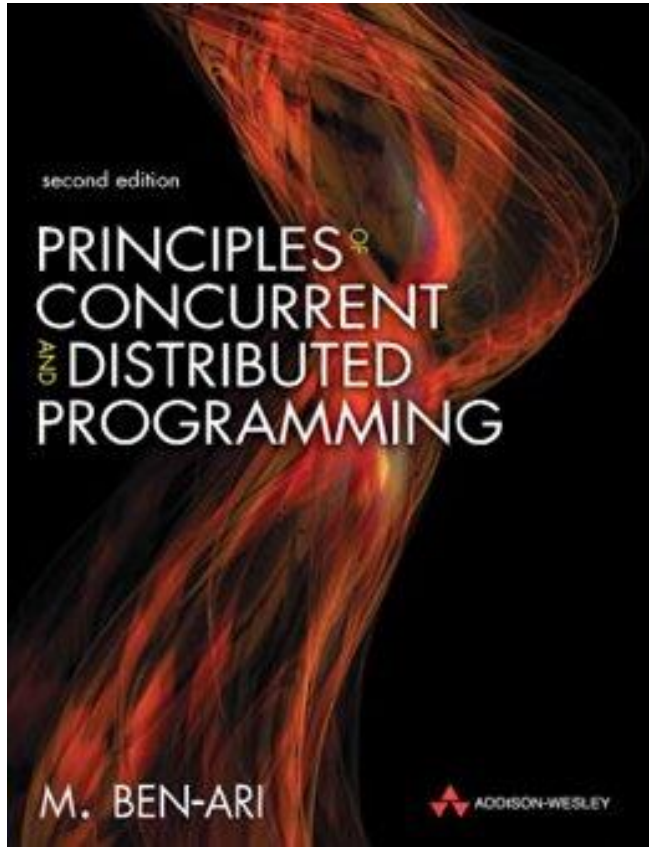
Comunicação e Sincronização entre Processos

Prof. Marcelo Veiga Neves

marcelo.neves@pucrs.br

Conteúdo

- Tipos de Comunicação
  - Compartilhamento de memória
  - Troca de mensagens
- Tipos de Sincronização
  - De competição
  - De cooperação
- Mecanismos de sincronização
  - Semáforos
  - Monitores
  - Mensagens/Canais



Tipos de comunicação

- Processos que cooperam precisam eventualmente comunicar:
  - Troca de dados ou sincronização
- Dois tipos de comunicação:
  - Através de uma memória compartilhada
  - Através de troca de mensagens
- Escolha depende normalmente do hardware disponível

Comunicação por memória compartilhada

- Utilizada em multiprocessadores com memória compartilhada
- Processos precisam ter acesso ao **mesmo espaço de endereçamento**
  - Exemplo: variável global de um programa com *threads*
- Normalmente mais eficiente que troca de mensagens
- Acesso ao dado se dá por **referência**:
  - Variáveis globais
  - Índices de um vetor ou matriz
  - Ponteiros
  - Instância de objeto

Comunicação por troca de mensagem

- Utilizada em multicomputadores sem memória compartilhada
- Espaços de endereçamento são diferentes
- Como não consegue passar uma referência, precisa movimentar o conteúdo todo
  - Envia uma mensagem com o conteúdo
  - Normalmente menos eficiente que compartilhar memória
- Necessário um mecanismo de comunicação
  - Exemplo: biblioteca de mensagens via rede, comunicação entre processos.

Como escolher?

- Máquina sem memória compartilhada, só troca de mensagens
- Máquina com memória compartilhada, pode-se optar por
  - Memória compartilhada:
    - Normalmente é mais eficiente
    - Pode causar *data races*, precisa tratar problemas de seção crítica
  - Troca de mensagens:
    - Pode facilitar modelagem em algumas linguagens modernas (ex: canais em Go)
    - Pode ser utilizada como mecanismo sincronização
    - Pode causar problemas de bloqueio, se ficar esperando por uma mensagem que nunca vem

Tipos de sincronização

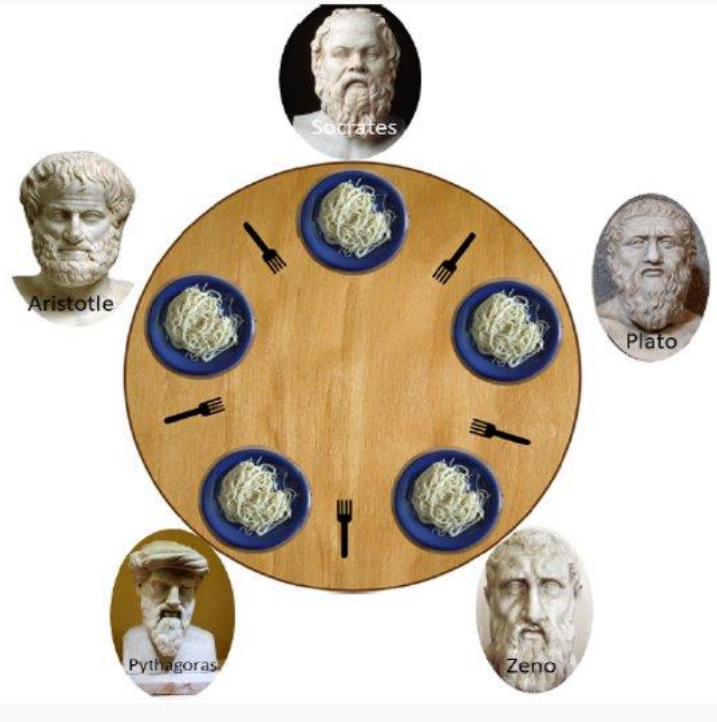
- Processos que cooperam precisam se coordenar, exemplos:
  - Esperar que alguém termine de fazer alguma coisa que eu preciso
  - Organizar a disputa por um recurso
  - Organizar a disputa por dados compartilhados
- Dois tipos de sincronização:
  - Sincronização de competição
  - Sincronização de cooperação
- Escolha depende do **problema** a ser resolvido

Sincronização de competição

- Necessária quando dois ou mais processos **disputam** algum dado ou recurso compartilhados
- A coordenação nestes compartilhamentos é fundamental para que o programa **funcione corretamente**
- Exemplos:
  - *Data race* (memória/dados)
  - Jantar dos filósofos (recursos)

Jantar dos filósofos

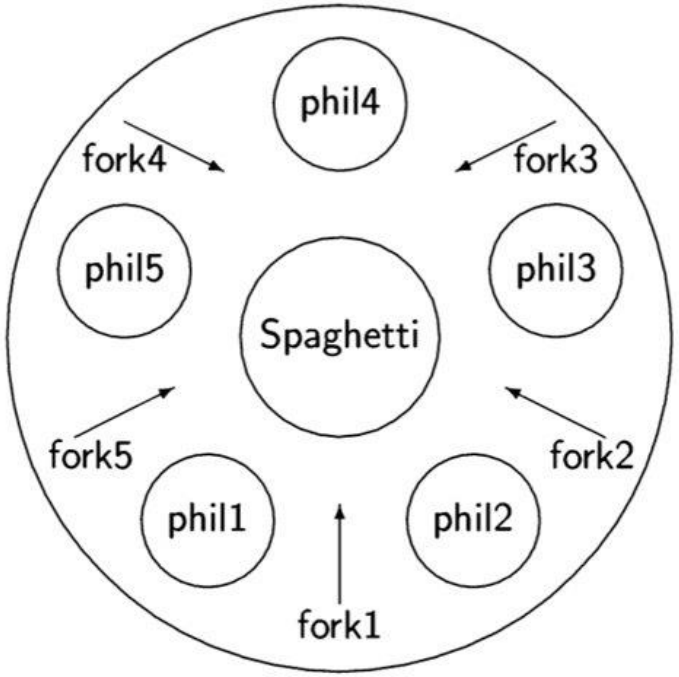
- O Jantar dos filósofos foi proposto por Dijkstra em 1965 como um problema de sincronização.
  - Cinco filósofos estão sentados em uma mesa redonda para jantar.
  - Cada filósofo tem um prato com espaguete à sua frente.
  - Cada prato possui um garfo para pegar o espaguete.
  - O espaguete está muito escorregadio e, para que um filósofo consiga comer, será necessário utilizar dois garfos.
  - Cada filósofo alterna entre duas tarefas: comer ou pensar.



Jantar dos filósofos

- Considerando cada filósofo como um processo, como garantir que eles executem as tarefas de comer e pensar?
  - Não pode travar (deadlock)
  - Não tem espera indefinida (starvation)
  - Deve manter um bom grau de concorrência (justiça)

- É um problema de seção crítica:
  - Pensar é uma seção não crítica
  - Comer é uma seção crítica (pegar os dois garfos)

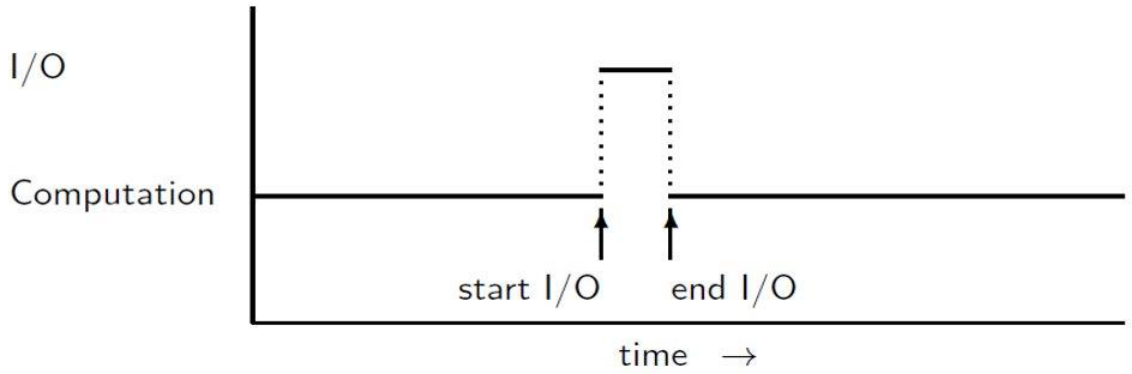


Sincronização de cooperação

- Necessária quando dois ou mais processos precisam se coordenar para **colaborar** em um problema
- A coordenação nestes casos é fundamental para que o programa funcione de forma **correta** e **eficiente**
- Exemplos:
  - Operação de entrada e saída (I/O)
  - Condição de parada
  - Produtor Consumidor

Operação de entrada e saída (I/O)

- Cooperação entre dois ou mais processos que estão realizando uma operação de entrada e saída em um sistema operacional
  - Processo que pediu a operação de I/O
  - Processos de atendimento do SO
- Exemplo: processo pediu para ler uma string do teclado



Condição de Parada

- Coordenação de dois ou mais processos que estão executando de forma concorrente para que o programa **termine graciosamente** (*gracefully*)
  - Nenhum processo deve ser interrompido antes de terminar a execução
  - Nenhum processo deve ficar "pendurado"
- Exemplo: programa com múltiplas threads em C ou Go

Produtor/Consumidor

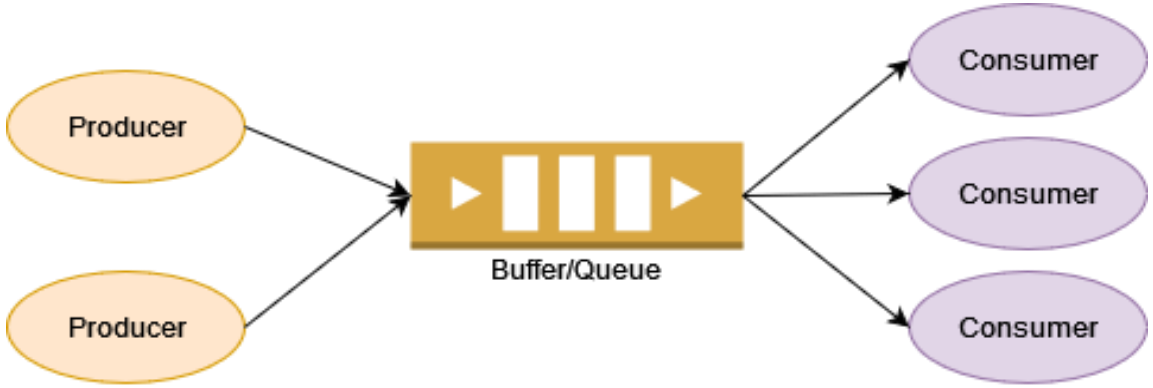
- Coordenação de dois ou mais processos que estão executando de forma concorrente para que uns consumam o que os outros produziram
- Resulta do **desacoplamento** entre produtores e consumidores aumentando a concorrência e permitindo paralelismo
  - Processos produtores e consumidores executam de forma concorrente

Produtor/Consumidor

- Modelo clássico de organização de trabalho proposto por Dijkstra que pode ser usado em várias situações
  - Recebimento de dados pela rede
  - Tratamento de I/O
- Também conhecido como problema do buffer limitado (*bounded buffer problem*):
  - Buffer tem um tamanho finito (N posições)
  - Produtores tem que bloquear se o buffer estiver cheio
  - Consumidores tem que bloquear se buffer estiver vazio

Produtor/Consumidor

- Pode-se ter vários produtores e vários consumidores para aumentar a concorrência
- Neste caso, também exige sincronização de competição para evitar *data races*
  - Produtores não escrevam na mesma posição
  - Consumidores não leiam da mesma posição



Mecanismos de sincronização

- Semáforos
- Monitores
- Mensagens/Canais

Semáforos

- Mecanismo criado por Djikstra em 1965 para ser aplicado nos problemas de sincronismo
- Um semáforo é uma estrutura de dados com um contador e uma fila de PIDs (identificadores de processos)
  - O contador controla quantos processos podem entrar na seção crítica
    - semáforo = 0: recurso está sendo utilizado
    - Semáforo > 0: recurso livre
  - A fila de PIDs guarda quais os processos que estão esperando para entrar
- Operações sobre semáforos:
  - Wait/Down: executada quando processo deseja entrar na seção crítica
  - Signal/Up: executada quando processo deseja sair da seção crítica (liberar o recurso)

Semáforos

- Wait/Down – original P() do holandês Probeer (tentar)
  - Verifica se o valor do contador do semáforo
  - Se for maior que zero, decrementa e contunua executando
  - Se for zero, bloqueia/dorme e coloca seu PID na lista de PIDs do semáforo
- Signal/Up – V() do holandês Verhoog (incrementar)
  - Incrementa contador
  - Se lista não está vazia, desbloqueia um destes processos (escolha arbitrária)

- As operações sobre semáforos são atômicas

Semáforos

- Semáforos podem ser dos tipos binários ou gerais
- **Semáforo binário:** só pode ter os valores 1 e 0
  - Utilizados para implementar exclusão mútua (mutex)
  - Inicializado com o valor 1 (pois só um entra de cada vez)
- **Semáforo geral:** pode ter qualquer valor não negativo
  - É inicializado com o valor de quantos processos queremos que possam entrar juntos na seção crítica

Semáforos

- Ambos os tipos podem ser fortes ou fracos
- **Semáforo fracos:** não mantém a ordem de acesso
  - Quando o processo liberado é escolhido arbitrariamente da fila de bloqueados
  - Mais concorrente, mas pode gerar *starvation*
- **Semáforos fortes:** mantém a ordem de acesso
  - Quando o processo é o primeiro da fila de bloqueados
  - Menos concorrente, mas evita *starvation*

Implementação de semáforos

- Semáforo pode ser utilizado para evitar condições de corrida, mas
- Como garantir acesso atômico ao contador do semáforo?
  - Possibilidade de data race

- Mecanismo *Test and Set Lock* (TSL):
  - Necessita suporte em hardware
  - Utiliza instrução *Test-and-Set* do processador
    - Permite ler o valor de um endereço de memória e escrever 1 (set) em uma única operação atômica
  - Será estudado em Sistemas Operacionais

Seção crítica com semáforos

- Utilização de semáforo para evitar *data races:*
  - Só deixar um processo por vez ler e atualizar o valor da variável compartilhada
  - Operação deve ser atômica e não permitir que outros processos a interrompam por entrelaçamento (exclusão mútua)
  - Usando semáfora binário (*mutex*):
    - Chama-se P()/Wait() para entrar na seção crítica
    - Chama-se e V()/Signal() quando terminarmos a atualização

Algorithm 6.1: Critical section with semaphores (two processes)	
binary semaphore S ← (1, ∅)	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

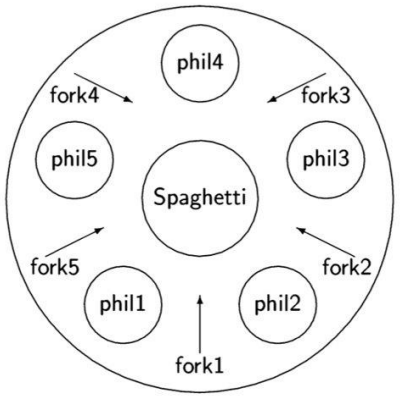
Jantar dos filósofos com semáforos

- Utilização de semáforo para resolver o problema dos filósofos
- Uma modelagem possível:
  - Envolve ter um processo para cada filósofo e um semáforo para cada garfo
  - Processos do tipo filósofo pegam (alocam) garfos fazendo P()/Wait() no respectivo semáforo e liberam fazendo V()/Signal () nele

Algorithm 6.10: Dining philosophers (first attempt)

semaphore array [0..4] fork ← [1,1,1,1,1]

```
      loop forever
p1:     think
p2:     wait(fork[i])
p3:     wait(fork[i+1])
p4:     eat
p5:     signal(fork[i])
p6:     signal(fork[i+1])
```



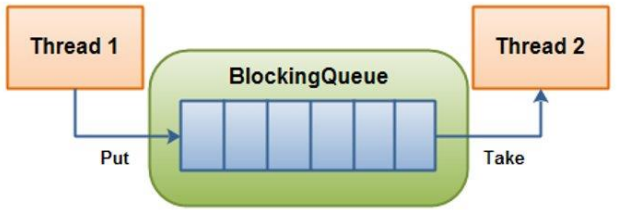
Produtor/consumidor com semáforos

- Utilização de semáforo para resolver a coordenação de produtores e consumidores
- Uma modelagem possível:
  - Usar um semáforo um binário para o controle do buffer vazio
  - Usar um semáforo geral para controle do buffer cheio
  - Inicializar o geral com a capacidade do buffer

Algorithm 6.8: Producer-consumer (finite buffer, semaphores)

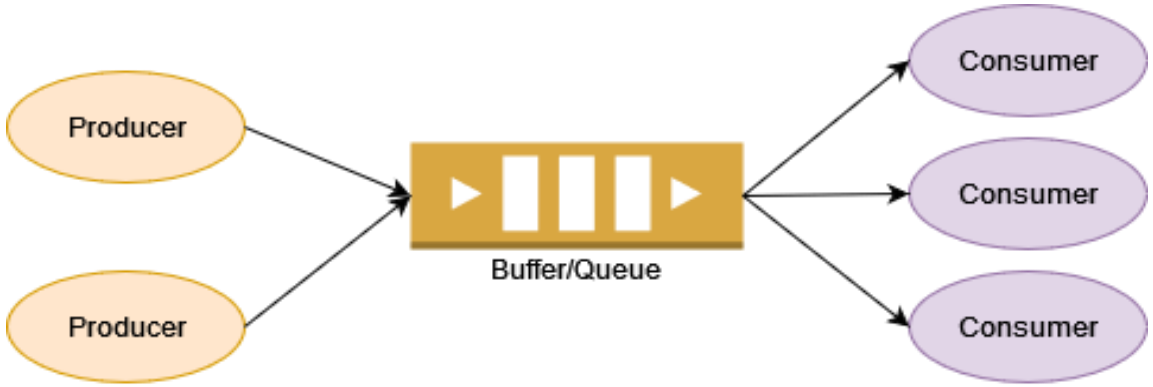
finite queue of dataType buffer ← empty queue
semaphore notEmpty ← $(0, \emptyset)$
semaphore notFull ← $(N, \emptyset)$

	producer		**consumer**
	dataType d		dataType d
	loop forever		loop forever
p1:	d ← produce	q1:	wait(notEmpty)
p2:	wait(notFull)	q2:	d ← take(buffer)
p3:	append(d, buffer)	q3:	signal(notFull)
p4:	signal(notEmpty)	q4:	consume(d)



Produtor/consumidor com semáforos

- No caso de vários processos produtores e consumidores, precisamos de mais dois semáforos para proteger *data-race* na escrita e leitura das posição do buffer

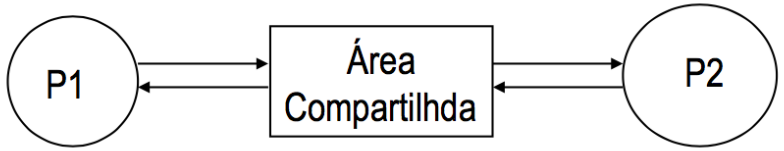


Monitor

- Idealizado por Hoare (1974) e Brinch Hansen (1975)
- Monitor é um mecanismo de sincronização de alto nível
  - Conjunto de procedimentos, variáveis e estruturas de dados agrupados em um único módulo ou pacote
- Monitor concentra o controle de acesso à seção crítica:
  - Somente um processo pode estar ativo dentro do monitor em um mesmo instante
  - Outros processos ficam bloqueados até que possam estar ativos no monitor
- Monitores se tornaram mecanismos de sincronização muito importantes por serem uma generalização do conceito de orientação a objetos
  - Usados em Ada, Java, C#
  - Encapsulam dados e métodos de uma classe
  - Ex: synchronized em Java

Comunicação entre processos

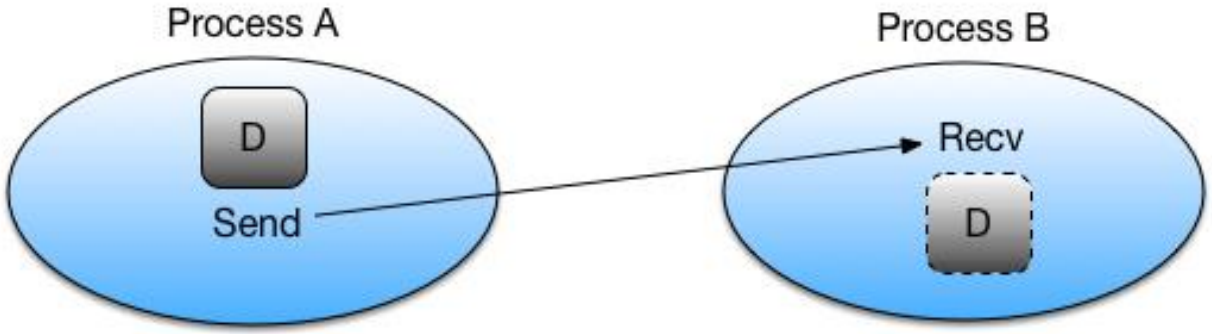
- Memória compartilhada:
  - os processo compartilham variáveis e trocam informações através do uso dessas



- Sem memória compartilhada:
  - os processos compartilham informações através de troca de mensagens
  - necessidade de mecanismo de comunicação entre os processos (geralmente envolve o Sistema Operacional), por exemplo:
    - Troca de mensagens
    - Canais

Troca de mensagens

- Método de comunicação em que entidades (threads, processos) trocam informações por meio de mensagens, sem compartilhar memória
- Normalmente utiliza rotinas do tipo send()/recv() ou similar
- Exemplos:
  - Sockets, filas de mensagens do sistema operacional (ex: POSIX message queue), bibliotecas de troca de mensagens (ex: MPI), canais em linguagem de programação (ex: Go e Rust), etc.

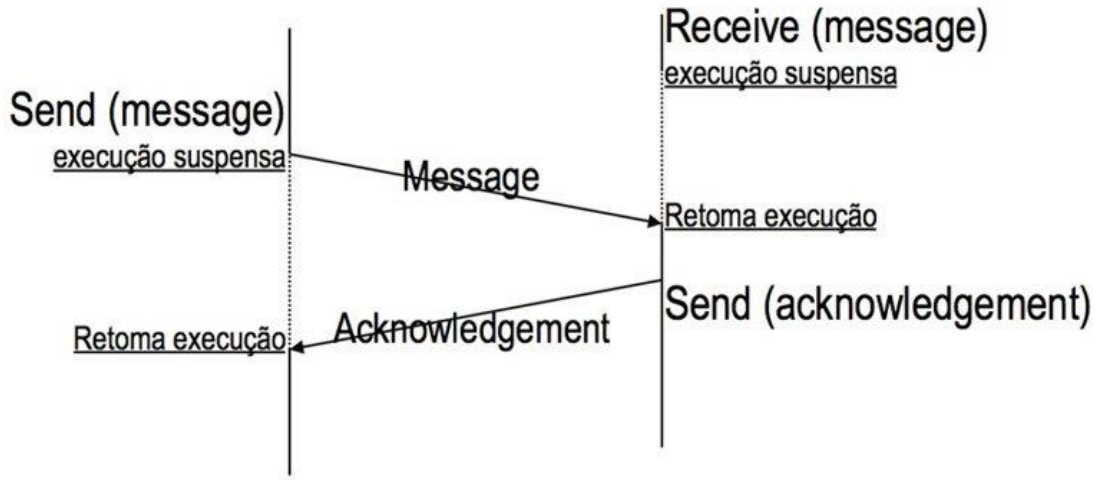


Troca de mensagens: sincronização

- Sincronização em operações *send/receive*
- Receive **bloqueante**:
  - processo receptor fica bloqueando na operação de *receive* até chegada da mensagem
  - problema: ficar bloqueado para sempre - mecanismo de *time-out*
- Receive **não-bloqueante**:
  - processo receptor informa onde armazenar mensagens e prossegue
    - Pode fazer outras coisas enquanto espera
  - processo receptor fica sabendo da chegada da mensagem por:
    - *Polling*/teste periódico
    - Interrupção

Troca de mensagens: sincronização

- Sincronização em operações *send/receive*
  - quando *send* e/ou *receive* são bloqueantes a comunicação é dita **síncrona**
  - senão é dita **assíncrona**



Troca de mensagens: bufferização

- **Buffer:** espaço em memória para armazenamento temporário
- Por que é necessário utilizar "bufferização"?
  - Transmissão da mensagem: copia corpo da mensagem do espaço de endereçamento do processo transmissor para espaço de endereçamento do receptor
    - Processo receptor pode não estar pronto para receber
    - Sistema operacional salva mensagem (SO do lado receptor implementa "bufferização")
- Tem relação direta com o **sincronismo da comunicação**

Troca de mensagens: tipos de bufferização

- *Null buffer* (um extremo)
  - Sem espaço de armazenamento temporário
  - Permite apenas para comunicação síncrona
- *Unbounded-capacity buffer* (outro extremo)
  - Capacidade de armazenamento ilimitada
  - Não existe na prática!
- Tipos intermediários de buffers:
  - *Single-message buffer*
  - *Bounded-capacity, finite-bound or multiple-message buffer*
  - Permitem comunicação assíncrona

Troca de mensagens: tipos de bufferização

- **Null buffer** - não há espaço temporário para armazenar mensagem
  - Usado apenas para **comunicação síncrona**
  - estratégia 1:
    - mensagem permanece no espaço de endereçamento do processo *sender*, que fica bloqueado no *send()*
    - quando receptor faz *receive()*, uma mensagem de confirmação é enviada ao *sender*, e o *send()* pode enviar os dados
  - estratégia 2:
    - *sender* manda a mensagem e espera confirmação
    - se processo receptor não está em *receive()*, SO descarta mensagem
    - se processo receptor está em *receive()*, manda confirmação
    - se não há confirmação dentro de um tempo (*time-out*) é sinal de que receptor não estava em *receive()*, e a mensagem é repetida

Troca de mensagens: tipos de bufferização

- **Single message buffer**
  - Apenas para **comunicação síncrona**
  - Ideia:
    - mantem a mensagem pronta para uso no local de destino
    - mensagem é bufferizada no lado receptor, caso o processo receptor não esteja pronto para recepção

Troca de mensagens: tipos de bufferização

- **Unbounded-capacity buffer**
  - para **comunicação assíncrona**, *sender* não espera *receiver*
  - podem existir várias mensagens pendentes ainda não aceitas pelo *receiver*
  - para garantir entrega de todas mensagens, um buffer ilimitado é necessário
  - Impossível na prática, pois não existe memória infinita!

Troca de mensagens: tipos de bufferização

- **Finite-bound buffer** (bounded capacity buffer)
  - Unbounded capacity buffer é impossível na prática
  - Buffer finito com capacidade para múltiplas mensagens
  - Permite **comunicação assíncrona**
  - Necessário uma estratégia para evitar estouro de buffer (*buffer overflow)*
    - comunicação sem sucesso: *send()* retorna código de erro
    - comunicação com controle de fluxo: o transmissor fica bloqueado no *send()* até que o receptor aceite alguma(s) mensagem(s);
      - Introduz sincronização entre transmissor e receptor
      - *Send* assíncrono não é assíncrono para todas mensagens

Sincronização de processo via mensagens

- Podemos utilizar troca de mensagens bloqueantes para sincronizar processos concorrentes
- Mensagem pode bloquear o *sender*, o *receiver*, ou ambos
- Veremos dois exemplos de sincronização:
  - Sincronização de competição: problema do jantar dos filósofos
  - Sincronização de cooperação: problema produtor/consumir

Jantar dos filósofos com troca de mensagens

- Possível solução do jantar dos filósofos utilizando troca de mensagens bloqueantes
  - **Pedido de garfo:** Quando um filósofo quer comer, ele pede o garfo ao vizinho da esquerda e aguarda a resposta.
  - **Resposta do Vizinho:** O vizinho pode:
    - **Aceitar** o pedido e enviar a resposta, se o garfo estiver disponível.
    - **Negar** o pedido e enviar a resposta, se o garfo estiver em uso.
  - **Segundo pedido:** Se o primeiro garfo for concedido, o filósofo pede o garfo ao vizinho da direita e espera pela resposta.
  - **Avaliação:** Se conseguir ambos os garfos, o filósofo come. Se não conseguir ambos ou esperar muito pelo segundo, ele devolve o primeiro garfo e volta a pensar.
  - **Liberação:** Após comer, o filósofo avisa ambos os vizinhos que liberou os garfos.

Produto/consumidor com troca de mensagens

- Possível solução simples envolve criar um processo que gerencia o acesso ao buffer
  - **Processos produtores:**
    - Criam um item.
    - Enviam REQUEST_TO_PRODUCE para o Buffer.
    - Ao receber ALLOW_PRODUCTION, enviam o item. Caso contrário, esperam.
  - **Processos consumidores:**
    - Enviam REQUEST_TO_CONSUME para o Buffer.
    - Ao receber ALLOW_CONSUMPTION, esperam pelo item. Caso contrário, esperam.
  - **Processo buffer:**
    - Ao receber REQUEST_TO_PRODUCE e ter espaço, responde com ALLOW_PRODUCTION.
    - Ao receber REQUEST_TO_CONSUME e ter um item, responde com ALLOW_CONSUMPTION e envia o item.
    - Gerencia capacidade e comunica negações quando necessário.

Canais

- Um canal (*channel*) conecta um processo emissor com um processo receptor
- Canais são tipados, ou seja, precisamos declarar o tipo da mensagem que será enviada antes do uso
- Em sistemas operacionais canais são chamados de *pipes*
  - Exemplo: ls -l *.txt | grep May
- Notação usada no nosso material
  - Ch <= x, escreve x no canal (em Go, Ch <- x)
  - Ch => y, lê do canal para y (em Go, y <- Ch)

Canais: sincronização

- Canais podem ser **síncronos** ou **assíncronos** dependendo de como a linguagem implementa
  - Algumas oferecem várias opções de chamada e diferentes níveis de sincronismo (ex: MPI, Go)
- **Canais síncronos** (ou não-bufferizados):
  - Quando um valor é enviado para um canal síncrono, a operação de envio bloqueia até que alguém leia esse valor do canal
  - Da mesma forma, quando alguém tenta ler de um canal síncrono e ele está vazio, a leitura bloqueia até que um valor seja enviado ao canal
- **Canais assíncronos** (ou bufferizados):
  - Enviar valores para um canal bufferizado não bloqueia até que o buffer esteja cheio.
  - Ler de um canal bufferizado não bloqueia até que o buffer esteja vazio.

Produtor/consumidor com canais

- Também podemos utilizar canais para solucionar o problema de sincronismo de cooperação com produtores e consumidores
  - *Bounded buffer problem*
- Uma modelagem possível envolve usar um canal com um limite de tamanho como buffer, assim resolvemos vários problemas
  - Bloqueio no caso de buffer vazio
  - Bloquei no caso de buffer cheio
  - *Data-race* na escrita e leitura das posição do *buffer*

Algorithm 8.1: Producer-consumer (channels)	
channel of integer ch	
producer	**consumer**
integer x	integer y
loop forever	loop forever
p1: x ← produce	q1: ch ⇒ y
p2: ch ⇐ x	q2: consume(y)

Jantar dos filósofos com canais

- Uma modelagem possível dos filósofos com canais:
  - Um processo para cada filósofo e um processo e um canal para cada garfo
    - Cada processo garfo cuida do seu respectivo canal
  - Processos do tipo filósofo pegam (alocam) garfos lendo do canal do respectivo garfo e liberam escrevendo nele
  - Processos do tipo garfo escrevem e leem nos seus respectivos canais

Algorithm 8.5: Dining philosophers with channels	
channel of boolean forks[5]	
philosopher i	**fork i**
boolean dummy	boolean dummy
loop forever	loop forever
p1: think	q1: forks[i] ⇐ true
p2: forks[i] ⇒ dummy	q2: forks[i] ⇒ dummy
p3: forks[i+1] ⇒ dummy	q3:
p4: eat	q4:
p5: forks[i] ⇐ true	q5:
p6: forks[i+1] ⇐ true	q6:

Referências

- Ben-Ari, M. Principles of Concurrent and Distributed Programming, 2nd Edition,