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.**

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

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

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;
}
```

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

	T	F	
1			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			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			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			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			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			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			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			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.

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.

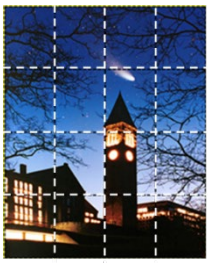
Q2.3. Suppose you need to decide between keeping data in a std::vector<int> or keeping it in a std::array<int>. Is one a better choice when you really need MMX parallelism? Why?

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.

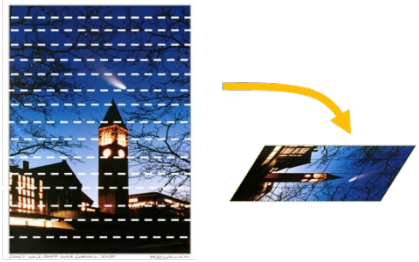
Q3. 20 points, 5 points per subpart. Efficient file system interactions.

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

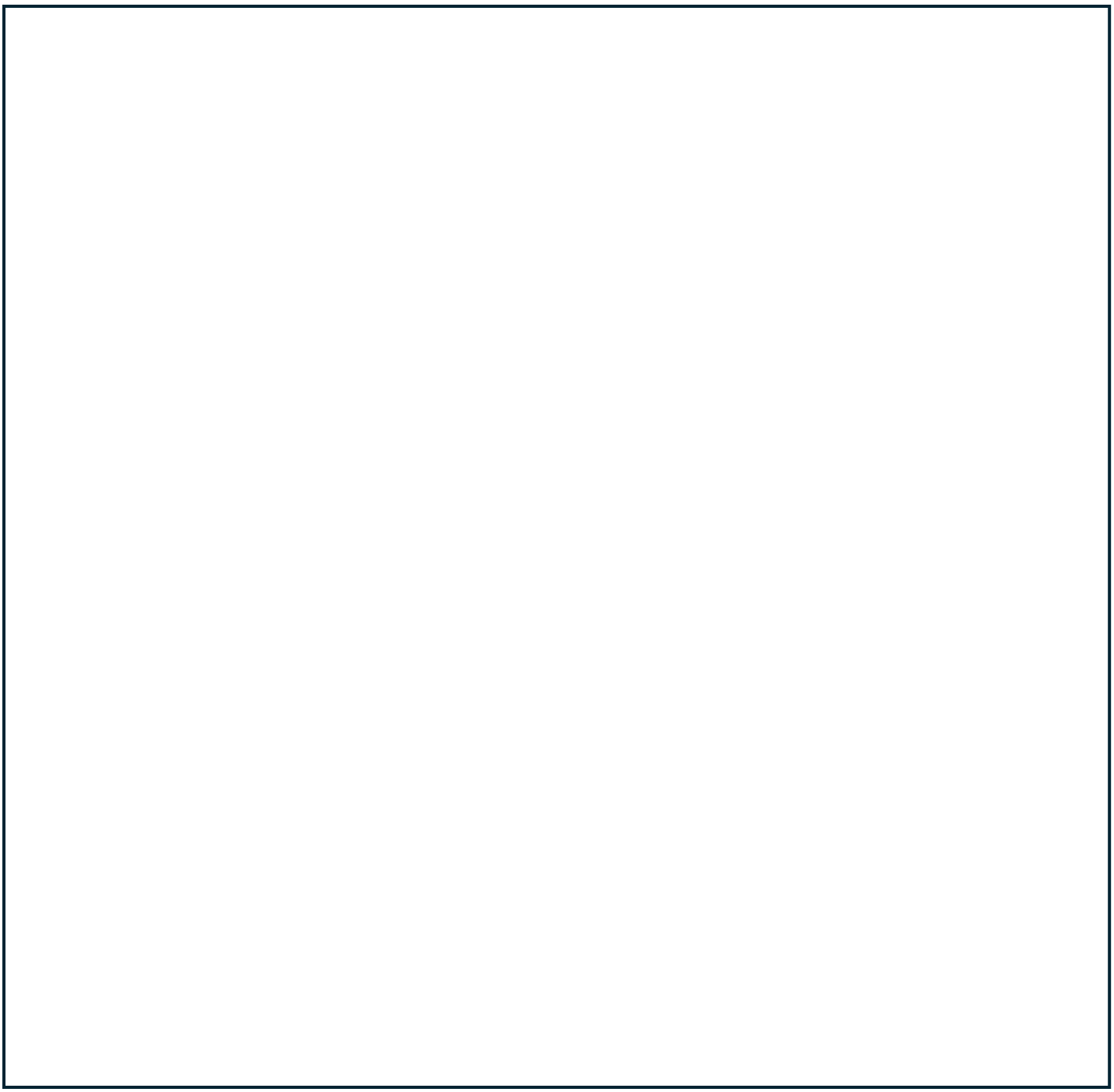
Q3.1. Explain how the data in this photo is represented in computer memory, and what this tells us about what we would find in each block of the file on disk. Also, explain which threading approach (chunked or stripped) would be more performant and why. In your explanation, you can assume that each pixel 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).

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:

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

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?



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:

Inodes:

Data blocks:

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

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 Q4.2)
 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 inside kn.pups? Explain. **Do not show us the expected output.**

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?

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;
}
```

Q4.4

Our Kennel class uses smart pointers. What is the advantage of this? Are there any overheads?

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, what will it print, 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;
}
```

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;
}
```

Q5.3:

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

Q5.4:

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