INE5412 Sistemas Operacionais I

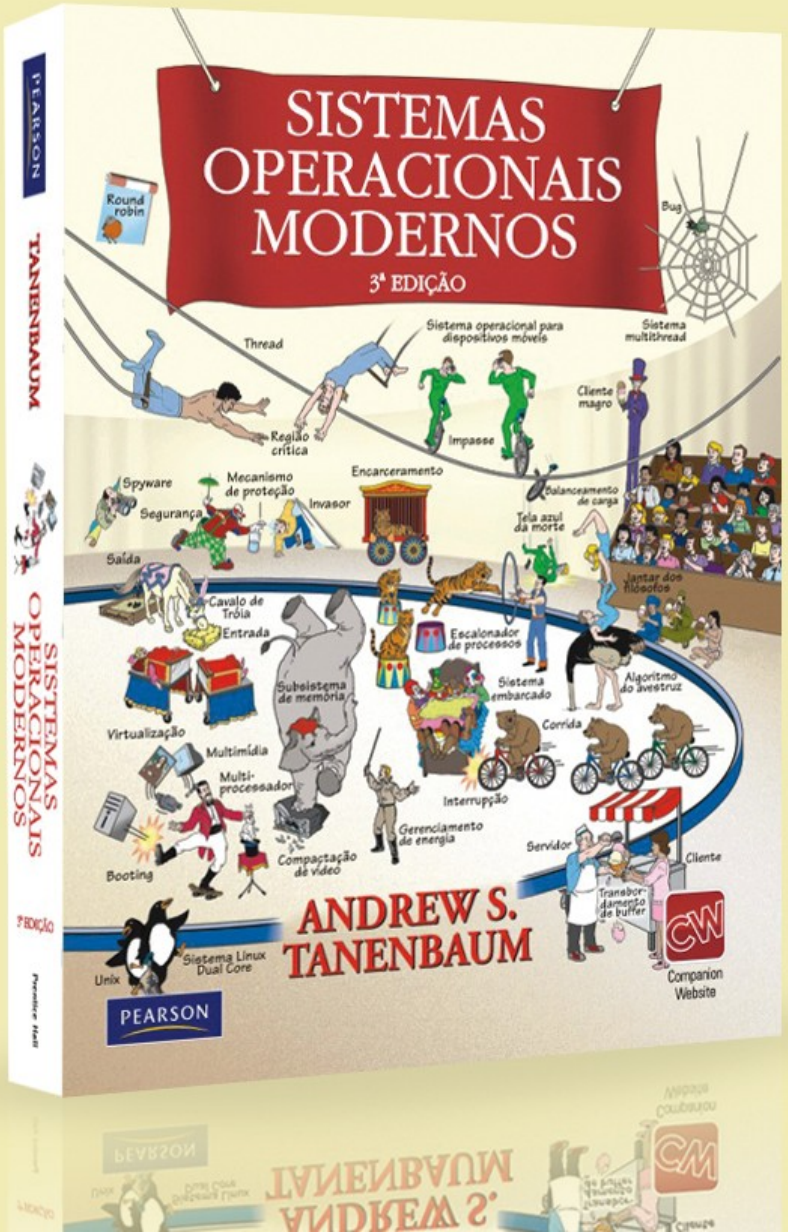
L. F. Friedrich

Gerenciamento de Memória: Introdução

Sistemas operacionais modernos
Terceira edição

ANDREW S. TANENBAUM

Capítulo 3
Gerenciamento de memória



Gerenciamento de Memória

- Memória principal (RAM) é um recurso importante que deve ser gerenciado com muito cuidado.
- Lei de Parkinson: “programas tendem a se expandir a fim de ocupar toda a memória disponível”
- Soluções: memória infinitamente grande, rápida e não volátil (também a baixo custo) ou a utilização do conceito de **hierarquia de memórias**
- Função do sistema operacional é abstrair a hierarquia em um modelo útil e gerenciar a abstração.
- Abstração mais simples?

GM – Processo

- De acordo com a sua experiência escrevendo programas, podes responder as seguintes questões?

Memória	Espaço total fixo?	Só leitura?
Constantes		
Variáveis globais		
Variáveis locais		
Alocação dinâmica		
Código programa		

GM – Processo

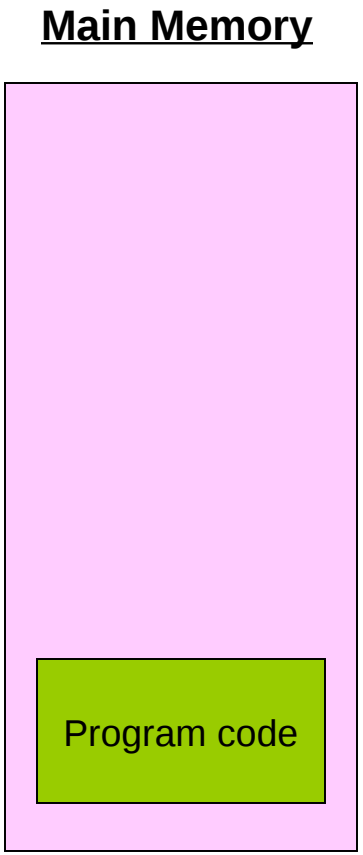
- De acordo com a sua experiência escrevendo programas, podes responder as seguintes questões?

Memória	Espaço total fixo?	Só leitura?
Constantes	**Sim**	**Sim**
Variáveis globais	**Sim**	**Não**
Variáveis locais	**Não**	**Não**
Alocação dinâmica	**Não**	**Não**
Código programa	**Não***	**Não***

* Reference: http://en.wikipedia.org/wiki/Self-modifying_code

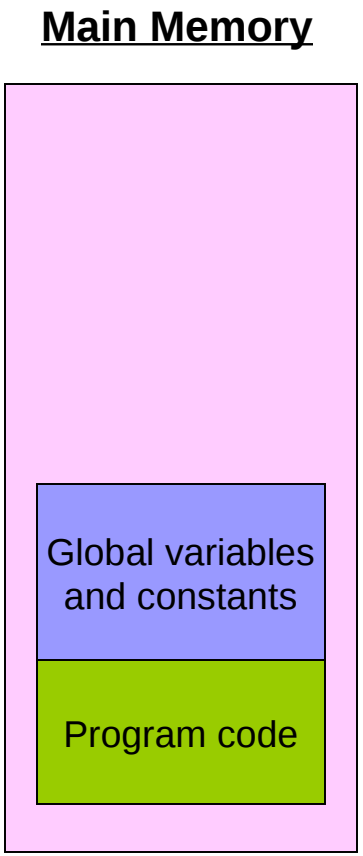
Processo e Programa

- Um processo esta sempre limitado a um programa.
  - Um processo pode mudar esta relação usando a syscall exec.
  - É necessário que o código do programa executando esteja na memória.
    - Porque?
    - A CPU precisa buscar as instruções para a execução.



Variáveis Globais e Constantes

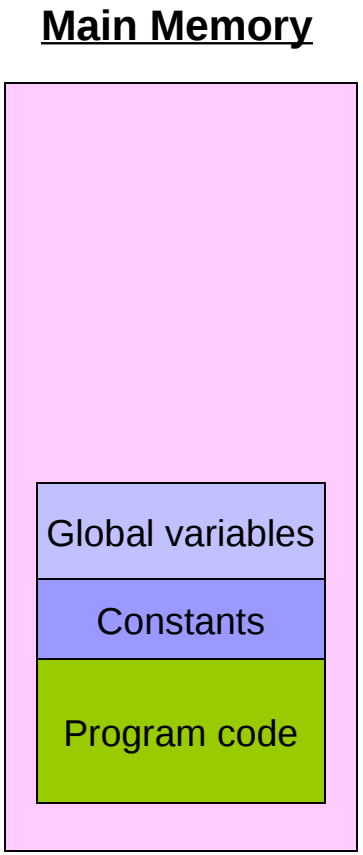
- Variáveis globais e constantes são anexados ao código do programa.
  - O número de variáveis globais e constantes é fixado em tempo de compilação.
  - Quando um novo código de programa inicia execução (por uma syscall exec), as variáveis globais e as constantes são criadas na memória.



Variáveis Globais e Constantes

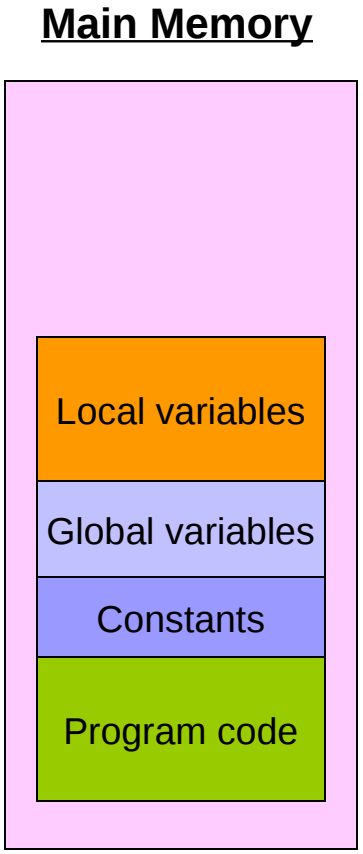
- Variáveis globais e constantes recebem na verdade diferentes gerenciamentos!
  - Constantes são “read-only” e devem ser protegidas.
  - Tente isto:

```
int main (void) {
    char *string = “hello”;
    string[0] = ‘\0’;
    return 0;
}
```



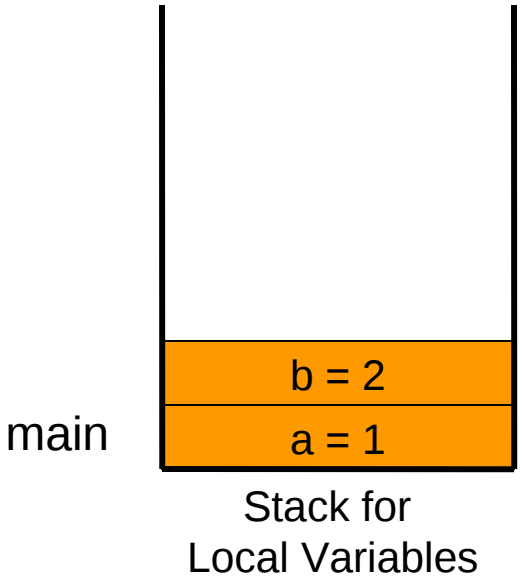
Variáveis locais e Programas

- Variáveis locais estão associadas apenas a funções (procedimentos).
  - Quando uma função é chamada, as variáveis locais são produzidas.
  - Quando uma função retorna, as variáveis locais são destruídas.
  - Durante chamadas de função, não existe envolvimento do kernel. O código compilado contém todos os controles.
    - Detalhes, ver código assembly.
    - Usando “gcc –S any_program.c”

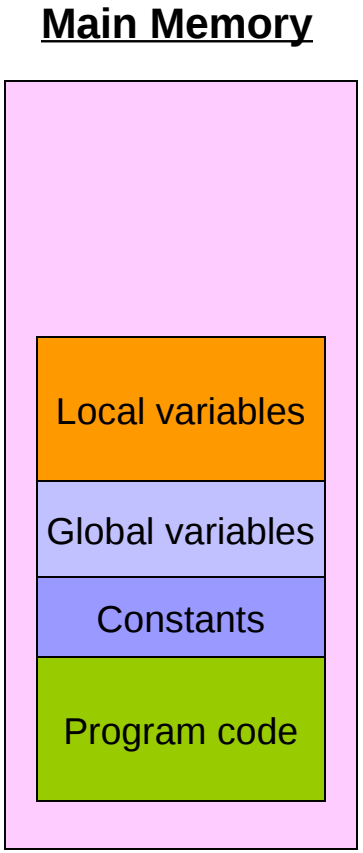


Variáveis locais e Programas

- Variáveis locais são organizadas em pilhas.
  - Quando o “main” é invocado, suas variáveis locais são colocadas na pilha.

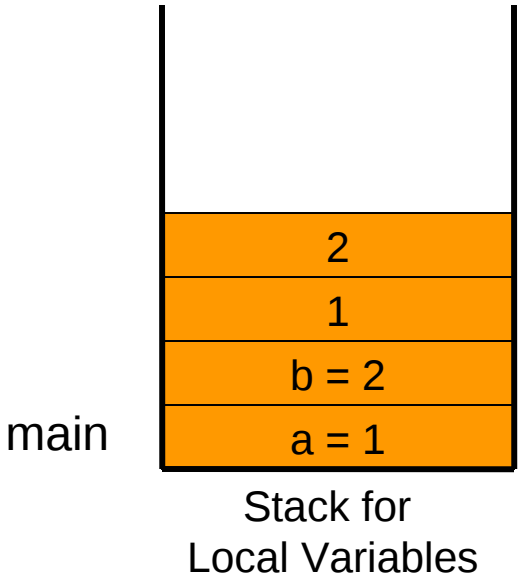



```
int fun2(int x, int y) {
    int c = 10
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(u, v);
}

int main(void) {
    int a = 1, b = 2;
    fun1(a, b);
    return 0;
}
```

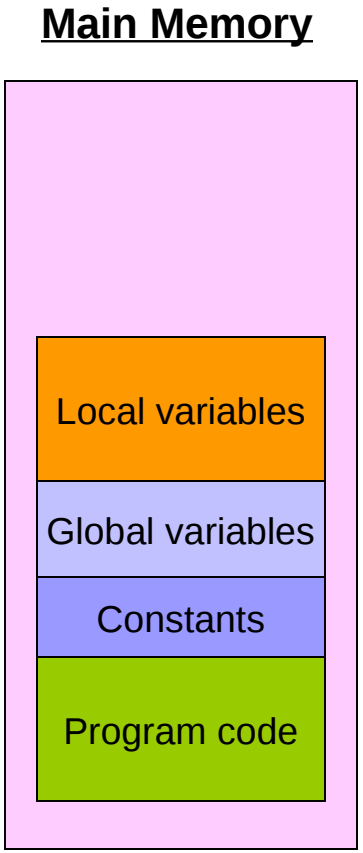


Variáveis locais e Programas

- Passagem de parâmetro também é feito via pilha.
  - Antes de o PC pular para “fun1”, os parâmetros são colocados na pilha.

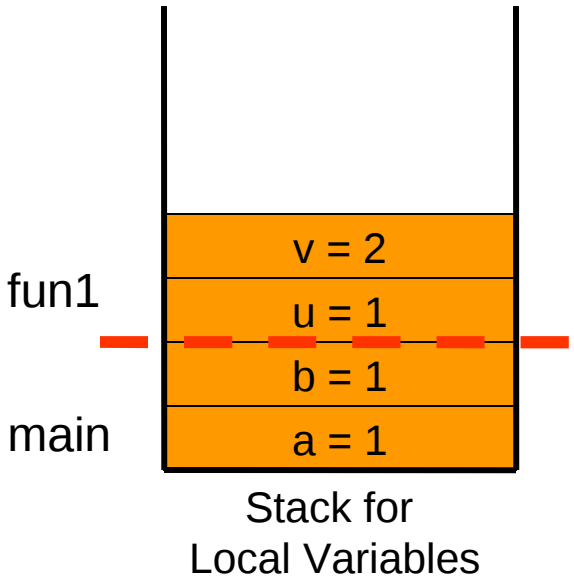



```
int fun2(int x, int y) {
    int c = 10
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(u, v);
}

int main(void) {
    int a = 1, b = 2;
    fun1(a, b);
    return 0;
}
```

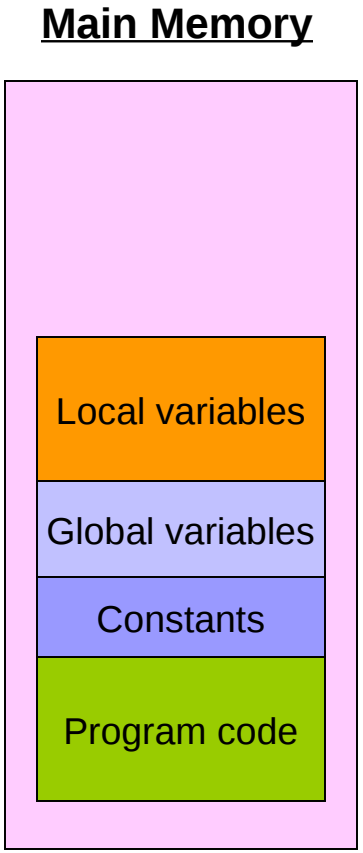


Variáveis locais e Programas

- Uma função refere-se aos parâmetros como se refere as variáveis locais.
  - Todos estão na pilha.




```
int fun2(int x, int y) {
    int c = 10
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(u, v);
}

int main(void) {
    int a = 1, b = 2;
    fun1(a, b);
    return 0;
}
```

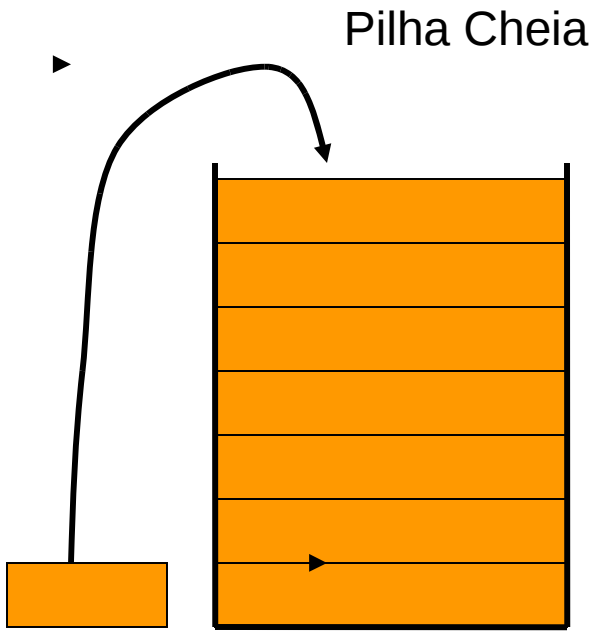


Variáveis locais e Programas

- Note que:
  - O tamanho total das variáveis locais muda dinamicamente.
  - O compilador não pode estimar o tamanho da pilha em tempo de compilação.
    - Porque o mumero de chamadas de função depende de: estado do programa, entradas do usuário, etc.
    - O que o kernel pode fazer é reservar um espaço suficientemente grande para a pilha.

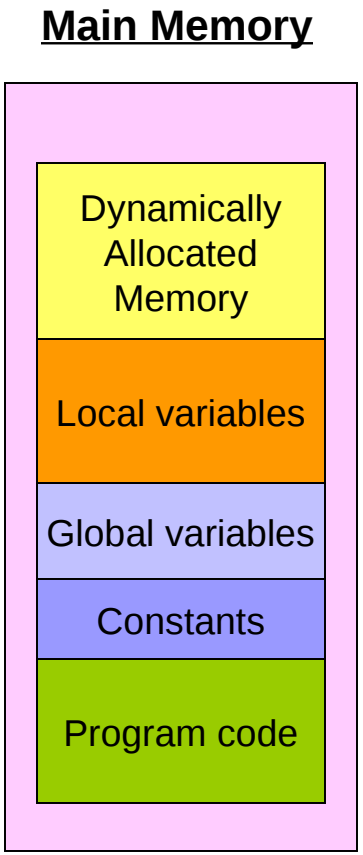
Variáveis locais e Programas

- O que acontece se:
  - Não existe mais espaço na pilha?
  - Aumentar a pilha?
  - O que acontece em uma cadeia sem fim de chamadas de funções recursivas?
- Solução:
  - Exception caught!
  - Stack overflow exception!
  - Program terminated!



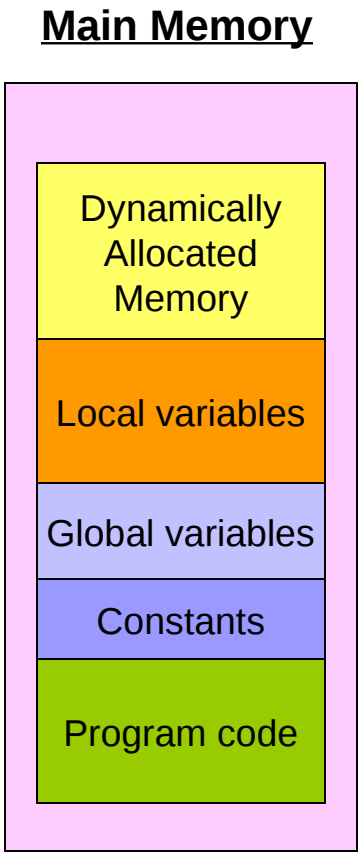
Memoria Alocada Dinamicamente

- Memoria alocada dinamicamente esta restrita ao programa todo.
  - O espaço é também chamado de heap.
  - Uma vez alocada, a memoria pode ser acessada por toda função do programa.
  - Diferente das variáveis locais, a alocação da memoria é implementada pelo kernel.



Memoria Alocada Dinamicamente

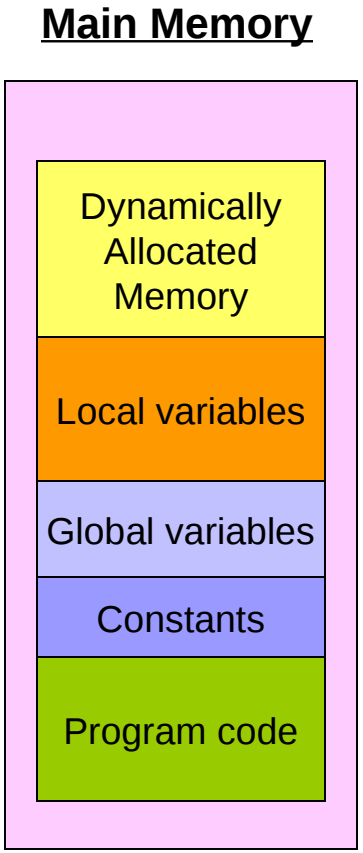
- O kernel realmente regula toda a memoria no sistema.
  - Quando a função malloc() é chamada, ela tenta encontrar memoria.
- A propósito:
  - É possível fazer o sistema executar out of memory (OOM)?



Memoria Alocada Dinamicamente

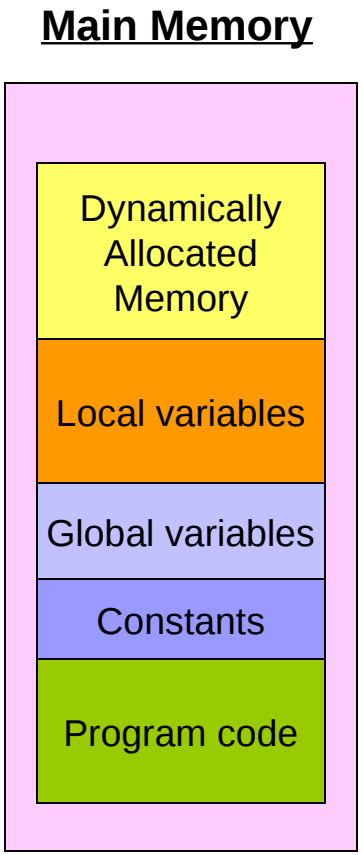
- OOM generator!

```
#define ONE_MEG  1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```



Memoria Alocada Dinamicamente

- No Linux, onde o OOM generator irá parar? Porque?
  - Razão:
    - Todo sistema de 32-bits tem espaço de memoria endereçável de 4G-1 bytes.
    - O kernel reserva cerca de 1GBytes.
    - Assim, 4G – 1G ¼ ~ 3G.



Memoria Alocada Dinamicamente

- O que o malloc() faz realmente?
  - Primeiro, malloc() não é uma chamada de sistema!
    - malloc() invoca a chamada de sistema brk().
  - Na verdade, malloc() é uma chamada de biblioteca!
    - Alocação de memoria é feita por brk().
    - malloc() apenas gerencia o espaço livre retornado do kernel.

http://www.ibm.com/developerworks/linux/library/l-memory/