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. Se retornar 0, então inicia-se um novo processo (processo 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.

exemplo aula

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 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) e anormal (O termination status do processo é gerado pelo kernel)

Processos Orfãos - pai termina antes do filho
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.

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 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.

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.

O argumento status de waitpid() pode ser NULL ou apontar para um inteiro; no caso de ser ≠ NULL - o código de terminação do processo que terminou é guardado na posição indicada por status; no caso de ser = NULL - o código de terminação é ignorado.

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]).

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