Processos em UNIX

1. Introdução

A norma POSIX fornece dois mecanismos para a criação de atividades concorrentes. O primeiro, abordado neste capítulo, corresponde ao tradicional mecanismo fork do UNIX e a sua chamada de sistema associada, o wait. A chamada fork dentro de um processo provoca a criação de um clone perfeito deste processo para a execução.

POSIX permite ainda que cada um dos processos criados contenham diversas threads (tarefas) de execução. Essas threads têm acesso às mesmas posições de memória e rodam num espaço de endereçamento único. Uma introdução ao uso dessas primitivas será apresentada na seqüência deste texto.

1.1. Processo: Uma definição

Um processo é um ambiente de execução que consiste em um segmento de instruções, e dois segmentos de dados (data e stack). Deve-se, entretanto, notar a diferença entre um processo e um programa: um programa nada mais é que um arquivo contendo instruções e dados utilizados para inicializar segmentos de instruções e de dados do usuário de um processo.

1.2. Identificadores de um processo

Cada processo possui um identificador (ou ID) único denominado pid. Como no caso dos usuários, ele pode estar associado a um grupo, e neste caso será utilizado o identificador denominado pgpr. As diferentes primitivas permitindo o acesso aos diferentes identificadores de um processo são as seguintes:

```
 #include <unistd.h>

 pid_t getpid()                       /* retorna o ID do processo  */
 pid_t getppid()                        /* retorna o ID do pai do processo
*/
 int setpgid(pid_t pid, pid_t pgid);  /* seta o valor do ID do grupo do */
                                      /* especificado por pid para pgid */
 pid_t getpgid(pid_t pid);            /* retorna o ID do grupo de processos
*/
                                      /* especificado por pid */
 int setpgrp(void);                   /* equivalente a setpgid(0,0) */
 pid_t getpgrp(void);                 /* equivalente a getpgid(0) */
```

Valor de retorno: 0 se setpgid e setpgrp são executados com sucesso e, -1 em caso de erro.

Exemplo:

```
/* arquivo test_idf.c */
#include <stdio.h>
#include <unistd.h>
```

```
int main()
{
     printf("Eu sou o processo %d de pai %d e de grupo %d\n",getpid()
             ,getppid(),getpgrp()) ;
     exit(0);
}
```

Resultado da execução:

test_idf
Eu sou o processo 28448 de pai 28300 e de grupo 28448

Observe que o pai do processo executando test_idf é o processo tcsh. Para confirmar a afirmação, faça um ps na janela de trabalho:

```
# ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:00 -tcsh
28451  ?  R    0:00 ps
```

Observação:
Grupos de processo são usados para distribuição de sinais, e pelos terminais para controlar as suas requisições. As chamadas setpgid e setpgrp são usadas por programas como o csh() para criar grupo de processos na implementação de uma tarefa de controle e não serão utilizadas no decorrer do curso.

2. As primitivas envolvendo processos

2.1. A primitiva fork()

```
#include <unistd.h>

pid_t fork(void)       /* criação de um processo filho */
pid_t vfork(void);     /* funciona como um alias de fork */
```

Valor de retorno: 0 para o processo filho, e o identificador do processo filho para o processo pai; -1 em caso de erro (o sistema suporta a criação de um número limitado de processos).

Esta primitiva é a única chamada de sistema que possibilita a criação de um processo em UNIX. Os processos pai e filho partilham o mesmo código. O segmento de dados do usuário do novo processo (filho) é uma cópia exata do segmento correspondente ao processo antigo (pai). Por outro lado, a cópia do segmento de dados do filho do sistema pode diferir do segmento do pai em alguns atributos específicos (como por exemplo, o pid, o tempo de execução, etc.). Os filhos herdam uma duplicata de todos os descritores dos arquivos abertos do pai (se o filho fecha um deles, a cópia do pai não será modificada). Mais ainda, os ponteiros para os arquivos associados são divididos (se o filho movimenta o ponteiro dentro de um arquivo, a próxima manipulação do pai será feita a partir desta nova posição do ponteiro). Esta noção é muito importante para a implementação dos pipes (tubos) entre processos.

Exemplo:

```
/* arquivo test_fork1.c */
/* Descritores herdados pelos processos filhos */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>

int main()
{
int pid ;
int fd ; /* descritor de arquivo associado ao arquivo agenda */
char *telephone ;
int r ; /* retorno de read */
int i ;
char c ;
     printf("Oi, eu sou o processo %d\n",getpid()) ;
     printf("Por favor, envie-me o seu numero de telefone\n") ;
     printf("E o 123456789 ? Ok, ja anotei na minha agenda\n") ;
     if((fd=open("agenda",O_CREAT|O_RDWR|O_TRUNC,S_IRWXU))==-1)
     {
          perror("impossivel de abrir a agenda") ;
          exit(-1) ;
     }
     telephone="123456789" ;
     if(write(fd,telephone,9)==-1)
     {
          perror("impossivel de escrever na agenda") ;
          exit(-1) ;
     }
     printf("Enfim, acabei de anotar e estou fechando a agenda\n") ;
     close(fd) ;
     printf("O que ? Eu me enganei ? O que foi que eu anotei ?\n") ;
     printf("\tO pai reabre sua agenda\n") ;
                  if((fd=open("agenda",O_RDONLY,S_IRWXU))==-1)      {
          perror("impossivel de abrir a agenda") ;
          exit(-1) ;
     }
     printf("\tNesse instante, o pai gera um filho\n") ;
     pid=fork() ;
     if(pid==-1) /* erro */
     {
          perror("impossivel de criar um filho") ;
          exit(-1) ;
     }
     else if(pid==0) /* filho */
     {
          sleep(1) ; /* o filho dorme para agrupar as mensagens */
          printf("\t\tOi, sou eu %d\n",getpid()) ;
          printf("\t\tVoces sabem, eu tambem sei ler\n") ;
          printf("\tO filho entao comeca a ler a agenda\n") ;
          for(i=1;i<=5;i++)
          {
               if(read(fd,&c,1)==-1)
               {
                    perror("impossivel de ler a agenda") ;
                    exit(-1) ;
```

```
                }
                printf("\t\tEu li um %c\n",c) ;
           }
           printf("\tMinha agenda ! Diz o pai\n") ;
           printf("\te supreso o filho fecha a agenda...\n") ;
           close(fd) ;
           sleep(3) ;
           printf("\tO filho entao se suicida de desgosto!\n") ;
           exit(1) ;
     }
     else /* pai */
     {
           printf("De fato, eu apresento a voces meu filho %d\n",pid) ;
           sleep(2) ;
           printf("Oh Deus ! Eu nao tenho mais nada a fazer\n");
           printf("Ah-ha, mas eu ainda posso ler minha agenda\n") ;
           while((r=read(fd,&c,1))!=0)
           {

                if(r==-1)
                {
                     perror("impossivel de ler a agenda") ;
                     exit(-1) ;
                }
                printf("%c",c) ;
           }
           printf("\n") ;
           printf("ENFIM ! Mas onde estao todos ?\n") ;
           sleep(3) ;
           close(fd) ;
     }
     exit(0);
}
```

Resultado da Execução:

test_fork1
Oi, eu sou o processo 28339
Por favor, envie-me o seu numero de telefone
E o 123456789 ? Ok, ja anotei na minha agenda
Enfim, acabei de anotar e estou fechando a agenda
O que ? Eu me enganei ? O que foi que eu anotei ?
    O pai reabre sua agenda
    Nesse instante, o pai gera um filho
        Oi, sou eu 28340
        Voces sabem, eu tambem sei ler
    O filho entao comeca a ler a agenda
        Eu li um 1
        Eu li um 2
        Eu li um 3
        Eu li um 4
        Eu li um 5
    Minha agenda ! Diz o pai
    e supreso o filho fecha a agenda...
    O filho entao se suicida de desgosto!
Oi, eu sou o processo 28339

Por favor, envie-me o seu numero de telefone
E o 123456789 ? Ok, ja anotei na minha agenda
Enfim, acabei de anotar e estou fechando a agenda
O que ? Eu me enganei ? O que foi que eu anotei ?
    O pai reabre sua agenda
    Nesse instante, o pai gera um filho
De fato, eu apresento a voces meu filho 28340
Oh Deus ! Eu nao tenho mais nada a fazer
Ah-ha, mas eu ainda posso ler minha agenda
6789
ENFIM ! Mas onde estao todos ?

Observação:

Três pontos principais devem ser observados no exemplo anterior:

1. filho herda os descritores "abertos" do pai - uma vez que o filho pode ler a agenda sem que seja necessário abri-la.
2. filho pode fechar um descritor aberto pelo pai, sendo que esse descritor continuará aberto para o pai.
3. Os dois processos compartilham do mesmo ponteiro sobre o arquivo duplicado na chamada da primitiva fork! Note que quando o pai vai ler o arquivo, ele vai se movimentar dentro desse arquivo da mesma forma que seu filho.

Comportamento da saída no console:

Note que se o pai e o filho vivem, uma interrupção de teclado (via CTRL-c) irá destruir os dois processos. Entretanto, se um filho vive enquanto seu pai está morto, uma interrupção pode não matá-lo. Veja o exemplo de programa a seguir.

Exemplo:

```
/* arquivo test_fork2.c */
/* Testa as reacoes do console quando um pai
 * morre e o filho continua vivo */

#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
int pid ;
   printf("Eu sou o pai %d e eu vou criar um filho \n",getpid()) ;
   pid=fork() ; /* criacao do filho */
   if(pid==-1) /* erro */
   {
      perror("impossivel de criar um filho\n") ;
   }
   else if(pid==0) /* acoes do filho */
   {
      printf("\tOi, eu sou o processo %d, o filho\n",getpid()) ;
```

```
      printf("\tO dia esta otimo hoje, nao acha?\n") ;
      printf("\tBom, desse jeito vou acabar me instalando para sempre\n");
      printf("\tOu melhor, assim espero!\n") ;
      for(;;) ; /* o filho se bloqueia num loop infinito */
   }
   else /* acoes do pai */
   {
       sleep(1) ; /* para separar bem as saidas do pai e do filho */
       printf("As luzes comecaram a se apagar para mim, %d\n",getpid()) ;
       printf("Minha hora chegou : adeus, %d, meu filho\n",pid) ;
       /* e o pai morre de causas naturais */
    }
    exit(0);
}
```

Resultado da Execução:

test_fork2
Eu sou o pai 28637 e eu vou criar um filho
    Oi, eu sou o processo 28638, o filho
    O dia esta otimo hoje, nao acha?
    Bom, desse jeito vou acabar me instalando para sempre
    Ou melhor, assim espero!
As luzes comecaram a se apagar para mim, 28637
Minha hora chegou : adeus, 28638, meu filho

Se o comando shell ps é executado no console, a seguinte saída é obtida:

```
# ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:00 -tcsh
28638  ?  R    0:04 test_fork2
28639  ?  R    0:00 ps
```

Note que o filho permanece rodando! Tente interrompê-lo via teclado usando CTRL-c ou CTRL-d. Ele não quer morrer, não é verdade? Tente matá-lo diretamente com um sinal direto através do comando shell kill.

kill -9 <pid>

No caso do exemplo anterior, deveria ser feito:

kill -9 28638

Desta vez estamos certos que Jason morreu! ?

```
# kill -9 28638
# ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:00 -tcsh
28666  ?  R    0:00 ps
```

2.1.1. Problema com os buffers de saída:

```
/* arquivo test_fork3.c */
/* O filho herda uma copia do buffer de saida do pai */

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

int main()
{
int pid ;
     printf(" 1") ;
     pid=fork() ;
     if(pid==-1) /* error */
     {
          perror("impossivel de criar um filho") ;
          exit(-1) ;
     }
     else if(pid==0) /* filho */
     {
          printf(" 2") ;
          exit(0) ;
     }
     else /* pai */
     {
          wait(0) ; /* o pai aguarda a morte de seu filho */
          printf(" 3") ;
          exit(0);
     }
}
```

Resultado da execução:

Contrariamente ao que poderia ser intuitivamente imaginado, o resultado da execução não será

1 2 3

mas

1 2 1 3

Parece estranho... O que teria acontecido? A resposta é que o filho, no seu nascimento, herda o "1" que já estava colocado no buffer de saída do pai (note que nem o caracter de retorno de linha, nem um comando para esvaziar a saída padrão foram enviados antes da criação do filho). Mais tarde, na sua morte, o buffer de saída do filho é esvaziado, e a seguinte saída de impressão será obtida: 1 2. Finalmente, o pai terminará sua execução e imprimirá por sua vez: 1 3.

Uma possível solução para o problema é mostrada no programa a seguir, através da utilização da primitiva fflush, que será detalhada no fim do capítulo.

```
/* arquivo test_fork4.c */
/* Solucao para o filho que herda uma copia do buffer nao vazio
 * de saida do pai */

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

int main()
{
  int pid ;
     printf(" 1") ;
     fflush(stdout); /* o buffer vai ser esvaziado pelo flush */
     pid=fork() ;
     if(pid==-1) /* error */
     {
          perror("impossivel de criar um filho") ;
          exit(-1) ;
     }
     else if(pid==0) /* filho */
     {
          printf(" 2") ;
          exit(0) ;
     }
     else /* pai */
     {
          wait(0) ; /* o pai aguarda a morte de seu filho */
          printf(" 3") ;
          exit(0);
     }
}
```

2.1.2. A primitiva wait()

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status)    /* espera a morte de um filho */
pid_t waitpid(pid_t pid, int *status, int options)

int *status                /* status descrevendo a morte do filho */
```

Valor de retorno: identificador do processo morto ou -1 em caso de erro.

A função wait suspende a execução do processo até a morte de seu filho. Se o filho já estiver morto no instante da chamada da primitiva (caso de um processo zumbi, abordado mais a frente), a função retorna imediatamente.

A função waitpid suspende a execução do processo até que o filho especificado pelo argumento pid tenha morrido. Se ele já estiver morto no momento da chamada, o comportamento é idêntico ao descrito anteriormente.

O valor do argumento pid pode ser:

< -1 : significando que o pai espera a morte de qualquer filho cujo o ID do grupo é igual so valor de pid;

-1 : significando que o pai espera a morte de qualquer filho;

0 : significando que o pai espera a morte de qualquer processo filho cujo ID do grupo é igual ao do processo chamado;

> 0 : significando que o pai espera a morte de um processo filho com um valor de ID exatamente igual a pid.

Se status é não nulo (NULL), wait e waitpid armazena a informação relativa a razão da morte do processo filho, sendo apontada pelo ponteiro status. Este valor pode ser avaliado com diversas macros que são listadas com o comando shell man 2 wait.

O código de retorno via status indica a morte do processo que pode ser devido uma:

- uma chamada exit(), e neste caso, o byte à direita de status vale 0, e o byte à esquerda é o parâmetro passado a exit pelo filho;
- uma recepção de um sinal fatal, e neste caso, o byte à direita de status é não nulo. Os sete primeiros bits deste byte contém o número do sinal que matou o filho.

Exemplo:

```
/* arquivo test_wait1.c */

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

int main()
{
   int pid ;
   printf("\nBom dia, eu me apresento. Sou o processo %d.\n",getpid()) ;
   printf("Estou sentindo uma coisa crescendo dentro de minha barriga...");
   printf("Sera um filho?!?!\n") ;

   if (fork() == 0) {
      printf("\tOi, eu sou %d, o filho de %d.\n",getpid(),getppid()) ;
      sleep(3) ;
      printf("\tEu sao tao jovem, e ja me sinto tao fraco!\n") ;
      printf("\tAh nao... Chegou minha hora!\n") ;
      exit(7) ;
   }
   else {
      int ret1, status1 ;
      printf("Vamos esperar que este mal-estar desapareca.\n") ;
      ret1 = wait(&status1) ;
      if ((status1&255) == 0) {
         printf("Valor de retorno do wait(): %d\n",ret1) ;
         printf("Parametro de exit(): %d\n",(status1>>8)) ;
```

```
         printf("Meu filho morreu por causa de um simples exit.\n") ;
      }
      else
         printf("Meu filho nao foi morto por um exit.\n") ;
         printf("\nSou eu ainda, o processo %d.", getpid());
         printf("\nOh nao, recomecou! Minha barriga esta crescendo
                 de novo!\n");
      if ((pid=fork()) == 0) {
         printf("\tAlo, eu sou o processo %d, o segundo filho de %d\n",
              getpid(),getppid()) ;
         sleep(3) ;
         printf("\tEu nao quero seguir o exemplo de meu irmao!\n") ;
         printf("\tNao vou morrer jovem e vou ficar num loop infinito!\n")
;
         for(;;) ;
      }
      else {
         int ret2, status2, s ;
         printf("Este aqui tambem vai ter que morrer.\n") ;
         ret2 = wait(&status2) ;
         if ((status2&255) == 0) {
            printf("O filho nao foi morto por um sinal\n") ;
         }
         else {
            printf("Valor de retorno do wait(): %d\n",ret2) ;
            s = status2&255 ;
            printf("O sinal assassino que matou meu filho foi: %d\n",s) ;
         }
      }
   }
   exit(0);
}
```

Resultado da execução:

test_wait &
[1] 29079

Bom dia, eu me apresento. Sou o processo 29079.
Estou sentindo uma coisa crescendo dentro de minha barriga...Sera um filho?!?!
Vamos esperar que este mal-estar desapareca.
    Oi, eu sou 29080, o filho de 29079.
    Eu sao tao jovem, e ja me sinto tao fraco!
    Ah nao... Chegou minha hora!
Valor de retorno do wait(): 29080
Parametro de exit(): 7
Meu filho morreu por causa de um simples exit.

Sou eu ainda, o processo 29079.
Oh nao, recomecou! Minha barriga esta crescendo de novo!
Este aqui tambem vai ter que morrer.
    Alo, eu sou o processo 29081, o segundo filho de 29079
    Eu nao quero seguir o exemplo de meu irmao!
    Nao vou morrer jovem e vou ficar num loop infinito!

```
# ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:01 -tcsh
29079  ?  S    0:00 test_wait
29081  ?  R    5:06 test_wait
29103  ?  R    0:00 ps
# kill -8 29081
# Valor de retorno do wait(): 29081
```

O sinal assassino que matou meu filho foi: 8.

[1] Done test_wait
#

O programa é lançado em background e, após o segundo filho estiver bloqueado num laço infinito, um sina será lançado para interromper sua execução através do comando shell

kill <numero-do-sinal> <pid-filho2>

Observações:

Após a criação dos filhos, o processo pai ficará bloqueado na espera de que estes morram. O primeiro filho morre pela chamada de um exit(), sendo que o parâmetro de wait() irá conter, no seu byte esquerdo, o parâmetro passado ao exit(); neste caso, este parâmetro tem valor 7.

O segundo filho morre com a recepção de um sinal, o parâmetro da primitiva wait() irá conter, nos seus 7 primeiros bits, o número do sinal (no exemplo anterior ele vale 8).

Observações relativas aos processos zumbis

Um processo pode se terminar quando seu pai não está a sua espera. Neste caso, o processo filho vai se tornar um processo denominado zumbi (zombie). Ele é neste caso identificado pelo nome <defunct> ou <zombie> ao lado do nome do processo. Seus segmentos de intruções e dados do usuário e do sistema são automaticamente suprimidos com sua morte, mas ele vai continuar ocupando a tabela de processo do kernel. Quando seu fim é esperado, ele simplesmente desaparace ao fim de sua execução.

Exemplo:

```
/* arquivo test_defunct.c */

#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
int pid ;
     printf("Eu sou %d e eu vou criar um filho\n",getpid()) ;
     printf("Vou em bloquear em seguida num loop infinito\n") ;
     pid = fork() ;
     if(pid == -1) /* erro */
```

```
     {
          perror("E impossivel criar um filho") ;
          exit(-1) ;
     }
     else if(pid == 0) /* filho */
     {
          printf("Eu sou %d o filho e estou vivo\n",getpid()) ;
          sleep(10) ;
          printf("Vou me suicidar para manter minha consciencia
                  tranquila\n") ;
          exit(0) ;
     }
     else /* pai */
     {
          for(;;) ; /* pai bloqueado em loop infinito */
     }
}
```

Resultado da execução:

Lançando a execução em background, tem-se o seguinte resultado:

test_defunct &
Vou em bloquear em seguida num loop infinito
Eu sou 29733 o filho e estou vivo
#
Fazendo-se um ps, obtém-se:

```
# ps
PID TTY STAT  TIME COMMAND
28300  ?  S    0:02 -tcsh
29732  ?  R    0:01 test_defunct
29733  ?  S    0:00 test_defunct
29753  ?  R    0:00 ps
```

Após os 10 segundos, o processo filho anuncia sua morte na tela:

Vou me suicidar para manter minha consciência tranqüila

Refazendo-se um ps, obtém-se:

```
# ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:02 -tcsh
29732  ?  R    1:35 test_defunct
29733  ?  Z    0:00 (test_defunct <zombie>)
29735  ?  R    0:00 ps
```

Tente matar o processo filho usando a primitiva kill:

```
# kill -INT 29733
# ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:02 -tcsh
29732  ?  R    2:17 test_defunct
29733  ?  Z    0:00 (test_defunct <zombie>)
29736  ?  R    0:00 ps
```

Não funciona, não é mesmo! Óbvio, ele é um zumbi!!! Note o resultado do ps para se certificar. Tente agora matar o pai.

```
# kill -INT 29732
[1]    Interrupt                     test_defunct
# ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:02 -tcsh
29750  ?  R    0:00 ps
```

Finalmente o processo pai, junto com seu filho zumbi foram finalizados.

2.2. Primitiva exit()

```
#include <unistd.h>

void _exit(int status);  /* terminacao do processo */
int status  /* valor retornado ao processo pai como status
             * saida do processo filho */
```

Valor de retorno: única primitiva que não retorna.

Todos os descritores de arquivos abertos são automaticamente fechados. Quando um processo faz exit, todos os seus processos filho são herdados pelo processo init de ID igual a 1, e um sinal SIGCHLD é automaticamente enviado ao seu processo pai.

Por convenção, um código de retorno igual a 0 significa que o processo terminou normalmente (veja o valor de retorno dos procedimentos main() dos programas de exemplo. Um código de retorno não nulo (em geral -1 ou 1) indicará entretanto a ocorrência de um erro de execução.

2.3. As Primitivas exec()

```
#include <unistd.h>

extern char **environ;

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[]);
```

As primitivas exec() constituem na verdade uma família de funções (execl, execlp, execle, execv, execvp) que permitem o lançamento da execução de um programa externo ao processo. Não existe a criação efetiva de um novo processo, mas simplesmente uma substituição do programa de execução.

Existem seis primitivas na família, as quais podem ser divididas em dois grupos: os execl(), para o qual o número de argumentos do programa lançado é conhecido; e os execv(), para o

qual esse número é desconhecido. Em outras palavras, estes grupos de primitivas se diferenciam pelo número de parâmetros passados.

O parâmetro inicial destas funções é o caminho do arquivo a ser executado.

Os parâmetros char arg, ... para as funções execl, execlp e execle podem ser vistos como uma lista de argumentos do tipo arg0, arg1, ..., argn passadas para um programa em linha de comando. Esses parâmetros descrevem uma lista de um ou mais ponteiros para strings não-nulas que representam a lista de argumentos para o programa executado.

As funções execv e execvp fornecem um vetor de ponteiros para strings não-nulas que representam a lista de argumentos para o programa executado.

Para ambos os casos, assume-se, por convenção, que o primeiro argumento vai apontar para o arquivo associado ao nome do programa sendo executado. A lista de argumento deve ser terminada pelo ponteiro NULL.

A função execle também especifica o ambiente do processo executado após o ponteiro NULL da lista de parâmetros ou o ponteiro para o vetor argv com um parâmetro adicional. Este parâmetro adicional é um vetor de ponteiros para strings não-nulas que deve também ser finalizado por um ponteiro NULL. As outras funções consideram o ambiente para o novo processo como sendo igual ao do processo atualmente em execução.

Valor de retorno: Se alguma das funções retorna, um erro terá ocorrido. O valor de retorno é -1 neste caso, e a variel global errno será setada para indicar o erro.

Na chamada de uma função exec(), existe um recobrimento do segmento de instruções do processo que chama a função. Desta forma, não existe retorno de um exec() cuja execução seja correta (o endereço de retorno desaparece). Em outras palavras, o processo que chama a função exec() morre.

O código do processo que chama uma função exec() será sempre destruído, e desta forma, não existe muito sentido em utilizá-la sem que ela esteja associada a uma primitiva fork().

Exemplo:

```
/* arquivo test_exec.c */

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

int main()
{
      execl("/bin/ls","ls","test_exec.c",NULL) ;
      printf ("Eu ainda nao estou morto\n") ;
      exit(0);
}
```

Resultado da execução:

```
# test_exec
test_exec.c
```

O comando ls é executado, mas o printf não. Isto mostra que o processo não retorna após a execução do execl.

O exemplo seguinte mostra a utilidade do fork neste caso.

Exemplo:

```
/* arquivo test_exec_fork.c */

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

int main()
{
     if ( fork()==0 ) execl( "/bin/ls","ls","test_exec.c",NULL) ;
     else {
          sleep(2) ;  /* espera o fim de ls para executar o printf() */
          printf ("Eu sou o pai e finalmente posso continuar\n") ;
     }
     exit(0);
}
```

Resultado da execução:

```
# test_exec_fork
test_exec.c
Eu sou o pai e finalmente posso continuar
```

Neste caso, o filho morre após a execução do ls, e o pai continuará a viver, executando então o printf.

2.3.1. Comportamento em relação aos descritores abertos

A princípio, os descritores de arquivos abertos antes da chamada exec() continuama abertos, exceto se for determinado o contrário (via primitiva fcntl(). Um dos efeitos do recobrimento dos segmentos dos processo numa chamada exec é a destruição do buffer associado ao arquivo na região usuário, e portanto a perda de informações contidas por ele. Para contornar o problema, deve-se forçar o esvaziamento completo do buffer antes da chamada exec de forma a não perder seus dados, utilizando-se para isso a função flush().

Exemplo 1:

```
/* arquivo test_buffer1.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
     printf("Voce nao vai conseguir ler este texto") ;
     execl("/bin/ls","ls","test_buffer1.c",NULL) ;
     exit(0);
}
```

Resultado da execução:
test_buffer1
test_buffer1.c

A mensagem não é impressa, pois o buffer de saída não foi esvaziado antes de ser destruído pela chamada exec.

Exemplo 2:

```
/* arquivo test_buffer2.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
     printf("Voce nao vai conseguir ler este texto\n") ;
     execl("/bin/ls","ls","test_buffer1.c",NULL) ;
     exit(0);
}
```

Resultado da execução:
test_buffer2
Voce nao vai conseguir ler este texto
test_buffer1.c

A mensagem agora é impressa, pois o caracter \n esvazia o buffer de saída e retorna à linha.

Exemplo 3:

```
/* arquivo test_buffer3.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
     printf("Voce nao vai conseguir ler este texto\n") ;
     fflush(stdout) ;
     execl("/bin/ls","ls","test_buffer1.c",NULL) ;
     exit(0);
}
```

Resultado da execução:

test_buffer3
Voce nao vai conseguir ler este texto
test_buffer1.c

O resultado é semelhante ao anterior, só que desta vez, o buffer é esvaziado através da primitiva fflush.

Observações: stdout corresponde à saída padrão em UNIX, que é neste caso a tela. Note também que comandos internos do shell não podem ser executados através de exec(). Por exemplo, não teria nenhum efeito a utilização de um processo filho associado a exec() para a

execução do comando shell cd. Isto porque o atributo mudado no processo filho (no caso o diretório corrente) não pode ser remontado ao pai uma vez que o filho morrerá imediatamente após a execução do exec. Verifique a afirmação através do exemplo a seguir:

Exemplo:

```
/* arquivo test_cd.c */
/* a mudanca de diretorio so e valida */
/* durante o tempo de execucao do processo */

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

int main()
{
     if(chdir("..")==-1) /* retorno ao diretorio predecente */
       {  perror("impossivel de achar o diretorio especificado") ;
       exit(-1);
       }
     /* sera executado um pwd que vai matar o processo */
     /* e que vai retornar o diretorio corrente onde ele esta */
     if(execl("/bin/pwd","pwd",NULL)==-1)
     {
          perror("impossivel de executar o pwd") ;
          exit(-1) ;
     }
     exit(0);
}
```

Faça um pwd no diretorio corrente. Lance o programa test_cd e verifique então que o diretório foi momentaneamente alterado durante a execução deste. Espere o fim do programa e execute novamente um pwd. O diretório corrente não foi alterado como era de se esperar.

2.4. Primitiva system()

```
#include <stdlib.h>

int system (const char * string)
```

Esta primitiva executa um comando especificado por string, chamando o programa /bin/sh/ -c string, retornando após o comando ter sido executado. Durante a execução do comando, SIGCHLD será bloqueado e SIGINT e SIGQUIT serão ignorados (estes sinais serão detalhados no próximo capítulo).

Valor de retorno: O código de retorno do comando. Em caso de erro, retorna 127 se a chamada da primitiva execve() falhar, ou -1 se algum outro erro ocorrer.

Bibliografia

SANTOS, Celso Alberto Saibel. **Programação em Tempo Real**: Módulo II - Comunicação InterProcessos. UFRN, Laboratório de Engenharia de Computação e Automação. http://www.dca.ufrn.br/~adelardo/cursos/DCA409/all.html. Acessado em:16/03/2005.