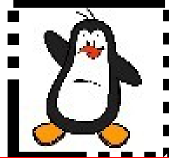


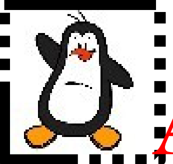
Video Lecture # 18
Process Management
Part - II

Course: SYSTEM PROGRAMMING

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**



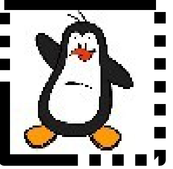
Agenda

- 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

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

Parent

(Reserved for OS)

Stack

Heap

Uninitialized vars

Initialized vars

Code

Parent's page tbl

Child's page tbl

RW

Copy page

Child

(Reserved for OS)

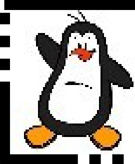
Stack

Heap

Uninitialized vars

Initialized vars

Code



Copy-On-Write Semantics

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

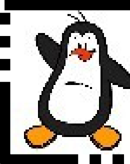
Before modification | **After modification**

Parent page table | Physical page frames | Child page table | PT entry 211 | Frame 1998 | *Unused page frames* | Frame 2038

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

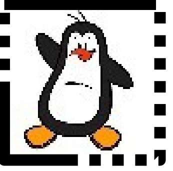
Orphan Processes

- 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

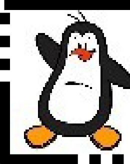


Zombie Processes

- 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

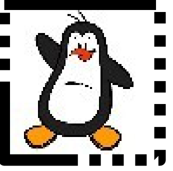
```
pid_t wait(int *status)
```

- 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

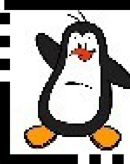
```
while(wait(null) > 0);
```

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`



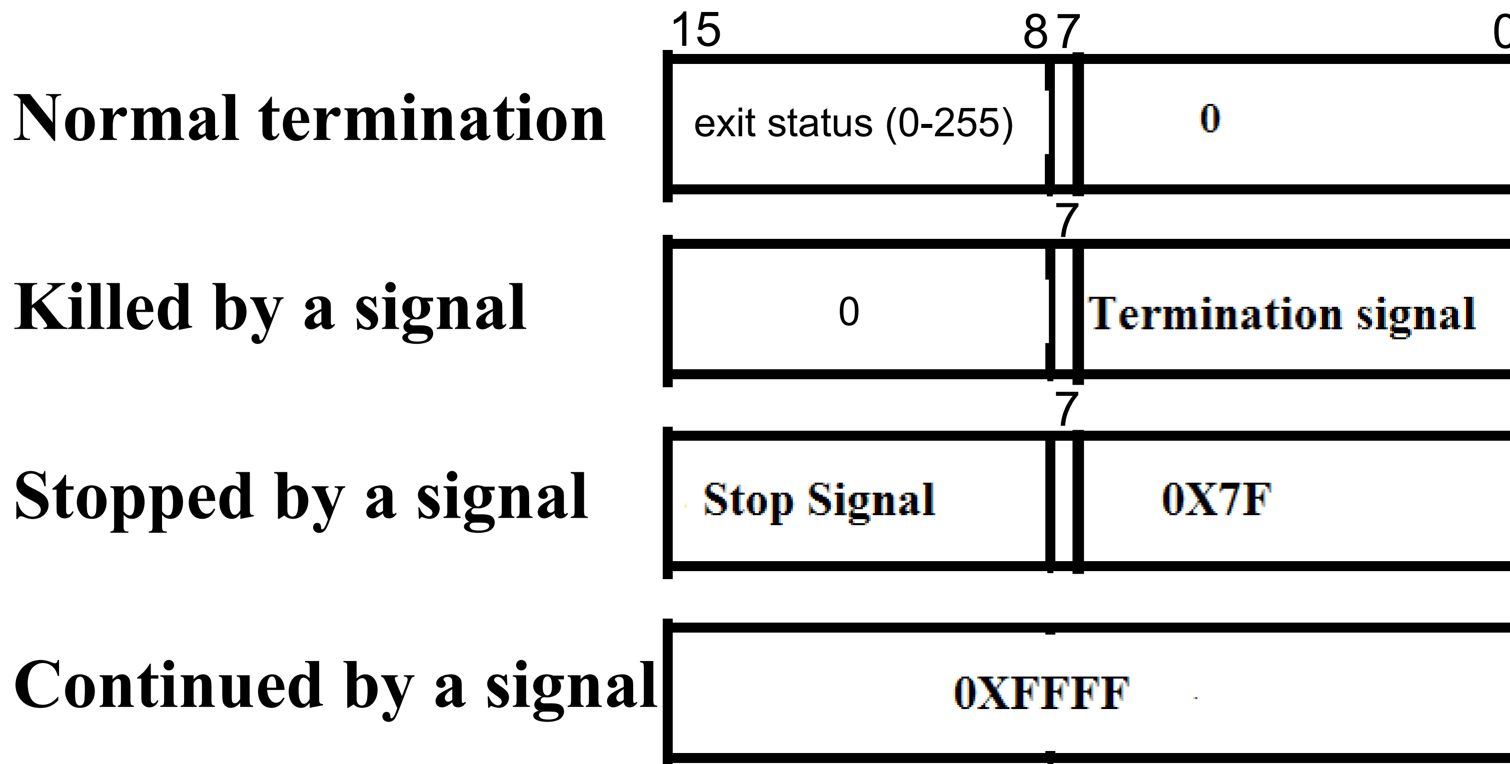
`wait()` Status Argument

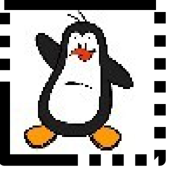
A process can end in four ways:

- **Success /Failure:** 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, from the kernel, or from another process
- **Stopped by a Signal:** A process might get SIGTSTP signal and temporary suspend its execution
- **Continued by a Signal:** A process might get SIGCONT 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



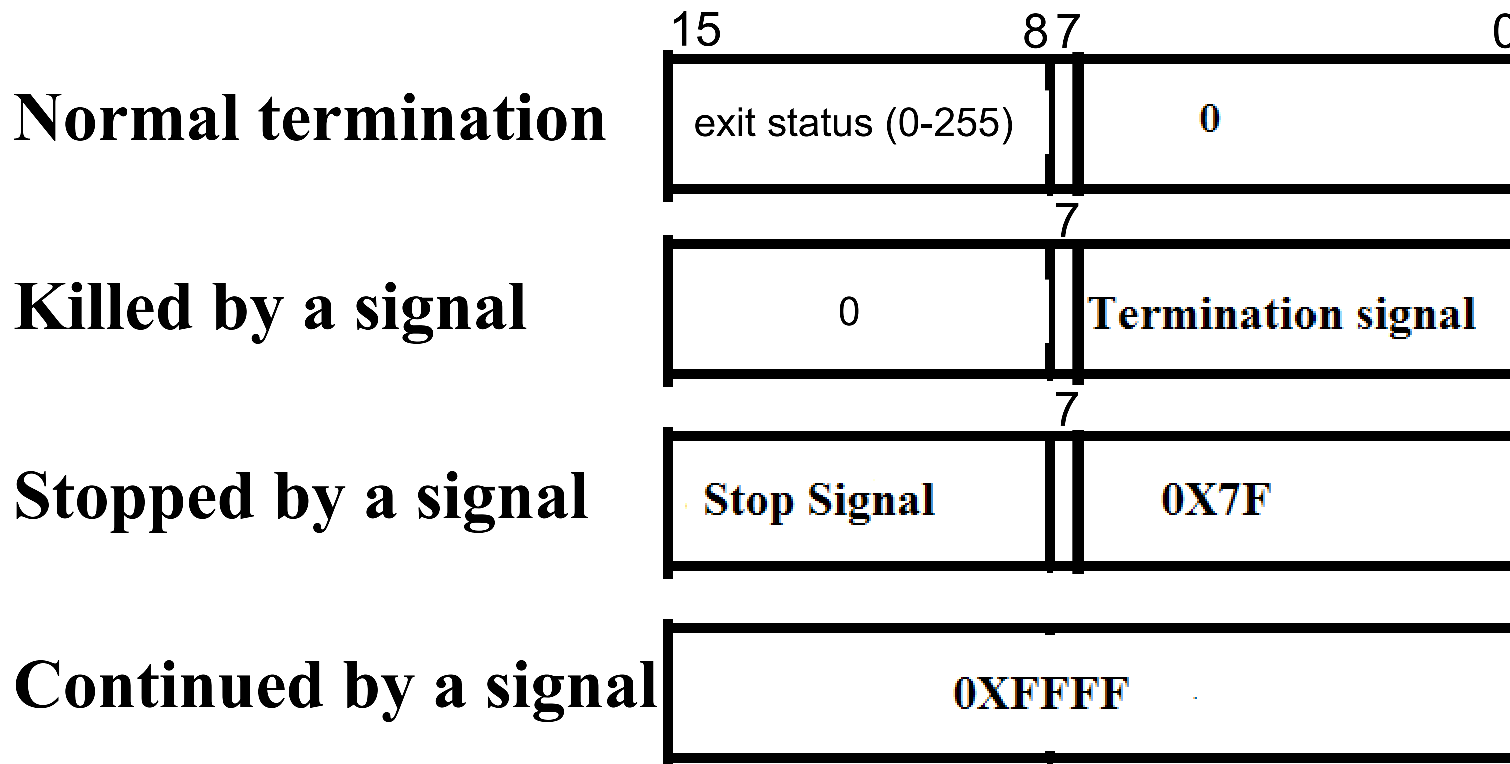


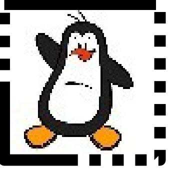
Monitoring Child Processes Proof of concept

`wait2.c`

`wait()` Status Argument

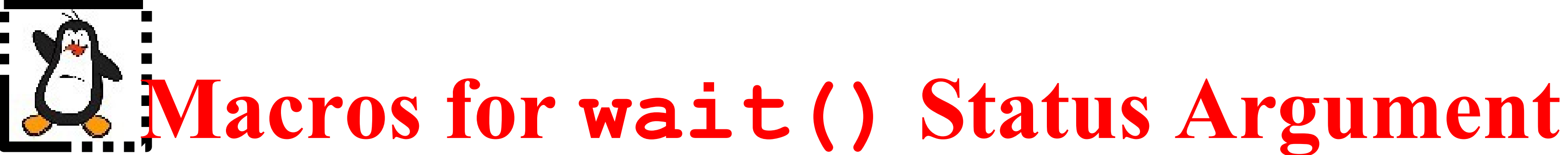
All this information is encoded in the status argument of the `wait()` system call. A programmer can decipher this information using bit operators or using available macros





Monitoring Child Processes
Proof of concept

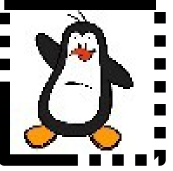
`wait3.c`



Macros for `wait()` Status Argument

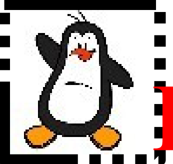
Instead of bit operators, we can use macros to decipher the **`status`** argument of **`wait()`** , defined in **`/usr/include/x86_64-linux-gnu/bits/waitstatus.h`**

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**



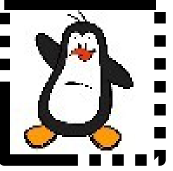
Monitoring Child Processes
Proof of concept

`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

`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)
```

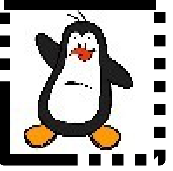
- **If pid == 0 :** waits for any child process whose process Group ID is the same as the calling/parent process
- **If pid < -1:** waits for any child process whose process Group ID equals the absolute value of **pid** argument

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
`waitpid.c`

wait3() & wait4() system calls

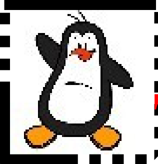
```
pid_t wait3(int *status , int options , struct rusage *rusage);
pid_t wait4(pid_t pid,int* status,int options,struct rusage *rusage);
```

- **`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

O.k., and now you'll do exactly what I'm telling you !

Access denied

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