

Conteúdo

1	Processos e sinais	2
1	Processos	2
1.1	Chamadas ao sistema (system call)	2
1.2	Hierarquia de processos	3
1.3	Shell: comandos e programas	4
1.4	Criação de processos: fork()	4
1.5	Mais sobre o fork(): processo pai e processo filho	5
1.6	Execução de outros programas: exec	5
1.7	Executar outro programa: fork e exec	6
1.8	wait	6
1.9	Exemplo: uma pequena shell	7
2	Sinais	7
2.1	Enviar um sinal	8
2.2	Tratar um sinal	9
2.3	alarm	10
2.4	Espera ativa e passiva	11

Processos e sinais

1 Processos

Um programa em execução diz-se um *processo*. Cada processo é identificado pelo sistema operativo através de um número único - o *pid* (process identifier).

Experimente o seguinte programa:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
main() {
    char s[80];
    printf ("PID=%d\n", getpid() );
    while ( 1 ) {
        printf ("Comando: ");
        fgets(s,80,stdin);
        s[strlen(s)-1]=0;
        if ( strcmp ( s, "exit" ) == 0 )
            exit(0);
    }
}
```

O programa começa por escrever no ecrã o seu identificador de processo; depois fica à espera que o utilizador introduza um comando. O único comando com efeito é o "exit", que faz o programa terminar. Tendo este programa em execução abra outra janela e execute o comando:

```
ps -u
```

Este comando permite ver a lista de todos processos pertencentes ao mesmo utilizador. Verifique que o programa aparece na lista (pode identificá-lo pelo *pid*).

Se puser este mesmo programa em execução em várias janelas terá vários processos diferentes (todos a executar o mesmo programa). Experimente e verifique com `ps -u`.

1.1 Chamadas ao sistema (system call)

O programa do ponto anterior usa a função `getpid()` para saber o seu *pid*.

A função `getpid()` origina uma chamada ao kernel do sistema operativo (system call). Da mesma forma, a função `exit()` origina uma chamada ao sistema operativo, neste caso pedindo ao sistema operativo para terminar o processo. Para saber mais pormenores sobre estas funções pode consultar o manual. Por exemplo:

```
man getpid
```

Um dos aspectos importantes do manual é a indicação dos "*includes*" que deve fazer para usar a função (pormenor que neste caso descurámos...).

Deve executar o comando `man nome_da_função` para saber quais os includes que deve fazer no seu programa em C no qual faça uso de uma qualquer função de sistema.

Estas funções de sistema operativo (às vezes chamadas também de "primitivas") são apresentadas na secção 2 dos manuais do Unix/Linux. Ocasionalmente o nome de uma função do sistema pode coincidir com o nome de um comando que esteja noutra secção do manual. Nesses casos pode ser preciso explicitar a secção do manual onde se pretende procurar. Por exemplo:

```
man 2 getpid
```

pede explicitamente o manual de `getpid` existente na secção 2.

Na realidade `getpid()` ou `exit()` são funções de biblioteca da linguagem C a que, com alguma liberdade perfeitamente aceitável, chamamos *system calls*. Mais rigorosamente, o que estas funções fazem é preparar a chamada ao sistema, a qual, especificamente, é feita através de uma interrupção de software ("trap") ao sistema operativo.

1.2 Hierarquia de processos

Quando abrimos uma janela "consola" no interface gráfico ou fazemos login num ecrã alfa-numérico ficamos perante um programa que lê e executa os nossos comandos. Um programa deste tipo chama-se uma "shell". No Linux a variante de shell mais provável é a designada `bash` (programa `/bin/bash`).

O trabalho da shell é repetitivo. Basicamente, consiste em:

- pedir um comando (fazendo aparecer o "pronto">);
- ler e interpretar o comando;
- executar o comando (ou dar erro, se o comando for inválido)

e repetir, fazendo reaparecer o "pronto" para pedir um novo comando.

Os comandos podem ter efeito e duração variados. Por exemplo um `ls -l` ou um `ps` listam informação no ecrã e terminam de imediato. Um comando como o `vi` toma conta do ecrã e termina apenas quando o utilizador terminar a edição saindo do `vi`. Seja como for, no fim do comando, a shell reaparece para pedir outro comando.

Para executar um comando, a shell cria um novo processo. O novo processo fica ligado (diz-se que é um processo filho) do processo que o criou (que se diz o seu processo pai). Um processo pode saber o número do seu processo pai através da função `getppid()`. O seguinte programa exemplifica esta relação:

```
#include <stdio.h>
main() {
    printf ("PID = %d\n", getpid() );
    printf ("Processo pai = %d\n", getppid() );
}
```

O programa escreve no ecrã o seu número de processo e o número do seu processo pai. Sem surpresa, como pode verificar facilmente executando este programa e depois fazendo `ps`, o `pid` do processo pai é justamente o `pid` da shell que o criou.

1.3 Shell: comandos e programas

A generalidade dos comandos que damos à shell para executar são, na realidade, programas externos à própria shell. Nestes casos o que a shell faz é localizar o programa correspondente ao comando dado e criar um novo processo para o executar.

Por exemplo, o comando `ls` corresponde a um programa que se pode encontrar, tipicamente, no directório `/bin` (pode verificar fazendo `ls /bin/ls`). Assim, executar um comando como `ls -l` é, na realidade, executar o programa `/bin/ls`, passando-lhe `-l` como argumento.

No mesmo directório `/bin` pode também encontrar muitos outros comandos (programas) de uso comum: `cp`, `ps`, etc. Possivelmente encontrará também aí o próprio programa shell, num ficheiro `/bin/bash`. Se fizer o comando

```
bash
```

iniciará a execução de uma nova shell, que ficará "sobrepota" à anterior, passando a aceitar comandos na janela/terminal. Nestas circunstâncias, fazendo `ps`, verá aparecer dois processos `bash` (ou mais, se executar o comando `bash` várias vezes). Para terminar a shell dá-se o comando:

```
exit
```

Este é um exemplo de um comando interno, isto é, um comando executado pela própria `bash`.

1.4 Criação de processos: `fork()`

Para criar um novo processo usa-se a função `fork()`. Esta função faz uma chamada ao sistema que cria um novo processo, duplicando o original. O seguinte exemplo simples ilustra a execução do `fork()`:

```
#include <stdio.h>
main() {
    printf ("Início\n" );
    fork();
    printf ("Fim\n" );
}
```

O programa começa por escrever "Início"; depois invoca o `fork()` que cria um novo processo clone do processo original. A partir daí passam a existir dois processos e ambos vão executar o segundo `printf`, escrevendo a mensagem "Fim" no ecrã.

O clone criado pelo `fork()` é, à partida, inteiramente igual ao original: o mesmo programa, com as mesmas variáveis, começando a ser executado a partir do ponto do `fork()`. Mas, obviamente, sendo outro processo, o `pid` é diferente do original. O exemplo seguinte ilustra essa diferença:

```
main() {
    printf ("Início PID=%d\n", getpid() );
    fork();
    printf ("Fim PID=%d\n", getpid() );
}
```

uma das mensagens "Fim" dá um `pid` igual ao original, outra dá um `pid` diferente - sendo esta última, obviamente, a que é escrita pelo novo processo, criado pelo `fork()`.

O novo processo criado pelo `fork()` diz-se um processo filho, sendo o processo pai o original. O exemplo seguinte ilustra esta relação:

```

main() {
    printf ("Início PID=%d filho de %d\n", getpid(), getppid() );
    fork();
    printf ("Fim PID=%d filho de %d\n", getpid(), getppid() );
}

```

A mensagem de fim escrita pelo programa original indica a shell como processo pai; a mensagem de fim escrita pelo novo processo indica o processo original como processo pai.

1.5 Mais sobre o fork(): processo pai e processo filho

Como se disse, o processo pai e processo filho originado por um `fork()` são inicialmente inteiramente semelhantes. Há apenas um pequeno, mas decisivo, pormenor distintivo: o valor que a própria função `fork()` devolve para cada um deles. Para o novo processo, o `fork()` devolve o valor 0; para o processo original, devolve um valor diferente de zero.

Esta diferença permite distinguir o processo original e o processo filho depois do `fork()` e, com base nisso, traçar um caminho diferente para cada um deles. O exemplo seguinte ilustra essa técnica:

```

main() {
    int n;
    printf ("Início\n" );
    n = fork();
    if ( n == 0 )
        printf ("Eu sou o processo filho\n");
    printf ("Fim \n" );
}

```

Consideremos cada um dos dois processos depois do `fork()`. O processo filho segue para a condição do `if`, `n==0`, que dá Verdade; desta forma executa o `printf` que escreve no ecrã a mensagem "Eu sou..."; depois disso escreve a mensagem "Fim" e termina. O processo original segue também para a condição do `if` que, nesse caso, dá Falso; depois escreve a mensagem "Fim" e termina.

1.6 Execução de outros programas: exec

A chamada ao sistema `exec` permite a um processo em curso alterar o programa que está a executar. O seguinte exemplo ilustra o conceito:

```

#include <stdio.h>
main() {
    printf ("Executar outro programa...\n");
    execl( "/bin/ls", "ls", "-l", NULL );
    printf ("Não resultou!\n");
}

```

O programa começa por escrever a mensagem "Executar outro programa...". Depois faz um `exec`, pedindo ao sistema operativo para passar a executar o programa `/bin/ls`. Se a chamada tiver sucesso, aparece no ecrã o resultado de "ls -l".

Atenção que não se trata de mandar executar o outro programa e voltar; trata-se de mudar, definitivamente, de um programa para outro. Neste caso, trata-se de mudar deste programa para o programa `ls`; se a mudança resultar, será executado o `ls` e a mensagem "Não resultou!" já não aparecerá.

A função `execl()` é uma das formas de fazer a chamada `exec` ao sistema operativo. Se fizer

```
man execl
```

poderá ver outras funções para o mesmo efeito, que variam na forma de disposição dos argumentos que configuram a chamada ao sistema.

1.7 Executar outro programa: `fork` e `exec`

A conjugação de `fork` e `exec` permite a um programa mandar executar um outro programa sem se "autodestruir" (como aconteceria de fizesse apenas um `exec`). O seguinte exemplo ilustra a técnica:

```
#include <stdio.h>
#include <unistd.h>
main() {
    int n;
    printf ("Início\n" );
    n = fork();
    if ( n == 0 )    {
        execl ( "/bin/ls", "ls", "-l", NULL );
    }
    printf ("Fim \n" );
}
```

O programa faz o `fork()` e encaminha o processo filho para um `exec`. Desta forma, o processo filho passa a executar um outro programa (no caso, “ls -l”) enquanto o programa original prossegue (neste caso para escrever a mensagem “Fim” e depois terminar).

1.8 `wait`

Consideremos ainda o exemplo anterior. Vamos supor que queremos garantir que a mensagem “Fim”, escrita pelo processo original, apareça no ecrã depois do resultado do `ls` produzido pelo processo filho.

Uma vez criado o novo processo filho, ambos os processos competem no sistema por tempo de processamento. O sistema operativo executa os processos pela ordem e sequência que entende; não podemos, por isso, presumir o que quer que seja sobre qual dos processos é executado primeiro.

Há, entretanto, mecanismos do sistema operativo que nos permitem controlar a sequência dos processos. Um desses mecanismos é a primitiva `wait()`, que nos permite parar o processo original até que o processo filho termine. A utilização do `wait()` para esse efeito é ilustrada no seguinte exemplo:

```
#include <stdio.h>
#include <unistd.h>
main() {
    printf ("Início\n" );
    if ( fork() == 0 )    {
        execl ( "/bin/ls", "ls", "-l", NULL );
        exit(1);
    }
    wait(NULL);
    printf ("Fim \n" );
}
```

O `wait()` faz com que o processo fique bloqueado (parado) até que um processo filho termine. Assim, depois do `fork()`, enquanto o processo filho segue para o `execl` executando “ls -l”, o processo original vai para o `wait()`, onde fica parado até que o processo filho acabe – só depois passa o `wait()` e prossegue escrevendo a mensagem “Fim”.

Devemos também precaver a possibilidade de o `execl` falhar. Neste caso haverá poucas possibilidades, mas em situações mais gerais pode acontecer (se o programa invocado não existir, não tiver permissões de execução, etc.); se o `execl` falhar, o processo filho segue para `exit()` e termina sem mais efeitos.

1.9 Exemplo: uma pequena shell

O exemplo seguinte ilustra uma pequena shell capaz de receber e mandar executar um comando (os comandos têm que ser dados com nome completo, por exemplo `/bin/ls`)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
main() {
    char s[80];
    while ( 1 ) {
        printf ("Comando> ");
        fgets ( s, 80, stdin );
        s[strlen(s)-1] = 0;
        if ( strcmp ( s, "exit" ) == 0 )
            exit(0);
        if ( fork() == 0 ) {
            execl ( s , "", NULL );
            printf ("%s: comando inválido.\n", s );
            exit(1);
        }
        wait(NULL);
    }
}
```

2 Sinais

O mecanismo mais rudimentar de comunicação entre processos é o envio de sinais. Vamos começar por ver este mecanismo através de um exemplo simples, conjugando dois processos:

- Processo 1: executa um programa que publica o seu PID e depois entra em espera;
- Processo 2: manda um sinal ao processo P1;

Os programas são os seguintes:

`p1.c`

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
main() {
    char s[100];
    printf ("PID=%d\n", getpid() );
    while ( 1 ) {
        printf ("> ");
```

```

        scanf("%s", s);
        if ( strcmp(s, "exit") == 0 ) exit(0);
    }
    printf ("Fim do Processo P1");
}

```

envia.c

```

#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
main() {
    int n;
    printf ("Enviar sinal para o processo: ");
    scanf("%d", &n);
    kill ( n, SIGKILL );
}

```

Ponha o programa P1 em execução e anote o respectivo PID; indique esse mesmo PID no programa P2, enviando assim um sinal ao processo P1. De momento, o efeito visível do envio do sinal é a terminação do processo P1.

O SIGKILL é um dos sinais que podem ser enviados de um processo para outro. Além deste há diversos outros sinais, como se pode ver através do comando:

```
man 7 signal
```

Cada sinal é identificado por um número. Por exemplo, o sinal enviado no exemplo anterior tem o número 9.

O ficheiro `signal.h` contém a definição de uma mnemónica para cada um desses números. Por exemplo:

```
#define SIGKILL 9
```

Exercício. experimente o envio do sinal SIGUSR1 em vez de SIGKILL (de momento, o efeito não será muito diferente: na mesma termina o processo de destino).

2.1 Enviar um sinal

O comando `kill` tem o mesmo nome e um efeito semelhante ao da função `kill`: permite o envio de um sinal a um processo. A forma geral para este efeito é:

```
kill -sinal pid
```

Sendo `sinal` o número do sinal a enviar e `pid` o número de processo ao qual o sinal vai ser enviado. Exemplo:

```
kill -9 1000
kill -16 1000
```

O primeiro envia o sinal SIGKILL ao processo 1000; o segundo envia ao mesmo processo o sinal SIGUSR1.

Apesar da função se chamar `kill`, nem sempre o envio de um sinal tem por consequência a "morte" do processo a quem o sinal é enviado. Dentro de certos limites o processo pode "capturar" o sinal que lhe é enviado e controlar a forma de resposta. Basicamente, há duas alternativas:

- Ignorar o sinal;
- Executar determinada função quando o sinal for recebido;

Ambas as acções são indicadas através da função `signal`:

```
signal ( sinal, handler)
```

A função tem dois argumentos: `sinal` e `handler`. No primeiro indica-se o número do sinal que se pretende capturar. No segundo especifica-se a acção a tomar se e quando (algures no tempo) o sinal vier a ser recebido.

Para ignorar o sinal indica-se no argumento `handler` a constante `SIG_IGN`. Exemplo: modifique o programa anterior de forma a ignorar o sinal `SIGUSR1`:

`ignora.c`

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
main() {
    char s[100];
    signal ( SIGUSR1, SIG_IGN ); //ignorar o sinal SIGUSR1
    printf ("PID=%d\n", getpid() );
    while ( 1 ) {
        printf ("> ");
        gets(s);
        if ( !strcmp(s, "exit") ) exit(0);
    }
    printf ("Fim do Processo P1");
}
```

Experimente agora o envio do sinal `SIGUSR1`, através do programa `p2` ou do comando `kill`. O processo deverá ser insensível a este sinal.

2.2 Tratar um sinal

Em vez de ignorar o sinal, podemos indicar uma função para ser executada quando o sinal for recebido. Exemplo:

`trata.c`

```
#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

void f_resposta(int sinal) {
    printf ("Recebi o sinal nº %d\n", sinal);
}

main() {
    char s[100];

    signal (SIGUSR1, f_resposta ); //armar o sinal...
    printf ("PID=%d\n", getpid() );
    while ( 1 ) {
        printf ("> ");
        gets(s);
        if ( !strcmp(s, "exit") ) exit(0);
    }
}
```

```

    }
    printf ("Fim do Processo P1");
}

```

A chamada à função `signal`

```

signal (SIGUSR1, f_resposta );

```

Estabelece que, em resposta ao sinal `SIGUSR1`, deve ser executada a função `f_resposta`.

Experimente pôr o processo em execução e enviar-lhe o sinal `SIGUSR1`. Em resposta deverá aparecer no ecrã a mensagem escrita pela função `f_resposta`.

Note que a chamada `signal` apenas indica a função a executar como resposta ao sinal (dizemos "arma o sinal"). A função indicada só será realmente executada se e quando o processo em causa receber um sinal.

Note ainda que o sinal permanece armado da mesma forma até nova chamada a `signal`. No caso do exemplo anterior, o processo responde com a mensagem indicada em `f_resposta` de cada vez que receber um sinal `SIG_USR1`. Este comportamento só poderia ser alterado com nova chamada a `signal`.

Exemplo: neste caso o processo responde uma única vez ao sinal:

```

f_resposta ( int sinal ) {
    printf ("SINAL Recebido. Desligar...\n");
    signal ( SIGUSR1, SIG_IGN );
}

```

2.3 alarm

A primitiva `alarm` permite a um processo mandar um sinal temporizado a si próprio. Por exemplo, a chamada:

```

alarm ( 5 )

```

faz com o próprio processo receba um sinal `SIGALRM` passado o número de segundos indicado no argumento. Exemplo:

```

#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int n;
void f(int sinal) {
    if ( sinal == SIGALRM ) {
        printf ("Contagem: %d\n", n--);
        if ( n > 0 )
            alarm(1);
    }
}

main() {
    char s[100];
    n = 5;
    signal ( SIGALRM, f );
    alarm (1);
    while ( 1 ) {
        printf ("> ");

```

```

        gets(s);
        if ( !strcmp( s, "exit" ) ) exit(0);
    }
}

```

O mecanismo de alarm é a base para a implementação de temporizações no processo. Por exemplo, para implementar um tempo de espera, pode-se lançar um *alarm* ficando o processo a fazer tempo até receber o respectivo sinal:

```

#include <signal.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int n;
void f(int sinal) {
    if ( sinal == SIGALRM ) n = 1;
}
main() {
    // pausa de 5 segundos...
    printf ("esperar 5 segundos...\n");
    signal ( SIGALRM, f );
    alarm (5);
    while ( n == 0 ) ;
    printf ("ok\n");
}

```

Atenção: não é tempo real.

2.4 Espera ativa e passiva

O problema com a implementação anterior é que o processo enquanto espera, apesar de não estar a fazer nada de útil, está ainda assim a consumir recursos de máquina como um processo normal (e dos piores porque, não tendo i/o nem sequer bloqueia). Diz-se, neste caso que o processo está em *espera activa*: executando normalmente, instruções inúteis, à espera de determinado acontecimento. Em alternativa, o processo pode esperar na situação de bloqueado. Ou seja: lança o alarme e depois pede ao sistema operativo que o bloqueie até receber o sinal. A vantagem é que, enquanto bloqueado, não consome recursos significativos, em particular o SO não o põe a executar.

Para um processo de bloquear chama a função `pause()`. A implementação alternativa pode então ser a seguinte:

```

{
    // pausa de 5 segundos...
    printf ("esperar 5 segundos...\n");
    signal ( SIGALRM, f );
    alarm (5);
    while ( n == 0 ) ;
    printf ("ok\n");
}

```

Este procedimento é semelhante ao que é implementado pela primitiva `sleep(sec)` que põe o processo em espera durante o número de segundos indicado no argumento.

Exercício: use o comando `ps` para ver diferenças entre as duas versões do processo.