UNIX System Programming

Process Management

Module 2

Agenda

1. Process Management
2. Program Execution
3. Proc File System
4. Process Management Commands
4. Process Layout & Exploring Executables

Objectives

At the end of this module you will be able to

- Understand the role of process management
- Use process related system calls
- Know the organization of proc file system
- Explore proc file system and per process entries in /proc
- Understand the memory layout of a process
- Use commands to explore object and executable files

Process Management

Process Management Subsystem

- Process is a program in execution
- Process management subsystem of Unix OS is responsible for
  - Process creation & termination
  - Process scheduling
  - Process accounting

- An executable program, which resides on disk, once executed results in having a process corresponding to the executable.

- A process has a single flow of execution of instructions, and has a context.

Process related system calls & library functions

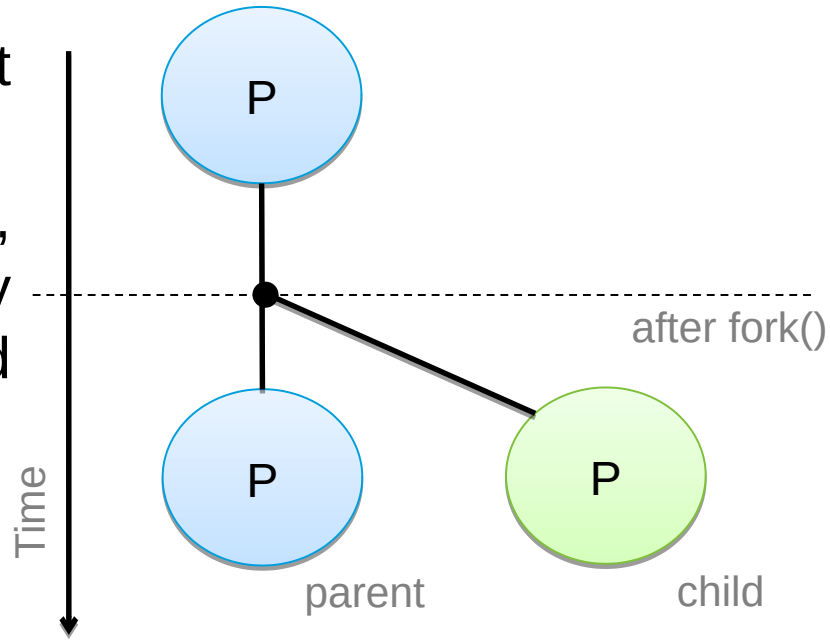
- Process related system calls that will be discussed are:-
  - fork create a process
  - getpid get process id
  - getppid get parent process id
  - getuid get user id
  - getgid get group id
  - wait wait for child process to terminate
  - waitpid wait for a specific or any child process to terminate

- exec family replace current process image with new image
  These are C Library functions

Process creation

- The only way to create a process in UNIX is to fork a process.
- The system call fork(), creates a copy of the calling process.
- After fork(), there will be two processes in execution, one the original process, and the other being a copy of the first.
- The process calling fork() is called *parent process* and the created process is called the *child process*.

- Virtual address space of the parent gets replicated for the child.
- If a process is successfully created, there is no guarantee that the newly created process (i.e., child) would get immediately chance to run.



Process creation (Contd.).

```
#include <unistd.h>
pid_t fork(void);
```

- If **fork()** is successful, it returns the PID of the child process to the calling process and 0 to the child process.
- If fork() fails, then return value is -1. Hence to check for the value of pid in case of error too.

Process creation (Contd.).

```
int pid;
pid = fork();
switch (pid)
{
   case -1:
         // error encountered
         break;
   case 0:
         // in child
         // implement logic for child process
         break;
     default:
         // in parent
         // implement logic for parent process
         break;
}
```

Process Attributes

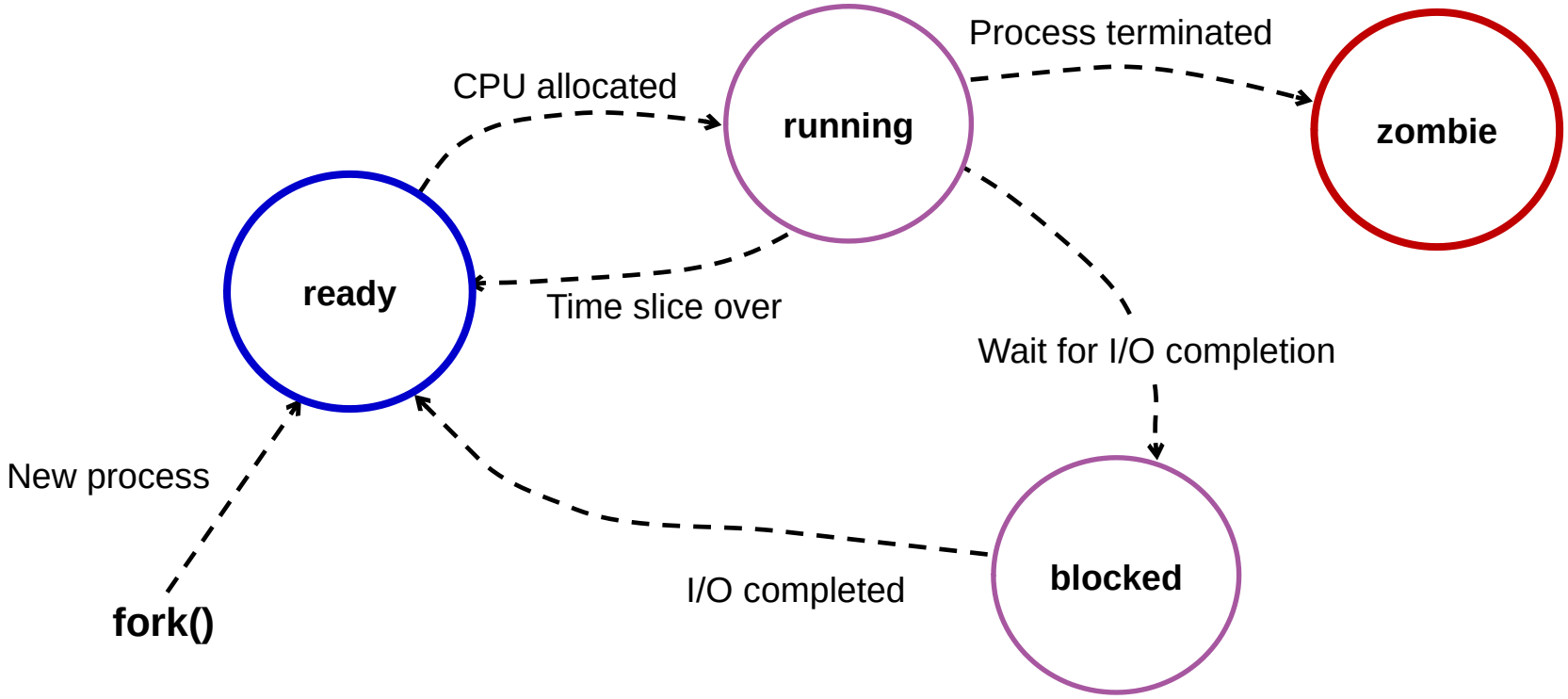
- Every process has a variety of attributes, some of them unique and some inherited from parent process.

- Unique attributes of a process
  - Process ID (PID)
  - Unique task structure

- Inherited attributes of a process
  - File descriptors
  - File size limit
  - Resource limits
  - Nice value

Process States

- A process from the time it got created till its termination, would make transition into several states.



Process Termination

- A process terminates execution, when any one of the following actions happen on a process.
  - Process executes **`exit()`** **or** **`_exit()`**
  - **`exit()`** is library function, where as **`_exit()`** is system call.

- **`exit()`** invokes all registered cleanup() routines and exits the process gracefully.
- **`_exit()`** does not invoke cleanup() routines and hence terminates the process abruptly.

getpid(), getppid()

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

pid_t getpid(void);
```

- Returns process id (pid) of the calling process

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

- Returns parent process id (ppid) of the calling process

Waiting for child process termination

- In some cases parent process wants to wait for the child to terminate
  - Example – shell waits for the command in interactive mode to terminate

- In such cases parent process has to call wait() or waitpid()
- wait() puts the calling process suspended, until one of its child processes terminates
  - If a child has already exited, this function returns immediately.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

`status` – to capture the exit status of the terminated process
If the exit status is not required, NULL can be passed as argument.

Returns the pid of the terminated child process/

waitpid()

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

- Similar to **`wait()`**, but provides the flexibility to wait for a specific child process to terminate.
  - If pid = -1, wait for any child to terminate. This is equivalent to **`wait()`**
  - If pid > 0, wait for specific child to terminate

  Additionally, if **`option`** is **`WNOHANG`**, then returns immediately.

  Return value will be the **`pid`** of the child that exited.
  Return value will be the 0, if **`WNOHANG`** specified for options and no child had exited.

Zombie process

- A process which creates a child, may or may not be interested to wait for the child's termination
- This causes the child process to be put into zombie or defunct state on termination.
  - On termination of a process, all resources that were acquired by the process get released.
  - However, process exit status is maintained, till the parent process asks for it (using wait()/waitpid()) or the parent terminates, whichever is earlier.
- There are situations where in a process is not interested in the exit status of its child processes.
- To avoid zombie processes, it requires handling of SIGCHLD signal.

Example – process.c

```
void  do_parent()
{
  printf("in parent
      ( pid = %d)\n", getpid());
  sleep(10);
  printf("parent terminated\n");
}

void do_child()
{
  printf("in child
      ( pid = %d)\n", getpid());
  sleep(3);
  printf("child terminated\n");
  exit(0);
}
```

```
int main()
{
    int pid;
    pid = fork();
    switch (pid) {
        case -1:
            printf("failed to
             fork a process\n");
            break;
        case 0:
            do_child();
            break;
        default:
            do_parent();
            break;
    }
    return 0;
}
```

Program Execution

Program execution – execve()

- Once a process is created, child process executes the same logic as in the parent process. This is because the “text segment” is shared.
- execve() system call lets a new program be loaded into the process image and be run

```
#include <unistd.h>
int execve(const char *file_path_name,
        char *const argv[], char *const envp[]);
```

- The system call takes the name of the executable, command arguments and environment string (last two parameters are optional)
- If successful, **`execve()`** never returns, because it replaces the program that called it; otherwise returns -1

exec() Library Functions

- C Library provides a variety of functions to perform exec()
- They internally call execve()
- They differ in the way program name, arguments and environment strings are provided to the functions.

  - ```
    int execle(const char *pathname, const char *arg, ...,
                (char *) NULL, char *const envp[] );
    ```
  - ```
    int execlp(const char *filename, const char *arg, …,
                (char *) NULL);
    ```
  - ```
    int execvp(const char *filename, char *const argv[]);
    ```
  - ```
    int execv(const char *pathname, char *const argv[]);
    ```
  - ```
    int execl(const char *pathname, const char *arg, ...,
                (char *) NULL);
    ```
    ```

How to use exec() library functions

```
int   ret, pid;
pid = fork();
switch (pid)  {
  case 0:  // in child
    // execute target program
    ret = any exec function with suitable args
     if (ret == -1)  // exec failed
      _exit(EXIT_FAILURE);
 case -1:  // notify fork failed
 default: // in parent
      // parent specific logic
}
```

Using execvp()

- Uses the PATH environment variable

```
int ret;
char* arglist[ ] = { "echo", "hello", "world",
                     NULL };
      /* list has to be terminate with NULL */
ret = execvp(arglist[0], arglist);
```

Proc File System

About proc file system

- Proc file system is a virtual file system
- Contains information about the system environment such as h/w, processes, file systems, device drivers etc.
- /proc is the mount point for the proc file system
- the file system resides in memory; never gets saved on disk.

Directories in /proc

```
1/
13/
1896/
1897/
1898/
1899/
1900/
2/
2266/
2270/
2298/
2318/
2330/
2388/
2455/
2471/
2500/
2509/
2520/
2531/
2555/
2565/
2577/
2578/
2579/
2580/
2581/
2582/
25821/
25823/
25824/
2583/
26115/
26277/
2637/
2638/
2653/
3/
4/
5/
6/
68/
7/
8/
9/
```

Process specific directories
Directory exists as long as the corresponding process exists

About proc file system

System specific directories/files

File contents keep changing dynamically, reflecting the system at that instance

```
apm
bus/
cmdline
cpuinfo
crypto
devices
dma
dri/
driver/
execdomains
fb
filesystems
fs/
ide/
interrupts
iomem
Ioports
irq/
kcore
kmsg
ksyms
loadavg
locks
mdstat
meminfo
misc
modules
mounts@
mtrr
net/
partitions
pci
scsi/
self@
slabinfo
stat
swaps
sys/
sysrq-trigger
sysvipc/
tty/
uptime
version
```

Directory entries of a process in /proc

```
$ ls -l  25824
total 0
-r--r--r--  1 wipro18  wipro18  0 May 21 10:46 cmdline
lrwxrwxrwx  1 wipro18  wipro18  0 May 21 10:46 cwd ->
/proc/25824
-r--------  1 wipro18  wipro18  0 May 21 10:46 environ
lrwxrwxrwx  1 wipro18  wipro18  0 May 21 10:46 exe -> /bin/bash
dr-x------  2 wipro18  wipro18  0 May 21 10:46 fd
-r--------  1 wipro18  wipro18  0 May 21 10:46 maps
-rw-------  1 wipro18  wipro18  0 May 21 10:46 mem
-r--r--r--  1 wipro18  wipro18  0 May 21 10:46 mounts
lrwxrwxrwx  1 wipro18  wipro18  0 May 21 10:46 root -> /
-r--r--r--  1 wipro18  wipro18  0 May 21 10:46 stat
-r--r--r--  1 wipro18  wipro18  0 May 21 10:46 statm
-r--r--r--  1 wipro18  wipro18  0 May 21 10:46 status
$
```

Directory entries of a process in /proc (Contd.).

```
$ ls -l 25824/fd
total 0
lrwx------  1 wipro18  wipro18  4 May 21 11:02 0 -> /dev/pts/0
lrwx------  1 wipro18  wipro18  64 May 21 11:02 1 -> /dev/pts/0
lrwx------  1 wipro18  wipro18  64 May 21 11:02 2 -> /dev/pts/0
lrwx------  1 wipro18  wipro18  64 May 21 11:02 255 -> /dev/pts/
0
$
```

Process Management Commands

Process Management Commands

- Process management commands that will be discussed are :-

  `ps` – prints a report of processes
  `top` – displays the topmost CPU intensive processes
  `fg` – resume specific job in foreground
  `jobs` – list active jobs
  `nice` – change scheduling priority of a process

- Note: **`bg`**, **`fg`** and **`jobs`** are built-in commands of bash shell.

ps command

```
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:04 init
    2 ?        SW     0:00 [keventd]
    3 ?        SW     0:00 [kapmd]
    4 ?        SWN    0:00 [ksoftirqd/0]
    7 ?        SW     0:00 [bdflush]
    5 ?        SW     0:00 [kswapd]
    6 ?        SW     0:00 [kscand]
    8 ?        SW     0:00 [kupdated]
  ...
21380 pts/1    S      0:00 -bash
21460 ?        S      0:00 sshd: wipro18 [priv]
21462 ?        S      0:00 sshd: wipro18@pts/0
21463 pts/0    S      0:00 –bash
29258 pts/1    R      0:00 ps ax
$
```

Process state code

R runnable (on run queue)
S sleeping
T traced or stopped
Z zombie process
N low-priority task
W no resident pages

ps command (Contd.).

```
$ ps  -u wipro18
  PID TTY          TIME CMD
21379 ?        00:00:00 sshd
21380 pts/1    00:00:00 bash
21462 ?        00:00:00 sshd
21463 pts/0    00:00:00 bash
29557 pts/1    00:00:00 ps
$
```

 get process listing specific to a user

```
$ ps -u wipro18  -O pid,ppid,state,cmd
 PID  PPID  S CMD
21379 21377 S sshd: wipro18@pts/1
21380 21379 S -bash
21462 21460 S sshd: wipro18@pts/0
21463 21462 S -bash
30253 21463 R ps -u wipro18 -o pid,ppid,state,cmd
$
```

use –o option to choose output columns

top command

- displays the topmost CPU intensive processes

```
wipro18@localhost:~
 17:22:46  up 4 days, 23:08,  3 users,  load average: 0.00, 0.01, 0.00
54 processes: 47 sleeping, 4 running, 0 zombie, 3 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
           total    0.0%    0.0%    0.0%   0.2%     0.0%    0.0%   99.8%
Mem:  2050480k av,  903840k used, 1146640k free,       0k shrd,  187856k buff
       419928k active,             208840k inactive
Swap: 4096532k av,       0k used, 4096532k free                  476604k cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND
    1 root      15   0   508  508   448 S     0.0  0.0   0:04   0 init
    2 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 keventd
    3 root      15   0     0    0     0 RW    0.0  0.0   0:00   0 kapmd
    4 root      34  19     0    0     0 SWN   0.0  0.0   0:00   0 ksoftirqd/0
    7 root      25   0     0    0     0 SW    0.0  0.0   0:00   0 bdflush
    5 root      15   0     0    0     0 RW    0.0  0.0   0:00   0 kswapd
    6 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 kscand
    8 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 kupdated
    9 root      25   0     0    0     0 SW    0.0  0.0   0:00   0 mdrecoveryd
   13 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 kjournald
   68 root      25   0     0    0     0 SW    0.0  0.0   0:00   0 khubd
 1896 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 kjournald
 1897 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 kjournald
 1898 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 kjournald
 1899 root      15   0     0    0     0 SW    0.0  0.0   0:00   0 kjournald
 1900 root      15   0     0    0     0 SW    0.0  0.0   2:15   0 kjournald
 2266 root      15   0   592  592   512 S     0.0  0.0   0:22   0 syslogd
 2270 root      15   0   460  460   396 S     0.0  0.0   0:00   0 klogd
 2298 rpc       21   0   564  564   488 S     0.0  0.0   0:00   0 portmap
```

Job control

- Commands associated with job control are
  - jobs
  - fg
  - bg
  - nice
  - renice

- Jobs command lists the jobs

```
$ sleep 100&
[1] 32202
$ sleep 200&
[2] 32203
$ sleep 300&
[3] 32204
$ jobs
[1]   Running   sleep 100 &
[2]-  Running   sleep 200 &
[3]+  Running   sleep 300 &
$ fg 3
sleep 300
[1]   Done      sleep 100
CTRL+Z
[3]+  Stopped   sleep 300
$
```

bg & fg commands

- fg resume specified job foreground
  fg *jobnumber*
- bg resume specified suspended job in background
  bg *jobnumber*

```
$ sleep 100&
[1] 32202
$ sleep 200&
[2] 32203
$ sleep 300&
[3] 32204
$ jobs
[1]   Running   sleep 100 &
[2]-  Running   sleep 200 &
[3]+  Running   sleep 300 &
$ fg 3
sleep 300
[1]   Done      sleep 100
CTRL+Z
[3]+  Stopped   sleep 300
$
```

nice command

- Nice command is used to run a command with a specific priority

nice [*option*] *command* [*args*]

-N or **-n N** where N is a number is the adjustment value

Default adjustment value is 10

nice command – example

```
$ sleep 100&
[1] 29015
$ ps -l 29015
...    PID  PPID  C PRI  NI ... CMD
... 29015 20685  0  75   0 ... sleep 100
$ nice sleep 100&
[2] 29017
$ ps -l 29017
...    PID  PPID  C PRI  NI ... CMD
... 29017 20685  0  90  10 ... sleep 100
$ nice -15 sleep 100&
[3] 29045
$ ps -l 29045
...    PID  PPID  C PRI  NI ... CMD
... 29045 20685  0  95  15 ... sleep 100
```

renice command

- change scheduling priority of a running process

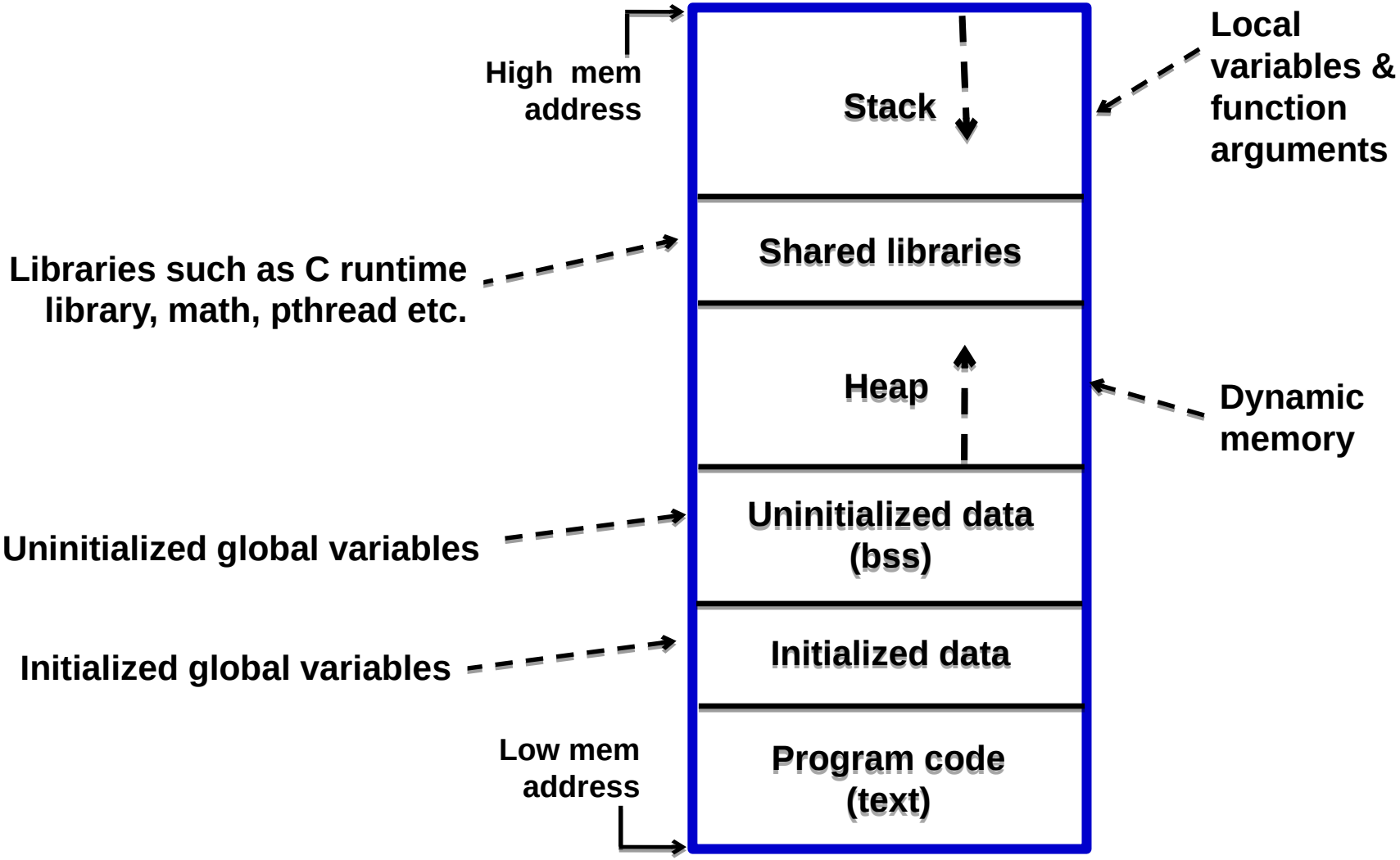
renice *priority* *pid*

renice command – example

```
$ sleep 500&
[2] 29777
$ ps -l 29777
...   PID  PPID  C PRI  NI  ... CMD
... 29777 20685  0  75   0  ... sleep 500
$ renice 15 29777
29777: old priority 0, new priority 15
$ ps -l 29777
...   PID  PPID  C PRI  NI   ... CMD
... 29777 20685  0  95  15  ...     sleep 500
$ renice 17 29777
29777: old priority 15, new priority 17
$ ps -l 29777
...   PID  PPID  C PRI  NI    ...   CMD
... 29777 20685  0  97  17    ...   sleep 500
```

Process Layout & Exploring Executables

UNIX Process Layout



Exploring executables

- The following bin utilities are useful to explore an executable
  - file determine file type
  - nm to print symbols of executable or object file symbol names include variables or function names
  - size print section sizes of executable or object files

file command

```
$ file sample.o
sample.o: ELF 32-bit LSB relocatable, Intel
 80386, version 1 (SYSV), not stripped
$

$ file a.out
a.out: ELF 32-bit LSB executable, Intel
 80386, version 1 (SYSV), for GNU/Linux
 2.2.5, dynamically linked (uses shared
 libs), not stripped
$
```

Sample program

```
static int a;
int b = 10;
int c;
void foo_2(int x)    {
   printf("in foo_2()\n");
   printf("foo_2  x+a = %d\n", x+a);
}
void foo_1(int x)    {
   printf("in foo_1()\n");
   c = 1;
   foo_2(x+b+c);
}
```

```
int main()
{
   int k = 3;

   foo_1(k);

   return 0;
}
```

nm command

```
$ nm -l a.out
080495ec b a
080494f0 D b
080495f0 B c
08048375 T foo_1  /home/wipro18/.x/sample.c:14
08048344 T foo_2  /home/wipro18/.x/sample.c:7
08048250 T _init
080483b1 T main /home/wipro18/.x/sample.c:23
08048298 T _start
```

D initialized data
B global uninitialized data
b uninitialized data
T code section

lower case indicates local identifier
UPPER case indicates global identifier

size command

```
$ size a.out
   text   data    bss    dec    hex filename
    968    260     12   1240    4d8 a.out
```

size of text, data, bss in decimal & hex notation

text executable code in bytes (in decimal format)

data initialized data in bytes (in decimal format)

bss uninitialized data in bytes (in decimal format)

Hands-on and Assignments

Exercises

1) Write a program which forks a process and the parent waits for the child to terminate and prints the exit status, considering the following cases
   a) Child terminates without a call to exit()
   b) Child terminates calling exit(n)
   c) Child terminates calling return(n)

   Try with values n = 0, n > 0 and n < 0 for argument **n**

2) Write an application that creates N child processes each sleeping for random amount of time.
   use waitpid(), to see that the parent process waits for each of the child processes to terminate and prints the pid of the child process and its exit status.

1) Write a program which forks a process and the parent waits(wait()) for the child to terminate and prints the exit status, considering the following cases

   a) wait()
   b) exit status of the terminated process

Give one example for renice command

Exercises (Contd.).

3) On the executables of the programs that you have already created run the following commands:
   a) file
   b) size
   c) nm

Summary

In this module, we discussed

- process related system calls
- how to create a process
- about the organization of proc file system
- per process entries in /proc
- memory layout of a process
- commands to explore object & executable files

Review Questions

1) What are the different states of a process?
2) When does a process get into zombie state?
3) Identify one invalid process state transition.
4) Give one example for uninitialized data.
5) How a command to be executed along with its arguments is passed to execvp()?

References

1) W. Richard Stevens and Stephen A. Rago, Ed 2., New Delhi: Pearson Education, 2009.

2) Kay A. Robbins and Steven Robbins, UNIX Systems Programming, New Delhi: Pearson Education, 2009.

3) Rochkind, Advanced Unix Programming, Ed 2. New Delhi: Pearson Education, 2008.

4) Arnold Robbins, Linux Programming by Example, New Delhi: Pearson Education, 2008.

Thank You