

CSCI 210: Computer Architecture

Lecture 12: Procedures & The Stack

Stephen Checkoway
Slides from Cynthia Taylor

Announcements

- Problem Set 3 Due TONIGHT
 - PS 4 available
- Lab 2 due
 - Lab 3 up now

CS History: IBM System 360



- 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


Register values across function calls

- “Preserved” registers
 - You can trust them to persist past function calls
 - Functions must ensure not to change them or to restore them if they do
- Not “Preserved” registers
 - Contents can be changed when you call a function
 - If you need the value, you need to put it somewhere else

Aside: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
\$zero	0	constant 0 (hardware)	n.a.
\$at	1	reserved for assembler	n.a.
\$v0 - \$v1	2-3	returned values	no
\$a0 - \$a3	4-7	arguments	no
\$t0 - \$t7	8-15	temporaries	no
\$s0 - \$s7	16-23	saved values	yes
\$t8 - \$t9	24-25	temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return addr (hardware)	yes

Programmer's responsibility

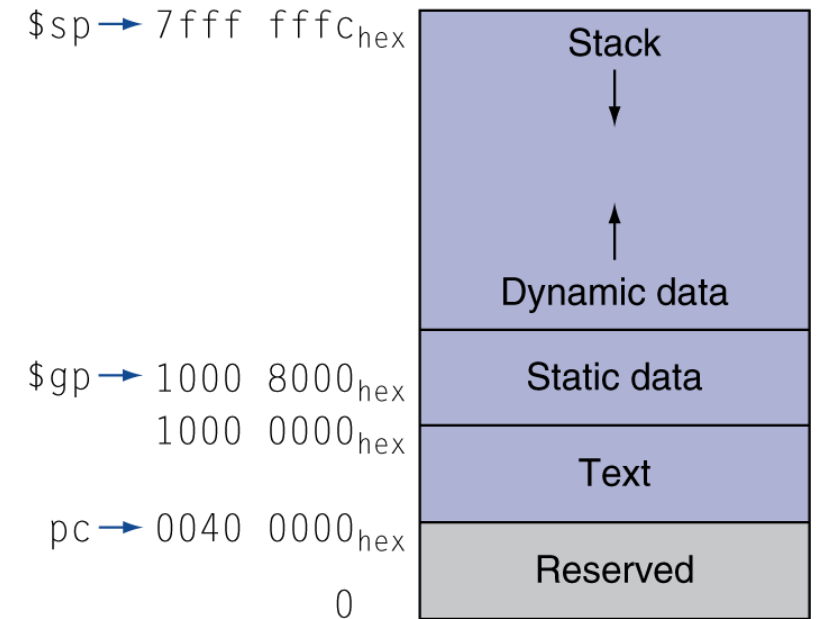


“Spill” and “Fill”

- Spill register **to** memory
 - Whenever you have too many variables to keep in registers
 - Whenever you call a method and need values in non-preserved registers
 - Whenever you want to use a preserved register and need to keep a copy
- Fill registers **from** memory
 - To restore previously spilled registers

Memory Layout

- Text: program code
- Static data: global variables
 - e.g., static variables in C, constant arrays and strings
- Dynamic data: heap
 - E.g., malloc in C, new in Java
- Stack: “automatic” storage for procedures



Before and after a function

Assembly Code

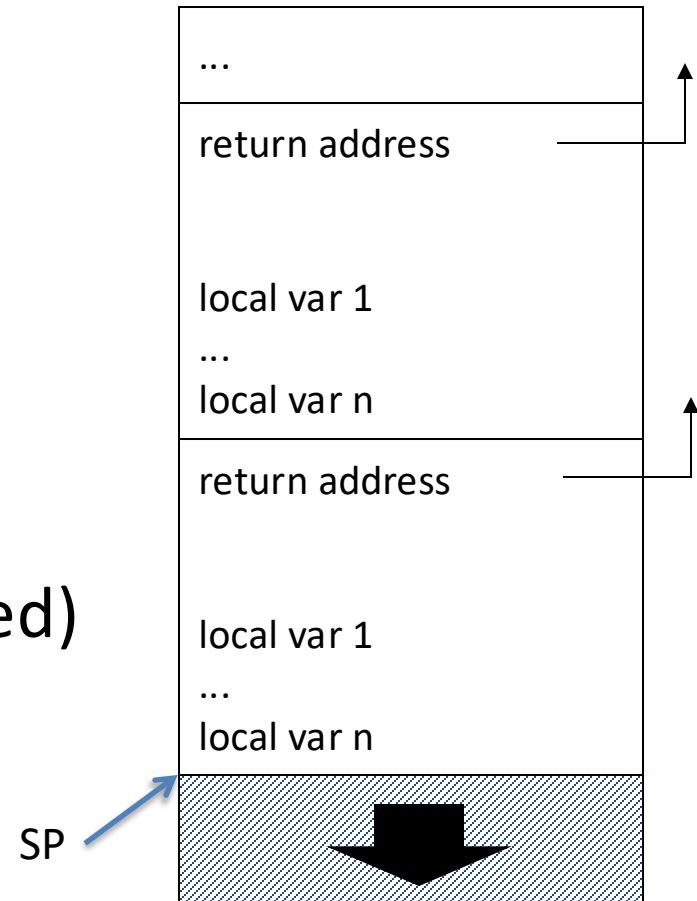
```
sw    $t0, 4($sp)
jal   myFunction
lw    $t0, 4($sp)
```

Which register is being spilled and filled?

- A. \$ra
- B. \$t0
- C. \$sp
- D. No register is spilled/filled
- E. No need to spill/fill any registers

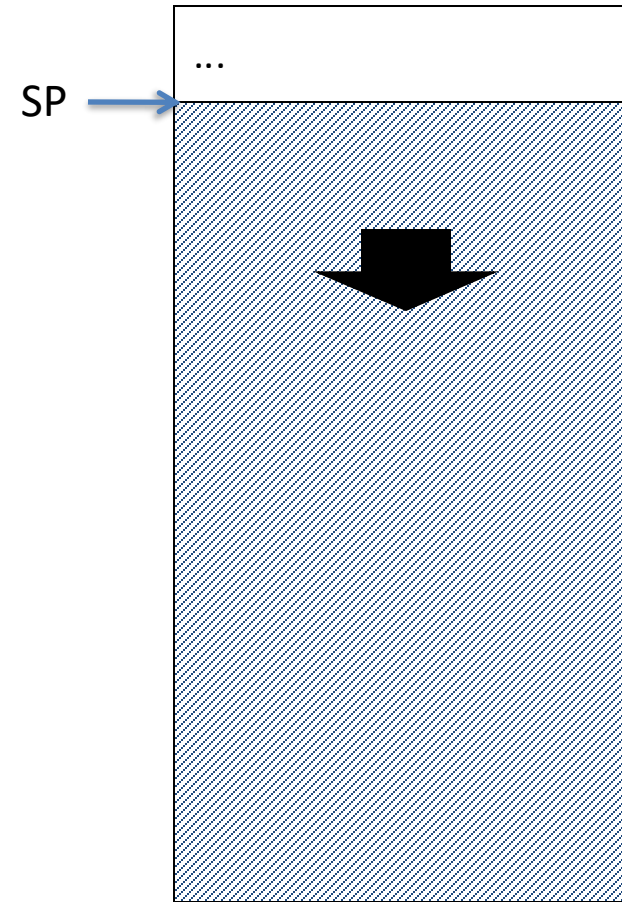
Stack

- Stack of stack frames
 - One per pending procedure
- Each stack frame stores
 - Where to return to
 - Local variables
 - Arguments for called functions (if needed)
- Stack pointer points to last record

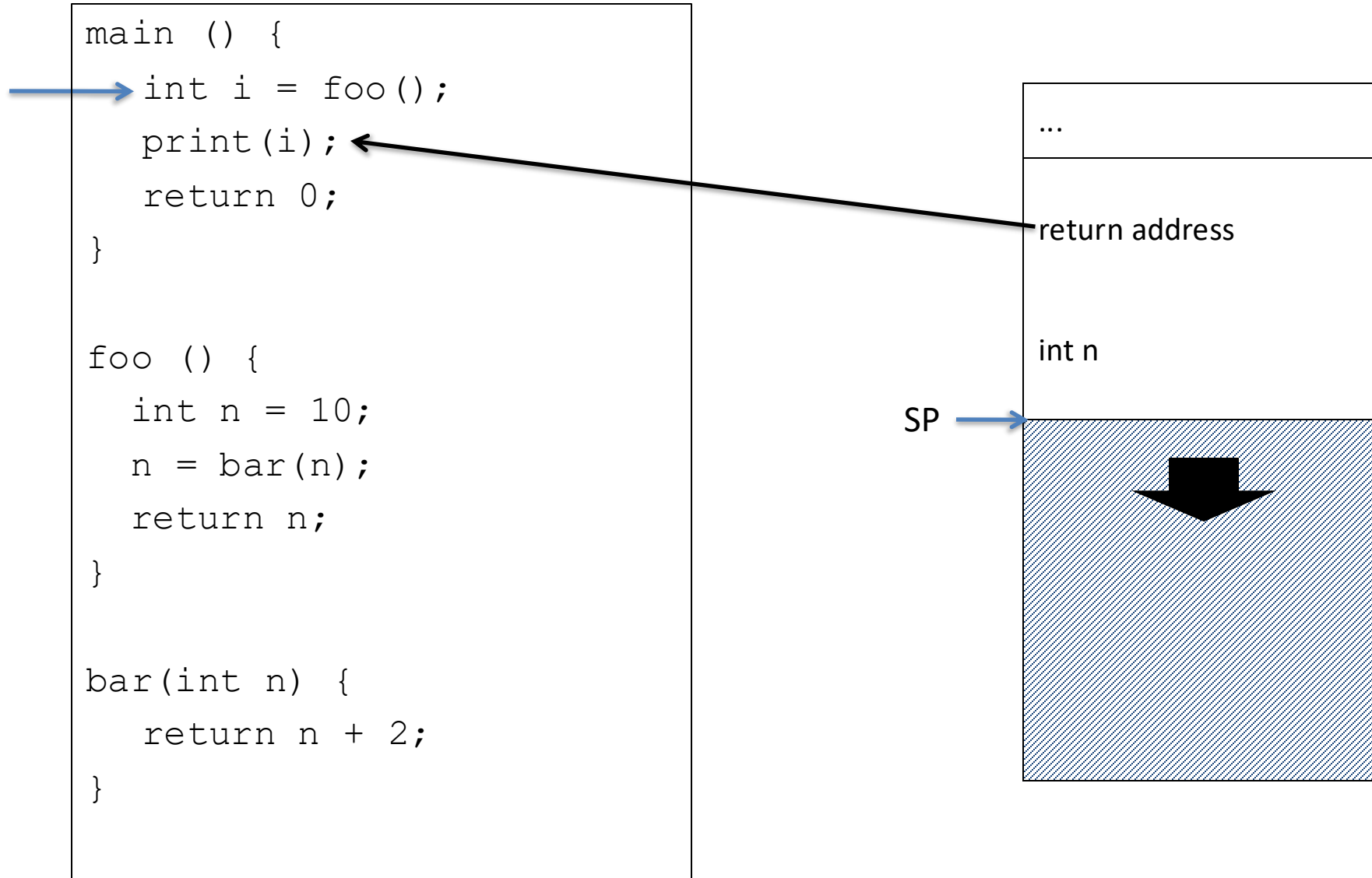


Process Stack

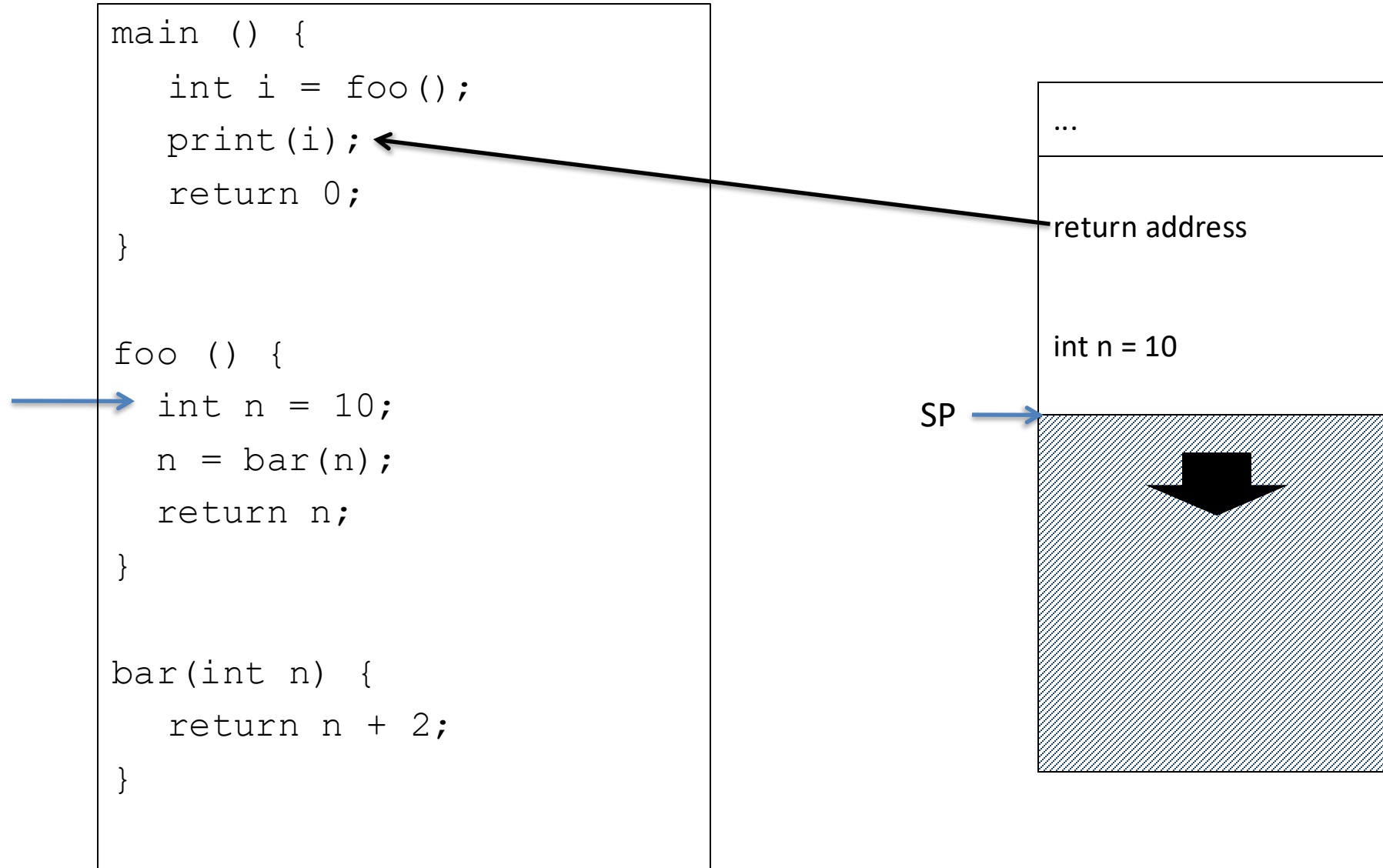
```
main () {  
    int i = foo();  
    print(i);  
    return 0;  
}  
  
foo () {  
    int n = 10;  
    n = bar(n);  
    return n;  
}  
  
bar(int n) {  
    return n + 2;  
}
```



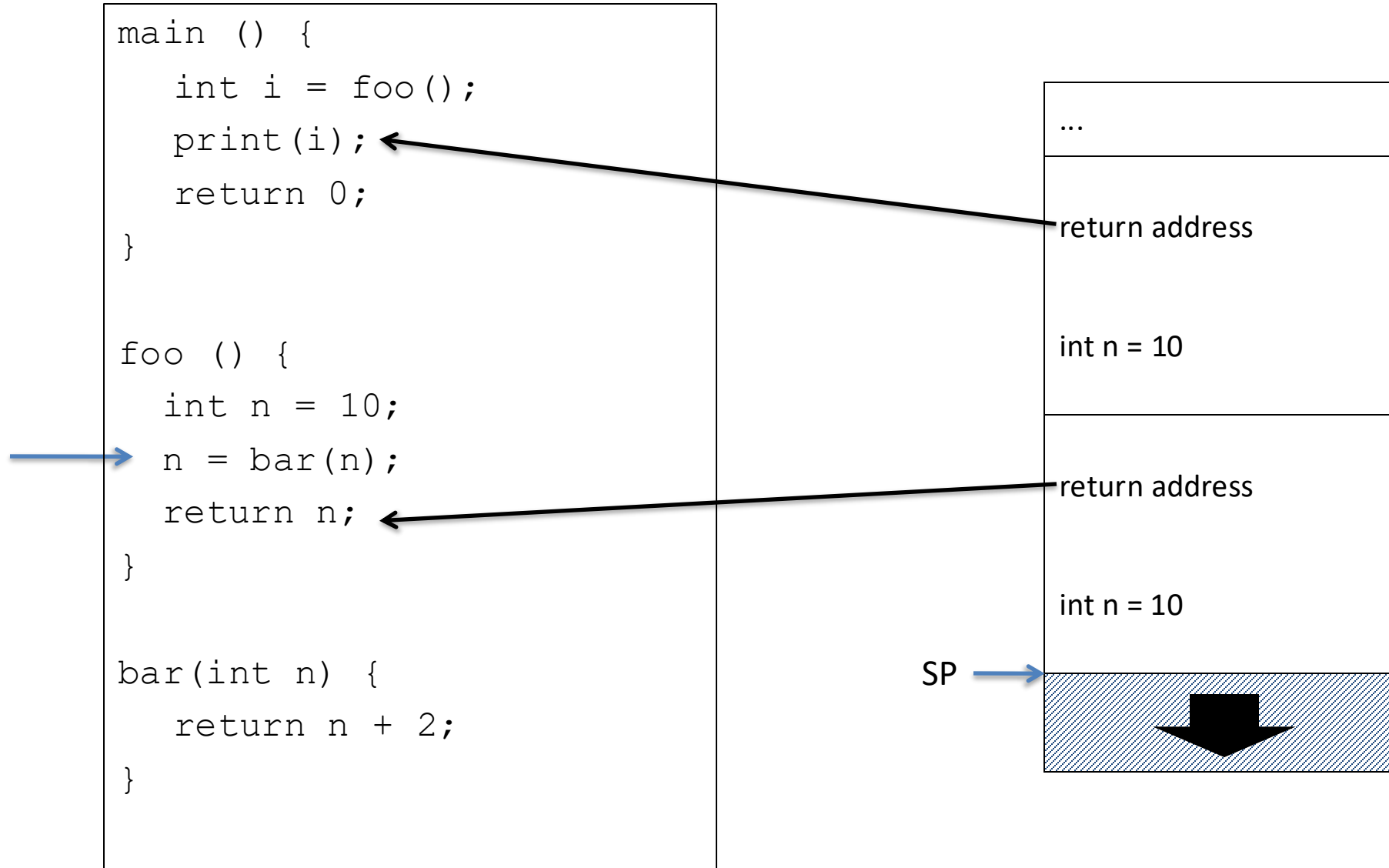
Process Stack



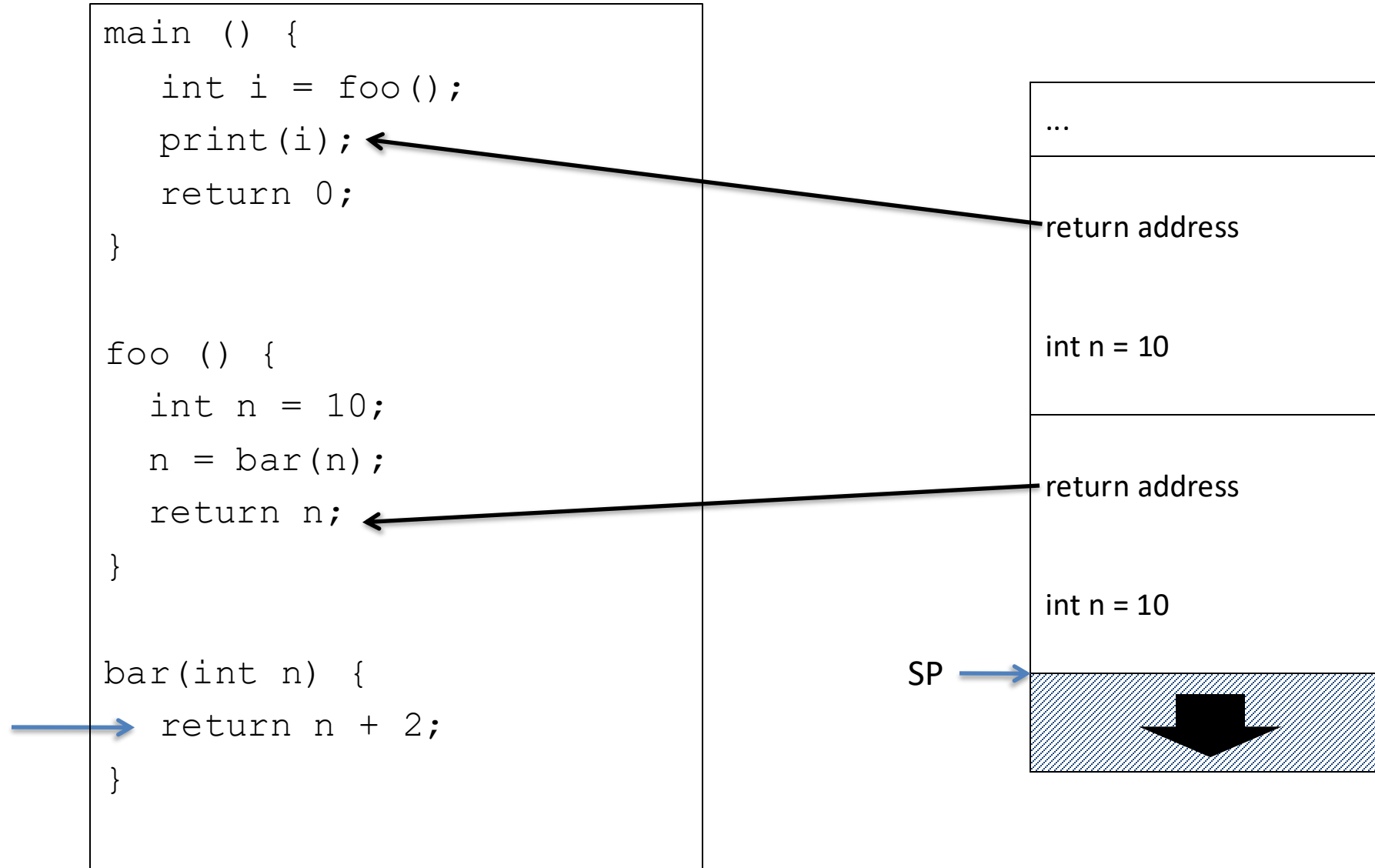
Process Stack



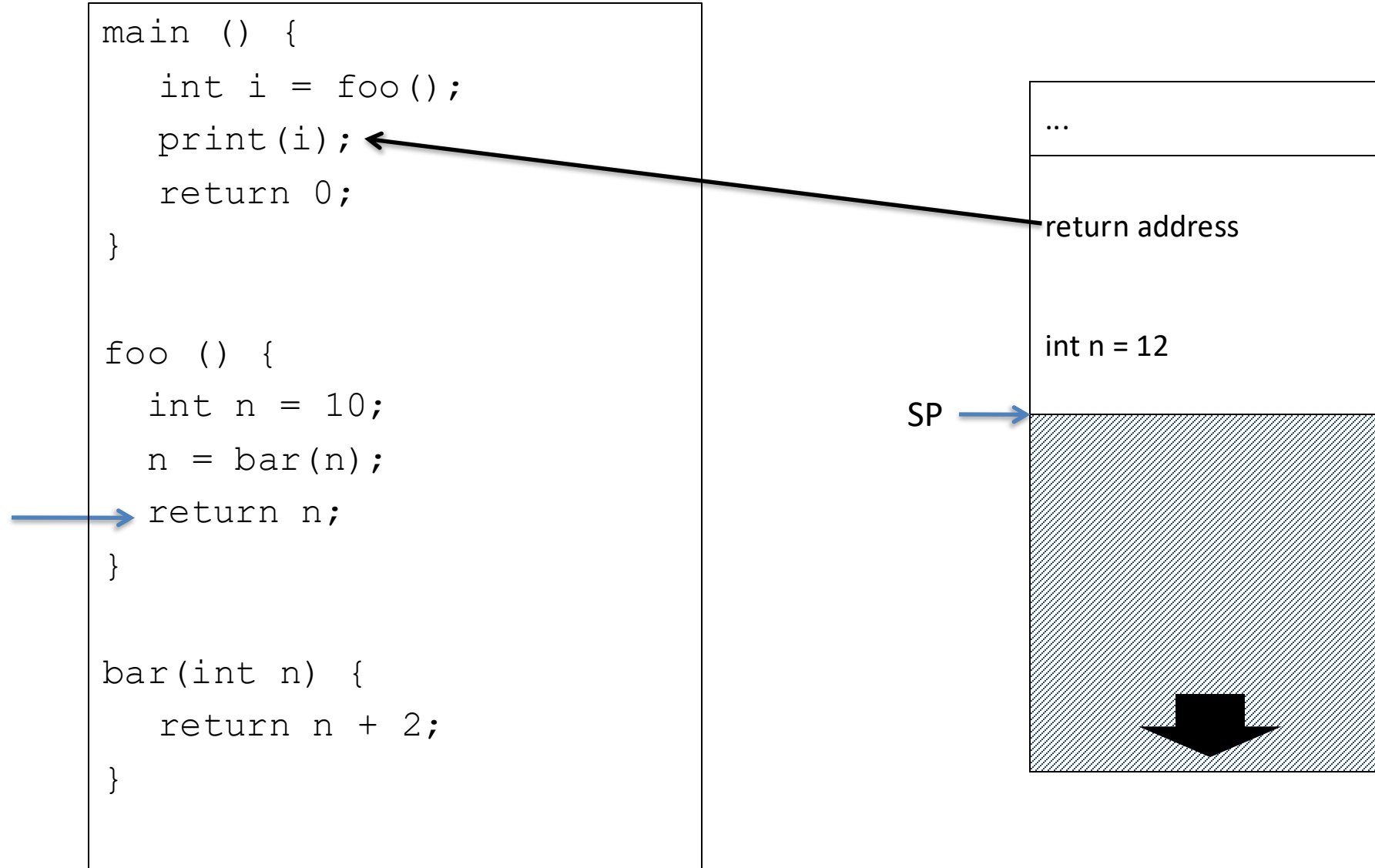
Process Stack



Process Stack

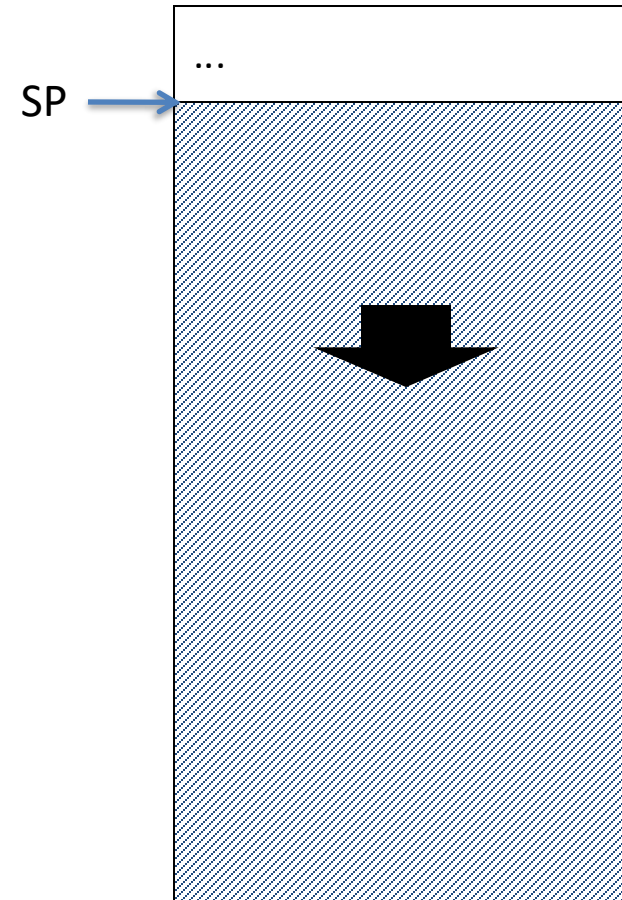


Process Stack



Process Stack

```
main () {  
    int i = foo();  
    print(i);  
    return 0;  
}  
  
foo () {  
    int n = 10;  
    n = bar(n);  
    return n;  
}  
  
bar(int n) {  
    return n + 2;  
}
```



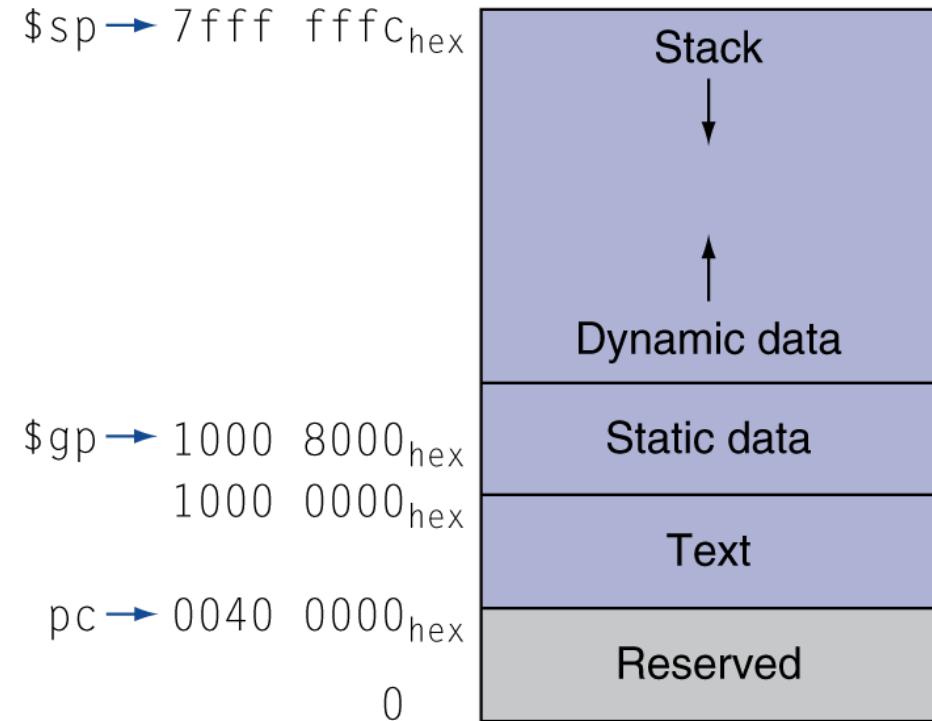
To add a variable to the stack in MIPS

- Change the stack pointer `$sp` to create room on the stack for the variable
- Use `sw` to store the variable on the stack
- The stack pointer in MIPS points after the last stack slot so the valid slots to access are `4($sp)`, `8($sp)`, `12($sp)`, etc.

Stack

If you wish to **push** an integer variable to the top of the stack, which of the following is true:

- A. You should decrement the stack pointer (\$sp) by 1
- B. You should decrement \$sp by 4
- C. You should increment \$sp by 1
- D. You should increment \$sp by 4
- E. None of the above



Manipulating the Stack

- To add the contents of \$s0 to the stack
 - addi \$sp, \$sp, -4
 - sw \$s0, 4(\$sp) ; The stack pointer points after the last stack slot
- To get the value back from the stack
 - lw \$s0, 4(\$sp)
- To “erase” the value from the stack
 - addi \$sp, \$sp, 4

Think-Pair-Share: Why do we spill and fill the return address when we call a function from inside another function?

```
func1:
    . . .
    addi $sp, $sp, -4
    sw   $ra, 4($sp)
    jal  func2
    lw   $ra, 4($sp)
    addi $sp, $sp, 4
    . . .
    jr   $ra
```

A better approach

- In the function “prologue,” reserve space on the stack for all of the variables and saved registers you’ll need
- Use sw/lw to spill and fill as needed to the space reserved in the prologue
- In the function “epilogue,” restore any saved registers you need and update the stack pointer

Complete example

foo:

```
addi    $sp, $sp, -12    # Reserve space for 3 vars
sw      $ra, 12($sp)     # Stores (spills) $ra, return address
sw      $s0, 8($sp)      # Stores (spills) s0, callee-saved reg
...
li      $s0, 25          # Set s0 to 25
sw      $t3, 4($sp)      # Stores (spills) t3, caller-saved reg
add     $a0, $t1, $t3
jal     myFunction
lw      $t3, 4($sp)      # Restores (fills) t3
...
lw      $s0, 8($sp)      # Restores (fills) s0, must restore
lw      $ra, 12($sp)     # Restores (fills) $ra, return address
addi    $sp, $sp, 12     # Restore the stack pointer
jr      $ra              # Return
```

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
```

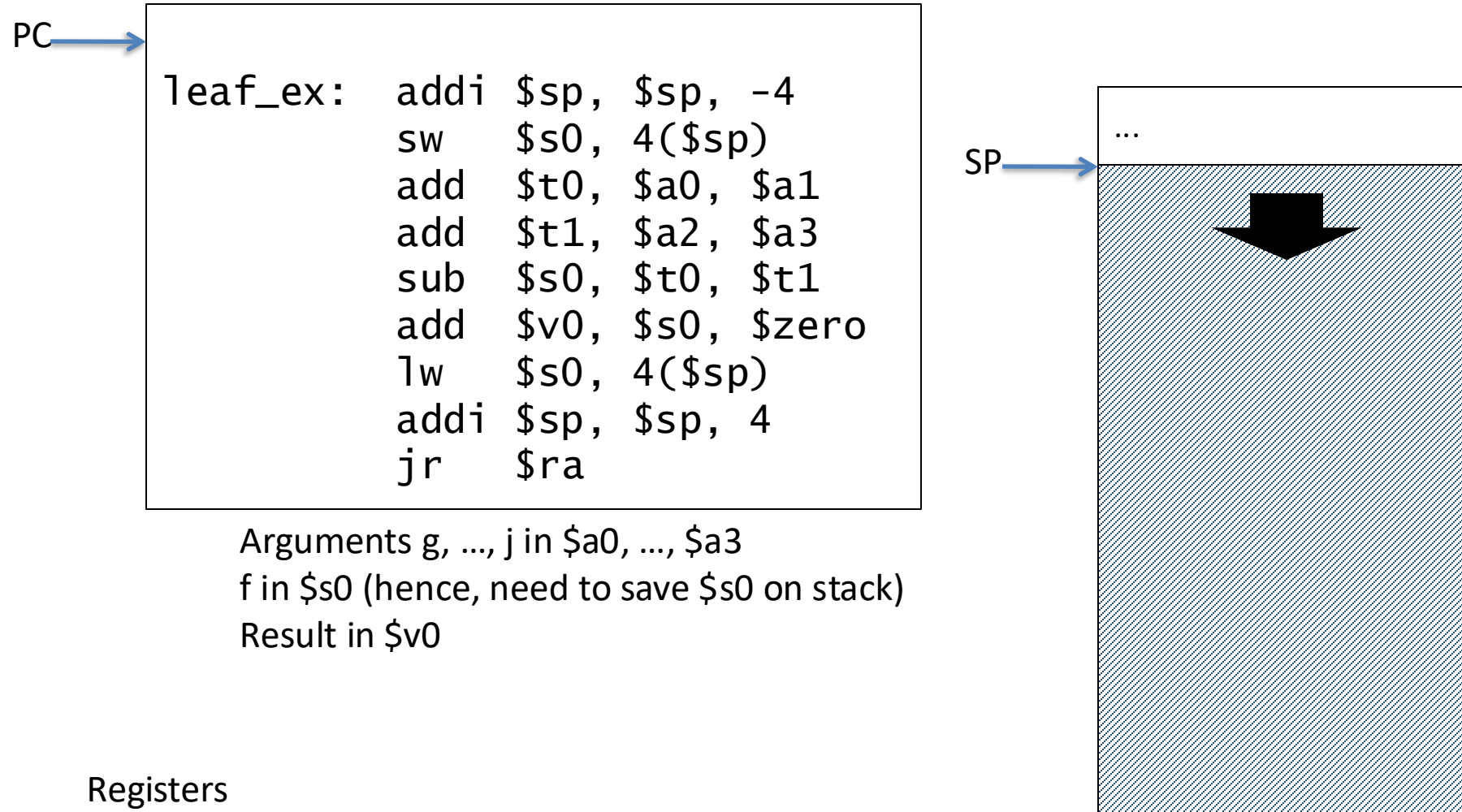
Leaf Procedure Example

```
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
- f in \$s0 (hence, need to save \$s0 on stack)
- Result in \$v0

```
leaf_example:  
    addi $sp, $sp, -4  
    sw    $s0, 4($sp)  
  
    add   $t0, $a0, $a1  
    add   $t1, $a2, $a3  
    sub   $s0, $t0, $t1  
    move  $v0, $s0  
  
    lw    $s0, 4($sp)  
    addi  $sp, $sp, 4  
    jr    $ra
```


Process Stack



Registers

\$s0: 25 \$a0: 5 \$a1: 2 \$a2: 3 \$a3: 1
\$t0: \$t1: \$v0:

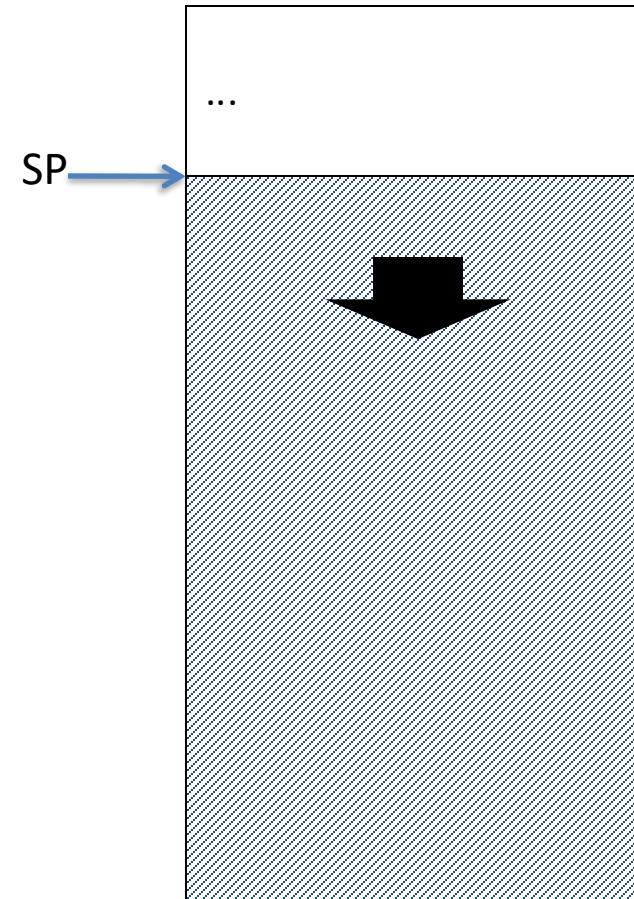
Process Stack

PC → leaf_ex: addi \$sp, \$sp, -4
 sw \$s0, 4(\$sp)
 add \$t0, \$a0, \$a1
 add \$t1, \$a2, \$a3
 sub \$s0, \$t0, \$t1
 add \$v0, \$s0, \$zero
 lw \$s0, 4(\$sp)
 addi \$sp, \$sp, 4
 jr \$ra

Arguments g, ..., j in \$a0, ..., \$a3
f in \$s0 (hence, need to save \$s0 on stack)
Result in \$v0

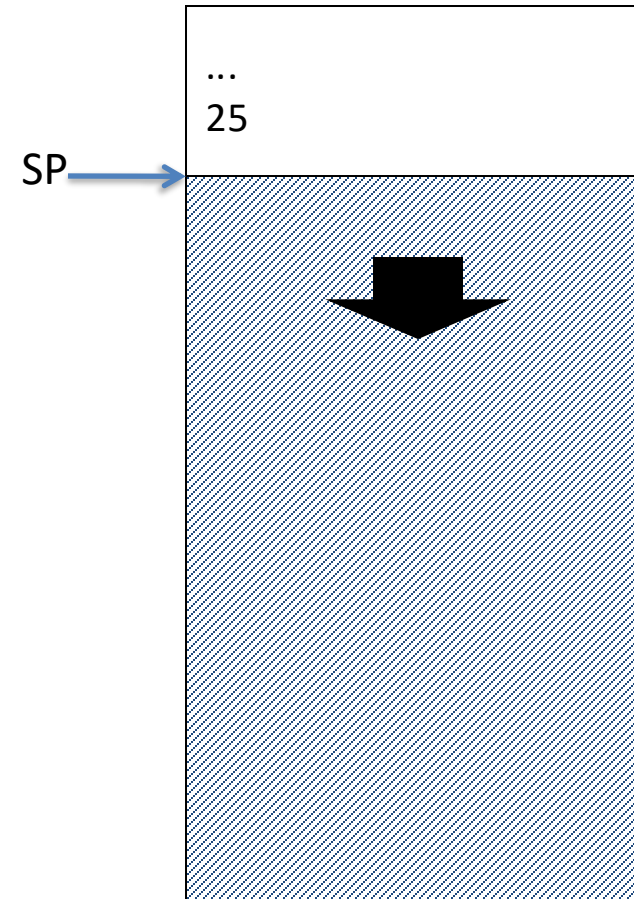
Registers

\$s0: 25 \$a0: 5 \$a1: 2 \$a2: 3 \$a3: 1
\$t0: \$t1: \$v0:



Process Stack

```
leaf_ex:  addi $sp, $sp, -4
          PC → sw  $s0, 4($sp)
          add  $t0, $a0, $a1
          add  $t1, $a2, $a3
          sub  $s0, $t0, $t1
          add  $v0, $s0, $zero
          lw   $s0, 4($sp)
          addi $sp, $sp, 4
          jr   $ra
```



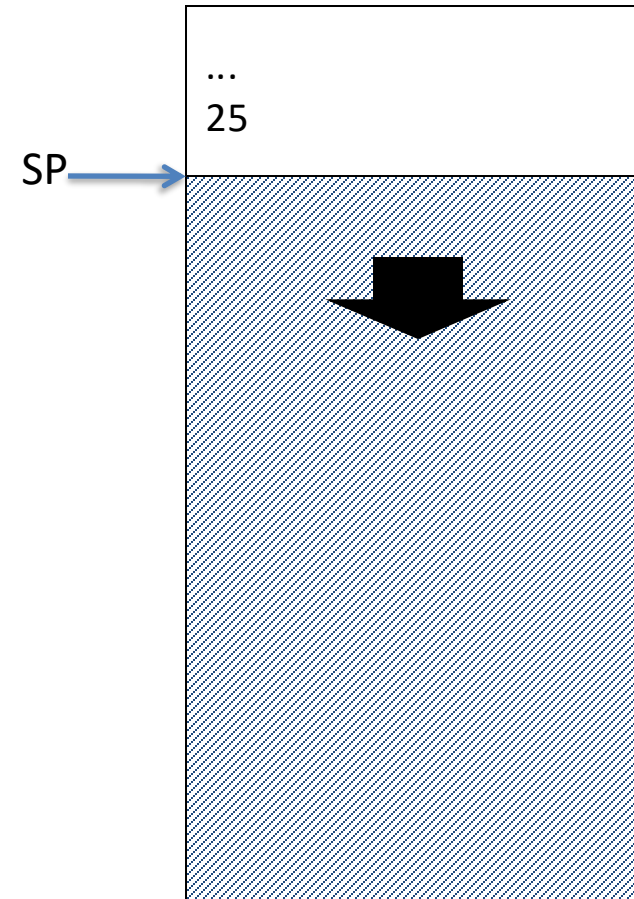
Arguments g, ..., j in \$a0, ..., \$a3
f in \$s0 (hence, need to save \$s0 on stack)
Result in \$v0

Registers

\$s0: 25	\$a0: 5	\$a1: 2	\$a2: 3	\$a3: 1
\$t0:	\$t1:	\$v0:		

Process Stack

```
leaf_ex:  addi $sp, $sp, -4
          sw   $s0, 4($sp)
          PC → add $t0, $a0, $a1
          add $t1, $a2, $a3
          sub  $s0, $t0, $t1
          add  $v0, $s0, $zero
          lw   $s0, 4($sp)
          addi $sp, $sp, 4
          jr   $ra
```



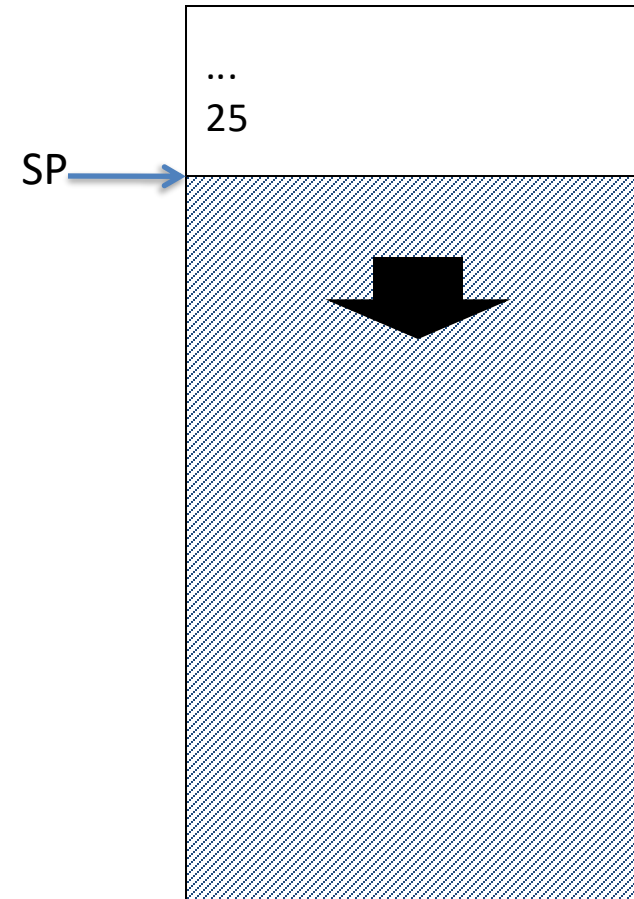
Arguments g, ..., j in \$a0, ..., \$a3
f in \$s0 (hence, need to save \$s0 on stack)
Result in \$v0

Registers

\$s0: 25 \$a0: 5 \$a1: 2 \$a2: 3 \$a3: 1
\$t0: 7 \$t1: \$v0:

Process Stack

```
leaf_ex:  addi $sp, $sp, -4
          sw   $s0, 4($sp)
          add  $t0, $a0, $a1
PC →      add  $t1, $a2, $a3
          sub  $s0, $t0, $t1
          add  $v0, $s0, $zero
          lw   $s0, 4($sp)
          addi $sp, $sp, 4
          jr   $ra
```



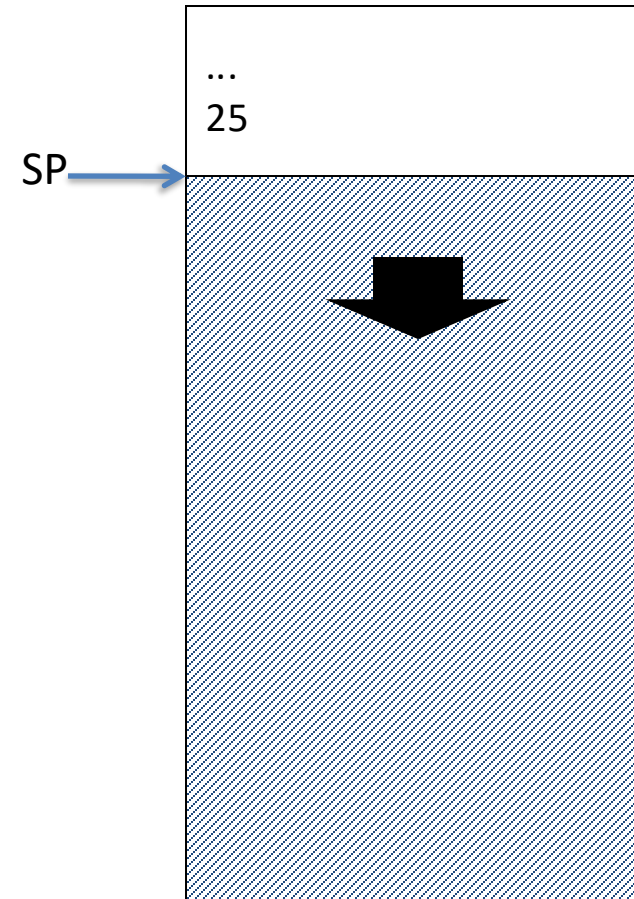
Arguments g, ..., j in \$a0, ..., \$a3
f in \$s0 (hence, need to save \$s0 on stack)
Result in \$v0

Registers

\$s0: 25 \$a0: 5 \$a1: 2 \$a2: 3 \$a3: 1
\$t0: 7 **\$t1: 4** \$v0:

Process Stack

```
leaf_ex:  addi $sp, $sp, -4
          sw   $s0, 4($sp)
          add  $t0, $a0, $a1
          add  $t1, $a2, $a3
PC →      sub  $s0, $t0, $t1
          add  $v0, $s0, $zero
          lw   $s0, 4($sp)
          addi $sp, $sp, 4
          jr   $ra
```



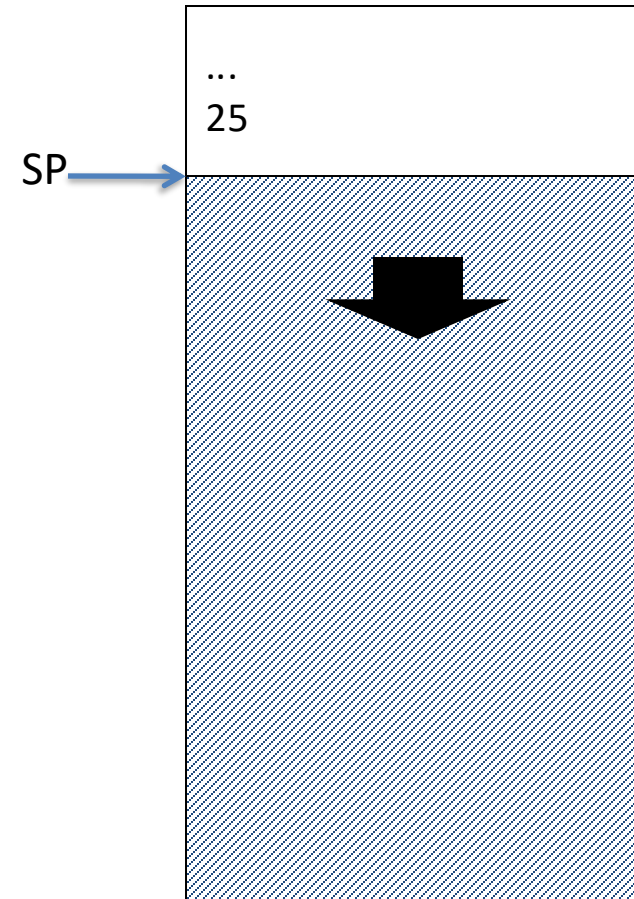
Arguments g, ..., j in \$a0, ..., \$a3
f in \$s0 (hence, need to save \$s0 on stack)
Result in \$v0

Registers

\$s0: 3 \$a0: 5 \$a1: 2 \$a2: 3 \$a3: 1
\$t0: 7 \$t1: 4 \$v0:

Process Stack

```
leaf_ex:  addi $sp, $sp, -4
          sw   $s0, 4($sp)
          add  $t0, $a0, $a1
          add  $t1, $a2, $a3
          sub  $s0, $t0, $t1
PC →      add  $v0, $s0, $zero
          lw   $s0, 4($sp)
          addi $sp, $sp, 4
          jr   $ra
```



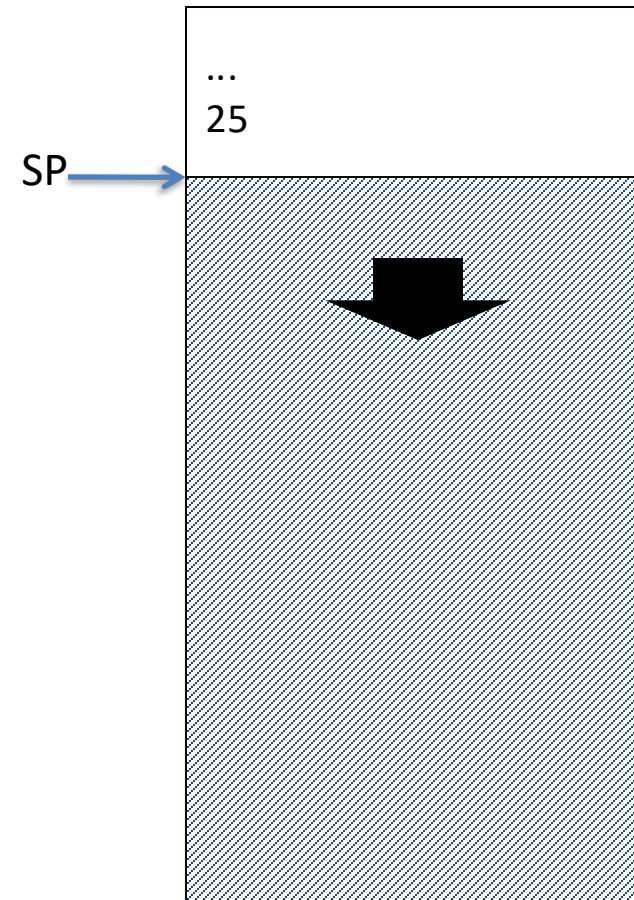
Arguments g, ..., j in \$a0, ..., \$a3
f in \$s0 (hence, need to save \$s0 on stack)
Result in \$v0

Registers

\$s0: 3	\$a0: 5	\$a1: 2	\$a2: 3	\$a3: 1
\$t0: 7	\$t1: 4	\$v0: 3		

Process Stack

```
leaf_ex:  addi $sp, $sp, -4
          sw   $s0, 4($sp)
          add  $t0, $a0, $a1
          add  $t1, $a2, $a3
          sub  $s0, $t0, $t1
          add  $v0, $s0, $zero
PC →      lw   $s0, 4($sp)
          addi $sp, $sp, 4
          jr   $ra
```



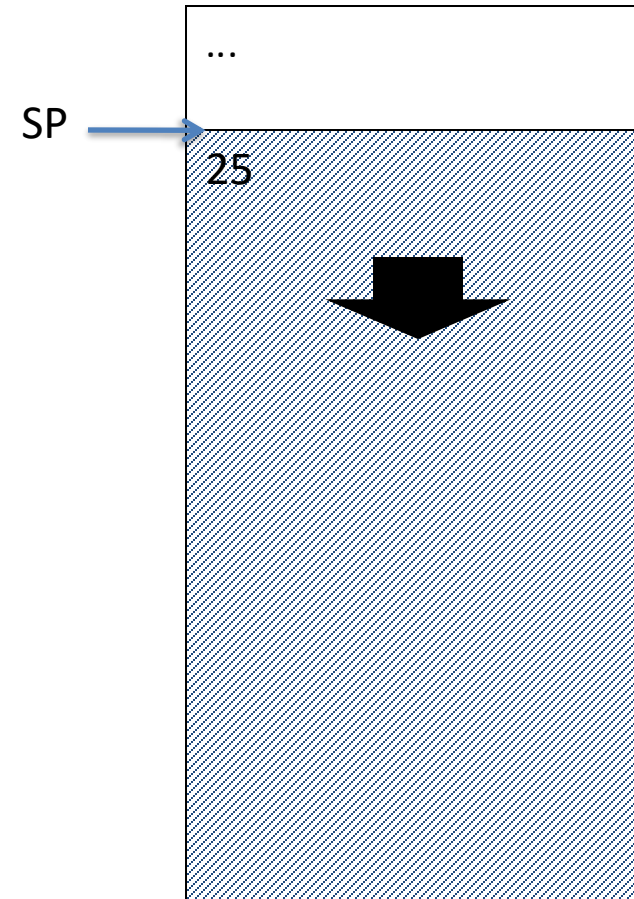
Arguments g, ..., j in \$a0, ..., \$a3
f in \$s0 (hence, need to save \$s0 on stack)
Result in \$v0

Registers

\$s0: 25 \$a0: 5 \$a1: 2 \$a2: 3 \$a3: 1
\$t0: 7 \$t1: 4 \$v0: 3

Process Stack

```
leaf_ex:  addi $sp, $sp, -4
          sw   $s0, 4($sp)
          add  $t0, $a0, $a1
          add  $t1, $a2, $a3
          sub  $s0, $t0, $t1
          add  $v0, $s0, $zero
          lw   $s0, 4($sp)
PC →      addi $sp, $sp, 4
          jr   $ra
```



Arguments g, ..., j in \$a0, ..., \$a3
f in \$s0 (hence, need to save \$s0 on stack)
Result in \$v0

Registers

\$s0: 25 \$a0: 5 \$a1: 2 \$a2: 3 \$a3: 1
\$t0: 7 \$t1: 4 \$v0: 3

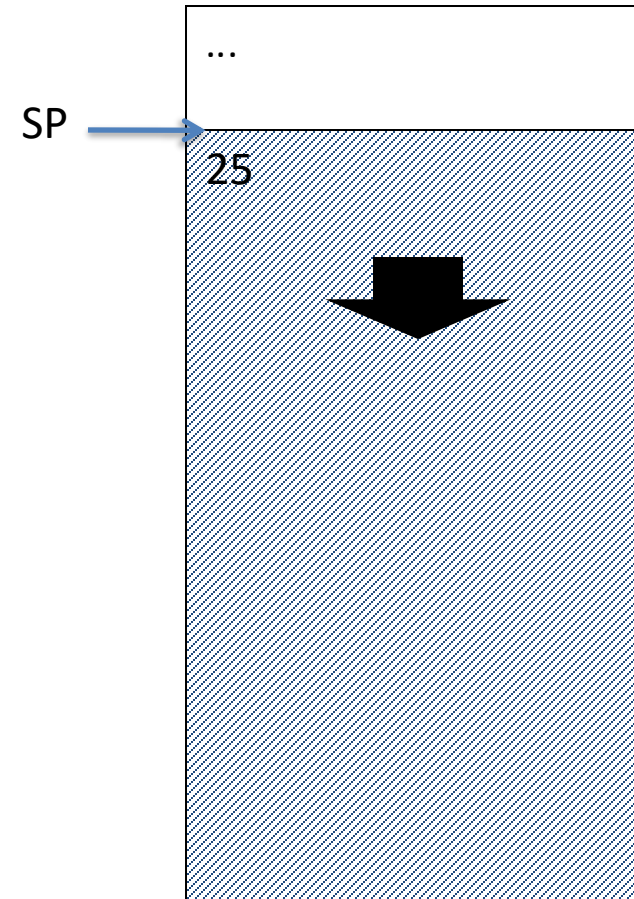
Process Stack

```
leaf_ex:  addi $sp, $sp, -4
          sw   $s0, 4($sp)
          add  $t0, $a0, $a1
          add  $t1, $a2, $a3
          sub  $s0, $t0, $t1
          add  $v0, $s0, $zero
          lw   $s0, 4($sp)
          addi $sp, $sp, 4
PC →      jr   $ra
```

Arguments g, ..., j in \$a0, ..., \$a3
f in \$s0 (hence, need to save \$s0 on stack)
Result in \$v0

Registers

\$s0: 25 \$a0: 5 \$a1: 2 \$a2: 3 \$a3: 1
\$t0: 7 \$t1: 4 \$v0: 3



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, 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

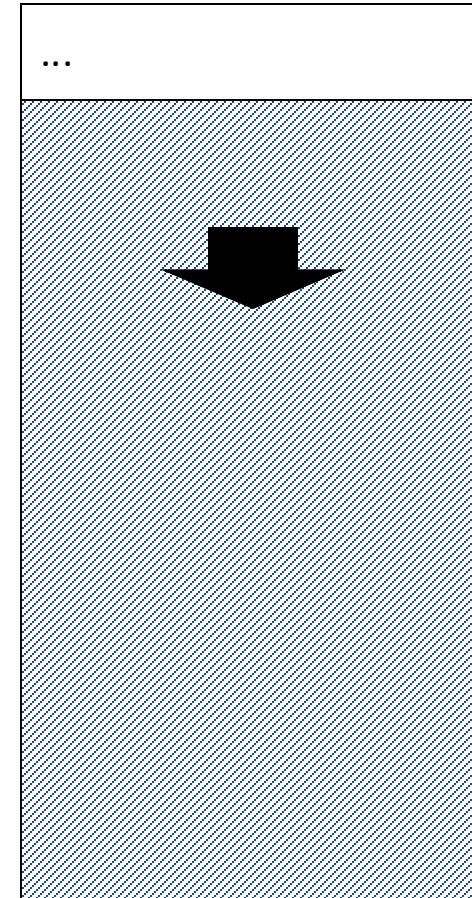
- C code:

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

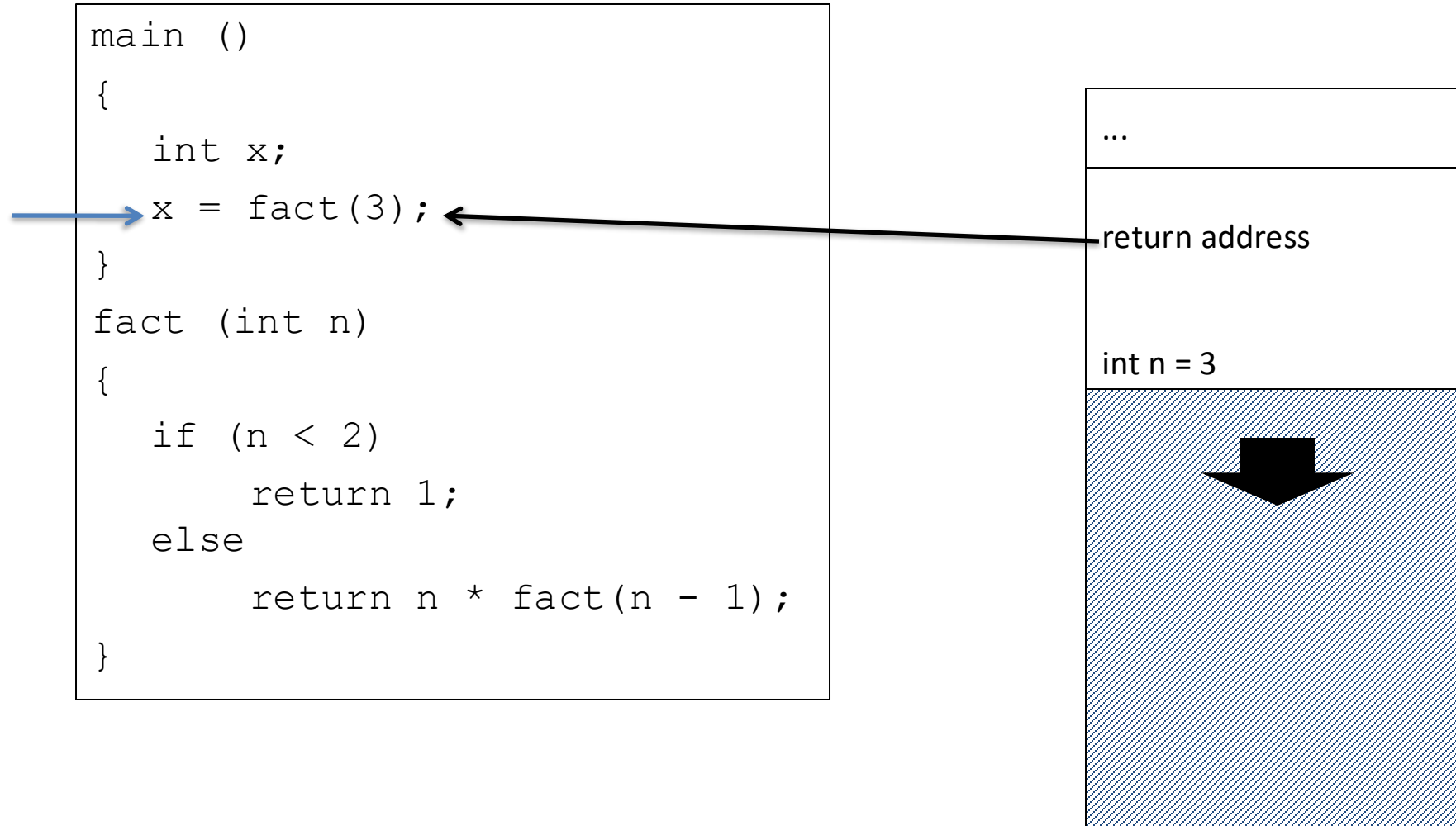
- Argument n in \$a0
- Result in \$v0

Process Stack

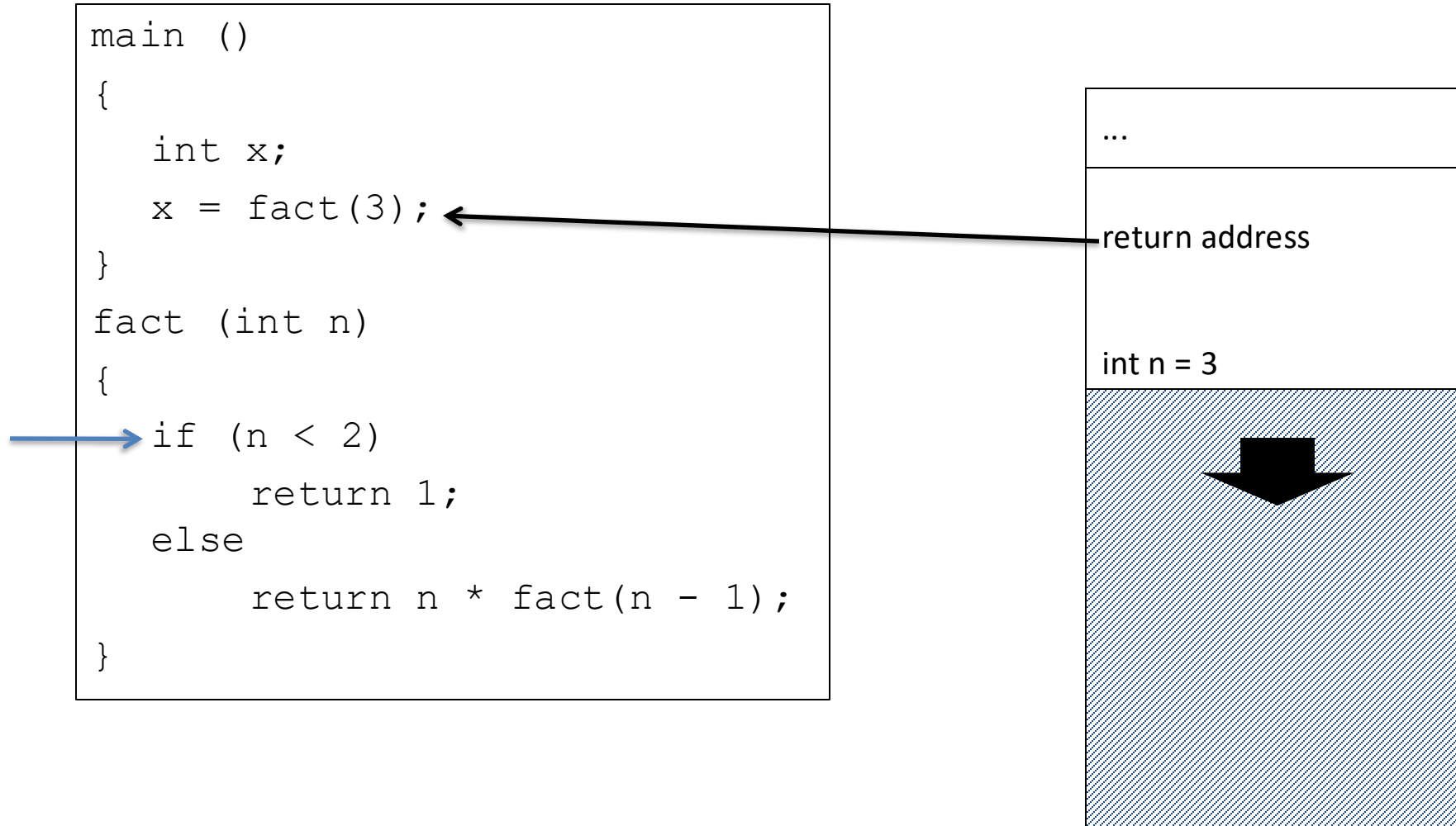
```
main ()
{
    int x;
    x = fact(3);
}
fact (int n)
{
    if (n < 2)
        return 1;
    else
        return n * fact(n - 1);
}
```



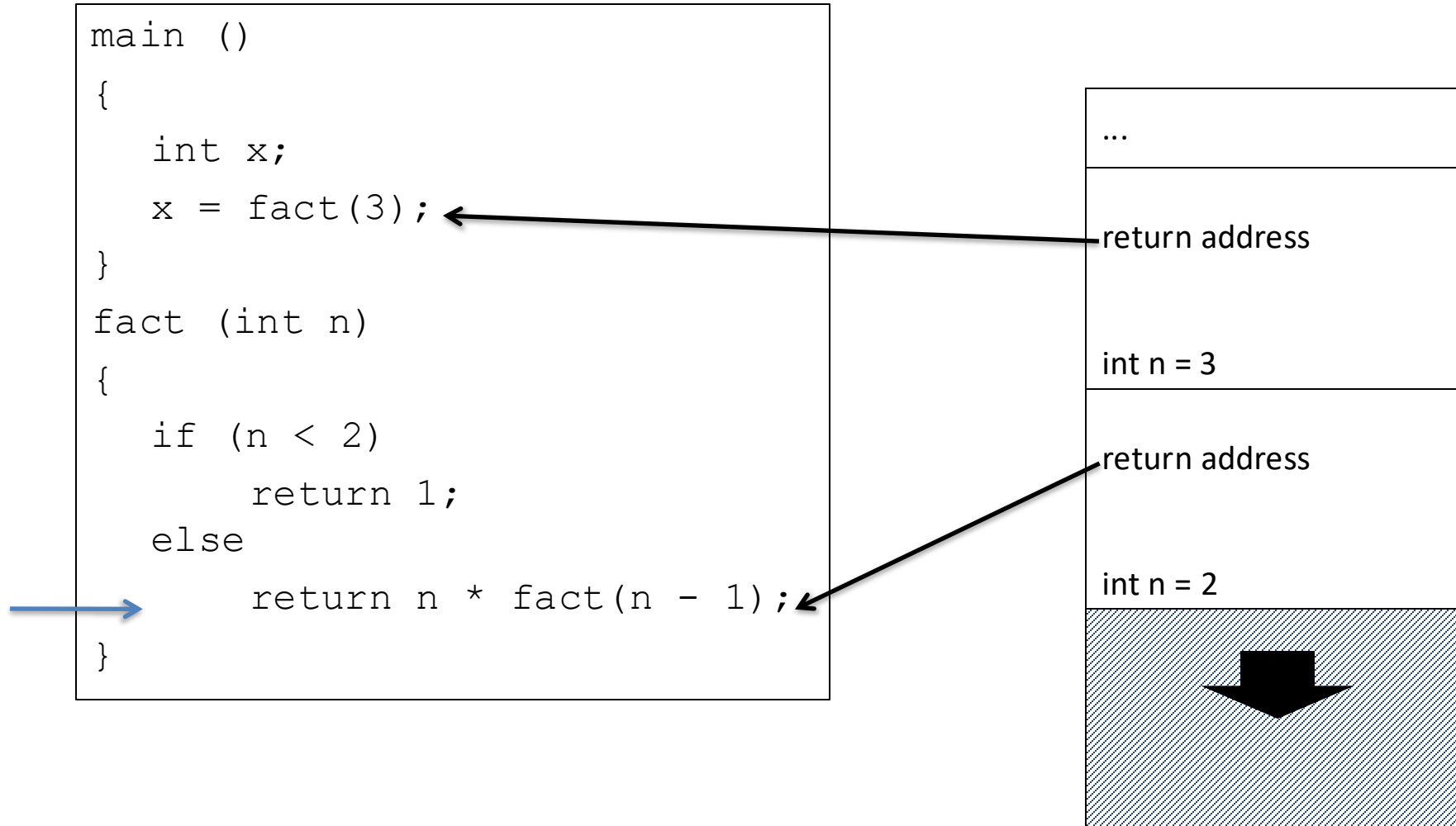
Process Stack



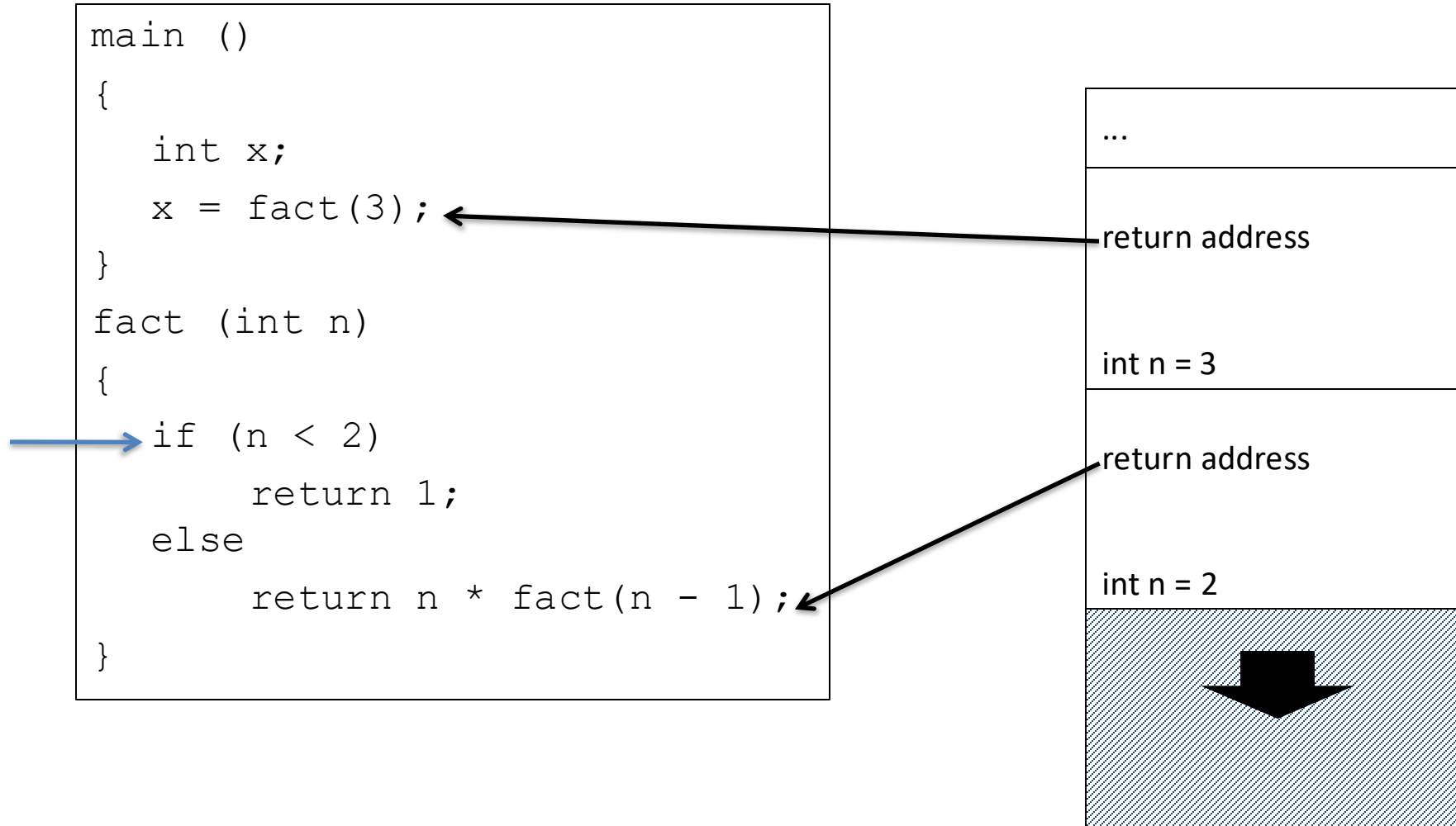
Process Stack



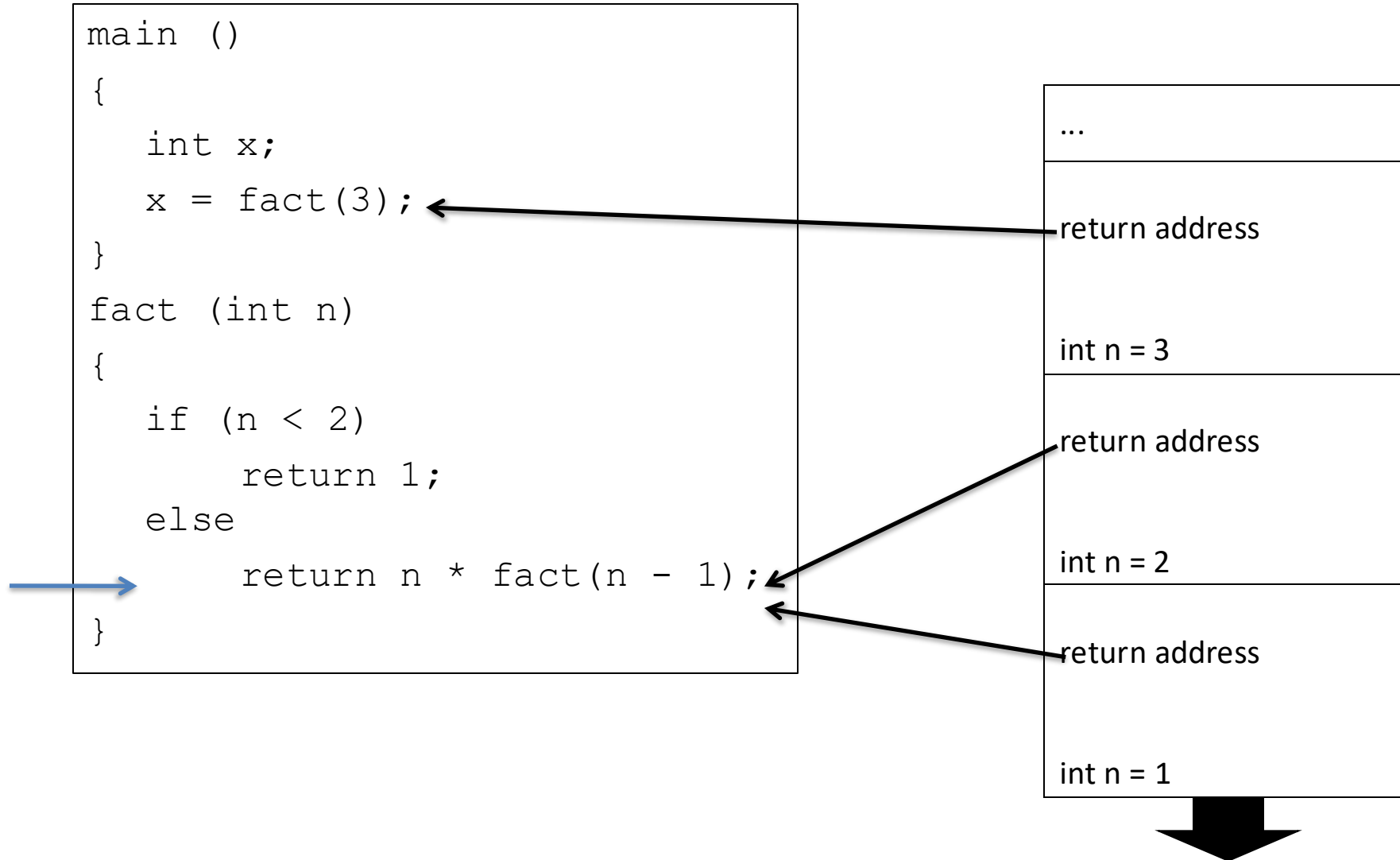
Process Stack



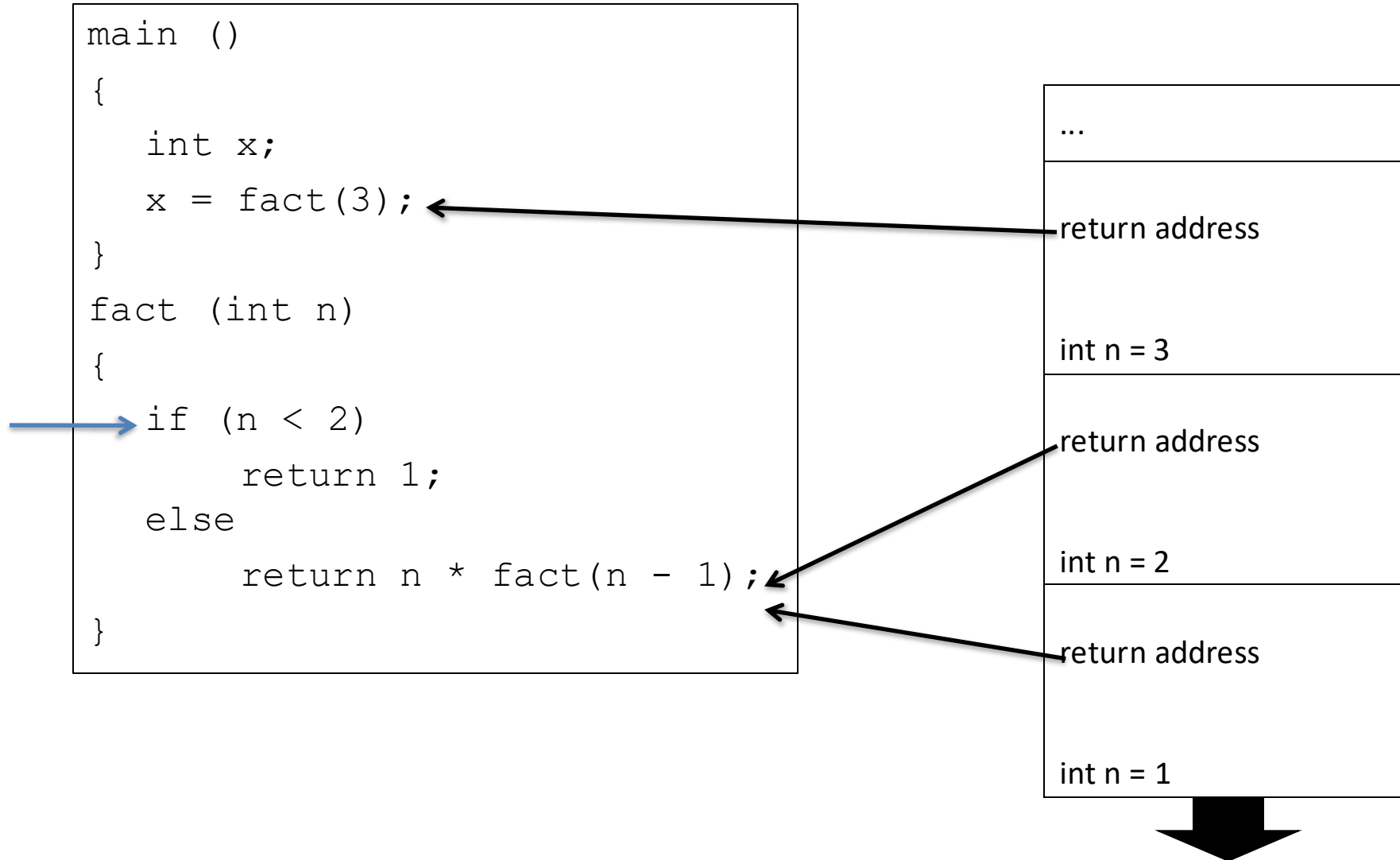
Process Stack



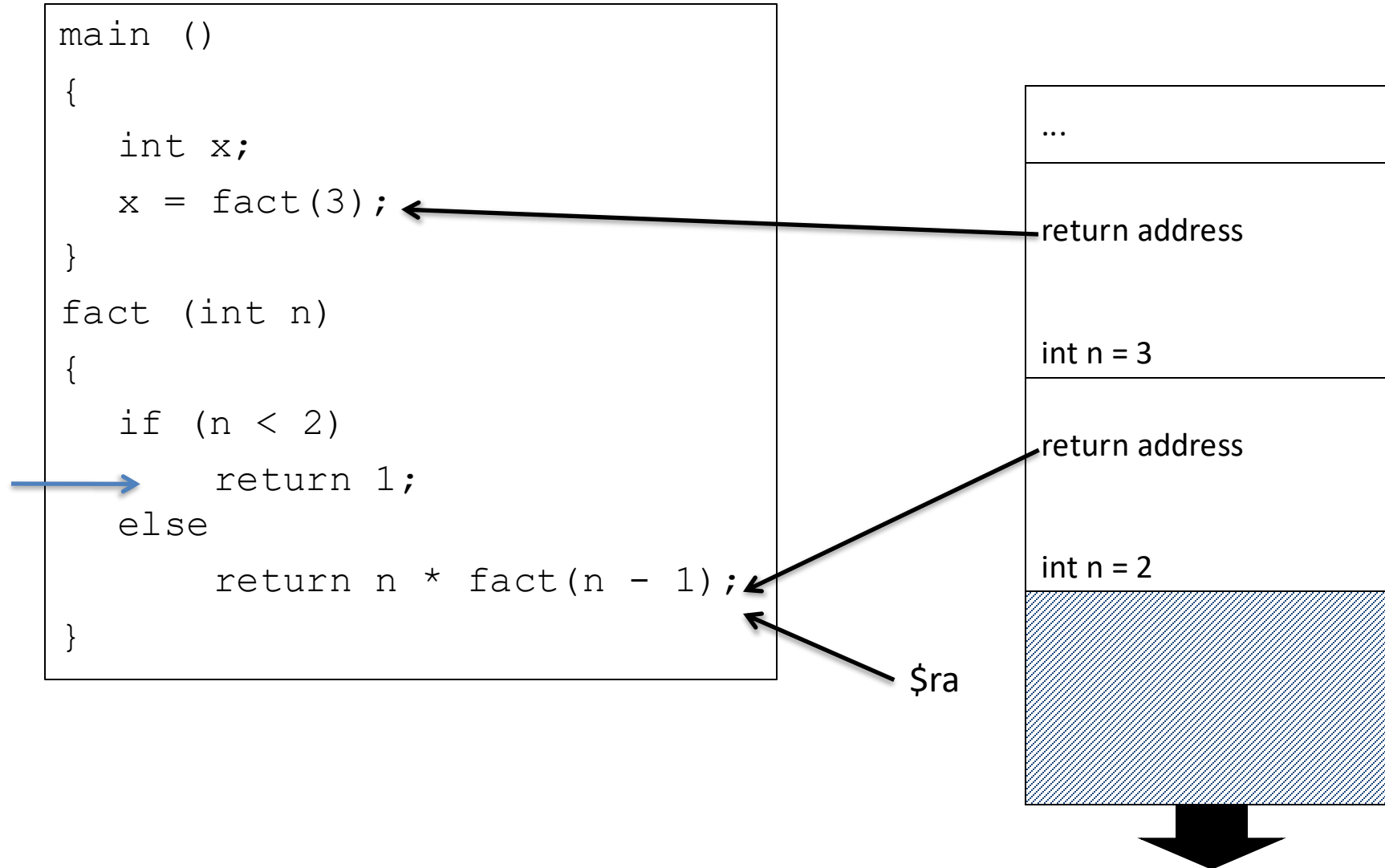
Process Stack



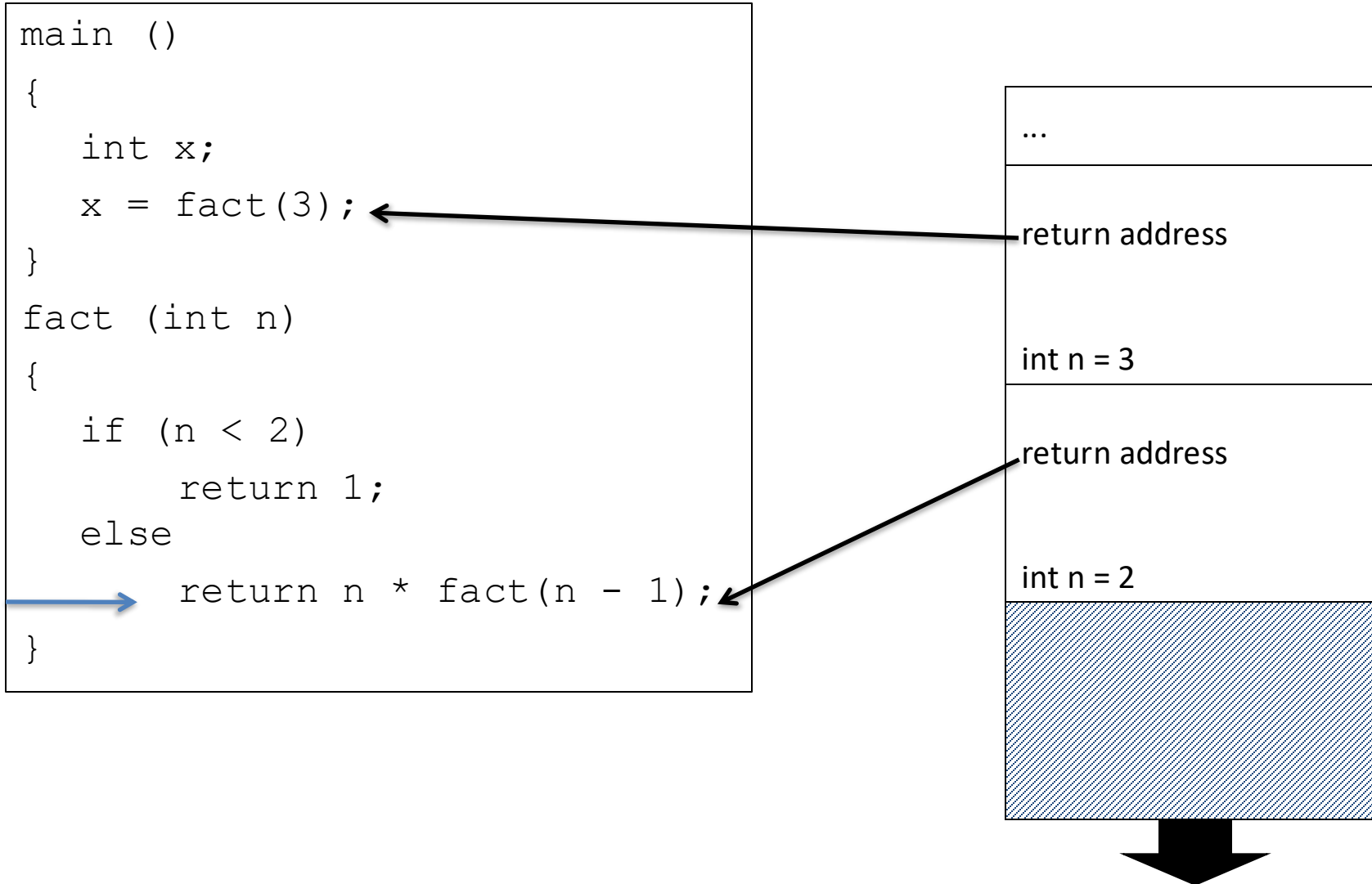
Process Stack



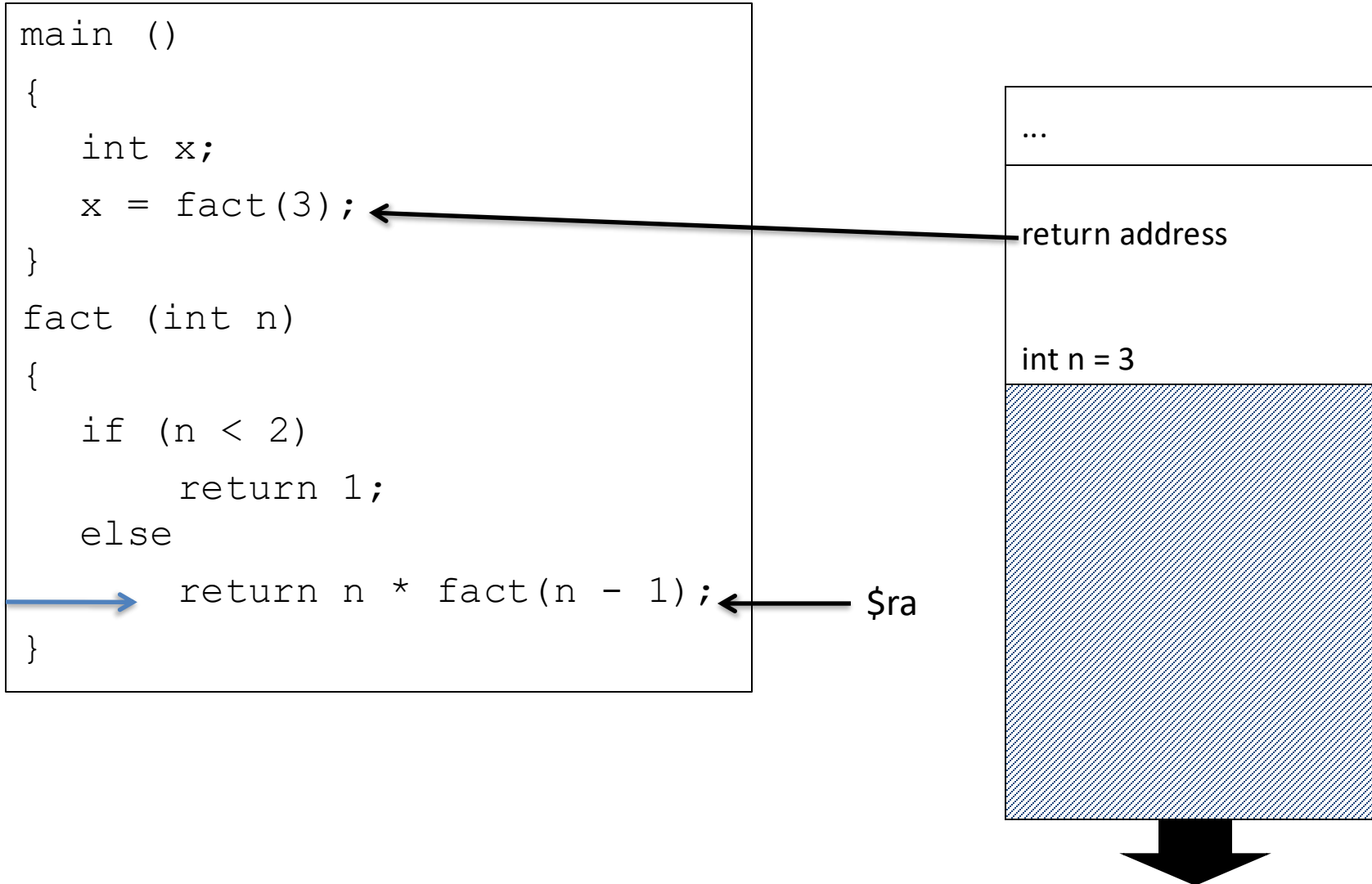
Process Stack



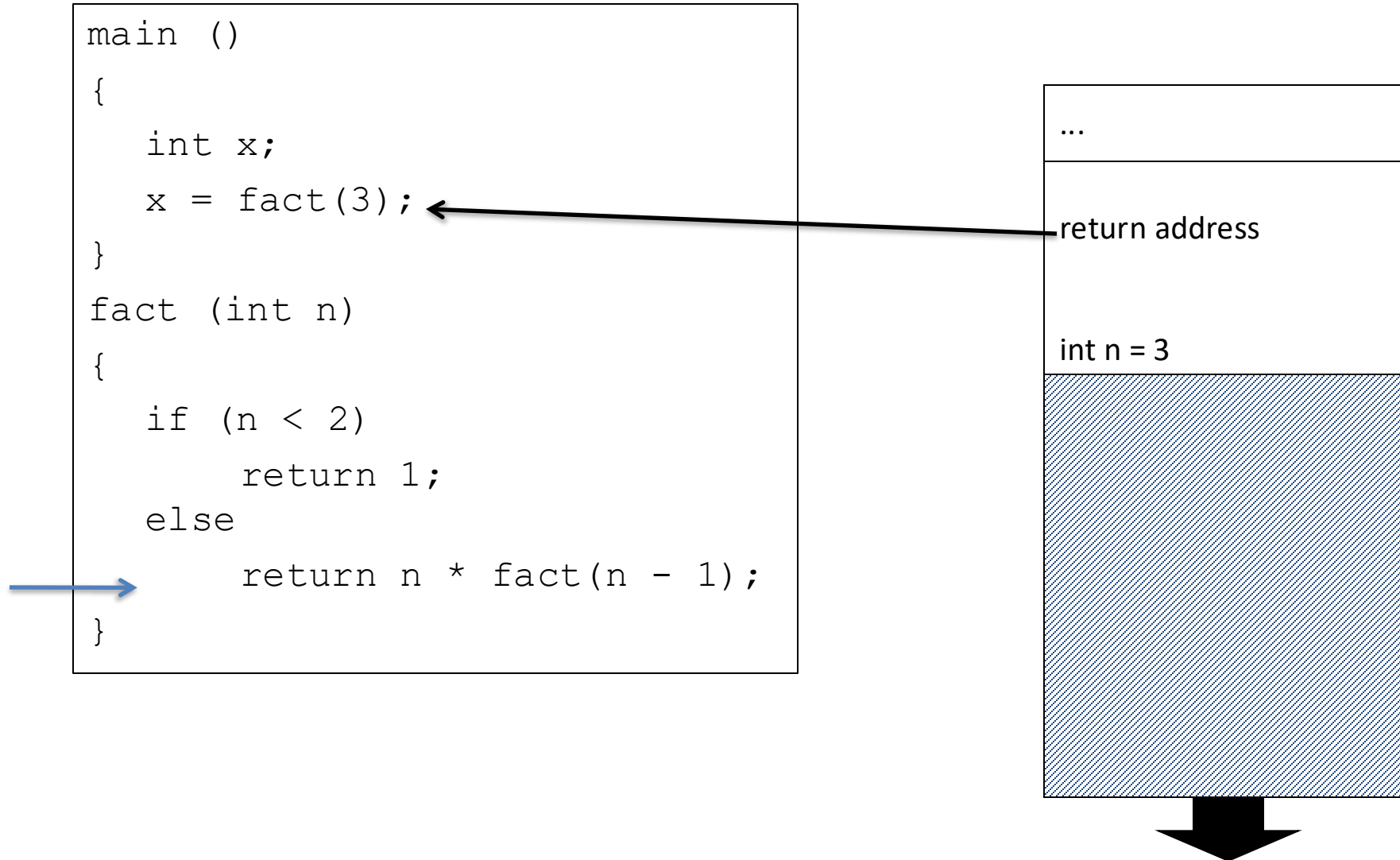
Process Stack



Process Stack



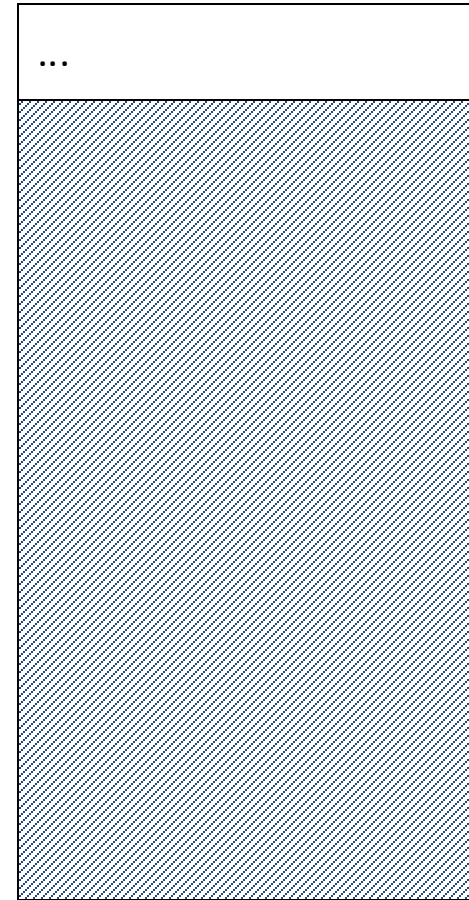
Process Stack



Process Stack

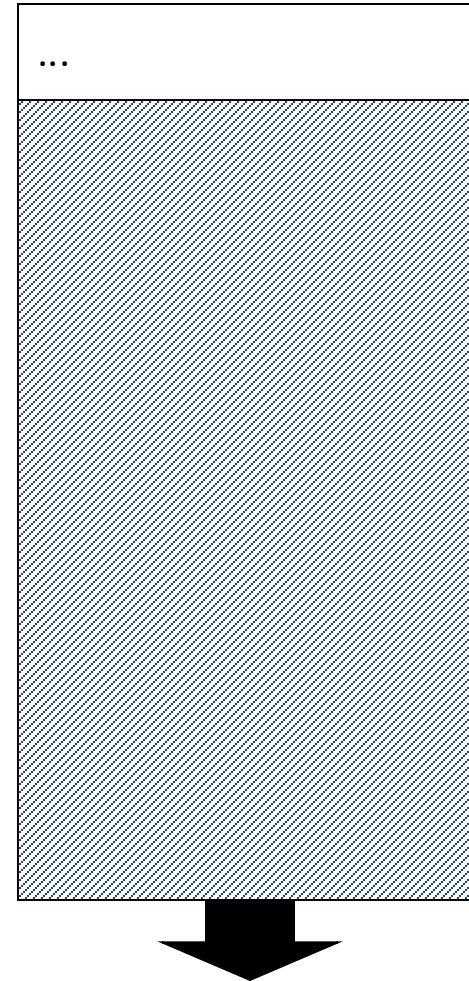
```
main ()  
{  
    int x;  
    x = fact(3);  
}  
fact (int n)  
{  
    if (n < 2)  
        return 1;  
    else  
        return n * fact(n - 1);  
}
```

\$ra



Process Stack

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



Questions?

Non-Leaf Procedure Example

- MIPS code:

fact:

```
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 8($sp)       # save return address
    sw   $a0, 4($sp)       # save argument
    slti $t0, $a0, 2       # test for n < 2
    beq  $t0, $zero, L1    # if so, result is 1
    addi $v0, $zero, 1     #   pop 2 items from stack
    addi $sp, $sp, 8       #   and return
    jr   $ra
L1: addi $a0, $a0, -1       # else decrement n
    jal  fact              # recursive call
    lw   $a0, 4($sp)       # restore original n
    lw   $ra, 8($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra              # and return
```

\$ra = 0x864

\$a0 = 3

\$v0 =

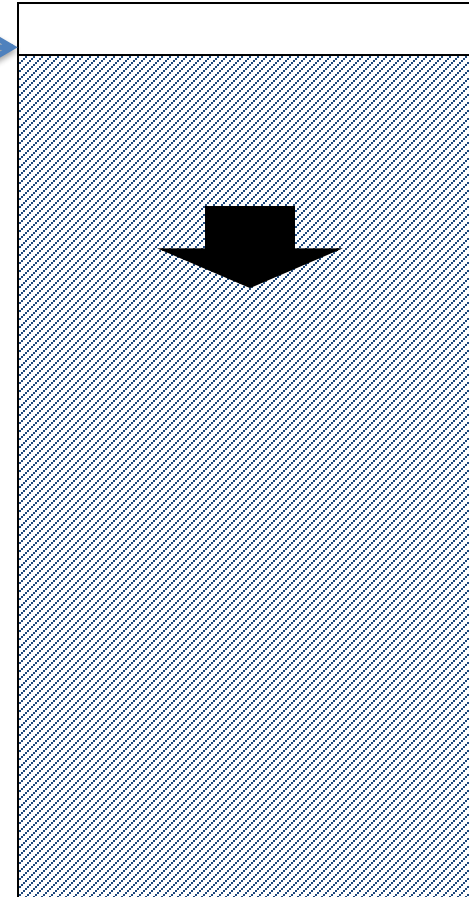
\$t0 =

fact(3)

PC →

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 8($sp)      # save return address
    sw   $a0, 4($sp)      # save argument
    slti $t0, $a0, 2      # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8      # pop 2 items from stack
    jr   $ra              # and return
L1:    addi $a0, $a0, -1    # else decrement n
    jal  fact              # recursive call
    lw   $a0, 4($sp)      # restore original n
    lw   $ra, 8($sp)      # and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra              # and return
```

SP →



\$ra = 0x864

\$a0 = 3

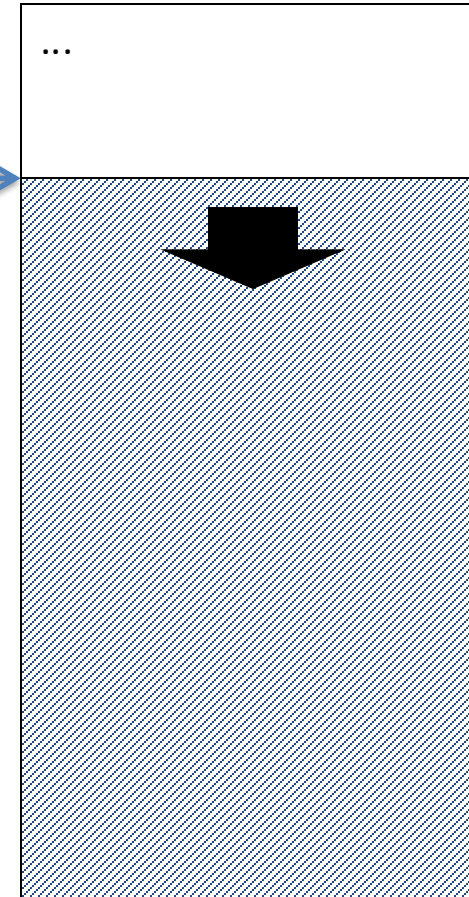
\$v0 =

\$t0 =

fact

```
fact:
PC → addi $sp, $sp, -8    # adjust stack for 2 items
      sw  $ra, 8($sp)     # save return address
      sw  $a0, 4($sp)     # save argument
      slti $t0, $a0, 2    # test for n < 2
      beq $t0, $zero, L1
      addi $v0, $zero, 1  # if so, result is 1
      addi $sp, $sp, 8    # pop 2 items from stack
      jr  $ra             # and return
L1:   addi $a0, $a0, -1    # else decrement n
      jal fact            # recursive call
      lw  $a0, 4($sp)     # restore original n
      lw  $ra, 8($sp)     # and return address
      addi $sp, $sp, 8    # pop 2 items from stack
      mul $v0, $a0, $v0   # multiply to get result
      jr  $ra             # and return
```

SP →

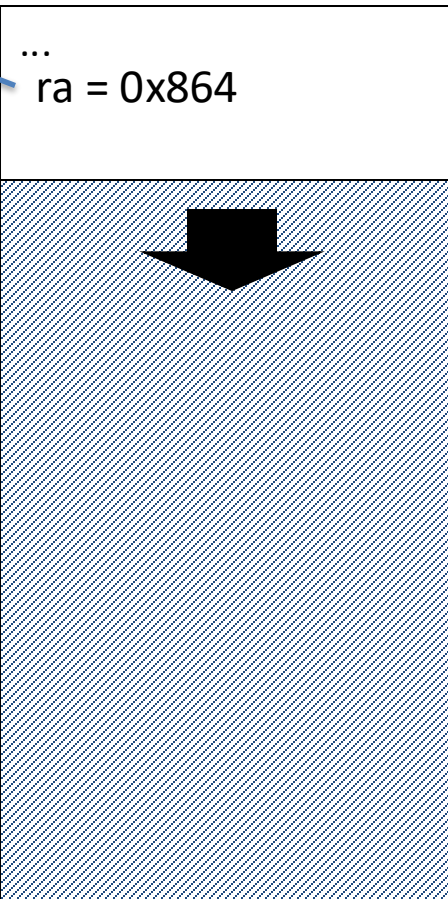


\$ra = 0x864
\$a0 = 3
\$v0 =
\$t0 =

fact

```
fact:
PC → addi $sp, $sp, -8      # adjust stack for 2 items
      sw  $ra, 8($sp)      # save return address
      sw  $a0, 4($sp)      # save argument
      slti $t0, $a0, 2     # test for n < 2
      beq $t0, $zero, L1
      addi $v0, $zero, 1   # if so, result is 1
      addi $sp, $sp, 8     # pop 2 items from stack
      jr  $ra              # and return
L1:   addi $a0, $a0, -1     # else decrement n
      jal fact             # recursive call
      lw  $a0, 4($sp)      # restore original n
      lw  $ra, 8($sp)      # and return address
      addi $sp, $sp, 8     # pop 2 items from stack
      mul $v0, $a0, $v0    # multiply to get result
      jr  $ra              # and return
```

SP →



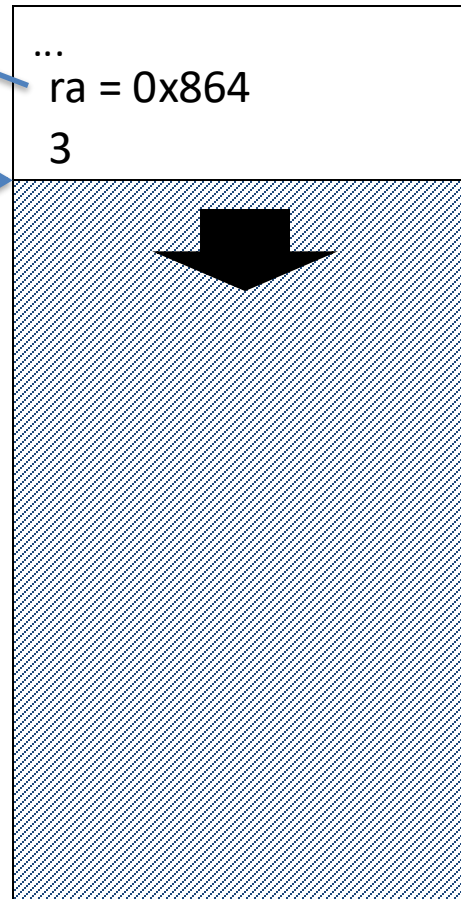
\$ra = 0x864
\$a0 = 3
\$v0 =
\$t0 =

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:    addi $a0, $a0, -1  # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra             # and return
```

PC →

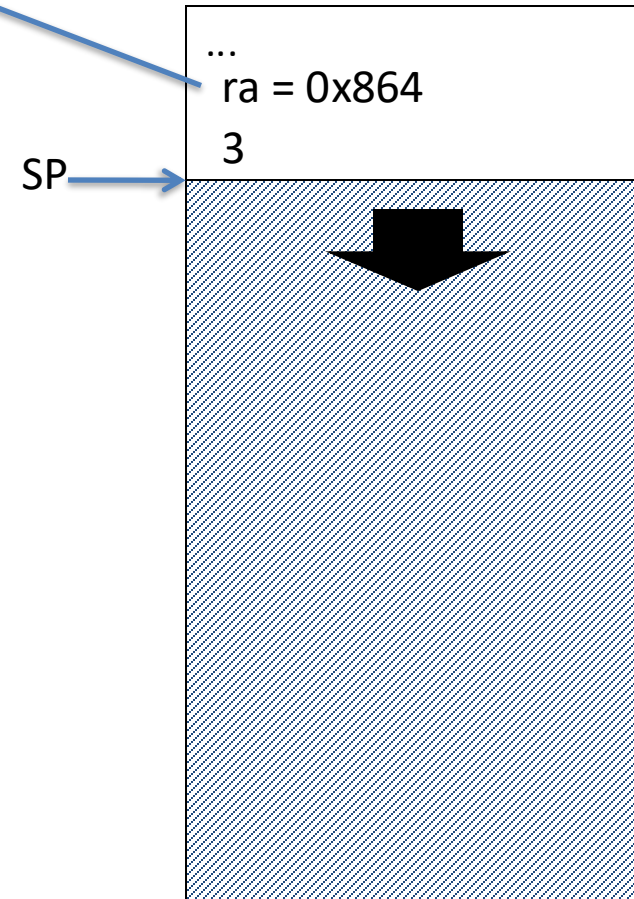
SP →



\$ra = 0x864
\$a0 = 3
\$v0 =
\$t0 = 0

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
PC →    slti $t0, $a0, 2  # test for n < 2
        beq $t0, $zero, L1
        addi $v0, $zero, 1 # if so, result is 1
        addi $sp, $sp, 8   # pop 2 items from stack
        jr   $ra          # and return
L1:     addi $a0, $a0, -1  # else decrement n
        jal  fact         # recursive call
        lw   $a0, 4($sp)  # restore original n
        lw   $ra, 8($sp)  # and return address
        addi $sp, $sp, 8   # pop 2 items from stack
        mul  $v0, $a0, $v0 # multiply to get result
        jr   $ra          # and return
```

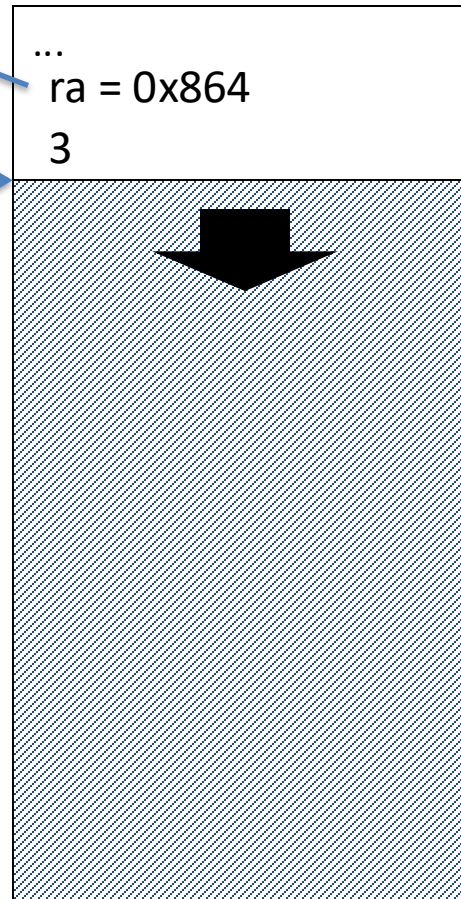


\$ra = 0x864
\$a0 = 3
\$v0 =
\$t0 = 0

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
PC → beq $t0, $zero, L1  # if so, result is 1
    addi $v0, $zero, 1   #   pop 2 items from stack
    addi $sp, $sp, 8     #   and return
    jr   $ra
L1:  addi $a0, $a0, -1    # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     #   and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra            # and return
```

SP →



\$ra = 0x864

\$a0 = 2

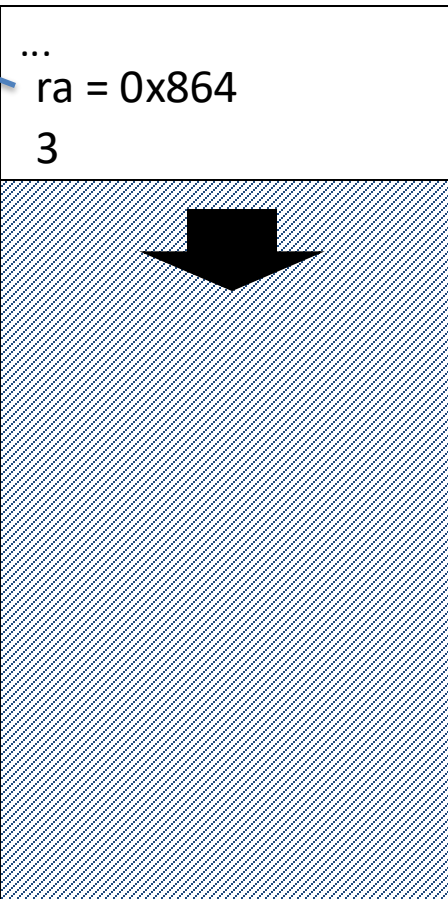
\$v0 =

\$t0 = 0

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)    # save return address
    sw   $a0, 4($sp)    # save argument
    slti $t0, $a0, 2    # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1  # if so, result is 1
    addi $sp, $sp, 8    # pop 2 items from stack
    jr   $ra            # and return
PC → L1 → addi $a0, $a0, -1 # else decrement n
        jal  fact        # recursive call
        lw   $a0, 4($sp)  # restore original n
        lw   $ra, 8($sp)  # and return address
        addi $sp, $sp, 8  # pop 2 items from stack
        mul  $v0, $a0, $v0 # multiply to get result
        jr   $ra          # and return
```

SP →



After this line of code, the next line of code we run will be

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 8($sp)      # save return address
    sw   $a0, 4($sp)      # save argument
    slti $t0, $a0, 2      # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       # pop 2 items from stack
    jr   $ra              # and return
PC → L1: addi $a0, $a0, -1  # else decrement n
        jal  fact          # recursive call
        lw   $a0, 4($sp)   # restore original n
        lw   $ra, 8($sp)   # and return address
        addi $sp, $sp, 8   # pop 2 items from stack
        mul  $v0, $a0, $v0 # multiply to get result
        jr   $ra          # and return
```

\$ra = 0x864

\$a0 = 2

\$v0 =

\$t0 = 0

A. `lw $a0, 4($sp)`

B. `addi $a0, $a0, -1`

C. `addi $sp, $sp, -8`

D. `jr $ra`

E. None of the above

$\$ra = L1 + 8$

$\$a0 = 2$

$\$v0 =$

$\$t0 = 0$

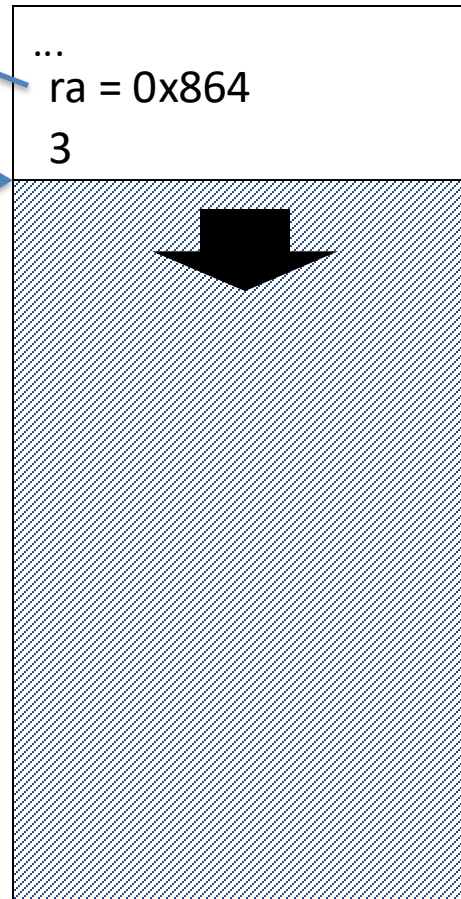
fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:   addi $a0, $a0, -1   # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra            # and return
```

PC



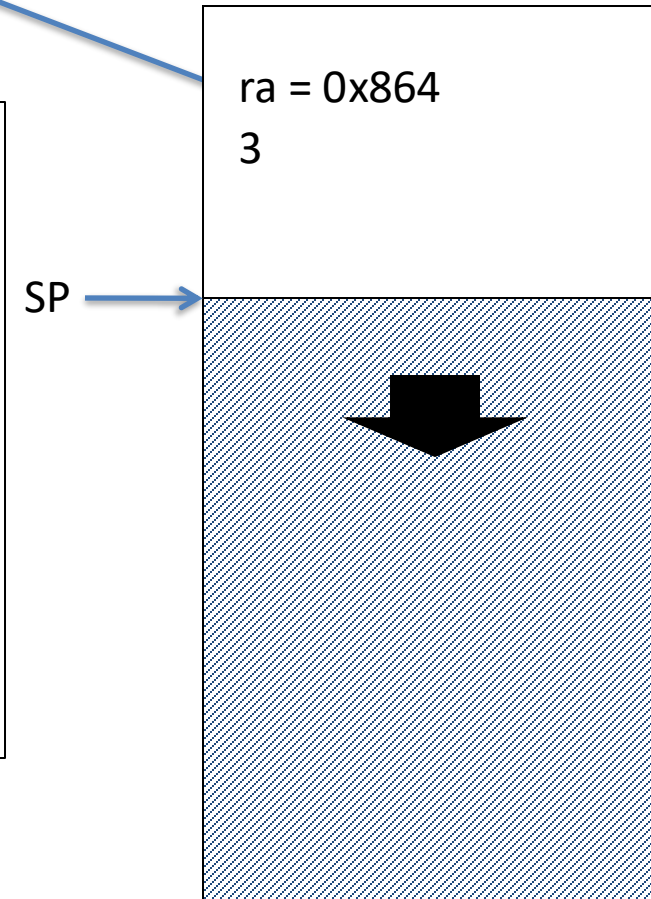
SP



\$ra = L1 + 8
\$a0 = 2
\$v0 =
\$t0 =

fact

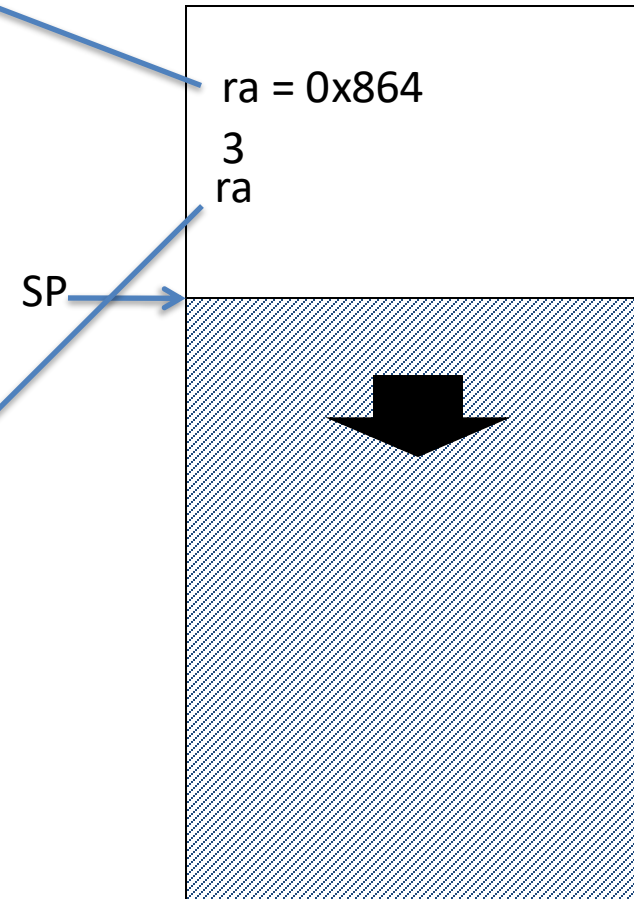
```
fact:
PC → addi $sp, $sp, -8    # adjust stack for 2 items
      sw  $ra, 8($sp)     # save return address
      sw  $a0, 4($sp)     # save argument
      slti $t0, $a0, 2    # test for n < 2
      beq $t0, $zero, L1
      addi $v0, $zero, 1  # if so, result is 1
      addi $sp, $sp, 8    # pop 2 items from stack
      jr  $ra             # and return
L1:   addi $a0, $a0, -1    # else decrement n
      jal fact            # recursive call
      lw  $a0, 4($sp)     # restore original n
      lw  $ra, 8($sp)     # and return address
      addi $sp, $sp, 8    # pop 2 items from stack
      mul $v0, $a0, $v0   # multiply to get result
      jr  $ra             # and return
```



\$ra = L1 + 8
\$a0 = 2
\$v0 =
\$t0 = 0

fact

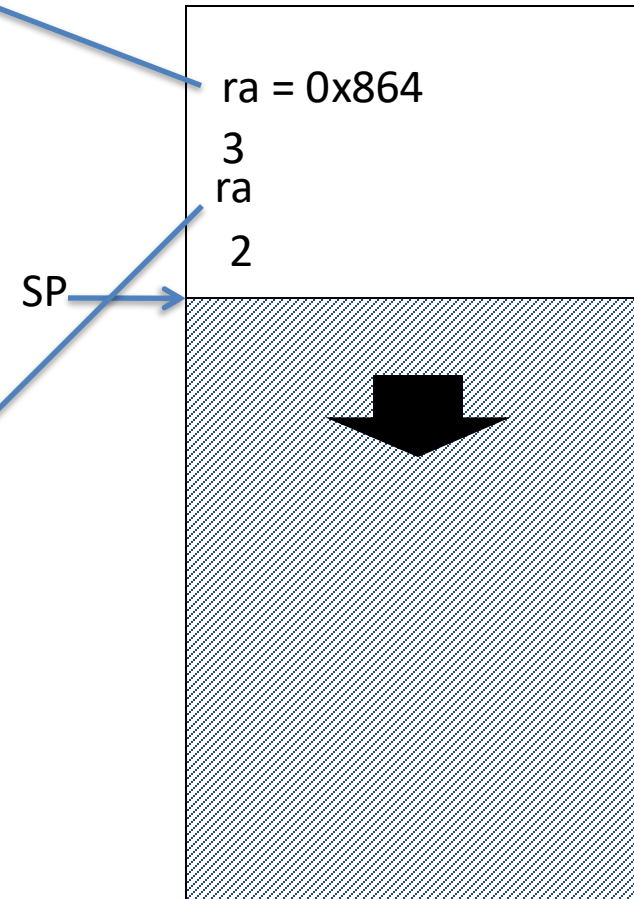
```
fact:
PC → addi $sp, $sp, -8      # adjust stack for 2 items
      sw  $ra, 8($sp)       # save return address
      sw  $a0, 4($sp)       # save argument
      slti $t0, $a0, 2      # test for n < 2
      beq  $t0, $zero, L1
      addi $v0, $zero, 1    # if so, result is 1
      addi $sp, $sp, 8      # pop 2 items from stack
      jr   $ra              # and return
L1:   addi $a0, $a0, -1      # else decrement n
      jal  fact              # recursive call
      lw   $a0, 4($sp)       # restore original n
      lw   $ra, 8($sp)       # and return address
      addi $sp, $sp, 8      # pop 2 items from stack
      mul  $v0, $a0, $v0     # multiply to get result
      jr   $ra              # and return
```



\$ra = L1 + 8
\$a0 = 2
\$v0 =
\$t0 = 0

fact

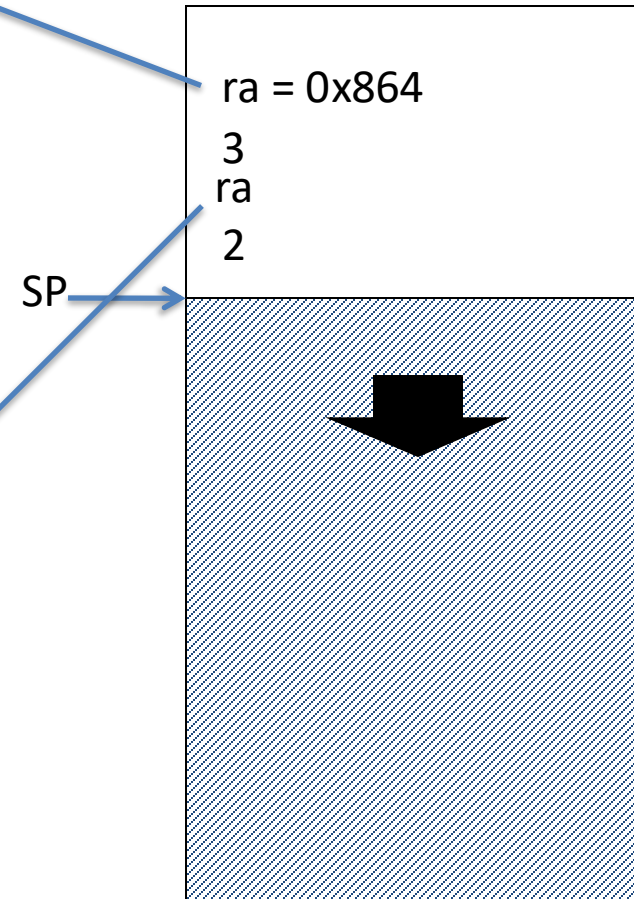
```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 8($sp)      # save return address
PC → sw   $a0, 4($sp)      # save argument
    slti $t0, $a0, 2      # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       # pop 2 items from stack
    jr   $ra              # and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact             # recursive call
    lw   $a0, 4($sp)      # restore original n
    lw   $ra, 8($sp)      # and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra              # and return
```



\$ra = L1 + 8
\$a0 = 2
\$v0 =
\$t0 = 0

fact

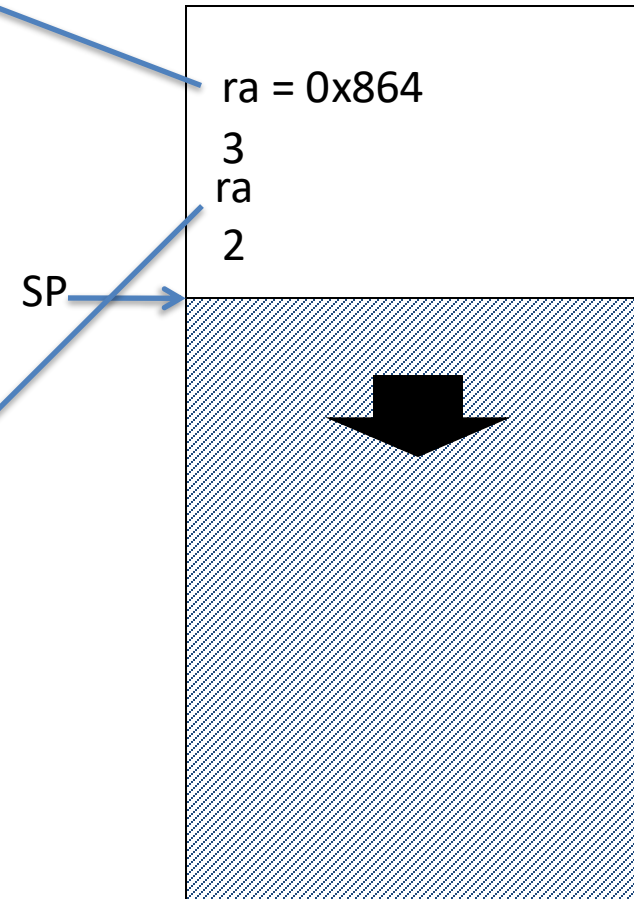
```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
PC →    slti $t0, $a0, 2  # test for n < 2
        beq $t0, $zero, L1
        addi $v0, $zero, 1 # if so, result is 1
        addi $sp, $sp, 8   # pop 2 items from stack
        jr   $ra           # and return
L1:     addi $a0, $a0, -1  # else decrement n
        jal  fact         # recursive call
        lw   $a0, 4($sp)  # restore original n
        lw   $ra, 8($sp)  # and return address
        addi $sp, $sp, 8   # pop 2 items from stack
        mul  $v0, $a0, $v0 # multiply to get result
        jr   $ra           # and return
```



\$ra = L1 + 8
\$a0 = 2
\$v0 =
\$t0 = 0

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
PC → beq  $t0, $zero, L1  # if so, result is 1
    addi $v0, $zero, 1   #   pop 2 items from stack
    addi $sp, $sp, 8     #   and return
    jr   $ra
L1:  addi $a0, $a0, -1    # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     #   and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra            # and return
```



\$ra = L1 + 8

\$a0 = 1

\$v0 =

\$t0 = 0

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)    # save return address
    sw   $a0, 4($sp)    # save argument
    slti $t0, $a0, 2    # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1  # if so, result is 1
    addi $sp, $sp, 8    # pop 2 items from stack
    jr   $ra            # and return
PC → L1 → addi $a0, $a0, -1 # else decrement n
        jal  fact        # recursive call
        lw   $a0, 4($sp) # restore original n
        lw   $ra, 8($sp) # and return address
        addi $sp, $sp, 8 # pop 2 items from stack
        mul  $v0, $a0, $v0 # multiply to get result
        jr   $ra        # and return
```

SP

ra = 0x864

3
ra
2



\$ra = L1 + 8
\$a0 = 1
\$v0 =
\$t0 = 0

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:   addi $a0, $a0, -1   # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra             # and return
```

PC →

SP →

ra = 0x864

3
ra
2



\$ra = L1 + 8
\$a0 = 1
\$v0 =
\$t0 = 0

fact

```
fact:
PC → addi $sp, $sp, -8    # adjust stack for 2 items
      sw  $ra, 8($sp)     # save return address
      sw  $a0, 4($sp)     # save argument
      slti $t0, $a0, 2    # test for n < 2
      beq  $t0, $zero, L1
      addi $v0, $zero, 1  # if so, result is 1
      addi $sp, $sp, 8    # pop 2 items from stack
      jr   $ra            # and return
L1:   addi $a0, $a0, -1    # else decrement n
      jal  fact           # recursive call
      lw   $a0, 4($sp)    # restore original n
      lw   $ra, 8($sp)    # and return address
      addi $sp, $sp, 8    # pop 2 items from stack
      mul  $v0, $a0, $v0  # multiply to get result
      jr   $ra            # and return
```

SP

ra = 0x864

3
ra
2

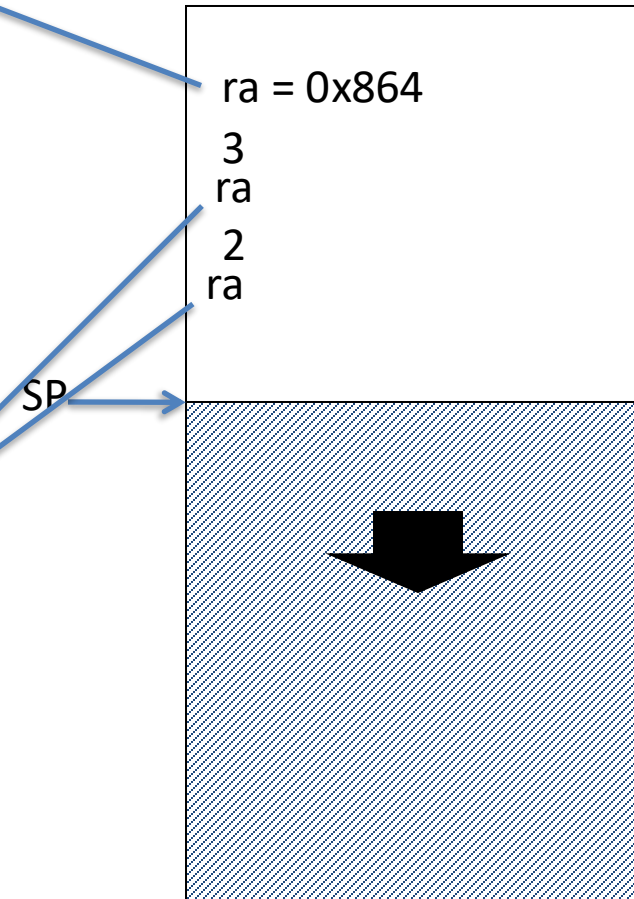


\$ra = L1 + 8
\$a0 = 1
\$v0 =
\$t0 = 0

fact

PC →

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 8($sp)      # save return address
    sw   $a0, 4($sp)      # save argument
    slti $t0, $a0, 2      # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       # pop 2 items from stack
    jr   $ra              # and return
L1:    addi $a0, $a0, -1    # else decrement n
    jal  fact             # recursive call
    lw   $a0, 4($sp)      # restore original n
    lw   $ra, 8($sp)      # and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra              # and return
```

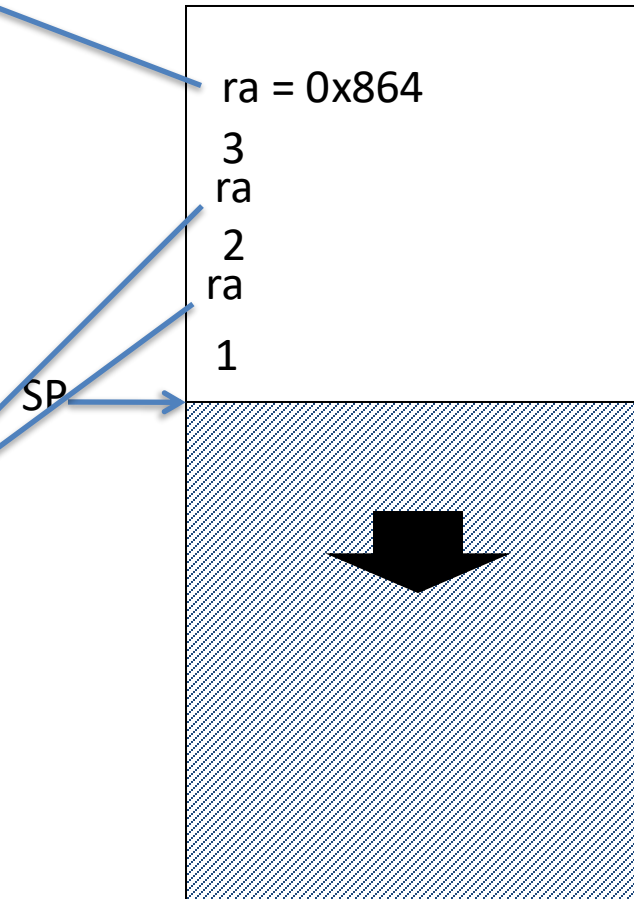


\$ra = L1 + 8
\$a0 = 1
\$v0 =
\$t0 = 0

fact

PC →

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 8($sp)       # save return address
    sw   $a0, 4($sp)       # save argument
    slti $t0, $a0, 2       # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       # pop 2 items from stack
    jr   $ra               # and return
L1:    addi $a0, $a0, -1    # else decrement n
    jal  fact              # recursive call
    lw   $a0, 4($sp)       # restore original n
    lw   $ra, 8($sp)       # and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

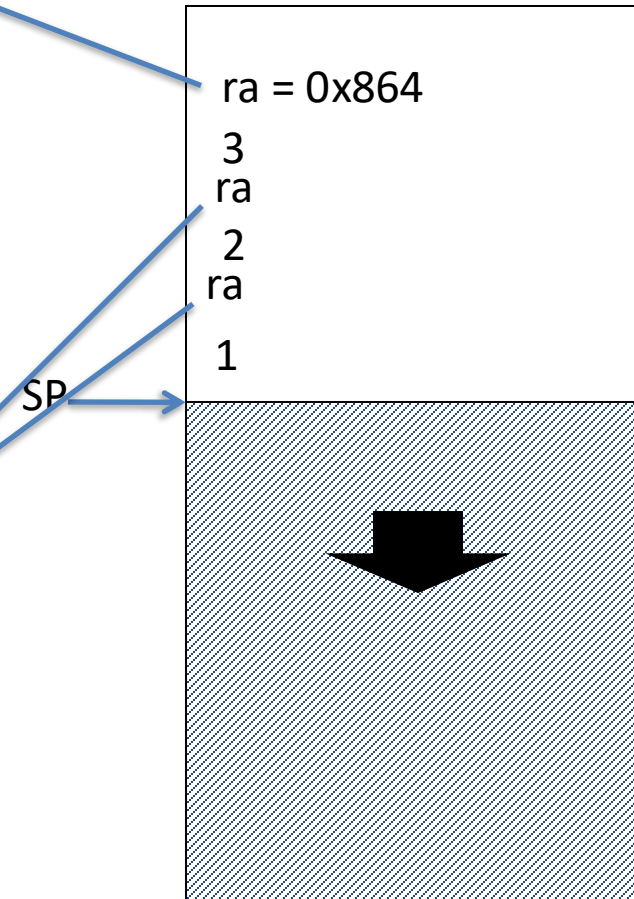


\$ra = L1 + 8
\$a0 = 1
\$v0 =
\$t0 = 1

fact

PC →

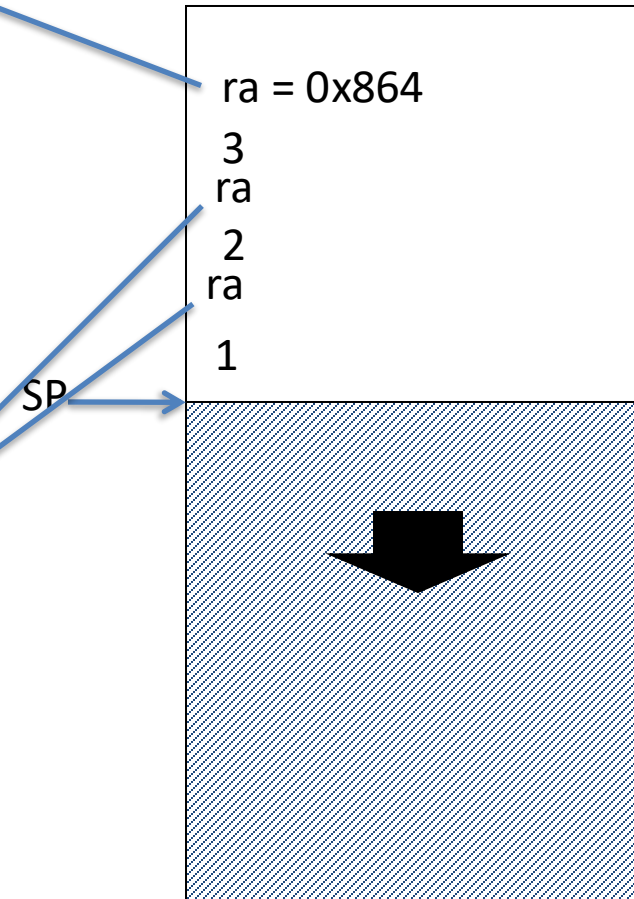
```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      # pop 2 items from stack
    jr   $ra             # and return
L1:   addi $a0, $a0, -1   # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra             # and return
```



\$ra = L1 + 8
\$a0 = 1
\$v0 =
\$t0 = 1

fact

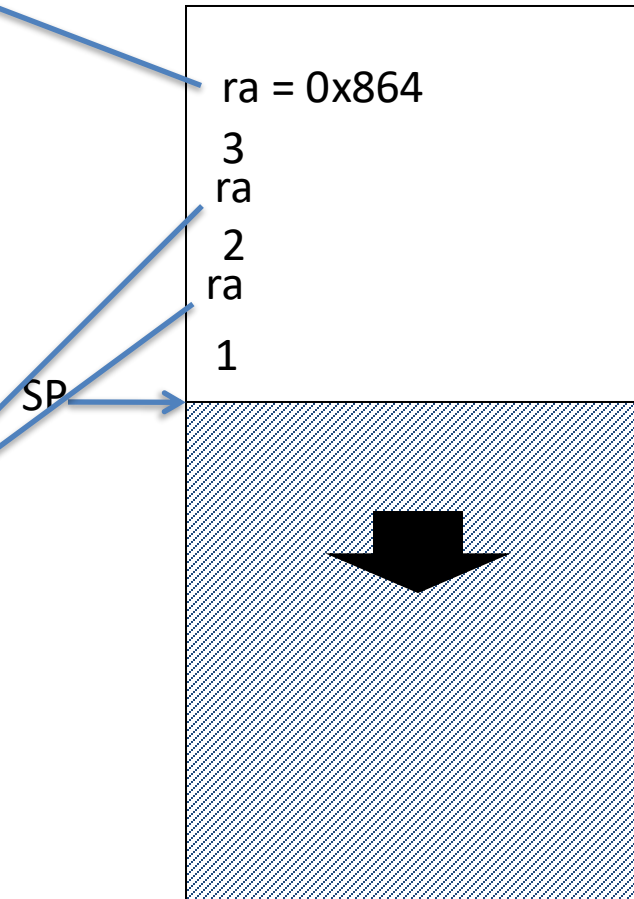
```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
PC → beq  $t0, $zero, L1  # if so, result is 1
    addi $v0, $zero, 1   #   pop 2 items from stack
    addi $sp, $sp, 8     #   and return
    jr   $ra
L1:  addi $a0, $a0, -1    # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     #   and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra            # and return
```



\$ra = L1 + 8
\$a0 = 1
\$v0 = 1
\$t0 = 1

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    PC → addi $v0, $zero, 1 # if so, result is 1
        addi $sp, $sp, 8  # pop 2 items from stack
        jr   $ra          # and return
    L1: addi $a0, $a0, -1 # else decrement n
        jal  fact         # recursive call
        lw   $a0, 4($sp)  # restore original n
        lw   $ra, 8($sp)  # and return address
        addi $sp, $sp, 8  # pop 2 items from stack
        mul  $v0, $a0, $v0 # multiply to get result
        jr   $ra          # and return
```



\$ra = L1 + 8
\$a0 = 1
\$v0 = 1
\$t0 = 1

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2    # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1   # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:    addi $a0, $a0, -1  # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra             # and return
```

PC →

SP →

ra = 0x864

3

ra

2

ra

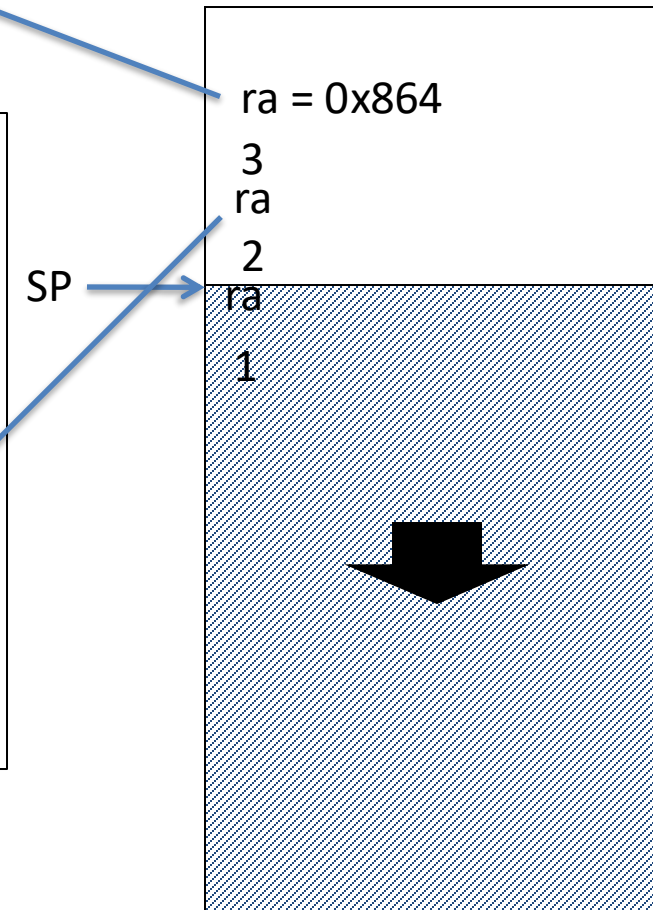
1



\$ra = L1 + 8
\$a0 = 1
\$v0 = 1
\$t0 = 1

fact

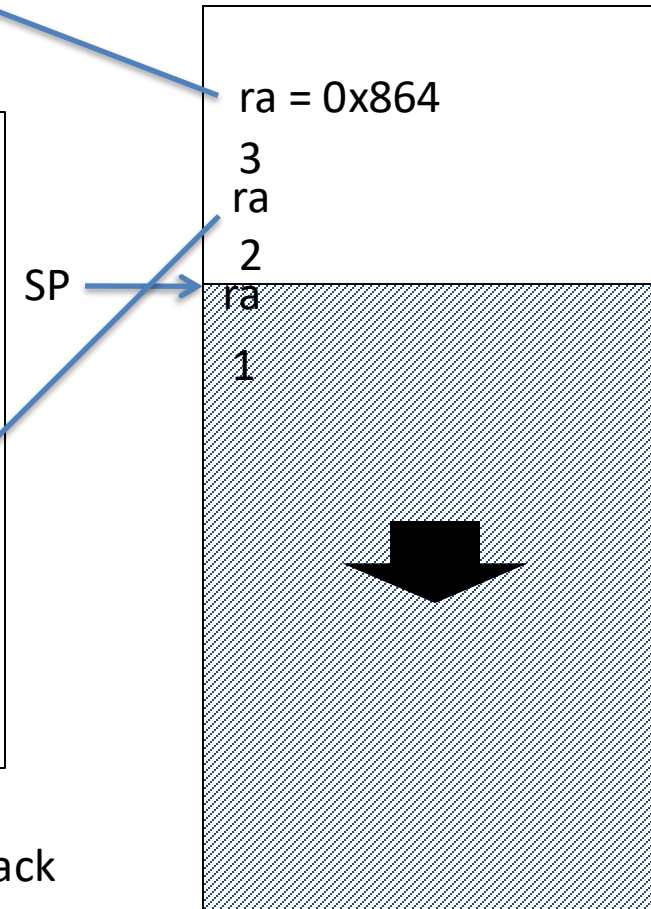
```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 8($sp)       # save return address
    sw   $a0, 4($sp)       # save argument
    slti $t0, $a0, 2       # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       # pop 2 items from stack
    jr   $ra               # and return
PC → L1: addi $a0, $a0, -1  # else decrement n
        jal  fact          # recursive call
        lw   $a0, 4($sp)   # restore original n
        lw   $ra, 8($sp)   # and return address
        addi $sp, $sp, 8   # pop 2 items from stack
        mul  $v0, $a0, $v0 # multiply to get result
        jr   $ra           # and return
```



$\$ra = L1 + 8$
 $\$a0 = 1$
 $\$v0 = 1$
 $\$t0 = 1$

We will return to

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1   # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
PC → L1: addi $a0, $a0, -1 # else decrement n
        jal  fact        # recursive call
        lw   $a0, 4($sp)  # restore original n
        lw   $ra, 8($sp)  # and return address
        addi $sp, $sp, 8  # pop 2 items from stack
        mul  $v0, $a0, $v0 # multiply to get result
        jr   $ra         # and return
```



- A. $L1 + 8$, because it is in $\$ra$
- B. $L1 + 8$, because it's the most recent value on the stack
- C. $0x864$, because it's the top value on the stack
- D. `fact`, because it's the procedure call
- E. None of the above

\$ra = L1 + 8

\$a0 = 2

\$v0 = 1

\$t0 = 1

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw    $ra, 8($sp)    # save return address
    sw    $a0, 4($sp)    # save argument
    slti  $t0, $a0, 2    # test for n < 2
    beq   $t0, $zero, L1
    addi  $v0, $zero, 1   # if so, result is 1
    addi  $sp, $sp, 8     # pop 2 items from stack
    jr    $ra            # and return
L1:     addi $a0, $a0, -1  # else decrement n
    jal   fact           # recursive call
    lw    $a0, 4($sp)    # restore original n
    lw    $ra, 8($sp)    # and return address
    addi  $sp, $sp, 8     # pop 2 items from stack
    mul   $v0, $a0, $v0  # multiply to get result
    jr    $ra            # and return
```

PC

SP

ra = 0x864

3

ra

2

ra

1



$\$ra = L1 + 8$

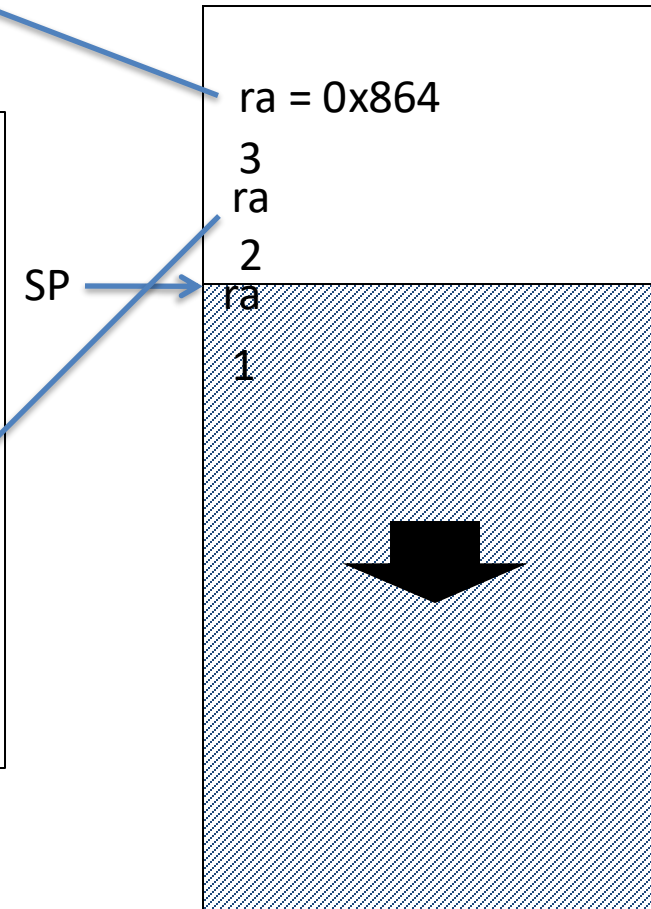
$\$a0 = 2$

$\$v0 = 1$

$\$t0 = 1$

fact

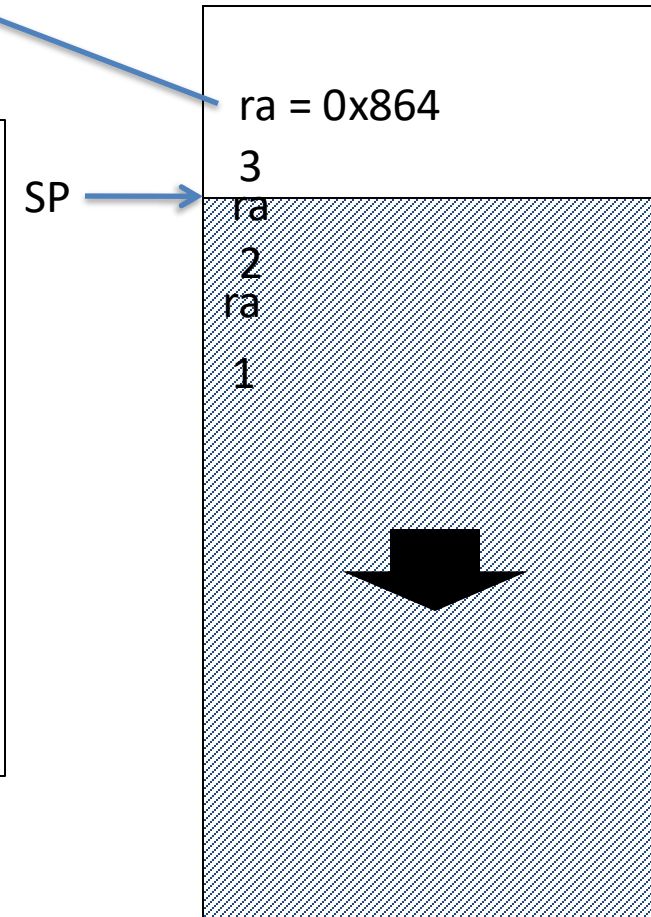
```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:    addi $a0, $a0, -1  # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
PC →   lw   $ra, 8($sp)  # and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra            # and return
```



\$ra = L1 + 8
\$a0 = 2
\$v0 = 1
\$t0 = 1

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:    addi $a0, $a0, -1  # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
PC →   addi $sp, $sp, 8  # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra            # and return
```



\$ra = L1 + 8
\$a0 = 2
\$v0 = 2
\$t0 = 1

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1   # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:    addi $a0, $a0, -1  # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra             # and return
```

PC →

SP →

ra = 0x864

3

ra

2

ra

1



\$ra = L1 + 8
\$a0 = 2
\$v0 = 2
\$t0 = 1

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1   # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:    addi $a0, $a0, -1  # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra             # and return
```

PC →

SP →

ra = 0x864

3

ra

2

ra

1



\$ra = L1 + 8

\$a0 = 3

\$v0 = 2

\$t0 = 1

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)    # save return address
    sw   $a0, 4($sp)    # save argument
    slti $t0, $a0, 2    # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1  # if so, result is 1
    addi $sp, $sp, 8    # pop 2 items from stack
    jr   $ra            # and return
L1:   addi $a0, $a0, -1  # else decrement n
    jal  fact           # recursive call
    lw   $a0, 4($sp)    # restore original n
    lw   $ra, 8($sp)    # and return address
    addi $sp, $sp, 8    # pop 2 items from stack
    mul  $v0, $a0, $v0  # multiply to get result
    jr   $ra            # and return
```

PC

SP

ra = 0x864

3

ra

2

ra

1



\$ra = 0x864

\$a0 = 3

\$v0 = 2

\$t0 = 1

fact

fact:

```
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw    $ra, 8($sp)    # save return address
    sw    $a0, 4($sp)    # save argument
    slti  $t0, $a0, 2    # test for n < 2
    beq   $t0, $zero, L1
    addi  $v0, $zero, 1  # if so, result is 1
    addi  $sp, $sp, 8    # pop 2 items from stack
    jr    $ra            # and return
L1:  addi  $a0, $a0, -1   # else decrement n
     jal   fact          # recursive call
     lw    $a0, 4($sp)   # restore original n
PC →   lw    $ra, 8($sp) # and return address
     addi  $sp, $sp, 8    # pop 2 items from stack
     mul   $v0, $a0, $v0 # multiply to get result
     jr    $ra           # and return
```

SP →

ra = 0x864

3

ra

2

ra

1

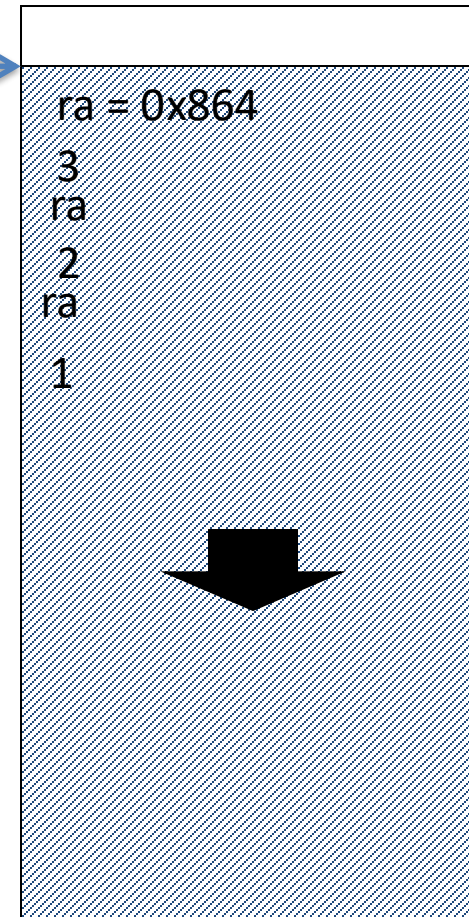


\$ra = 0x864
\$a0 = 3
\$v0 = 2
\$t0 = 1

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:    addi $a0, $a0, -1  # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
PC →   addi $sp, $sp, 8  # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra            # and return
```

SP →



\$ra = 0x864

\$a0 = 3

\$v0 = 6

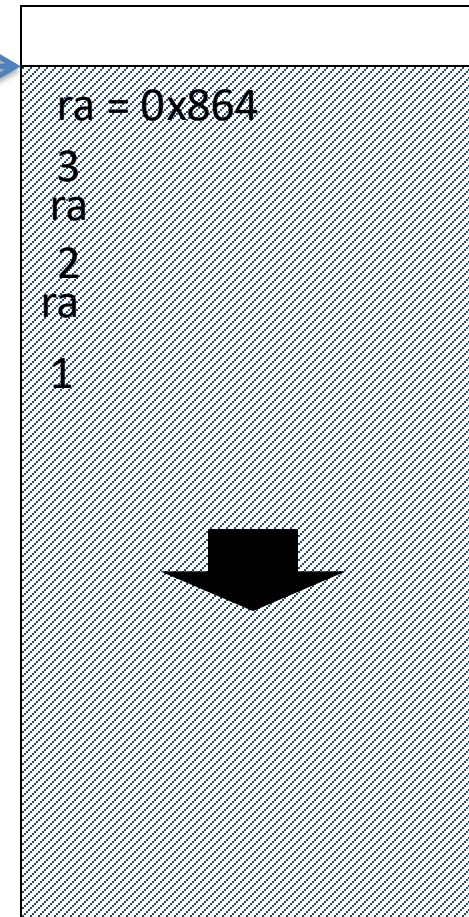
\$t0 = 1

fact

fact:

```
        addi $sp, $sp, -8      # adjust stack for 2 items
        sw   $ra, 8($sp)      # save return address
        sw   $a0, 4($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1     # if so, result is 1
        addi $sp, $sp, 8       # pop 2 items from stack
        jr   $ra              # and return
L1:      addi $a0, $a0, -1      # else decrement n
        jal  fact             # recursive call
        lw   $a0, 4($sp)      # restore original n
        lw   $ra, 8($sp)      # and return address
        addi $sp, $sp, 8       # pop 2 items from stack
PC →     mul  $v0, $a0, $v0    # multiply to get result
        jr   $ra              # and return
```

SP →



\$ra = 0x864

\$a0 = 3

\$v0 = 6

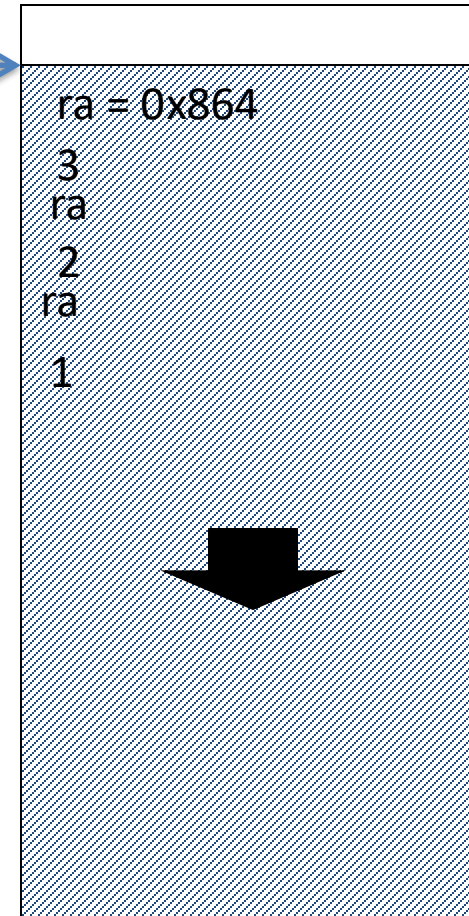
\$t0 = 1

fact

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 8($sp)     # save return address
    sw   $a0, 4($sp)     # save argument
    slti $t0, $a0, 2     # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1   # if so, result is 1
    addi $sp, $sp, 8     # pop 2 items from stack
    jr   $ra             # and return
L1:    addi $a0, $a0, -1  # else decrement n
    jal  fact            # recursive call
    lw   $a0, 4($sp)     # restore original n
    lw   $ra, 8($sp)     # and return address
    addi $sp, $sp, 8     # pop 2 items from stack
    mul  $v0, $a0, $v0   # multiply to get result
    jr   $ra             # and return
```

PC

SP



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
 - `.ascii` “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:

```

D. More than one of the above

E. None of the above

```

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

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

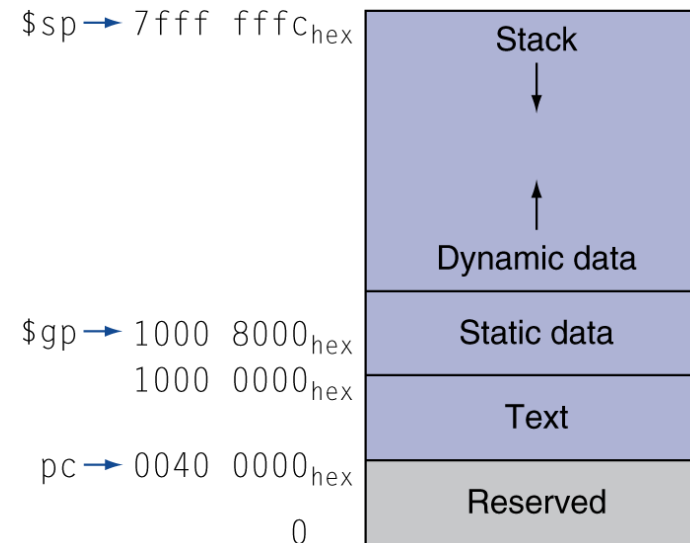
- How to use:
 - Put syscall number into register \$v0
 - Load arguments into argument registers
 - Issue syscall instruction
 - Retrieve return values

- 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
```

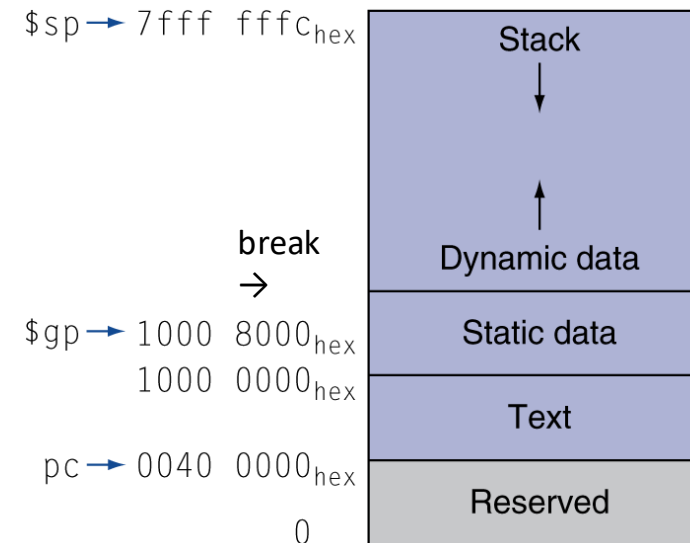
sbrk allocates memory from which region?

- 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



Reading

- Next lecture: More Stack
- Problem set 3 due Today
- Lab 2 due Sunday