Calling Conventions

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

See: P&H 2.8, 2.12

calc.c

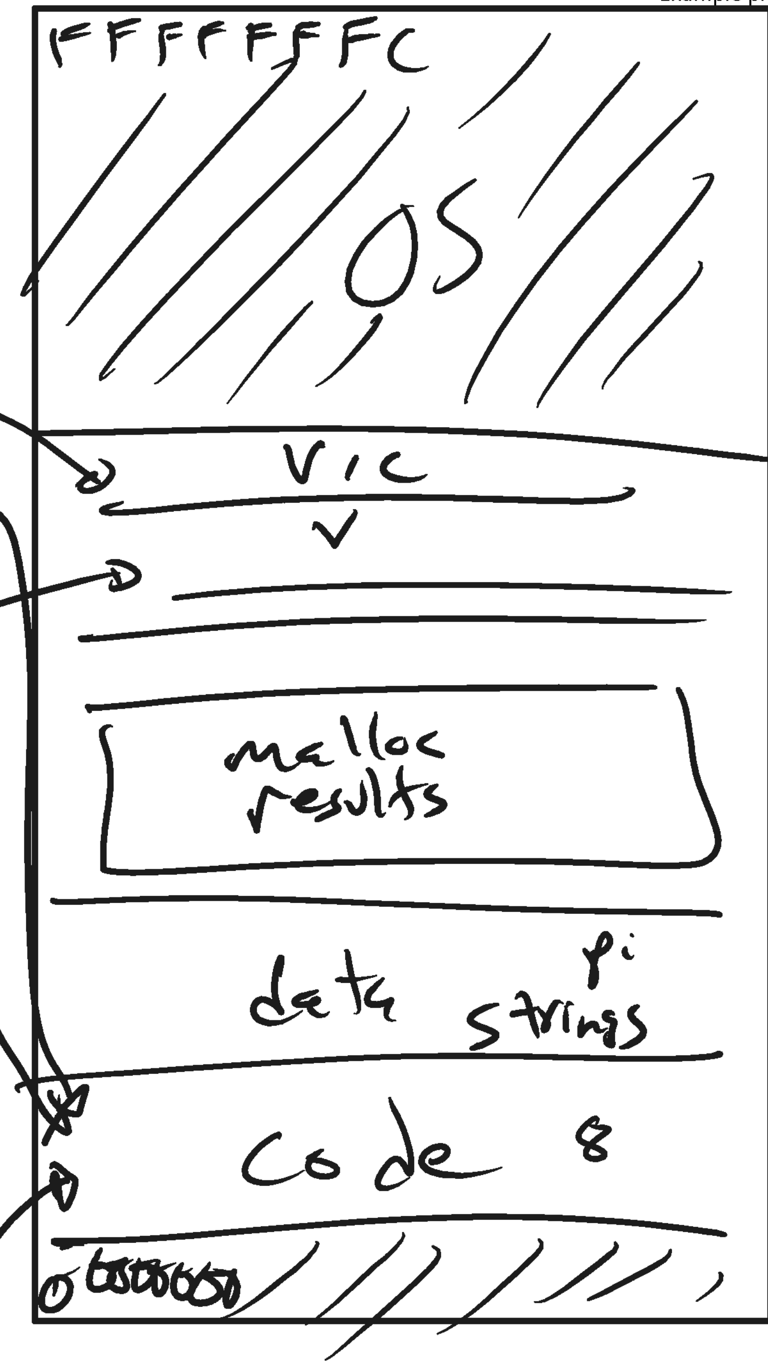
```
vector v = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int c = pi + tnorm(v);
print("result", c);
```

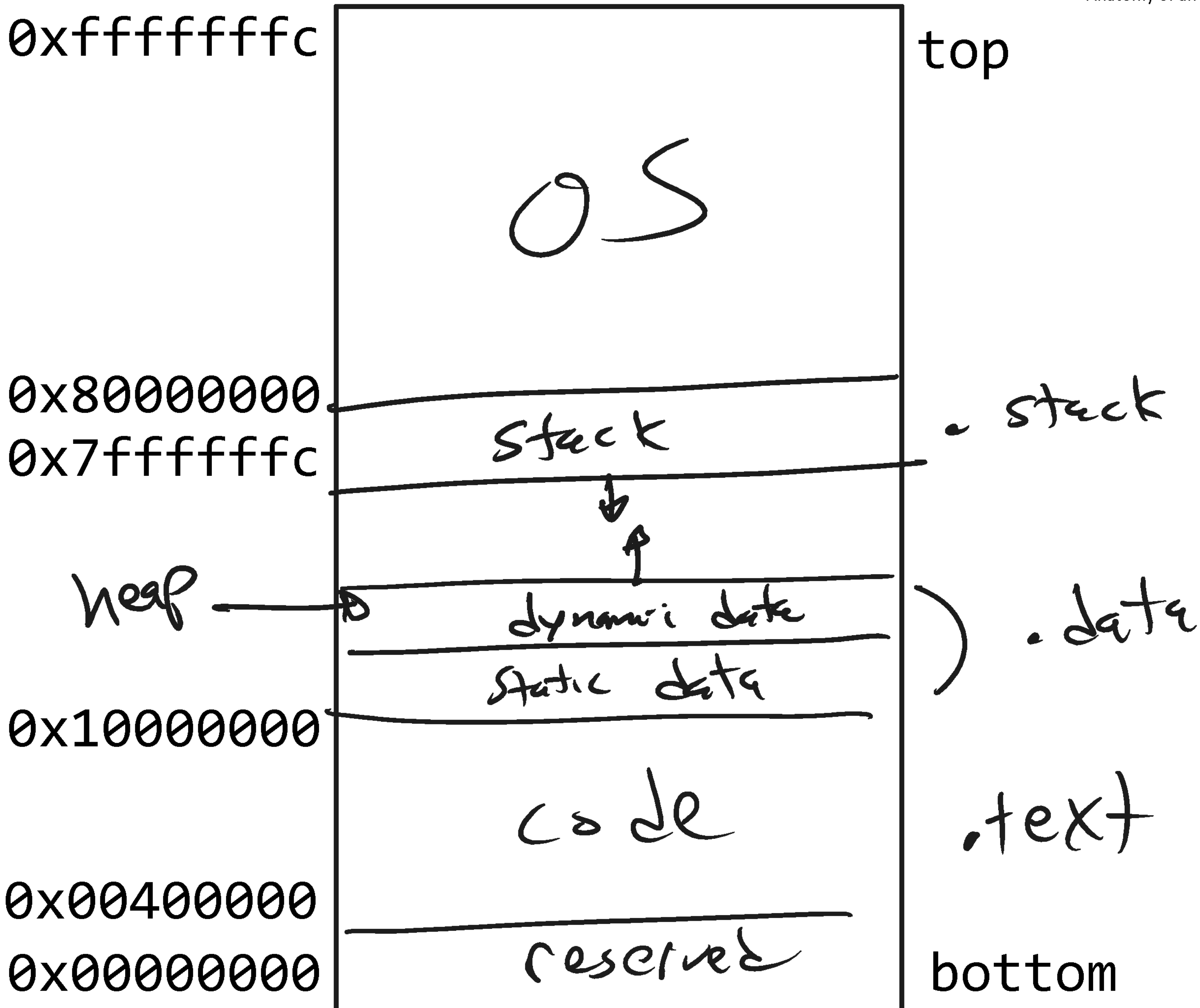
math.c

```
int tnorm(vector v) {
  return abs(v->x)+abs(v->y);
}
```

lib3410.o

```
global variable: pi
entry point: prompt
entry point: print
entry point: malloc
```




0xfffffffc
top
OS
0x80000000
stack
0x7ffffffc
. stack
heap
dynamic data
static data
. data
0x10000000
code
. text
0x00400000
reserved
0x00000000
bottom

math.c

```
int abs(x) {
  return x < 0 ? -x : x;
}
int tnorm(vector v) {
 return abs(v->x)+abs(v->y);
}
```

abs:

arg in r3, return address in r31

leaves result in r3

```
BLZ r3, neg
J r31

neg:
  SUB r3, r0, r3
  J r31
```

.global tnorm

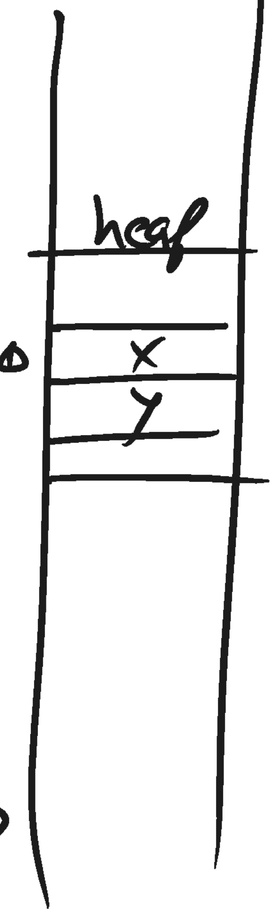
tnorm:

arg in r4, return address in r31

leaves result in r4

```
MOVE r30, r31
LW r3, 0(r4)
JAL abs
~~Move r6, r3~~
LW r6, 4(r4)
JAL abs

ADD r4, r6, r3
JR r30
```



calc.c

```
vector v = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int c = pi + tnorm(v);
print("result", c);
```

```
.data
str1: .asciiz "enter x"
str2: .asciiz "enter y"
str3: .asciiz "result"
.text
   .extern prompt
   .extern print
   .extern malloc
   .extern tnorm
   .global dostuff
```

```
dostuff:
# no args, no return value, return addr in r31
MOVE r30, r31
LI r3, 8          # call malloc: arg in r3, ret in r3
JAL malloc
MOVE r6, r3 # r6 now holds v
LA r3, str1       # call prompt: arg in r3, ret in r3
JAL prompt
SW r3, 0(r6)
LA r3, str2       # call prompt: arg in r3, ret in r3
JAL prompt
SW r3, 4(r6)
MOVE r4, r6 # call tnorm: arg in r4, ret in r4
JAL tnorm
LA r5, pi
LW r5, 0(r5)
ADD r5, r4, r5
LA r3, str3       # call print: args in r3 and r4
MOVE r4, r5
JAL print
JR r30
```

r6 clobbered, r30, r3

Calling Conventions

- where to put function arguments
- where to put return value
- who saves and restores registers, and how
- stack discipline

Why?

- Enable code re-use (e.g. functions, libraries)
- Reduce chance for mistakes

Warning: There is no one true MIPS calling convention.
lecture != book != gcc != spim != web

```
void main() {
  int x = ask(“x?”);
  int y = ask(“y?”);
  test(x, y);
}

void test(int x, int y) {
  int d = sqrt(x*x + y*y);
  if (d == 1)
     print(“unit”);
  return d;
}
```

r0	$zero	zero	r16		
r1	$at	assembler temp	r17		
r2	$v0	function return values	r18		
r3	$v1		r19		
r4	$a0	function arguments	r20		
r5	$a1		r21		
r6	$a2		r22		
r7	$a3		r23		
r8			r24		
r9			r25		
r10			r26	$k0	reserved for OS kernel
r11			r27	$k1	
r12			r28		
r13			r29		
r14			r30		
r15			r31	$ra	return address

```
void main() {
  int x = ask("x?");
  int y = ask("y?");
  test(x, y);
}
```

```
main:

LA $a0, strX
JAL ask # result in $v0
MOVE r16, $v0
LA $a0, strY
JAL ask # result in $v0
MOVE r17, $v0

MOVE $a0, r16
MOVE $a1, r17

JAL test
```

Call stack

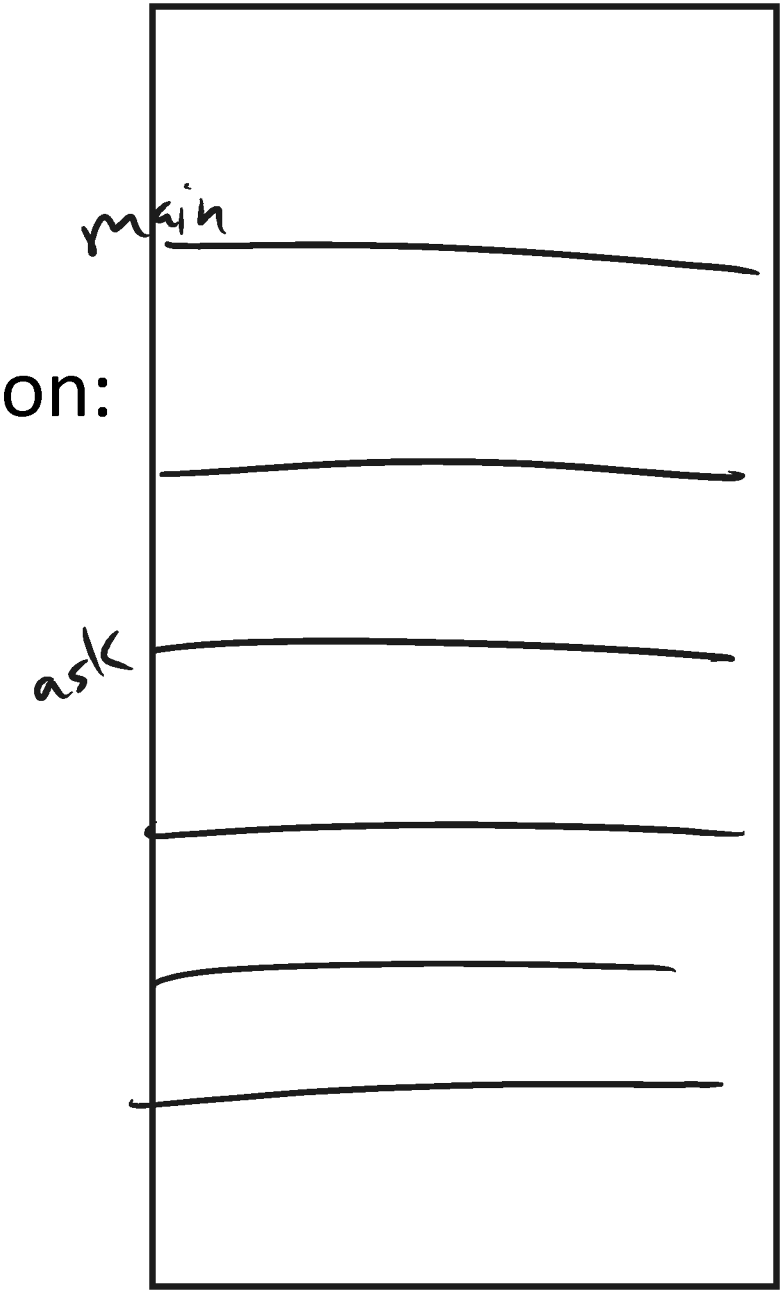
- contains *activation records* (aka *stack frames*)

One for each function invocation:

- saved return address
- local variables
- … and more

Simplification:

- frame size & layout decided at compile time for each function

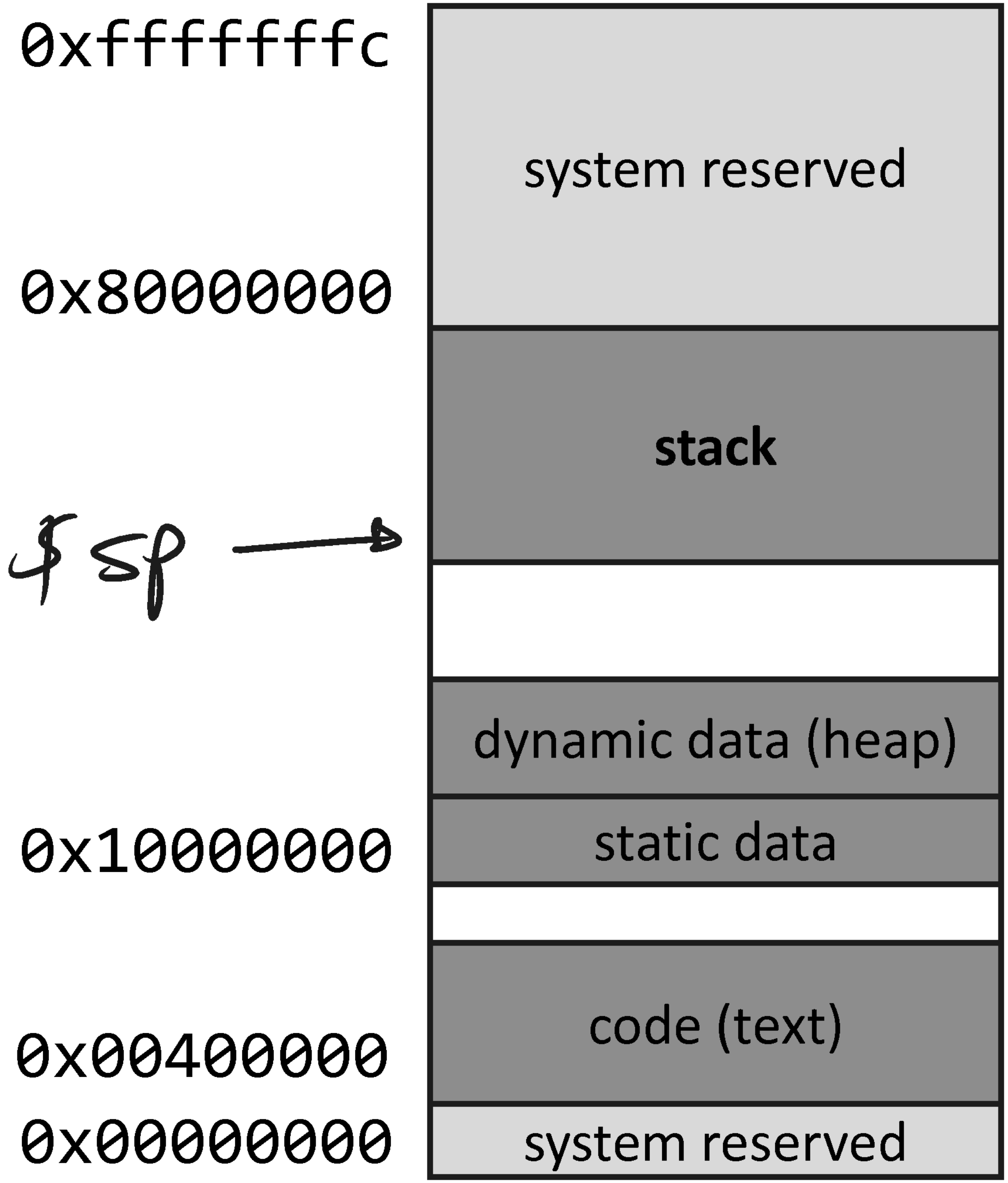


Convention:

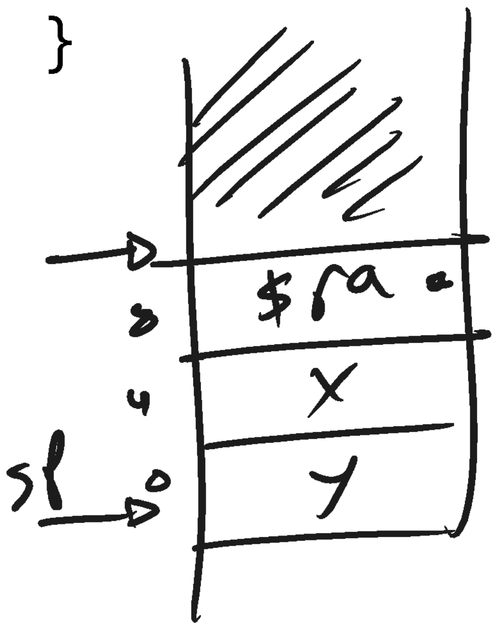
- r29 is $sp (bottom elt of call stack)

Stack grows **down**

Heap grows **up**



```
void main() {
  int x = ask("x?");
  int y = ask("y?");
  test(x, y);
}
```



main:

\# allocate frame

ADDUI $sp, $sp, -12 # $ra, x, y

\# save return address in frame

SW $ra, 8($sp)

sw $v0, 4($sp)

sw $v0, 0($sp)

\# restore return address

LW $ra, ~~4~~8($sp)

\# deallocate frame

ADDUI $sp, $sp, 12

JR $ra

Conventions so far:

- args passed in \$a0, \$a1, \$a2, \$a3
- return value (if any) in \$v0, \$v1
- stack frame at \$sp
  - contains \$ra (clobbered on JAL to sub-functions)
  - contains local vars (possibly clobbered by sub-functions)

Q: What about real argument lists?

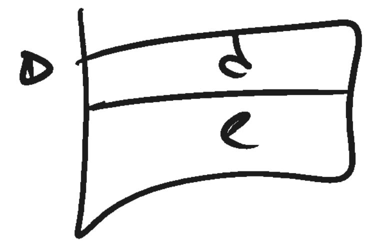
int min(int a, int b); $a0 $a1

int paint(char c, short d, struct point p); $a0 a1 a2 a3

int treesort(struct Tree *root, int[] A);

struct Tree *createTree(); 32 bits $a0, $a1

int max(int a, int b, int c, int d, int e);



Conventions: a, b, c,

- align everything to multiples of 4 bytes
- first 4 words in $a0...$a3, “spill” rest to stack

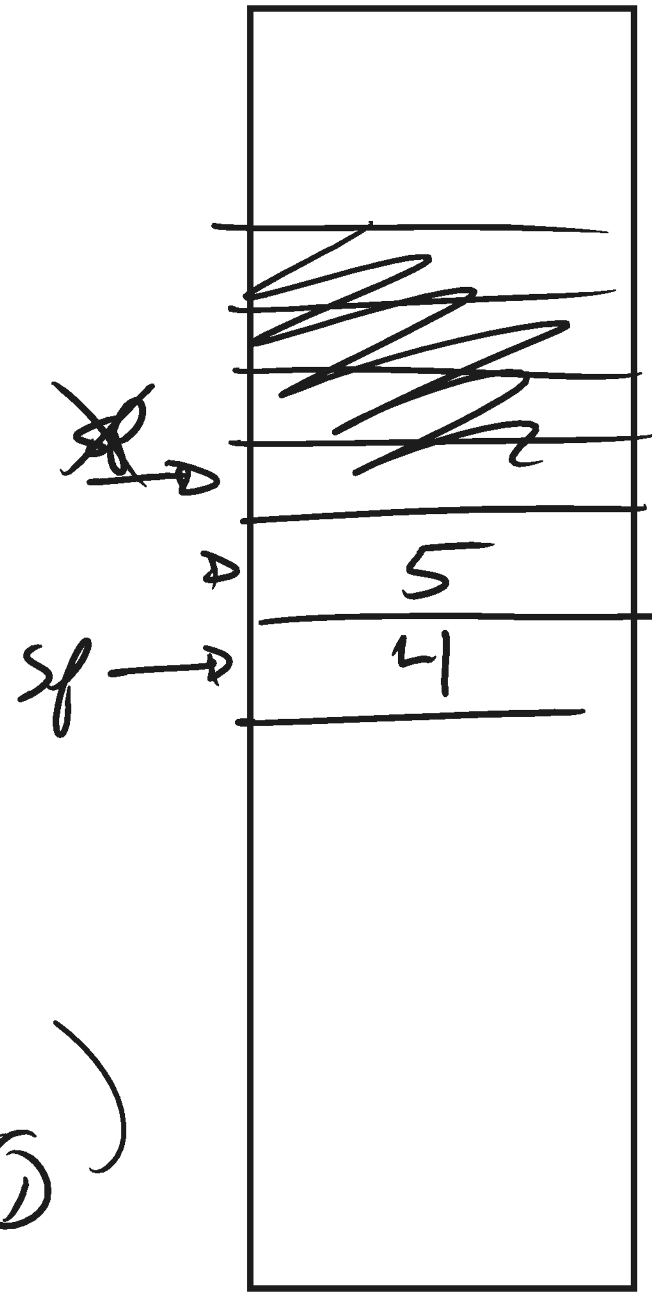
invoke sum(0, 1, 2, 3, 4, 5);

```
main:

...
LI $a0, 0
LI $a1, 1
LI $a2, 2
LI $a3, 3
ADDI $sp, $sp, -8
LI r8, 4
SW r8,   0($sp)
LI r8, 5
SW r8,   4($sp)
JAL sum
nop
ADDI $sp, $sp, 8
```

```
sum:

...
ADD $v0, $a0, $a1
ADD $v0, $v0, $a2
ADD $v0, $v0, $a3
LW $v1, 0($sp)
ADD $v0, $v0, $v1
LW $v1, 4($sp)
ADD $v0, $v0, $v1
...
JR $ra
```



printf(fmt, …)

```
main:
…
LI $a0, str0
LI $a1, 1
LI $a2, 2
LI $a3, 3
# 2 slots on stack
LI r8, 4
SW r8,   0($sp)
LI r8, 5
SW r8,   4($sp)
JAL sum
```

$a0 $a1 a2 a3

```
printf:
…
if (argno == 0)
    use $a0
else if (argno == 1)
    use $a1
else if (argno == 2)
    use $a2
else if (argno == 3)
    use $a3
else
    use $sp+4*argno
…
```

sw a0 sp

sw a1 sp+4

sw a2 sp+8

for(i

arg = M[sp+4·i]

worse?

~~Variable Length~~ Arguments

Initially confusing but ultimately simpler approach:

- Pass the first four arguments in registers, as usual
- Pass the rest on the stack (in order)
- Reserve space on the stack for all arguments, including the first four

Simplifies varargs functions

- Store a0-a3 in the slots allocated in parent's frame
- Refer to all arguments through the stack

Conventions so far:

- first four arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed on the stack
- return value (if any) in $v0, $v1
- stack frame at $sp
  - contains $ra (clobbered on JAL to sub-functions)
  - contains local vars (possibly clobbered by sub-functions)
  - contains extra arguments to sub-functions
  - contains **space** for first 4 arguments to sub-functions

init():	0x400000
printf(s, ...):	0x4002B4
vnorm(a,b):	0x40107C
main(a,b):	0x4010A0
pi:	0x10000000
str1:	0x10000004

CPU:
$pc=0x004003C0
$sp=0x7FFFFFAC
$ra=0x00401090

	0x00000000
	0x0040010c
	0x0040010a
	0x00000000
	0x00000000
	0x00000000
	0x00000000
main	0x004010c4
	0x00000000
	0x00000000
	0x00000015
0x7FFFFFB0	0x10000004
$p →	0x00401090

What func is running? printf.

Who called it? vnorm

Has it called anything? NO.

Will it? NO - No space for 4 args.

Args? str1, 15,

Stack depth? printf, vnorm, main, init

Call trace?

Frame pointer marks boundaries

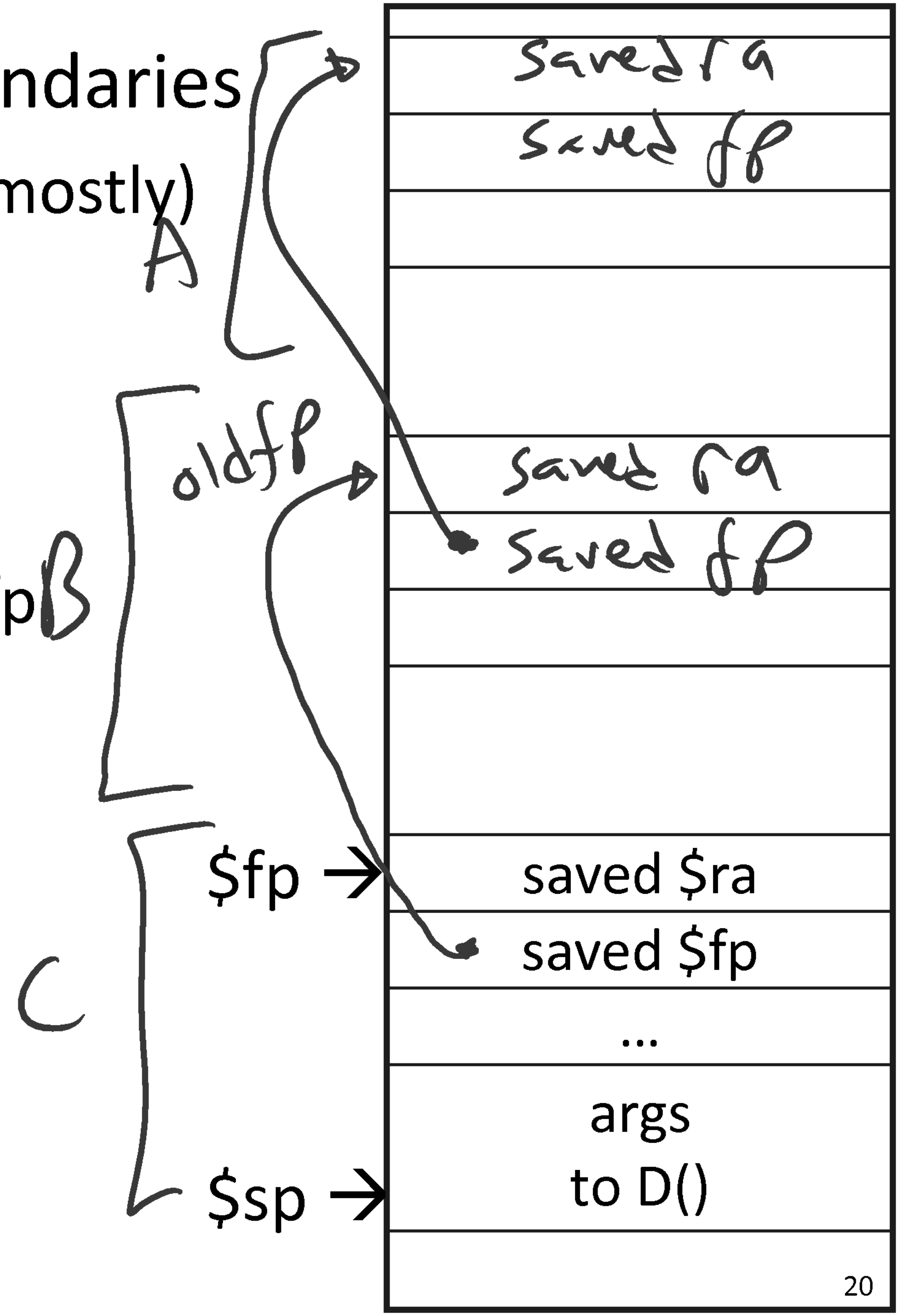
- Optional (for debugging, mostly)

Convention:

- r30 is $fp
  (top elt of current frame)
- Callee: always push old $fp on stack

E.g.:

A() called B()
B() called C()
C() about to call D()



r0	$zero	zero	r16		
r1	$at	assembler temp	r17		
r2	$v0	function return values	r18		
r3	$v1		r19		
r4	$a0	function arguments	r20		
r5	$a1		r21		
r6	$a2		r22		
r7	$a3		r23		
r8			r24		
r9			r25		
r10			r26	$k0	reserved for OS kernel
r11			r27	$k1	
r12			r28		
r13			r29	$sp	**stack pointer**
r14			r30	$fp	**frame pointer**
r15			r31	$ra	return address

How does a function load global data?

- global variables are just above 0x10000000

Convention: *global pointer*

- r28 is \$gp (pointer into *middle* of global data section)
  \$gp = 0x10008000
- Access most global data using LW at \$gp +/- offset
  LW \$v0, 0x8000(\$gp)
  LW \$v1, 0x7FFF(\$gp)

r0	$zero	zero	r16		
r1	$at	assembler temp	r17		
r2	$v0	function return values	r18		
r3	$v1		r19		
r4	$a0	function arguments	r20		
r5	$a1		r21		
r6	$a2		r22		
r7	$a3		r23		
r8			r24		
r9			r25		
r10			r26	$k0	reserved for OS kernel
r11			r27	$k1	
r12			r28	$gp	**global pointer**
r13			r29	$sp	stack pointer
r14			r30	$fp	frame pointer
r15			r31	$ra	return address

Q: Remainder of registers?

A: Any function can use for any purpose

- places to put extra local variables, local arrays, …
- places to put callee-save

```
int main() {
  int x = prompt("x?");
  int y = prompt("y?");
  int v = tnorm(x, y)
  printf("result is %d", v);
}
```

Callee-save: Always…

- save before modifying
- restore before returning

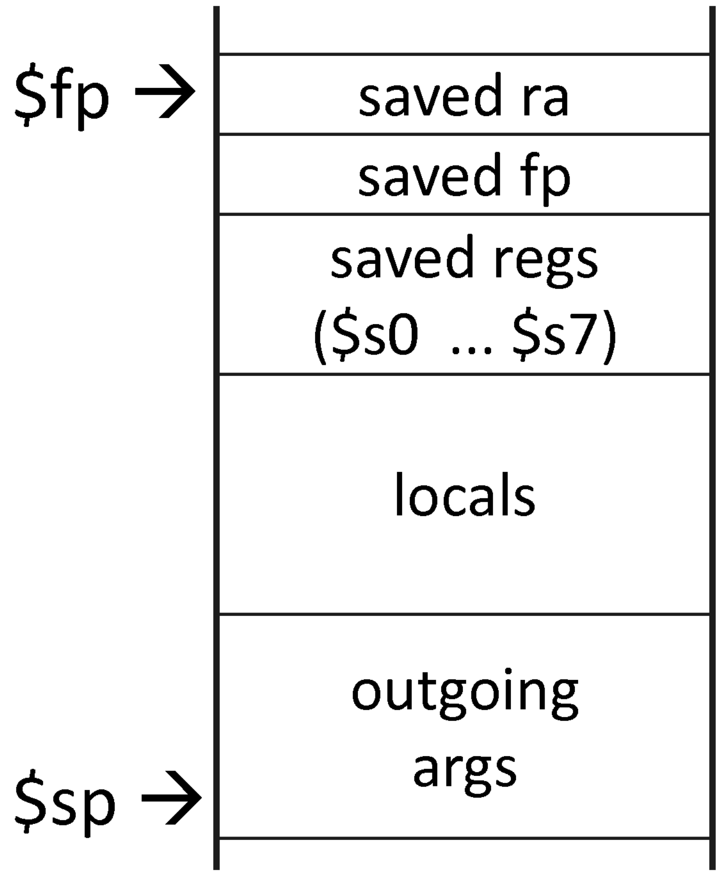
Caller-save: If necessary…

- save before calling anything
- restore after it returns

r0	$zero	zero	r16	$s0	
r1	$at	assembler temp	r17	$s1	
r2	$v0	function	r18	$s2	
r3	$v1	return values	r19	$s3	**saved**
r4	$a0		r20	$s4	**(callee save)**
r5	$a1	function	r21	$s5	
r6	$a2	arguments	r22	$s6	
r7	$a3		r23	$s7	
r8	$t0		r24	$t8	**more temps**
r9	$t1		r25	$t9	**(caller save)**
r10	$t2		r26	$k0	reserved for
r11	$t3	**temps**	r27	$k1	kernel
r12	$t4	**(caller save)**	r28	$gp	global data pointer
r13	$t5		r29	$sp	stack pointer
r14	$t6		r30	$fp	frame pointer
r15	$t7		r31	$ra	return address

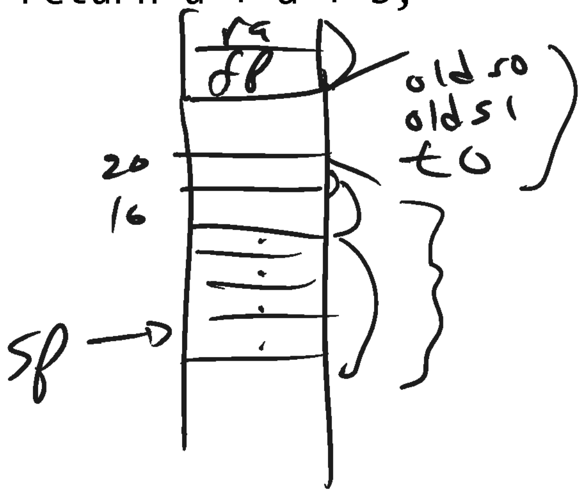
Conventions so far:

- first four arg words passed in $a0, $a1, $a2, $a3
- remaining arg words passed in parent's stack frame
- return value (if any) in $v0, $v1
- globals accessed via $gp
- callee save regs are preserved
- caller save regs are not



```
int test(int a, int b) {
    int tmp = (a&b)+(a|b);
    int s = sum(tmp,1,2,3,4,5);
    int u = sum(s,tmp,b,a,b,a);
    return u + a + b;
}
```

a0 a1



Prolog

MOVE s0, a0
MOVE s1, a1

AND t0, a0, a1

OR t1, a0, a1

ADD t0, t0, t1

MOVE a0, t0

LI a1, 1

LI a2, 2

LI a3, 3

LI t1, 4

SW t1, 16(sp)

LI t1, 5

SW t1, 20(sp)

SW t0, 24(sp)

JAL sum

NOP

LW t0, 24(sp)

MOVE a0, v0

MOVE a1, t0

a2 s1

a3 s0

SW s1, 16(sp)

SW s0, 20(sp)

JAL sum

NOP

u is in v0

ADD v0, v0, s0

ADD v0, v0, s1

EPILOG

test:

```
ADDIU sp, sp, -44
SW ra, 40(sp)
SW fp, 36(sp)
SW s0, 32(sp)
SW s1, 28(sp)
ADDIU fp, sp, 40
```

$\boxed{\text{BODY}}$

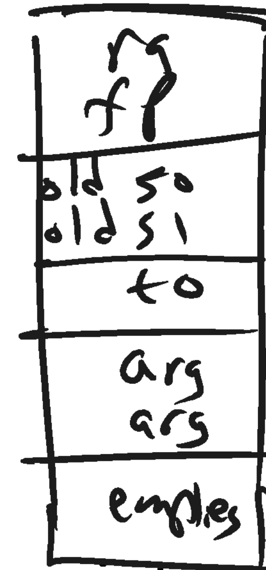
```
LW s1, 28(sp)
LW s0, 32(sp)
LW fp, 36(sp)
LW ra, 40(sp)
ADDIU sp, sp, +44
JR ra
NOP
```

uses…
allocate frame
save $ra
save old $fp
save …
save …
set new frame pointer

…

…

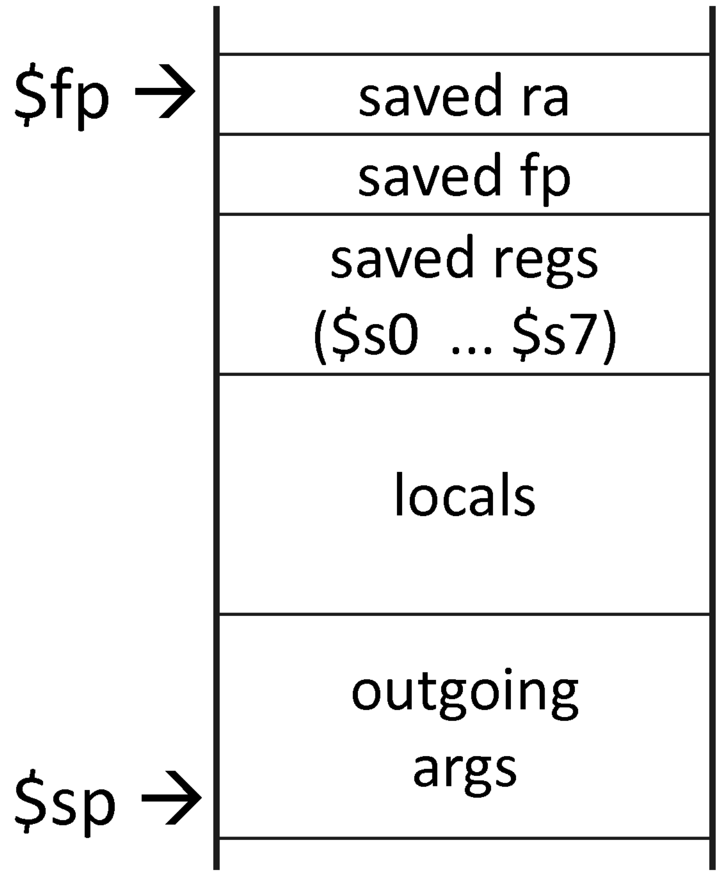
restore …
restore …
restore old $fp
restore $ra
dealloc frame



Minimum stack size for a standard function?

4×2 if it doesn't make any calls.

4×6 otherwise

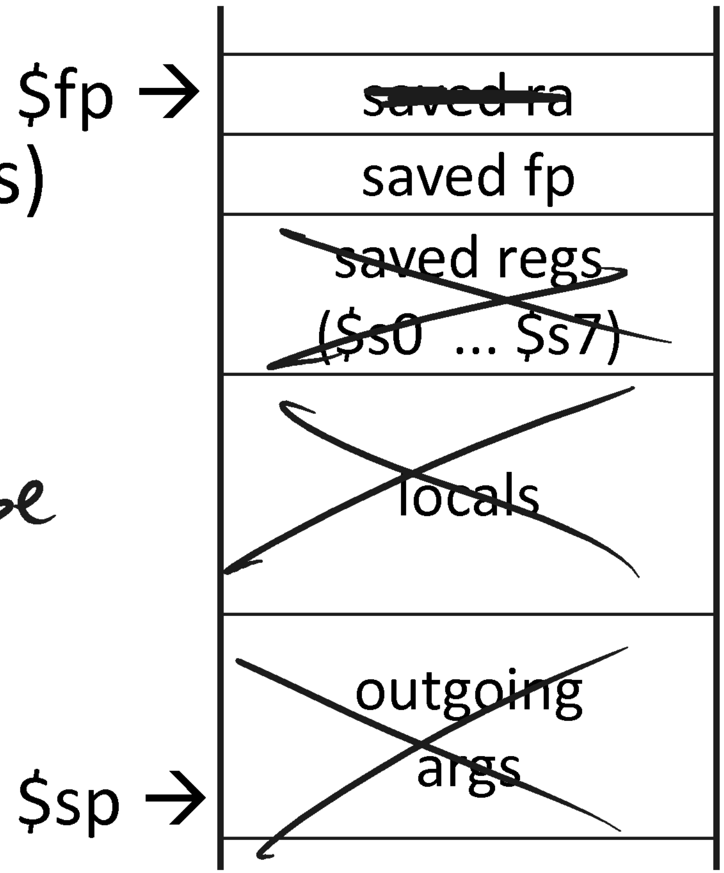


Leaf function does not invoke any other functions

int f(int x, int y) { return (x+y); }

Optimizations?

- No saved regs (or locals)
- No outgoing args
- Don't push $ra
- No frame at all? Maybe



Global variables in data segment

- Exist for all time, accessible to all routines

Dynamic variables in heap segment

- Exist between malloc() and free()

Local variables in stack frame

- Exist solely for the duration of the stack frame

Dangling pointers into freed heap mem are bad

Dangling pointers into old stack frames are bad

- C lets you create these, Java does not
- `int *foo() { int a; return &a; }`

```
#include <stdio.h>
int n = 100;
int main (int argc, char* argv[]) {
    int i, m = n, count = 0;
    for (i = 1; i <= m; i++) { count += i; }
    printf (“Sum 1 to %d is %d\n", n, count);
}
```

```
[csug01] mipsel-linux-gcc –S add1To100.c
```

```
        .data
        .globl  n
        .align  2
n:      .word   100
        .rdata
        .align  2
$str0:  .asciiz
           "Sum 1 to %d is %d\n"
        .text
        .align  2
        .globl  main
main:   addiu   $sp,$sp,-48
        sw      $31,44($sp)
        sw      $fp,40($sp)
        move    $fp,$sp
        sw      $4,48($fp)
        sw      $5,52($fp)
        la      $2,n
        lw      $2,0($2)
        sw      $2,28($fp)
        sw      $0,32($fp)
        li      $2,1
        sw      $2,24($fp)
```

```
$L2:    lw      $2,24($fp)
        lw      $3,28($fp)
        slt     $2,$3,$2
        bne     $2,$0,$L3
        lw      $3,32($fp)
        lw      $2,24($fp)
        addu    $2,$3,$2
        sw      $2,32($fp)
        lw      $2,24($fp)
        addiu   $2,$2,1
        sw      $2,24($fp)
        b       $L2
$L3:    la      $4,$str0
        lw      $5,28($fp)
        lw      $6,32($fp)
        jal     printf
        move    $sp,$fp
        lw      $31,44($sp)
        lw      $fp,40($sp)
        addiu   $sp,$sp,48
        j       $31
```

```
main: addiu $sp, $sp, -64
      sw $fp, 56($sp)
      addu $fp, $0, $sp
      sw $0, 48($fp)        s
      sw $0, 52($fp)        i
$L2:  lw $2, 52($fp)
      nop
      slt $4, $2, 7
      beq $4, $0, $L3
      nop
      sll $4, $2, 2
      …
      addiu $2, $2, 1
      sw $2, 52($fp)
      J $L2
```

```
int strlen(char *a) {
    int n;
    while (a[n] != 0) n++;
    return n;
}
strlen:       addiu   sp, sp, -16
              sw      fp, 8(sp)
              addiu   fp, sp, 16
              lw      v0, 0(fp)
              nop
strlen_top:           addu    t0, a0, v0
              lb      t1, 0(t0)
              nop
              beqz    t1, strlen_done
              nop
              addiu   v0, v0, 1
              b               strlen_top
              nop
strlen_done:lw        fp, 8(sp)
              addiu   sp, sp, 16
              jr      ra
              nop
```