Recitation 6

—

fork()

1. Creates a new process that is called the child process. The process that makes the fork() call is called the parent process. Both the processes run concurrently.
2. After a child process is created, both the processes will execute the next line in the code following the fork() system call.
3. Both the child process uses the same files and the same registers as the parent process.
4. Note that fork() does not take in any arguments. It also returns integral values. A negative value is returned if the fork() call was unsuccessful. Zero returned to the newly created child process and a positive value returned to the caller.

Sample Programs

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main() {
            fork();

                printf("Hello world!\n");
                     return 0;
                     }
~
~
~
~
~
~
~
~
```

Sample Programs

```
-sh-4.2$ gcc fork1.c
-sh-4.2$ vi fork1.c
-sh-4.2$ ./a.out
Hello world!
Hello world!
-sh-4.2$
```

How many times is “Hello World” printed?

```c
#include  <stdio.h>
#include  <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
~
~
~
~
~
~
~
~
~
~
~
~
~
```

Output

```
-sh-4.2$ gcc fork2.c
-sh-4.2$ ./a.out
hello
hello
hello
hello
hello
-sh-4.2$ hello
hello
hello
```

In general

```
fork ();   // Line 1
fork ();   // Line 2
fork ();   // Line 3

       L1       // There will be 1 child process
    /     \     // created by line 1.
  L2      L2    // There will be 2 child processes
 /  \    /  \   //  created by line 2
L3  L3  L3  L3  // There will be 4 child processes
                // created by line 3
```

Number of times “Hello” printed is equal to 2^n where n is the number of times the fork() statement is called.

Another Example

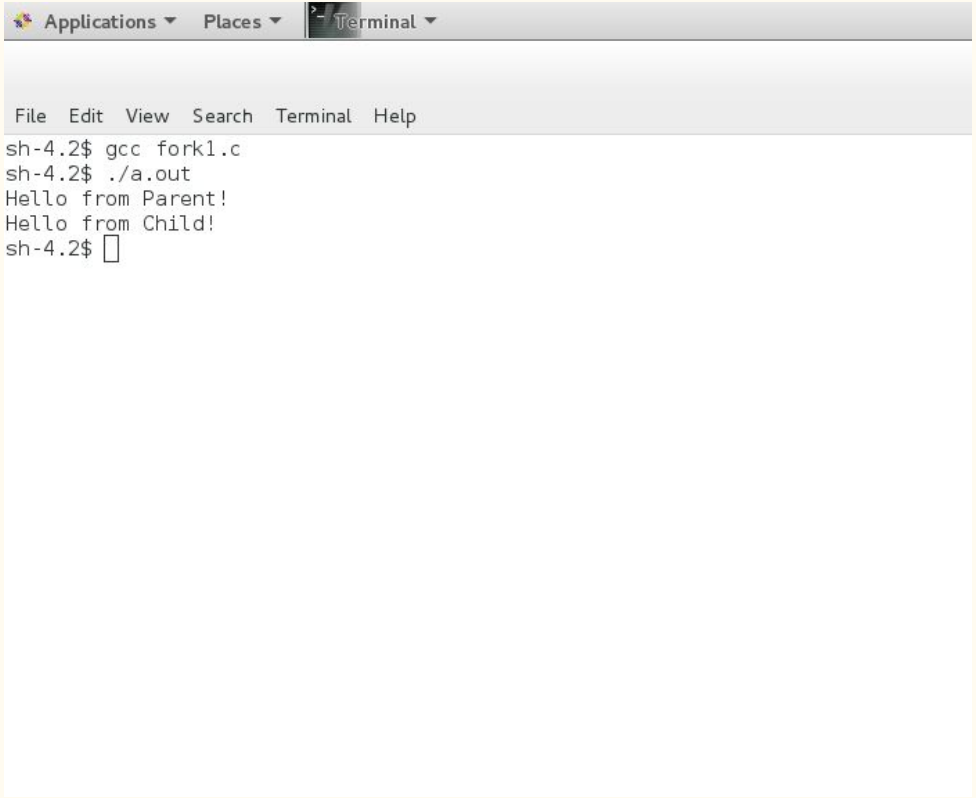
Applications Places Terminal

File Edit View Search Terminal Help

```
i#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork()==0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

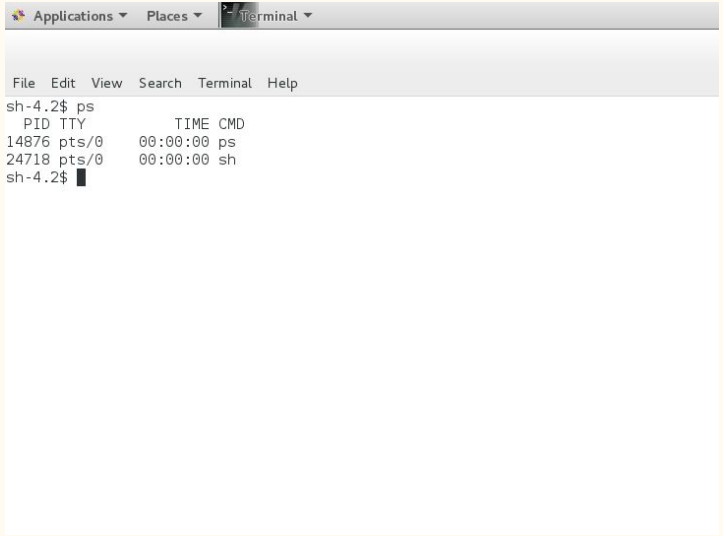
The Output


Applications
Places
Terminal
File Edit View Search Terminal Help
sh-4.2$ gcc fork1.c
sh-4.2$ ./a.out
Hello from Parent!
Hello from Child!
sh-4.2$

Process management

1. The operating system tracks processes through a unique 5 digit ID called a PID.
2. PIDs eventually repeat because all possible numbers are used up and the next pid rolls over. At any point of time, no two processes with the same PID exist in the system.
3. Another important identification is PPID. This is basically the ID of the parent process that started this process.

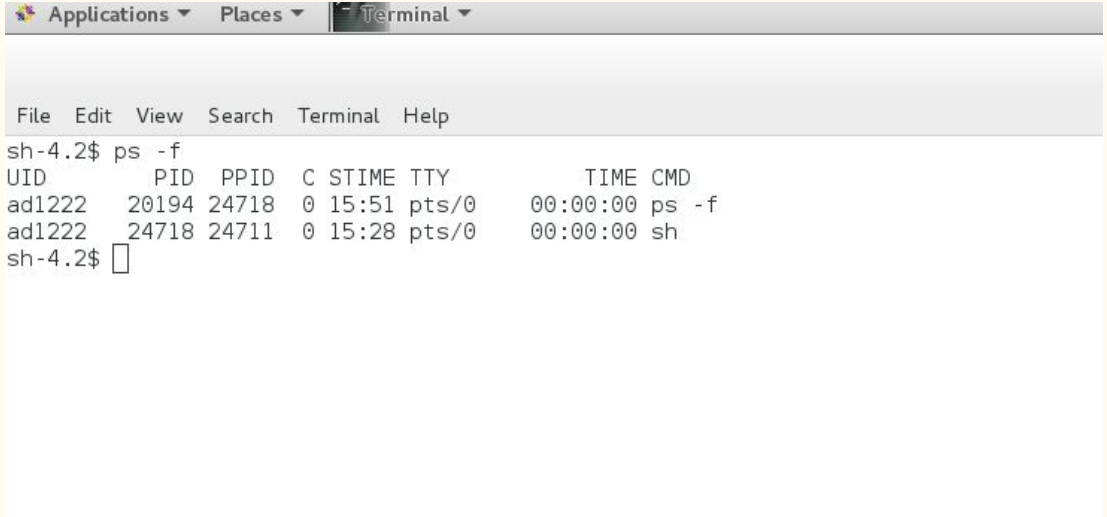
Listing Running Processes



You can see your processes by running the ps (process state) command.

Listing Running Processes

The ps -f command gives us more information:



What do these columns mean?

1. UID: User ID of the person who is running the process.
2. PID: Process ID
3. PPID: Parent Process ID
4. C: CPU utilization of the process.
5. STIME: Process start time
6. TIME: CPU time taken by the process
7. CMD: command that started this process

Use kill PID to stop a process.

top

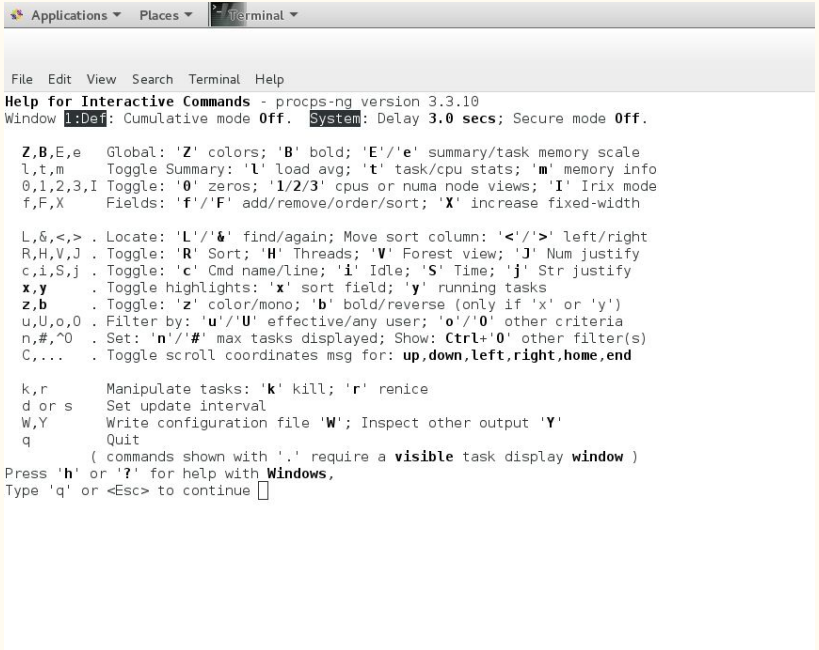
```
Applications  Places  Terminal

File  Edit  View  Search  Terminal  Help
top - 16:48:49 up 14 days,  1:38,  2 users,  load average: 0.32, 0.49, 0.47
Tasks: 288 total,   1 running, 287 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.8 us,  0.7 sy,  0.0 ni, 95.3 id,  0.2 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem : 32687824 total,   284280 free,  3640332 used, 28763212 buff/cache
KiB Swap: 41746424 total, 41746424 free,        0 used. 27751672 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
25081 ad1222    20   0 4357028 1.772g 105528 S  30.3  5.7 327:34.91 firefox
20017 root       0 -20       0      0      0 S   0.7  0.0   0:00.21 kworker/7:0H
24299 ad1222    20   0  708688  12644   9004 S   0.7  0.0   0:09.03 pulseaudio
  921 root      20   0    4368    588    496 S   0.3  0.0 178:48.98 rngd
15321 ad1222    20   0  168152   2544   1608 R   0.3  0.0   0:00.40 top
    1 root      20   0  191148   4248   2432 S   0.0  0.0   7:48.36 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.22 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:01.24 ksoftirqd/0
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.34 migration/0
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0   7:59.80 rcu_sched
   10 root      rt   0       0      0      0 S   0.0  0.0   0:02.66 watchdog/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:02.63 watchdog/1
   12 root      rt   0       0      0      0 S   0.0  0.0   0:00.29 migration/1
   13 root      20   0       0      0      0 S   0.0  0.0   0:01.14 ksoftirqd/1
   16 root      rt   0       0      0      0 S   0.0  0.0   0:02.42 watchdog/2
   17 root      rt   0       0      0      0 S   0.0  0.0   0:00.25 migration/2
   18 root      20   0       0      0      0 S   0.0  0.0   0:01.02 ksoftirqd/2
   21 root      rt   0       0      0      0 S   0.0  0.0   0:02.52 watchdog/3
   22 root      rt   0       0      0      0 S   0.0  0.0   0:00.39 migration/3
   23 root      20   0       0      0      0 S   0.0  0.0   0:01.00 ksoftirqd/3
   26 root      rt   0       0      0      0 S   0.0  0.0   0:02.87 watchdog/4
   27 root      rt   0       0      0      0 S   0.0  0.0   0:18.22 migration/4
   28 root      20   0       0      0      0 S   0.0  0.0   0:01.76 ksoftirqd/4
   31 root      rt   0       0      0      0 S   0.0  0.0   0:02.82 watchdog/5
   32 root      rt   0       0      0      0 S   0.0  0.0   0:12.31 migration/5
   33 root      20   0       0      0      0 S   0.0  0.0   0:01.82 ksoftirqd/5
   36 root      rt   0       0      0      0 S   0.0  0.0   0:02.94 watchdog/6
   37 root      rt   0       0      0      0 S   0.0  0.0   0:15.08 migration/6
   38 root      20   0       0      0      0 S   0.0  0.0   0:01.17 ksoftirqd/6
   41 root      rt   0       0      0      0 S   0.0  0.0   0:03.23 watchdog/7
   42 root      rt   0       0      0      0 S   0.0  0.0   0:26.20 migration/7
   43 root      20   0       0      0      0 S   0.0  0.0   0:27.91 ksoftirqd/7
   47 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   48 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
   49 root      20   0       0      0      0 S   0.0  0.0   0:00.47 khungtaskd
   50 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 writeback
   51 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kintegrityd
   52 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 bioset
   53 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kblockd
   54 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 md
   60 root      20   0       0      0      0 S   0.0  0.0   0:01.22 kswapd0
   61 root      25   5       0      0      0 S   0.0  0.0   0:00.00 ksmd
   62 root      39  19       0      0      0 S   0.0  0.0   0:52.99 khugepaged
   63 root      20   0       0      0      0 S   0.0  0.0   0:01.90 fsnotify_mark
   64 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 crypto
   72 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kthrotld
   75 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kmpath_rdacd
   76 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kpsmoused
   78 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 ipv6_addrconf
   97 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 deferwq
  134 root      20   0       0      0      0 S   0.0  0.0   5:58.60 kauditd
  285 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 ata_sff
  307 root      20   0       0      0      0 S   0.0  0.0   0:00.00 scsi_eh_0
  308 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 scsi_tmf_0
  309 root      20   0       0      0      0 S   0.0  0.0   0:00.00 scsi_eh_1
  310 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 scsi_tmf_1
  312 root      20   0       0      0      0 S   0.0  0.0   0:00.00 scsi_eh_2
  313 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 scsi_tmf_2
  314 root      20   0       0      0      0 S   0.0  0.0   0:00.00 scsi_eh_3
  315 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 scsi_tmf_3
  316 root      20   0       0      0      0 S   0.0  0.0   0:00.00 scsi_eh_4
  317 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 scsi_tmf_4
  318 root      20   0       0      0      0 S   0.0  0.0   0:00.00 scsi_eh_5
  319 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 scsi_tmf_5
```

top

To learn more about top press ‘h’ when it is running:



top vs ps

1. top is used interactively while ps is used non-interactively. Non-interactive use include uses in scripts, extracting information through shell pipelines.
2. ps gives you a single snapshot. top displays statistics until stopped. You can stop top using the ctrl+z command.
3. top displays processes in order of the processor usage.

Key switches for top command

1. top -o: Sorts by the named field
2. top -p: Only shows processes with specified process IDs
3. top -u: Shows processes by the specified users
4. top -i: Don’t show idle processes

wait() system call

1. A call to wait() blocks the calling process until one of its child processes exits or a signal is received.
2. After child process terminates, parent ***continues*** its execution after wait system call instruction.
3. If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates.
4. If only one child process is terminated, then return a wait() returns process ID of the terminated child process.
5. If more than one child processes are terminated then wait() considers any ***arbitrarily child*** and return a process ID of that child process.
6. If any process has no child process then wait() returns immediately “-1”.

Programs: wait()

Applications Places Terminal

File Edit View Search Terminal Help

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    pid_t cpid;
    if (fork()== 0)
        exit(0);            /* terminate child */
   else
       cpid = wait(NULL); /* reaping parent */
    printf("Parent pid = %d\n", getpid());
    printf("Child pid = %d\n", cpid);

    return 0;
}
~
~
~
~
~
~
~
~
~
```

Applications Places Terminal

File Edit View Search Terminal Help

```
sh-4.2$ gcc wait.c
sh-4.2$ ./a.out
Parent pid = 28695
Child pid = 28696
sh-4.2$ vi wait.c
sh-4.2$
```

Another Example

Applications Places Terminal

File Edit View Search Terminal Help

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    if (fork()== 0)
        printf("HC: hello from child\n");
    else
    {
        printf("HP: hello from parent\n");
        wait(NULL);
        printf("CT: child has terminated\n");
    }

    printf("Bye\n");
    return 0;
}
~
~
~
~
~
~
~
~
~
~
~
```

The output

Applications Places Terminal

File Edit View Search Terminal Help

```
sh-4.2$ vi wait2.c
sh-4.2$ gcc wait2.c
sh-4.2$ ./a.out
HP: hello from parent
HC: hello from child
Bye
CT: child has terminated
Bye
sh-4.2$
```

Without wait()

Applications Places Terminal

File Edit View Search Terminal Help

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    if (fork()== 0)
        printf("HC: hello from child\n");
    else
    {
        printf("HP: hello from parent\n");
        printf("CT: child has terminated\n");
    }

    printf("Bye\n");
    return 0;
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
```

The output

Applications Places Terminal

File Edit View Search Terminal Help

```
sh-4.2$ gcc wait3.c
sh-4.2$ ./a.out
HP: hello from parent
CT: child has terminated
Bye
HC: hello from child
Bye
sh-4.2$
```

Some tips for last times HW and today

1. DIR opendir(const char *dir_name): opens the directory specified by dir_name and returns a pointer to the directory stream.
2. struct dirent *readdir(DIR dirp): The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp. It returns NULL on reaching the end of the directory stream or if an error occurred.
3. DIR: The dir datatype represents a directory stream (which is an ordered sequence of all the directory entries).

Some tips for last times HW and today

3. The dirent data structure:

```
struct dirent
{
    ino_t          d_ino;              /* inode number */
    off_t          d_off;              /* offset to the next dirent */
    unsigned short d_reclen;           /* length of this record */
    unsigned char  d_type;             /* type of file; not supported by all file system types
*/
    char           d_name[256];        /* filename */
};
```

Some tips for last times HW And today

4. Under Linux, /**proc** includes a **directory** for each running process, including kernel processes, in **directories** named /**proc**/PID, where PID is the process number. Each **directory** contains information about one process, including: /**proc**/PID/cmdline, the command that originally started the process.

Today's HW

1. Implement ps:

Some hints

Applications Places Terminal

File Edit View Search Terminal Help

```
char * base = "/proc"
DIR * stuff = opendir(base, RD_ONLY)
dirent * pidnumber = NULL;

char * newCmdline = NULL;
char * workingName = NULL;

int fd = -1;

while
{
pidnumber = readdir(stuff);
if( pidnumber != NULL && pidnumber->d_type == DT_DIR) { newCmdline = ... malloc strlen(base)+strlen(pidnumber->d_name)+9;
newCmdline[0] = '\0';
strcat(newCmdline, base);
strcat(newCmdline, "/");
strcat(newCmdline, pidnumber->d_name);
strcat(newCmdline, "/cmdline"

fd = open(newCmdline, RD_ONLY);

... read loop ...
  .. .while read from fd != 0, printf it out ...

close(fd);

}

}
do(pidnumber != NULL);

... this gets you all command lines run for all pids for all procs on the system

printf( command run and its pid(i.e. directory name of cmdline) )

~
~
~
~
~
```

Some hints

Applications Places Terminal

Terminal

File Edit View Search Terminal Help

```
1. Then open the file 'status', look for the uid section and extract the owner's uid. Using pwd.h, determine the name of the user who owns the process and print it out as well.

open status alongside/after cmdline (but before clocking your readdir loop! readdir is destructive!) read through and parse the status file looking for 'uid'

while reading in status file ...
if buffer[i] == 'u'
if bufferLength - 1 >= 2
  if (buffer[i+1] == 'i' && buffer[i+2] == 'd')
   ... can start reading in uid that called this code ... w00t!

printf( command run, its pid and the uid that called it) .. boring .. want userNAME, not UID .. :^P .. but only place in system this information is together is in passwd file ... very well, then...


struct passwd * getUname = getpwuid( UID parsed out of status above )

new can print out:
  command run (from cmdline file)
  username that called it ( passwd->pw_name )

.. for every current PID
```

Some hints

2. Then open the file schedstat and read in order: time spent running on CPU (in nanoseconds), time spent waiting on a runqueue, # of times context switched

(be careful of decimals! you may want to check the status file to be sure your degree is correct)