BLOCK PRINTED NAME: _____ BLOCK PRINTED NETID: _____

CS4414/CS5416 Prelim 1. September 25, 2025. 7:30pm-9:00pm

Instructions: Created as a 75m exam, but you can stay for up to 1 ½ hours. Closed book, no technology, but you may use a one-page (front and back) crib sheet created by you. Try to write clearly and to stay in the answer boxes: our scanner doesn't handle scrawled writing or light colors well. We award partial credit but also deduct for wrong statements.

Q1. C++ programming. (20 points, 4 points per sub-question)

Q1.1. Explain the differences between stack and heap memory in C++ in terms of allocation and lifetime. Give one example of a variable (ideally, write the example as code) for each case. **Don't say things here that belong in Q1.2.**

If you declare a variable inline in C++ it will be allocated on the stack:

```
....
Cat kitty("Fluffy");
....
```

The compiler creates the needed space on entering the scope, and later will pop that space and in effect, free it for other uses.

Heap memory is used when you call malloc() or use new to create an object. You get back a pointer to the object in a managed memory segment called the heap, and it is your job to make sure to free the pointer later:

```
Cat* kitty = new Cat("Fluffy");
....
free(kitty);
```

Q1.2. Give one advantage and one disadvantage of storing data on each of the stack and the heap.

Data on the stack makes sense if the variable is playing a temporary role linked to the lifetime of the scope. But precisely because the stack memory is reclaimed when exiting the scope, you can't leave references to the object around after you exit the scope where it was created.

With the heap, the object lifetime is potentially unbounded. But this also means memory will leak (not be reclaimed) if you forget to free the malloc'ed area when finished, or forget to call delete.

Q1.3: Modern computers have NUMA architectures. Explain what this acronym stands for and why it can cause surprising performance impacts.

The acronym means “nonuniform memory access”, which is to say that the speed of access memory varies depending on which CPU a thread is running on and which part of memory is being accessed. There is normally “close-by” memory, and the allocator will try to have your stack and any heap allocations occur close to the thread that did the allocstion

Q1.4: What will this output, and why might its use of consteval be relevant for performance optimization?

```
#include <iostream>
using namespace std;

template<int N>
constevalr int weird_factorial() {
   if consteval (N <= 1) return 1;
   else return (N+1) * weird_factorial<N-1>();
}
```

```
int main() {
   const int result = weird_factorial<5>();
   cout << result << endl;
   return 0;
}
```

Const, constexpr and consteval are all ways to tell the compiler to try and fully compute some value at compile time and not generate runtime code. Consteval is the most “demanding” and it requires compile-time evaluation, precluding generated code.

The compiler will expand the template and end up with a constant calculation of 6*5*4*3*1, hence the program will print 360.

Q1.5: C++ True/False, for 1/2 pt each. Put an X or a check-mark in the appropriate box.

	T	F	
1	T		In a C++ program with exactly 2 threads that read but never update shared memory locations, no synchronization (like mutexes, locks or monitors) is needed.
2		F	Standard objects like std::vector, std::map, etc, do their own locking. Threads can share these objects and read or write to them without adding any additional synchronization.
3		F	When using a circular buffer, deadlock occurs if you don't make the buffer large enough so that producer and consumer threads never need to wait because the buffer is full or empty.
4		F	In a multithreaded C++ program, if a function receives arguments by reference, then no synchronization is needed when accessing those referenced variables inside the function.
5	T		When using the performance profiler gprof on a -O3 optimized C++ program, timing output for templated methods could sometimes be omitted or attributed to the code that used the template rather than to the templated method itself.
6	T		Looking at the code in Q1.4, whereas weird_factorial would always use stack space in Python, in this C++ code it might not consume any runtime stack space at all.
7		F	It is best to never use the C++ "auto" type declaration, because the programmer always needs to know the exact type of every variable.
8	T		When a process is created in Linux, any DLLs it uses will not be loaded into its memory until the first time a method in the DLL is invoked.

Q2. Parallelism. 20 points, 5 pts per sub-part.

Q2.1. Define the terms SISD and SIMD. Don't repeat things here that belong in Q2.2 or Q2.3.

SISD: Single Instruction, Single Data (each instruction operates on a single data item)

SIMD: Single Instruction, Multiple Data (each instruction operates on a vector of items)

Q2.2. Explain what the term “cache-line length” means, and why this information is important when using SIMD programming using the Intel or AMD MMX vector-parallel instructions.

A cache line is a term referring to the width of the connection from a CPU to a cache or to Dram memory. On Intel and AMD machines cache-lines are currently 64 bytes in width. Each CPU to memory interaction transfers an entire cache line as a single action.

SIMD instructions operate on a cache line at a time. For example, 16 4-byte ints, or 512 bits. But to be used, the base address of the data must be an exact multiple of the cache line length, and the size of the vector being processed should be an exact number of cache lines.

Q2.3. Suppose you need to decide between keeping data in a std::vector<int> or keeping it in a std::array<int>. Is one a better choice when you really need MMX parallelism? Why?

With std::vector you do have more features, but many of them result in copying and data might not start on a cache-line boundary. Even if you know it actually does, the C++ compiler won’t be able to figure this out. So there really is no choice: to use the MMX features of C++ you must go with std::array or with a native array.

Q2.4. Suppose that you are computing Boolean-logic operations on arrays of bits, and that the requirements for SIMD MMX parallelism are met. What is the most speedup possible? Explain.

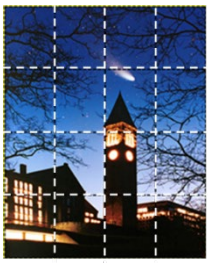
In this case you should have a reasonable chance of getting 512-times faster execution. Obviously this depends on C++ -O3 actually using the vectorizable code compilation features but there is a way to force it to do so and be sure of warnings or errors if it can't.

A fancier answer would refer to Amdahl's law, which is that the peak speedup possible with N CPUs will be limited by the percentage of the code that actually uses parallel instructions. If half the logic is sequential, we aren't going to get better than a 50% speedup.

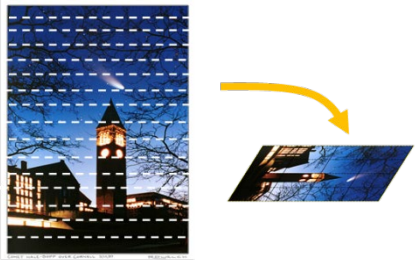
The fanciest answers of all will cite the actual formula Amdahl came up with is $1/(1-p+\frac{p}{N})$ but this year on prelim1, our graders will *not* be awarding extra points for giving this equation or deducting for not having it memorized. One reason is that our slides in lecture initially were for the case where N is infinity, and the equation then simplifies to $1/(1-p)$. Later Ken changed the slides to show the full equation for cases where N might be much smaller, like 2, but we don't want people to worry about that.

Q3. 20 points, 5 points per subpart. Efficient file system interactions.
Q3.1-Q3.3 all relate to the image shown below, which is a famous photo from when comet Hale Bopp was visible over the Cornell clock tower:

Assume that the photo itself is on disk in a very large file. The threads will be processing their data raster by raster from left to right, reading data by using the OS **read** system call, one file block per read. Notice that the number of pixels each thread is responsible for is unchanged.



Chunked approach



Cut into strips

Q3.1. Explain how the data in this photo is represented in computer memory, and what this tells us about what we would find in each block of the file on disk. Also, explain which threading approach (chunked or stripped) would be more performant and why. In your explanation, you can assume that each pixel of our photo is a 4-byte representation of RGB color plus intensity. Assume that disk blocks are 4KB each and that the photo is very high resolution (thousands of pixels per raster).

C++ stores data in row-major order, and each raster is a row in the matrix holding the photo. A photo element would be a single pixel, with an RGB+I representation: 4 bytes. The same data order is used on the disk.

But this tells us that one disk block can only hold 1024 pixels. A raster of our photo will span many data blocks.

The result is that the chunked layout on the left where each thread owns one chunk is not going to perform very well at all. One of those chunks could easily span 2 or 3 blocks per raster, and the same raster in the chunk to the left or right will include data from some of those same blocks. So, to perform image projection on 1000 rasters or so (whatever the height of a chunk) a thread probably reads 2000 blocks.

The full raster representation allows one thread to read chunk by chunk in genuinely sequential order. And it will only need to read half as many blocks.

Q3.2. Now, **focusing only on prefetching**, first **explain how Linux decides to prefetch blocks of a file.** Then, thinking about the comment in lecture about how with 16 threads all reading the same photo file, it would be important to open the same file 16 times, so that each thread would have its own private file descriptor. **Why is this potentially important if we want Linux to realize it should prefetch blocks?**

How Linux decides to prefetch:

Linux will prefetch if it sees 3 or more sequential block accesses from the same source.

Why each of our 16 threads must separately open the file and get its own file descriptor:

With a threaded process that has just one shared file descriptor, each thread will need to lock the file descriptor, then use lseek to jump to the desired block where its raster starts, then read two blocks, then release the lock. This won't do any prefetching.

In the approach where we open the same file once per thread, the different threads are reading from different file descriptors. Now the question becomes: do they read 3 or more blocks sequentially? With the long thin strips, each thread owns a few rasters and can read blocks one by one in order. No locking is needed and Linux will prefetch.

Q3.3. Reading files block by block in 4KB chunks is not the only option. We could also read the entire file into memory at the start of the program, before we even launch the threads that perform the rotation operation. Assume that we are working with a poster-quality copy of the photo and that the size on disk is 2GB (500,000 disk blocks). Our goal is to maximize parallelism: will this be as parallel as the approach where each thread reads block by block?

In this proposal, no computing can really happen until the data is entirely loaded into memory.

So first the process launches, then it loads the entire file, and then the threads can do shared read-only access to the bytes from the file (2GB should fit in memory).

So we have zero concurrency until that reading finishes, but then we get full concurrency from our 16 threads.

Bottom line: we get less parallelism because the file I/O isn't overlapped with the rotation operation.

Q3.4. When we learned about the file system data structure, one surprise was that deleting a file and emptying the recycle bin might not actually delete the data. Briefly explain the role of directory entries, inodes and data blocks in the Linux file system, and then tell us why a file that was deleted could still be accessible.

Directory entries:	A directory is like a file containing a table in which each entry is a "row". It has an inode number and a file name. Deleted files have a 0 inode number.
Inodes:	This is the Linux kernel data structure holding file metadata such as owner, permissions, access times and part of the list of blocks in the file
Data blocks:	Actual data is in disk blocks, 4KB per block.

What Linux does when a file is deleted and why it might still be accessible with forensic tools:

As we learned in lecture 3, a file is deleted by zeroing the inode number in the directory and putting the inode itself and the data blocks on the respective freelists, which tend to be FIFO. That is, to even wear and tear, newly allocated blocks and inodes come from the front of these lists and ones freed recently go to the rear.

Thus if the disk isn't in insanely heavy use, our free file probably still has an intact inode and the blocks have probably not be reassigned and overwritten. A forensic tool will instantly find it. To get the file name right, it will look for directories that were modified recently and do its best to match possible names with files based on things like the file type or contents. That step might not work but at least the file itself can usually be opened and a human will figure out what it seems to be.

Q4. 20 points, 5 points per subpart:

Consider the following code:

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>

class Puppy {
public:
  std::string name;

  Puppy(): name("unnamed"){
    std::cout << "Puppy(name): " << name << "\n";
  }

  Puppy(std::string n): name(n) {
    std::cout << "Puppy(name): " << name << "\n";
  }

  Puppy(const Puppy& other): name(other.name + " [copy]") {
    std::cout << "Puppy(copy): " << name << "\n";
  }

  Puppy(Puppy&& other): name(std::move(other.name)) {
    std::cout << "Puppy(move): " << name << "\n";
  }

  ~Puppy() {
    std::cout << "~Puppy: " << name << "\n";
  }
};

class Kennel {
public:
  std::vector<std::shared_ptr<Puppy>> pups;

  // (for part 4.2)
  void admit(std::shared_ptr<Puppy> p) {
    pups.push_back(std::move(p));
  }
};
```

Q4.1:

If we have a global (“free”) function:

```
void admit(Kennel k, std::shared_ptr<Puppy> p) {
   k.pups.push_back(p);
}
```

and then call:

```
Kennel kn;
auto pup = std::make_shared<Puppy>("Goji");
admit(kn, pup);
```

After this sequence, is pup in kn.pups? Explain. **Do not show us the expected output.**

The code is buggy: it passes kn by value, hence makes and then modifies a copy of the Kennel object, not the original. kn is unchanged and its version of pups will be empty

Q4.2:

If we instead use the member function of Kennel:

```
Kennel kn;
auto pup = std::make_shared<Puppy>("Goji");
kn.admit(pup);
```

Does pup get added to kn.pups? What happens to the reference count of pup?

Yes, this time the kn.pups field will have a new shared_ptr<Puppy> added to it. Along the way the initial reference count was set to 1 by make_shared. Then, the admit function parameter p copies the pointer, setting the reference count to 2. Then, the copied pointer is moved into the pups vector, keeping the reference count at 2.

Q4.3

What will this program output? Walk us through the sequence of events and reference count updates.

```
void addGojiPuppy(Kennel& k) {
   Puppy pup("Goji");
   auto pointer = std::make_shared<Puppy>(pup);
   k.admit(pointer);
}

int main() {
   Kennel kn;
   addGojiPuppy(kn);
   return 0;
}
```

```
Puppy(name): Goji                 // Original construction, from line 2 of addGojiPuppy
Puppy(copy): Goji [copy]          // Copy created by the call to make_shared
~Puppy: Goji                      // When the instance created in line 2 goes out of scope
~Puppy: Goji [copy]               // When the Kennel kn goes out of scope
```

Q4.4

Our solution used smart pointers. What is the advantage of this? Are there any overheads?

Shared_ptr<T> is a convenient way to do a kind of memory management in C++. The object has a pointer to the underlying object, which it allocates when make_shared<T> is called, and a reference count (initially 1). It has a getter that overloads the -> operator to use this pointer, but if a copy is made, the reference counts will be shared and will increment for each copy and decrement when the destructors run. The underlying object is freed if the count goes to 0.

The advantage is that you won't leak memory as references go out of scope, but the implementation of shared_ptr<T> has some overheads (for counting references), does call delete which means a call to free might occur at moments you might have hoped not to see free run (free is not, in fact, free), and you need to do locking if threads share the same shared ptr.

Q5. 20 points, 5 points per subpart:

Q5.1:

This code includes some logic that probably was not what the programmer intended. What is it doing and why is this not likely to be correct? How would you fix the issue?

```
#include <iostream>

void swap(int* x, int y) {
   int tmp = *x ;
   *x = y ;
   y = tmp;
}

int main() {
   int x = 10;
   int y = 20;
   swap(&x, y);

   std::cout << x << "\n";
   std::cout << y << "\n";

   return 0;
}
```

It looks like the code is supposed to swap the value in the integer pointed to be x with the value in integer y. But because y is passed by value, in fact y itself won't end up holding the original value x pointed to: x and y will both be 20 after swap runs.

The fix is to declare y as a by reference argument: "int& y"

Q5.2:

Explain the & notation used in the addOne method declaration. What will this program output? We often combine & with const, as is “const int&”. Can that be done here? Why or why not?

```
#include <iostream>

void addOne(int& x) {
   x += 1;
}

int main() {
   int x = 10;
   addOne(x) ;

   std::cout << x << "\n";

   return 0;
}
```

The by reference notation means that x in AddOne will be an alias (a secondary name) for the version of x in main(). When we add 1 in AddOne, it actually mutates that original copy.

So, the program outputs 11.

Q5.3:

Define the term “memory leak” and give a very small piece of code illustrating a leak.

A memory leak occurs if we allocate something but never free it. For example:

```
for(int k = 0; k < 10; k++)
    malloc(100 + k*10);
```

allocates a series of objects of increasing size, but does nothing with the pointers and never calls free. That memory has “leaked”.

Objects can leak too, if they are heap allocated and the pointers to them are dumb pointers and all of them go out of scope or are reused to point to other things.

Q5.4:

How is the Valgrind tool used and why is it important when building high-quality C++ programs?

Valgrind is a tool that tracks all memory use in C++ programs. It is very slow but will report on any leaks it finds. It also notices and reports array out of bounds errors.