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!
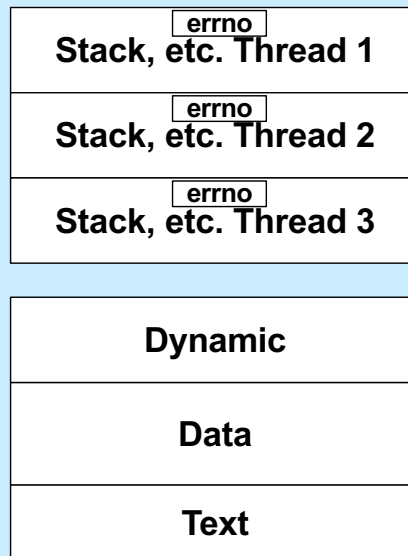
However, somehow it works.

What's done???

A Solution ...

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

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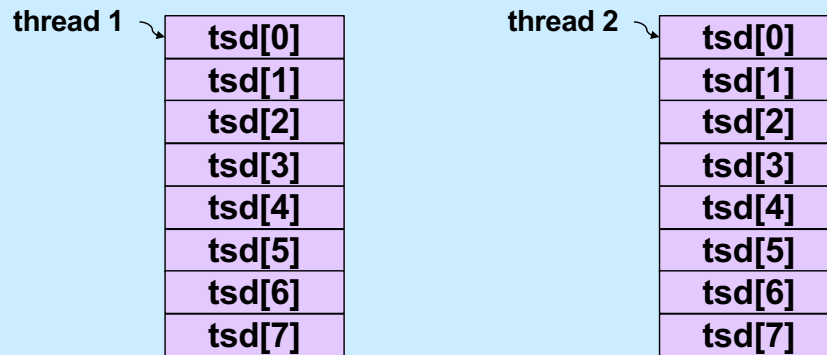   - **thus each thread has, effectively, its own copy of errno**

When you give gcc the –pthread flag, it, among other things, defines some preprocessor variables that cause some code in the standard header files to be compiled (that otherwise wouldn't be). In particular the #define statement given in the slide is compiled.

Process Address Space

errno
Stack, etc. Thread 1

errno
Stack, etc. Thread 2

errno
Stack, etc. Thread 3

Dynamic
Data
Text

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**

thread 1

tsd[0]
tsd[1]
tsd[2]
tsd[3]
tsd[4]
tsd[5]
tsd[6]
tsd[7]

thread 2

tsd[0]
tsd[1]
tsd[2]
tsd[3]
tsd[4]
tsd[5]
tsd[6]
tsd[7]

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**

So that we can be certain that it's the calling thread's array that is accessed, rather than access the TSD array directly, one uses a set of POSIX threads library routines. To find an unused slot, one calls *pthread_key_create*, which returns the index of an available slot in its first argument. Its second argument is the address of a routine that's automatically called when the thread terminates, so as to do any cleanup that might be necessary (it's called with the key (index) as its sole argument, and is called only if the thread has actually stored a non-null value into the slot). To put a value in a slot, i.e., perform the equivalent of TSD[i] = x, one calls *pthread_setspecific(i,x)*. To fetch from the slot, one calls *pthread_getspecific(i)*.

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);
```

Using the thread-specific data functions we can create an alternative (but equivalent) implementation of thread-specific errno.

Before any threads (other than the original thread in the process) are created, pthread_key_create is called to initialize errno_key, a global variable. We define errno to be the value returned by pthread_getspecific – the calling thread's value of errno. Then to set a thread's errno, we use pthread_setspecific.

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!
   */
```

ELF stands for "executable and linking format" and is the standard format for executable and object files used on most Unix systems. The __**thread** attribute tells gcc that each thread is to have its own copy of the variable. A detailed description of how it is implemented can be found at http://people.redhat.com/drepper/tls.pdf.

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");
  …
}
```

In this example, we put together per-thread windows for thread output. Threads call **getWindow** to set up a window for their exclusive use, then call **threadWrite** to send output to their windows. Individual threads can now set up their own windows and write to them without having to pass around information describing which are their windows. Each thread's window is referred to by the static global variable **my_win**.

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);
}
```

An example of the single-thread mentality in early Unix is the use of static local storage in a number of library routines. An example of this is **strtok**, which saves a pointer into the input string for use in future calls to the function (which we've used extensively in earlier projects). This works fine as long as just one thread is using the function, but fails if multiple threads use it – each will expect to find its own saved pointer in **saveptr**, but there's only one **saveptr**.

Coping

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

As the slide shows, there are at least three techniques for coping with this problem. We could use thread-local storage, but this would entail associating a fair amount of storage with each thread, even if it is not using **strtok**. We might simply allocate storage (via **malloc**) inside **strtok** and return a pointer to this storage. The problem with this is that the calls to **malloc** and **free** could turn out to be expensive. Furthermore, this makes it the caller's responsibility to free the storage, introducing a likely storage leak.

The solution taken is to redesign the interface. The "thread-safe" version is called **strtok_r** (the **r** stands for **reentrant**, an earlier term for "thread-safe"); it takes an additional parameter pointing to storage that holds **saveptr**. Thus the caller is responsible for both the allocation and the liberation of the storage containing **saveptr**; this storage is typically a local variable (allocated on the stack), so that its allocation and liberation overhead is negligible, at worst.

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);
}
```

Here's the thread-safe version of **strtok**.

Shared Data

- **Thread 1:**
  ```
  printf("goto statement reached");
  ```
- **Thread 2:**
  ```
  printf("Hello World\n");
  ```

- **Printed on display:**

  go to Hell

Yet another problem that arises when using libraries that were not designed for multithreaded programs concerns synchronization. The slide shows what might happen if one relied on the single-threaded versions of the standard I/O routines.

Coping

- **Wrap library calls with synchronization constructs**
- **Fix the libraries**

To deal with this **printf** problem, we must somehow add synchronization to **printf** (and all of the other standard I/O functions). A simple way to do this would be to supply wrappers for all of the standard I/O functions ensuring that only one thread is operating on any particular stream at a time. A better way would be to do the same sort of thing by fixing the functions themselves, rather than supplying wrappers (this is what is done in most implementations).

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()

After making a library thread-safe, we may discover that many functions have become too slow. For example, the standard-I/O functions **getc** and **putc** are expected to be fast — they are usually implemented as macros. But once we add the necessary synchronization, they become rather sluggish — much too slow to put in our innermost loops. However, if we are aware of and willing to cope with the synchronization requirements ourselves, we can produce code that is almost as efficient as the single-threaded code without synchronization requirements.

The POSIX-threads specification includes unsynchronized versions of **getc** and **putc** — **getc_unlocked** and **putc_unlocked**. These are exactly the same code as the single-threaded **getc** and **putc**. To use these new functions, one must take care to handle the synchronization oneself. This is accomplished 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()	ecvt()	gethostent()	getutxline()	putc_unlocked()
basename()	encrypt()	getlogin()	gmtime()	putchar_unlocked()
catgets()	endgrent()	getnetbyaddr()	hcreate()	putenv()
crypt()	endpwent()	getnetbyname()	hdestroy()	pututxline()
ctime()	endutxent()	getnetent()	hsearch()	rand()
dbm_clearerr()	fcvt()	getopt()	inet_ntoa()	readdir()
dbm_close()	ftw()	getprotobyname()	l64a()	setenv()
dbm_delete()	gcvt()	getprotobynumber()	lgamma()	setgrent()
dbm_error()	getc_unlocked()	getprotoent()	lgammaf()	setkey()
dbm_fetch()	getchar_unlocked()	getpwent()	lgammal()	setpwent()
dbm_firstkey()	getdate()	getpwnam()	localeconv()	setutxent()
dbm_nextkey()	getenv()	getpwuid()	localtime()	strerror()
dbm_open()	getgrent()	getservbyname()	lrand48()	strtok()
dbm_store()	getgrgid()	getservbyport()	mrand48()	ttyname()
dirname()	getgrnam()	getservent()	nftw()	unsetenv()
dlerror()	gethostbyaddr()	getutxent()	nl_langinfo()	wcstombs()
drand48()	gethostbyname()	getutxid()	ptsname()	wctomb()

According to IEEE Std. 1003.1 (POSIX), all functions it specifies must be thread-safe, except for those listed above.

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**
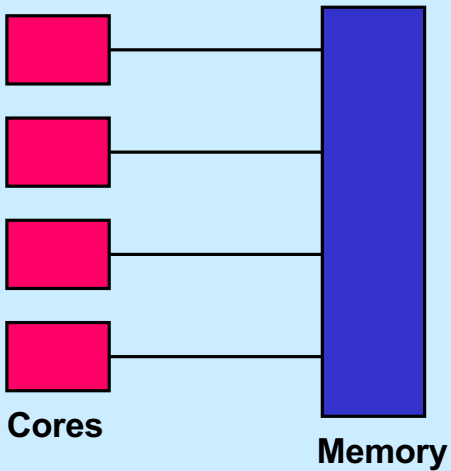
Cores

Memory

This slide illustrates the common view of the architecture of a multi-core processor: a number of processors are all directly connected to the same memory (which they share). If one core (or processor) stores into a storage location and immediately thereafter another core loads from the same storage location, the second core loads exactly what the first core stored.

Unfortunately, as we learned earlier in the course, things are not quite so simple.

**Multi-Core Processor:
More Realistic View**

Cores L1Caches Bus Memory
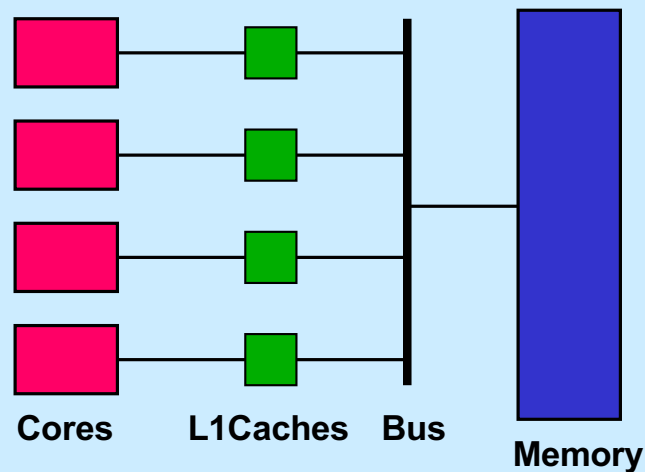
Real multi-core processors have L1 caches that sit between each core and the memory bus; there is a single connection between the bus and the memory. When a core issues a store, the store affects the L1 cache. When a core issues a load, the load is dealt with by the L1 cache if possible, and otherwise goes to memory (perhaps via a shared L2 cache). Most architectures have some sort of cache-consistency logic to insure that the shared-memory semantics of the previous page are preserved.

However, again as we learned earlier in the course, even this description is too simplistic.

Multi-Core Processor: Even More Realistic

Cores   **L1 Caches**   **Bus**

Memory
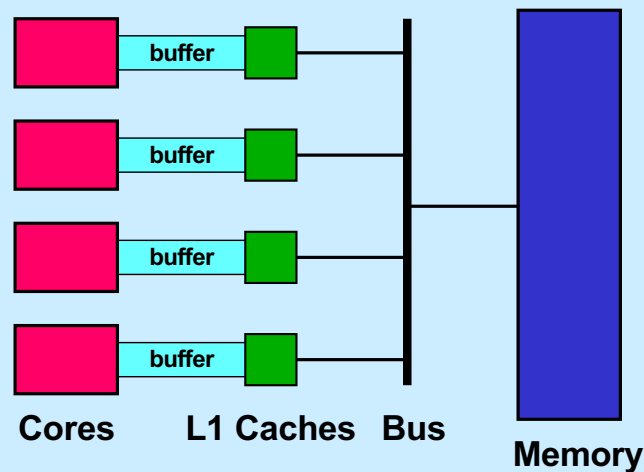
This slide shows an even more realistic model, pretty much the same as what we saw is actually used in recent Intel processors. Between each core and the L1 cache is a buffer. Stores by a core go into the buffer. Sometime later the effect of the store reaches the L1 cache. In the meantime, the core is issuing further instructions. Loads by the core are handled from the buffer if the data is still there; otherwise they go to the L1 cache, and then perhaps to memory.

In all instances of this model the effect of a store, as seen by other cores, is delayed. In some instances of this model the order of stores made by one core might be perceived differently by other cores. Architectures with the former property are said to have **delayed stores**; architectures with the latter are said to have **reordered stores** (an architecture could well have both properties).

Concurrent Reading and Writing

Thread 1:   **Thread 2:**

```
i = shared_counter;     shared_counter++;
```

In this example, one thread running on one processor is loading from an integer in storage; another thread running on another processor is loading from and then storing into an integer in storage. Can this be done safely without explicit synchronization?

On most architectures, the answer is yes. If the integer in question is aligned on a natural (e.g., eight-byte) boundary, then the hardware (perhaps the cache) insures that loads and stores of the integer are atomic.

However, one cannot assume that this is the case on all architectures. Thus a portable program must use explicit synchronization (e.g., a mutex) in this situation.

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;
}
```

Shown on the slide is Peterson's algorithm for handling mutual exclusion for two threads without explicit synchronization. (The **me** argument for one thread is 0 and for the other is 1.) This program works given the first two shared-memory models. Does it work with delayed-store architectures?

The algorithm is from "Myths About the Mutual Exclusion Problem," by G. L. Peterson, Information Processing Letters 12(3) 1981: 115–116.

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

Sunlab computers (as do most modern computers) employ the delayed-store architecture.

Busy-Waiting Producer/Consumer

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

  buf[in%BSIZE] = item;             item = buf[out%BSIZE];

  in++;                             out++;
}
                                    return(item);
                                  }
```

This example is a solution, employing "busy waiting," to the producer-consumer problem for one consumer and one producer.

This solution to the producer-consumer problem is from "Proving the Correctness of Multiprocess Programs," by L. Lamport, IEEE Transactions on Software Engineering, SE-3(2) 1977: 125-143.

Quiz 2

```
void producer(char item) {        char consumer( ) {
                                    char item;
  while(in – out == BSIZE)           while(in – out == 0)
   ;                                   ;

  buf[in%BSIZE] = item;              item = buf[out%BSIZE];

  in++;                              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**

The point of the previous several slides is that one cannot rely on expected properties of shared memory to eliminate explicit synchronization. Shared memory can behave in some very unexpected ways. However, it is the responsibility of the implementers of the various synchronization primitives to make certain not only that they behave correctly, but also that they synchronize memory with respect to other threads.

Which Runs Faster?

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

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

void *thread2(void *arg) {
  int i;                              void *thread2(void *arg) {
  for (i=0; i<reps; i++) {             int i;
    b = 1;                             for (i=0; i<reps; i++) {
  }                                      b = 1;
}                                        }
                                      }
```
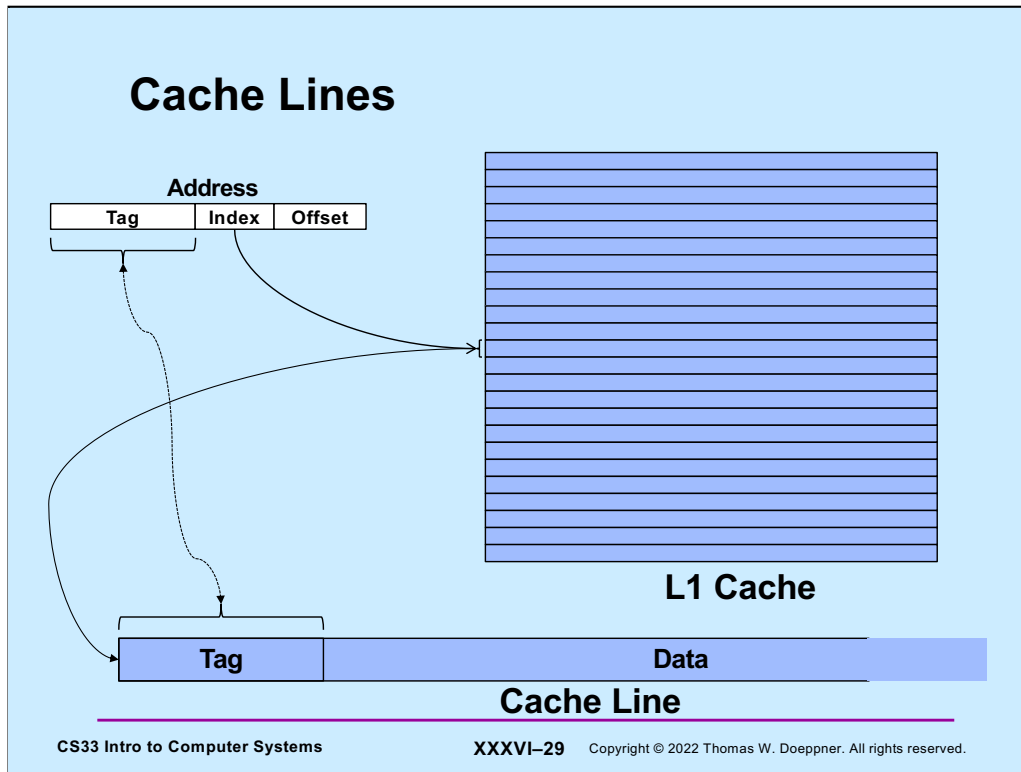
Assume these are run on a two-processor system: why does the two-threaded program on the right run faster than the two-threaded program on the left?

Cache Lines

Address

Tag	Index	Offset

L1 Cache

Tag	Data

Cache Line

Processors usually employ data caches that are organized as a set of cache lines, typically of 64 bytes in length. Thus data is fetched from and stored to memory in units of the cache-line size. Each processor has its own data cache.

False Sharing
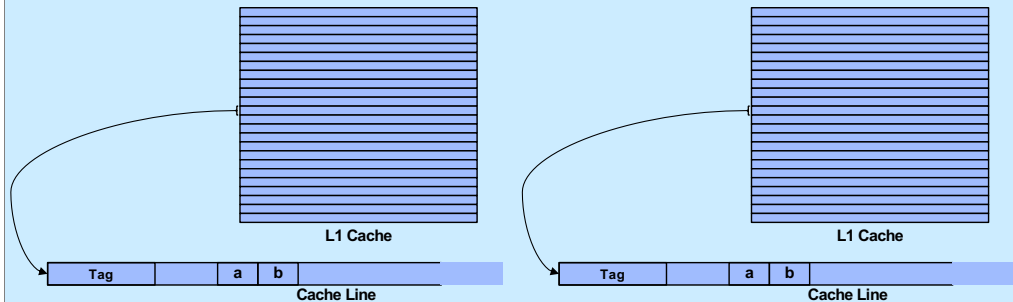
Getting back to our example: we have a two-processor system, and thus two data (L1) caches. If **a** and **b** are in the same cache line, then when either processor accesses **a**, it also accesses **b**. Thus if **a** is modified on processor 1, memory coherency will cause the entire cache line to be invalidated on processor 2. Thus when processor 2 attempts to access **b**, it will get a cache miss and be forced to go to memory to update the cache line containing **b**. From the programmer's perspective, **a** and **b** are not shared. But from the cache's perspective, they are. This phenomenon is known as **false sharing**, and is a source of performance problems.

For further information about false sharing and for tools to deal with it, see http://emeryblogger.com/2011/07/06/precise-detection-and-automatic-mitigation-of-false-sharing-oopsla-11/.

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**

For details on futexes, avoid the Linux man pages, but look at http://people.redhat.com/drepper/futex.pdf, from which this material was obtained. Note that there's actually just one **futex** system call; whether it's a **wait** or a **wakeup** is specified by an argument.

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;
  }

These functions are available on most architectures, particularly on the x86. Note that their effect must be **atomic**: everything happens at once.

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);
}
```

If the futex's value is 0,it's unlocked, otherwise it's locked.

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 <u>won't</u> 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);
  }
}
```

In this version, if the futex's value is 0, it's unlocked, if it's one it's locked and no threads are waiting for it; if it's greater than one it's locked and there might be threads waiting for it.

Memory Allocation

- **Multiple threads**

- **One heap**

Bottleneck?

In a naïve multithreaded implementation of malloc/free, there is one mutex protecting the heap, resulting in a 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;
}
```

In this test program, each thread does a sequence of mallocs and frees.

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
```

The code was run on an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz.

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
```

This was run on a current CS department computer: Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz (4 cores).

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
 …
```

strace is a system facility that supplies information about the system calls a process uses. The –c flag tell is to print the cumulative statistics after the process terminates. The –f flag tells it to include information on all threads and child processes.

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

```c
#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;
```

This program creates pairs of threads: one thread allocates storage, the other deallocates storage. They communicate using producer-consumer communication.

Test Program 2, part 2

```c
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;
}
```

The main function creates **npairs** (16) of communicating pairs of threads.

Test Program 2, part 3

```c
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;
}
```

To reduce the number of calls to **sem_wait** and **sem_post**, at each iteration the thread calls malloc **allocsPerIter** (1024) times.

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
```

The code was run on a SunLab machine (an Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz).

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
```

This was run on a current CS department computer: Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz (4 cores).

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
 …
```