Run-Time Memory Organization

<table>
<thead>
<tr>
<th>Low Addresses</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heap</td>
</tr>
</tbody>
</table>

Run-Time Memory Organization...

- This is a common organization of memory on Unix systems.
- The **Text Segment** holds the code (instructions) of the program. The **Initialized Data** segment holds strings, etc, that don’t change. **Static Data** holds global variables. The **Stack** holds procedure activation records and the **Heap** dynamic data.

Procedure Calls

- 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)?
Global Variables – MIPS

- How do we allocate space for and access global variables? We’ll examine three ways.

--- Running Example: 

PROGRAM P;
VAR X : INTEGER; (* 4 bytes. *)
VAR C : CHAR; (* 1 byte. *)
VAR R : REAL; (* 4 bytes. *)
END.

Global Variables – Allocation by Name

- Allocate each global variable individually in the data section. Prepend an underscore to each variable to avoid conflict with reserved words.

- Remember that every variable has to be aligned on an address that is a multiple of its size.

.data
_X: .space 4
_C: .space 1
    .align 2 # 4 byte boundary.
_R: .space 4
.text
main: lw $2, _X

Storage Allocation

Global Variables are stored in the Static Data area.
Strings (such as "Bart!") are stored in the Initialized Data section.
Dynamic Variables are stored on the Heap:

PROCEDURE P ();
    VAR X : POINTER TO CHAR;
    BEGIN
        NEW(X);
    END P

Own Variables are stored in the Static Data area. An Own variable can only be referenced from within the procedure in which it is declared. It retains its value between procedure calls.

PROCEDURE P (X : INTEGER);
    OWN W : INTEGER;
    VAR L : INTEGER;
    BEGIN
        W := W + X; END P
Storage 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.

Global Variables – Allocation in Block

- Allocate one block of static data (called _Data, for example), holding all global variables. Refer to individual variables by offsets from _Data.

```
.data
_Data: .space 48
.text
main: lw $2, _Data+0 # X
      lb $3, _Data+4  # C
      l.s $f4, _Data+8 # R
```

Global Variables – Allocation on Stack

- Allocate global variables on the bottom of the stack. Refer to variables through the Global Pointer $gp, which is set to point to the beginning of the stack.

```
main:  subu $sp,$sp,48
       move $gp,$sp
       lw  $2, 0($gp)     # X
       lb  $3, 4($gp)    # C
       l.s $f4, 8($gp)   # R

_X: .space 4 Each access lw $2, _X takes 2 cycles.
_Data: .space 48 Each access lw $2, _Data+32 takes 2 cycles.
subu $sp,$sp,48 1 cycle to access the first 64K global variables.
```
### 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 $B(1)+B(0)$.

### Procedure Call Conventions

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

---

**PROCEDURE F (n:INTEGER) : INTEGER;**

```
VAR L:INTEGER;
BEGIN
(1) IF n <= 1
(2) THEN L:=1;
(3) ELSE
(4) R_0:=F(n-1);
(5) R_1:=n;
(6) L:=R_0 + R_1;
(7) ENDIF;
(8) RETURN L;
END F;
BEGIN
(9) C:=F(3);
(10) END
```
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>
<th>FP of last activation record.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td></td>
</tr>
<tr>
<td>Control Link</td>
<td></td>
</tr>
<tr>
<td>Actual Parameters</td>
<td></td>
</tr>
</tbody>
</table>

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: \( \text{lw} \ $2, 8($fp) \).
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!): \( \text{subu } sp, sp, ss \). 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 across calls, must make room for them in the activation record as well.

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 $K - 4$ 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...

Caller's Actions:
- new $sp

Parameter 1
Parameter 2
Parameter n
Saved $fp
Saved $ra
Local var #1
Local var #2
Unused space

Callee's Actions:
- subu $sp,$sp,SS
  (where SS is the size of the AR)
- sw $fp,F0($sp)
- sw $ra,R0($sp)
- addu $fp,$fp,SS

old $sp
new $fp

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)

old $sp
new $fp

MIPS Procedure Returns...

Caller's Actions:
- old $sp

Parameter 1
Parameter 2
Parameter n
Saved $fp
Saved $ra
Local var #1
Local var #2
Unused space

Callee's Actions:
- mov $sp,$fp
- lw $ra,R0($sp)
- lw $fp,F0($sp)
- j $ra

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

P returns to Q

Q 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.

Readings and References

- Read Scott, pp. 115–122, 427–437
- Read the Dragon Book: Procedures 389–394
- Storage Organization 396–397, 401–404
- Activation Records 398–400
- Calling Sequences 404–408
- Lexical Scope 411, 415–418
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.