Processos API

Controle de Processos

UNIX/LINUX

Processos - pid

- **Todos os processos:**
  - Têm um identificador - pid
  - O pid é do tipo `pid_t`
  - Obter o pid de um processo `pid_t getpid()`
  - Possuem um processo pai excepto o primeiro processo do sistema – `init` com o pid=1
  - Obter o pid do processo pai `pid_t getppid()`

Processos - ps

- **Observar os processos em execução:**
  - `ps` lista processos controlados pelo terminal
  - Opções:
    - `-e` todos os processos em execução no sistema
    - `-o` definição da informação a apresentar e.g.:

      ```
      ps -e -o pid,ppid,command
      ```

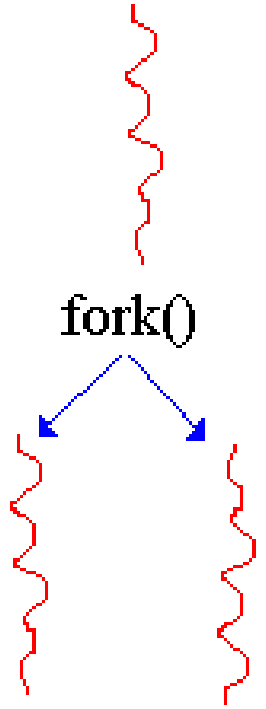
    - Mais opções consultar: `man ps`

Processos - kill

- **Terminar um processo**
  - `kill <pid>` - por omissão envia signal SIGTERM (signals)
  - `kill -9` <pid> - força a finalização do processo
  - `killall <nome>` - termina processos pelo nome

Processos - fork

- **A criação de processos é feita através da primitiva fork()que efectua uma "clonagem" do processo pai**
- **Esta primitiva é chamada 1 vez mas retorna duas vezes!**
  - Retorna o valor zero no contexto do processo filho
    - Se o filho pretender obter o pid do processo pai pode recorrer à função getppid()
  - Retorna um valor diferente de zero no contexto do processo pai
    - O valor devolvido representa o pid do processo criado (pois não existe maneira de um pai saber o pid dos seus filhos)



Processos - fork

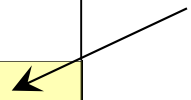
P1

```
int main ()
{


   printf("Parent process starts...\n");

   fork();

   printf("My pid = %d\n", getpid());

   printf("process %d terminating\n",
           getpid());
   return 0;
}
```

Processos - fork

P1

```
int main ()
{


  printf("Parent process starts...\n");

  fork();

  printf("My pid = %d\n", getpid());

  printf("process %d terminating\n",
          getpid());
  return 0;
}
```



O fork é chamado

Processos - fork

P1

```
int main ()
{


  printf("Parent process starts...\n");

  fork();

  printf("My pid = %d\n", getpid());

  printf("process %d terminating\n",
          getpid());
  return 0;
}
```

P2

```
int main ()
{


  printf("Parent process starts...\n");

  fork();

  printf("My pid = %d\n", getpid());

  printf("process %d terminating\n",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{


  printf("Parent process starts...\n");

  fork();

  printf("My pid = %d\n", getpid());

  printf("process %d terminating\n",
          getpid());
  return 0;
}
```

P2

```
int main ()
{


  printf("Parent process starts...\n");

  fork();

  printf("My pid = %d\n", getpid());

  printf("process %d terminating\n",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{


  printf("Parent process starts...\n");

  fork();

  printf("My pid = %d\n", getpid());

  printf("process %d terminating\n",
          getpid());
  return 0;
}
```

P2

```
int main ()
{


  printf("Parent process starts...\n");

  fork();

  printf("My pid = %d\n", getpid());

  printf("process %d terminating\n",
          getpid());
  return 0;
}
```

Processos - fork

- **Os processos pai e filho podem divergir na sua execução em função do valor devolvido pelo `fork()`**

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();

  printf("My pid = %d\n", getpid());

  printf("process %d terminating\n",
         getpid());
  return 0;
}
```

retfork = pid p2

P2

```
int main ()
{
  pid_t pid;

  printf("Parent process starts...\n");

  retfork = fork();

  printf("My pid = %d\n", getpid());

  printf("process %d terminating\n",
         getpid());
  return 0;
}
```

retfork = 0

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing\n",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing\n",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing\n",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing\n",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing\n",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing\n",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing\n",
          getpid());
  return 0;
}
```

Processos - fork

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if (retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing",
          getpid());
  return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process
    printf("new process pid = %d\n",
            getpid());
    sleep(2);
  }
  else { // parent process continues...
    printf("My child process has pid =
            %d\n", retfork);
    printf("My pid = %d and ppid = %d\n",
            getpid(), getppid());
  }
  printf("process %d finishing\n",
          getpid());
  return 0;
}
```

Processos - fork

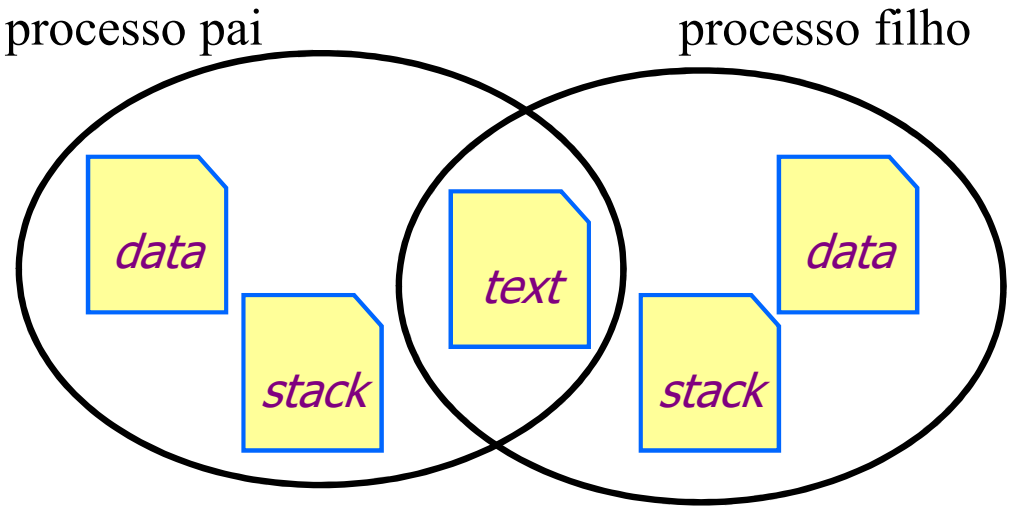
- **Pai e filho seguem os seus fios de execução na instrução seguinte ao `fork()`**

- **Nunca se sabe qual dos dois processos vai correr primeiro, é um factor que depende do núcleo**

- **O processo pai pode sincronizar com a terminação do(s) processo(s) filho(s) (ver wait)**

Processos

- **Depois de criado, o processo filho é uma cópia do processo pai:**
  - espaço de dados;
  - heap;
  - stack;
  - text (partilhado);



Processos

- **Contudo existem excepções:**
  - Normalmente a secção text (código) é partilhado:
    - Como normalmente os processos não alteram a sua zona de text em tempo de execução, o sistema pode classificar esta zona como de *read-only* e os vários processos podem partilhar o mesmo text (apenas com diferentes pontos de execução)
  - Sistemas que possam utilizar `COW` (*Copy-On-Write*):
    - Muitas vezes fazer a cópia do pai é desperdiçar recursos pois grande parte da informação não é utilizada pelo filho. Inicialmente, pai e filho partilham o mesmo espaço de endereçamento (dados, *heap*, *stack* e *text*) que tem protecção de *read-only*. Se um dos processos tenta alterar uma destas regiões, então o núcleo faz uma cópia dessa região de memória.

FUNÇÕES EXEC

Processos - exec

- **Razões para se usar um `fork()`:**
  - Quando um processo se quer duplicar para distribuir a execução de código (por exemplo servidores de rede);
  - Quando um processo quer executar um programa diferente e é usado em conjunto com o **`exec`** (por exemplo *shell*).

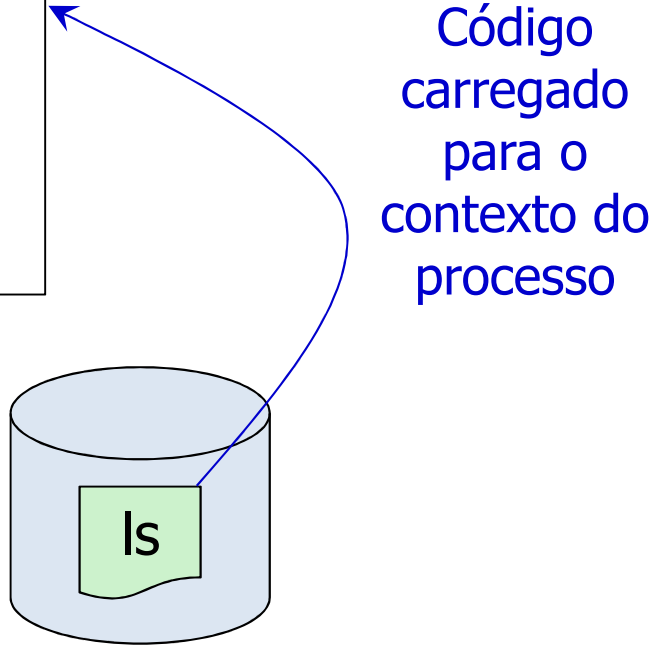
Processos - exec

- **Quando um processo evoca uma função `exec`, os seus segmentos (*text*, *data*, *heap* e *stack*) são substituídos pelo novo programa, que recomeça a executar a sua função `main`:**
  - O pid não muda (não se cria um novo processo)
  - NOTAS:
    - A primitiva **`fork`** cria um novo processo
    - A primitiva **`exec`** inicia um novo programa

Processos - exec

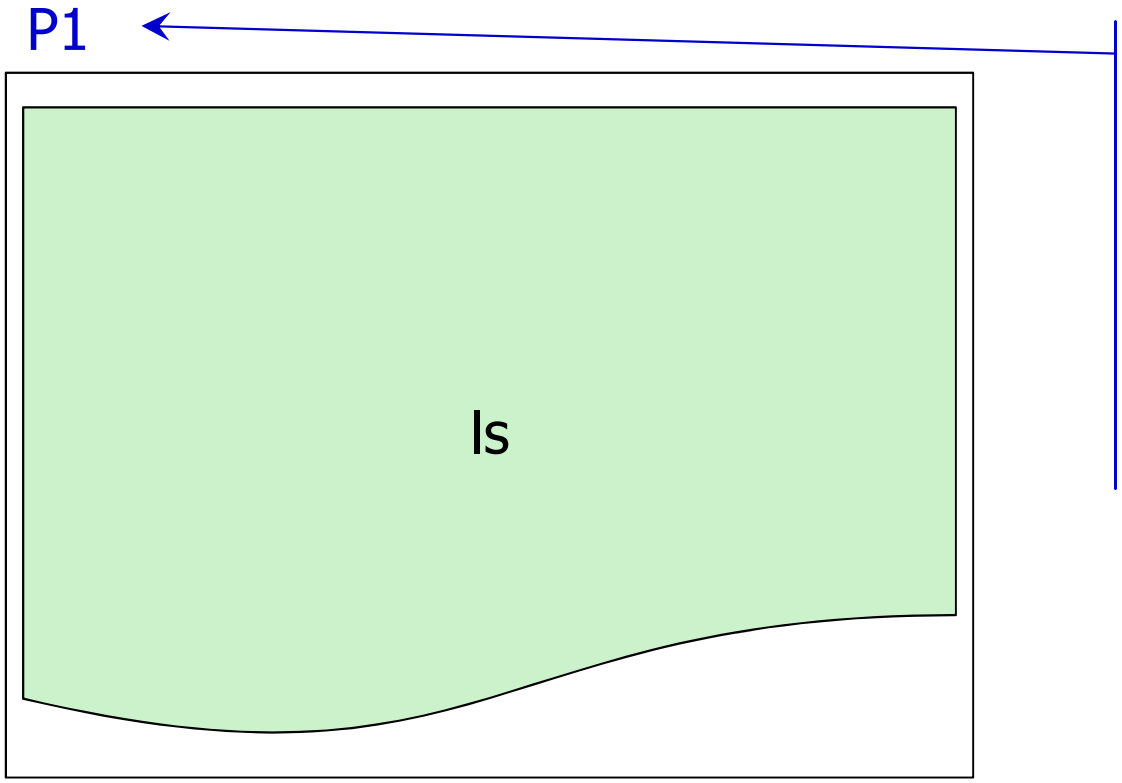
P1

```
int main ()
{
    printf("new process pid = %d\n",
            getpid());

    execlp("ls", "ls", "-l", (char*)0);

    perror("Erro no execlp");

    printf("process %d finishing\n",
            getpid());

    return 0;
}
```

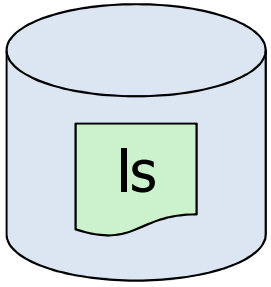


Código carregado para o contexto do processo

Processos - exec



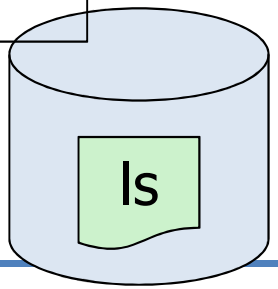
- O processo é o mesmo (pid mantém-se)
- Estrutura do processo idêntica (ficheiros abertos, etc)
- Alterações do conteúdo do espaço de endereçamento



Processos - exec

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process

    execlp("ls", "ls", "-l", (char*)0 );

    perror("Erro no execlp");
  }
  else { // parent process continues...
   ...
   printf("process %d finishing\n",
           getpid());
    return 0;
}
```



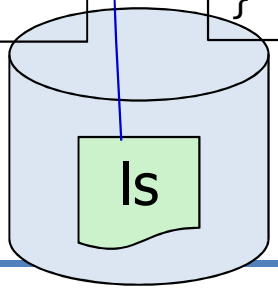
Processos - exec

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process

    execlp("ls", "ls", "-l", (char*)0 );

    perror("Erro no execlp");
  }
  else { // parent process continues...
   ...
   printf("process %d finishing\n",
           getpid());
    return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process

    execlp("ls", "ls", "-l", (char*)0 );

    perror("Erro no execlp");
  }
  else { // parent process continues...
   ...
   printf("process %d finishing\n",
           getpid());
    return 0;
}
```

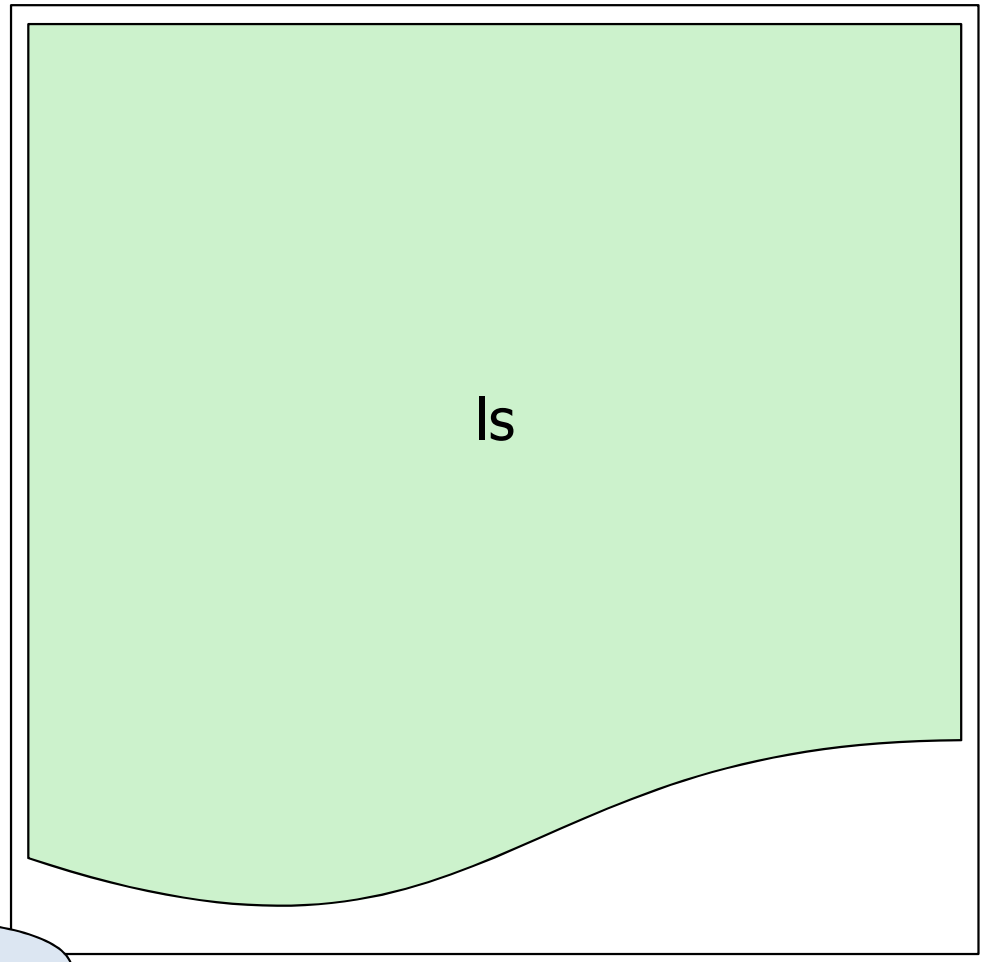


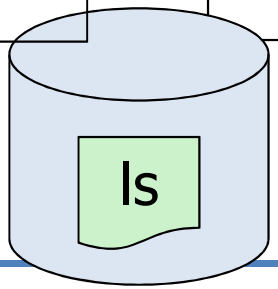
Processos - exec

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process

    execlp("ls", "ls", "-l", (char*)0 );

    perror("Erro no execlp");
  }
  else { // parent process continues...
    ...
   printf("process %d finishing\n",
           getpid());
    return 0;
}
```

P2





Processos - exec

Variantes da função exec

```
#include <unistd.h>

  execl(char *path, char *arg0, char *arg1, ..., char *argn,(char *)0);

  execv(char *path, char *argv[]);

  execle(char *path, char *arg0, char *arg1, ..., char *argn,(char *)0, char*envp[]);

  execve(char *path, char *argv, char *envp[]);

  execlp(char *file, char *arg0, char *arg1, ..., char *argn,(char *)0);

  execvp(char *file, char *argv[]);
```

Devolve -1 se erro, se não OK

Processos - exec

- **Significado das diferentes letras no nome das funções:**
  - *p*
    utiliza o nome de ficheiro binário e pesquisa no `PATH` até o encontrar.
  - *l* e *v*
    são mutuamente exclusivos e significam respectivamente *list* e *vector*
  - *e*
    environment

- **Diferenças entre as funções:**
  - As primeiras 4 usam *`pathname`* e as restantes duas *`filename`*.
  - Significa que quando se trata de um *`filename`*, o executável é procurado nas directorias especificadas pela variável de ambiente `PATH`

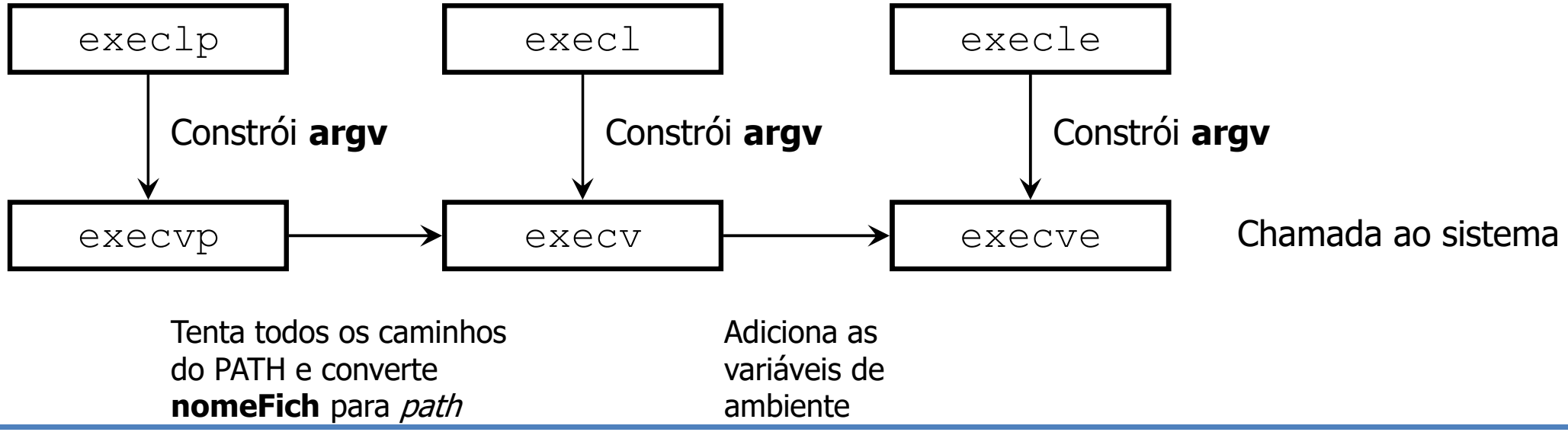
- **Lista/vector de argumentos**
  - As funções `execl`, `execlp` e `execle` requerem a passagem de argumentos separadamente
  - As funções `execv`, `execvp` e `execve` recebem um *array* de *pointers* para os argumentos

Processos - exec

- **Existência de lista de environment**
  - **`execle`** e **`execve`** permitem a passagem de um array com as variáveis de ambiente (environment) - o pai escolhe o ambiente para o filho).
  - As restantes 4 usam o ambiente do pai para o copiar para o novo programa.

Processos - exec

- **Normalmente apenas uma destas funções é uma chamada de sistema. As restantes são funções de biblioteca que evocam a chamada de sistema**
- **A utilização de uma das funções da família exec está, normalmente, associada ao serviço fork. O processo shell executa um comando através da chamada de uma função exec**



Processos - exec

- **Os processos podem terminar de "duas" maneiras distintas:**
  - De forma "normal":
    - Invocando return
    - Invocando exit
  - De forma "anormal":
    - Invocando abort
    - Quando um processo recebe um sinal do núcleo ou do shell

- **Seja qual for a forma de terminar, quando um processo termina são realizadas as seguintes operações pelo núcleo:**
  - Todos os descritores do processo são fechados
  - A memória usada pelo processo é libertada

ESPERAR PELA TERMINAÇÃO DE UM PROCESSO

Processos - terminação

- **Como é que o processo pai pode saber se o processo filho terminou e como terminou a sua execução?**
  - **O núcleo gere e mantém um *estado de terminação* que pode ser obtido pelo pai através das funções `wait`**

- **O que acontece quando:**
  - **O pai termina primeiro que o filho?**
    - O processo `init` (pid=1) lançado no fim do *boot strap* do UNIX toma conta de todos os processos "órfãos"
  - **O filho termina primeiro que o pai?**
    - *zombie* (ou *defunct*) – processo que termina mas cujo pai ainda não efectuou um `wait` sobre ele

Processos - wait

- **Sempre que um processo termina, o pai é notificado pelo núcleo através de um sinal (signal SIGHLD).**
- **O pai pode ignorar ou providenciar uma função para tratar este sinal (por omissão é ignorado)**

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

  pid_t wait(int *statloc);

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

PID se OK, ou -1 se erro

Processos - wait

P1

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process

    ...

    return 0;
  }
  else { // parent process continues...

   wait(NULL);

   printf("process %d finishing\n",
           getpid());
    return 0;
}
```

P2

```
int main ()
{
  pid_t retfork;

  printf("Parent process starts...\n");

  retfork = fork();
  if ( retfork == 0 ) { // child process

    ...

    return 0;
  }
  else { // parent process continues...

   wait(NULL);

   printf("process %d finishing\n",
           getpid());
    return 0;
}
```

Espera que um dos processos filho termine (neste caso P2)

Processos - wait

- **A função `waitpid` quando usa a opção `WNOHANG` é não bloqueante, ou seja, se o filho a que corresponde o `PID` ainda não terminou, a função retorna 0.**

- **O argumento `statloc` toma um dos valores consoante a forma como terminou o processo filho. Em `statloc` é depositado o estado de terminação do processo (quando diferente de `NULL`)**

- **Ao evocar `wait` o processo :**
  - bloqueia, se o filho ainda está a correr
  - se um processo filho já terminou retorna de imediato com o seu estado de terminação
  - se não possui nenhum processo filho devolve erro

- **A função `wait` espera pelo primeiro processo filho que termine**

Processos - wait

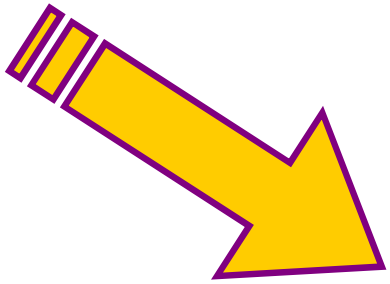
- **As condições de terminação de um processo podem ser determinadas com base no estado recebido. Esta operação deve ser efectuada recorrendo às macros que a seguir se sintetizam:**

Macro	Descrição	Acção
WIFEXITED(status)	TRUE se terminação normal	WEXITSTATUS(status)
WIFSIGNALED(status)	TRUE se terminação anormal	WTERMSIG(status)
WIFSTOPPED(status)	TRUE se o processo foi parado	WSTOPSIG(status)

Processos - exemplo do shell

- **Estrutura interna de um shell:**

```
~$ cat Makefile
```

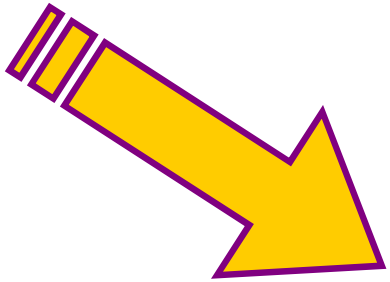


```
...
if ( fork()==0 ) {
  execlp( “cat”, “cat”, “Makefile”, (char*)0 );
}
else {
  wait( (int*)0 );
}
...
```

Processos - exemplo do shell

- **Estrutura interna de um shell:**

```
~$ cat Makefile &
```



```
...
if ( fork()==0 ) {
  execlp( “cat”, “cat”, “Makefile”, (char*)0 );
}
else {
  if ( linha não terminada por & ) {
    wait( (int*)0 );
  }
}
...
```

O utilizador pode optar pela execução do comando em concorrência com o shell e/ou outros processos, bastando para tal, terminar a linha de comando com o caracter ‘**&**’.

Referências

- **R. Arpaci-Dusseau, A. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, Mar 2015 [ch 1-6]**

- **Mark Mitchell, Jeffrey Oldham, and Alex Samuel, Advanced Linux Programming, 1st ed, 2001, online:** http://advancedlinuxprogramming.com/alp-folder/advanced-linux-programming.pdf **[3. Processes p. 45]**