Programação Concorrente e Paralela

Noemi Rodriguez

2016

o que é programação concorrente e paralela?

programação concorrente: composição de linhas de atividades independentes

programação paralela: execução simultânea de linhas de atividades

Go blog (Rob Pike)

Concurrency is about dealing *with lots of things at once. Parallelism is about* doing *lots of things at once.*

Programação Concorrente

- não necessariamente envolve diferentes linhas de execução
  - alternativas para modelos de servidores concorrentes
- mas modelo mais comum de concorrência é multi*tarefa*

- necessidade de *comunicação* entre diferentes linhas de execução

programação concorrente e paralela — comunicação entre processos

- acesso compartilhado a dados

  concorrência: bases de dados, informações sobre conexões, ...
  paralelismo: dados de problemas científicos, ...

- espera por estados desejáveis

  concorrência: nova requisição de cliente, ...
  paralelismo: novos resultados, novos dados para processamento, ...

- sincronização!

Programação Concorrente

- linhas de controle independentes ativas simultaneamente
  - o que é "atividade"?
  - um processador X múltiplos processadores
- estudo nasceu com sistemas operacionais
- importância hoje: atendimentos concorrentes

- disputa por recursos!

Programação Paralela

- execução realmente simultânea
- linhas de execução que colaboram (ou não) em busca de solução
- paralelismo para melhoria de desempenho (ou viabilidade de execução)

paralelismo e concorrência

- em ambos os casos necessidade de *coordenação* entre atividades
- cooperação x competição

Objetivos da disciplina

- princípios e técnicas de programação concorrente
  - multiprocessadores
    - memória compartilhada
    - troca de mensagens
  - memória distribuída
    - troca de mensagens
  - padrões de programação paralela
    - exemplos clássicos: multiplicação de matrizes, resolução de sistemas
    - casos mais irregulares: espaços de busca

que princípios e técnicas são esses?

- notações para controle de fluxo
  - criação de processos e threads, ou outras formas de expressão de paralelismo
- abstrações de comunicação
- sincronização em sistemas de memória compartilhada
  - ... mas problemas também existem sem memória compartilhada...
- propriedades: *safety* e *liveness*
- avaliação de desempenho
- balanceamento de carga

Referências

- M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- M. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.
- P. Pacheco. *An Introduction to Parallel Programming*. McGraw-Hill, 2011.
- artigos técnicos.

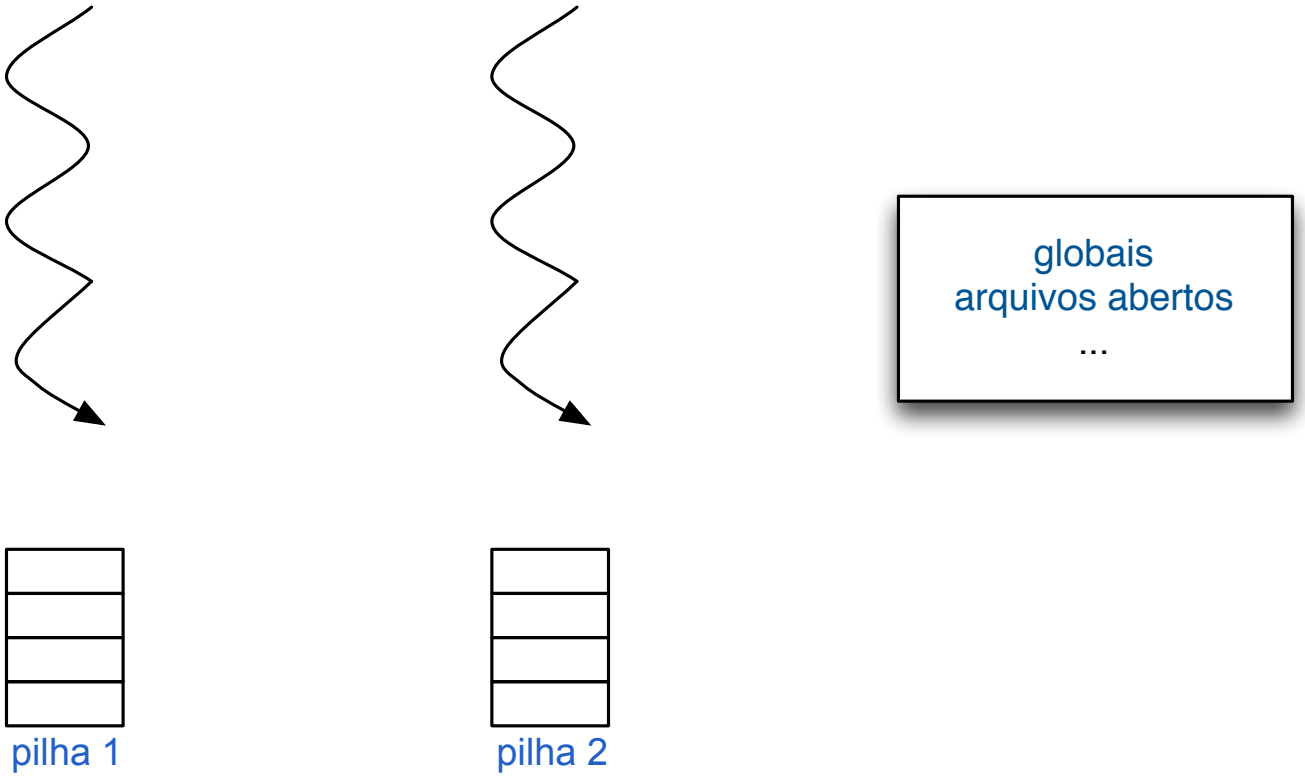
Fluxos de Execução

- classificação

Fluxos de Execução

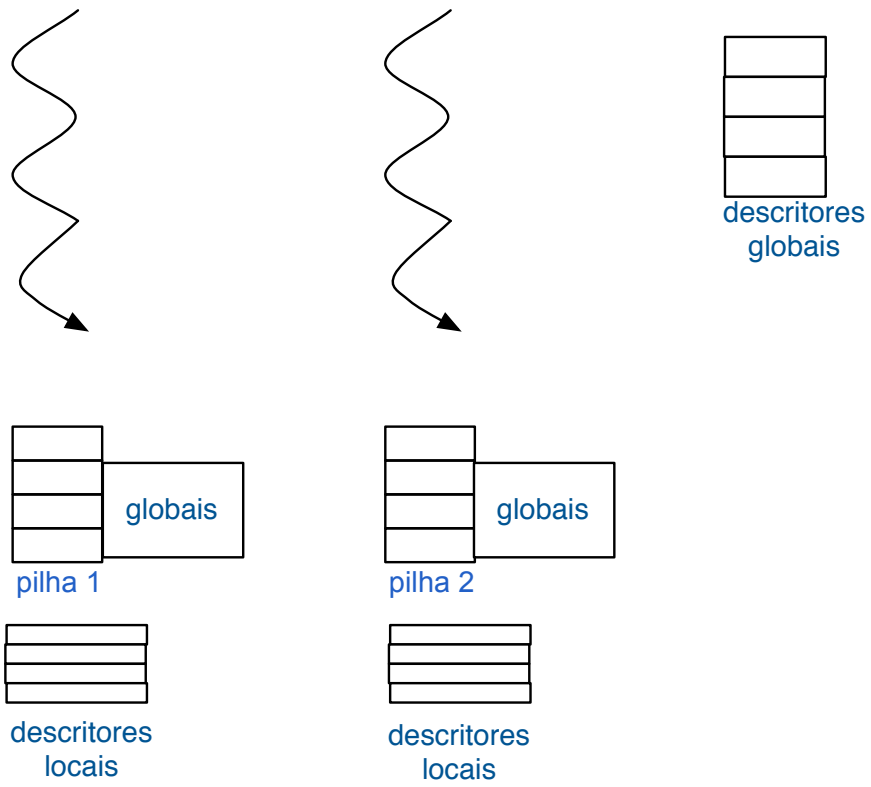
- processos, threads preemptivos, threads cooperativos
  - pilha de execução
  - dados globais?
  - espaços de endereçamento protegidos?
  - preempção?

Fluxos Independentes



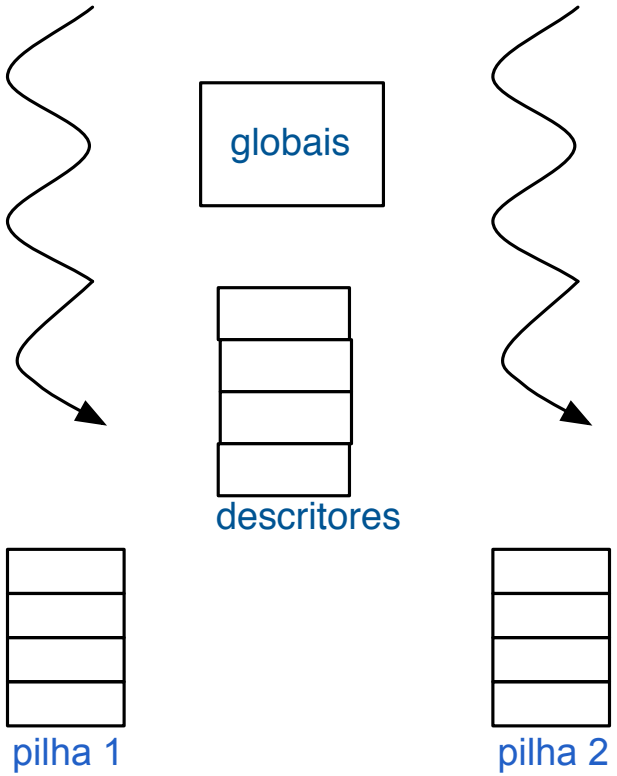
- pilhas independentes?
- acesso a globais?

Processos



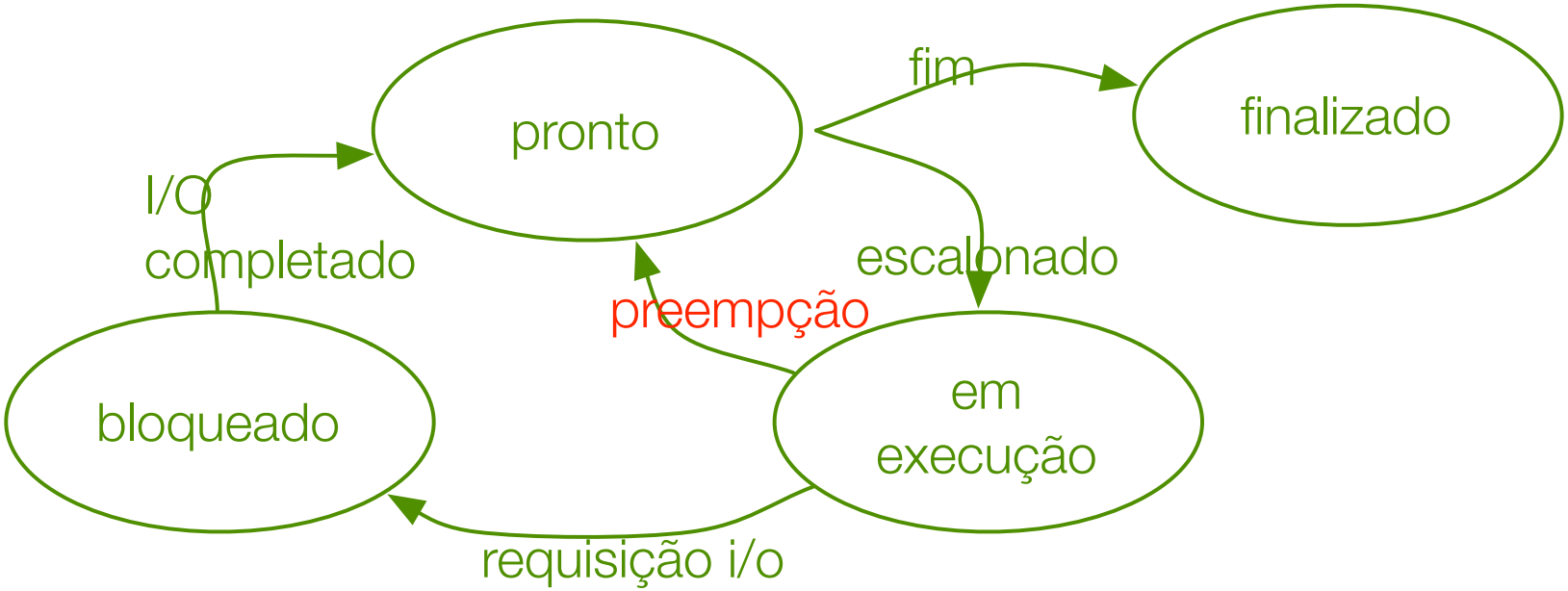
- pilhas independentes
- globais independentes
- descritores podem ser compartilhados
- preempção

Threads Preemptivas

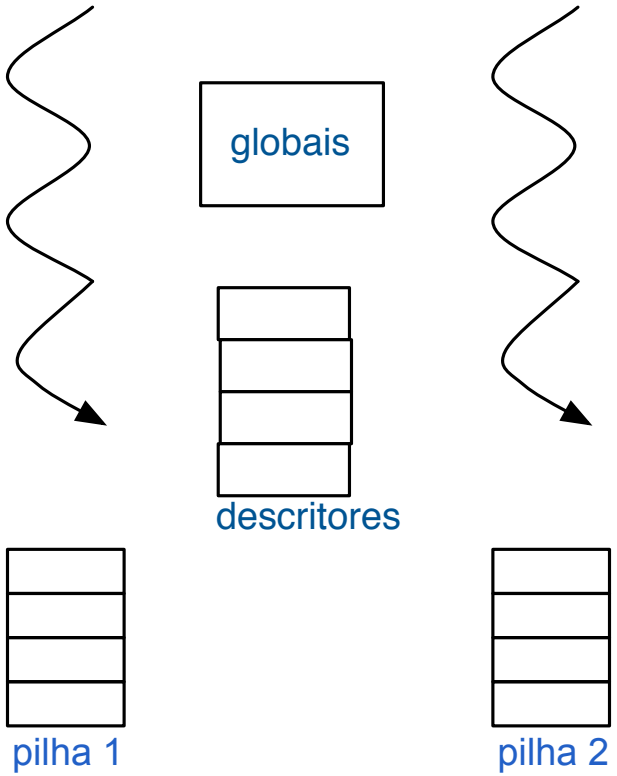


- pilhas independentes
- globais compartilhadas
- descritores compartilhados
- preempção

Preempção


pronto
fim
finalizado
I/O
completado
escalonado
preempção
bloqueado
em
execução
requisição i/o

Threads Cooperativas



- pilhas independentes
- globais compartilhadas
- descritores compartilhados
- sem preempção

Concorrência: modelo mais comum

threads preemptivos com memória compartilhada

- pthreads

Exemplo: Busca de primos com memória compartilhada

- imprimir primos entre 1 e 10^{10}
- máquina de 10 processadores
- um thread por processador

Busca de Primos com Memória Compartilhada

```
void primePrint (void) {
  long int j;
  for (j = 1, j<100000000000; j++) {
    if (isPrime(j))
      printf("%ld\n", j);
  }
}
```

Busca de Primos com Memória Compartilhada

- idéia 1: cada thread testa um intervalo pré-definido
- cada thread executa:

```
void primePrint (void){
  long int j;
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = i*109+1, j<(i+1)*109; j++) {
    if (isPrime(j))
      printf("%ld\n", j);
  }
}
```

  - nem todos os intervalos têm números iguais de primos
  - números grandes: maior dificuldade computacional
  - desbalanceamento

Busca de Primos com Memória Compartilhada

- idéia 2: contador compartilhado

```
int counter = new Counter(1);
```

- cada thread executa:

```
void primePrint {
  long j = 0;
  while (...) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Contador Compartilhado

- o objeto counter é único para todos os threads

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

Contador Compartilhado

- o objeto counter é único para todos os threads
- e o problema é que podemos ter:

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

```
temp = value;
value = value + 1;
return temp
```

Memória Compartilhada e Preempção

- muitos entrelaçamentos são possíveis...

```
thread 1:                    thread 2:

temp = value
                             temp = value
temp = temp + 1
                             temp = temp +1
value = temp
                             value = temp
```

que mundo é esse em que `value++` *não tem o efeito que esperamos...?*

Memória Compartilhada e Preempção

- e mais do que isso...
- não basta pensar nos entrelaçamentos possíveis!

Memória Compartilhada e Preempção

You don't know Jack about shared variables or memory models. Hans-J. Boehm and Sarita V. Adve. 2012. *Commun. ACM* 55, 2 (February 2012), 48-54.

Memória Compartilhada e Preempção

- imagine que o programa usa valores maiores do que a palavra do hardware
- `x++` traduzido para algo como:

```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++;
x_hi = tmp_hi;
x_lo = tmp_lo;
```

Memória Compartilhada e Preempção

- ler artigo Boehm para a próxima aula

Concorrência: programador sempre terá que lidar com dificuldades de programação?

mecanismos e técnicas

- mecanismos para lidar com memória compartilhada
  - semáforos e locks
  - monitores
  - mecanismos não bloqueantes
  - memória transacional
- eliminar memória compartilhada
  - troca de mensagens e outras abstrações
- utilizar multithreading cooperativo
- soluções mais recentes
  - propriedades garantidas pelo compilador
    - D, DPJ, ...

Paralelismo: questões

- como paralelizar a solução de um problema?
  - objetivo maior normalmente é obter menor tempo de execução
  - desenho da solução paralela normalmente é dependente da plataforma de execução
- que plataformas facilitam o desenvolvimento e depuração?
- como medir o tempo de execução e comparar soluções?
- tolerância a falhas: como garantir que o trabalho feito até o momento da falha não seja perdido?

Inicialmente: Sincronização

- como criar mecanismos que garantam acesso exclusivo aos dados em ambientes de memória compartilhada...?
- propriedades de *safety* e *liveness*

Seções Críticas

- conceito clássico em sistemas de memória compartilhada
- seção crítica: trecho de código que apenas uma thread pode estar executando em determinado momento

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
    int temp = value;
    value = value + 1;
    return temp;
  }
}
```

Garantia de Exclusão Mútua

- protocolos de entrada e saída de regiões críticas

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }
}
```

- se executado em threads 0 e 1: $RC_0 \rightarrow RC_1$ ou $RC_1 \rightarrow RC_0$

Exclusão Mútua – Propriedades

- garantia da exclusão mútua
- ausência de deadlock
- entrada em algum momento (ausência de starvation)
- ausência de esperas desnecessárias
  - thread não espera quando não há competição pela RC

Classes de Propriedades

- *safety*
  - programa nunca entra em estado ruim
- *liveness*
  - programa em algum momento entra em estado bom

Soluções de exclusão mútua utilizando espera ocupada

- estudo clássico
- alguma utilidade com múltiplos processadores
- algoritmos interessantes (!)
  - complexidade: caso com 2 threads é o mais inteligível

- Peterson
- Filtros
- Padaria

Exclusão Mútua – Propriedades

- garantia da exclusão mútua
  - safety
- ausência de deadlock
  - safety
- entrada em algum momento (ausência de starvation)
  - liveness
- ausência de esperas desnecessárias
  - nível de concorrência

Exclusão mútua: algoritmo de Peterson

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Art of Multiprocessor Programming

Exclusão Mútua

```
public void lock(0) {
  flag[A] = true;
  victim  = A;
  while (flag[B] && victim == A)
     {};
```

```
public void lock(0) {
  flag[B] = true;
  victim  = B;
  while (flag[A] && victim == B)
     {};
```

funciona?

- vamos supor, por contradição, que ambos estão na seção crítica

... funciona? prova por contradição

- do código:
  - $write_A$(flag[A]=true) → $write_A$(victim=A) → $read_A$(flag[B]) → $read_A$(victim) → CS_A
  - $write_B$(flag[B]=true) → $write_B$(victim=B) → $read_B$(flag[A]) → $read_B$(victim) → CS_B
- vamos assumir:
  - $write_B$(victim=B) → $write_A$(victim=A)
- como A entrou na RC:
  - $write_A$(victim=A) → $read_A$(flag[B] == false)
- mas então
  - $write_B$(flag[B]=true) → $write_B$(victim=B) → $write_A$(victim=A) → $read_A$(flag[B] == false)

```
public void lock(1) {
  flag[A] = true;
  victim  = A;
  while (flag[B] && victim == A)  {};
```

Exclusão Mútua: problemas!

- soluções baseadas em variáveis dependem de ordenação que pode se perder na compilação...

```
public void lock(0) {
  victim  = 0;   [2]
  flag[0] = true; [3]
  while (flag[1] && victim == 0)   [4]
     {};
```

```
public void lock(1) {
  victim  = 1; [1]
  flag[1] = true; [5]
  while (flag[0] && victim == 1) [6]
   {};
```