Supporting Procedure Call

Supporting Procedure Call

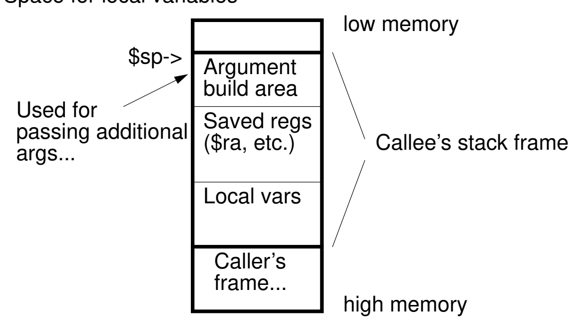
- Procedures (or functions) are a crucial program structuring mechanism.
- To support procedures we need to define a calling convention: a sequence of steps followed by the calling procedure and the called procedure to assure correct passage of parameters, results, and control flow...
- Each machine (compiler, really) uses its own calling convention
- The convention is controlled by software and/or hardware
- In RISC machines, the hardware performs only simple instructions, so most of the onus is on the programmer/compiler to issue the proper sequence of instructions to assure correct implementation of procedure calls

Program Stack

- Each executing program (process) has a stack
- A stack is a dynamic data structure that is accessed in a LIFO manner (you knew this already)
- The program stack is automatically allocated by the OS when the program starts up
- The register $sp (register 29 on the MIPS) is automatically loaded to point to the first empty slot on the top of the stack
- By convention, the stack grows towards *lower* memory addresses
  - To allocate space on the stack, decrement $sp
  - To free old stack space, increment $sp
- Stacks are used to:
  - Pass parameters
  - Save registers
  - Hold local variables

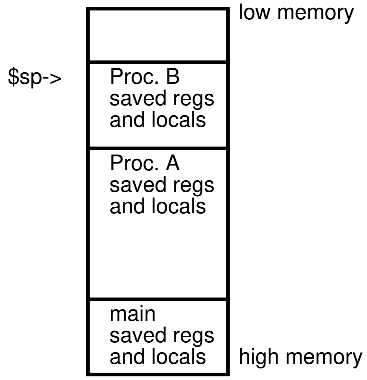
Procedure Stack Frame

- A stack frame is a block of memory on the stack that is used for:
  - Passing arguments
  - Saving registers
  - Space for local variables



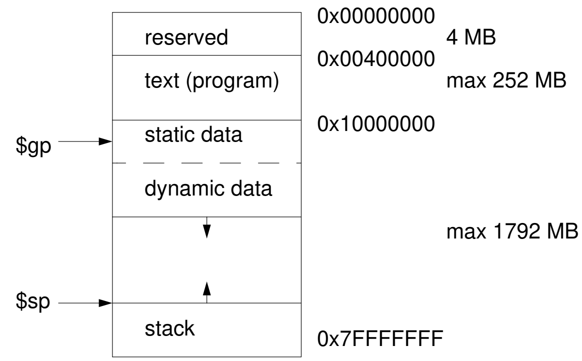
Multiple calls

- Assume your main program calls procedure A, which in turn calls procedure B. During the execution of B, the stack will look like:



Program and Memory Layout

- By MIPS convention, memory is laid out as follows:



- Note that the user only gets half of the address space.

Procedure Call Sequence

- Definitions: *callee* (the procedure that is called); *caller* (the procedure that does the calling). Note that a given procedure can be, at different times, both caller and callee...
- Here is a generic sequence of events surrounding a procedure call. What needs to be done during the calling sequence? (And who does it?)
  1. Caller must pass the return address (where to continue execution after the call) to the callee
  2. Caller must pass parameters to callee
  3. Caller must save registers that the callee might want to use
  4. Jump to the first instruction of the callee
  5. Callee must allocate space for local variables, and possibly save registers
  6. Callee executes...
  7. Callee has to restore registers (possibly) and return to caller
  8. Caller continues....

Mechanisms

- How do we save information? Pass information? Make space for locals? Again, this is defined mainly by convention:
- In the MIPS convention the registers are used for:
  1. passing the return address (by jal target, which places the address of the next instruction into $ra)
  2. passing a small number of parameters (up to 4, in $a0-$a3)
  3. keeping track of the stack pointer ($sp)
  4. returning function values (in $v0-$v1)
- The stack is used for:
  1. save registers that the callee might use
  2. save information about the caller (its $ra, for instance: why?)
  3. pass additional parameters
  4. allocate space for a procedures local variables

Register conventions

- The following conventions dictate the use of registers during procedure call:

Register	Name	Function
$2-3	$v0-v1	return function value
$4-7	$a0-a3	for passing the first 4 parameters; caller-saved (volatile)
$8-15	$t0-t7	caller-saved temporaries (volatile)
$16-23	$s0-s7	callee-saved temporaries
$24-25	$t8-t9	caller-saved temporaries (volatile)
$28	$gp	pointer to global static memory (don't mess)
$29	$sp	stack pointer
$30	$fp	frame pointer (used by some compilers)
$31	$ra	return address (callee-saved)

Who saves/restores the registers?

- *Caller saves.* The caller saves any registers that it wants preserved before making the call, and restores them afterwards.
- *Callee saves.* The callee saves any registers it intends to use, and restores them before it returns.
- MIPS takes a hybrid approach, by classifying some registers as *caller-saved* and some as *callee-saved*.
- *Caller-saved* registers ($t0-$t9) are those that the caller must save/restore (if they need the value after the call). Sometimes these registers are described as *volatile*, because the callee is free to change them without saving/restoring them.
- *Callee-saved* registers ($s0-$s7, $ra) are those that the callee must save/restore if they want to change them.
- Compilers are good at deciding how to allocate the registers to optimize their use, e.g. by placing short-lived values into caller-saved registers, and long-lived values into callee saved registers.

A Convention of My Invention

- The trouble with conventions is that no one agrees on them. For instance:
  1. The text presents two different procedure call conventions, both of which are confusing.
  2. The MIPS manual presents another, which is also confusion.
  3. The cc compiler uses another (close to the one in the MIPS manual)
  4. The gcc compiler uses yet another...
- At a minimum, our convention should agree (in principle) with the ones used by typical compilers. (Ours almost does...)
- We're concerned primarily with 4 points in program execution:
  1. The entry to a called procedure.
  2. The exit from a called procedure.
  3. Just before calling a procedure.
  4. Just after the return from a procedure call.

Procedure Entry

- Allocate stack space by:

```
subu   $sp, $sp, framesize
```

- Framesize is calculated by determining how many bytes are required for
  1. Local variables
  2. Saved registers: usually at least $ra (if we intend to make a call) + space for the callee save registers we intend to use.
  3. Procedure call arguments: If we intend to make a call with more than 4 parameters, we'll need to allocate extra words at the top of our stack frame.
- Save callee-saved registers. A callee must save $s0-$s7 before altering them, since the caller expects to find them unchanged after the call. Register $ra need only be saved if the callee intends to make further calls.
- *It is a good habit to only grow the stack on procedure entry, and not to mess with it again until procedure exit.*

Procedure exit

- Return values. If the procedure is a value returning function, the value should be placed in $v0.
- Restore all callee-saved ($s0-$s7) registers.
- Restore $ra, if necessary.
- Pop the stack frame by adding framesize to $sp:

```
addu   $sp, $sp, framesize
```

- Return to the caller by jumping to the address in $ra:

```
jr     $ra
```


Prior to a Call

- Pass arguments. Place the first 4 arguments into $a0-$a3. The remaining arguments are placed on the stack at offset($sp). The offset is 0 for the 5th argument, 4 for the 6th argument, etc. *This works because we allocated a large enough argument build area on procedure entry (see above)*.
- Save the caller-saved registers. The callee can modify $a0-$a3 and $t0-$t9 with impunity, so if the caller expects to use one of these registers after the call, it must save them.
- Jump to the caller by executing the jal instruction, which will deposit the return address into $ra.

After a call

- Restore any caller-saved registers you saved before the call.
- Often there will be nothing to do here, because we (or the compiler) have been clever and have used the callee saved registers for long-lived values, and the caller-saved registers for short-lived values.

Example: Recursive Factorial

```
int factorial (int n) {
  if (n==0)
    return 1;
  else
    return n * factorial(n-1);
}
```

- How large does the stack frame for factorial need to be?

Assembly Version

```
factorial:
        subu    $sp, $sp, 8         # create stack frame
        sw      $ra, 0($sp)         # save ra
        beq     $a0, $0, base       # base case?
        sw      $a0, 4($sp)         # RECURSIVE CASE:
        addi    $a0, $a0, -1        # save $a0 on stack,
        jal     factorial           # make recursive call
        lw      $t0, 4($sp)         # now restore $a0...
        mul     $v0, $v0, $t0       # ...and multiply
        j       ret                 # drop to bottom
base:   li      $v0, 1              # BASE CASE, return 1
ret:    lw      $ra, 0($sp)         # procedure exit
        addu    $sp, $sp, 24        # restore $ra, $sp
        jr      $ra                 # return to caller
```


Larger Example

```
int doubleIt (int x) {
  return 2*x;
}

int
sumAndDouble (int a, int b, int c, int d, int e, int f) {
  int temp;
  temp = a+b+c+d+e+f;
  temp = doubleIt(temp);
  return temp
}

int main () {
  int x, f[6];
  x = sumAndDouble(f[0], f[1], f[2], f[3], f[4], f[5]);
  printInt(foo);
}
```