Atividade Prática 5

Paula Raissa Silva

1 Gestão de Processos

Um processo é uma instância em execução de um programa. Os processos são criados através de chamadas de sistema. As mais populares são o *fork()* e *exec()*.

Os principais assuntos abordados nessa lista de exercícios serão:

- Novas bibliotecas: $< unistd.h >$, $< sys/wait.h >$, $< errno.h >$
- Tipo de dados para processos pid_t
- *fork()*
- *exec()*, *execlp()*
- *wait()*, *waitpid()*

2 Criação de Processos

No sistema UNIX, há apenas uma chamada de sistema para criar um novo processo: *fork*. Essa chamada cria um clone exato do processo que a chamou. Após o *fork*, os dois processos, pai e filho, têm a mesma imagem de memória, as mesmas variáveis de ambiente, e os mesmos ficheiros abertos. Normalmente o processo filho executa o *exec* ou uma chamada de sistema similar para mudar sua imagem de memória e executar um novo programa.

Tanto no sistema UNIX quanto no sistema Windowns, após um processo ser criado, o pai e o filho têm seus próprios espaços de endereços de memória

2.1 exec

```
#include <unistd.h>

int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int  execle( const char *path, const char *arg , ...,
char* const envp[]);

int execv( const char *path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
```

As primitivas *exec*() constituem na verdade uma família de funções (execl, execlp, execle, execv, execvp) que permitem o lançamento da execução de um programa externo ao processo. Não existe a criação efetiva de um novo processo, mas simplesmente uma substituição do programa de execução.

Existem seis primitivas na família, as quais podem ser divididas em dois grupos: os *execl*(), para o qual o número de argumentos do programa lançado é conhecido; e os *execv*(), para o qual esse número é desconhecido. Em outras palavras, estes grupos de primitivas se diferenciam pelo

número de parâmetros passados. O parâmetro inicial destas funções é o caminho do arquivo a ser executado.

Os parâmetros *chararg* para as 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. Esses parâmetros descrevem uma lista de um ou mais apontadores para strings não-nulas que representam a lista de argumentos para o programa executado.

As funções execv e execvp fornecem um vetor de ponteiros para strings não-nulas que representam a lista de argumentos para o programa executado. Para ambos os casos, assume-se, por convenção, que o primeiro argumento vai apontar para o ficheiro associado ao nome do programa a ser executado. A lista de argumento deve ser terminada pelo apontador $NULL$.

A função execle também especifica o ambiente do processo executado após o ponteiro NULL da lista de parâmetros ou o ponteiro para o vetor argv com um parâmetro adicional. Este parâmetro adicional é um vetor de ponteiros para strings não-nulas que deve também ser finalizado por um ponteiro NULL. As outras funções consideram o ambiente para o novo processo como sendo igual ao do processo atualmente em execução.

Valor de retorno: Se alguma das funções retorna, um erro terá ocorrido. O valor de retorno é -1 neste caso, e a variel global errno será setada para indicar o erro.

Na chamada de uma função *exec*(), existe um recobrimento do segmento de instruções do processo que chama a função. Desta forma, não existe retorno de um *exec*() cuja execução seja correta (o endereço de retorno desaparece). Em outras palavras, o processo que chama a função *exec*() morre logo após executar.

2.2 fork

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

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). Por outro lado, a cópia do segmento de dados do filho do sistema pode diferir do segmento do pai em alguns atributos específicos (como por exemplo, o pid, o tempo de execução, etc.). 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 apontadores para os ficheiros associados são divididos (se o filho movimenta o apontador dentro de um ficheiro, a próxima manipulação do pai será feita a partir desta nova posição do apontador). Esta noção é muito importante para a implementação dos *pipes* entre processos.

Valor de retorno: 0 para o processo filho, e o identificador do processo filho para o processo pai; -1 em caso de erro (o sistema suporta a criação de um número limitado de processos).

Quando fork é chamado, o espaço de endereço virtual (assim como os descritores de arquivo) é copiado para o novo processo. Isso significa que, para todos os efeitos, os dois processos são idênticos. Agora, para garantir que os dois processos permaneçam independentes, toda a memória física é somente leitura. Quando uma tentativa de gravação é feita, uma exceção do processador é gerada. O kernel então pagina em uma nova página e copia os dados da página original. Em seguida, ele executa novamente o processo e permite a gravação. Para que isso funcione, o processador deve ter uma Unidade de Gerenciamento de Memória que mapeie a memória física para o espaço virtual. Isso significa que o Linux não modificado não pode ser executado em unidades de microcontroladores.

Uma lógica parecida também é utilizada no GitHub. É possível fazer um *fork()* de um repositório. Nesse caso o comando *fork()* permite que faça uma cópia de repositório que estás a gerenciar. Esse repositório criado através de um *fork* permite que faças modificações no projeto, sem afetar o repositório original. Você pode buscar atualizações ou enviar alterações para o repositório original usando *pull requests*.

2.3 wait

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

pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options)
int *status
```

Valor de retorno: identificador do processo morto ou -1 em caso de erro.

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 (caso de um processo zumbi), a função retorna imediatamente. Após a criação dos filhos, o processo pai ficará bloqueado na espera de que estes morram.

A função waitpid suspende a execução do processo até que o filho especificado pelo argumento pid tenha morrido. Se ele já estiver morto no momento da chamada, o comportamento é idêntico ao descrito anteriormente.

O valor do argumento *pid* pode ser:

- $<$ -1 : significa que o pai espera a morte de qualquer filho cujo o ID do grupo é igual so valor de pid;
- 0 : significa que o pai espera a morte de qualquer processo filho cujo ID do grupo é igual ao do processo chamado;
- $>$ 0 : significa que o pai espera a morte de um processo filho com um valor de ID exatamente igual a pid.

Se status é não nulo (NULL), wait e waitpid armazena a informação relativa a razão da morte do processo filho, sendo apontada pelo ponteiro status. Este valor pode ser avaliado com diversas macros que são listadas com o comando shell man 2 wait.

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.

Um processo pode se terminar quando seu pai não está a sua espera. Neste caso, o processo filho vai se tornar um processo denominado zumbi (zombie). Ele é neste caso identificado pelo nome $< defunct >$ ou $< zombie >$ ao lado do nome do processo. Seus segmentos de intruções e dados do usuário e do sistema são automaticamente suprimidos com sua morte, mas ele vai continuar ocupando a tabela de processo do kernel. Quando seu fim é esperado, ele simplesmente desaparace ao fim de sua execução.

3 Compartilhamento de Memória

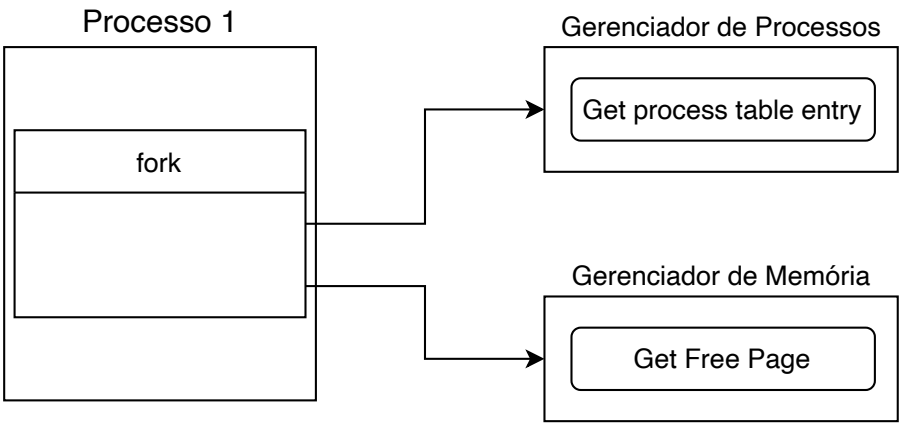

Figura 1: Diagrama de controle de fluxo para a chamada de sistema fork

As partes estáticas, como o segmento de código e a memória estática podem ser compartilhadas entre os dois processos e não requerem uma cópia da memória. Muitas vezes, uma técnica chamada *copy-on-write* é usada ao criar o heap e a pilha do filho. O kernel marca as regiões na memória como somente leitura e somente quando um dos processos acessa a memória com uma solicitação de gravação, a página específica é copiada. Finalmente, todos os descritores de ficheiros abertos e descritores de soquete são copiados para que o filho possa continuar qualquer operação, incluindo I/O do mesmo estado que o pai.

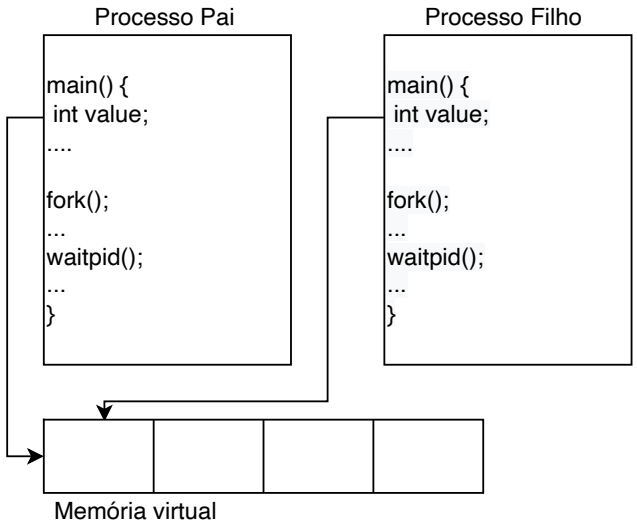

Figura 2: Partilha de memória entre processo pai e filho

A *copy-on-write* encontra seu uso principal no compartilhamento da memória virtual dos processos do sistema operativo, quando invocamos a chamada de sistema *fork*. Normalmente, o processo não modifica nenhuma memória e executa imediatamente um novo processo, substituindo totalmente o espaço de endereço. Ele pode ser implementado de maneira eficiente usando a tabela de páginas ao marcar determinadas páginas de memória como somente leitura e mantendo uma contagem do número de referências à página. Quando os dados são gravados nessas páginas, o kernel intercepta a tentativa de gravação e aloca uma nova página física, inicializada

com os dados de cópia em gravação, embora a alocação possa ser ignorada se houver apenas uma referência. O kernel então atualiza a tabela de páginas com a nova página (gravável), decrementa o número de referências e executa a gravação. A nova alocação garante que uma mudança na memória de um processo não seja visível em outro. Quando a memória é alocada, todas as páginas retornadas são marcadas como *copy-on-write*. Dessa forma, a memória física não é alocada para o processo até que os dados sejam gravados, permitindo que os processos reservem mais memória virtual do que a memória física e usem a memória de forma esparsa, correndo o risco de ficar sem espaço de endereço virtual.

Quando executamos a questão 3, temos o seguinte resultado:

```
CHILD: value = 1, addr = 0x7ff7b3b9e158
PARENT: value = 0, addr = 0x7ff7b3b9e158
```

Nesse caso temos o processo filho e o processo pai a alterar uma variável *value*. O processo filho e o pai partilham o mesmo endereço virtual de memória, portanto esse é o endereço *addr* que vemos no retorno. No momento em que a variável *value* vai ser alterada pelo processso filho, é acionada a técnica chamada *copy-on-write*, o kernel marca essa região da memória virtual como somente leitura e faz uma cópia da página da memória. As alterações feitas na variável *value* pelo processo filho não são visíveis pelo processo pai, porque todas essas operações são feitas na cópia da página da memória e o valor final fica armazenado nesse espaço dedicado à paginação de memória. Quando o processo pai executa, o valor que está no endereço de memória virtual ainda é o valor inicial 0, porque ao acionar a técnica *copy-on-write* a página em que está o processo pai não foi alterada. Portando, o que os processos partilham é um apontador para um endereço de memória virtual (Figura 2). É importante destacar que o endereço físico dos processos não tem como ser acedido pelo usuário/programador, porque é escondido pelo sistema operativo.

Uma lógica parecida também é utilizada no GitHub. É possível fazer um *fork()* de um repositório. Nesse caso o comando *fork()* permite que faça uma cópia de repositório que estás a gerenciar. Esse repositório criado através de um *fork* (que seria como um processo filho) permite que faças modificações no projeto, sem afetar o repositório original. Você pode buscar atualizações ou enviar alterações para o repositório original usando *pull requests*. Nesse caso a atualização do projeto original (semelhante ao processo pai) é feita manualmente e controlada pelo gestor do projeto (que equivale ao gestor de processos do SO).

Se ainda ficaste curioso sobre o assunto, pesquise um pouco mais sobre paginação de memória, memória física e virtual.

4 Resolução dos Exercícios

Ver o ficheio em anexo ficha5.zip. No ficheiro *.zip* estão os exercícios resolvidos e os comentários para algumas soluções.