

# Advanced Unix Programming

## Module 05

Raju Alluri

askraju @ spurthi.com

# Advanced Unix Programming: Module 5

- Processes & Signals
  - Process Structure
  - Process Creation etc.
  - Process return status
  - Signals & Signal Handlers

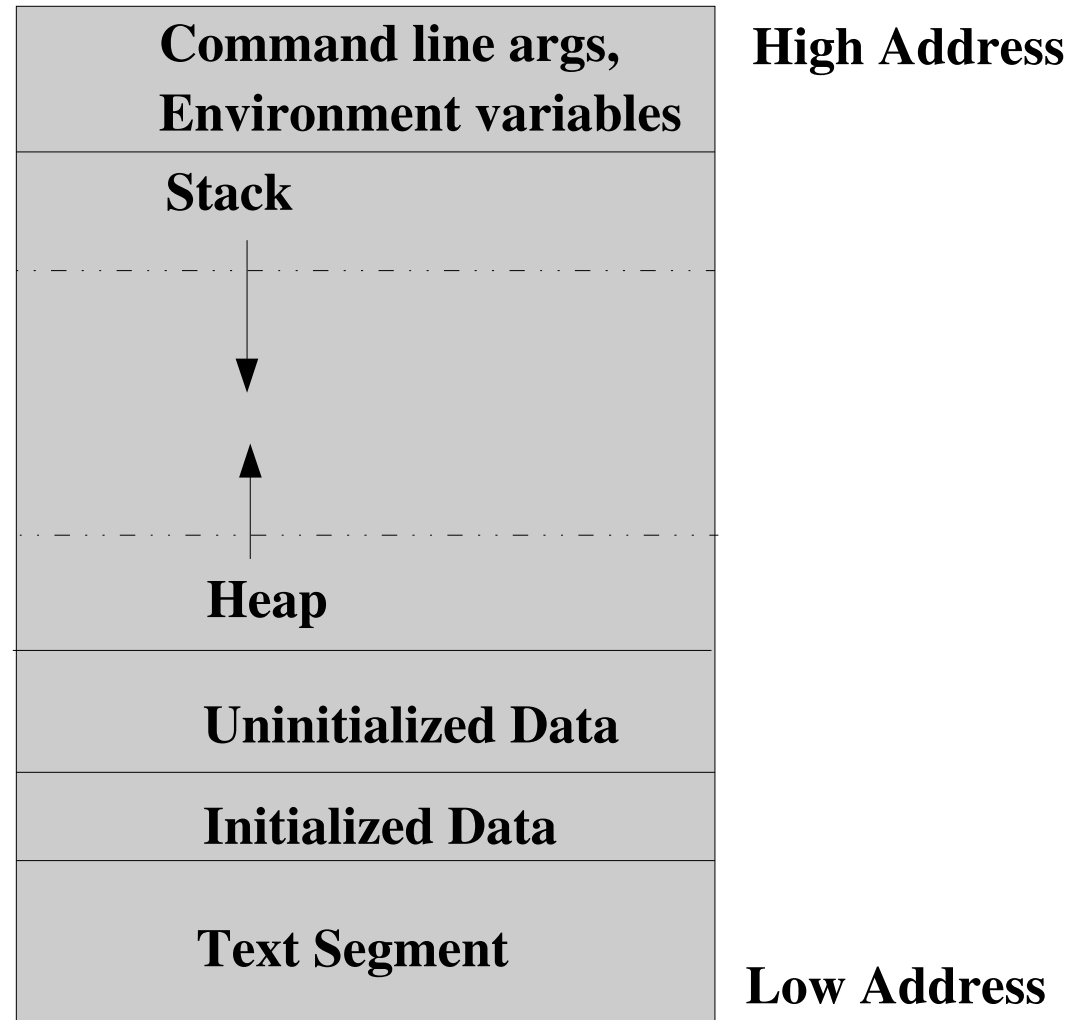
# Processes

- Whenever a program is run, it runs as a process
  - Each process is identified by a unique ID called PID
  - Processes can create other processes when needed
  - Each process gets some memory to store the program instructions, data etc.
  - Each process, upon completion, returns a status to the OS.

# Processes

- Whenever a program is run, it runs as a process
  - Each process is identified by a unique ID called PID
  - Processes can create other processes when needed
  - Each process gets some memory to store the program instructions, data etc.
  - Each process, upon completion, returns a status to the OS.

# Structure of a Process



# Process Identifiers

- Has
  - Process ID (Unique)
  - Parent Process ID
  - Real User ID, Effective User ID
  - Real Group ID, Effective Group ID

# System Calls for Process Identifiers

- Process Identifiers

```
pid_t  getpid();  
pid_t  getppid();  
uid_t  getuid();  
uid_t  geteuid();  
gid_t  getgid();  
gid_t  getegid();
```

# Process Creation

- The `fork()` system call spawns a new process

```
pid_t fork();
```

- In the parent, the `fork()` system call returns the PID of the child.
- In the child, the `fork()` system call returns zero
- Both processes continue with the code after `fork()` system call

# Typical code using fork()

- Code using fork() would look like:

```
pid_t pid;

if((pid = fork() < 0)) {
    /* error condition here */
}
else if (pid ==0) {
    /* in child process */
}
else {
    /* in parent process */
}
```

# Process Termination

- There are several ways a process can terminate
  - Normal termination
    - Do a return from main
      - Same as `exit()`
    - Call `exit()` function
      - Calls exit handlers and calls `_exit()`
    - Call `_exit()` function
      - Does system level cleanup.
  - Abnormal termination
    - Call `abort`
    - Receive certain signals (discussed later)

# Parent/Child Termination Sequence

- Parent terminates before child (orphaned child)
  - Child is reparented to a process called init (pid 1)
- Child terminates before the parent
  - Parent might need the exit status of the child
  - If child is not running, there is no way for parent to get exit status
    - The child's memory and open files can be discarded
    - The process entry will still be maintained
    - The child's process entry will still be maintained (PID, status, CPU time taken etc.)
    - The child is said to be in “Zombie” status

# Waiting for a child to terminate

- The `wait()` system call waits for a child process to terminate

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

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

- The `wait()` system call blocks until
  - A child terminates or a signal to terminate (discussed later)
- The `waitpid()` call has an option to continue without waiting for a child to continue.
  - `Pid` is `-1` means wait for any child, `pid` is `>0` means wait for a specific child, `pid` is `0` means wait for child whose `PGID` is same, `pid` is `<-1` means wait for child whose `PGID` is `| pid |`

# Other variants for wait

- The `wait3()` and `wait4()` are very useful too.

```
pid_t wait3(int *status, int options,  
            struct rusage *ru)
```

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

- The pointer to `rusage` struct is filled with resource usage of the child.

# How to execute other programs

- To start a new program, the normal sequence is

- A `fork()` call is done

- The parent continues

- The child does an `exec()` call to overlay a new program

```
pid_t wait3(int *status, int options,  
            struct rusage *ru)
```

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

- The pointer to `rusage` struct is filled with resource usage of the child.

# The exec() functions

- 6 variants

```
int execl(const char *path, const char
    *arg0, ... /* (char *)0 */);
int execv(const char *path, char *const
    argv[]);
int execlp(const char *path, const char *arg0,
    ... /* (char *)0, char *const envp[] */);
int execve(const char *path, char *const
    argv[], char *const envp[]);
int execlp(const char *file, const char *arg0,
    ... /* (char *)0 */);
int execvp(const char *file, char *const
    argv[]);
```

# Effect of an exec()

- After an exec()
  - No new process will be created
  - The memory of the process is completely replaced with the new program
  - Execution of the new program starts from main()

# The vfork()

- The vfork() system call has same semantics as fork(), except
  - The child normally calls exec() immediately after the fork()
  - Until then, the child will continue to operate on the address space of the parent
  - Upon exec, a new process layout will be created with the new program
  - Efficient execution, but can lead to errors if not used properly

# Chaning user & groupd IDs

- The `setuid()`, `setgid()` calls

```
int setuid(uid_t uid);
```

```
int setgid(gid_t gid);
```

- For a process with super user previleges, the real and effective user/group id is set to uid/gid
- For a process that has real user id same as uid/gid, the effective user/group ID is set of uid/gid
- Otherwise it is an error condition

# The system() function

- Execute a program directly and get the return status

```
int system(const char *string);
```

- Execute the command in a shell and send the return value

# User Identification

- Get the user name

```
char *getlogin();
```

- Get the login name of the current user

# Process Times

- Get the user name

```
clock_t *times(struct tms *buf);
```

- Get the elapsed clock time and fill the buffer

- tms\_utime
- tms\_stime
- tms\_cutime
- tms\_cstime

# Signals

- Signals are software interrupts
- They allow handling of asynchronous events
- Each signal has
  - A name (and an associated positive integer constant)
    - e.g: SIGKILL, SIGABRT, SIGSEGV
    - See signal.h
  - A default action

# What we can do with a signal

- When a signal occurs, we can do one of the following activity (called action)
  - Let the default action happen
  - Ignore the signal
    - Except for SIGKILL and SIGSTOP
  - Catch the signal
    - Tell kernel which function to call when the signal occurs
    - Each signal can be associated with a specific signal handler

# Default actions

- The default actions for signals are:
  - Ignore
  - Stop process
  - Terminate
  - Terminate with core

# Registering a signal handler

- Use the `signal()` call to register the signal handler

```
void (*signal(int sig, void (*func)(int)))(int);
```

- Register `func` as the handler for signal `sig` and return the previous handler for the signal
- Predefined actions
  - `SIG_ERR`
    - Returned by `signal()` in case of error
  - `SIG_DFL`
  - `SIG_IGN`

# Registering a signal handler

- Use the `signal()` call to register the signal handler

```
void (*signal(int sig, void (*func(int)))(int);
```

- Register `func` as the handler for signal `sig` and return the previous handler for the signal
- Predefined actions
  - `SIG_ERR`
    - Returned by `signal()` in case of error
  - `SIG_DFL`
  - `SIG_IGN`

# Effect of a fork()

- When parent process does a fork(), the child inherits all the signal handlers

# Unreliable Signals

- Earlier versions of unix used to reset the signal handler to default action when a signal happens
  - As a result, the first line in a signal handler is to typically re-register the signal handler
    - What happens if the signal occurs twice before the re-registration?
- You want to wait until a signal has occurred. How do you do that?
  - What if the signal happens after the check (for the signal occurrence) and before the call that waits for a signal?

# Semantics of Reliable Signals

- Signals have the following states
  - Generated: By the time they occur
  - Delivered: By the time the signal handler executes
  - Pending: The intermediate time between generated and delivered state
- Using `sigpending()`, the process can determine the list of pending signals

# Sending Singnals to a Process

- The `kill()` call sends a given signal

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

- `Pid > 0`: Send signal to process with PID as `pid`
  - `Pid == 0`: Send signal to all processes with PGID same as that of sender (and sender has permissions for the process)
  - `Pid < 0`: Send signal to all processes with PGID as `|pid|`
- The `raise()` call sends a given signal to the same process

```
int raise(int signo);
```

-

# Sending Singnals to a Process, #2

- The alarm() call sends a SIGALRM signal after the timer expires

```
unsigned int alarm(unsigned int seconds);
```

- If seconds is 0, cancel the previous timer and return when the previous timer would have sent a signal
- If seconds is non-zero, then send a signal after “seconds”.
  - Return the number of seconds after which a previously registered alarm would be sent.

# Waiting for Signals

- The `pause()` call waits until a signal happens

```
int pause(void);
```

# The abort() function

- Calling the abort() generates SIGABRT to the process

```
void abort(void);
```

- To be POSIX compliant, the signal handler for SIGABRT should not return
  - So call exit(), \_exit(), longjmp(), siglongjmp() etc.
- Do all the cleanup needed within the signal handler
- Posix compliant implementations override the blocking or ignoring of the signal

# The system() function

- Calling the system() needs special handling of some signals
  - Ignore SIGINT and SIGQUIT
  - Block SIGCHLD
- Note that the system() call returns the return status of the shell that executes the command

# The sleep() function

- Suspend the process execution for a specified time
  - Unsigned int sleep(unsigned int seconds);
  - Suspend the calling process until
    - The amount of clock time elapsed is “seconds”
    - A signal is caught by the process and the signal handler is executed