FACULDADE DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

Fundamentos de Sistemas de Operação

LEI - 2023/2024

Vitor Duarte
Mª. Cecília Gomes

1

Aula 4

- Processos
  - API de processos: fork, wait, exit, execve
- OSTEP: cap. 5

2

Clonagem de processo

- Fork cria um novo processo
  - A sua descrição é idêntica ao processo original
  - Mesmo mapa de memória e CPU, mesmo estado de execução (READY), mesmo UID e permissões, mesmos canais IO, mesma diretoria corrente,...
  - Novo PID, memória virtual cópia da original
- A chamada ao sistema vai retornar para dois processos (original e clone), devolvendo valores diferentes
- O novo processo fica pronto a executar como qualquer outro

Interface Posix/Unix

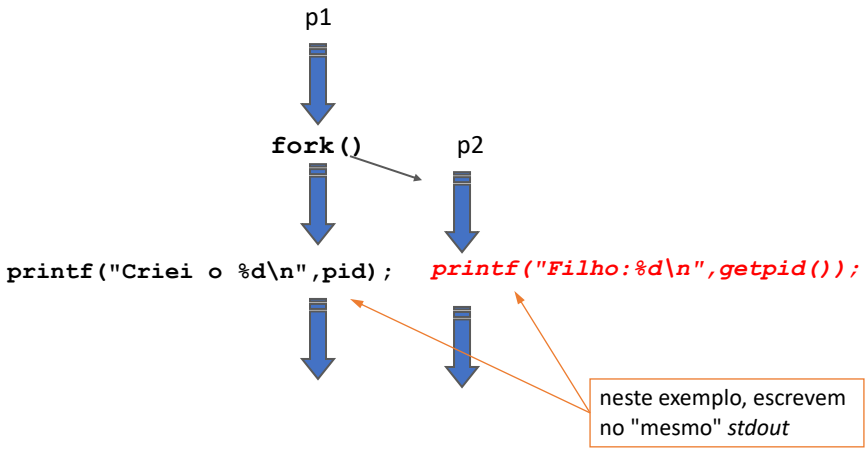
`int fork(void)`

- cria um novo processo (filho) cópia do original (pai)
  - incluindo: programa, atributos e informação sobre canais abertos (o *pid* é diferente, claro!)
- devolve o *pid* do filho ao pai
  - devolve -1 se erro
- o filho, sendo uma cópia do pai (tem o mesmo IP), começa a executar na instrução imediatamente a seguir ao `fork()`!
  - o SO também tem de devolver um resultado do `fork` no novo processo: devolve zero!

Exemplo:

```
pid = fork();
switch (pid) {
  case 0:
      printf( "Filho:%d\n", getpid() );
      exit(0);
  case -1:
       perror("fork");
       break;
  default: printf("Criei o %d\n", pid);
}
```


Exemplo:



Exemplo 2:

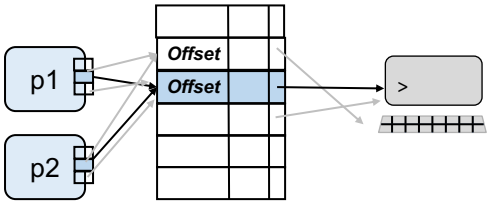
```
void hello(int n) {
    pid_t pid = getpid();
    for (int i=0; i<n; i++) {
        printf("ola de %d\n", pid);
        sleep(1);
    }
}
-------------------------------------------------
    int pid = fork();   // assumindo que não há erro
    printf("%d está vivo!\n", getpid() );
    if (pid==0) {
        hello(10);
        exit(0);
    } else
        hello(5);
```


Processos e I/O no UNIX

- Em cada processo são assumidos 3 canais standard
  - descritores: 0=stdin, 1=stdout, 2=stderr
- Estes podem representar muitas coisas
  - Tipicamente o terminal, mas podem ser outros ficheiros
  - Quando se executa na shell, pode-se alterar:
    - Ligar estes a ficheiros em disco, ou a outros processos!
    - ver notações: >, <, | no manual
- Após fork?
- Canais partilhados entre pai e filho
- No exemplo anterior: escrita em concorrência no ecrã por vários processos:
  - o processo original
  - o processo filho
  - o shell, quando o pai acaba e o filho continua em execução

Fork e IO

- Onde aparecem as mensagens?
  - Filho "herda" os canais!
  - descritores copiados, logo apontam os mesmos canais
- Por que ordem fica o output?
  - As mensagens aparecem misturadas
  - Depende do **escalonamento** do SO
  - Até podem executar em simultâneo, se existirem vários CPU!
- Como sincronizar, fazendo o pai esperar que o filho termine?



se *standard output* de ambos é um terminal

p2 filho de p1

Tabela de fich. abertos

Terminação de processos

- Normal (vontade própria):
  - retorno de **`main`**: acaba por chamar **`exit`**/**`_exit`**
  - **`exit`**: chama ações finais (**`atexit`**), todas as escritas são *flushed*, chama **`_exit`**
  - chamada de **`_exit`**: chamada ao SO, entrega o 'exit status' do processo. Os IO fechados, o processo destruído
  - Convenção: *status* 0 se correu tudo normalmente; *status* > 0 se houve algum problema
- Anormal:
  - ocorrência de uma notificação (sinal) vinda do *hardware* ou *software.*
  - Exemplos: *Floating point exception*, *Invalid memory reference*, *Kill*, Ctrl-C, …
  - Estes sinais normalmente terminam o processo.

Esperar pela terminação

`int wait(int *status)`

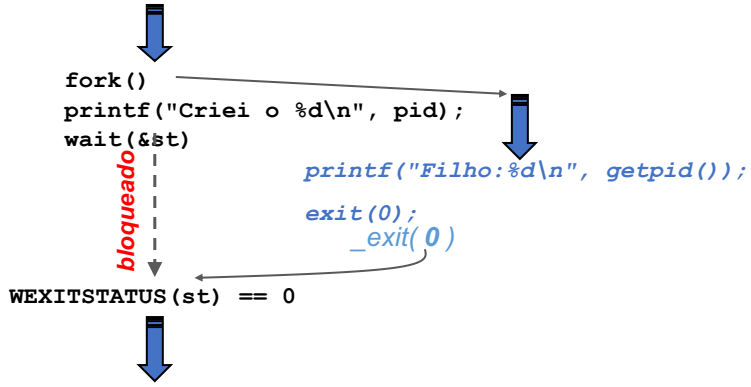
- Permite aguardar pela terminação de um dos filhos (ou pela sua paragem/recomeço)
- status: retorna estado da saída do processo
- A macro **`WIFEXITED()`** permite testar se a terminação foi normal ou não
- se terminou com _exit pode-se obter o exit satus com a macro **`WEXITSTATUS()`**

Voltando ao Exemplo:

```
pid = fork();
switch (pid) {
  case 0: printf( "Filho:%d\n", getpid() );
          exit(0);
  case -1: perror("fork");
           break;
  default: printf("Criei o %d\n", pid);
           childpid = wait( &st );
           if (WIFEXITED(st))
               printf("%d saiu com %d\n",
                      childpid, WEXITSTATUS(st));
           else
               printf("%d terminou ou parou\n");
}
```

Esperar pela terminação

- **`int wait(int *status)`**
  - Bloqueia esperando pela terminação de um dos filhos
  - `status`: retorna o *status* do processo (se terminou com `_exit` inclui o *exit satus* passado no exit)



voltando ao Exemplo 2:

```
void hello(int n) {
    pid_t pid = getpid();
    for (int i=0; i<n; i++) {
        printf("ola de %d\n", pid);
        sleep(1);
    }
}
-------------------------------------------------
    int pid = fork();   // assumindo que não há erro
    printf("%d está vivo!\n", getpid() );
    if (pid==0) {
        hello(10);
        exit(0);
    } else {
        hello(5);
        wait(NULL);     // espera que filho termine
    }                   // nao quer status para nada
```

wait: cenários possíveis

- O pai e o filho são executados concorrentemente pelo sistema de operação.
- Cenários possíveis:
- Pai chega primeiro à chamada de **`wait`**
  - fica bloqueado até o filho terminar (**`_exit`**)
- Filho chama termina antes do pai chamar **`wait`**
  - o filho fica zombie (*defunct*)
  - quando pai chamar **`wait`** , não bloqueia, recebe o status e o que resta do filho é eliminado.
- Pai não chama **`wait`** e termina
  - o filho fica órfão: pode ser adotado pelo *init*

waitpid

```
int waitpid(int pid,
            int *status, int options)
```

- se pid > 0 aguarda pelo filho indicado
- se pid = -1 qualquer filho (como **`wait`**)
- status: como no **`wait`**
- options: **`WNOHANG`** - não bloquear se nenhum filho terminou (testa apenas)

`wait(&status)` é equivalente a

```
waitpid(-1, &status, 0);
```

Usos do fork

- Tirar partido da multiprogramação
  - Criar várias instâncias do mesmo programa
  - p. ex: lançar várias instâncias de servidores Web
  - p. ex: um programa inclui diversas ações que podem ser executadas em concorrência/paralelo
- Combinar **fork+exec** para criar processos com novos programas

Novo programa no mesmo processo

- Exec... troca de programa
  - Novo mapa de memória com novo conteúdo: o programa a executar
  - Novo estado do CPU com valores iniciais para o novo programa (IP, SP, etc)
  - Mesmo PID, mesmo UID
  - Permissões, canais IO, diretoria corrente, etc. como estavam antes do exec
- A chamada ao sistema não retorna (o programa que chamou já não existe).
  - exceto se houver erro

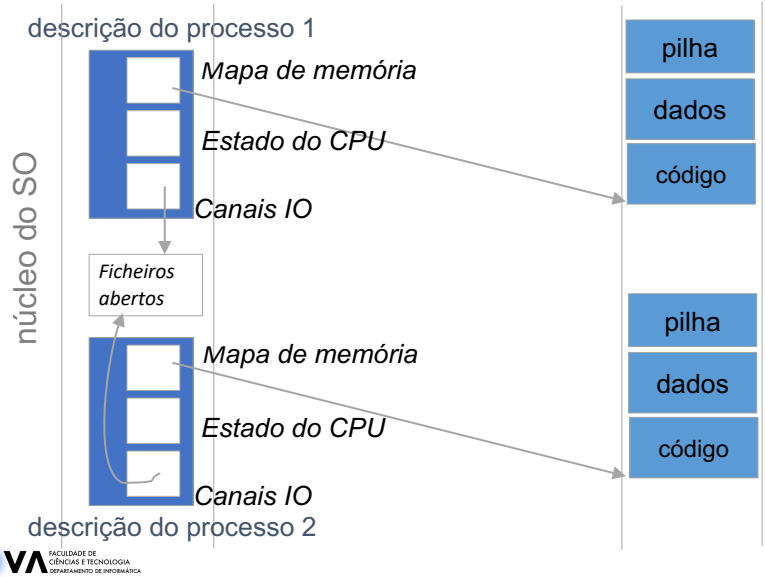
Executar um programa: execve

```
int execve(char*path, char*argv[],
                      char*envp[])
```

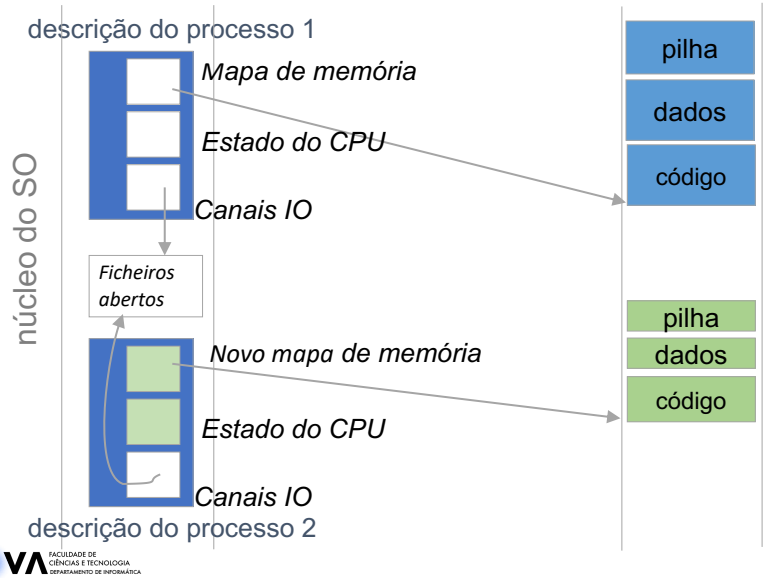
- Cria um novo mapa de memória no processo (substitui o antigo mapa)
  - na prática, passa a executar um novo programa! (o antigo deixa de existir)
  - copia para memória no novo mapa, o argc, argv e envp
- O resto do processo mantém-se
  - atributos e canais de I/O (há excepções)
- Exemplo:

```
char *argv[] = {"ls", NULL};
execve("/bin/ls", argv, environ);
```


Dois processos...



Processos após exec em processo 2



21

Início de um novo programa

- O código de inicialização (no "start address" especificado pelo compilador) é chamado:
  - p.e. **`_start`**
- Se programa em C, no *Run Time* (**`crt`**):
  - obtém argumentos e vars. ambiente iniciando a pilha
  - chama a função main() com os argumentos recebidos do execve
  - Equivale a:

```
exit( main(argc, argv, envp) );
```


22

execve e Companhia

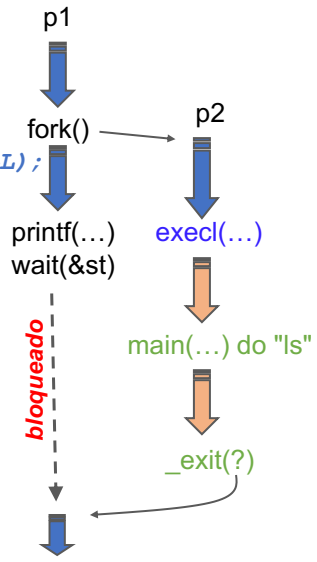
- Funções de biblioteca:

```
execl(char*path, char*arg0, char*arg1,…)
execle(char*path, char*arg0,
       char*arg1,…,0, char*envp[])
execv(char*path, char*argv[])
execlp(char*file, char*arg0, char*arg1,…)
execvp(char*file, char*argv[])
```

  - as duas últimas usam as diretorias em `PATH` do ambiente corrente para encontrar `file`

fork + exec - executar "ls /bin"

```
pid = fork();
switch (pid)
{
  case 0:
       execl("/bin/ls","ls","/bin",NULL);
       perror("ls");
       exit(1);
  case -1:
       perror("fork");
       break;
  default:
       printf("Criei o %d\n", pid);
       wait(&st);
}
```



onde escreve o ls?

Alterando os canais

- Onde o ls vai escrever agora?

```
switch (pid = fork()) {
  case 0:
        close(1);
        creat("meu-output", 0666);
        execl("/bin/ls","ls","/bin",NULL);
        perror("ls");
        exit(1);
  case -1:
        perror("fork");
        break;
  default:
        printf("Criei o %d\n", pid);
        wait(&st);
}
```

alterando o standard output

msg se erro no execl e saindo com status 1 (!=0)

onde escreve o ls?