Criação e Terminação de Processos

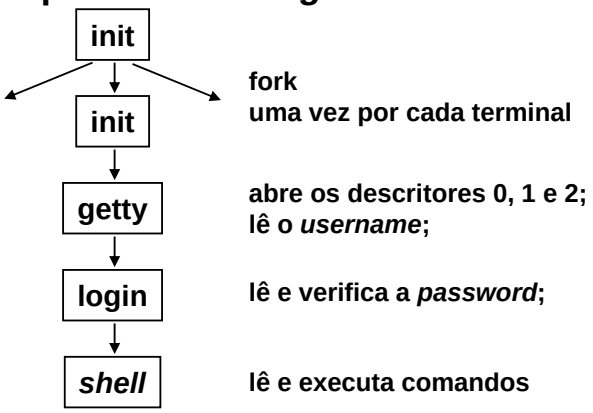
Objectivos

No final desta aula, os estudantes deverão ser capazes de:

- **Explicar a criação de um novo processo em Unix, usando a função *fork()***
- **Usar as funções *fork()* e *exec()* para o lançamento em execução de novos processos/programas**
- **Usar mecanismos de sincronização básicos entre processos**
  - **processos que esperam que outros terminem**
- **Descrever alguns dos estados por que passa um processo durante a sua execução e compreender o significado desses estados**

Criação de novos processos

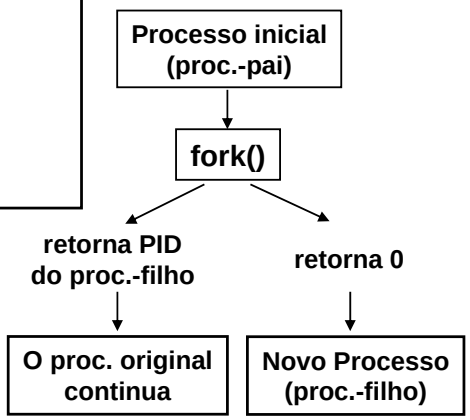
- A forma de um processo existente (processo-pai) criar um novo processo (processo-filho) é invocar a função *fork( )*. A única excepção são processos especiais criados pelo *kernel*.
- O processo *init* (*PID*=1) é um processo especial, criado pelo *kernel*.
- Este processo é responsável por criar outros processos de sistema e por desencadear o processo de *login* dos utilizadores.



A função *fork*

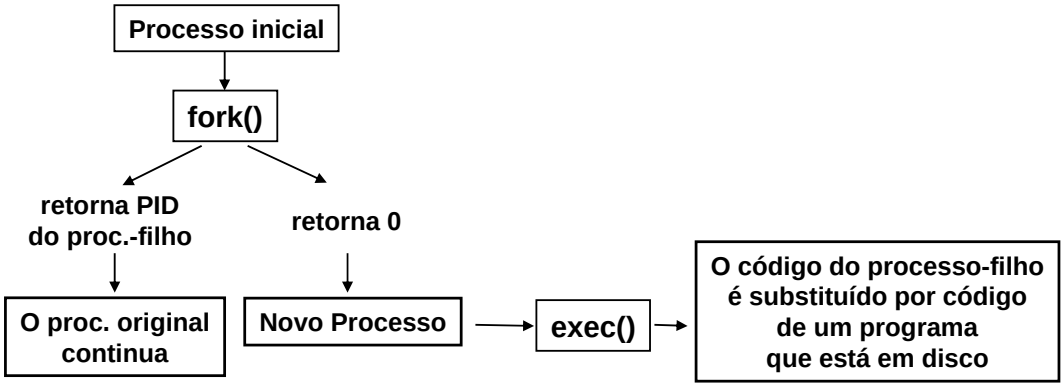
- A função *fork( )* é invocada 1 vez mas retorna 2 vezes !
- Após a chamada *fork( )* pai e filho executam o mesmo (!...) código.

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

pid_t fork(void);

Retorna:
                0 - p/o processo-filho
   PID do filho - p/o processo-pai
               -1 - se houve erro
```



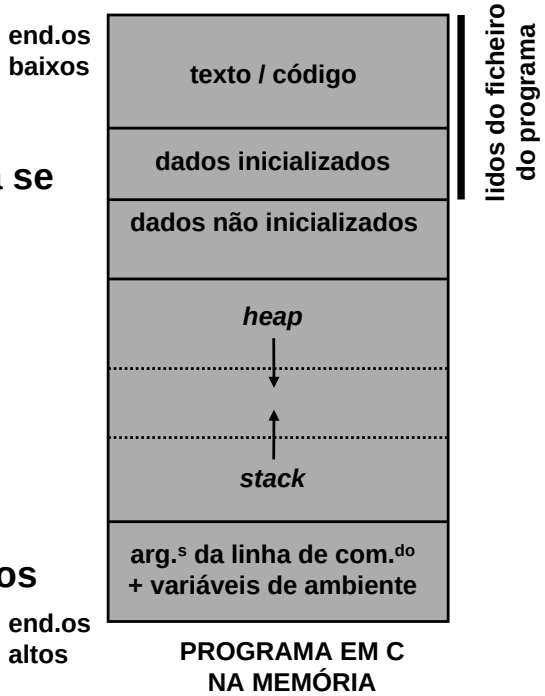
Criação de novos processos

- Em geral, pretender-se-á que pai e filho executem diferentes sequências de instruções.
- Isso consegue-se usando instruções condicionais, uma vez que o valor de retorno de *fork( )* é diferente para pai e filho.
- Frequentemente o processo-filho substitui o seu código por um novo código invocando uma das funções *exec( )* (v. adiante).



A função *fork*

- O filho é uma cópia do pai ficando com uma cópia do segmento de dados, *heap* e *stack*.
- Em alguns sistemas a cópia só é feita se um dos processos tentar modificar um destes segmentos.
- O segmento de texto é muitas vezes partilhado por ambos.
- Após fork(), em geral, não se sabe quem começa a executar primeiro (o pai ou o filho).
- Se for necessária sincronização, é preciso usar mecanismos adequados (v. adiante).



As funções *getpid* e *getppid*

- **Um processo pode obter a sua *PID* e a *PID* do seu pai usando, respectivamente, as funções seguintes:**

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

pid_t getpid(void);    /* Obter a PID do próprio processo */

pid_t getppid(void);   /* Obter a PID do processo-pai */

Notas:
   Estas funções são sempre bem sucedidas.
   A PID do processo-pai do processo 1 é 1.
```


Exemplo

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int     glob = 6;        /* external variable in initialized data */

int main(void)
{
 int    var;             /* automatic variable on the stack */
 pid_t  pid;

 var = 88;

 printf("before fork\n");

 if ( (pid = fork()) < 0) fprintf(stderr,"fork error\n");
 else if (pid == 0) {    /* child */
         glob++;         /* modify variables */
         var++; }
     else
        sleep(2);        /* parent ; try to guarantee that child ends first*/

 printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
 exit(0);
}
```

Resultado possível:

```
before fork
pid = 430, glob = 7, var = 89
pid = 429, glob = 6, var = 88
```

as var.s do filho foram modificadas
as do pai permanecem inalteradas

A função *fork*

Algumas propriedades do pai que são herdadas pelo filho:

- **ficheiros abertos**
- ***ID*'s*(real user, real group, effective user, effective group)***
- ***process group ID***
- **terminal de controlo**
- **directório de trabalho corrente**
- **directório raíz**
- **limites dos recursos**
- **...**

Algumas diferenças entre pai e filho:

- **o valor retornado por *fork ( )***
- **a *ID* do processo**
- **as *ID*'s do processo-pai de cada um deles**
- **os alarmes pendentes são anulados para o filho**
- **o conjunto de sinais pendentes para o filho fica vazio**
- **...**

A função *fork*

***fork( )* pode falhar quando**

- **o nº total de processos no sistema for demasiado elevado (constante MAXPID em sys/param.h);**
- **o nº total de processos do utilizador for demasiado elevado (constante CHILD_MAX em limits.h).**

Utilizações de *fork( )*

- **Quando um processo quer duplicar-se de tal modo que pai e filho executam diferentes secções de código ao mesmo tempo.**
  - » **Ex: servidor de rede**
    - **O pai espera por um pedido de serviço de um cliente.**
    - **Q.do o pedido chega ele faz *fork* e deixa o filho tratar do pedido.**
    - **O pai volta a ficar à espera do próximo pedido.**
- **Quando um processo quer executar um programa diferente**
  - » **Ex: *shells***
    - **O filho faz *exec* (do comando) depois de retornar do *fork*.**

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***

As funções *exit* e *_exit*

```
#include <stdlib.h>

void exit (int status);


#include <unistd.h>

void _exit (int status);
```

As funções *exit* e *_exit*

Independentemente do modo como um processo termina (normal/anormal) será eventualmente executado certo código do *kernel*, código este que

- fecha os descritores abertos pelo processo
- liberta a memória que o processo usava
- ...

Terminação normal

- O argumento das funções *exit* (o *exit status*) indica ao proc.-pai como é que o proc.-filho terminou (*termination status*).

Terminação 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*.

Processos órfãos

- Se o pai terminar antes do filho, o filho é automaticamente adoptado por *init* (proc. c/ *PID*=1)
- Fica assim garantido que qualquer processo tem um pai.
- Quando um processo termina (neste caso o pai) o *kernel* percorre todos os processos activos para ver se algum deles é filho do processo que terminou.

  Se houver algum nestas condições, a *PID* do pai 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. (Em geral, na saída do comando *ps* o estado destes processos aparece como Z)
- Excepção: quando um processo que foi adoptado por *init* terminar não se torna *zombie*, porque *init* executa um dos *wait*'s para obter o seu *termination status*.

Processos *zombie*

- A informação acerca do filho não pode desaparecer completamente.
- O *kernel* mantém essa informação (PID do processo, *termination status*, tempo de *CPU* usado, ...) de modo a que ela esteja disponível quando o pai executar um dos *wait*'s.
- O resto da memória usada pelo filho é libertada.
- Os ficheiros são fechados.

As funções *wait* e *waitpid*

- Um pai pode esperar que um dos seus filhos termine e, então, aceitar o seu *termination status*, executando uma destas funções.
- Quando um processo termina (normalmente ou anormalmente) 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 ficarão *zombies*.
  - Dispor de um *signal handler*
    - » Em geral, o *handler* poderá executar um dos *wait*'s para obter a *PID* do filho e o seu *termination status*.

As funções *wait* e *waitpid*

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

pid_t wait(int *statloc);

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

Retornam:
   PID do processo - se OK
                -1 - se houve erro


int waitid(idtype_t idtype, id_t id, siginfo_t *infop,
int options); // wait for a child process to change state
```

Detalhes em:
http://pubs.opengroup.org/onlinepubs/9699919799/nframe.html

As funções *wait* e *waitpid*

Um processo que invoque *wait* ou *waitpid* pode

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

Diferenças entre *wait* e *waitpid*:

- ***wait* pode bloquear o processo que o invoca até que um filho qualquer termine**
- ***waitpid* tem uma opção que impede o bloqueio**
  (útil quando se quer apenas obter o *termination status* do filho)
- ***waitpid* não espera que o 1º filho termine, tem opções para indicar o processo pelo qual se quer esperar**

As funções *wait* e *waitpid*

O argumento *statloc*

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

O *status* retornado por *wait* / *waitpid* tem certos *bits* que indicam:

- **se a terminação foi normal**
- **o número de um sinal, se a terminação foi anormal**
- **se foi gerada uma *core file***

O *status* pode ser examinado (os *bits* podem ser testados) usando macros, definidas em `<sys/wait.h>`.

Os nomes destas macros começam por `WIF`.

A função *waitpid*

O argumento *pid* de *waitpid*:

- ***pid* == –1 - espera por um filho qualquer (⇔ *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*)**

***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 .**

A função *waitpid*

O argumento *options* de *waitpid*:

- **0 (zero) ou**
- **OR, *bit* a *bit* (operador |) das constantes**
  - » **WNOHANG**
    ***waitpid* não bloqueia se o filho especificado por *pid* não estiver imediatamente disponível.**
    Neste caso o valor de retorno=0.
  - » **WUNTRACED**
    se a implementação suportar *job control*
    o *status* de qualquer filho especificado por PID que tenha terminado e cujo *status* ainda não tenha sido reportado desde que ele parou é retornado.

    (*job control* - permite iniciar múltiplos *jobs*=grupos de processos a partir de um único terminal e controlar quais os *jobs* que podem aceder ao terminal e quais os *jobs* que são executados em *background*)

Exemplo

```
#include ...

int main (void)
{
 int pid, status, childpid;

 printf ("I'm the parent proc. w/PID %d\n", getpid());
 pid = fork();
 if (pid != 0)   //PARENT
  { printf ("I'm the parent proc. w/PID %d and PPID %d\n", getpid(), getppid());
    childpid = wait(&status);      /* wait for the child to terminate */
    printf("A child w/PID %d terminated w/EXIT CODE %d\n",childpid,
            WEXITSTATUS(status) );
  }
 else   //CHILD
  { printf("I'm the child proc. w/ PID %d and PPID %d\n",getpid(),getppid());
    exit(31);    /*exit with a silly number*/
  }
 printf("PID %d terminated\n",getpid()); exit(0);
}
```


Macros p/ testar o *termination status*

WIFEXITED(status)

- **==*True*, se o filho terminou normalmente.**
- **Neste caso,**
  WEXITSTATUS(status) permite obter o exit status do filho
  (8 bits menos significativos de *_exit* / *exit*) .

WIFSIGNALED(status)

- **==*True*, se o filho terminou anormalmente,**
  porque recebeu um sinal que não tratou.
- **Neste caso,**
  WTERMSIG(status) permite obter o nº do sinal
  (não há maneira portável de obter o nome do sinal em vez do número)
  WCOREDUMP(status) == *True*, se foi gerada uma *core file* .

WIFSTOPPED(status)

- **==*True*, se o filho estiver actualmente parado (*stopped*) .**
  O filho pode ser parado através de um sinal
  - **SIGSTOP, enviado por outro processo**
  - **SIGTSTP, enviado a partir de um terminal (CTRL-Z)**
- **Neste caso,**
  WSTOPSIG(status) permite obter nº do sinal que provocou o *stop*.

Exemplo

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

void pr_exit(int status);

int main(void)
{
 pid_t   pid;
 int     status;

 if ( (pid = fork()) < 0) fprintf(stderr,"fork error\n");
 else if (pid == 0) exit(7);       /* child */

 if (wait(&status) != pid) fprintf(stderr,"wait error\n");
 pr_exit(status);   /* wait for child and print its status */
//
 if ( (pid = fork()) < 0) fprintf(stderr,"fork error\n");
 else if (pid == 0) abort();       /* child generates SIGABRT */

 if (wait(&status) != pid) fprintf(stderr,"wait error\n");
 pr_exit(status);   /* wait for child and print its status */
//
 if ( (pid = fork()) < 0) fprintf(stderr,"fork error\n");
 else if (pid == 0) status /= 0;   /* child - divide by 0 generates SIGFPE */

 if (wait(&status) != pid) fprintf(stderr,"wait error\n");
 pr_exit(status);                  /* wait for child and print its status */

 exit(0);
}
```

(continua)

Exemplo (cont.)

```
void pr_exit(int status)
{
 if (WIFEXITED(status))
  printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
 else if (WIFSIGNALED(status))
       printf("abnormal termination, signal number = %d%s\n",WTERMSIG(status),
              #ifdef WCOREDUMP
               WCOREDUMP(status) ? " (core file generated)" : "");
              #else
               "");
              #endif
      else if (WIFSTOPPED(status))
            printf("child stopped, signal number = %d\n",WSTOPSIG(status));}

}
```

As funções *exec*

Existem 6 funções que designaremos genericamente por *exec*

***fork* - criar novos processos**

***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
```


As funções *exec*

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)*

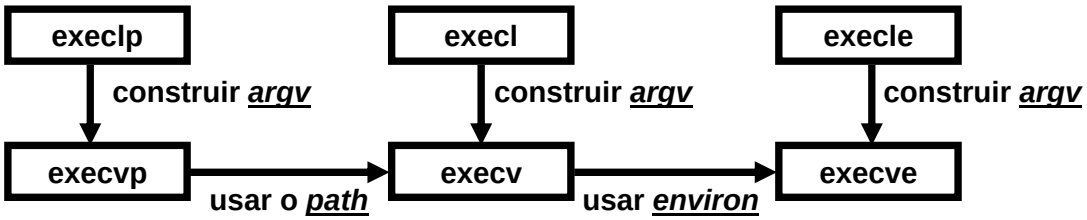
- **e - Passa-se um apontador para um *array* de apontadores para as *strings* .**
- **~~e~~ - Usar a variável *environ* se for necessário aceder às variáveis de ambiente no novo programa.**

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*.
- **~~p~~ - O nome do ficheiro executável deve incluir o *path*.**

As funções *exec*

```
# include <unistd.h>

int execlp(const char *filename, const char *arg0, … /* (char *)0 */);
int execvp(const char *filename, char *const argv[]);
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[]);

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



Exemplos

```
#include <unistd.h>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

 execl("/bin/ls","/bin/ls","-l",NULL);

_____________________________________________________________________

 char   *env_init[] = {"USER=unknown", "PATH=/tmp", NULL};
 ...
 execle("./prog","./prog","arg1","arg2",NULL, env_init);

_____________________________________________________________________

 execlp("prog","prog","arg1",NULL); /* o executável é procurado no PATH */

_____________________________________________________________________
...

int main (int argc, char *argv[])
{
 if (fork() == 0)
  { execvp(argv[1],&argv[1]);
    fprintf(stderr,"Can't execute\n",argv[1]);
  }
}
```

Se este programa se chamar ***back***,
o que faz o comando
o comando ***back gcc prog1.c*** ?

As funções *exec*

- Limite ao tamanho total da (lista de argumentos + lista de ambiente)
  - especificado por ARG_MAX (em <limits.h>)
- Propriedades que o novo programa herda do processo que o invocou:
  - *PID* e *PID* do pai
  - *real user ID*, *real group ID*
  - *process group ID*
  - *session ID*
  - terminal de controlo
  - directório corrente
  - sinais pendentes
  - limites dos recursos
  - ...
- O tratamento dado aos ficheiros abertos depende da *close-on-exec flag* (FD_CLOEXEC, actualizada pela função *fnctl*)
  - se estiver activada o descritor é fechado
  - se não estiver activada o descritor é mantido aberto (situação por omissão)

A 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);

Retorna: (v. a seguir)
```

A função *system*

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.**