CS 33

Multithreaded Programming VI

A Problem ...

- **In thread 1:**

```
if ((ret = open(path,
    O_RDWR) == -1) {
  if (errno == EINTR) {
    ...
  }
  ...
}
```

- **In thread 2:**

```
if ((ret = socket(AF_INET,
    SOCK_STREAM, 0)) {
  if (errno == ENOMEM) {
    ...
  }
  ...
}
```

There's only one errno!

However, somehow it works.

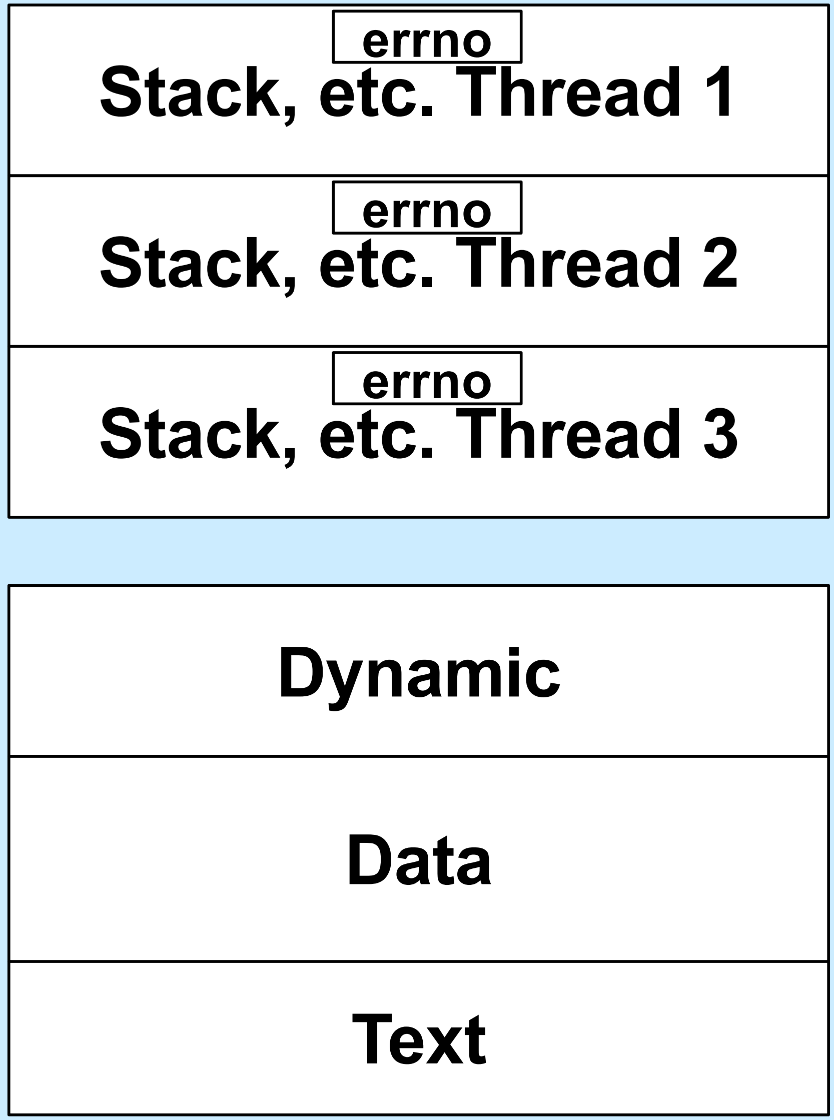
What's done???

A Solution ...

```
#define errno (*__errno_location())
```

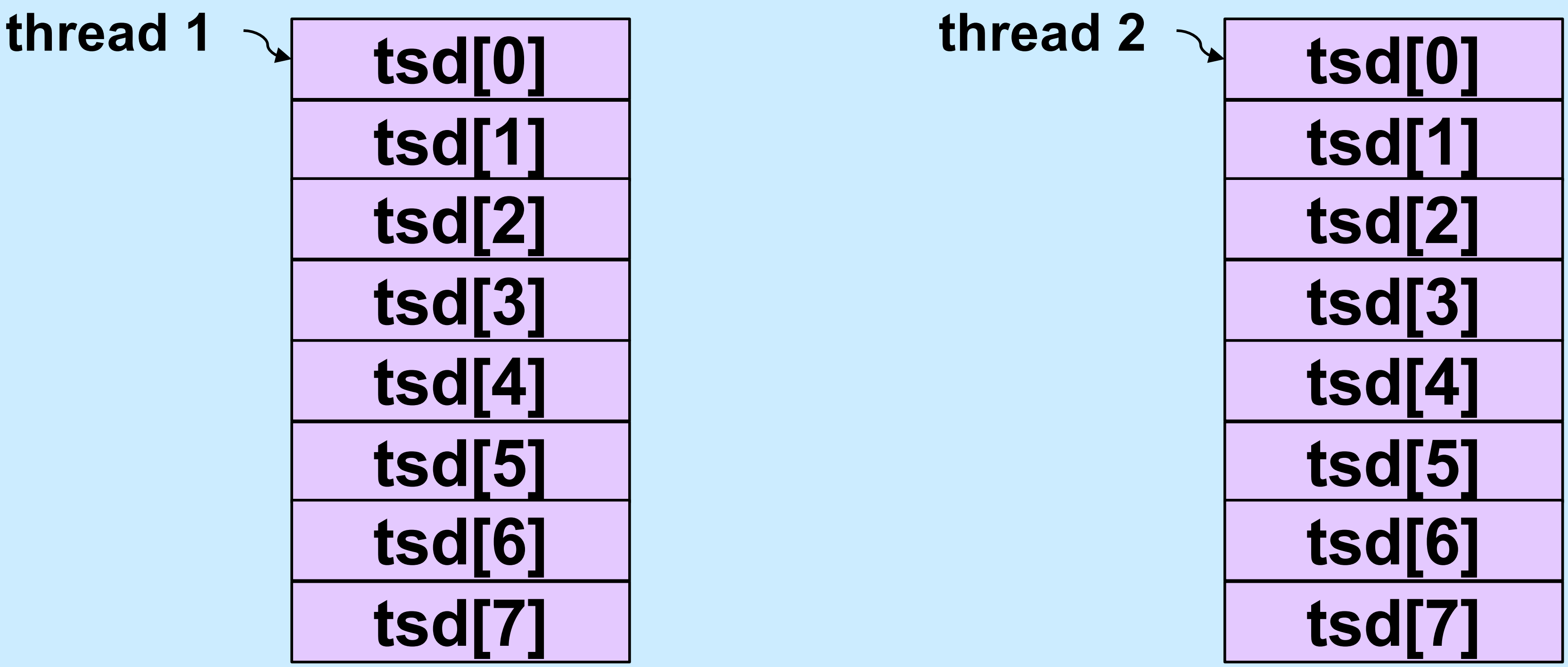
- __errno_location returns an int * that's different for each thread
  - thus each thread has, effectively, its own copy of errno

Process Address Space



Generalizing

- *Thread-specific data* (sometimes called *thread-local storage*)
  - data that's referred to by global variables, but each thread has its own private copy



Some Machinery

- `pthread_key_create(&key, cleanup_routine)`
  - **allocates a slot in the TSD arrays**
  - **provides a function to cleanup when threads terminate**
- `value = pthread_getspecific(key)`
  - **fetches from the calling thread's array**
- `pthread_setspecific(key, value)`
  - **stores into the calling thread's array**

errno (Again)

```
// executed before threads are created
pthread_key_t errno_key;
pthread_key_create(&errno_key, NULL);

// redefine errno to use thread-specific value
#define errno pthread_getspecific(errno_key);

// set current thread’s errno
pthread_set_specific(errno_key, (void *)ENOMEM);
```


Beyond POSIX
TLS Extensions for ELF and gcc

- **Thread Local Storage (TLS)**

```
__thread int x=6;
  /*
   * Each thread has its own copy of x,
   * each initialized to 6.
   * Linker and compiler do the setup.
   * May be combined with static or extern.
   * Doesn’t make sense for non-static
   * local variables!
   */
```


Example: Per-Thread Windows

```
typedef struct {
  wcontext_t win_context;
  int file_descriptor;
} win_t;
__thread static win_t my_win;

void getWindow() {
  my_win.win_context = … ;
  my_win.file_decriptor = … ;
}

int threadWrite(char *buf) {
  int status = write_to_window(
      &my_win, buf);

  return(status);
}
```

```
void *tfunc(void * arg) {
  getWindow();

  threadWrite("started");
  …

  func2(…);
}

void func2(…) {

  threadWrite(
       "important msg");
  …
}
```

Static Local Storage and Threads

```
char *strtok(char *str, const char *delim) {
    static char *saveptr;

    ...  // find next token starting at either
     ... // str or saveptr
     ... // update saveptr

    return(&token);
}
```

Coping

- Use thread local storage
- Allocate storage internally; caller frees it
- Redesign the interface

Thread-Safe Version

```
char *strtok_r(char *str, const char *delim,
               char **saveptr) {

    ...  // find next token starting at either
     ... // str or *saveptr
     ... // update *saveptr

    return(&token);
}
```

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?

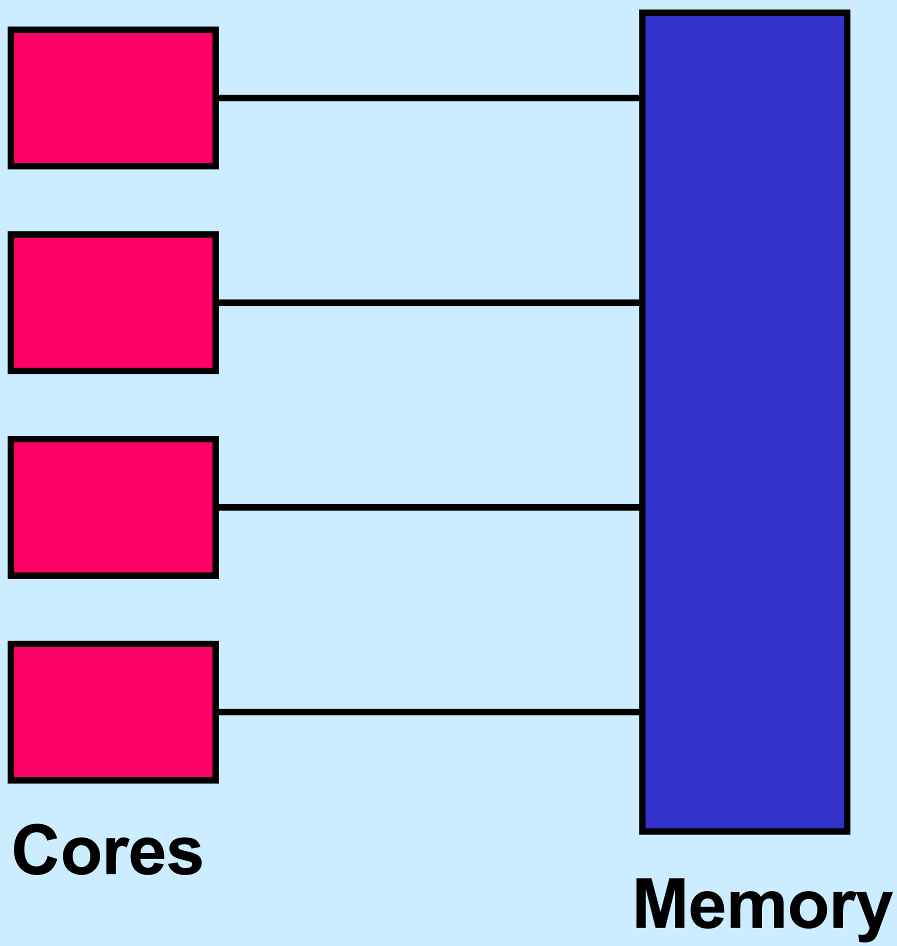
- **Everything except**

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()

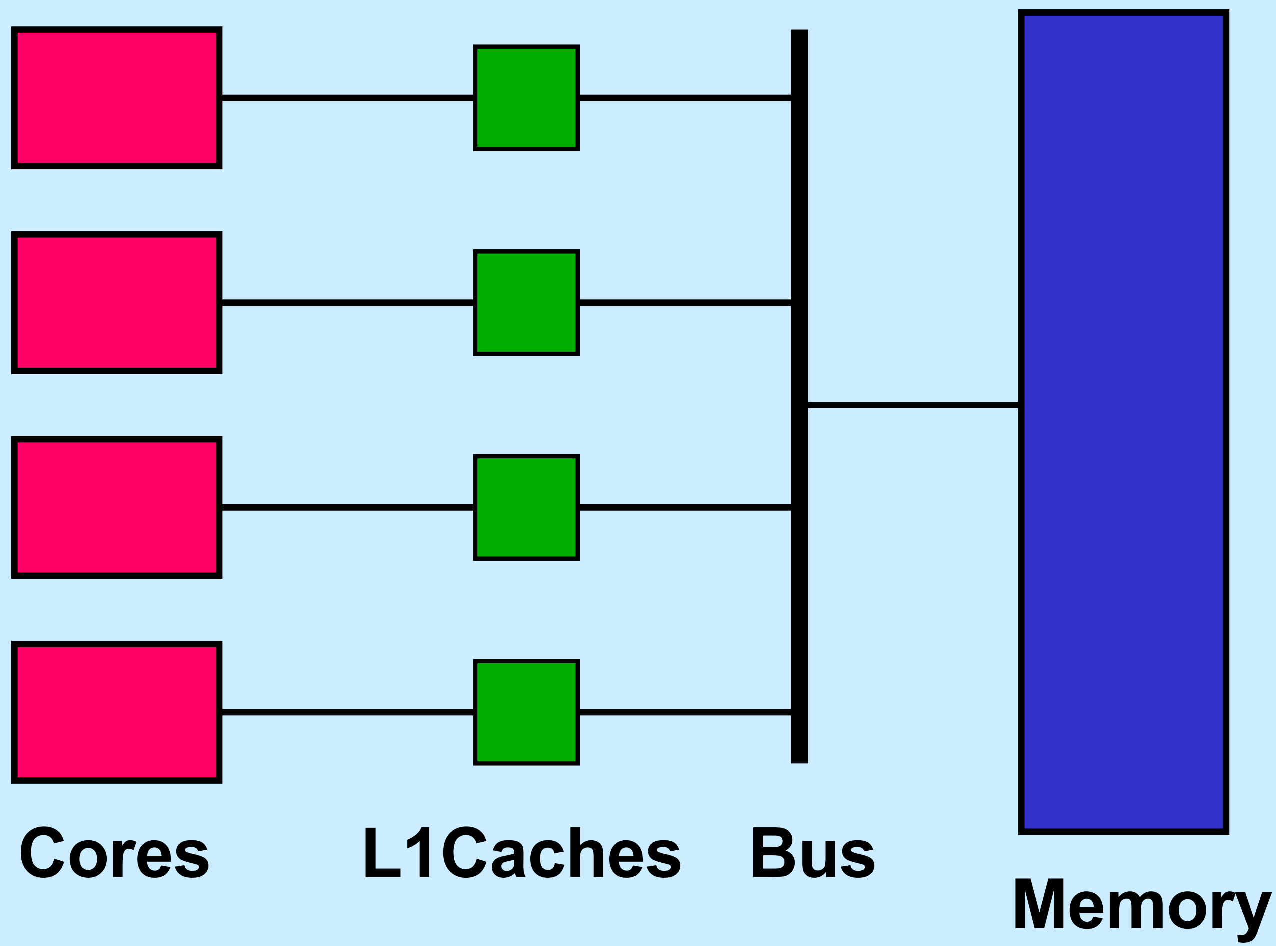
Concurrency

- **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

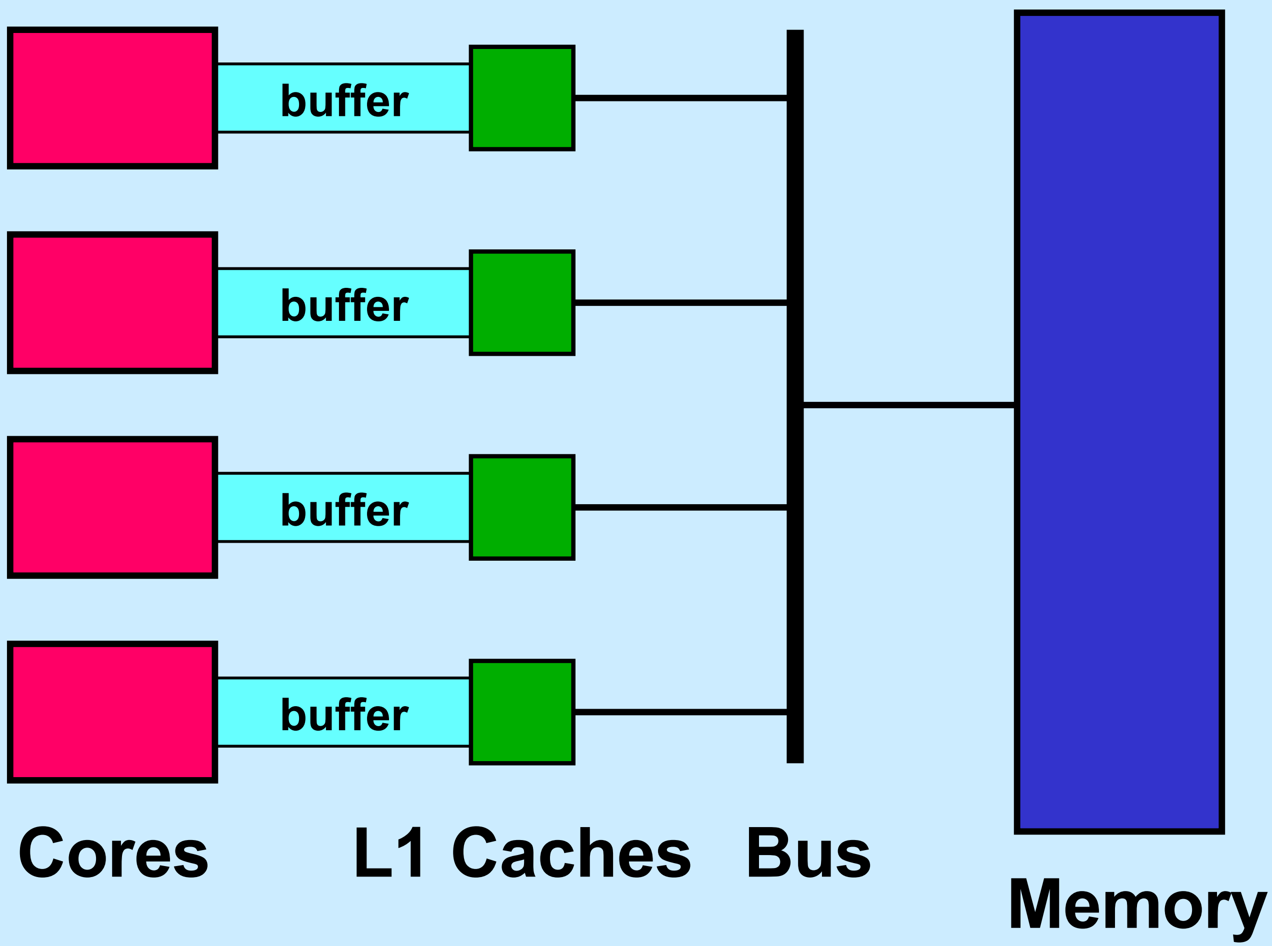
Multi-Core Processor: Simple View



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) true

b) false

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) true

b) false

Coping

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

Which Runs Faster?

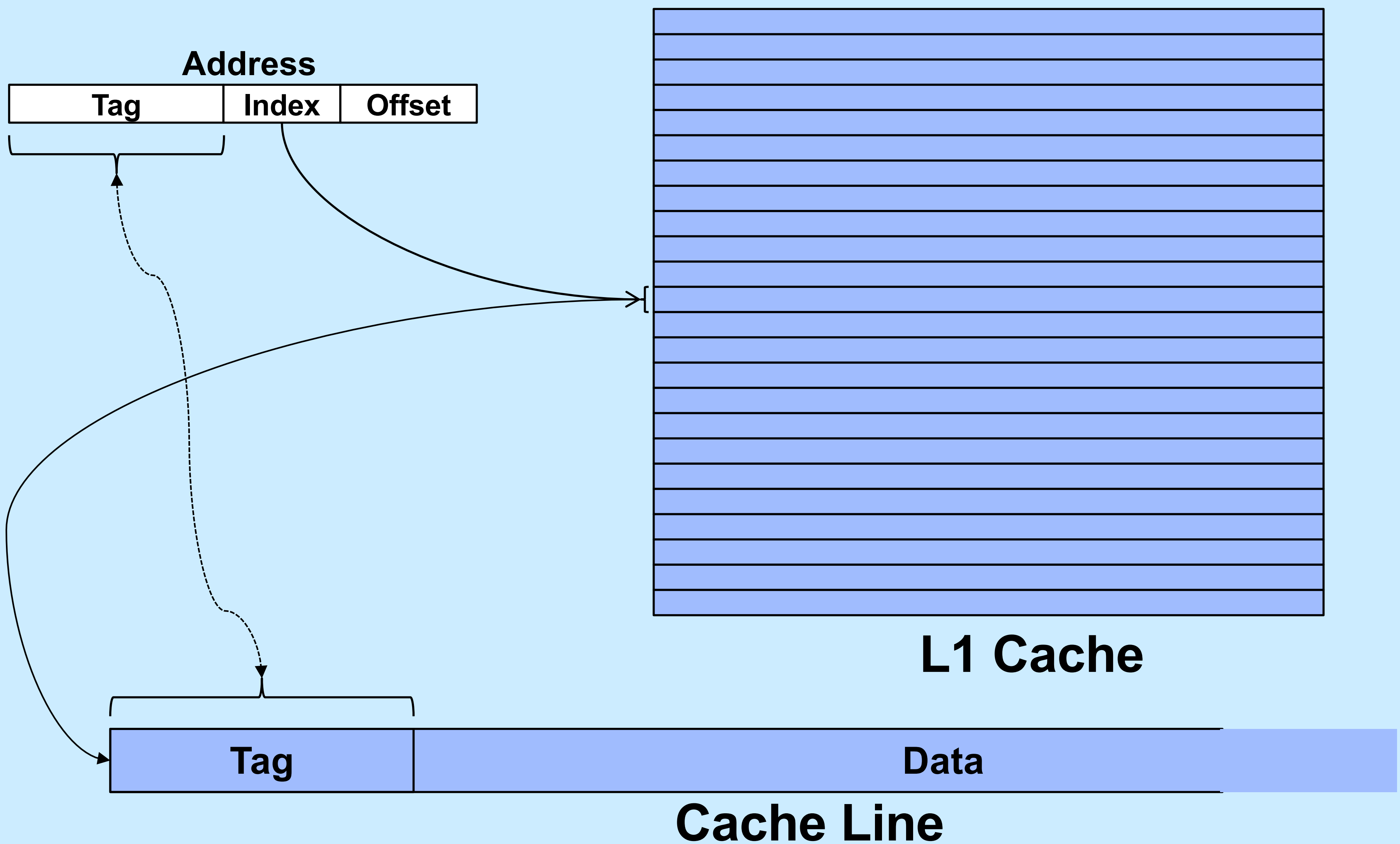
```c
volatile int a, b;

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;
  }
}
```

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

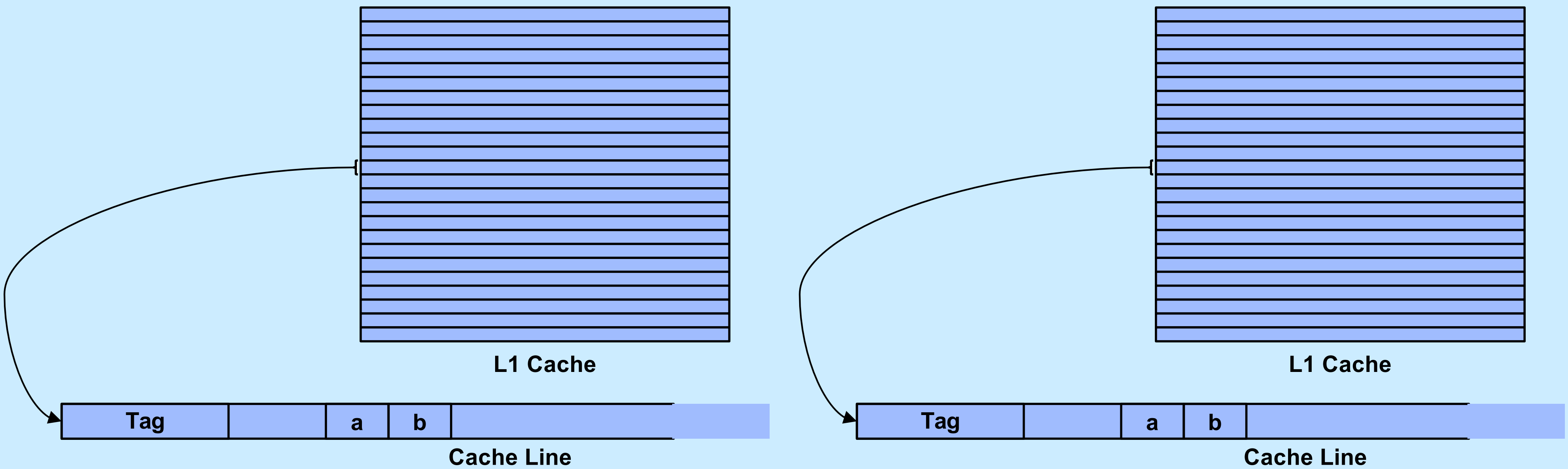
void *thread1(void *arg) {
  int i;
  for (i=0; i<reps; i++) {
    a = 1;
  }
}

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


Cache Lines



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 3

```
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);
}
```

Which of the following won't happen if the futex's value is zero and three threads call lock at the same time?

a) **one will return immediately, two will call futex_wait.**

b) **even though unlock is called appropriately, one thread will never return from futex_wait.**

c) **threads might return from futex_wait immediately, because the futex's value is not equal to c+1.**

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);
  }
}
```

