2024 Fall Final

Course: ECE454: Computer Systems Software
Examiners: Jon Eyolfson, Jianwen Zhu
Date: December 19, 2024
Duration: 2 hours 30 minutes (150 minutes)

Exam Type: A

A “closed book” examination.
No aids are permitted other than the information printed on the examination paper.

Calculator Type: 3

Non-programmable calculators from a list of approved calculators as issued by the Faculty Registrar.

Instructions:

Do not write answers on the back of pages as they will not be graded. Use the blank sheet at the end of the exam for extra space, and clearly indicate in the provided answer space if your response continues.

If a question seems unclear or ambiguous, state your assumptions and answer accordingly. In case of an error, identify it, provide a corrected version, and respond as if the question has been fixed.

Provide brief and specific answers. Clear and concise responses will receive higher marks compared to vague and wordy ones. Note that marks will be deducted for incorrect statements in your answers.

Below is a partial listing of GCC's optimize options.

-01 turns on the following optimization flags:

```
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fmove-loop-stores
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dse
-funit-at-a-time
```

-02 turns on all previous optimization flags, and additionally:

```
-fcode-hoisting
-fcse-follow-jumps  -fcse-skip-blocks
-ffinite-loops
-fgcse  -fgcse-lm
-finline-functions
-finline-small-functions
-fipa-bit-cp  -fipa-cp  -fipa-icf
-fipa-ra  -fipa-sra  -fipa-vrp
-flra-remat
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-loop-vectorize
-ftree-pre
-ftree-slp-vectorize
-ftree-tail-merge
```

Short Answer (15 marks total)

Q1 (1 mark). **True / False** (Circle the correct answer)
An atomic instruction in a processor typically has a latency comparable to an arithmetic add instruction.

Q2 (1 mark). **True / False** (Circle the correct answer)
`gcov` runs slower than `gprof` when profiling the same program.

Q3 (1 mark). **True / False** (Circle the correct answer)
A multi-threaded program is said to be *blocking* if the progress of one thread may depend on the progress of another thread; it is non-blocking otherwise.

Q4 (1 mark). **True / False** (Circle the correct answer)
It is possible for a multi-threaded program to reach deadlock by using only non-blocking synchronizer.

Q5 (1 mark). **True / False** (Circle the correct answer)
All processor hardware must strictly satisfy memory coherence and sequential memory consistency.

Q6 (1 mark). **True / False** (Circle the correct answer)
On a machine equipped exclusively with hard disk drives, the major page fault rate can exceed 1 million faults per second.

Q7 (1 mark). **True / False** (Circle the correct answer)
If one of a process's memory addresses is bigger than a second one, then its corresponding value must appear before the second one's value in physical memory.

Q8 (1 mark). **True / False** (Circle the correct answer)
Ready-Copy-Update (RCU) maintains multiple versions of a data structure simultaneously and is suitable for use cases with many readers and a single writer.

Q9 (1 mark). **True / False** (Circle the correct answer)
In a shared memory multicore machine, the processor may reorder a read of a memory location before a write to the same location in a single thread, thus relaxing the program order, in order to speed it up.

Q10 (1 mark). **True / False** (Circle the correct answer)
For a program running on a superscalar processor, since the hardware is doing the job of instruction scheduling, there is no benefit for a compiler to reorder the instructions.

Q11 (1 mark). **True / False** (Circle the correct answer)
Memory consistency model governs the legal ordering of memory read/write events on multiple memory locations seen by different processor cores.

Q12 (1 mark). **True / False** (Circle the correct answer)
`sbrk` is a primitive used by a memory allocator to extend the heap.

Q13 (1 mark). **True / False** (Circle the correct answer)
False sharing is a form of software bug and a program with false sharing problem is functionally incorrect.

Q14 (1 mark). **True / False** (Circle the correct answer)
By dividing a large critical section protected by a coarse grained lock, into multiple smaller ones, each protected with a separate lock, fine-grained locking is a common method of improving performance of the original program.

Q15 (1 mark). **True / False** (Circle the correct answer)
A program with the longest running loop is made faster by 100 times, but it is still possible that the running time of the entire program runs only slightly faster than before.

Compiler Optimization (6 marks total)

Consider the following code:

```
1  int foo(int *p1, int *p2, int *p3, int x) {
2      int a, b, c;
3      a = *p1;
4      *p1 = 2 + x;
5      smb_wmb();
6      *p2 = 4;
7      b = a + *p1;
8      smb_rmb();
9      c = *p3;
10     d = *p1;
11     return a+b+c+d;
12 }
```

As we know, compilers can take the liberty of reordering statements and expressions to optimize for performance. For the following statement orders, mark true if it is legal for compiler to do so, and false otherwise.

Q16 (1 mark). **True /** **False** (Circle the correct answer)
Line (7) before (3) violation of dependency

Q17 (1 mark). **True /** **False** (Circle the correct answer)
Line (6) before (4) violation of write memory barrier @5

Q18 (1 mark). **True /** **False** (Circle the correct answer)
Line (9) before (6) violation of read memory barrier @ 8

Q19 (1 mark). **True /** **False** (Circle the correct answer)
Line (7) before (6) p1 & p2 may alias, therefore dependency

Q20 (1 mark). **True** **/ False** (Circle the correct answer)
Line (10) before (9)

Q21 (1 mark). **True /** **False** (Circle the correct answer)
Line (8) before (5)

Cache Hierarchy (12 marks total)

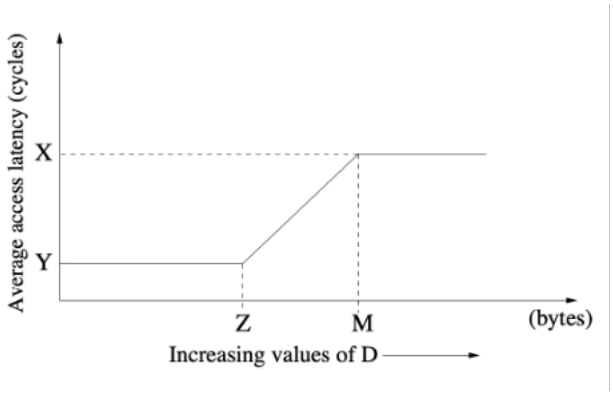
Suppose an architect is designing a simple in-order pipeline processor that blocks on data misses. The data cache of this processor has the following configurations:

Cache size	C bytes
Cache line	B bytes
Cache associativity	4-way, LRU replacement policy
Cache latency	H cycle
Latency between cache and DRAM	P cycles

To help the architect making better decisions, we use micro-benchmark programs. One of them is shown below, which sequentially accesses D elements of array A 100 times. To simplify your reasoning, assume the only memory accesses are to the entries of A; A starts at memory address 0; the cache is initially empty before the outer loop; `sizeof(char)=1`.

```
char A[N]; /* N >> D */
int sum = 0;
    for (i = 0; i < 100; i ++)
        for (j = 0; j < D; j ++)
            sum += A[j];
```

We run the above benchmark code on the processor simulator. By sweeping different values of D, we get the following graph. The horizontal axis is the total number of bytes in the D elements and the vertical axis is the average latency to access an element in the array A.



Determine the following values with the configuration parameters in Table 1 (C, B, 4, H, and P):

Q22 (2 marks).
Y = H
Z = C

Q23 (2 marks). Number of cache lines = C/B

Q24 (2 marks). Number of cache sets = C/(4*B)

Q25 (2 marks). M = 1.25C or 1.25C-B+1

Q26 (2 marks). What's the hit rate when D > M (in the range of the second plateau)?
Hit rate = (B-1)/B

Q27 (2 marks). X = P/B + H

Program Optimization (8 marks total)

Consider the following code that computes the minimum of the pairwise sum of two vectors:

```
#include <limits.h>
#include <pthread.h>
#include <stdlib.h>

#define MIN(x,y) ((x) < (y) ? (x) : (y))
#define INFINITY INT_MAX

#define N (1<<20)
#define P (8)

typedef struct { int* data; int len; } vec_t;
typedef struct { vec_t* a; vec_t* b; int part; int result; } worker_t;

int vec_len(vec_t* v) { return v->len; }
int vec_elem(vec_t* v, int i) { return v->data[i]; }
void vec_init(vec_t* v, int len) { v->len = len; v->data = calloc(len, sizeof(int)); }
void vec_fini(vec_t* v ) { free(v->data); }

void minsum(vec_t* a, vec_t* b, int part, int* result) {
    *result = INFINITY;
    for (int i = 0; i < vec_len(a); i += P)
        *result = MIN(*result, vec_elem(a,i+part) + vec_elem(b,i+part));
}

void* thread_func(void* arg) {
    worker_t* worker = (worker_t*) arg;
    minsum(worker->a, worker->b, worker->part, &worker->result);
    return NULL;
}

int main() {
    pthread_t tid[P];
    worker_t workers[P];
    vec_t a, b;
    int result = INFINITY;

    vec_init(&a, N); vec_init(&b, N);

    for (int i = 0; i < N; i++)  { a.data[i] = i; b.data[i] = N-i-1; }
    for (int p = 0; p < P; p++) {
        workers[p].a = &a; workers[p].b = &b; workers[p].part = p;
        pthread_create(&tid[p], NULL, thread_func, &workers[p]);
    }
    for (int p = 0; p < P; p++) {
      pthread_join(tid[p], NULL);
      result = MIN(result, workers[p].result);
    }
    vec_fini(&a); vec_fini(&b);
    return result;
}
```

Q28 (2 marks). Use the techniques learned in class to improve the implementation of `minsum()` above. Clearly name the techniques utilized (You do not need to do loop unrolling). Show your final optimized `minsum` function.

```
void minsum(vec_t* a, vec_t* b, int64_t* result) {
   int64_t sum = INFINITY;
   int len = a->len;
   for (int i = 0; i < len; i++) {
       int index = i + part;
       sum = MIN(temp, a->data[index], b->data[index]);
   }
   *result = sum;
```

Name 2 of:

1. Inlining
2. Loop-Invariant Code Motion
3. Common subexpression elimination

```
}
```

Q29 (2 marks). The `minsum()` function contains a performance problem when executed on a modern **shared memory multi-core processor**. Clearly identify the problem, and explain why program performance cannot achieve linear speedup with respect to the number of threads (N).

False sharing problem: since each thread access the array with a stride of P, different thread may access the neighboring elements beloning to the same cache line, therefore the processor cores running different threads will be keep invalidating each other.

Q30 (4 marks). Reimplement the `minsum()` function such that the performance problem identified in Part (b) can be mitigated.

```
void minsum(vec_t* a, vec_t* b, int64_t* result) {


    int sum = INFINITY;
    int len = vec_len(a) / P; // we can assume P divides vec_len(a)
    int index = len*part;
    for (int i = 0; i < len; i++) {
        sum = MIN( sum, a->data[index+i]+a->data[index+i] );
    }
    *result = sum;




}
```

Concurrent Memory Allocator (14 marks total)

You are tasked with writing software that needs to process network packets at hardware line rate. An important component of the software is the memory manager for network packets, which needs to perform allocation and releasing at extreme speed in order to keep up with the line rate.

Your manager came up with the idea of implementing a pool allocator, which is very similar to the arena allocator we learned in class, but with the following simplifying assumptions:

- Each network packet has a constant size of `SIZE_PACKET`;
- The number of packet slots (which can be returned as result of allocation) managed by the entire pool is fixed as a power-of-two constant `SIZE_POOL`.
- The allocation state of the packet pool is maintained by a bitmap (freemap), where a i-th bit of the freemap assuming the value of 1 indicates that the i-th slot is free and available for allocation.
- No other metadata other than the freemap is allowed in order to keep the space utilization high.

The header file of the pool allocator, with skeleton of the required data structure and API is shown below.

```
#define SIZE_PACKET (1 << 12)
#define SIZE_POOL (1 << 16)
#define BITS_PER_WORD (sizeof(long)*8)

typedef struct {
    char slots[SIZE_POOL][SIZE_PACKET];
    long freemap[SIZE_POOL/BITS_PER_WORD];
} packet_pool_t;

extern void pool_init(packet_pool_t* pool); /* initialization of packet pool, called @ start */
extern void pool_fini(packet_pool_t* pool); /* finalization of packet pool, called @ exit */
extern char* pool_alloc(packet_pool_t* pool); /* allocate packet memory */
extern void pool_free(packet_pool_t* pool, char* packet); /* free packet memory */
```

Assume the existence of a function called **`ffsl(long i);`** that returns the position of the first (least significant) bit set in the word `i`.

Q31 (2 marks). Implement the pool initializer.

```
void pool_init(packet_pool_t* pool) {


    for(int = 0; i < SIZE_POOL / BITS_PER_WORD; i++)
        pool->freemap[i] = -1;









}
```

Q32 (4 marks). Implement the pool allocator, assuming that the networking software is single threaded.

```
char* pool_alloc(packet_pool_t* pool) {


    int leading_one, i, nwords = SIZE_POOL/BITS_PER_WORD;
    for (i = 0; i < nwords; i++) {
        if(pool->freemap[i] == 0) continue;
        leading_one = ffs(pool->freemap[i]);
        pool->freemap[i] &= ~(1 << leading_one); // clear bit
        return pool->slots[i*BITS_PER_WORD + leading one];
    }
    return NULL;









}

void pool_free(packet_pool_t* pool, char* packet) {


    assert(packet > pool->slots[0] && packet < packet->slots[SIZE_POOL]);
    int index = packet - pool->slots[0];
    int i = index / BITS_PER_WORD;
    pool->freemap[i] |= 1 << (index % BITS_PER_WORD);









}
```

Q33 (2 marks). Assuming the allocators you implemented in the previous question is called by a multi-threaded program running on a shared memory multi-core machine, explain the scenario where the code might not behavior correctly.

> Different threads maybe updating the same freemap word simultaneously. Since the bit update is a read-modify-write, and not done atomically, freemap values may become inconsistent.

Q34 (4 marks). Reimplement the pool allocator, such that it can be correctly used by multi-threaded programs running on shared memory multi-core processor, without using locks or any other blocking synchronization primitives. But you can use one or more of the following atomic primitives.

```
/* test_and_set (TAS): */
bool TAS(bool *lock) {
    /* pseudocode: atomic instruction */
    bool old = *lock;
    *lock = true;
    return old;
}
```

```
/* compare_and_swap (CAS): */
bool CAS(long* p, long old, long new) {
    /* pseudocode: atomic instruction */
    if (*p == old) {
        *p = new;
        return true;
    }
    else {
        return false;
    }
}
```

```
char* pool_alloc(packet_pool_t* pool) {

    int leading_one, i, nwords = SIZE_POOL/BITS_PER_WORD;
    long old, new;
    for (i = 0; i < nwords; i++) {
        if(pool->freemap[i] == 0) continue;
        do {
            leading_one = ffs(pool->freemap[i]);
            old = pool->freemap[i];
            new = old & ~(1 << leading_one);
        } while(!CAS(&pool->freemap[i], old, new));
        return pool->slots[i*BITS_PER_WORD + leading one];
    }
    return NULL;









}
```

```
void pool_free(packet_pool_t* pool, char* packet) {

    assert(packet > pool->slots[0] && packet < packet->slots[SIZE_POOL]);
    int index = packet - pool->slots[0];
    int i = index / BITS_PER_WORD;
    long old, new;
    do {
        old = pool->freemap[i];
        new = old | 1 << (index % BITS_PER_WORD);
    } while(!CAS(&pool->freemap[i], old, new));









}
```

Q35 (2 marks). Comment on the performance of the implementation in the previous question as the number of threads grows. If performance is less than ideal, discuss the potential ways to mitigate the performance problem.

While the program in the previous question is correct, under contention, there will be cache thrashing problem as the different processors are accessing the same word, and they invaldiate each other like a ping-pong.

This situation can be mitigated by:

- Keep a separate pool of packets for each processor (arena idea);
- or, keep a cache of packets for each processor to reduce the frequency of real allocation/free to the shared common pool.

Parallel Programming (8 marks total)

Consider the following code:

```
#include <pthread.h>
int counter = 2;

void foo() {
    counter++;
    printf("%d", counter);
}

pthread_t tid[2];
```

Q36 (2 marks). With the `main()` function below:

```
int main(void) {
    for (int i = 0; i < 2; i ++) {
        pthread_create(&tid[i], 0, foo, 0);
    }
    count++;
  printf("%d", counter);
}
```

What is the `first` number that gets printed? (Circle the correct answer)
(a) 3
(b) 4
(c) 5
(d) Either 3, 4, or 5 **[Correct]**
(e) Either 3, or 4

Q37 (2 marks). With the `main()` function below:

```
int main(void) {
    for (int i = 0; i < 2; i++) {
        pthread_create(&tid[i], 0, foo, 0);
        pthread_join(tid[i], 0);
    }
    count++;
    printf("%d", counter);
}
```

What is the `first` number that gets printed? (Circle the correct answer)
(a) 3 **[Correct]**
(b) 4
(c) 5
(d) Either 3, 4, or 5
(e) Either 3, or 4

Q38 (2 marks). With the `main()` function below:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main(void) {
    for (int i = 0; i < 2; i ++) {
        pthread_mutex_lock(&mutex);
        pthread_create(&tid[i], 0, foo, 0);
        pthread_mutex_unlock(&mutex);
    }
    count++;
  printf("%d", counter);
}
```

What is the `first` number that gets printed? (Circle the correct answer)
(a) 3
(b) 4
(c) 5
(d) Either 3, 4, or 5 **[Correct]**
(e) Either 3, or 4

Q39 (2 marks). With the `main()` function below:

```
int main(void) {
    for (int i = 0; i < 2; i++) {
        pthread_create(&tid[i], 0, foo, 0);
    }
    for (int i = 0; i < 2; i++) {
        pthread_join(tid[i], 0);
    }
    count++;
    printf("%d", counter);
}
```

What is the `first` number that gets printed? (Circle the correct answer)
(a) 3
(b) 4
(c) 5
(d) Either 3, 4, or 5
(e) Either 3, or 4 **[Correct]**

Synchronization (10 marks total)

Consider the following code (assume there are 3 mutexes: `m1`, `m2`, and `m3`):

```
1  void thread1(void) {
2      pthread_mutex_lock(&m1);
3      pthread_mutex_lock(&m2);
4      pthread_mutex_lock(&m3);
5      /* Do something */
6      pthread_mutex_unlock(&m3);
7      pthread_mutex_unlock(&m2);
8      pthread_mutex_unlock(&m1);
9  }
10
11 void thread2(void) {
12     pthread_mutex_lock(&m1);
13     pthread_mutex_lock(&m2);
14     pthread_mutex_lock(&m3);
15     /* Do something */
16     pthread_mutex_unlock(&m1);
17     pthread_mutex_unlock(&m2);
18     pthread_mutex_unlock(&m3);
19 }
```

Q40 (3 marks). Does this code contain a deadlock? If so, write a sequence of line numbers that, when executed in that order, will cause the deadlock.

No.

Now consider the following code:

```
1  void thread1(void) {
2      pthread_mutex_lock(&m1);
3      pthread_mutex_lock(&m2);
4      pthread_mutex_lock(&m3);
5      /* Do something */
6      pthread_mutex_unlock(&m3);
7      pthread_mutex_unlock(&m2);
8      pthread_mutex_unlock(&m1);
9  }
10
11 void thread2(void) {
12     pthread_mutex_lock(&m3);
13     pthread_mutex_lock(&m2);
14     pthread_mutex_lock(&m1);
15     /* Do something */
16     pthread_mutex_unlock(&m3);
17     pthread_mutex_unlock(&m2);
18     pthread_mutex_unlock(&m1);
19 }
```

Q41 (3 marks). Does this code contain a deadlock? If so, write a sequence of line numbers that, when executed in that order, will cause the deadlock.

Line 2, line 12, then either line 3 or line 13.

Consider the following code:

```
1  void thread1(void) {
2      pthread_mutex_lock(&m1);
3      pthread_mutex_lock(&m2);
4      /* Do something */
5      pthread_mutex_unlock(&m2);
6      pthread_mutex_unlock(&m1);
7  }
8
9  void thread2(void) {
10     pthread_mutex_lock(&m3);
11     pthread_mutex_lock(&m1);
12     /* Do something */
13     pthread_mutex_unlock(&m1);
14     pthread_mutex_unlock(&m3);
15 }
16
17 void thread3(void) {
18     pthread_mutex_lock(&m2);
19     pthread_mutex_lock(&m3);
20     /* Do something */
21     pthread_mutex_unlock(&m3);
22     pthread_mutex_unlock(&m2);
23 }
```

Q42 (4 marks). Does this code contain a deadlock? If so, write a sequence of line numbers that, when executed in that order, will cause the deadlock.

Yes, any ordering of lines 2, 10, and 18.