PROCESSOS

✓ DEFINIÇÃO

Um processo é um programa de execução independente que tem seu próprio conjunto de recursos. Para um mesmo programa (entidade estática), podem existir vários processos (entidades dinâmicas).
O UNIX, como sistema operacional multitarefa, permite a existência de vários processos ao mesmo tempo. Em máquinas monoprocessadas, o *kernel* do UNIX se encarrega de escalonar os recursos de execução do único processador, para os vários processos do sistema. Já em máquinas multiprocessadas, pode-se ter processos executando em paralelo, e não concorrentemente como nas máquinas monoprocessadas.
Quando for digitado algum comando no ambiente do UNIX, este comando é executado como um processo subordinado – **processo filho** – do processo corrente chamado **processo pai**. Todos os processos criados pelo usuário são filhos do processo de *login*.

O sistema operacional tem muitos recursos que devem ser gerenciados, incluíndo recursos de hardware e software. Um dos recursos de software que deve ser gerenciado é um programa em execução. As características do processo são:

- É um programa que pode ser executado concorrentemente.
- Pode ser criado e destruido
- Possui recursos alocados para ele
- Possui um ambiente associado a ele que: é herdado do processo pai, consiste de todas as informações relativas ao processo e pode ser alterado através de comandos no ambiente de shell
- Pode criar outros processos
- Pode se comunicar com outros processos

No UNIX cada processo possui o seu próprio ambiente onde existem todas as informações relativas ao processo e que afetam a sua execução:

- Dados
- Arquivos abertos
- Diretório corrente
- User ID
- Process ID
- Parente Process ID
- Conjunto de variáveis

✓ CRIANDO UM PROCESSO

Quando o sistema operacional é inicializado, o processo init é inicializado. Este processo é responsável pelo processo de login que aguarda pela entrada de comandos através dos terminais dos usuários. O processo init pertence ao superusuário e é controlado pelo console. O console é o terminal para onde o kernel escreve as mensagens de erro de sistema.

No momento que o usuário se conecta ao sistema, o processo init inicia um processo shell de usuário com um ambiente padrão. A partir deste ponto o usuário cria outros processos executando comandos, rodando programas ou shell scripts.

✓ EXECUTANDO PROCESSOS EM BACKGROUND E FOREGROUND

Quando você executa um comando shell ou um script, por default é executado em foreground (primeiro plano). Quando termina a execução do comando só então é possível executar outro pelo prompt do UNIX. Isto ocorre porque quando um comando está sendo executado em foreground o shell não pode aceitar outra entrada até que a execução seja concluída.

Para permitir mais do que um comando sendo executado ao mesmo tempo, os comandos precisam ser executados em background que não possui controle direto da entrada e saída de dados do terminal. Este método é aconselhável quando o comando a ser executado consome tempo de CPU e não requer entrada de dados interativa. Como exemplo citamos programas de classificação de dados, compilação, cálculos matemáticos complexos.

Para que um programa seja executado em background acrescente o caracter **&** após o comando no prompt do UNIX:

```
$ ls -la &
```

COMANDOS DE CONTROLE DE PROCESSOS

✓ PS

Sintaxe: **ps** [-fe]

Descrição Lista os processos do sistema.

Opções: -fe → inclui informações sobre todos os processos do sistema;
Nenhuma → exibe somente os processos do usuário;

Exemplo
```
$ ps
PIDTTY    STAT  TIME  CMD
1058      pp0   S     21:09 -bash
1071      pp0   R     21:09 ps
```

OBS: No ambiente simulador JSLinux usar: **ps -o pid,ppid,user,comm**

✓ KILL

Sintaxe: **kill** [-sinal] pid
kill [-l]

Descrição Envia sinais para os processos.

Opções: -sinal envia o sinal especificado ao(s) processo(s) ("pid");
Sinal -9: encerra o processo.
-l apresenta lista de sinais que podem ser usados com o comando **kill**.

Exemplo:
```
$ kill -9 1234
$ kill -SIGKILL 1234
```

PRIMITIVAS C PARA MANIPULAÇÃO DE PROCESSOS

✓ Compilador gcc

Utilizado para pré-processar, compilar e linkeditar programas fonte da linguagem C em ambiente Linux (em alguns sistemas precisa ser instalado adicionalmente). Deve ser invocado via linha de comando do shell no formato básico:

gcc –o *outfile infile*

Exemplo:
```
gcc –o exemplo_01 exemplo_01.c
```

✓ Identificadores de um processo

Cada processo possui um identificador (ou ID) único denominado **pid**. As primitivas permitindo o acesso aos diferentes identificadores de um processo são as seguintes:

getpid() /* retorna o ID do processo */
getppid() /* retorna o ID do pai do processo */

Exemplo:
```
================================================================================
/* arquivo exemplo_01.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
     printf("\n...EXECUCAO DO PROGRAMA EXEMPLO_01.C\n");
     printf("...Eu sou o processo PID=%d de pai PPID=%d \n",getpid(),getppid());
     exit(0);
}
================================================================================
```

Resultado da execução:

```
Server01:~> ./exemplo_01
...EXECUCAO DO PROGRAMA EXEMPLO_01.C
...Eu sou o processo PID=28448 de pai PPID=28300
```

Observe que o pai do processo executando exemplo_01 é o processo bash. Para confirmar a afirmação, faça um ps na janela de trabalho:

```
Server01:~> ps
  PID TTY STAT  TIME COMMAND
28300  ?  S    0:00 bash
28451  ?  R    0:00 ps
```

✓ Primitiva exit()

O processo é finalizado. Quando um processo faz exit, todos os seus processos filho são herdados pelo processo *init* de pid igual a 1.

Por convenção, um código de retorno igual a 0 significa que o processo terminou normalmente. Um código de retorno não nulo (em geral -1 ou 1) indicará a ocorrência de um erro de execução.

Exemplo: `exit(0);`

✓ Primitiva execl()

A família de comandos exec() cria uma nova imagem de processo a partir de um processo chamador. O processo chamador é funcionalmente substituído pelo novo processo.

Exemplo:

```
==========================================================================
/* arquivo exemplo_02.c */
#include <stdio.h>
#include <unistd.h>

int main()
{
     printf("\n....EXECUCAO DO PROGRAMA EXEMPLO_02.C\n");
     printf("\n....Eu sou o processo PID=%d de pai PPID=%d",getpid(),getppid());
     printf("\n....Agora esse processo vai ser substituido pelo comando 'ps -f'
            em seguida \n");
     execl("/bin/ps","ps","-f",NULL);
     printf("Este printf nunca sera executado!!!");
     exit(0);
}
==========================================================================
```

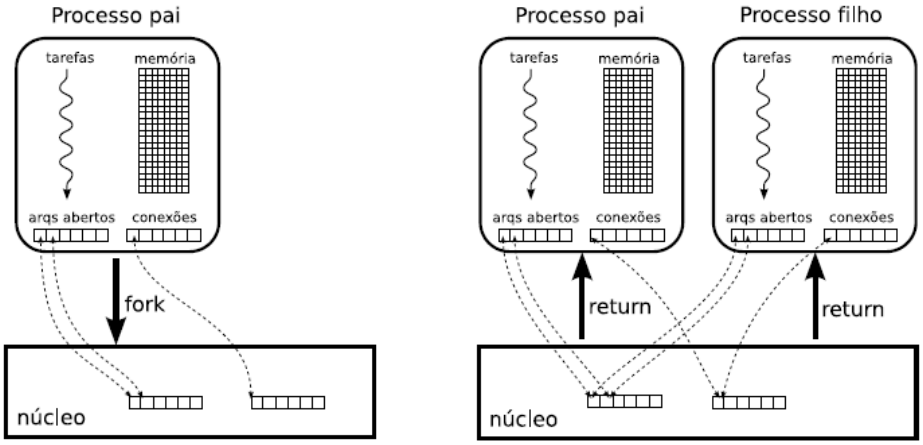
O primeiro argumento do execl() é o caminho do comando que será executado com o nome do comando; em seguida devem ser fornecidos os demais argumentos iniciando com o nome do comando. O último argumento deverá ser NULL.

Resultado da execução:

```
Server01:~> ./exemplo_02
....EXECUCAO DO PROGRAMA EXEMPLO_02.C
....Eu sou o processo PID=28448 de pai PPID=28300
....Agora esse processo vai ser substituido pelo comando 'ps -f' em seguida
UID    PID   PPID STIME  TTY  TIME  COMMAND
User1 28300 28251 10:27   ?   0:00  bash
User1 28448 28300 10:27   ?   0:00  ps -f
```

✓ Primitiva fork()

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). 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).



Primitiva fork(): antes (esquerda) e depois (direita) de sua execução

A chamada de sistema fork() é invocada por um processo (o pai), mas dois processos recebem seu retorno: o processo pai, que a invocou, e o processo filho, recém-criado. A chamada à função fork retorna os seguintes códigos de retorno:

zero=no processo filho **>zero**=no processo pai **<zero**=falha na execução do fork)

A execução do processo cópia inicia exatamente na instrução seguinte à instrução onde o fork() foi fornecido.

Exemplo:

```
==========================================================================
/* arquivo exemplo_03.c */
#include <stdio.h>
#include <unistd.h>
int main()
{
     int retcode;
     printf("\n...INICIANDO EXECUCAO DO PROGRAMA EXEMPLO_03.C\n");
     retcode = fork();
     if(retcode != 0) {
        printf("\n...Eu sou o processo PAI de PID=%d e meu filho tem o PID=%d
               \n",getpid(),retcode);
     }
     else {
        printf("\n...Eu sou o processo FILHO \n");
     }
     printf("\n...Terminando a minha execucao. Goodbye!!! \n\n");
     exit(0);
}
==========================================================================
```

Resultado da execução:

```
Server01:~> ./exemplo_03
...INICIANDO EXECUCAO DO PROGRAMA EXEMPLO_03.C
...Eu sou o processo FILHO
...Terminando a minha execucao. Goodbye!!!
...Eu sou o processo PAI de PID=5481 e meu filho tem o PID=5482
...Terminando a minha execucao. Goodbye!!!
```

Se um filho vive (continua executando) enquanto seu pai está morto, ele se transformará em um processo orfão tendo como novo pai o processo "init". Veja o exemplo de programa a seguir:

Exemplo:

```
==========================================================================
/* arquivo exemplo_04.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("INICIANDO EXECUCAO DO PROGRAMA EXEMPLO_04.C \n");
   printf("PAI- Eu sou o pai %d e eu vou criar um filho \n",getpid()) ;
   pid=fork() ;        /* criacao do filho */
   if(pid == 0)        /* acoes do filho */
   {
      printf("\tFILHO- Oi, eu sou o processo %d, o filho\n",getpid()) ;
      printf("\tFILHO- O dia esta otimo hoje, nao acha?\n") ;
      printf("\tFILHO- Bom, desse jeito vou acabar me instalando para sempre\n");
      printf("\tFILHO- Ou melhor, assim espero!\n") ;
      for(;;) ;        /* o filho se bloqueia num loop infinito */
   }
   else                /* acoes do pai */
   {
       sleep(3) ;      /* para separar bem as saidas do pai e do filho */
       printf("PAI- As luzes comecaram a se apagar para mim, %d\n",getpid()) ;
       printf("PAI- Minha hora chegou : adeus, %d, meu filho\n",pid) ;
       exit(0);        /* e o pai morre de causas naturais */
    }
}
==========================================================================
```

Resultado da execução:

```
Server01:~> ./exemplo_04
INICIANDO EXECUCAO DO PROGRAMA EXEMPLO_04.C
PAI- Eu sou o pai 5995 e eu vou criar um filho
        FILHO- Oi, eu sou o processo 5996 o filho
        FILHO- O dia esta otimo hoje, nao acha?
        FILHO- Bom, desse jeito vou acabar me instalando para sempre
        FILHO- Ou melhor, assim espero!
PAI- As luzes comecaram a se apagar para mim, 5995
PAI- Minha hora chegou : adeus, 5996, meu filho
```

Se executarmos os comando "ps –f" na console:

```
UID    PID   PPID STIME  TTY  TIME  COMMAND
User1  5874  5201 10:27 tty2  0:00  bash
User1  5996     1 10:27 tty2  0:00  exemplo_04
User1  6008  5874 10:27 tty2  0:00  ps -f
```

veremos que o filho continua executando. Podemos matá-lo com o comando "kill"

```
Server01:~> kill -9 5996
```

Exercício 01: Criar um programa "exercicio_01.c" que internamente emita um comando "**ls –l**", mas que o processo original continue executando, isto é, ele não deve ser substituído pelo comando emitido.

Exercício 02: Elaborar um programa "exercicio_02.c" que durante sua execução emita as seguintes mensagens (não necessariamente nesta ordem):

```
PAI-EXECUCAO DO PROGRAMA EXEMPLO_02.C
PAI-EU SOU O PROCESSO PID=5903 DE PAI PPID=5564
        FILHO-EU SOU O PROCESSO PID=5904 DE PAI PPID=5903
        FILHO-AGORA EU VOU CRIAR MAIS UM FILHO
                NETO-EU SOU O PROCESSO PID=5905 DE PAI PPID=5904
                NETO-AGORA VOU TERMINAR. BYE. PID=5905
        FILHO-AGORA VOU TERMINAR TAMBEM. TCHAU. PID=5904
PAI-VOU ENCERRAR NORMAL. PID=5903
```

✓ Primitiva wait()

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, a função retorna imediatamente. O código de retorno devolvido é o número do processo (PID) do processo morto ou -1 em caso de erro.

Seu formato é: codret = wait(&status)

Onde status é uma variável tipo *int* ou é NULL. Se status é não nulo (NULL), wait armazena a informação relativa a razão da morte do processo filho, sendo apontada pelo ponteiro status. 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: no exemplo abaixo (exemplo_05.c) a execução do programa deverá ser feita em *background*, no seguinte formato:

```
Server01:~> ./exemplo_01 &
```

```
================================================================================
/* arquivo exemplo_05.c */
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
   int rcode ;
   printf("\n\nPAI- Ola, sou o processo EXEMPLO_05 de PID=%d.\n",getpid());
   printf("\nPAI- Vou gerar o meu primeiro filho...\n");
   rcode = fork();
   if (rcode == 0) {
      printf("\n\tFILHO1- Oi, eu sou %d, o filho de %d.\n",getpid(),getppid());
      sleep(3);
      printf("\n\tFILHO1- Estou executando normalmente...\n");
      printf("\n\tFILHO1- Pois e, Chegou minha hora! Encerrando com exit().\n");
      return 7;
   }
   else {
      int ret1, status1, ret2;
      printf("\nPAI- Olha ai, meu primeiro filho %d esta executando...\n",rcode);
      ret1 = wait(&status1);
      if ((status1&255) == 0) {
         printf("\nPAI- Meu primeiro filho morreu. Valor meu wait():%d\n",ret1);
         printf("\nPAI- Ele morreu com um exit de codigo %d.\n",(status1>>8));
      }
      else {
         printf("\nPAI- Meu filho nao foi morto por um exit.\n");
      }
      printf("\nPAI- Sou eu ainda, o processo %d.\n", getpid());
      printf("\nPAI- Vou gerar agora meu segundo filho\n");
      ret2 = fork();
      if (ret2 == 0) {
         printf("\n\tFILHO2- Sou %d, segundo filho de %d\n",getpid(),getppid());
         sleep(3);
         printf("\n\tFILHO2- Eu nao quero seguir o exemplo de meu irmao!\n");
         printf("\n\tFILHO2- Nao vou morrer... Vou ficar num loop eterno!\n");
         for(;;);
      }
      else {
         int ret3, status3, s;
         printf("\nPAI- Sou eu ainda (de novo), o processo %d.\n", getpid());
         ret3 = wait(&status3);
         if ((status3&255) == 0) {
            printf("\nPAI- Meu segundo filho nao foi morto por um sinal!!\n");
         }
         else {
            printf("\nPAI- Valor de retorno do wait(): %d\n",ret3);
            s = status3&255;
            printf("\nPAI- O sinal que matou meu filho foi: %d\n",s);
            printf("\nPAI- Agora sou eu que vou partir... Tchau!\n");
         }
      }
   }
   return 0;
}
================================================================================
```

Durante a execução do programa acima, quando o segundo filho der a mensagem que entrou em um loop infinito, devemos dar um comando "ps –f" para verificarmos os processos ativos. Em seguida devemos cancelar com o comando "kill" esse processo filho e analisar o comportamento do processo pai original. Podemos concluir com essa execução que:

- 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.

✓ Primitiva waitpid()

A função waitpid suspende a execução do processo até a morte de um filho específico ou simplemesmente verifica o status de um processo filho sem bloquear o processo pai.

Seu formato é: codret = waitpid(pid, &status, option)

Onde:

- pid: número do PID do processo filho específico ou -1 para todos os filhos;
- &status: idêntico à primitiva wait();
- option:
  0: bloqueia o processo pai até o término do filho;
  WNOHANG: não bloqueia o processo pai. Se o filho ainda não terminou, retorna 0.