Fundamentos de Processamento Paralelo e Distribuído

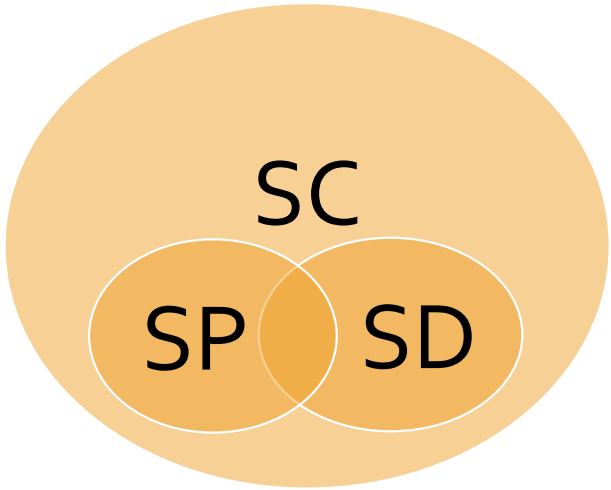
Comunicação e Sincronização entre Processos

Prof. César A. F. De Rose

Contextualização

Por que Concorrência

- Por que estudar concorrência?
- Sistemas Paralelos e Distribuídos são sistemas concorrentes
  - Processos que disputam recursos
  - Modelagem: responsividade, desempenho, corretude
- Sendo assim propriedades de concorrência serão herdadas por estas sistemas



Desafios

- Temos dois principais desafios na programação concorrente
  - Permitir a **comunicação** entre os processos
  - E a necessidade de **sincronizar** a sua execução
- Se pertencem ao mesmo programa, estes processo estão colaborando na resolução de um mesmo problema, de forma que precisam se comunicar
  - Como no exemplo da entrada e saída no SO
- E pelo mesmo motivo acima, vai ser necessário um trabalho articulado entre eles, ou seja sua sincronização
  - Processo que precisa uma tecla, só pode continuar depois que ela for lida
- **Spoiler: é muito difícil implementar comunicação e sincronização nestes sistemas de forma eficiente e segura!**
  - Quando algo "congela" se trata normalmente de um erro de comunicação e/ou sincronização

Desafios

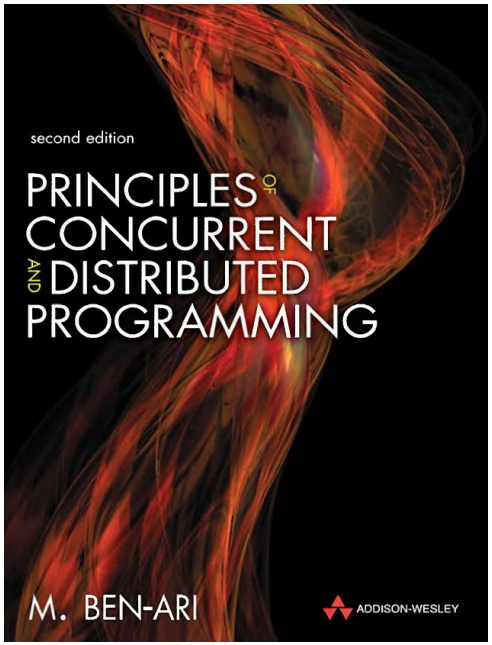
- Perda de controle que pode comprometer o resultado
  - Perda em relação ao acesso aos recursos e/ou a dados compartilhados
  - Perda em relação ao controle de fluxo

Desafios

- Objetivo desta parte do conteúdo é:
- Apresentar mecanismos, padrões, algoritmos e sistemas que são usados para se obter o funcionamento correto de programas concorrentes
  - Evitar problemas de comunicação e sincronização vistos no slide anterior
  - Ou seja, que execute **sempre** corretamente!
- **A escolha de qual mecanismo usar vai ficar a critério do desenvolvedor, dependendo dos requisitos do sistema e da arquitetura em que executa**

Roteiro

- Tipos de Comunicação
  - Compartilhamento de memória
  - Troca de mensagens
- Tipos de Sincronização
  - De competição
    - Seção crítica
    - Filósofos
  - De cooperação
    - I/O
    - Condição de parada
    - Produtor consumidor
- Mecanismos
  - Mensagens/Canais
  - Semáforos
  - Monitores



Tipos de Comunicação

Tipos de Comunicação

- Como vimos, um dos desafios na programação concorrente é permitir a **comunicação** entre os processos do programa
- Se colaboram em alguma coisa precisam em algum momento se comunicar
- Comunico, ou seja **troco informações**, basicamente de duas formas
  - Através de uma memória compartilhada
  - Através de troca de mensagens
- Esta escolha depende normalmente do hardware que tenho disponível

Compartilhamento de Memória

- Comunico, ou seja troco informações, através da memória
  - Precisa estar **compartilhada** entre os envolvidos
  - Mesmo **espaço de endereçamento** entre as partes
    - Ex: endereço 100 é o mesmo para os dois!
- Depende de uma característica do hardware
  - Máquina **multiprocessada**
- Como o conteúdo está onde todos conseguem "ver", não preciso movimentá-lo, só passar uma **referência**
  - Variáveis globais
  - Índices de um vetor ou matriz
  - Ponteiros
- Em função disto é normalmente mais eficiente que trocar mensagens

Troca de Mensagens

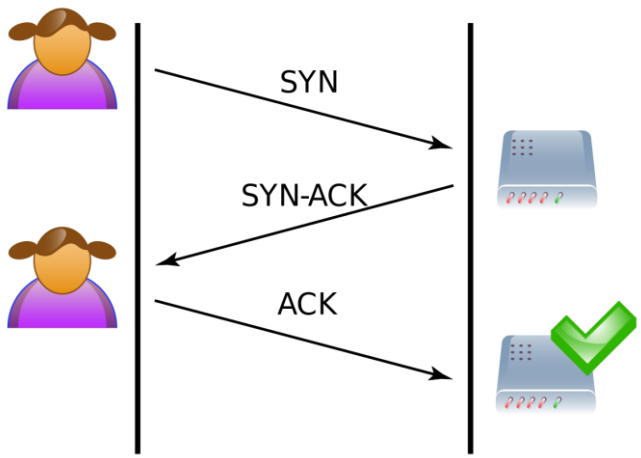
- Comunico, ou seja troco informações, através da uma mensagem
  - Não tenho uma memória **compartilhada** entre os envolvidos
    - Ou seja, não consigo acessar a memória do meu interlocutor
  - **Espaços de endereçamento** diferentes entre as partes
    - Ex: o endereço 100 NÃO é o mesmo para os dois!
- Consequência de uma característica do hardware
  - Máquina **multicomputadora**
- Como não consigo passar uma referência, preciso movimentar o conteúdo todo
- Mensagem usada como recurso para escrever na memória do outro
  - Tipo de dados canal
  - Biblioteca de troca de mensagens
- Em função disto é normalmente menos eficiente que compartilhar memória

Como escolher

- Em uma máquina sem memória compartilhada só posso trocar mensagens
  - Ex: multicomputadador/cluster
- Se tenho memória compartilhada posso optar por qualquer um dos dois paradigmas
  - Compartilhar memória é normalmente mais eficiente nestas máquinas
  - Mas em algumas linguagens modernas pode ser interessante trocar mensagens pela facilidade de modelagem
    - Ex: canais em Go
- Se uso memória compartilhada posso ter **data races** e preciso tratar o problema com seções críticas
- Se uso mensagens posso ter mais facilmente bloquear
  - Se ficar esperando por uma mensagem que nunca vem

Também Sincronizam

- Comunicação não transfere apenas dados entre as partes, mas também acaba sincronizando quando feita de forma síncrona!
  - Espera de um valor para uso imediato
    - Leitura de tecla ou opção de um menu
  - Troca de valores ou confirmação de recebimento
    - Aperto de mão (*handshake*)



Tipos de Sincronização

Tipos de Sincronização

- Como vimos, um dos desafios na programação concorrente é a necessidade de **sincronização** entre os processos do programa
- Se colaboram em alguma coisa precisam em algum momento se coordenar na resolução deste problema
  - Esperar que alguém termine de fazer alguma coisa que eu preciso
  - Organizar a disputa por um recurso
  - Organizar a disputa por dados compartilhados
- Esta coordenação tem dois tipos
  - Sincronização de competição
  - Sincronização de cooperação
- Esta escolha depende do **problema** que tenho que resolver

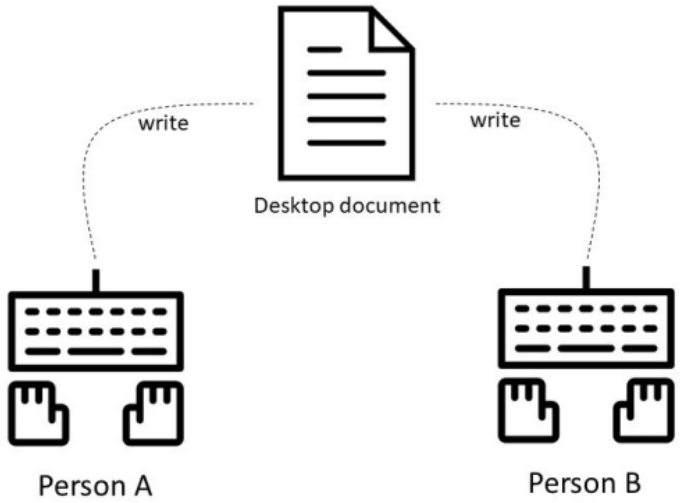
De Competição

- A **sincronização de competição** se faz necessária quando dois ou mais processos **disputam** alguma memória ou recurso compartilhados
- A coordenação nestes compartilhamentos é fundamental para que o programa funcione corretamente
- Veremos dois exemplos de aplicação
  - Data race (memória)
  - Filósofos Jantando (recursos)

Data race

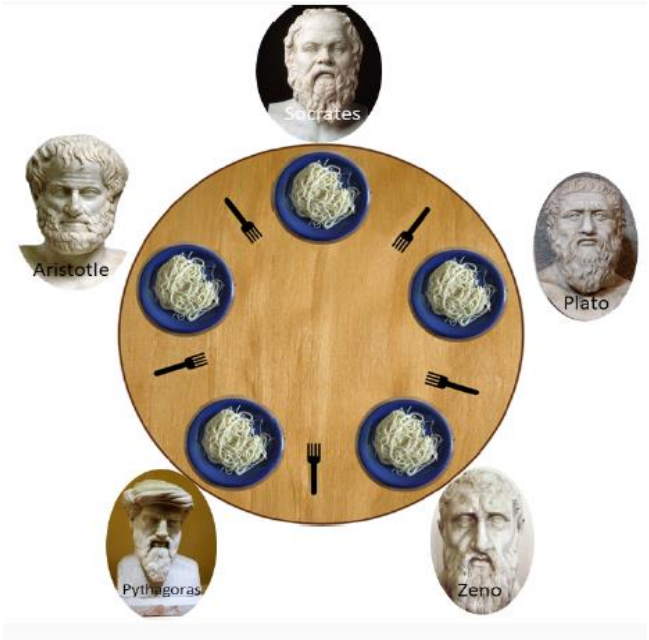
- Sincronização de competição que envolve a aplicação de técnicas de **exclusão mútua** para evitar que os programas atualizem uma variável compartilhada sem proteção
- Resulta na execução desta parte do código – seção crítica - de forma **atômica** garantindo assim o resultado correto mesmo com concorrência

[executar exemplo v3.go com e sem as linhas 21 e 25]



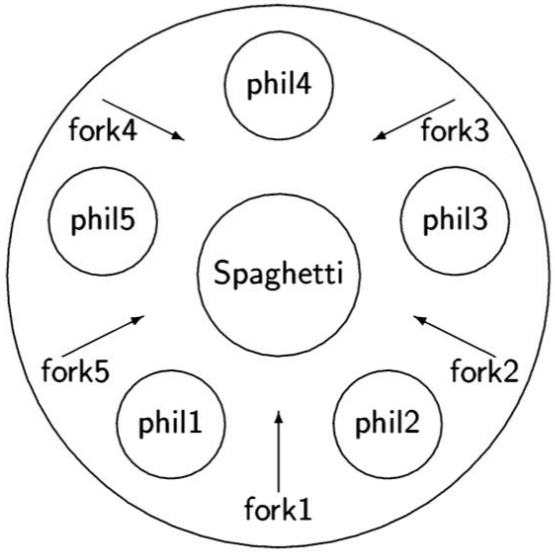
Filósofos Jantando

- Sincronização de competição que envolve a aplicação de técnicas de **exclusão mútua** para evitar que os programas se bloqueiem mutuamente – **deadlock** – no acesso a recursos compartilhados
- Resulta na execução desta parte do código - seção crítica - de forma **atômica** garantindo assim o resultado correto mesmo com concorrência
- Problema clássico de sincronização introduzido por Dijkstra



Filósofos Jantando

- Cinco filósofos estão sentados ao redor de uma mesa de jantar, com um garfo entre dois filósofos adjacentes
- Cada filósofo alterna entre pensar (*non-critical section*) e comer (*critical section*)
  - Para conseguir comer um filósofo precisa de dois garfos, o da esquerda e o da direta
- Solução deve garantir que não ocorra ***deadlock*** e ***starvation*** mantendo um bom grau de concorrência

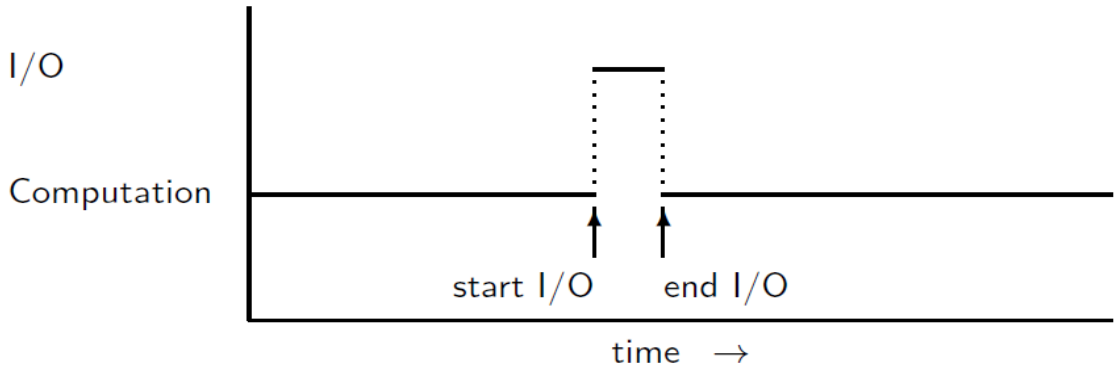


De Cooperação

- A **sincronização de cooperação** se faz 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**
- Veremos três exemplos de aplicação
  - Operação de entrada e saída (I/O)
  - Condição de parada
  - Produtor Consumidor

Operação de entrada e saída I/O

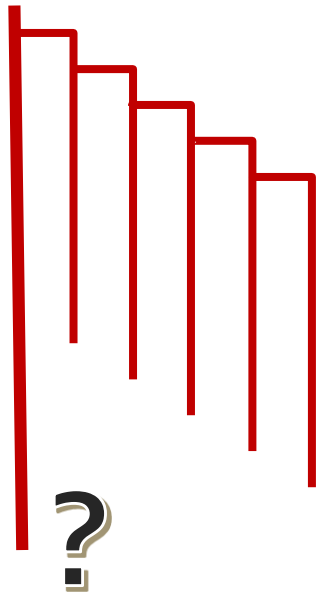
- Sincronização de cooperação que envolve a coordenação de dois ou mais processos que estão realizando uma operação de entrada e saída (I/O) em um sistema operacional
- Resulta na parada do processo que pediu a operação de I/O até que ela seja realizada pelo processo de atendimento do SO garantindo assim o resultado correto mesmo com concorrência
  - Ex: leitura de uma string do teclado



Condição de Parada

- Sincronização de cooperação que envolve a coordenação de dois ou mais processos que estão executando de forma concorrente para que o programa termine graciosamente
- Resulta na espera de um processo "pai" até que todos os processos filhos terminem de executar
  - **Problema da condição de parada** em função da perda do controle de fluxo

[executar exemplo v5.go]

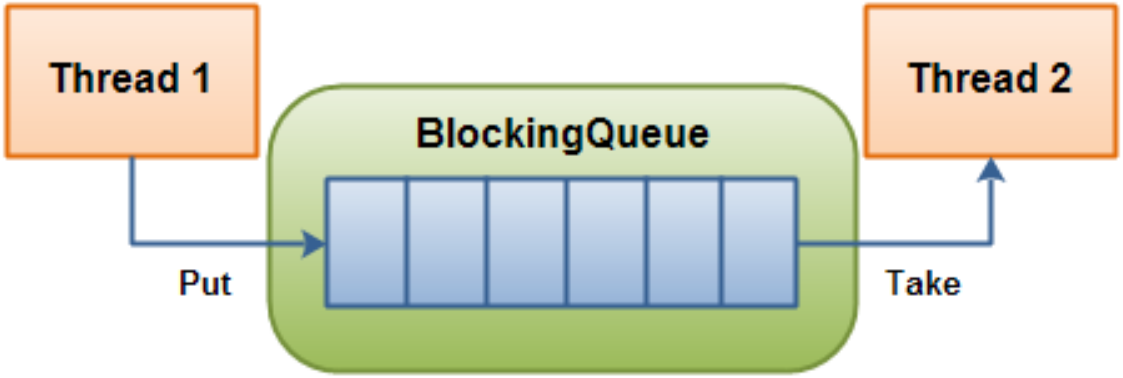


Produtor Consumidor

- Sincronização de cooperação que envolve a coordenação de dois ou mais processos que estão executando de forma concorrente para que uns consumam o que os outros produziram
- Resulta no **desacoplamento** entre produtores e consumidores aumentando a concorrência e permitindo paralelismo
  - Ex: múltiplos consumidores

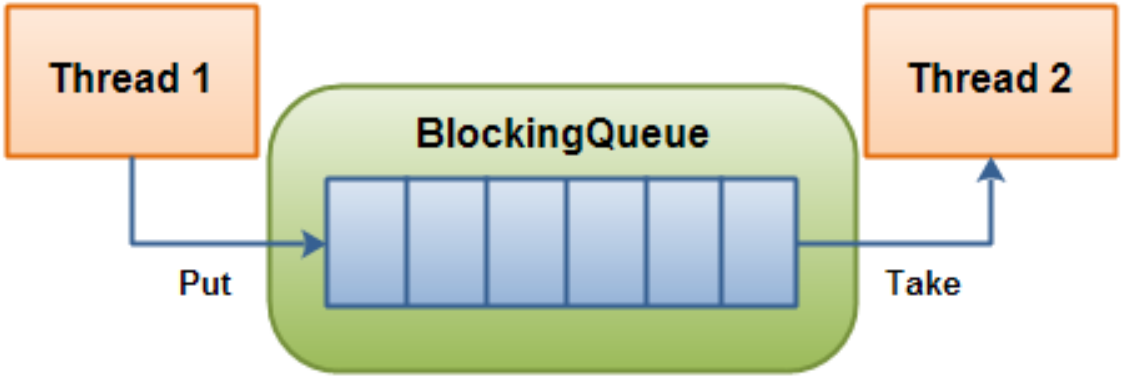
[executar exemplo v6.go]

- 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



Produtor Consumidor

- Também conhecido como ***bounded buffer problem***
  - Buffer acaba tendo um tamanho finito
  - Produtores tem que bloquear se o buffer estiver cheio
  - Consumidores tem que bloquear se buffer estiver vazio
- Posso ter o problema com vários produtores e vários consumidores
- Neste caso tenho que cuidar também que (**data race,** que seria um problema de competição):
  - Produtores não escrevam na mesma posição
  - Consumidores não leiam da mesma posição



Mecanismos

Mecanismos

- Agora que já vimos situações que podem apresentar problemas em programas concorrentes, veremos os mecanismos (*constructs*) que podem ser usados para evitar estes problemas
- Cada um tem caraterísticas próprias que podem resultar em vantagens e desvantagens dependendo do caso
  - Necessidade de memória compartilhada
  - Nível de abstração
- Cabe ao programador identificar qual o melhor mecanismo a ser usado em cada caso
  - Pode depender da disponibilidade na linguagem usada
- Vamos ver
  - Mensagens/Canais
  - Semáforos
    - *Mutex/Atomic/Critical*
  - Monitores
  - Barreiras

Mensagens/Canais

Dois em um

- Como já vimos processos podem se comunicar enviando e recebendo mensagens uns dos outros
- E que estas mensagens também acabam sincronizado estes processos
- Ou seja, podemos ter mecanismos de sincronização baseados em comunicação
  - Prático, um único mecanismo pode ser usado para resolver as duas questões (comunicação e sincronização)
  - Funciona sem memória compartilhada
  - Mas em função da praticidade em algumas linguagens modernas são usados mesmo em ambientes de memória compartilhada
    - Ex: Go possui o tipo canal
    - Já vimos vários exemplos com canais

Mensagens Síncronas

- Comunicação requer dois processos, um que envia a mensagem e outro que recebe
- Precisamos definir o grau de cooperação entre os dois processos
- Na comunicação **síncrona**, a troca de uma mensagem é uma operação **atômica** que requer a participação do emissor e do receptor
  - Se o recebedor não estiver pronto pra receber o emissor é bloqueado
  - E se o recebedor por sua vez estiver pronto pra receber antes que o emissor estiver pronto pra enviar, o recebedor é bloqueado
  - Ou seja, a ação de enviar uma mensagem sincroniza o fluxo de execução das duas partes neste modo
- Apesar de ser mais fácil de usar precisamos tomar cuidado
  - Para não matar a concorrência
  - Para não gerar deadlock

Rendezvous – encontro

- Conceito usado neste contexto
- Remete a imagem de duas pessoas que marcam um encontro em um local determina
  - O primeiro que chegar precisa esperar pelo segundo
- Por que isto nos interessa: **sincronimo**!
  - Em go um canal que não indico capacidade é síncrono (capacidade = 0)
- Na metáfora a relação é simétrica e as duas pessoas se encontram em um lugar neutro
- No entanto, quando usamos diretivas de sincronismo normalmente o local do encontro pertence a um dos processos, o que aceita o convite
- O processo que faz o convite precisa conhecer a identidade do processo que vai convidar e a identidade do local do encontro
  - Também chamado de ponto de entrada (*entry)*
- O processo que aceita o convite não precisa necessariamente saber a identidade de quem convida
  - Por isto que *rendezvous* é apropriado para a implementação de servidores que exportam seus serviços para potencias clientes

Mensagens Assíncronas

- Mas também podemos fazer uma comunicação **assíncrona**, onde emissor e recebedor não ficam bloqueados
- Desta forma não teremos um sincronismo entre os fluxos das duas partes
  - Emissor envia mensagem sem ficar bloqueado
  - Receptor pode estar fazendo outra coisa no momento do envio e só mais tarde verificar se chegou uma mensagem (e neste momento bloquear)
- A escolha do modelo de comunicação depende da capacidade do canal de comunicação de armazenar mensagens (*buffer*), pois se não foram lidas imediatamente (síncrono) tem que ficar armazenadas até uma leitura posterior (assíncrono)
  - Se acontecerem várias escritas antes de uma leitura posso precisar de um buffer que comporte várias mensagens
  - Como um buffer tem tamanho finito, eventualmente vou ter que bloquear quando estiver cheio, ou corro o risco de perder mensagens

Endereçamento

- Para iniciar uma chamada telefônica, quem liga precisa indicar um número de com quem quer falar, o "endereço" do destinatário
  - O endereçamento neste caso é **assimétrico**, pois o destinatário recebe a ligação sem necessariamente saber quem ligou
- Em uma mensagem de e-mail, o endereçamento é **simétrico** pois a mensagem contem o endereço do destino e do emissor
- Endereçamento simétrico é preferível quando vários processos estão colaborando na resolução de um problema
  - Endereçamento pode ser resolvido na compilação resultando em uma programação mais fácil e uma execução mais eficiente
  - Em alguns casos o canal de comunicação recebe um nome usado pelos dois lados
    - Ex: linguagem Go
  - Outra opção é usar o ID dos processos (PID)
    - Ex: MPI
- Comunicação assimétrica é preferível quando se programa serviços ou servidores
  - Cliente só indica o nome do serviço, o que dá mais flexibilidade de implementação para o servidor (ou servidores)

Sentido da comunicação

- Uma comunicação pode ter dados fluindo em um único sentido (***one-way***) ou nos dois sentidos (***two-way***)
  - e-mail (unidirecional)
  - chamada telefônica (bidirecional)
- Comunicação assíncrona é necessariamente em um único sentido
- Na comunicação síncrona as duas possibilidades são possíveis
- Enviar uma mensagem em sentido único é extremamente eficiente pois o emissor não fica bloqueado enquanto o receptor processa a mensagem
- Se no entanto o emissor precisa de uma confirmação do envio, talvez seja melhor ficar bloqueado do que ter que fazer depois outra operação síncrona

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
- 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
- Em sistemas operacionais canais são chamados de *pipes* e permitem que se conecte a saída de um programa na entrada de outro
  - Ex: `$ ls -l *.ps | grep May`
- Notação usada no nosso material
  - Ch <= x escrevo x no canal
  - Ch => y leio do canal para y

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)

Sincronizando com canais

- Veremos exemplos dos diferentes tipos de sincronização usando mensagens/canais
- Tipos de Sincronização
  - De competição
    - Seção crítica
    - Filósofos
  - De cooperação
    - Entrada e Saída (I/O)
    - Condição de parada
    - Produtor consumidor
- Veremos um exemplo de comunicação em um pipeline

Seção Crítica

- Aplicaremos canais para solucionar o problema de sincronismo de competição com seção crítica em uma **data race**
- Uma modelagem possível envolve transferir a responsabilidade de atualizar uma varável compartilhada para apenas um processo
- Assim encapsulamos a variável em questão que deixa de ser compartilhada
- Os outros processos enviam a este processo a operação por mensagem

[executar exemplo v31.go]

Filósofos

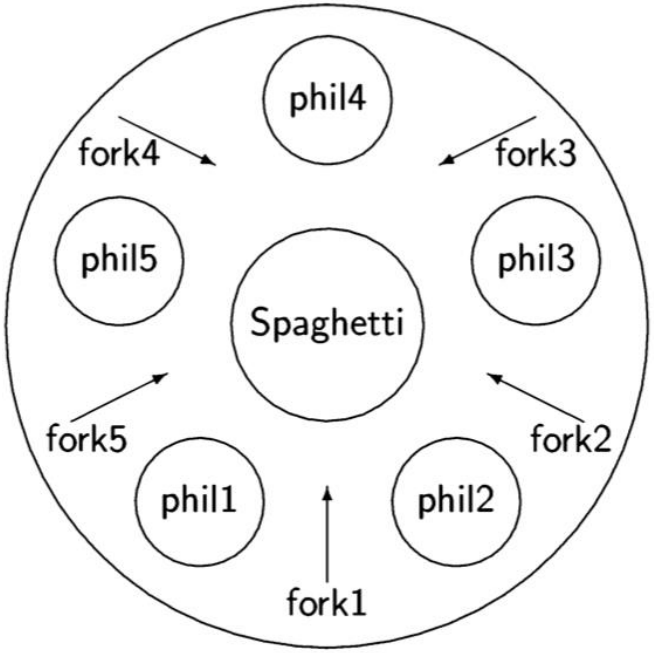
- Aplicaremos canais para solucionar o problema de sincronismo de competição dos filósofos
- Uma modelagem possível envolve ter um processo para cada filósofo e um processo e um canal para cada garfo
  - Cada processo garfo cuidando do seu respectivo canal
- Processos do tipo filósofo pegam (alocam) garfos lendo do canal do respectivo garfo e liberam escrevendo nele
  - ```
    Loop forever
        Fork[i] => dummy
        eat
        Fork[i] <= true
    ```
- Processos do tipo garfo escrevem e leem nos seus respectivos canais
  - ```
    Loop forever
        Fork[i] <= true
        Fork[i] => dummy
    ```

Filósofos

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:	



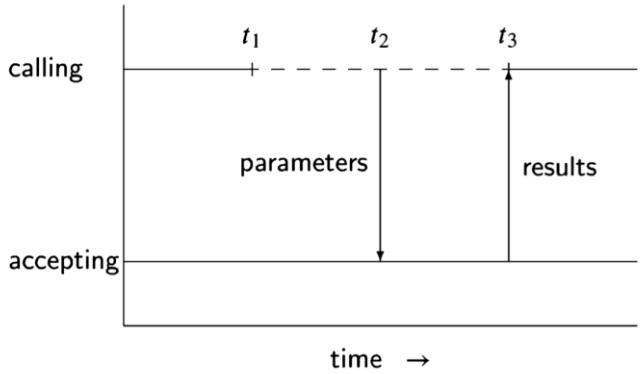
Entrada e Saída I/O

- Aplicaremos canais para solucionar o problema de sincronismo de cooperação na entrada e saída
- Uma modelagem possível utiliza canais síncronos para chamar o serviço que desejo do SO, passar os parâmetros e retornar o resultado da operação
- Assim garanto que o processo principal aguarde o término da chamada ao sistema para continuar

[executar exemplo v7.go]

Algorithm 8.6: Rendezvous

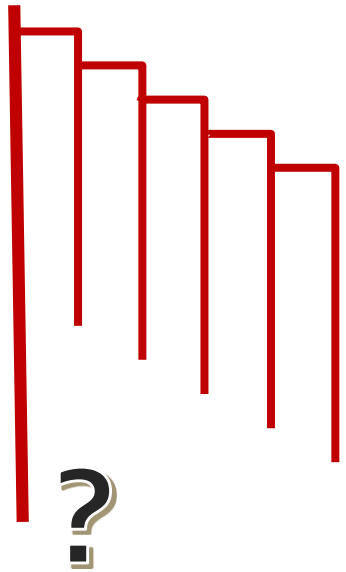
	client		server
	integer parm, result		integer p, r
	loop forever		loop forever
p1:	parm ← . . .	q1:	
p2:	server.service(parm, result)	q2:	accept service(p, r)
p3:	use(result)	q3:	r ← do the service(p)



Condição de parada

- Aplicaremos canais para solucionar o problema de sincronismo de cooperação da condição de parada
- Uma modelagem possível utiliza canais para avisar o processo principal que as sub-rotinas concorrentes terminaram
- Assim garanto que o processo principal aguarde o término dos sub-programas e o programa termine graciosamente

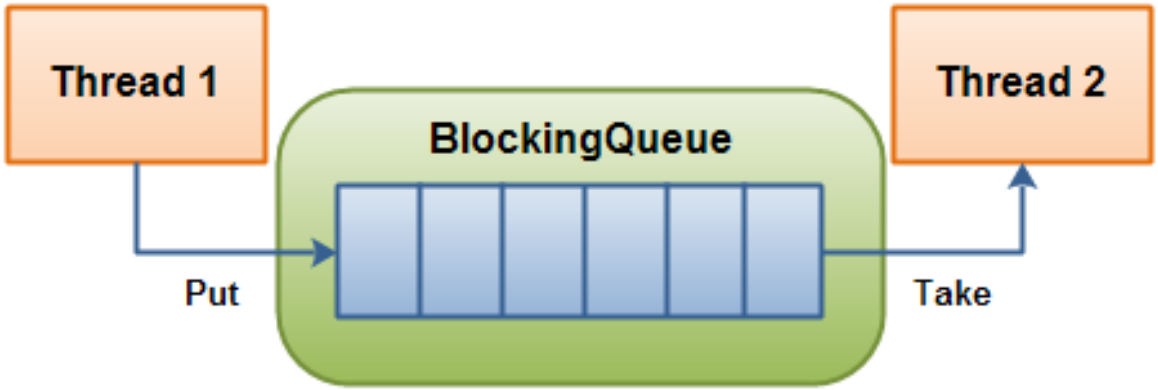
[executar exemplo v5.go]



Produtor Consumidor

- Aplicaremos 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

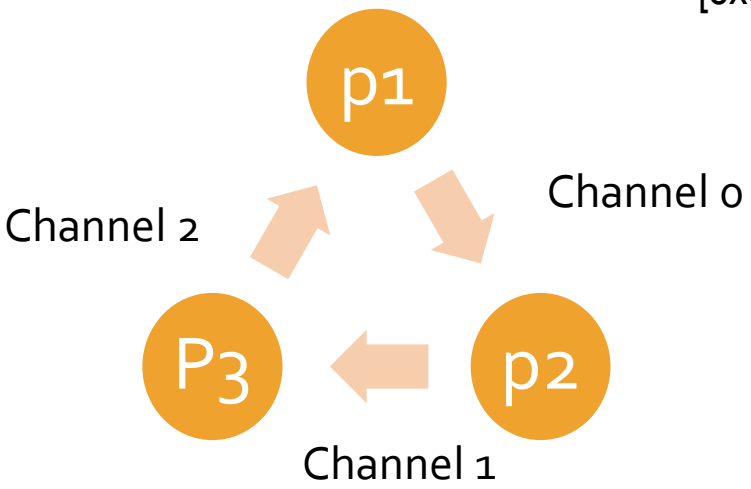
[executar exemplo v61.go]



Pipeline

- Aplicaremos canais para implementar a comunicação em um pipeline de processos
- Cada processo do pipeline recebe de um canal de entrada, processa o dado recebido e escreve em um canal de saída
- Desta forma realizamos a articulação a comunicação entre os processos
- Pode ser usado para implementar **exclusão mútua**
  - Técnica de passagem de bastão
  - Mensagem é o bastão

[executar exemplo v8.go]



Semáforos

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
  - O contador controla quantos processos podem entrar na seção crítica (inteiro não negativo >=0)
  - A fila de pids quais os processo que estão esperando para entrar
- Implementa duas operações atômicas
  - Wait – original P() do holandês *Probeer* (tentar)
    - Decrementa o contador em um se for maior que zero e continua executando. Se for zero bloqueia e coloca seu pid na lista de Pids do semáforo
  - Signal - V() do holandês *Verhoog* (incrementar)
    - Se fila está vazia, incrementa o contador. Se fila não está vazia, desbloqueia um destes processos (escolha arbitrária)

Semáforos

- Dois tipos de semáforo
  - Binário: só pode ter os valores 1 e 0
    - Chamado em algumas linguagens de *mutex*
    - Inicializado com o valor 1 (pois só um entra de cada vez)
  - Geral: pode ter qualquer valor não negativo
    - É inicializado com o valor de quantos processos queremos que possam entrar juntos na seção crítica
- Ambos podem ser **fortes** ou **fracos**
  - Fracos: quando um processo é liberado é escolhido **arbitrariamente** da fila de bloqueados
    - Mais concorrente mas pode gerar *starvation*
  - Fortes: quando um processo é liberado é escolhido o **primeiro** da fila de bloqueados
    - Menos concorrente mas evita *starvation*

Semáforos Implementação

- Como para gerencia o contador de um semáforo preciso ler e incrementar, posso ter problemas se isto não for feito de forma atômica
  - **Data race**!
- Como fazer para implementar se a solução do problema de data race também é sensível a data race?
- Será visto em detalhes em Sistema Operacionais, mas a solução mais comum usa uma instrução em linguagem de máquina que testa e incrementa de forma atômica
  - `Test_and _set (lock)`

Seção Crítica com Semáforos

- Aplicaremos semáforos para solucionar o problema de sincronismo de competição uma **data race**
- Uma modelagem possível envolve só deixar um processo por vez ler e atualizar o valor da variável compartilhada
  - O mesmo que fizemos em todos os exemplos quando imprimimos na tela
- Ou seja, tratar esta operação como atômica, não deixando que outros processos a interrompam por entrelaçamento
  - Exclusão mútua
- Assim usamos um semáforo binário para tratar estas instruções como seção critica
  - chamando P()/W()no início do trecho
  - e V()/S() quando terminarmos a atualização

Algorithm 6.1: Critical section with semaphores (two processes)	
binary semaphore S $\leftarrow (1, \emptyset)$	
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)

Seção Crítica com Semáforos em Go

- Não temos uma implementação completa de semáforos em Go, pois a ideia é sincronizar por canais
- Go implementa exclusão mútua com um tipo mutex e dois métodos
  - Mutex.Lock()
  - Mutex.Unlock()
- Isto seria equivalente a um semáforo binário e só permite que um processo entre na seção crítica

[executar exemplo v3.go]

Filósofos com Semáforos

- Aplicaremos semáforos para solucionar o problema de sincronismo de competição dos filósofos
- Uma modelagem possível envolve ter um processo para cada filósofo e um semáforo para cada garfo
  - Cada semáforo garfo cuidando do seu respectivo garfo
- Processos do tipo filósofo pegam (alocam) garfos fazendo Wait() no respectivo semáforo e liberam fazendo Signal () nele

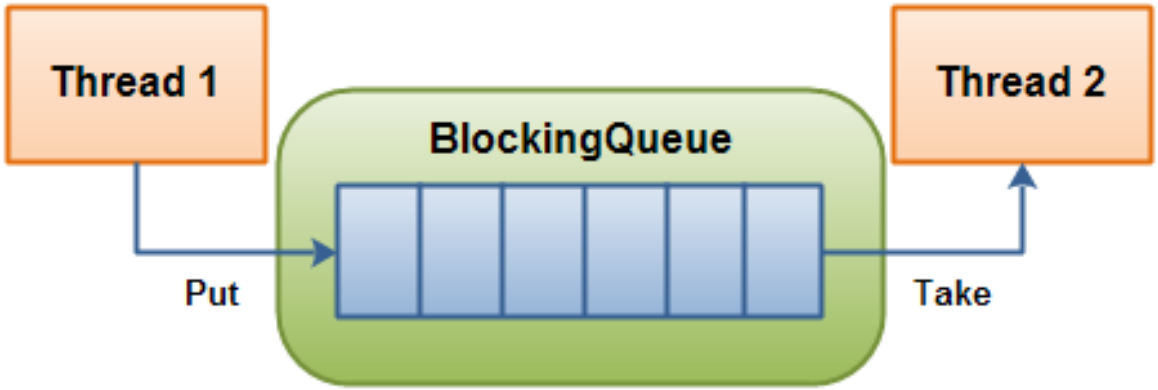
Algorithm 6.10: Dining philosophers (first attempt)

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

- Aplicaremos semáforos para solucionar o problema de sincronismo de cooperação com produtores e consumidores
  - ***Bounded buffer problem***
- Uma modelagem possível envolve usar dois semáforos, um binário para o controle do buffer vazio e um geral para controle do buffer cheio
  - Inicializamos o geral com a capacidade do buffer
- No caso de vários processos produtores e consumidores precisamos mais dois semáforos para proteger *data-race* na escrita e leitura das posição do buffer

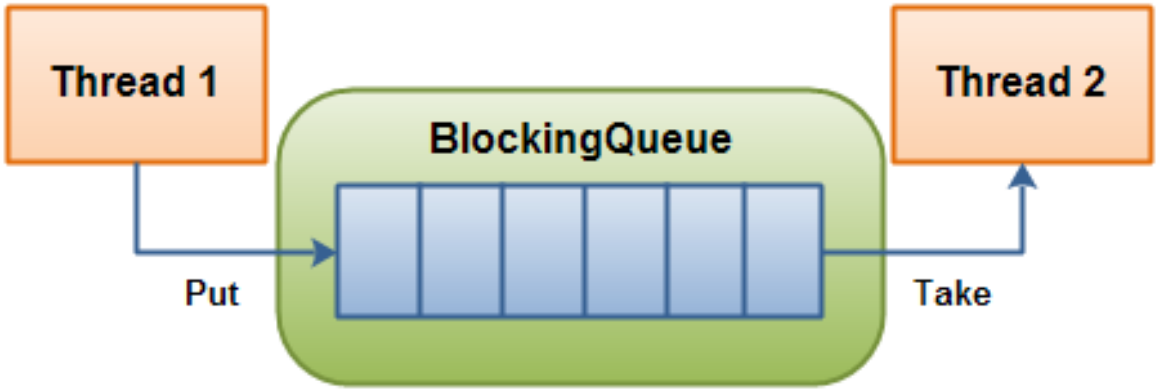


Produtor Consumidor com Semáforos

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)



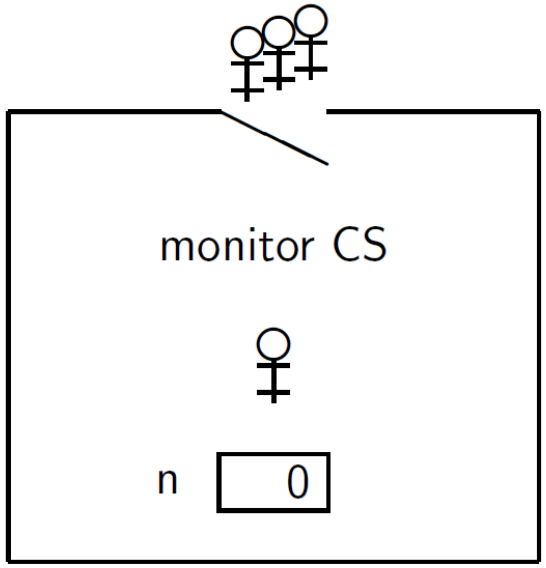
Monitor

Monitores

- Canais e Semáforos são mecanismos de sincronização que, como vimos, podem ser usados para resolver os potenciais problemas de programas concorrentes
- Mas são considerados mecanismos de baixo nível por não serem estruturados
- Se formos construir um sistema grande, a responsabilidade do uso correto destas primitivas fica espalhada pelos processos que compõem o sistema
  - Se apenas um deles não usar o mecanismo corretamente poderemos ter uma falha difícil de diagnosticar
- Monitores propiciam um mecanismo estruturado de sincronismo que concentra a responsabilidade de corretude em um único lugar
  - Um monitor para cada objeto ou conjunto de objetos que precise de sincronismo

Monitores

- Ideia: encapsular o dado compartilhado e suas operações para restringir o acesso
  - Todos os acesso são gerenciados pelo monitor
  - Só um processo é liberado para operar o dado de cada vez
  - Se o monitor estiver ocupado, outros processos que desejam acesso esperam em uma fila
- 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



Dúvidas?