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.

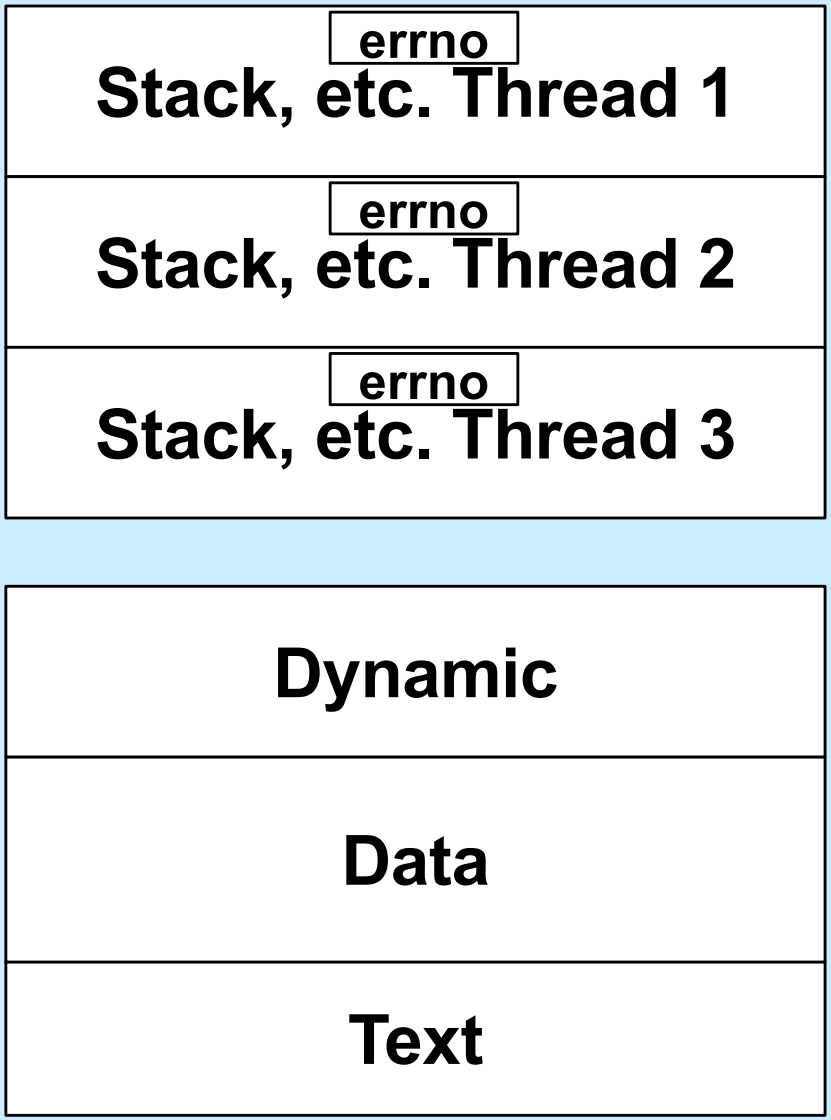
What's done???

A Solution ...

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

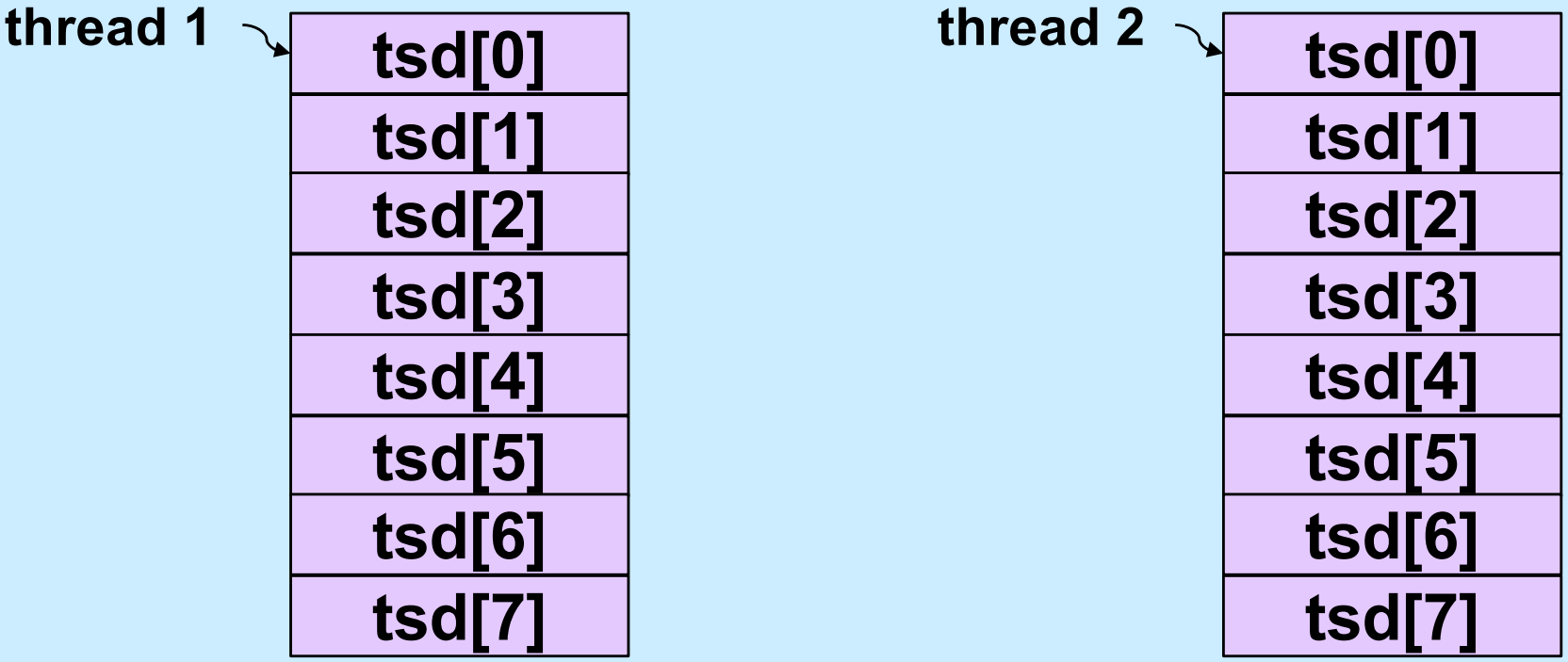
- **__errno_location returns an int * that's different for each thread**
  - **thus each thread has, effectively, its own copy of errno**

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?

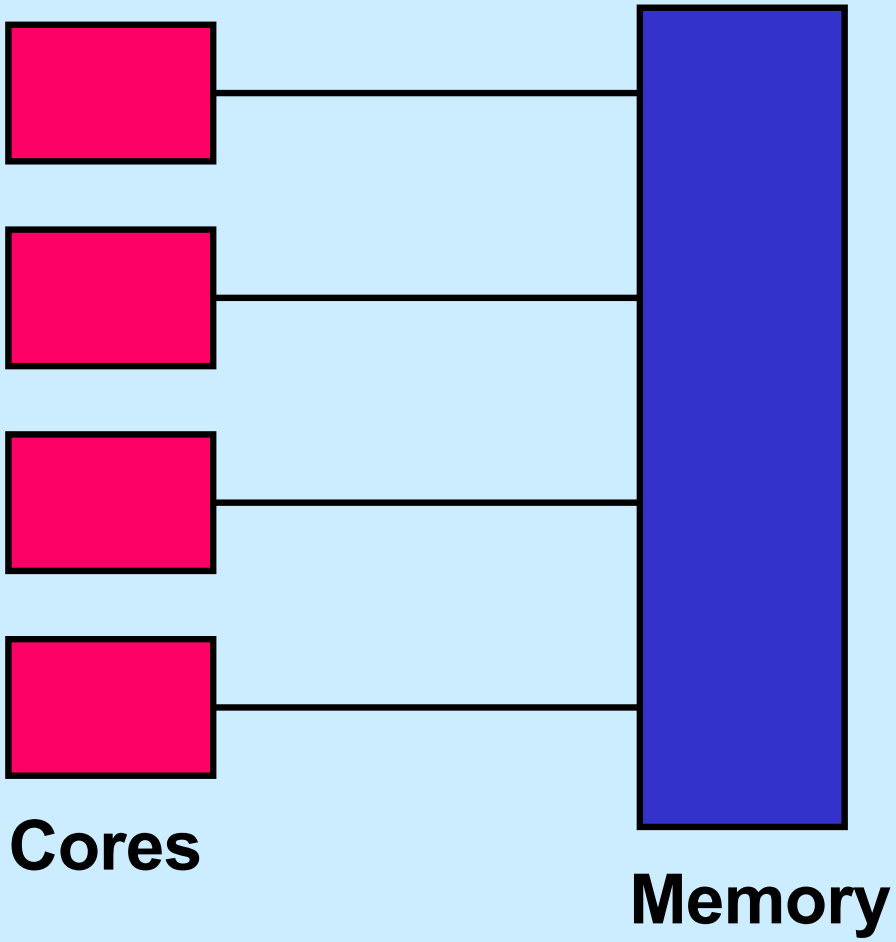
- **Everything except**

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()

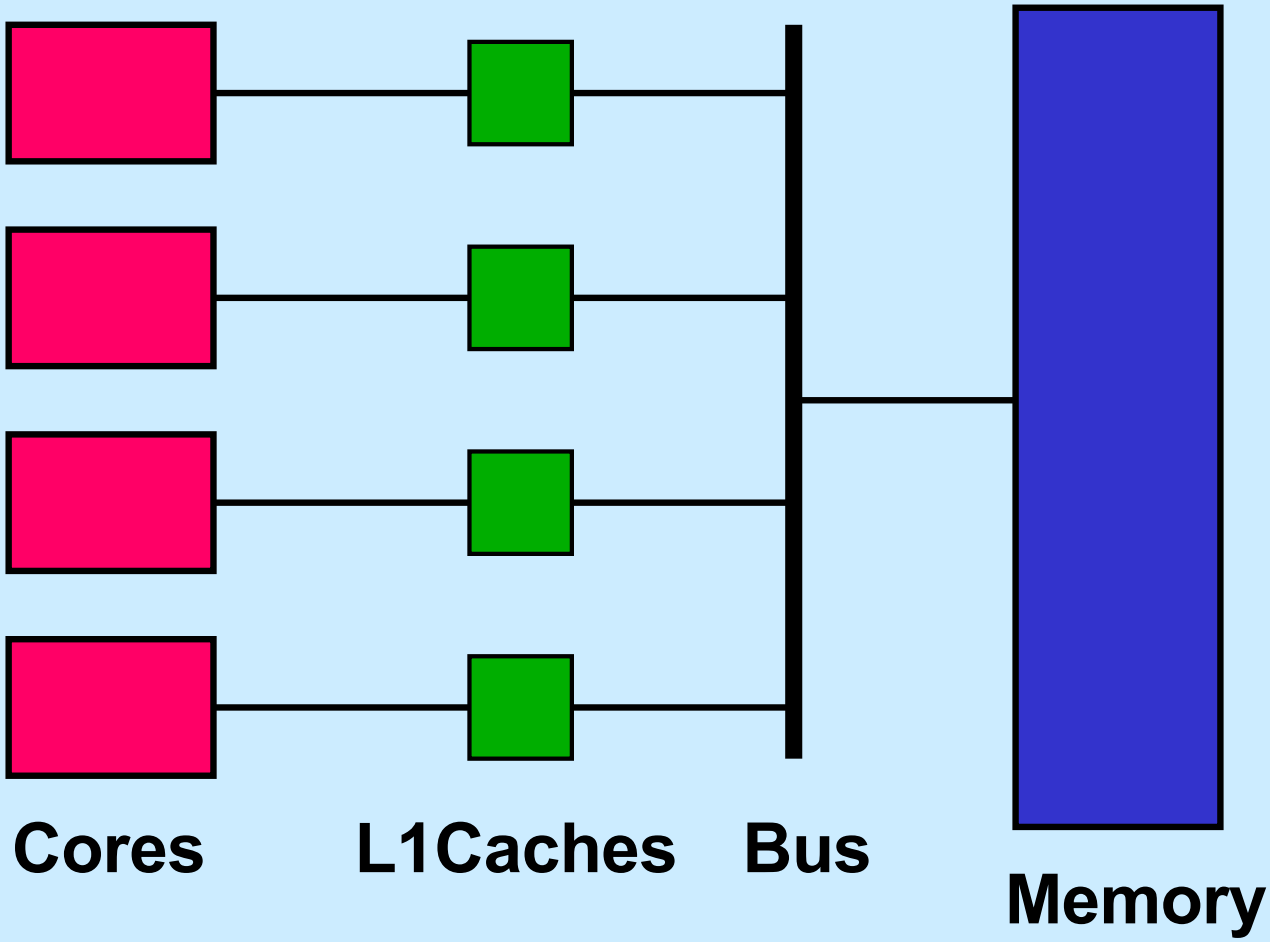
Concurrency

- 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

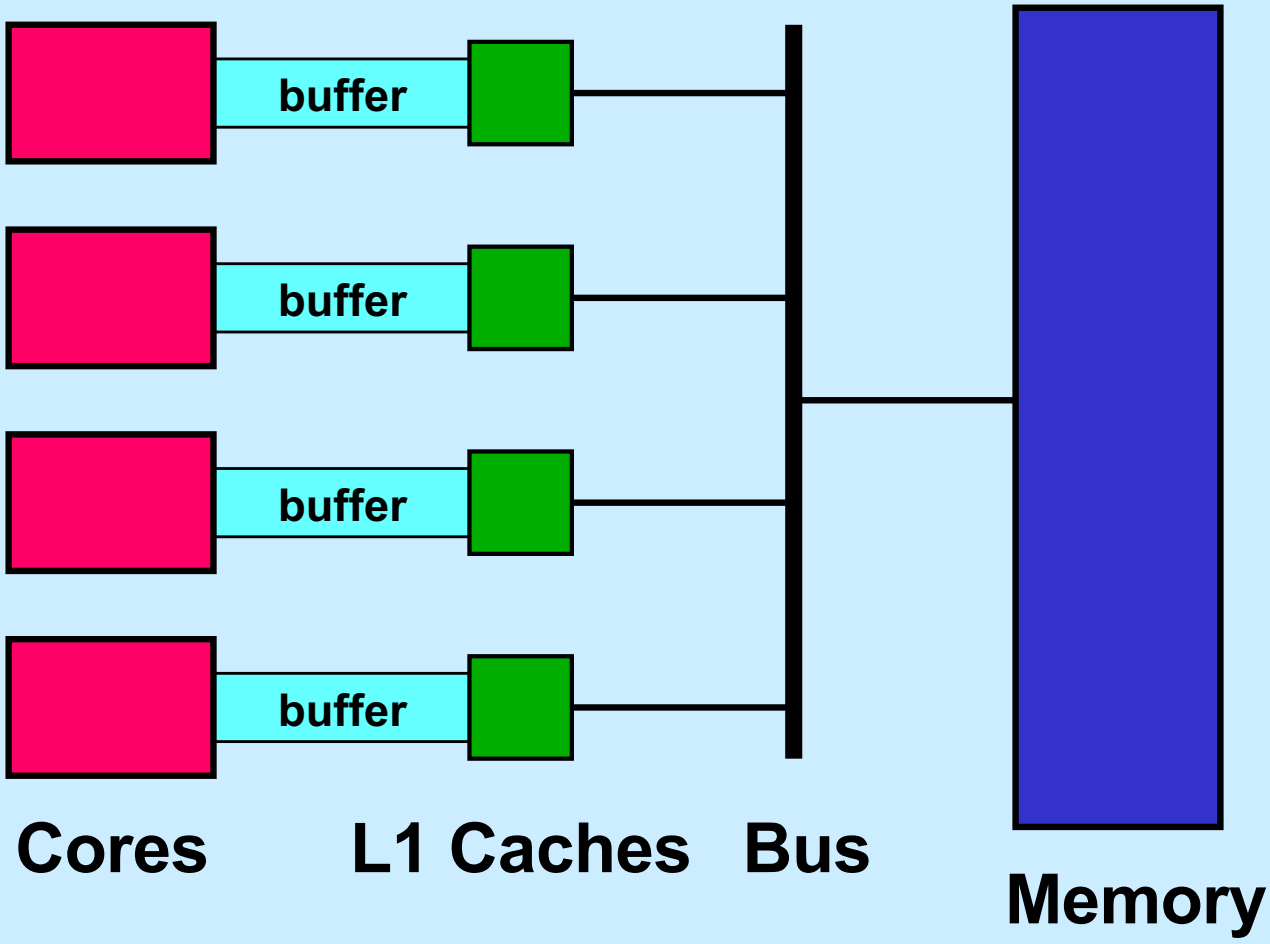
Multi-Core Processor: Simple View



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 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 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 machines.

a) true

b) false

Coping

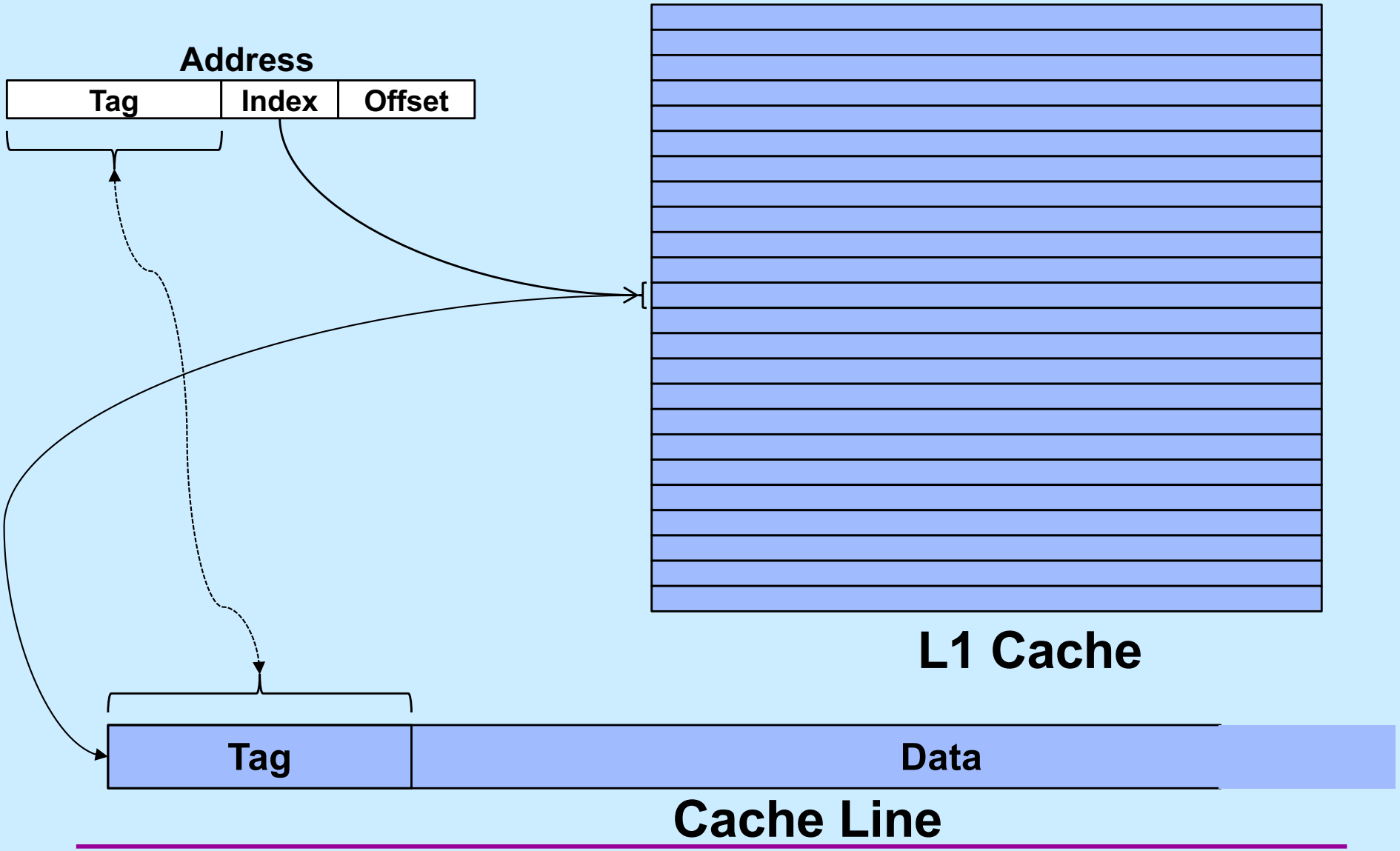
- Don’t rely on shared memory for synchronization
- Use the synchronization primitives

Which Runs Faster?

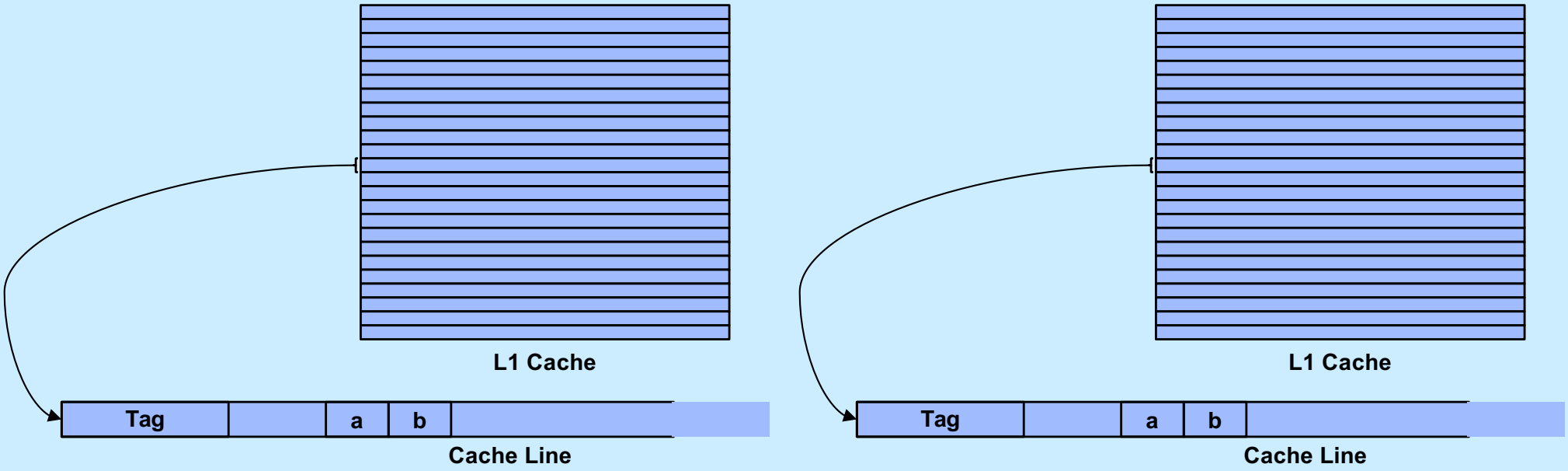
```
volatile int a, b;

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



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);
  }
}
```


Memory Allocation

- Multiple threads
- One heap

} Bottleneck?

Solution 1

- Divvy up the heap among the threads
  - each thread has its own heap
  - no mutexes required
  - no bottleneck
- How much heap does each thread get?

Solution 2

- Multiple “arenas”
  - each with its own mutex
  - thread allocates from the first one it can find whose mutex was unlocked
    - » if none, then creates new one
  - deallocations go back to original arena

Solution 3

- Global heap plus per-thread heaps
  - threads pull storage from global heap
  - freed storage goes to per-thread heap
    - » unless things are imbalanced
      - then thread moves storage back to global heap
  - mutex on only the global heap
- What if one thread allocates and another frees storage?

Malloc/Free Implementations

- ptmalloc
  - based on solution 2
  - in glibc (i.e., used by default)
- tcmalloc
  - based on solution 3
  - from Google
- Which is best?

Test Program

```
const unsigned int N=64, nthreads=32, iters=10000000;
int main() {
  void *tfunc(void *);
  pthread_t thread[nthreads];
  for (int i=0; i<nthreads; i++) {
    pthread_create(&thread[i], 0, tfunc, (void *)i);
    pthread_detach(thread[i]);
  }
  pthread_exit(0);
}
void *tfunc(void *arg) {
  long i;
  for (i=0; i<iters; i++) {
    long *p = (long *)malloc(sizeof(long)*((i%N)+1));
    free(p);
  }
  return 0;
}
```

Compiling It …

```
% gcc -o ptalloc alloc.cc -lpthread
% gcc -o tcalloc alloc.cc -lpthread -ltcmalloc
```

Running It (2014) …

```
$ time ./ptalloc
real    0m5.142s
user    0m20.501s
sys     0m0.024s
$ time ./tcalloc
real    0m1.889s
user    0m7.492s
sys     0m0.008s
```

Running It (2022) …

```
$ time ./ptalloc
real    0m1.156s
user    0m3.456s
sys     0m0.004s
$ time ./tcalloc
real    0m0.876s
user    0m3.460s
sys     0m0.004s
```

What's Going On (2014)?

```
$ strace -c -f ./ptalloc
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.040002          13      3007       520 futex
 …


$ strace -c -f ./tcalloc
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 …
  0.00    0.000000           0        59        13 futex
 …
```


What's Going On (2022)?

```
$ strace -c -f ./ptalloc
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 …
 31.23    0.019968         416        48         6 futex
 …


$ strace -c -f ./tcalloc
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 …
 0.00    0.000000           0        42         3 futex
 …
```

Test Program 2, part 1

```
#define N 64
#define npairs 16
#define allocsPerIter 1024
const long iters = 8*1024*1024/allocsPerIter;
#define BufSize 10240
typedef struct buffer {
  int *buf[BufSize];
  unsigned int nextin;
  unsigned int nextout;
  sem_t empty;
  sem_t occupied;
  pthread_t pthread;
  pthread_t cthread;
} buffer_t;
```

Test Program 2, part 2

```
int main() {
  long i;
  buffer_t b[npairs];
  for (i=0; i<npairs; i++) {
    b[i].nextin = 0;
    b[i].nextout = 0;
    sem_init(&b[i].empty, 0, BufSize/allocsPerIter);
    sem_init(&b[i].occupied, 0, 0);
    pthread_create(&b[i].pthread, 0, prod, &b[i]);
    pthread_create(&b[i].cthread, 0, cons, &b[i]);
  }
  for (i=0; i<npairs; i++) {
    pthread_join(b[i].pthread, 0);
    pthread_join(b[i].cthread, 0);
  }
  return 0;
}
```

Test Program 2, part 3

```
void *prod(void *arg) {
  long i, j;
  buffer_t *b = (buffer_t *)arg;
  for (i = 0; i<iters; i++) {
    sem_wait(&b->empty);
    for (j = 0; j<allocsPerIter; j++) {
      b->buf[b->nextin] = malloc(sizeof(int)*((j%N)+1));
      if (++b->nextin >= BufSize)
        b->nextin = 0;
    }
    sem_post(&b->occupied);
  }
  return 0;
}
```


Test Program 2, part 4

```
void *cons(void *arg) {
  long i, j;
  buffer_t *b = (buffer_t *)arg;
  for (i = 0; i<iters; i++) {
    sem_wait(&b->occupied);
    for (j = 0; j<allocsPerIter; j++) {
      free(b->buf[b->nextout]);
      if (++b->nextout >= BufSize)
        b->nextout = 0;
    }
    sem_post(&b->empty);
  }
  return 0;
}
```

Running It (2014) …

```
$ time ./ptalloc2
real    0m1.087s
user    0m3.744s
sys     0m0.204s
$ time ./tcalloc2
real    0m3.535s
user    0m11.361s
sys     0m2.112s
```

Running It (2022) …

```
$ time ./ptalloc2
real    0m0.367s
user    0m1.187s
sys     0m0.179s
$ time ./tcalloc2
real    0m0.426s
user    0m1.211s
sys     0m0.290s
```

What's Going On (2014)?

```
$ strace -c -f ./ptalloc2
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 94.96    2.347314          44     53653     14030 futex
…
$ strace -c -f ./tcalloc2
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 93.86    6.604632          36    185731     45222 futex
 …
```


What's Going On (2022)?

```
$ strace -c -f ./ptalloc2
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
92.26    4.544802          66     68250     13340 futex
…
$ strace -c -f ./tcalloc2
 …
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 91.40    3.439416          52     65165     12182 futex
 …
```

