Linguagem C
- Zonas de Memória -

Arquitetura de Computadores 2024/2025

Variáveis Globais

- Até agora falámos de duas maneiras diferentes de alocar memória:
  - Declaração de variáveis locais

    ```
    int i; char *string; int ar[n];
    ```
  - Alocação dinâmica em runtime usando "malloc"

    ```
    ptr = (struct Node *) malloc(sizeof(struct Node)*n);
    ```

- Existe uma terceira possibilidade ...
  - Declaração de variáveis fora de qualquer função (i.e. antes do `main`)

```
int myGlobal;
main() {
}
```

    - É similar às variáveis locais mas tem um âmbito global, podendo ser lida e escrita a partir de qualquer ponto do programa

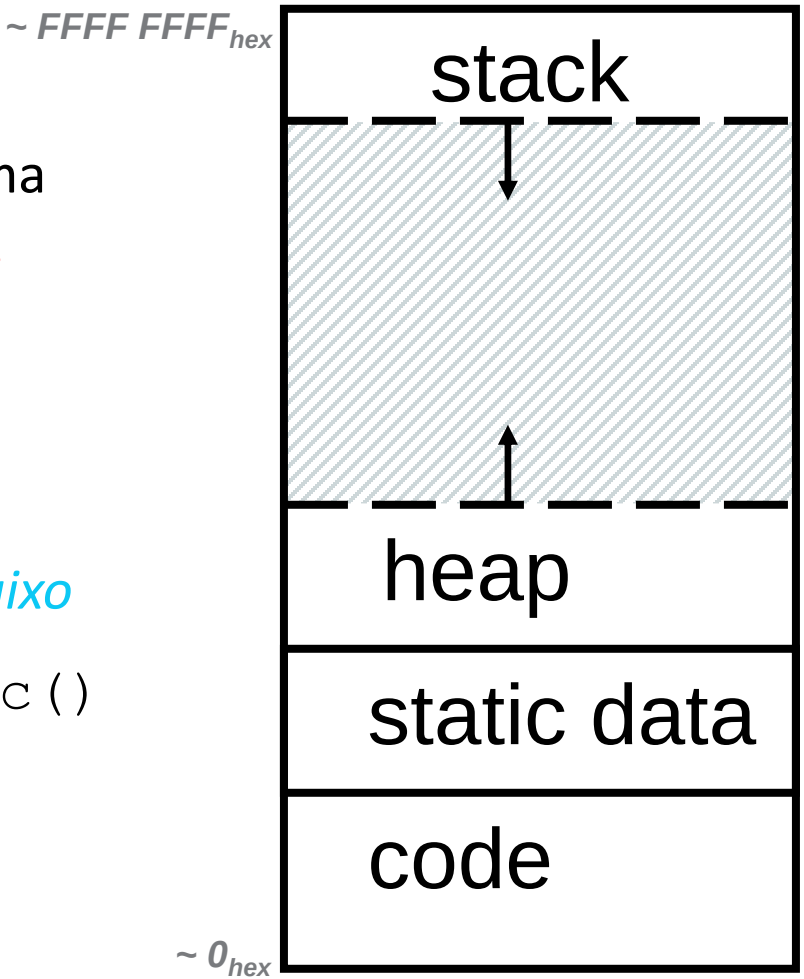
Gestão de Memória em C (1/2)

- Um programa em C define três zonas de memória distintas para o armazenamento de dados:
  - Armazenamento Estático/*Static Storage*: onde ficam as variáveis globais que podem ser lidas/escritas por qualquer função do programa. Este espaço está alocado permanentemente durante todo o tempo em que o programa corre (daí o nome estático)
  - A Pilha/*Stack*: armazenamento de variáveis locais, parâmetros, endereços de retorno, etc.
  - A Heap (*dynamic malloc storage*): os dados são válidos até ao instante em que o programador faz a desalocação manual com `free()`.
- O C precisa de conhecer a localização dos objectos na memória, senão as coisas não funcionam correctamente.

Gestão de Memória em C (2/2)

- O *espaço de endereçamento* de um programa contém 4 regiões:
  - código: Carregado quando o programa começa, o tamanho não se modifica.
  - Dados estáticos: variáveis globais declaradas fora do main(), *tamanho constante durante a execução.*
  - stack: variáveis locais, *cresce para baixo*
  - heap: espaço requisitado via `malloc()` ; *cresce para cima.*

O Sistema Operativo evita a sobreposição da Stack com a Heap



Onde é que as variáveis são alocadas?

- Se são declaradas fora de qualquer função/procedimento, então são alocadas na zona estática.

- Se são declaradas dentro da função, então são alocadas na pilha sendo o espaço libertado quando o procedimento termina.

  – Nota: `main()` é uma função

```
int myGlobal;

main() {
   int myTemp;
}
```

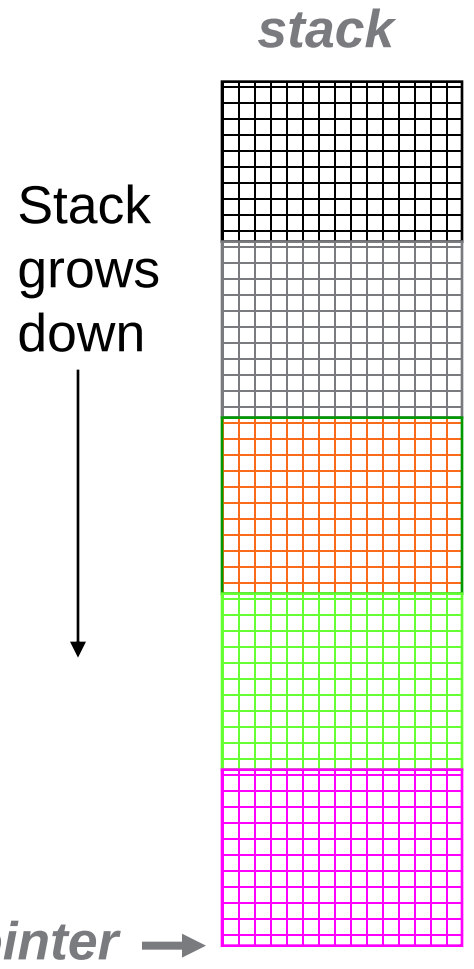
A Pilha/Stack (1/2)

- Um "*Stack Frame*" inclui:
  - Endereços de retorno
  - Parâmetros
  - Espaço para variáveis locais
- Os "*Stack frames*" são blocos contíguos de memória; o "*stack pointer*" indica qual é o "*frame*" no topo da pilha
- Quando uma rotina termina, o seu "*stack frame*" é descartado (não explicitamente apagado). Isto permite libertar memória para futuras utilizações

A Pilha/Stack (2/2)

- Last In, First Out (LIFO) data structure

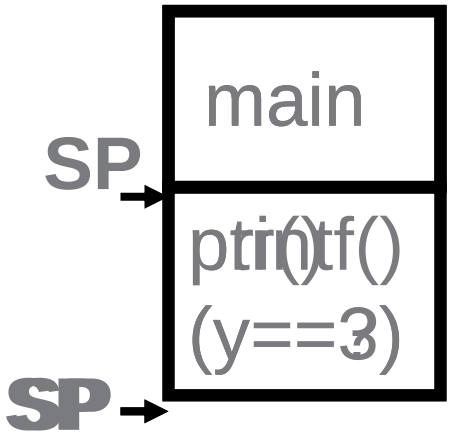
```
main ()
{ a(0);
}
  void a (int m)
  { b(1);
  }
    void b (int n)
    { c(2);
    }
      void c (int o)
      { d(3);
      }
        void d (int p)
        {
        }
```



Quem gere a pilha ?

- Os ponteiros em C permitem-nos aceder a zonas de memória que foram entretanto desalocadas. Isto pode levar a problemas de consistência e bugs difíceis de encontrar !

```
int *ptr () {
    int y;
    y = 3;
    return &y;
};
main () {

    int *stackAddr,content;
    stackAddr = ptr();
    content = *stackAddr;
    printf("%d", content); /* 3 */
    content = *stackAddr;
    printf("%d", content); /*13451514 */
};
```



A Heap (Memória Dinâmica)

- Grande bloco de memória, onde a alocação não é feita de forma contígua. É uma espécie de "*espaço comunitário*" do programa.

- Em C, é necessário especificar o número exacto de bytes que se pretende alocar:

```
int *ptr;
ptr = (int *) malloc(sizeof(int));
/* malloc returns type (void *),
so need to cast to right type */
```

  - `malloc()`: aloca memória não inicializada na área da *heap*

Características das diferentes zonas de memória

- Variáveis estáticas
  - Espaço de memória acessível a partir de qualquer zona do programa
  - O espaço de memória permanece alocado durante todo o "*runtime*" (pouco eficiente)

- Pilha/Stack
  - Guarda variáveis locais, endereços de retorno, etc.
  - A memória é libertada sempre que uma função termina a sua execução, permitindo a reutilização da memória para uma nova função
  - Funciona como o "bloco de notas" das funções/procedimentos
  - Não é adequada para armazenar dados de grandes dimensões (*stack overflow*)
  - Não permite a partilha de dados entre diferentes procedimentos

Características das diferentes zonas de memória

- Heap / Alocação dinâmica
  - Alocação em "*runtime*" de blocos de memória
  - A alocação não é contígua, e os blocos podem ficar muito distantes no espaço de endereçamento
  - Em C, a libertação de memória tem de ser feita de forma explícita pelo programador (não existe nenhum *Garbage Collector*)
  - Os mecanismos de gestão de memória são complexos de forma a evitar a fragmentação

Gestão de Memória

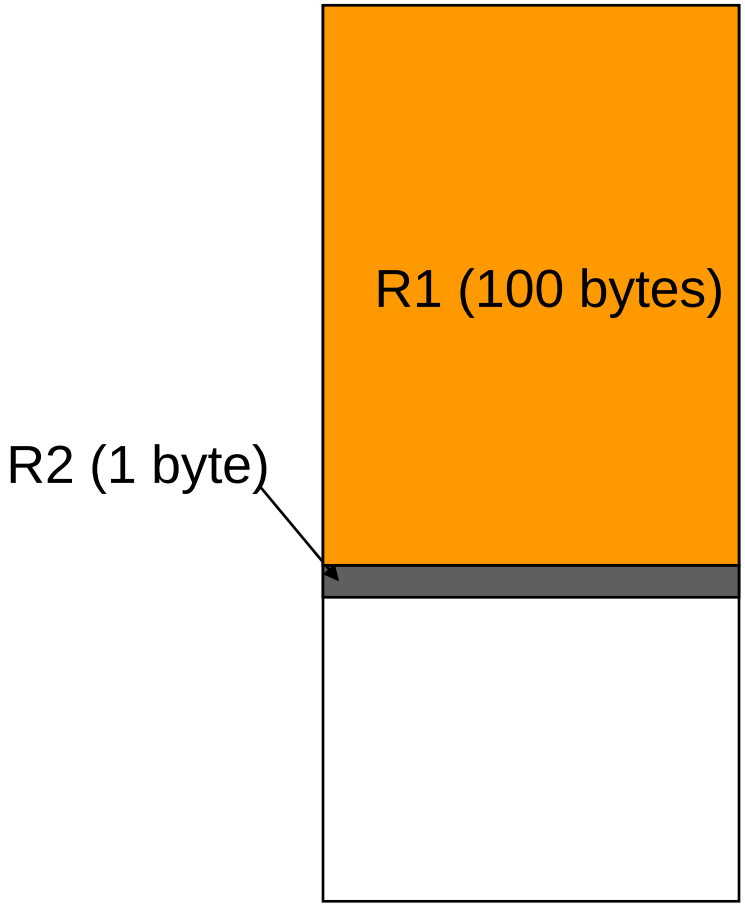
- Como é feita a gestão de memória?
  - Zona do código e variáveis estáticas é fácil:
    estas zonas nunca aumentam ou diminuem
  - O espaço da pilha também é fácil:
    As "stack frames" são criadas e destruídas usando uma ordem last-in, first-out (LIFO)
  - Gerir a heap já é mais complicado:
    a memória pode ser alocada / desalocada em qualquer instante

Requisitos da Gestão da Heap

- As funções `malloc()` e `free()` devem executar rapidamente.
- Pretende-se o mínimo de *overhead* na gestão de memória
- Queremos evitar *fragmentação (externa)** – quando a maior parte da memória está dividida em vários blocos pequenos
  - Neste caso podemos ter muito bytes disponíveis mas não sermos capazes de dar resposta a uma solicitação de espaço porque os bytes livres não são contíguos.

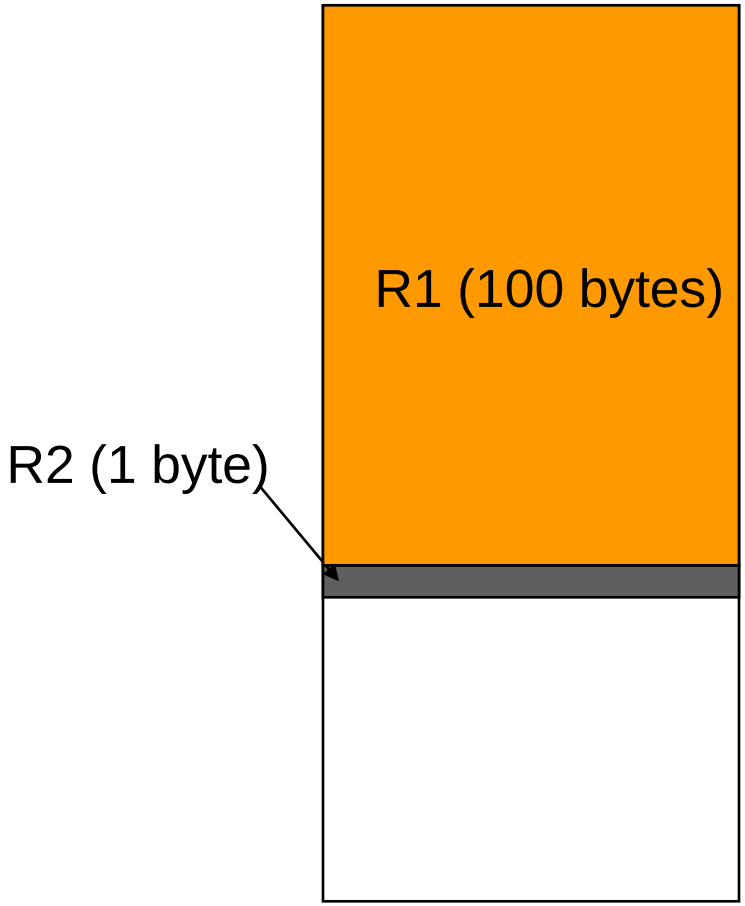
Gestão da Heap (1/2)

- Exemplo
  - Pedido R1 com 100 bytes
  - Pedido R2 com 1 byte
  - Memória de R1 é libertada
  - Pedido R3 com 50 bytes



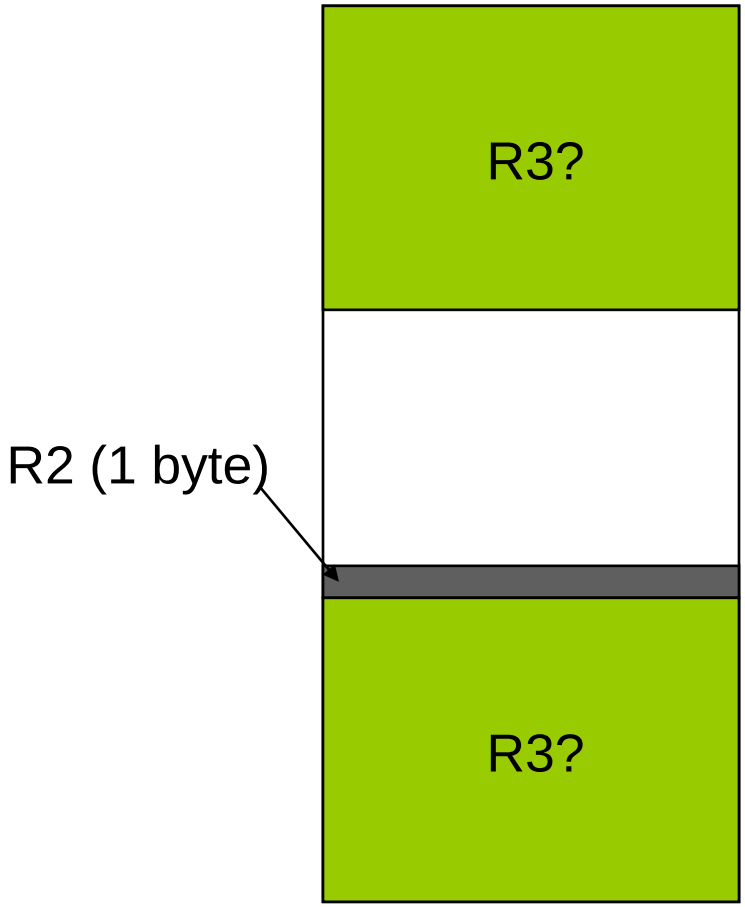
Gestão da Heap (1/2)

- Exemplo
  - Pedido R1 com 100 bytes
  - Pedido R2 com 1 byte
  - Memória de R1 é libertada
  - Pedido R3 com 50 bytes



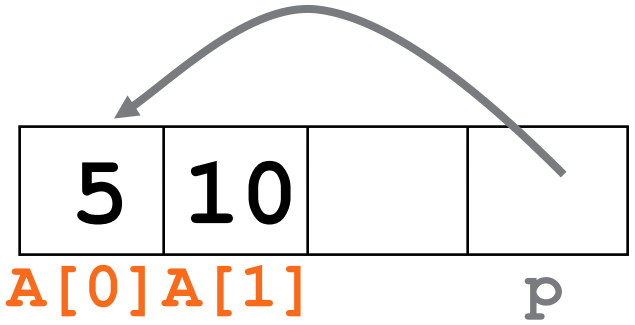
Gestão da Heap (2/2)

- Exemplo
  - Pedido R1 com 100 bytes
  - Pedido R2 com 1 byte
  - Memória de R1 é libertada
  - Pedido R3 com 50 bytes



QUIZ

```
int main(void){
  int A[] = {5,10};
  int *p = A;

  printf("%u %d %d %d\n",p,*p,A[0],A[1]);
   p =  p + 1;
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);
  *p = *p + 1;
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);
  }
```



Se o primeiro `printf` mostrar `100 5 5 10`, qual será o output dos outros dois `printf` ?

```
1: 101 10 5 10        depois 101 11 5 11
2: 104 10 5 10        depois 104 11 5 11
3: 101 <other> 5 10   depois 101 <3-others>
4: 104 <other> 5 10   depois 104 <3-others>
5: Um dos dois printfs causa um ERRO
6: Desisto!
```

Introdução ao MIPS
- Funções e Procedimentos -

Arquitetura de Computadores 2024/2025

Funções em C

```
main() {
   int i,j,k,m;
   ...
   i = mult(j,k); ...
   m = mult(i,i); ...
}
```

Numa chamada a função, que informação é que o compilador/programador precisa de registar ?

```
/* ineficiente implementação de mult */
int mult (int mcand, int mlier){
   int product;
   product = 0;
   while (mlier > 0){
        product = product + mcand;
        mlier = mlier -1;
   }
   return product;
}
```

Que instruções permitem fazer isto?

Chamada de funções (Revisões)

- No MIPS os registos são fundamentais para guardar a informação necessária à chamada de funções.

- Convenção de utilização de registos:
  - Endereço de retorno. `$ra`
  - Argumentos / Parâmetros: `$a0, $a1, $a2, $a3`
  - Retorno de valores: `$v0, $v1`
  - Variáveis locais: `$s0, $s1, … , $s7`

- Veremos mais tarde que a *stack* também é utilizada.

Instruções de suporte a funções (2/6)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
 return x+y;
}
```

MIPS

address

```
1000 add  $a0,$s0,$zero  # x = a
1004 add  $a1,$s1,$zero  # y = b
1008 addi $ra,$zero,1016 #$ra=1016
1012 j  sum              #jump to sum
1016 ...
2000 sum: add $v0,$a0,$a1
2004 jr   $ra # salta para o endereço
              # apontado pelo registo
```

Instruções de suporte a funções (3/6)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
 return x+y;
}
```

MIPS

- Pergunta: Porquê utilizar `jr`? Porque não `j`?
- Resposta: A função `sum` pode ser chamada de muitos sítios diferentes. Assim, não podemos regressar para um endereço fixo pré-definido. É preciso disponibilizar um mecanismo para dizer "regressa aqui" !

```
2000 sum: add $v0,$a0,$a1
2004      jr   $ra    # new instruction
```

Instruções de suporte a funções (4/6)

- Instrução para simultaneamente saltar e fazer a salvaguarda do endereço de retorno: jump and link (`jal`)

- Sem jal:

```
1008 addi $ra,$zero,1016  #$ra=1016
1012 j sum                #goto sum
```

- Com jal:

```
1008 jal sum              # $ra=1012,goto sum
```

- Será que `jal` é imprescindível?
  - "Make the common case fast": a chamada a funções é uma operação muito frequente.
  - Para além disso, com `jal` o programador não precisa de saber onde é que o código vai ser carregado.

Instruções de suporte a funções (5/6)

- A sintaxe do `jal` (jump and link) é semelhante à do `j` (jump):

```
jal label
```

- Na verdade o `jal` deveria ser chamado `laj` (link and jump):
  - Passo 1 (link) - Guarda o endereço da próxima instrução em `$ra`
  - Passo 2(jump) - Salta para a instrução assinalada por label

- Porque é que é guardado o endereço da instrução seguinte em vez da instrução actual?

Instrução de Suporte a Funções (6/6)

- Sintaxe do `jr` (*jump register*):

```
jr register
```

- Em vez de darmos um "*label*" ao *jump*, passamos um registo que contém o endereço para onde queremos saltar.

- Estas duas instruções são muito úteis para chamada de funções:
  - `jal` guarda o endereço de retorno no registo (`$ra`)
  - `jr $ra` salta de volta para o sítio onde a função foi chamada (se entretanto não alterarmos o conteúdo do registo)

Funções em Cascata (1/2)

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y;
}
```

- Alguém chamou `sumSquare`, e agora `sumSquare` está a chamar `mult`.
- Assim o endereço que está em `$ra` é o sítio para onde `sumSquare` vai ter de regressar. No entanto, esse registo vai ser alterado/escrito pela chamada a `mult`.
- Vamos ter de guardar o endereço de retorno de `sumSquare` antes de fazer a chamada a `mult`.

Funções em Cascata (2/2)

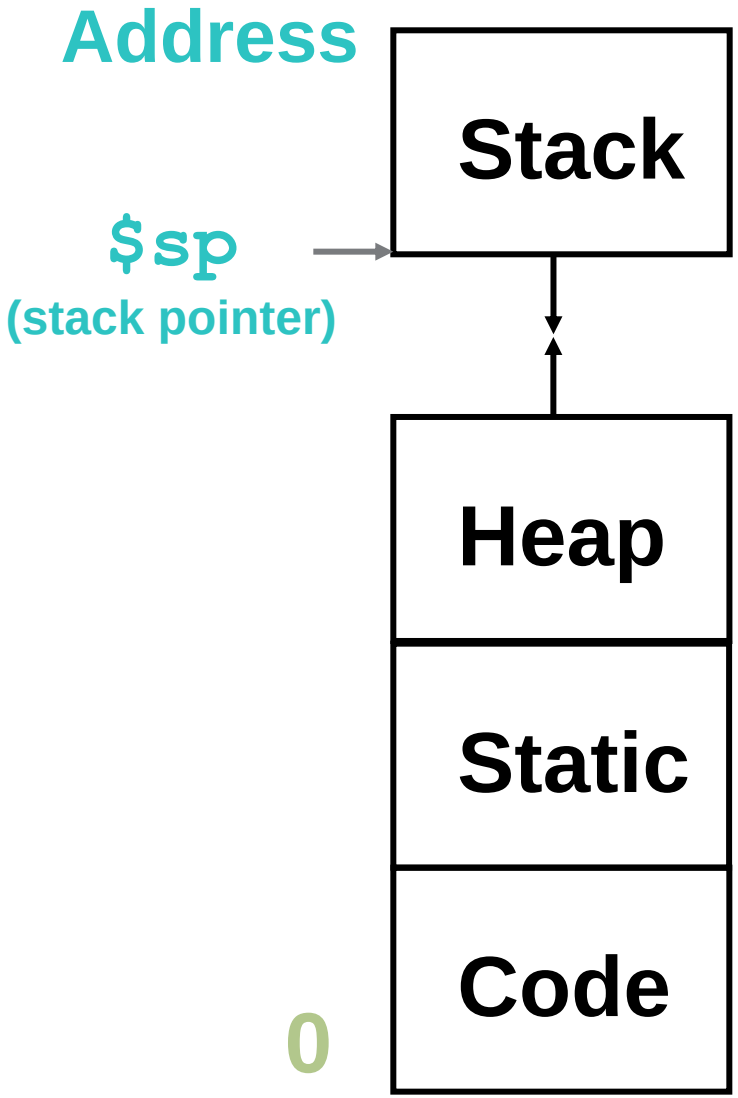
- Iremos ver mais à frente que normalmente precisamos de guardar outras informações para além do conteúdo de `$ra`.

- Onde será que podemos guardar essa informação?

- Quando um programa em C está a correr, existem 3 zonas diferentes de memória:
  - Static: Variáveis declaradas uma única vez no início do programa. Esta zona só é desalocada quando o programa termina.
  - Heap: Variáveis declaradas de forma dinâmica
  - Stack: Espaço para ser utilizado por funções/procedimentos durante a execução.

Este é a zona onde fazemos a salvaguarda de contexto!

Revisão da alocação de memória em C



Espaço para guardar informação dos procedimentos

Espaço criado explicitamente, e.g. malloc(); ponteiros do C

Variáveis Globais

Programa

Utilização da Pilha (1/2)

- O registo `$sp` contém sempre o endereço da última zona de memória que está a ser ocupada pela *stack* (topo da pilha ... ou melhor, fundo da pilha!).

- Para utilizar a pilha, devemos decrementar o ponteiro `$sp` pelo número de bytes que vamos precisar para guardar a informação.

- Como é que devemos então compilar o programa?

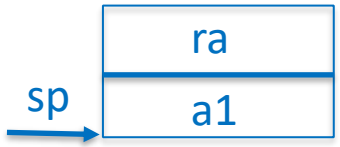
```
int sumSquare(int x, int y) {
  return mult(x,x)+ y;
 }
```

Utilização da Pilha (2/2)

- Compile "à mão a"

```
int sumSquare(int x, int y){
   return mult(x,x)+ y;
}
```

```
                                    # x e y estão em $a0 e $a1
sumSquare:
  "push"  addi $sp,$sp,-8           # espaço na stack 2 words
          sw   $ra, 4($sp)          # guardar ret addr
          sw   $a1, 0($sp)          # guardar y

          add  $a1,$a0,$zero        # mult(x,x)
          jal  mult                 # chamar mult

          lw   $a1, 0($sp)          # restaurar y

          add  $v0,$v0,$a1          # mult()+y

  "pop"   lw   $ra, 4($sp)          # obter ret addr
          addi $sp,$sp,8            # libertar a stack

          jr   $ra

mult: ...
```



Passos na chamada de uma função

1) Salvaguardar a informação necessária na pilha (e.g. endereço de retorno em $ra)

2) Fazer a passagem de parâmetro(s), se houver

3) Saltar para a função chamada usando `jal`

4) Restabelecer valores a partir da pilha

Regras a respeitar pela função chamada

- A função é chamada através da instrução `jal`, e regressa usando `jr $ra`
- Aceita um máximo de 4 parâmetros passados através dos registos `$a0`, `$a1`, `$a2` e `$a3`
- O retorno de valores é sempre feito através de `$v0` (e se necessário de `$v1`)
- Tem de obedecer às convenções de registos

Registos Gerais do MIPS

Constante 0	$0	$zero
Reservado para o Assembler	$1	$at
Retorno de Valores	$2-$3	$v0-$v1
Parâmetros	$4-$7	$a0-$a3
Variáveis Temporárias	$8-$15	$t0-$t7
Variáveis (saved)	$16-$23	$s0-$s7
Mais variáveis temporárias	$24-$25	$t8-$t9
Reservado para o Kernel	$26-27	$k0-$k1
Ponteiro Global	$28	$gp
Ponteiro da Pilha	$29	$sp
Ponteiro de “Frame”	$30	$fp
Endereço de Retorno	$31	$ra

Existem ainda: Registos reservados (e.g. PC), e registos de vírgula flutuante

Registos desconhecidos

- `$at`: pode ser utilizado pelo assembler em qualquer altura; não é seguro utilizar

- `$k0-$k1`: podem ser usados pelo OS em qualquer altura; não é seguro utilizar.

- `$gp`, `$fp`: vamos ignorar estes registos. Podem ler sobre eles no apêndice A do livro, mas vamos passar sem eles na escrita do nosso código.

Estrutura básica de uma função

Prólogo

```
entry_label:
        addi $sp,$sp, -framesize
        sw   $ra, framesize-4($sp)   # guarda $ra
```

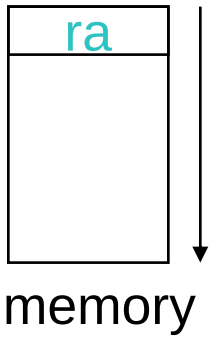
(salvaguarda outros registos se necessário)

Corpo ... **(chama outras funções…)**

Epilógo

(recupera outros registos)

```
        lw   $ra, framesize-4($sp)   # recupera $ra
        addi $sp,$sp, framesize
        jr   $ra
```



Convenção dos Registos (1/4)

- Chamante ou CalleR: a função que chama
- Chamada ou CalleE: a função chamada
- Quando a função chamada regressa, a função chamante precisa de saber que registos foram alterados e que registos mantiveram o valor.
- Convenção de registos: conjunto de regras ou convenções, a ser respeitadas pelo programador/compilador, que define quais os registos que podem ser alterados depois da chamada a `jal`, e quais têm de ser preservados no regresso.

Convenção dos Registos (2/4) - SAVED

- `$0`: Não Altera. Sempre 0.
- `$s0-$s7`: Repôr se modificado. É por isso que são chamados "saved registers". Se a função chamada alterar estes registos deverá restaurá-los antes de regressar à função chamante.
- `$sp`: Repôr se modificado. O *stack pointer* deverá apontar para o mesmo endereço de memória antes e depois da instrução `jal` que passa a execução para a função chamada.
- DICA -- Todos os registos "saved" começam por S!

Convenção dos Registos (3/4) - VOLÁTEIS

- `$ra`: Pode ser alterado. A própria instrução `jal` modifica este registo. A função Chamante tem a obrigação de o salvaguardar na pilha antes de passar a execução a outra função.

- `$v0-$v1`: Podem ser alterados. Este registos contêm os valores de retorno

- `$a0-$a3`: Podem ser alterados. Servem para passar parâmetros à função chamada. A função chamante tem de salvaguardá-los se precisar de manter estes valores depois da função chamada regressar.

- `$t0-$t9`: Podem ser alterados. Por alguma razão são chamados temporários ...

Convenção de Registos (4/4)

- Se **R** é a função chamante, e **E** é a função chamada, temos em resumo que ...
  - A função **R**, antes de fazer o `jal` para E, tem de guardar na pilha todos os registos temporários que tencione usar mais tarde (isto para além de `$ra`)
  - A função **E** tem de guardar na pilha todos os registos S (saved) que pretende utilizar, de forma a poder repôr os seus valores antes de regressar com `jr`
  - Atenção: Caller/Callee só precisam de guardar os registos temporários/saved que precisem/utilizem, e não todos os registos.

Outro Exemplo: Séries de Fibonacci (1/4)

- Os números de Fibonacci definem-se da seguinte forma:
  F(n) = F(n – 1) + F(n – 2),
  F(0) e F(1) são sempre 1

- Assim a série de Fibonacci para n=9 é:

F(0)=1;	F(3)=3;	F(6)=13;	F(9)=55;
F(1)=1;	F(4)=5;	F(7)=21;	
F(2)=2;	F(5)=8;	F(8)=34;	

- E o código recursivo em C é

```
int fib(int n) {
 if(n == 0) { return 1; }
 if(n == 1) { return 1; }
 return (fib(n - 1) + fib(n - 2));
}
```

Outro Exemplo: Séries de Fibonacci (2/4)

```
int fib(int n) {
 if(n == 0) { return 1; }
 if(n == 1) { return 1; }
 return (fib(n - 1) + fib(n - 2));
}
```

- Vamos compilar "à mão"!
- Argumento de entrada => `$a0`
- Passagem de resultado => `$v0`
- Precisamos de guardar 3 *words* na pilha:
  - $ra (a função chama outras funções)
  - Um registo para acumular o resultado (e.g. `$s0`)
  - Guardar o valor de "n" para passar correctamente o parâmetro na chamada seguinte
- Durante a resolução use o seu espírito crítico para ver se conseguiria resolver o problema guardando menos de 3 *words* na pilha

Outro Exemplo: Séries de Fibonacci (3/4)

```
int fib(int n) {
 if(n == 0) { return 1; }
 if(n == 1) { return 1; }
 return (fib(n - 1) + fib(n - 2));
}
```

Prólogo

```
fib:
   addi $sp, $sp, -12     # Espaço para 3 words
   sw $ra, 8($sp)         # Guardar endereço de retorno
   sw $s0, 4($sp)         # Salvaguardar $s0
```

Epílogo

```
fim:
   lw $s0, 4($sp)         #Repôr $s0
   lw $ra, 8($sp)         #Repôr o endereço de retorno
   addi $sp, $sp, 12      #Colocar a pilha como foi recebida
   jr $ra                 #Regressar à função chamante
```

Outro Exemplo: Séries de Fibonacci (4/4)

```
int fib(int n) {
 if(n == 0) { return 1; }
 if(n == 1) { return 1; }
 return (fib(n - 1) + fib(n - 2));
}
```

Corpo

```
# Retornar 1 quando $a0 é 0 ou 1
addiu    $v0, $zero, 1
beq      $a0, $zero, fim    #Preparar para sair ($a0=0)
addiu    $t0, $zero, 1      #Será que poderíamos não usar $t0?
beq      $a0, $t0, fim      #Preparar para sair ($a0=1)

addiu    $a0, $a0, -1       #Preparar argumento 1ª chamada
sw       $a0, 0($sp)        #Salvaguardar para a 2ª chamada
jal      fib                #fib(n-1)
addi     $s0, $v0, $zero    #Salvaguardar o result. preliminar
lw       $a0, 0($sp)        #Preparar argumento 2ª chamada
addiu    $a0, $a0, -1
jal      fib                #fib(n-2)

addi     $v0, $v0, $s0      #resultado final
```

Exemplo B - Faça a Compilação (1/3)

```
main() {
   int i,j,k,m;                    /* i-m:$s0-$s3 */
   ...
   i = mult(j,k); ...
   m = mult(i,i); ...
}

int mult (int mcand, int mlier){
   int product;
   product = 0;
   while (mlier > 0)  {
    product += mcand;
    mlier -= 1; }
   return product;
}
```

Exemplo B - Faça a Compilação (2/3)

```
main:
   ...
   add $a0,$s1,$0      # arg0 = j
   add $a1,$s2,$0      # arg1 = k
   jal mult            # call mult
   add $s0,$v0,$0      # i = mult()
   ...

   add $a0,$s0,$0      # arg0 = i
   add $a1,$s0,$0      # arg1 = i
   jal mult            # call mult
   add $s3,$v0,$0      # m = mult()
   ...
```

```
main() {
int i,j,k,m; /* i-m:$s0-$s3 */
...
i = mult(j,k); ...
m = mult(i,i); ... }
```

- Nota: todas as variáveis a ser preservadas na função `main` estão em registos "*saved*" e portanto não precisam de ser salvaguardadas na pilha.

Exemplo B - Faça a Compilação (3/3)

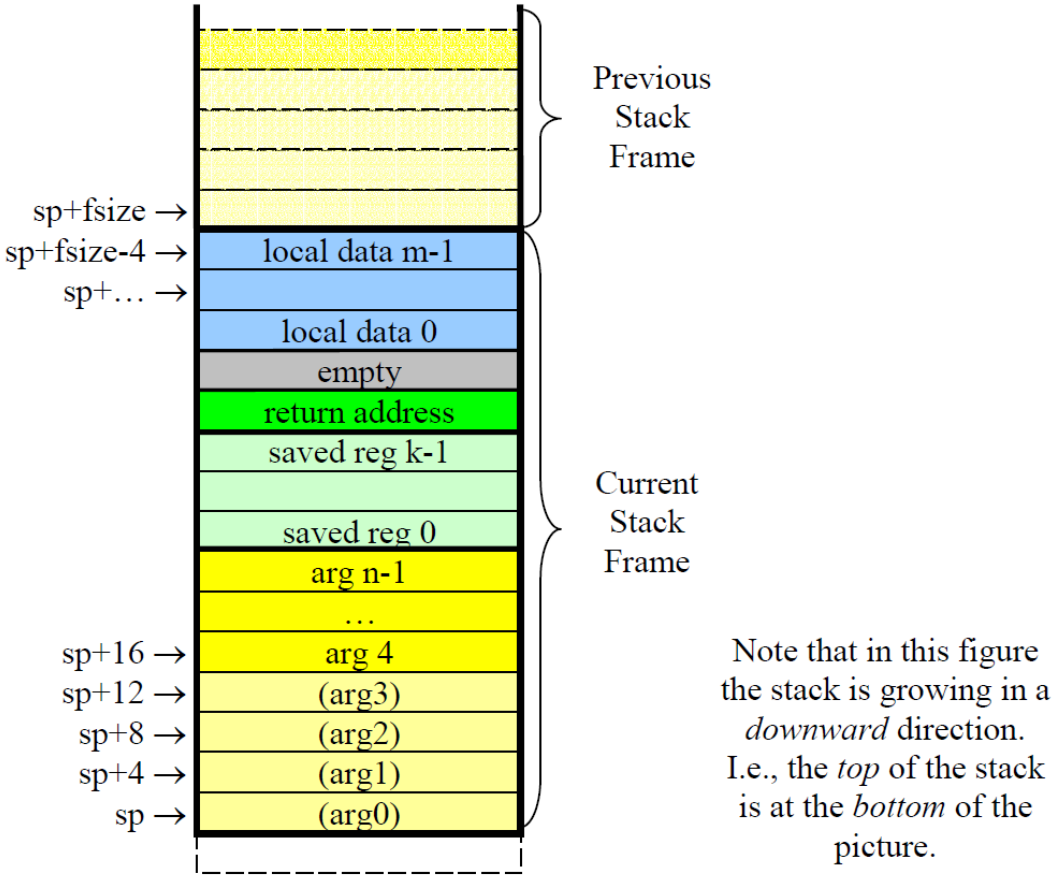
```
mult:
      add $t0,$0,$0       # prod=0

Loop:
      slt  $t1,$0,$a1     # mlier > 0?
      beq  $t1,$0,Fim     # no=>Fim
      add  $t0,$t0,$a0    # prod+=mcand
      addi $a1,$a1,-1     # mlier-=1
      j    Loop           # goto Loop

Fim:
      add  $v0,$t0,$0     # $v0=prod
      jr   $ra            # return
```

```
int mult (int mcand, int mlier){
int product = 0;
while (mlier > 0)  {
 product += mcand;
 mlier -= 1; }
return product;
}
```

- Notas:
  - Não há chamadas a jal feitas dentro do mult, assim não é preciso fazer a salvaguarda de $ra
  - Também não são usados saved registers o que significa que não há contexto a ser guardado na pilha

Mapeamento do *Stack Frame* em Funções

Os compiladores modernos baseados no MIPS fazem o seguinte mapeamento do *stack frame* quando uma função é chamada:



Para saber mais ...

- P&H - Capítulos 2.6 e 2.7
- P&H - Capítulo 2.9 páginas 95 e 96
- Anexo A-6 no CD que vem com o livro