CSCI 210: Computer Architecture
Lecture 12: Procedures & The Stack

Stephen Checkoway

Slides from Cynthia Taylor

CS History: IBM System 360

Photo by Arnold Reinhold, CC BY-SA 3.0, via Wikimedia Commons

- Family of mainframes developed in 1964
- Introduced:
  - 8-bit byte
  - Byte-addressable memory
  - 32-bit words
- Featured BAL (Branch and Link) and BR (Branch Register) instructions
- IBM's current System z mainframes will still run code written for the 360 series

Complete example

```
foo:
        addi    $sp, $sp, -32  # Allocate space for stack frame
        sw      $ra, 28($sp)   # Stores (spills) $ra, return address
        sw      $s0, 24($sp)   # Stores (spills) s0, callee-saved reg
        …
        li      $s0, 25        # Set s0 to 25
        sw      $t3, 20($sp)   # Stores (spills) t3, caller-saved reg
        add     $a0, $t1, $t3
        jal     myFunction
        lw      $t3, 20($sp)   # Restores (fills) t3
        …
        lw      $s0,  24($sp)  # Restores (fills) s0, must restore
        lw      $ra,  28($sp)  # Restores (fills) $ra, return address
        addi    $sp, $sp, 32   # Restore the stack pointer
        jr      $ra            # Return
```

Complete example

```
foo:
        addi    $sp, $sp, -32
        sw      $ra, 28($sp)
        sw      $s0, 24($sp)
        …
        li      $s0, 25
        sw      $t3, 20($sp)
        add     $a0, $t1, $t3
        jal     myFunction
        lw      $t3, 20($sp)
        …
        lw      $s0,  24($sp)
        lw      $ra,  28($sp)
        addi    $sp, $sp, 32
        jr      $ra
```

Stack frame for foo (32 bytes in size)
Arguments are in $a0, …, $a3 and then on the stack at ($sp+32)+16, ($sp+32)+20, … for argument 5, 6, …

$sp + 28	Saved return address $ra
$sp + 24	Saved register $s0
$sp + 20	Saved register $t3
$sp + 16	Unused space to preserve 8-byte alignment
$sp + 12	Space for argument 4 (for use by myFunction)
$sp + 8	Space for argument 3 (for use by myFunction)
$sp + 4	Space for argument 2 (for use by myFunction)
$sp + 0	Space for argument 1 (for use by myFunction)

Leaf function

- If the function doesn't call any other functions, it's a "leaf"
- If a leaf function doesn't need to use any of the callee-saved registers (e.g., \$s0–\$s7), then it doesn't need to change the stack pointer or spill/fill \$ra
- Example:

```
# myFunction(int a0, int a1, int a2)
myFunction:
        add    $t0, $a0, $a2
        sub    $v0, $t0, $a1
        jr     $ra
```

Leaf Procedure Example

```
int leaf_example(
  int g, int h, int i, int j
) {
    int f = (g + h) - (i + j);
    return f;
}
```

- Arguments g, …, j in $a0, …, $a3
- Result in $v0

```
leaf_example:
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $v0, $t0, $t1
    jr   $ra
```

Non-Leaf Procedures

- Procedures that call other procedures
- Caller needs to allocate a stack frame
- Caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

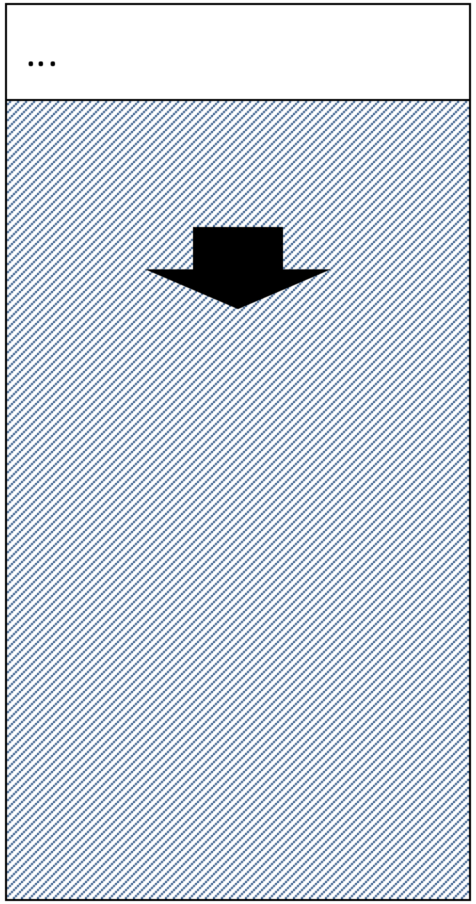
- C code:

```
int fact (int n) {
  if (n < 2)
    return 1;
  else
    return n * fact(n - 1);
}
```

  - Argument n in $a0
  - Result in $v0

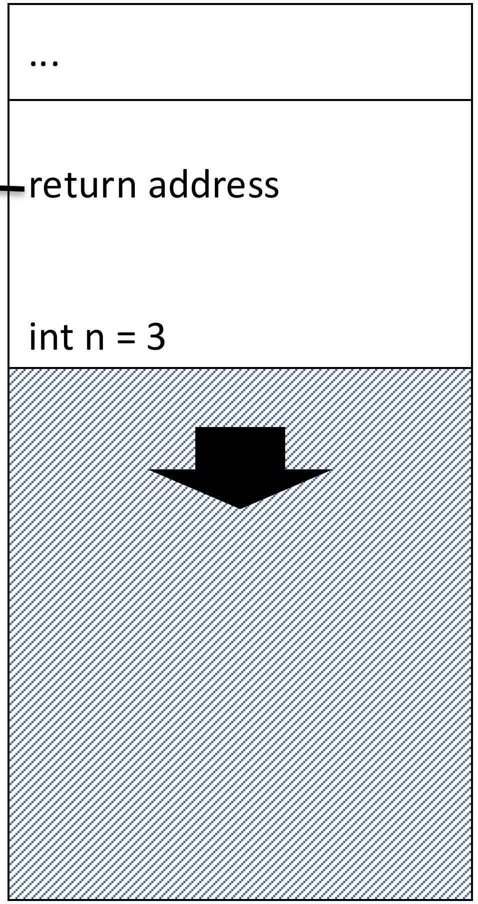
Process Stack

```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
        return 1;
   else
        return n * fact(n - 1);
}
```



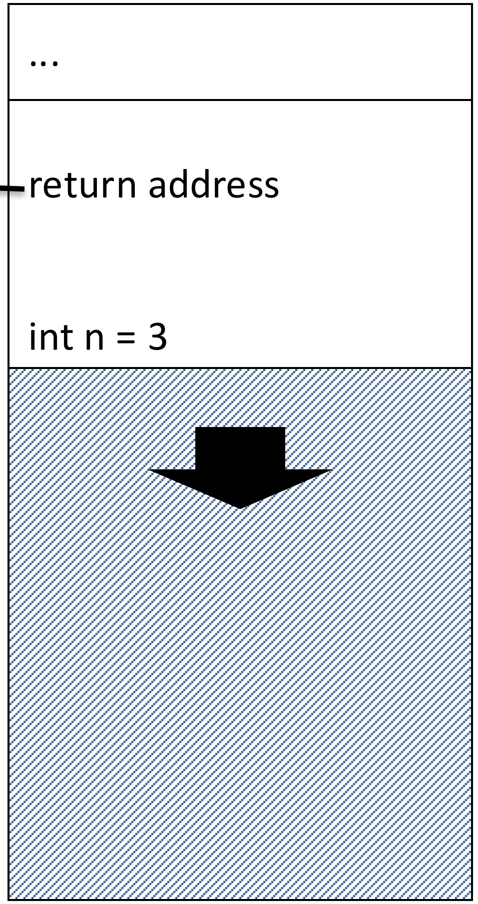
Process Stack

```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
        return 1;
   else
        return n * fact(n - 1);
}
```



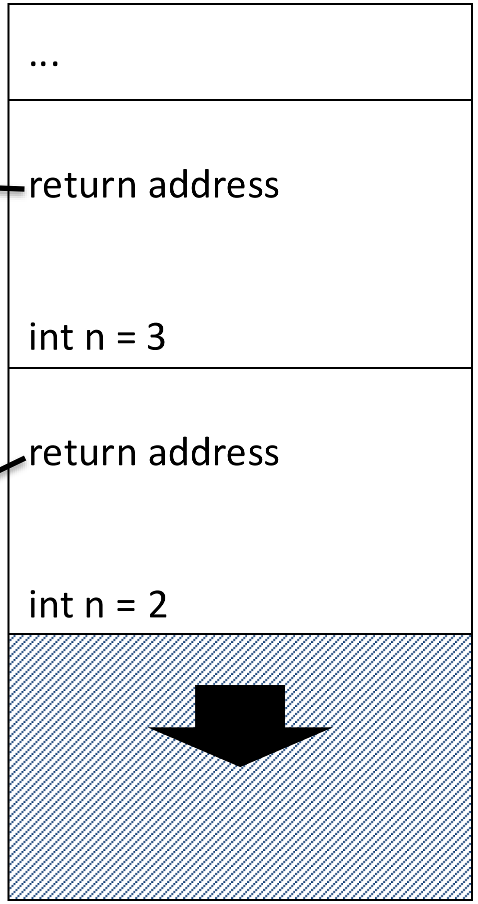
Process Stack

```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
        return 1;
   else
        return n * fact(n - 1);
}
```



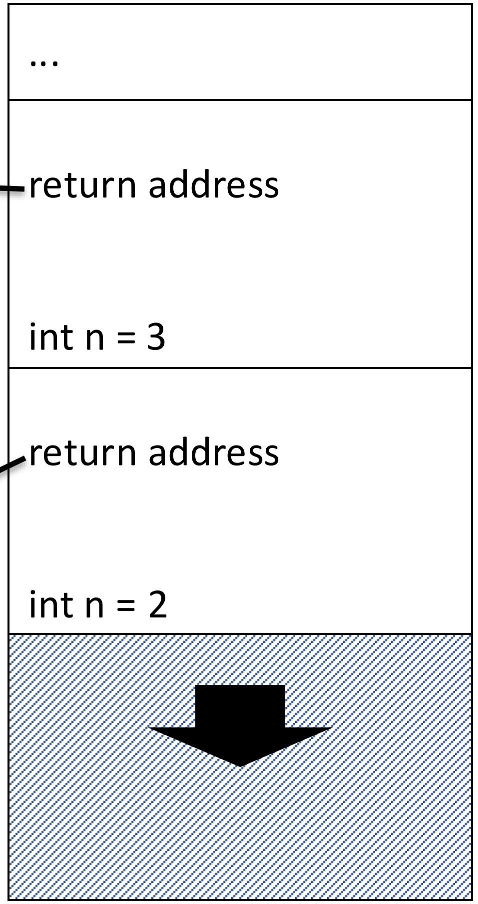
Process Stack

```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
        return 1;
   else
        return n * fact(n - 1);
}
```



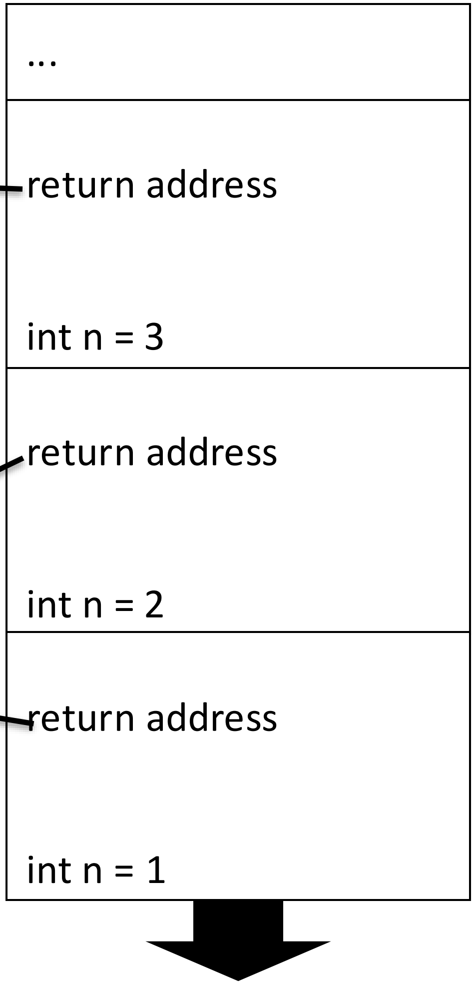
Process Stack

```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
        return 1;
   else
        return n * fact(n - 1);
}
```



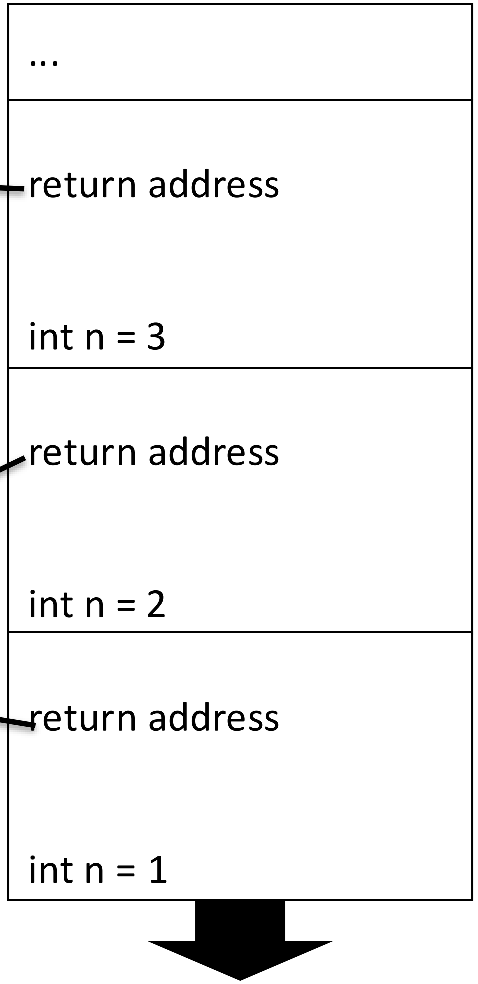
Process Stack

```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
      return 1;
   else
      return n * fact(n - 1);
}
```



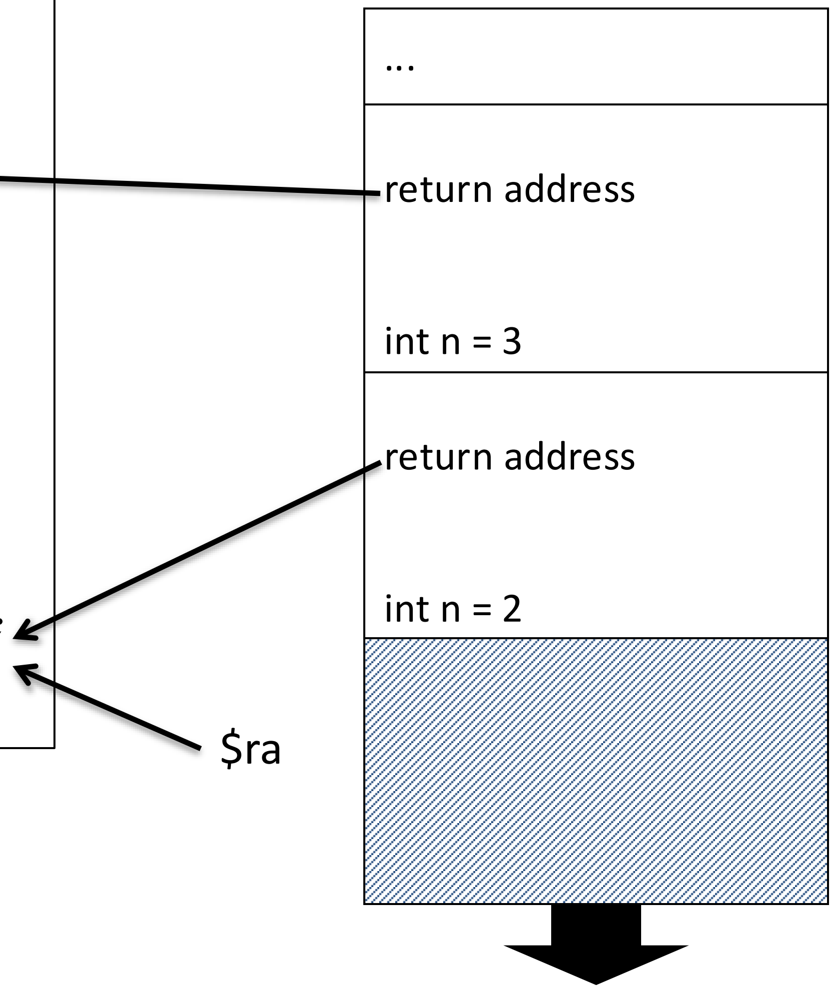
Process Stack

```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
       return 1;
   else
       return n * fact(n - 1);
}
```



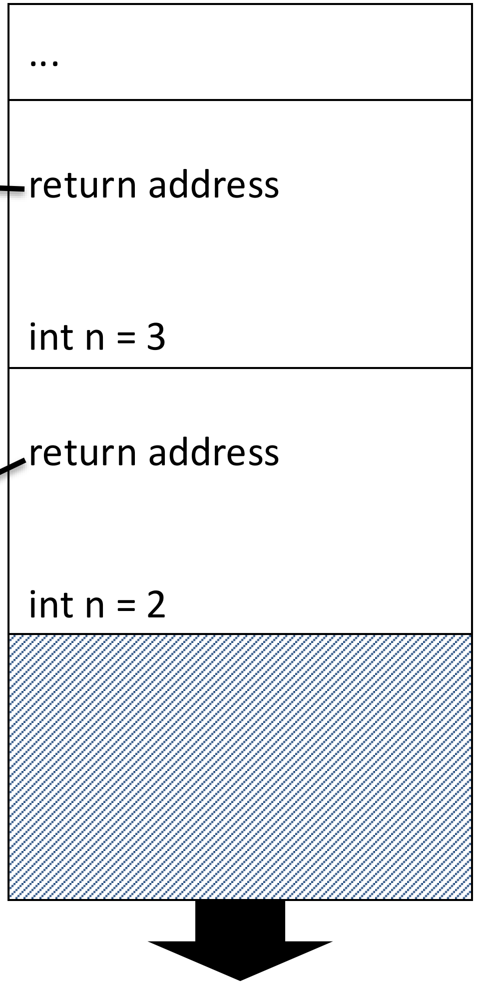
Process Stack

```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
      return 1;
   else
      return n * fact(n - 1);
}
```



Process Stack

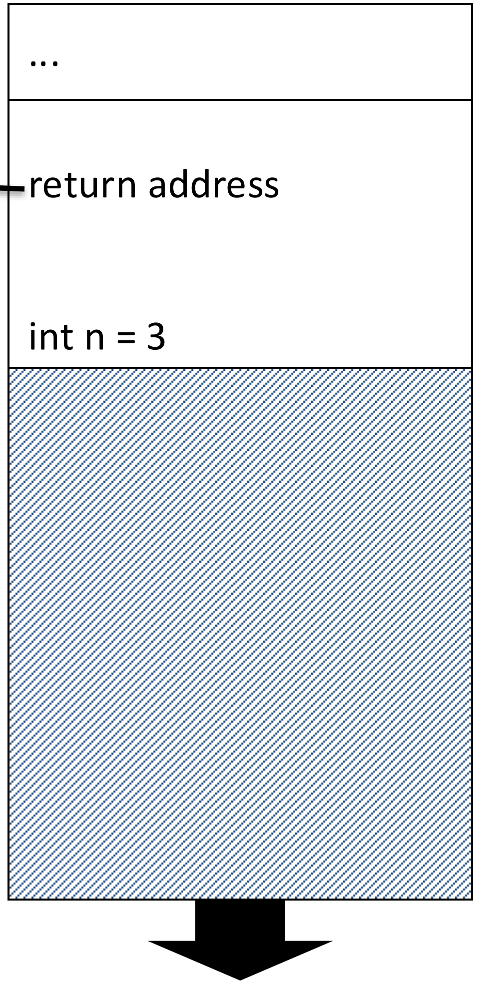
```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
       return 1;
   else
       return n * fact(n - 1);
}
```



Process Stack

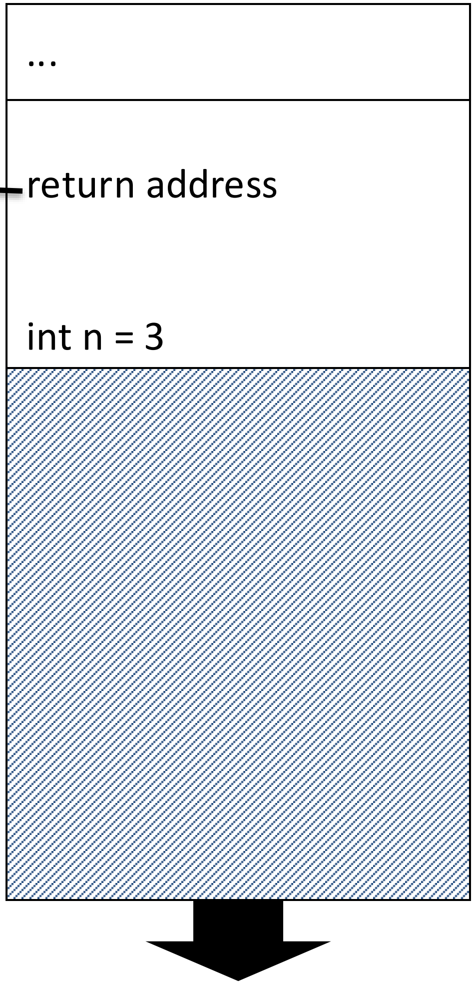
```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
        return 1;
   else
        return n * fact(n - 1);
}
```

$ra



Process Stack

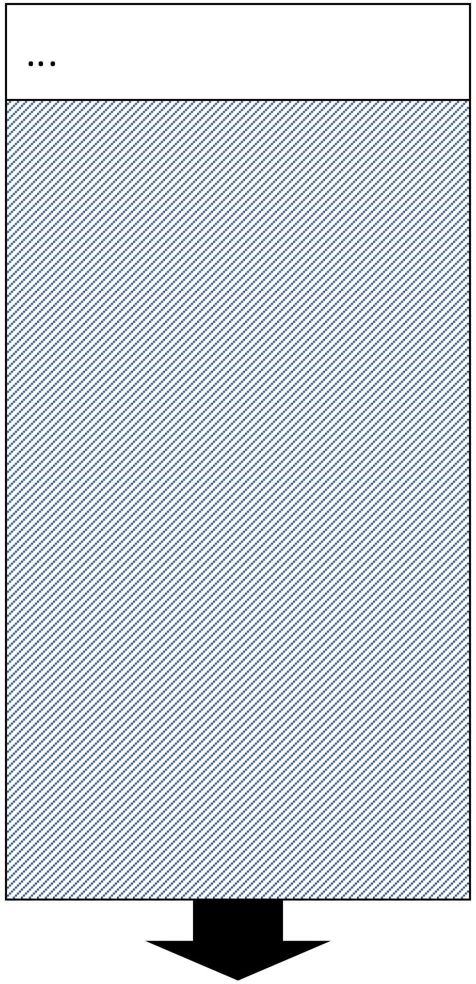
```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
       return 1;
   else
       return n * fact(n - 1);
}
```



Process Stack

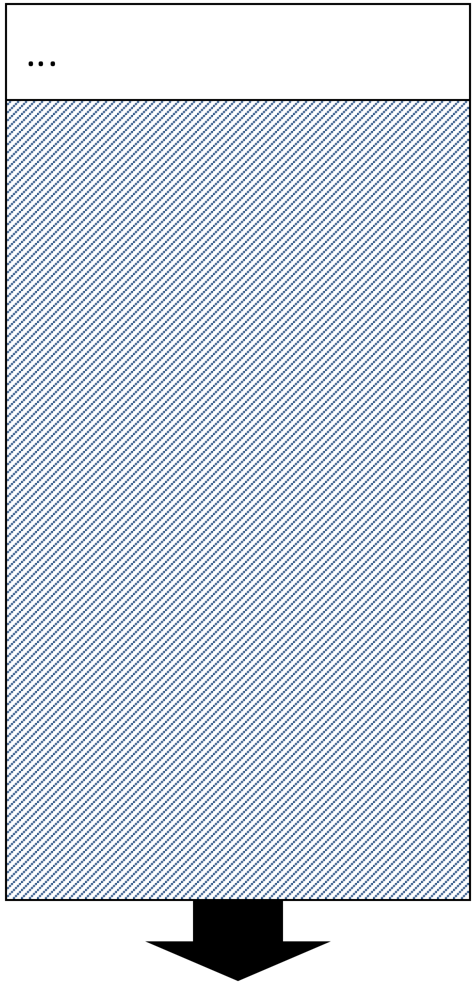
```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
        return 1;
   else
        return n * fact(n - 1);
}
```

$ra



Process Stack

```
int main ()
{
   int x;
   x = fact(3);
}
int fact (int n)
{
   if (n < 2)
        return 1;
   else
        return n * fact(n - 1);
}
```



Questions?

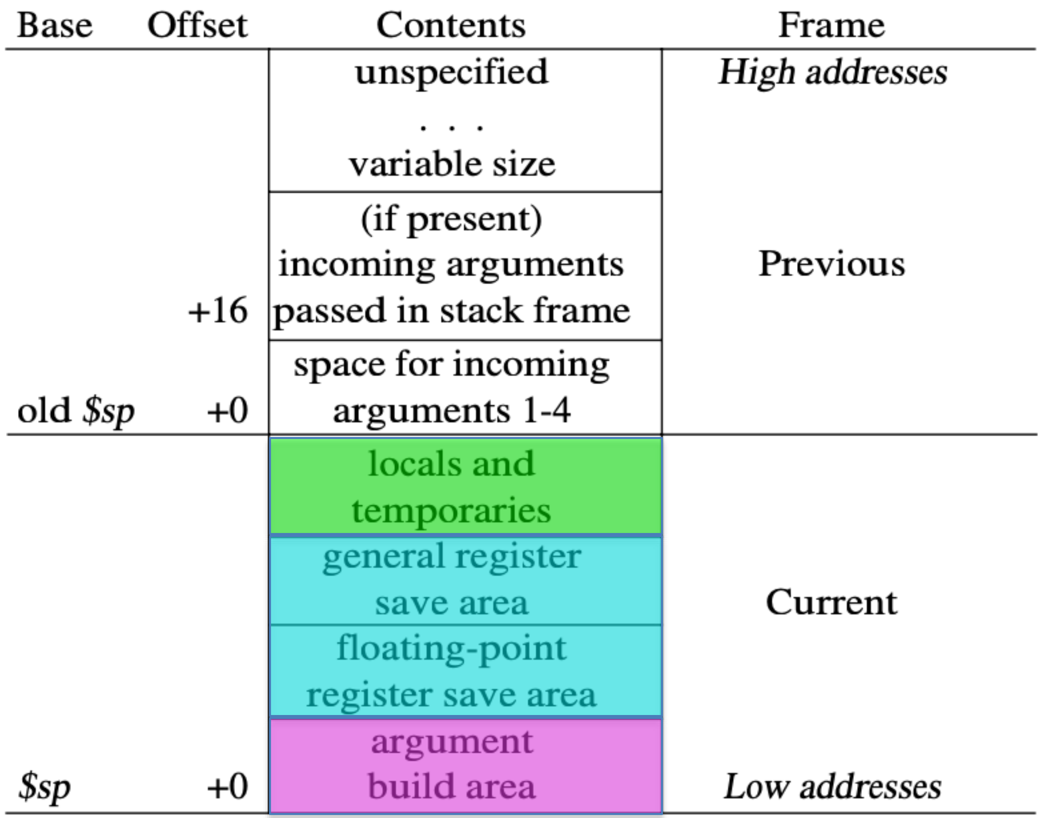
Rules for allocating a stack frame for a nonleaf-procedure

Size of stack frame is sum of

- Local variables and temporaries
- 4 * number of saved registers
- min(16, 4 * number of words of arguments for called functions)

Round the whole thing up to a multiple of 8 for stack alignment

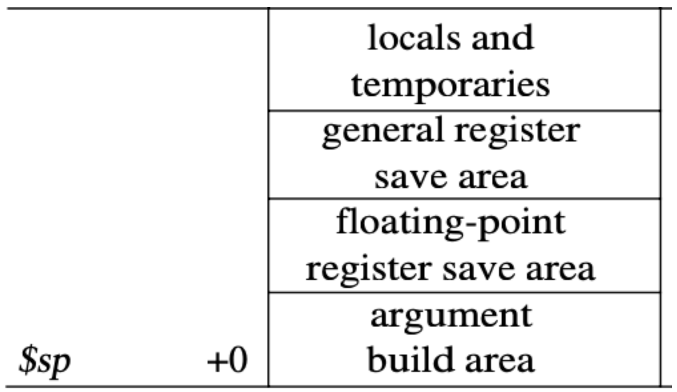
Figure 3-21: Stack Frame



How many local variables does caller need to allocate space for on the stack? (Hint: How many need to persist beyond a function call?)

A. 1 (4 bytes)
B. 2 (8 bytes)
C. 3 (12 bytes)
D. 4 (16 bytes)
E. It depends

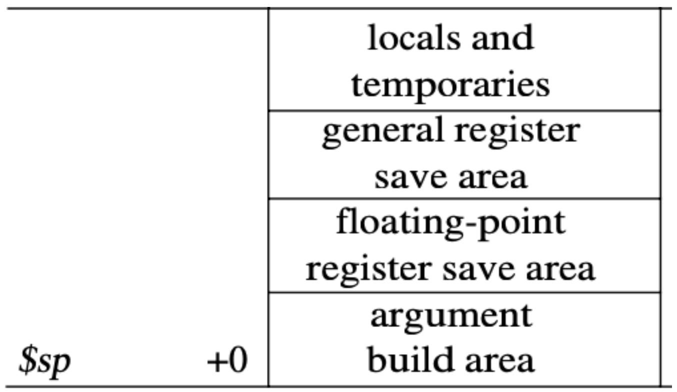
```
int caller(int a, int b) {
  int x = fun1();
  int y = fun2(10, 3, a, b, 5);
  int z = fun3(b, a);
  return x + y + z;
}
```



How many **bytes** of the argument build area does caller need to allocate?

A. 0 bytes (all args passed in registers)
B. 4 bytes (first 4 args passed in registers, 5th on the stack)
C. 20 bytes (all arguments passed on the stack)
D. 20 bytes (first 4 args passed in registers, 5th on the stack)

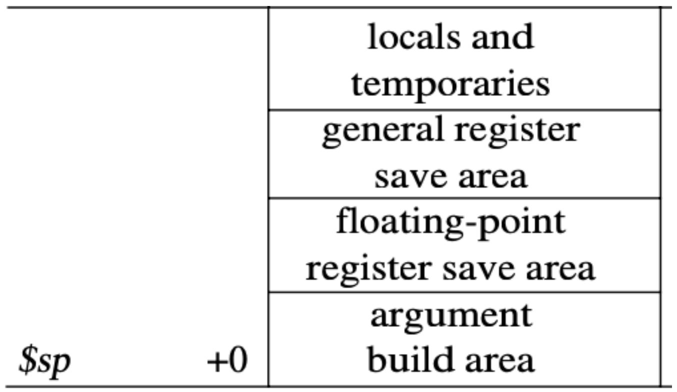
```
int caller(int a, int b) {
  int x = fun1();
  int y = fun2(10, 3, a, b, 5);
  int z = fun3(b, a);
  return x + y + z;
}
```



After the call to fun1, $a0 and $a1 can no longer be assumed to hold values a and b. Where should caller save them prior to calling fun1?

A. Allocate space on the stack to store them
B. Use the argument build area in the stack frame of the function that called caller
C. Allocate space in the heap for them
D. Store them in saved registers $s0 and $s1

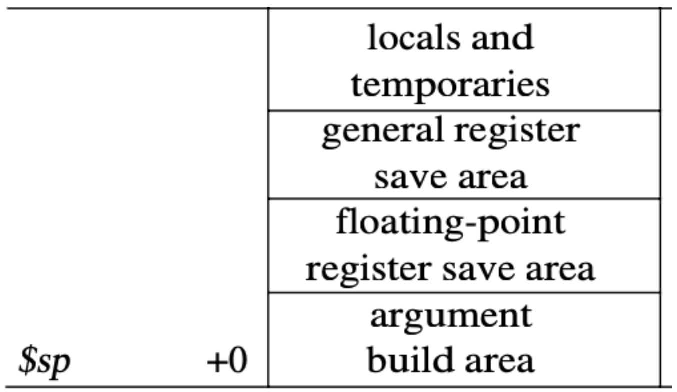
```
int caller(int a, int b) {
  int x = fun1();
  int y = fun2(10, 3, a, b, 5);
  int z = fun3(b, a);
  return x + y + z;
}
```



How many saved registers does caller need to allocate space for on the stack and which ones?

A. 0
B. 1 ($pc)
C. 1 ($ra)
D. 2 ($pc, $ra)
E. 3 ($ra, $s0, $s1)

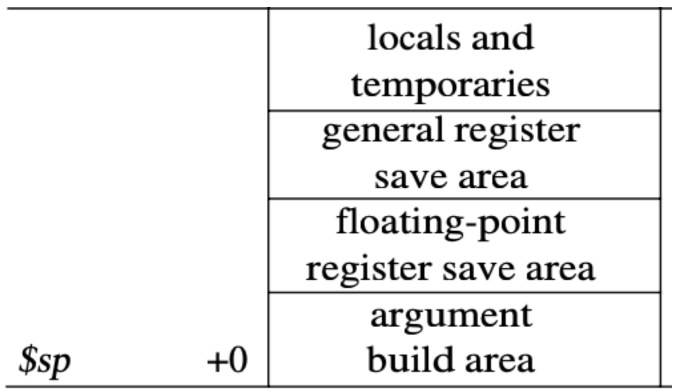
```
int caller(int a, int b) {
  int x = fun1();
  int y = fun2(10, 3, a, b, 5);
  int z = fun3(b, a);
  return x + y + z;
}
```



What is the total size of caller's stack frame if it needs to store 2 local variables, 1 saved register, and 20 bytes of argument build area

A. 23 bytes
B. 24 bytes
C. 32 bytes
D. 36 bytes
E. 40 bytes

```
int caller(int a, int b) {
  int x = fun1();
  int y = fun2(10, 3, a, b, 5);
  int z = fun3(b, a);
  return x + y + z;
}
```



Non-leaf recursive example

```
fact:   addi    $sp, $sp, -24   # allocate stack frame
        sw      $ra, 20($sp)    # save return address
        sw      $a0, 24($sp)    # save in arg build area

        slti    $t0, $a0, 2     # test for n < 2
        beq     $t0, $zero, L1
        addi    $v0, $zero, 1   # if so, result is 1
        j       L2

L1:     addi    $a0, $a0, -1    # else decrement n
        jal     fact            # recursive call
        lw      $a0, 24($sp)    # restore original n
        mul     $v0, $v0, $a0   # multiply to get result

L2:     lw      $ra, 20($sp)    # restore $ra
        addi    $sp, $sp, 24    # deallocate stack frame
        jr      $ra             # return
```

At start of fact(3)

$pc →

```
fact:   addi    $sp, $sp, -24
        sw      $ra, 20($sp)
        sw      $a0, 24($sp)

        slti    $t0, $a0, 2
        beq     $t0, $zero, L1
        addi    $v0, $zero, 1
        j       L2

L1:     addi    $a0, $a0, -1
        jal     fact
        lw      $a0, 24($sp)
        mul     $v0, $v0, $a0

L2:     lw      $ra, 20($sp)
        addi    $sp, $sp, 24
        jr      $ra
```

Reg	Value
$a0	3
$v0	
$ra	main+20

$sp →

After prologue

```
        fact:   addi    $sp, $sp, -24
                sw      $ra, 20($sp)
                sw      $a0, 24($sp)

$pc →           slti    $t0, $a0, 2
                beq     $t0, $zero, L1
                addi    $v0, $zero, 1
                j       L2

        L1:     addi    $a0, $a0, -1
                jal     fact
                lw      $a0, 24($sp)
                mul     $v0, $v0, $a0

        L2:     lw      $ra, 20($sp)
                addi    $sp, $sp, 24
                jr      $ra
```

Reg	Value
$a0	3
$v0	
$ra	main+20

	3
	Return address to main
	(unused for alignment)
$sp →	

At start of fact(2)

$pc →
```
fact:   addi    $sp, $sp, -24
        sw      $ra, 20($sp)
        sw      $a0, 24($sp)

        slti    $t0, $a0, 2
        beq     $t0, $zero, L1
        addi    $v0, $zero, 1
        j       L2

L1:     addi    $a0, $a0, -1
        jal     fact
        lw      $a0, 24($sp)
        mul     $v0, $v0, $a0

L2:     lw      $ra, 20($sp)
        addi    $sp, $sp, 24
        jr      $ra
```

Reg	Value
$a0	2
$v0	
$ra	L1+8

3
Return address to main
(unused for alignment)
$sp →

After prologue

```
       fact:   addi    $sp, $sp, -24
               sw      $ra, 20($sp)
               sw      $a0, 24($sp)

$pc            slti    $t0, $a0, 2
               beq     $t0, $zero, L1
               addi    $v0, $zero, 1
               j       L2

       L1:     addi    $a0, $a0, -1
               jal     fact
               lw      $a0, 24($sp)
               mul     $v0, $v0, $a0

       L2:     lw      $ra, 20($sp)
               addi    $sp, $sp, 24
               jr      $ra
```

Reg	Value
$a0	2
$v0	
$ra	L1+8

	3
	Return address to main
	(unused for alignment)
	2
	Return address to L1+8
	(unused for alignment)
$sp	

At start of fact(1)

```
$pc → fact:   addi    $sp, $sp, -24
              sw      $ra, 20($sp)
              sw      $a0, 24($sp)

              slti    $t0, $a0, 2
              beq     $t0, $zero, L1
              addi    $v0, $zero, 1
              j       L2

      L1:     addi    $a0, $a0, -1
              jal     fact
              lw      $a0, 24($sp)
              mul     $v0, $v0, $a0

      L2:     lw      $ra, 20($sp)
              addi    $sp, $sp, 24
              jr      $ra
```

Reg	Value
$a0	1
$v0	
$ra	L1+8

	3
	Return address to main
	(unused for alignment)
	2
	Return address to L1+8
	(unused for alignment)
$sp →	

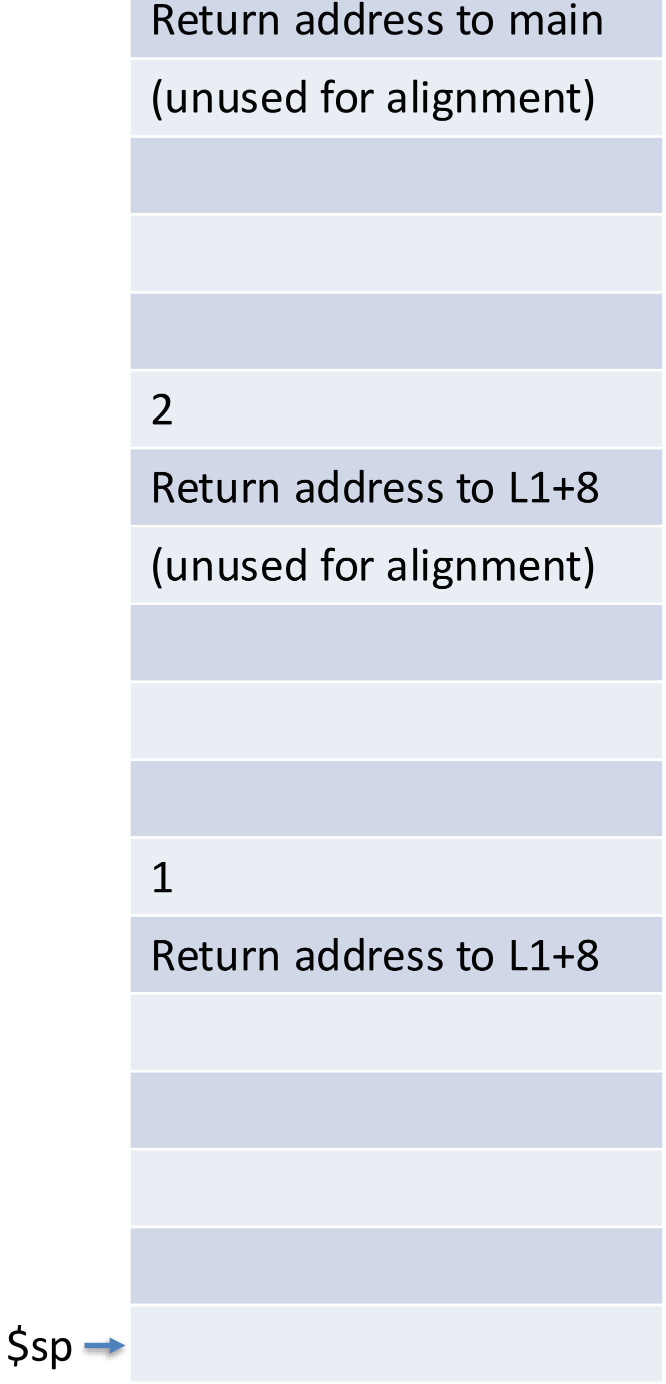
After prologue

```
fact:   addi    $sp, $sp, -24
        sw      $ra, 20($sp)
        sw      $a0, 24($sp)

$pc →   slti    $t0, $a0, 2
        beq     $t0, $zero, L1
        addi    $v0, $zero, 1
        j       L2

L1:     addi    $a0, $a0, -1
        jal     fact
        lw      $a0, 24($sp)
        mul     $v0, $v0, $a0

L2:     lw      $ra, 20($sp)
        addi    $sp, $sp, 24
        jr      $ra
```



Before Epilogue

```
fact:     addi    $sp, $sp, -24
          sw      $ra, 20($sp)
          sw      $a0, 24($sp)

          slti    $t0, $a0, 2
          beq     $t0, $zero, L1
          addi    $v0, $zero, 1
          j       L2

L1:       addi    $a0, $a0, -1
          jal     fact
          lw      $a0, 24($sp)
          mul     $v0, $v0, $a0

$pc → L2: lw      $ra, 20($sp)
          addi    $sp, $sp, 24
          jr      $ra
```

Reg	Value
$a0	1
$v0	1
$ra	L1+8

Return address to main
(unused for alignment)
2
Return address to L1+8
(unused for alignment)
1
Return address to L1+8
$sp →

After Epilogue

```
fact:    addi    $sp, $sp, -24
         sw      $ra, 20($sp)
         sw      $a0, 24($sp)

         slti    $t0, $a0, 2
         beq     $t0, $zero, L1
         addi    $v0, $zero, 1
         j       L2

L1:      addi    $a0, $a0, -1
         jal     fact
         lw      $a0, 24($sp)
         mul     $v0, $v0, $a0

L2:      lw      $ra, 20($sp)
         addi    $sp, $sp, 24
$pc →    jr      $ra
```

Reg	Value
$a0	1
$v0	1
$ra	L1+8

Stack	
	Return address to main
	(unused for alignment)
	2
	Return address to L1+8
	(unused for alignment)
$sp →	1
	Return address to L1+8

After fact(1)

```
       fact:   addi    $sp, $sp, -24
               sw      $ra, 20($sp)
               sw      $a0, 24($sp)

               slti    $t0, $a0, 2
               beq     $t0, $zero, L1
               addi    $v0, $zero, 1
               j       L2

       L1:     addi    $a0, $a0, -1
               jal     fact
$pc →          lw      $a0, 24($sp)
               mul     $v0, $v0, $a0

       L2:     lw      $ra, 20($sp)
               addi    $sp, $sp, 24
               jr      $ra
```

Reg	Value
$a0	1
$v0	1
$ra	L1+8

	3
	Return address to main
	(unused for alignment)
	2
	Return address to L1+8
	(unused for alignment)
$sp →	1

After fact(2)

```
fact:   addi    $sp, $sp, -24
        sw      $ra, 20($sp)
        sw      $a0, 24($sp)

        slti    $t0, $a0, 2
        beq     $t0, $zero, L1
        addi    $v0, $zero, 1
        j       L2

L1:     addi    $a0, $a0, -1
        jal     fact
$pc →   lw      $a0, 24($sp)
        mul     $v0, $v0, $a0

L2:     lw      $ra, 20($sp)
        addi    $sp, $sp, 24
        jr      $ra
```

Reg	Value
$a0	2
$v0	2
$ra	L1+8

	3
	Return address to main
	(unused for alignment)
$sp →	2
	Return address to L1+8
	(unused for alignment)
	1

Before return from fact(3)

```
fact:   addi    $sp, $sp, -24
        sw      $ra, 20($sp)
        sw      $a0, 24($sp)

        slti    $t0, $a0, 2
        beq     $t0, $zero, L1
        addi    $v0, $zero, 1
        j       L2

L1:     addi    $a0, $a0, -1
        jal     fact
        lw      $a0, 24($sp)
        mul     $v0, $v0, $a0

L2:     lw      $ra, 20($sp)
        addi    $sp, $sp, 24
$pc →   jr      $ra
```

Reg	Value
$a0	3
$v0	6
$ra	main+20

$sp →	3
	Return address to main
	(unused for alignment)
	2
	Return address to L1+8
	(unused for alignment)
	1

Stack pointer has been restored!

Why store registers relative to the stack pointer, rather than at some set memory location?

A. Saves space.

B. Easier to figure out where we stored things.

C. Functions won’t overwrite each other’s saves.

D. None of the above

Assembler directives

- Instructions to the assembler
  - .data / .text / .rodata / .bss are used to switch between global (mutable) data, executable code, read-only data, and uninitialized data in the output
  - .word x allocates space for 4 bytes with value x
  - .space n allocates n bytes of space
  - .asciiz “string” writes a 0-terminated string at that location

Review: Arrays!

- How do we declare a 10-word array in our data section?

- Could do

```
.data
x1:     .word  0
x2:     .word  0
x3:     .word  0
...
x10:    .word  0
```

Review: Declaring an Array

- Instead, just declare a big chunk of memory

```
.data
arr:  .space  40
```

```
.data
arr:    .space 40

.text
    li     $t0, 0
    addi   $t1, $t0, 10
    la     $s0, arr
loop:
    beq        $t0, $t1, end
    What goes here?
    addi   $t0, $t0, 1
    j          loop
end:
```

```c
int i;
for (i = 0; i < 10; i++){
        arr[i] = i;
}
```

```
sw      $t0, $t1($s0)
```

A

```
add     $t2, $s0, $t1
sw      $t0, 0($t2)
```

B

```
sw      $t0, 0($s0)
addi    $s0, $s0, 4
```

C

D. More than one of the above

E. None of the above

But what if we don’t know how big the array will be before runtime?

sbrk system call

- Allocates memory and returns its address in $v0
- Amount of memory is specified in bytes in $a0
- Used by malloc, new

System Calls

- Syscalls (when we need OS intervention)
  - I/O (print/read stdout/file)
  - Exit (terminate)
  - Get system time
  - Random values

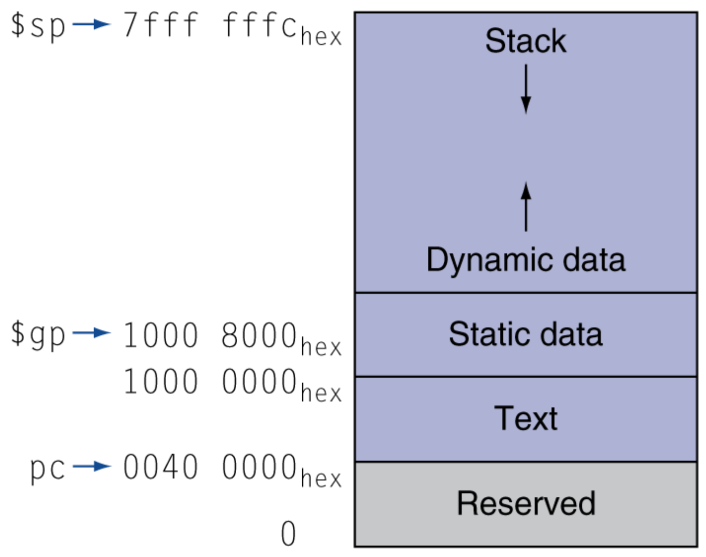
System Calls Review

- How to use:
  - Put syscall number into register $v0
  - Load arguments into argument registers
  - Issue syscall instruction
  - Retrieve return values
- Example (allocate $t4 bytes of memory with sbrk):

```
li      $v0, 9    # sbrk system call number
move    $a0, $t4  # allocate $t4 bytes of mem
syscall
move    $s0, $v0  # $s0 holds a pointer to mem
```

sbrk allocates memory from which region?

A. Stack
B. Dynamic data
C. Static data
D. Text
E. Reserved



What about freeing memory?

- Some operating systems maintain a "program break" which controls the size of the dynamic data
- sbrk requests the OS increment/decrement the break
- malloc()/free() carve the dynamic data up into chunks which the application can use and maintain lists of free chunks
- Freeing memory adds the chunk to a "free list"
- When more memory is needed, the break is changed