

VGP393C – Week 7

⇒ Agenda:

- Common multi-threading problems
 - Dead-lock / live-lock
 - Priority inversion
 - Thread-safe libraries
- Cache abuse / memory bandwidth



27-August-2008

© Copyright Ian D. Romanick 2008

Deadlock

- *Deadlock* occurs when forward progress is halted because every task is waiting for some other task to complete some action
 - Requires that *all* four of these conditions be met:
 - Access to each resource is exclusive
 - A task is allowed to acquire one resource while already holding another
 - No task is willing / able to release a resource that it has acquired
 - There is a cycle of tasks trying to acquire resources
 - Each resource is held by one task but is requested by another



27-August-2008

© Copyright Ian D. Romanick 2008

Deadlock

- ⇒ Allow non-exclusive access to resources
 - Data replication
 - Non-blocking algorithms
 - etc.



27-August-2008

© Copyright Ian D. Romanick 2008

Deadlock

⇒ Break the cycle of tasks

- Require that tasks acquire resources in a particular order
 - Order resources by name / ID
 - Order resources by sequence in data structure (list, tree, etc.)
 - Order resources by memory address of the resource
 - etc.



27-August-2008

© Copyright Ian D. Romanick 2008

Deadlock

- Allow tasks to release resources when deadlock is possible
 - If a second resource cannot be acquired in a reasonable time, release the first resource

```
void acquire_lock(lock *L1, lock *L2)
{
    bool L1_held = true;
    for (unsigned i = base_timeout; /* empty */ ; i *= 2) {
        if (!L1_held)
            acquire(L1);

        if (try_acquire(L2, i))
            break;

        release(L1);
        L1_held = false;
    }
}
```



27-August-2008

© Copyright Ian D. Romanick 2008

Live-lock

- No task makes *progress* though all tasks are doing something
 - In the previous example, two tasks in lock-step would repeatedly:
 - Try to acquire a second resource
 - Release the first resource
 - Re-acquire the first resource
 - Lather, rinse, repeat
 - Usually fixed by randomizing timeouts or adding a priority scheme



27-August-2008

© Copyright Ian D. Romanick 2008

Live-lock

```
void acquire_lock(lock *L1, lock *L2)
{
    bool L1_held = true;
    for (unsigned i = base_timeout; /* empty */ ; i *= 2) {
        if (!L1_held)
            acquire(L1);

        unsigned try_timeout = random_value(base_timeout, i);
        if (try_acquire(L2, try_timeout))
            break;

        release(L1);
        L1_held = false;
    }
}
```



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

- ⇒ Imagine you have three threads:
 - Low priority thread: gathers meteorological data
 - Runs infrequently for a short period of time
 - Accesses a shared data structure via a mutex



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

- Imagine you have three threads:
 - Low priority thread: gathers meteorological data
 - Runs infrequently for a short period of time
 - Accesses a shared data structure via a mutex
 - Medium priority thread: performs communications
 - May run for a long period of time



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

⇒ Imagine you have three threads:

- Low priority thread: gathers meteorological data
 - Runs infrequently for a short period of time
 - Accesses a shared data structure via a mutex
- Medium priority thread: performs communications
 - May run for a long period of time
- High priority thread: performs *important* system management activities

Called a *watchdog timer*

- This thread runs frequently

- If this thread is unable to run for a long period of time, the system assumes it has crashed and reboots

- Access same shared data as the low priority thread



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

- The system usually runs just fine, but...
 - Every now and then, it randomly reboots
 - These reboots are traced to the watchdog timer
 - What happened?



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

- The system usually runs just fine, but...
 - Every now and then, it randomly reboots
 - These reboots are traced to the watchdog timer
 - What happened?
- Classic priority inversion!
 - Low priority thread acquires the mutex
 - Medium priority thread starts running and preempts the low priority thread
 - High priority thread needs to run
 - Can't acquire mutex because it is held by another thread
 - Low priority thread can't release the mutex because the medium priority thread is running



Priority Inversion

⇒ Does this sound contrived?



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

- ⇒ Does this sound contrived?
 - Maybe, but it this is exactly what happened to the Mars Pathfinder in 1997¹

¹ http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

- *Priority inversion* occurs when “a low priority task holds a shared resource that is required by a high priority task....execution of the high priority task [is] blocked until the low priority task has released the resource, effectively “inverting” the relative priorities of the two tasks.¹”

¹ http://en.wikipedia.org/wiki/Priority_inversion



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

- Three common solutions to priority inversion
 - Disable interrupts / task switching during critical sections
 - Priority inheritance
 - Priority ceilings



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

- Disable interrupts / multitasking during critical sections
 - Since a task cannot be interrupted while holding a lock, it will run to completion and release the lock
 - A higher priority thread cannot prevent a lower priority thread from finishing up with a mutex
 - Only one thread can run at a time with a mutex held, so deadlock is prevented single processor systems
 - Not very practical:
 - Can't disable multitasking on desktop / server OSs!
 - Limits scalability
 - Doesn't help on multiprocessor systems



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

⇒ Priority inheritance

- A thread holding a lock temporarily inherits the priority of the highest priority thread requesting the lock
- If a higher priority thread needs to run, the lower priority thread holding the lock is guaranteed to be scheduled so that it can finish
 - This is how the Mars Pathfinder team solved their priority inversion problem



27-August-2008

© Copyright Ian D. Romanick 2008

Priority Inversion

⇒ Priority ceiling

- Each mutex has an associated priority
 - Called the “ceiling,” it is the highest priority of any thread expected to ever hold the mutex
- When a thread acquires a mutex, its priority is bumped to that of the mutex



27-August-2008

© Copyright Ian D. Romanick 2008

Thread-Safe Libraries

- A *thread-safe* library function can be called concurrently by multiple library clients (threads)



27-August-2008

© Copyright Ian D. Romanick 2008

Thread-Safe Libraries

- Consider the C library function `fprintf`:
 - Processes the format and parameters, and writes characters to the specified file
 - The `FILE` structure contains a low-level OS file handle and a buffer
 - File I/O writes to the buffer and flushes it to the file when full
- What happens if multiple threads call a non-thread-safe `fprintf` concurrently?
 - Chaos



27-August-2008

© Copyright Ian D. Romanick 2008

Thread-Safe Libraries

⇒ Consider the C library function `strtok`:

```
char *strtok(char *str, const char *delim);
```

- Partitions `str` into “tokens” separated by characters in `delim`
- First call sets `str`, which is stored in *hidden* storage
 - Following calls pass `NULL` to get successive tokens from the same string
 - Usually happens in a loop

⇒ What happens if multiple threads call `strtok` concurrently?



Chaos

27-August-2008

© Copyright Ian D. Romanick 2008

Thread-Safe Libraries

⇒ `strtok` is just plain broken:

- Uses hidden data that is shared *across calls* to the function
 - A simple mutex in the function doesn't solve the problem

⇒ Two ways to fix:

- Add an explicit “state” parameter
 - `strtok_r` does just that
 - The `_r` in names of standard C library functions means *re-entrant*
- Use thread-local storage



27-August-2008

© Copyright Ian D. Romanick 2008

Thread-Safe Libraries

- ⇒ Interface conventions to follow:
 - Data used across calls should be passed in
 - Caller provides synchronization on objects passed-in
 - Different from the `fprintf` case!
 - Library function provides synchronization on global data
 - Provides thread-safety when called concurrently on different objects



27-August-2008

© Copyright Ian D. Romanick 2008

Thread-Safe Libraries

➤ Why force the caller to provide synchronization on objects passed-in?

- Imagine a linked list class that provides thread-safe `pop` and `is_empty` methods:

```
node = list.pop();  
if (list.is_empty()) {  
    foo();  
}
```

Race condition!



- Explicit synchronization is required anyway
- Implicit synchronization inside the methods becomes wasted overhead



27-August-2008

© Copyright Ian D. Romanick 2008

Thread-Safe Libraries

- System libraries usually have thread-safe and non-thread-safe versions
 - In VisualStudio, /MD (or /MDd) selects the thread-safe versions
 - Some interfaces are just plain broken, and these are typically documented as non-thread-safe
 - This can sometimes be worked around by using your own mutex



27-August-2008

© Copyright Ian D. Romanick 2008

Break



27-August-2008

© Copyright Ian D. Romanick 2008

Cache

- Two primary cause of decreased cache performance
 - Working set too large
 - Causes memory-to-cache data movement and cache-to-memory data movement
 - Data sharing
 - Causes processor-to-processor data movement
 - i.e., from one CPU's cache to another CPU's cache



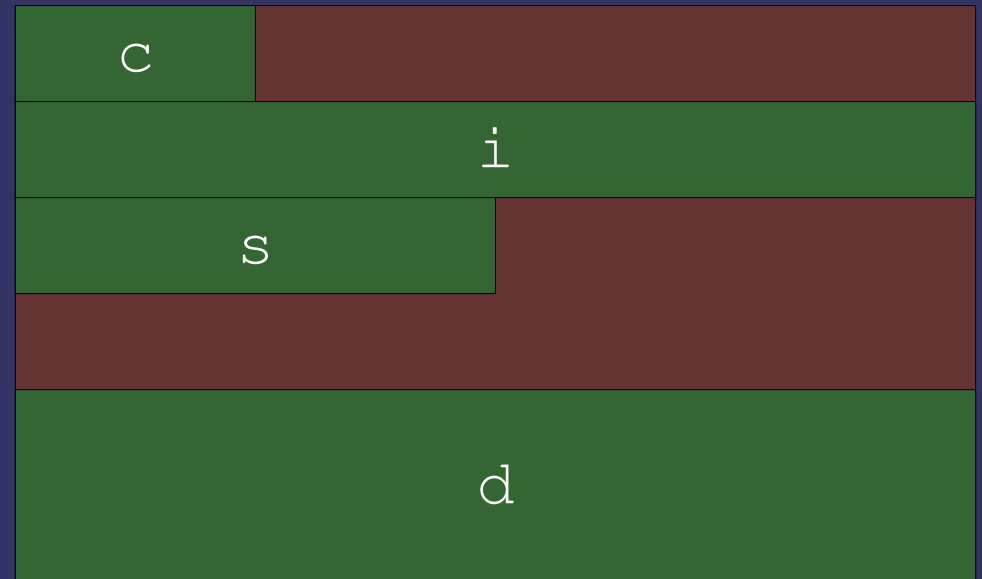
27-August-2008

© Copyright Ian D. Romanick 2008

Cache

- ⇒ CPUs have data alignment rules
 - Usually, data must be aligned to a multiple of its size
 - Results in holes or *padding* in structures

```
struct foo {  
    char c;  
    int i;  
    short s;  
    double d;  
};
```



27-August-2008

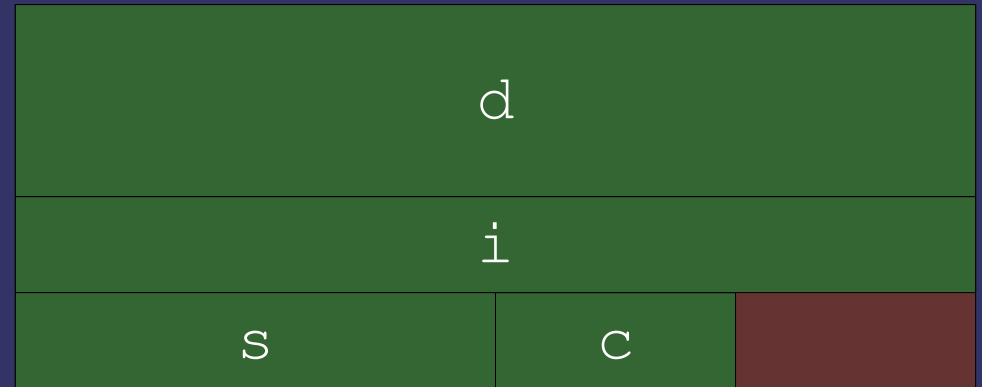
© Copyright Ian D. Romanick 2008

Cache

⇒ CPUs have data alignment rules

- Usually, data must be aligned to a multiple of its size
- Results in holes or *padding* in structures
- Ordering structure members by size fixes this

```
struct foo {  
    double d;  
    int i;  
    short s;  
    char c;  
};
```



- Now the whole structure fits in a single 16-byte cache-line!



27-August-2008

© Copyright Ian D. Romanick 2008

Cache

- Keeping data in the cache improves performance
 - Decrease the size of the *working set* by compacting the data
 - Use smaller data types
 - Improve alignment to compact structures
 - Operate on a *window* of a larger data set



27-August-2008

© Copyright Ian D. Romanick 2008

Cache

```
int strike(bool *composite, int i, int stride, int limit)
{
    for (/* empty */; i <= limit; i += stride)
        composite[i] = true;
    return i;
}
```

```
int sieve(int n)
{
    int count = 0;
    int m = (int) sqrt((double) n);
    bool *const composite = new bool[n + 1];

    (void) memset(composite, 0, sizeof(bool) * (n + 1));
    for (int i = 2; i <= m; i++) {
        if (!composite[i]) {
            count++;
            strike(composite, 2 * i, i, n);
        }
    }
    for (int i = m + 1; i <= n; i++) {
        if (!composite[i]) count++;
    }
    delete composite;
    return count;
}
```

If n is larger than the cache size,
`strike` will thrash the cache



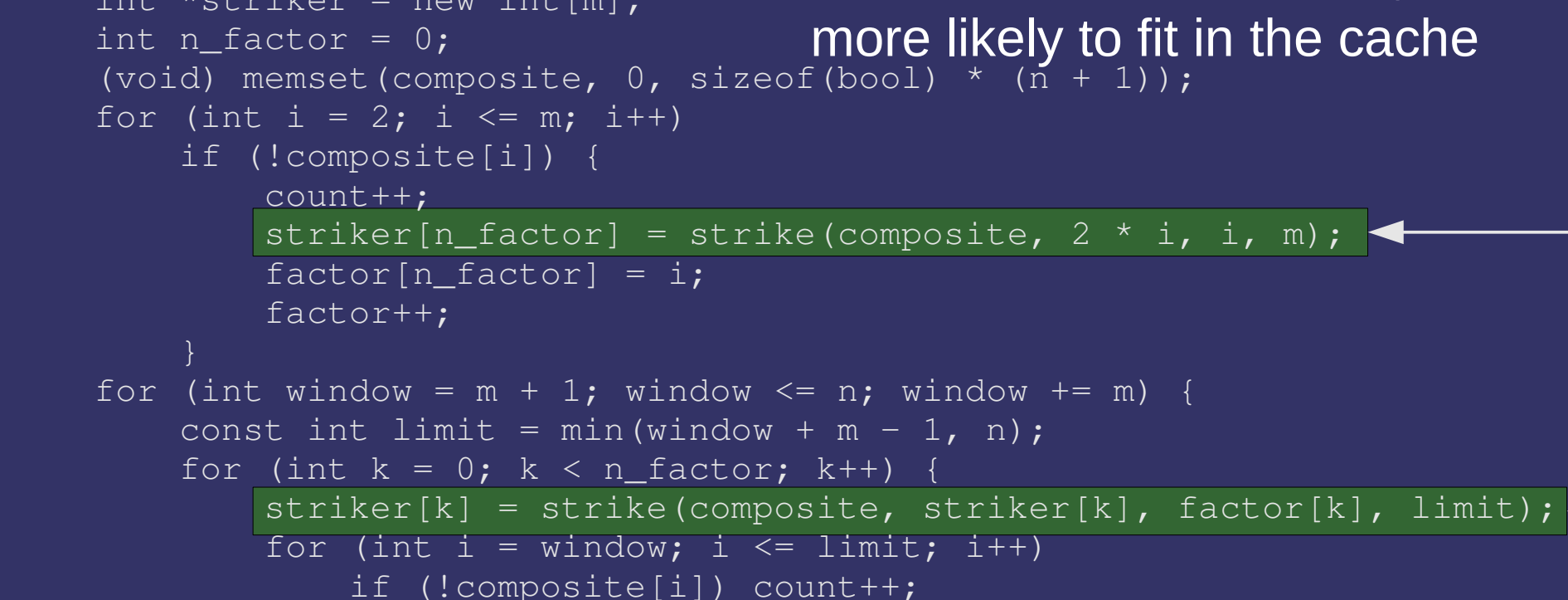
27-August-2008

© Copyright Ian D. Romanick 2008

Cache

```
int sieve_cache_friendly(int n)
{
    int count = 0;
    int m = (int) sqrt((double) n);
    bool *const composite = new bool[n + 1];
    int *factor = new int[m];
    int *striker = new int[m];
    int n_factor = 0;
    (void) memset(composite, 0, sizeof(bool) * (n + 1));
    for (int i = 2; i <= m; i++)
        if (!composite[i]) {
            count++;
            striker[n_factor] = strike(composite, 2 * i, i, m);
            factor[n_factor] = i;
            factor++;
        }
    for (int window = m + 1; window <= n; window += m) {
        const int limit = min(window + m - 1, n);
        for (int k = 0; k < n_factor; k++) {
            striker[k] = strike(composite, striker[k], factor[k], limit);
            for (int i = window; i <= limit; i++)
                if (!composite[i]) count++;
        }
    }
    delete composite; delete factor; delete striker;
    return count;
}
```

m is much smaller than n, and is more likely to fit in the cache



The diagram consists of a large white bracket on the right side of the slide, spanning from the text 'm is much smaller than n, and is more likely to fit in the cache' down to the two highlighted lines of code. Two white arrows point from the text towards the highlighted lines: one points to the line 'striker[n_factor] = strike(composite, 2 * i, i, m);' and the other points to the line 'striker[k] = strike(composite, striker[k], factor[k], limit);'.



27-August-2008

© Copyright Ian D. Romanick 2008

Cache

- Certain memory access patterns cause *memory contention*, which results in processor-processor transfers
 - Read-write – One processor writes a cache line, another processor reads it
 - Write-write – One processor writes a cache line, another processor writes it
 - Read-read – No contention in this case




27-August-2008

© Copyright Ian D. Romanick 2008

Cache

```
int sieve_cache_friendly(int n)
{
    int count = 0;
    int m = (int) sqrt((double) n);
    bool *const composite = new bool[n + 1];
    int *factor = new int[m];
    int *striker = new int[m];
    int n_factor = 0;
    (void) memset(composite, 0, sizeof(bool) * (n + 1));
    for (int i = 2; i <= m; i++)
        if (!composite[i]) {
            count++;
            striker[n_factor] = strike(composite, 2 * i, i, m);
            factor[n_factor] = i;
            factor++;
        }
    for (int window = m + 1; window <= n; window += m) {
        const int limit = min(window + m - 1, n);
        for (int k = 0; k < n_factor; k++) {
            striker[k] = strike(composite, striker[k], factor[k], limit);
            for (int i = window; i <= limit; i++)
                if (!composite[i]) count++;
        }
    }
    delete composite; delete factor; delete striker;
    return count;
}
```

Use loop parallelism here



27-August-2008

© Copyright Ian D. Romanick 2008

Cache

- Fix contention using known patterns:
 - Let read-only memory be shared
 - Fill `factor` once and let all threads share it
 - Generate output to task-local buffers when possible
 - Each task has a private sub-range of `composite`
 - Each task has a private `striker`
 - Eliminate the loop-carried dependency by recalculating the first element of `striker`
 - Use reductions when possible
 - Calculate `count` per-task, reduce at the end



27-August-2008

© Copyright Ian D. Romanick 2008

Cache

- Cache line granularity is not usually the same as data item granularity
 - The `count` reduction array has a 4-byte granularity, but the L1 cache line granularity on a Core 2 Duo is 64-bytes
 - If two processors access separate data items that happen to reside in the same cache line there will be memory contention
 - This is called *false sharing*



27-August-2008

© Copyright Ian D. Romanick 2008

Cache

- Fix false sharing by doing cache line granular data partitioning
 - Partition distributed arrays a cache line boundaries
 - Pad elements of reduction arrays to the cache line size
 - Allocate per-task data at cache line boundaries
- Need to know the size of a cache line!
 - Need a memory allocator that can allocate memory at arbitrary alignments
 - Search for “aligned malloc”



27-August-2008

© Copyright Ian D. Romanick 2008

Next week...

⇒ No class next week

- Next class meeting is Wednesday 9/10
- SIMD
- Quiz #3
 - Will cover material from week 5 and week 6
 - Will *not* cover material from this week!
- Assignment #3 due
- Start assignment #4



27-August-2008

© Copyright Ian D. Romanick 2008

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.



27-August-2008

© Copyright Ian D. Romanick 2008