Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Sistemas de Computação

SSC 140 - SISTEMAS OPERACIONAIS I
Turmas A e B

Aula 7 –Comunicação e Sincronização de Processos

Profa. Sarita Mazzini Bruschi

Slides de autoria de
Luciana A. F. Martimiano baseados no livro
Sistemas Operacionais Modernos de A. Tanenbaum

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - **Monitores**;
  - Passagem de Mensagem;

Comunicação de Processos – Monitores

- Idealizado por Hoare (1974) e Brinch Hansen (1975)
- **Monitor**: primitiva (unidade básica de sincronização) de alto nível para sincronizar processos:
  - Conjunto de procedimentos, variáveis e estruturas de dados agrupados em um único módulo ou pacote;
- Somente um processo pode estar ativo dentro do monitor em um mesmo instante; outros processos ficam bloqueados até que possam estar ativos no monitor;

Comunicação de Processos – Monitores

```
monitor example
  int i;
  condition c;

  procedure A();
  .
  end;
  procedure B();
  .
  end;
end monitor;
```

Estrutura básica de um Monitor

Dependem da linguagem de programação → Compilador é que garante a exclusão mútua.
- JAVA

Todos os recursos compartilhados entre processos devem estar implementados **dentro** do **Monitor**;

Comunicação de Processos – Monitores

- Execução:
  - Chamada a uma rotina do monitor;
  - Instruções iniciais → teste para detectar se um outro processo está ativo dentro do monitor;
  - Se positivo, o processo novo ficará bloqueado até que o outro processo deixe o monitor;
  - Caso contrário, o processo novo executa as rotinas no **monitor**;

Comunicação de Processos – Monitores

- *Condition Variables* (`condition`): variáveis que indicam uma condição; e
- Operações Básicas: *WAIT e SIGNAL*
  - `wait (condition)` → bloqueia o processo;
  - `signal (condition)` → "acorda" o processo que executou um `wait` na variável `condition` e foi bloqueado;

Comunicação de Processos – Monitores

- Variáveis condicionais não são contadores, portanto, não acumulam sinais;
- Se um sinal é enviado sem ninguém (processo) estar esperando, o sinal é perdido;
- Assim, um comando `WAIT` deve vir antes de um comando `SIGNAL`.

Comunicação de Processos – Monitores

- Como evitar dois processos ativos no monitor ao mesmo tempo?

  (1) Hoare → colocar o processo mais recente para rodar, suspendendo o outro!!! (*sinalizar e esperar*)

  (2) B. Hansen → um processo que executa um `SIGNAL` deve deixar o monitor imediatamente;
  - O comando `SIGNAL` deve ser o último de um procedimento do monitor;

A condição (2) é mais simples e mais fácil de se implementar.

Comunicação de Processos – Monitores

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```


Comunicação de Processos – Monitores

- A exclusão mútua automática dos procedimentos do monitor garante que, por exemplo, se o produtor dentro de um procedimento do monitor descobrir que o buffer está cheio, esse produtor será capaz de terminar a operação de `WAIT` sem se preocupar, pois o consumidor não estará ativo dentro do monitor até que `WAIT` tenha terminado e o produtor tenha sido marcado como não mais executável;

Comunicação de Processos – Monitores

- Limitações de semáforos e monitores:
  - Ambos são boas soluções somente para CPUs com memória compartilhada. Não são boas soluções para sistema distribuídos;
  - Nenhuma das soluções provê troca de informações entre processo que estão em diferentes máquinas;
  - Monitores dependem de uma linguagem de programação – poucas linguagens suportam Monitores;

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - Monitores;
  - **Passagem de Mensagem**;

Comunicação de Processos – Passagem de Mensagem

- Provê troca de mensagens entre processos rodando em máquinas diferentes;
- Utiliza-se de duas primitivas de chamadas de sistema: *send* e *receive;*

13

Comunicação de Processos – Passagem de Mensagem

- Podem ser implementadas como procedimentos:
  - *send (destination,&message);*
  - *receive (source,&message);*
- O procedimento `send` envia para um determinado destino uma mensagem, enquanto que o procedimento `receive` recebe essa mensagem em uma determinada fonte; Se nenhuma mensagem está disponível, o procedimento `receive` é bloqueado até que uma mensagem chegue.

14

Comunicação de Processos – Passagem de Mensagem

- Problemas desta solução:
  - Mensagens são enviadas para/por máquinas conectadas em rede; assim mensagens podem se perder ao longo da transmissão;
  - Mensagem especial chamada ***acknowledgement*** → o procedimento `receive` envia um ***acknowledgement*** para o procedimento send. Se esse ***acknowledgement*** não chega no procedimento send, esse procedimento retransmite a mensagem já enviada;

15

Comunicação de Processos – Passagem de Mensagem

- Problemas:
  - A mensagem é recebida corretamente, mas o ***acknowledgement*** se perde.
  - Então o `receive` deve ter uma maneira de saber se uma mensagem recebida é uma retransmissão → cada mensagem enviada pelo send possui uma identificação – seqüência de números; Assim, ao receber uma nova mensagem, o `receive` verifica essa identificação, se ela for semelhante a de alguma mensagem já recebida, o `receive` descarta a mensagem!

16

Comunicação de Processos – Passagem de Mensagem

- Problemas:
  - Desempenho: copiar mensagens de um processo para o outro é mais lento do que operações com semáforos e monitores;
  - Autenticação → Segurança;

17

Comunicação de Processos – Passagem de Mensagem

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

18

Comunicação de Processos
Outros mecanismos

- RPC – *Remote Procedure Call*
  - Rotinas que permitem comunicação de processos em diferentes máquinas;
  - Chamadas remotas;
- MPI – *Message-passing Interface*;
  - Sistemas paralelos;
- RMI Java – *Remote Method Invocation*
  - Permite que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java, independentemente da localização dessas máquinas virtuais;
- Web Services
  - Permite que serviços sejam compartilhados através da Web

19

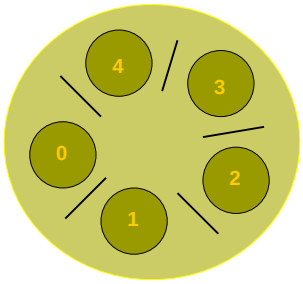
Comunicação de Processos
Outros mecanismos

- *Pipe*:
  - Permite a criação de filas de processos;
  - **`ps -ef | grep luciana`**;
  - Saída de um processo é a entrada de outro;
  - Existe enquanto o processo existir;
- *Named pipe*:
  - Extensão de pipe;
  - Continua existindo mesmo depois que o processo terminar;
  - Criado com chamadas de sistemas;
- *Socket*:
  - Par endereço IP e porta utilizado para comunicação entre processos em máquinas diferentes;
  - Host X (192.168.1.1:1065) Server Y (192.168.1.2:80);

20

Problemas clássicos de comunicação entre processos

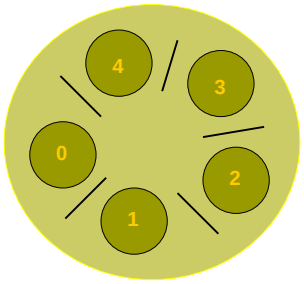
- Problema do Jantar dos Filósofos
  - Cinco filósofos desejam comer espaguete; No entanto, para poder comer, cada filósofo precisa utilizar dois garfo e não apenas um. Portanto, os filósofos precisam compartilhar o uso do garfo de forma sincronizada.
  - Os filósofos comem e pensam;



21

Problemas clássicos de comunicação entre processos

- Problemas que devem ser evitados:
  - *Deadlock* – todos os filósofos pegam **um garfo** ao mesmo tempo;
  - *Starvation* – os filósofos **fiquem indefinidamente pegando garfos simultaneamente**;



22

Solução 1 para Filósofos (1/2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

23

Solução 1 para Filósofos (2/2)

- Problemas da solução 1:
  - Execução do `take_fork(i)` → Se todos os filósofos pegarem o garfo da esquerda, nenhum pega o da direita → *Deadlock*;
- Se modificar a solução (mudança 1):
  - Verificar antes se o garfo da direita está disponível. Se não está, devolve o da esquerda e começa novamente → *Starvation (Inanição);*
  - Tempo fixo ou tempo aleatório (rede Ethernet);
    - Serve para sistemas não-críticos;

24

Solução 1 para Filósofos (2/2)

- Se modificar a solução (mudança 2):

```
#define N 5                                   /* number of philosophers */
semaphore mutex = 1;
void philosopher(int i)                       /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                             /* philosopher is thinking */
        down(&mutex);
        take_fork(i);                         /* take left fork */
        take_fork((i+1) % N);                 /* take right fork; % is modulo operator */
        eat( );                               /* yum-yum, spaghetti */
        put_fork(i);                          /* put left fork back on the table */
        put_fork((i+1) % N);                  /* put right fork back on the table */
    }
}
        up(&mutex);
```

Somente um filósofo come!

Solução 2 para Filósofos usando Semáforos (1/3)

- Não apresenta:
  - *Deadlocks*;
  - *Starvation*;
- Permite o máximo de "paralelismo";

26

Solução 2 para Filósofos usando Semáforos (2/3)

```
#define N                5                /* number of philosophers */
#define LEFT             (i+N−1)%N        /* number of i's left neighbor */
#define RIGHT            (i+1)%N          /* number of i's right neighbor */
#define THINKING         0                /* philosopher is thinking */
#define HUNGRY           1                /* philosopher is trying to get forks */
#define EATING           2                /* philosopher is eating */
typedef int semaphore;                    /* semaphores are a special kind of int */
int state[N];                             /* array to keep track of everyone's state */
semaphore mutex = 1;                      /* mutual exclusion for critical regions */
semaphore s[N];                           /* one semaphore per philosopher */

void philosopher(int i)                   /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                        /* repeat forever */
        think( );                         /* philosopher is thinking */
        take_forks(i);                    /* acquire two forks or block */
        eat( );                           /* yum-yum, spaghetti */
        put_forks(i);                     /* put both forks back on table */
    }
}
```

Solução 2 para Filósofos usando Semáforos (3/3)

```
void take_forks(int i)                    /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);                         /* enter critical region */
    state[i] = HUNGRY;                    /* record fact that philosopher i is hungry */
    test(i);                              /* try to acquire 2 forks */
    up(&mutex);                           /* exit critical region */
    down(&s[i]);                          /* block if forks were not acquired */
}

void put_forks(i)                         /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);                         /* enter critical region */
    state[i] = THINKING;                  /* philosopher has finished eating */
    test(LEFT);                           /* see if left neighbor can now eat */
    test(RIGHT);                          /* see if right neighbor can now eat */
    up(&mutex);                           /* exit critical region */
}

void test(i)                              /* i: philosopher number, from 0 to N−1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

28

Problemas clássicos de comunicação entre processos

- Sugestão de Exercícios:
  - Entender a solução para o problema dos **Filósofos** utilizando semáforos:
    - Identificando a(s) **região(ões) crítica(s)**;
    - Descrevendo **exatamente** como a solução funciona;
  - Entender a solução para o problema dos Produtores/Consumidores utilizando monitor:
    - Identificando a(s) **região(ões) crítica(s)**;
    - Descrevendo **exatamente** como a solução funciona;

29