INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

ECM225 – Sistemas Operacionais

Atividade – Processos e Threads em Linux – I

Prof. Marco Furlan

14 de março de 2019

Nota: O nome dos programas C desta atividade aparecem na primeira linha de comentário dentro do texto do programa. Para compilar um programa `prog.c`, utilizar o comando a seguir, na linha de comando: `gcc -o prog prog.c`. Para executar o programa na linha de comando, executar: `./prog`.

1 Programas e processos

Um **programa** é uma sequência de instruções preparadas para realizar alguma tarefa. Um **processo** é uma **instância** de um programa em execução, e que **possui** seu próprio **espaço de endereço** e **estado de execução**. Um programa torna-se um processo quando o sistema operacional carrega o programa na memória.

Podem haver vários processos de um mesmo programa – cada um possui um **identificador** diferente. Um **processo** possui, no mínimo, um fluxo de controle denominado **thread**, responsável pela execução das instruções. Processos podem ser executados **concorrentemente** e comunicar entre si, por diversas técnicas.

Um programa em execução (processo) no Linux possui pode ser visualizado conforme a Figura 1 onde (no exemplo, 32 bits de endereçamento):

- O **segmento de texto** contém as instruções de **código de máquina do programa** sendo executado pelo processo. O segmento de texto possui **acesso apenas de leitura**, evitando que o programa acidentalmente modifique suas próprias instruções por meio de um valor inválido de ponteiro, por exemplo. **Vários processos** podem estar **executando** o **mesmo programa**, então o **segmento** de **texto** é **compartilhado** de modo que apenas uma única cópia do código do programa possa ser mapeada para o espaço de endereçamento virtual de todos seus processos;
- O **segmento de dados inicializado** contém **variáveis globais e estáticas** que foram **explicitamente inicializadas**. Os valores dessas variáveis são lidos do arquivo executável quando o programa é carregado na memória;

- O **segmento de dados não inicializados** contém variáveis globais e estáticas que não são explicitamente inicializadas. Antes de iniciar o programa, o sistema inicializa toda a memória deste segmento com zeros.
- A **pilha** é um segmento de memória que pode aumentar ou diminuir contendo ***stack frames***. Um *stack frame* é criado para cada **chamada de função** e armazena as variáveis locais da função, argumentos e valor de retorno;
- O ***heap*** é uma área na qual variáveis podem ser alocadas dinamicamente em tempo de execução. O topo final do *heap* é denominado de ***program break***.

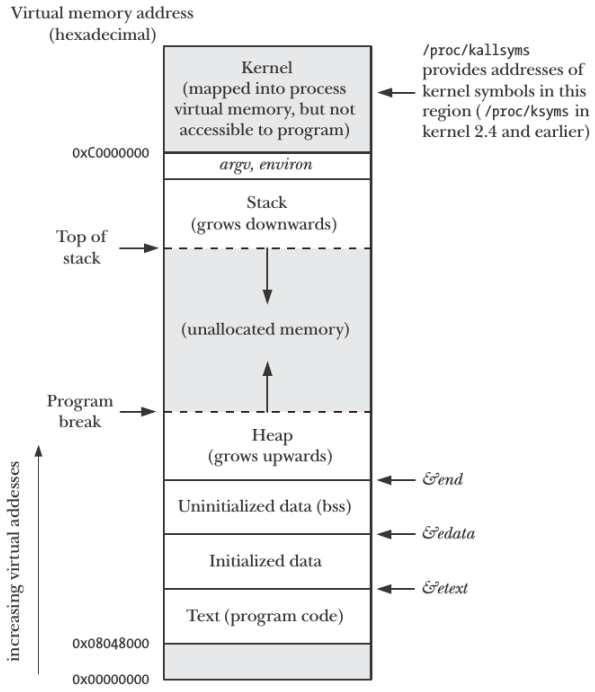

Figura 1: Leiaute de um processo Linux

2 Manipulação de processos no Linux

2.1 Identificadores de processos

No UNIX e Linux, todo processo possui um número inteiro que o identifica – **ID do processo**. Existe, ainda, o **ID** do **processo-pai** que o iniciou. Funções para determinar o ID de processos:

```
#include <unistd.h>
/*ID do processo*/
pid_t getpid(void);
/*ID do processo-pai*/
pid_t getppid(void);
```

Compile e execute o exemplo a seguir (`procid.c`):

```c
/*procid.c*/
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    printf("Meu ID de processo e %ld\n", (long)getpid());
    printf("ID do processo-pai e %ld\n", (long)getppid());
    return 0;
}
```

2.2 Identificadores de usuários e grupos

Cada processo no UNIX ou Linux possui vários identificadores de usuários e grupos:

- **ID real do usuário**: aquele que foi determinado para o usuário pelo administrador do sistema;
- **ID real do grupo**: aquele que foi determinado para o usuário pelo administrador do sistema;
- **ID efetivo do usuário**: normalmente o mesmo que o real, porém o processo pode mudá-lo em algumas circunstâncias. Exemplo: um programa que é executado com privilégios de **root** pode querer criar um arquivo como se fosse um usuário comum;
- **ID efetivo do grupo**: a mesma explicação que a do usuário efetivo.

Funções para determinar usuários e grupos:

```c
#include <unistd.h>
gid_t getegid(void);    /*ID efetivo do grupo*/
uid_t geteuid(void);    /*ID efetivo do usuário*/
gid_t getgid(void);     /*ID real do grupo*/
uid_t getuid(void);     /*ID real do usuário*/
```

Compile e execute o exemplo a seguir (`procuid.c`):

```c
/*procuid.c*/
#include<stdio.h>
#include<unistd.h>

int main(void)
{
    printf("Meu ID real de usuario e %5ld\n", (long)getuid());
    printf("Meu ID efetivo de usuario e %5ld\n", (long)geteuid());
    printf("Meu ID real de grupo e %5ld\n", (long)getgid());
    printf("Meu ID efetivo de grupo e %5ld\n", (long)getegid());
    return 0;
}
```

2.3 Estados de um processo

No Linux, todos os processos seguem um diagrama de estados semelhante ao da Figura 2, onde cada estado pode ser assim descrito:

- **Novo**: representa quando um programa está sendo carregado na memória, transformando-se em um processo;
- **Pronto**: é a situação quando o processo está na fila de processos do sistema operacional aguardando ser executado;
- **Execução**: é a situação quando o processo é selecionado da fila de processos pelo escalonador e está em execução;
- **Bloqueado**: ocorre quando o processo está aguardando por um evento (ex.: E/S) e não pode ser selecionado para ser executado – essa situação pode ser forçada pela função **`sleep()`**;
- **Terminado**: é o estado de término de um processo – de modo normal ou anormal.

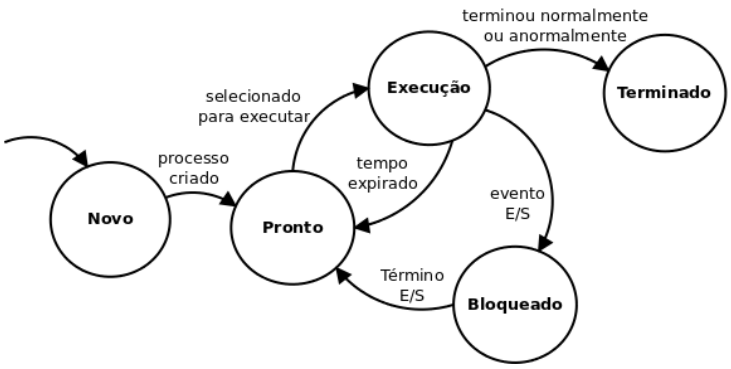

Figura 2: Estados de um processo no Linux

2.4 Criação de processos

A função **`fork()`** cria um novo processo. O processo que chamou torna-se o processo "pai" e o novo torna-se o processo "filho":

```
#include <unistd.h>
pid_t fork(void);
```

Essa função copia a imagem da memória de modo que o novo processo recebe uma cópia do espaço de endereçamento do processo pai. Ambos os processos continuam suas execuções a partir do ponto de chamada de **`fork()`**. O retorno de **`fork()`** é sempre **0 (zero)** para o **processo filho**. O valor de PID para o processo filho é único e o valor do **PID** do **processo pai do filho** é o PID do processo pai que se executou o fork. No caso de erro, **`fork()`** não cria o processo filho e retorna o valor -1.

Exemplo de execução de **`fork()`** (`forkintro.c`):

```c
/*forkintro.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
    pid_t pidFilho = fork();
    if(pidFilho == -1){
        perror("Falhou no fork()!");
        return EXIT_FAILURE;
    }
    if(pidFilho == 0) {
        printf("Aqui se executa o código do filho...\n");
        printf("PID filho: %ld\n", (long)getpid());
    }
    else {
        printf("Aqui se executa o código do pai...\n");
        printf("PID pai: %ld\n", (long)getpid());
        printf("PID do processo filho retornado por fork(): %ld\n", (long)pidFilho);
    }
    return EXIT_SUCCESS;
}
```

Observar a lógica de **`fordk()`** quanto a atribuição de variáveis `forkatrib.c`:

```c
/*forkatrib.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
    int x;
    x = 0;
    fork();
    ++x;
    printf("ID do processo: %ld, valor de x = %d\n",
                (long)getpid(), x);
    return 0;
}
```

2.5 Sincronização de processos

Quando um processo cria um filho, ambos procedem com a execução a partir do **`fork()`**. O processo pai pode executar a função **`wait()`** ou **`waitpid()`** para bloquear o processo pai até que o processo filho termine. As interfaces dessas funções são:

```c
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

As funções **`wait()`** e **`waitpid()`** retornam o ID do processo filho ou -1 se houver erro. O parâmetro de **`wait()`** é um ponteiro para uma variável que armazenará o estado da execução ou **`NULL`** caso não se deseje armazenar tal valor. O estado da execução é um número inteiro que representa uma situação de término do filho e que pode ser descoberta por conjunto de macros (vide `man 2 wait`). Os parâmetros de **`waitpid()`** são o pid do filho que se aguardará, um ponteiro para uma variável que armazenará o estado da execução (como em **`wait()`**) e um conjunto de opções (vide `man 2 waitpid`). O valor do parâmetro pid influi no modo como o processo pai aguardará a execução dos processos filho , nas seguintes condições:

- Se pid for -1, o processo pai aguardará qualquer filho;
- Se pid for 0, o processo pai aguardará todos os filhos do mesmo grupo de processos[1];
- Se pid for < -1, o processo pai aguardará todos os filhos do grupo de processos do valor absoluto de pid;
- Se pid for > 0, o processo pai aguardará o processo de número pid específico.

No exemplo a seguir (`forkwait.c`) o processo pai aguarda o término de todos os processos filho.

```c
/*forkwait.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[]){
    pid_t childpid = 0;
    int i, n;
    if(argc != 2){
        fprintf(stderr, "Uso: %s numero\n", argv[0]);
        return EXIT_FAILURE;
    }
    n = atoi(argv[1]);
    for(i = 1; i < n; i++)
        if((childpid= fork()) <= 0) break;
    while( wait(NULL) > 0 );
    fprintf(stderr, "i: %d - ID: %ld -  ID pai: %ld - ID filho: %ld\n",
                i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```

Compare o resultado da execução deste programa com o programa `forkfan`.

2.6 Executando programas externos em processos

A função **`execlp()`** é uma das funções (vide `man 3 execlp`) que permite a execução de um programa externo dentro de algum processo em execução:

[1] Um grupo de processos é uma coleção de um ou mais processos associados a uma sessão

```c
#include <unistd.h>
int execlp(const char *filename, const char *arg0, ...);
```

Esta função substitui a imagem do processo atual por uma nova imagem de processo. Nesta função passa-se no argumento `filename` o nome da imagem (programa em disco) que se deseja executar e, em seguida, obrigatoriamente o nome do programa a ser executado seguido opcionalmente por seus argumentos de linha de comando, terminados por `NULL`.

O programa a seguir (`exec.c`) executa um comando que pode ser escolhido pelo usuário (observe atentamente o que será produzido em tela):

```c
/*exec.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    static char *cmd[]={"who","ls","date","bongobongo"};
    int i;
    printf("0=who,1=ls,2=date,3=bongobongo: ");
    scanf("%d", &i);
    execlp(cmd[i],cmd[i],0);
    printf("command not found\n");
    return EXIT_SUCCESS;
}
```

A seguir, tem-se um exemplo de outra função similar, **`execl`** junto com a função **`wait`**:

```c
/*execwait.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("Falhou fork!");
       return 1;
   }
   if (childpid == 0) {/*codigo do processo filho*/
       execl("/bin/ls", "ls", "-l", NULL);
       perror("Filho não executou ls!");
       return 1;
   }
   if (childpid != wait(NULL)) {/*codigo do pai*/
       perror("Falha no pai - provocado por sinal ou erro");
       return 1;
   }
```

```
    return 0;
}
```

A diferença entre **execlp** e **execl** é que a primeira assume que o programa a ser executado está em um dos diretórios do caminho do sistema enquanto que o segundo exige que seja informado o caminho do programa.

3 Tarefas

(3.1) Pode-se criar processos em diferentes estilos. Uma **cadeia de processos** é criada quando um processo cria outro processo que, por sua vez cria outro e assim por diante, conforme o programa a seguir (`forkchain.c`):

```
/*forkchain.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[]){
    pid_t childpid = 0;
    int i, n;
    if(argc != 2){
        fprintf(stderr, "Uso: %s numero\n", argv[0]);
        return EXIT_FAILURE;
    }
    n = atoi(argv[1]);
    for(i = 1; i < n; i++)
        if(childpid = fork())
            break;
    fprintf(stderr, "#: %d - ID: %ld -  ID pai: %ld - ID filho: %ld\n",
            i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```

Um outro estilo para criar processos é criar um **leque de processos**, onde um único processo pai é o responsável direto por criar vários processos filho, como no programa a seguir (`forkfan.c`):

```
/*forkfan.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc, char *argv[]){
    pid_t childpid = 0;
    int i, n;
    if(argc != 2){
        fprintf(stderr, "Uso: %s numero\n", argv[0]);
        return EXIT_FAILURE;
```

```
        }
        n = atoi(argv[1]);
        for(i = 1; i < n; i++)
            if((childpid = fork() ) <= 0)
                break;
        fprintf(stderr, "#: %d - ID: %ld -  ID pai: %ld - ID filho: %ld\n",
                    i, (long)getpid(), (long)getppid(), (long)childpid);
        return 0;
    }
```

Pede-se: explicar as diferenças na criação de processos entre os programas `forkchain` e `forkfun`. Faça um desenho que simule as execuções dos programas para um número de processos igual à 4, por exemplo.

(3.2) O que acontece com o programa `forkwait.c` se as instruções **`while`** e **`fprintf`** forem trocadas de lugar entre si?

(3.3) Execute o programa a seguir:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <signal.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  int main (void) {
8      pid_t childpid;
9      childpid = fork();
10     if (childpid == -1) {
11         perror("Failed to fork");
12         return 1;
13     }
14     if (childpid == 0)
15         fprintf(stderr, "Filho - PID = %ld\n", (long)getpid());
16     else if (wait(NULL) != childpid)
17         fprintf(stderr, "Sinal interrompeu wait!\n");
18     else
19         fprintf(stderr, "Pai - PID = %ld com filho %ld\n", (long)getpid(), (long)childpid);
20     return 0;
21 }
```

O que deve acontecer na execução deste programa para que as linhas 11, 15, 17 e 19 sejam executadas?

(3.4) Escreva um programa que execute um leque de processos e então aguarde o término de cada um deles, identificando um a um aqueles que forem terminando. Sugestão: adapte o programa `forkwait.c`.

(3.5) Como identificar o estado do término de um processo filho (o parâmetro de **`wait()`**)? Pesquise no manual.

(3.6) Pesquise e explique as funções da família **`execxx`**.