Software Básico

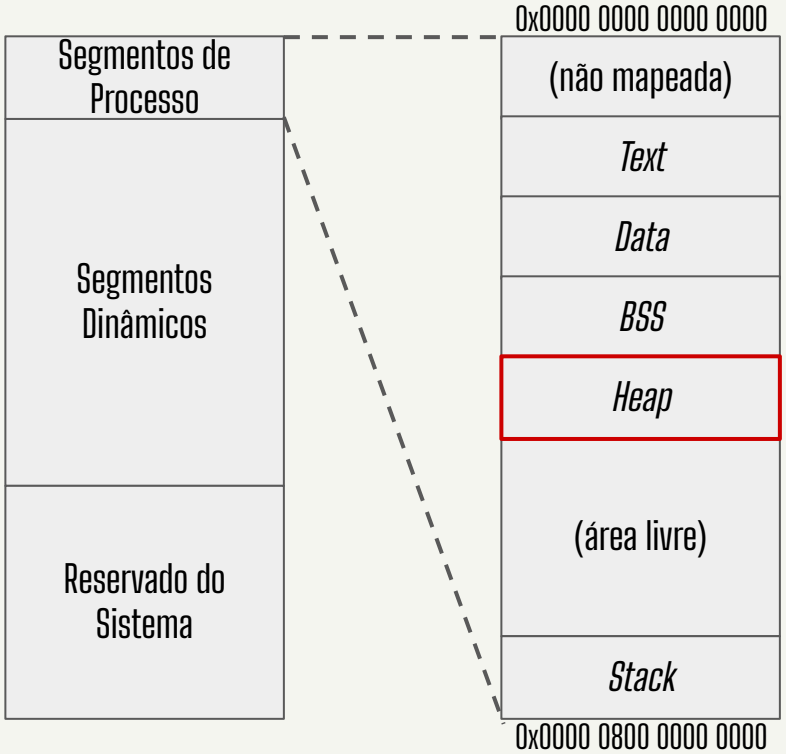
Aula #16

A Seção *Heap*: Alocação Dinâmica de Memória

Para que serve e como funciona a seção *heap*? Quais são as principais estratégias de gerenciamento de memória na *heap*? Como funciona um coletor de lixo?

Ciência da Computação - BCC2 - 2023/02
Prof. Vinícius Fülber Garcia

Relembrando



A seção *heap* faz parte dos **segmentos de um processo**.

Essa seção está localizada após a BSS e é utilizada no contexto de alocações dinâmicas de memória.

Tem esse nome uma vez que o gerenciamento da memória **não ocorre apenas "em uma ponta".**

A Seção *Heap*

Mas... uma vez que já temos a seção *data*, *bss* e *stack*, **para que mais uma seção dedicada à alocação de memória?**

Quais são...

- as limitações da seção *data*?
- as limitações da seção *bss*?
- as limitações da seção *stack*?

A Seção *Heap*

Mas... uma vez que já temos a seção *data*, *bss* e *stack*, **para que mais uma seção dedicada à alocação de memória?**

Quais são...

- as limitações da seção *data*?
  - Espaços de dados com tamanho e valor preestabelecidos!
- as limitações da seção *bss*?
  - Espaços de dados com tamanho preestabelecido!
- as limitações da seção *stack*?
  - Espaços de dados vinculados a registros de ativação específicos!

A Seção *Heap*

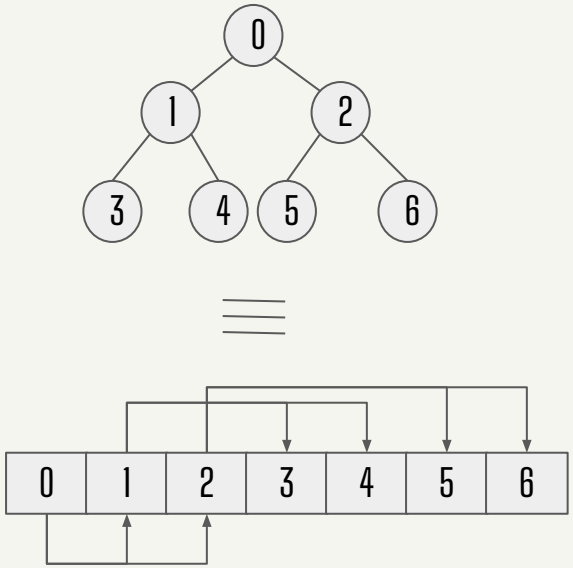
Mas... uma vez que já temos a seção *data*, *bss* e *stack*, **para que mais uma seção dedicada à alocação de memória?**

Além disso, algumas estruturas de dados não se beneficiam de espaços de memória grandes e sequencialmente alocados (vetores)!

Quais são exemplos dessas estruturas?

A Seção *Heap*

Vamos considerar, para exemplo, uma árvore binária!



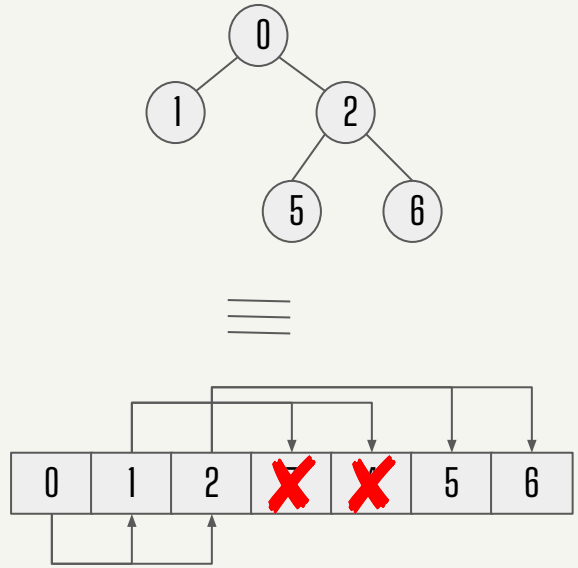
É possível implementar uma árvore binária via um vetor predefinido?

SIM!

Limitação #01: deve haver um tamanho máximo para a árvore!

A Seção *Heap*

Vamos considerar, para exemplo, uma árvore binária!



É possível implementar uma árvore binária via um vetor predefinido?

SIM!

Limitação #02: desperdício de memória em nodos não necessários!

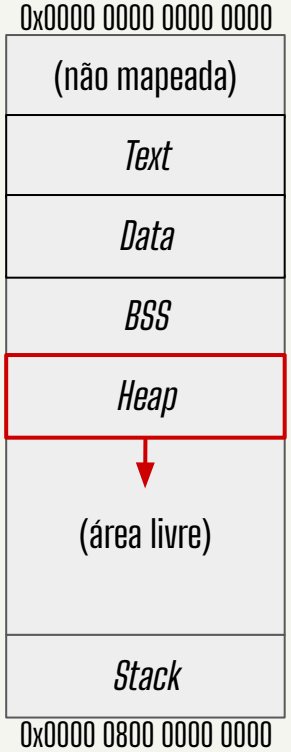
A Seção *Heap*

Sabendo dos cenários descritos, seria conveniente contar com uma área de memória que pode ser segmentada em **quantidades aleatórias de bytes consecutivos, alocados em tempo de execução**.

Bem-vindo à ideia da *heap*!

Ao melhor do nosso conhecimento, todos os modelos de execução (elf, coff, pe, exe...) preveem um espaço de memória como a *heap*.

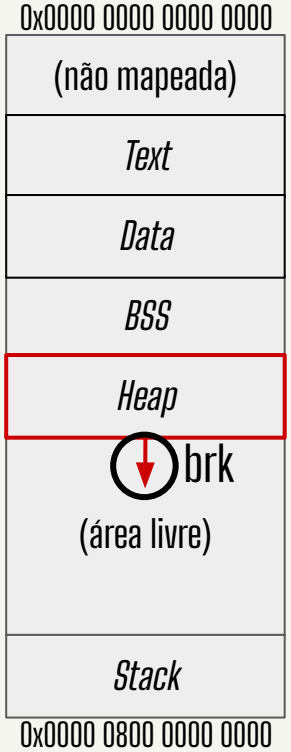
A Seção *Heap*



A seção *heap* apresenta várias características interessantes!

- Ela cresce "para baixo" no espaço de endereçamento...
  - Qual é a operação de para aumentar o espaço da *heap*?
- Ela cresce em oposição à *stack*...
  - O que acontece quando elas se encontram?

A Seção *Heap*: o brk



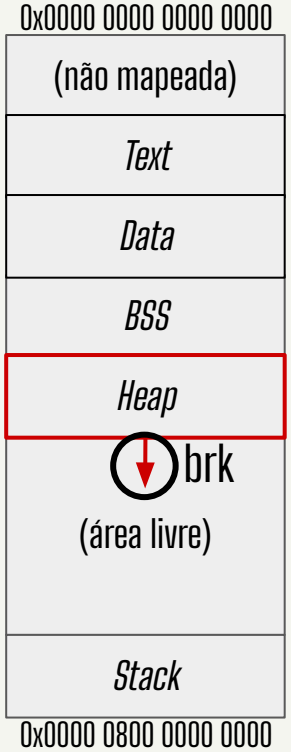
Bem... mas se gerenciamento de memória é possível através da modificação do tamanho da *heap*, naturalmente deve existir um **sentinela indicando o limite de seu espaço de memória**.

Sim, e ele é chamado de **brk**!

Informalmente, podemos pensar nele como o equivalente ao rsp da pilha.

Mas uma diferença importante! Palpites?

A Seção *Heap*: o brk



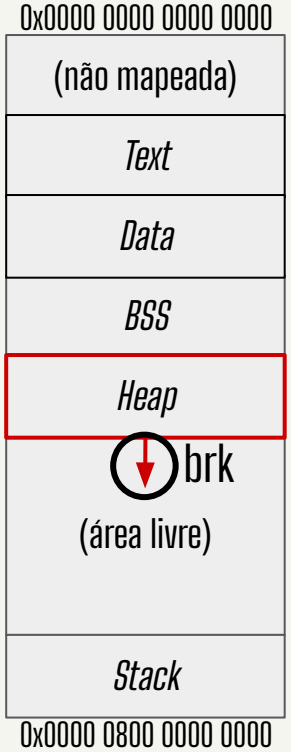
O brk não é um registrador, mas sim uma posição de memória!

O brk é armazenado por processo! Ou seja, ele **consta na PCB vinculada a um PID** na tabela de processos.

Em termos teóricos, tudo que está entre o final da seção BSS e o brk pode ser acessado com permissão de leitura e escrita!

E o que está depois de brk?

A Seção *Heap*: o brk

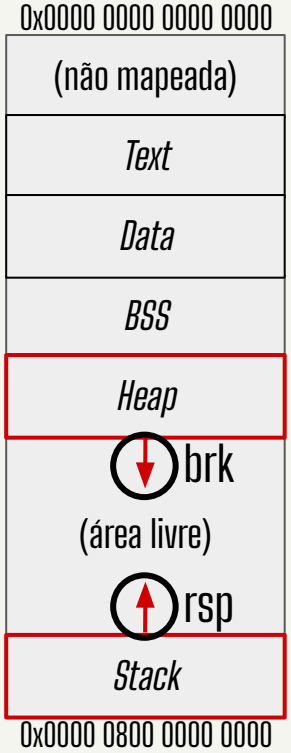


A dicotomia: o que eu posso Vs. o que eu não devo fazer

Teoricamente, eu não devo acessar posições de memória entre a *heap* (brk) e a *stack* (rsp)... porém, isso não gera *segmentation fault*, na prática.

Mas claro, isso pode (e provavelmente vai) gerar **problemas lógicos no processo**.

Curioso!



Se o brk passar o rsp (ou vice-versa), um *segmentation fault* pode acontecer...

- ao fazer um malloc?
- ao chamar uma função?

É possível gerar um *segmentation fault* por estouro de pilha **sem alocação dinâmica e por meio de uma única chamada de função**?

A Seção *Heap*: o brk

%rax	Nome	Argumentos	Comentários
12	brk	%rdi: novo valor de brk	Se o valor de %rdi for zero, o retorno é a posição atual de brk; caso contrário, o valor é atualizado

O brk é acessado via *syscall*

O serviço de acesso ao brk é 12 (0xc); como de praxe, após setado o registrador de serviço (%rax) e o de argumento (%rdi), uma instrução *syscall* deve ser executada.

A Seção *Heap*: o brk

```
1)    pointer = brk(0);
2)    pointer += 1024;
3)    new_pointer = brk(pointer);
```

Analise o fragmento de código em C ao lado...

- Qual é o tipo da variável *pointer*?
- Qual é o tamanho do bloco de dados alocado, com endereço retornado em *pointer*?
- Qual é o valor de *new_pointer*?

sbrk(1024)

Gerenciamento de Memória

```
1)    void* pointer = malloc(1024);
2)    if (!pointer) return 1;
3)    free(pointer);
4)    return 0;
```

Naturalmente, manipular o brk diretamente consiste em enfrentar diversos desafios de **gerenciamento de memória.**

As funções **malloc** e **free**, por exemplo, lidam com esses desafios pelo programador!

Gerenciamento de Memória

É de se imaginar que existem diversas estratégias para lidar com o gerenciamento de memória relacionada a *heap*.

Algumas estratégias são bastante **simples**, mas **pouco ou nada eficientes** no reuso de memória; outras estratégias são complexas, mas são bastante eficientes em reutilizar memória já liberada pelo processo.

Gerenciamento de Memória

```
1)   void setup_brk();
2)   void dismiss_brk();
3)   void* memory_alloc(unsigned long int
     bytes);
4)   int memory_free(void *pointer);
```

A partir de agora, vamos considerar uma API simples de gerenciamento de memória, apresentada ao lado.

Nosso objetivo é considerar a implementação da **API em C** (não em *assembly*).

Gerenciamento de Memória

```
1) void setup_brk(); //Obtém o endereço de brk
2) void dismiss_brk(); //Restaura o endereço de brk
3) void* memory_alloc(unsigned long int bytes); //Executa brk para abrir um bloco de "bytes" bytes
4) int memory_free(void *pointer); //Não executa nada
```

Estratégia #01: INGÊNUA

Nessa estratégia, uma requisição de alocação de memória **sempre avança o brk**. Não há reaproveitamento de memória liberada.

Gerenciamento de Memória

```
1)   #include <unistd.h>
2)   #include <stdio.h>
3)   void *brk_original = 0, *brk_current = 0;
4)   void setup_brk(){
5)      brk_original = sbrk(0);
6)      brk_current = brk_original;
7)   }
8)   void dismiss_brk(){
9)      brk_current = brk_original;
10)     brk(brk_original);
11)  }
12)  void* memory_alloc(unsigned long int bytes){
13)     brk_current += bytes; return brk(bytes);
14)  }
15)  int memory_free(void *pointer){
16)     return 0;
17)  }
```

Estratégia #01: INGÊNUA

A implementação ao lado é uma opção para a implementação ingênua.

Por que usar sbrk(0) ao invés de brk(0)?

Qual é o grande problema?

Gerenciamento de Memória

Estratégia #02: **REAPROVEITAMENTO DE MEMÓRIA**

Existem múltiplas alternativas para realizar a implementação de reaproveitamento de memória.

Vamos analisar uma alternativa possível e funciona... mas não a melhor, e nem mesmo eficiente!

Beneficiar a simplicidade em prol da compreensão!

Gerenciamento de Memória

```
1) void setup_brk(); //Obtém o endereço de brk
2) void dismiss_brk(); //Restaura o endereço de brk
3) void* memory_alloc(unsigned long int bytes);
    a) //1. Procura bloco livre com tamanho igual ou maior que a requisição
    b) //2. Se encontrar, marca ocupação e retorna o endereço do bloco
    c) //3. Se não encontrar, abre espaço para um novo bloco
4) int memory_free(void *pointer); //Marca um bloco ocupado como livre
```

Estratégia #02:
REAPROVEITAMENTO DE MEMÓRIA

Analise o código ao lado!

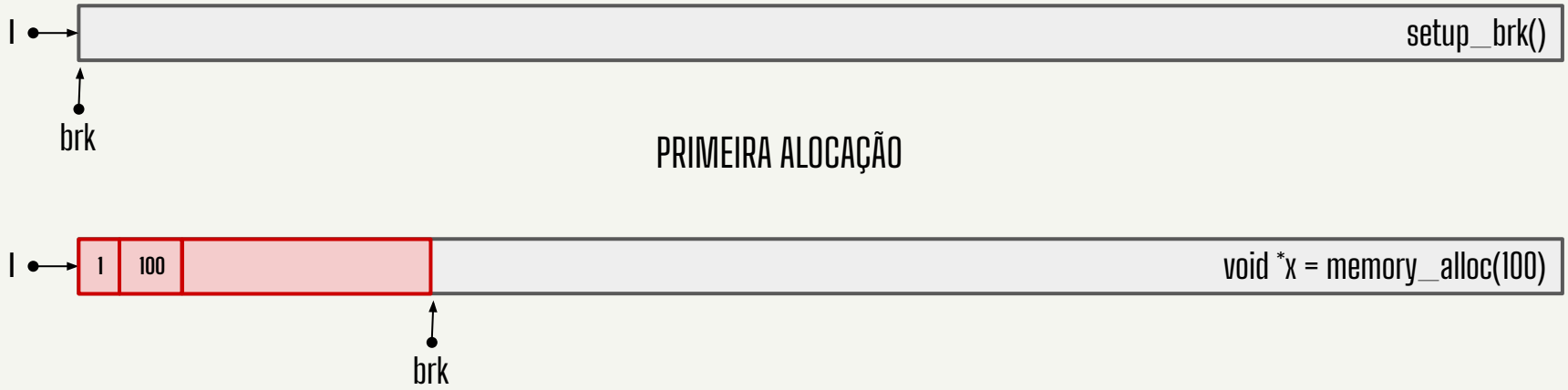
- Essa estratégia funciona?
- Qual é o principal problema da mesma?

Gerenciamento de Memória

Estratégia #02: **REAPROVEITAMENTO DE MEMÓRIA**

Flags: bloco em uso (1 byte); tamanho do bloco (8 bytes)

Dados: bloco alocado em frente às *flags*

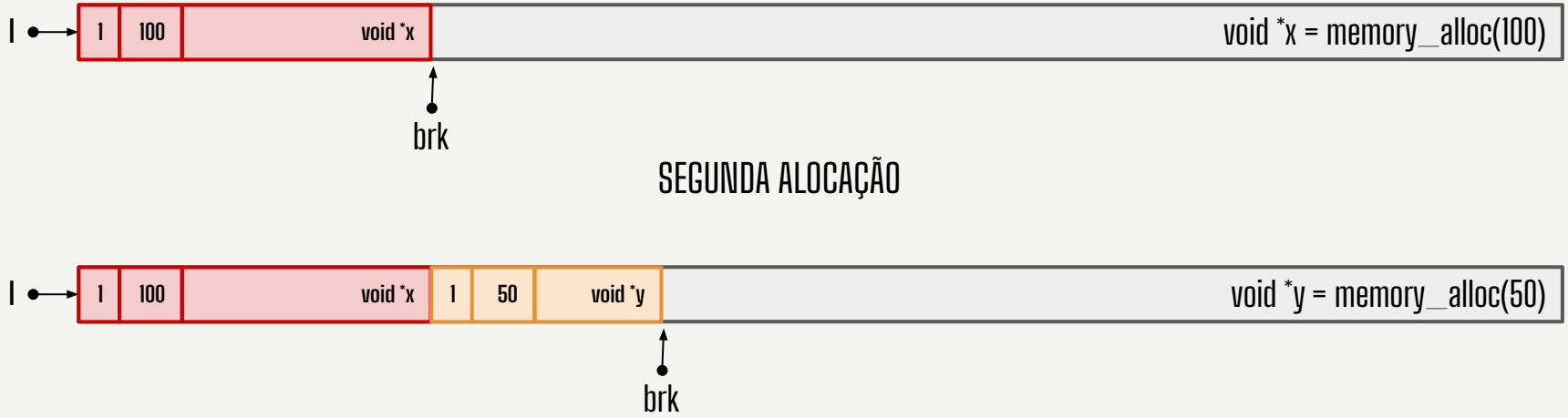


Gerenciamento de Memória

Estratégia #02: **REAPROVEITAMENTO DE MEMÓRIA**

Flags: bloco em uso (1 byte); tamanho do bloco (8 bytes)

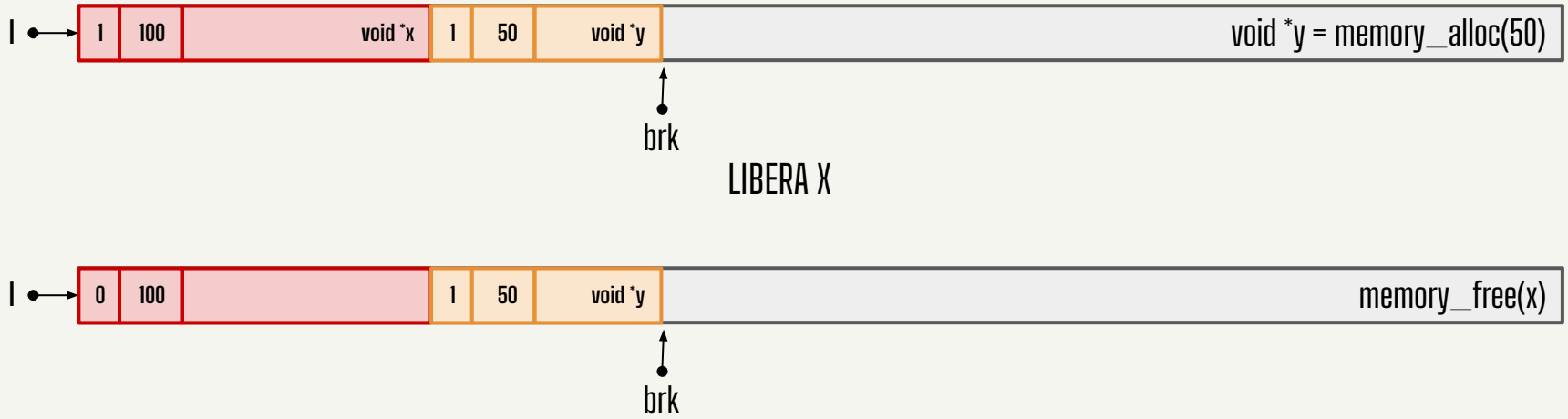
Dados: bloco alocado em frente às *flags*



Gerenciamento de Memória

Estratégia #02: REAPROVEITAMENTO DE MEMÓRIA

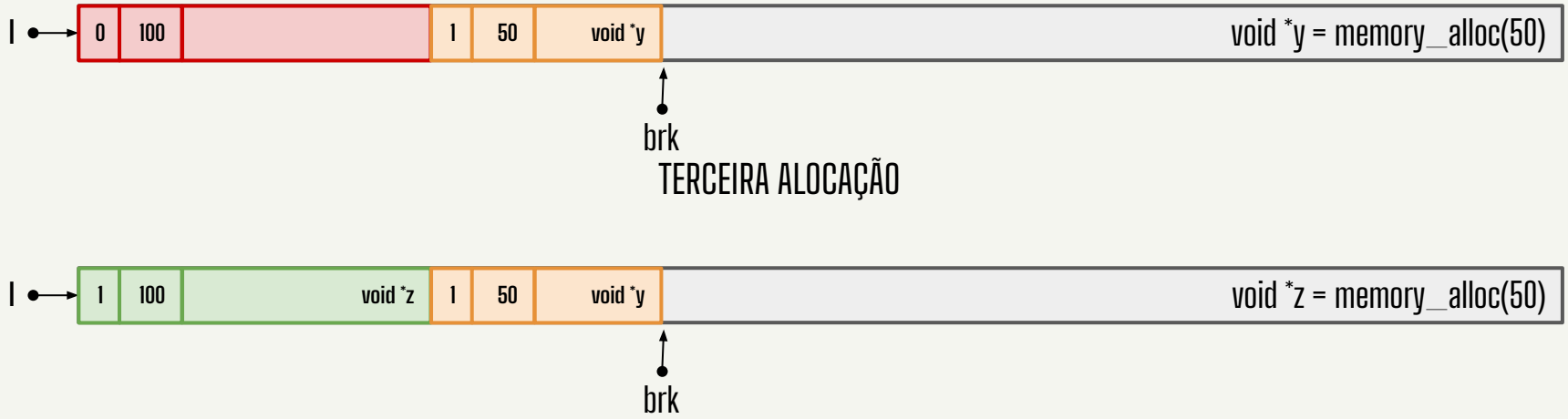
Para a desalocação, basta atribuir à *flag* de uso o valor zero (0).



Gerenciamento de Memória

Estratégia #02: REAPROVEITAMENTO DE MEMÓRIA

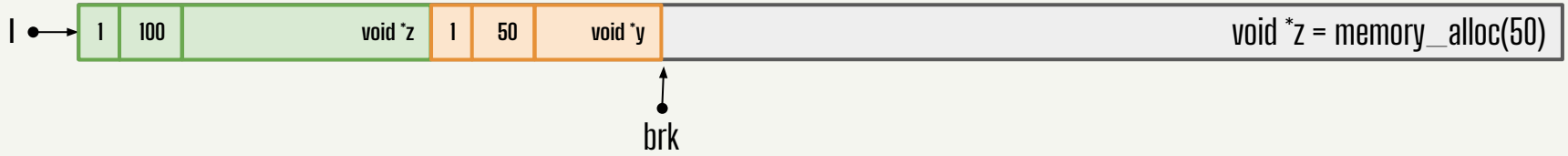
Eu uma nova alocação, existindo um bloco já alocado compatível, usar o mesmo!



Gerenciamento de Memória

Estratégia #02: **REAPROVEITAMENTO DE MEMÓRIA**

Porém, 100 bytes são alocados para uma requisição de 50!



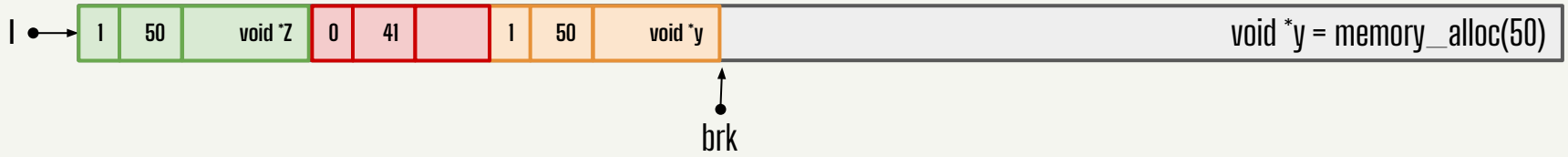
Esse problema pode acontecer tanto em *first fit*, quanto em *best fit*...

COMO É POSSÍVEL MELHORAR?

Gerenciamento de Memória

Estratégia #02: REAPROVEITAMENTO DE MEMÓRIA

Que tal utilizar apenas o segmento necessário de memória?



Por que sobrou apenas 41 bytes?
Como os blocos de alocação são percorridos?

Trabalho da Disciplina

Estratégia #02: **REAPROVEITAMENTO DE MEMÓRIA**

VAMOS IMPLEMENTAR?

Esta estratégia não, pois será o trabalho de vocês!

Implementar um alocador de memória nos moldes da ESTRATÉGIA #02, adotando uma técnica de *first fit* para reuso de memória que quebra o bloco de alocação apenas se existirem pelo menos 10 bytes restantes.

Assembly AMD64

Gerenciamento de Memória

Estratégia #03: *BUDDY* (KNUTH)

O algoritmo *buddy* consiste em uma estratégia de gerenciamento de alocação de memória **um pouco mais próxima do que é de fato implementado** em sistemas reais.

Buddy quer dizer algo como “camarada”, em inglês...
Esse nome fará mais sentido quando entendermos o algoritmo!

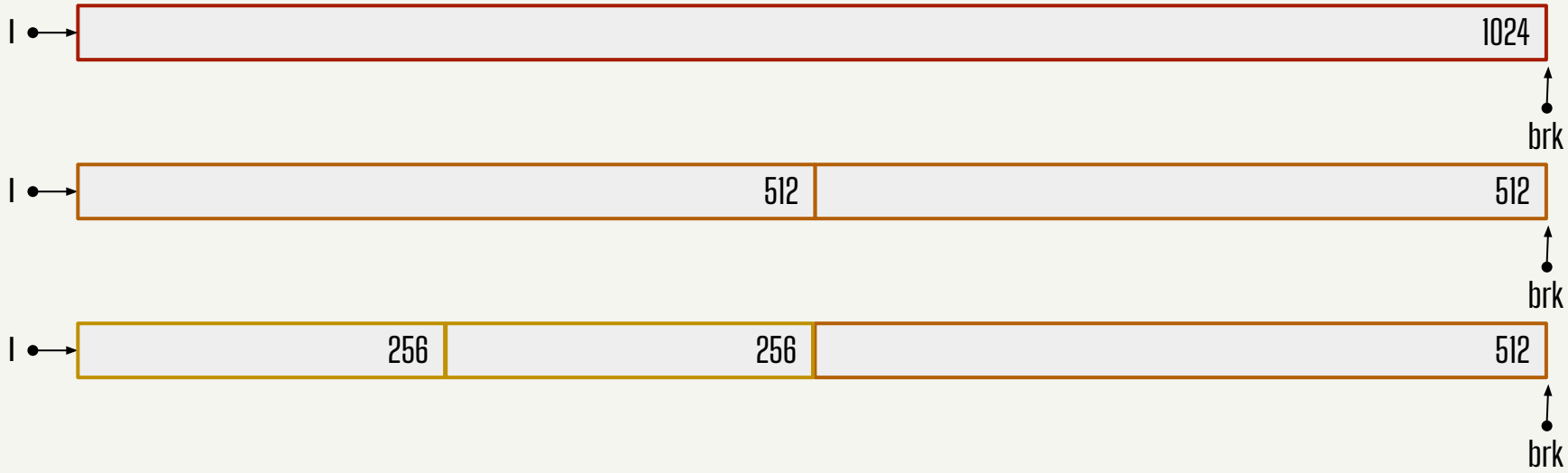
Gerenciamento de Memória

Estratégia #03: *BUDDY* (KNUTH)

- Ao iniciar o alocador, um grande espaço é alocado na *heap* (imediatamente)
- Ao receber um pedido de alocação, o espaço é dividido em dois até que se encontre um bloco do tamanho requisitado, ou imediatamente maior (potência de 2)
  - Blocos vizinhos são chamados de *buddy*
  - O endereço do bloco mais adequado é retornado

Gerenciamento de Memória

Estratégia #03: *BUDDY* (KNUTH)



Gerenciamento de Memória

Estratégia #03: *BUDDY* (KNUTH)

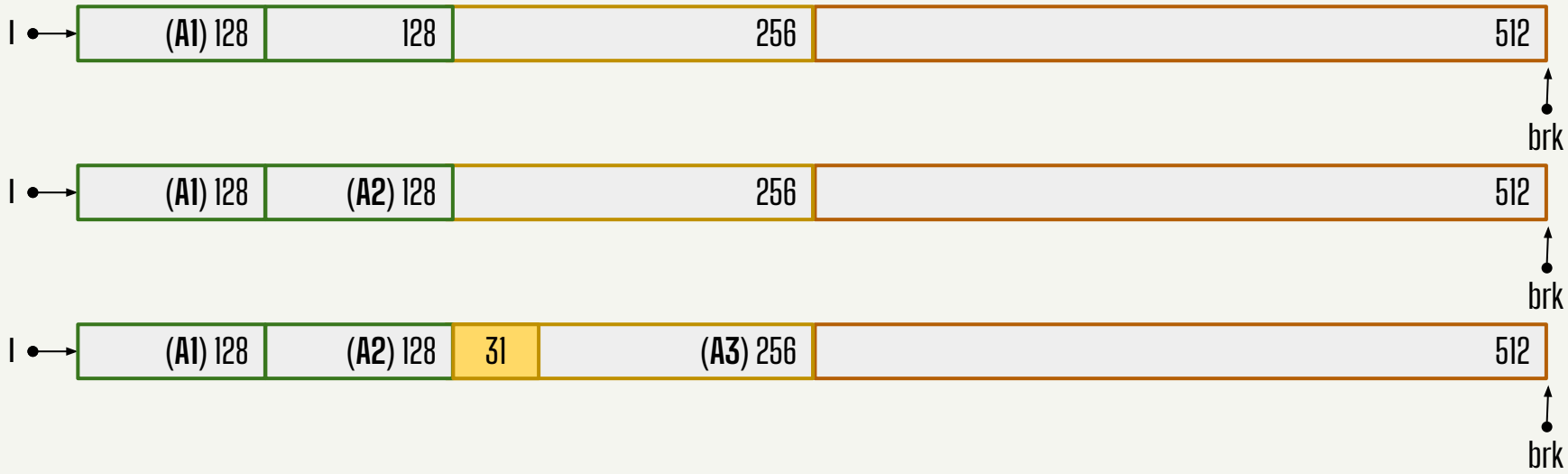
A divisão ocorre quantas vezes forem necessárias, dado o pedido de alocação.
Se considerarmos a seguinte sequência:

128 (A1)**; 128** (A2)**; 225** (A3)**; 256** (A4)

Teremos o seguinte...

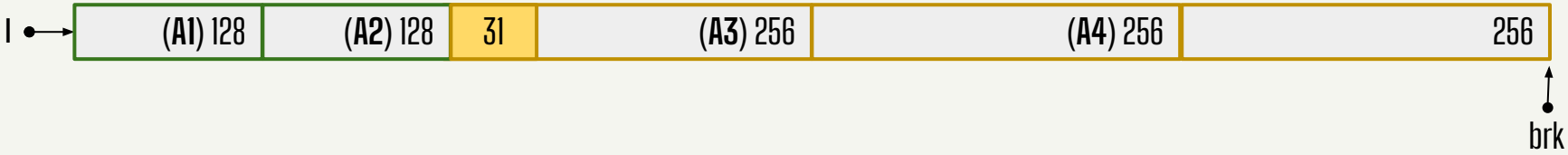
Gerenciamento de Memória

Estratégia #03: *BUDDY* (KNUTH)



Gerenciamento de Memória

Estratégia #03: *BUDDY* (KNUTH)

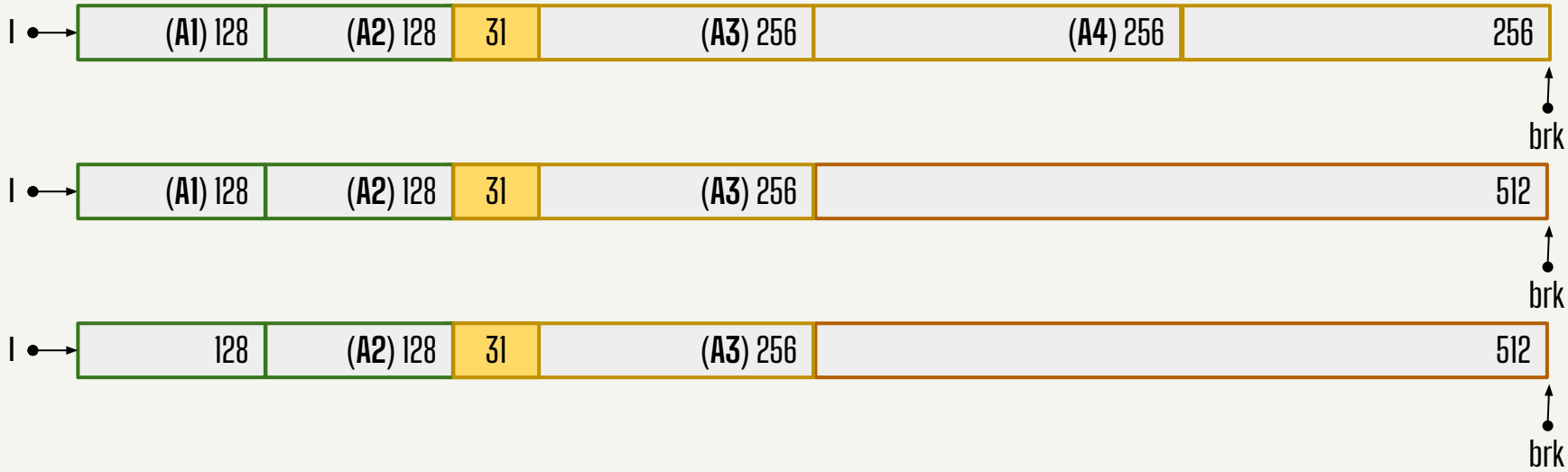


Para a desalocação, os blocos **são unidos ao seu respectivo *buddy***, caso os seus dados não estejam alocados para nenhum bloco.
Vamos considerar a seguinte ordem de desalocação:

A4; A1; A3; A2

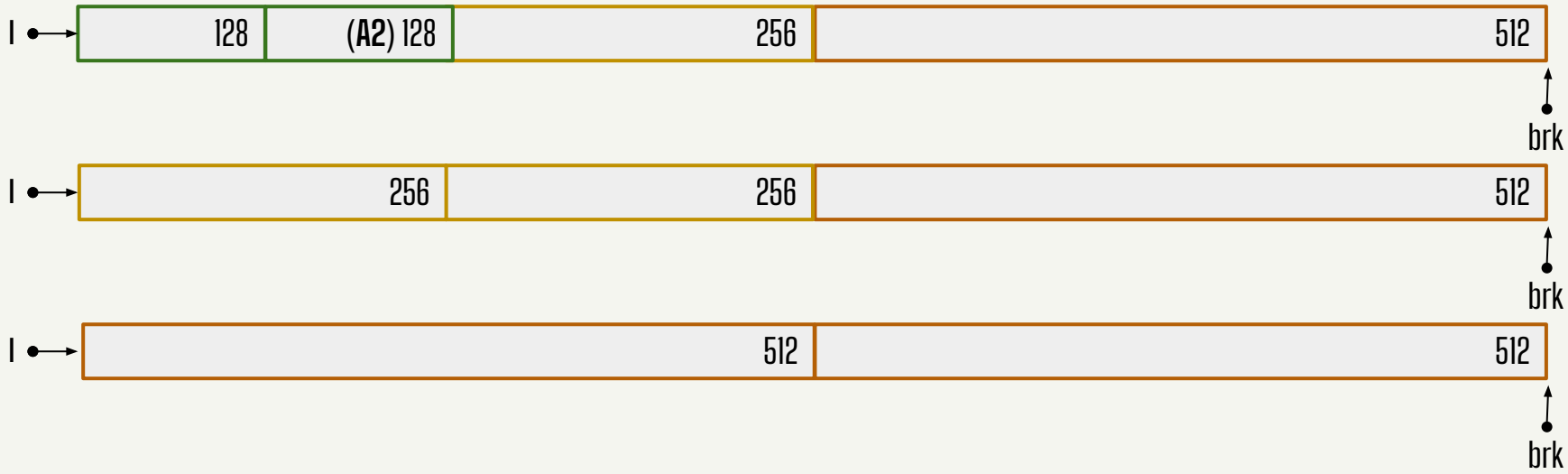
Gerenciamento de Memória

Estratégia #03: *BUDDY* (KNUTH)



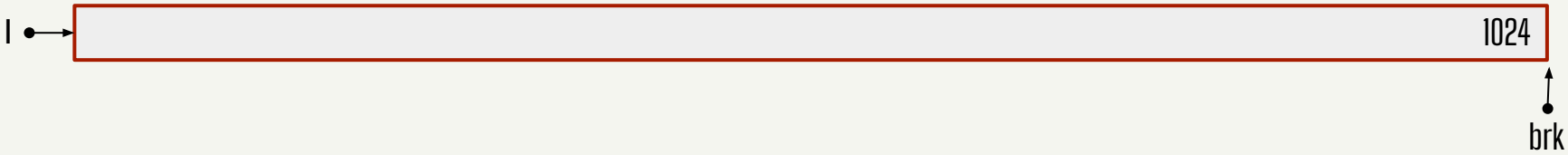
Gerenciamento de Memória

Estratégia #03: *BUDDY* (KNUTH)



Gerenciamento de Memória

Estratégia #03: *BUDDY* (KNUTH)



Como todos os blocos foram desalocados, o espaço de memória retorna ao originalmente alocado.

Estratégia eficiente: logarítmica

Problema: fragmentação interna

Erros Comuns

```
int main (int argc , char** argv){
  void *a;
  int i;
  for (i=0; i <100; i ++) {
    a = malloc (100);
    strcpy (a , " TESTE "'<);
    printf (" %p %s \n " , a , (char*) a );
  }
  free(a);
  return(0);
}
```

Analise o código ao lado...

Existe um problema em relação à alocação de memória neste código.

Qual?

Erros Comuns

```
int main (int argc , char** argv){
  void *a;
  int i;
  for (i=0; i <100; i ++) {
    a = malloc (100);
    strcpy (a , " TESTE "'<);
    printf (" %p %s \n " , a , (char*) a );
  }
  free(a);
  return(0);
}
```

Analise o código ao lado...

Existe um problema em relação à alocação de memória neste código.

Qual?

MEMORY LEAK

Erros Comuns

```c
int main(int argc, char **argv) {
  char *p, *q;
  int i;
  p = malloc(TAMANHO);
  q = malloc(TAMANHO);
  if (argc >= 2) strcpy(p, argv[1]);
  free(q);
  free(p);
  return 0;
}
```

Analise o código ao lado...

Existe um problema **EM POTENCIAL** em relação à alocação de memória neste código.

Qual?

Erros Comuns

```c
int main(int argc, char **argv) {
  char *p, *q;
  int i;
  p = malloc(TAMANHO);
  q = malloc(TAMANHO);
  if (argc >= 2) strcpy(p, argv[1]);
  free(q);
  free(p);
  return 0;
}
```

Analise o código ao lado...

Existe um problema **EM POTENCIAL** em relação à alocação de memória neste código.

Qual?

BUFFER OVERFLOW

Erros Comuns

```c
int main(int argc, char **argv) {
  char *p, *q, *t;
  int i;
  p = malloc(1024);
  q = malloc(1024);
  for (i=0; i<100; i++){
    free (p);
    p = malloc(1);
    p = malloc(1024);
    t=p; p=q; q=t;
  }
  free(q);
  free(p);
  return 0;
}
```

Analise o código ao lado...

Existe um problema **(QUE NÃO É O *MEMORY LEAK*)** em relação à alocação de memória neste código.

Qual?

Erros Comuns

```c
int main(int argc, char **argv) {
  char *p, *q, *t;
  int i;
  p = malloc(1024);
  q = malloc(1024);
  for (i=0; i<100; i++){
    free (p);
    p = malloc(1);
    p = malloc(1024);
    t=p; p=q; q=t;
  }
  free(q);
  free(p);
  return 0;
}
```

Analise o código ao lado...

Existe um problema **(QUE NÃO É O *MEMORY LEAK*)** em relação à alocação de memória neste código.

Qual?

FRAGMENTAÇÃO

Coletor de Lixo

O coletor de lixo é um programa existente no contexto de diversas linguagens de programação, como Java e Python (principalmente as orientadas a objetos).

Quando em execução, o coletor de lixo tem dois objetivos principais:

- Reduzir a fragmentação da *heap*
- Desalocar memória cuja referência foi perdida

ESSAS TAREFAS SÃO REALIZADAS EM TEMPO DE EXECUÇÃO!

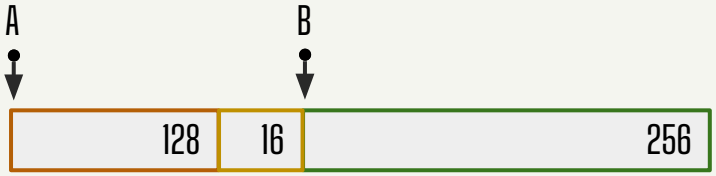
Coletor de Lixo

Implementar um coletor de lixo em C é um tanto complicado, já que as variáveis de ponteiro **indicam endereços absolutos na *heap***, ou seja, o endereço de um ponteiro pode ser calculado em função de outro ponteiro.

Já em Java, por exemplo, existe uma tabela de endereços, e o **equivalente a um ponteiro em Java não aponta diretamente um endereço**, mas sim um índice desta tabela.

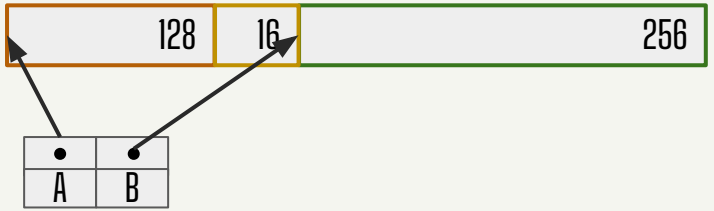
POR QUE ISSO FACILITA TODO O PROCESSO?

Coletor de Lixo



Caso C

Mudanças devem ser feitas diretamente em todas as variáveis após A



Caso JAVA

Mudanças centralizadas na tabela

Exercício #16

Implemente um procedimento em *assembly* AMD64 que **aloque dinamicamente um vetor com uma certa quantidade (parâmetro da função) de letras "A"**, retornando uma *string* correspondente.

O procedimento principal do programa deve **executar o procedimento descrito anteriormente e exibir o resultado na tela** usando a função printf (libc).

Obs: verifique o efeito da função printf no brk.

Obrigado!

Vinícius Fülber Garcia
inf.ufpr.br/vinicius/
viniciusfulber@ufpr.br