CS 33

Multithreaded Programming V

Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```

Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue,
      &s->mutex);
  pthread_mutex_unlock(&s->mutex);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```

Quiz 1

You're in charge of designing POSIX threads. Should *pthread_cond_wait* be a cancellation point?

a) no
b) yes; cancelled threads must acquire mutex before invoking cleanup handler
c) yes; but they don't acquire mutex

Start/Stop

- **Start/Stop interface**

```
void wait_for_start(state_t *s){
  pthread_mutex_lock(&s->mutex);
  pthread_cleanup_push(
    pthread_mutex_unlock, &s);
  while(s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_cleanup_pop(1);
}
void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```

Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(pthread_mutex_unlock, &m);
while(should_wait)
  pthread_cond_wait(&cv, &m);

// … (code perhaps containing other cancellation points)

pthread_cleanup_pop(1);
```

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.

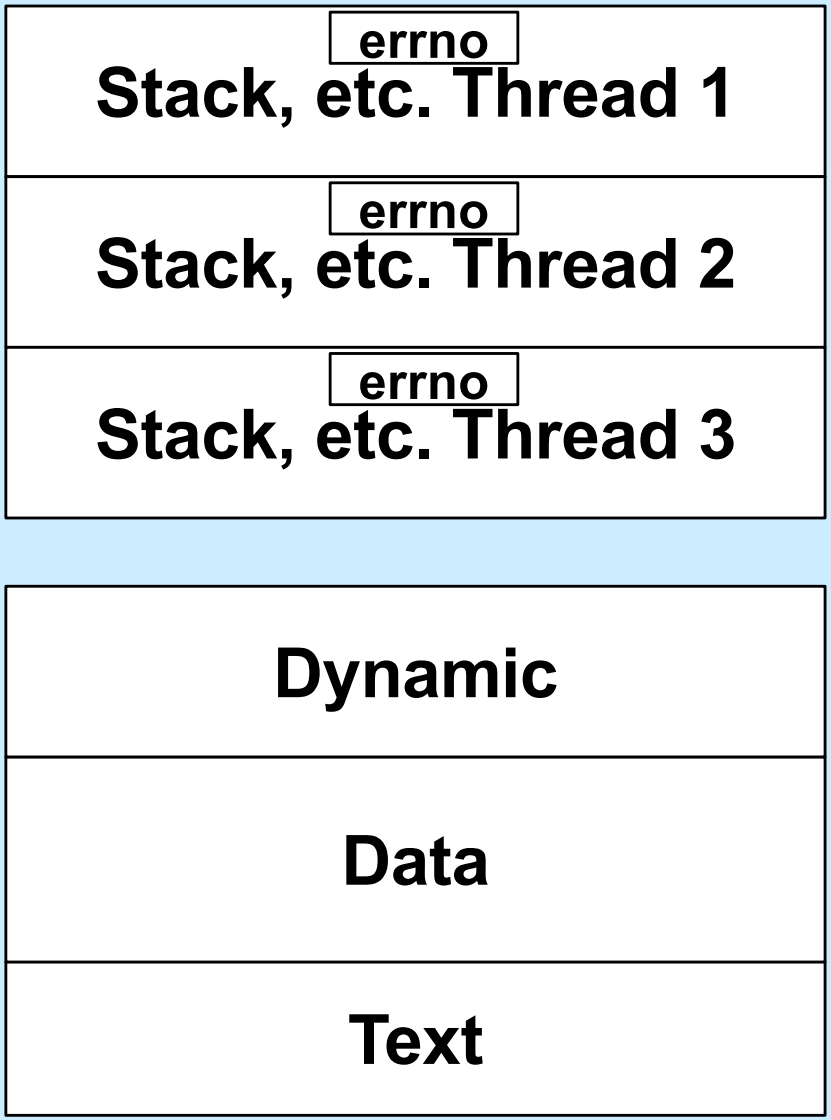
What's done???

A Solution ...

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

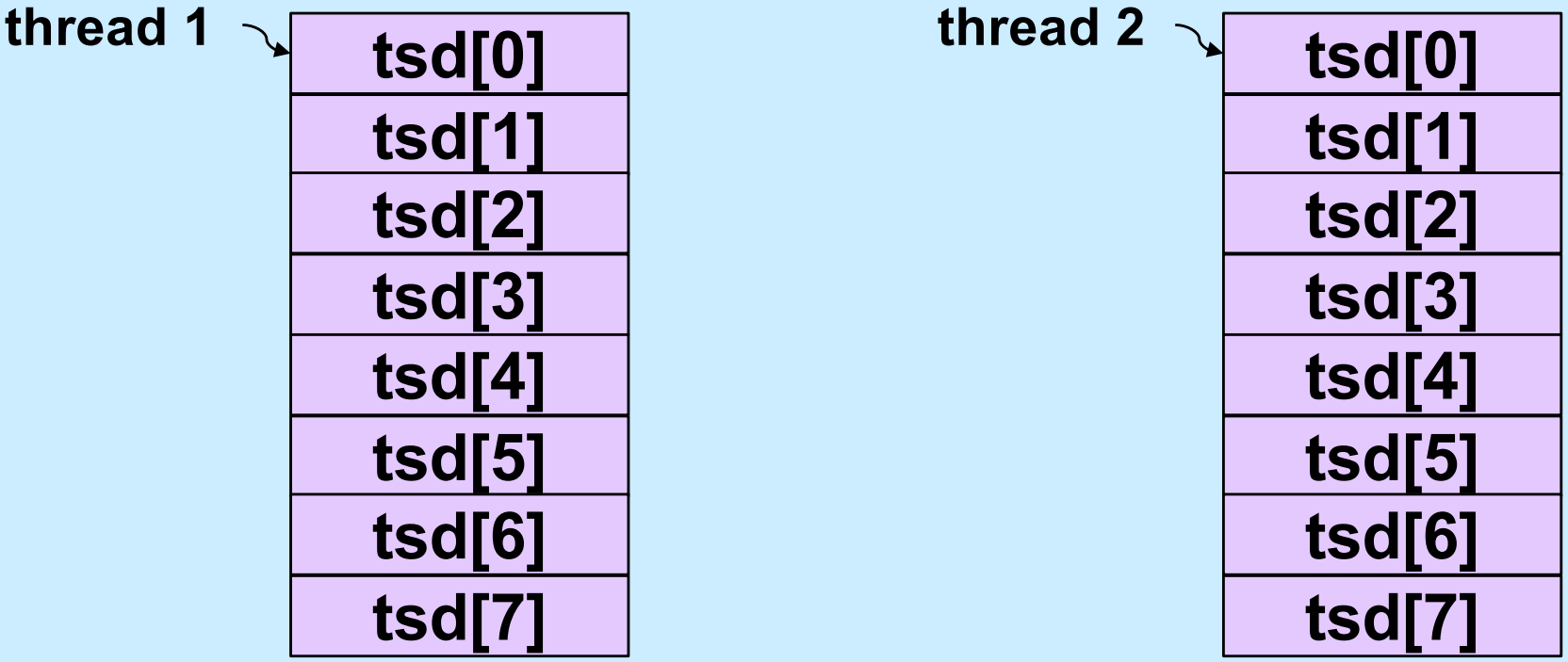
- **__errno_location returns an int * that's different for each thread**
  - **thus each thread has, effectively, its own copy of errno**

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**

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 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

```
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**

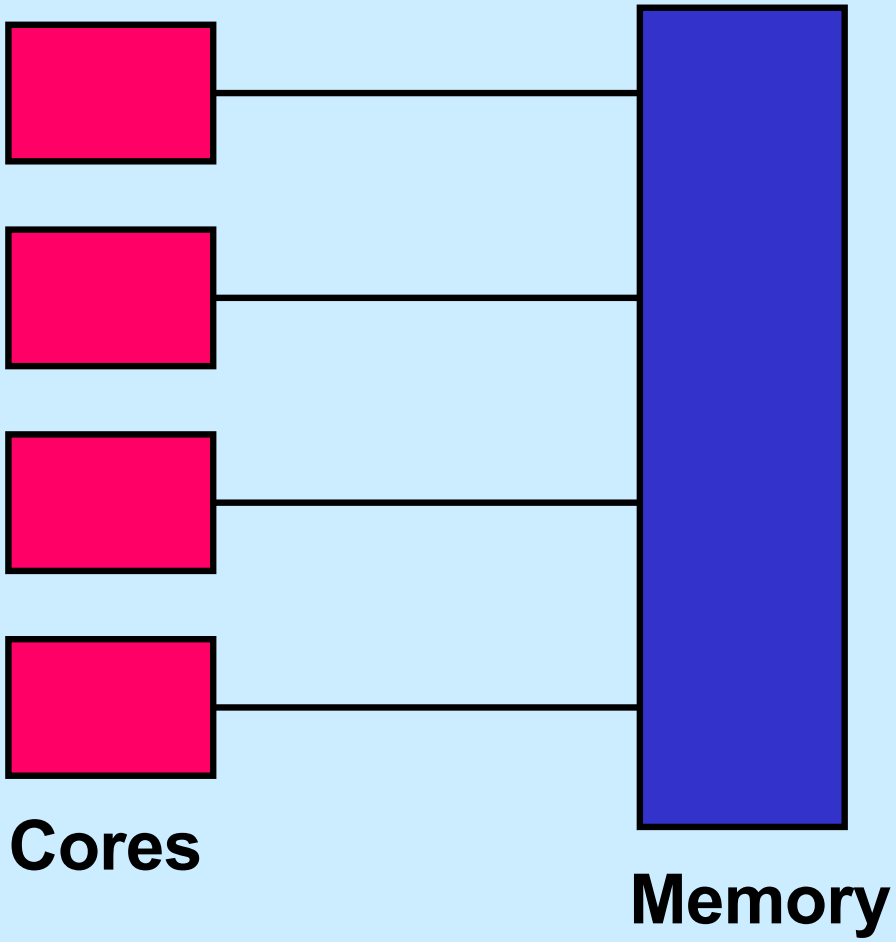
```
flockfile(stdout);
for(i=0; i<lim; i++)
  putc_unlocked(out[i]);
funlockfile(stdout);
```

What's Thread-Safe?

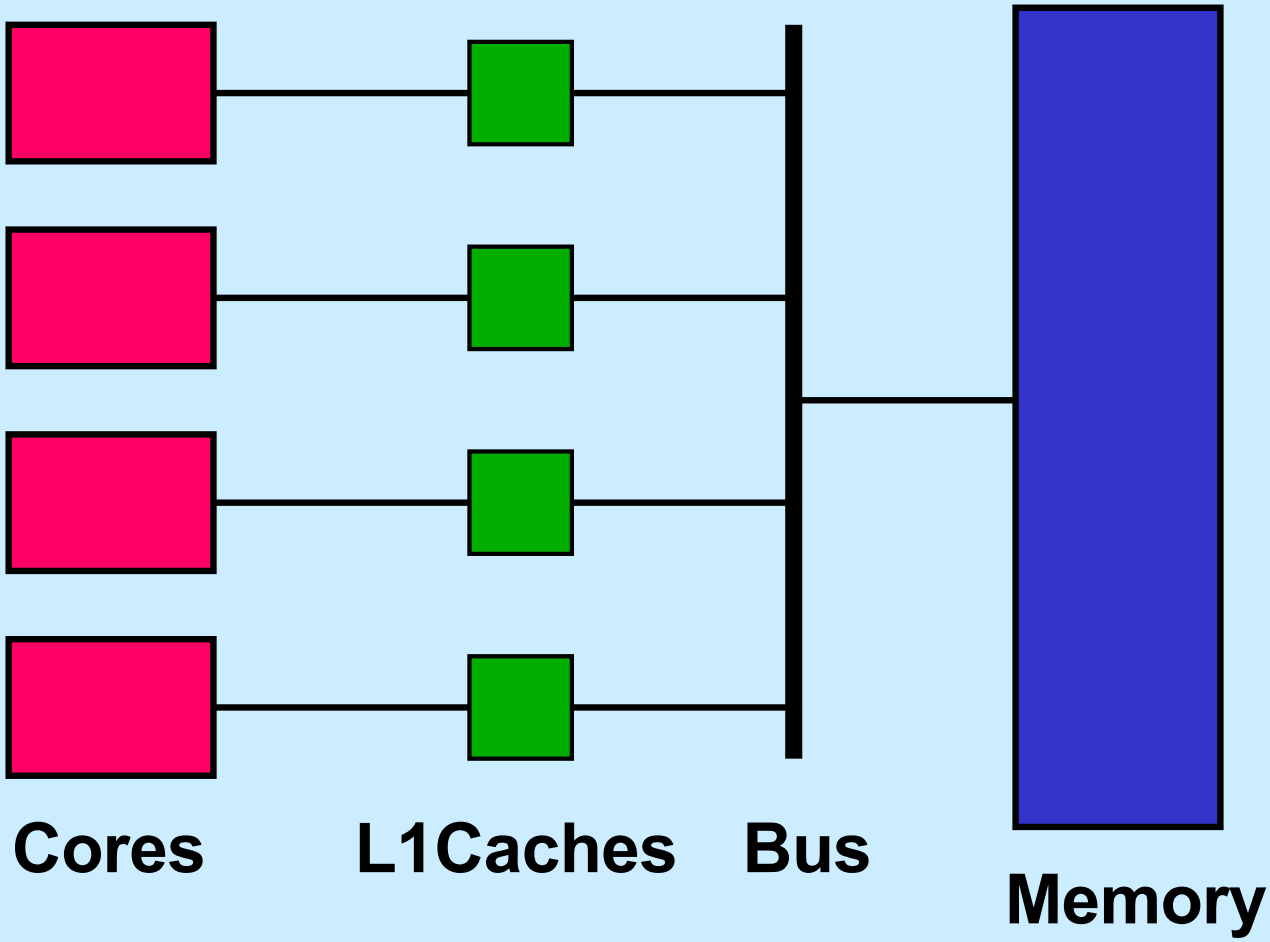
- **Everything except**

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()

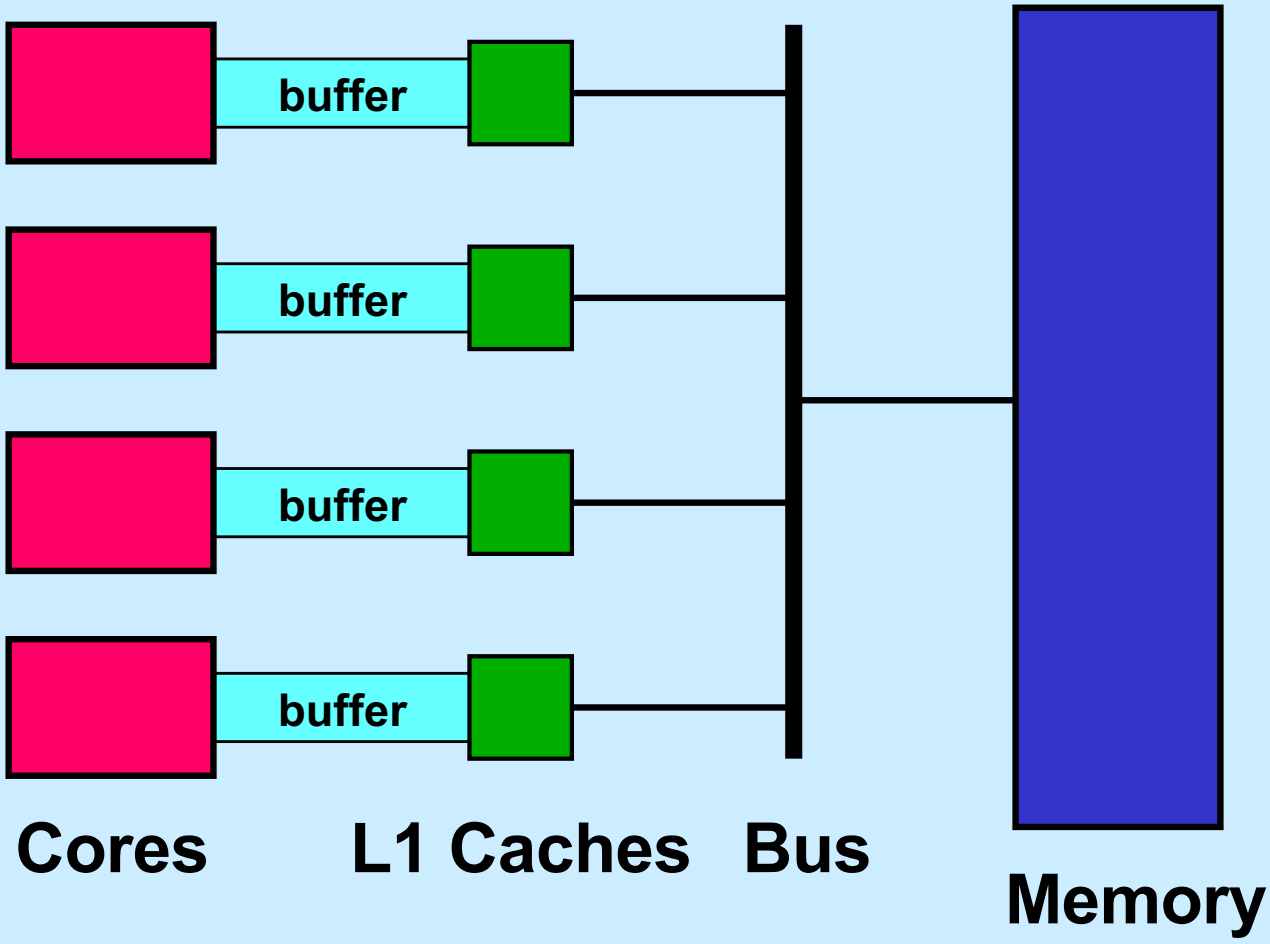
Multi-Core Processor: Simple View



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 2

```
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 machines.

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 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 sunlab machines.

a) **true**
b) **false**

Coping

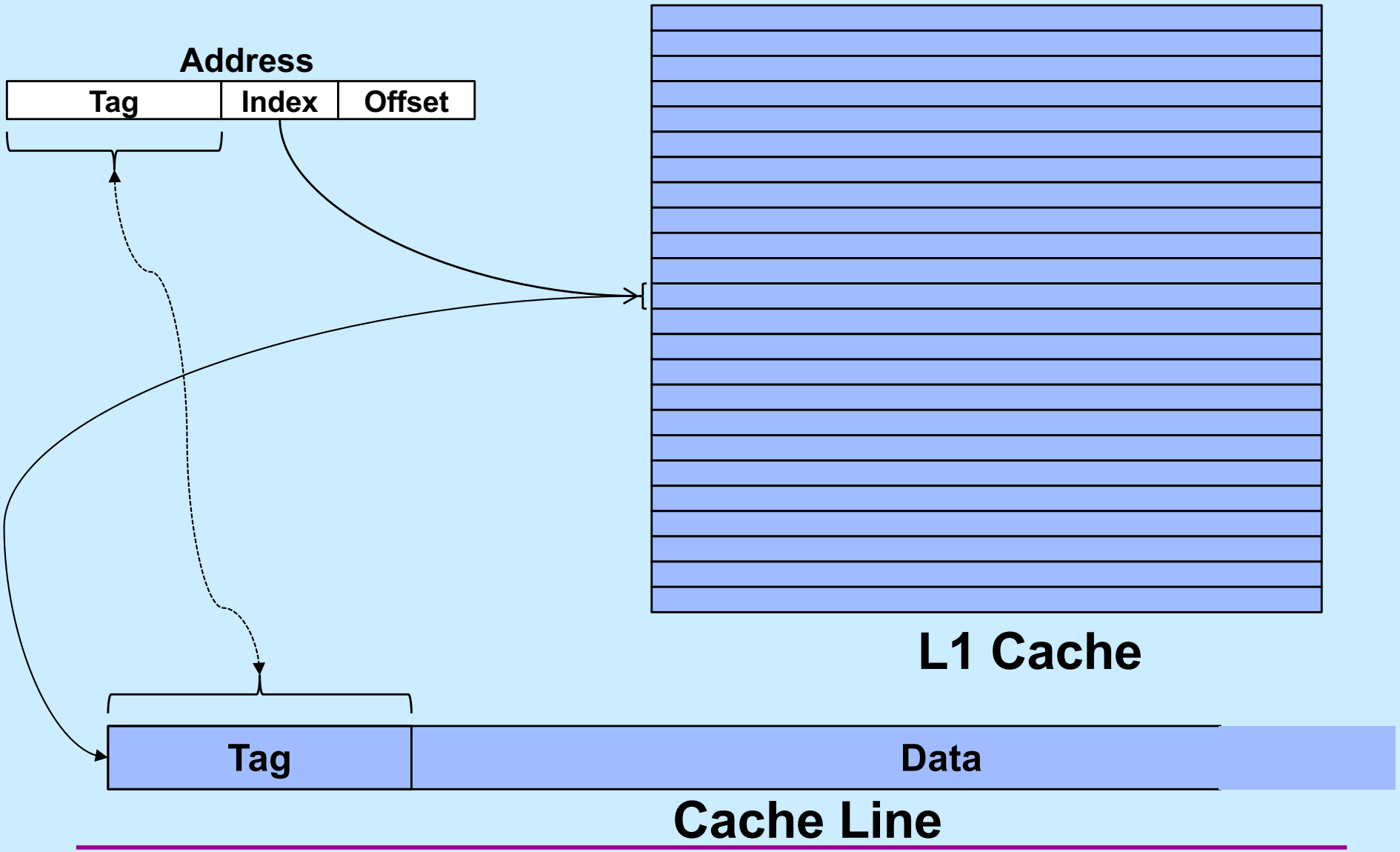
- Don’t rely on shared memory for synchronization
- Use the synchronization primitives

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



False Sharing