Universidade Federal
de Mato Grosso
Campus Rondonópolis

Sistemas Operacionais

Aula 7 – Comunicação interprocessos (*Interprocess Communication) (2)*

Prof. Msc. Cleyton Slaviero

cslaviero@gmail.com

Dormir e acordar

- Soluções TSL (ou XCGH em proc. Intel x86) dependem de espera ociosa
  - Gasta tempo de CPU
  - Inversão de prioridade
    - Imagine dois processos, H e L
    - H sempre é executando quando no estado pronto
    - L entra em região crítica, H fica pronto pra executar
    - H tenta executar, mas região crítica → espera ociosa
    - H nunca irá executar, pois L nunca será escalonado (H tem prioridade)
- Solução: bloquear
  - Wake e sleep

Universidade Federal
de Mato Grosso
Campus Rondonópolis

Dormir e acordar

- Exemplo: Produtor e consumidor (ou buffer limitado – *bounded buffer*
  - Um produtor insere itens num buffer, outro remove
  - Problema para o produtor
    - Buffer cheio
  - Problema para o consumidor
    - Buffer vazio

Dormir e acordar

- Uma solução
  - Coloca-se o produtor/consumidor para dormir
  - O outro o acorda quando a condição for satisfeita

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Dormir e acordar

- Desvantagens:
  - Mesmos problemas de antes
  - Condição de corrida: variável count tem acesso irrestrito
    - Escalonador pode trocar de processo na hora em que count é verificado pelo consumidor
    - Sinal de acordar é perdido
  - E se usarmos um bit de espera (guarda sinais de acordar)?
    - Ótimo pra dois processos, péssimo para n

Semáforos

- E se usarmos uma variável para contar o número de sinais salvos para uso futuro?
  - Semáforo
- Proposto por Dijkstra (1965)
  - Duas operações (*up* e *down*)
    - Generalizações do sleep e wake, respectivamente
  - **Down** verifica se o valor é maior que 0; caso positivo, decrementa; se for negativo, põe o processo para dormir
  - **Ação atômica**
    - Fundamental para que semáforos funcionem
  - **Up** incrementa o valor de um semáforo
    - Se um processo estiver dormindo, sistema escolhe (e.g. aleatório) um processo para acordar

Semáforos

- Semáforos resolvem o problema de perda do sinal de acordar
  - Precisam estar implementados de maneira indivisível para não haver problemas

- Up em down geralmente são programados como chamadas de sistema
  - Se várias CPUs estiverem sendo usadas, cada semáforo deverá ser protegido por uma variável de trava, com o uso da instrução TSL

Semáforos

- Só para lembrar
  - TSL/XCGH é diferente de espera ocupada do produtor/consumidor
  - Operação de semáforo dura apenas alguns microssegundos
  - Produtor/consumidor pode demorar um tempo arbitrariamente longo

Semáforos

- Uma solução para o produtor/consumidor
  - Três semáforos
    - Full – número de lugares preenchidos
    - Empty – conta número de lugares vazios
    - Mutex – garante a exclusão mútua
      - Semáforo binário

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );               /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Semáforo

- Duas formas de usar o semáforo
  - Sincronização
    - Garante que produtor e consumidor não tentem acessar o buffer se estiver cheio ou vazio , respectivamente
  - Como mecanismo de exclusão mútua
    - Semáforo binário

Mutex

- Um semáforo binário também é conhecido como mutex
- Variável compartilhada
  - Dois estados – travado (lock) e destravado (unlocked)
- Se um thread/processo precisa acessar uma região crítica
  - mutex_lock() – caso esteja travada a região, block até destravar
  - É possível implementar usando TSL
  - Se um thread não puder usar pode também chamar thread_yield(), e quando entrar de novo, tenta novamente
    - Não bloqueia, mas volta para a fila de pronto
  - Algumas bibliotecas fornecem um ”trylock”
    - O procedimento decide o que fazer

Mutex

- Como podemos compartilhar o mutex?
  - Via kernel
  - SO permitindo compartilhar alguns recursos entre processos/threads
  - Se nada é possível: Um arquivo
- Quer dizer que thread = processo?
  - Não. Mesmo com esse compartilhamento, processos e threads ainda tem diferenças

Futex

- Espera ocupada/spin lock
  - Pode ser boa se a espera é curta, mas desperdiçam recursos se a espera é longa
- Bloquear o processo
  - pode ser bom, mas depende de troca para o modo kernel
    - Custoso!
- Qual a solução?

Universidade Federal de Mato Grosso
Campus Rondonópolis

Futex

- O melhor dos dois mundos!
- Futex - *Fast User Space Mutex*
  - Um processo só faz solicitações ao modo kernel se necessário
  - Se não há contenção, simplesmente executa
  - Se houver, faz a chamada de sistema para solicitar o lock ou acessar a fila de processos bloqueados
  - Quando acabar, se não houver ninguém esperando não é necessário liberar o lock; logo não se faz necessária qualquer chamada de sistema
- É ”rápido” no espaço de usuário

Monitores

- Situação hipotética
  - Se no exemplo do produtor-consumidor, eu desse down no mutex antes do buffer no produtor…
  - Se o buffer estiver cheio, travaria com mutex em 0
  - Consumidor tentaria consumir...
    - Não consegue, pois o mutex foi travado
  - O buffer não consegue ser consumido, logo…
    - Temos um deadlock!
- Hansen e Hoare propuseam uma primitiva de sincronização de alto nível chamada **monitor**

Monitores

- Um monitor é uma coleção de procedimentos, variáveis e estruturas de dados agrupadas em um módulo ou pacote
- Processos chamam o monitor, mas não acessam suas estruturas internas
- Monitores são dependentes da linguagem
  - C, por exemplo, não especifica monitores!

Monitores

- Num monitor, somente um processo pode estar ativo em um determinado momento
  - O compilador da linguagem entende isso e garante
- Numa chamada, o monitor vai verificar se é possível aquele processo utilizará uma região crítica
- O monitor é responsável pelos mecanismos de exclusão mútua
  - Geralmente se usa mutex

Monitores

- No entanto, monitores podem não ser suficientes
- Um processo pode querer bloquear quando não pode prosseguir
- **Variáveis condicionais + wait() e signal()**
  - Permitem ou bloquear um processo de acordo com uma condição
  - Exemplo: o produtor encontra o buffer cheio
    - Wait() até que alguém dê o signal()

```
#include <stdio.h>
#include <pthread.h>

#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                        /* put item in buffer */
                pthread_cond_signal(&condc);       /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                        /* take item out of buffer */
                pthread_cond_signal(&condp);       /* wake up producer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Monitores

- Variáveis de condição não são contadores!
- Se uma variável de condição é sinalizada, sem alguém esperando, o sinal é perdido
  - Wait() tem que vir antes do signal()
- Com monitores não há problema em um tentando dormir enquanto o outro tenta acordar
  - O monitor tem controle da execução e garante que o wait() é completado

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;
```

Monitores

- Um processo pode acordar outro processo dormindo enviando um signal() na variável de condição que estava esperando
  - Precisamos no entanto garantir que dois processos não fiquem no monitor ao mesmo tempo
- O que fazer depois do signal() ?
  - Hoare propôs deixar o processo recém acordado rodar, suspendendo quem estava executando (voltando a rodar depois)
  - Hansen propôs que o processo em execução saia logo depois do sinal
    - Signal() seria então o último procedimento do processo

Monitores

- Desvantagens
  - Conceito da linguagem de programação
    - No caso de semáforos, é possível adicionar
    - Para monitores, o design da linguagem deve suportar
  - Acesso distribuído com múltiplas CPUs
    - Cada um com memória, e conectados a uma rede, fica impossível
- Conclusão
  - Semáforos são muito baixo-nível
  - Monitores são usáveis em algumas linguagens apenas
  - Além disso há o problema de troca de informações entre máquinas diferentes…

Troca de mensagens

- Os mecanismos já considerados exigem do S.O. somente a sincronização
  - Asseguram a exclusão mútua, mas não garantem um controle sobre as operações desempenhadas sobre o recurso.
  - Deixam para o programador a comunicação de mensagens através da memória compartilhada
- A troca de mensagens é um mecanismo de comunicação e sincronização
  - Exige do S.O., tanto a sincronização quanto a comunicação entre os processos.
  - É um mecanismo mais elaborados de comunicação e sincronização entre processos

Troca de mensagens

- Esquema de troca de mensagens:
  - Os processos enviam e recebem mensagens, em vez de ler e escrever em variáveis compartilhadas.
  - Sincronização entre processos:
    - Garantida pela restrição de que uma mensagem só poderá ser recebida depois de ter sido enviada.
    - A transferência de dados de um processo para outro, após ter sido realizada a sincronização, estabelece a comunicação.

Troca de mensagens

- Possui duas primitivas:
- Send
  - **send(destino, &mensagem);**
- Receive
  - **receive(fonte, &mensagem);**
- Se não houver mensagem disponível, o receptor pode:
  - Bloquear, até que haja alguma mensagem (Blocked)
  - Retornar com mensagem de erro (Unblocked)
- Implementadas como chamadas ao sistema

Mensagens - primitivas

- As primitivas podem ser de dois tipos
  - **Bloqueantes**: quando o processo que a executar ficar bloqueado até que a operação seja bem sucedida
    - Quando ocorrer a **entrega** efetiva da mensagem ao processo destino, no caso da emissão
    - Quando ocorrer o **recebimento** da mensagem pelo processo destino, no caso de recepção.
  - **Não bloqueantes:** quando o processo que executar a primitiva, continuar sua execução normal, independentemente da entrega ou do recebimento efetivo da mensagem pelo processo destino.

Mensagens - primitivas

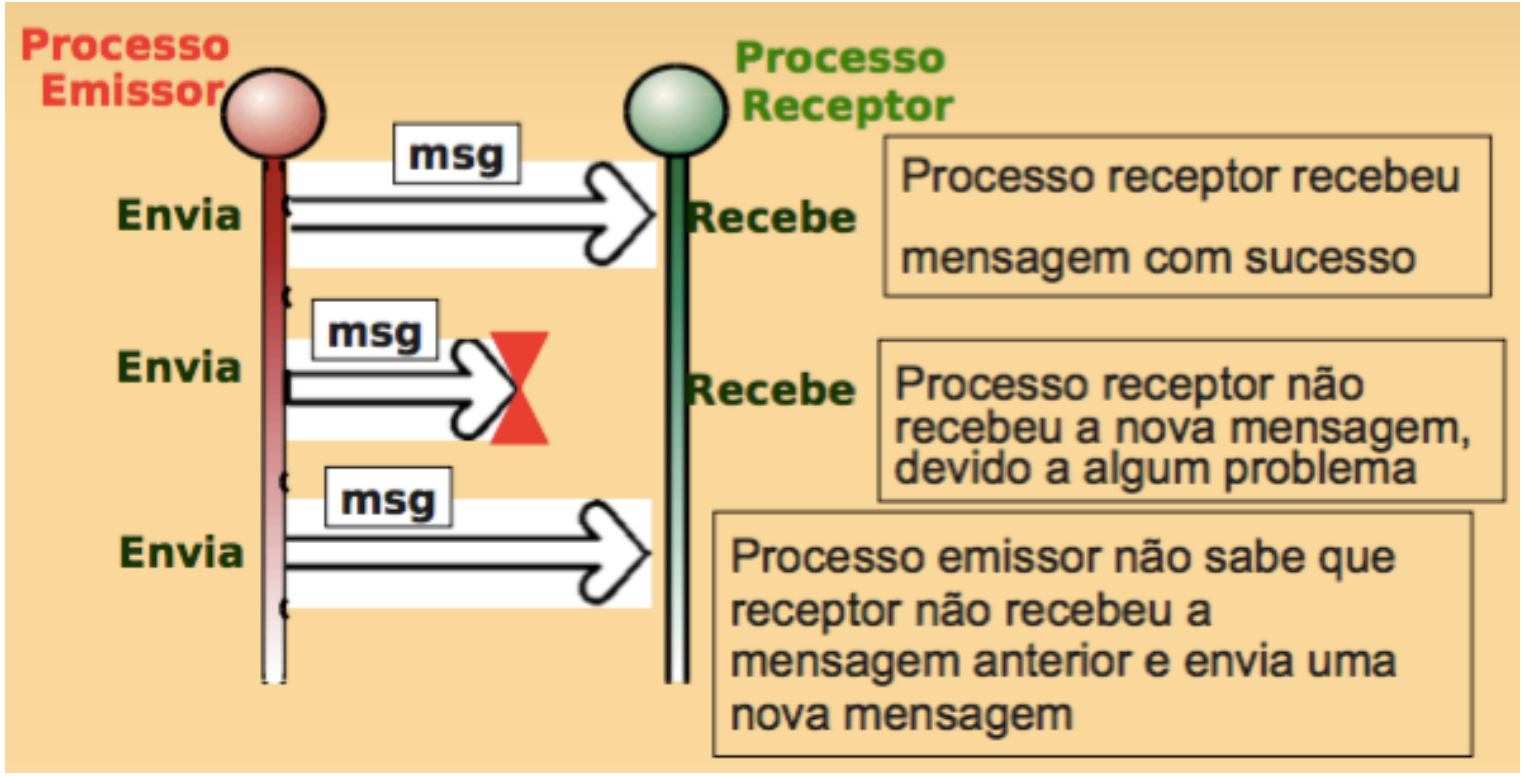
- Combinação de primitivas
- Existem quatro maneiras de se combinar as primitivas de troca de mensagens:
  - Envia bloqueante → recebe bloqueante (síncrono)
  - Envia bloqueante → recebe não bloqueante (semi-síncrono)
  - Envia não bloqueante → recebe bloqueante (semi-síncrono
  - Envia não bloqueante → recebe não bloqueante (assíncrono)

Mensagens

- Os sistemas de troca de mensagens possuem alguns problemas e estudos de projetos interessantes
  - Principalmente quando os processos comunicantes estão em máquinas diferentes, conectadas por uma rede de comunicação.
  - Os principais são:
    - Perda de mensagens
    - Perda de reconhecimento
    - Nomeação de Processos
    - Autenticação
    - Estudos de Projeto para quando emissor e receptor estiverem na mesma máquina

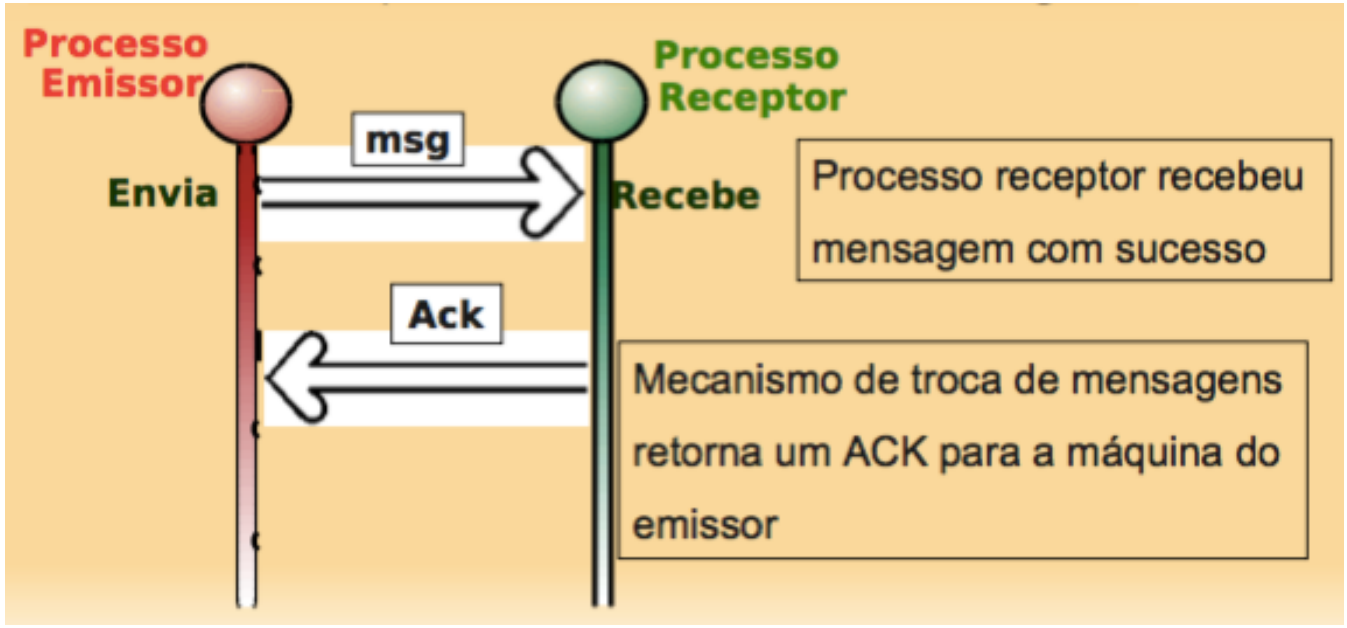
Mensagens - Problemas

- Perda de mensagens
  - Podem ocorrer por falhas na rede, por exemplo



Perda de mensagens

- Possível solução
  - O receptor, ao receber uma nova mensagem, envia uma mensagem especial de reconhecimento (ACK).
  - Se o emissor não receber um ACK dentro de um determinado intervalo de tempo, deve retransmitir a mensagem.



Universidade Federal de Mato Grosso
Campus Rondonópolis

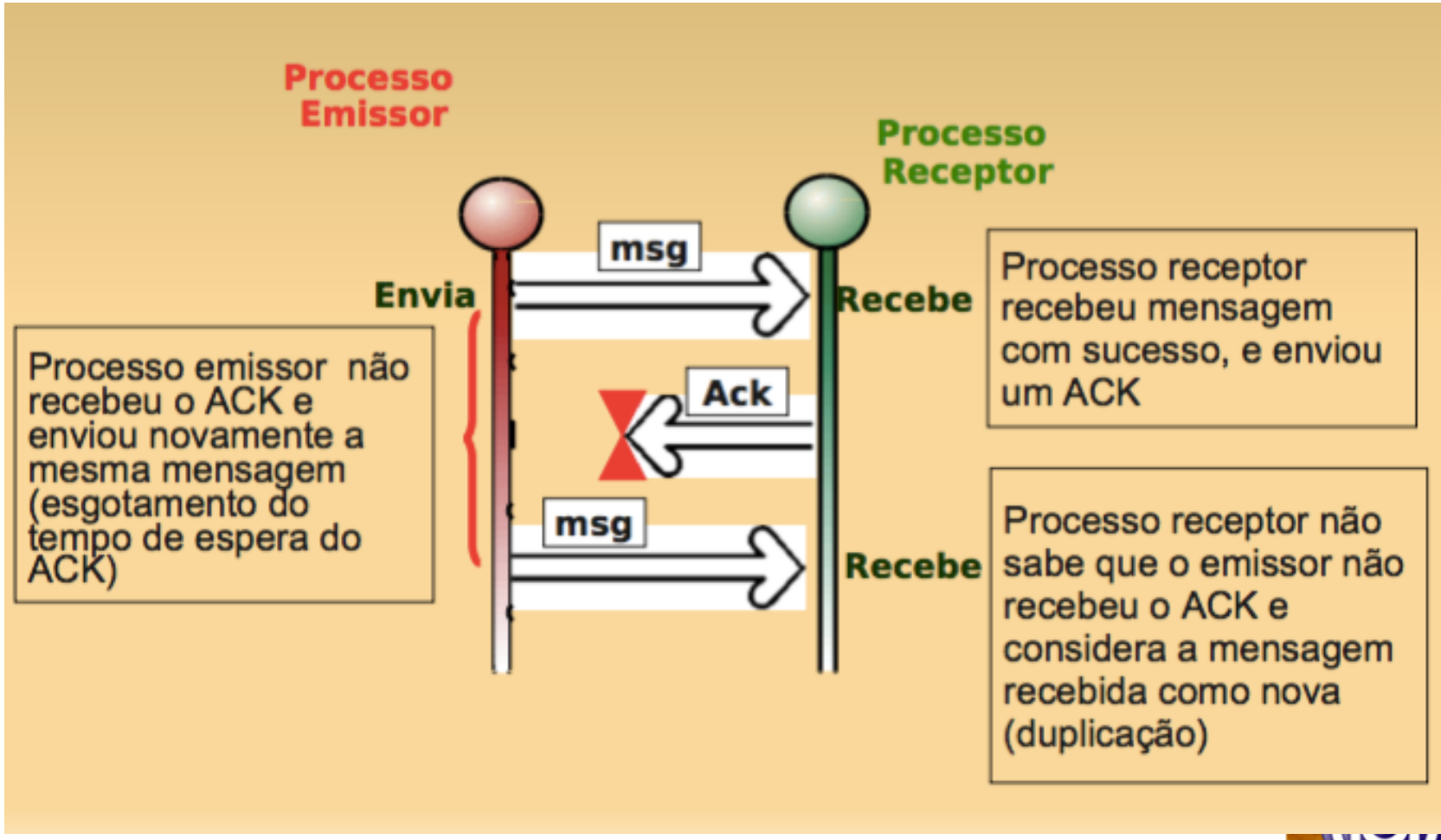
Perda de mensagens

- Possível solução
  - O que acontece se a mensagem é recebida corretamente, mas o ACK é perdido?

Perda de reconhecimento

- Possível solução
  - O que acontece se a mensagem é recebida corretamente, mas o reconhecimento (ACK) é perdido?
  - O emissor retransmitirá a mensagem
  - O receptor irá recebê-la duas vezes

Perda de reconhecimento



Perda de reconhecimento

- Possível solução
  - É essencial que o receptor seja capaz de distinguir uma nova mensagem de uma retransmissão
- Numerar as mensagens
  - Se o receptor receber mensagem com o mesmo número que alguma anterior, sabe que é duplicata, ignorando-a.

Mensagens – Nomeação de processos

- Os processos devem ser nomeados de maneira única, para que o nome do processo especificado no Send ou Receive não seja ambíguo.
  - Ex: processo@máquina (normalmente existe uma autoridade central que nomeia as máquinas).
- Quando o número de máquinas é muito grande:
  - processo@máquina.domínio.

Mensagens – Autenticação

- Como o cliente pode dizer que está se comunicando com o servidor real, e não um impostor?
- Devemos permitir a comunicação e acessos apenas dos usuários autorizados.
- Uma solução é criptografar as mensagens com uma chave conhecida apenas por usuários autorizados.

Mensagens

- Questões de estudo de projeto para quando emissor e receptor estão na mesma máquina
  - Desempenho:
    - Copiar mensagens de um processo para o outro é mais lento do que operações com semáforos e monitores;
  - Sugestão:
    - Limitar o tamanho das mensagens ao dos registradores Passar as mensagens usando os registradores

Mensagens

- Produtor-consumidor com mensagens
  - Todas as mensagens tem o mesmo tamanho
  - N mensagens são usadas
  - O consumidor envia N mensagens vazias para o produtor
  - Se o produtor tem um item, tira uma mensagem vazia e adiciona uma cheia
  - Se o consumidor trabalha mais rápido, o oposto acontece

```
#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 */
     }
}
```

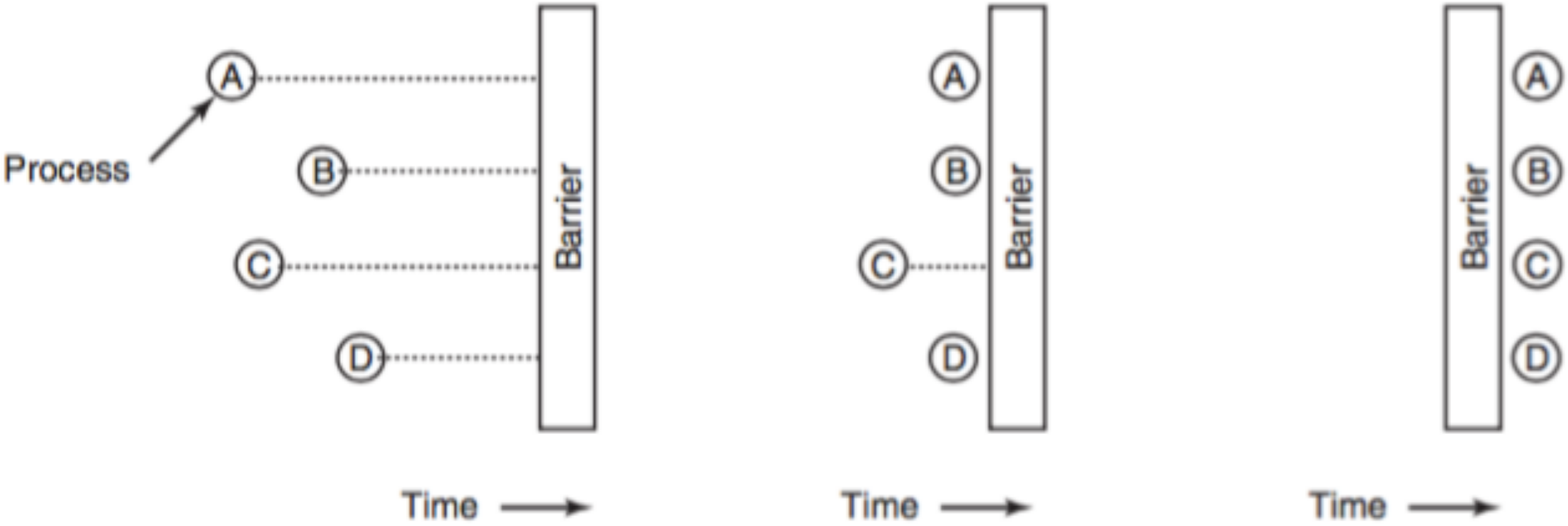
Mensagens

- Diferentes variantes
  - Uma forma é pensar que mensagens vão endereçadas a processos
  - Outra forma é pensar em uma caixa de correio (mailbox)
    - Mensagens passam a ser enviadas para o correio, e não para o processo
  - Outra forma: Rendezvous
    - Elimina-se o buffer
      - Se não houver mensagem, bloqueia

Barreiras

- Mecanismo para grupos de processos ao invés de situações como a do produtor consumidor
- Em alguns casos, é necessário que certas etapas do processamento terminem para que o processo continue
- Para isso, cria-se uma **barreira** ao fim de cada fase
- Quando o processo/thread alcança a barreira, ele é bloqueado até que todos os outros processos/threads terminem
  - Sincronização

Barreiras



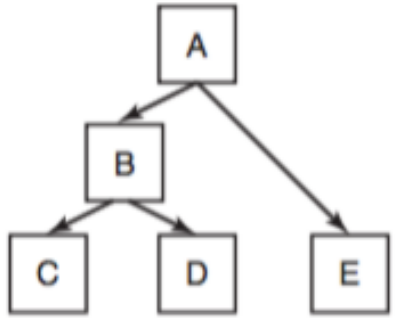
Barreiras

- Exemplo:
  - Imagine um programa que meça variações de temperatura em uma placa de metal
  - Matriz NxN com temperaturas
    - A cada instante de tempo, novas temperaturas sào calculadas
    - O cálculo depende de células adjacentes
    - Pensem em ma matriz muito grande
    - É preciso que toda a matriz seja calculada antes de prosseguir

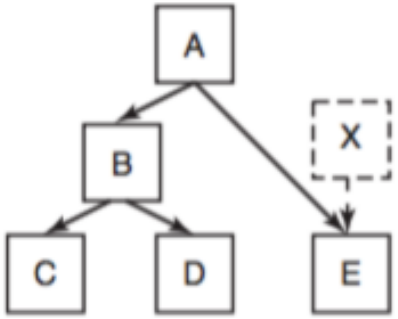
Evitando Locks

- O ideal é não termos travas
  - Como fazer isso?
  - Em alguns casos é inevitável
    - Exemplo: ordenação+ média de um vetor (array)
- Em alguns casos, podemos permitir que alguém escreva mesmo enquanto outros o usam
  - Garante-se que cada leitor leia apenas a versão antiga dos dados ou a nova, mas nunca uma mistura das duas
- **Read-copy-update**

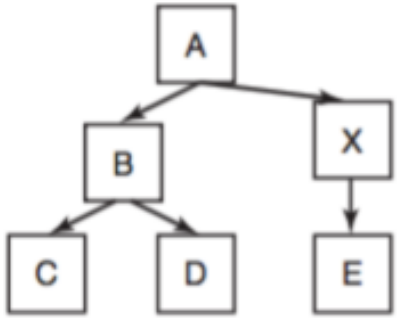
Adding a node:



(a) Original tree.

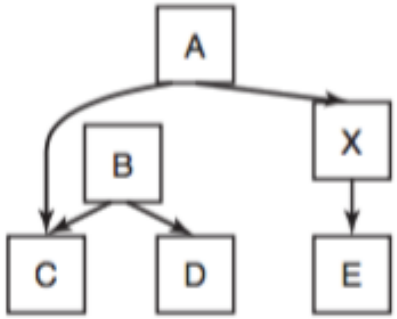


(b) Initialize node X and connect E to X. Any readers in A and E are not affected.

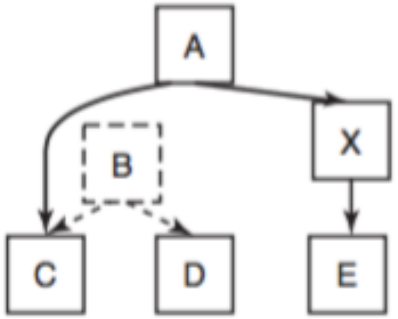


(c) When X is completely initialized, connect X to A. Readers currently in E will have read the old version, while readers in A will pick up the new version of the tree.

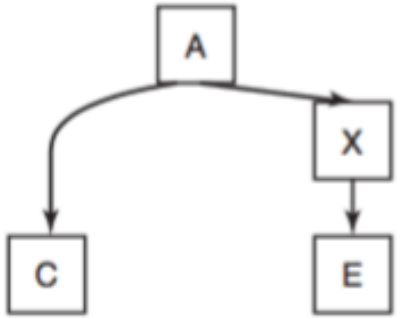
Removing nodes:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.



(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed any more.



(f) Now we can safely remove B and D

Evitando locks

- Problema
  - Enquanto não se sabe se há leitores, não podemos liberar eles (a versão antiga da árvore)
  - **Quanto tempo esperar?**
    - RCU determina o tempo máximo que um leitor pode guardar uma referência a estrutura de dados
    - Os leitores acessam a ”região crítica de leitura”
    - Define-se então um ”período de graça” como o tempo que sabem que cada thread estará fora da seção crítica ao menos uma vez
      - Por exemplo: esperar todas as threads trocarem de contexto