Fundamentos de Processamento Paralelo e Distribuído

Concorrência = vêm de coexistência , correr junto, enfatiza a coexistência e não a competição. Se algo é concorrente, algo está acontecendo ao mesmo tempo

fenômenos emergentes do uso de concorrência: deadlock, livelock, starvation, interleaving

sistema concorrente = mais de uma atividade está ativa no sistema

elementos básicos de sistemas concorrentes: (São os elementos fundamentais para construir todos os sistemas concorrentes existentes)

- processos = conjunto de processos sequenciais
- sincronização: canais ou memória compartilhada

**Control pointer de um processo

comportamento de um sistema concorrente aborda o conjunto de todas as possíveis combinações de cada um dos processos

starvation = um determinado processo não consegue competir pq os outros não deixam
deadlock = processos esperam mutuamente pera liberação de recursos e não progridem
livelock = processos continuam executando mas não progridem

Conceitos básicos de concorrência

não se pode supor nada sobre a velocidade relativa dos processos em concorrência (Mantra)

escalonador que decide qual thread que vai rodar

- **computação** = descreve uma possível execução de um programa concorrente (caminho dentro do diagrama de estados)
- **Interleaving** = entrelaçamento de comandos atômicos de programa
- **comportamento** = conjunto de todas as computações de um programa

  Semântica de interleaving = forma de representar a execução de um sistema concorrente, mais utilizada na literatura

ciclos = no diagrama representam a possibilidade de computações infinitas

estados sem arestas representam situações de bloqueio ou terminação

considerar interleaving arbritário RETIRA QUALQUER SUPOSIÇÃO TEMPORAL ao ambiente de execução dos processos

Apesar de considerar interleaving arbritário, temos que fazer uma restrição, justiça

Justiça = ñ faz sentido supor a possibilidade de os comandos de um processo NUNCA SEREM SELECIONADOS PARA EXECUÇÃO

Computação é justa se um comando que está continuamente habilitado nela acaba por ser executado e, não é deixado para trás

ACHEI PIKA (é óbvio, mas é uma formalização do q ja sabemos): quando uma função acaba, todas as rotinas que foram criadas dentro do escopo dessa função acabam também, independentemente se as rotinas acabaram ou não, é o lance da main acabar e as go routines não terminarem

Escalonador que cuida da justiça FRACA, garantido que um processo repetidas vezes, receba o direito de executar

Justiça forte é garantida por mecanismos de sincronização

- Justiça Fraca: o programa abaixo acaba ?

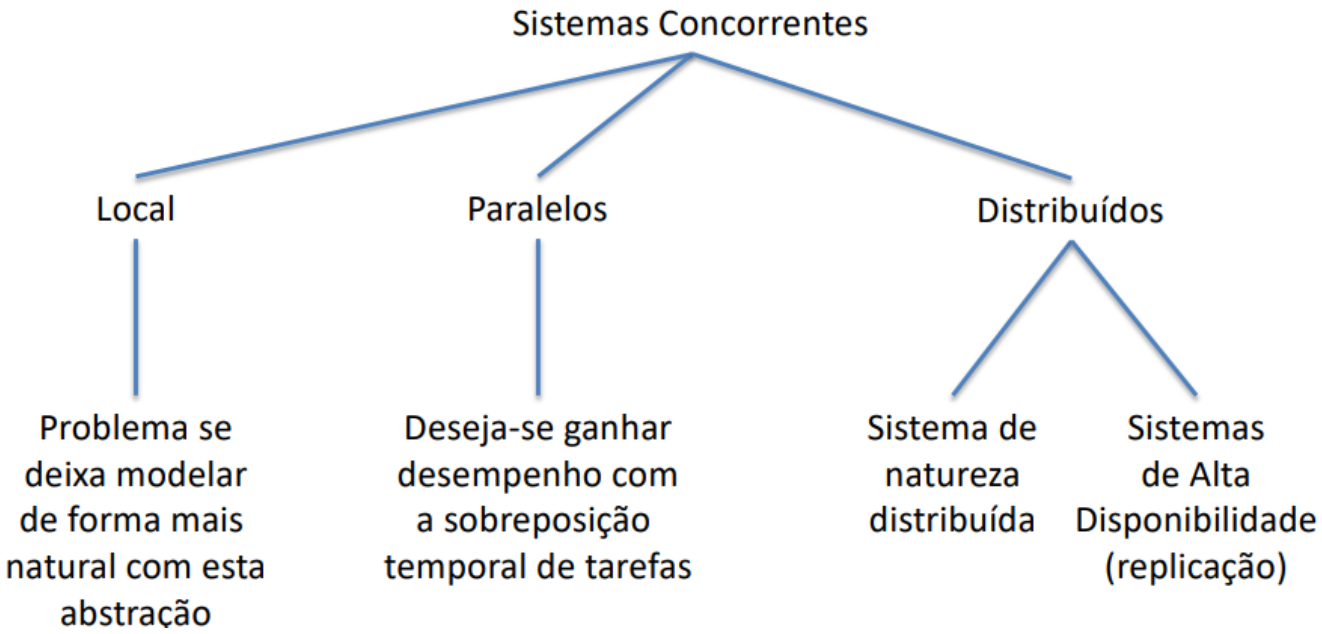
Algorithm: Stop the loop A	
integer n ← 0 boolean flag ← false	
p	**q**
p1: while flag = false p2: n ← 1 − n	q1: flag ← true q2:

- Justiça Forte: o programa abaixo acaba ?

Algorithm: Stop the loop A	
integer n ← 0 boolean flag ← false	
p	**q**
p1: while flag = false p2: n ← 1 − n	q1: await (n==1) q2: flag ← true

Relembrando

- Justiça Fraca:
  - se um processo está continuamente habilitado a prosseguir, ele progredirá
- Justiça Forte:
  - se um processo continuamente se torna habilitado a prosseguir, ele progredirá



Canais (troca de mensagens)

- tipados
- escritos e lidos por processos
- semântica FIFO / First In First Out
- Síncronos ou Assíncronos

Canal Síncrono = Leitura e escrita acontecem ao mesmo tempo

Canal Assíncrono (Bufferizado) = qualquer canal com tamanho maior que zero, channel com tamanho 1, suporta dois itens na fila

Não determinismo

Deadlock, Livelock & Starvation

dead = sem atividade ;
lock = bloqueio

processos formam uma cadeia circular de espera por recursos

Condições de Coffman para deadlock

para ter deadlock é necessário que as 4 condições ocorram simultaneamente, eliminando uma das condições, já não ocorre deadlock

- hold and wait = processo segura um recurso e entra em espera por outro
- recursos não são preemtáveis = recursos não são retiráveis de processos
- exclusão mútua = processos não usam um recurso ao mesmo tempo
- formação de ciclo de espera

**Jantar dos filósofos

Starvation = processo fica em estado de espera por recurso indefinidamente

Livelock = programa roda, mais não progride

Memória Compartilhada

processos vão ter variáveis em memória que vão ser comuns a eles

condição de corrida

O problema da sessão crítica

região onde atualizam os dados compartilhados

deve ser feita de forma mutuamente exclusiva

sessão crítica deve prover:

- exclusão mutua = só um processo por vez na sessão crítica
- espera limitada (não postergação) = thread não é indefinidamente passada para trás
- não bloqueio = processos de fora da sessão crítica não podem bloquear outros processos

- progresso =

Atômica = indivisível

Garantir o nível de atomicidade na sessão crítica

// Protocolo de entrada da sessão crítica
// Sessão Crítica
// Protocolo de saída da sessão crítica

Abordagens para solucionar o problema da sessão crítica

- Soluções de SW:
  2 ou n processos
  Algoritmo de Peterson = utilizar 3 flags, apenas 2 processos
  N processos = Algoritmo de Bakery, Algoritmo do Lamport
  Espera ocupada (busy wait)
- Suporte de HW
  Test and Set
  Exchange ou Swap
  São aplicáveis a qualquer número de processos
  Espera ocupada é empregada (busy wait)
- Semáforos
- Monitores

Semáforos

abstração onde uma linha de execução pode ser bloqueada sem consumir CPU

implementa uma espera **BLOQUEADA**

possui:

- V = valor inteiro não negativo = n° de vezes que wait( ) pode ser chamado sem bloquear
- L = Lista de processos administrados como fila

duas operações:

- wait( ) = decrementa o valor do semáforo ou se bloqueia
- signal( ) = incrementa o semáforo ou se desbloqueia

Semáforo é um canal. que pode ou não ter um determinado tamanho

```
type Semaphore struct {
        sChan chan struct {}
}

func newSemaphore(int init){
        s := &Semaphore(
                sChan: make(chan struct{} , init)
        )
        return s
}

func (s *Semaphore) Wait(){
        s.sChan <- struct{}{}
}

func (s *Semaphore) Signal(){
        <-s.sChan
}
```

pode-se construir canais com semáforos, ou semáforos com canais

PROBLEMA
leitores e escritores

readers = podem ler concorrentemente, pois não mudam o recurso
writers = tem q escrever no recurso apenas quando ninguém estiver lendo

Barreiras

mecanismo de sincronização que se pode construir com semáforos

implementa um ponto de encontro entre processos

se eu implemento uma barreira para 10 processo e, um já chegou na barreira e já pode seguir, ele deve esperar os outros 9 chegarem na barreira também poderem prosseguir

processos devem sincronizar em um ponto para depois prosseguirem

PONTO CRÍTICO != SESSÃO CRÍTICA

```
P 1 () {
    // computação.
    b.arrive()
    // PONTO CRÍTICO
    }
}
```

```
P 2 () {
    // computação .
    b.arrive()
    // PONTO CRÍTICO
    }
}
```

...

```
P N () {
    // computação .
    b.arrive()
    // PONTO CRÍTICO
    }
}
```

ponto onde se tem uma determinada restrição de sincronização

Listing 3.3: Barrier hint

```
1  n = the number of threads
2  count = 0
3  mutex = Semaphore(1)
4  barrier = Semaphore(0)
```

código de "rendezvouz"

Listing 3.5: Barrier solution

```
1  rendezvous
2
3  mutex.wait()
4      count = count + 1
5  mutex.signal()
6
7  if count == n: barrier.signal()
8
9  barrier.wait()
10 barrier.signal()
11
12 critical point
```

barreira normal tu passa só uma vez, por isso, se criou a barreira **REUTILIZÁVEL**