CS 3214
Computer Systems
MULTI-THREADING SYNCHRONIZATION
Accessing global variables

```c
/* Define a mutex and initialize it. */
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

static int counter = 0; /* A global variable to protect. */

/* Function executed by each thread. */
static void *
increment(void *__)
{
    int i;
    for (i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
}
```

```asm
movl    counter, %eax
incl    %eax
movl    %eax, counter
```
Anatomy of a Race Condition

Thread 1

```assembly
movl counter, %eax
incl %eax
movl %eax, counter
```

Thread 2

```assembly
movl counter, %eax
incl %eax
movl %eax, counter
```

IRQ
OS decides to context switch

%eax – Thread 1’s copy
%eax – Thread 2’s copy
counter – global variable, shared

Assume counter == 0 initially
Final result: counter is 1, should be 2
Race Conditions

• Definition: *two or more threads read and write a shared variable, and final result depends on the order of the execution of those threads*

• Usually timing-dependent and intermittent
  – Hard to debug

• Technically not a race condition if all execution orderings lead to same result
  – Chances are high that you misjudge this

• How to deal with race conditions:
  – Ignore (!?)
    • Can be ok if final result does not need to be accurate
    • Not an option in CS 3214
  – Don’t share: duplicate or partition state
  – Avoid “bad interleavings” that can lead to wrong result
Not Sharing: Duplication or Partitioning

• **Undisputedly** best way to avoid race conditions
  – Always consider it first
  – Usually faster than alternative of sharing + protecting
  – But duplicating has space cost; partitioning can have management cost
  – Sometimes must share (B depends on A’s result)

• **Examples:**
  – Each thread has its own counter (then sum counters up after join(())
  – Every CPU has its own ready queue
  – Each thread has its own memory region from which to allocate objects

• Truly ingenious solutions to concurrency involve a way to partition things people originally thought you couldn’t
Thread-Local Storage

• A concept that helps to avoid race conditions by giving each thread a copy of a certain piece of state

• Recall:
  – All local variables are already thread-local
    • But their extent is only one function invocation!
  – All function arguments are also thread-local
    • But must pass them along call-chain

• TLS creates variables of which there’s a separate location for each thread to store their value in.

• In PThreads/C (compiler or library-supported)
  – Dynamic: pthread_create_key(), pthread_get_key(), pthread_set_key()
    • E.g. myvalue = keytable(key_a)→get(pthread_self());
  – Static: using __thread storage class
    • E.g.: __thread int x;

• Java: java.lang.ThreadLocal
# TLS Example

```c
#include <pthread.h>
#include <stdio.h>

__thread int x;

static void *
thread_func(void * _tn)
{
    int i, tn = (int)_tn;
    for (i = 0; i < 3; i++)
        printf("thread=%d x=%d\n", tn, x++);
    return NULL;
}

int
main()
{
    int i, N = 4;
    pthread_t t[N];
    for (i = 0; i < N; i++)
        pthread_create(t + i, NULL,
                        thread_func, (void *)i);
    for (i = 0; i < N; i++)
        pthread_join(t[i], NULL);
    return 0;
}
```

Compiler assigns offset in “per-thread” area; all accesses use this offset
Note that scope of ‘x’ is global.
Atomic Operations

• All architectures support at least 1 atomic operation - so that no bad interleavings can occur because
  – a) CPU will not accept interrupts – so no context switch on same CPU
  – b) multiple cores synchronize by locking caches/buses
• Variations include “test-and-set”, “compare-and-swap”, “compare-and-set”, etc.
• Other atomic ops can be built on top of that
  – Example: java.util.concurrent.AtomicInteger
    • incrementAndGet()
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}

public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}

public final int get() { return value; }
Motivation for Critical Section

- Partitioning or the use of atomic operations is not always possible
- Must then prevent race conditions by imposing constraints on execution order so the final result is the same regardless of actual execution order
  - That is, exclude “bad” interleavings
  - *Specifically*: disallow other threads to start updating shared variables while one thread is in the middle of doing so; make those updates as a group atomic with respect to what other threads “see” – threads either see old or new value, but none in between
Critical Sections

• Critical Section
  – A synchronization technique to ensure atomic execution of a segment of code
    • Has entry() and exit() operations

```c
pthread_mutex_lock(&lock);  /* entry() */
counter++;
pthread_mutex_unlock(&lock);  /* exit() */
```

• Critical Section Problem also known as mutual exclusion problem
Critical Sections (cont’d)

- Only one thread can be inside critical section; others attempting to enter CS must wait until thread that’s inside CS leaves it.

- Different solutions known: entirely software, or entirely hardware possible.
  - Usually combined.
  - Different implementations (but same API) for uniprocessor vs multiprocessor scenarios.
Myths about CS

• A thread in a CS executes entire section without being preempted
  – No – not usually true
• A thread in a CS executes the entire section
  – No – may exit or crash
• There can be only one critical section in a program
  – No – as many as the programmer decides to use
• Critical sections protect blocks of code
  – No – they protect data accesses (but note role of encapsulation!)
Locks

• Thread that enters CS locks it
  – Others can’t get in and have to wait
• Thread unlocks CS when leaving it
  – Lets in next thread
  – which one?
    • FIFO guarantees bounded waiting
    • Highest priority in priority-based systems
• Can view Lock as an abstract data type
  – Provides (at least) init, acquire, release
Locks, Take 2

Shared Data
Managing Locks

• Programmer should consider locks resources
  – Must ensure that they are released on all paths

• In Java/C#
  – Either use built-in “synchronized” or use try/finally

• In C: requires careful reasoning about all code paths
  – Idiomatic structure of functions helps:
    • minimize number of returns
    • can use goto’s to create a structure that resembles a try/finally clause if a function acquires multiple locks (not usually done for a single lock, though)

• In C++: use RAII pattern
void f() {
    lock(l1);
    try {
        ....
        if (some error)
            return;
        ....
    lock(l2);
    try {
        ....
        if (some other err)
            return;
        ....
    } finally {
        unlock(l2);
    }
} finally {
    unlock(l1);
}
synchronized in Java

• Built-in ‘locking’ in Java
• Any Java object can be used as a lock
• 2 forms: block & method attribute
  – If block form: synchronized (o) { … } uses L = o
  – If instance method: uses L = ‘this’
  – If static method of class C: uses L = C.class
• Equivalent to (and best thought of as)
  \( L\).lock(); try { … } finally { L.unlock(); }
• Recursive
  – Synchronized methods can call each other without deadlocking
Locks: Ownership & Recursion

• Locks semantically have notion of ownership
  – Only lock holder is allowed to unlock
  – Some systems allow querying `lock_held_by_current_thread()`
    – But: POSIX does not enforce this and in fact Linux’s implementation allows non-holders to unlock a lock

• What if lock holder tries to acquire locks it already holds?
  – Nonrecursive locks: deadlock!
    • default semantics of POSIX locks
  – Recursive locks:
    • inc counter
    • dec counter on lock_release
    • release when zero
    • default semantics of Java synchronized
Thread-Safety

- Property of a function to yield correct result when called from multiple threads
- Attribute that must be documented as part of the API documentation of a library
- Functions are not thread-safe if they
  1. Fail to protect shared variables
  2. Rely on persistent state across invocations
  3. Return a pointer to a static variable
  4. Call other functions that aren’t thread safe.
• Class 2: Relying on persistent state across multiple function invocations.
  – Random number generator relies on static state
  – Fix: Rewrite function so that caller passes in all necessary state (see `rand_r()`)

```c
static unsigned int next = 1;
/* rand - return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```
Class 3: Returning a ptr to a **static** variable

• Fixes:
  – 1. Rewrite code so caller passes pointer to **struct**.
    • Issue: Requires changes in caller and callee.
  – 2. **Lock-and-copy**
    • Issue: Requires only simple changes in caller (and none in callee)
    • However, caller must free memory.

```c
struct hostent
*gethostbyname(char *name)
{
    static struct hostent h;
    <contact DNS and fill in h>
    return &h;
}
```

```c
hostp = malloc(...);
gethostbyname_r(name, hostp);
```

```c
struct hostent
*gethostbyname_ts(char *p)
{
    struct hostent *q = Malloc(...);
    lock(&mutex); /* lock */
    p = gethostbyname(name);
    *q = *p;    /* copy */
    unlock(&mutex);
    return q;
}
```
Performance Cost of Locking

• Direct cost: at a minimum, 1 atomic instruction per lock acquire (uncontended)

• Indirect cost if lock is contended (i.e., held by other thread when thread attempts to acquire it)
  – Context switch cost: Thread needs to move into blocked state, context switch to other thread (if any is ready), later – thread must be made ready again, another context switch back to resume thread
  – Opportunity cost due to introduced serialization/loss of parallelism: blocked threads cannot use CPU/cores, may lead to underutilization and/or idling of cores

• Remember: nobody cares about the optimization of incorrect code. Safety is always paramount
Critical Section Efficiency

- Lock Acquisition ($T_a$)
- Critical Section ($T_c$)
- Lock Release ($T_r$)

\[
\text{Efficiency} = \frac{T_c}{T_c + T_a + T_r}
\]

- As processors get faster, CSE decreases because atomic instructions become relatively more expensive (even in uncontended case!)

Source: McKenney, 2005
How many locks should I use?

• Could use one lock for all shared variables
  – Disadvantage: if a thread holding the lock blocks, no other thread can access *any* shared variable, even unrelated ones
  – Sometimes used when retrofitting non-threaded code into threaded framework
  – Examples:
    • “BKL” Big Kernel Lock in Linux
    • Interpreter Lock in Python
    • GUI Lock in gtk

• Ideally, want fine-grained locking
  – One lock only protects one (or a small set of) variables – how to pick that set?
Mapping Locks To Variables

• Choosing which lock should protect which shared variable(s) is not easy – must weigh:
  – Whether all variables are always accessed together (use one lock if so)
  – Whether there is an atomicity requirement if multiple variables are accessed in related sequence (must hold single lock if so)
  – Cost of multiple calls to lock/unlock (advantage of increased parallelism may be offset by those costs)
  – Whether code inside critical section may block (if not, no throughput gain from fine-grained locking on uniprocessor)
Simple Rules for Locking

• Every shared variable must be protected by a lock
  – Establish this relationship with code comments
    • /* protected by … <lock>*/
  – Acquire lock before touching (reading or writing) variable
  – Release when done, on all paths
  – One lock may protect more than one variable, but not too many
    • If in doubt, use fewer locks (may lead to worse efficiency, but less likely to lead to race conditions or deadlock)

• If manipulating multiple variables, acquire locks assigned to protecting each
  – Acquire locks always in same order (doesn’t matter which order, but must be same)
  – Release in opposite order
  – Don’t release any locks before all have been acquired (two-phase locking)
Race Detection Tools

• A number of tools help to detect race conditions (Helgrind, Intel Thread Checker)
  – Dynamic analysis tools
• Typically based one or both of these approaches:
  – Locksets: detect if no lock is consistently held when a given variable x is accessed
  – “Happens-before” relationship: (intuition) if we can’t prove that access A must happen before B due to the synchronization the programmer used, then it’s a race; example:
    • All accesses before a thread is started “happen before” all accesses by a thread; and these accesses “happen before” all accesses done after the thread is joined
• Typically do not detect atomicity violations
Atomicity Violations

• Locking all accesses to shared variables does not guarantee that a program is free of concurrency bugs

• Atomicity violations are another, more insidious type of concurrency violations

• Bug pattern is
  - lock
  - get information I
  - unlock
  (other threads may invalidate information I)
  - lock
  - act on now possibly out-of-date information I
  - unlock

```c
char *p = ….; /* shared variable */
lock lp; /* protects ‘p’ */
...
int getplen() {
    lock(&lp);
    int len = strlen(p);
    unlock(&lp);
    return len;
}
...
int nchars = getplen();
char *copy = malloc(nchars + 1);
lock(&lp);
strcpy(copy, p);
unlock(&lp);
```
Atomicity Violation (Example 2)

• Incorrect even though individual accesses to “sb” are synchronized (protected by a lock)
  – But “len” may no longer be equal to “sb.length” in call to getChars()
• This means simply slapping lock()/unlock() around every access to a shared variable does not thread-safe code make
• Found by Flanagan/Freund

```java
public synchronized StringBuffer append(StringBuffer sb) {
    int len = sb.length(); // note: StringBuffer.length() is synchronized
    int newcount = count + len;
    if (newcount > value.length)
        expandCapacity(newcount);
    sb.getChars(0, len, value, count); // StringBuffer.getChars() is synchronized
    count = newcount;
    return this;
}
```
Atomicity Constraints

- Atomicity violations disregard atomicity constraints
  - Information read in critical section A must not be used in a critical section B if B relies on it not having been changed

```java
lock();
var x = read_var();
unlock();
....
lock();
use(x);
unlock();
```

atomic

atomicity required to maintain consistency