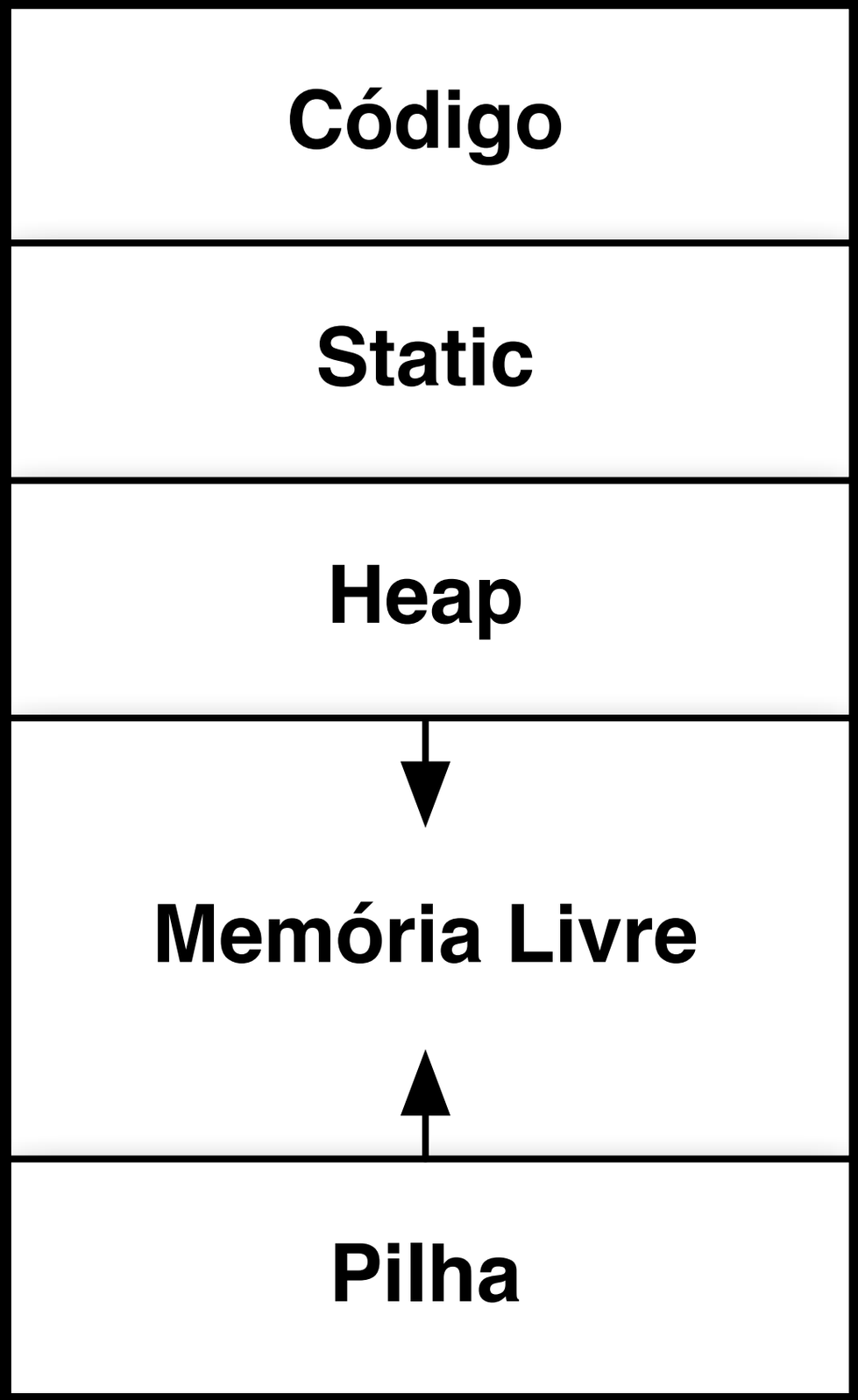
Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt



Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

Heap

- A heap é a porção da memória utilizada para dados que vivem ‘indefinidamente’
  - ou até que o programa explicitamente remova
- Muitas linguagens permitem a criação de objetos cuja existência não é ligada à ativação de procedimentos
- Estes objetos são guardados na heap

Memory Manager

- Subsistema que aloca espaço na heap
- Serve como interface entre os programas e o sistema operacional
- Para linguagens como C ou C++ que desalocam espaços de memória manualmente, responsável também por desalocar memória

Tarefas do *Memory Manager - Allocation*

- Quando um programa requisita memória para uma variável ou objeto, aloca uma porção de memória contígua na heap do tamanho pedido
- Se possível, satisfaz uma requisição usando espaço livre da heap
- Se não há espaço suficiente, tenta aumentar o espaço da heap com o SO
- Se o espaço está esgotado, esta informação é repassada ao programa

Tarefas do *Memory Manager* - *Deallocation*

- Libera espaço deslocado ao *pool* de espaço livre na heap
- Desta forma, a heap pode reusar o espaço para outras requisições de alocação
- Normalmente, *memory managers* não devolvem memória ao SO, mesmo que o uso da heap diminua

Gerenciamento de Memória

- Seria mais simples se:
  - (a) todos os pedidos de alocação fossem para porções do mesmo tamanho
  - (b) espaço fosse liberado de forma previsível, algo como FIFO, primeiro a alocar, primeiro a desalocar
- Na maioria das linguagens, estas duas condições não são satisfeitas

Memory manager tem de estar preparado para servir requisições em qualquer ordem, de qualquer tamanho, de 1 byte ao espaço inteiro do programa

Propriedades desejadas

- *Space Efficiency*
  - minimizar o espaço total necessário pela heap
- *Program Efficiency*
  - deve fazer bom uso do subsistema de memória, para que programas rodem rápido
- Baixo *Overhead*
  - tempo gasto (des)alocando memória

Hierarquia de memória

- Gerenciamento de memória e otimização de compiladores deve levar em conta como a memória é organizada no computador

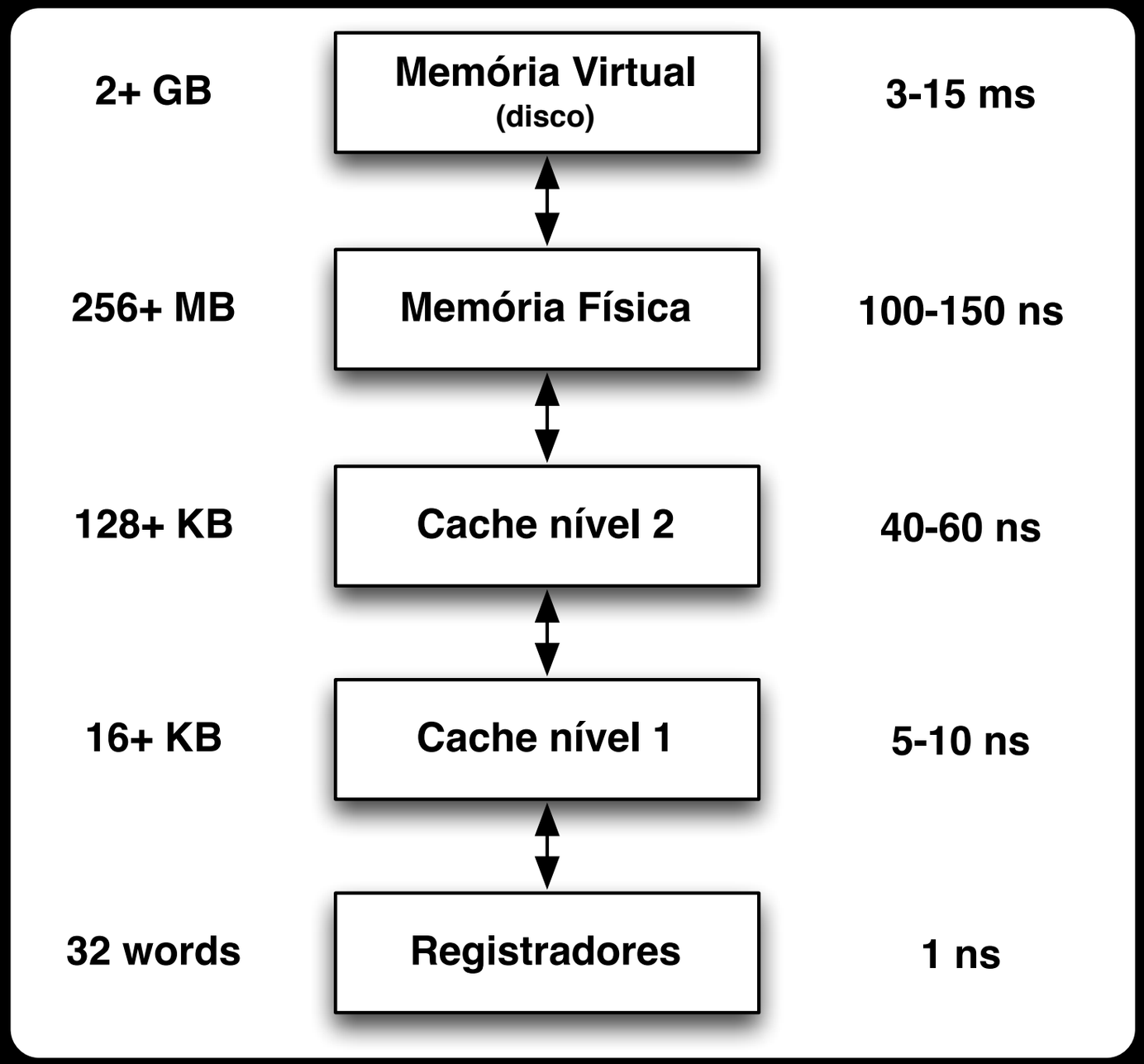
- A eficiência de um programa não é determinada apenas pela quantidade de instruções, mas também com o tempo para executá-las

- Acesso a diferentes partes da memória pode variar significativamente

Hierarquia de memória

- Há uma grande variância nos tempos de acesso à memória
  - *Small and fast vs. large and slow*
- Computadores organizam armazenamento em uma hierarquia de memória
  - elementos de armazenamento com os menores e mais rápidos *próximos* ao processador

Hierarquia de memória



Adaptado de Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman
Compilers: Principles, Techniques & Tools

Locality

- Muitos programas tem alto grau de *locality*
  - gastam maior parte do tempo executando uma pequena fração do código, que toca apenas em uma pequena fração dos dados
- *Temporal locality*
- *Spatial locality*

Costuma-se dizer que programas gastam 90% do tempo executando 10% do código. Quais as razões?

Locality

- Programas tem muitas instruções que nunca são executadas (código legado, *components, libraries*)
- Várias partes do código não são exercitadas em geral, como tratamento de exceções
- Gasta-se muito tempo em loops e ciclos recursivos

Lidando com *Locality*

- Manter dados usados mais recentemente na parte mais rápida da hierarquia
  - pode funcionar com alguns programas, mas nem todos
- Em geral, não dá para saber, olhando apenas para o código, que partes serão mais utilizadas
  - deve ser possível ajustar dinamicamente

Otimizações visando Memória

- A política de manter instruções recentemente usadas em cache tende a funcionar bem
  - o uso da memória no passado geralmente é um bom preditor do futuro
- Ao executar uma nova instrução, há uma alta probabilidade que a próxima seja executada
  - colocar blocos básicos contíguos na mesma *página*, se possível

Fragmentação

- No início da execução do programa, a heap é uma grande unidade de espaço livre
- Na medida que memória é utilizada e liberada, este espaço é dividido entre partes livres e ocupadas de memória
- As partes livres (*holes*) não residem em áreas contíguas da heap

Fragmentação

- A cada requisição, devemos encontrar um *hole* grande o suficiente para alocar os dados
- A não ser que seja exatamente do tamanho solicitado, temos que dividir o *hole* ao alocar espaço
- Isto pode gerar fragmentação, grandes quantidades de espaços livres pequenos e não contíguos

Que estratégias podemos utilizar para reduzir fragmentação?

Controlar antes...

- Podemos reduzir fragmentação controlando a maneira como alocamos objetos na heap
  - *first-fit vs. best-fit vs. next-fit*
- Best-fit divide espaços livres em *bins*, de tamanhos variáveis - *space utilization*
- Next-fit tenta melhorar *spatial locality*, usando best-fit e alocando objetos próximos

Controlar depois…

- Ao desalocar objetos da heap, combinar (*coalesce*) o espaço livre com espaços livres adjacentes da heap

- Marcar *bins* com um bit indicando se está ocupado ou livre

- Se *bins* não forem utilizados, temos que marcar as fronteiras dos espaços livres

Manual Deallocation

- Gerenciamento manual de memória tende a gerar erros
  - Esquecer de deletar dados que não podem mais ser referenciados (*memory leak*)
  - Referenciar dados deletados (*dangling reference*)

Memory Leaks

- É difícil afirmar que um programa não irá mais fazer referências a um objeto
- Uma estratégia comum é não desalocar nada
- Este tipo de erro não gera problemas de corretude (enquanto máquina tem memória)
- Problemático em programas que precisam executar por longos períodos de tempo

Dangling References

- Espaço de memória que foi liberado eventualmente será preenchido por outros objetos, variáveis, etc.

- Referenciar um ponteiro *dangling* pode gerar erros difíceis de serem depurados

- Potencialmente afeta a corretude do programa

- Outro tipo de erro é acesso a endereços ilegais

Garbage Collection

Garbage Collection

- Dados que não podem mais ser referenciados são denominados de *garbage*
- Muitas linguagens de programação liberam do programador a tarefa de gerenciar manualmente a memória, oferecendo *garbage collection*
- Conceito já presente na implementação inicial de Lisp, em 1958
    - Java, Perl, Modula-3, ML, Prolog, Smalltalk…

Requisitos

- O tipo dos objetos deve ser possível de ser determinado em tempo de execução
- A partir do tipo, devemos ser capazes de dizer o tamanho do objeto, e quais componentes deste objeto tem referências a outros objetos
- Referências são sempre endereços para o início dos objetos
- Objetos se tornam *garbage* quando o programa não pode mais *alcançar* os objetos

Desempenho

- Custo previne adoção
- Diversas abordagens foram propostas
- Algumas métricas de desempenho:
  - Tempo de execução total
  - Uso do espaço
  - *Pause time*
  - *Program locality*

Requisitos podem ser conflitantes

Projeto da Linguagem pode influenciar no uso da memória

Reachability

- Dados que podem ser acessados diretamente por um programa, sem precisar dereferenciar um ponteiro, formam o *root set*

- Um programa pode alcançar qualquer membro deste conjunto a qualquer momento

- Recursivamente, qualquer objeto cujas referências são armazenadas nos membros do *root set* é também alcançável

Reachability

- O conjunto de objetos alcançáveis muda durante a execução do programa
- Existem operações que alteram este conjunto:
  - *object allocation*
  - *parameter passing and return values*
  - *reference assignments*
  - *procedure returns*

Como encontrar objetos inalcançáveis?

A ideia é desalocar implicitamente

Estratégias

- *Incremental*
  - A cada instrução realiza alguma tarefa
- *Batch-oriented*
  - roda sob demanda, quando espaço esgota

Reference Counting

- Ideia: adicionar um contador para cada objeto alocado na *heap*
- O contador rastreia o número de ponteiros para aquele objeto
- Quando o contador alcança zero, o sistema pode liberar aquele objeto
    - Liberar um objeto pode levar a liberação de outros

Reference Counting

- *object allocation*: +1
- *parameter passing*: +1 para objetos passados
- *reference assignments*: u = v
  - antigo objeto de u -1; objeto v +1
- *procedure returns*:
  - objetos em variáveis locais -1 (para toda referência)
- *transitive loss of reachability*
  - quando a contagem alcança 0, todo objeto referenciado pelo objeto zerado tem contagem decrementada

Problemas

- O código em execução precisa de um mecanismo para diferenciar ponteiros
- Custo
    - toda operação tem custo adicional de atualizar a contagem
    - pode levar muito tempo (montar uma fila)
    - no entanto, permite *garbage collection* incremental
- Estruturas de dados cíclicas
    - objetos que apontam para o nó pai ou tem recursão mútua
    - contagem nunca alcançará 0

Batch collectors

- geralmente são executados quando espaço livre está esgotado ou abaixo de um certo limiar
- *collector* pausa a execução do programa, examina memória alocada para descobrir objetos inutilizados e libera o espaço
- geralmente rodam em duas fases
  - descoberta de objetos mortos
  - desalocação e “reciclagem” de objetos mortos

Identificando Objetos *Live*

- Em geral, usa-se o que é chamado de algoritmo de *marking*

- O coletor usa um bit para cada objeto na *heap*, chamado de *mark bit*

- Este bit é armazenado no cabeçalho do objeto, junto à informação para registro de localização e tamanho do objeto

Intuição de *marking*

- Limpa todos os *mark bits* e constrói uma worklist
  - todos os ponteiros em registradores e em variáveis acessíveis aos procedimentos
- Caminha nesta *worklist* e segue quaisquer referências a partir destes ponteiros como alcançável

Intuição de *sweep*

- Ao término do algoritmo, objetos *unmarked* são inalcançáveis e podem ser liberados
- Travessia nos objetos da *heap* liberando objetos inalcançáveis
    - opcionalmente, já reseta o *mark bit* para a fase de *marking* evitar a travessia inicial

Mark-and-Sweep

- **<u>Entrada:</u>** root set, heap, e uma lista (*Free*) com todos os espaços livres da heap

- **<u>Saída:</u>** lista *Free* modificada, após todo o lixo ser removido (liberado)

- **<u>Método:</u>** o algoritmo consiste de uma lista (*Unscanned*) com objetos alcançáveis, mas seus sucessores ainda não foram considerados

```
/* marking phase */
insere todos de root set em Unscanned
while (Unscanned ≠ ∅) {
   remove objeto o de Unscanned
   foreach (o' referenciado em o) {
      if (o'.reached == 0) {
         o'.reached = 1;
         insere o' em Unscanned
      }
   }
}
…
```

```
…
/* sweeping phase */
Free = ∅
foreach (espaço de memória o na heap) {
   if (o.reached ==0) {
      insere o em Free
   }
   else {
      o.reached = 0;
   }
}
```

Abstrações

- Todos os algoritmos em *batch*, também conhecidos como *trace-based* computam o conjunto de objetos alcançáveis e usam o seu complemento para liberar memória
- Memória é reciclada da seguinte maneira:
  - o programa faz requisições de alocação
  - *garbage collector* descobre *reachability*
  - *garbage collector* libera espaço dos objetos inalcançáveis

Estados da Memória

- Embora os algoritmos *trace-based* possam diferir em sua implementação, em geral são descritos de acordo com os estados
  - *Free*: espaço de memória pronto para ser alocado; não pode conter objeto alcançável
  - *Unreached*: espaço normalmente é denominado inalcançável, a não ser que o *tracing* prove o contrário
  - *Unscanned*: espaço alcançável, mas seus ponteiros ainda não foram escaneados
  - *Scanned*: todo objeto *Unscanned* eventualmente será observado e transiciona para este estado

Variações

- *Baker's mark-and-sweep:* ao invés de examinar a heap inteira, mantém uma lista de objetos alocados
- *Mark-and-compact*: move objetos na heap para eliminar fragmentação de memória, ao invés de apenas marcar como livre
- *Copying collectors*: divide espaço em duas áreas, A e B, que alternam papéis; quebra dependência entre tracing e procura de espaço livre
- *Incremental*: intercalam GC e programa, são conservadores, portanto

Copying Collectors

- Divide a *heap* em duas *pools*, *old* e *new*
- Aloca memória sempre a partir de *old*
- *Stop and copy*: quando alocação falha, copia todos os dados *live* de *old* para *new* e inverte identidade
    - pode usar *mark-and-sweep* ou *incremental*

Comparando

- Garbage collection liberta o programador de se preocupar com liberação de memória, leaks, etc
- Em geral, argumenta-se que os benefícios superam as desvantagens, independente da técnica escolhida

Comparando

- Reference counting distribui o custo de deslocar “mais uniformemente” durante execução
  - Aumenta o custo de cada instrução que envolve valores da heap, mesmo que o programa nunca esgote espaço disponível
- Batch collectors, por sua vez, não tem custo algum a não ser que espaço fique indisponível
  - neste ponto, há o custo total da coleção

Comparando

- *Mark-and-sweep* costuma analisar a *heap* inteira, enquanto *copying collectors* costumam analisar apenas *live data*
  - *tradeoff* depende da aplicação
- Tanto *reference counting* como *batch collectors* mais conservadores tem dificuldades com estruturas cíclicas

Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt