Sistemas Operacionais
Laboratorio - System Calls (parte 2)
Adaptação do Laboratório 2 - Prof. Eduardo Zambon

1 A API para criação de processos no UNIX

Como vimos aula passada, para se criar um novo processo a partir de outro já em execução, o padrão POSIX define a função `fork()`. Uma implementação típica que utiliza essa função pode ser vista abaixo:

```c
#include <stdio.h>
#include <unistd.h>
// For the syscall functions.
int main() {
   pid_t pid = fork(); // Fork a child process.
   if (pid < 0) { // Error occurred.
      fprintf(stderr, "Fork failed!\n");
      return 1;
   } else if (pid == 0) { // Child process.
      printf("[CHILD]: PID: %d - PPID: %d\n", getpid(), getppid());
   } else { // Parent process.
      printf("[PARENT]: PID: %d - PPID: %d\n", getpid(), getppid());
   }
   return 0;
}
```

No momento de execução do `fork()`, um novo processo filho é criado, e a seguir, tanto o pai quanto o filho seguem a execução *concorrentemente* do mesmo código. No entanto, o valor de retorno do `fork()` muda conforme o processo. No processo filho o retorno é zero. No processo pai o retorno é o PID (*Process ID*) do filho. Compilando e executando esse programa, obtemos o seguinte resultado (PPID – *Parent Process ID*):

```
$ gcc -o fork0 fork0.c
$ ./fork0
[PARENT]: PID: 17607 - PPID: 17593
[CHILD]: PID: 17608 - PPID: 17607
$ ps
  PID TTY          TIME CMD
17593 pts/0    00:00:00 bash
17609 pts/0    00:00:00 ps
```

O processo filho recém criado é um *clone* do seu pai, o que significa que toda a área de memória do filho é copiada do pai, inclusive a área de texto (comandos) e de dados. Qualquer variável declarada no processo pai vai ter o seu valor copiado para o filho no momento da invocação do `fork()`. (Mais informações em: `man 2 fork`.)

2 Término de Processos no Unix

Um processo pode terminar normalmente ou anormalmente nas seguintes condições:

```
Normal:
     - Executa return na função main(), o que é equivalente à chamar exit();
     - Invoca diretamente a função exit() da biblioteca C;
     - Invoca diretamente o serviço do sistema _exit().
Anormal:
     - Invoca o função abort();
     - Recebe sinais de terminação gerados pelo próprio processo, ou
       por outro processo, ou ainda pelo Sistema Operacional.
```

A função `abort()` Destina-se a terminar o processo em condições de erro e pertence à biblioteca padrão do C. Em Unix, a função `abort()` envia ao próprio processo o sinal `SIGABRT`, que tem como consequência terminar o processo. Esta terminação deve tentar fechar todos os arquivos abertos.

A figura a seguir ilustra os diferentes caminhos de término via `exit`. A chamada `exit()` termina o processo, portanto, `exit()` nunca retorna. Ela (i) chama todos os *exit handlers* que foram registrados na função `atexit()`; (ii) memória alocada ao segmento físico de dados é liberada; (iii) todos os arquivos abertos são fechados; (iv) é enviado um sinal para o pai do processo, se este estiver bloqueado esperando o filho, ele é acordado. Se o processo que invocou o `exit()` tiver filhos, esses serão "adotados" pelo processo init (ou systemd). Ao final da chamada `exit()`, o escalonador é invocado.

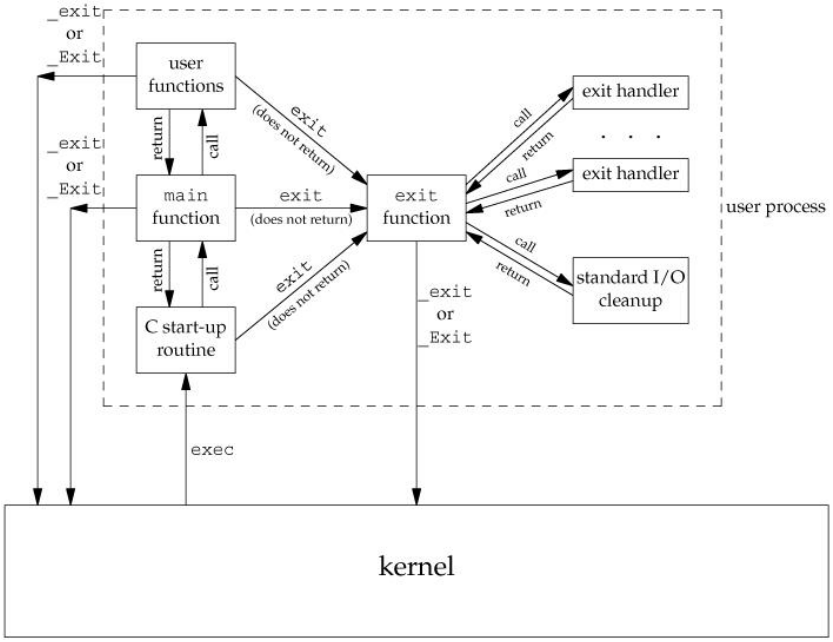

Figura 1: Possibilidades de exit's.

3 Como coordenar a execução de processos relacionados (pais &filhos)?

Diferentes estratégias podem ser aplicadas quando processos relacionados estão trabalhando juntos. Por exemplo: os processos podem executar de forma independente e cada um realiza uma tarefa, ou o pai ficar parado esperando pelos seus filhos e usar os resultados destes. Essa

segunda estratégia é implementada com o uso da função `wait()` no processo pai. Um código de exemplo pode ser visto a seguir.

```c
// FILE: testa_zombie_2.c
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
   int pid ;
   printf("Eu sou o processo pai, PID =  %d, e eu vou criar um filho.\n
      ",getpid()) ;
   pid = fork() ;
   if(pid == -1) /* erro */
  {
      perror("E impossivel criar um filho") ;
      exit(-1) ;
  }
  else if(pid == 0) /* filho */
  {
     printf("Eu sou o filho, PID = %d. Estou vivo mas vou dormir um
        pouco. Use o comando ps -l para conferir o meu estado e o do
        meu pai. Daqui a pouco eu acordo.\n",getpid()) ;
     sleep(60) ;
     printf("Sou eu de novo, o filho. Acordei mas vou terminar agora.
        Use ps -l novamente.\n") ;
     exit(0) ;
  }
  else /* pai */
  {
     printf("Bem, agora eu vou esperar pelo termino da execucao do meu
        filho. Tchau!\n") ;
     wait(NULL) ;  /* pai esperando pelo termino do filho */
  }
}
```

A chamada `wait()` é usada para esperar por mudanças de estado nos filhos do processo chamador (pai) e obter informações sobre aqueles filhos cujos estados tenham sido alterados (ex: morte de um filho). Quando o pai executa o `wait()`, se o filho já teve o seu estado alterado (ex: já morreu) no momento da chamada, ela retorna imediatamente; caso contrário, o processo chamador é bloqueado até que ocorra uma mudança de estado do filho ou então um "signal handler" interrompa a chamada (isso será explicado mais adiante). A figura a seguir ilustra esse comportamento.

3.1 Como fazer "autópsia"em "processos filhos"?

A função `wait()`, como mostrado no exemplo anterior, suspende (bloqueia) a execução do processo pai até que o filho termine. Agora observe no exemplo a seguir que a função `wait()` pode receber como parâmetro o endereço de uma variável:

```c
// FILE: fork1.c
#include <stdio.h>
#include <unistd.h>   // For the syscall functions.
```

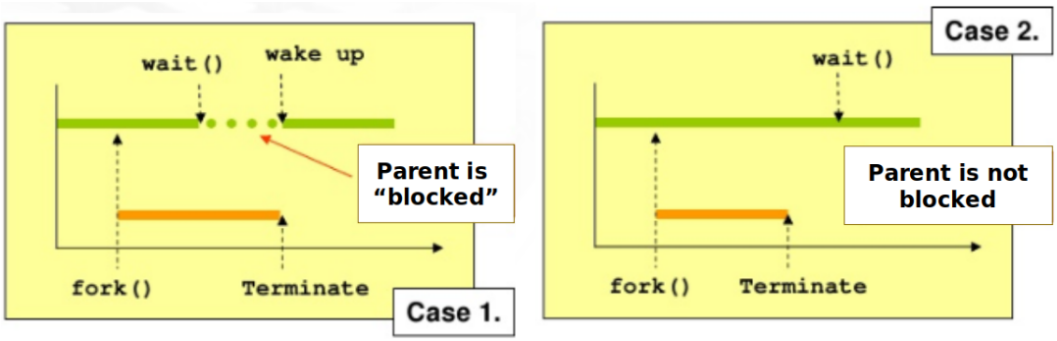

Figura 2: Comportamentos possíveis após um wait.

```c
#include <sys/wait.h> // For wait and related macros.

int main() {
   pid_t pid = fork(); // Fork a child process.
   if (pid < 0) { // Error occurred.
      fprintf(stderr, "Fork failed!\n");
      return 1;
   } else if (pid == 0) { // Child process.
      printf("[CHILD]: I'm finished.\n");
      return 42;
   } else { // Parent process.
      printf("[PARENT]: Waiting on child.\n");
      int wstatus;
      wait(&wstatus);
      if (WIFEXITED(wstatus)) {
         printf("[PARENT]: Child returned with code %d.\n",
                  WEXITSTATUS(wstatus));
      }
   }
   return 0;
}
```

Quando isso acontece, a variável `wstatus` é preenchida com uma série de informações. Essa variável inteira corresponde a uma série de *flags* binárias, portanto para determinar se alguma *flag* foi marcada, é necessário o uso de macros. Por exemplo, a macro `WIFEXITED` retorna verdadeiro se o processo filho terminou normalmente, através de uma chamada para `_exit()` ou retorno da função `main()`. (Mais informações em: `man 2 wait`.) Compilando e executando esse exemplos, temos o seguinte resultado:

```
$ gcc -o fork1 fork1.c
$ ./fork1
[PARENT]: Waiting on child.
[CHILD]: I'm finished.
[PARENT]: Child returned with code 42.
```

O POSIX expecifica seis macros, projetadas para operarem em pares:

```
WIFEXITED(status) - permite determinar se o processo filho terminou normalmente.
                    Se WIFEXITED avalia um valor não zero, o filho terminou
                    normalmente. Neste caso, WEXITSTATUS avalia os 8-bits de menor
```

```
                    ordem retornados pelo filho através de  _exit(), exit() ou
                    return de main.
WEXITSTATUS(status) - retorna o código de saída do processo filho.
--------------------------------------------------------------------------------
WIFSIGNALED(status)- permite determinar se o processo filho terminou devido a um
                    sinal
WTERMSIG(status) - permite obter o número do sinal que provocou a finalização do
                    processo filho
--------------------------------------------------------------------------------
WIFSTOPPED(status) - permite determinar se o processo filho que provocou o retorno
                     se encontra congelado/suspenso (stopped)
WSTOPSIG(status) - permite obter o número do sinal que provocou o congelamento do
                    processo filho

Linux: WIFCONTINUED(status) (Linux 2.6.10)
```

Tarefa

1. Altere o exemplo "fork1.c"de forma que o processo pai imprima uma mensagem caso seu filho tenha morrido devido a um sinal (incluindo o número do sinal "assassino").

   DICA1: o sinal `SIGKILL` (9) é um dos sinais que causa o término do processo; Use o terminal para enviar um SIGKILL para o processo (`kill -SIGKILL PID_DO_PROCESSO`).

3.2 waitpid

Outra opção que o programador tem é usar a função `waitpid(pid_t pid, int *status, int options)`. Esta função 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 com a chamada `wait()`.

```
1 #include <sys/wait.h>
2
3 pid_t wait(int *status);
4 pid_t waitpid(pid_t pid, int *status, int options);
```

Diferenças entre wait() e waitpid():

- wait() bloqueia o processo que o invoca até que um filho qualquer termine (o primeiro filho a terminar desbloqueia o processo pai);
- waitpid() espera um filho específico morrer (a não ser que seja passado o valor -1)
- waitpid() tem uma opção que impede o bloqueio do processo chamador (útil quando se quer apenas obter o código de terminação do(s) filho(s) que já morreram... veremos mais a diante);

Se `wait()` ou `waitpid()` retornam devido ao status de um filho ter sido reportado (por exemplo, o filho morreu), então elas retornam o PID daquele filho. Caso contrário, será retornado `-1`.

Tarefas

2. Altere o exemplo "fork1.c"de forma que o processo pai imprima uma mensagem caso seu filho tenha morrido devido a um sinal (incluindo o número do sinal "assassino"), ou caso seu filho tenha sido suspenso (incluindo o número do sinal recebido pelo filho que tenha causado sua suspensão).

   DICA1: o sinal `SIGKILL` (9) causa o término do processo; o sinal `SIGSTOP` (19) faz com que o processo seja suspenso

   DICA2: A flag `WUNTRACED` especifica que o `waitpid()` também deve reportar mudança de estados dos filhos que foram suspensos (não somente os filhos terminados). Se ela não for usada, o LINUX não consegue fazer com que o pai retorne da chamada `waitpid()` caso um filho tenha sido suspenso.

3. E se o processo pai recebe um sinal enquanto ele está bloqueado em um `wait()` ou `waitpid()`? Implemente um programa C em que o pai cria um filho. O filho, assim que é criado, deve fazer uma chamada `sleep(1000)`. Já o pai, após criar o filho, deve bloquear em uma chamada `int test=wait()`. Se `test==-1`, o pai deve imprimir uma mensagem explicando porque aconteceu aquele erro.

   DICA: Use a variável global `errno` para recuperar códigos de erros retornados por funções C padrão. Além da variável global `errno`o arquivo `errno.h`também define constantes que representam códigos de erro, como:

   ECHILD: não existem filhos para terminar (wait), ou pid não existe (waitpid)

   EINTR: função foi interrompida por um sinal

4 Trocando o código de um processo

Para trocar o código binário do processo filho para algo diferente do código do programa principal (pai), o programador deve usar alguma função da família `exec()`, que carrega o arquivo binário passado como argumento como a imagem do processo filho. Exemplo:

```
1  //FILE: fork2.c
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5  // For the syscall functions.
6  // For wait and related macros.
7  int main() {
8     pid_t pid = fork(); // Fork a child process.
9     if (pid < 0) { // Error occurred.
10       fprintf(stderr, "Fork failed!\n");
11       return 1;
12    } else if (pid == 0) { // Child process.
13       printf("[CHILD]: About to load command.\n");
14       execlp("/usr/bin/ls", "ls", "-la", (char*) NULL);
15       printf("[CHILD]: Great! It worked!\n");
16    } else { // Parent process.
17       printf("[PARENT]: Waiting on child.\n");
18       wait(NULL);
19       printf("[PARENT]: Child finished.\n");
20    }
21    return 0;
```

```
22 }
```

Tarefa

4. Compile e execute o arquivo `fork2.c` e veja o que acontece. Alguma coisa estranha?

   No código anterior, a função `execlp` carrega o programa binário `/usr/bin/ls` e o executa com os argumentos `ls` e `-la`. (Lembre que por convenção do C, o primeiro argumento é sempre o nome do executável.) É importante destacar que, na função `execlp`, a lista de argumentos *deve* ser terminada por um ponteiro `NULL`, e que esse ponteiro deve sofrer *cast* para `char*`. (Mais informações em: `man 3 exec`.) Note também que neste exemplo a função `wait` recebeu um ponteiro nulo, indicando que o processo pai não está interessado no status de retorno do filho .

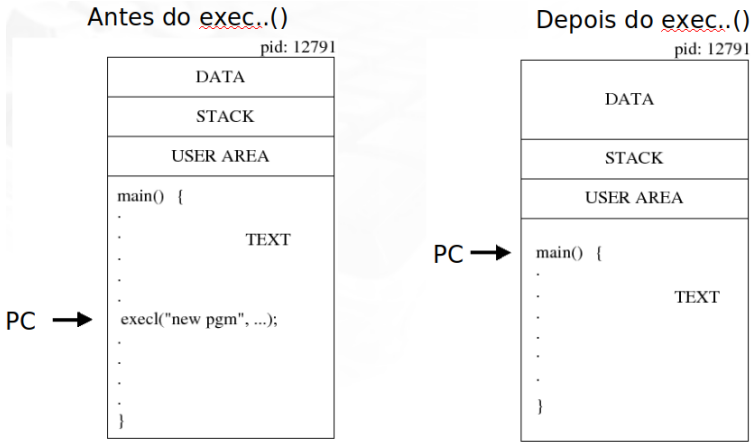

Figura 3: ... Imediatamente após um exec()

Você não sentiu falta do texto `"[CHILD]: Great! It worked!"` na saída? Bom... na verdade seria mal sinal se ele aparecesse. É que, como ilustrado na figura 3, imediatamente após um `exec(...)` a imagem do processo é toda reconstruída (são mantidas apenas algumas informações de controle como o PID e o PPID). E quando a SVC é finalizada, a próxima instrução a ser executada pelo processo será a PRIMEIRA INSTRUÇÃO definida pelo arquivo executável que foi passado como parâmetro na chamada `exec(...)`. Com isso, a linha `printf("[CHILD]: Great! It worked!\n");` que sucede a chamada `exec(...)` SÓ é executada SE a chamada `exec(...)` FALHAR (der erro)! A listagem a seguir mostra os diferentes tipos de erro que podem ocorrer nessa chamada.

```
E2BIG           Lista de argumentos muito longa
EACCES          Acesso negado
EINVAL          Sistema não pode executar o arquivo
ENAMETOOLONG    Nome de arquivo muito longo
ENOENT          Arquivo ou diretório não encontrado
ENOEXEC         Erro no formato de arquivo exec
ENOTDIR         Não é um diretório
```

OBSERVAÇÃO: Antigamente no UNIX, uma chamada de `fork()` era bastante demorada pois exigia a cópia de toda a área de memória do processo pai para o processo filho. Por conta

disso, foi criada uma função `vfork()` que não realiza essa cópia. Após a execução de `vfork()` o filho deve imediatamente chamar a função `exec()`. No entanto, nas implementações atuais do UNIX (Linux, BSD, etc), a função `fork()` é muito mais eficiente (usam a estratégia *Copy-on-Write*), pois evita qualquer cópia desnecessária da memória. Por conta disso, o uso de `vfork()` não é mais recomendado. (Mais informações em: `man 2 vfork`.)

Tarefa

4. Implemente um programa que recebe de 1 a 2 parâmetros: o primeiro parâmetro é o nome de um arquivo executável, e o segundo parâmetro (que é opcional) é um possível parâmetro para esse arquivo executável... Exemplos:

   ```
   $ myProgram ls -l

   //OU
   $ myProgram xcalc
   ```

   Seu programa deve criar um processo filho para executar o comando (e eventualmente seu parâmetro) passado como parâmetro.

   DICA: O UNIX implementa diferentes versões da chamada `exec()`. Veja aqui a descrição desses comandos: `https://linuxhint.com/linux-exec-system-call/`; Você também pode visualizar nos slides do curso: `http://inf.ufes.br/~rgomes/so_fichiers/aula6c.pdf`. No caso desta tarefa, como o comando a ser executado é passado via argv, é mais prático usar a versão `execvp(const char *filename, *cont array[])`