Computer Science & Engineering Department
I. I. T. Kharagpur

Operating System: CS33007
3rd Year CSE: 5th Semester (Autumn 2006 - 2007)
Lecture II (Linux System Calls I)

Goutam Biswas *Date:* 26th July, 2006

1 Process Creation

1. **fork()** - create a child process in its own image. The call returns values to both the processes - the child gets zero (0) and the parent gets the process id of the child.

2. **waitpid()** - the parent may be asked to wait for the child to terminate.

```
/*************************************
 *  Creation of new process by fork() *
 *************************************/

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int n ; // global data
int main() {
   int chPID, status ;

   if((chPID = fork()) != 0) {  // parent
      n = 25 ;
      printf("\tInside parent: pid = %d\n", getpid()) ;
      printf("\tChild pid = %d\n", chPID) ;
      waitpid(chPID, &status, 0) ;
      printf("\tn = %d, &n = %p\n", n, &n) ;
   }
   else {  // child
      printf("\t\t\t\tInside child: pid = %d\n", getpid()) ;
      printf("\t\t\t\tParent pid = %d\n", getppid()) ;
      n = 100 ;
      printf("\t\t\t\tn = %d, &n = %p\n", n, &n) ;
   }
   return 0 ;
}
```

3. A system call has a software interrupt within it. The command e.g. fork is passed through the register **eax**, and other parameters are passed through different registers.

The software interrupt used by all the system calls of Linux on Pentium is **int 0x80**. The value returned by the system call is in the register **eax**.

```
/************************************
 *  System call within fork          *
 ************************************/

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int n ; // global data
int main() {
   int chPID, status ;

__asm__ __volatile__(
                 "movl $2,%%eax\n\t"
                 "int $0x80\n\t"
                 :"=a" (chPID)
                 ) ;

   if(chPID != 0) {  // parent
      n = 25 ;
      printf("\tInside parent: pid = %d\n", getpid()) ;
      printf("\tChild pid = %d\n", chPID) ;
      waitpid(chPID, &status, 0) ;
      printf("\tn = %d, &n = %p\n", n, &n) ;
   }
   else {  // child
      printf("\t\t\t\tInside child: pid = %d\n", getpid()) ;
      printf("\t\t\t\tParent pid = %d\n", getppid()) ;
      n = 100 ;
      printf("\t\t\t\tn = %d, &n = %p\n", n, &n) ;
   }
   return 0 ;
}
```

The key word `__asm__` specifies that a sequence of inline assembly code follows. The keyword `__volatile__` specifies that the compiler should not change the position of the code.

4. **getpid()** - returns the process id of the calling proces.

5. **getppid()** - returns the process id of the parent of the calling process.

2 Load & Execute a Process Image

1. A shell while executing an external command[1] forks a process and loads the executable module (process image) corresponding to the command.

2. **execve()** - is the system call to load the process image from an executable module in an existing process and exute it. There are different wrapper function known as exec-calls written using this - execl, execlp, execle, execv, execvp.

3. The **execve()** system call takes the following parameters:

   - The command (11) in the register eax
   - The path of the executable file as a string in ebx
   - The command line arguments as **char** *[] in ecx
   - The environment variable and value pairs as **char** *[] in edx

```
/***********************************
 * This program uses execve system  *
 * call. Execute -                  *
 * $ ./a.out ./factorial 6          *
 * *********************************/

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[], char *envp[]) {
 int chPID, status, i = -1;
 char **agv = argv+1;

 while(agv[++i]) printf("agv[%d] = %s\n", i, agv[i]) ;

 if((chPID = fork()) != 0) { // Parent
     printf("\tInside Parent\n") ;
     waitpid(chPID, &status, 0) ;
     printf("\tEnd of child: %d\n", chPID) ;
 }
 else { // Child
     printf("\t\t\tInside Child\n") ;
     execve(agv[0], agv, envp) ;
     printf("\t\t\tCannot be printed: %d\n", getppid()) ;
 }
```

[1]There are two types of shell commands, internal e.g. cd, echo, pwd etc., and external e.g. a.out, ls, mkdir etc. The shell generates a system call corresponding to an internal command e.g. chdir(), echo() etc. But it loads the executable module corresponding to an external command e.g. ./a.out, /bin/ls, /bin/mkdir etc.

```
 return 0 ;
}
```

4. The factorial computing function takes the input as a command line argument.

```
/**********************************************
 * Factorial: cc -Wall factorial.c -o factorial *
 * *********************************************/

#include <stdio.h>
#include <stdlib.h>

int main(int count, char *vects[]) {
 int n, i, fact = 1 ;

 if(count < 2) {
      printf("Less command line argument\n") ;
      exit(0) ;
 }
 n = atoi(vects[1]) ;
 for(i=1; i<=n; ++i) fact *=i ;
 printf("%d! = %d\n", n, fact) ;

 return 0 ;
}
```

5. Now we replace the execve() call by the corresponding assembly code of software interrupt.

```
/*************************************
 * This program uses execve system  *
 * call. Execute -                  *
 * $ ./a.out ./factorial 6          *
 * **********************************/

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[], char *envp[]) {
 int chPID, status, i = -1;
 char **agv = argv+1;

 while(agv[++i]) printf("agv[%d] = %s\n", i, agv[i]) ;

 if((chPID = fork()) != 0) { // Parent
```

```
     printf("\tInside Parent\n") ;
     waitpid(chPID, &status, 0) ;
     printf("\tEnd of child: %d\n", chPID) ;
 }
 else { // Child
        printf("\t\t\tInside Child\n") ;
        __asm__ __volatile__(
                            "movl $11, %%eax\n\t"
                            "int $0x80\n\t"
                            :
                            :"b" (agv[0]), "c" (agv), "d" (envp)
                            ) ;
       //execve(agv[0], agv, envp) ;
       printf("\t\t\tCannot be printed: %d\n", getppid()) ;
      }

 return 0 ;
}
```

3 Terminating and Blocking a Process

1. **_exit()** - terminates the current process. The status is reurned to the parent process. If the parent of a child is killed, it is inherited by the *init* process (processid 1).

   ```
   /*****************************************
    * _exit() terminates a process
    *****************************************/
   #include <stdio.h>
   #include <sys/types.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main() {
      int chPID, status ;

      if((chPID = fork()) != 0) {  // parent
         printf("\tInside parent: pid = %d\n", getpid()) ;
         printf("\tChild pid = %d\n", chPID) ;
         waitpid(chPID, &status, 0) ;
         printf("\tExit status is: %d\n", WEXITSTATUS(status)) ; // Exit status
      }
      else {  // child
         printf("\t\t\t\tInside child: pid = %d\n", getpid()) ;
         printf("\t\t\t\tParent pid = %d\n", getppid()) ;
         _exit(10) ; // Small value
      }
      return 0 ;
   }
   ```

2. The code with software interrupt is

   ```
   /*****************************************
    * _exit() using software interrupt
    *****************************************/
   #include <stdio.h>
   #include <sys/types.h>
   #include <unistd.h>
   #include <sys/wait.h>

   int main() {
      int chPID, status ;

      if((chPID = fork()) != 0) {  // parent
         printf("\tInside parent: pid = %d\n", getpid()) ;
         printf("\tChild pid = %d\n", chPID) ;
         waitpid(chPID, &status, 0) ;
   ```

```
      printf("\tExit status is: %d\n", WEXITSTATUS(status)) ; // Exit status
   }
   else {  // child
      printf("\t\t\t\tInside child: pid = %d\n", getpid()) ;
      printf("\t\t\t\tParent pid = %d\n", getppid()) ;
      __asm__ __volatile__ (
                           "movl $1, %%eax\n\t"
                           "int $0x80\n\t"
                           :
                           :"b" (10)
                           ) ;
      //_exit(10) ; // Small value
   }
   return 0 ;
}
```

3. **sleep()** - a process may be put to sleep for a number of seconds.

```
/****************************************
 * Process goes to sleep()
 ****************************************/
#include <stdio.h>
#include <unistd.h>
#define TIME 10

int main(){

       printf("Before the sleep\n") ;
       sleep(TIME) ;
       printf("After %d secs of sleep\n", TIME) ;

       return 0 ;
}
```

References

[1] http://asm.sourceforge.net//syscall.htmli#3

[2] http://www-128.ibm.com/developerworks/library/l-ia.html#h1