Programação Concorrente em Ambientes *UNIX*

Rui Carlos A. Gonçalves

29 de Agosto de 2008

Introdução

Nos últimos tempos os processadores chegaram a um ponto onde se tornou praticamente impossível aumentar a sua frequência, de modo a que a solução foi passar para as arquitecturas *multi-core*. No entanto, nesta abordagem os ganhos não são tão lineares como podem parecer, i.e., duplicar o número de núcleos, não significa duplicar a velocidade das aplicações. Isto porque se a aplicação não for projectada para este tipo de arquitectura, só irá fazer uso de um dos núcleos.

Neste artigo será abordado uma forma de responder a esta situação (não necessariamente a melhor). Assim, serão abordadas algumas *system calls* disponíveis em ambientes *UNIX*, para criação e manutenção de processos, bem como para a comunicação entre eles. Será usada a linguagem de *C*.

Processos e *Threads*

Um processo é aquilo que executa um conjunto de instruções associadas a um programa (programa este que pode ter vários processos a si associados), tendo cada um uma zona de memória independente. Cada processo possui também um identificador único (o *process identification* ou simplesmente *pid*). Por sua vez, cada processo pode ser constituído por várias *threads* (linhas de execução), que partilham as variáveis globais e a *heap*, mas têm uma *stack* independente. Ter vários processos e/ou *threads* em execução em simultâneo é um requisito essencial para que se tire partido de um processador *multi-core*. O facto de partilharem recursos, torna as *threads* mais leves do que os processos, permitindo também uma comunicação mais simples entre elas. Por estes motivos, a utilização de várias *threads* é habitualmente a melhor opção para programação concorrente, no entanto, não será esse o tema abordado neste artigo, centrando-se em vez disso nos processos.

Criação e Manutenção de Processos

As principais *system calls* disponíveis para a criação e manipulação de processos são as seguintes:

- `fork` cria um processo *filho*;
- `wait` espera que um processo termine;
- `getpid` devolve o *pid* de um processo;
- `exit` termina o processo actual.

Para as usar serão necessárias as bibliotecas `unistd.h`, `sys/types.h`, `stdlib.h` e `sys/wait.h`. De seguida analisar-se-á cada uma das *system calls* indicadas (excepto a `exit`, que é do conhecimento geral da maioria dos programadores).

fork

```
pid_t fork(void);
```

Esta *system call* cria um processo filho (em relação ao que a invoca). Todo o código que estiver depois da sua invocação passará a ser executado por duas *entidades*. É claro que na maior parte dos casos o que se pretende é que cada um dos processos execute uma determinada parte do código, não tendo grande utilidade a execução do código em duplicado. Mas como é que se indica o é que deve ser feito por um processo e pelo outro?

A resposta está no valor devolvido pelo `fork`. Este devolve para o processo que o invocou (o processo *pai*) o valor correspondente ao *pid* do novo processo[1] (o processo *filho*), enquanto que ao novo processo devolve o valor 0. Assim, verificando este valor, pode-se controlar aquilo que será executado pelo processo pai e o que é para o processo filho (isto é valido para os casos mais simples, quando se cria mais do que um processo filho as coisas podem não ser assim tão fáceis). De seguida apresenta-se um pequeno exemplo.

```
1  int main() {
2    pid_t pid=fork();
3
4    if(pid==0) puts("Processo filho");
5    else puts("Processo pai");
6
7    puts("Fim");
8
9    return 0;
10 }
```

Ao executar este código a mensagem "Fim" será mostrada duas vezes, enquanto que as outras duas só são mostradas uma vez cada. Experimentando executar o programa várias vezes, certamente será possível ver que a ordem pela qual as mensagens são mostradas varia. Este é um comportamento típico das aplicações concorrentes.

wait

```
pid_t wait(int *status);
```

Esta *system call* permite esperar pela finalização de um processo filho. Quando isso acontecer, é colocada em `status` informação relativa à forma como o processo terminou[2]. Devolve o *pid* do processo que terminou, a menos que ocorra um erro, devolvendo -1 nesse caso.

De seguida mostra-se um exemplo de utilização desta *system call*.

```
1  int main() {
2    pid_t pid=fork();
3
4    if(pid==0) {
```

[1] Também pode ser devolvido o valor -1 caso ocorra algum erro.

[2] Existem várias macros já definidas que permitem extrair a informação do `status` (que é apenas um inteiro); para mais informação consultar as *man pages* do `wait`.

```
5       puts("Processo filho arrancou...");
6       sleep(10);
7       puts("Processo filho terminou.");
8     } else {
9       puts("Processo pai");
10      if(pid!=wait(NULL)) puts("Erro!!!");
11      puts("Fim");
12    }
13
14    return 0;
15  }
```

Certamente a mensagem "Fim" só irá aparecer depois da mensagem "Processo filho terminou.". Se se retirar a linha correspondente à função `wait`, o mais provável é que o programa encerre antes da mensagem "Processo filho terminou." ser mostrada.

De notar que, caso fossem criados vários processos filho, esta função só espera por um deles (para vários será necessário executar várias vezes a função, como seria de esperar). Mais do que isso, esta função não permite controlar qual o processo filho pelo qual se vai esperar. Para isso existe uma variante desta *system call*, a `waitpid` (`pid_t waitpid(pid_t wpid, int *status, int options);`). Esta já permite a indicação de qual processo se está à espera (através do parâmetro `wpid`), assim como algumas opções adicionais que não são relevantes para este texto.

Mais um pequeno exemplo de utilização:

```
1   int main() {
2     pid_t pid1, pid2;
3     pid1=fork();
4
5     if(pid1==0) {
6       puts("Processo filho 1 arrancou...");
7       sleep(4);
8       puts("Processo filho 1 terminou.");
9     } else {
10      pid2=fork();
11
12      if(pid2==0){
13        puts("Processo filho 2 arrancou...");
14        sleep(1);
15        puts("Processo filho 2 terminou.");
16      } else {
17        puts("Processo pai");
18        waitpid(pid1, NULL, 0);
19        puts("Fim");
20      }
21    }
22
23    return 0;
24  }
```

Tal como está, só com o fim do primeiro processo filho a ser criado é que aparece a mensagem "Fim". Se se tivesse usado a função `wait` (ou colocado o valor -1 no primeiro parâmetro da função `waitpid`), mal um dos processos filho terminasse, o pai também terminaria.

getpid

```
pid_t getpid(void);
```

Esta *system call* permite saber o *pid* do processo que está a correr. Existe também uma variante que devolve o *pid* do processo que lançou o processo actual (ou seja, o *pid* do processo pai): `pid_t getppid(void);`.

Um exemplo da sua utilização:

```
1  int main() {
2    pid_t pid;
3    pid=fork();
4
5    if(pid==0) {
6      printf("Processo filho - pid: %d / pai: %d\n", getpid(), getppid());
7    } else {
8      printf("Processo pai - pid: %d / filho: %d\n", getpid(), pid);
9      wait(NULL);
10   }
11
12   return 0;
13 }
```

Ambos os programas deverão mostrar os mesmos valores, apenas com a ordem trocada. Agora pode-se experimentar remover a chamada ao `wait` (linha 9), e executar o programa várias vezes. É provável que em algumas delas, a função `getppid` devolva o valor 1. Isto aconteceria no caso da chamada à função ocorrer depois do processo pai já ter terminado, e nestas situações, o processo pai passa a ser o processo 1.

Por fim apresenta-se um exemplo onde se mostra uma possível utilização das funções aqui abordadas. Pretende-se procurar um determinado valor num vector de inteiros, e caso este exista, determinar a posição no vector em que esse mesmo valor se encontra. O vector terá 200 posições, e será inicializado com uma sequência de inteiros de 1 a 200. Vão criar-se 4 processos para efectuar a procura, e devolver a posição onde o elemento está para o processo principal. O valor a procurar será o 90.

```
1  int main() {
2    pid_t pid;
3    int vals[200];
4    int i, j, n=90, status;
5
6    for(i=0; i<200; i++)
7      vals[i]=i+1;
8
9    for(i=0; i<4; i++) {
10     pid=fork();
11     if(pid==0) {
12       for(j=50*i; j<50*(i+1); j++)
13         if(vals[j]==n) exit(j);
14
15       exit(255);
16     }
17   }
```

```
18
19    for(i=0; i<4; i++) {
20      wait(&status);
21
22      if(WEXITSTATUS(status)!=255)
23        printf("O valor esta na posicao: %d\n", WEXITSTATUS(status));
24    }
25
26    return 0;
27  }
```

Aqui já não se usa apenas o resultado do `fork` para saber qual é o processo que está a correr. Recorre-se também ao valor que a variável *i* possui num determinado momento. De notar também que os processos filho são encerrados ainda dentro do *if*, algo que é bastante frequente neste tipo de problemas, pois permite um maior controlo sob aquilo que é executado por cada processo.

Neste exemplo foi usada a macro `WEXITSTATUS`, que extrai o valor devolvido por um processo, a partir do valor dado pela função `wait`.

Este método possui algumas limitações. Isto porque o valor devolvido pelo `exit` não pode ser superior a 255. Assim, cada processo poderia, no máximo, pesquisar 255 posições. No entanto, existem formas de comunicação entre processos mais funcionais, uma das quais será abordada na próxima secção.

Comunicação Entre Processos com *Pipes*

As *pipes* são uma forma de comunicação entre processos bastante usada em *UNIX*, que muitos utilizadores estão habituados a ver numa *shell*. Agora mostrar-se-á como usar este mecanismo em conjunto com as restantes *system calls* aqui apresentadas.

Uma *pipe* funciona como um ficheiro. Na sua criação, obtêm-se dois descritores, um onde se podem escrever dados e outro onde se podem ler dados. No entanto, um processo só pode ler, e o outro só pode escrever, i.e., a informação só circula num sentido. Para a criar, usa-se a função `int pipe(int filedes[2]);`. Em `filedes[0]` é colocado o descritor aberto em modo de leitura, e em `filedes[1]` o descritor aberto em modo de escrita (em cada processo, só se usa um deles, como foi abordado atrás, razão pela qual se deve fechar o descritor que não vai ser usado).

Vai então resolver-se um problema semelhante ao anteriormente apresentado, mas agora com um vector de 10000 posições, com valores de 0 a 999 (repetidos várias vezes), procurando pelo valor 0.

```
1   int main() {
2     pid_t pid;
3     int vals[10000];
4     int i, j, n=0;
5
6     int filho_pai[2];
7
8     if(pipe(filho_pai)==-1) exit(1);
9
10    for(i=0; i<10000; i++)
11      vals[i]=i%1000;
12
13    for(i=0; i<4; i++) {
14      pid=fork();
```

```
15      if(pid==0) {
16        close(filho_pai[0]);
17        for(j=2500*i; j<2500*(i+1); j++)
18          if(vals[j]==n) write(filho_pai[1], &j, sizeof(int));
19
20        close(filho_pai[1]);
21        exit(0);
22      }
23    }
24
25    close(filho_pai[1]);
26
27    for(i=0; i<4; i++)
28      wait(NULL);
29
30    while(read(filho_pai[0], &j, sizeof(int)))
31      printf("O valor esta na posicao: %d\n", j);
32
33    close(filho_pai[0]);
34    return 0;
35  }
```

O método é semelhante ao usado anteriormente. A única diferença é mesmo a forma como os valores são devolvidos ao processo pai. Como se pode ver, as *pipes* já não têm limitações com o tamanho da informação, podendo também ser utilizadas por vários processos em simultâneo. Neste exemplo, foi usada para enviar informações dos processos filho, para o processo pai, mas também podem ser usadas em sentido inverso (é claro que no caso de haver vários processos a ler numa mesma *pipe*, é necessário algum cuidado).

Conclusão

Ao longo deste texto, foram abordados alguns dos mecanismos básicos disponíveis em ambientes *UNIX*, que permitem construir aplicações concorrentes, tendo sido também apresentados alguns exemplos (simples) de aplicação. Estes mecanismos possuem ainda outra grande utilidade, a execução de aplicações externas, assunto que, no entanto, sai fora do tema deste texto. Sugere-se o estudo das *system calls* da família `exec` para que estiver interessado no assunto. Informações mais aprofundadas sobres as funções aqui tratadas, podem ser facilmente encontradas nas respectivas *man pages*.

Referências

[1] J. P. Oliveira. *Apontamentos de Sistemas Operativos I.* Universidade do Minho, 2005.

[2] W. R. Stevens, S. A. Rago. *Advanced Programming in the UNIX Environment.* 2ª edição, 2005.