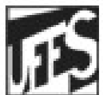
Laboratório de Pesquisa em Redes e Multimídia

SVCs para Controle de Processos no Unix (cont.)



Universidade Federal do Espírito Santo
Departamento de Informática

Sistemas Operacionais

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()*?

Antes do exec..()

pid: 12791

DATA
STACK
USER AREA
main() { . . TEXT . . . execl("new pgm", ...); . . . . }

PC → execl("new pgm", ...);

Depois do exec..()

pid: 12791

DATA
STACK
USER AREA
main() { . . TEXT . . . . }

PC → main() {

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.)

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

- 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);
```

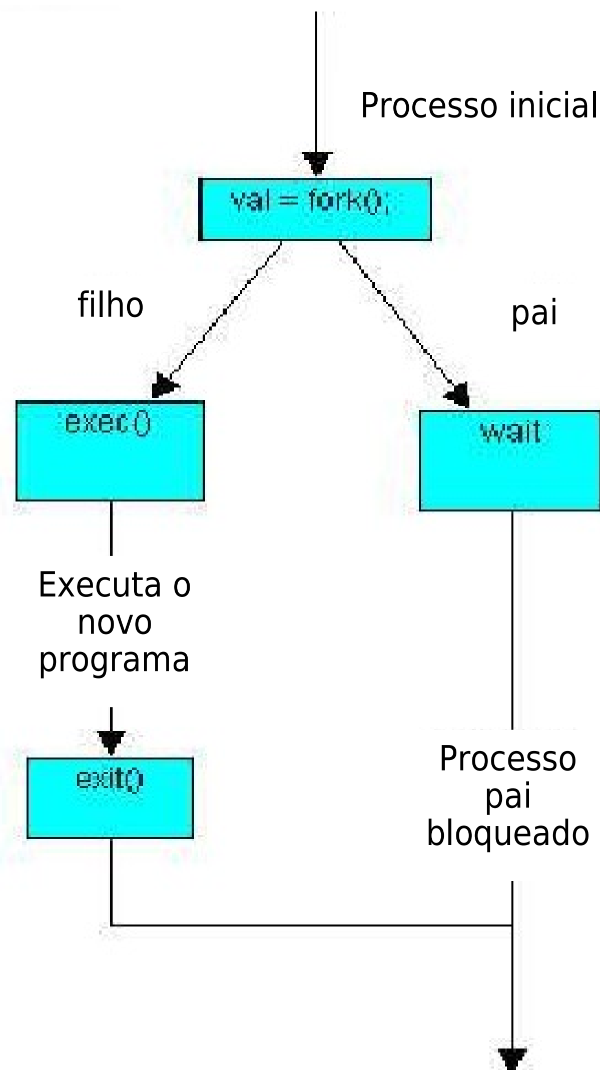
Exemplo (arquivo testa_exec_0.c)

```
#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...

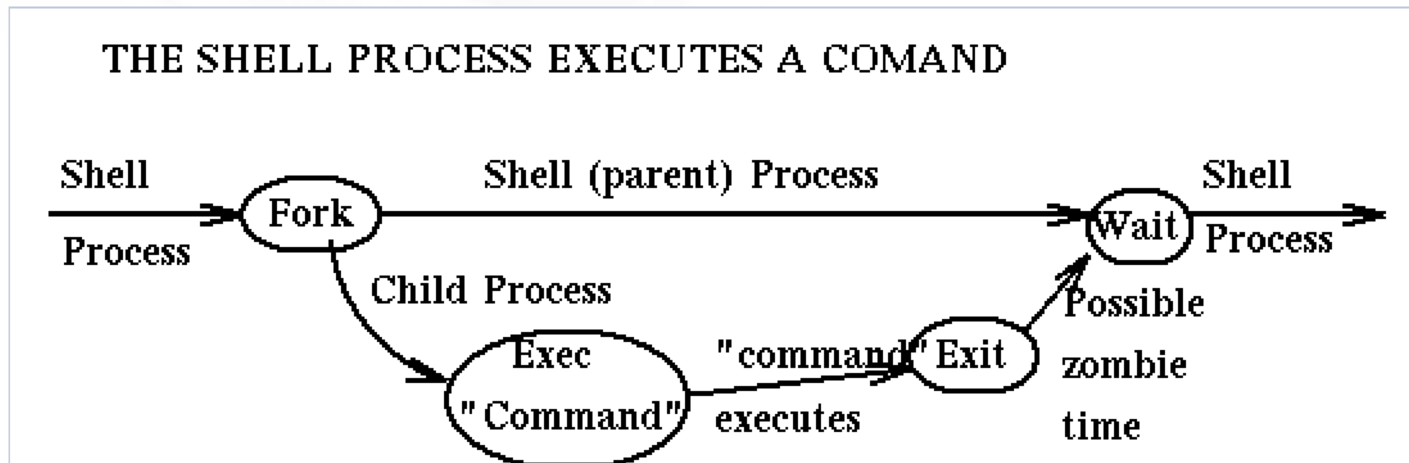
- O processo que executou a função *exec()* mantém as seguintes informações:
  - pid e o ppid
  - user, group, session id
  - Máscara de sinais
  - Alarmes
  - Terminal de controle
  - Diretórios raiz e corrente
  - Informações sobre arquivos abertos
  - Limites de uso de recursos
  - Estatísticas e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

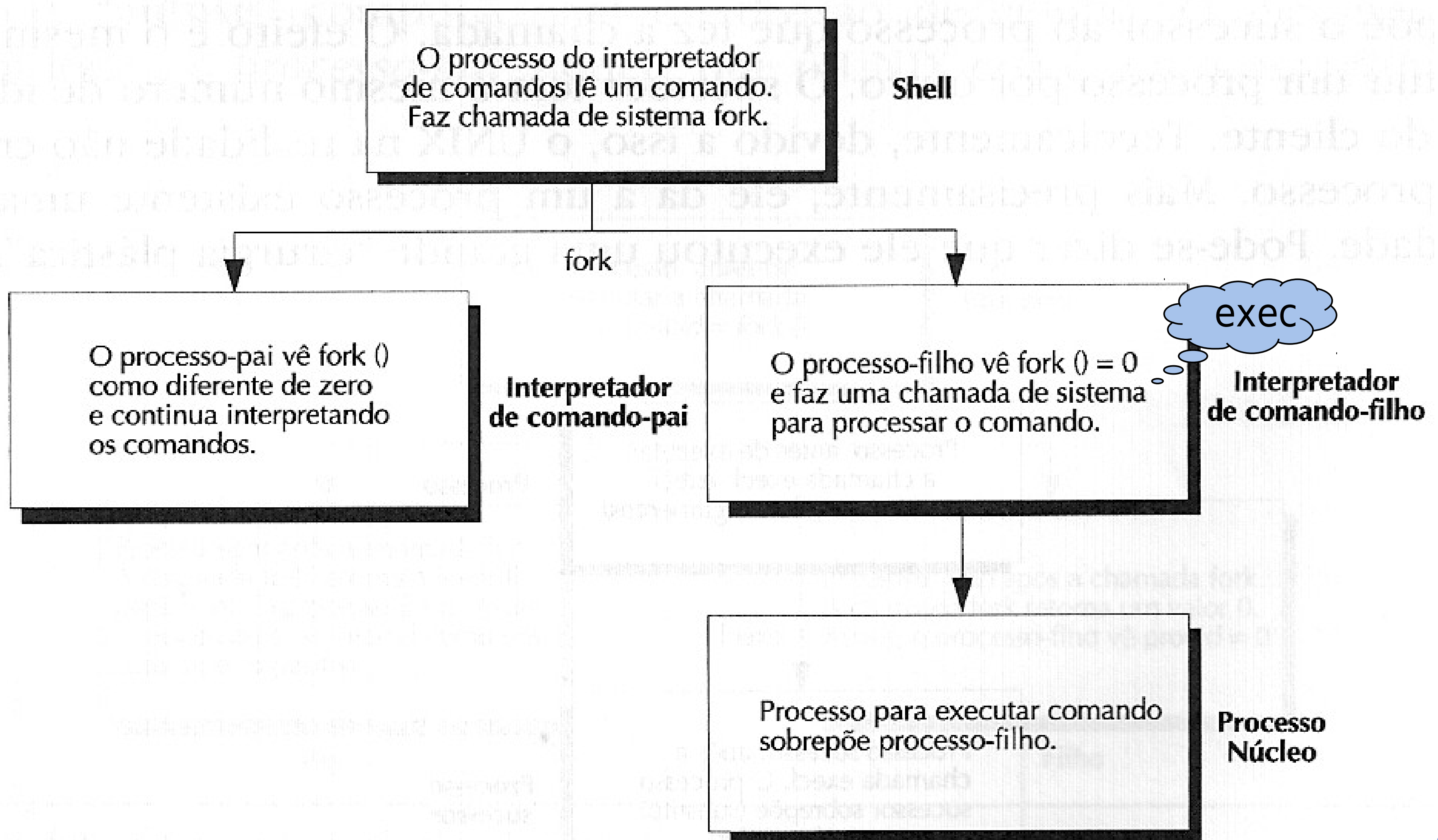
Exemplo Clássico: o Shell do UNIX

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

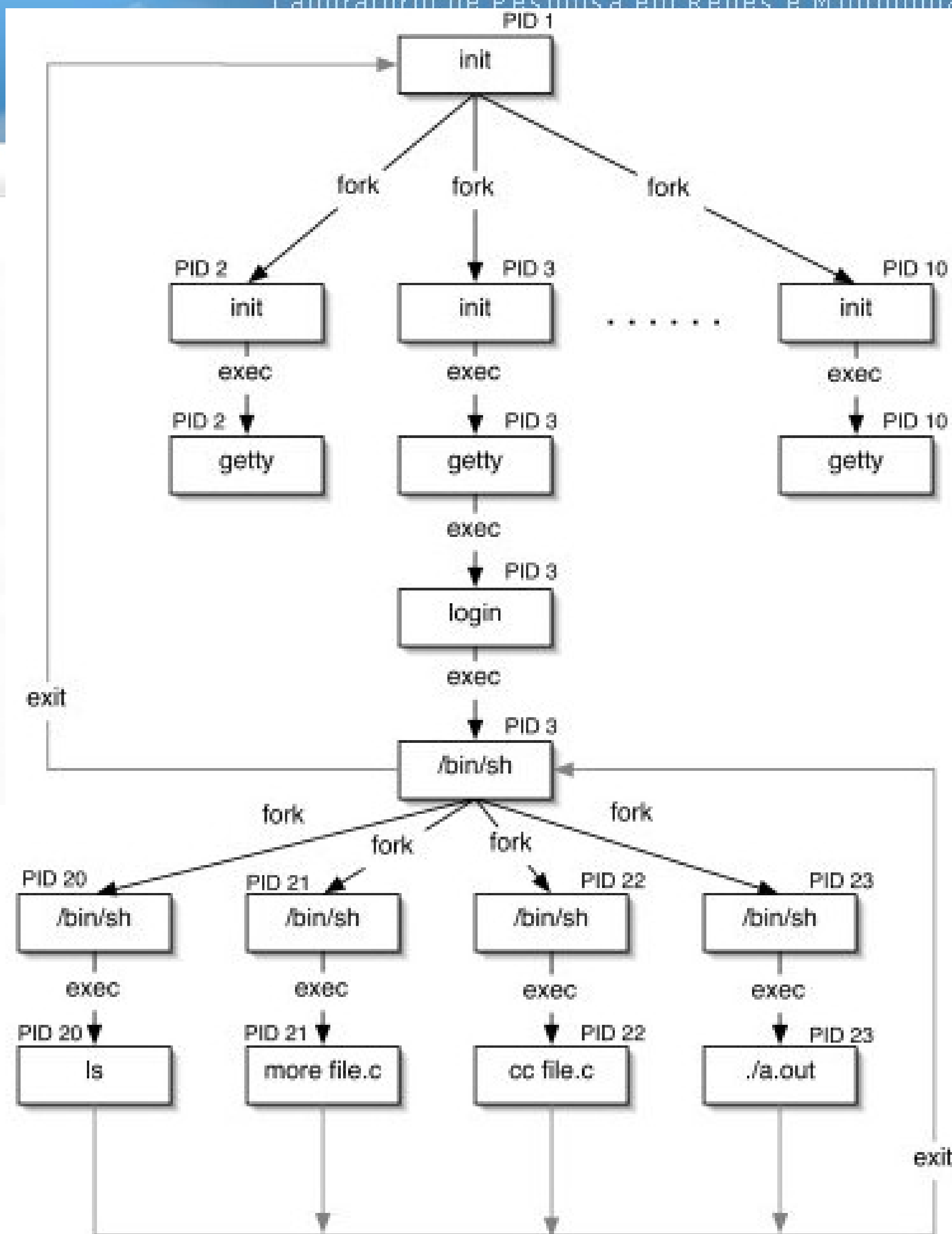
```
...
Lê comando para o interpretador de comandos
...
If (fork()==0)
   exec...(command, lista_arg ...)
```



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

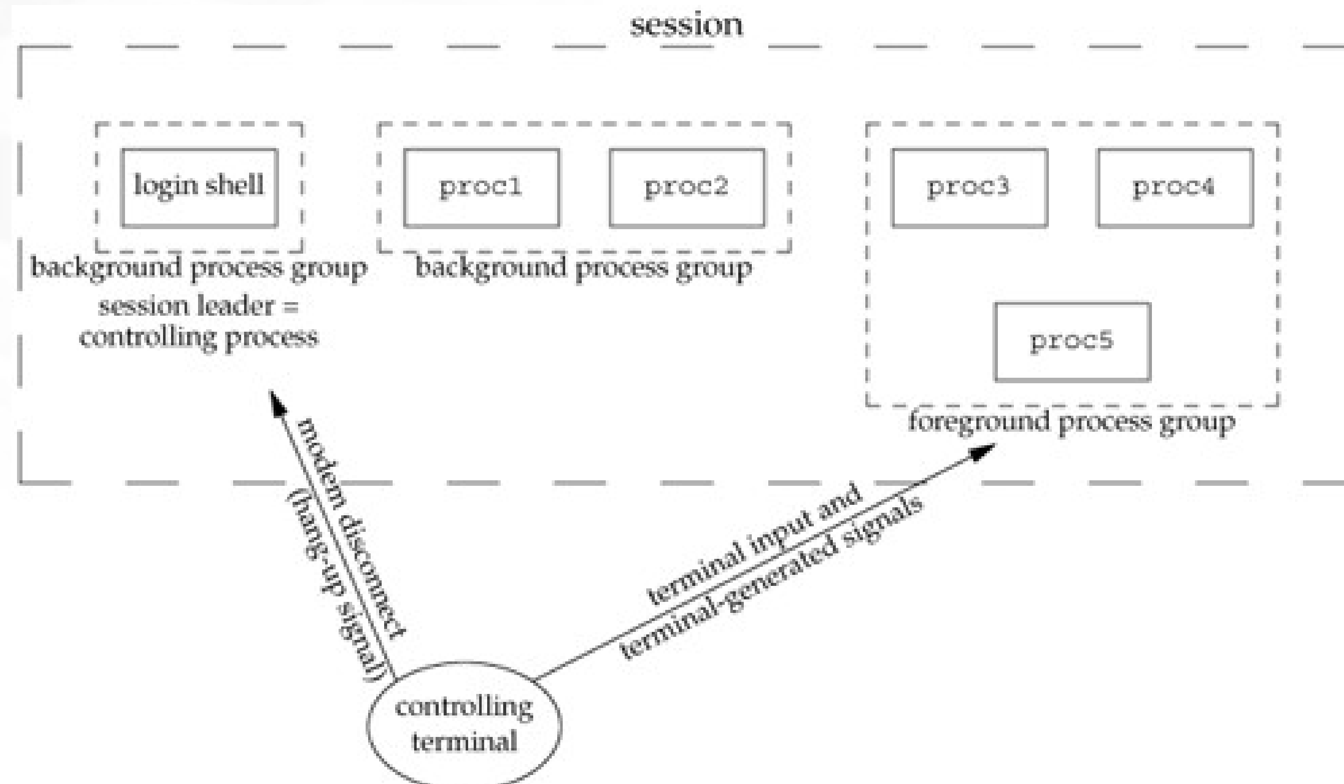
- No Unix, além de ter um PID, todo processo também pertence a um grupo. Um *process group* é uma coleção de um ou mais processos.
- Todos os processos dentro de um grupo são tratados como uma única entidade. A função `getpgrp()` retorna o número do grupo do processo chamador.
- Cada grupo pode ter um 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*



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