CMPSCI 377 Operating Systems **Fall 2008**

Lecture 3: September 9

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

3.1 Operator Overloading in C++

It allows you to provide an intuitive interface to users of your class, plus makes it possible for templates to work equally well with classes and built-in/intrinsic types. Operator overloading allows C/C++ operators to have user-defined meanings on user-defined types (classes). Overloaded operators are syntactic sugar for function calls:

```
class Fred {
 public:
   ...
 };

 #if 0

   // Without operator overloading:
   Fred add(const Fred& x, const Fred& y);
   Fred mul(const Fred& x, const Fred& y);

   Fred f(const Fred& a, const Fred& b, const Fred& c)
   {
     return add(add(mul(a,b), mul(b,c)), mul(c,a));    // Yuk...
   }

 #else

   // With operator overloading:
   Fred operator+ (const Fred& x, const Fred& y);
   Fred operator* (const Fred& x, const Fred& y);

   Fred f(const Fred& a, const Fred& b, const Fred& c)
   {
     return a*b + b*c + c*a;
   }

 #endif
```

By overloading standard operators on a class, you can exploit the intuition of the users of that class. This lets users program in the language of the problem domain rather than in the language of the machine. The ultimate goal is to reduce both the learning curve and the defect rate.

3.2 The C++ Standard Template Library

As you know already, it is tough to program without good data structures. Fortunately, most C++ implementations have them builtin! They are called the Standard Template Library (STL). The two that you may need for 377 are queues and maps. You should know what these are already. Using the STL is pretty straightforward. Here is a simple example:

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

queue<int> myQueue;
int main(int argc, char * argv[]){

      myQueue.push(10);
      myQueue.push(11);

      cout << myQueue.front() << endl; // 10
      myQueue.pop();

      cout << myQueue.front() << endl; // 11
      myQueue.pop();

      cout << myQueue.size() << endl; // Zero
}
```

The type inside of the angle brackets says what the queue myQueue will hold. Push inserts a **COPY** of what you pass it. Front gives you a reference to the object, unless you copy it. Pop, pops it off the queue and discards it. Often you will have a queue or a map of pointers. Here is a more complex example you should now understand:

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

class IntCell
{
   public:

   IntCell( int initialValue );
   int getValue( );
   int setValue( int val );

   private:

   int storedValue;
};

IntCell::IntCell (int initialValue = 0){
   storedValue = initialValue;
}

int IntCell::getValue( ){
   return storedValue;
```

```
}

int IntCell::setValue( int val )
{
   storedValue = val;
}

// In a map the first parameter is the key
map<int, IntCell *> myMap;

int main(int argc, char * argv[]){
   IntCell *a;
   int i, max = 100;

   for (i = 0; i < max; i++){
      a = new(IntCell);
      a->setValue(max-i);
      myMap[i] = a; // Inserts a copy of the pointer
   }

   for (i = 0; i < max; i++){
      a = myMap[i];
      cout << a->getValue() << endl;
      delete (a);
      myMap[i] = NULL; // Good idea?
   }
   myMap[0]->setValue(0); // Quiz: can I do this?
}
```

Think about what the output from this should be.

3.3 Computer Architecture

Although some computer architecture problems underly a lot of the science involved in designing Operating Systems, in this class we will review just some of the important topics that are relevant to *make OSs **FAST***. Specifically, in this class we will focus on memory hierarchy and on good policies for improving caching mechanisms.

3.3.1 Memory Hierarchy

Modern computers have several types of memory. The most obvious ones are the RAM and the hard disk. There are, however, at least two other types of memory that are important: the *registers* and the *cache memory*.

3.3.1.1 Registers (and a little bit on context changing)

Registers are specialized pieces of memory in the CPU, and can be seen as dedicated names for specific *words* of memory. Some of them are of general purpose (such as AX, BX, CX, which on a x86 can be used for addition, multiplication, etc). Other registers have special purposes: the SP (Stack Pointer), for instante,

is used to point to one of the ends of the stack; the PC (Program Counter) is used to point to the current instruction being executed; and so on.

One of the problems that an Operating System has to deal with is how to make a machine with several simultaneous processes get by with just only one set of registers. In other words, we know that the CPU has only one Program Counter, one Stack Pointer, etc; also, each process has a different instruction being executed, a different stack pointer, and so on; how, then, is it possible for the OS to manage multiple concurrent processes with just one set of registers available? The secret is to realize that, in fact, *on a single CPU there are no two processes running at the same time*. What happens is that the Operating System keeps alternating between processes vary fast, giving us the impression of true concurrency. From now on, we will call this replacement of the active process by another one by the name of *context switching*.

During a context switch, what the OS has to do is to **save the current register's values** (ie, copy them from the CPU to the RAM) and to **reload the saved registers** of the process being activated (ie, to copy them from the RAM to the CPU). In this way, even though at any given time there might be a bunch of saved registers, there will be only *one set of registers* actually being used: that of the currently active process.

3.3.1.2 Cache memory

As mentioned before, registers are incredibly fast pieces of memory, located straight into the CPU; they are, however, few. RAM, on the other hand, is much slower than registers, but has a much bigger capacity. Is there any type of memory in between? Yes, the *cache memory*.

Consider that every access to the main memory is expensive in terms of CPU cycles (typical access times are around 100 cycles). The role of the cache memory is to a faster-than-RAM memory, but at a more affordable cost. Since cache memory is not so big as the main memory, we need smart ways to manage it.

Usually, caches hold **recently-accessed data or instructions**. The crucial assumption that justifies this approach is that data recently accessed might be needed again in a near future, and that data *near* the one just accessed might also be needed soon.

Notice that even though cache is the generic term to denote this type of memory, there are in fact several types of cache: the **L1 cache**, which is an on-chip (on the CPU) memory; smallish; very fast; very expensive; but with very low capacity (usually $\tilde{3}$2k, $\tilde{6}$4k); the **L2 cache**, usually on or next the the processor; larger than L1 and still more expensive than RAM, but faster; and with capacity in the order of magnitude of megabytes; and the **L3 cache**, which is pretty large, on bus[1].

One of the most important ways to measure the performance gained by means of caching is the *hit rate*. The hit rate refers to the percentage of accesses to data that is cached, and that therefore does not need to be fetched from the main memory. Very small differences in the hit rate can make a **huge** difference on performance (eg: if the hit rate falls from 99% to 99%, the computer can become even thousands of times slower).

When the cache starts being used we say that it is **cold**, meaning that the data we will soon need is not yet cached. Notice that when the cache is cold, it is unavoidable to incur in some initial misses; these are called **compulsory misses**. When they occur, the computer needs to fetch the needed data on the main memory and then uses it to populate the cache. One important detail to mention, however, is that the cache is **not** populated with individual bytes; instead, full *lines* are brought to the cache (eg: 128 bytes at a time). The assumption behind this decision is that usually data presents spacial locality, ie, bytes near the one just accessed are likely to be needed in a near future. After the cache has been filled in, we said it has

[1] Compare the orders of magnitude of the cycles required to access each type of memory: registers=1 cycle of latency; L1 cache = 2 cycles; L2 cache = 7 cycles; RAM = 100 cycles; Disk = 40 million cycles; Network = 200 million cycles

been *warmed up*. Cache memory, then, is expected to be holding the most-frequently used data. Besides compulsory misses, we can also mention other two types of misses: **capacity misses**, which occur due to the finite size of the cache[2]; and **conflict misses**, which occur whenever a different memory location had to be loaded into the same cache line as the one used by the needed data.

Since the cache is finite (and because of associativity – to be discussed next), a policy to manage the cache memory is needed. Ideally, the cache should be as large as the memory (in which case we wouldn't *need* any main memory, of course!); since it cannot be that large, we must find a smart way to decide which data to keep and which data to remove from the cache, when we run out of space. The most popular policy for managing cache is known as the **Least Recently Used** (LRU) policy. According to LRU, whenever we need to free some space in the cache, we throw away the data we used the further in the past. The assumption that justifies this behavior is that we often can use the past as a predictor of the future. Thus, it s reasonable to guess that the things we've used the further in the past will probably be needed the farthest in the future. Of course this is not a certainty; maybe the line just evicted will be needed two instructions ahead. We don't know. Even then, LRU turns out to be a pretty decent policy for managing cache.

Another important issue related to caching is that of **associativity**. Ideally, we would want that any memory position from the main memory could be stored in any part of the cache memory. This would be a *fully associative cache*; however, this type of cache is *very* expensive to manufacture, and requires very complicated logic. Thus, it is usual to restrict the places in the cache where each memory position (of the main memory) can be saved. For example: in a 2-way set associative cache, any piece of memory can end up in one of two places; in a 1-way direct mapping cache, on the other hand, each piece of memory always goes to the same place on the cache[3].

Finally, it is important to understand the implications of a context switch on the cache. Since the new process being activated is unlikely to share data or instructions with the current process, it is reasonable to assume that the cache will turn cold again. This brings us to the conclusion that context switches make the data on the cache useless, causing misses and increasing the latency.

3.3.2 Kernel mode and system calls

It is a function of the kernel to protect of OS itself from users, and users from other users. In order to do so, the kernel only allows privileged code to execute in what is called the *kernel mode*. Eg: when a system call is made for reading something from the disk, the OS makes sure (among other things) that the user which requested the reading is not trying to access other user's files.

Everytime the CPU goes into kernel mode, all pipelines are flushed, the context is saved, the corresponding code is executed in kernel land, and then the system returns to user mode, restoring the context. It is also important to realize that the hardware actually does have some support to ease the implementation of this separation between user and kernel modes.

3.3.3 Timers and interrupts

It is obvious that modern OSs have to deal with multiple processes running at the same time. Also, the system must remain responsive, in the sense that IO must be permitted even when the CPU is busy processing. In order to make the system respond to certain events periodically, timers and interrupts are used. Eg: when the timer that gives the time limit for process execution goes off, the processor is interrupted; the current process stops; the OS takes control through the interrupt handler; and, finally, the scheduler chooses the

[2]Ie, these occur when the needed data had to be evicted due to lack of space, and thus is no longer present in the cache

[3]This can be a problem if, for instance, A and B map to the same place, and we need to read A and B in turns.

next process. Interrupts also signal IO events, such as the arrival of a network packet, when a disk read is complete, etc.

3.3.4 Traps and Interrupts

A great deal of the kernel consists of code that is invoked as the result of a interrupt or a trap. While the words "interrupt" and "trap" are often used interchangeably in the context of operating systems, there is a distinct difference between the two. An interrupt is a CPU event that is triggered by some external device. A trap is a CPU event that is triggered by a program. Traps are sometimes called software interrupts. They can be deliberately triggered by a special instruction, or they may be triggered by an illegal instruction or an attempt to access a restricted resource. When an interrupt is triggered by an external device the hardware will save the the status of the currently executing process, switch to kernel mode, and enter a routine in the kernel. This routine is a first level interrupt handler. It can either service the interrupt itself or wake up a process that has been waiting for the interrupt to occur. When the handler finishes it usually causes the CPU to resume the processes that was interrupted. However, the operating system may schedule another process instead. When an executing process requests a service from the kernel using a trap the process status information saved, the CPU is placed in kernel mode, and control passes to code in the kernel. This kernel code is called the system service dispatcher. It examines parameters set before the trap was triggered, often information in specific CPU registers, to determine what action is required. Control then passes to the code that performs the desired action. When the service is finished, control is returned to either the process that triggered the trap or some other process. Traps can also be triggered by a fault. In this case the usual action is to terminate the offending process. It is possible on some systems for applications to register handlers that will be evoked when certain conditions occur – such as a division by zero.

3.3.5 Synchronous and Asynchronous I/O

There are two types of input/output (I/O) synchronization: synchronous I/O and asynchronous I/O. Asynchronous I/O is also referred to as overlapped I/O. In synchronous I/O, a thread starts an I/O operation and immediately enters a wait state until the I/O request has completed. A thread performing asynchronous I/O sends an I/O request to the kernel by calling an appropriate function. If the request is accepted by the kernel, the calling thread continues processing another job until the kernel signals to the thread that the I/O operation is complete. It then interrupts its current job and processes the data from the I/O operation as necessary. In situations where an I/O request is expected to take a large amount of time, such as a refresh or backup of a large database or a slow communications link, asynchronous I/O is generally a good way to optimize processing efficiency. However, for relatively fast I/O operations, the overhead of processing kernel I/O requests and kernel signals may make asynchronous I/O less beneficial, particularly if many fast I/O operations need to be made. In this case, synchronous I/O would be better.

3.3.6 Memory-Mapped I/O

Memory-mapped I/O is an I/O scheme where the device's own on-board memory is mapped into the processor's address space. Data to be written to the device is copied by the driver to the device memory, and data read in by the device is available in the shared memory for copying back into the system memory. Memory-mapped I/O is frequently used by network and video devices. Many adapters offer a combination of programmed I/O and memory-mapped modes, where the data buffers are mapped into the processor's memory space and the internal device registers that provide status are accessed through the I/O space. The adapter's memory is mapped into system address space through the PCI BIOS, a software setup program, or by setting jumpers on the device. Before the kernel can access the adapter's memory, it must map the

adapter's entire physical address range into the kernel's virtual address space using the functions supplied by the driver interface.

3.3.7 Virtual Memory

Virtual memory is a technique which gives an application program the impression that it has contiguous working memory, while in fact it may be physically fragmented and may even overflow on to disk storage. Systems that use this technique make programming of large applications easier and use real physical memory (e.g. RAM) more efficiently than those without virtual memory. Note that "virtual memory" is not just "using disk space to extend physical memory size". Extending memory is a normal consequence of using virtual memory techniques, but can be done by other means such as overlays or swapping programs and their data completely out to disk while they are inactive. The definition of "virtual memory" is based on tricking programs into thinking they are using large blocks of contiguous addresses. All modern general-purpose computer operating systems use virtual memory techniques for ordinary applications, such as word processors, spreadsheets, multimedia players, accounting, etc. Few older operating systems, such as DOS of the 1980s, or those for the mainframes of the 1960s, had virtual memory functionality.