CSC 4304 - Systems Programming
Fall 2008

LECTURE - VIII

UNIX PROCESS ENVIRONMENT

Tevfik Koşar

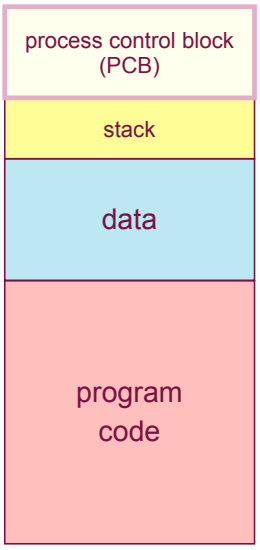
Louisiana State University
September 25th, 2008

In Today's Class

- Unix Process Environment
  - Process Concept
  - ps -- get process info
  - Shell & its implementation
  - Exec() & Fork()
  - Creation & Termination of Processes

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack: temporary data

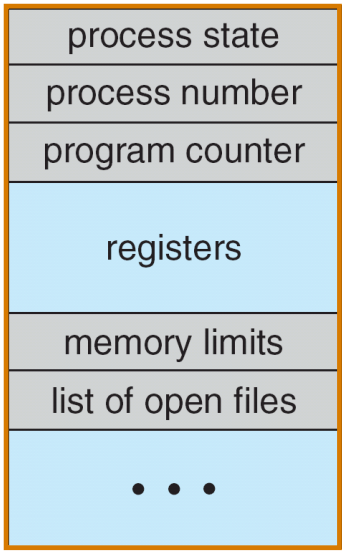


Process in Memory

Process Control Block (PCB)

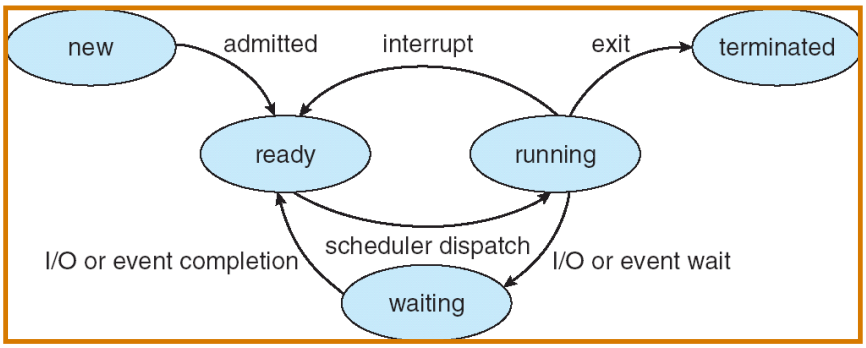
Information associated with each process

- Process state
  - (running, waiting..)
- Program counter
- CPU registers
- CPU scheduling information
  - (i.e. process priority)
- Memory-management information
  - (i.e. page & segment tables)
- Accounting information
- I/O status information



Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a process
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution




```
$ ps
PID   TTY      TIME     CMD
18684 pts/4    00:00:00 bash
18705 pts/4    00:00:00 ps
```

$ ps a

```
  PID TTY      STAT   TIME COMMAND
 6702 tty7     Ss+   15:10 /usr/X11R6/bin/X :0 -audit 0
 7024 tty1     Ss+    0:00 /sbin/mingetty --noclear tty1
 7025 tty2     Ss+    0:00 /sbin/mingetty tty2
 7026 tty3     Ss+    0:00 /sbin/mingetty tty3
 7027 tty4     Ss+    0:00 /sbin/mingetty tty4
 7028 tty5     Ss+    0:00 /sbin/mingetty tty5
 7029 tty6     Ss+    0:00 /sbin/mingetty tty6
17166 pts/6    Ss     0:00 -bash
17191 pts/6    S+     0:00 pico program3.cc
17484 pts/5    Ss+    0:00 -bash
17555 pts/7    Ss+    0:00 -bash
17646 pts/8    Ss     0:00 -bash
17809 pts/10   Ss     0:00 -bash
17962 pts/8    S+     0:00 pico prog2.java
17977 pts/1    Ss     0:00 -bash
18014 pts/9    Ss+    0:00 -bash
18259 pts/10   T      0:00 a.out
18443 pts/2    Ss     0:00 -bash
18511 pts/1    S+     0:00 pico program3.cc
18684 pts/4    Ss     0:00 -bash
```


$ ps la

```
 F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
4     0  6702  6701  15   0  25416  7204 -      Ss+  tty7      15:10 /usr/X11R6/bin/X :0 -
audit 0 -auth /var/lib/g
4     0  7024     1  17   0   3008     4 -      Ss+  tty1       0:00 /sbin/mingetty --
noclear tty1
4     0  7025     1  16   0   3008     4 -      Ss+  tty2       0:00 /sbin/mingetty tty2
4     0  7026     1  16   0   3012     4 -      Ss+  tty3       0:00 /sbin/mingetty tty3
4     0  7027     1  17   0   3008     4 -      Ss+  tty4       0:00 /sbin/mingetty tty4
4     0  7028     1  17   0   3008     4 -      Ss+  tty5       0:00 /sbin/mingetty tty5
4     0  7029     1  17   0   3008     4 -      Ss+  tty6       0:00 /sbin/mingetty tty6
0  2317 17166 17165  15   0   9916  2300 wait   Ss   pts/6      0:00 -bash
0  2317 17191 17166  16   0   8688  1264 -      S+   pts/6      0:00 pico program3.cc
0  2238 17484 17483  16   0   9916  2300 -      Ss+  pts/5      0:00 -bash
0  2611 17555 17554  15   0   9912  2292 -      Ss+  pts/7      0:00 -bash
0  2631 17646 17644  16   0   9912  2300 wait   Ss   pts/8      0:00 -bash
0  2211 17809 17808  15   0   9916  2324 wait   Ss   pts/10     0:00 -bash
0  2631 17962 17646  16   0   8688  1340 -      S+   pts/8      0:00 pico prog2.java
0  2320 17977 17976  16   0   9912  2304 wait   Ss   pts/1      0:00 -bash
```

$ ps -fa

```
UID        PID  PPID  C STIME TTY          TIME CMD
cs1253as 17191 17166  0 10:13 pts/6    00:00:00 pico program3.cc
cs135032 17962 17646  0 10:57 pts/8    00:00:00 pico prog2.java
cs1253av 18511 17977  0 11:20 pts/1    00:00:00 pico program3.cc
15059    19109 18684  0 11:44 pts/4    00:00:00 ps -fa
```


$ ps -ax

```
 PID TTY      STAT   TIME COMMAND
   1 ?        S      0:02 init [5]
   2 ?        S      0:00 [migration/0]
   3 ?        SN     0:00 [ksoftirqd/0]
   4 ?        S      0:00 [migration/1]
   5 ?        SN     0:01 [ksoftirqd/1]
   6 ?        S      0:00 [migration/2]
   7 ?        SN     0:16 [ksoftirqd/2]
   8 ?        S      0:00 [migration/3]
   9 ?        SN     0:16 [ksoftirqd/3]
  10 ?        S<     0:00 [events/0]
  11 ?        S<     0:00 [events/1]
  12 ?        S<     0:00 [events/2]
  13 ?        S<     0:00 [events/3]
  14 ?        S<     0:00 [khelper]
  15 ?        S<     0:00 [kthread]
 653 ?        S<     0:00 [kacpid]
 994 ?        S<     0:00 [kblockd/0]
 995 ?        S<     0:00 [kblockd/1]
 996 ?        S<     0:01 [kblockd/2]
 997 ?        S<     0:00 [kblockd/3]
```

Shell

- A tool for process and program control
- Three main functions
  - Shells run programs
  - Shells manage I/O
  - Shells can be programmed
- Main Loop of a Shell

```
while (!end_of_input){
   get command
   execute command
   wait for command to finish
}
```


How does a Program run another Program?

- Program calls **execvp**

```
int execvp(const char *file, char *const argv[]);
```

- Kernel loads program from disk into the process
- Kernel copies arglist into the process
- Kernel calls main(argc,argv)

Exec Family

```
int execl(const char *path, const char *arg, ...);

int execlp(const char *file, const char *arg, ...);

int execle(const char *path, const char *arg , ...,
                             char * const envp[]);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);
```


Running “ls -l”

```
   #include <unistd.h>
#include <stdio.h>

main()
{
   char   *arglist[3];

   arglist[0] = "ls";
   arglist[1] = "-l";
   arglist[2] = 0 ;

   printf("* * * About to exec ls -l\n");
   execvp( "ls" , arglist );
   printf("* * * ls is done. bye\n");
}
```

execvp is like a Brain Transplant

- execvp loads the new program into the current process, replacing the code and data of that process!

Writing a Shell v1.0

```
     int main()
{
     char *arglist[MAXARGS+1];        /* an array of ptrs    */
     int  numargs;           /* index into array    */
     char argbuf[ARGLEN];             /* read stuff here */
     char *makestring();              /* malloc etc      */

     numargs = 0;
     while ( numargs < MAXARGS )
     {
          printf("Arg[%d]? ", numargs);
          if ( fgets(argbuf, ARGLEN, stdin) && *argbuf != '\n' )
               arglist[numargs++] = makestring(argbuf);
          else
          {
               if ( numargs > 0 ){         /* any args?  */
                    arglist[numargs]=NULL; /* close list */
                    execute( arglist );    /* do it */
                    numargs = 0;       /* and reset  */
               }
          }
     }
     return 0;
}
```

```
#include <stdio.h>
#include <signal.h>
#include <string.h>

#define MAXARGS 20
#define ARGLEN 100
```

Writing a Shell v1.0 *(cont.)*

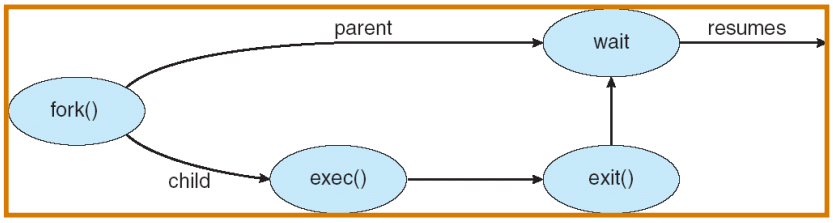
```
     int execute( char *arglist[] )
{
     execvp(arglist[0], arglist);           /* do it */
     perror("execvp failed");
     exit(1);
}


char * makestring( char *buf )
{
     char *cp, *malloc();

     buf[strlen(buf)-1] = '\0';            /* trim newline   */
     cp = malloc( strlen(buf)+1 );         /* get memory */
     if ( cp == NULL ){                    /* or die */
          fprintf(stderr,"no memory\n");
          exit(1);
     }
     strcpy(cp, buf);                     /* copy chars  */
     return cp;                          /* return ptr   */
}
```


How to Create a New Process?

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program



How fork works?

```
pid_t fork(void);
```

- Allocates a new chunk of memory and data structures
- Copies the original process into the new process
- Adds the new process to the set of running processes
- Returns control back to both processes

Fork Implementation

```
int main()
{
   Pid_t  pid;
  /* fork another process */
  pid = fork();
  if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      exit(-1);
  }
  else if (pid == 0) { /* child process */
      execlp("/bin/ls", "ls", NULL);
  }
  else { /* parent process */
      /* parent will wait for the child to
  complete */
```


Fork Example 1

```
    #include   <stdio.h>

main()
{
    int ret_from_fork, mypid;

    mypid = getpid();            /* who am i?    */
    printf("Before: my pid is %d\n", mypid);   /* tell pid */

    ret_from_fork = fork();

    sleep(1);
    printf("After: my fork returns pid : %d, said %d\n",
           ret_from_fork, getpid());
}
```

Fork Example 2

```
    #include   <stdio.h>

main()
{
    fork();
    fork();
    fork();
    printf("my pid is %d\n", getpid() );
}
```

How many lines of output will this produce?

Writing a Shell v2.0

```
   execute( char *arglist[] )
{
    int pid,exitstatus;               /* of child*/

    pid = fork();                  /* make new process */
    switch( pid ){
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            execvp(arglist[0], arglist);      /* do it */
            perror("execvp failed");
            exit(1);
        default:
            while( wait(&exitstatus) != pid )
                ;
            printf("child exited with status %d,%d\n",
                    exitstatus>>8, exitstatus&0377);
    }
}
```

Exercise

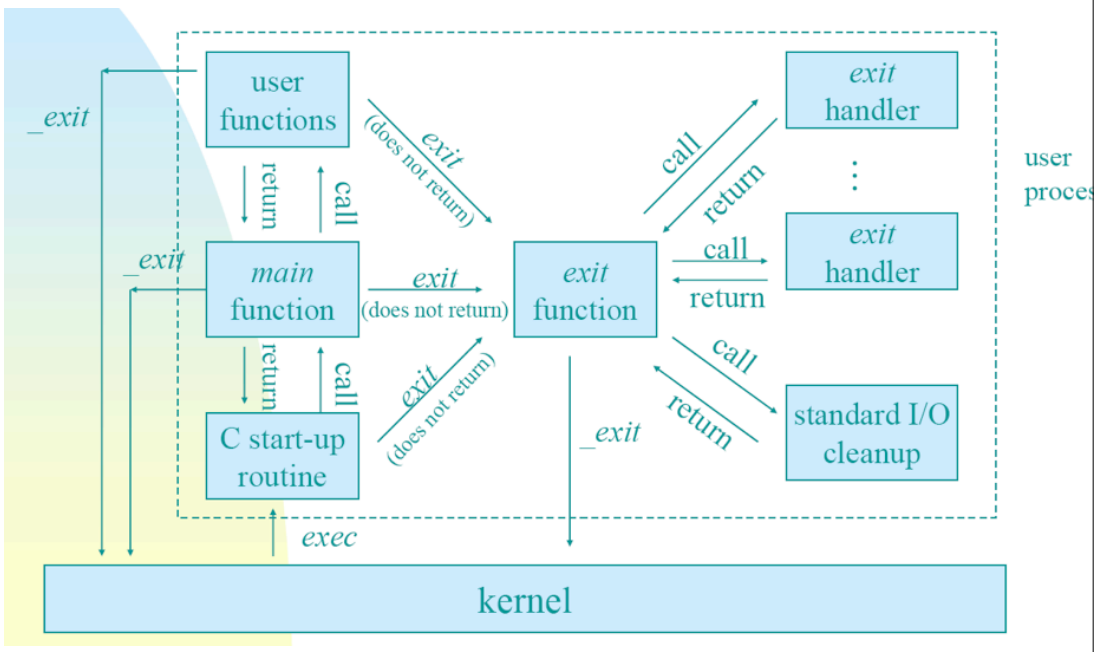
Improve the Shell v2.0 by:

- Allow the user to type all the arguments on one line
- Allow the user to quit by typing exit

Process Termination

- Five ways to terminate:
- Normal Termination
  - Return from main()
  - Call exit()
  - Call _exit()
- Abnormal termination
  - Call abort()
  - Be terminated by a signal

Process Start and Termination



Summary

- Unix Process Environment
  - Process Concept
  - ps -- get process info
  - Shell & its implementation
  - Exec() & Fork()
  - Creation & Termination of Processes

- Next Class: Process Control
- Try “fork” and “shell” examples

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), and B. Knicki (WPI).