Laboratório de Pesquisa em Redes e Multimídia

SVCs para Controle de Processos no Unix

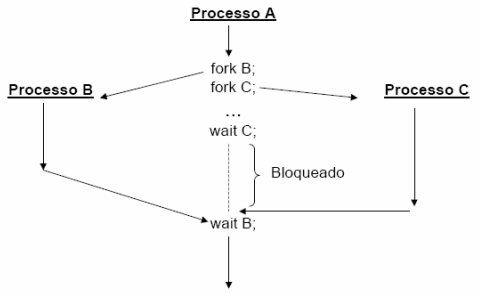
Aula 9

Profa. Patrícia Dockhorn Costa

Universidade Federal do Espírito Santo
Departamento de Informática

Criação de Processos (1)

- A maioria dos sistemas operacionais usa um mecanismo de *spawn* para criar um novo processo a partir de um outro executável.



Criação de Processos (2)

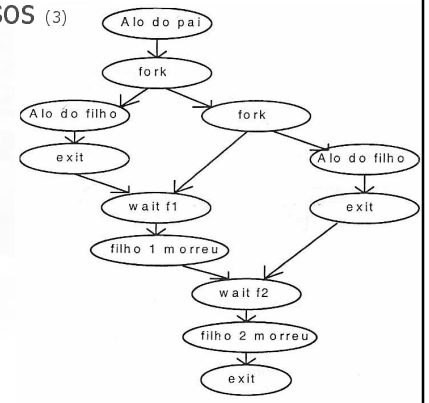
Exemplo

Não é exatamente assim no Unix!

```
/* Programa principal */
main()
{
        int f1;      /* Identifica processo filho 1*/
        int f2;      /* Identifica processo filho 2*/

        printf("Alo do pai\n");

        f1 = fork( codigo_do_filho );          /* Cria filho 1 */

        f2 = fork( codigo_do_filho );          /* Cria filho 2 */

        wait( f1);
        printf("Filho 1 morreu\n");

        wait( f2);
        printf("Filho 2 morreu\n");

        exit();
}

/* Funcao executada pelos dois processos filhos */
codigo_do_filho()
{
        printf("Alo do filho\n");
        exit();
}
```


Criação de Processos (3)

Exemplo
Grafo de Precedência



Criação de Processos (4)

Exercício: Desenhar o grafo de precedência

```
main()
{
        int f1;      /* Identifica processo filho 1*/
        int f2;      /* Identifica processo filho 2*/

        printf("Alo do pai\n");

        f1 = fork( codigo_do_filho );          /* Cria filho 1 */
        wait( f1);
        printf("Filho 1 morreu\n");

        f2 = fork( codigo_do_filho );          /* Cria filho 2 */
        wait( f2);
        printf("Filho 2 morreu\n");

        exit();
}

/* Funcao executada pelos dois processos filho */
codigo_do_filho()
{
        printf("Alo do filho\n");
        exit();
}
```

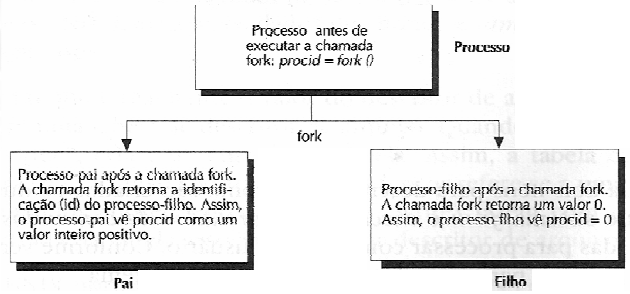

Criação de Processos no UNIX

- No Unix, são usadas duas funções distintas:
  - `fork()`
  - `exec()`
- `fork()` cria um processo filho idêntico ao pai, exceto pelo PID, PPID, e alguns recursos, tais como:
  - estatísticas do processo e sinais pendentes...
- `exec()`carrega e executa um novo programa.
- A sincronização é feita com `wait()` , que bloqueia o processo que a executa até que um processo filho (criado pelo que executou a operação wait) termine.

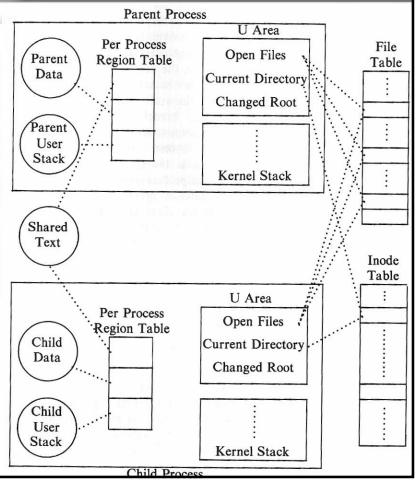
A SVC *fork()* do Unix (1)

- Duplica o processo que executa a chamada e os dois executam o mesmo código;
- Cada processo tem seu próprio espaço de endereçamento, com **cópia** de todas as variáveis, que são independentes em relação às variáveis do outro processo;
- O processo filho herda do pai atributos como
  - Privilégios e prioridade de escalonamento. Herda também alguns recursos, tais como arquivos abertos e *devices*.
- Ambos executam a instrução seguinte à chamada de sistema;
- *id*, no retorno da chamada, contém, no processo pai, o identificador do processo filho criado;
- Para o processo filho o valor da variável *id* será zero;
- Pode-se selecionar o trecho de código que será executado pelos processos com o comando *if*;

A SVC *fork()* do Unix (2)



A SVC *fork()* do Unix (3)



Copy-on-Write

- Como alternativa a significante ineficiência do `fork()`, no Linux o `fork()` é implementado usando uma técnica chamada *copy-on-write* (COW).
- Essa técnica atrasa ou evita a cópia dos dados.
  - Ao invés de copiar o espaço de endereçamento do processo pai, ambos podem compartilhar uma única cópia somente de leitura.
  - Se uma escrita é feita, uma duplicação é realizada e cada processo recebe uma cópia.
  - Conseqüentemente, a duplicação é feita apenas quando necessário, economizando tempo e espaço.

SVCs para Identificação do Processo no UNIX

- Process id
  - pid_t getpid(void)
  - pid_t getppid(void)
- User id
  - uid_t getuid(void)
- Group id
  - gid_t getgid(void)

Exemplo 1:

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

int main (void) {
   printf("I am process %ld\n", (long)getpid());
   printf("My parent is %ld\n", (long)getppid());

   printf("My user ID is       %5ld\n", (long)getuid());
   printf("My group ID is      %5ld\n", (long)getgid());

   return 0;
}
```

Exemplo 2: *simplefork.c*

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

int main(void) {
   int x;

   x = 0;
   fork();
   x = 1;
   printf("I am process %ld and my x is %d\n",
          (long)getpid(), x);
   return 0;
}
```


Comando ps

```
$ ps -l
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   508 11436  9181  0  70   0    -   542 rt_sig ttyp2    00:00:00 tcsh
000 S   508 11520 11436  0  60   0    -  2386 do_sel ttyp2    00:00:00 xemacs
000 S   508 11620 11436  0  60   0    -  1061 do_sel ttyp2    00:00:00 xterm
000 R   508 11624 11436  0  73   0    -   637 -      ttyp2    00:00:00 ps
```


Estrutura Geral do *fork()*

```
pid=fork();
if(pid < 0) {
      /* falha do fork */
}
else if (pid > 0) {
      /* código do pai */
}
else { //pid == 0
      /* código do filho */
}
```

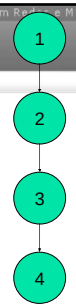

Exemplo 3: *twoprocs.c*

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

int main(void) {
   pid_t childpid;

   childpid = fork();
   if (childpid == -1) {  //error
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)                                  /* child code */
      printf("I am child %ld\n", (long)getpid());
   else                                                /* parent code */
      printf("I am parent %ld\n", (long)getpid());
   return 0;
}
```

Exercício 1: Montar o Grafo de Precedência

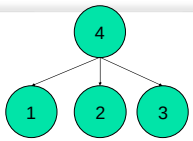
```
c2 = 0;
c1 = fork();                /* fork number 1 */
if (c1 == 0)
   c2 = fork();             /* fork number 2 */
fork();                     /* fork number 3 */
if (c2 > 0)
   fork();                  /* fork number 4 */
exit();
```


Exemplo 5: *simplechain.c*




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

int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   ...

   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (childpid = fork())  //only the parent enters
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```


Exemplo 6: *simplefan.c*




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

int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   /* check for valid number of command-line arguments */
   ...
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if ((childpid = fork()) <= 0)  //only the child (or error) enters
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```


A SVC *exit()* do Unix (1)

- void exit(code);
  - O argumento *code* é um número de 0 a 255, escolhido pela aplicação e que será passado para o processo pai na variável status
- A chamada *exit* termina o processo e portanto *exit* nunca retorna
  - A memória alocada ao segmento físico de dados é liberada
  - Todos os arquivos abertos são fechados
  - É enviado um sinal para o pai do processo
  - Se o pai do processo estiver bloqueado esperando o filho, ele é acordado
  - Se o processo tiver filhos, eles serão "adotados" pelo processo *init*
- A rotina faz o *scheduler* ser invocado
- Processo *Zombie*
  - Kernel mantém informações do filho até o pai executar *wait*
  - Para processos filhos herdados pelo *init*, este último toma conta dos processos e executa *wait* para eliminar *zombies* periodicamente

A SVC *wait()* do Unix (1)

- Como processos descobrem se seu filho terminou?
  - Chamada `wait()` feita pelo pai – retorna o PID do processo filho que terminou execução
- Normalmente, pai executando `wait()` é bloqueado até filho terminar
  - Se não existir filhos no estado zombie, esperar que um filho termine
- `waitpid()`
  - Para esperar um filho específico
  - Também pode esperar por qualquer filho
    - `waitpid(-1,...,...)` ~ `wait(...)`

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

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```


A SVC *wait()* do Unix (2)

- Em caso de erro
  - retorna -1
  - Seta a variável global *errno*
    - ECHILD: não existem filhos para termitar (*wait*), ou pid não existe (*waitpid*)
    - **EINTR: função foi interrompida por um sinal**
    - EINVAL: o parâmetro *options* do *waitpid* estava inválido
- Solução para que um processo pai continue esperando pelo término de um processo filho, mesmo que o pai seja interrompido por um sinal:

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

pid_t r_wait(int *stat_loc) {
   int retval;

   while (((retval = wait(stat_loc)) == -1) && (errno == EINTR)) ;
   return retval;
}
```


A SVC *wait()* do Unix (3)

- A opção WNOHANG na chamada *waitpid* permite que um processo pai verifique se um filho terminou, sem que este primeiro bloqueie caso filho não tenha terminado
  - Neste caso *waitpid* retorna 0

Exemplo 7: *fanwait.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "restart.h"

int main(int argc, char *argv[]) {
   pid_t childpid;
   int i, n;

   ...
   for (i = 1; i < n; i++)
      if ((childpid = fork()) <= 0)
         break;

   while(r_wait(NULL) > 0) ;    /* wait for all of your children */
   fprintf(stderr, "i:%d proc.ID:%ld parentID:%ld  child ID:%ld\n",
            i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

Exercício 3

- Referente ao programa `fanwait.c`
  - Explique o que acontece quando coloca o `printf` antes do `while` ?
  - Explique o que acontece se substituirmos a chamada `r_wait` por `wait(NULL)` simplesmente ?

Valores de *stat_loc* (1)

- O argumento *stat_loc*: ponteiro p/a uma variável inteira
- Um filho sempre retorna seu status ao chamar *exit* ou ao retornar do *main*
  - 0: indica EXIT_SUCCESS
  - outro valor: indica EXIT_FAILURE

MACROS
WIFEXITED(status) – permite determinar se o processo filho terminou normalmente **WEXITSTATUS(status) –** retorna o código de saída do processo filho
WIFSIGNALED(status)– permite determinar se o processo filho terminou devido a um sinal **WTERMSIG(status)** – permite obter o número do sinal que provocou a finalização do processo filho
WIFSTOPPED(status) – permite determinar se o processo filho que provocou o retorno se encontra congelado (stopped) **WSTOPSIG(status)** – permite obter o número do sinal que provocou o congelamento do processo filho

Valores de *stat_loc* (2)

- Exemplo

```
q = wait(&status);

if (q == -1) {
    /* Erro */
} else if (q > 0) {
    /* q -> pid do processo que terminou */

    if (WIFEXITED(status)) {
        /* Processo q terminou normalmente */
        /* Codigo de saida = WEXITSTATUS(status) */
    } else {
        /* Processo q terminou anormalmente! */
    }
}
```


A SVC *exec()* do Unix (1)

- Quando um processo invoca uma das funções *exec*, ele é completamente substituído por um novo programa
  - Substitui o processo corrente (os seus segmentos text, data, heap e stack) por um novo programa carregado do disco
  - O novo programa começa a sua execução a partir da função main()
- O identificador do processo não é alterado
  - De fato, nenhum novo processo é criado
- Valor de retorno
  - **Sucesso - não retorna**
  - Erro - retorna o valor -1 e seta a variável errno com o código específico do erro
- Quando um processo executando um programa A quer executar outro programa B:
  - Primeiramente ele deve criar um novo processo usando fork()
  - Em seguida, o processo recém criado deve substituir todo o seu programa pelo programa B, chamando uma das primitivas da família *exec*

Exemplo de Uso: *fork() – exec()*

```
#incude <stdio.h>
main( int argc, char *argv[] ) {
      int pid;
      /* fork a child process */
      pid = fork();
      /* check fork() return code */
      if ( pid < 0 ) {
            /* some error occurred */
            fprintf( stderr, "Fork failed!\n" );
            exit( -1 );
       } else if ( pid == 0 ) {
            /* this is the child process */
            execl( "/bin/ls", "ls", NULL ); /* morph into "ls" */
       } else {
            /* this is the parent process. Wait for child to complete */
            wait( NULL );
            printf( "Child completed -- parent now exiting.\n" );
            exit( 0 );
       }
}
```


A SVC *exec()* do Unix (2)

- O processo que executou a função *exec* mantém as seguintes informações
  - pid e o ppid
  - user, group, session id
  - Mascara de sinais
  - Alarmes
  - Terminal de controle
  - Diretórios raiz e corrente
  - Informações sobre arquivos abertos
  - Limites de uso de recursos
  - Estatísticas e informações de *accounting*

attribute	relevant library function
process ID	`getpid`
parent process ID	`getppid`
process group ID	`getpgid`
session ID	`getsid`
real user ID	`getuid`
real group ID	`getgid`
supplementary group IDs	`getgroups`
time left on an alarm signal	`alarm`
current working directory	`getcwd`
root directory	
file mode creation mask	`umask`
file size limit*	`ulimit`
process signal mask	`sigprocmask`
pending signals	`sigpending`
time used so far	`times`
resource limits*	`getrlimit, setrlimit`
controlling terminal*	`open, tcgetpgrp`
interval timers*	`ualarm`
nice value*	`nice`
`semadj` values*	`semop`

Funções exec (1)

- int execl(const char *pathname, const char *arg0, ...)
- int execv(const char *pathname, char *const argv[])
- int execlp(const char *filename, const char *arg0, ...)
- int execvp(const char *filename, char *const argv[])
- int execle(const char *pathname, const char *arg0, ..., char *const envp[])
- int execve(const char *pathname, char *const argv[], char *const envp[])

System Call	*Argument Format*	*Environment Passing*	*PATH Search?*
`execl`	list	auto	no
`execv`	array	auto	no
`execle`	list	manual	no
`execve`	array	manual	no
`execlp`	list	auto	yes
`execvp`	array	auto	yes

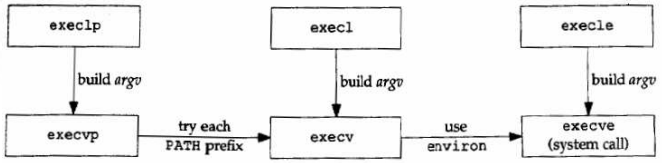
Funções exec (2)

- Sufixos
  - l
    - lista de argumentos (terminada com NULL)
  - v
    - argumentos num array de strings (terminado com NULL)
  - e
    - variáveis de ambiente num array de strings (terminado com NULL)
  - p
    - procura executável nos diretórios definidos na variável de ambiente PATH (echo $PATH)

System Call	*Argument Format*	*Environment Passing*	*PATH Search?*
`execl`	list	auto	no
`execv`	array	auto	no
`execle`	list	manual	no
`execve`	array	manual	no
`execlp`	list	auto	yes
`execvp`	array	auto	yes

Funções exec (3)

- Relações entre as funções exec



Funções exec (4)

- Se alguma das funções retorna, um erro terá ocorrido
- Valores possíveis na variável global *errno*

E2BIG	Lista de argumentos muito longa
EACCES	Acesso negado
EINVAL	Sistema não pode executar o arquivo
ENAMETOOLONG	Nome de arquivo muito longo
ENOENT	Arquivo ou diretório não encontrado
ENOEXEC	Erro no formato de exec
ENOTDIR	Não é um diretório

Funções exec (4)

- Executar os seguintes comandos
  - ls -l /etc
  - ls -l /etc/s*.conf

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

int main(void)
{
    execlp("ls", "ls", "-l", "/etc", NULL);

    perror("execlp");

    return EXIT_FAILURE;
}
```

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

int main(void)
{
    execlp("ls", "ls", "-l", "/etc/s*.conf", NULL);

    perror("execlp");

    return EXIT_FAILURE;
}
```


Funções exec (5)

- Um programa que cria um processo filho para executar "ls -l"

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int  main(void) {
   pid_t childpid;

   childpid = fork();
   if (childpid == -1)  {
       perror("Failed to fork");
       return 1;
   }
   if (childpid == 0) {                              /* child code */
       execl("/bin/ls", "ls", "-l", NULL);
       perror("Child failed to exec ls");
       return 1;
   }
   if (childpid != wait(NULL)) {                     /* parent code */
       perror("Parent failed to wait due to signal or error");
       return 1;
   }
   return 0;
}
```

Funções exec (5)

- Um programa que cria um processo filho para executar um comando (com ou sem parâmetros) passado como parâmetro

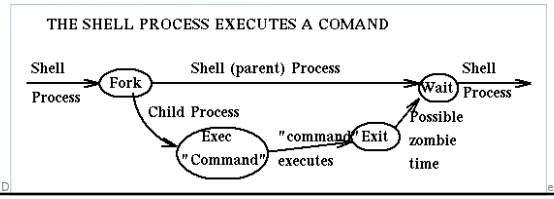
execcmd.c

```
...
int main(int argc, char *argv[]) {
   pid_t childpid;

   if (argc < 2){      /* check for valid number of command-line arguments */
      fprintf (stderr, "Usage: %s command arg1 arg2 ...\n", argv[0]);
      return 1;
   }
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0) {                                    /* child code */
      execvp(argv[1], &argv[1]);
      perror("Child failed to execvp the command");
      return 1;
   }
   if (childpid != r_wait(NULL)) {                         /* parent code */
      perror("Parent failed to wait");
      return 1;
   }
   return 0;
}
```

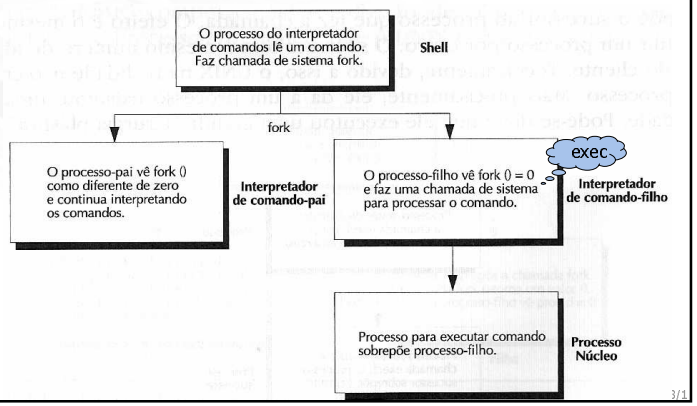

O shell do UNIX (1)

- Quando o interpretador de comandos UNIX interpreta comandos, ele chama fork e execl

```
...
Lê comando para o interpretador de comandos
...
If (fork()==0)
   exec...(command, lista_arg ...)
```



O shell do UNIX (2)



Processos de backgroud e deamons (1)

- Quando usamos o shell p/ executar o comando "ls -l", ele espera o processo filho criado terminar par continuar executando
  - Eventualmente, quando a entrada e a saída padrão vêm de um terminal, o usuário pode terminar o comando em execução entrando um caractere de interrupção (default é Ctrl-c)
- Para executar um comando em background, usa-se &
  - "ls -l &"
- **Ctrl-c não afeta processos em background**
- *Daemons* ("***d**isk **a**nd **e**xecution **mon**itor*")
  - Processos em background que sempre estão executando
  - Pageout, in.rlogind, mail, printer, finger

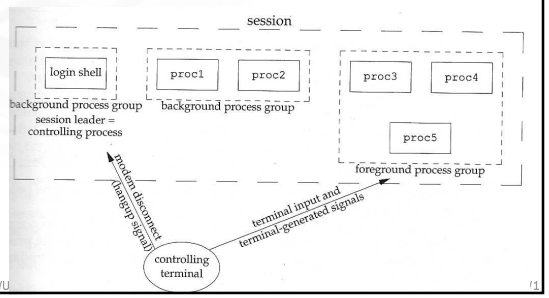
Processos de backgroud e deamons (2)

```
int makeargv(const char *s, const char *delimiters, char ***argvp);

int main(int argc, char *argv[]) {
   pid_t childpid;
   char delim[] = " \t";
   char **myargv;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s string\n", argv[0]);
      return 1;
   }
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0) {                    /* child becomes a background process */
     if (setsid() == -1)
        perror("Child failed to become a session leader");
     else if (makeargv(argv[1], delim, &myargv) == -1)
        fprintf(stderr, "Child failed to construct argument array\n");
     else {
        execvp(myargv[0], &myargv[0]);
        perror("Child failed to exec command");
     }
     return 1;                                    /* child should never return */
   }
   return 0;                                              /* parent exits */
}
```

Uso de *setsid* para que o processo pertença a uma outra sessão , e a um outro grupo, se tornando de *background*

Sessões e grupos de processos (1)

- Uma sessão é um conjunto de grupos de processos
- Cada sessão pode ter
  - um único terminal controlador
  - no máximo 1 grupo de processos de *foregroud*
  - *n* grupos de processos de *background*



Sessões e grupos de processos (2)

- Quando um processo chama a função

```
pid_t setsid(void)
```

  - O processo torna-se um líder de uma nova sessão e líder de um novo grupo
  - Se o processo já for lider de grupo, a chamada retorna erro
  - Usada para tornar um processo de *foreground* em *background*

Resumo SVCs: Processos

- **fork**: cria um novo processo que é uma cópia do processo pai. O processo criador e o processo filho continuam em paralelo, e executam a instrução seguinte à chamada de sistema.
- **wait**: suspende a execução do processo corrente até que um filho termine. Se um filho terminou antes desta chamada de sistema(estado *zombie*), os recursos do filho são liberados e o processo não fica bloqueado, retornando imediatamente.
- **exit**: termina o processo corrente. Os filhos, se existirem, são herdados pelo processo *init* e o processo pai é sinalizado.
- **exec**: executa um programa, substituindo a imagem do processo corrente pela imagem de um novo processo, identificado pelo nome de um arquivo executável, passado como argumento.
- **kill**: usada para enviar um sinal para um processo ou grupo de processos. O sinal pode indicar a morte do processo.
- **sleep**: suspende o processo pelo tempo especificado como argumento.