

# *Fundamentos de Sistemas de Operação*

Unix Windows NT Netware Mac OS DOS/V/VS Vax/VMS  
Linux Solaris HP/UX AIX Mach Chorus

*A API de Processos:  
Conclusão?*

# *Operações fundamentais: exec\*()*

- O exec não é uma única função de sistema, mas sim uma família de várias, que diferem no número e tipo dos argumentos.
- Porque é assim? Porque umas são mais fáceis de usar numa situação, outras noutra, etc.
- São elas: `exec1()`, `execlp()`, `execle()`, `execv()`, `execvp()`, `execvpe()`
- Os sufixos têm um significado: *l*—lista; *v*—vector, *e*—environment; *p*—path. Isto é, um exec que tem um *l* no nome, usa uma lista de argumentos, enquanto um que tem *v* usa vector(es) de argumentos. Se tem *p* usa a variável PATH do ambiente,...

# *exec\*() para que serve?*

- As funções da família exec permitem que um processo “oblitere” ☺ (“escreva por cima” – overwrite) da suas zonas do EE (código, dados, stack, heap) um novo mapa obtido por leitura do ficheiro executável indicado no argumento da função.
- Exemplo: imagine que um processo (pode ser uma shell) executa a seguinte sequência de instruções:

```
if (fork())
    wait(...);                                // espera pelo filho
else
    execl("/bin/ls", "ls", "-l", NULL);      // shell filho
```

- O *fork()* lança um filho (outra shell), mas este executa o *ls*, ou seja, esmaga o código */bin/bash* com */bin/ls*, e esmaga também o resto do EE

# Demo: execl ()

```
int main()
{
    ...
    pid=fork();
    if (!pid) {
        printf("Sou o processo filho, PID:%d\n", getpid());
        printf("vou tentar listar os ficheiros existentes na...\\n");
        execl("/bin/ls", "ls", "-l", NULL);
        printf("Se esta mensagem aparece, e' porque o execl falhou\\n");
    } else {
        printf("Sou o processo pai, PID:%d\n", getpid());
        wait(NULL);
    }
}
```

# *Fundamentos de Sistemas de Operação*

Unix Windows NT Netware Mac OS DOS/V/S Vax/VMS  
Linux Solaris HP/UX AIX Mach Chorus

*Programas, Processos e o SO:  
Comunicação entre processos: I*

# Comunicação entre Processos no Unix

- No UNIX estão disponíveis diferentes mecanismos para possibilitar a troca de informação entre processos que residem no mesmo computador
  - PIPEs (*canos ou tubos*), usados como se fossem ficheiros onde se lê (`read()`) e escreve (`write()`)
  - MESSAGE QUEUES, onde se colocam (`msgsnd()`) e retiram (`msgrcv()`) mensagens
  - SHARED MEMORY, zona partilhada onde dois ou mais processos podem ler e escrever usando “variáveis” visíveis para todos
  - Outros...

# *Comunicar usando um ficheiro* (1)

```
int main()
{
    ...
    p=fork();
    ...
    if (p) {                                // pai escreve
        fd=open("ficheiro", O_WRONLY...);
        write(fd, "ola", 3);
    } else {                                 // filho lê
        fd=open("ficheiro", O_RDONLY...);
        read(fd, buf, 3);
    }
    ...
}
```

- Que acontece se o *filho ler e o pai ainda não escreveu?*
- Como garantir que o *filho só lê depois do pai escrever?*

# Comunicar usando um ficheiro (2)

```
int main()
{
    ...
    p=fork();
    if (!p) {                                // filho escreve... e termina!
        fd=open("ficheiro", O_WRONLY...);
        write(fd, "ola", 3);
    } else {
        wait(NULL);                          // pai espera...
        fd=open("ficheiro", O_RDONLY...);
        read(fd, buf, 3);                  // ... depois lê
    }
}
```

- Assim garantimos que o pai só lê depois do filho escrever...
- Mas só corre um de cada vez ☹ ... má utilização de recursos... CPUs desaproveitados... programa demora mais...

# Comunicar usando um ficheiro (3)

## Demo

```
int main()
{
    ...
    fd=open("dem-01a.txt", O_RDWR...O_TRUNC,...);
    p=fork();
    if (!p) {                                // filho escreve... e termina!
        write(fd, "ola", 3);
    } else {                                 // pai
        wait(NULL);                         // espera...
        read(fd, buf, 3);                  // ... depois lê
    }
}
```

- Será que funciona? [NÃO]
- Porquê? [veremos mais tarde...]

# Comunicar usando um ficheiro (4)

## Demo

```
pid=fork();
if (!pid) {
    printf("Sou o processo filho, PID:%d\n", getpid());
    if ( (fd= open("dem-01b.txt", O_RDWR|O_CREAT|O_TRUNC, 0660) ...
        write(fd, "ola", 3);
} else {
    printf("Sou o processo pai, PID:%d\n", getpid()); wait(NULL);
    fd= open("dem-02.txt", O_RDONLY);
    read(fd, buf, 3);
    buf[3]= '\n'; printf("Tenho buf=%s\n",buf);
}
```

- Será que funciona? [SIM] Porquê? [veremos mais tarde...]

# *Comunicação vs. Sincronização*

- Há comunicação entre processos quando
  - Há transferência de informação (bytes) entre os espaços de endereçamento dos processos: há emissor(es) e receptor(es).
- Há sincronização entre processos quando
  - Um processo assinala a outro(s) a ocorrência de um dado evento; por ex., um processo espera que outro termine. O evento acontece quando termina!
- Note-se que há uma certa dualidade entre as duas
  - quando um processo espera por uma mensagem de outro, há simultaneamente sincronização e comunicação...

# Pipes (1)

- Um pipe (“cano”) é uma abstracção do SO que permite a (um, dois ou até mais...) processos comunicarem unidireccionalmente
  - Numa “extremidade” um descritor para leitura e na “outra extremidade” um descritor para escrita permite a troca de informação (fluxo de bytes)...



# Pipes (2)

- `int pipe(int fd[2])`

Se a execução da função tiver sucesso, o conteúdo do vector de dois inteiros `fd` é tal que `fd[0]` identifica a extremidade de leitura do pipe enquanto `fd[1]` identifica a de escrita no pipe.

```
int fd[2];  
  
if ( pipe(fd) == -1) abort();  
...  
// fd[0] é o canal para ler do pipe  
// fd[1] é o canal para escrever no pipe
```

# *Semântica das operações de E/S sobre pipes (1)*

## □ `int read(int fd, void *buf, size_t sz)`

*Um `read()`, quando executado sobre um pipe, comporta-se por vezes de forma diferente do que quando executado sobre um ficheiro:*

- *Se não há dados para ler, o processo fica bloqueado, à espera... [no caso do ficheiro retornava zero]*
- *Se a quantidade de dados disponível é inferior à pedida retorna a quantidade existente [tal como no caso do ficheiro]*
- **Caso especial:** se o canal de escrita está fechado e o pipe está vazio, a leitura não bloqueia, retorna zero. É assim que o escritor assinala que não mais quer comunicar...

# *Semântica das operações de E/S sobre pipes (2)*

## □ **int write(int fd, void \*buf, size\_t sz)**

*Um **write()**, quando executado sobre um pipe, comporta-se por vezes de forma diferente do que quando executado sobre um ficheiro.*

*O SO define uma capacidade máxima para o pipe armazenar (ainda que temporariamente) dados*

- Se **sz** é menor do que a capacidade máxima, mas o espaço disponível no pipe não chega para armazenar os **sz** bytes, o escritor bloqueia até que um leitor tenha lido o suficiente para que caibam os **sz** bytes, duma só vez...
- Se o valor **sz** é maior do que a capacidade máxima, apenas uma parte dos dados são escritos (a que couber), e o escritor bloqueia até que um leitor tenha lido alguns bytes... nessa altura são escritos mais alguns...

# Comunicação usando um pipe (1)

```
int main()
{
    ...
    if ( pipe(fd) == -1 ) abort();           // cria pipe e abre fd's
    pid=fork();
    if (pid) {
        write(fd[1], "ola", 3);             // pai escreve
        ...
    } else {
        read(fd[0], buf, 3);               // filho lê
        ...
    }
}
```

- Pai e filho executam concorrentemente ☺...
- O SO sincroniza os processos de forma indirecta, através das operações **read()** e **write()** ...

# Demo: Comunicação usando um pipe (2)

```
int main()
{
    int pid, fd[2];

    if ( pipe(fd) == -1 ) abort(); // cria pipe e abre fd's para R e W
    pid=fork();
    if (pid) {
        close(fd[0]); // pai vai W, não precisa de R
        write(fd[1], "ola", 3); // pai escreve
    } else {
        close(fd[1]); // filho vai R, não precisa de W
        read(fd[0], buf, 3); // filho lê
    }
}
```

- Código praticamente completo! ☺...

# Comunicação usando um pipe (3)

Unix Windows NT Netware Mac OS DOS/VMS Vax/VMS  
Linux Solaris HP/UX AIX Mach Chorus

- Um pipe (“cano”) é uma abstracção do SO que permite a (um, dois ou até mais...) processos comunicarem unidireccionalmente
  - Numa “extremidade” um descritor para leitura e na “outra extremidade” um descritor para escrita permite a troca de informação (fluxo de bytes)...



# *Pipes: notas finais*

- *Unnamed pipes (pipes anónimos, ou “sem nome”)*
  - Este tipo de pipes, que acabamos de ver, só pode ser usado pelo processo que cria o pipe e seus descendentes, pois o “objecto” criado não pode ser visto no sistema de ficheiros, que é onde (quase) todos os objectos visíveis aparecem...
- *Named pipes*
  - Criados (hoje em dia) com uma função de biblioteca, `mkfifo()`, aparecem no sistema de ficheiros com um nome e permissões, e qualquer processo pode tentar ☺ “abri-los” (com `open()`) e “aceder-lhes” (com `read()` e/ou `write()`)