Criação e terminação de processos

A função fork é invocada 1 vez, mas tem 2 retornos e tem 2 fluxos de execução.
Após a chamada fork() pai e filho executam o mesmo código.

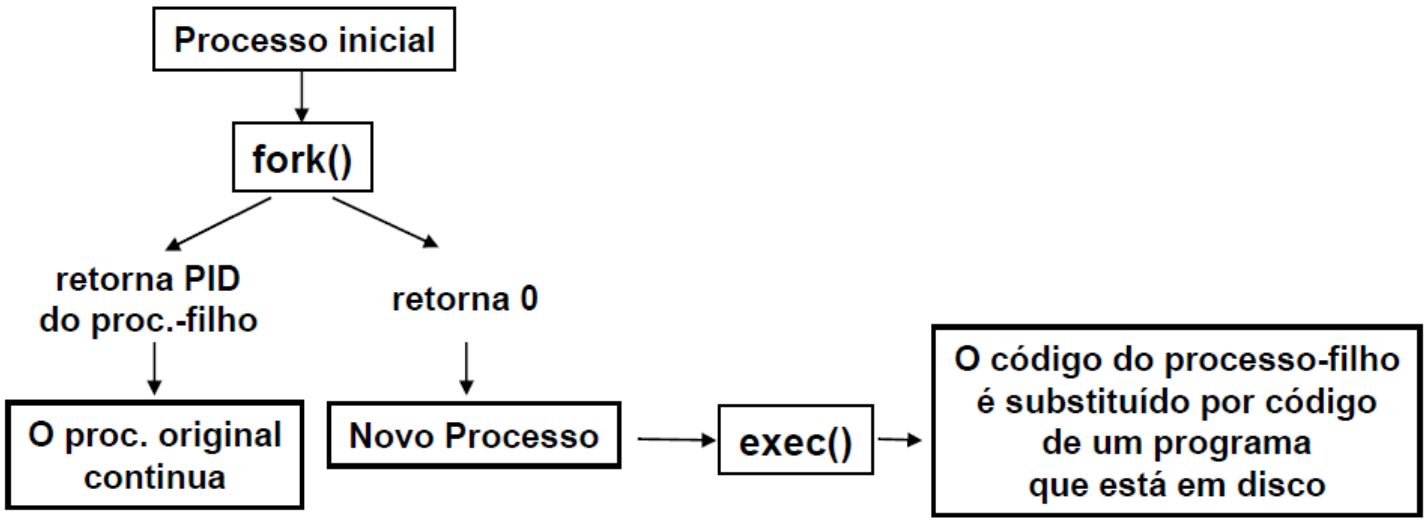
```
int main(){
    pid_t pid = fork();
    printf("Hello");
}
```

Na sequência de um fork, vão resultar 2 processos, que vão ser executados 2 vezes, uma pelo fork e outra pelo que resultou do fork, pelo que neste caso vão ser imprimidas 2 vezes "Hello". Se o fork retornar o pid do processo filho, o processo original continua (processo pai). Se retornar 0, então inicia-se um novo processo (processo filho). Se retornar -1, então ocorreu um erro no processo.

Para que pai e filho executem diferentes segmentos de código (objetivo de utilizar a função fork()), tem de se utilizar as intruções condicionais. Deste modo, o valor de retorno do fork(), já vai ser diferente para pai e filho.

```
int main(){
    pid_t pid = fork();
    printf("Hello");
    if(pid > 0)
        printf("Hello from parent\n");
    else
        printf("Hello from son\n");
}
```

Deste modo, só vai haver uma execução de um processo, caso pid > 0 será o processo pai, case pid = 0 será o processo filho.



```
pid_t pid = fork(); (program counter pai) (retorna 1235)
pid_t pid = fork(); (program counter filho) (retorna 0)
```

Estes 2 processos correm em paralelo, não há garantia que haja um que execute primeiro do que o outro.

Normalmente o que se pretende fazer com o fork é que o processo pai e o processo filho executem diferentes tarefas.
Para que os processos executem diferentes códigos, utiliza-se uma chamada **exec("Prog_novo", ...)** (exs: servidores - o pai espera por um cliente; quando o cliente chega, passa o processo para o filho que vai resolve-lo; o pai volta a ficar a espera de um cliente). Outra hipótese é nos casos em que um processo quer executar um programa diferente (exs: shells - o filho faz exec (do comando) depois de retornar do fork.

funções **getpid** (retorna pid do filho) e **getppid** (retorna pid do pai).

A funçao fork pode falhar quando o nº total de processos no sistema ou do utilizador é muito elevado.

Os processos podem terminar de duas formas:

- **Normal -** O argumento das funções exit (o exit status) indica ao proc. pai como é que o proc. filho terminou (termination status);
- **Anormal -** O termination status do processo é gerado pelo kernel.

O processo-pai pode obter o valor do termination status através das funções wait ou waitpid.

Terminação de um processo

Modos de terminação de um processo:

- **Normal**
  - » Executa ***return*** na função ***main***.
  - » Invoca a função ***exit ( )*** - biblioteca do C
    - Os ***exit handlers***, definidos c/ chamadas ***atexit***, são executados.
    - As ***I/O streams*** standard são fechadas.
  - » Invoca a função ***_exit*** - chamada ao sistema

- **Anormal**
  - » Invoca ***abort***.
  - » Recebe certos sinais gerados por
    - próprio processo
    - outro processo
    - *kernel*

Processos Orfãos

Casos em que o pai termina antes do filho. Neste caso, o filho é automaticamente aotado por init (processo com PID igual a 1).
Deste modo, qualquer processo tem um pai.
Quando o processo pai termina, o kernel percorre todos os processoa ativos a ver se algum deles é filho deste processo que terminou. Nesse caso, o PID desse processo passa a ser 1.

Processos Zombie

Um processo que termina não pode deixar o sistema até que o seu pai aceite o seu código de retorno, através da execução de uma chamada wait / waitpid.
Zombie - Um processo que terminou, mas cujo pai ainda não executou um dos wait's.
A informação sobre o processo filho não pode desaparecer completamente. Assim, o kernel mantém essa informação de modo a que ela esteja disponível quando o pai executar um dos wait's. O resto da memória ocupada pelo processo filho é libertada e o processo é fechado.

Funções wait e waitpid

Quando um processo termina, o kernel notifica o seu pai enviando-lhe um sinal (SIGCHLD). O pai pode:

- Ignorar o sinal - Se o processo indicar que quer ignorar o sinal, os filhos não ficam zombies.
- Dispor de signal handler - O handler poderá executar um dos wait's para obter a PID do filho e o seu termination status.

```
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

O wait espera por que um qualquer filho termine, e depois recolhe o seu sinal.
O waitpid sabe qual é o pid do filho pelo qual tem de esperar.

O argumento statloc:

- $\neq$ NULL - o termination status do processo que terminou é guardado na posição indicada por statloc;
- = NULL - o termination status é ignorado.

O argumento pid de waitpid:

- pid == -1 - espera por um filho qualquer ($\equiv$ wait);
- pid > 0 - espera pelo filho com a PID indicada;
- pid == 0 - espera por um qualquer filho do mesmo process group;
- pid < -1 - espera por um qualquer filho cuja process group ID seja igual a valor_absoluto(pid).

O argumento options de waitpid é útil quando toma o valor WNOHANG que permite que o pai não fique pendurado à espera que o processo filho especificado por pid esteja disponível. Se o pid em waitpid for -1 e options for WNOHANG, isto permite que o processo pai não tenha de esperar por nenhum dos processos filhos caso estes demorem mais.

WNOHANG - indica que waitpid() não bloqueia se o filho especificado por pid não estiver imediatamente disponível (terminado). Neste caso o valor de retorno é igual a 0.
WUNTRACED - indica que, se a implementação suportar job control, o estado de terminação de qualquer filho especificado por pid que tenha terminado e cujo status ainda não tenha sido reportado desde que ele parou, é agora retornado.

waitpid retorna um erro (valor de retorno = -1) se:

- o processo especificado não existir ;
- o processo especificado não for filho do processo que a invocou ;
- o grupo de processos não existir .

Um processo que invoque wait() ou waitpid() pode:

- bloquear - se nenhum dos seus filhos ainda não tiver terminado;
- retornar imediatamente com o código de terminação de um filho - se um filho tiver terminado e estiver à espera de retornar o seu código de terminação (filho zombie).
- retornar imediatamente com um erro - se não tiver filhos.

Diferenças entre wait() e waitpid():

- serviço wait() pode bloquear o processo que o invoca até que um filho qualquer termine;
- serviço waitpid() tem uma opção que impede o bloqueio (útil quando se quer apenas obter o código de terminação do filho);
- waitpid() não espera que o 1º filho termine, tem um argumento para indicar o processo pelo qual se quer esperar.

Quando se usam vários forks consecutivos, convém que de vez enquanto se faça um ciclo com um wait para ver se algum dos processos filho ja terminou. Para isso, à medida que se vai criando filhos, guarda-se os pid's destes num array, por exemplo, e depois no ciclo faz-se por ex. waitpid(array[pid-1]).

Funções exec()

exec - iniciar novos programas (quando um processo invoca exec, o processo é completamente substituído por um novo programa).

```
# include <unistd.h>
int execl (const char *pathname, const char *arg0, … /* (char *)0 */);
int execv (const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, … /* (char *)0, char *const envp[] */);
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, … /* (char *)0 */);
int execvp(const char *filename, char *const argv[]);

Retorno:
    não há - se houve sucesso
    -1 - se houve erro
```

Diferenças entre as 6 funções: estão relacionadas com as letras l, v e p acrescentadas a exec.
Lista de argumentos:

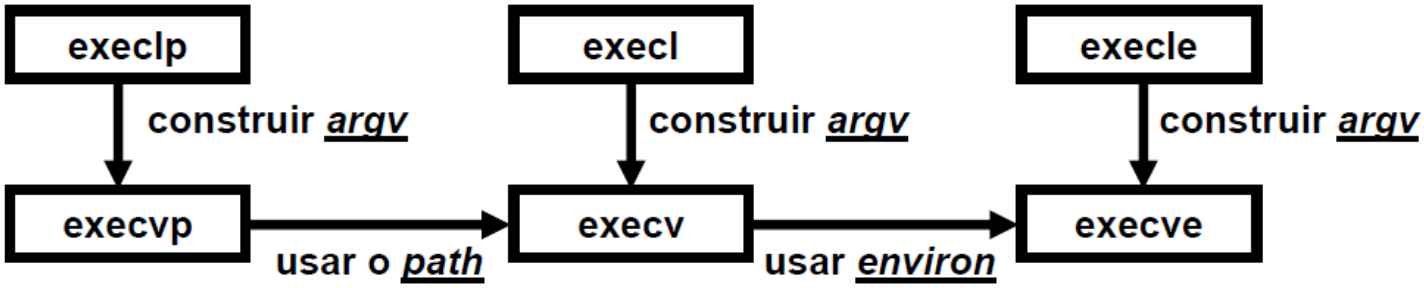
- **l** - Lista, passados um a um separadamente, terminados por um apontador nulo.
- **v** - Vector, passados num vector.

Passagem das strings de ambiente (environment):

- **e** - Passa-se um apontador para um array de apontadores para as strings .
- **sem e** - Usar a variável environ se for necessário aceder às variáveis de ambiente no novo programa.

Path:

- **p** - O argumento é o nome do ficheiro executável.
  - Se o path não for especificado, o ficheiro é procurado nos directórios especificados pela variável de ambiente PATH.
  - Se o ficheiro não for um executável (em código máquina) assume-se que pode ser um shell script e tenta-se invocar /bin/sh com o nome do ficheiro como entrada p/ a shell.
- **sem p** - O nome do ficheiro executável deve incluir o path.



Função system

- Usada para executar um comando do interior de um programa.
  - Ex: system ("date > file")
- Não é uma interface para o sistema operativo mas para uma shell.
- É implementada recorrendo a fork, exec e waitpid.

```
# include <stdlib.h>
int system(const char *cmdstring);
```

Retorno de system:

- Se cmdstring = null pointer
  - retorna valor □ 0 só se houver um processador de comandos disponível (útil p/ saber se esta função é suportada num dado S.O.; em UNIX é sempre suportada)
- Senão retorna
  - -1
    • se fork falhou ou
    • waitpid retornou um erro ≠ EINTR (indica que a chamada de sistema foi interrompida)
  - 127
    • se exec falhou
  - termination status da shell, no formato especificado por waitpid, quando as chamadas fork, exec e waitpid forem bem sucedidas.

Sinais

Um sinal é:

- uma notificação, por software, de um acontecimento
- uma forma, muito limitada, de comunicação entre processos

Um sinal ode ter origem no teclado (certas combinações de teclas), no hardware (referência inválida à memória), na função de sistema kill (permite que um processo envie um sinal a outro processo ou conjunto de processos), no comando kill (permite enviar um sinal a um processo ou conjunto de processosa partir de shell) ou no software (certos acontecimentos gerados por software dão origem a sinais).

Os sinais podem ser gerados sincronamente ou assincronamente.
Todos os sinais têm um nome simbólico que começa por SIG.
Respostas possíveis a um sinal:

- Ignorar - a maior parte pode ser ignorada, exceto SIGKILL e SIGSTOP;
- Tratar (catch) - indicar uma função a executar quando o sinal correr;
- Executar a ação por omissão - todos os sinais têm uma ação por omissão

Chamada signal permite associar uma rotina de tratamento (signal handler) a um determinado sinal. (ex: signal(SIGINT, inthandler) ou signal(SIGINT, SIG_IGN) quando é possível ignorar o sinal)

Signal é uma função que tem como

- **argumentos**
  um inteiro (o número de um sinal)
  um apontador p/ uma função do tipo sigfunc (o novo signal handler)
- **valor de retorno**
  um apontador p/uma função do tipo sigfunc (o signal handler anterior) ou SIG_ERR se aconteceu um erro

Outras constantes declaradas em <signal.h>:

- **SIG_ERR**
  usada para testar se signal retornou erro
  if (signal(SIGUSR1, usrhandler) == SIG_ERR) …;
- **SIG_DFL**
  usada como 2º argumento de signal
  indica que deve ser usado o handler por omissão para o sinal especificado como 1º argumento
- **SIG_IGN**
  usada como 2º argumento de signal
  indica que o sinal especificado como 1º argumento deve ser ignorado

ex: define SIG_IGN (void (*) (int)) 1 - apontador para uma função que tem como parametro um inteiro e que retorna void