause our application to terminate, when it receives such signals



Synchronous & Asynchronous Signals

- Signals may be generated synchronously or asynchronously
- **Synchronous signals** pertains to a specific action in the program and is delivered (unless blocked) during that action. Examples:
  - Most errors generate signals synchronously
  - Explicit request by a process to generate a signal for the same process
- **Asynchronous signals** are generated by the events outside the control of the process that receives them. These signals arrive at unpredictable times during execution. Examples include:
  - External events generate requests asynchronously
  - Explicit request by a process to generate a signal for some other process



Signal Delivery and Handler Execution

Program Code

Start of program

Flow of execution

1

Signal delivery

Instruction n

Instruction n+1

5

`exit()`

2

Kernel calls S.H on behalf of process

Signal Handler

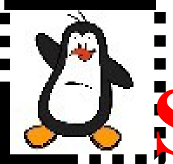
3

Code of signal handler is executed

`return`

4

Program resumes at point of interruption

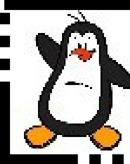


Signal Numbers and Strings

- Every signal has a symbolic name and an integer value associated with it, defined in `/usr/include/asm-generic/signal.h`
- You can use following shell command to list down the signals on your system:

  `$ kill -l`

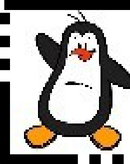
- Linux supports 32 real time signals from SIGRTMIN (32) to SIGRTMAX (63). Unlike standard signals, real time signals have no predefined meanings, are used for application defined purposes. The default action for an un-handled real time signal is to terminate the receiving process. See also **`$ man 7 signal`**



Sending Signals to Processes

A signal can be issued in one of the following ways:

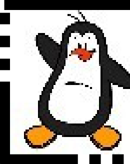
- **Using Key board**
  - `<Ctrl+c>` gives `SIGINT(2)`
  - `<Ctrl+\>` gives `SIGQUIT(3)`
  - `<Ctrl+z>` gives `SIGTSTP(20)`
- **Using Shell command**
  - `kill -<signal> <PID>` OR `kill -<signal> %<jobID>`
  - If no signal name or number is specified then default is to send `SIGTERM(15)` to the process
  - Do visit man pages for `jobs, ps, bg` and `fg` commands
  - `bg` gives `SIGTSTP(20)` while `fg` gives `SIGCONT(18)`
- **Using `kill() or raise()` system call**
- **Implicitly by a program** (division by zero, issuing an invalid addr, termination of a child process)



Signal Disposition

Upon delivery of a signal, a process carries out one of the following default actions, depending on the signal: `[$man 7 signal]`

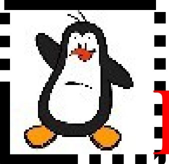
1. The signal is **ignored**; that is, it is discarded by the kernel and has no effect on the process. (The process never even knows that it occurred)
2. The process is **terminated** (killed). This is sometimes referred to as abnormal process termination, as opposed to the normal process termination that occurs when a process terminates using `exit()`
3. A **core dump** file is generated, and the process is terminated. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated
4. The process is **stopped**—execution of the process is suspended (SIGSTOP, SIGTSTP)
5. Execution of the process is **resumed** which was previously stopped (SIGCONT, SIGCHLD)



Signal Disposition (cont...)

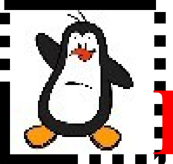
- Each signal has a current disposition which determines how the process behave when the OS delivers it the signal
- If you install no signal handler, the run time environment sets up a set of default signal handlers for your program. Different default actions for signals are:

TERM	Abnormal termination of the program with `_exit( )` i.e, no clean up. However, status is made available to `wait()` & `waitpid()` which indicates abnormal termination by the specified signal
CORE	Abnormal termination with additional implementation dependent actions, such as creation of core file may occur
STOP	Suspend/stop the execution of the process
CONT	Default action is to continue the process if it is currently stopped



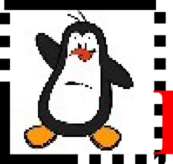
Important Signals (Default Behavior: Term)

Signal	Description
SIGHUP(1)	Informs the process when the user who run the process logs out. When a terminal disconnect (hangup) occurs, this signal is sent to the controlling process of the terminal. A second use of SIGHUP is with daemons. Many daemons are designed to respond to the receipt of SIGHUP by reinitializing themselves and rereading their configuration files.
SIGINT(2)	When the user types the terminal interrupt character (usually <Control+C>, the terminal driver sends this signal to the foreground process group. The default action for this signal is to terminate the process.
SIGKILL(9)	This is the sure kill signal. It can't be blocked, ignored, or caught by a handler, and thus always terminates a process.
SIGPIPE(13)	This signal is generated when a process tries to write to a pipe, a FIFO, or a socket for which there is no corresponding reader process. This normally occurs because the reading process has closed its file descriptor for the IPC channel
SIGALRM(14)	The kernel generates this signal upon the expiration of a real-time timer set by a call to `alarm()` or `setitimer()`
SIGTERM(15)	Used for terminating a process and is the default signal sent by the kill command. Users sometimes explicitly send the SIGKILL signal to a process, however, this is generally a mistake. A well-designed application will have a handler for SIGTERM that causes the application to exit gracefully, cleaning up temporary files and releasing other resources beforehand. Killing a process with SIGKILL bypasses SIGTERM handler.



Important Signals (Default Behavior: Core)

SIGQUIT(3)	When the user types the quit character (Control+\) on the keyboard, this signal is sent to the foreground process group. Using SIGQUIT in this manner is useful with a program that is stuck in an infinite loop or is otherwise not responding. By typing Control-\ and then loading the resulting core dump with the gdb debugger and using the backtrace command to obtain a stack trace, we can find out which part of the program code was executing
SIGILL(4)	This signal is sent to a process if it tries to execute an illegal (i.e., incorrectly formed) machine-language instruction module
SIGFPE(9)	Generate by floating point Arithmetic Exception
SIGSEGV(11)	Generated when a program makes an invalid memory reference. A memory reference may be invalid because the referenced page doesn't exist (e.g., it lies in an unmapped area somewhere between the heap and the stack), the process tried to update a location in read-only memory (e.g., the program text segment or a region of mapped memory marked read-only), or the process tried to access a part of kernel memory while running in user mode. In C, these events often result from dereferencing a pointer containing a bad address. The name of this signal derives from the term segmentation violation



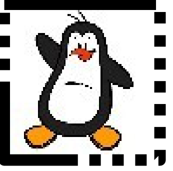
Important Signals (cont...)

Default Behavior: Stop

`SIGSTOP(19)`	This is the sure stop signal. It can't be blocked, ignored, or caught by a handler; thus, it always stops a process
`SIGTSTP(20)`	This is the job-control stop signal, sent to stop the foreground process group when the user types the suspend character (usually <Control+Z>) on the keyboard.. The name of this signal derives from "terminal stop"

Default Behavior: Cont

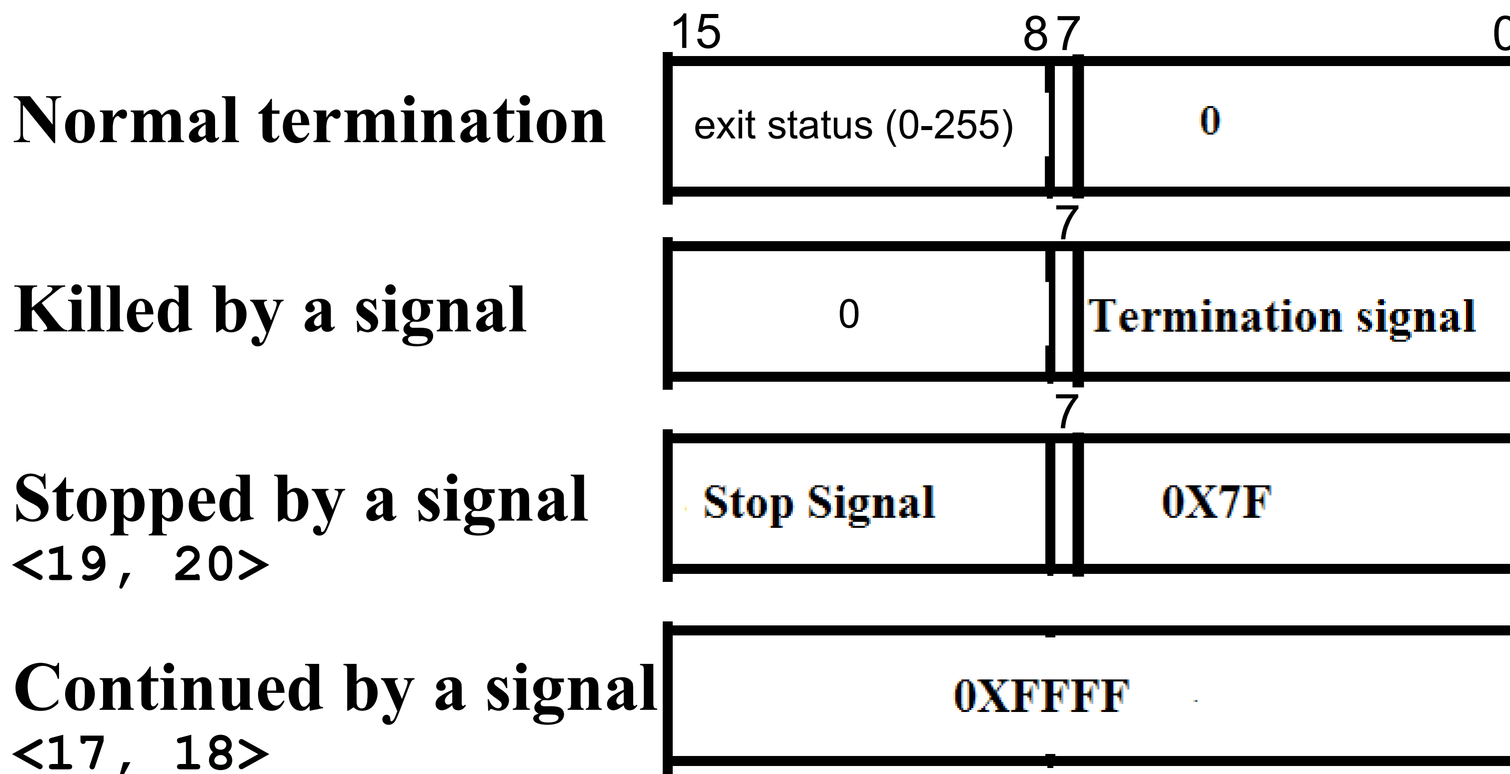
`SIGCHILD(17)`	This signal is sent (by the kernel) to a parent process when one of its children terminates (either by calling `exit()` or as a result of being killed by a signal). It may also be sent to a process when one of its children is stopped or resumed by a signal
`SIGCONT(18)`	When sent to a stopped process, this signal causes the process to resume (i.e., to be rescheduled to run at some later time). When received by a process that is not currently stopped, this signal is ignored by default. A process may catch this signal, so that it carries out some action when it resumes

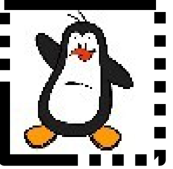


Back to
Monitoring Child Processes
using `wait()` System Call

wait() Status Argument

All this information is encoded in the status argument of the `wait()` system call. A programmer can decipher this information using bit operators or using available macros





Monitoring Child Processes
Proof of concept

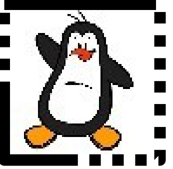
Retrieving the termination signal when killed by a signal (using bit operators)

`wait3.c`

Macros for `wait()` Status Argument

Instead of bit operators, we can use macros to decipher the **`status`** argument of **`wait()`** , defined in **`/usr/include/x86_64-linux-gnu/bits/waitstatus.h`**

Macro	Description
WIFEXITED (status)	• This macro returns true if child process exited normally • **`WEXITSTATUS (status)`** returns exit status of the child process
WIFSIGNALED (status)	• This macro returns true if child process is killed by a signal • **`WTERMSIG(status)`** returns the number of signal that killed the process • **`WCOREDUMP(status)`** returns a non-zero value if the child process created a core dump file
WIFSTOPPED (status)	• This macro returns true if child process is stopped by a signal • **`WSTOPSIG(status)`** returns the number of signal that stopped the process
WIFCONTINUED (status)	• This macro returns true if child process was resumed by **SIGCONT**



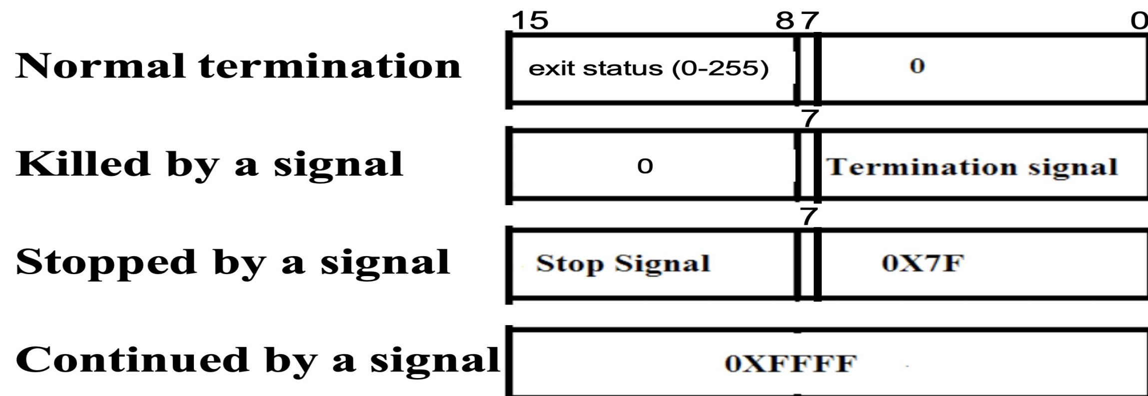
Monitoring Child Processes
Proof of concept

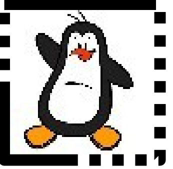
Retrieving the termination signal when killed by a signal (using macros)

`wait4.c`

Limitations of `wait()` System call

- Using `wait()`, it is not possible for parent to retrieve the signal number using which the execution of a child process is stopped (`SIGSTOP(19)`,`SIGTSTP(20)`). Moreover, it is also not possible to be notified when a stopped child is resumed by delivery of a (`SIGCHILD(17)`, `SIGCONT(18)`) signal
- It is not possible to wait for a particular child, parent can only wait for the first child that terminates
- It is not possible to perform a non blocking wait so that if no child has yet terminated, parent get an indication of this fact





Monitoring Child Processes
Proof of concept
Limitation of `wait()` System Call
`wait5.c`

waitpid() System call

With the passage of time, UNIX designers have added a number of variants of the `wait()` system call, like `waitpid()`, `waitid()`, `wait3()`, `wait4()`...

```
pid_t waitpid(pid_t pid, int* status,int options);
```

The **pid** argument enables the selection of the child to be waited for:

- **If pid > 0 :** waits for the child whose PID equals the value of pid
- **If pid == -1:** waits for any child

```
wait(&status) <=> waitpid(-1, &status, 0)
```

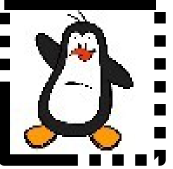
- **If pid == 0 :** waits for any child process whose process Group ID is the same as the calling/parent process
- **If pid < -1:** waits for any child process whose process Group ID equals the absolute value of **pid** argument

waitpid() System call

```
pid_t waitpid(pid_t pid, int* status,int options);
```

The third argument of **waitpid()** call is a bit mask of zero or more of the following flags, defined in `/usr/include/wait.h` file:

WUNTRACED	Also returns information when a child is stopped by a signal
WCONTINUED	Also return information about stopped children that have been resumed by delivery of SIGCONT signal
WNOHANG	Performs polling. If no child specified by pid has yet changed state, then return immediately, instead of blocking



Monitoring Child Processes
Proof of concept

Retrieving the stop signal when stopped by a signal

`waitpid.c`

wait3() & wait4() system calls

```
pid_t wait3(int *status , int options , struct rusage *rusage);
pid_t wait4(pid_t pid,int* status,int options,struct rusage *rusage);
```

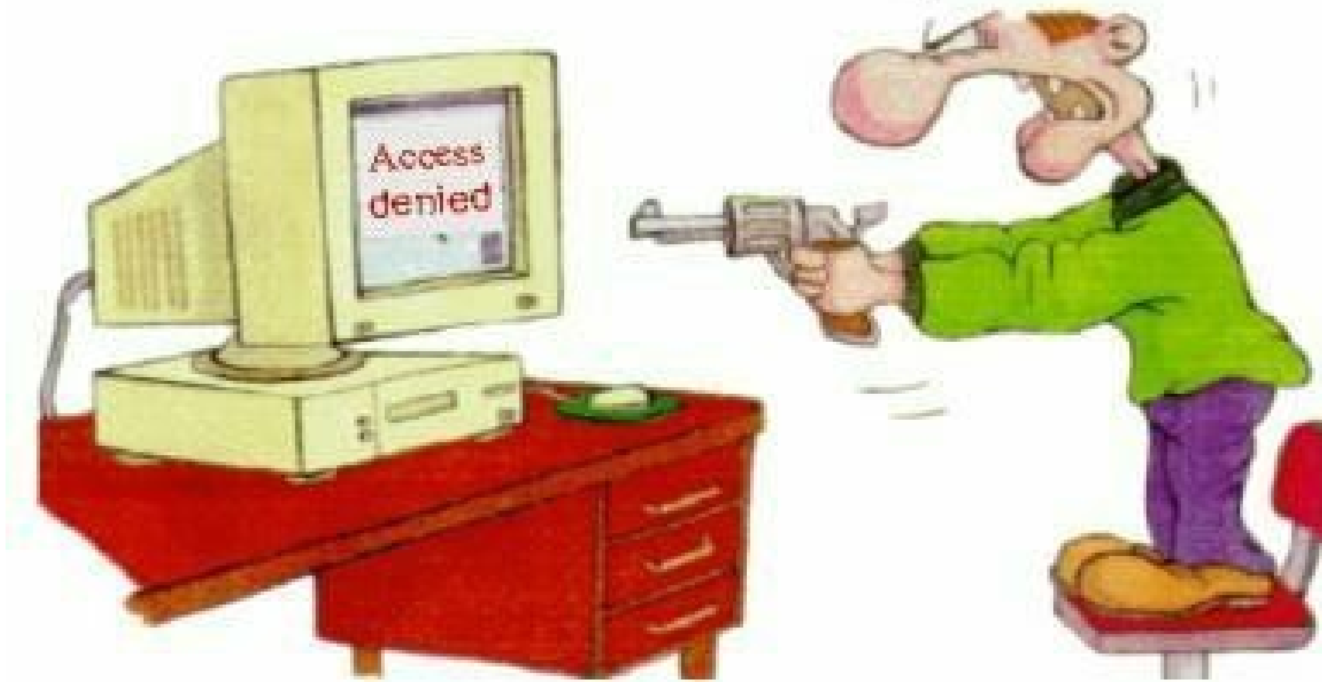
- **wait3() & wait4()** system calls are similar to **waitpid()** but also returns resource usage information about the terminated child in the structure pointed to by **rusage** arguments,which contains information like amount of CPU time used by the process, memory management statistics.
- Waiting for any of the children:

```
wait3(&status,options,null) <=> waitpid (-1,&status,options);
```

- Waiting for a particular child with id **pid**:

```
wait4(pid,&status,options,null)<=>waitpid(pid,&status,options);
```

Things To Do



If you have problems visit me in counseling hours. . . .