CSC 4304 - Systems Programming
Fall 2010

Lecture - VII

Unix Process Environment

Tevfik Koşar

Louisiana State University

September 14th, 2010

In Today's Class

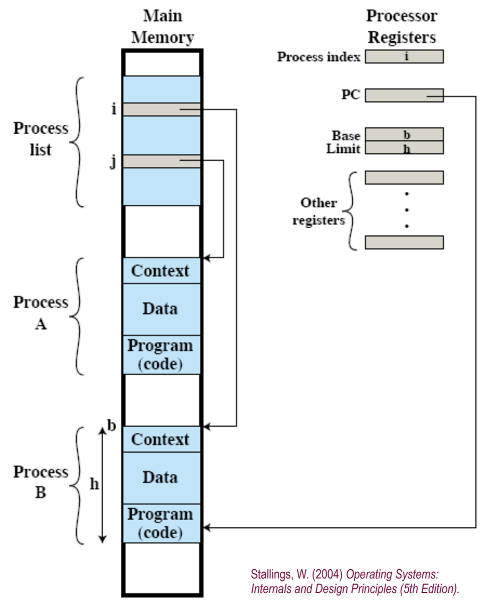
- Unix Process Environment
  - Process Concept
  - Creation & Termination of Processes
  - Exec() & Fork()
  - ps -- get process info
  - Shell & its implementation

Process Concept

- a Process is a program in execution;

➢ A process image consists of three components

user address space
1. an executable program
2. the associated data needed by the program
3. the execution context of the process, which contains all information the O/S needs to manage the process (ID, state, CPU registers, stack, etc.)



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Typical process image implementation

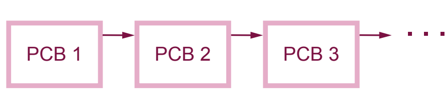
Process Control Block

➢ The Process Control Block (PCB)

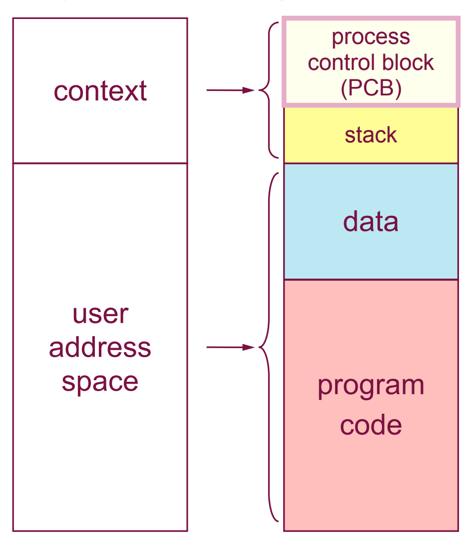
✓ is included in the context, along with the stack

✓ is a “snapshot” that contains all necessary and sufficient data to restart a process where it left off (ID, state, CPU registers, etc.)

✓ is one entry in the operating system’s **process table** (array or linked list)

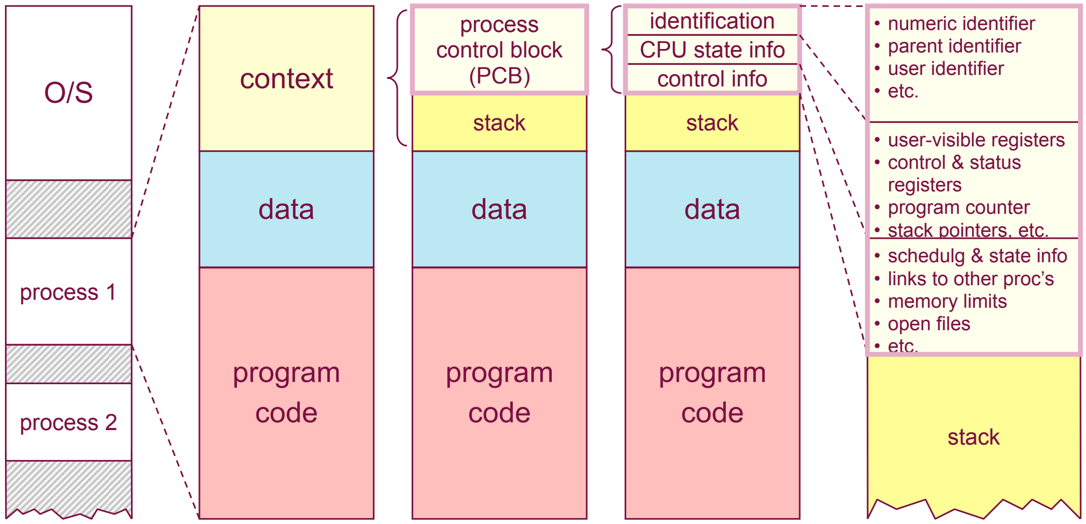


Typical process image implementation



Process Control Block

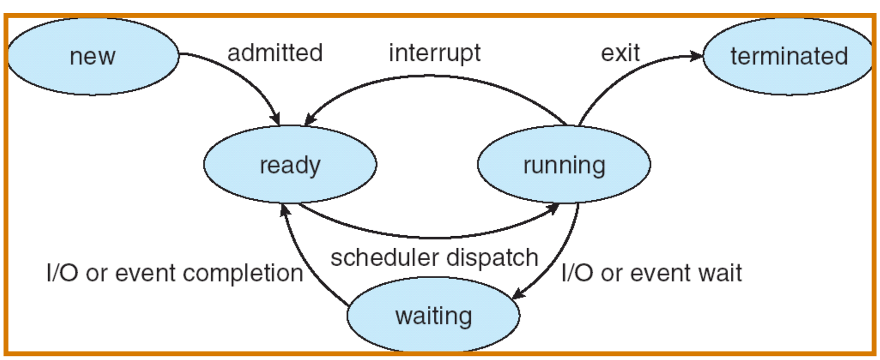
➢ Example of process and PCB location in memory



Illustrative contents of a process image in (virtual) memory

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 -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]
```

Process Creation

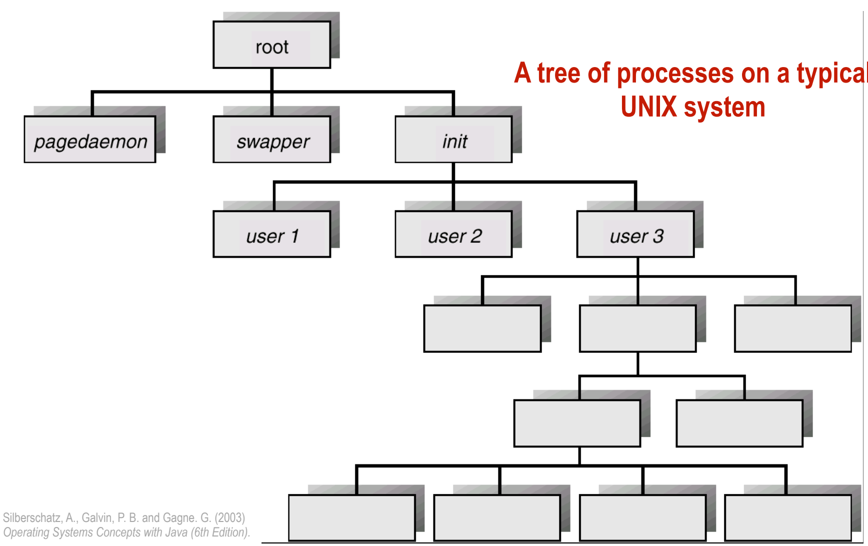
➢ <u>Some events that lead to process creation (enter)</u>

all cases of process spawning

- ✓ the system boots
  - when a system is initialized, several background processes or “daemons” are started (email, logon, etc.)
- ✓ a user requests to run an application
  - by typing a command in the CLI shell or double-clicking in the GUI shell, the user can launch a new process
- ✓ an existing process spawns a child process
  - for example, a server process (print, file) may create a new process for each request it handles
  - the *init* daemon waits for user login and spawns a shell
- ✓ a batch system takes on the next job in line

Process Creation

➢ <u>Process creation by spawning</u>



A tree of processes on a typical UNIX system

Silberschatz, A., Galvin, P. B. and Gagne. G. (2003) *Operating Systems Concepts with Java (6th Edition).*

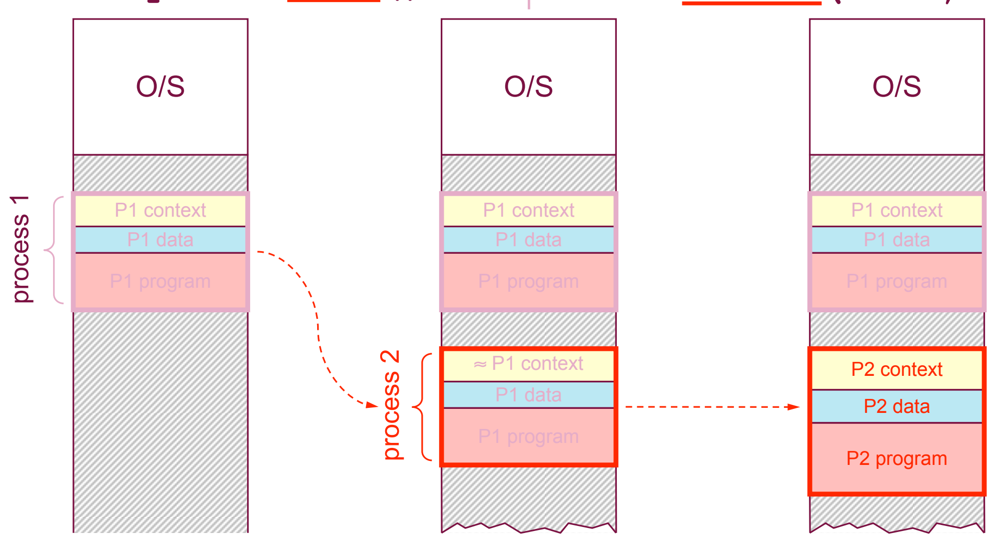
Process Creation

```
...
int main(...)
{
    ...
    if ((pid = fork()) == 0)                    // create a process
    {
        fprintf(stdout, "Child pid: %i\n", getpid());
        err = execvp(command, arguments);       // execute child
                                                //   process
        fprintf(stderr, "Child error: %i\n", errno);
        exit(err);
    }
    else if (pid > 0)                           // we are in the
    {                                           //   parent process
        fprintf(stdout, "Parent pid: %i\n", getpid());
        pid2 = waitpid(pid, &status, 0);        // wait for child
        ...                                     //   process
    }
    ...

    return 0;
}
```


Process Creation

1. Clone child process
   - ✓ `pid = fork()`
2. Replace child's image
   - ✓ `execve(name, ...)`



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?

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[]);
```


execvp is like a Brain Transplant

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

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");
}
```


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   */
}
```


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);
     }
 }
```

Process Termination

➤ <u>Some events that lead to process termination (exit)</u>

- ✓ regular completion, with or without error code

  process-triggered

  - the process voluntarily executes an **`exit(err)`** system call to indicate to the O/S that it has finished

- ✓ fatal error (uncatchable or uncaught)

  O/S-triggered (following system call or preemption)

  - service errors: no memory left for allocation, I/O error, etc.
  - total time limit exceeded

  hardware interrupt-triggered

  - arithmetic error, out-of-bounds memory access, etc.

- ✓ killed by another process via the kernel

  software interrupt-triggered

  - the process receives a **`SIGKILL`** signal
  - in some systems the parent takes down its children with it

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

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).