

Mecanismo *select* – Resumo, API e exemplo

Este documento resume o mecanismo *select* abordado nas aulas e apresenta um exemplo de utilização.

Conteúdo

1. Mecanismo *select* (explicação e cenário de utilização)
2. Forma de uso
3. API e Estruturas de dados envolvidas
4. Exemplo com código

1. Mecanismo *select*

O mecanismo *select* permite aguardar por mais do que uma operação de entrada/saída sem ser necessário saber qual delas se completa em primeiro lugar. Este mecanismo possibilita, por exemplo, aguardar informação em mais do que um *pipe*, ou num *pipe* e num teclado, sem usar sinais para sinalização nem *threads*.

Cenários de utilização.

O mecanismo é útil quando um programa está a usar operações bloqueantes e deseja efectuar mais do que uma. Como as operações são bloqueantes, corre-se o risco de aguardar por aquela que ocorre em último lugar, perdendo-se tempo e eventualmente deixando de atender um acontecimento (ex.: chegada de informação) por se estar bloqueado à espera da “outra” operação.

Exemplo de aplicação:

Considere-se um programa que necessita de dar atenção simultaneamente a um *pipe* (leitura) e ao teclado. Usam-se funções bloqueantes e é um requisito responder imediatamente à chegada de informação. Só existem as seguintes soluções em Unix:

- **Recurso a sinais:** o programa associa o atendimento de um sinal à rotina de leitura do *pipe* e fica a aguardar na leitura do teclado. O outro programa responsável por enviar informação a este pelo *pipe* terá também que lhe enviar um sinal para o “avisar” e causar a execução da sua rotina de leitura do *pipe*.
 - Pouco vantajosa: envolve mais um mecanismo (sinais). É pouco versátil e só funciona para situações onde há um programa do outro lado capaz de enviar sinais (no caso do teclado, não é útil).
- **Utilização de *threads*:** o programa lança uma *thread* cujo código só lê o *pipe*, e outra *thread* só lê o *pipe*.
 - É simples, aplicável a muitas situações, e costuma ser a melhor solução. Neste momento ainda não se falou de *threads* e irá ser dado um exemplo mais tarde com este cenário. Por esta razão não se vai considerar esta hipótese neste documento.
- **Utilização de operações não bloqueantes:** os *read/write* deixam de bloquear e o programa nunca fica bloqueado “à espera na outra fonte de dados”.
 - Resolve o problema, mas levanta outros piores. As operações assíncronas exigem ao programador algoritmos mais complicados, e exigem ao programa o teste contínuo acerca da disponibilidade de dados/conclusão da operação (*polling*), ocupando o processador e prejudicando a performance geral da máquina.
- **Mecanismo *select*:** o programador indica quais as fontes de dados/mecanismo de comunicação que deseja usar e o sistema aguarda em todas, detectando qual a primeira em que a operação pretendida está pronta. O programa apenas espera numa coisa.
 - É relativamente simples de usar e apenas a solução com *threads* será melhor (por ser mais genérica).
 - Esta é a solução tratada e exemplificada neste documento.

2. *select* - Forma de uso:

O programa encontra-se eventualmente (mas não obrigatoriamente) num ciclo onde desencadeia operações E/S. Podem ser de leitura, de escrita, ou de atendimento de acontecimentos de excepção. Dentro do ciclo (se houver) vai fazer:

- Prepara o conjunto **fd_set** que vai conter os descritores de ficheiros correspondentes aos dispositivos/mecanismos de comunicação onde deseja efectuar operações.
 - Coloca o conteúdo a zero – **FD_ZERO**.
 - Acrescenta os descritores – **FD_SET**.
 - Indica qual o descritor numericamente mais elevado +1.
 - Irá existir um conjunto para descritores para operações de leitura, outro para os descritores em operações de escrita, e outro para as notificações de excepção.
- Coloca o *timeout* (prazo máximo a aguardar no *select*) numa estrutura *timeval*.
- O programa invoca a função **select** indicando o ponteiro para cada um dos descritores e a estrutura **timeval** com o *timeout*.
- A função *select* bloqueia e devolve apenas quando uma das operações pretendidas está em condições de ser concluída de imediato.
- Após a função *select* retornar, o programa deve averiguar com **FD_ISSET** cada um dos descritores para saber qual deles é que está pronto a usar.
- A função *select* modifica a estrutura *fd_set* e *timeval*. Caso se esteja em clico, é necessário colocar a informação de início nestas estruturas.

3. API e Estruturas de dados envolvidas

- Set de flags (prontidão) de descritores de ficheiros

fd_set

- Aguardar dados disponíveis/possibilidade de escrita através dos descritores indicados

```
int select(int nfd,  
          fd_set *readfds, fd_set *writefds, fd_set *exceptfds,  
          struct timeval *timeout);
```


- **Remover descritor ao conjunto de flags a observar**

```
void FD_CLR(int fd, fd_set *set);
```

- **Verificar disponibilidade de dados/ possibilidade de escrita num descritor**

```
int FD_ISSET(int fd, fd_set *set);
```

- **Adicionar descritor ao conjunto de flags a observar**

```
void FD_SET(int fd, fd_set *set);
```

- **Limpar conjunto de flags de descritores**

```
void FD_ZERO(fd_set *set);
```

4. Código exemplo:

O exemplo consiste num programa que consegue ler de dois pipes e do teclado em simultâneo.

Função principal

```
int main(int argc, char* argv[]) {
    int fd_a, fd_b, fd_c;    // file descriptor dos pipes
    int nfd;                 // valor retorno select()
    fd_set read_fds;         // conjunto das flags para desc. ficheiros
    struct timeval tv;       // timeout para select

    signal(SIGINT, trataCC); // para interromper via ^C

    // cria os pipes
    mkfifo("pipe_a", 00777);
    mkfifo("pipe_b", 00777);

    // abre os pipes. RDRW vs RD – notar isto
    fd_a = open("pipe_a", O_RDWR | O_NONBLOCK);
```

```

if (fd_a == -1)
    sayThisAndExit("Erro no open pipe_a");
fd_b = open("pipe_b", O_RDWR | O_NONBLOCK);
if (fd_b == -1)
    sayThisAndExit("Erro no open pipe_b");

while (1) {

    tv.tv_sec = 10;    // segundos    ( 10 = apenas um exemplo)
    tv.tv_usec = 0;    // micro-segundos (se ambos a 0 então faz polling)

    FD_ZERO(& read_fds);        // inicializa conjunto de fd (watch list)
    FD_SET(0, & read_fds);      // adiciona stdin ao conj de fd a observar
    FD_SET(fd_a, & read_fds);    // adiciona pipe_a ao conj de fd a observ.
    FD_SET(fd_b, & read_fds);    // adiciona pipe_b ao conj de fd a observ.

    // vê se há dados em alguns dos fd (stdin, pipes) - modifica os sets
    // bloqueia ate: sinal, timeout, há dados para ler EOF, exception
    nfd = select(                // bloqueia até haver dados ou EOF no read-set
        max(fd_a, fd_b)+1,      // max valor dos vários fd + 1
        & read_fds,              // read fd set
        NULL,                   // write fd set - (nenhum aqui)
        NULL,                   // exception fd set - (nenhum aqui)
        & tv);                  // timeout - se ambos = 0-> retorna logo (p/ polling)
    // actualiza tv -> quanto tempo faltava para o timeout

    if (nfd == 0) {
        printf("\n(Estou a espera....)\n"); fflush(stdout);
        continue;
    }

    if (nfd == -1) {
        perror("\nerro no select");
        close(fd_a); close(fd_b);
        unlink("pipe_a"); unlink("pipe_b");
        return EXIT_FAILURE;
    }

    if (FD_ISSET(0, & read_fds)) {    // stdin tem algo para ler?
        trataTeclado();
        // sem "continue" -> não vai logo para a próxima iteração
    } // porque pode ser que pipe_a ou pipe_b também tenha algo

```

```

    if (FD_ISSET(fd_a, & read_fds)) {    // fd_a tem algo para ler?
        leEMostraPipes("A", fd_a);      // função auxiliar para ler pipes
        // sem "continue" -> não vai logo para a próxima iteração
    } // porque pode ser que pipe_b também tenha algo

    if (FD_ISSET(fd_b, & read_fds)) {    // fd_b tem algo para ler?
        leEMostraPipes("B", fd_b);
        // o código dentro do ciclo acaba já a seguir e volta a esperar por dados
    }
}

// em princípio, neste exemplo, não deve chegar aqui
return EXIT_SUCCESS;
}

```

Funções seguintes:

São de natureza auxiliar. O código de tratamento de leitura de caracteres é mais complexo que o necessário e deixa-se a sua simplificação para trabalho em casa

Atendimento da chegada de informação via pipe(s)

```

void leEMostraPipes(char * quem, int fd) {
    char buffer[200];
    int bytes;
    bytes = read(fd, buffer, sizeof(buffer));
    buffer[bytes] = '\0';
    if ( (bytes > 0) && (buffer[strlen(buffer)-1] == '\n') )
        buffer[strlen(buffer)-1] = '\0';
    printf("\n%s: (%d bytes) [%s]\n", quem, bytes, buffer);
    if (strcmp(buffer, "sair")==0) {
        unlink("pipe_a"); unlink("pipe_b");
        exit(EXIT_SUCCESS);
    }
}
}

```

Atendimento da chegada de informação via stdin

```
void trataTeclado() {
    char buffer[200];
    int bytes;
    fgets(buffer, sizeof(buffer), stdin); // scanf("%s",buffer);
    if ( (strlen(buffer)>0) && (buffer[strlen(buffer)-1]=='\n') )
        buffer[strlen(buffer)-1] = '\0';
    printf("\nKBD: [%s]\n", buffer);
    fflush(stdout);
    if (strcmp(buffer,"sair")==0) {
        unlink("pipe_a"); unlink("pipe_b");
        exit(EXIT_SUCCESS);
    }
}
```

Funções secundárias: max, trataCC, sayThisAndExit

Terminação em caso de erro

```
void sayThisAndExit(char * p) {
    perror(p);
    exit(EXIT_FAILURE);
}
```

Obtenção do máximo de dois inteiros

```
int max(int a, int b) {
    return (a>b) ? a: b;
}
```

Atendimento de SIGINT (recepção de ^C)

```
void trataCC(int s) {
    unlink("pipe_a"); unlink("pipe_b");
    printf("\n ->CC<- \n\n");
    exit(EXIT_SUCCESS);
}
```


Ficheiros *header* usados

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

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