CSc 520

Principles of Programming Languages

5 : Memory Management — Stack Allocation

Christian Collberg

collberg+520@gmail.com

Department of Computer Science
University of Arizona

Copyright © 2008 Christian Collberg
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.

<table>
<thead>
<tr>
<th>returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual parameter 1</td>
</tr>
<tr>
<td>actual parameter 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>static link</td>
</tr>
<tr>
<td>control link</td>
</tr>
<tr>
<td>saved registers, etc</td>
</tr>
<tr>
<td>local variable 1</td>
</tr>
<tr>
<td>local variable 2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
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 B (n:INTEGER):INTEGER;
VAR L:INTEGER;
BEGIN
(1) IF n <= 1
(2) THEN L := 1;
(3) ELSE
(4) \( R_0 := B(n-1) \);
(5) \( R_1 := B(n-2) \);
(6) \( L := R_0 + R_1 \)
(7) ENDIF;
(8) RETURN L;
END B;
BEGIN
(9) C := B(4);
(10)
END

\[\begin{align*}
\text{B(0)} &: n = 1; L = 1 \\
\text{B(2)} &: n = 2; L = ?; R_0 = 1 \\
\text{B(3)} &: n = 3; L = ? \\
\text{B(4)} &: n = 4; L = ? \\
\text{main} &: C = ?
\end{align*}\]
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
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: \texttt{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.

```
<table>
<thead>
<tr>
<th>Local Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Link</td>
</tr>
<tr>
<td>Actual Parameters</td>
</tr>
</tbody>
</table>

FP of last activation record.
```
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.

The Activation Record for the main program.
Procedure Call on the MIPS
MIPS Activation Record

Low memory addresses
Stack grows this way!

Space for params for any procedure P may call

Control Link
Return address

$sp before the call

$fp after the call

High memory addresses

Parameter 1
Parameter 2
....... Parameter n

Saved $fp

Saved $ra

Local variable 1
Local variable 2

Unused space

Argument 1 = $a0
Argument 2 = $a1
Argument 3 = $a2
Argument 4 = $a3
Argument 5
Argument 6

Available Stack Space
new $sp

Callee’s activation record

P

O calls P

Caller’s Activation Record

Q
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 $K - 4$ 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.
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

Caller's Actions:
- new $sp

Callee's Actions:
- subu $sp,$sp,SS where SS is the size of the AR
- addu $fp,$fp,SS
- lw $a0,a
- lw $a1,b
- lw $a2,c
- lw $a3,d
- lw $2,e
- sw $2,16($sp)
- lw $2,f
- sw $2,20($sp)
- sw $a0,0($fp)
- sw $a1,4($fp)
- sw $a2,8($fp)
- sw $a3,12($fp)

Unused space

Parameter 1
Parameter 2
......
Parameter n

Saved $fp
Saved $ra

Local var #1
Local var #2

Arg #1 = $a0
Arg #2 = $a1
Arg #3 = $a2
Arg #4 = $a3
Arg #5
Arg #6

new $fp

old $sp

Q calls P

Saved $fp
Saved $ra
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**

**Caller's Actions:**
- old $sp

**Callee's Actions:**
- mov $sp,$fp
- lw $ra,R0($sp)
- lw $fp,FO($sp)
- j $ra

**Parameter 1**
**Parameter 2**
**.........**
**Parameter n**

**Saved $fp**
**Saved $ra**

**Local variable 1**
**Local variable 2**

**Unused space**

**Argument 1 = $a0**
**Argument 2 = $a1**
**Argument 3 = $a2**
**Argument 4 = $a3**
**Argument 5**
**Argument 6**

**P returns to O**
Read 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.
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.