- Note -

Examples on following slides assume a slightly different version of Project 0 (used last semesters) where used blocks were also kept on a list, the "used list."

- mem_alloc would add a block to used list
- mem_free would remove block from used list

In that case, the code needed to protect both the free and used list

The following slides discuss correct and incorrect way of doing so

Using Locks

Associate each shared variable with lock L

- "lock L protects that variable"

```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock listlock; /* Protects usedlist & freelist */
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&listlock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&listlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&listlock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&listlock);
}
```

How many locks should I use?

Could use one lock for all shared variables

- Disadvantage: if a thread holding the lock blocks, no other thread can access *any* shared variable, even unrelated ones
- Sometimes used when retrofitting non-threaded code into threaded framework
- Examples:
  - "BKL" Big Kernel Lock in Linux
  - fslock in Pintos Project 2

Ideally, want fine-grained locking

- One lock only protects one (or a small set of) variables – how to pick that set?

Multiple locks, the wrong way


```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock alloclock; /* Protects allocations */
static struct lock freelock; /* Protects deallocations */
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&alloclock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&alloclock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&freelock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
}
```

Wrong: locks protect data structures, not code blocks! Allocating thread & deallocating thread could collide

Multiple locks, 2nd try


```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock usedlock; /* Protects usedlist */
static struct lock freelock;   /* Protects freelist */
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&usedlock);
    list_remove(&b->elem);
    lock_acquire(&freelock);
    coalesce_into_freelist(&freelist, b);
    lock_release(&usedlock);
    lock_release(&freelock);
}
```

Also wrong: deadlock!
Always acquire multiple locks in same order -
Or don't hold them simultaneously

Multiple locks, correct (1)


```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock usedlock; /* Protects usedlist */
static struct lock freelock;   /* Protects freelist */
```


```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&usedlock);
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&usedlock);
    lock_acquire(&freelock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
}
```

Correct, but inefficient!
Locks are always held simultaneously,
one lock would suffice

Multiple locks, correct (2)


```
static struct li
static struct li

static struct lo
static struct lo
```

Correct, but not necessarily better!

On uniprocessor:

No throughput from fine-grained locking, since no blocking inside critical sections – but pay twice the price compared to one-lock solution

On multiprocessor:

Gain from being able to manipulate free & used lists in parallel, but increased risk of contended locks

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    lock_release(&freelock);
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    lock_release(&usedlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&usedlock);
    list_remove(&b->elem);
    lock_release(&usedlock);
    lock_acquire(&freelock);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
}
```


Conclusion

Choosing which lock should protect which shared variable(s) is not easy – must weigh:

- Whether all variables are always accessed together (use one lock if so)
- Whether code inside critical section can block (if not, no throughput gain from fine-grained locking on uniprocessor)
- Whether there is a consistency requirement if multiple variables are accessed in related sequence (must hold single lock if so)
    - See “Subtle race condition in Java” below
- Cost of multiple calls to lock/unlock (increasing parallelism advantages may be offset by those costs)

Rules for Easy Locking

Every shared variable must be protected by a lock
- One lock may protect more than one variable, but not too many
- Acquire lock before touching (reading or writing) variable
- Release when done, on all paths

If manipulating multiple variables, acquire locks protecting each
- Acquire locks always in same order (doesn't matter which order, but must be same)
- Release in opposite order
- Don't mix acquires & release (two-phase locking)

Infinite Buffer Problem


```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  lock_release(buffer);
}
```

```
consumer()
{
  lock_acquire(buffer);
  while (buffer is empty) {
    lock_release(buffer);
    thread_yield();
    lock_acquire(buffer);
  }
  item = buffer[tail++];
  lock_release(buffer);
  return item
}
```

Trying to implement infinite buffer problem with locks alone leads to a very inefficient solution (busy waiting!)

Locks cannot express precedence constraint: A must happen before B.

Infinite Buffer Problem, Take 2


```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  if (#consumers > 0)
      for c in consumers {
        thread_unblock(c);
      }
  lock_release(buffer);
}
```

```
consumer()
{
   lock_acquire(buffer);
   while (buffer is empty) {
      lock_release(buffer);
      consumers.add(current);
      thread_block(current);
      lock_acquire(buffer);
   }
   item = buffer[tail++];
   lock_release(buffer);
   return item
```

Context switch here would cause *Lost Wakeup* problem: producer will put item in buffer, but won't unblock consumer thread (since consumer thread isn't in consumers yet)

Infinite Buffer Problem, Take 3


```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  if (#consumers > 0)
      for c in consumers {
        thread_unblock(c);
      }
  lock_release(buffer);
}
```

```
consumer()
{
   lock_acquire(buffer);
   while (buffer is empty) {
      consumers.add(current);
      lock_release(buffer);
      thread_block(current);
      lock_acquire(buffer);
   }
   item = buffer[tail++];
   lock_release(buffer);
   return item
}
```

What if consumers.add is done before lock is released?

Infinite Buffer Problem, Take 4


```
producer(item)
{
  lock_acquire(buffer);
  buffer[head++] = item;
  if (#consumers > 0)
      for c in consumers {
        thread_unblock(c);
      }
  lock_release(buffer);
}
```

```
consumer()
{
   lock_acquire(buffer);
   while (buffer is empty) {
      consumers.add(current);
      lock_release(buffer);
      thread_block(current);
      lock_acquire(buffer);
   }
   item = buffer[tail++];
   lock_release(buffer);
   return item
}
```

This is correct, but complicated and very easy to get wrong

- Want abstraction that does not require direct block/unblock call

Low-level vs. High-level Synchronization

Low-level synchronization primitives:

- Disabling preemption, (Blocking) Locks, Spinlocks
- implement mutual exclusion

Implementing precedence constraints directly via thread_unblock/thread_block is problematic because

- It's complicated (see last slides)
- It may violate encapsulation from a software engineering perspective
- You may not have that access at all (unprivileged code!)

We need well-understood higher-level constructs

- Semaphores
- Monitors