

# Implementing Locks in C

Recitation

# How to implement atomics?

Two ways:

- Use intrinsics offered by your compiler
- Use inline assembly

Today we will take a look at both options.

We use the gcc compiler as an example. Other compilers (Intel, XLC) have very similar features.

# GCC Atomic Builtins

```
type __sync_fetch_and_add(*ptr, type val)  
type __sync_add_and_fetch(*ptr, type val)
```

- Full memory Barrier
- Instead of "add" also available as "sub", "xor", etc.
- Returns old value (FAA) or the new one (AAF)

# **GCC Atomic Builtins**

```
bool __sync_bool_compare_and_swap(type *ptr, type  
oldval, type newval)
```

```
type __sync_val_compare_and_swap(type *ptr, type  
oldval, type newval)
```

- Full memory Barrier
- Set \*ptr to newval if \*ptr is equal to oldval
- Return oldval (`_val_`) or truth-value of the comparison (`_bool_`)

# **GCC Atomic Builtins**

```
void __sync_synchronize()
```

- Full memory barrier (mfence)

```
type __sync_lock_test_and_set(type *ptr, type  
value)
```

- Full memory barrier
- Write value into \*ptr, return the previous value

# **GCC Inline Assembly**

- GCC inline Assembly uses **AT&T** Syntax: The first operand is the source, second operand the destination
- Intel Syntax: destination is first operand
- Register names are prefixed with %
- Constants are prefixed with \$ (\$0x for Hex)

Example: `mov $42, %eax`

# **GCC Inline Assembly**

- The size of the operands is part of the mnemonic (optional! - assembler will guess)
  - `movq $0x42, %rax`            64 Bit
  - `movl $0x42, %eax`            32 Bit
  - `movw $0x42, %ax`            16 Bit
  - `movb $0x42, %al`            8 Bit

# **GCC Inline Assembly**

- Memory accesses are specified as `$disp(%base, %offset, $multiply)` this refers to  $*(base + disp + offset * multiply)$
- Everything except base is optional
- Common Case:
  - `movl -4(%ebp), %eax`
  - `movl (%ecx), %edx`



# **GCC Inline Assembly**

- C compiler does not "understand" inline assembly - it simply copies it into the output stream
- We need to tell the compiler
  - Which registers we overwrite in assembly
  - Which values we access

# **GCC Inline Assembly**

```
int a=10, b;  
asm ("movl %1, %%eax;\n"  
    "movl %%eax, %0;\n"  
    : "=r"(b)          /* output */  
    : "r"(a)           /* input */  
    : "%eax"           /* clobbered register */);
```

Input/Output Values are referenced by %0, %1, etc.

Constraints tell the compiler where to put values:

"r" -> any register (a -> eax, b->ebx, etc)

"=r" -> register is used write-only

"m" -> memory location

# Compare and Swap in Assembler

```
unsigned long cas(volatile unsigned long* ptr,
                 unsigned long old, unsigned long new) {
    unsigned long prev;
    asm volatile("lock; cmpxchgq %1, %2;"
                : "=a"(prev)
                : "r"(new), "m"(*ptr), "a"(old) : );
    return prev;
}
```

```
cmpxch: if (eax == dest) {ZF=1; dest=src}
        else {ZF=0; eax=dest}
```

# Fetch and Add in Assembler

```
int fetch_and_add(int* ptr, int val){
    asm volatile(
        "lock; xaddl %%eax, %2;"
        : "=a" (val)
        : "a" (val), "m" (*ptr) : );
    return val;
}
```

xadd: src = dest; dest += src;

Always use volatile to prevent the compiler from reordering!