Processos

O UNIX é um sistema operacional *multitarefa* (suporta *multiprogramação*). Isso significa que ele é capaz de gerenciar a execução simultânea de vários programas. O termo *processo* é usado para caracterizar a *ativação* de um programa. É importante, neste ponto, caracterizar bem a distinção entre *programa* e processo:

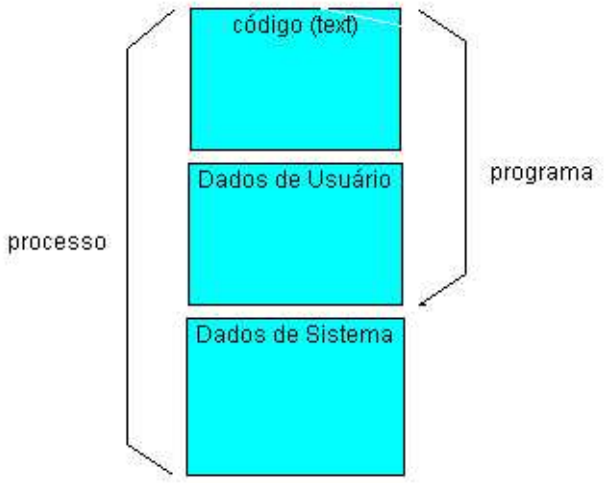
- um programa define estaticamente um padrão de comportamento através das instruções de máquina.
- um processo é um programa em execução (além das instruções de máquina, um processo é caracterizado por um *estado*, definido pelos dados com os quais o mesmo opera e pela instrução sendo executada num dado instante).
- um mesmo programa pode dar origem a mais de um processo.

Estrutura de um processo

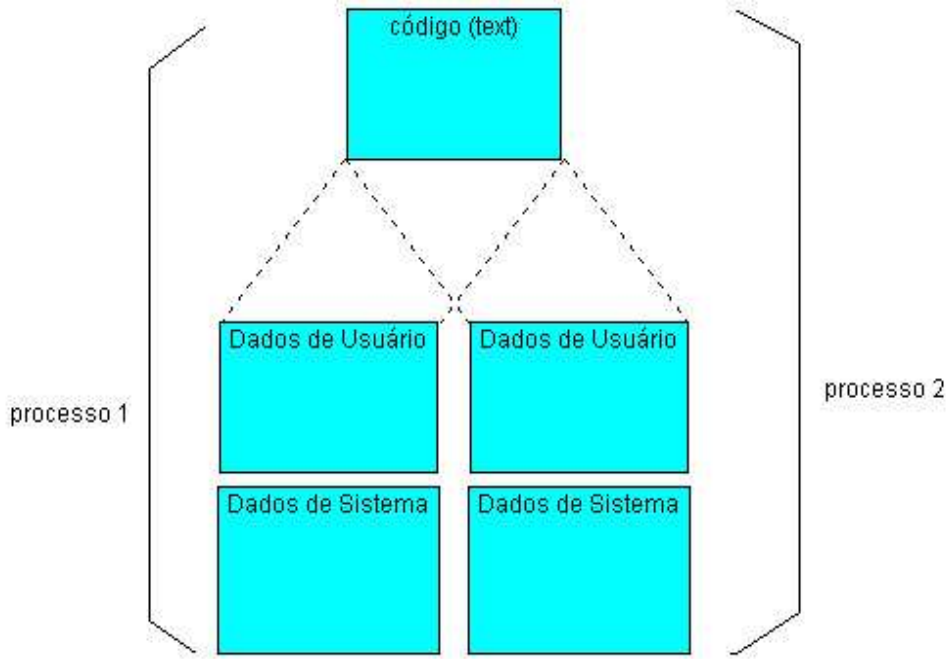
Um processo sob UNIX ocupa uma área de memória formada basicamente por 3 partes:

- segmento de código (*text segment*)
- segmento de dados de usuário (*user data segment*).
- segmento de dados de sistema (*system data segment*)

O segmento de código contém basicamente as instruções de máquina geradas ao se compilar o programa. Alguns compiladores mantém nesse segmento as constantes definidas no programa. O segmento de dados de usuário contém as variáveis declaradas pelo usuário no seu programa. O segmento de dados de sistema mantém informações utilizadas pelo sistema operacional necessárias ao gerenciamento do processo. Nesse segmento são mantidos os descritores de arquivos usados pelo processo.



Um mesmo programa pode ser ativado mais de uma vez, dando origem a vários processos. Nesse caso o segmento de código (*text segment*) é compartilhado pelos mesmos, conforme mostrado na figura abaixo.



Identificação de um processo

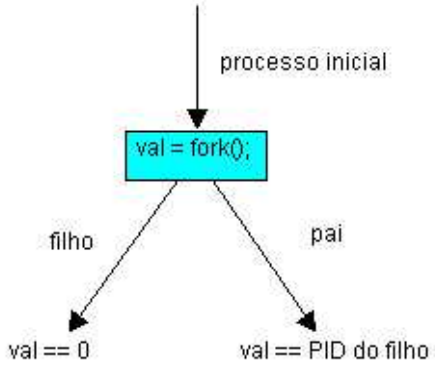
Todo processo gerenciado pelo UNIX tem uma identificação, *PID - process id*, definida por um número inteiro único no sistema. Essa identificação é utilizada nas *system calls* voltadas ao gerenciamento, comunicação e sincronização entre processos.

Criação de um processo

No Unix, a criação de um processo é feita através da chamada ao systema *fork()*: através dela, o processo que a executa é duplicado. O novo processo, chamado *processo filho* é executado em paralelo com o *processo pai*. A função *fork()* retorna um valor diferente para cada um dos processos:

- o valor de retorno para o processo pai é a identificação (PID) do processo filho.
- o valor de retorno para o processo filho é 0.

O objetivo desse valor diferente em cada caso é permitir que os dois processos possam ter comportamentos diferentes após a execução da operação *fork()*. A figura abaixo ilustra o funcionamento de *fork()*:



O programa abaixo, fork1.c, mostra o uso da chamada *fork()*:

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

main()
{
  pid_t val;

  printf("PID antes do fork(): %d\n", (int)getpid());

  if(val = fork())
    printf("PID do processo pai: %d\n",(int)getpid());
  else
    printf("Child do processo filho: %d\n",(int)getpid());
}
```

O exemplo abaixo, fork2.c, permite observar a execução "simultânea" dos dois processos:

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

main()
{
  int i;
  int n = 10;
  pid_t val;

  if(val = fork())
    for(i=1; i < n; i++)
      {
        sleep(random()%3); printf("=====> i: %d\n",i);
      }
  else
    for(i=n; i >0; i--)
      {
        sleep(random()%3); printf("==============> i: %d\n",i);
      }
}
```

Sincronização dos processos através de *wait()*

A chamada ao sistema *wait()* bloqueia um processo até que o seu *processo filho* encerre sua execução:

```
#include < sys/types.h>
#include < unistd.h>
...
pid_t wait(int* status);
```

```
...
```

O valor de retorno de *wait()* é a identificação (PID) do processo filho que encerrou sua execução. O parâmetro *status* é usado para devolver o "status" de finalização do processo. Um exemplo de uso de *wait()* é mostrado a seguir (fork3.c):

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

main()
{
  int i, status;
  int n = 10;
  pid_t val, wval;

  if(val = fork())
    {
      wval = wait(&status);
      for(i=1; i < n; i++) printf("pai ==> i: %d\n",i);

    }
  else
    for(i=n; i >0; i--)
       printf("filho =========> i: %d\n",i);

}
```

A função *exec()*

A chamada *fork()*, da forma como apresentada serve apenas para "clonar" um processo, o que é de utilidade limitada. Normalmente *fork()* é usada em conjunto com a chamada *exec()*, que permite que um processo passe a executar o código de outro programa. A função *exec()* é disponível sob várias formas. Por ora, vamos nos restringir a uma delas, que é chamada de *execv()*:

```
#include < unistd.h>
...
int execv(char* pathname, char* argv[]);
...
```

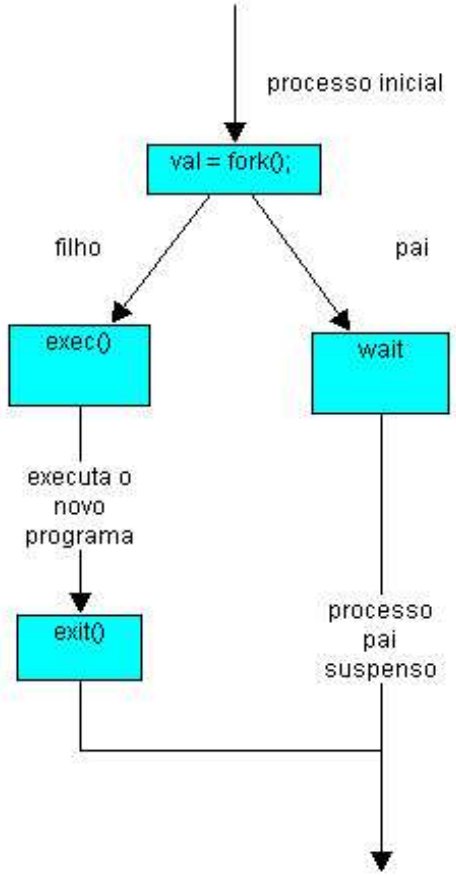
Um exemplo

O programa mostrado a seguir, *run.c*, usa *execv()* para executar um um programa cujo nome é passado como argumento na linha de comando. Eventuais argumentos adicionais são passados também na linha de comando.

```
int main(int argc, char* argv[])
{
  if(argc == 1)
    {
      printf("uso: run  [  ]\n");
      exit(1);
    }

  execv(argv[1],&argv[1]);
  printf("não é possível executar %s\n",argv[1]);
}
```

A figura abaixo ilustra a situação mostrada no exemplo.



Exemplos de uso

```
./run  /bin/ls   programa1.c
./run  /bin/cat  programa1.c
./run  /bin/sort  arq2
```

Outro exemplo

A *shell()* utiliza *exec()* para executar os comandos que lhe são passados pela entrada padrão. O programa abaixo, nanosh.c, ilustra esse tipo de utilização.

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

int myexec(char* nome, char* argv[])
{

  if(strcmp(nome,"dir")==0) nome = "/bin/ls";
  else
    if(strcmp(nome,"type")==0) nome = "/bin/cat";
  else
    if(strcmp(nome,"copy")==0) nome  = "/bin/cp";
  else
    if(strcmp(nome,"exit")==0) exit(0);

  execv(nome,&argv[0]);
  printf("nao e' possivel executar %s\n",nome);
}

int main(int argc, char* argv[])
{
  int i, status;
  pid_t val, wval;

  if(val = fork()) wval = wait(&status);
  else myexec(argv[1],&argv[1]);
}
```