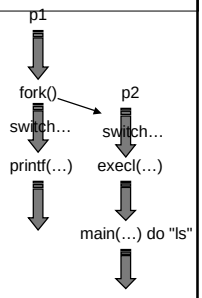
Usos do fork

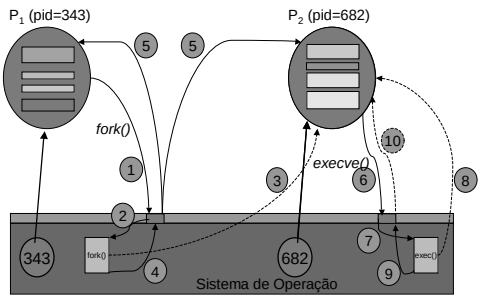
- **Criar várias instâncias do mesmo programa**
  - **p. ex: lançar o login em vários terminais**
- **Tirar partido da multiprogramação**
  - **p. ex: um programa inclui diversas acções que podem ser executadas em concorrência**
- **Combinar fork+execve para criar processos com novos programas**
  - **(antes do execve o novo processo pode alterar o ambiente para o novo programa)**

fork + exec*

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



sequência fork() + execve()



fork vs. execve

`fork`	`execve`
mapa mem. pai copiado para filho	mapa de mem. substituído pelo novo programa
retorna pid do filho ao pai (filho recebe zero)	nunca retorna (só em caso de erro)
pid filho diferente do do pai (processos diferentes!)	o pid mantém-se (mesmo processo!)
filho herda uma copia dos canais de i/O	mantém os mesmos canais (mesmo processo!)

Chamadas Unix para Processos

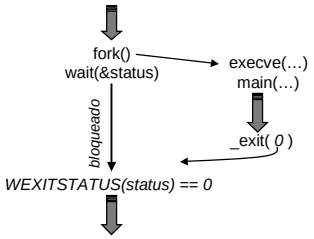
- **`fork`: cria um duplicado exacto do processo original incluindo informação sobre canais abertos**
- **`exec*`: troca o programa do processo, a partir de um ficheiro executável (um novo mapa de memória)**
- **`_exit/exit`: termina o processo corrente**
- **`wait`: permite a um processo aguardar pela terminação de um filho**

Um *shell* simplificado

```
while (true) {
  read_command(command, params);
  if ( fork() > 0 )
    // o pai espera
    wait (&status);
  else
    // o filho executa o comando
    execve(command,params,0);
}
```

Esperar pela terminação

- **`int wait(int *status)`**
  - **aguarda pela terminação de um dos filhos**
  - **`status`: retorna como terminou o processo (se terminou com `_exit` pode-se obter o *exit satus*)**



wait: cenários possíveis

- **O pai e o filho são executados concorrentemente pelo sistema de multiprogramação. Cenários possíveis:**

1. **Pai chega primeiro à chamada de `wait`**
   - **fica bloqueado até o filho chamar `_exit`**
2. **Filho chama `_exit` antes do pai chamar `wait`**
   - **o filho fica *ZOMBIE (defunct)***
   - **quando pai chamar `wait` não bloqueia, e o resta do filho é eliminado.**
3. **Pai não chama `wait` e termina**
   - **o filho fica ÓRFÃO: é adoptado pelo `init`**

waitpid

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

Exemplo

```
pid = fork();
switch (pid)
{
  case 0:
      chdir("/bin");
      execl("/bin/ls","ls",0);
      printf(stderr,"erro no execl\n");
  case -1: perror("fork");
           break;
  default:
      printf("Criei o %d\n", pid);
      wait( &st );
      if ( WIFEXIT(st) )
         printf("exit status: %d\n", WEXITSTATUS(st) );
}
```

Exemplo (2)

```
pid = fork();
switch (pid)
{
  case 0:
      { char *av[3];

        av[0]="ls"; av[1]="/bin"; av[2]=NULL;
        execvp( "ls", av );
        perror( "exec ls" );
        exit(1);
      }
  case -1: perror("fork");
           break;
  default:
      wait( &st );
}
```


Exemplo (3): redirecção de I/O

```
pid = fork();
switch (pid)
{
  case 0:
        close(1);
        creat("/tmp/resultado", 0666);
        execlp( "ls", "ls","/bin",0 );
        perror( "exec ls" );
        exit(1);
  case -1: perror("fork");
           break;
  default:
      wait( &st );
}
```