Laboratório de Pesquisa em Redes e Multimídia

# SVCs para Controle de Processos no Unix (cont.)

Universidade Federal do Espírito Santo
Departamento de Informática

Sistemas Operacionais

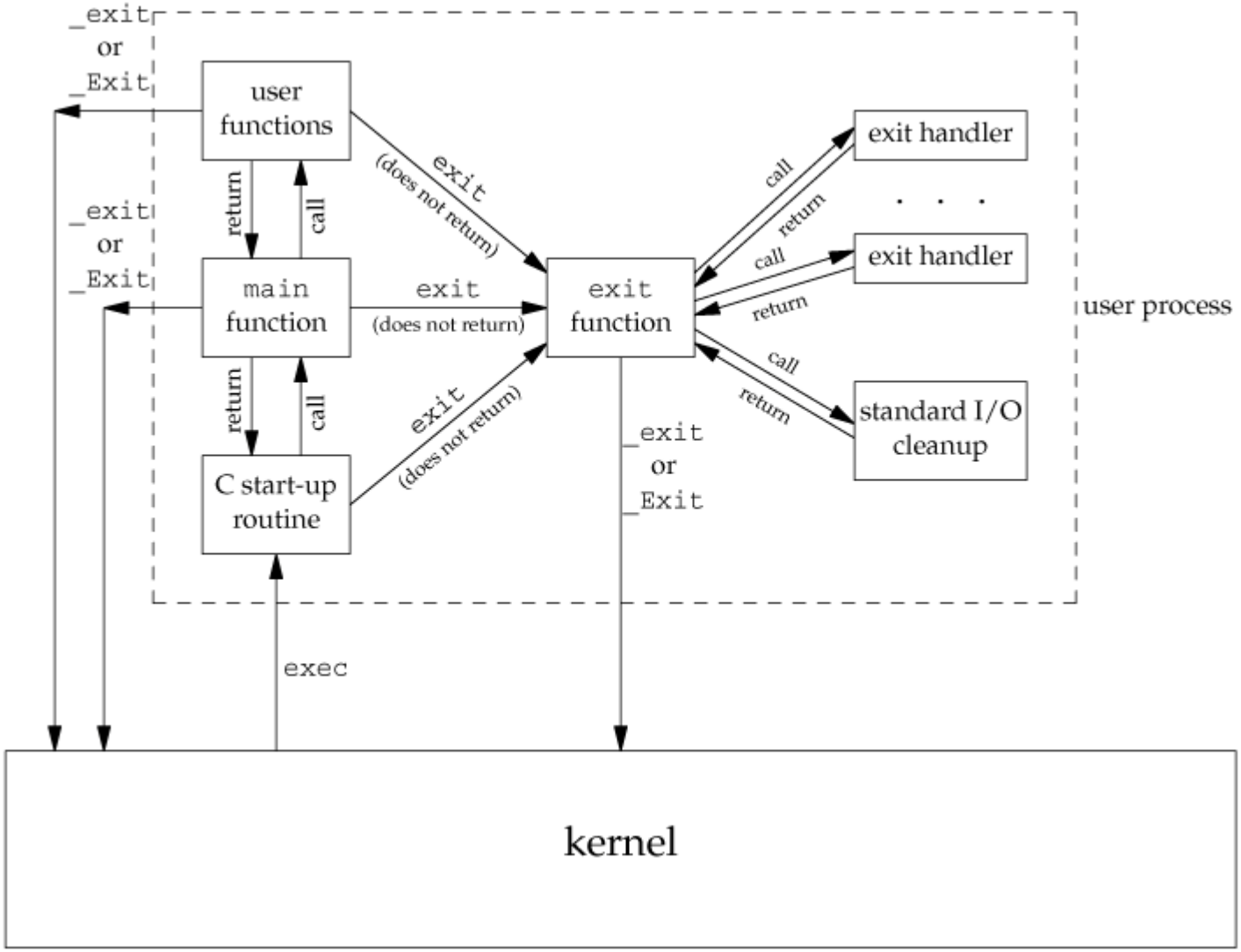
# Término de Processos no Unix

- Um processo pode terminar normalmente ou anormalmente nas seguintes condições:
- Normal:
  - Executa `return` na função `main()`, o que é equivalente à chamar `exit()`;
  - Invoca diretamente a função `exit()`da biblioteca C;
  - Invoca diretamente o serviço do sistema `_exit()`.
- Anormal:
  - Invoca o função `abort()`;
  - Recebe sinais de terminação gerados pelo próprio processo, ou por outro processo, ou ainda pelo Sistema Operacional.
- A função `abort()`
  - Destina-se a terminar o processo em condições de erro e pertence à biblioteca padrão do C.
  - Em Unix, a função `abort()` envia ao próprio processo o sinal `SIGABRT`, que tem como conseqüência terminar o processo. Esta terminação deve tentar fechar todos os arquivos abertos.

# A Chamada exit()

- `void exit(code)`
  - O argumento `code` é um número de 0 a 255, escolhido pela aplicação e que será passado para o processo pai via parâmetro `status` da função `wait()`

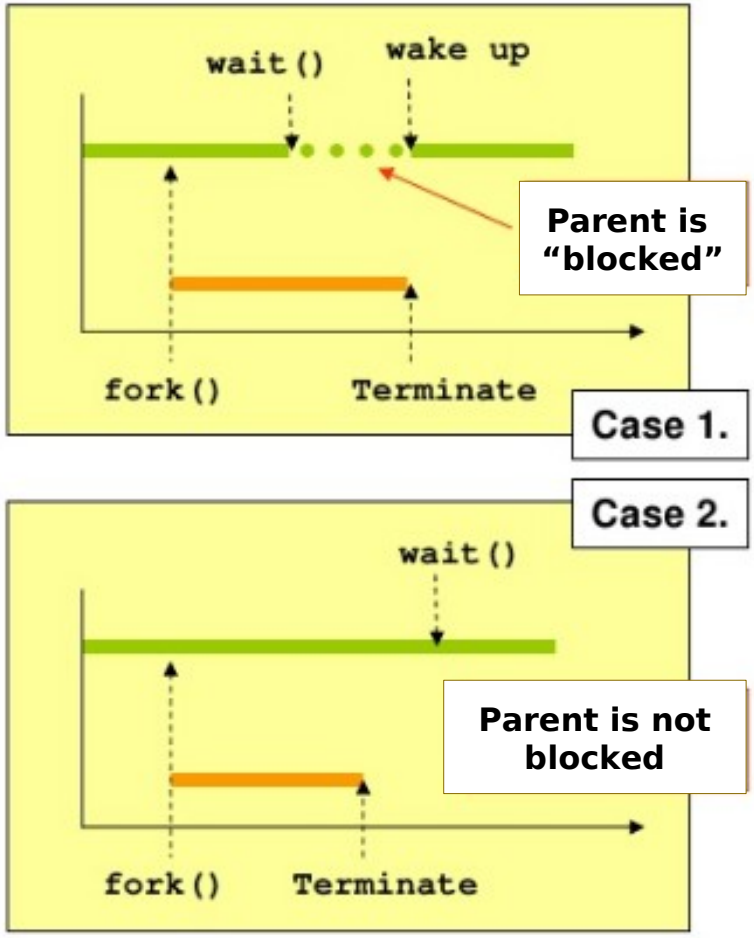
- A chamada `exit()` termina o processo; portanto, `exit()` nunca retorna
  - Chama todos os *exit handlers* que foram registrados na função *atexit()*.
  - A memória alocada ao segmento físico de dados é liberada.
  - Todos os arquivos abertos são fechados.
  - É enviado um sinal para o pai do processo. Se este estiver bloqueado esperando o filho, ele é acordado.
  - Se o processo que invocou o `exit()` tiver filhos, esses serão "adotados" pelo processo `init`.
  - Faz o escalonador ser invocado.


_exit
or
_Exit
user
functions
exit
(does not return)
exit handler
. . .
exit handler
return
call
_exit
or
_Exit
main
function
exit
(does not return)
exit
function
user process
return
call
exit
(does not return)
standard I/O
cleanup
C start-up
routine
_exit
or
_Exit
exec
kernel

# As Chamadas wait() e waitpid()

- São usadas para esperar por mudanças de estado nos filhos do processo chamador e obter informações sobre aqueles filhos cujos estados tenham sido alterados.
  - Ex: quando um processo termina (executando → terminado) o kernel notifica o seu pai enviando-lhe o sinal SIGCHLD.
- Considera-se uma alteração de estado:
  - o término de execução de um filho (exit);
  - o filho foi parado devido a um sinal (CTRL-z);
  - o filho retornou à execução devido a um sinal (SIGCONT).
- Se o filho já teve o seu estado alterado no momento da chamada, elas retornam imediatamente; caso contrário, o processo chamador é bloqueado até que ocorra uma mudança de estado do filho ou então um "*signal handler*" interrompa a chamada.

# As Chamadas wait() e waitpid() (cont.)



# As Chamadas wait() e waitpid() (cont.)

- Um processo pode esperar que seu filho termine e, então, aceitar o seu código de terminação, executando uma das seguintes funções:
  - **wait(int *status):** 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), a função retorna imediatamente.
  - **waitpid(pid_t pid, int *status, int options):** 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.

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

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

# As Chamadas wait() e waitpid()

- Em resumo, um processo que invoque wait() ou waitpid() pode:
  - bloquear - se nenhum dos seus filhos ainda não tiver terminado;
  - retornar imediatamente com o código de terminação de um filho - se um filho tiver terminado e estiver à espera de retornar o seu código de terminação (filho zombie).
  - retornar imediatamente com um erro - se não tiver filhos.
- Se wait() ou waitpid() retornam devido ao *status* de um filho ter sido reportado, então elas retornam o PID daquele filho.

# As Chamadas wait() e waitpid() (cont.)

- Diferenças entre wait() e waitpid():
  - wait() bloqueia o processo que o invoca até que um filho qualquer termine (o primeiro filho a terminar desbloqueia o processo pai);
  - waitpid() espera um filho específico morrer (a não ser que seja passado o valor -1)
  - waitpid() tem uma opção que impede o bloqueio do processo chamador (útil quando se quer apenas obter o código de terminação do filho);

# As Chamadas wait() e waitpid() (cont.)

- O argumento *pid* de waitpid() pode ser:
  - > 0 : espera pelo filho com o *pid* indicado;
  - -1 : espera por um filho qualquer (= wait());
  - 0 : espera por um filho qualquer do mesmo *process group*
  - < -1 : espera por um filho qualquer cujo *process group ID* seja igual a |*pid*|.
- waitpid() retorna um erro (valor de retorno = -1) se:
  - o processo especificado não existir;
  - o processo especificado não for filho do processo que o invocou;
  - o grupo de processos não existir.

# As Chamadas wait() e waitpid() (cont.)

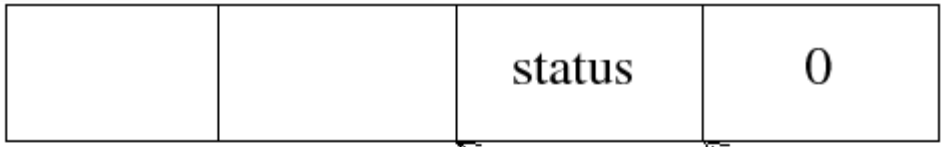
- Se um erro ocorre as funções retornam -1 e **setam a variável global *errno*.**
- Os erros mandatórios para wait() e waitpid() são:
  - ECHILD: não existem filhos para terminar (*wait*), ou pid não existe (*waitpid*)
  - EINTR: função foi interrompida por um sinal
  - **EINVAL: o parâmetro *options* do *waitpid* estava inválido**

# Valores de *status*

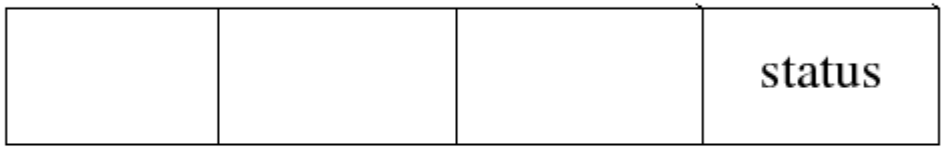
- O argumento *status* de waitpid() pode ser NULL ou apontar para um inteiro. No caso de status ser ≠ NULL, o código de terminação do processo que finalizou é guardado na posição indicada por *status.* No caso de ser = NULL, este código de terminação é ignorado.
- A morte do processo pode ser devido a:
  - 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, e neste caso, o byte à direita de *status* é não nulo e os sete primeiros bits deste byte contém o número do sinal que matou o filho.
- O estado do processo filho retornado por *status* tem certos bits que indicam se a terminação foi normal, o número de um sinal, se a terminação foi anormal, ou ainda se foi gerado um *core file*.
- O estado de terminação pode ser examinado (os bits podem ser testados) usando macros, definidas em <sys/wait.h>. Os nomes destas macros começam por WIF e podem ser são listadas com o comando shell *man 2 wait*.

# Valores de *status* (cont.)

PROCESSO "PAI"
**wait (&*stat*)**

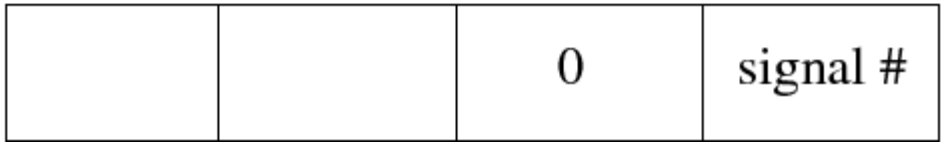



PROCESSO "FILHO"
No caso de um **exit(*var*)**




Mas e se o processo "FILHO" for finalizado devido a um sinal recebido?!

PROCESSO "PAI"
**wait (*stat*)**



# Valores de *status* (cont.)

- O POSIX expecifica seis macros, projetadas para operarem em pares:

**`WIFEXITED(status)`** – permite determinar se o processo filho terminou normalmente. Se `WIFEXITED` avalia um valor não zero, o filho terminou normalmente. Neste caso, `WEXITSTATUS` avalia os 8-bits de menor ordem retornados pelo filho através de _exit(), exit() ou return de main.
**`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/suspenso (stopped)
**`WSTOPSIG(status)`** – permite obter o número do sinal que provocou o congelamento do processo filho

- Linux: `WIFCONTINUED(status) (Linux 2.6.10)`

# Valores de *status* (cont.)

- Estrutura Geral:

```
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! */
    }
}
```

# Exemplo 1: Process *fan wait*

*(testa_wait_1.c )*

```
// O programa é lançado em background. Após o segundo filho ser bloqueado no laço infinito, um
// sinal é lançado para interromper a sua execução, por meio de comando shell
//"kill <número-do-sinal> <pid-filho2>"
#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(30) ;
                printf("\tEu sou tao jovem, e ja me sinto tao fraco!\n") ;
                printf("\tAh nao... Chegou minha hora!\n") ;
                exit(7) ;
        }
        else {
                int ret1, status1 ;
                printf("Vou 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 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);
}
```

# Exemplo 2: *wait e init* *(testa_wait_2.c)*

```
/* O programa é lançado em background.
   Primeiro, rode normalmente o programa. Verifique que o pai sai do wait e é concluído assim que um dos filhos
termina.
   Na segunda vez, rode o programa matando o primeiro filho logo depois que o Filho2 for dormir.
   Verifique que agora o pai sai do Wait(), terminando antes do Filho2. Verifique que Filho2 foi adotado pelo
init. */
#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("\nOi, eu sou o pai PID = %d. Vou criar um filho.\n",getpid()) ;


            if ((pid=fork()) == 0) {
                        printf("\tOi, eu sou o Filho1, PID = %d, PPID = %d.\n",getpid(),getppid()) ;
                        printf("\tVou ficar num loop infinito.\n") ;
                        for(;;) ;
            }
            else {
                        printf("Oi, sou eu, o pai, de novo. Vou criar mais um filho e depois vou entrar em
                                    wait().\n") ;
                        if ((pid=fork()) > 0)
                                    wait(NULL);
                        else {
                                    printf("\tOi, eu sou Filho2, PID = %d, PPID = %d.\n",getpid(),getppid()) ;
                                    printf("\tVou dormir um pouco. Use ps -l agora\n");
                                    sleep(60);
                                    printf("\tOpa, sou o Filho2. Acordei mas estou terminando agora. Use ps -l
                                                novamente.\n") ;
                        }
            }
}
```

# Exemplo 3: *wait all children*

```
// Para rodar o programa: $testa_wait_3 <número de processos>
// Pai espera por todos os filhos -
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <errno.h>

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

          if (argc != 2) {    // check number of command-line arguments
                    fprintf(stderr, "Usage: %s n\n", argv[0]);
                    return 1;
          }

          n = atoi(argv[1]);
          for (i = 1; i < n; i++)
                    if ((childpid = fork()) <= 0)   //only the child (or error) enters
                              break;

          for ( ; ; ) {
                    childpid = wait(NULL);
                    if ((childpid ==-1) && (errno != EINTR))
                              break;
          }

          fprintf(stderr, "I am process %ld, my parent is %ld\n", (long)getpid(),
                    (long)getppid());
          return 0;
}
```

# Como usar *wait* sem bloquear?

- A opção WNOHANG na chamada *waitpid* permite que um processo pai verifique se um filho terminou, sem que o pai bloqueie caso o status do filho ainda não tenha sido reportado (ex: o filho não tenha terminado)
  - Neste caso *waitpid* retorna 0

```
pid_t child pid;

while (childpid = waitpid(-1, NULL, WNOHANG))
   if ((childpid == -1) && (errno != EINTR))
      break;
```

Vejam o exemplo
testa_wait_3.c ...

# E se o processo pai receber um sinal?

- 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;
}
```

Isso não é um problema pro Linux...

# Exemplo 4: r_*wait*

```
// Para rodar o programa em background: $test_wait_4 <número de processos> &
// Pai espera todos os filhos terminarem, mesmo se um deles for morto durante o sleep().
// Usa a função r_wait() para esperar por todos os filhos.
// Observar a diferença entre EINTR e ECHILD (o processo pai fica esperando eternamente).

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


pid_t r_wait(int *status) {
        int retval;

        while (((retval = wait(status)) == -1) && (errno == EINTR)) ;
//      while (((retval = wait(status)) == -1) && (errno == ECHILD));
        return retval;
}
```

# Exemplo 4: r_*wait*

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

        if (argc != 2) { // check for valid number of command-line arguments
            fprintf(stderr, "Usage: %s n\n", argv[0]);
            return 1;
        }

        n = atoi(argv[1]);
        for (i = 1; i < n; i++)
            if ((childpid = fork()) <= 0) { //only the child (or error) enters
                  sleep(10); break;
        }

        while (r_wait(NULL) > 0) ;  // wait for all of your children

         fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld \n",

           i, (long)getpid(), (long)getppid(), (long)childpid);
        return 0;
}
```

# Exemplo 5: r_*wait* *(testa_wait_5.c – example 3.15)*

```
/* Determina o status de exit de um processo filho - TEM ERRO - FALTA ACERTAR!! */
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
pid_t r_wait(int *status) {
          int retval;
          while (((retval = wait(status)) == -1) && (errno == EINTR)) ;
          return retval;
}
int main(void) {
          pid_t pid;
          int status;

          if ((pid = fork()) < 0)
                    fprintf(stderr,"fork error\n");
          else if (pid == 0) exit(7);                    /* child1 finishes normally */

          if (wait(&status) != pid)                      /* parent code */
                    fprintf(stderr,"wait error\n");
          pr_exit(status);                               /* wait for child and print its status */

          if ((pid = fork()) < 0)
                    fprintf(stderr,"fork error\n");
          else if (pid == 0) abort();                    /* child2 generates SIGABRT */

          if (wait(&status) != pid)
                    fprintf(stderr,"wait error\n");
          pr_exit(status);
```

# Exemplo 5: r_*wait* *(testa_wait_5.c – example 3.15)*

```
        if ((pid = fork()) < 0)
                fprintf(stderr,"fork error\n");
        else if (pid == 0) status /= 0;             /* child3 - divide by 0 generates SIGFPE */

        if (wait(&status) != pid)
                fprintf(stderr,"wait error\n");
        pr_exit(status);                            /* wait for child and print its status */

        if ((pid = fork()) < 0)
                printf(stderr,"fork error\n");
        else if (pid == 0)
                sleep(30);                          /* child4 - waiting SIGSTOP */

        if (wait(&status) != pid)
                fprintf(stderr,"wait error\n");
        pr_exit(status);                            /* wait for child and print its status */
        exit();
}
void show_return_status(void) {
        pid_t childpid;
        int status;
        childpid = r_wait(&status);
        if (childpid == -1)
                perror("Failed to wait for child");
        else if (WIFEXITED(status))
                printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
             else if (WIFSIGNALED(status))
                printf("abnormal termination, signal number = %d%s\n", WTERMSIG(status),
                  else if (WIFSTOPPED(status))
                printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}
```

# Exemplo 6: Process chain wait *(testa_wait_6.c – exemple 3.21)*

```
// Para rodar o programa: $testa_wait_6 <número de processos>
// Cada filho criado espera por seu próprio filho completar antes de imprimir a msg.
// As mensagens aparecem na ordem reversa da criação.
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main (int argc, char *argv[]) {
   pid_t childpid;
   int i, n;
   pid_t waitreturn;
   if (argc != 2){   /* check for number of command-line arguments */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (childpid = fork()) break;
   while (childpid != (waitreturn = wait(NULL)))
      if ((waitreturn == -1) && (errno != EINTR))
         break;
   fprintf(stderr, "I am process %ld, my parent is %ld\n", (long)getpid(),(long)getppid());
   return 0;
}
```

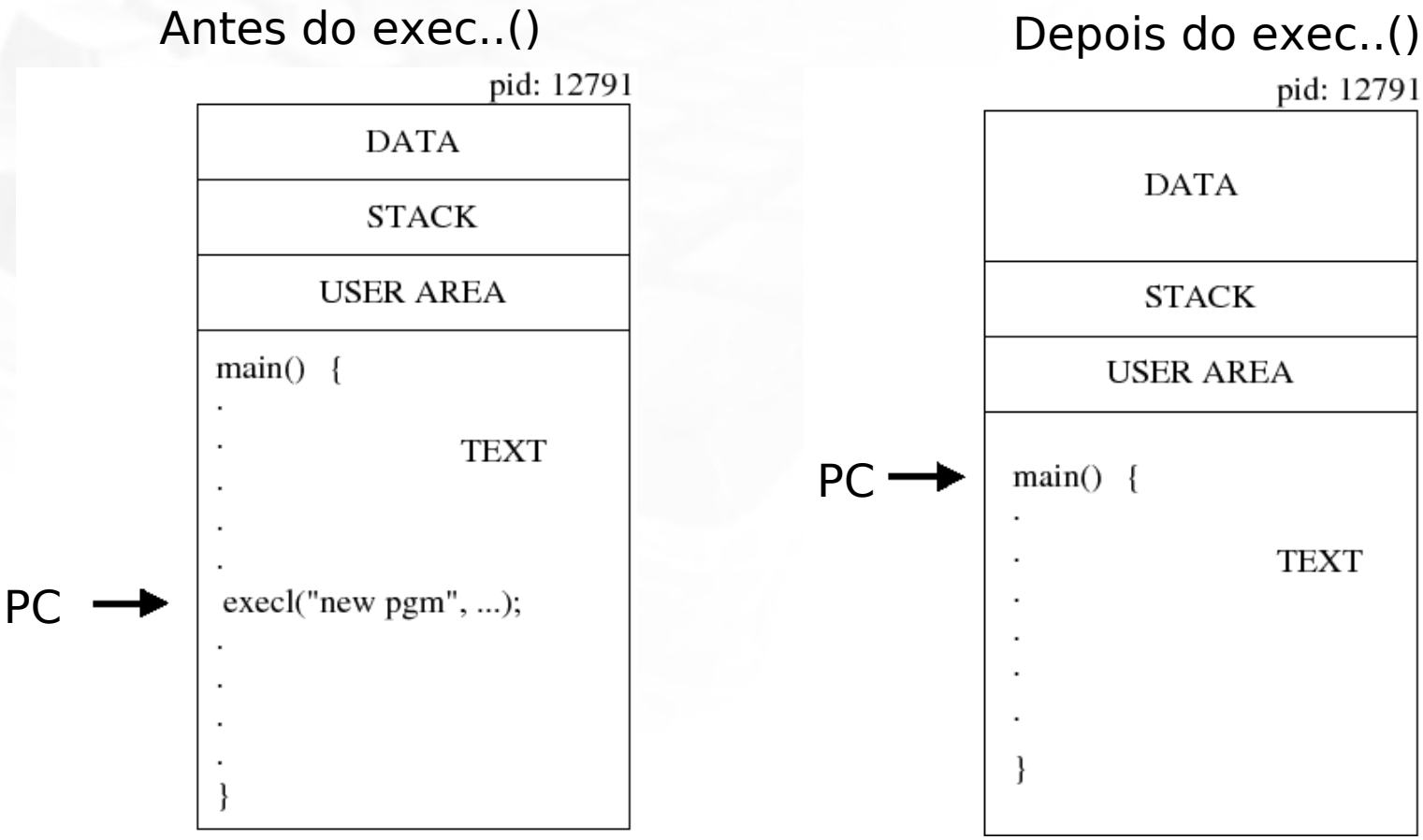
# Referências

- Kay A. Robbins, Steven Robbins, *UNIX Systems Programming: Communication, Concurrency and Threads, 2nd Edition*
  - Capítulo 3

# Primitivas exec..()

- As primitivas *exec* constituem, na verdade, uma família de funções que permitem a um processo executar o código de outro programa.
- Não existe a criação efetiva de um novo processo, mas simplesmente uma substituição do programa de execução.
- Quando um processo chama *exec..()* ele imediatamente cessa a execução do programa atual e passa a executar o novo programa, a partir do seu início.
  - O processo NÃO retorna do exec..(), em caso de sucesso.

# O que ocorre quando um processo faz *exec()?*



# A Família de SVC's *exec..()*

- Existem seis primitivas na família, as quais podem ser divididas em dois grupos:
  - **execl()**, para o qual o número de argumentos do programa lançado é conhecido em tempo de compilação. Nesse caso, os argumentos são pasados um a um, terminando com a string nula.
    - execl(), execle() e execlp()
  - **execv()**, para o qual esse número é desconhecido. Nesse caso, os argumentos são passados em um array de strings.
    - execv(), execve() e execvp().
- Em ambos os casos, o primeiro argumento deve ter o nome do arquivo executável.

# A Família de SVC's *exec..()* (cont.)

| Function | Specification of program file *(–, p)* | Specification of arguments *(v, l)* | Source of environment *(e, –)* |
|---|---|---|---|
| *execve()* | pathname | array | *envp* argument |
| *execle()* | pathname | list | *envp* argument |
| *execlp()* | filename + PATH | list | caller's *environ* |
| *execvp()* | filename + PATH | array | caller's *environ* |
| *execv()* | pathname | array | caller's *environ* |
| *execl()* | pathname | list | caller's *environ* |

- 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)

# A Família de SVC's *exec()* (cont.)

```
#include <unistd.h>

int execl (const char *pathname, const char *arg,...);
int execv (const char *pathname, char *const argv[]);

int execle (const char *pathname, const char *arg ,...,
            char *const envp[]);
int execve (const char *pathname, char *const argv[],
            char *const envp[]);

int execlp (const char *filename, const char*arg,...);
int execvp (const char *filename, char *const argv[]);
```

# A Família de SVC's *exec()* (cont.)

- Os parâmetros *char arg, ...* das 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. Elas descrevem uma lista de um ou mais ponteiros para strings não nulas que representam a lista de argumentos para o programa.
- Já as funções *execv()*, *execvp()* e *execve()* fornecem um vetor de ponteiros para strings não nulas que representam a lista de argumentos para o programa.
- A função *execle()* e *execve()* também especificam o ambiente do processo após o ponteiro NULL da lista de parâmetros. As outras funções consideram o ambiente para o novo processo como sendo igual ao do processo atualmente em execução.

# Exemplos de Uso

```
execl (“/bin/cat”, “cat”, “f1”, “f2”, NULL)
...
static char *args[] = { “cat”, “f1”, “f2”, NULL);
execv (“/bin/cat”, args);
...
execlp (“ls”, “ls”, “-l”, NULL)
execvp (argv[1], &argv[1])
...
static char *env[] = {"TERM=vt100","PATH=/bin:/usr/bin",
                      NULL };
execle (“/bin/cat”, “cat”, “f1”, “f2”, NULL, env)
execve("/bin/catl", args, env);
```

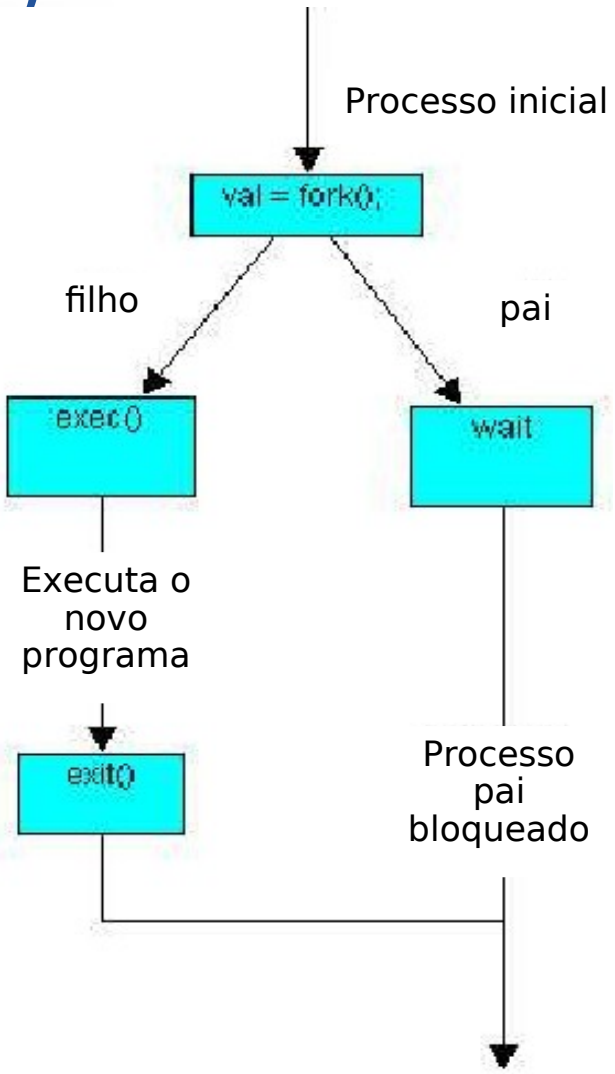
# Exemplo (arquivo testa_exec_0.c)

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

int main()
{
     printf ("Eu sou o processo PID=%d e estou executando o programa
                         testa_exec_0\n", getpid()) ;
     printf ("Vou fazer um exec() agora!\n") ;
     execl("/bin/ls","ls","-l", "testa_exec_0.c",NULL) ;
     printf ("Estou de volta! Vou continuar a execução do programa
                              testa_exec_0\n") ;
     return 1;
}
```

# Uso de fork() - exec()

- Um processo executando um programa A quer executar um outro programa B:
  - Primeiramente ele deve criar um processo filho usando fork().
  - Em seguida, o processo recém criado deve substituir o programa A pelo programa B, chamando uma das primitivas da família *exec.*
  - O processo pai espera pelo término do processo filho usando a chamada wait().



# Exemplo – Uso de fork-exec (arquivo testa_exec_0.c)

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

int main()
{
    if ( fork()==0 ) execl( "/bin/ls","ls","-l", "testa_exec_0a.c",NULL) ;
    else {
        sleep(2) ; /* espera o fim de ls para executar o printf() */
        printf ("Eu sou o pai e finalmente posso continuar\n") ;
    }
    return 1;
}
```

# Retorno do exec..()

- Sucesso - não retorna
- Se alguma das funções exec..() retornar, um erro terá ocorrido
  - retorna o valor -1
  - seta a variável *errno* com o código específico do erro
- Valores possíveis da 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 arquivo exec |
| **ENOTDIR** | Não é um diretório |

# Exemplo 1: (arquivo testa_exec_1.c - program 3.4)

Programa que cria um proceso filho para executar o comando ls -l.

```
#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;
        }
        printf("I am the parent. I am waiting for my child to complete...\n");
        if ( childpid != wait (NULL)) {
                perror("Parent failed to wait due to signal or error");
                return 1;
        }
        printf("Child completed – I am now exiting.\n");
        return( 0 );
}
```

# Exemplo 2: (arquivo testa_exec_2.c - program 3.5)

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

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
pid_t r_wait(int *status) {
        int retval;
        while (((retval = wait(status)) == -1) && (errno == EINTR)) ;
        return retval;  }
int main(int argc, char *argv[]) {
        pid_t childpid;
        if ( argc < 2 ) {
                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;
        }
        printf("Child completed -- parent now exiting.\n");
        return 0;
}
```

# Exemplo 3: (arquivo testa_exec_3.c)

Interpretador de comandos simples que usa execlp() para executar comandos digitados pelo usuário.

```
//  myshell.c
    #include <stdio.h>
    #include <unistd.h>
    #define  EVER   ;;

    int main()      {
      int process;
      char line[81];

      for (EVER) {
        fprintf(stderr, "cmd: ");
          if ( gets (line) == (char *) NULL)      /* blank line input */
            return 0;
        process = fork ();                        /* create a new process */
        if (process > 0)                          /* parent */
          wait ((int *) 0);               /* null pointer - return value not saved */
        else if (process == 0)  {     /* child */
              execlp (line, line, (char *) NULL);                 /* execute program */
              fprintf (stderr, "Can't execute %s\n", line);
              return 1; }
        else if ( process == -1) {    /* can't create a new process */
              fprintf (stderr, "Can't fork!\n");
              return 2; }
        }
    }
```

# Informações Mantidas...

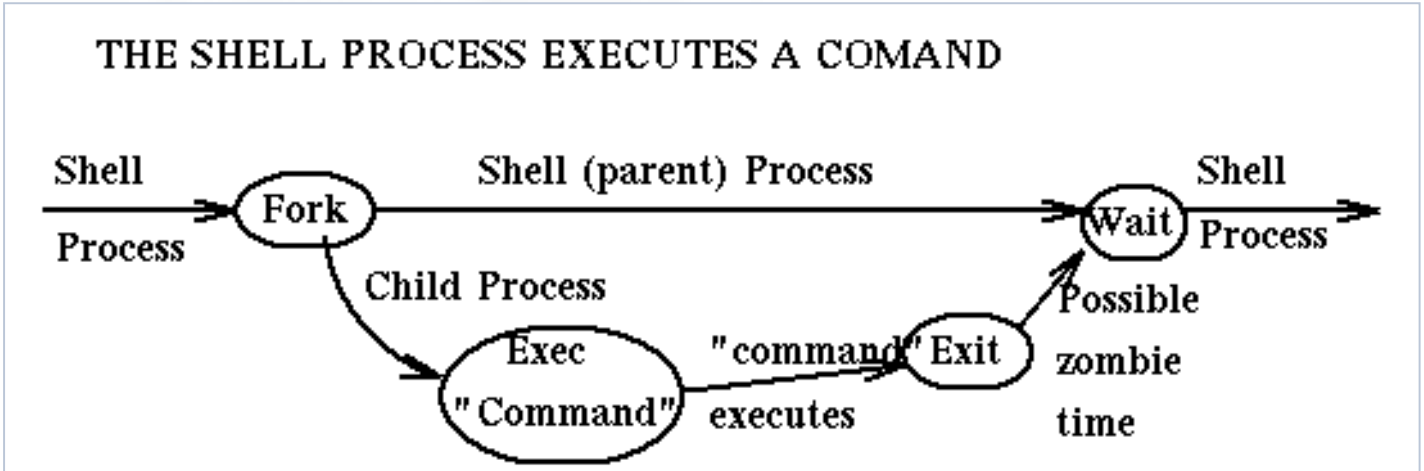
- O processo que executou a função *exec()* mantém as seguintes informações:
  - pid e o ppid
  - user, group, session id
  - Máscara de sinais
  - Alarmes
  - Terminal de controle
  - Diretórios raiz e corrente
  - Informações sobre arquivos abertos
  - Limites de uso de recursos
  - Estatísticas e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

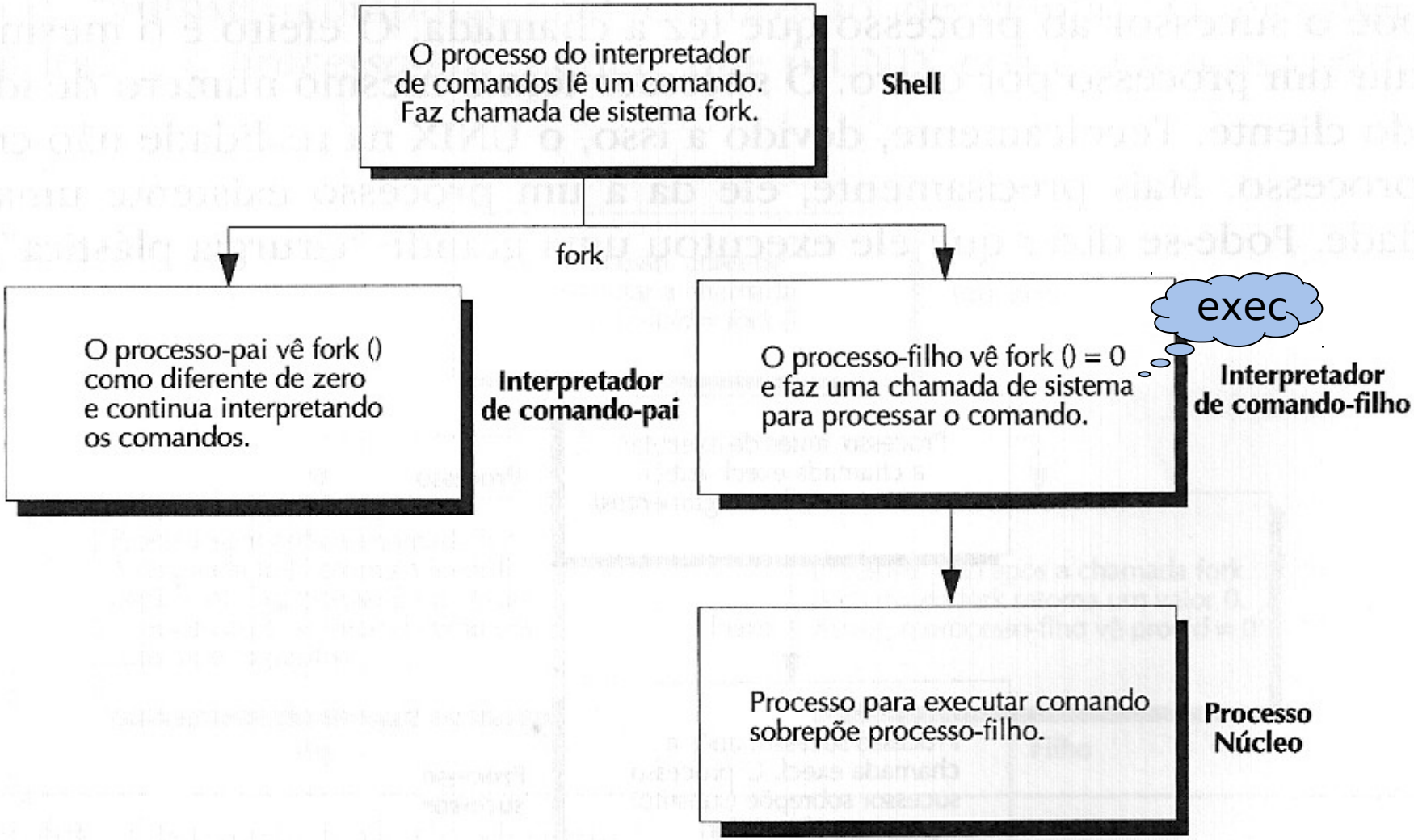
# Exemplo Clássico: o Shell do UNIX

- Quando o interpretador de comandos UNIX interpreta comandos, ele chama fork() e exec().

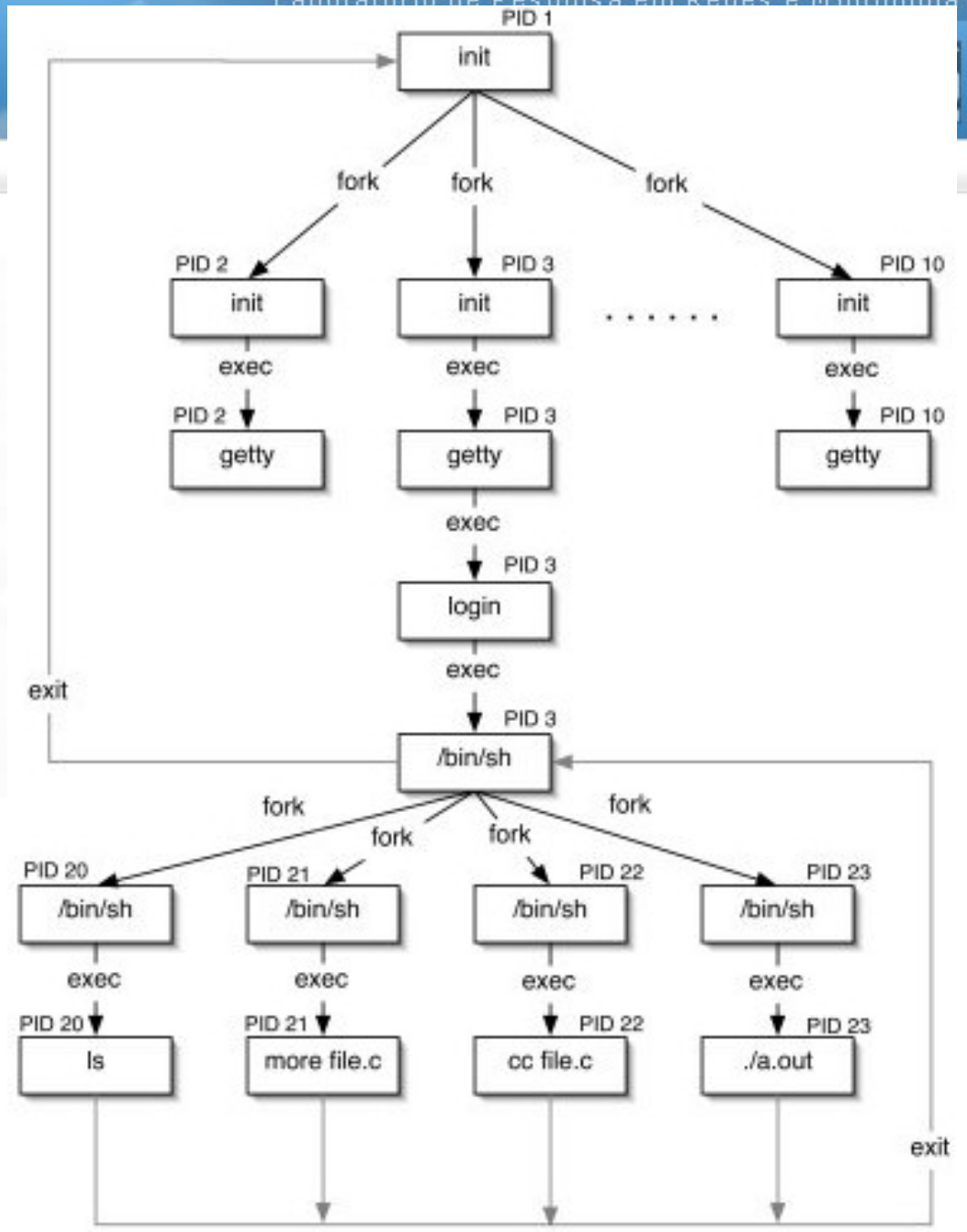
```
...
Lê comando para o interpretador de comandos
...
If (fork()==0)
   exec...(command, lista_arg ...)
```



# Exemplo Clássico: o Shell do UNIX (cont.)



# Exemplo Clássico: o Processo *init*



# Processos *background* e *foreground*

- Existem vários tipos de processos no Linux: processos interativos, processos em lote (*batch*) e *Daemons*. Processos interativos são iniciados a partir de uma sessão de terminal e por ele controlados. Quando executamos um comando do *shell*, entrando simplesmente o nome do programa seguido de <enter>, estamos rodando um processo em *foreground*.
- Um programa em *foreground* recebe diretamente sua entrada (*stdin*) do terminal que o controla e, por outro lado, toda a sua saida (*stdout* e *stderr*) vai para esse mesmo terminal. Digitando *Ctrl-Z,* suspendemos esse processo, e recebemos do *shell* a mensagem *Stopped* (talvez com mais alguns caracteres dizendo o número do *job* e a linha de comando).
- A maioria dos *shells* tem comandos para controle de *jobs*, para mudar o estado de um processo parado para *background*, listar os processos em *background*, retornar um processo de *back* para *foreground*, de modo que o possamos controlar novamente com o terminal. No *bash* o comando "`jobs`" mostra os *jobs* correntes, o `bg` restarta um processo suspenso em *background* e o comando `fg` o restarta em *foreground*.
- *Daemons* ou processos servidores, mais freqüentemente são iniciados na partida do sistema, rodando continuamente em *background* enquanto o sistema está no ar, e esperando até que algum outro processo solicite o seu serviço (ex: sendmail).

# Processos *background* e *foreground* (cont.)

- **O Comando Jobs**
  - Serve para visualizar os processos que estão parados ou executando em segundo plano (*background*). Quando um processo está nessa condição, significa que a sua execução é feita pelo *kernel* sem que esteja vinculada a um terminal. Em outras palavras, um processo em segundo plano é aquele que é executado enquanto o usuário faz outra coisa no sistema.
  - Para executar um processo em *background* usa-se o “&” (ex: ls –l &). Se o processo estiver parado, geralmente a palavra "stopped" (ou “T”) aparece na linha de exibição do estado do processo.
- **Os comandos fg e bg**
  - O `fg` é um comando que permite a um processo em segundo plano (ou parado) passar para o primeiro plano (*foreground*), enquanto que o `bg` passa um processo do primeiro para o segundo plano. Para usar o `bg`, deve-se paralisar o processo. Isso pode ser feito pressionando-se as teclas Ctrl + Z. Em seguida, digita-se o comando da seguinte forma: `bg %número`
  - O número mencionado corresponde ao valor de ordem informado no início da linha quando o comando `jobs` é usado.
  - Quanto ao comando fg, a sintaxe é a mesma: `fg %número`

# Sessões e grupos de processos

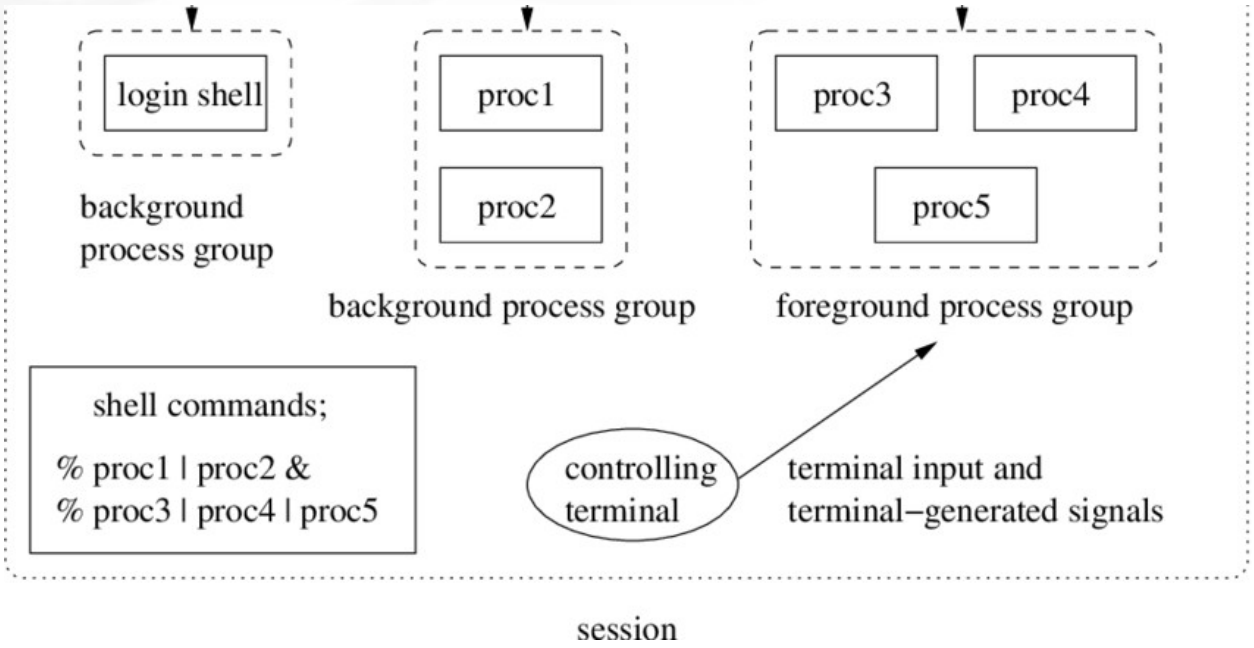
- No Unix, além de ter um PID, todo processo também pertence a um grupo. Um *process group* é uma coleção de um ou mais processos.
- Todos os processos dentro de um grupo são tratados como uma única entidade. A função *getpgrp()* retorna o número do grupo do processo chamador.
- Cada grupo pode ter um processo líder, que é identificado por ter o seu PID igual ao seu groupID.
- É possível ao líder criar novos grupos, criar processos nos grupos e então terminar (o grupo ainda existirá mesmo se o líder terminar; para isso, tem que existir pelo menos um processo no grupo - *process group lifetime*).

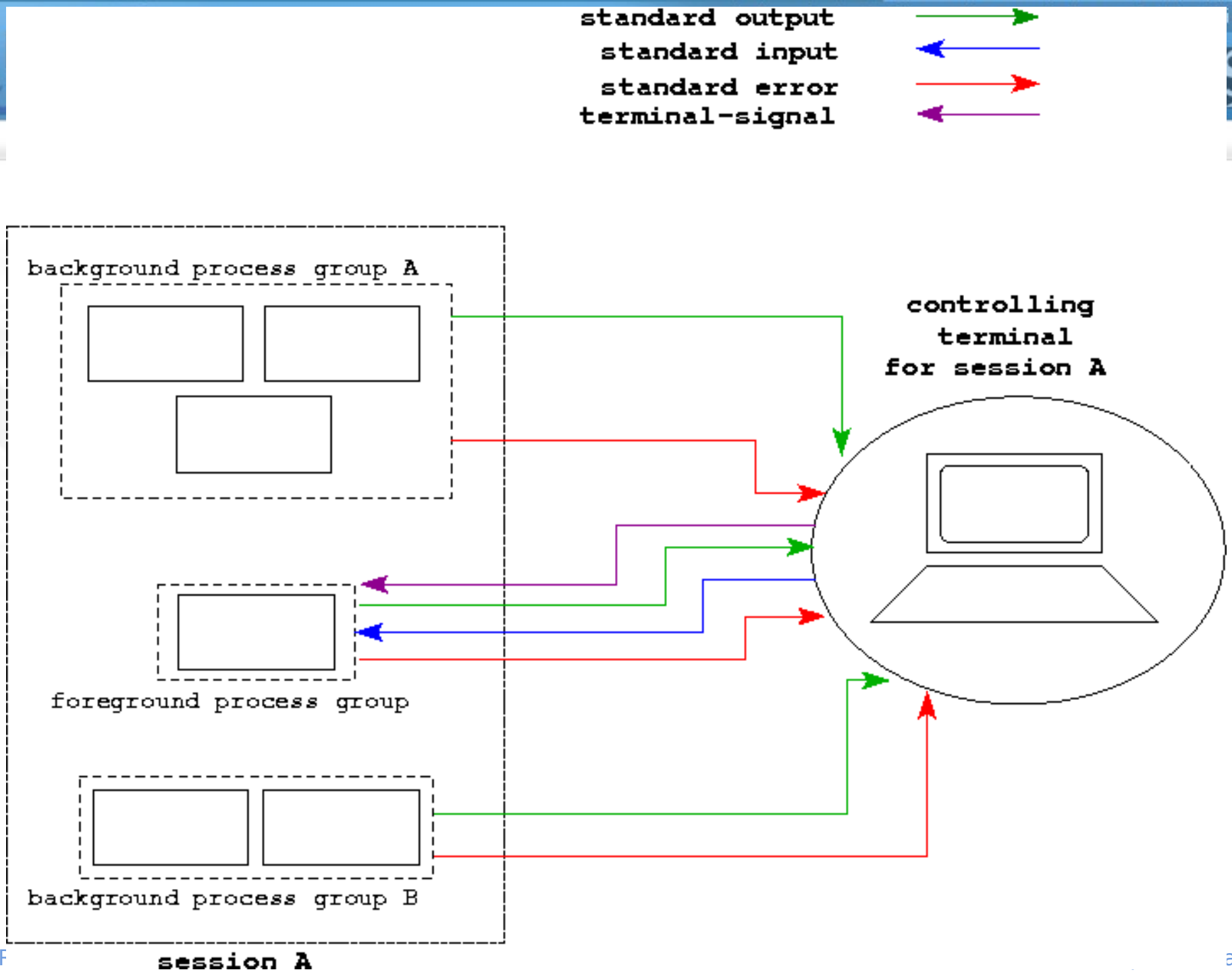
# Sessões e grupos de processos (cont.)

- Uma sessão é um conjunto de grupos de processos. Grupos ou sessões são também herdadas pelos filhos de um processo.
- Um servidor, por outro lado, deve operar independentemente de outros processos. Como fazer então que um processo servidor atenda a todos os grupos e sessões?
- A primitiva **`setsid()`**obtém um novo grupo para o processo. Ela coloca o processo em um novo grupo e sessão, tornando-o independente do seu terminal de controle (**`setpgrp()`** é uma alternativa para isso).
- É usada para passar um processo de *foreground* em *background.*

# Sessões e grupos de processos (cont.)

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




standard output
standard input
standard error
terminal-signal
background process group A
controlling terminal for session A
foreground process group
background process group B
session A

# Colocando um processo em background

```
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 um processo em *background*

# Referências

- Kay A. Robbins, Steven Robbins, *UNIX Systems Programming: Communication, Concurrency and Threads, 2nd Edition*
  - Capítulo 3