CSc 520

Principles of Programming Languages

5 : Memory Management — Stack Allocation

Christian Collberg

collberg+520@gmail.com

Department of Computer Science

University of Arizona

Questions

- How do we deal with recursion? Every new recursive call should get its own set of local variables.
- How do we pass parameters to a procedure?
    - Call-by-Value or Call-by-Reference?
    - In registers or on the stack?
- How do we allocate/access local and global variables?
- How do we access non-local variables? (A variable is non-local in a procedure P if it is declared in procedure that statically encloses P.)
- How do we pass large structured parameters (arrays and records)?

Stack Allocation

Local Variables: stored on the run-time stack.

Actual parameters: stored on the stack or in special argument registers.

- Languages that allow recursion cannot store local variables in the `Static Data` section. The reason is that every **Procedure Activation** needs its own set of local variables.
- For every new procedure activation, a new set of local variables is created on the run-time stack. The data stored for a procedure activation is called an **Activation Record**.
- Each **Activation Record** (or **(Procedure) Call Frame**) holds the local variables and actual parameters of a particular procedure activation.

Storage Allocation...

- When a procedure call is made the **caller** and the **callee** cooperate to set up the new frame. When the call returns, the frame is removed from the stack.

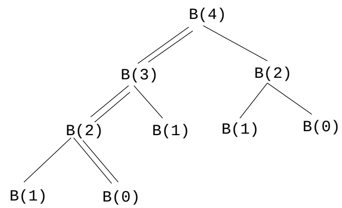
returned value
actual parameter 1 actual parameter 2 ...
return address
static link
control link
saved registers, etc
local variable 1 local variable 2 ...

Recursion

Recursion Examples

Example I (Factorial function): R_0 and R_1 are registers that hold temporary results.

Example II (Fibonacci function): We show the status of the stack after the first call to `B(1)` has completed and the first call to `B(0)` is almost ready to return.
The next step will be to pop `B(0)`'s AR, return to `B(2)`, and then for `B(2)` to return with the sum `B(1)+B(0)`.



Recursion Example

```
PROCEDURE F (n:INTEGER) :INTEGER;
  VAR L:INTEGER;
BEGIN
(1) IF n <= 1
(2) THEN L:=1;
(3) ELSE
(4)  R0:=F(n-1);
(5)  R1:=n;
(6)  L:=R0 * R1;
(7) ENDIF;
(8) RETURN L;
END F;
BEGIN
    (9) C:=F(3);
    (10)
END
```

n = 1	F(1)
L = 1	
RetAddr=(5)	
RetVal=1	
n = 2	F(2)
L = ?	
RetAddr=(5)	
RetVal=?	
n = 3	F(3)
L = ?	
RetAddr=(10)	
RetVal=?	
C = ?	main

Recursion Example

```
PROCEDURE B (n:INTEGER):INTEGER;
 VAR L:INTEGER;
BEGIN
(1)  IF n <= 1
(2)  THEN L:=1;
(3)  ELSE
(4)   R0:=B(n-1);
(5)   R1:=B(n-2);
(6)   L:=R0 + R1
(7)  ENDIF;
(8)  RETURN L;
END B;
BEGIN
     (9) C:=B(4);
     (10)
END
```

n = 1; L = 1	B(0)
RetAddr=(6)	
RetVal=1	
n=2; L=?; R_0=1	B(2)
RetAddr=(5)	
RetVal=?	
n = 3; L = ?	B(3)
RetAddr=(5)	
RetVal=?	
n = 4; L = ?	B(4)
RetAddr=(10)	
RetVal=?	
C = ?	main

Calling Conventions

Procedure Call Conventions

- **Who** does **what when** during a procedure call? Who pushes/pops the activation record? Who saves registers?
- This is determined partially the hardware but also by the conventions imposed by the operating system.
- Some work is done by the **caller** (the procedure making the call) some by the **callee** (the procedure being called).

Work During Call Sequence: Allocate Activation Record, Set up Control Link and Static Link. Store Return Address. Save registers.

Work During Return Sequence: Deallocate Activation Record, Restore saved registers, Return function result Jump to code following the call-site.

Example Call/Return Sequence

The Call Sequence

The caller: Allocates the activation record, Evaluates actuals, Stores the return address, Adjusts the stack pointer, and Jumps to the start of the **callee**'s code.

The callee: Saves register values, Initializes local data, Begins execution.

The Return Sequence

The callee: Stores the return value, Restores registers, Returns to the code following the `call` instr.

The caller: Restores the stack pointer, Loads the return value.

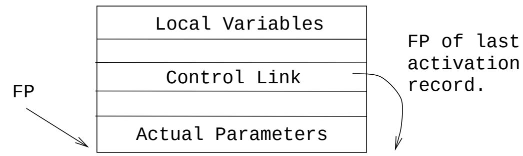
The Control Link

The Control Link

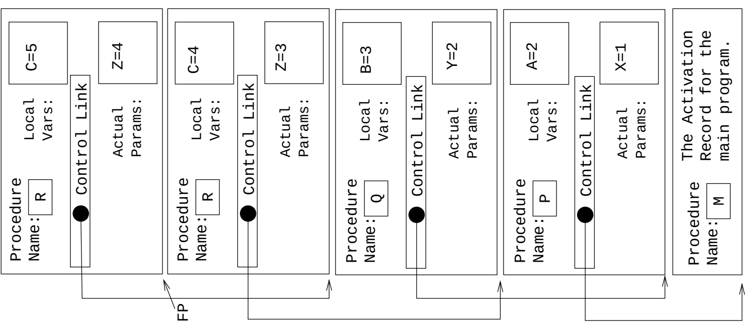
- Most procedure calling conventions make use of a **frame pointer** (FP), a register pointing to the (top/bottom/middle of the) current activation record.
- Local variables and actual parameters are accessed relative the FP. The offsets are determined at compile time.
- MIPS example: `lw $2, 8($fp)`.

The Control Link. . .

- Each activation record has a **control link** (aka **dynamic link**), a pointer to the previous activation record on the stack.
- The control link is simply the stored FP of the previous activation.



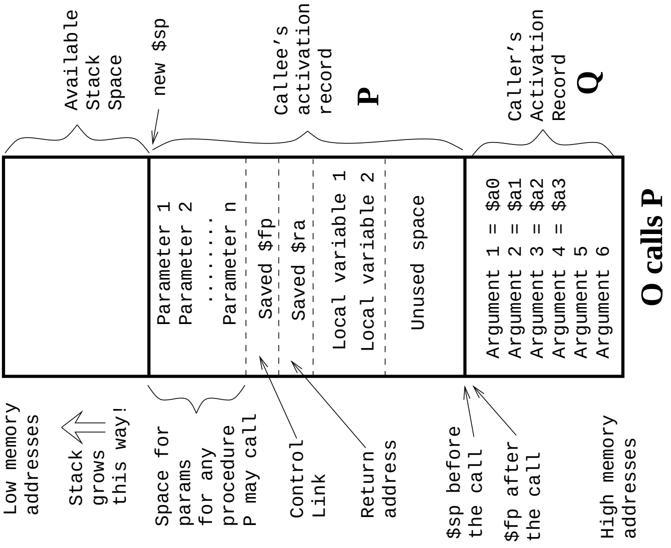
The Control Link. . .




```
PROGRAM M;
 PROC P(X);
  LOCAL A;
  PROC Q(Y);
   LOCAL B;
   PROC R(Z);
    LOCAL C;
    BEGIN
     C:=Z+1;
     R(C);
    END R;
   BEGIN
    B:=Y+1;
    R(B);
   END Q;
  BEGIN
   A:=X+1;
   Q(A);
  END P;
 BEGIN
  P(0);
 END M.
```


Procedure Call on the MIPS

MIPS Activation Record



MIPS Procedure Call

- Assume that a procedure **Q** is calling a procedure **P**. **Q** is the **caller**, **P** is the **callee**. **P** has **K** parameters.
- **Q** has an area on it's activation record in which it passes arguments to procedures that it calls. **Q** puts the first 4 arguments in registers (`$a0--$a3` $\equiv$ `$4--$7`). The remaining $\mathrm{K}-4$ arguments **Q** puts in its activation record, at `16+$sp, 20+$sp, 24+$sp` etc. (We're assuming that all arguments are 4 bytes long).
- Note that there is space in **Q**'s activation record for the first 4 arguments, we just don't put them in there.
- We must know the max number of parameters of an call **Q** makes, to know how large to make its activation record.

MIPS Procedure Call. . .

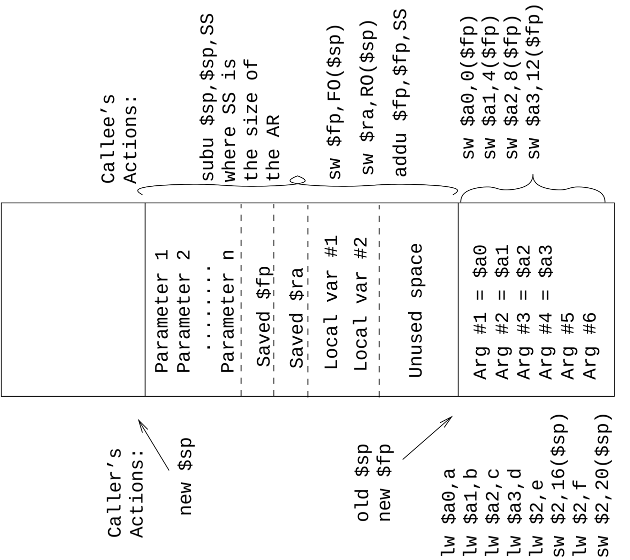
- Next, **Q** executes a `jal` (jump and link) instruction. This puts the return address (the address right after the `jal` instruction) into register `$ra` (`$31`), and then jumps to the beginning of **P**.
- Before **P** starts executing it's code, it has to set up it's stack frame (activation record). How much space does it need?
  1. Space for local variables,
  2. Space for the control link (old `$fp` 4 bytes).
  3. Space to save the return address `$ra` (4 bytes).
  4. Space for parameters **P** may want to pass when making calls itself.

  Furthermore, the size of the activation record must be a multiple of 8! This can all be computed at compile-time.

MIPS Procedure Call. . .

- Given the size of the stack frame (`SS`) we can set it up by subtracting from `$sp` (remember that the stack grows towards lower addresses!): `subu $sp,$sp,SS`. We also set `$fp` to point at the bottom of the stack frame.
- If **P** makes calls itself, it must save `$a0--$a3` into their stack locations.
- Procedures that don't make any calls are called **leaf routines**. They don't need to save `$a0--$a3`.
- Procedures that make use of registers that need to be preserved accross calls, must make room for them in the activation record as well.

MIPS Procedure Call…

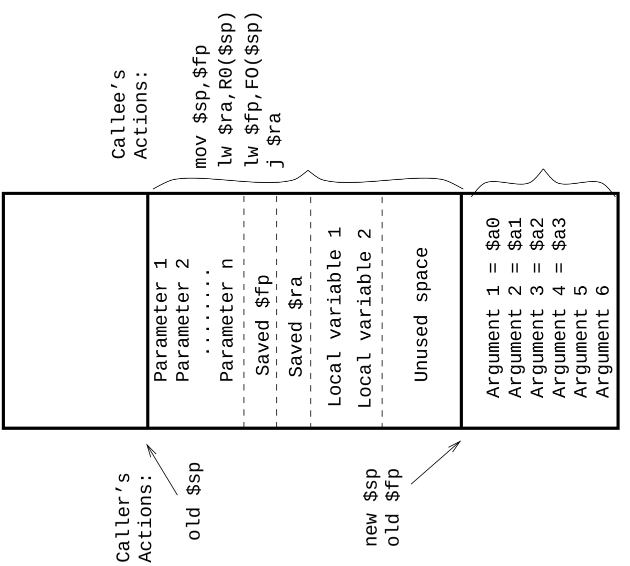


O calls P

MIPS Procedure Returns

- When **P** wants to return from the call, it has to make sure that everything is restored exactly the way it was before the call.
- **P** restores `$sp` and `$fp` to their former values, by reloading the old value of `$fp` from the activation record.
- **P** then reloads the return address into `$ra`, and jumps back to the instruction after the call.

MIPS Procedure Returns…



P returns to O

Readings and References

- Read Scott, pp. 106–111, 410–413

Summary

- Each procedure call pushes a new activation record on the run-time stack. The AR contains local variables, actual parameters, a static (access) link, a dynamic (control) link, the return address, saved registers, etc.
- The frame pointer (FP) (which is usually kept in a register) points to a fixed place in the topmost activation record. Each local variable and actual parameter is at a fixed offset from FP.

Summary. . .

- The dynamic link is used to restore the FP when a procedure call returns.
- The static link is used to access non-local variables, i.e. local variables which are declared within a procedure which statically encloses the current one.