CS 33

Multithreaded Programming VI

Shared Data

- **Thread 1:**

```
printf("goto statement reached");
```

- **Thread 2:**

```
printf("Hello World\n");
```

- **Printed on display:**

```
go to Hell
```

Coping

- Wrap library calls with synchronization constructs
- Fix the libraries

Efficiency

- Standard I/O example
  - `getc()` and `putc()`
    - » expensive and thread-safe?
    - » cheap and not thread-safe?
  - two versions
    - » `getc()` and `putc()`
      - expensive and thread-safe
    - » `getc_unlocked()` and `putc_unlocked()`
      - cheap and not thread-safe
      - made thread-safe with `flockfile()` and `funlockfile()`

Efficiency

- **Naive**

```
for(i=0; i<lim; i++)
  putc(out[i]);
```

- **Efficient**

```
flockfile(stdout);
for(i=0; i<lim; i++)
  putc_unlocked(out[i]);
funlockfile(stdout);
```

What’s Thread-Safe?

- Everything except

asctime()
basename()
catgets()
crypt()
ctime()
dbm_clearerr()
dbm_close()
dbm_delete()
dbm_error()
dbm_fetch()
dbm_firstkey()
dbm_nextkey()
dbm_open()
dbm_store()
dirname()
dlerror()
drand48()

ecvt()
encrypt()
endgrent()
endpwent()
endutxent()
fcvt()
ftw()
gcvt()
getc_unlocked()
getchar_unlocked()
getdate()
getenv()
getgrent()
getgrgid()
getgrnam()
gethostbyaddr()
gethostbyname()

gethostent()
getlogin()
getnetbyaddr()
getnetbyname()
getnetent()
getopt()
getprotobyname()
getprotobynumber()
getprotoent()
getpwent()
getpwnam()
getpwuid()
getservbyname()
getservbyport()
getservent()
getutxent()
getutxid()

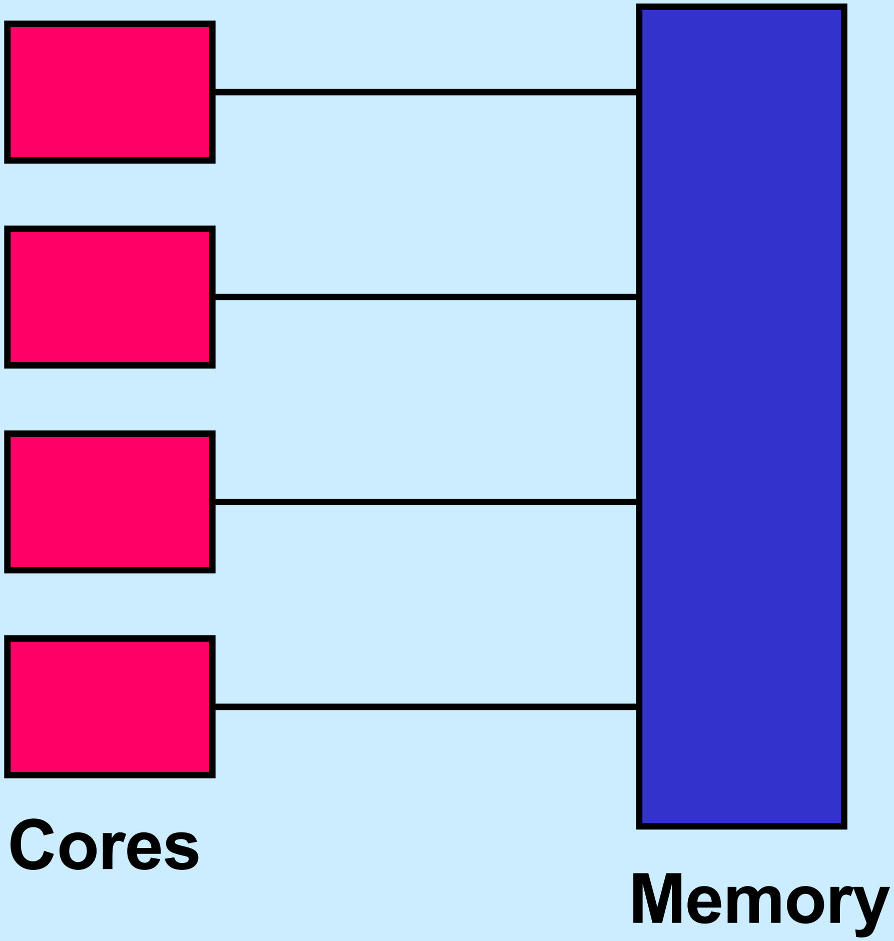
getutxline()
gmtime()
hcreate()
hdestroy()
hsearch()
inet_ntoa()
l64a()
lgamma()
lgammaf()
lgammal()
localeconv()
localtime()
lrand48()
mrand48()
nftw()
nl_langinfo()
ptsname()

putc_unlocked()
putchar_unlocked()
putenv()
pututxline()
rand()
readdir()
setenv()
setgrent()
setkey()
setpwent()
setutxent()
strerror()
strtok()
ttyname()
unsetenv()
wcstombs()
wctomb()

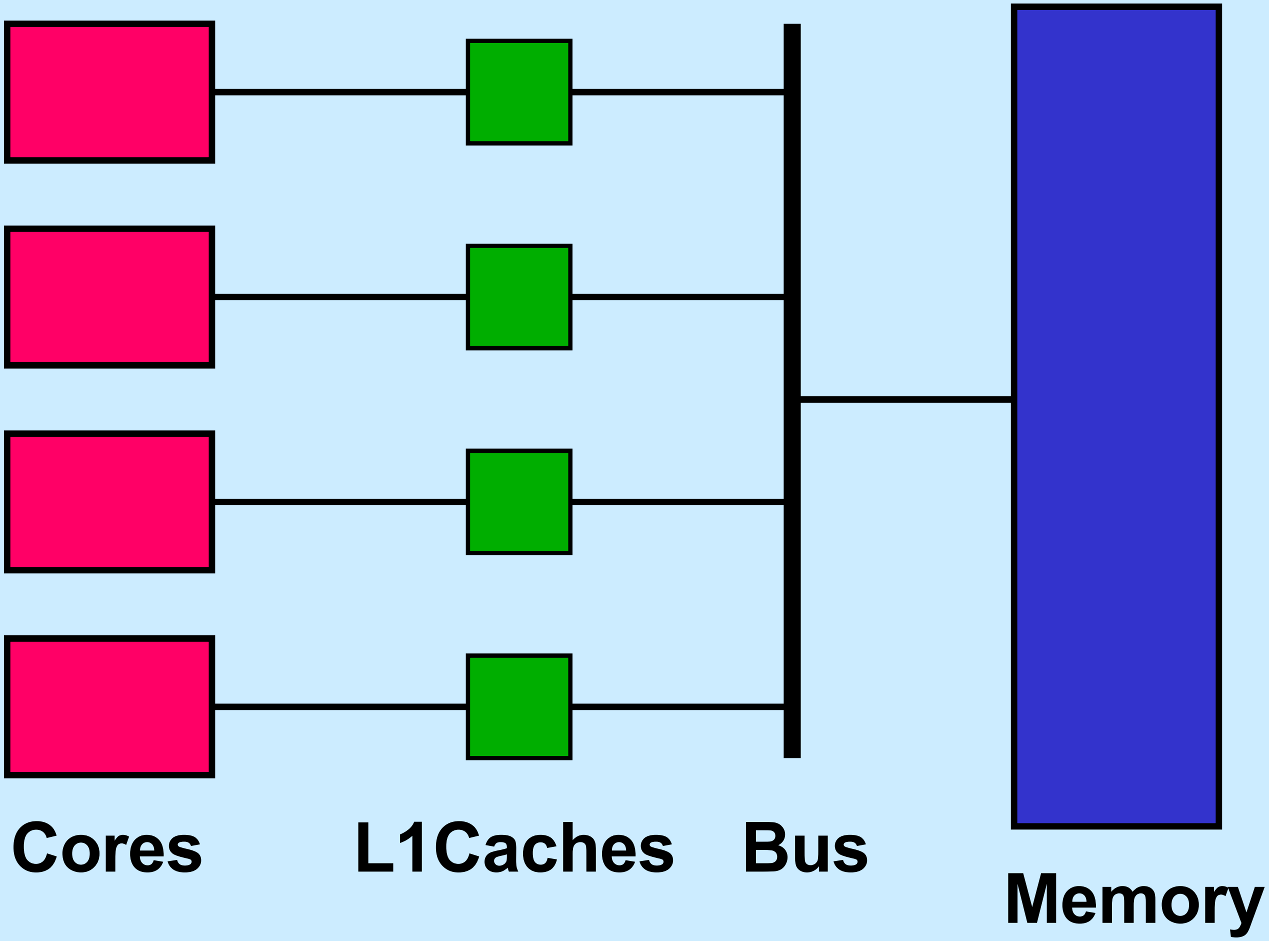
Concurrency

- Real
  - many things happen at once
  - multiple threads running on multiple cores
- Simulated
  - things appear to happen at once
  - a single core is multiplexed among multiple threads
    - time slicing

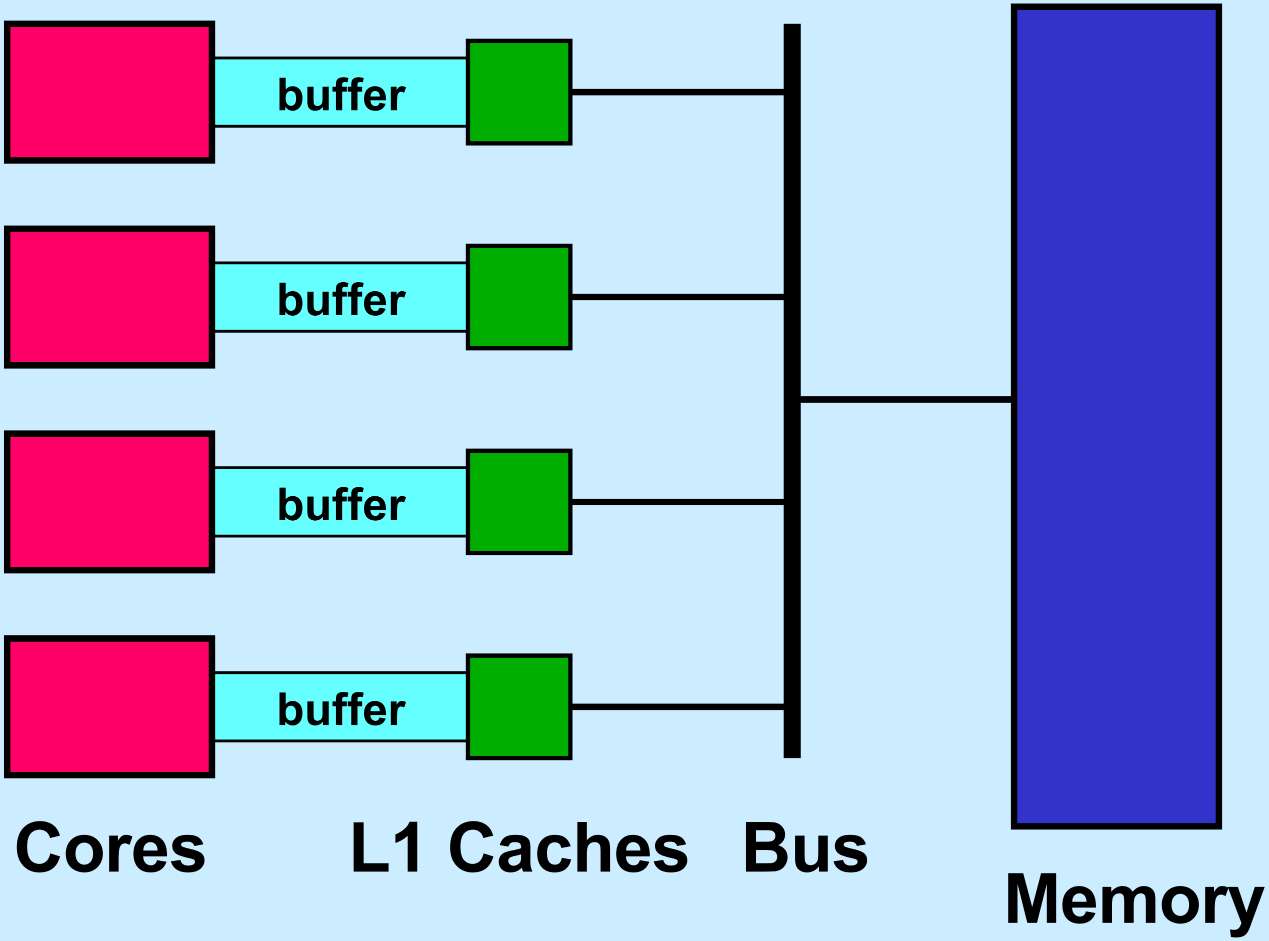
Multi-Core Processor: Simple View



Multi-Core Processor: More Realistic View



Multi-Core Processor: Even More Realistic



Concurrent Reading and Writing

Thread 1:

```
i = shared_counter;
```

Thread 2:

```
shared_counter++;
```

Mutual Exclusion w/o Mutexes

```
void peterson(long me) {
  static long loser;                // shared
  static long active[2] = {0, 0};   // shared
  long other = 1 - me;              // private
  active[me] = 1;
  loser = me;
  while (loser == me && active[other])
    ;
  // critical section
  active[me] = 0;
}
```

Quiz 1

```
void peterson(long me) {
  static long loser;                 // shared
  static long active[2] = {0, 0};    // shared
  long other = 1 - me;               // private
  active[me] = 1;
  loser = me;
  while (loser == me && active[other])
    ;
  // critical section
  active[me] = 0;
}
```

This works on sunlab computers.

a) never

b) usually

c) always

Busy-Waiting Producer/Consumer

```
void producer(char item) {

  while(in - out == BSIZE)
   ;

  buf[in%BSIZE] = item;

  in++;
}
```

```
char consumer( ) {
  char item;
  while(in - out == 0)
   ;

  item = buf[out%BSIZE];

  out++;

  return(item);
}
```

Quiz 2

```
void producer(char item) {

  while(in - out == BSIZE)
   ;

  buf[in%BSIZE] = item;

  in++;
}
```

```
char consumer( ) {
  char item;
  while(in - out == 0)
   ;

  item = buf[out%BSIZE];

  out++;

  return(item);
}
```

This works on sunlab computers.

a) never
b) usually
c) always

Quiz 3

```
void producer(char item) {

  while(in - out == BSIZE)
   ;

  buf[in%BSIZE] = item;

  in++;
}
```

```
char consumer( ) {
  char item;
  while(in - out == 0)
   ;

  item = buf[out%BSIZE];

  out++;

  return(item);
}
```

This works on computers with reordered stores.

a) never
b) usually
c) always

Coping

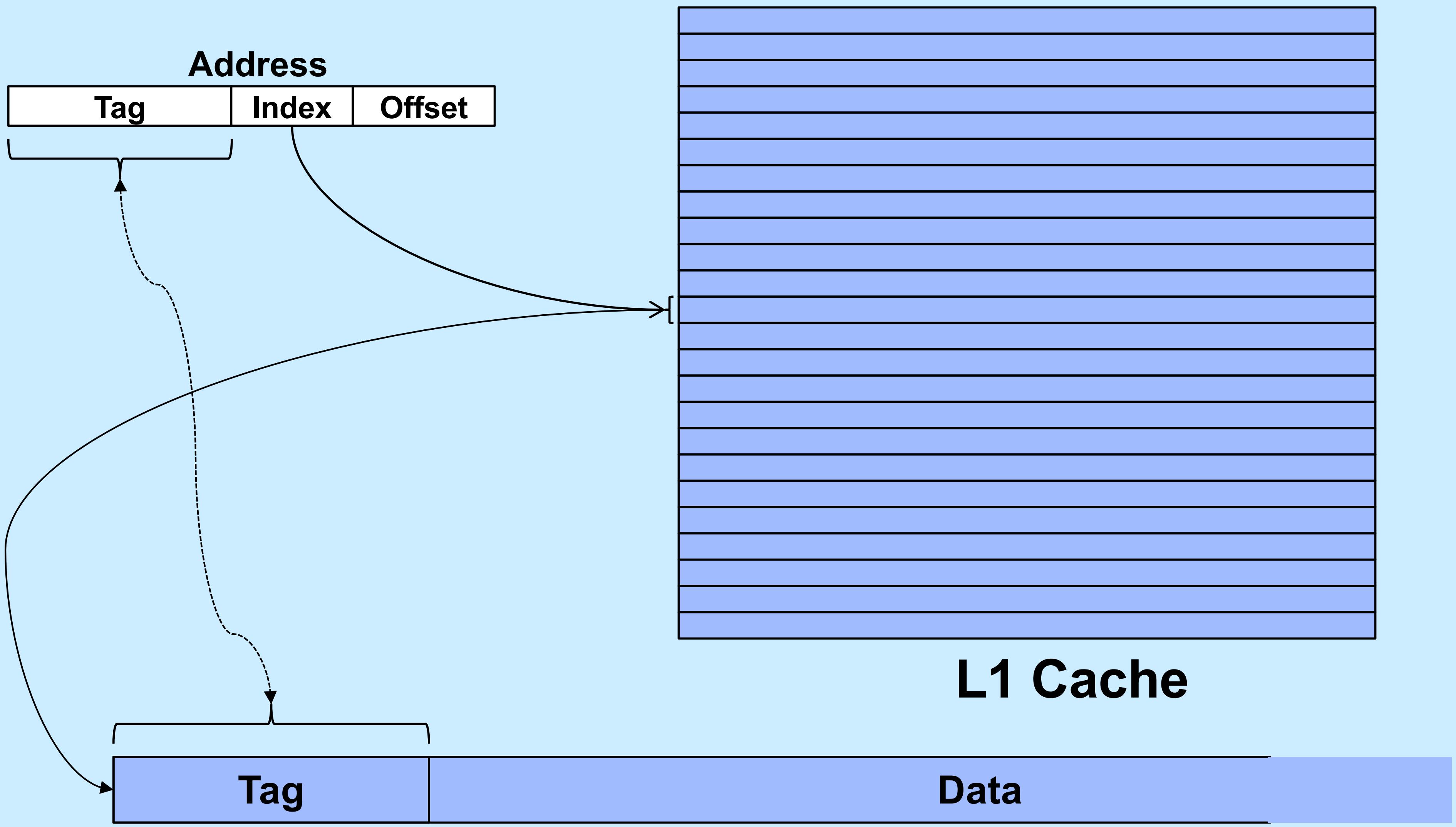
- Don’t rely on shared memory for synchronization
- Use the synchronization primitives

Which Runs Faster?

```
volatile int a, b;

void *thread1(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    a = 1;
  }
}

void *thread2(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    b = 1;
  }
}
```

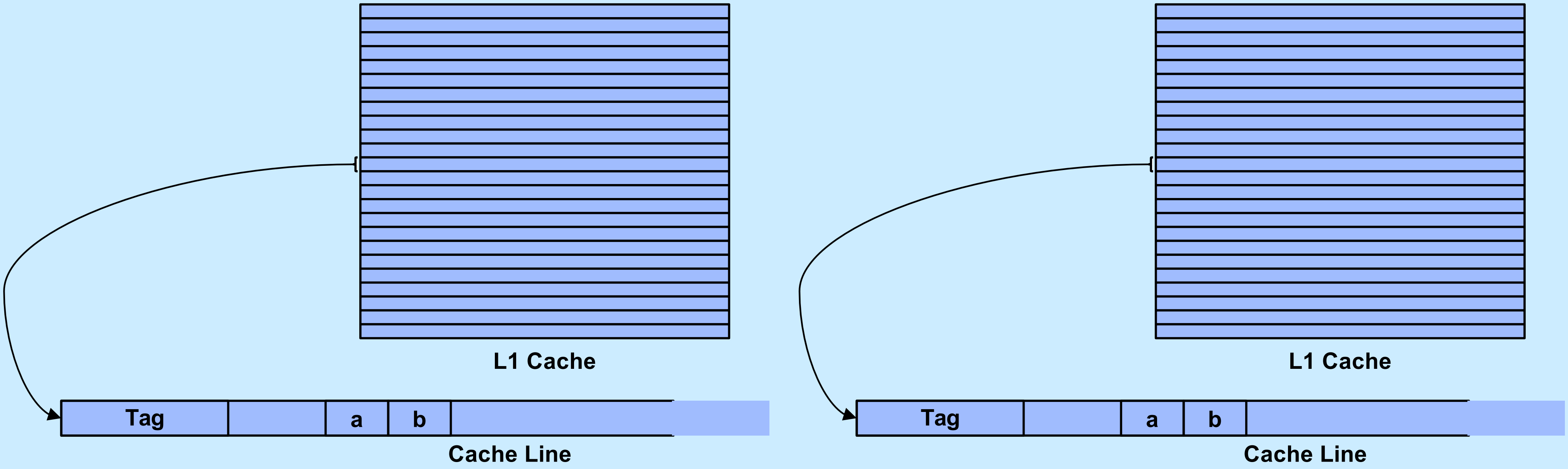
```
volatile int a, padding[128], b;

void *thread1(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    a = 1;
  }
}

void *thread2(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    b = 1;
  }
}
```

Cache Lines



Cache Line

False Sharing



Implementing Mutexes

- **Strategy**
  - make the usual case (no waiting) very fast
  - can afford to take more time for the other case (waiting for the mutex)

Futexes

- Safe, *efficient* kernel conditional queueing in Linux
- All operations performed atomically
  - `futex_wait(futex_t *futex, int val)`
    - » if `futex->val` is equal to `val`, then sleep
    - » otherwise return
  - `futex_wake(futex_t *futex)`
    - » wake up one thread from `futex`'s wait queue, if there are any waiting threads

Ancillary Functions

- `int atomic_inc(int *val)`
  - add 1 to `*val`, return its original value
- `int atomic_dec(int *val)`
  - subtract 1 from `*val`, return its original value

-
  ```
  int CAS(int *ptr, int old, int new) {
        int tmp = *ptr;
         if (*ptr == old)
              *ptr = new;
         return tmp;
    }
  ```

Attempt 1

```
void lock(futex_t *futex) {
  int c;
  while ((c = atomic_inc(&futex->val)) != 0)
    futex_wait(futex, c+1);
}

void unlock(futex_t *futex) {
  futex->val = 0;
  futex_wake(futex);
}
```

Quiz 4

```
void lock(futex_t *futex) {
  int c;
  while ((c = atomic_inc(&futex->val)) != 0)
    futex_wait(futex, c+1);
}

void unlock(futex_t *futex) {
  futex->val = 0;
  futex_wake(futex);
}
```

Why doesn't Attempt 1 work?

a) unlock fails to wake up a sleeping thread in certain circumstances

b) the while loop in lock doesn't terminate in certain circumstances

c) both of the above

d) none of the above

Attempt 2

```
void lock(futex_t *futex) {
  int c;
  if ((c = CAS(&futex->val, 0, 1) != 0)
    do {
      if (c == 2 || (CAS(&futex->val, 1, 2) != 0))
        futex_wait(futex, 2);
    while ((c = CAS(&futex->val, 0, 2)) != 0))
}

void unlock(futex_t *futex) {
  if (atomic_dec(&futex->val) != 1) {
    futex->val = 0;
    futex_wake(futex);
  }
}
```

Quiz 5

Does it work?

a) yes

b) no