BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN) – 333 031
IS C462 – Network Programming
Lab#1

Topic: Environment of a Single Process

- We are all aware of writing programs in c. When we execute a program, a process is created by the shell. Process in short is defined as 'a program in execution'. Kernel schedules the process to run on CPU. Kernel also allocates the space(memory) for the process to store its data, code etcetera. This space is 'process address space'. Every process has its own space. Kernel has data structures which maintain the information about processes. Anytime a process needs to access resources like a file, keyboard or a screen it requests the kernel.

- To see the active processes in the system, we use 'ps' command. 'ps' stands for process status.

Usage of ps command:

```
$ps
```

- This will display all the processes associated with your terminal. These are the processes associated with your current session terminal.

```
$ps x
```

- This will display all the processes owned by your user id. If you see output of this command, you will observe that some of them don't have a terminal. These processes are either background processes or daemons. They will be covered later. Can you identify any daemon from this output?

```
$ps ux
```

- This displays all your processes in user format. You can see that there is unique PID for every process. This is the process identifier, a positive integer. Also there is status of the process; running, sleeping, zombie etc.

```
$ps -el
```

- This command displays all the processes in the system. You can see that there is UID which is user id. This is the id of the user who owns this process. You can see that there is PPID which is parent process id. Every process has a parent pid. Can you find out a process which doesn't have a process? Can you find out the first user process in the system? Also you can see that there are so many processes without a terminal (TTY). These are all the services that are running in the system. They generally end with 'd' indicating that it is a daemon.

- Take any process in the list. Find out its parent. Now take this parent and find its parent. In this way, reach up to the process first created in the system. This is parent-child hierarchy in the system.

```
$ps –elH
```

- This command displays the parent-child relationships among processes.

```
$ps –Heo pid,ppid,user,state,command
```

- This will print the user specified attributes for each process. You can see that under command attribute, along with command-line arguments, there are environment variables such as HOME, USER, SHELL, PATH etc.

- To get a real time view of processes in the system use `$top or $htop`
- See the manual for more option on ps command. `$man ps`

Using environment variables:

- In the last command we have seen that a list of environment variables being passed to a program. When we run a program in the bash shell, the shell creates a new process and loads the program into it by calling system call. At that time, the environment variables are copied into the global area of the process address space. We will discuss this in the class.

- These environment variables can be accessed within the program. They are mainly used to access the settings of the shell. The environment variable-value pairs are stored in an array of character pointers. The address of this array is stored in environ.

```
extern char **environ;
```

The values are stored as

```
HOME=/home/students/f2006091
PATH=:/bin:/user/bin
SHELL=/bin/bash
USER=f2006091
LOGNAME=f2006091
```

- The following program displays the command line arguments as well as environment variables that are available in the process when this program is run.

```
//environ.c
int
main (int argc, char *argv[])
{
  int i;
  char **ptr;
  extern char **environ;

  for (i = 0; i < argc; i++)    /* echo all command-line args */
    printf ("argv[%d]: %s\n", i, argv[i]);

  for (ptr = environ; *ptr != 0; ptr++) /* and all env strings */
    printf ("%s\n", *ptr);
```

```
  exit (0);
}
```

Q?

- After executing this program, see the list of environment variables displayed. There are standard functions to manipulate these variables within the process. **`putenv(), setenv(), unsetenv().`** They are present in stadlib module. Find more about them using man pages.

  Insert the following line in the above program.

```
setenv("SOME_ENV_VAR", "SOME_ENV_VAL", 1);
putenv("NAME=URNAME");
```

- Run and see the change in the list of variables.

Finding about a process:

- The following program prints the some details of a process in which it is running.

```
//process.c
#include <unistd.h>

main ()
{
  printf ("I am running in a process whose details are as follows\n");
  printf("process id (pid) = %d, parent process id(ppid) = %d, user
id(uid) = %d\n",getpid (), getppid (), getuid ());
}
```

Q?

- When you run this program, it prints process id, parent id, and user who owns the process. Run the program several times. Did you observe something?
- Can you tell why the process id changing every time?
- Can you tell why the parent is same all the time?
- Can you think of a circumstance where the parent can be different? [Hint: fork()]

Process Creation:

- As we discussed in the class, a process is created by fork() system call. Consider the following program.

```
//process_creation.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int glob = 6;                    //global variable
int
main ()
{
  int var;
  pid_t pid;

  var = 88;
  printf ("Before fork\n");

  if ((pid = fork ()) < 0)
    perror ("fork");   //function to print error that occurred in the
process
  else if (pid == 0)
    {
      glob++;
      var++;
      printf ("pid = %d, glob=%d, var=%d\n", getpid (), glob, var);
      exit (0);
                    }
  else
    {
      printf ("pid = %d, glob=%d, var=%d\n", getpid (), glob, var);
      exit (0);
    }
}
```

Q?

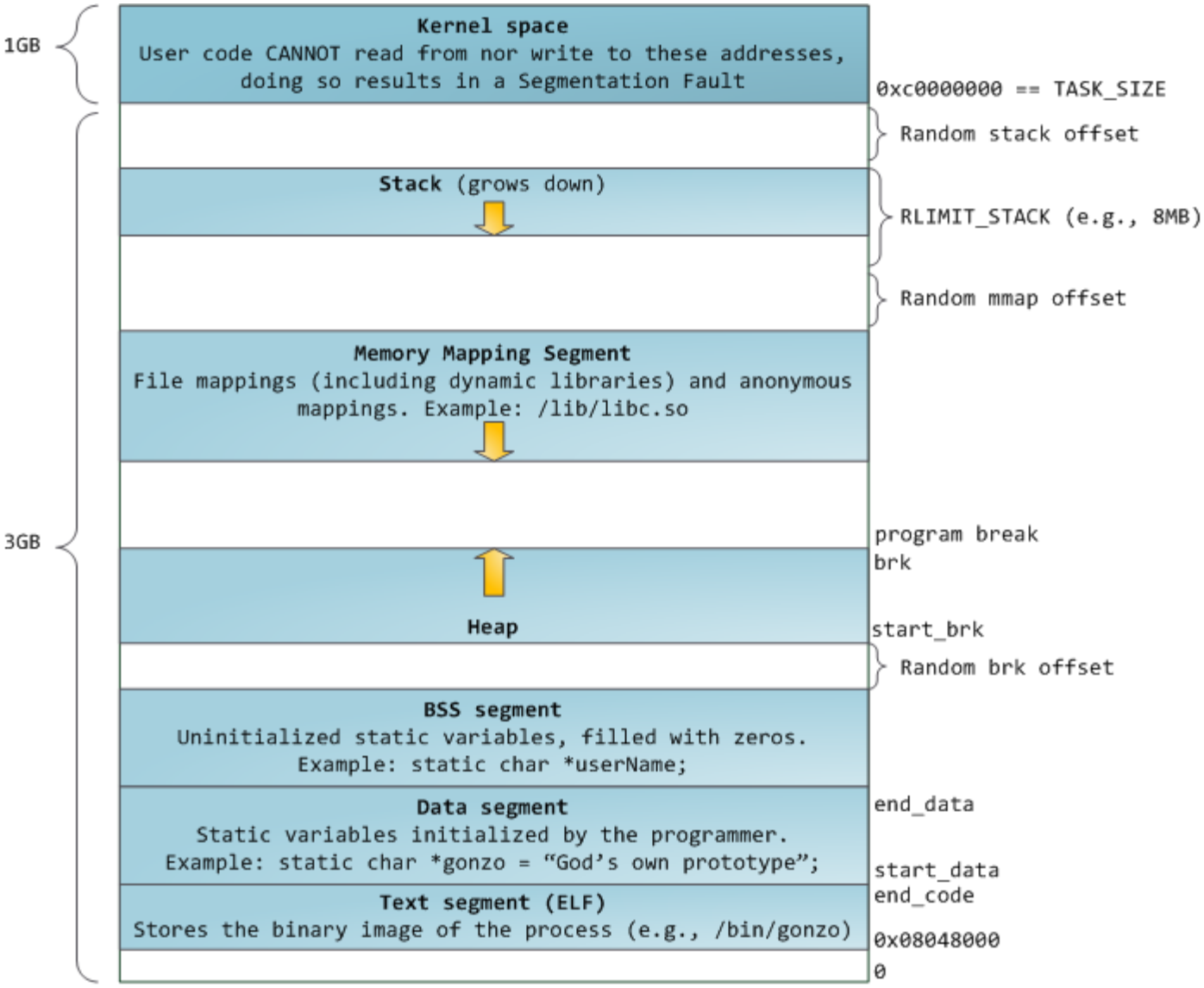
- What did you observe in the output? You will notice that the changes made by child are not affecting the parent's data.
- Change the fork() to vfork() in the above program and see the output. Did you find any change?

Process Memory:

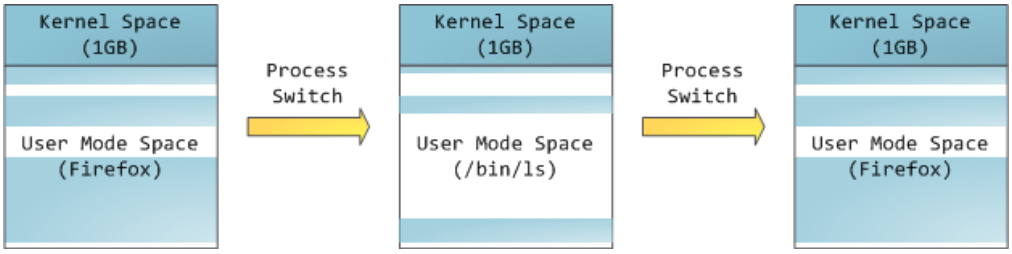
- We discussed in the class about the memory layout of a process. Let us get into some more details.
- To see the sizes of text, data and bss sections in any executable file, use the following code.

```
$size a.out
```

- in a 32 bit system, 4 GB address space divided as shown below. Some portion of this address space is mapped onto the physical memory. Blue colored regions are mapped to physical memory. Following diagram is taken from http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory.



- Since there can be only one process accessing CPU at a time, that has all the address space to use. When context switch happens, this address space is mapped on to a different set of physical memory pages. But kernel address space mappings will remain unchanged through a context switch.



- Use the following command to see the address space mappings of your current process. We can replace self with any valid pid to see another process address map.

```
$vi /proc/self/maps
```

- Please run the following program. This will give some idea of address mappings.

/*addressmap.c*/

```
#include <stdlib.h>
#include <stdio.h>
double t[0x02000000];
void segments()
{
static int s = 42;
void *p = malloc(1024);
printf("stack\t%010p\nbrk\t%010p\nheap\t%010p\n"
"static(bss)\t%010p\nstatic(data)\t%010p\ntext\t%010p\n",
&p, sbrk(0), p, t, &s, segments);
}
int main(int argc, char *argv[])
{
segments();
exit(0);
}
```

- This will print output like:

```
stack    0xbfc40fe0
brk      0x1806b000
heap     0x1804a008
static(bss)      0x08049720
static(data)     0x080496f0
text     0x080483f4
```

- sbrk() is a sys call that is used by malloc() when it needs to add memory to its pool. sbrk() takes number of bytes as input and adds that much memory to the process and changes the brk pointer.

Q?

- change the above program so that you allocate 1MB memory using malloc(). Then use printf statement. do you notice any change.

wait() and waitpid():

- We will use the following program to understand wait() and waitpid() calls. Run the following program and observe the result of synchronization using wait().

```
//wait.c
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
main ()
{
  int i = 0, j = 0;
  pid_t ret;
  int status;

  ret = fork ();
  if (ret == 0)
    {
      for (i = 0; i < 5000; i++)
        printf ("Child: %d\n", i);
      printf ("Child ends\n");
    }
  else
    {
      wait (&status);
      printf ("Parent resumes.\n");
      for (j = 0; j < 5000; j++)
        printf ("Parent: %d\n", j);
    }
}
```

Q?

- Remove the wait() and see the result. The output is interleaved randomly. It is because once the child is created, which one will run first parent or child is not predictable and the standard output descriptor is shared between parent and child. They write to the same file descriptor.
- Modify the above program.

1. Use waitpid() instead of wait().

```
waitpid (ret,0,0);
```

By default waitpid() call is a blocking call.

2. Make it non-blocking by using
   waitpid (ret,0,WNOHANG);

WNOHANG The `waitpid` function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.

- Can you think of ways to synchronize the printing of child and parent in such way that child and parent print the lines alternatively? {hint: use signals]

Q?

a) What is the output of the following Code?

```
main()
  {
      val = 5;
      if(fork())
          wait(&val);
      val++;
      printf("%d\n", val);
      return val;
  }
```

b)
```
int   main()
{  int pid1,pid2;
        printf("FIRST\n");
 pid1=fork();
      if(pid1==0) {
           printf("SECOND \n");
      pid2=fork();
            printf("SECOND \n");
    }  else   {
        printf("THIRD\n");
      }
}
```

i) How many total processes are created if the above code runs?

ii) How many times each of the strings "FIRST","SECOND" and "THIRD" in the above code printed?

c) Given the following piece of code

```
main(int argc, char ** argv)
 {
     forkme(4)
 }
void forkme(int n)   {   if(n > 0)
         {          fork();
            forkme(n-1);
```

```
}}
```

If the above piece of code runs, how many processes are created?

=========End of Lab#1=========