ECE 486/586
Computer Architecture

Prof. Mark G. Faust

Maseeh College of Engineering
and Computer Science

PORTLAND STATE UNIVERSITY

Outline

- Stacks
- Calling conventions
  - Procedure calls (functions, subroutines)
  - Stack frames (activation records)
  - Passing arguments
  - Saving registers
  - Returning values
  - Local variables (automatic variables)
  - Leaf
  - Static variables
- Buffer overflow attacks
- Linking

Procedures

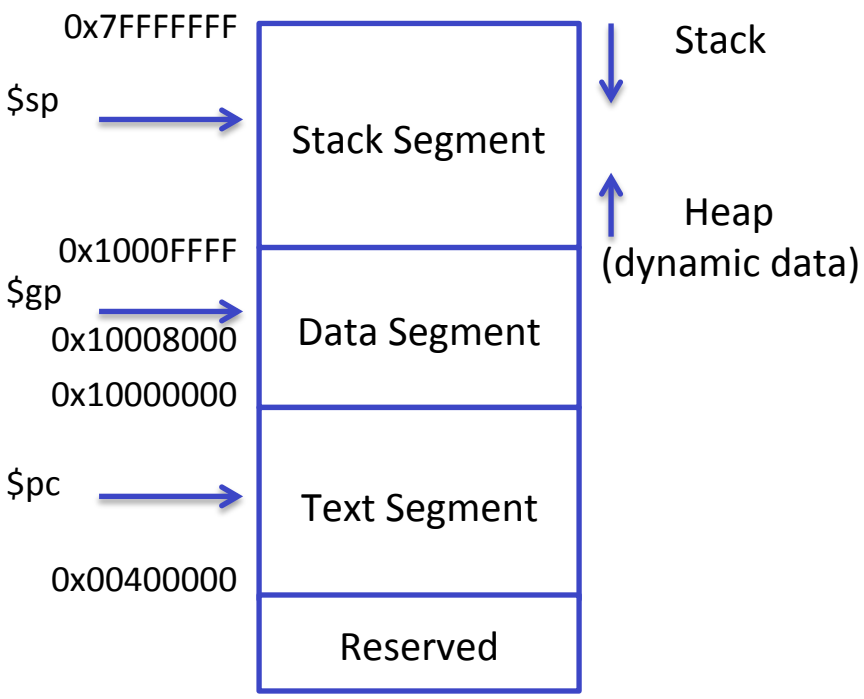
- Variously known as functions, subroutines
- Used to structure programs
  - Comprehension
  - Debugging
  - Reusability

Embark on mission with secret plan, acquires resources, performs task, covers his tracks, returns to point of origin with desired result

- Execution of a procedure requires
  1. Put parameters (arguments) where procedure can access them
  2. Transfer control to procedure
  3. Acquire storage resources needed for procedure
  4. Perform desired task
  5. Put result in place where caller can access it
  6. Return control to point of origin

     Procedure can be called from multiple places

MIPS Memory Layout



Address space above for kernel

Stack facilitates procedure calls
Heap used for dynamically allocated memory
malloc(), free()

Why are stack and heap arranged this way?

Data segment use for static storage
Global pointer ($gp) register points to middle
Why?
Hint: how big is the data segment?

Text segment contains code

Calling Conventions

- Determined by programming language or platform (architecture and operating system)
  - Allows separate compilation of modules and later linking
  - May permit linking of object code produced by different languages
  - Needed by debuggers
  - Must be observed by assembly language programmer
  - Observed in code generated by compilers targeting the platform
- Specify register and stack usage
- Specify how parameters are passed and values returned
- Can vary on same system
  - Multiple MIPS ABIs (Applications Binary Interface) {o32,n32,n64…}
  - MIPS C compiler vs. gcc (MIPS C compiler doesn't use $fp)
  - Compiler options (e.g. optimization) – may preclude debugging if info not available to debugger

MIPS Register Conventions

Generally pass arguments on stack, but first four via registers ($a0 - $a3). Why?
Some registers ($t0 - $t9) aren't expected by caller to be saved by callee. What should caller do?
Other registers (e.g. $s0 - $s7, $gp, $sp, $fp, $ra) must be saved by callee (if used by callee)
Return values via registers ($v0 - $v1)
Argument registers aren't saved so what does a procedure do if it calls another procedure?

MIPS `JAL` (jump-and-link) instruction puts return address in $ra
MIPS `JR` (jump register) instruction jumps to address in designated register

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

$at	$1	reserved for assembler
$k0	$26	reserved for OS
$k1	$27	reserved for OS

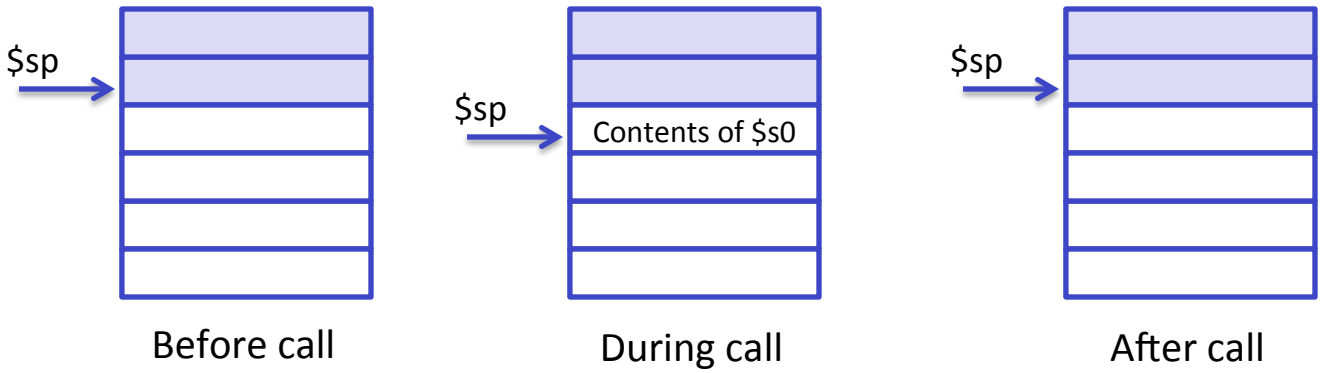
Simple Example

"Leaf" procedure (doesn't call another procedure)

```
int leaf(int g, int h, int i, int j) {
  int f;
  f = (g + h) - (i + j);
  return f;
}
```

Don't save $ra because leaf() doesn't call any other procedures
Don't allocate memory for **f** – optimizing compiler put in register
(can direct compiler with **register int** declaration)

```
addi $sp,$sp,-4    # adjust stack to make room for saved reg
sw $s0,0($sp)      # save $s0
add $t0,$a0,$a1    # $t0 = g + h (how did they get there?)
add $t1,$a2,$a3    # $t1 = i + j
sub $s0,$t0,$t1    # f = (g + h) - (i + j)
add $v0,$s0,$zero  # return f
lw $s0,0($sp)      # restore $s0 for caller
addi $sp,$sp,4     # restore stack
jr $ra             # return to caller
```



An Example: Sort

```
void sort(int v[], int n) {
  int i, j;
  for(i = 0; i < n; i+= 1){
    for (j = i-1; j >= 0 && v[j] > v[j+1]; j-=1) {
      swap(v,j);
    }
  }
}
```

```
void swap(int v[], int k) {
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

- swap() is a leaf function so doesn't need to save its return address on stack or save its own arguments
- It doesn't use any saved registers, so doesn't need to save them on stack
- It has a local variable but it's implemented in a non-saved register and doesn't require space on stack

```
swap:   sll $t1,$a1,2     # $t1 = k * 4
        add $t1,$a0,$t1   # $t1 = v + k*4 (address of v[k])
        lw $t0,0($t1)     # $t0(temp) = v[k]
        lw $t2,4($t1)     # $t2 = v[k+1]
        sw $t2,0($t1)     # v[k] = $t2
        sw $t0,4($t1)     # v[k+1] = $t0(temp)
        jr $ra
```

```
sort:     addi $sp,$sp,-20    # make room for 5 regs
          sw $ra,16($sp)      # save $ra
          sw $s3,12($sp)      # save $s3
          sw $s2,8($sp)       # save $s2
          sw $s1,4($sp)       # save $s1
          sw $s0,0($sp)       # save $s0
          move $s2,$a0        # save argument v
          move $s3,$a1        # save argument n
          move $s0,$zero      # i = 0
for1tst:  slt $t0,$s0,$s3     # $t0 = 1 if $s0 < $s3
          beq $t0,$zero,exit1 # exit if i >= n
          addi $s1,$s0,-1     # j = i-1
for2tst:  slti $t0,$s1,0      # $t0 = 1 if j < 0
          bne $t0,$zero,exit2 # exit if j < 0
          sll $t1,$s1,2       # j * 4
          add $t2,$s2,$t1     # v + j*4
          lw $t3,0($t2)       # $t3 = v[j]
          lw $t4,4($t2)       # $t4 = v[j+1]
          slt $t0,$t4,$t3     # $t0 = 1 if v[j] > v[j+1]
          beq $t0,$zero,exit2 # exit if v[j] <= v[j+1]
          move $a0, $s2       # prepare to pass v[]
          move $a1, $s1       # pass j
          jal swap            # call swap()
          addi $s1,$s1,-1     # j--
          j for2tst           # jump to inner loop test
exit2:    addi $s0, $s0,1     # i++
          j for1tst           # jump to outer loop test
exit1:    lw $s0,0($sp)       # restore $s0
          lw $s1,4($sp)       # restore $s1
          lw $s2,8($sp)       # restore $s2
          lw $s3,12($sp)      # restore $s3
          lw $ra,16($sp)      # restore $ra
          addi $sp,$sp,20     # restore $sp
          jr $ra              # return
```

Simple Example

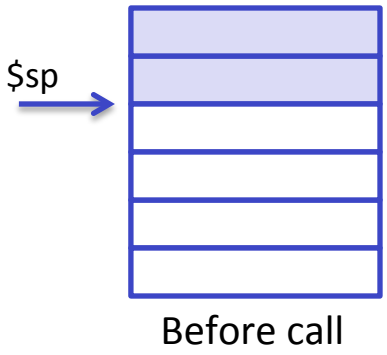
"Leaf" procedure (doesn't call another procedure)

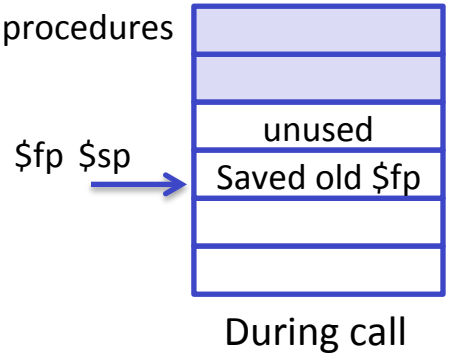
```
int empty(void)
{
return 0;
}
```

```
 1 empty:
 2          addiu   $sp,$sp,-8
 3          sw      $fp,0($sp)
 4          move    $fp,$sp
 5          move    $2,$0
 6          move    $sp,$fp
 7          lw      $fp,0($sp)
 8          addiu   $sp,$sp,8
 9          j       $31
10          nop
```

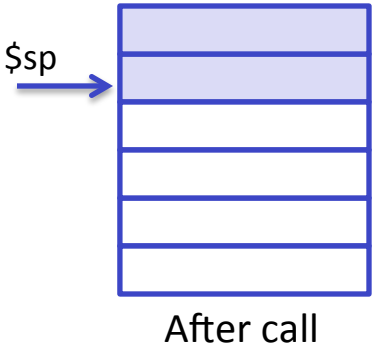
Register save area must by double word aligned

No need to save $ra ($31): not calling other procedures





line 2: reserve 8 bytes stack space
line 3: push the previous $fp to stack
line 4: take current $sp as the subroutine's frame pointer
line 5: set $v0 to zero ($v0 stores the return value)
line 6,7,8: pop out $fp from stack, restore back original $sp
line 9: jump to return address
line 10: branch delay slot



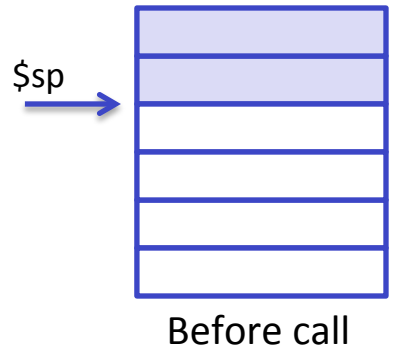
Simple Example

```
int emptycaller(void)
{
empty();
return 0;
}
```

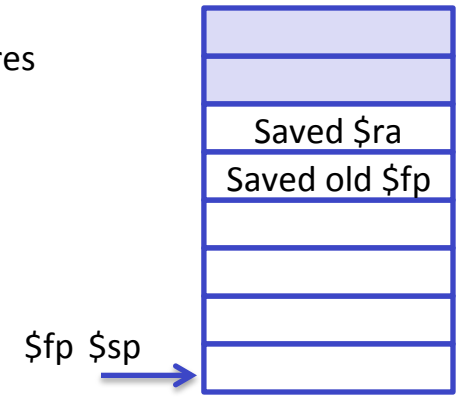
```
 1 emptycaller:
 2          addiu   $sp,$sp,-24
 3          sw      $31,20($sp)
 4          sw      $fp,16($sp)
 5          move    $fp,$sp
 6          jal     empty
 7          nop
 8          move    $2,$0
 9          move    $sp,$fp
10          lw      $31,20($sp)
11          lw      $fp,16($sp)
12          addiu   $sp,$sp,24
13          j       $31
14          nop
```

Minimum size for argument area

Save $ra ($31), will call other procedures



Before call



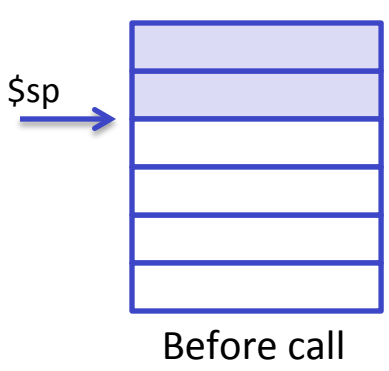
During call before call to empty()

line 3: push $ra onto the stack
line 4: push $fp onto the stack
line 10: pop $ra from the stack
line 11: pop $fp from the stack

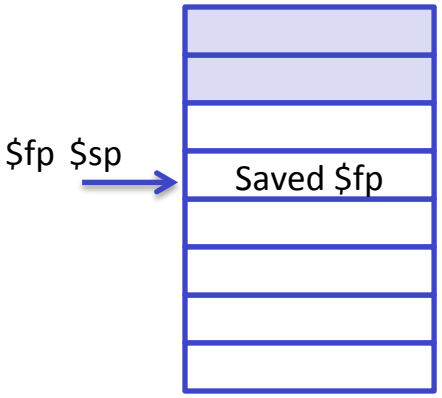
Simple Example

```
int args3(int x, int y, int z)
{
return (x + y + z);
}
```

```
 1 args3:
 2          addiu   $sp,$sp,-8
 3          sw      $fp,0($sp)
 4          move    $fp,$sp
 5          sw      $4,8($fp)
 6          sw      $5,12($fp)
 7          sw      $6,16($fp)
 8          lw      $3,8($fp)
 9          lw      $2,12($fp)
10          nop
11          addu    $2,$3,$2
12          lw      $3,16($fp)
13          nop
14          addu    $2,$2,$3
15          move    $sp,$fp
16          lw      $fp,0($sp)
17          addiu   $sp,$sp,8
18          j       $31
19          nop
```



Before call

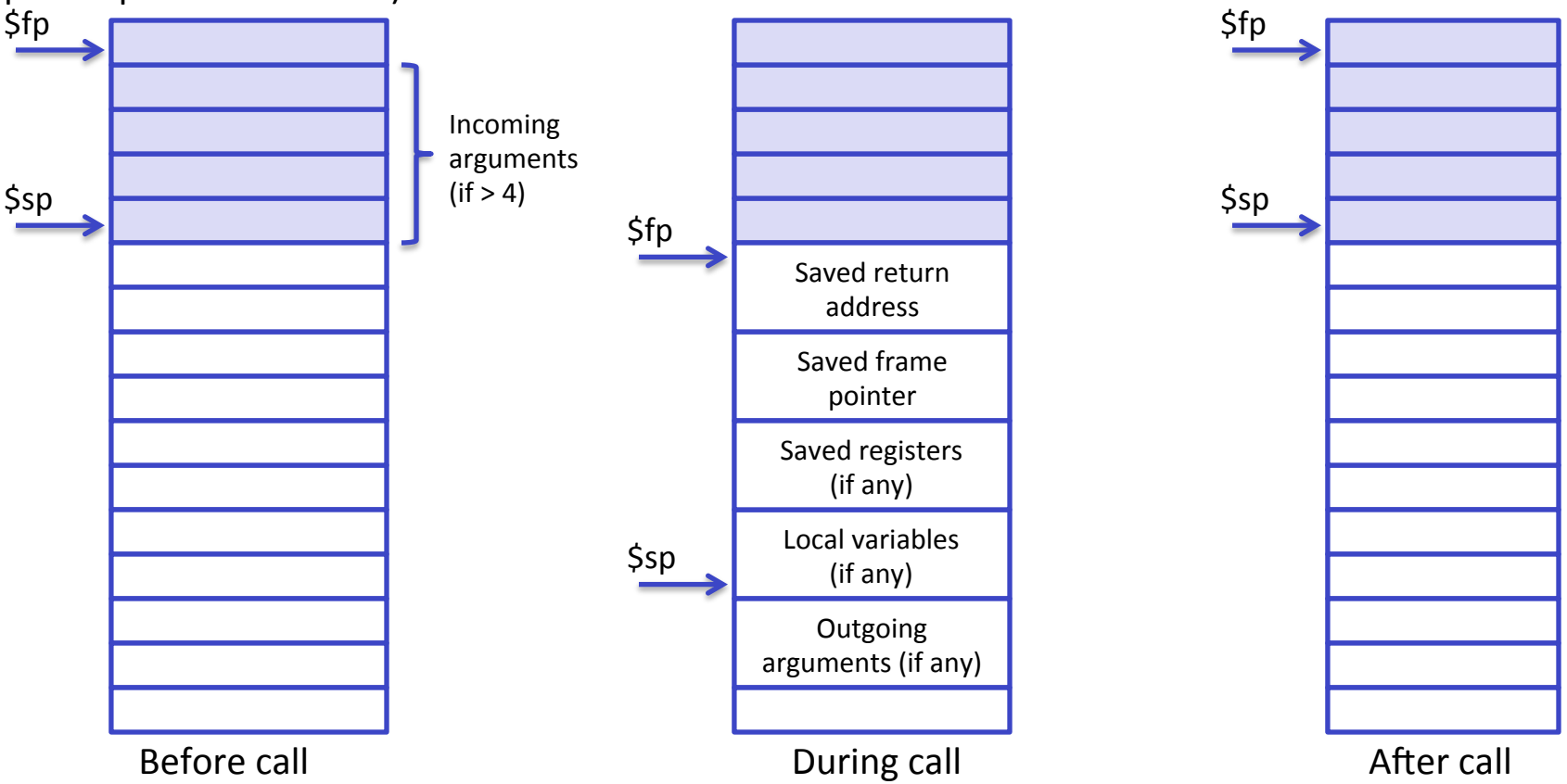


During call before call to empty()

line 2~4: the function prologue as described previously
line 5~7: push the arguments from $a0, $a1, and $a2 to the stack
line 8~14: add the 3 arguments, and put the sum in $v0 for the return value
line 15~17: the function epilogue as described previously
line 18~19: return to the caller
The arguments are pushed to 8($fp) to 16($fp), which are located in the caller's stack frame. A non-leaf function (Caller) will always reserve the top 4 words (16 bytes) at least to storing arguments to called functions. It is called the function call argument area.

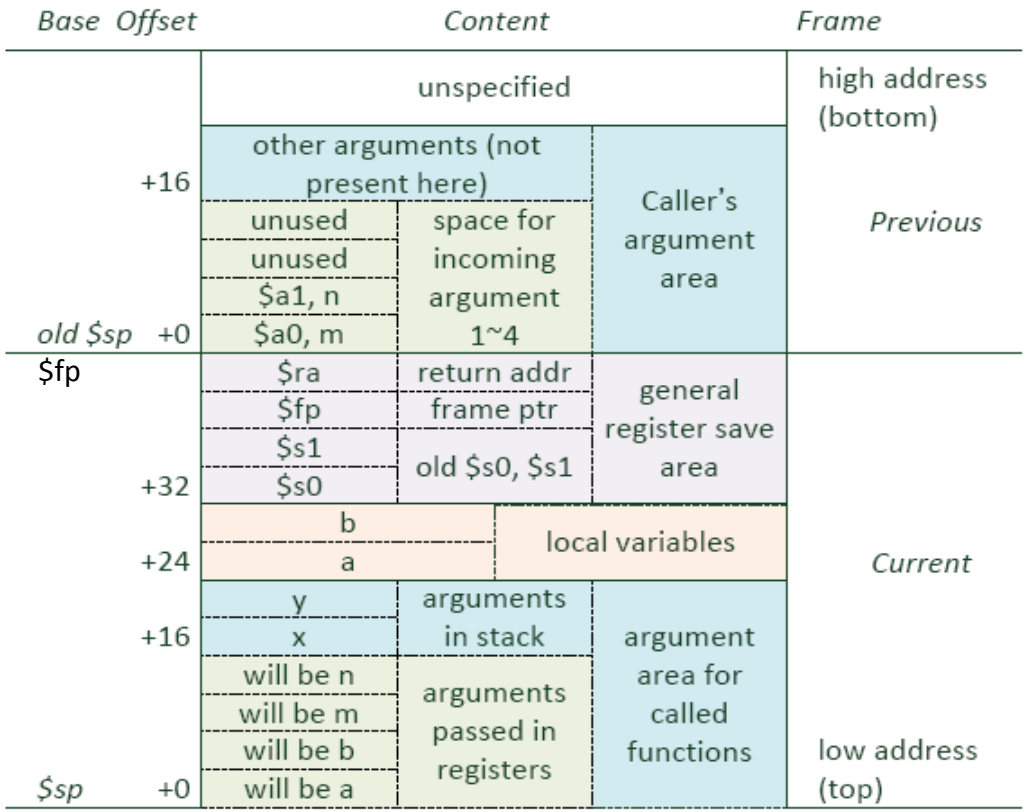
More Complicated Example

Activation record (procedure frame) created for every procedure call. Frame pointer points to first word in frame. Stack pointer may change during procedure execution (it may call other procedures, pushing outgoing arguments onto the stack) so location of local variables relative to $sp will be different at different points in the procedure, but always a constant offset from $fp. Makes code more readable, assists debugger. Not required. MIPS C compiler doesn't use $fp while GNU (gcc) does. Architectures may impose minimum frame sizes and alignments (e.g. double word to facilitate floating point operands on stack)



Complex Example

```
int comp(int m, int n)
{
int a = 1, b = 2;
register int x = 5, y = 6;
args6(a, b, m, n, x, y);
return (x + y);
}
```



Notes:

Caller allocates space in its stack frame for (minimum of) four arguments: comp() isn't a leaf function and may need to save its incoming argument registers before using them to pass arguments to a procedure it calls
Two incoming arguments (m, n) are passed via registers ($a0 and $a1), not on stack
Local (auto) variables a and b are allocated space on the stack
Registers $s0 and $s1 used for local variables x and y. Because they're declared as register the compiler will not assign stack space to them – optimizing compiler may do this anyway for some local variables (e.g. loop counters)).
The call to the function args6() requires six arguments. Space will be allocated for all six in comp()'s stack frame but only values for x and y are put in stack. Values of a,b,m,n are passed in registers $a0 - $a4.

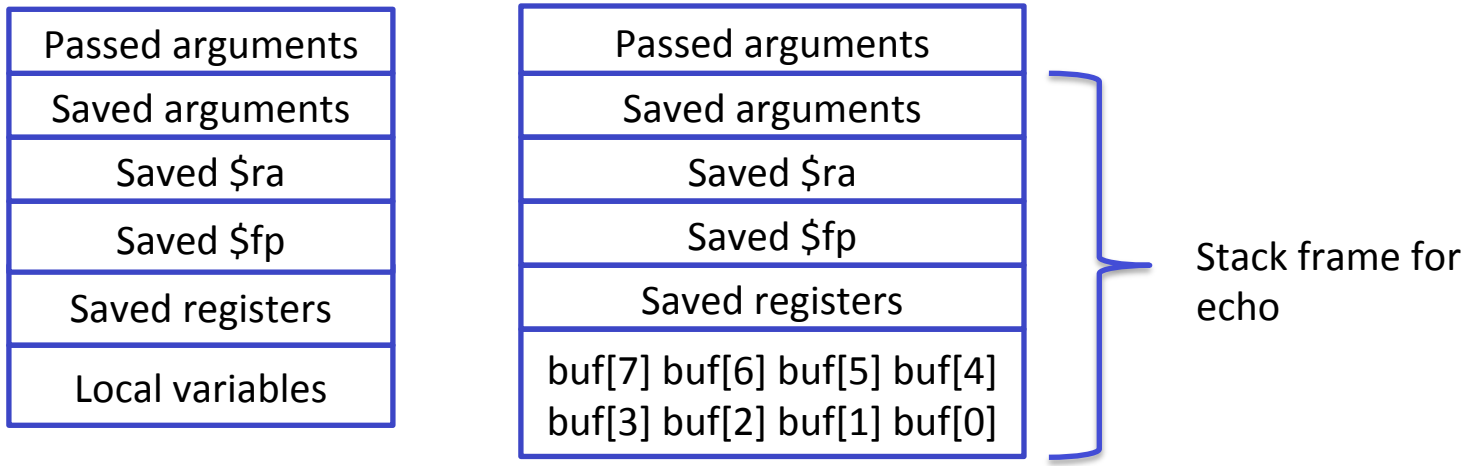
Out of Bounds Memory References and Buffer Overflow

Combination of no bounds checking on array (or pointer) references and storage of local variables along with state information (saved registers and return addresses) can lead to serious errors

```
char *gets(char *s) {
  int c;
  char *dest = s;
  int gotchar = 0;    /* read at least on character? */
  while ((c = getchar*()) != '\n' && c != EOF) {
    *dest++ = c;   /* no bounds checking ! */
    gotchar = 1;
  }
  *dest++ = '\0';  /* terminate string */
  if (c == EOF && !gotchar)   /* error */
    return NULL;
  return s;
}


void echo() {
  char buf[8];   /* Undersized */
  gets(buf);
  puts(buf);
}
```

Buffer Overflow Effects and "Stack Smashing"



As long as 7 or fewer characters typed, no problem
If more than 7 characters typed, begin to corrupt stack
Result depends on how many characters…

Corruption of data, "strange" behavior, segmentation violation

and what's typed…

Intentional overflowing buffer ("stack smashing") with "exploit code"
Causes stack to be overwritten with instructions
Stack location storing $ra overwritten with start address of exploit code
Return from procedure call invokes exploit code

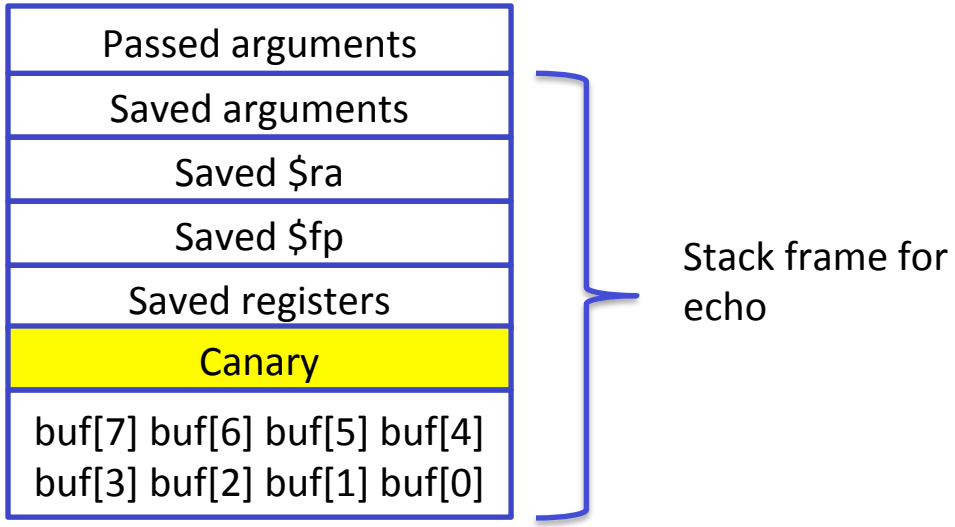
Thwarting Buffer Overflow Attacks

- Use better coding practices
  - Defensive coding
  - Avoid dangerous functions (*gets*, *strcpy*, *strcat*…)
- Randomize stack location
  - Attacks rely on knowledge of where stack is located
  - Randomize stack location – allows overwrite but no idea what value to place in saved $ra location in stack
  - Widely used on newer Linux systems
  - General address space layout randomization techniques
    - Code, stack, global variables loaded into different regions each time program is executed

```
int main() {
        int local;
        printf("local at %p\n", &local);
        return(0);
}
```

Thwarting Buffer Overflow Attacks

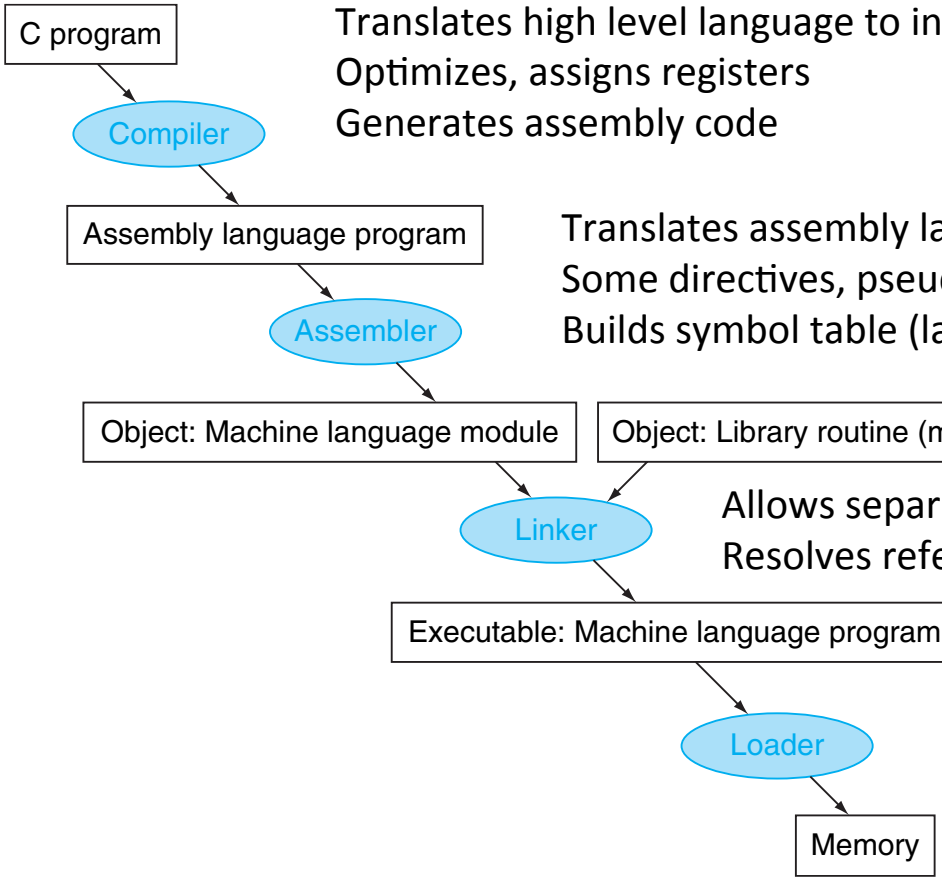
- Employ Canaries
  - Place a special value on stack
  - Check for corruption of this value before returning from call
  - *gcc* compiler employs ("stack protector" option)
  - Add minimal overhead to procedure call
  - Error handling if stack corrupted



Thwarting Buffer Overflow Attacks

- Limit Executable Regions
  - Use hardware segmentation or virtual memory to mark regions
    - Readable, writeable, executable
    - X86 merged read/execute into single 1-bit flag
      - Stack had to be readable and writeable – hence executable
    - AMD introduced NX (no execute) bit in 64-bit processors
    - Intel now does same

Translating and Starting Programs



Translates high level language to intermediate form
Optimizes, assigns registers
Generates assembly code

Translates assembly language to machine code (binary)
Some directives, pseudo-instruction replacement
Builds symbol table (labels)

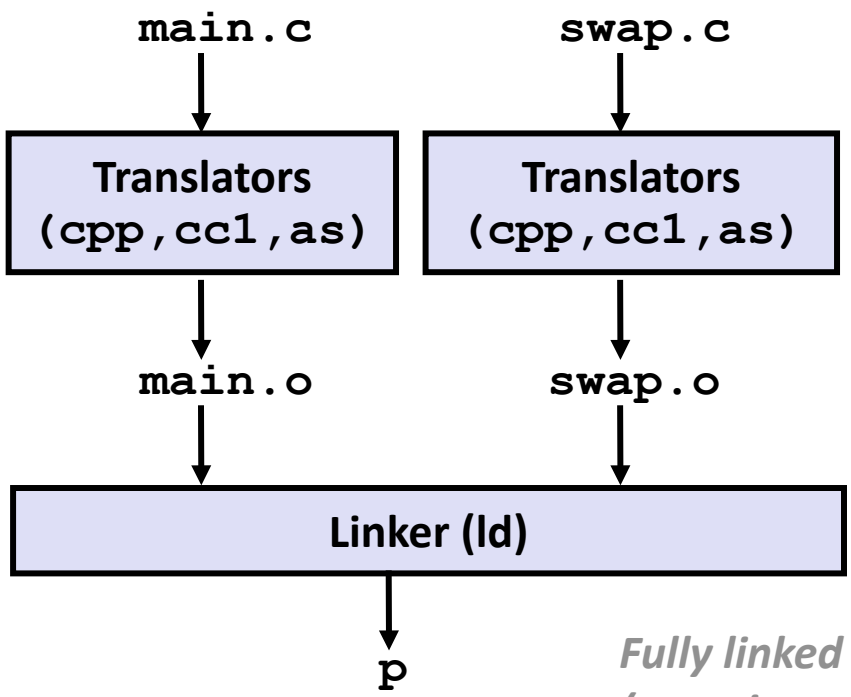
Allows separate compilation of modules, libraries
Resolves references between them, generates executable code

Loads executable image into memory
Sets up main arguments
Transfers control to program

Compiler Driver

Programs are translated and linked using a *compiler driver*:

```
unix> gcc -O2 -g -o p main.c swap.c
unix> ./p
```

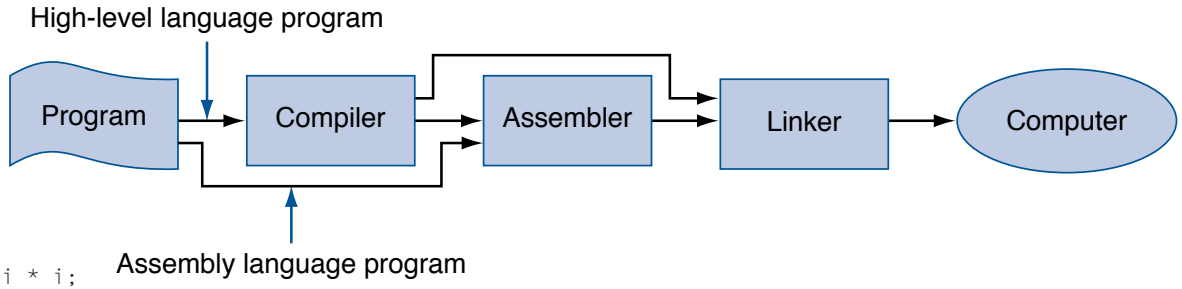


Source files

Separately compiled relocatable object files

Fully linked executable object file (contains code and data for all functions defined in main.c and swap.c

Compilers and Assemblers




```
   #include <stdio.h>

int
main (int argc, char *argv[])
{
     int i;
     int sum = 0;

     for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
     printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

```
     addiu     $29, $29, -32
     sw        $31, 20($29)
     sw        $4,  32($29)
     sw        $5,  36($29)
     sw        $0,  24($29)
     sw        $0,  28($29)
     lw        $14, 28($29)
     lw        $24, 24($29)
     multu     $14, $14
     addiu     $8,  $14, 1
     slti      $1,  $8, 101
     sw        $8,  28($29)
     mflo      $15
     addu      $25, $24, $15
     bne       $1,  $0, -9
     sw        $25, 24($29)
     lui       $4,  4096
     lw        $5,  24($29)
     jal       1048812
     addiu     $4,  $4, 1072
     lw        $31, 20($29)
     addiu     $29, $29, 32
     jr        $31
     move      $2,  $0
```

```
        .text
        .align    2
        .globl    main
main:
        subu      $sp, $sp, 32
        sw        $ra, 20($sp)
        sd        $a0, 32($sp)
        sw        $0,  24($sp)
        sw        $0,  28($sp)
loop:
        lw        $t6, 28($sp)
        mul       $t7, $t6, $t6
        lw        $t8, 24($sp)
        addu      $t9, $t8, $t7
        sw        $t9, 24($sp)
        addu      $t0, $t6, 1
        sw        $t0, 28($sp)
        ble       $t0, 100, loop
        la        $a0, str
        lw        $a1, 24($sp)
        jal       printf
        move      $v0, $0
        lw        $ra, 20($sp)
        addu      $sp, $sp, 32
        jr        $ra

        .data
        .align    0
str:
        .asciiz   "The sum from 0 .. 100 is %d\n"
```

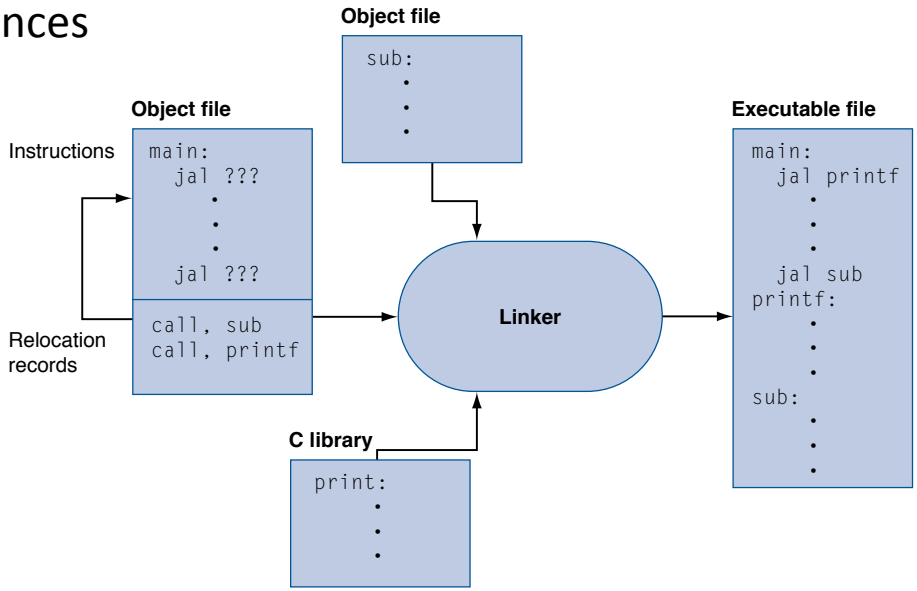
```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

Object file formats

- Several standards, depending upon platform
  - Unix a.out, ELF, COFF (variations), PE (portable executable)
  - Unix
    - Object file header: size and position of rest of file
    - Text segment: machine language code
    - Static data segment: static data
    - Relocation information: identifies instructions and data that rely on absolute addresses
    - Symbol table: labels not defined (e.g. external references)
    - Debugging information: how code was compiled
      - Allow debugger to associate machine code with high level language source code
      - Allow debugger to interpret data structures

Linker

- Waste of resources to recompile entire program when most portions unchanged, particularly standard library routines
- But assembler doesn't know address at which functions or data in a module will actually reside – makes them start at standard offset
- Assembler doesn't know where externally defined functions or data will reside so can't complete code (e.g. target of jal instruction)
- Three steps
  1. Place code and data modules symbolically in memory
  2. Determines addresses of data and instruction labels
  3. Patch internal and external references



Linking Example

Object file header			
	Name	Procedure A	
	Text Size	100_{16}	
	Data Size	20_{16}	
Text Segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	…		
Data Segment	0	(X)	
	…	…	
Relocation Information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol Table	Label	Address	
	X	-	
	B	-	

Linking Example

Object file header			
	Name	Procedure B	
	Text Size	200_{16}	
	Data Size	30_{16}	
Text Segment	Address	Instruction	
	0	sw $a1, 0($gp)	
	4	jal 0	
	…		
Data Segment	0	(Y)	
	…	…	
Relocation Information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol Table	Label	Address	
	Y	-	
	A	-	

Linking Example

Executable file header		
	Text Size	300_{16}
	Data Size	50_{16}
Text Segment	Address	Instruction
	$0040\ 0000_{16}$	lw \$a1, -8000_{16} (\$gp)
	$0040\ 0004_{16}$	jal 40 0100_{16}
	…	…
	$0040\ 0100_{16}$	sw \$a1, -7980_{16} (\$gp)
	$0040\ 0104_{16}$	jal 40 0000_{16}
	…	
Data Segment	Address	
	1000 0000	(X)
	…	
	1000 0020	(Y)
	…	

Linking

- External symbols
- Strong and weak symbols
  - Strong: functions, initializes global variables
  - Weak: uninitialized global variables
- Rules
  1. At most one strong symbol (more than one should result in error)
  2. Any number of weak symbols
  3. If strong symbol exists, use it
  4. If no strong symbol, choose arbitrarily among weak symbols

main.c

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Linker Symbols

Global symbols

- Symbols defined by a module that can be referenced by other modules
- E.g.: non-**`static`** C functions and non-**`static`** global variables

External symbols

- Global symbols that are referenced by a module but defined by some other module

Local symbols

- Symbols that are defined and referenced exclusively by a module
- E.g.: C functions and variables defined with the **`static`** attribute
- **Local linker symbols are *not* local program variables**

Global, External, or Local?

main.c

```
int buf[2] = {1, 2};

int main()
{
  swap();
  return 0;
}
```

swap.c

```
extern int buf[];

static int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

- **In main.c**
  - buf
  - main
  - swap
- **In swap.c**
  - buf
  - bufp0 / bufp1
  - swap
  - temp

Linking

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (p1)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to x will refer to the same uninitialized int.

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to x in p2 might overwrite y! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to x in p2 will overwrite y! Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to x will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

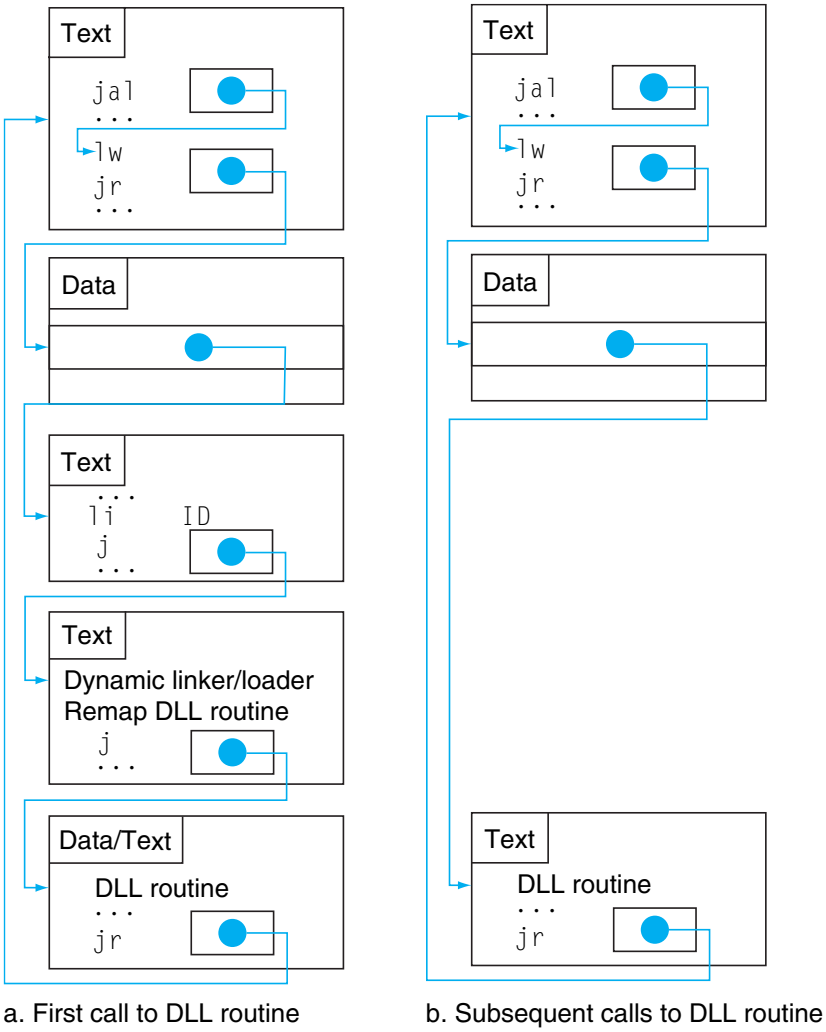
Loader

- Executable file on disk
- Unix loader
  1. Reads executable file header to determine size of text and data segments
  2. Creates address space large enough for text and data
  3. Copies instruction and data from executable file to memory
  4. Copies parameters (if any) to main program onto stack
  5. Initialized any machine state and sets stack pointer to first free location
  6. Calls main routine in program

Static and Dynamically Linked Libraries

- Statically linked libraries
  - Executable file contains copy of all library code used
    - Actually any library routine mentioned – may never be called
  - Updates to library requires relinking all code using library
  - Each file and program memory image using the library contains (redundant) copy of library code
- Dynamically linked libraries (DLLs or .so (shared objects))
  - Libraries aren't linked and loaded until program executes
  - Even better: "lazy" procedure linkage
    - Only link library routine first time call is made to it

DLLs: Lazy Procedure Linkage



a. First call to DLL routine

b. Subsequent calls to DLL routine

Exploits indirection

1. First call to procedure calls dummy entry
2. Executes indirect jump to a procedure that puts index identifying desired procedure into register
3. Jumps to dynamic linker/loader
4. Linker/loader finds desired routine (loading it if necessary)
5. Changes indirect pointer to point to the routine
6. Jumps to routine

Subsequent calls simply go to routine (with one level of indirection)

Adds a little overheard on first call
Minor overhead for subsequent calls