Sistemas Operacionais
Laboratorio 2 - System Calls (parte 2)
Adaptação do Laboratório 2 - Prof. Eduardo Zambon & Profa. Roberta L. Gomes

1 A API para criação de processos no UNIX

Como vimos no último roteiro, 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 (lembre-se que, inclusive, no Linux, processos pai e filho compartilham o seguimento de código, ou *Text*, uma vez que ele é *read-only*). 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 o programa acima, 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
```

Como já vimos, 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 código (*Text*), pilha (*Stack*) e de dados (*Data*). Qualquer variável declarada no processo pai vai ter o seu valor copiado para o filho no momento da invocação do `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 a 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()`[1]; (ii) libera a memória alocada ao segmento de dados; (iii) fecha todos os arquivos abertos; (iv) envia um sinal (`SIGCHLD`) para o pai do processo (se este estiver bloqueado esperando o filho, ele é desbloqueado). 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.

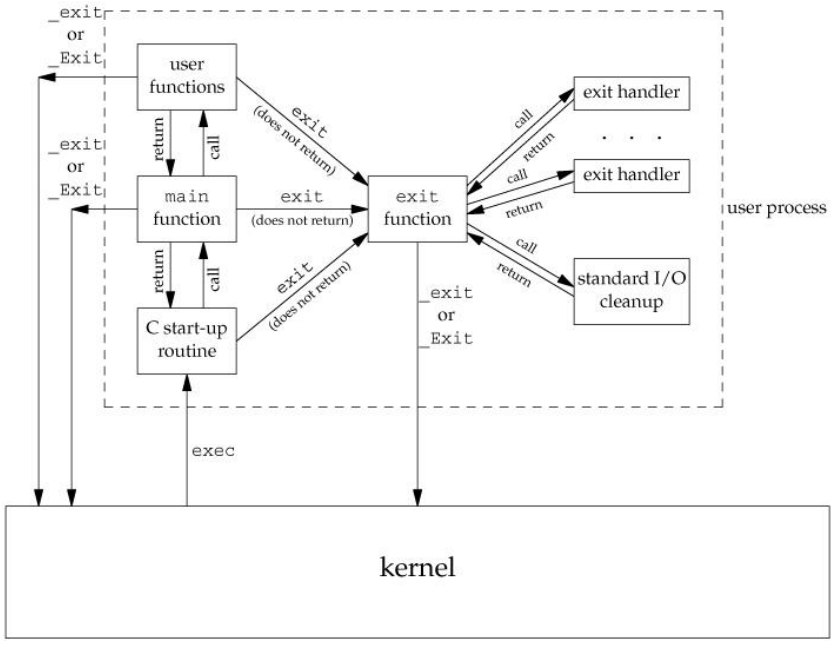

Figura 1: Possibilidades de exit's.

[1] Conforme definido no ANSI C, pode-se registrar até 32 funções que serão automaticamente executadas quando um processo termina. Essas funções são chamadas de *exit handlers* e são registradas por meio da chamada à função *atexit()*.

Tarefas

1. Faça o *download* dos arquivos exemplos para a aula de hoje: lab2.zip
2. Analise o código do arquivo `fork0.c` e em seguida compile e execute o programa. Observe o que aconteceu com valor retornado pelo `getppid()` no processo filho. Execute em seguida o seguinte comando de linha, substituindo `XX` pelo valor printado em `new-PPID`:
   `ps aux | grep XX`
   Você deve ter observado que após o término do processo pai, o processo filho foi "adotado" pelo processo `init` ou pelo processo `systemd`. Mas dependendo da versão do Linux, esse processo não possui o PID = 1. Por exemplo, nas versões mais recentes do Ubuntu, é criado um novo processo `systemd` para cada usuário que logar no sistema (`systemd` por sua vez é filho do `systemd` de PID=1). Com isso, quando um processo de usuário fica "órfão", ele é adotado pelo `systemd` criado para o respectivo usuário. Você pode usar o comando `pstree` para visualizar toda a hierarquia de processos criados no sistema.

init vs systemd
O **init** é um processo (*daemon*) que começa assim que o computador é iniciado e continua funcionando até seu desligamento. Ele é o primeiro processo que o kernel inicia quando um computador é inicializado, tornando-o pai de todos os demais processos em execução, e portanto, lhe é atribuído PID = 1.
O **systemd** é um *daemon* de gerenciamento de sistema, cuja convenção UNIX definiu que seu nome leve "d" no final, significando "daemon": systemd = SystemDaemon. O systemd foi projetado para superar as deficiências do init. Ele próprio é um processo em segundo plano que é projetado para iniciar processos em paralelo, reduzindo assim o tempo de inicialização da máquina e o consumo computacional. Além disso, o systemd conta com muito mais recursos em comparação com init. Com isso, o systemd, inicialmente implementado pelo Fedora, suplantou o init tradicional nas versões mais recentes das distribuições mais usadas de Linux.

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 totalmente independente, ou o pai pode ficar parado (bloqueado) esperando pelos seus filhos e usar os resultados retornados por eles. Essa segunda estratégia é implementada com o uso da função `wait()` no processo pai.

A chamada `wait()` é usada para fazer o processo pai esperar por **mudanças de estado** nos processos filhos. Por meio dessa chamada o processo chamador (pai) pode obter informações sobre aqueles filhos cujos estados tenham sido alterados (ex: término de um filho). A figura 2 ilustra o comportamento da chamada `wait()`. Na figura, a linha verde representa o processo pai e a linha laranja o processo filho. Como ilustrado no *Case 2*, quando o pai executa o `wait()`, se o filho já teve o seu estado alterado (ex: já terminou) no momento da chamada, ele (o pai) retorna imediatamente, não havendo bloqueio deste processo. Caso contrário, como ilustrado no *Case 1*, o processo chamador (o pai) é bloqueado até que ocorra uma mudança de estado do filho (ex: o filho termina)[2].

[2]... ou então até que um "signal handler" interrompa a chamada (isso será explicado mais adiante).

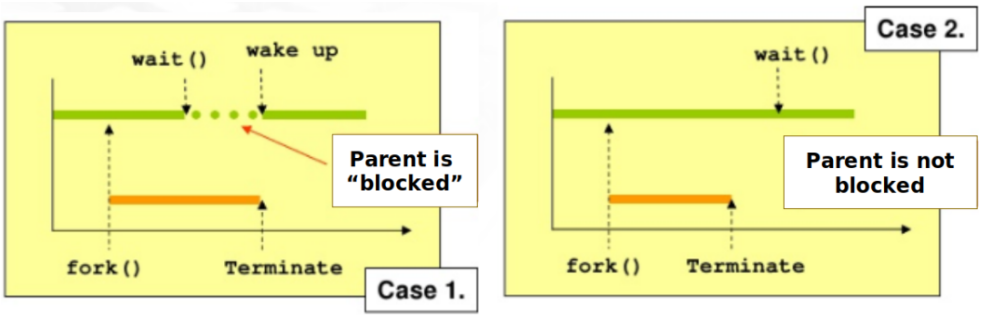

Figura 2: Comportamentos possíveis após um wait.

Um código exemplo ilustrando o uso do `wait()` pode ser visto a seguir:

```
1  // FILE: testa_zombie.c
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <stdlib.h>
5  #include <sys/wait.h>
6
7  int main()
8  {
9     int pid ;
10    printf("Eu sou o processo pai, PID =  %d, e eu vou criar um filho.\n
          ",getpid()) ;
11    pid = fork() ;
12    if(pid == -1) /* erro */
13   {
14       perror("E impossivel criar um filho") ;
15       exit(-1) ;
16   }
17   else if(pid == 0) /* filho */
18   {
19      printf("Eu sou o filho, PID = %d. Estou vivo mas vou dormir um
           pouco. Use o comando ps -lt para conferir o meu estado e o do
           meu pai. Daqui a pouco eu acordo.\n",getpid()) ;
20      sleep(10) ;
21      printf("Sou eu de novo, o filho. Acordei mas vou terminar agora.
           Use ps -lt novamente.\n") ;
22      exit(0) ;
23   }
24   else /* pai */
25   {
26      printf("Bem, agora eu vou bloquear e esperar pelo termino do meu
           filho.\n") ;
27      wait(NULL);  /* pai esperando pelo termino do filho */
28      printf("Pronto... meu filho terminou... agora vou terminar tamb m
           ! Tchau!\n") ;
29      //for(;;) ; /* pai bloqueado em loop infinito */
30
31   }
32 }
```

Tarefas

3. Compile e execute o arquivo `testa_zombie.c` em **background**. Siga as instruções: enquanto o processo filho estiver bloqueado no `sleep()`, rode no shell o comando `ps -lt`; após o filho terminar, rode novamente `ps -lt`. O que você observou quando você executou o `ps -lt` enquanto o filho estava bloqueado?
   Preste atenção na coluna `STAT`. Ali você consegue visualizar o estado `S` (que no Linux indica que o processo está bloqueado) tanto para o pai quanto para o filho. Já na coluna `WCHAN` você consegue ver o nome da função de kernel na qual o processo encontra-se bloqueado. No Ubuntu, no caso do pai temos `do_wai` (função de kernel utilizada para bloquear o processo na chamada de sistema `wait()`) e no caso do filho temos `hrtime` (função de kernel utilizada para bloquear o processo na chamada de sistema `sleep()`).

4. Agora altere o código de `testa_zombie.c` da seguinte forma: (i) comente a linha 27 (inclua `//` no começo da linha); (ii) descomente a linha 29 (exclua o `//` no começo da linha). Se você quiser, você também pode comentar a linha 20 (apenas para permitir que o filho termine mais rápido). Rode novamente o programa em **background** e execute no shell `ps -lt` em seguida.
   Como resultado, você observou que como o processo pai não executa mais o `wait()`, apesar do filho já ter terminado ele ainda é listado pelo comando `ps`. Mas você consegue notar que o estado desse processo filho é *Zumbi* (letra `Z` na coluna `STAT`). Mas esse processo filho vai continuar no estado *Zumbi* "para sempre"? [Responda no formulário online] *Dica: o quadro abaixo pode ajudar...*
   IMPORTANTE: Ao final, não esqueça de terminar o processo pai, caso contrário você terá um processo em loop infinito consumindo 100% de uma de suas CPUs. Para isso você pode usar o comando `kill -9 PID_DO_PAI;`

Processos Zumbis
Um processo zumbi (ou *zombie*, chamado também de *defunct*) é um processo que finalizou a execução mas ainda possui uma entrada na tabela de processos, pois seu processo pai ainda não "tomou conhecimento" que ele terminou.
Os processos zumbis são assim chamados porque eles já "morreram" (finalizaram a execução), tiveram seus recursos desalocados (memória, descritores de arquivo, etc), mas ainda não foram "expurgados" do sistema (permanece sua entrada na tabela de processos do sistema, e ele ainda ocupa um bloco de controle de processo - BCP). Estão "mortos", mas ainda existem de alguma forma no sistema.
Se processos zumbis ocupam PIDs, por que eles existem? A razão é que, muitas vezes, o processo pai precisa tomar conhecimento que o filho morreu, saber qual código este retornou, e executar alguma ação em função destas informações. E para poder "tomar conhecimento", o processo pai precisa executar a chamada **wait** (ou uma de suas variantes). Então sempre que um processo termina ele permanece no estado Zumbi até que seu pai faça uma chamada **wait**.
Mas e se o pai do Zumbi também já tiver terminado? Isso não é um problema! Você se lembra que quando um processo fica "órfão", ele é "adotado" pelo processo systemd ou init? Então, isso também ocorre mesmo que o processo órfão esteja no estado Zumbi. Com isso, uma das tarefas do systemd (ou do init) é ficar de forma periódica executando a chamada **wait** de forma a "expurgar" todos os processos Zumbis que eventualmente ele tenha "adotado", liberando assim entradas na tabela de processos e blocos de controle.

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 (linha 17):

```
1  // FILE: fork1.c
2  #include <stdio.h>
3  #include <unistd.h>   // For the syscall functions.
4  #include <sys/wait.h> // For wait and related macros.
5
6  int main() {
7     pid_t pid = fork(); // Fork a child process.
8     if (pid < 0) { // Error occurred.
9        fprintf(stderr, "Fork failed!\n");
10       return 1;
11    } else if (pid == 0) { // Child process.
12       printf("[CHILD]: I'm finished.\n");
13       return 42;
14    } else { // Parent process.
15       printf("[PARENT]: Waiting on child.\n");
16       int wstatus;
17       wait(&wstatus);
18       if (WIFEXITED(wstatus)) {
19          printf("[PARENT]: Child returned with code %d.\n",
20                   WEXITSTATUS(wstatus));
21       }
22    }
23    return 0;
24 }
```

Quando isso acontece, a variável inteira cujo endereço foi passado como parâmetro na chamada `wait` (no exemplo acima, `wstatus`) é preenchida com uma série de informações. Essa variável inteira passa a corresponder a uma série de *flags* binárias. A forma mais prática de se determinar se alguma *flag* foi marcada, é utilizando macros. Por exemplo, a macro `WIFEXITED` retorna verdadeiro se o processo filho terminou normalmente (o filho executou `_exit()` ou retornou da função `main()`)[3]. Quando a macro `WIFEXITED` retornar TRUE (ou seja, o processo filho retornou normalmente), o processo pai pode utilizar a macro `WEXITSTATUS` para descobrir qual foi o valor (inteiro) retornado pelo processo filho quando este último terminou.

Compilando e executando o código acima 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 (isto é, o processo deve primeiramente verificar o tipo de término que o filho teve e, em seguida, utilizar uma segunda macro para obter mais informações):

[3]Mais informações em: **man 2 wait**

```
WIFEXITED(int 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(int status) - retorna o código de saída do processo filho.
--------------------------------------------------------------------------------
WIFSIGNALED(int status)- permite determinar se o processo filho terminou devido
                  a um sinal.
WTERMSIG(int status) - permite obter o número do sinal que provocou a finalização
                  do processo filho.
--------------------------------------------------------------------------------
WIFSTOPPED(int status) - permite determinar se o processo filho que provocou o
                  retorno se encontra congelado/suspenso (stopped).
WSTOPSIG(int status) - permite obter o número do sinal que provocou o congelamento
                  do processo filho.

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

É importante ressaltar que a chamada `wait` (bloqueante ou não) retorna não somente quando o filho morre... mas sempre que o filho **mudar de estado**. Mas o que significa "mudar de estado" neste contexto? Neste caso, temos 3 possibilidades: o filho terminou (essa você já sabia!), o filho foi suspenso, ou o filho foi *des-suspenso*. Percebam que o terceiro par de macros mostrado acima (`WIFSTOPPED` e `WSTOPSIG`) faz todo sentido agora! Além disso, para versões mais atuais do Linux, também temos a macro `WIFCONTINUED`. Mas para que essas macros funcionem corretamente, teremos que utilizar uma variante da chamada `wait`, como veremos na subseção 3.2.

Tarefa

5. Altere o exemplo `fork1.c` de forma que o processo pai imprima uma mensagem caso seu filho tenha terminado devido a um sinal (incluindo o número do sinal). Em um segundo terminal envie um sinal SIGUSR1[4] para o processo filho (`kill -SIGUSR1 PID_DO_PROCESSO_FILHO`). Qual foi o valor retornado pela macro `WTERMSIG`? [Responda no formulário online]

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 terminado (mais especificamente, mudado de estado... mas daqui para frente, **por simplicidade**, vamos considerar apenas o término do filho). Se ele já tiver terminado 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);
```

[4]SIGUSR1 é um dos sinais que causa o término do processo.

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 terminar (a não ser que seja passado o valor -1 no seu primeiro parâmetro, neste caso ele tem o mesmo comportamento do `wait()`, isto é, espera **qualquer** filho);
- `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 terminaram... veremos mais a diante);

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

Tarefas

6. Continuando o que você fez na [Tarefa 5], agora, além de imprimir uma mensagem caso seu filho tenha terminado devido a um sinal (incluindo o número do sinal), o processo pai também deve imprimir uma mensagem caso o seu filho tenha sido suspenso (incluindo o número do sinal que tenha causado a suspensão do filho). Durante a execução do seu código, abra um outro terminal e envie um sinal `SIGSTOP` para o processo filho, fazendo com que o mesmo seja suspenso. Qual foi o valor retornado pela macro `WSTOPSIG`? [Responda no formulário online]

   DICA: A *flag* `WUNTRACED`, passada como o 3o parâmetro na chamada `waitpid()` especifica que o `waitpid()` também deve reportar mudança de estados dos filhos que foram suspensos (não somente os filhos terminados). É fundamental que o processo pai use essa *flag* para que as macros `WIFSTOPPED` e `WSTOPSIG` funcionem corretamente. Se ela não for usada, o Linux não consegue fazer com que o pai retorne da chamada `waitpid()` caso um filho tenha sido suspenso.

7. Crie um programa em que o processo pai crie 3 filhos. Esses filhos devem dormir por 2 segundos cada um e depois terminar. Após criar os filhos, o pai deve ficar em um loop rodando o seguinte algoritmo:

   - dorme 2 segundos;
   - verifica se algum de seus filhos terminou;
   - se um filho terminou ele deve imprimir “Meu filho pid=PID-DO-FILHO terminou”;
   - se nenhum filho terminou, ele deve imprimir “Nenhum filho terminou”;

   ... O pai deve sair do loop apenas quando todos os seus filhos tiverem terminado. Como ficou codificado o loop criado para o processo pai? [Transcreva no formulário online o trecho de código contendo o loop]

   DICA1: A opção WNOHANG como 3o parâmetro na chamada `waitpid` permite que um processo pai verifique se um filho terminou, sem que o pai bloqueie caso o status do filho ainda não tenha sido reportado (ex: o filho não tenha terminado). Neste caso, o `waitpid` retorna 0 (zero). Mas se todos os filhos já tiverem terminado, ela retorna -1.

DICA2 ... SOLUÇÃO ELEGANTE! A solução mais simples para o problema mostrado é ficar no loop e, sempre que um filho morrer, incrementar um contador. Quando esse contador chegar a 3, o pai sai do loop. No entanto, muitas vezes o processo pai NÃO tem essa informação e não sabe quantos filhos ele tem (eita pai promíscuo!!). Neste caso, deve-se usar 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. Assim, quando a chamada `waitpid` retornar `-1` é possível verificar a razão do erro. O valor `ECHILD` indica que não existem filhos para terminar ou pid passado como parâmetro não existe ou existe mas não é filho do processo chamador.

```
if ((waitpid(-1, NULL, WNOHANG)) == -1) && (errno == ECHILD)) break;
```

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, criando **uma nova imagem** para o 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 }
```

No código anterior, a função `execlp` (linha 14), executada pelo processo filho, 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*`[5]. 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.

A figura 3 ilustra o que acontece com um processo quando ele executa um `exec(...)` **de forma bem sucedida.** A imagem do processo é **toda reconstruída** (são mantidas apenas algumas informações de controle como o PID e o PPID). E quando a chamada de sistema

[5] Mais informações em: **man 3 exec**.

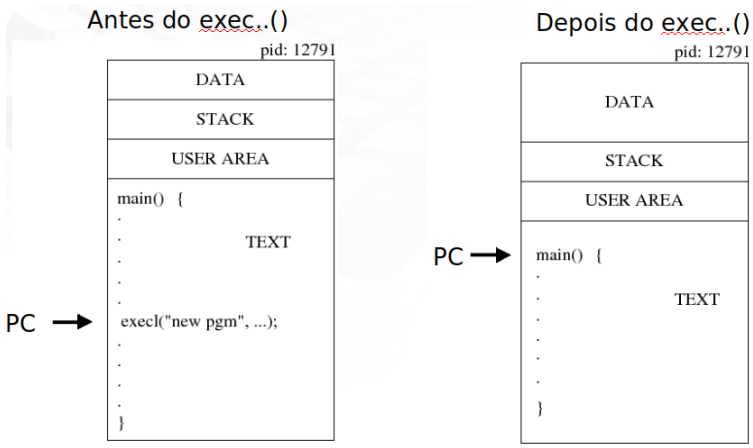

Figura 3: ... À direita, vemos o PC Imediatamente após um exec() BEM SUCEDIDO

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(...)`. Assim, após um `exec(...)`, o código onde esta chamada aparece é **todo substituído** pelo código do arquivo executável passado como parâmetro. Observe na figura que o PC (registrador *Program Counter*) passa a apontar para a primeira instrução do novo código.

A Família de SVC's exec()

```
execl ("/bin/ls", "ls", "-l", NULL)
execlp ("ls", "ls", "-l", NULL)
*env[]="TERM=vt100","PATH=/bin:/usr/bin", NULL;
execle ("/bin/ls", "ls", "-l", NULL, env)
static char *args[] = ( "ls", -l", NULL);
execv ("/bin/ls", args);
execvp (argv[1], argv[1])
execve("/bin/ls", args, env);
```

- Sufixo `l`: a função exec() recebe a lista de argumentos para o executável, um a um, como parâmetros diretos, seguidos do parâmetro NULL.
- Sufixo `v`: os argumentos para o arquivo executável devem ser armazenados em um array de strings (terminado com NULL), e este array é passado como parâmetro na chamada exec()
- Sufixo `e`: o processo, ao executar o exec() terá suas variáveis de ambiente alteradas, sendo essas configuradas num array de strings (terminado com NULL) que deve ser passado como último parâmetro da chamada.
- Sufixo `p`: procura o arquivo executável nos diretórios definidos na variável de ambiente PATH, não sendo necessário passar o caminho absoluto do executável no 1o. parâmetro da chamada).

Eventualmente algum erro pode ocorrer durante a execução de um `exec(...)`. Nesse caso, o código do processo **NÃO** é substituído, e a instrução que segue o `exec(...)` é executada normalmente. A listagem a seguir mostra os diferentes tipos de erro que podem ocorrer:

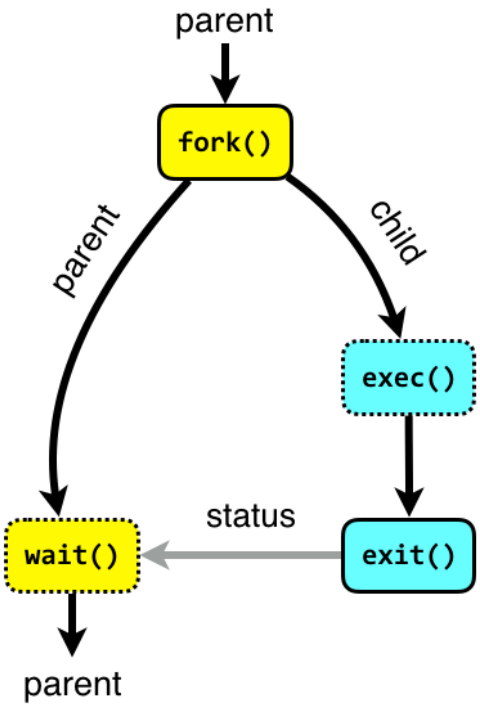

Figura 4: Uso típico das chamadas fork(), exec(), wait()

```
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
```

Com isso, para concluir, a figura 4 ilustra o comportamento típico dos processos ao usarem as chamadas `fork()`, `exec()` e `wait()`. Mas entenda que não há uma obrigatoriedade nisso. Por exemplo, se um processo pai faz o fork(), ele poderia fazer um `exec()` em seguida. Com isso o processo pai teria a sua imagem reconstruída usando o programa passado como parâmetro no `exec()`. Já o processo filho NÃO tendo feito nenhum `exec()` continua executando o código original do pai.

fork() vs vfork()
Antigamente no UNIX, uma chamada **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[a].

[a] Mais informações em: **man 2 vfork**

Tarefa

8. Compile e execute o arquivo `fork2.c` e veja o que acontece. A linha 15 é executada? [Responda no formulário online]

5 Sessões e grupos de processos

No Unix, como vimos no roteiro passado, além de ter um PID, todo processo também pertence a um grupo. Um *process group* é uma coleção de um ou mais processos. Cada grupo pode ter um **processo líder**, que é identificado por ter o seu PID igual ao seu groupID.

É possível ao líder criar novos grupos, criar processos nos grupos e então terminar (o grupo ainda existirá mesmo se o líder terminar; para isso, tem que existir pelo menos um processo no grupo).

Um novo conceito apresenta do aqui é o de **Sessão** (*Session*). Uma sessão é um conjunto de grupos de processos. Grupos ou sessões são também herdadas pelos filhos de um processo. Para mudar de sessão, usa-se a chamada `setsid()`: ala coloca o processo em um novo grupo e sessão (*group ID* e *session ID* novos, iguais ao PID do processo chamador), tornando-o independente do seu terminal de controle.

Terminal de controle?!? A figura 5 vai ajudar a entender. Uma sessão é um conjunto de grupos de processos. Cada sessão pode ter:

- um único terminal de controle;
- no máximo 1 grupo de processos de foreground;
- n grupos de processos de background;

Tarefa

9. Analise o código do arquivo chsession.c e em seguida compile e execute o programa. Observe os valores de PID, GID e SID (*Session ID*).

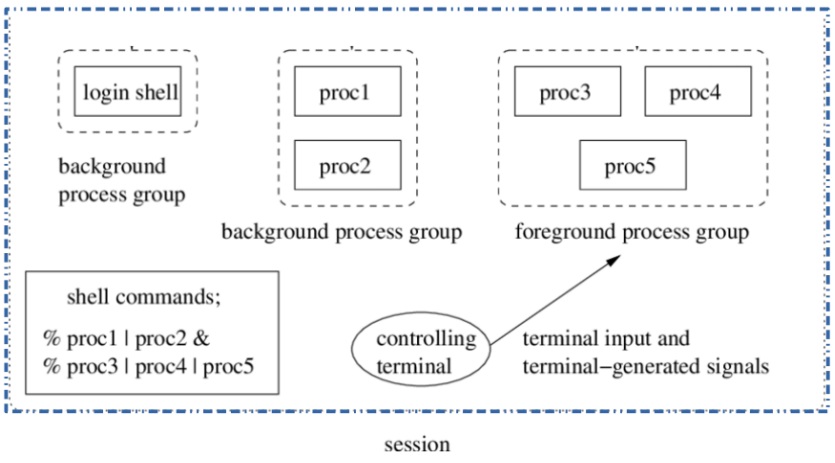

Figura 5: Uma sessão é um conjunto de grupos de processos

========== Exercícios de Consolidação ==========

O objetivo desses exercícios é ajudar no estudo individual dos alunos. Soluções de questões específicas poderão ser discutidas em sala de aula, conforme interesse dos alunos.

1. Descreva o funcionamento da função `fork()`. Após o *fork*, como os processos pai e filho podem se comunicar/sincronizar (considere apenas as chamadas `fork()`, `exec()`, `exit()` e `wait()`)?

2. 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. 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[])`

3. Um grafo de precedência é um grafo direcionado em que a relação (a) → (b) indica que 'a' precede 'b'. Neste exercício, os nós do grafo devem representar as instruções numeradas no código abaixo. Desta forma o grafo de precedência representará a ordem em que as instruções serão executadas. Observe que se o programa não tivesse fork(), o grafo seria linear pois a execução desse programa seria uma sequência de instruções (com desvios ou não). Ex:

   (1) → (2) → (8) → (9) → (8) ... *//números representam as respectivas linhas de instrução*

   No entanto sempre que há um `fork()`, passamos a ter execuções em paralelo. Ex:

   ```
                → (3) → (5)...
   (1) → (2) /
             \
                → (3) → (4)...
   ```

   Desenhe o grafo de precedência referente ao código a seguir:

```c
int f1, f2, f3; /* Identifica processos filho*/
int main(){
    printf("Alo do pai\n");
    f1 = fork;
    if (f1==0)
       execlp("codigo_filho","codigo_filho",NULL);
    printf("Filho 1 criado\n");
    f2 = fork;
```

```
    if (f2==0)
       execlp("codigo_filho","codigo_filho",NULL);
    printf("Filho 2 criado\n");
    waitpid(f1,null,0);
    printf("Filho 1 morreu\n");
    f3 = fork;
    if (f3==0)
      execlp("codigo_filho",    codigo_filho    ,NULL);
    printf("Filho 3 criado\n");
    waitpid(f3,null,0);
    printf("Filho 3 morreu\n");
    waitpid( f2,null,0);
    printf("Filho 2 morreu\n");
    exit();
}
```

4. [EXTRA] Implemente um programa C que possui uma variável do tipo array contendo 10 números desordenados. Esse processo MAIN deve criar um filho. Em seguida o MAIN deve ordenar o array usando "ordenação simples" enquanto o filho deve fazer "quick sort". Ao final da ordenação, cada processo deve exibir o tempo gasto para realizar a mesma. O processo que acabar primeiro deve finalizar (kill()) o seu "parente"e imprimir uma msg avisando sobre o "assassinato"(ex. "Sou o pai, matei meu filho!"). Observem que não deve ser possível que os dois processos mostrem as mensagens de assassinato.

   Dicas:

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

int kill(pid_t pid, int sig);
/*
- If pid is positive, then signal sig is sent to the process with
    the ID specified by pid.
- SIGKILL and SIGINT are examples of signals that can cause the
    process to be terminated
- Return Value: On success (at least one signal was sent), zero is
    returned. On error, -1 is returned, and errno is set
    appropriately.
*/
```

```
#include <time.h>
...
clock_t c1, c2; /* variaveis que contam ciclos de processador */
float tmp;
c1 = clock();
//... codigo a ser executado
c2 = clock();
tmp = (c2-c1)*1000/CLOCKS_PER_SEC; //tempo de execucao em milisec.
```

```
void quickSort(int valor[], int esquerda, int direita)
{
    int i, j, x, y;
    i = esquerda;
    j = direita;
    x = valor[(esquerda + direita) / 2];
    while(i <= j){
```

```
        while(valor[i] < x && i < direita){
            i++;
        }
        while(valor[j] > x && j > esquerda){
            j--;
        }
        if(i <= j){
            y = valor[i];
            valor[i] = valor[j];
            valor[j] = y;
            i++;
            j--;
        }
    }
    if(j > esquerda){
        quickSort(valor, esquerda, j);
    }
    if(i < direita){
        quickSort(valor,  i, direita);
    }
}
```