UMBC
AN HONORS UNIVERSITY

# Procedures and Stacks

Ergun Simsek

CMSC 411 | Lectures 8

## What are Procedures?

a.k.a. functions, methods, and subroutines

*An independent code module that fulfills some concrete task and is referenced within a larger body of source code*

Key Idea:

- main routine *M* **calls** a procedure *P*
- *P* does some work, then **returns** to *M*
  - execution in *M* picks up where left off
  - i.e., the instruction in *M* right after the one that called *P*

# Why Use Procedures?

Readability

- divide up long program into smaller procedures

Reusability

- call same procedure from many parts of code
- programmers can use each others' code

Parameterizability

- same function can be called with different arguments/parameters at runtime

Any other reason…?



# And It "Sort Of" Works

**Example**

```
.globl x
.data
x:   .word 9

.globl fee
.text
fee:
    add   $v0,$a0,$a0
    addi  $v0,$v0,-1
    jr    $ra

.globl main
.text
main:
     lw     $a0,x
     jal    fee
     jr     $ra
```

Callee

Caller

Works for special cases where the Callee needs few resources and calls no other functions.

This type of function is called a LEAF function.

But there are lots of issues:

- How can fee call functions?
- More than 4 arguments?
- Local variables?
- Where will main return to?

## Using Procedures

A "calling" program (Caller) must:

- Provide procedure "parameters" / "arguments"
  - ➢ put the arguments in a place where the procedure can access them
- Transfer control to the procedure
  - ➢ jump to it

A "called" procedure (Callee) must:

- Acquire the resources needed to perform the function
- Perform the function
- Place results in a place where the Caller can find them
- Return control back to the Caller



## What might go wrong?

Linkage Problems

- We need a way to pass arguments into procedures and receive results from procedures
- Procedures need storage for their LOCAL variables
- Procedures need to call other procedures
- Procedures might call themselves (Recursion)

Q: How can we solve these problems?

(Partial) Solution:

1. Make an agreement between Caller and Callee, e.g.
   - Give the Callee some "scratch" registers($t registers) to play with
     - If the Caller cares about these, it must preserve them ($t registers)
   - Give the Caller some registers ($s registers) that the Callee won't clobber
     - If the Callee touches them, it must restore them ($s registers)
2. Allocate registers for some specific functions ➔ *Next Slide*

# MIPS Register Usage

Conventions designate registers for procedure arguments ($4-$7) and return values ($2-$3).

Transfer control to Callee using the **jal** instruction

The ISA designates a "linkage register" for calling procedures ($31) allowing Callee to go back to Caller once done

Return to Caller with the **jr $31** or **jr $ra** instruction

| Name | Register number | Usage |
|---|---|---|
| $zero | 0 | the constant value 0 |
| $at | 1 | assembler temporary |
| $v0-$v1 | 2-3 | procedure return values |
| $a0-$a3 | 4-7 | procedure arguments |
| $t0-$t7 | 8-15 | temporaries |
| $s0-$s7 | 16-23 | saved by callee |
| $t8-$t9 | 24-25 | more temporaries |
| $k0-$k1 | 26-27 | reserved for operating system |
| $gp | 28 | global pointer |
| $sp | 29 | stack pointer |
| $fp | 30 | frame pointer |
| $ra | 31 | return address |

We'll talk about these very soon

CMSC 411 - Lectures 8 7

# Procedures <--> Stacks

Let's calculate $x^2$ recursively

$x^2 = x^2-2x+1 + 2x -1$
$= (x-1)^2+ 2x-1$

So if we can get $(x-1)^2$, then we can calculate $x^2$ by adding x+x-1 on it

$4^2 = 3^2 + 4 + 4 - 1$
$3^2 = 2^2 + 3 + 3 - 1$
$2^2 = 1^2 + 2 + 2 - 1$
$1^2 = 0^2 + 1 + 1 - 1$

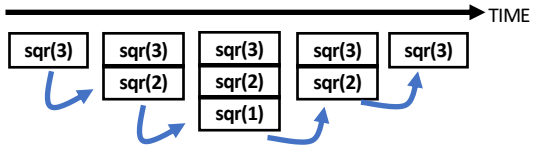
Then let's do this

1. Go down from x to 1
2. When we are doing this, let's put markers where we want to go back
3. Once we reach 1, then start going back to x from 1 by adding 2*current value -1

CMSC 411 - Lectures 8

## Lives of Activation Records

```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}
```




Where do we store activation records?

A procedure call creates a new activation record.
Caller's record needs to be preserved because we'll need it when call finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

9

## We Need Dynamic Storage!

What we need is a SCRATCH memory for holding temporary variables.

We'd like for this memory
(i) to grow and shrink as needed.
(ii) to have an easy management policy.

One possibility is a

**STACK**

A last-in-first-out (LIFO) data structure.

Some interesting properties of stacks:
- SMALL OVERHEAD. Only the top is directly visible, the so-called "top-of-stack"
- Add things by PUSHING new values on top.
- Remove things by POPPING off values.

10

# MIPS Stack Convention

We use a register for the Stack Pointer

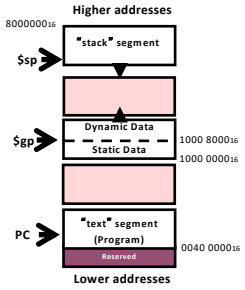
$sp = $29

Stack grows DOWN (towards lower addresses) on pushes and allocates

$sp points to the TOP *used* location.

Place stack far away from our program and its data

Another implementation possibility:
Stacks that grow "UP" and SP points to first UNUSED location




**Stack Management Primitives**

| | | |
|---|---|---|
| ALLOCATE k: reserve k WORDS of stack | Reg[SP] = Reg[SP] - 4*k | addi $sp, $sp, -4*k |
| DEALLOCATE k: release k WORDS of stack | Reg[SP] = Reg[SP] + 4*k | addi $sp, $sp, 4*k |
| PUSH rx: push Reg[x] onto stack | Reg[SP] = Reg[SP] - 4<br>Mem[Reg[SP]] = Reg[x] | addi $sp, $sp, -4<br>sw $rx, 0($sp) |
| POP rx: pop the value on the top of the stack into Reg[x] | Reg[x] = Mem[Reg[SP]]<br>Reg[SP] = Reg[SP] + 4; | lw $rx, 0($sp)<br>addi $sp, $sp, 4 |

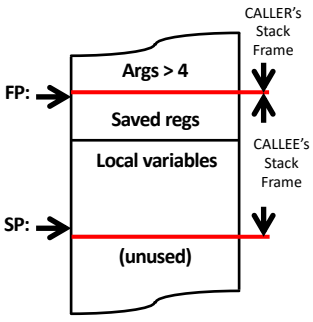
11

# Solving Procedure Linkage "Problems"

Let's "waste" one more register:

- $30 = **$fp** (frame pointer) points to the start of Callee's activation record on the stack
  - Note that if there are more than 4 arguments, then we need to store them in Caller's Stack Frame
- Remember $29 = **$sp** points to TOP of stack (iow, the end of Callee's activation record on the stack)
- Together $29 and $30 are bookends to activation record
- $31 = **$ra** (return address back to Caller)

The STACK FRAME contains storage for the Caller's volatile state that it wants preserved after the invocation of Callees.




The Callee will use the stack for

1) Accessing the arguments that the CALLER passes to it (specifically, the 5th and greater)
2) Saving non-temporary registers that it wishes to modify
3) Accessing its own local variables

It is possible to use only the SP to access a stack frame, but offsets may change due to ALLOCATEs and DEALLOCATEs. For convenience a $fp is used to provide CONSTANT offsets to local variables and arguments

CMSC 411 - Lectures 8 12

# Procedure Stack Usage

ADDITIONAL space must be allocated in the stack frame for:

1. Any SAVED registers the procedure uses ($s0-$s7)
2. Any TEMPORARY registers that the procedure wants preserved IF it calls other procedures ($t0-$t9)
3. Any LOCAL variables declared within the procedure
4. Other TEMP space IF the procedure runs out of registers (RARE)
5. Enough "outgoing" arguments to satisfy the worse case ARGUMENT SPILL of ANY procedure it calls. (SPILL is the number of arguments greater than 4)

•The CALLEE is free to use $t0-$t9, $a0-$a3, and $v0-$v1, and the memory below the stack pointer.

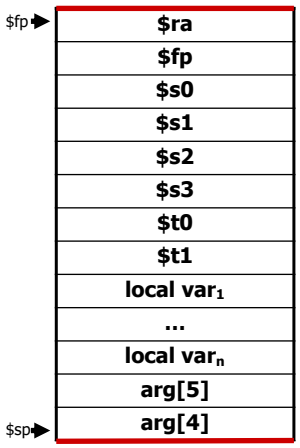
•The registers $s0-$s7, $sp, $ra, $gp, $fp, and the memory above the stack pointer must be preserved by the CALLEE

13

# Stack Snap Shot: A typical procedure

A typical "activation record" or "stack frame"

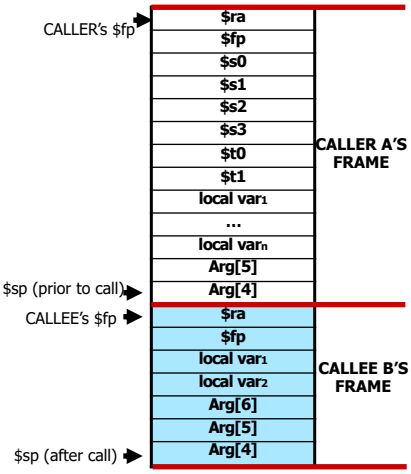
- save $ra and $fp first
- save values of "saved registers" modified by this procedure
  - e.g.: $s0, $s1, $s2, $s3
- save values of "temp registers" that must survive calls to other procedures from this procedure
  - e.g.: $t0, $t1
  - should be saved immediately before calling other procedure; restored immediately after
- any local variables needed (that are not in registers) reside on the stack
  - e.g.: locals $var_1$ … $var_n$
- any spillover arguments for calling other procedures [in reverse order]
  - e.g.: arg[4], arg[5]



# Stack Snap Shot: Caller + Callee

procedure A calls procedure B

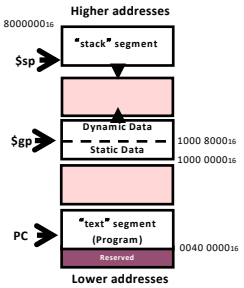
- B has less stuff that needs to be saved on stack
- Can you tell the number of args for B?
  - NOPE!
- Can you tell the max number of args needed by any procedure called by A?
  - Yes, 6
- Where in CALLEE's stack frame might one find CALLER's $fp?
  - At -4($fp)






# What about $gp?

- Constants and global variables are stored in the heap!
- Register $gp (28) is a global pointer that points to heap
- The objects in this heap can be quickly accessed with a single load or store instruction.



Let's review Procedure Linkage one more time!



## Procedure Linkage is Nontrivial

The details can be overwhelming. How do we manage this complexity?

- Abstraction: High-level languages hide the details

There are great many implementation choices:

- which variables are saved
- who saves them
- where are arguments stored?

Solution: CONTRACTS!

- Caller and Callee must agree on the details

# Procedure Linkage: Caller Contract

The **CALLER** will:

1) Save all temp registers that it wants to survive subsequent calls in its stack frame (t0-$t9, $a0-$a3, and $v0-$v1)
2) Pass the first 4 arguments in registers **$a0**-**$a3**, and save subsequent arguments on stack, in reverse order.
3) Call procedure, using a **jal** instruction (places return address in **$ra**).
4) Access procedure's return values in **$v0**-**$v1**



# Procedure Linkage: Callee Contract

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 used: (**$ra**, **$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

7) Fix **$sp** to its original value

8) Return to CALLER with **jr $ra**