

ECE260: Fundamentals of Computer Engineering

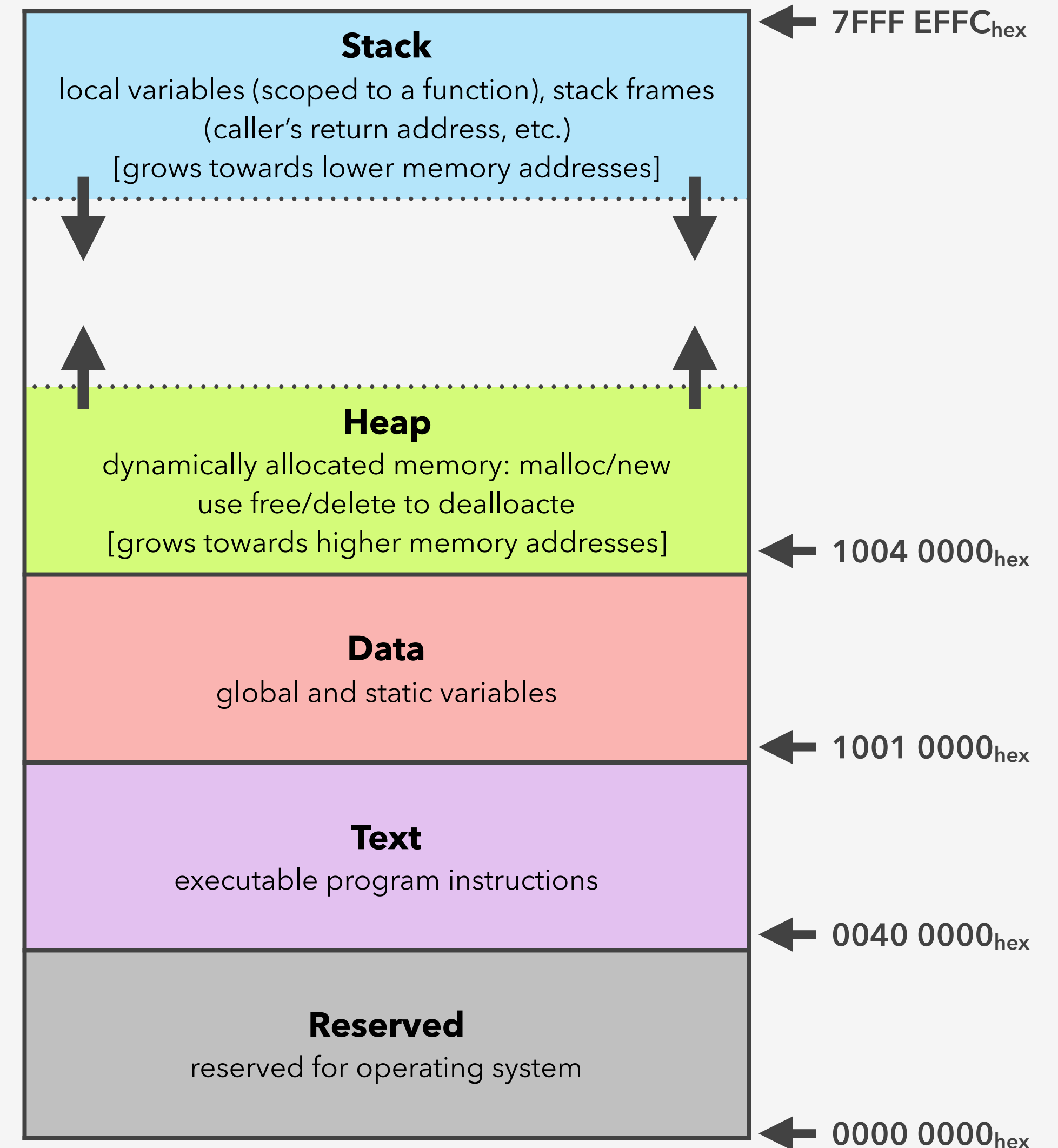
Supporting Nested Procedures

James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania



Memory Layout [in MARS Simulator]

- **Reserved** – used by operating system, not for you
- **Text** – executable program instructions (i.e. your code)
 - Create with a `.text` directive
- **Data** – global and static variables
 - Create with a `.data` directive
- **Heap** – dynamically allocated memory
 - Request this from the operating system
- **Stack** – for storing local variables and stack frames
 - Grows towards the heap



MIPS Registers (now with more info!)

- MIPS architecture has a 32 × 32-bit register file (e.g. it has 32 32-bit registers)

Register Number	Register Name	Use
0	\$zero	Constant value 0
1	\$at	Assembler temporary
2 - 3	\$v0 - \$v1	Procedure return values
4 - 7	\$a0 - \$a3	Procedure arguments
8 - 15	\$t0 - \$t7	Temporary values
16 - 23	\$s0 - \$s7	Saved temporary values
24 - 25	\$t8 - \$t9	More temporary values
26 - 27	\$k0 - \$k1	Reserved for OS
28	\$gp	Global pointer
29	\$sp	Stack pointer
30	\$fp	Frame pointer
31	\$ra	Return Address

Not for you!

Callee puts return value here

Caller puts arguments here

Callee may overwrite these

Must be saved by callee if used

Callee may overwrite these

Also, not for you!

Easy access to constants/globals

Top of stack

Points to local variables on stack

**Where to go when returning
from procedure**

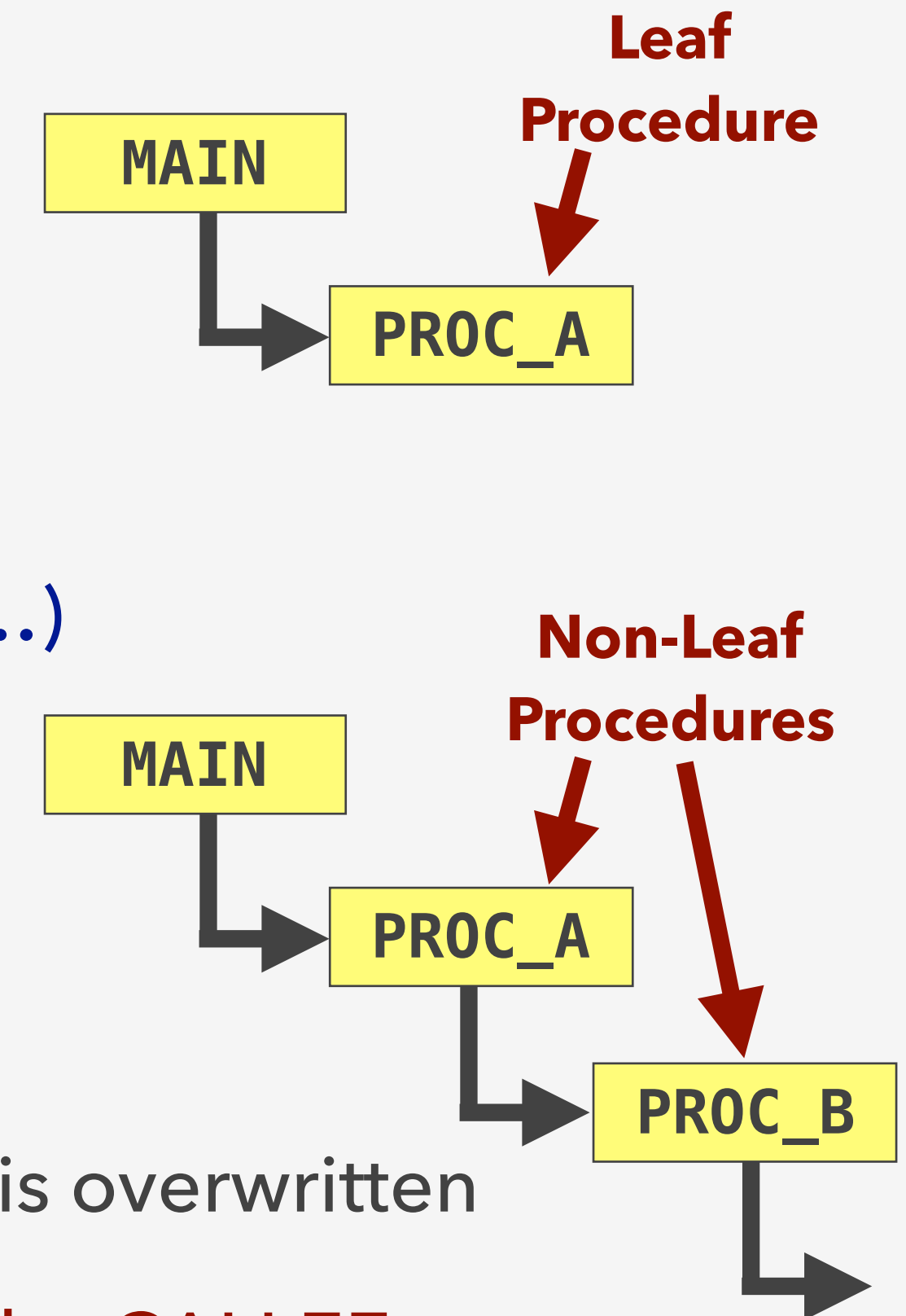
What is Preserved Across a Procedure Call?

- MIPS conventions dictate that a CALLER can expect the following behavior when calling a procedure
 - Some registers and data are expected to be “preserved”
 - Other registers and data are NOT expected to be “preserved”
- Registers and data that are “preserved” are expected by the CALLER to contain the same values both before and after a procedure call (i.e. they should be saved/restored by a CALLEE)

Preserved	Not Preserved
Saved registers: \$s0 - \$s7	Temporary registers: \$t0 - \$t9
Stack/Frame pointer registers: \$sp, \$fp	Argument registers: \$a0 - \$a3
Return address register: \$ra	Return value registers: \$v0 - \$v1
Stack above stack pointer	Stack below stack pointer

Non-Leaf Procedures

- A **leaf procedure** is a procedure that does NOT call another procedure
 - CALLEE (PROC_A) must save/restore and \$sX registers that it uses
- A **non-leaf procedure** is a procedure that calls other procedures
 - All CALLEEs must save/restore \$sX registers that they use (PROC_A, PROC_B, ...)
 - Each CALLER needs to save/restore info on the stack prior to transferring control to next CALLEE
 - **CALLER must save its return address**
 - When the CALLER issues the **jal** instruction to call CALLEE the \$ra register is overwritten
 - **Any arguments (\$aX registers) and temporaries \$(tX registers) needed after the CALLEE returns**
 - CALLER may reassign \$aX registers to pass arguments to CALLEE, so save contents
 - CALLEE may become a CALLER itself and overwrite \$aX registers!
 - No guarantee that a CALLEE will preserve values in \$tX registers, so save contents



Non-Leaf Procedure – A Recursive Example

- Example C code
 - This function is **both** a CALLER and a CALLEE

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

← **BASE Case – stops recursion**

← **RECURSIVE Case – function calls itself**

- Assume the following:
 - Parameter n is passed in register \$a0
 - Factorial function (fact) is called using a **jal** instruction that sets the \$ra register
 - Return value is placed in register \$v0 before the fact function returns

Non-Leaf Procedure – A Recursive Example (continued)

FACT:

```
addi $sp, $sp, -8      # adjust $sp to make room on stack to save contents of 2 registers
sw    $ra, 4($sp)      # As CALLER: save return address onto the stack ... may need restoring later
sw    $a0, 0($sp)      # As CALLER: save argument onto the stack since $a0 may be used to call fact
```

```
slti $t0, $a0, 1      # test for  $n < 1$  ... $t0 is set to 1 if true, 0 otherwise
beq  $t0, $zero, ELSE # branch to ELSE if  $!(n < 1)$ 
```

BASE Case

```
addi $v0, $zero, 1    # otherwise, set return value to 1
addi $sp, $sp, 8      # pop stack, no need to restore $a0 or $ra since never overwritten
jr    $ra             # return from base case with a return value of 1
```

ELSE:

RECURSIVE Case

```
addi $a0, $a0, -1     # decrement n into $a0 to set argument for recursive call
jal  FACT             # recursive call, writes $ra – good thing we backed it up on the stack!
### ##### recursing, will eventually return here with a return value in $v0
lw    $a0, 0($sp)     # As CALLER: restore original value for n when recursion returns
lw    $ra, 4($sp)     # As CALLER: restore original value for $ra when recursion returns
addi $sp, $sp, 8      # pop stack now that values have been restored to registers

mul   $v0, $a0, $v0    # multiply n by result of last recursive call –  $n * \text{fact}(n - 1)$ 
jr    $ra             # previous mul instruction put result in $v0, now return it
```

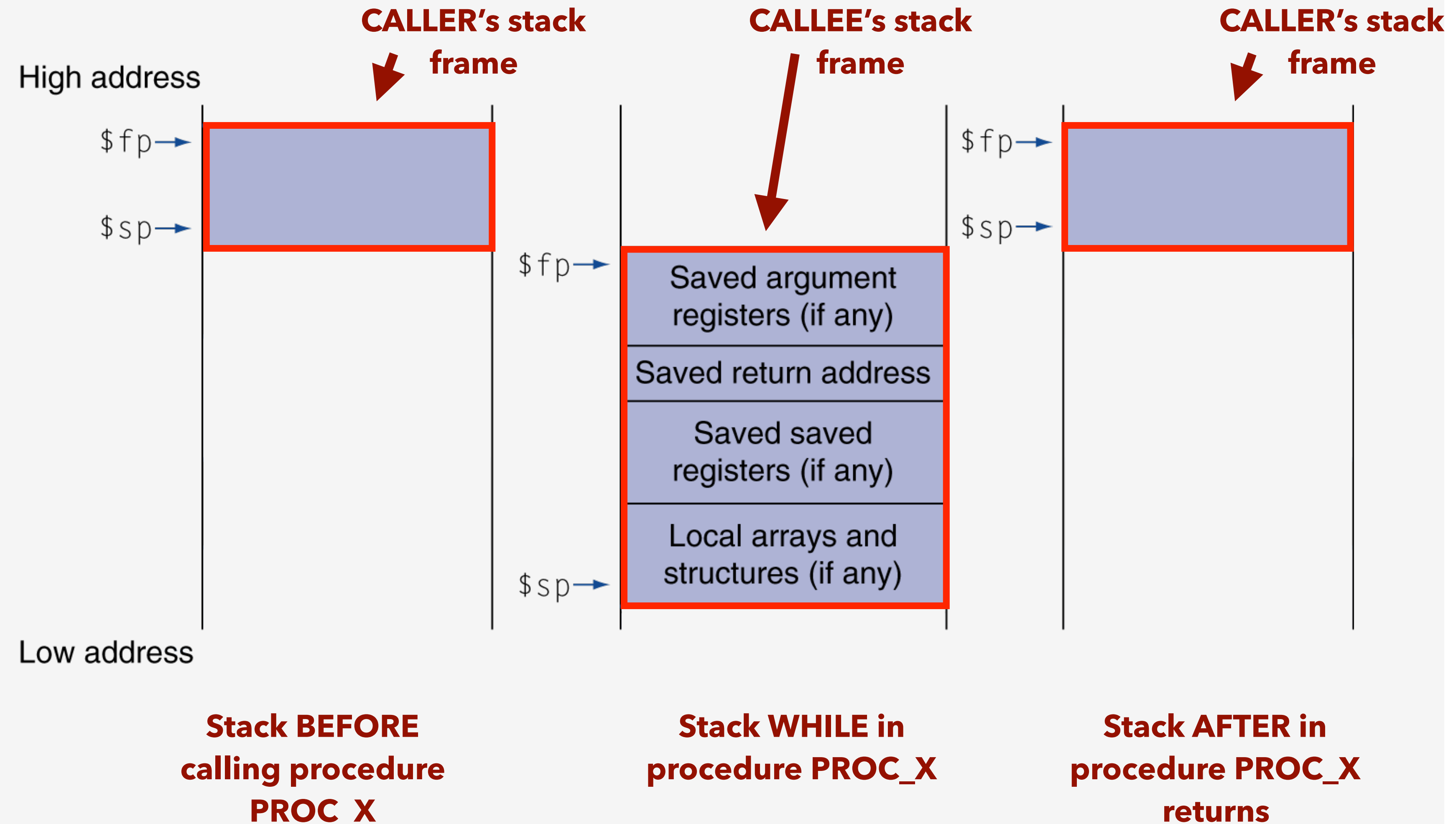
Storing Local Data on the Stack

- A procedure may need memory space for local variables
 - May have locally defined array or simply too much data to store in registers
- If necessary, memory space for local variables is allocated on the stack
 - If not much local data, registers may be sufficient
 - **NOT** stored in .data segment of memory
 - Data is conserved on stack if procedure calls another procedure
 - Data is removed from stack when procedure returns
- **Procedure frame** (a.k.a. **activation record**, a.k.a. **stack frame**)
 - Segment of the stack containing a procedures saved registers and local variables

Storing Local Data on the Stack

Frame pointer (\$fp) points to first word of the frame and doesn't move during a procedure

Stack pointer (\$sp) points to the top of the stack and may move as stack grows/shrinks in a procedure



Passing More than Four Arguments

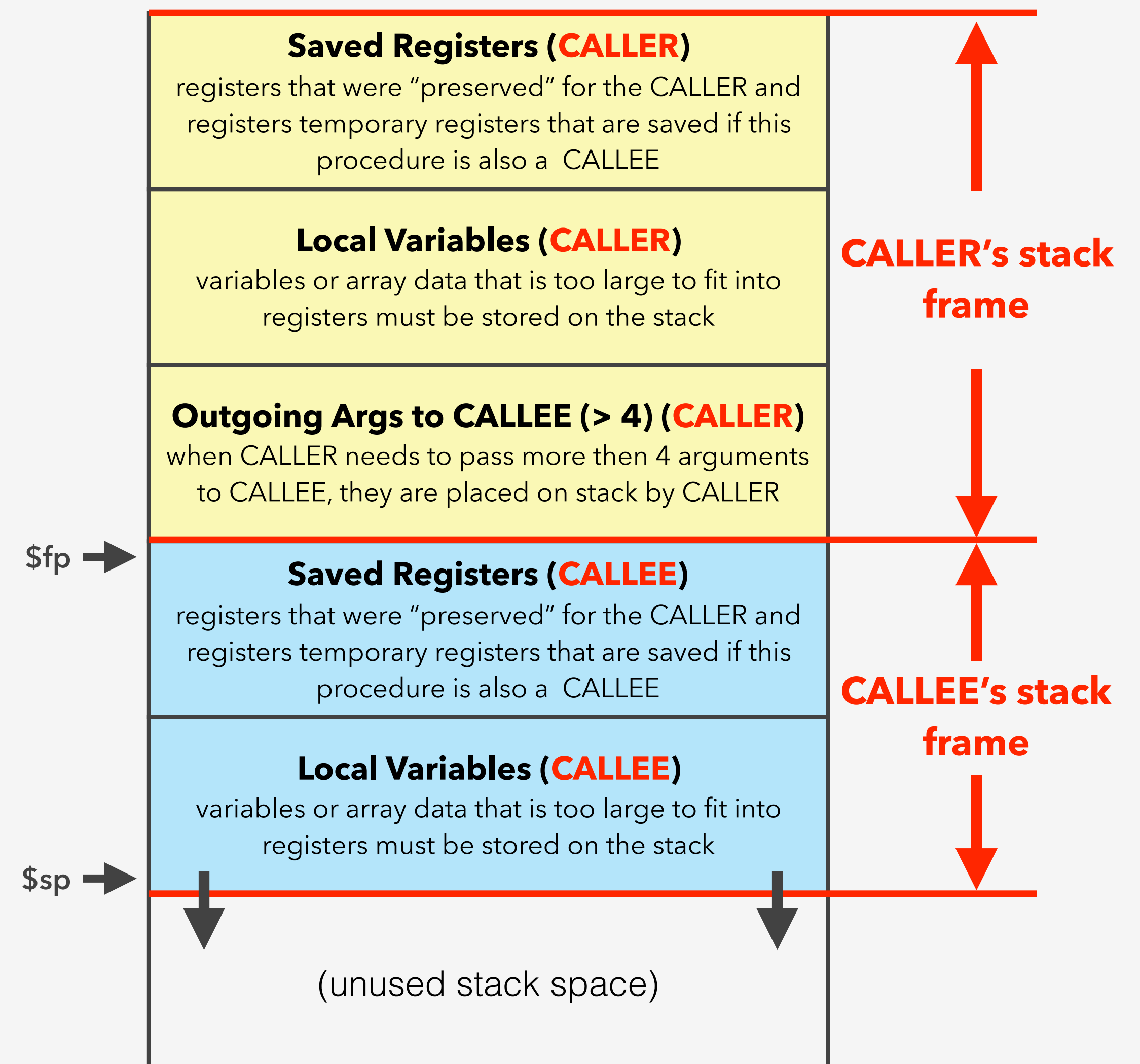
- MIPS provides four registers, \$a0 - \$a3 for passing arguments to a procedure
- If a procedure expects more than four arguments they must be passed on the stack
 - CALLER places first four arguments in registers \$a0 - \$a3
 - CALLER places arguments 5 and up on stack immediately before executing **jal** instruction
 - Arguments should be **LAST** thing placed in CALLER's stack frame – easily accessible by CALLEE
 - Arguments placed on stack are placed in **reverse** order
- CALLEE can access arguments 5 and up by using the **frame pointer** (set \$fp upon entering CALLEE)

```
add $t1, $zero, $a0      # access 1st argument
add $t2, $zero, $a1      # access 2nd argument

lw  $t5, 4($fp)          # access 5th argument
lw  $t6, 8($fp)          # access 6th argument
lw  $t7, 12($fp)         # access 7th argument
```

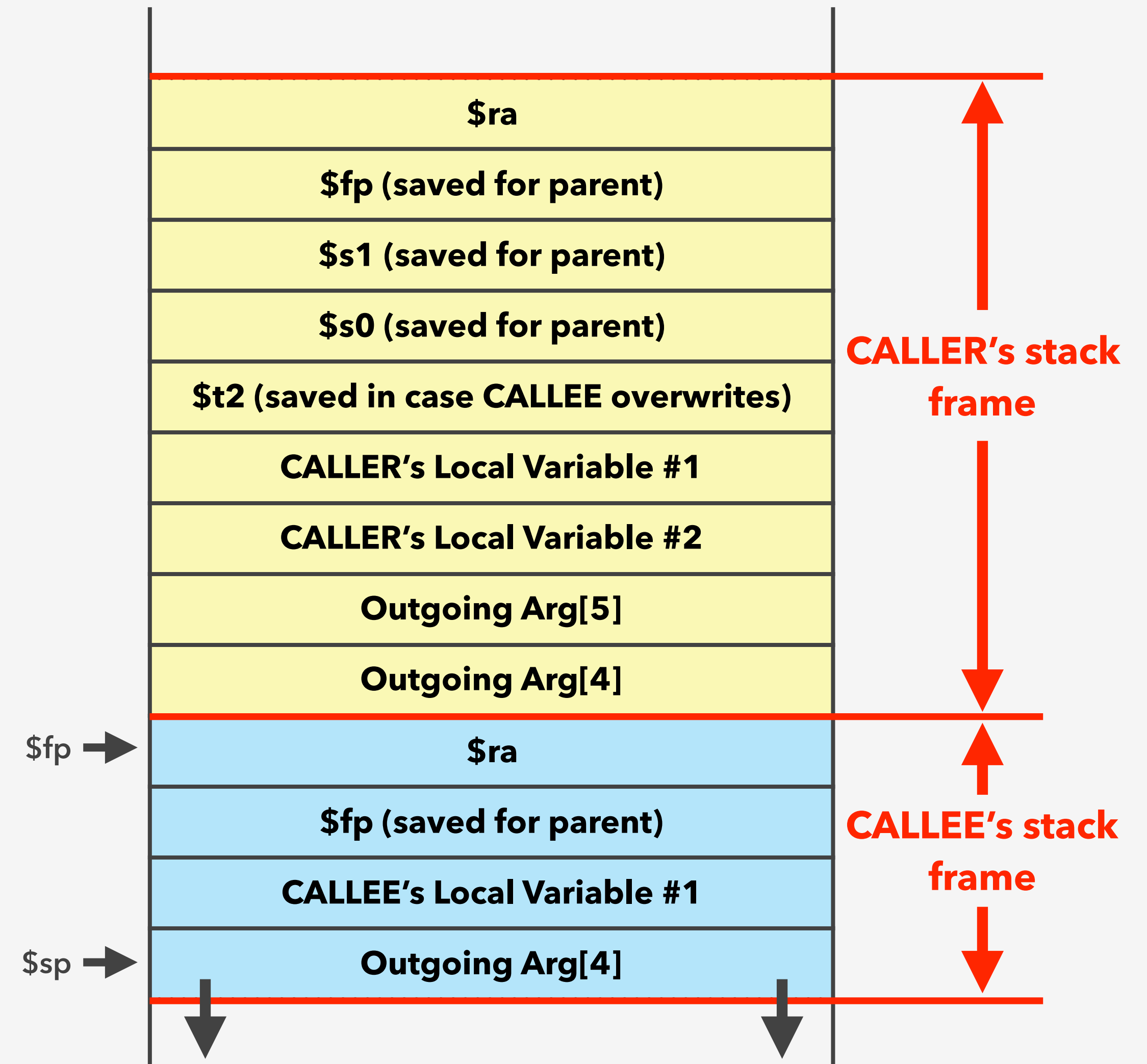
Stack Frame Overview

- The **stack frame** contains storage for the CALLER's data that it wants preserved after the invocation of CALLEEs
- The CALLEE uses the stack for the following:
 1. Accessing the arguments that the CALLER passes to it (specifically, the 5th and greater)
 2. "Preserving" non-temporary registers that it wishes to modify
 3. Storing/accessing its own local variables
- The **frame pointer** keeps track of the boundary between stack frames



Stack Frame Overview

- The **stack frame** contains storage for the CALLER's data that it wants preserved after the invocation of CALLEEs
- The CALLEE uses the stack for the following:
 1. Accessing the arguments that the CALLER passes to it (specifically, the 5th and greater)
 2. "Preserving" non-temporary registers that it wishes to modify
 3. Storing/accessing its own local variables
- The **frame pointer** keeps track of the boundary between stack frames



Example: A Sort Procedure in C

- Another procedure example

```
void sort (int array[], int len) {  
    int i, j;  
    for (i = 0; i < len; i += 1) {  
        for (j = i - 1; j >= 0 && array[j] > array[j + 1]; j -= 1) {  
            swap(array, j, j+1);  
        }  
    }  
}
```

Assume:
\$a0 has base address of array
\$a1 has len

```
void swap (int array[], int i, int j) {  
    int temp;  
    temp = array[i];  
    array[i] = array[j];  
    array[j] = temp;  
}
```

Assume:
\$a0 has base address of array
\$a1 has i
\$a2 has j

Example: A Sort Procedure in C – Swap Procedure

swap:

load array[i] into register

sll \$t1, \$a1, 2 **# compute byte offset of i**

add \$t1, \$a0, \$t1 **# compute effective address of array[i]**

lw \$t3, 0(\$t1) **# load array[i] into register**

load array[j] into register

sll \$t0, \$a2, 2 **# compute byte offset of j**

add \$t0, \$a0, \$t0 **# compute effective address of array[j]**

lw \$t2, 0(\$t0) **# load array[j] into register**

store values into memory swapped

sw \$t3, 0(\$t0) **# store array[i] into array[j]**

sw \$t2, 0(\$t1) **# store array[j] into array[i]**

jr \$ra **# return to caller**

Example: A Sort Procedure in C – Procedure Body

	move \$s2, \$a0 move \$s3, \$a1	# save \$a0 into \$s2 # save \$a1 into \$s3	Move input params
	move \$s0, \$zero	# i = 0	Outer Loop
for1tst:	slt \$t0, \$s0, \$s3	# \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n)	
	beq \$t0, \$zero, exit1	# go to exit1 if \$s0 ≥ \$s3 (i ≥ n)	Inner Loop
	addi \$s1, \$s0, -1	# j = i - 1	
for2tst:	slti \$t0, \$s1, 0	# \$t0 = 1 if \$s1 < 0 (j < 0)	
	bne \$t0, \$zero, exit2	# go to exit2 if \$s1 < 0 (j < 0)	
	sll \$t1, \$s1, 2	# \$t1 = j * 4	
	add \$t2, \$s2, \$t1	# \$t2 = array_base + (j * 4)	
	lw \$t3, 0(\$t2)	# \$t3 = array[j]	
	lw \$t4, 4(\$t2)	# \$t4 = array[j + 1]	
	slt \$t0, \$t4, \$t3	# \$t0 = 0 if \$t4 ≥ \$t3	Setup and call swap
	beq \$t0, \$zero, exit2	# go to exit2 if \$t4 ≥ \$t3	
	move \$a0, \$s2	# 1st param of swap is array (old \$a0)	
	move \$a1, \$s1	# 2nd param of swap is j	
	addi \$a2, \$s1, 1	# 3rd param of swap is j+1	Inner Loop
	jal swap	# call swap procedure	
	addi \$s1, \$s1, -1	# j -= 1	Outer Loop
	j for2tst	# jump to test of inner loop	
exit2:	addi \$s0, \$s0, 1	# i += 1	Outer Loop
	j for1tst	# jump to test of outer loop	

Example: A Sort Procedure in C – Full Procedure

sort:	addi \$sp, \$sp, -20	# make room on stack for 5 registers
	sw \$ra, 16(\$sp)	# save \$ra on stack
	sw \$s3, 12(\$sp)	# save \$s3 on stack
	sw \$s2, 8(\$sp)	# save \$s2 on stack
	sw \$s1, 4(\$sp)	# save \$s1 on stack
	sw \$s0, 0(\$sp)	# save \$s0 on stack
	...	# procedure body
	...	
exit1:	lw \$s0, 0(\$sp)	# restore \$s0 from stack
	lw \$s1, 4(\$sp)	# restore \$s1 from stack
	lw \$s2, 8(\$sp)	# restore \$s2 from stack
	lw \$s3, 12(\$sp)	# restore \$s3 from stack
	lw \$ra, 16(\$sp)	# restore \$ra from stack
	addi \$sp, \$sp, 20	# restore stack pointer
	jr \$ra	# return to calling routine

Caller Conventions

- The CALLER will:
 - Save all temp registers that it wants to survive subsequent procedure calls into its stack frame (\$t0-\$t9, \$a0-\$a3, and \$v0-\$v1)
 - Pass the first 4 arguments to a CALLEE in registers \$a0-\$a3 – save subsequent arguments on stack, in **reverse** order
 - Call CALLEE procedure, using a **jal** instruction which places the return address in register \$ra
 - If this CALLER is also a CALLEE, you must save \$ra before using **jal**
 - Access CALLEE procedure's return values in registers \$v0-\$v1 after CALLEE returns
 - Restore all temp registers that were saved prior to calling CALLEE
 - Be sure to grab return value from CALLEE prior to restoring any saved \$v0-\$v1 from stack or you will overwrite the CALLEE's return value
- **IMPORTANT NOTE: A CALLER MAY ALSO BE ALL CALLEE**

Callee Conventions

- If needed the CALLEE will:
 - 1) Allocate a stack frame with space for saved registers, local variables, and spilled args
 - 2) Save any “preserved” registers that it will use/overwrite: \$ra, \$sp, \$fp, \$gp, \$s0-\$s7
 - 3) If CALLEE has local variables -or- needs access to args on the stack, save CALLER’s frame pointer and set \$fp to 1st entry of CALLEE’s stack
 - 4) EXECUTE procedure
 - 5) Place return values in \$v0-\$v1
 - 6) Restore saved registers including those that were preserved for CALLER
 - 7) Restore \$sp to its original value
 - 8) Return to CALLER with jr \$ra
- **IMPORTANT NOTE: A CALLEE MAY ALSO BE ALL CALLER**