Universidade Federal de Itajubá
SRSC02 – Sistemas Operacionais
Prof. Minoru

Exercício Prático 01 - EP01 Processos

1 Recomendações

- Para os programas utilize das recomendações de boas práticas de programação disponível no SIGAA tópico "Apresentação da disciplina (14/08/2024)".
- Para as questões, as respostas devem ser **completas** e **justificadas**. Poderá utilizar o Word ou o Writer para as respostas, utilize o cabeçalho (acima) da proposta do exercício para o documento das respostas, não esqueça de colocar o nome e número de matrícula substituindo o nome do professor. Antes de entregar salve o documento no formato PDF.
- **TODOS** os programas fonte deverão ser entregues, inclusive os que não funcionaram nos micros do laboratório devido a segurança.
- Para a entrega você deverá compactar somente os arquivos **PDF e .C**, não compacte diretórios.
- É necessário que somente um dos componentes do grupo entregue o arquivo ZIP no SIGAA.

2 Criação de Processos

Nos sistemas Linux e Unix, `fork()` é o nome da system call usada para se criar um *novo* processo (usando C/C++). O novo processo é chamado de *processo filho* (*child*), enquanto que o processo original é chamado de *processo pai* (*parent*). Por padrão, o processo filho é uma duplicata do processo pai. Isso significa que o filho possui o mesmo código do pai, porém, o espaço de memória dos dois processos é separado.

A sintaxe do comando `fork()` é:

```
#include <unistd.h>

pid_t fork(void);
```

Para ilustrar a criação de um processo, considere o Código 1, a seguir:

Após digitar o Código 1, salve-o e abra uma janela de terminal na mesma pasta em que o código está salvo. Use os seguintes comandos para compilar e executar o código:

```
aluno@lab:~$ gcc -o saida exemplo1.c
aluno@lab:~$ ./saida
```

Exercício 1: Responda: o Código 1 executou corretamente?

Código 1: `exemplo1.c`

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main()
5  {
6      printf("inicio do codigo\n");
7
8      fork();
9
10     printf("fim do codigo\n");
11
12     return 0;
13 }
```

Exercício 2: Comente o resultado da execução do Código 1.

A função `fork()` não recebe nenhum parâmetro (`void`) e devolve um inteiro longo não sinalizado (`unsigned long int`), representado pelo tipo `pid_t`. O tipo `pid_t` é definido no arquivo de cabeçalho `sys/types.h`, que é incluído pelo arquivo de cabeçalho `unistd.h`. Este último contém a declaração da função `fork()`.

Em caso de falha na criação de um processo, `fork()` retorna o valor −1. Em caso de sucesso, seu valor retornado varia: para o processo filho, `fork()` retorna 0. Já para o processo pai, `fork()` retorna o PID do processo filho recém criado.

No Código 1, após a invocação de `fork()` na linha 8, ambos os processos passam a executar o mesmo código. Por esta razão, a mesma mensagem presente na linha 10 foi impressa duas vezes: o processo pai, ao continuar a sua execução, imprimiu e finalizou. O mesmo ocorreu com o processo filho. É tarefa do programador ajustar o código de modo que os dois processos realizem ações diferentes.

Para ilustrar esta separação, seja o Código 2. Este código faz uso das funções `getpid()` e `getppid()`. A primeira função devolve o PID do processo que está em execução. A segunda função devolve o PID do processo que é o pai do processo em execução. O retorno de ambas é do tipo `pid_t`.

Vamos observar o resultado de duas execuções do Código 2:

```
aluno@lab:~$ gcc -o saida exemplo2.c
aluno@lab:~$ ./saida
inicio do codigo
este eh o processo filho
o PID do filho eh: 13525
o PID de seu pai eh: 13524
codigo comum a pai e filho
este eh o processo pai
o PID do pai eh: 13524
o PID do filho gerado eh: 13525
codigo comum a pai e filho
aluno@lab:~$
```

```
aluno@lab:~$ ./saida
inicio do codigo
```

Código 2: `exemplo2.c`

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

int main()
{
    pid_t child_pid;

    printf("inicio do codigo\n");

    child_pid = fork();

    if(child_pid < 0)
    {
        fprintf(stderr, "falha na criacao do processo...\n");
        return 1;
    }
    else if(child_pid == 0)
    {
        printf("este eh o processo filho\n");
        printf("o PID do filho eh: %d\n", getpid() );
        printf("o PID de seu pai eh: %d\n", getppid() );
    }
    else
    {
        printf("este eh o processo pai\n");
        printf("o PID do pai eh: %d\n", getpid() );
        printf("o PID do filho gerado eh: %d\n", child_pid );
    }

    printf("codigo comum a pai e filho\n");

    return 0;
}
```

```
este eh o processo pai
o PID do pai eh: 13599
o PID do filho gerado eh: 13600
codigo comum a pai e filho
este eh o processo filho
o PID do filho eh: 13600
o PID de seu pai eh: 13599
codigo comum a pai e filho
aluno@lab:~$
```

Se o Código 2 for executado várias vezes, a ordem em que as mensagens serão mostradas no terminal irá diferir. Uma vez que há dois processos concorrentes, o processador da máquina pode ser atribuído a qualquer um deles podendo, inclusive, haver *preempção* entre eles. De forma análoga, os valores de PID de ambos os processos irão diferir entre máquinas e execuções distintas.

Exercício 3: Escreva um programa usando a system call `fork()` para criar dois filhos do mesmo processo, isto é, o Pai P possui P1 e P2 como processos filhos.

Exercício 4: Escreva um programa usando a system call `fork()` para criar uma hierarquia de 3 processos, isto é, P2 é filho de P1, e P1 é filho de P.

Exercício 5: Responda: o que a system call `fork()` retorna em caso de sucesso?

Exercício 6: Responda: qual o PID de um processo filho?

Exercício 7: Responda: qual a função usada para se obter o PID de um processo?

Exercício 8: No total, quantos processos são criados no código abaixo?

```
int main()
{
    fork();
    fork();
}
```

Exercício 9: Descreva o que acontece se os processos pai e filho alteram o valor de uma variável declarada no código.

3 System calls `wait()` e `sleep()`

Em muitas aplicações, quando um processo pai cria um processo filho, é interessante que o pai possa monitorar o filho e saber como ele terminou sua execução. Uma das maneiras de se realizar isso é através da system call `wait()`.

A função `wait()` faz com que o processo que a invoca aguarde pelo término de um de seus filhos e seta o estado de seu término em um buffer referenciado pelo parâmetro `status`. O retorno da função é o PID do processo que terminou, como segue:

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

pid_t wait(int *status);
```

A system call `wait()` realiza as seguintes operações:

1. Caso nenhum filho do processo pai tenha terminado sua execução, a chamada de `wait()` bloqueia o pai, até que algum filho encerre.
2. Se algum filho do processo pai já encerrou sua execução no momento da invocação de `wait()`, `wait()` retorna imediatamente.
3. Se o parâmetro `status` for diferente de `NULL`, a informação sobre o término do filho é gravada no inteiro apontado por `status`.
4. Se o parâmetro `status` for `NULL`, `wait()` retorna assim que o primeiro filho finalizar.
5. O kernel adiciona às estatísticas do processo pai o tempo total de CPU e o uso de recursos de todos os seus filhos.

Os tempos de execução de um processo pai e de seus processos filhos nem sempre são os mesmos: frequentemente o tempo de execução do pai supera o dos filhos ou vice-versa. Dessa maneira, temos duas questões a serem verificadas:

1. Se o pai finaliza antes de algum filho, quem se torna o pai de um "filho órfão"?
2. O que acontece se um filho finaliza antes que o pai realize a invocação de `wait()`?

Para a questão 1, no Linux, todos os "filhos órfãos" são adotados pelo processo *init*, que é o ancestral de todos os outros processos da máquina, com PID = 1 (a depender da distribuição, também pode ser o processo *systemd* – um dos filhos de *init* – também responsável pela inicialização de grande parte do sistema). Dessa forma, após o pai encerrar sua execução, a invocação da função `getppid()` no filho retornará 1. Note que este recurso pode ser usado para descobrir se o pai verdadeiro de um processo se encerrou ou não.

Para a questão 2, mesmo que o filho tenha encerrado, o pai deve ter a oportunidade de invocar a função `wait()` para determinar o status de encerramento do filho. O kernel do SO lida com essa situação transformando o processo filho em um *processo zumbi*. Isso significa que a maioria dos recursos alocados ao filho são retomados pelo SO, podendo ser alocados a outros processos. As únicas informações do processo zumbi que se mantêm são a sua entrada na tabela de processos do kernel (PID), seu status de encerramento e suas estatísticas de execução.

Exercício 10: Como verificar, em código, se o pai verdadeiro de um processo já se encerrou ou não?

O Código 3 ilustra o caso de um processo filho "órfão". Na execução do filho, em sua linha 10, o Código 3 invoca a função `sleep()`, fazendo que o filho espere por 3 segundos, antes de continuar. Este tempo é suficiente para que o pai encerre sua execução e o filho torne-se "órfão".

Código 3: `exemplo3.c`

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main()
5  {
6      pid_t child_pid = fork();
7
8      if(child_pid == 0)
9      {
10         sleep(3);
11         printf("PID do pai adotante: %d\n", getppid() );
12     }
13     else
14     {
15         printf("PID do pai verdadeiro: %d\n", getpid() );
16     }
17
18     return 0;
19 }
```

O resultado da execução do Código 3 pode ser visto a seguir:

```
aluno@lab:~$ gcc -o saida exemplo3.c
aluno@lab:~$ ./saida
PID do pai verdadeiro: 14205
PID do pai adotante: 1423
aluno@lab:~$
```

Observação: não se deve usar `wait()` e `sleep()` para fazer com que um processo execute antes de outro. A sincronização de processos será abordada futuramente na disciplina.

Já a criação de um processo zumbi pode ser observada na execução do Código 4, cujo resultado da execução é mostrado a seguir:

```
aluno@lab:~$ gcc -o saida exemplo4.c
aluno@lab:~$ ./saida
PID do pai: 16211
PID do filho: 16212
   16211 pts/0    00:00:00 saida
   16212 pts/0    00:00:00 saida <defunct>
aluno@lab:~$
```

Código 4: `exemplo4.c`

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h> // system()
4
5  int main(int argc, char *argv[])
6  {
7      char cmd[100];
8      pid_t child_pid;
9
10     printf("PID do pai: %d\n", getpid() );
11
12     child_pid = fork();
13
14     if(child_pid == 0)
15     {
16         printf("PID do filho: %d\n", getpid() );
17     }
18     else
19     {
20         sleep(3);
21         sprintf(cmd, "ps | grep -i %s", &argv[0][2]); // vc entendeu esta linha?
22         system(cmd);
23     }
24
25     return 0;
26 }
```

O Código 4 inicia imprimindo o PID do processo pai e então cria um processo filho (linhas 10 e 12). A execução do filho simplesmente imprime o seu PID e finaliza. Já na execução do pai, inicialmente,

aguarda-se 3 segundos através do uso da função `sleep()`, para garantir que o filho encerrou sua execução e tornou-se um zumbi (linha 20). A linha 21 compõe uma string usando o comando `ps` do shell do Linux, filtrando apenas as linhas cujo nome do processo em execução (`saida`) apareça. Na linha 22, o comando é executado no sistema, o que faz com que sejam impressos os PIDs e nomes dos processos pai e filho.

No resultado da execução do Código 4, é possível ver que o comando `ps` imprimiu a string `<defunct>` (do inglês: extinto), para indicar que o filho terminou sua execução e se tornou um processo zumbi.

Exercício 11: Responda: o que a system call `wait()` retorna em caso de sucesso? E de falha?

Exercício 12: Responda: é possível usar `wait()` para fazer com que o processo filho espere o pai finalizar? Por quê?

Exercício 13: Escreva um programa que crie dois processos filhos do mesmo pai. O pai deve esperar ambos os filhos finalizarem antes de encerrar.

Na linha 21 do Código 4, a função `sprintf()` armazena o seu argumento (`"ps | grep -i %s"`) dentro da string `cmd`. Inicialmente o comando `ps` é executado e, devido ao *pipe* (`|`), sua saída é repassada como entrada do comando `grep`. O `%s` é substituído pelo valor de `&argv[0][2]`. No caso, `argv` é um vetor de strings, onde a posição 0 contém o nome do programa executável, como passado via terminal (`./saida`). Para o comando `grep`, queremos apenas o nome do executável, isto é, `saida` sem o "`./`". Por isso, passamos uma referência a esta string, porém, a partir de sua posição 2.

4 A família de funções `exec()`

Em diversas situações, ao executar o comando `fork()`, o programador não deseja escrever o código do processo filho, ou então deseja que outro programa já disponível na máquina seja executado como processo filho.

Neste caso, é necessário fazer com que o contexto do processo filho (que é uma cópia do pai), seja substituído por um outro processo, para que este outro processo possa ser executado. Assim, para subsituir a imagem de um processo, usamos alguma função da família de funções `exec`. São elas:

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ... , (char *)0);
int execle(const char *path, const char *arg0, ... , (char *)0, char *const envp[]);
int execlp(const char *file, const char *arg0, ... , (char *)0);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

As funções `exec()` pode ser divididas em dois grupos:

- `execl`: onde o número de argumentos do programa lançado é conhecido.
- `execv`: onde o número de argumentos do programa lançado não é conhecido.

Basicamente, o que diferencia um grupo de outro é a quantidade de parâmetros utilizados na invocação de uma função. Como ponto em comum, todas elas fazem com que um processo entre em execução.

Para o grupo de funções `execl`, o primeiro parâmetro é o caminho (ou nome) do arquivo executável, o segundo parâmetro é o nome do programa e os demais parâmetros são os argumentos a serem passados para o programa. O último parâmetro deve ser sempre nulo (`NULL`).

Para o grupo de funções `execv`, o primeiro parâmetro é o caminho (ou nome) do arquivo executável, e o segundo parâmetro é um vetor de strings que contém os argumentos a serem passados para o programa, onde a última string é sempre nula.

Adicionalmente, as funções que possuem um `p` em seu nome (`execlp`, `execvp`) replicam a ação do terminal ao procurar por um programa executável. Dessa maneira, uma vez que o programa executável seja encontrado nos diretórios especificados pela variável de ambiente `PATH`, seu nome pode ser usado na invocação, sem a necessidade de se passar o caminho completo. Já as funções que possuem um `e` em seu nome (`execle`, `execve`) permitem que o programador defina o ambiente de execução para o processo. Neste caso, `*envp[]` deve ser um vetor de strings, onde a última string é sempre nula.

A função `fexecve()` é idêntica à `execve()`, com a exceção de que seu primeiro parâmetro, ao invés do caminho do arquivo executável, é o descritor do arquivo.

Como exemplo de uso da função `execl`, considere o Código 5. Neste código, o contexto do processo filho é substituído pelo programa `ls`, que lista os arquivos e pastas do diretório atual do teminal do Linux. Em sua linha 12, a função `execl` recebe três parâmetros: o caminho deste programa no sistema de arquivos (`"/bin/ls"`), o nome do programa (`"ls"`) e, como nenhum parâmetro foi repassado a comando `ls`, o último parâmetro é `NULL`. Caso o programador queira passar algum parâmetro para o comando `ls`, basta inserir outras strings com os valores correspondentes. Por exemplo, para executar o comando `"ls -l"`, a função seria chamada como: `execl("/bin/ls", "ls", "-l", NULL);`. Novamente, note que o último parâmetro é sempre `NULL`.

Código 5: `exemplo5.c`

```
1  #include <unistd.h>
2  #include <sys/wait.h>
3
4  int main()
5  {
6      pid_t child_pid;
7
8      child_pid = fork();
9
10     if(child_pid == 0)
11     {
12         execl("/bin/ls", "ls", NULL);
13     }
14     else
15     {
16         wait(NULL);
17     }
18
19     return 0;
20 }
```

Observe, a seguir, o resultado da execução do Código 5, porém, com a passagem do parâmetro `"-l"`.

```
aluno@lab:~$ gcc -o saida exemplo5.c
aluno@lab:~$ ./saida
total 84
```

```
drwxr-xr-x 2 aluno aluno  4096 ago 15 11:44 Desktop
drwxr-xr-x 2 aluno aluno  4096 ago 15 11:44 Documents
drwxr-xr-x 4 aluno aluno  4096 ago 18 17:54 Downloads
-rw-rw-r-- 1 aluno aluno   161 ago 19 21:20 exemplo1.c
-rw-rw-r-- 1 aluno aluno   688 ago 19 21:20 exemplo2.c
-rw-rw-r-- 1 aluno aluno   287 ago 19 21:23 exemplo3.c
-rw-rw-r-- 1 aluno aluno   433 ago 19 21:24 exemplo4.c
-rw-rw-r-- 1 aluno aluno   248 ago 20 16:09 exemplo5.c
drwxr-xr-x 2 aluno aluno  4096 ago 15 11:44 Music
drwxr-xr-x 2 aluno aluno  4096 ago 15 11:44 Pictures
drwxr-xr-x 2 aluno aluno  4096 ago 15 11:44 Public
-rwxrwxr-x 1 aluno aluno 16200 ago 20 21:08 saida
drwxr-xr-x 2 aluno aluno  4096 ago 15 11:44 Templates
drwxr-xr-x 2 aluno aluno  4096 ago 15 11:44 Videos
aluno@lab:~$
```

De modo análogo ao comando `execl()`, caso o comando `execlp()` fosse usado, não seria necessário especificar o caminho completo do executável como primeiro parâmetro. Assim, a instrução ficaria: `execlp("ls", "ls", NULL);`.

Como exemplo do grupo de funções `execv`, considere o Código 6, que faz uso da função `execvp()`.

Código 6: `exemplo6.c`

```
1  #include <unistd.h>
2  #include <sys/wait.h>
3
4  int main()
5  {
6      pid_t child_pid;
7      char *cmd[] = {"ls", "-l", NULL};
8
9      child_pid = fork();
10
11     if(child_pid == 0)
12     {
13         execvp("ls", cmd);
14     }
15     else
16     {
17         wait(NULL);
18     }
19
20     return 0;
21 }
```

Como pode ser observado no Código 6, em sua linha 13, a função `execvp()` recebe apenas dois parâmetros: o nome do programa executável (`ls`) e um vetor de strings (linha 7: `cmd`). No caso das funções do grupo `execl`, os argumentos eram repassados como uma lista de parâmetros para a função. No caso do grupo `execv`, estes mesmos argumentos são inseridos dentro do vetor de strings. A última string desse vetor deve ser, obrigatoriamente, nula.

Exercício 14: Quando a família de funções `exec()` retorna um valor inteiro?

Exercício 15: O comando passado para a função `exec()` sempre executa. A afirmação é verdadeira ou falsa? Por quê?

Exercício 16: No Código 4, você usou a função `system()` para fazer com que outro processo fosse executado. Nos Códigos 5 e 6, você usou a função `exec()`. Explique como essas duas funções se diferenciam.

Documentação

A documentação de todas as funções e bibliotecas vistas neste guia pode ser encontrada em:
`https://pubs.opengroup.org/onlinepubs/9699919799/`

(No frame direto, procure pelo link *Alphabetic Index*. Use este índice para buscar pelo termo que desejar.)