Lecture 4

A closer look at synchronization
Performance measurement and characterization
Announcements

• A1 has been posted
• Section will be held during Thursday’s lecture time (10/10)
• Lecture will be held on Friday 2:00 to 3:20, Podcast?
• There will be no quiz, but a double quiz next Friday
• Friday 11/7 (not 10/17) is the makeup date for 11/18, on a Friday, at a time TBA
• Quiz 1: avg: 1.45 σ: 0.88 median 1.5
Back to synchronization
Semaphores

- Locks handle mutual exclusion
- But they aren’t an appropriate solution for long critical sections
- Introducing Semaphores
  - Counting
  - Binary (mutex)
What is a semaphore?

• An abstract data type that provides mutual exclusion to critical sections [Dijkstra, 1968]

• Semaphore S: integer variable that supports two operations
  – \text{wait}(S): wait until $S>0$, then decrement [Dijkstra’s P() or down()]
  – \text{signal}(S): increment, allow another thread to enter [Dijkstra’s V(), or up()]
Two types of Semaphores

- **Binary semaphore** (or *mutex* semaphore)
  - 0 or 1

- **Counting semaphore**
  - Initialized with general integer values
  - Keeps track of multiple resources, also manages certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore
Implementation

struct Semaphore {
    int value;
    Queue q;
} S;

Atomic wait (S) {
    Begin Critical Section:
    S->value = S->value – 1;
    if (S->value< 0) {
        enqueue(S->q, current_thread);
        yield(current_thread);
    }
    End Critical Section
}

Atomic signal (S) {
    Begin Critical Section:
    S->value = S->value + 1;
    if (S->value< 0) {
        thread = dequeue(S->q);
        thread_start(thread);
    }
    End Critical Section
}
Discussion follows
Voelker and Marzullo
(CSE 120)
Implementing locks with Atomic Instructions

• Hardware executes test-and-set atomically
  – Record the old value
  – Set the value to indicate available
  – Return the old value

Atomic bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}

• After T&S returns
• What is value of flag afterwards if it was initially False? True?
• What is the return result if flag was initially False? True?
Using Test-And-Set

```c
struct lock {
    int held = 0;
};
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0;
}
```
Using Semaphores

• Use is similar to our locks, but different semantics

```
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

```
wait(S);
balance = get_balance(account);
balance = balance – amount;
wait(S);
put_balance(account, balance);
signal(S);

wait(S);

...signal(S);

...signal(S);
```
Readers/Writers Problem

• An object is shared among several threads
• Some threads only read the object, others only write it
• There are multiple readers but only one writer
  – \( #r = \) number of readers, \( #w = \) number of writers
  – Safety: \( (#r \geq 0) \land (0 \leq #w \leq 1) \land ((#r > 0) \implies (#w = 0)) \)
The Code

int readcount = 0;
Semaphore mutex = 1;
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}
    all readers can fall through

reader {
    wait(mutex); // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex); // unlock readcount
    Read;
    wait(mutex); // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex); // unlock readcount}
Questions

• Once a writer exits, all readers can fall through
  – Which reader gets to go first?
  – Is it guaranteed that all readers will fall through?

• If readers and writers are waiting, and a writer exits, who goes first?

• Why do readers use mutex?

• Why not the writers?

• Are there any problems that can be solved with counting semaphores that can’t be solved with mutex semaphores?
Drawbacks

- Shared global variables
- No connection between the semaphore and the data it controls
- No control or guarantee of proper usage
- Alternatives
  - Monitors
  - PL Support
End Discussion
Posix Interface

#include <semaphore.h>

sem_t sem;
sem_init(&sem,0,Count)
sem_wait(&sem);
sem_post(&sem);

sem_t mutex;
sem_init(&mutex,0,1)
Performance measurement and characterization
Why do we measure performance?

- Determine if we have met our design goals
- Obtain feedback to refine a design
- Establish competition and pricing
Measures of Performance

- Completion time
- Processor time product
  \[ \text{Completion time} \times \# \text{processors} \]
- Throughput: amount of work that can be accomplished in a given amount of time
- Relative performance: given a reference architecture or implementation
  AKA *Speedup*
Parallel speedup and efficiency

- **Definition**
  
  The *parallel speedup* on \( P \) processors is \( S_P \)

  \[
  \text{Execution time on 1 processor} = T_1 \\
  \text{Execution time on } P \text{ processors} = T_p
  \]

- Parallel efficiency \( E_P \)
  
  \[
  S_P / P
  \]

- \( T_1 \) running time for the “best serial algorithm”

- May not be the running time of the parallel algorithm on 1 processor
Performance Anomalies

- Super-linear speedup: $S_P > P$
- Is it real?
- A better serial algorithm may be lurking
What’s wrong with speedup?

- Not always an accurate way to compare different algorithms….
- or the same algorithm running on different machines
- For an individual user the bottom line is running time $T_P$ or the *space time cost* $P T_P$
- Can we obtain a better speedup at the cost of a longer running time?
Scalability

- We want performance to scale linearly with the number of processors.
- Difficulties:
  - Serial sections: code that runs on only one processor.
  - “Non-productive” work associated with parallel execution, e.g. communication.
  - Load imbalance: uneven work assignments over the processors.
- Some algorithms present intrinsic barriers to scalability leading to alternatives:
  \[
  \text{for } i=0:n-1 \quad \text{sum} = \text{sum} + x[i]
  \]
Serial section

- Limits scalability
- Let $f = \text{the fraction of } T_1 \text{ that runs serially}$
- $T_1 = f \times T_1 + (1-f) \times T_1$
- $T_P = f \times T_1 + (1-f) \times T_1 / P$
- Thus $S_P = 1/[f + (1 - f)/p]$
- As $P \to \infty$, $S_P \to 1/f$
- This is known as Amdahl's Law (1967)
Amdahl’s law (1967)

- \( f = \) fraction of \( T_1 \) that runs serially
- As \( P \to \infty \), \( S_p \to \frac{1}{f} \)
Scaled Speedup

- Is Amdahl’s law pessimistic?
- Observation: Amdahl’s law assumes that the workload \((W)\) remains fixed
- But parallel computers are used to tackle more ambitious workloads

\[ W \text{ increases with } P \]
\[ f \text{ often decreases with } W \]
Computing scaled speedup

• Instead of asking what the speedup is, let’s ask how long a parallel program would run on a single processor [J. Gustafson 1992]

• Let $T_P = 1$

• $f' = \text{fraction of serial time spent on the parallel program}$

• $T_I = f' + (1-f') \times P = S'_P = \text{scaled speedup}$

• Scaled speedup is linear in $P$
Isoefficiency

- Consequence of Gustafson’s observation is that we increase N with P
- Kumar: We can maintain constant efficiency so long as we increase N appropriately
- The *isoefficiency* function specifies the growth of N in terms of P
- If N is linear in P, we have a *scalable* computation
- What if N is super-linear in P?
Reporting and Displaying Performance

• Give the viewer sufficient information to…
  – Draw their own conclusions
  – Reproduce your results

• Tabulate and display the results fairly
  – Avoid misleading techniques
  – See the Bailey paper for examples of how not to display and report performance data
Challenges to measuring performance

• Reproducibility
  – Transient system operating conditions
  – Differing systems or program configuration

• Measurements are imprecise
  – “Heisenberg uncertainty principle:” measurement technique may affect performance
  – Overheads and inaccuracy

• Explain anomalous behavior, but ignore anomalies that are not significant
Complications

• Cost of measuring a full run is prohibitive
  – Ignore startup code if you plan to run for a much longer time in production

• Transient behavior
  – Repeat your measurements
  – “Warm up” the code before collecting measurements
  – Ignore outliers unless their behavior is important to you
  – Average time, maximum time, minimum time?
Measurement collection

• Report the best timings
  ▶ Repeat results 3 to 5 times until at least 2 measures agree to within... 5%, 10%
  ▶ Report the minimum time
• Also report outliers
• A scatter plot or error bar can be useful
Timing collection

• Measures of time
  ▶ Elapsed, or “wall clock” time
  ▶ CPU time = system + user time
  ▶ Overhead, resolution, and quantization effects

• Measurement tools
  ▶ Unix time command does a reasonable job for long-running programs
  ▶ Hardware performance monitors
  ▶ System clocks
    • Often platform dependent, especially library routines
    • Gettimeofday() [*IX], read_real_time() [IBM]