CS61 – Lecture 25 – 12/25/2012
Scribe Notes by Farnham, Buffalo Hird, & Ana Nast

<u>Agenda</u>
1) Concurrency
2) Parallelism in Speeding up programs
3) pthread calls; mutexes and locks on critical sections

*Code in L25 Directory
- example for the day: reads in lines, sorts them, prints it back out. Mostly in Hungarian.
  - Function: read_lines - reads lines into a lineset - struct - array of line, each line is a string and a length; lineset also has size and some other stuff
  - Function: sort_lines - uses qsort function - predefined function: we know it works and is fast, but we don't know whether it's as fast as possible

When we compare this with system sort, system sort does almost twice as well as rsort. Why?
- Key to figuring it out is in cpu usage - system sort uses multiple cores, reason for ~300% usage
- qsort is on a single core

Sort program using mergesort
Mergesort –
1) breaks data down into halves
2) sorts halves individually
3) combines sorted halves element by element

· Mergesort cannot be done in place – uses 0n extra space for spare array
· Quicksort preferred for small data sets because it requires no extra space
· Mergesort's divide-and-conquer design is easily parallelized
    -sorting of halved arrays done in parallel

Using this parallelism sort02.c is faster!

<u>Threads</u>
· Every thread requires a threadstart function
· Example given for a thread with ID left_thread calling function sort_lines(left)

```
Pthread_create(&left_thread, NULL, &sort_lines, &left)
```

Arguments:
1) storing of thread ID
2) ignore (thread attributes not needed in cs61)
3) thread function (function thread will run)
4) argument passed to thread function

- This is a great idea in computer science, but sort03 crashed?
    - 1800% of CPU?
        - merging lines that have not been sorted yet -> creates a race condition
        - synchronization problem -> garbage pointers
    - solution: pthread_join()
        - blocks until the threads eists
        - now no merging until each side is sorted!
    - Better, but how parallel is it?
        - recursive calls replaced with new threads
        - thread number proportional to size of the input and merge threshold

- why are we creating two threads and waiting on both of them?
- what utilization problems does this create?
    - parent thread inactive while waiting for children
    - solution: create new thread for only 1 child half
        - reuse parent thread for other child half
        - this increases performance by 2.5%! which is still pretty small, so we'll try to make it better.

<u>Hypothesis and Scientific Method for Computer Science Problems</u>

Question: What is slowing down code?
Hypothesis: Tremendous number of threads making our sort slower than system

How do we test our hypothesis?

- Track threads!
    - Cur_nthreads = current active threads
    - Total_nthreads = total used threads
    - Max_nthreads = max active threads

Are our counters correct or is this a critical section?
- Critical section: code which depends upon order threads run

How to find a critical section

- Look for shared variables modified (written) by multiple threads
- Sort arrays separate for each thread = no critical section
- Thread counters modified by each thread = critical section

<u>Locks for Threads</u>

Static pthread_mutex_t lock = PTHREAD_MUTEX_INTIALIZED

- Creates a lock (mutex) named lock to enforce critical sections

```
Pthread_mutex_lock(&lock);
// critical section code
pthread_mutex_unlock(&lock);
```

But what of the performance overhead of locks?

- Largest speedup is when a program is correct for first time!
- Art of parallel programming: minimizing locks and lock duraction

Common issue debugging:

- You may need to run a code many times to see bugs
- “Heisenbugs” - they disappear when you try to see what causes them :)

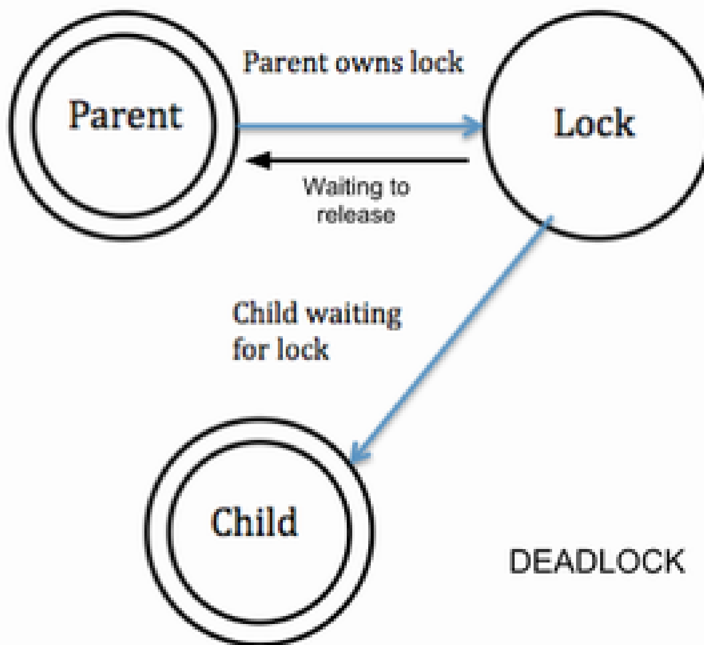
Common error: lock and unlock in wrong places

- Why not make critical sections larger and less error-prone?
    - no longer parallel as only one thread can execute
    - can cause fatal deadlocks - example in our program - runs forever without doing anything
        - main thread retains lock
        - recursive calls can’t acquire mutex

Circular Wait – infinite wait caused by poor mutex management

- Deadlock cycle – worse than crash because program continues with no alert
    - detecting deadlock is an important problem in concurrency

Resource Apposition graph / Lock Apposition graph



How can we beat system sort?

- A lot of thread forking –forks to threshold of 1024 array elements
- We are concerned with overall thread latency

- Using n threads, where n is the number of cores
  - as a fallback, call qsort if we can't parallelize further - sort08

  - slightly faster than overloading with thousands of threads - gets us pretty close to system sort
  - why call qsort if we could simply wait for open cores?

- Potential bottlenecks of this code? (still in sort08.c)
  - Contention – too many threads trying to acquire mutex, waiting on locks
    - -threads must wait, wasting cpu cycles
      - wait in correlation to number of threads
    - -performance increase with more cores until drastically falling
    - -critical section rough bound on contention's effect

    → probably not the source of our bottleneck
  - imbalance hypothesis – small number of threads take longest
    - -fork diagram – last fork (qsort) takes exponentially longer
    - -if left sorts before right, all of right will be left to qsort

Probable issue::

Amdahl's Law : The performance benefit of an optimization to part of a program is limited by the overall performance of the rest of the program

- Possible that read_lines takes 80% of the time and sort_lines only 20%
    - read_lines is only 1 thread -> unaddressed possible bottleneck
    - optimization of 20% will still leave majority unoptimized

    -> could get at most 20% improvement, even if optimized sorting to no time

- read_lines can be speed up with cache use!

(when in doubt, saying "cache" as a way to improve performance is a good plan)

- merging is sequential, but each item merged is a pointer
    - these pointers dereference sporadic memory addresses
    - solutions?
        - store n letters of words locally
        - dereference only when necessary

Big Data & Map Reduce

Big Data – challenge of working with ever increasing dataset sizes

Map Reduce – ships off portions of dataset to multiple computers

- extreme end of concurrency!

What comes next for us?

- CS 161
- CS 205
- CS 207