Procedures

Class 34

Introduction

- we must support function calls (i.e., procedures)
- to execute a procedure, the following steps occur
    1. put parameters (if any) in a place where the procedure can find and access them
    2. transfer control to the code at the beginning of the procedure
    3. acquire the storage space needed by the procedure
    4. put the return value (if any) in a place where the calling scope can find and access it
    5. return control to the place from which the procedure was called
- and this must be done in such a way that the calling scope is not damaged or inadvertently affected

Instructions

in my directory I have the machine code for hello world

```
$ od -Ax -v -x -w4 hello.o
0000 457f 464c
0004 0201 0001
0008 0100 0800
000c 0000 0100
0010 0000 0404
0014 0070 0710
0018 3400 0000
001c 0000 2800
0020 1100 1000
0024 bd27 c8ff
0028 bfaf 3400
002c beaf 3000
0030 a003 25f0
0034 1c3c 0000
```

- the operating system loads this file into memory
- each instruction is at an address
- first the instruction at 0000 runs
- then the instruction at 0004 runs
- one instruction after another
- unless there's a branch or a jump

The Program Counter

- every CPU has a register called the program counter PC
- a misnomer — should be called instruction-address (in Intel CPUs it's actually called the instruction pointer IP)
- it always contains the address of the current executing instruction

```
0020 1100 1000
0024 bd27 c8ff  ←
0028 bfaf 3400
```

when this instruction is executing, PC is 0024

- at the end of every normal instruction, the PC is incremented by 4, ready to fetch the next instruction
- look at the branch and jump instructions: they modify the PC

Procedure Register Conventions

- we wish to use registers if we can
- `$a0` through `$a3` are used for passing actual parameters to functions
- `$v0` and `$v1` are used for returning a value from a procedure
- `$ra` is used to store the return address

- the actual instruction to call a procedure is jump-and-link `jal`
- it causes control to jump unconditionally to a new instruction and simultaneously records the return address in `$ra`
- to return from a procedure we use the jump register instruction `jr`, using the address in `$ra`

Simple Example: C

```c
int main(void)
{
  ... other stuff
  a = sum(b, c);
  ... other stuff
}

int sum(int x, int y)
{
  int z = x + y;
  return z;
}
```

Simple Example: MIPS

```
main:
    ... other stuff
    ... get values into $a0 and $a1
    jal    sum
    sw     $v0, 0($s0)
    ... other stuff

sum:
    add    $v0, $a0, $a1    # v0 = x + y
    jr     $ra              # return to PC + 4
```

Simple Example

- however, the simple example is misleading, and has shortcomings
- there are a bunch of things it doesn't address:
    - what happens to $ra if one procedure calls another procedure (including the same procedure (i.e. recursion))?
    - what happens if we have more than 4 parameters?
    - what happens when we declare local variables?
    - what happens if the calling scope was using a bunch of the general-purpose registers?

The Stack

- the stack is one of the most central and important concepts in all of computer science
- stacks show up in many places
- they are essential to our understanding of how programs work
- inspired by the spring-loaded stack of plates found in university cafeterias

Stack Operations

- when you push a plate onto the stack, it now has one more plate than it had before
- when you pop a plate off of the stack, it now has one fewer plates than it had before

- the plate that you get with pop is the most recent plate added by the last push
- last in, first out
- a synonym for stack is LIFO

- this is the opposite of a queue, which is FIFO

Memory Stack

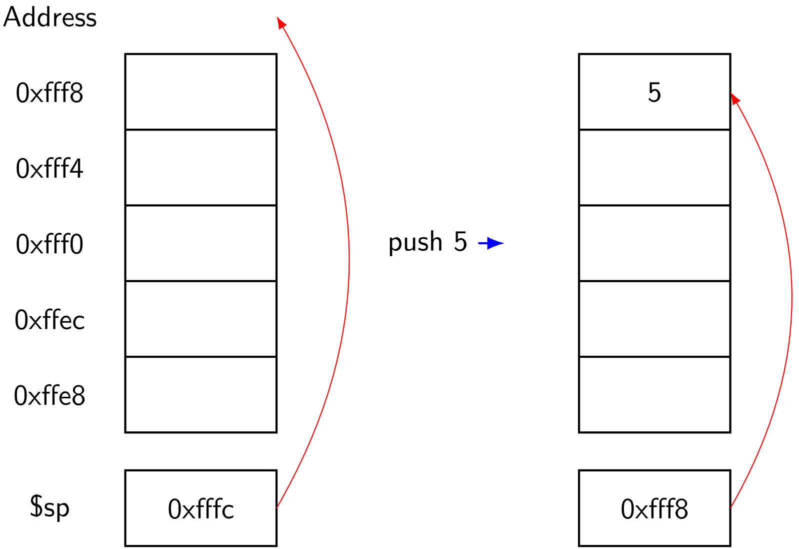
- memory is just one huge linear address space
- but a computer system reserves portions of it for particular purposes
- every computer system has a region of memory for the stack
- in some CPUs the stack is at a fixed location
- in most CPUs the compiler decides where the stack will be

- MIPS has a dedicated register that always knows where the stack is: $sp the stack pointer

Stack Implementation

- the stack pointer always points to the last (most recent) value pushed onto the
- in the beginning, the stack is empty, so the stack pointer doesn't point to any value
- by historical convention, the stack is at a very high memory address, and grows "downward" toward smaller addresses

Push


Address
0xfff8
0xfff4
0xfff0
0xffec
0xffe8
$sp
0xfffc
push 5
5
0xfff8

Push and Pop

- the steps in push are ($xx stands for some register):

```
addiu   $sp, $sp, -4      # decrement sp by 4
sw      $xx, 0($sp)       # copy register to *sp
```

- the steps in pop are:

```
lw      $xx, 0($sp)       # copy *sp into register
addiu   $sp, $sp, 4       # increment sp by 4
```

Stack Pointer

- the “top” of the stack is where the stack pointer is currently pointing
- as more values are pushed, the stack “grows” downward, toward lower addresses
- each push decrements the stack pointer
- thus the “top” of the stack is the lowest current stack address

Exercise

- starting with an empty stack, and `$sp = 0xfffc`, diagram the state of the stack, and the value of $sp, after the following operations:
    1. push 5
    2. push 10
    3. push 8
    4. pop
    5. push 3
    6. pop
    7. pop
    8. push 2

Register Conventions

- consider the following code:

```
    lw     $t2, 0($s4)      # fetch a into t2
    jal    sum              # compute v0 = b + c
    add    $s6, $t2, $v0    # s6 = a + sum(b + c)
```

- all looks well, but what happens if sum clobbers $t2?

- in high level language, we go to significant lengths to avoid side effects
- that's why strtok has a bad reputation
- we need to come to some agreement on what we can count on and what we cannot

Register Conventions

- on green sheet panel 2, bottom
- column PRESERVED ACROSS A CALL?
- a little misleading: a column marked “Yes” is not automatically preserved during a procedure call
- rather, you the programmer must ensure that any register marked “Yes” has the same value at the end of a procedure call that it had at the beginning
- if not, very bad things will happen

- conversely, for any register marked “No”, you cannot depend on it having the same value it had after a procedure
- this means it does not need to be preserved during a procedure

Register Conventions

- thus we have two classes of registers:
  1. those we must preserve across procedure calls
  2. those we can clobber indiscriminately during a procedure call
- there are two ways to preserve a register:
  1. don't change it
  2. store it first, use it, then restore it to its original value
- how do we store and then restore a register?
- save it on the stack!

A Slightly More Complex Example

```c
int compute(int g, int h, int i, int j)
{
  int result;

  result = (g + h) - (i + j);
  return result;
}
```

```
compute:
    add    $t0, $a0, $a1      # t0 = g + h
    add    $t1, $a2, $a3      # t1 = i + j
    sub    $s0, $t0, $t1      # s0 = t0 - t1
    add    $v0, $s0, $zero    # copy s0 to v0
    jr     $ra                # return
```

A Slightly More Complex Example

```
compute:
   add    $t0, $a0, $a1     # t0 = g + h
   add    $t1, $a2, $a3     # t1 = i + j
   sub    $s0, $t0, $t1     # s0 = t0 - t1
   add    $v0, $s0, $zero   # copy s0 to v0
   jr     $ra               # return
```

- this works
- it clobbers t0 and t1, but that's ok: they're temporary registers
- they are not expected to maintain their values over a procedure call
- but what about s0? its value has changed
- this violates the convention
- we are required to save and restore s0 if we use it

A Slightly More Complex Example

```
compute:
   addiu  $sp, $sp, -4      # 1st half of push
   sw     $s0, 0($sp)       # push s0 onto the stack
   add    $t0, $a0, $a1     # t0 = g + h
   add    $t1, $a2, $a3     # t1 = i + j
   sub    $s0, $t0, $t1     # s0 = t0 - t1
   add    $v0, $s0, $zero   # copy s0 to v0
   lw     $s0, 0($sp)       # 1st half of pop
   addu   $sp, $sp, 4       # pop the stack
   jr     $ra               # return
```

Saving Registers

- this example saved and restored only one register
- that was all that was needed, because you only have to save a register that you modify
- what if we needed to save two registers, say, $s0 and $fp?

```
foo:
   addiu $sp, $sp, -8       # two pushes
   sw    $s0, 4($sp)        # push s0 onto the stack
   sw    $fp, 0($sp)        # push fp onto the stack
   ... body of the procedure
   lw    $fp, 0($sp)        # pop fp from the stack
   lw    $s0, 4($sp)        # pop s0 from the stack
   addiu $sp, $sp, 8        # two pops
   jr    $ra                # return
```

- because it's a stack, we restore in reverse order of save

Nested Procedures

- when procedure A is called from main via jal, main's instruction return address is stored in $ra
- if procedure A then calls another procedure B via jal, the original value of $ra is overwritten to be A's instruction return address
- when B finishes, control is returned to A, and all is well
- but when A finishes, $ra still contains A's return address, and there's no way to get back to main

- no worries; we know how to save and restore a register
- if a procedure A calls another procedure B, then A must save $ra prior to calling B, and restore $ra prior to exiting

Local Variables

- if a procedure needs a few local variables for temporarily storing intermediate results, it can just use the t registers
- they're available for scratch use, and don't have to be saved
- but what if a procedure needs more local variable space than will fit in the t registers?
- for example, consider a flush function:

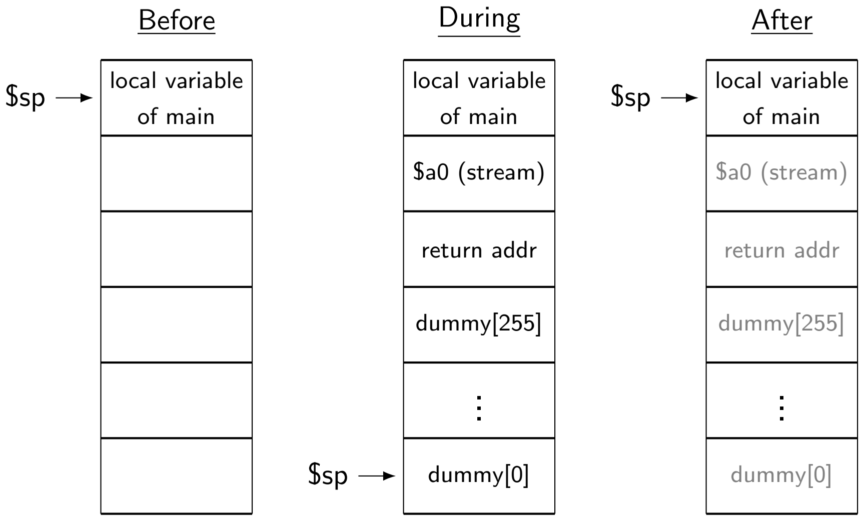
```
void flush_stream(FILE* stream)
{
  char dummy[255];

  do
  {
    fgets(dummy, 255, stream);
  } while(!strstr(dummy, "\n"));
}
```

Local Variables

- where do we put that local string?
- on the stack!
- the stack is where all local variables are stored
- this includes main's local variables — remember, main is a procedure

- the stack before, during, and after the call to flush_stream appears like this:

Local Variables



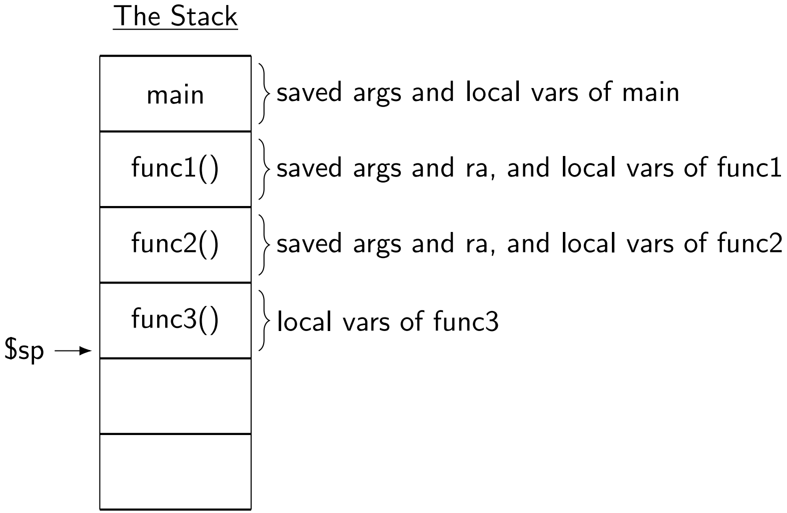
The Stack

- at the beginning of a procdure call, we set up a stack frame
- for the duration of the procedure call, everything is relative to the current value of the stack pointer
- at the end of the procedure call, we tear down the stack frame

- $sp has a value prior to the begining of the procedure
- at the end of the procedure, $sp must have the same value

- the portion of the stack for this procedure is called the activation record

Activation Records



The Stack
main
func1()
func2()
func3()
$sp
saved args and local vars of main
saved args and ra, and local vars of func1
saved args and ra, and local vars of func2
local vars of func3

Frame Pointer

- your text mentions the frame pointer $fp
- we will not use the frame pointer at all
- it is mostly used for locally scoped variables, which C rarely uses
- heavily used by C++
- all of our stack use and manipulation will strictly involve the stack pointer $sp

Global Variables

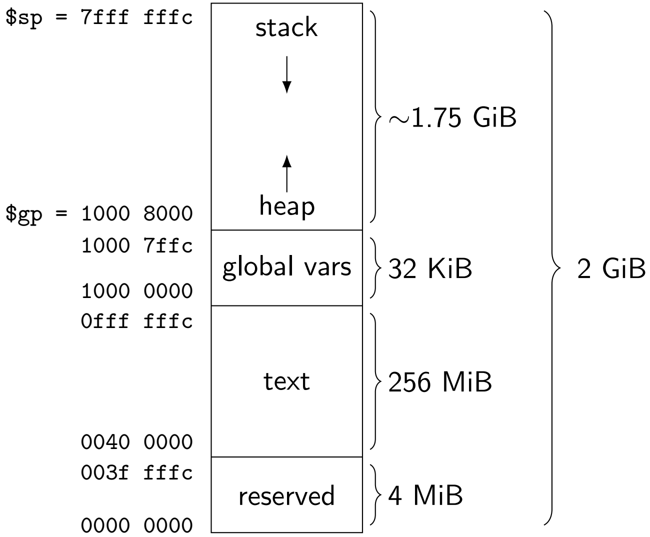
- local variables are always allocated on the stack
- a program can have global variables
- where are they?

- MIPS has the global pointer register $gp
- specifically to point to the area of memory where global variables are stored
- $gp never changes
- available to all functions

Dynamically Allocated Variables

- when you use malloc in C, or new in C++
- you allocate memory that does not go away when a function ends
- where is that?
- we have said that it's in the heap, but where is the heap?
- there's no heap pointer

Memory


$sp = 7fff fffc
$gp = 1000 8000
1000 7ffc
1000 0000
0fff fffc
0040 0000
003f fffc
0000 0000
stack
heap
global vars
text
reserved
∼1.75 GiB
32 KiB
256 MiB
4 MiB
2 GiB

Memory Notes

- the address space of MIPS is 4 GiB
- the top ½ (and bottom 4 MiB) is reserved for the OS, memory-mapped IO, etc
- the remaining $\sim$ 2 GiB is available for you the programmer

- text is the historical name for the machine code of the program

- in general, $gp never changes, while $sp moves up and down as the program progresses

- if either the stack or the heap grows too large and runs into the other, the program crashes
- if the stack grows too large, you get a stack overflow