3. Criação e terminação de processos

Um processo é uma instância em execução de um programa. No sistema operativo Unix a única forma de se criar um novo processo (processo-filho) é através da invocação, por parte de um processo existente e em execução, do serviço `fork()`:

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

pid_t fork(void);
```

Retorna: 0 - para o processo filho
pid do filho - para o processo pai
-1 - se houve erro e o serviço não foi executado

A função `fork()` é invocada uma vez, no processo-pai, mas retorna 2 vezes, uma no processo que a invocou e outra num novo processo agora criado, o processo-filho. Este serviço cria um novo processo que é uma cópia do processo que o invoca. Este novo processo-filho (assim como o pai) continua a executar as instruções que se seguem a `fork()`. O filho é uma cópia fiel do pai ficando com uma cópia do segmento de dados, heap e stack; no entanto o segmento de texto (código) é muitas vezes partilhado por ambos. Em geral, não se sabe quem continua a executar imediatamente após uma chamada a `fork()` (se é o pai ou o filho). Depende do algoritmo de escalonamento.

Todos os processos em Unix têm um identificador, geralmente designados por *pid* (*process identifier*). É sempre possível a um processo conhecer o seu próprio identificador e o do seu pai. Os identificadores dos processos em Unix são números inteiros (melhor, do tipo `pid_t` definido em `sys/types.h`) diferentes para cada processo. Os serviços a utilizar para conhecer *pid*'s (além do serviço `fork()`) são:

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

pid_t getpid(void);            /* obtém o seu próprio pid */
pid_t getppid(void);           /* obtém o pid do pai */
```

Estas funções são sempre bem sucedidas.

No exemplo seguinte pode ver-se uma utilização destes três serviços:

```
#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);
  return 0;
}
```

Um resultado possível será (os *pid*'s serão concerteza outros):

before fork
pid = 430, glob = 7, var = 89 as variáveis do filho foram modificadas
pid = 429, glob = 6, var = 88 as do pai permanecem inalteradas

Um processo pode terminar normalmente ou anormalmente nas seguintes condições:
Normal:
- Executa `return` na função `main()`;
- Invoca directamente a função `exit()` da biblioteca do C;
- Invoca directamente 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 Operativo.

A função `abort()` destina-se a terminar o processo em condições de erro e pertence à biblioteca standard 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 ficheiros abertos.

```
#include <stdlib.h>

void abort(void);
```

Esta função nunca retorna.

O Unix mantém sempre uma relação pai-filho entre os processos. Se o pai de um processo terminar antes do filho, este fica momentaneamente orfão. Quando um processo termina, o SO percorre todos os seus processos activos e verifica se algum é filho do que terminou. Quando encontra algum nessas condições o seu pai passa a ser o processo 1 (que existe sempre no sistema). Assim os processos que ficam orfãos são adoptados pelo processo 1, ficando assim garantido que todos os processos têm um pai.

Um processo que termina não pode deixar o sistema até que o seu pai aceite o seu código de terminação (valor retornado por `main()` ou passado a `exit()` ou `_exit()`) , através da execução de uma chamada aos serviços `wait()` / `waitpid()`. Um processo que terminou, mas cujo pai ainda não executou um dos wait's passa ao estado de zombie. (Na saída do comando *ps* o estado destes processos aparece identificado como Z). Quando um processo que foi adoptado por *init* (processo 1) terminar, não se torna zombie, porque init executa um dos wait's para obter o seu código de terminação. Quando um processo passa ao estado de zombie a sua memória é libertada mas permanece no sistema alguma informação sobre o processo (geralmente o seu PCB - *process control block*).

Um pai pode esperar que um dos seus filhos termine e, então, aceitar o seu código de terminação, executando uma das funções `wait()`. Quando um processo termina (normalmente ou anormalmente) o kernel notifica o seu pai enviando-lhe um sinal (`SIGCHLD`). O pai pode ignorar o sinal ou instalar um *signal handler* para receber aquele sinal. Nesse handler poderá executar um dos wait's para obter o identificador (pid) do filho e o seu código de terminação.

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

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

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

O argumento **`options`** de `waitpid` pode ser:
0 (zero) ou
or, bit a bit (operador |) das constantes
`WNOHANG` - que 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` - que 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.

(*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)

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.
O estado retornado por `wait()` / `waitpid()` na variável apontada por **`status`** tem certos bits que indicam se a terminação foi normal, o número de um sinal, se a terminação foi anormal, ou ainda se foi gerada uma *core file*. O estado de terminação pode ser examinado (os bits podem ser testados) usando macros, definidas em `<sys/wait.h>`. Os nomes destas macros começam por `WIF`.
O argumento **`pid`** de `waitpid()` pode ser:
`pid == -1` - espera por um filho qualquer (= `wait()`);
`pid > 0` - espera pelo filho com o pid indicado;

`pid == 0` - espera por um qualquer filho do mesmo *process group*;
`pid < -1` - espera por um qualquer filho cujo *process group ID* seja igual a `|pid|`.
`waitpid()` retorna um erro (valor de retorno = -1) se:
o processo especificado não existir;
o processo especificado não for filho do processo que o invocou;
o grupo de processos não existir.

Eis um exemplo:

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

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

  printf ("I'm the parent proc. with 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,
           status>>8);
  }
  else {  /* ***child*** */
    printf("I'm the child proc. w/ pid %d and ppid %d\n", getpid(),
           getppid());
    return 31;                                /*exit with a silly number*/
  }
  printf("pid %d terminates\n", getpid());
  return 0;
}
```

No ficheiro de inclusão `sys/wait.h` estão definidas algumas macros que podem ser usadas para testar o estado de terminação retornado por `wait()` ou `waitpid()`. São elas:
`WIFEXITED(status)` que é avaliada como verdadeira se o filho terminou normalmente.
Neste caso, a macro `WEXITSTATUS(status)` é avaliada como o código de terminação do filho (ou melhor os 8 bits menos significativos passados a `_exit()` / `exit()` ou o valor retornado por `main()`).
`WIFSIGNALED(status)` é avaliada como verdadeira se o filho terminou anormalmente, porque recebeu um sinal que não tratou. Neste caso a macro `WTERMSIG(status)` dá-nos o nº do sinal (não há maneira portável de obter o nome do sinal em vez do número).
`WCOREDUMP(status)` é avaliada como verdadeira se foi gerada uma *core file*.
Finalmente `WIFSTOPPED(status)` é avaliada como verdadeira se o filho estiver actualmente parado (*stopped*). O filho pode ser parado através de um sinal `SIGSTOP`, enviado por outro processo ou pelo sinal `SIGTSTP`, enviado a partir de um terminal (`CTRL-Z`). Neste caso, `WSTOPSIG(status)` dá-nos o nº do sinal que provocou a paragem do processo.

Outro exemplo:

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

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

void pr_exit(int status);

void 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);
}

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               /* nem sempre está definida em sys/wait.h */
           WCOREDUMP(status) ? " (core file generated)" : "");
#else
           "");
#endif
  else if (WIFSTOPPED(status))
      printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}
```

Como se viu atrás, a função `fork()` é a única no Unix capaz de criar um novo processo. Mas fá-lo duplicando o código do pai e não substituindo esse código por outro residente num ficheiro executável. Para esta última função existem no Unix seis serviços (designados genericamente por serviços `exec()`) que, embora não criando um novo processo, substituem totalmente a imagem em memória do processo que os invoca por uma imagem proveniente de um ficheiro executável, especificado na chamada.

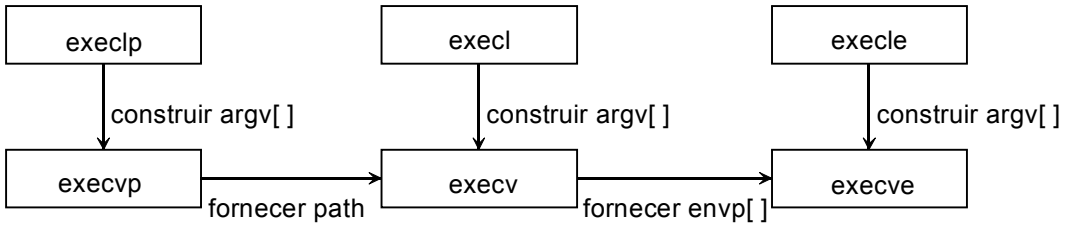
Os seis serviços são:

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, … /* NULL */ );
int execv(const char *pathname, const char *argv[]);
int execle(const char *pathname, const char *arg0, … /* NULL,
           char * const envp[] */ );
int execve(const char *pathname, const char *argv[ ],
           char * const envp[]);
int execlp(const char *filename, const char *arg0, … /* NULL */ );
int execvp(const char *filename, const char *argv[]);
```

Se os serviços tiverem sucesso não retornam (o código que os chamou é substituído); retornam –1 se houver erro.

O parâmetro **`pathname`** (string) deve indicar o nome do ficheiro executável que se pretende venha a substituir o código do processo que chama o serviço. Este nome deve ser completo no sentido de incluir todo o *path* desde a raiz. Nos serviços que contêm um p no nome (2 últimos), basta indicar o nome do ficheiro executável para o parâmetro **`filename`**. O SO procurará esse ficheiro nos directórios especificados pela variável de ambiente PATH.
Os serviços que contêm um l (letra éle) no nome aceitam a seguir ao primeiro parâmetro, uma lista de strings, terminada por `NULL`, que constituirão os parâmetros de chamada do programa que agora se vai passar a executar (deve incluir-se, como primeiro parâmetro desta lista, o nome do programa). Nos serviços que contêm a letra v esses parâmetros de chamada devem ser previamente colocados num vector `argv[]` (ver capítulo 1). Finalmente os serviços que contêm a letra e, aceitam como último parâmetro um vector (`envp[]`) de apontadores para string com as variáveis de ambiente e suas definições (ver capítulo 1).



Alguns exemplos:

```
#include    <unistd.h>
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
execl("/bin/ls", "ls", "-l", NULL);
```

```
char   *env_init[] = {"USER=unknown", "PATH=/tmp", NULL};
 ...
execle("/users/stevens/bin/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]);
    }
}
```

Neste último pedaço de código mostra-se um pequeno programa que executa outro, cujo nome lhe é passado como 1º argumento. Se este programa se chamar run, a invocação run cc prog1.c executará cc prog1.c se cc for encontrado no PATH.

Como último serviço que referiremos temos o serviço `system()`. Este serviço permite executar um programa externo do interior do processo que o invoca. O processo que invoca o serviço fica bloqueado até que a execução pedida termine. Este serviço é assim equivalente à sequência `fork()`, `exec()`, `waitpid()`.

```
#include <stdlib.h>

int system(const char* cmdstring);
```

Retorna:
-1 - se `fork()` falhou ou se `waitpid()` retornou um erro
127 - se `exec()` falhou
código de terminação do programa invocado, no formato especificado por `waitpid()` se tudo correu bem.

Exemplo:

```
system("date > file");
```