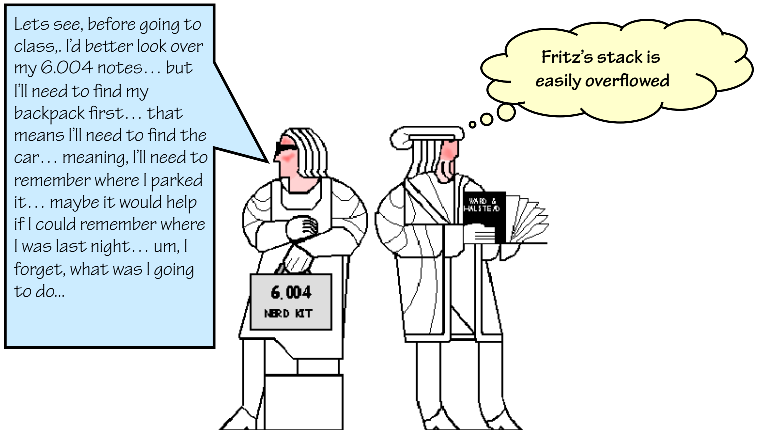
6.004 Computation Structures
Spring 2009

Stacks and Procedures



Lab 4 due tonight!

Where we left things last week…

```
int fact(int n)
{
    int r = 1;
    while (n>0) {
        r = r*n;
        n = n-1;
    }
    return r;
}

fact(4);
```

Procedures & Functions

- Reusable code fragments that are called as needed
- Single "named" entry point
- Parameterizable
- Local state (variables)
- Upon completion control is transferred back to caller

Procedure Linkage: First Try

```
int fact(int n)
{
    if (n>0)
        return n*fact(n-1);
    else
        return 1;
}

fact(4);
```

Oh no, not recursion! Didn't we get enough of that in 6.001?

fact(4) = 4*fact(3)
fact(3) = 3*fact(2)
fact(2) = 2*fact(1)
fact(1) = 1*fact(0)
fact(0) = 1

Proposed convention:
- pass arg in R1
- pass return addr in R28
- return result in R0
- questions:
  - nargs > 1?
  - preserve regs?

Let's just use some registers. We've got plenty…

Procedure Linkage: First Try

```
int fact(int n)
{
    if (n>0)
        return n*fact(n-1);
    else
        return 1;
}

fact(3);
```

```
fact:
        CMPLEC(r1,0,r0)
        BT(r0,else)
        MOVE(r1,r2)     | save n
        SUBC(r2,1,r1)
        BR(fact,r28)
        MUL(r0,r2,r0)
        BR(rtn)
else:   CMOVE(1,r0)
rtn:    JMP(r28,r31)

main:   CMOVE(3,r1)
        BR(fact,r28)
        HALT()
```

OOPS!

Proposed convention:
- pass arg in R1
- pass return addr in R28
- return result in R0
- questions:
  - nargs > 1?
  - preserve regs?

Need: O(n) storage locations!

Revisiting Procedure's Storage Needs

Basic Overhead for Procedures/Functions:
- Arguments
  f(x,y,z) or perhaps... sin(a+b)
- Return Address back to caller
- Results to be passed back to caller.

In C it's the caller's job to evaluate its arguments as expressions, and pass their resulting *values* to the callee... Thus, a variable name is just a simple case of an expression.

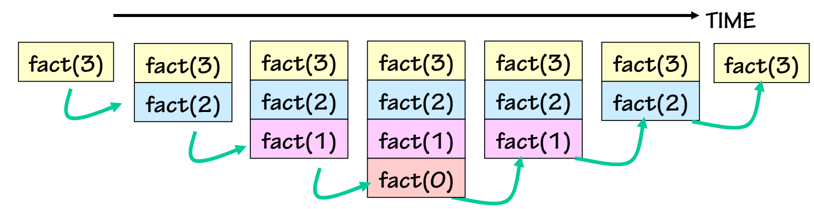
Temporary Storage:
intermediate results during expression evaluation.
(a+b)*(c+d)

Local variables:
```
{ int x, y;
  ... x ... y ...;
}
```

Each of these is specific to a particular *activation* of a procedure; collectively, they may be viewed as the procedure's *activation record.*

Lives of Activation Records

```
int fact(int n)
 { if (n > 0) return n*fact(n-1);
   else return 1;
 }
```



A procedure call creates a new activation record. Caller's record is preserved because we'll need it when call finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

Insight (ca. 1960): We need a STACK!

Suppose we allocated a SCRATCH memory for holding temporary variables. We'd like for this memory to grow and shrink as needed. And, we'd like it to have an easy management policy.

One possibility is a

STACK

A last-in-first-out (LIFO) data structure.

Figure by MIT OpenCourseWare.

Some interesting properties of stacks:

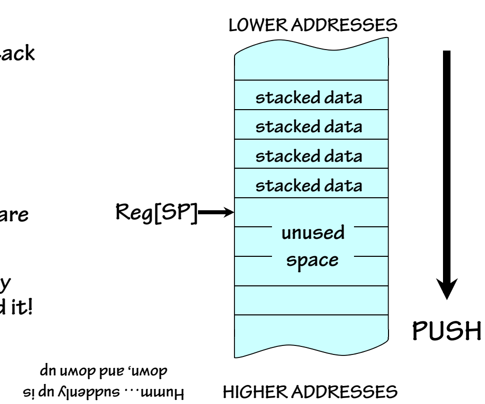
- Low overhead: Allocation, deallocation by simply adjusting a pointer.
- Basic PUSH, POP discipline: strong constraint on deallocation order.
- Discipline matches procedure call/return, block entry/exit, interrupts, etc.

Stack Implementation

CONVENTIONS:

- Dedicate a register for the Stack Pointer (SP), R29.
- Builds UP (towards higher addresses) on push
- SP points to first UNUSED location; locations below SP are allocated (protected).
- Discipline: can use stack *at any time;* but leave it as you found it!
- Reserve a block of memory well away from our program and its data

We use only *software conventions* to implement our stack (many architectures dedicate hardware)



Other possible implementations include stacks that grow "down", SP points to top of stack, etc.

Stack Management Macros

PUSH(RX): push Reg[x] onto stack
Reg[SP] = Reg[SP] + 4;
Mem[Reg[SP]-4] = Reg[x]

```
ADDC(R29, 4, R29)
ST(RX,-4,R29)
```

POP(RX): pop the value on the top of the stack into Reg[x]
Reg[x] = Mem[Reg[SP]-4]
Reg[SP] = Reg[SP] - 4;

```
LD(R29, -4, RX)
ADDC(R29,-4,R29)
```

Why?

ALLOCATE(k): reserve k WORDS of stack
Reg[SP] = Reg[SP] + 4*k

```
ADDC(R29,4*k,R29)
```

DEALLOCATE(k): release k WORDS of stack
Reg[SP] = Reg[SP] - 4*k

```
SUBC(R29,4*k,R29)
```


Fun with Stacks

We can squirrel away variables for later. For instance, the following code fragment can be inserted anywhere within a program.

```
           |
           | Argh!!! I'm out of registers Scotty!!
           |
           PUSH(R0)            | Frees up R0
           PUSH(R1)            | Frees up R1
           LD(R31,dilithum_xtals, R0)
           LD(R31,seconds_til_explosion, R1)
suspense:  SUBC(R1, 1, R1)
           BNE(R1, suspense, R31)
           ST(R0, warp_engines,R31)
           POP(R1)             | Restores R1
           POP(R0)             | Restores R0
```

Data is popped off the stack in the opposite order that it is pushed on

AND Stacks can also be used to solve other problems...

Solving Procedure Linkage "Problems"

A reminder of our storage needs:

1) We need a way to *pass arguments* into procedures
2) Procedures need their own *LOCAL variables*
3) Procedures need to *call other procedures*
4) Procedures might *call themselves* (Recursion)

BUT FIRST, WE'LL COMMIT SOME MORE REGISTERS:

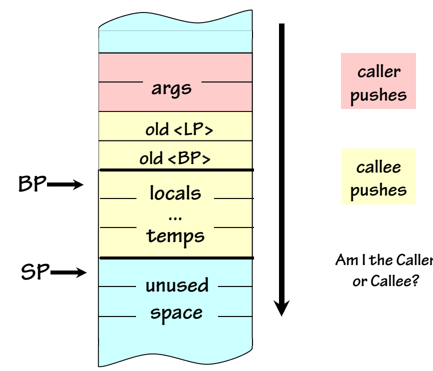
r27 = BP. Base ptr, points into stack to the local variables of callee
r28 = LP. Linkage ptr, return address to caller
r29 = SP. Stack ptr, points to 1st unused word

PLAN: CALLER puts args on stack, calls via something like
BR(CALLEE, LP)
leaving return address in LP.

"Stack frames" as activation records

The CALLEE will use the stack for all of the following storage needs:

1. saving the RETURN ADDRESS back to the caller
2. saving the CALLER's base ptr
3. Creating its own local/temp variables



In theory it's possible to use SP to access stack frame, but offsets will change due to PUSHs and POPs. For convenience we use BP so we can use constant offsets to find, e.g., the first argument.

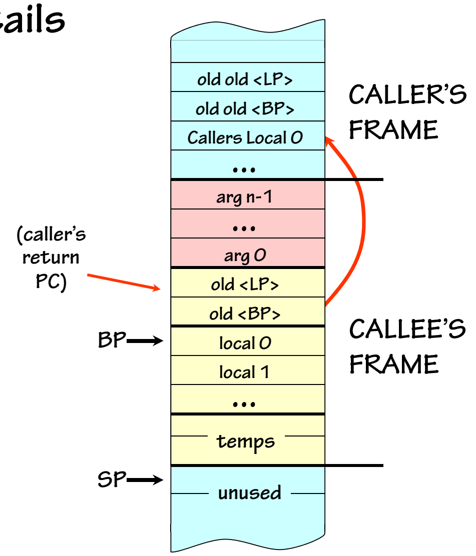
Stack Frame Details

The CALLER passes arguments to the CALLEE on the stack in REVERSE order

F(1,2,3,4) is translated to:

```
ADDC(R31,4,R0)
PUSH(R0)
ADDC(R31,3,R0)
PUSH(R0)
ADDC(R31,2,R0)
PUSH(R0)
ADDC(R31,1,R0)
PUSH(R0)
BEQ(R31, F, LP)
```

Why push args in REVERSE order???



Order of Arguments

Why push args onto the stack in *reverse order?*

1) It allows the BP to serve double duties when accessing the local frame

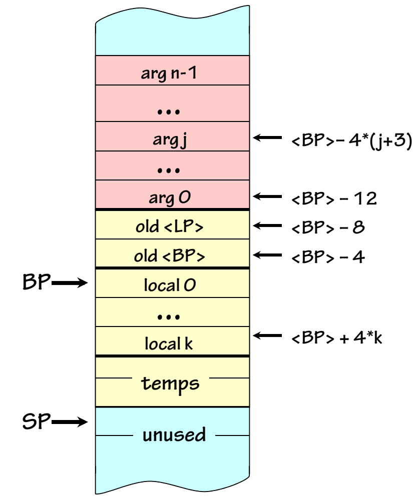
To access k^{th} local variable ($k \geq 0$)

```
LD(BP, k*4, rx)
  or
ST(rx, k*4, BP)
```

To access j^{th} argument ($j \geq 0$):

```
LD(BP, -4*(j+3), rx)
  or
ST(rx, -4*(j+3), BP)
```

2) The CALLEE can access the first few arguments without knowing how many arguments have been passed!



Procedure Linkage: The Contract

The CALLER will:

- Push args onto stack, in reverse order.
- Branch to callee, putting return address into LP.
- Remove args from stack on return.

The CALLEE will:

- Perform promised computation, leaving result in R0.
- Branch to return address.
- Leave stacked data intact, including stacked args.
- Leave regs (except R0) unchanged.

Procedure Linkage

typical "boilerplate" templates

Calling Sequence

```
PUSH(arg_n)           | push args, last arg first
...
PUSH(arg_1)
BEQ(R31,f, LP)        | Call f.
DEALLOCATE(n)         | Clean up!
...                   | (f's return value in r0)
```

Entry Sequence

```
f:  PUSH(LP)              | Save LP and BP
    PUSH(BP)              | in case we make new calls.
    MOVE(SP,BP)           | set BP=frame base
    ALLOCATE(nlocals)     | allocate locals
    (push other regs)     | preserve any regs used
```

Where's the Deallocate?

Return Sequence

```
    (pop other regs)      | restore regs
    MOVE(val, R0)         | set return value
    MOVE(BP,SP)           | strip locals, etc
    POP(BP)               | restore CALLER's linkage
    POP(LP)               | (the return address)
    JMP(LP,R31)           | return.
```

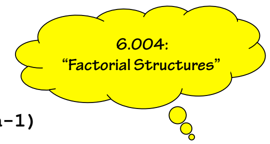
Our favorite procedure…

```
fact:  PUSH(LP)            | save linkages
       PUSH(BP)
       MOVE(SP,BP)         | new frame base
       PUSH(r1)            | preserve regs
       LD(BP,-12,r1)       | r1 ← n
       BNE(r1,big)         | if (n != 0)
       ADDC(r31,1,r0)      | else return 1;
       BR(rtn)

big:   SUBC(r1,1,r1)       | r1 ← (n-1)
       PUSH(r1)            | push arg1
       BR(fact,LP)         | fact(n-1)
       DEALLOCATE(1)       | pop arg1
       LD(BP,-12,r1)       | r0 ← n
       MUL(r1,r0,r0)       | r0 ← n*fact(n-1)

rtn:   POP(r1)             | restore regs
       MOVE(BP,SP)         | Why?
       POP(BP)             | restore links
       POP(LP)
       JMP(LP,R31)         | return.
```

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



Recursion?

But of course!

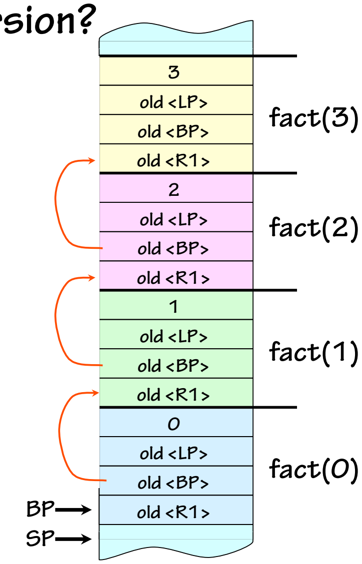
- Frames allocated for each recursive call...
- De-allocated (in inverse order) as recursive calls return.

Debugging skill:

"stack crawling"

- Given code, stack snapshot – figure out what, where, how, who...
- Follow old <BP> links to parse frames
- Decode args, locals, return locations, etc etc etc

Particularly useful on 6.004 quizzes!

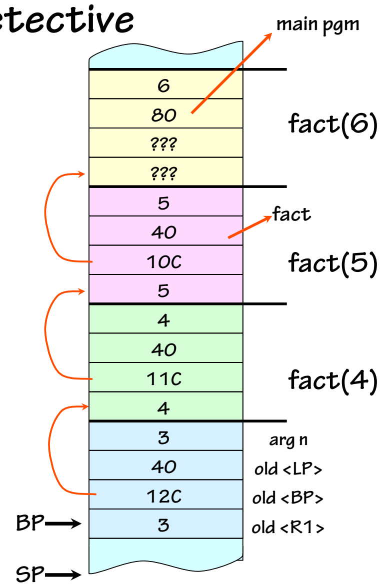


Stack Detective

fact(n) is called. During the calculation, the computer is stopped with the PC at **0x40**; the stack contents are shown (in **hex**).

- What's the argument to the *most recent* call to fact? 3
- What's the argument to the *original* call to fact? 6
- What's the location of the original calling (BR) instruction? 80 – 4 = 7C
- What instruction is about to be executed? DEALLOCATE(1)
- What value is in BP? 13C
- What value is in SP? 13C+4+4=144
- What value is in R0? fact(2) = 2
- What follows the call to fact(n)?

  another call to fact. Its the only program these guys have.



Man vs. Machine

Here's a C program which was fed to the C compiler*. Can you generate code as good as it did?

```
int ack(int i, int j)
{
  if (i == 0) return j+j;
  if (j == 0) return i+1;
  return ack(i-1, ack(i, j-1));
}
```

* GCC Port courtesy of Cotton Seed, Pat LoPresti, & Mitch Berger; available on Athena:

```
Athena% attach 6.004
Athena% gcc-beta -S -O2 file.c
```

Tough Problems

1. NON-LOCAL variable access, particularly in nested procedure *definitions.*

"FUNarg" problem of LISP:

```
(((lambda (x)
    (lambda(y)(+ x y)))
  3)
 4)
```

Python:

```
def f(x):
    def g(y): return x+y
    return g
z = f(3)(4)
```

```
f(int x)
 { int g(int y)
     { return x+y;
   return g;
 }

z = f(3)(4);
```

Conventional solutions:

- Environments, closures.
- "static links" in stack frames, pointing to frames of statically enclosing blocks. This allows a run-time discipline which correctly accesses variables in enclosing blocks.

  BUT… enclosing block may no longer exist (as above!).

(C avoids this problem by outlawing nested procedure declarations!)

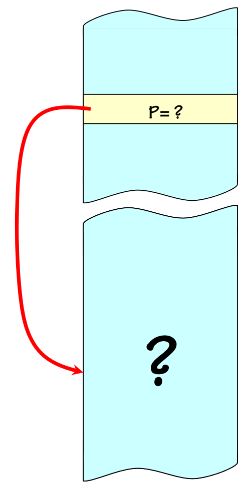
2. "Dangling References" - - -

Dangling References

```
int *p; /* a pointer */

int h(x)
{
   int y = x*3;
   p = &y;
   return 37;
}

h(10);
print(*p);
```



What do we expect???

Randomness. Crashes. Smoke. Obscenities. Furious calls to Redmond, WA.

Dangling References:

different strokes…

C and C++: real tools, real dangers.

"You get what you deserve".

Java / Scheme / Python / ...: kiddie scissors only.

- No "ADDRESS OF" operator: language restrictions *forbid* constructs which could lead to dangling references.
- Automatic storage management: garbage collectors, reference counting: local variables allocated from a "heap" rather than a stack.

"Safety" as a language/runtime property: guarantees against stray reads, writes.

- Tension: (manual) algorithm-specific optimization opportunites vs. simple, uniform, non-optimal storage management
- Tough language/compiler problem: abstractions, compiler technology that provides simple safety yet exploits efficiency of stack allocation.

Next Time: Building a Beta

Figure MIT OpenCourseWare.

```
ack:    PUSH (LP)
        PUSH (BP)
        MOVE (SP, BP)
        PUSH (R1)
        PUSH (R2)
        LD (BP, -12, R2)
        LD (BP, -16, R0)
_L4:    SHLC (R0, 1, R1)
        BEQ (R2, _L1)
        ADDC (R2, 1, R1)
        BEQ (R0, _L1)
        SUBC (R2, 1, R1)
        SUBC (R0, 1, R0)
        PUSH (R0)
        PUSH (R2)
        BR (ack, LP)
        DEALLOCATE (2)
        MOVE (R1, R2)
        BR (_L4)
_L1:    MOVE (R1, R0)
        POP (R2)
        POP (R1)
        POP (BP)
        POP (LP)
        JMP (LP)
```