1.Problemas clássicos de comunicação entre processos

1.1 O problema da janta dos filósofos

Em 1965, Dijkstra propôs e resolveu um problema de sincronização que chamou de ***problema da janta dos filósofos***. O problema pode ser exposto de uma maneira simples. Cinco filósofos estão sentados ao redor de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete é tão escorregadio que o filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos há um garfo. A vida de um filósofo consiste em alternar períodos de se alimentar e de pensar. Quando um filósofo sente fome, ele tenta pegar os garfos da esquerda, ele come por algum tempo e, então, coloca os garfos na mesa e continua a pensar. A pergunta fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e nunca entre em *deadlock*.

solução errada para o problema

```
#define N 5                     /* número de filósofos */

void philosopher(int i)         /* i: número do filósofo, de 0 a 4 */
while(TRUE){
  think();                      /* o filósofo está pensando */
  take_fork(i);                         /* pega o garfo da esquerda */
  take_fork((i+1) % N);         /* pega o garfo da direita; % é o operador de módulo */
  eat();                                /* come o espaguete */
  put_fork(i);                  /* coloca o garfo da esquerda de volta na mesa */
  put_fork((i+1) % N);          /* coloca o garfo da direita de volta na mesa */
}
}
```

O procedimento *take_fork* espera até que o garfo especificado esteja disponível e, então, apodera-se dele. Suponha que os cinco filósofos peguem os garfos da esquerda simultaneamente. Nenhum será capaz de pegar os garfos da direita e haverá um impasse.

Uma situação como essa, na qual todos os programas continuam a executar indefinitivamente, mas não conseguem fazer progresso algum é chamado de *starvation*.

A observação de que os filósofos devem esperar um tempo aleatório é válida. Entretanto, devido aos atrasos na transmissão, dois podem enviar dados simultaneamente sobrepondo-os, nesse caso, uma colisão. Quando é detectada uma colisão, cada um espera por um tempo aleatório e tenta novamente, que na prática, funciona bem. Porém, em alguns aplicativos, é necessário que isso funcione sempre como o controle de uma usina nuclear.

```
#define N 5                     /* número de filósofos */
#define LEFT (i+N-1)%N          /* o número do vizinho à esquerda de i */
#define RIGHT (i+1)%N           /* número do vizinho à direita de i */
#define THINKING 0                      /* o filósofo está esperando */
#define HUNGRY 1                /* o filósofo está tentando pegar garfos */
#define EATING 2                /* o filósofo está comendo */
```

```
typedef int semaphore;      /* os semáforos são um tipo especial de int */
int state[N];               /* array para controlar o estado de todos */
semaphore mutex = 1;              /* exclusão mútua para regiões críticas */
semaphore s[N];             /* um semáforo por filósofo */

void philosopher(int i){    /* i: número do filósofo, de 0 a N -1 */
while(TRUE){                      /* repete eternamente */
  think();                  /* o filósofo está pensando */
  take_forks(i);            /* pega dois garfos ou bloqueia */
  eat();                          /* come o espaguete */
  put_forks(i);                   /* coloca os dois garfos de volta na mesa */
}
}

void take_forks(int i){           /* i: número do filósofo, de 0 a N - 1 */
  down(&mutex);             /* entra na região crítica */
  state[i] = HUNGRY               /* registra o fato de que o filósofo i está com fome */
  test(i);                  /* tenta pegar 2 garfos */
  up(&mutex);                     /* sai da região crítica */
  down(&s[i]);              /* bloqueia se os garfos não foram pegos */
}

void put_forks(i){          /* i: número do filósofo, de 0 a N - 1 */
  down(&mutex);             /* entra na região crítica */
  state[i] = THINKING;      /* o filósofo acabou de comer */
  test(LEFT);               /* verifica se o vizinho da esquerda pode comer agora */
  test(RIGHT);              /* verifica se o vizinho da direita pode comer agora */
  up(&mutex);                     /* sai da região crítica */
}

void test(i){               /* i: número do filósofo, de 0 a N - 1 */
  if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != eating){
    state[i] = EATING;
    up(&s[i];
  }
}
```

A solução acima não apresenta impasse e permite o máximo de paralelismo para um número arbitrário de filósofos. Ela utiliza um *array*, *state*, para controlar se um filósofo está comendo, pensando ou se está com fome. Um filósofo só pode passar para o estado "comendo" se nenhum vizinho estiver comendo. Os vizinhos do filósofo *i* são definidos pelas macros *LEFT* e *RIGHT*. Em outras palavras, se *i* é 2, *LEFT* é 1 e *RIGHT* é 3.

O programa utiliza um *array* de semáforos, um por filósofo, de modo que os filósofos que estão com fome podem ser bloqueados, caso os garfos necessários estejam ocupados.

Note que cada processo executa a função *philosopher* como seu código principal, mas *take_fork*, *put_forks* e *test*, são as funções comuns e não processos separados.

2.O problema dos leitores e escritores

Outro problema de E/S é dos leitores e escritores que modela um acesso a um banco de dados. Imagine um sistema de reservas de uma companhia aérea, com muitos processos querendo ler e escrever. É aceitável ter vários processos lendo o banco de dados ao mesmo tempo, mas se um processo estiver atualizando, o banco de dados, nenhum outro processo poderá acessar esse banco, nem mesmo o leitor.

A solução para esse problema, o primeiro leitor a obter acesso ao banco de dados executa uma operação *down* no semáforo *db*. Os leitores subsequentes precisam apenas incrementar um contador *rc*. À medida que os leitores saem, eles decrementam o contador e o último deles executa uma operação *up* no semáforo, permitindo a entrada de um escritor bloqueado, caso haja um. Como ter dois leitores ao mesmo tempo não é problema, o segundo leitor é admitido. Um terceiro leitor e os leitores subsequentes também podem ser admitidos, caso apareçam.

Agora, suponha que apareça um escritor. O escritor não pode ser admitido no banco de dados, pois os escritores devem ter acesso exclusivo, de modo que ele é bloqueado. Posteriormente, aparecem outros leitores. O escritor será mantido bloqueado até que nenhum leitor esteja presente. Se um novo leitor chegar, o escritor nunca executará.

Para evitar essa situação, o programa pode ser escrito de maneira ligeiramente diferente: quando um leitor chegar e um escritor está esperando, o leitor é bloqueado atrás do escritor em vez de ser admitido imediatamente. Dessa maneira, um escritor precisa esperar o término dos leitores que estavam ativos quando ele chegou, mas não precisa esperar os leitores que apareceram depois dele. A desvantagem dessa solução é que ela gera menos concorrência e, portanto, tem desempenho inferior.