Imediatamente acima dos dados estáticos estão os *dados dinâmicos*. Esses dados, como seu nome sugere, são alocados pelo programa enquanto ele é executado. Nos programas C, a rotina de biblioteca `malloc` localiza e retorna um novo bloco de memória. Como um compilador não pode prever quanta memória um programa alocará, o sistema operacional expande a área de dados dinâmica para atender à demanda. Conforme indica a seta para cima na figura, `malloc` expande a área dinâmica com a chamada do sistema `sbrk`, que faz com que o sistema operacional acrescente mais páginas ao espaço de endereçamento virtual do programa (veja Seção 5.4, no Capítulo 5) imediatamente acima do segmento de dados dinâmico.

A terceira parte, o **segmento de pilha** do programa, reside no topo do espaço de endereçamento virtual (começando no endereço $7fffffff_{hexa}$). Assim como os dados dinâmicos, o tamanho máximo da pilha de um programa não é conhecido antecipadamente. À medida que o programa coloca valores na pilha, o sistema operacional expande o segmento de pilha para baixo, em direção ao segmento de dados.

segmento de pilha A parte da memória usada por um programa para manter frames de chamada de procedimento.

Essa divisão de três partes da memória não é a única possível. Contudo, ela possui duas características importantes: os dois segmentos dinamicamente expansíveis são bastante distantes um do outro, e eles podem crescer para usar o espaço de endereços inteiro de um programa.

B.6 Convenção para chamadas de procedimento

As convenções que controlam o uso dos registradores são necessárias quando os procedimentos em um programa são compilados separadamente. Para compilar um procedimento em particular, um compilador precisa saber quais registradores pode usar e quais são reservados para outros procedimentos. As regras para usar os registradores são chamadas de **convenções para uso dos registradores** ou **convenções para chamadas de procedimento**. Como o nome sugere, essas regras são, em sua maior parte, convenções seguidas pelo software, em vez de regras impostas pelo hardware. No entanto, a maioria dos compiladores e programadores tenta seguir essas convenções estritamente, pois sua violação causa bugs traiçoeiros.

convenção para uso dos registradores Também chamada **convenção para chamadas de procedimento**. Um protocolo de software que controla o uso dos registradores por procedimentos.

A convenção para chamadas descrita nesta seção é aquela utilizada pelo compilador gcc. O compilador nativo do MIPS utiliza uma convenção mais complexa, que é ligeiramente mais rápida.

A CPU do MIPS contém 32 registradores de uso geral, numerados de 0 a 31. O registrador `$0` contém o valor fixo 0.

- Os registradores `$at` (1), `$k0` (26) e `$k1` (27) são reservados para o montador e o sistema operacional e não devem ser usados por programas do usuário ou compiladores.
- Os registradores `$a0`—`$a3` (4-7) são usados para passar os quatro primeiros argumentos às rotinas (os argumentos restantes são passados na pilha). Os registradores `$v0` e `$v1` (2, 3) são usados para retornar valores das funções.
- Os registradores `$t0`—`$t9` (8-15, 24, 25) são **registradores salvos pelo caller**, usados para manter quantidades temporárias que não precisam ser preservadas entre as chamadas (veja Seção 2.8, no Capítulo 2).

registrador salvo pelo caller Um registrador salvo pela rotina que faz uma chamada de procedimento.

- Os registradores `$s0`—`$s7` (16-23) são **registradores salvos pelo callee**, que mantêm valores de longa duração os quais devem ser preservados entre as chamadas.

registrador salvo pelo callee Um registrador salvo pela rotina sendo chamada.

- O registrador `$gp` (28) é um ponteiro global que aponta para o meio de um bloco de 64K de memória no segmento de dados estático.
- O registrador `$sp` (29) é o stack pointer, que aponta para o último local na pilha. O registrador `$fp` (30) é o frame pointer. A instrução `jal` escreve no registrador `$ra` (31) o endereço de retorno de uma chamada de procedimento. Esses dois registradores são explicados na próxima seção.

As abreviações e os nomes de duas letras para esses registradores – por exemplo, `$sp` para o stack pointer – refletem os usos intencionados na convenção de chamada de procedimento. Ao descrever essa convenção, usaremos os nomes em vez de números de registrador. A Figura B.6.1 lista os registradores e descreve seus usos intencionados.

Nome do registrador	Número	Uso
$zero	0	constante 0
$at	1	reservado para o montador
$v0	2	avaliação de expressão e resultados de uma função
$v1	3	avaliação de expressão e resultados de uma função
$a0	4	argumento 1
$a1	5	argumento 2
$a2	6	argumento 3
$a3	7	argumento 4
$t0	8	temporário (não preservado pela chamada)
$t1	9	temporário (não preservado pela chamada)
$t2	10	temporário (não preservado pela chamada)
$t3	11	temporário (não preservado pela chamada)
$t4	12	temporário (não preservado pela chamada)
$t5	13	temporário (não preservado pela chamada)
$t6	14	temporário (não preservado pela chamada)
$t7	15	temporário (não preservado pela chamada)
$s0	16	temporário salvo (preservado pela chamada)
$s1	17	temporário salvo (preservado pela chamada)
$s2	18	temporário salvo (preservado pela chamada)
$s3	19	temporário salvo (preservado pela chamada)
$s4	20	temporário salvo (preservado pela chamada)
$s5	21	temporário salvo (preservado pela chamada)
$s6	22	temporário salvo (preservado pela chamada)
$s7	23	temporário salvo (preservado pela chamada)
$t8	24	temporário (não preservado pela chamada)
$t9	25	temporário (não preservado pela chamada)
$k0	26	reservado para o kernel do sistema operacional
$k1	27	reservado para o kernel do sistema operacional
$gp	28	ponteiro para área global
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	endereço de retorno (usado por chamada de função)

FIGURA B.6.1 Registradores do MIPS e convenção de uso.

Chamadas de procedimento

Esta seção descreve as etapas que ocorrem quando um procedimento (*caller*) invoca outro procedimento (*callee*). Os programadores que escrevem em uma linguagem de alto nível (como C ou Pascal) nunca veem os detalhes de como um procedimento chama outro, pois o compilador cuida dessa manutenção de baixo nível. Contudo, os programadores assembly precisam implementar explicitamente cada chamada e retorno de procedimento.

frame de chamada de procedimento Um bloco de memória usado para manter valores passados a um procedimento como argumentos, a fim de salvar registradores que um procedimento pode modificar mas que o caller não deseja que sejam alterados, e fornecer espaço para variáveis locais a um procedimento.

A maior parte da manutenção associada a uma chamada gira em torno de um bloco de memória chamado **frame de chamada de procedimento**. Essa memória é usada para diversas finalidades:

- Para manter valores passados a um procedimento como argumentos.
- Para salvar registradores que um procedimento pode modificar, mas que o caller não deseja que sejam alterados.
- Para oferecer espaço para variáveis locais a um procedimento.

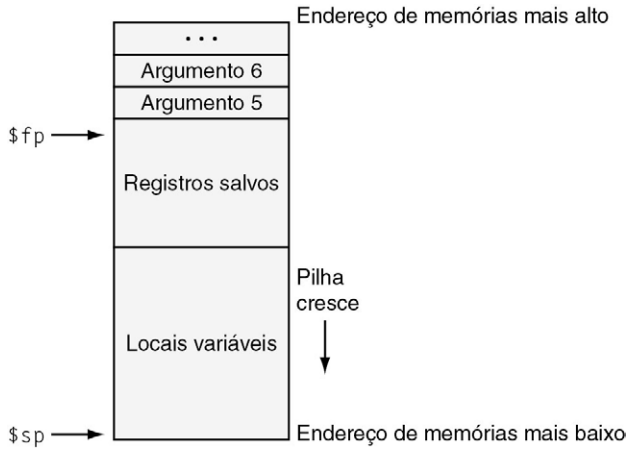

FIGURA B.6.2 Layout de um frame de pilha. O frame pointer (`$fp`) aponta para a primeira palavra do frame de pilha do procedimento em execução. O stack pointer (`$sp`) aponta para a última palavra do frame. Os quatro primeiros argumentos são passados em registradores, de modo que o quinto argumento é o primeiro armazenado na pilha.

Na maioria das linguagens de programação, as chamadas e retornos de procedimento seguem uma ordem estrita do tipo último a entrar, primeiro a sair (LIFO – Last-In, First-Out), de modo que essa memória pode ser alocada e liberada como uma pilha, motivo pelo qual esses blocos de memória às vezes são chamados frames de pilha.

A Figura B.6.2 mostra um frame de pilha típico. O frame consiste na memória entre o frame pointer (`$fp`), que aponta para a primeira palavra do frame, e o stack pointer (`$sp`), que aponta para a última palavra do frame. A pilha cresce para baixo a partir dos endereços de memória mais altos, de modo que o frame pointer aponta para cima do stack pointer. O procedimento que está executando utiliza o frame pointer para acessar rapidamente os valores em seu frame de pilha. Por exemplo, um argumento no frame de pilha pode ser lido para o registrador `$v0` com a instrução

```
lw $v0, 0($fp)
```

Um frame de pilha pode ser construído de muitas maneiras diferentes; porém, o caller e o callee precisam combinar a sequência de etapas. As etapas a seguir descrevem a convenção de chamada utilizada na maioria das máquinas MIPS. Essa convenção entra em ação em três pontos durante uma chamada de procedimento: imediatamente antes de o caller invocar o callee, assim que o callee começa a executar e imediatamente antes de o callee retornar ao caller. Na primeira parte, o caller coloca os argumentos da chamada de procedimento em locais padrões e invoca o callee para fazer o seguinte:

1. Passar argumentos. Por convenção, os quatro primeiros argumentos são passados nos registradores `$a0`—`$a3`. Quaisquer argumentos restantes são colocados na pilha e aparecem no início do frame de pilha do procedimento chamado.
2. Salvar registradores salvos pelo caller. O procedimento chamado pode usar esses registradores (`$a0`—`$a3` e `$t0`—`$t9`) sem primeiro salvar seu valor. Se o caller espera utilizar um desses registradores após uma chamada, ele deverá salvar seu valor antes da chamada.
3. Executar uma instrução `jal` (veja Seção 2.8 do Capítulo 2), que desvia para a primeira instrução do callee e salva o endereço de retorno no registrador `$ra`.

Antes que uma rotina chamada comece a executar, ela precisa realizar as seguintes etapas para configurar seu frame de pilha:

1. Alocar memória para o frame, subtraindo o tamanho do frame do stack pointer.
2. Salvar os registradores salvos pelo callee no frame. Um callee precisa salvar os valores desses registradores (`$s0`—`$s7`, `$fp` e `$ra`) antes de alterá-los, pois o caller espera

encontrar esses registradores inalterados após a chamada. O registrador `$fp` é salvo para cada procedimento que aloca um novo frame de pilha. No entanto, o registrador `$ra` só precisa ser salvo se o callee fizer uma chamada. Os outros registradores salvos pelo callee, que são utilizados, também precisam ser salvos.

3. Estabelecer o frame pointer somando o tamanho do frame de pilha menos 4 a `$sp` e armazenando a soma no registrador `$fp`.

Interface hardware/ software

A convenção de uso dos registradores do MIPS oferece registradores salvos pelo caller e pelo callee, pois os dois tipos de registradores são vantajosos em circunstâncias diferentes. Os registradores salvos pelo caller são usados para manter valores de longa duração, como variáveis de um programa do usuário. Esses registradores só são salvos durante uma chamada de procedimento se o caller espera utilizar o registrador. Por outro lado, os registradores salvos pelo callee são usados para manter quantidades de curta duração, que não persistem entre as chamadas, como valores imediatos em um cálculo de endereço. Durante uma chamada, o caller não pode usar esses registradores para valores temporários de curta duração.

Finalmente, o callee retorna ao caller executando as seguintes etapas:

1. Se o callee for uma função que retorna um valor, coloque o valor retornado no registrador `$v0`.
2. Restaure todos os registradores salvos pelo callee que foram salvos na entrada do procedimento.
3. Remova o frame de pilha somando o tamanho do frame a `$sp`.
4. Retorne desviando para o endereço no registrador `$ra`.

procedimentos recursivos Procedimentos que chamam a si mesmos direta ou indiretamente através de uma cadeia de chamadas.

Detalhamento: Uma linguagem de programação que não permite **procedimentos recursivos** – procedimentos que chamam a si mesmos direta ou indiretamente, por meio de uma cadeia de chamadas – não precisa alocar frames em uma pilha. Em uma linguagem não recursiva, o frame de cada procedimento pode ser alocado estaticamente, pois somente uma invocação de um procedimento pode estar ativa ao mesmo tempo. As versões mais antigas de Fortran proibiam a recursão porque frames alocados estaticamente produziam código mais rápido em algumas máquinas mais antigas. Todavia, em arquiteturas load-store, como MIPS, os frames de pilha podem ser muito rápidos porque o registrador frame pointer aponta diretamente para o frame de pilha ativo, que permite que uma única instrução load ou store acesse valores no frame. Além disso, a recursão é uma técnica de programação valiosa.

Exemplo de chamada de procedimento

Como exemplo, considere a rotina em C

```
main ()
{
  printf ("The factorial of 10 is %d\n",fact(10));
}
int fact (int n)
{
  if (n < 1)
      return (1);
  else
      return (n * fact (n-1))
}
```

que calcula e imprime 10! (o fatorial de 10, 10! = 10 × 9 × ... × 1). `fact` é uma rotina recursiva que calcula *n*! multiplicando *n* vezes (*n* – 1)!. O código assembly para essa rotina ilustra como os programas manipulam frames de pilha.

Na entrada, a rotina `main` cria seu frame de pilha e salva os dois registradores salvos pelo callee que serão modificados: `$fp` e `$ra`. O frame é maior do que o exigido para esses dois registradores, pois a convenção de chamada exige que o tamanho mínimo de um frame de pilha seja 24 bytes. Esse frame mínimo pode manter quatro registradores de argumento (`$a0`—`$a3`) e o endereço de retorno `$ra`, preenchidos até um limite de double palavra (24 bytes). Como `main` também precisa salvar o `$fp`, seu frame de pilha precisa ter duas palavras a mais (lembre-se de que o stack pointer é mantido alinhado em um limite de double palavra).

```
        .text
        .globl main
    main:
        subu   $sp,$sp,32   # Frame de pilha tem 32
        sw     $ra,20($sp)  # Salva endereço de
        sw     $fp,16($sp)  # Salva frame pointer
        addiu  $fp,$sp,28   # Prepara frame pointer
```

A rotina `main`, então, chama a rotina de fatorial e lhe passa o único argumento 10. Depois que `fact` retorna, `main` chama a rotina de biblioteca `printf` e lhe passa uma string de formato e o resultado retornado de `fact`:

```
    li      $a0,10      # Coloca argumento (10) em $a0
    jal     fact        # Chama função fatorial

    la      $a0,$LC     # Coloca string de formato em $a0
    move    $a1,$v0     # Move resultado de fact para $a1
    jal     printf      # Chama a função print
```

Finalmente, depois de imprimir o fatorial, `main` retorna. Entretanto, primeiro, ela precisa restaurar os registradores que salvou e remover seu frame de pilha:

```
        lw      $ra,20($sp) # Restaura endereço de retorno
        lw      $fp,16($sp) # Restaura frame pointer
        addiu   $sp,$sp,32  # Remove frame de pilha
        jr      $ra         # Retorna a quem chamou
        .rdata
    $LC:
        .ascii  "The factorial of 10 is %d\n\000"
```

A rotina de fatorial é semelhante em estrutura a `main`. Primeiro, ela cria um frame de pilha e salva os registradores salvos pelo callee que serão usados por ela. Além de salvar `$ra` e `$fp`, `fact` também salva seu argumento (`$a0`), que ela usará para a chamada recursiva:

```
        .text
fact:
        subu    $sp,$sp,32   # Frame de pilha tem 32 bytes
        sw      $ra,20($sp)  # Salva endereço de retorno
        sw      $fp,16($sp)  # Salva frame pointer
        addiu   $fp,$sp,28   # Prepara frame pointer
        sw      $a0,0($fp)   # Salva argumento (n)
```

O núcleo da rotina `fact` realiza o cálculo do programa em C. Ele testa se o argumento é maior do que 0. Caso contrário, a rotina retorna o valor 1. Se o argumento for maior do que 0, a rotina é chamada recursivamente para calcular `fact(n-1)` e multiplica esse valor por *n*:

```
        lw      $v0,0($fp)   # Carrega n
        bgtz    $v0,$L2      # Desvia se n > 0
        li      $v0,1        # Retorna 1
        jr      $L1          # Desvia para o código de retorno
$L2:
        lw      $v1,0($fp)   # Carrega n
        subu    $v0,$v1,1    # Calcula n - 1
        move    $a0,$v0      # Move valor para $a0
        jal     fact         # Chama função de fatorial
        lw      $v1,0($fp)   # Carrega n
        mul     $v0,$v0,$v1  # Calcula fact() * n
```

Finalmente, a rotina de fatorial restaura os registradores salvos pelo callee e retorna o valor no registrador `$v0`:

```
$L1:                         # Resultado está em $v0
    lw      $ra, 20($sp)     # Restaura $ra
    lw      $fp, 16($sp)     # Restaura $fp
    addiu   $sp, $sp, 32     # Remove o frame de pilha
    jr      $ra              # Retorna a quem chamou
```

EXEMPLO

Pilha em procedimentos recursivos

A Figura B.6.3 mostra a pilha na chamada `fact(7)`. `main`executa primeiro, de modo que seu frame está mais abaixo na pilha. `main` chama `fact(10)`, cujo frame de pilha vem em seguida na pilha. Cada invocação chama `fact` recursivamente para calcular o próximo fatorial mais inferior. Os frames de pilha fazem um parale-

lo com a ordem LIFO dessas chamadas. Qual é a aparência da pilha quando a chamada a `fact(10)` retorna?

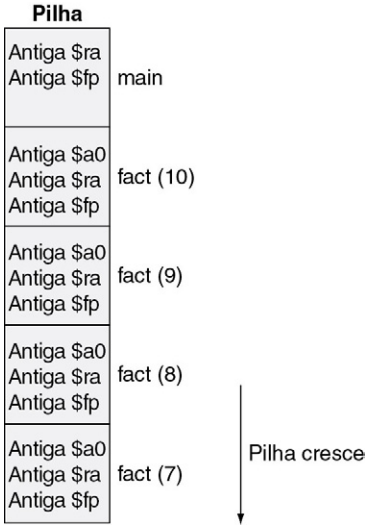

FIGURA B.6.3 Frames da pilha durante a chamada de `fact(7)`.

RESPOSTA

Detalhamento: A diferença entre o compilador MIPS e o compilador gcc é que o compilador MIPS normalmente não usa um frame pointer, de modo que esse registrador está disponível como outro registrador salvo pelo callee, `$s8`. Essa mudança salva algumas das instruções na chamada de procedimento e sequência de retorno. Contudo, isso complica a geração do código, porque um procedimento precisa acessar seu frame de pilha com `$sp`, cujo valor pode mudar durante a execução de um procedimento se os valores forem colocados na pilha.

Outro exemplo de chamada de procedimento

Como outro exemplo, considere a seguinte rotina que calcula a função `tak`, que é um benchmark bastante utilizado, criado por Ikuo Takeuchi. Essa função não calcula nada de útil, mas é um programa altamente recursivo que ilustra a convenção de chamada do MIPS.

```
int tak (int x, int y, int z)
{
    if (y < x)
        return 1 + tak tak (x - 1, y, z) ,
          tak (y - 1, z, x),
          tak (z - 1, x, y));
     else
        return z;
}
int main ( )
{
     tak(18, 12, 6);
}
```

O código assembly para esse programa está logo em seguida. A função `tak` primeiro salva seu endereço de retorno no frame de pilha e seus argumentos nos registradores salvos pelo callee, pois a rotina pode fazer chamadas que precisam usar os registradores `$a0`—`$a2` e `$ra`. A função utiliza registradores salvos pelo callee, pois eles mantêm valores que persistem por toda a vida da função, o que inclui várias chamadas que potencialmente poderiam modificar registradores.

```
    .text
    .globl  tak
tak:
    subu    $sp, $sp, 40
    sw      $ra, 32($sp)
    sw      $s0, 16($sp)  # x
    move    $s0, $a0
    sw      $s1, 20($sp)  # y
    move    $s1, $a1
    sw      $s2, 24($sp)  # z
    move    $s2, $a2
    sw      $s3, 28($sp)  # temporário
```

A rotina, então, inicia a execução testando se `y<x`. Se não, ela desvia para o rótulo`L1`, que aparece a seguir.

```
    bge  $s1, $s0, L1  # if (y < x)
```

Se `y<x`, então ela executa o corpo da rotina, que contém quatro chamadas recursivas. A primeira chamada usa quase os mesmos argumentos da progenitora:

```
addiu   $a0, $s0, -1
move    $a1, $s1
move    $a2, $s2
jal     tak           # tak (x - 1, y, z)
move    $s3, $v0
```

Observe que o resultado da primeira chamada recursiva é salvo no registrador `$s3`, de modo que pode ser usado mais tarde.

A função agora prepara argumentos para a segunda chamada recursiva.

```
addiu   $a0, $s1, -1
move    $a1, $s2
move    $a2, $s0
jal     tak           # tak (y - 1, z, x)
```

Nas instruções a seguir, o resultado dessa chamada recursiva é salvo no registrador `$s0`. No entanto, primeiro, precisamos ler, pela última vez, o valor salvo do primeiro argumento a partir desse registrador.

```
addiu   $a0, $s2, -1
move    $a1, $s0
move    $a2, $s1
move    $s0, $v0
jal     tak           # tak (z - 1, x, y)
```

Depois de três chamadas recursivas mais internas, estamos prontos para a chamada recursiva final. Depois da chamada, o resultado da função está em `$v0`, e o controle desvia para o epílogo da função.

```
move    $a0, $s3
move    $a1, $s0
move    $a2, $v0
jal     tak             # tak (tak(...), tak(...), tak(...))
addiu   $v0, $v0, 1
j       L2
```

Esse código no rótulo `L1` é a consequência da instrução *if-then-else*. Ele apenas move o valor do argumento `z` para o registrador de retorno e entra no epílogo da função.

```
L1:
   move  $v0, $s2
```

O código a seguir é o epílogo da função, que restaura os registradores salvos e retorna o resultado da função a quem chamou.

```
L2:
   lw      $ra, 32($sp)
   lw      $s0, 16($sp)
   lw      $s1, 20($sp)
   lw      $s2, 24($sp)
   lw      $s3, 28($sp)
   addiu   $sp, $sp, 40
   jr      $ra
```

A rotina `main` chama a função `tak` com seus argumentos iniciais, e depois pega o resultado calculado (7) e o imprime usando a chamada ao sistema do SPIM para imprimir inteiros:

```
     .globl main
main:
     subu    $sp, $sp, 24
     sw      $ra, 16($sp

     li      $a0, 18
     li      $a1, 12
     li      $a2, 6
     jal tak                 # tak(18, 12, 6)

     move    $a0, $v0
     li      $v0, 1          # syscall print_int
     syscall

     lw      $ra, 16($sp
     addiu   $sp, $sp, 24
     jr      $ra
```