

Review: MIPS Register Usage

R0, \$0 = ## zero
R2, \$2 = ## return value / system call number
R4, \$4 = ## 1st argument
R5, \$5 = ## 2nd argument
R6, \$6 = ## 3rd argument
R7, \$7 = ## 4th argument
R29, \$29 = ## stack pointer
R30, \$30 = ## frame pointer
R31, \$31 = ## return addr

How a System Call Works

```
/* In Nachos all user programs are linked
 * with start.s, it begins at virtual address 0
 */
.globl __start
.ent    __start
__start:
jal     main    ## store addr of next
move    $4,$0   ## instr in $31 then jump
jal     Exit    ## if return from main exit(0)
.end    __start
```

How a System Call Works

```
/* call.c
 * Show how a function/syscall is made.
 */
#include "syscall.h"

int
main()
{
    ## Q: What's wrong with this code?
    ## A: It isn't checking the return code
    Write("Hello World\n", 12, 1);
}
```

How a System Call Works

```
.file    1 "call.c"
.rdata   ## read only data segment
.align   2  ## to 1=byte 2=word 3=dw
$LC0:
.ascii   "Hello World\n\000"
.text     ## start text/code segment
.align    2
.globl    main
.ent      main
```

How a System Call Works

main:

```
.frame    $fp, 24, $31
.mask     0xc0000000, -4 ## bitmask saved regs
.fmask    0x00000000, 0  ## saved fp regs
subu     $sp, $sp, 24    ## min frame size 24
sw       $31, 20($sp)    ## save ret addr
sw       $fp, 16($sp)    ## save old frame ptr
move     $fp, $sp        ## setup new frame ptr
jal      __main          ## for system init
la       $4, $LC0        ## addr of string arg1
li       $5, 0x00000000c  ## 12                arg2
li       $6, 0x000000001  ## 1                arg3
jal      Write           ## call Write
```

How a System Call Works

\$L1:

```
.set    noreorder
move    $sp,$fp        ## set stack ptr
lw      $31,20($sp)     ## restore ret addr
lw      $fp,16($sp)     ## restore frame ptr
j      $31              ## return to __start
addu    $sp,$sp,24      ## branch delay slot!
.set    reorder        ## pops stack frame
.end    main
```

How a System Call Works

```
/* dummy function to keep gcc happy */  
    .globl  __main  
    .ent    __main  
__main:  
    j       $31      ## return to point of call  
    .end    __main  ## i.e. back to __start
```

How a System Call Works

```
## start.s Comments: System call stubs:  
## Assembly language assist to make system  
## calls to the Nachos kernel. There is one  
## stub per system call, that places the code  
## for the system call into register r2, and  
## leaves the arguments to the system call  
## alone (in other words, arg1 is in r4, arg2  
## is in r5, arg3 is in r6, arg4 is in r7)  
##  
## The return value is in r2. This follows the  
## standard C calling convention on the MIPS.
```

How a System Call Works

```
.globl Write  
.ent    Write
```

Write:

```
addiu $2,$0,SC_Write  
syscall  
j      $31  
.end Write
```

How a System Call Works

```
/* system call codes */  
#define SC_Halt          0  
#define SC_Exit          1  
#define SC_Exec          2  
#define SC_Join          3  
#define SC_Create        4  
#define SC_Open          5  
#define SC_Read          6  
#define SC_Write         7  
#define SC_Close         8  
#define SC_Fork          9  
#define SC_Yield        10
```

How a System Call Works

```
void ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);

    kernel->currentThread->SaveUserState();
    kernel->currentThread->space->SaveState();

    switch(which) {
    case SyscallException:
        switch(type) {

        case SC_Write:
            vaddr = kernel->machine->ReadRegister(4);
            len = kernel->machine->ReadRegister(5);
            fileID = kernel->machine->ReadRegister(6);
            retval = WriteHandler(fileID, vaddr, len);
            break;
```

C Code for Segments Example

```
#define N    (5)    ## Preprocessor replaces this

unsigned int x = 0xdeadbeef;  # initialized data
int y = 0xbb;                # ditto
const int blah = 0xff;       # rdata (read only)
int data[N];                 # uninit data

struct info {    ## Doesn't use any storage!
    int x;
    int y;
};
```

C Code for Segments Example (cont'd)

```
main()  
{  
    int i;  
    int j = 0xaa;  
    int k;  
    const int l = 0xee;  ## all above in regs/stack  
    char *str = "Hello World\n";  
    ## str in on stack or register  
    ## Hello World in .rdata  
  
    for (i=0; i<N; i++) {  
        data[i] = i;  ## code/instructs in text  
    }  
}
```

Coff2noff Output for Segments

Loading 4 sections:

".text"	filepos	52 (0x34)	mempos	0 (0x0)
	size	736 (0x2e0)		
".rdata"	filepos	788 (0x314)	mempos	768 (0x300)
	size	32 (0x20)		
".data"	filepos	820 (0x334)	mempos	896 (0x380)
	size	16 (0x10)		
".bss"	filepos	-1 (0xffffffff)	mempos	1024 (0x400)
	size	20 (0x14)		

<not in file>

See next few slides for explanations

Coff2noff Output for Segments

```
## .text  at 0x34 (52) in file
##      size = 0x2e0 (736 bytes)
##      starts at to virtual addr 0x0

## .rdata at .text start (52) + text size (736)
##      = 788 in file
##      Contains 4 bytes for int blah and 13 for
##      "Hello World\n\0" total = 17
##      (word aligned in file, page aligned in mem)
##      Page = 128 bytes
##      1 = 0x80, 2 = 0x100, 3 = 0x180, 4 = 0x200,
##      5 = 0x280, 6 = 0x300 7 = 0x380, 8 = 0x400
##      .rdata starts at 0x300 in virtual memory
```

Coff2noff Output for Segments

```
## .data in file at .rdata start (788) +  
##      .rdata size (32) = 820  
## Contains 4 bytes for int x + 4 bytes for yy = 8  
## .data starts at 0x380 in virtual memory  
  
## .bss (no where because no values to store)  
  
## (bss = Block Started by Symbol)  
## have only a name / symbol but no value  
## (uninitialized data)
```

Some Output from objdump

Contents of section .rdata:

```
0300 ff000000 48656c6c 6f20576f 726c640a
    ....Hello World.
```

```
0310 00000000 00000000 00000000 00000000
```

.....

Contents of section .data:

```
0380 efbeadde bb000000 00000000 00000000
```

.....

See next slides for explanation

Some Output from objdump

```
## H = 0x48 e = 0x65 l = 6c ... \n = 0x0a

## x = 0xdeadbeef /* 3735928559 */
## Byte ordering.  Mips - little endian
## Least significant byte at lowest address
## Word addressed by address of least sign. byte
#
## 0 .. 7 8.. 15 16..23 24..31
## [ ef ] [ be ] [ ad ] [ de ]

## Big Endian
## Most significant byte at lowest address
## Word addressed by address of most sign. byte

## 0 .. 7 8 ..15 16..23 24..31
## [ de ] [ ad ] [ be ] [ ef ]
```