Software Systems

UMASS AMHERST
Department of Computer Science

December 13, 2009

Contents

Chapter 1

General information

This class is taught by Prof. Emery Berger in ELAB 323, every Tuesday and Thursday from 9:30am to 11:00am. ELAB is the older brick building next to the computer science building. For additional information related to the course (syllabus, projects, etc), please visit `www.cs.umass.edu/~emery`.

Although the class name *is* Operating Systems, this course will deal mostly with large-scale computer systems. The main question to be studied is *how to build correct, reliable and high performance computer systems*. In order to answer this question, problems such as memory management, concurrency, scheduling, etc, will be studied.

1.1 Getting help

Prof. Berger is available by appointment in his office (CS344). The TA's office hours are Mondays and Wednesdays from 4:00 PM to 5:00 PM in the Edlab (LGRT 223, 225).

In addition to office hours, Discussion Sections will be led by the TA on Wednesdays from 12:20pm to 1:10pm in CMPSCI 142. Attendance is very important since sample test problems will be discussed, and the main concepts presented during the last couple of classes will be reinforced. Also, students looking for help can access the course's newsgroup (look for the link at Prof. Berger's website). Lecture notes will also be available. Although this class requires no textbook, interested students might look for classics of the area, such as *Operating System Concepts* (Silberschatz et all), 7th Edition.

1.2 Grading

Student's grades will depend both on exams (2 exams, 25% of the final grade each) and on projects. Prof. Berger adopts a very strict late policy, so please keep that in mind when deciding when to start working on your assignments. There will be four projects; the first one will be worth 5% of the overall grade, and the last three will each be worth 15% of the overall grade. Students will have 3 bonus submissions per project. The grading of all projects will be performed by an autograder system, so you will have the chance to assess how well you're doing several days before the due date.

Do not cheat! An automatic system for finding cheaters will be used, and you *will* be caught.

Chapter 2

Introduction to Operating Systems

The term **Operating System** (OS) is often misused. It is common, for example, for people to speak of an OS when they are in fact referring to an OS *and* to a set of additional applications (e.g. on Windows, Notepad, Windows Explorer, etc).

The traditional view of the area, however, defines an OS in a different way. The OS can be seen as the layer and interface that stands between the user-level applications and the hardware. Its main goal is to hide the complexity of the hardware from the applications. The important concept here is *abstraction*: an OS abstracts architectural details, giving programs the illusion of existing in a "homogeneous" environment. The OS effectively makes programs believe that they live in a reliable machine with large amounts of memory, a dedicated processor, and so on. It is also the OS's function to manage the computer's resources (e.g. the OS decides which process to runs when, for how long, etc).

2.1 A Brief History of Operating Systems

Operating Systems have evolved tremendously in the last few decades.

The first approach for building Operating Systems, taken during the 40s through early 60s, was to allow *only one user and one task at a time*. Users had to wait for a task to be finished before they could specify another task, or even interact with the computer. In other words, not only were OS's mono-user and mono-task, there was no overlapping between computation and I/O.

The next step in the development of OS's was to allow batch processing. Now, multiple "jobs" could be executed in a batch mode, such that a program was loaded, executed, output was generated, and then the cycle restarted with the next job. Although in this type of processing there was still no interference or communication between programs, some type of protection (from poorly or maliciously written programs, for instance) was clearly needed.

Allowing overlap between I/O and computation was the next obvious problem to be addressed. Of course, this new feature brought with itself a series of new challenges, such as the need for buffers, interrupt handling, etc.

Although the OS's from this time allowed users to interact with the computer while jobs were being processed, only one task at a time was permitted. Multiprogramming solved this, and it was a task of Operating System to manage the interactions between the programs (e.g. which jobs to run at each time, how to protect a program's memory from others, etc). All these were complex issues that effectively led to OS failures in the old days. Eventually, this additional complexity began to require that OS's be designed in a scientific manner.

During the 70s, hardware became cheap, but humans (operators, users) were expensive. During this decade, interaction was done via terminals, in which a user could send commands to a mainframe. This was the Unix era. Response time and thrashing became problems to be dealt with; OS's started to treat programs and data in a more homogeneous way.

During the 80s, hardware became even cheaper. It was then that PCs became widespread, and very simple OS's, such as DOS and the original Mac OS, were used. DOS, for example, was so simple that it didn't have any multiprogramming features.

From the 90s on (until today), hardware became *even* cheaper. Processing demands keep increasing since then, and "real" OS's, such as Windows NT, Mac OS X and Linux, finally became available for PCs. Operating systems are now used in a wide range of systems, from cell phones and car controller computers, to huge distributed systems such as Google.

Computer technology has advanced *9 orders of magnitude* (in terms of speed, size, price) in the last 50 years. Moore's law also seems to be running out of steam, mainly due to fundamental physics limits. Though details are difficult to predict, we can make some guesses on what to expect in the next few years: a continuing shift to multiple cores and processors, serious power/heat constraints, trading off computer power for reliability, dealing with unreliable memory, a continuing growth of large distributed systems, etc. Operating systems and systems software will need to continue to evolve to work with these types of systems.

2.2 More on the focus of this course

Again, this course will focus more on building large-scale computer systems rather than on traditional operating systems. We will focus on what is needed to build systems software which is reliable, fast, and scalable, at a layer above the traditional operating system kernel. Some of the topics we will cover include:

- On each computer:
  - C and C++ (providing access to details such as pointers and more)
  - concurrency and scheduling
  - memory management and locality
  - disks, network, and filesystems
- Across each cluster:

  - server architectures
  - distributed computing and filesystems

Chapter 3

C++

Most of this lecture is covered in the Appendix, Section A. Here are a few additional notes:

3.1 Why C/C++?

One reason to learn C and C++ is simply that so much software is written in these languages. A related, but more fundamental reason, is that C and C++ are relatively low-level, allowing efficient use of resources as well as direct access to pointers, critical for operating systems.

C is low-level, fairly close to what-you-see-is-what-you-get, with effectively no runtime system. (A runtime system is essentially an abstraction layer between the operating system and the application.) C is efficient in code space and execution time.

C++ is an upward-compatible (almost completely) object-oriented extension of C. This allows software-engineering benefits, including the use of classes, encapsulation ("private"), templates ("generics"), and other modularity advantages.

3.2 Pointers

Pointers are just numbers, representing addresses in memory. You can add to and subtract from pointers, for instance. It is pretty easy to make mistakes with pointer math, however, and sometimes those sorts of bugs can be hard to catch, so be careful. Don't dereference pointers unless you know that the address is valid, and that you know what data type is at that memory address.

With arrays, generally be careful. C/C++ does not check array bounds, and the problems with pointer math can occur with arrays as well, since arrays are just "syntactic sugar" barely covering up the underlying pointers.

3.3 Memory

C and C++ require explicit dynamic memory management, using `new` and `delete` or `malloc()` and `free()`.

It is helpful to understand where variables exist (usually the stack or the heap, sometimes the data segment[1]).

The stack is managed for you by the compiler, so it's usually the easiest memory to use. Local variables go on the stack, and passed function parameters go on the stack[2]. Since the stack frame can change significantly between uses, *do not return pointers to the stack!* This is analogous to dereferencing freed memory.

If a variable needs to exist longer than a function call, then you should put allocate space for it on the heap (with `new`, for instance). If you allocate space for a variable, remember to free that space when you're done with it! If you allocate memory but don't free it, you'll end up with memory leaks, which usually becomes a problem when a program is supposed to run for a long time, repeatedly allocating and forgetting to free. It can be helpful to write allocating functions/methods at the same time as freeing functions/methods, so that you don't forget to deallocate memory from the heap.

3.4 C++ Standard Template Library (STL)

The STL details are described in many places online (see the CS377 webpage for some links), and there's a very quick introduction in Section 14 of the *377 Student Guide to C++*. Here are just a few additional notes:

Function calls in C++ pass by value, making a copy of the argument. If you need to modify the argument, or if making a copy of a large object would be consume too much time and memory, then passing by reference (a pointer) is often preferable.

Similarly, many of the STL methods (e.g. insert(), etc) are set up to pass potentially large objects by value, rather than by reference. For this reason, one might want to create objects composed of pointers, rather than objects. For example, if you wanted a queue-of-queues, with larger datasets, you'd probably want to use a queue-of-pointers-to-queues instead, such as

```
queue<queue<int>*> my_queue;
```

rather than

[1]Global variables, and local variables which are declared `static`, are located in the program's data segment, which doesn't change dynamically the way the stack and heap do. As a general rule, especially for CS377, try to avoid using global variables and static local variables. Now that I've opened this can of worms, C/C++ use `static` to mean multiple different things (see Section 16.6 of the *377 Student Guide to C++*). So if you do use a global variable, it's often helpful to declare it as `static` so that it is only visible in that file.

[2]Usually passed function parameters go on the stack, but not always. Many compiler/runtime systems use a mixture of register and stack passing for function parameters. This depends on the details of the specific compiler/runtime system, and is *not* important for this class.

```
queue<queue<int> > my_queue;
```

(note the space in the second declaration, because >> is a C++ operator).

As a couple of very quick STL examples, consider

```
queue<int *> myqueue;
int *ptr = new int;
myqueue.push(ptr);
```

and

```
map<int,char> mymap;
mymap[10]='a';
```

The second example is a map indexed by `int`s, storing `char`s.

3.4.1 C/C++ `main(int argc, char *argv[])`

Command-line parameters are passed into programs using the arguments of `main()`. Here's a quick example, for a program called by typing "`progname file.txt 1 2.7`":

```
#include <iostream>
#include <stdlib.h>
using namespace std;

int main(int argc, char *argv[])
{
  char *progname, *filename;
  int value1;
  float value2;

  if (argc != 4) {
        cerr << "usage:␣" << argv[0] << "␣file.txt␣int␣float" << endl;
        exit(1);
  }

  progname = argv[0];          // argv[0] is the executable's name itself
  filename = argv[1];
  value1   = atoi(argv[2]);   // convert C-string to int with atoi()
  value2   = atof(argv[3]);   // convert C-string to float with atof()

  cout << progname << "␣" << filename << "␣" << value1 << "␣" << value2 <<endl;
  exit(0);
}
```

Chapter 4

Processes and Threads

Processes and threads each have their place in multi-programming, generally to hide latency and to maximize CPU utilization.

With the continuing spread of multi-core processors in personal computers, threads are becoming more and more important every day. There are now threads in almost every type of application, including client applications, not just server applications. Soon, there are likely to be multi-core processors even on cellphones, and there will be threaded applications on cellphones.

These multiple cores make it possible to run several different lines of processing at the same time, allowing the computer to run much faster than usual; because of this, however, programmers must now explicitly make use of multi-threaded programming. Unfortunately, parallel programming is very confusing and error prone.

In general, parallel programming can be implemented either by using several concurrent processes, or by using threads. While processes have each their own address space (separate program counters, heaps, stacks, etc), threads share their address spaces; the programmer can use either of these to obtain concurrency. Besides maximizing CPU utilization, the use of parallel programming also helps to hide latency (e.g. waiting for the disk while using the CPU) and to handle multiple asynchronous events.

As a rough analogy, different processes are like different housing subdivisions: quite separate and protected from each other, and with communication possible but relatively difficult. In this analogy, different threads within the same process are like roommates: communication and sharing is easy, but there's much less protection from each other's mistakes.

4.1 Unix process API

The two most important function calls to use when programming with several processes are `fork` and `exec`:

- **fork()** creates a copy of current process. It gives a different return value to each process and works based on Copy On Write;

- **exec()** replaces a process with an executable.

(The Windows CreateProcess(...), taking ten arguments, is analogous.)

Notice that fork() implies that each process descends from another process. In fact, in Unix everything descends from a single process called *init*: basically, init forks a process and then "replaces its code" with, say, the code of bash, using exec().

Example of how to use fork:

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int parentid = getpid();
char program_name[1024];
gets(program_name);   // reads the name of program we want to start
int cid = fork();
if (cid==0) { // i'm the child
    execlp(program_name, program_name, 0); // loads the program and runs it
    printf("if the above worked, this line will never be reached\n");
}
else { // i'm the parent
    sleep (1);   // give my child time to start
    waitpid(cid, 0, 0);   // waits for my child to terminate
    print("program %s finished\n", program_name);
}
```

Is the `sleep(1)` call necessary to allow the child process to start? The answer is *no*, it is not at all necessary. In general, if you think you need to sleep in a program, you are probably doing something wrong, and just slowing down your program. The call to `waitpid()` is a blocking wait, and will first wait to let the child process start (if it hasn't already), then will wait until it ends.

4.2 Interprocess Communication

Now we consider the following questions: how can the parent process communicate with its child? Or how can children processes communicate with other children? The exact answer depends on the exact problem being treated, but in general we call the several different approaches to this question by the name of Interprocess Communication (IPC).

On possibility for IPC is to use sockets. This approach is based on explicit message passing, and has the advantage that processes can be distributed anywhere on the Internet, among several different machines. An application designed to use sockets can fairly easily be re-deployed as a multi-server application when its needs outgrow a single server.

Another possibility is the use of mmap, which is a hack, although a very common one. Mmap uses memory sharing as an indirect way of communicating. The way that this works is that all processes map the same file into a fixed memory location. In this case, however, since we have objects being read from and written to a shared memory region, we must use some kind of process-level synchronization. Unfortunately, these synchronization procedures are in general much more expensive than the use of threads itself.

One can also use *signals*; in this case, processes can send and receive integer numbers associated with particular signal numbers. In order to be able to receive and treat these numbers, processes must first set up signal handlers. This approach works best rare events[1]. In general, signals are not very useful for parallel or concurrent programming.

The other possibility is the use of *pipes*. These are unidirectional communication channels which make it possible for the output of one program to be used as input to another one, just like in Unix (for example, when we use the pipe symbol, as in "ls | wc -l". The advantage of using pipes is that they are easy and fast.

4.3 Threads

First, remember that different processes keep their own data in distinct address spaces. Threads, on the other hand, explicitly share their entire address space with one another. Although this can make things a lot faster, it comes with the cost of making programming **a lot** more complicated.

In Unix/POSIX, the threads API is composed by two main calls:

- pthread_create(), which starts a separate thread;
- pthread_join(), which waits for a thread to complete.

Notice that the general syntax for using these is:

```
pid = pthread_create(&tid, NULL, function_ptr, argument);
pthread_join(tid, &result);
```

Example:

```
void *run(void *d)
{
    int q = ((int) d);
    int v = 0;
    for (int i=0; i<q; i++)
        v = v + some_expensive_function_call();
    return (void *) v;
}

int main()
```

[1] Example: processes treating a SIGSEGV.

```
{
   pthread_t t1, t2;
   int *r1, *r2;
   int arg1=100;
   int arg2=666;
   pthread_create(&t1, NULL, run, &arg1);
   pthread_create(&t2, NULL, run, &arg2);

   pthread_join(t1, (void **) &r1);
   pthread_join(t2, (void **) &r2);

   cout << r1= << *r1 << r2= << *r2 << endl;
}
```

Notice that the above threads maintain *different* stacks and different sets of registers; except for those, however, they share *all* their address spaces. Also notice that if you were to run this code in a 2 core machine, it would be expected that it ran roughly twice as fast as it would in a single core machine. If you ran it in a 4 core machine, however, it would run as fast as in the 2 core machine, since there would be no sufficient threads to exploit the available parallelism.

4.4 Processes vs. threads

One might argue that in general processes are more flexible than threads. For one thing, they can live in two different machines, and communicate via sockets; they are easy to spawn remotely (e.g. ssh foo.cs.umass.edu "ls -l"); etc. However, using processes requires explicit communication and risks hackery. Threads also have their own problems: because they communicate through shared memory, they must run on the same machine and they require thread-safe code. So even though threads are faster, they are much harder to program. In a sense, we can say that processes are far more robust than threads, since they are completely isolated from other another. Threads, on the other hand, are not that safe, since whenever one thread crashes the whole process terminates.

When comparing processes and threads, we can also analyze the cost of context switches. Whenever we need to switch between two processes, we must invalidate the TLB cache (the so called *TLB shootdown*; see later lectures on virtual memory). This of course makes everything slower. When we switch between two threads, on the other hand, it is not necessary to invalidate the TLB, because all threads share the same address space, and thus have the same contents in the cache. In other words, on many operating systems, the cost of switching between threads is **much** smaller than the cost of switching between processes.

Most large-scale systems use a mixture of processes and threads: threads within a process on one server, communicating via a network socket to similar processes on other servers.

Chapter 5

Synchronization

As we already know, threads must ensure consistency; otherwise, race conditions (non-deterministic results) might happen.

Now consider the "too much milk problem": two people share the same fridge and must guarantee that there's always milk, but not too much milk. How can we solve it? First, we consider some important concepts and their definitions:

- **Mutex**: prevents things from operating on the same data at the same time;
- **Critical section**: a piece of code that only one thread can execute at a time;
- **Lock**: a mechanism for mutual exclusion; the program locks on entering a critical section, accesses the shared data, and then unlocks. Also, a program waits if it tries to enter a locked section.
- **Invariant**: something that must always be true when not holding the lock.

For the above mentioned problem, we want to ensure some correctness properties. First, we want to guarantee that only one person buys milk when it is need (this is the *safety* property, aka "nothing bad happens"). Also, we want to ensure that someone *does* buy milk when needed (the *progress* property, aka "something good eventually happens"). Now consider that we can use the following atomic operations when writing the code for the problem:

- "leave a note" (equivalent to a lock)
- "remove a note" (equivalent to a unlock)
- "don't buy milk if there's a note" (equivalent to a wait)

An *atomic* operation is an unbreakable operation. Once it has started, no other thread or process can interrupt it until it has finished.

Our first try could be to use the following code on both threads:

```
if (no milk and no note)
   leave note
   buy milk
   remove note
```

Unfortunately, this doesn't work because both threads could simultaneously verify that there's no note and no milk, and then both would simultaneously leave a note, and buy more milk. The problem in this case is that we end up with too much milk (safety property not met).

Now consider our solution #2:

Thread A:

```
leave note "A"
if (no note "B")
   if (no milk)
      buy milk
remove note "A"
```

Thread B:

```
leave note "B"
if (no note "A")
   if (no milk)
      buy milk
remove note "B"
```

The problem now is that if both threads leave notes at the same time, neither will ever do anything. Then, we end up with no milk at all, which means that the progress property not met.

Let us now consider an approach that *does* work:

Thread A

```
leave note A
while (note B)
   do nothing
if (no milk)
   buy milk
remove note A
```

Thread B

```
leave note B
if (no note A)
   if (no milk)
      buy milk
remove note B
```

This approach, unlike the two examples considered on the previous class, does work.

However, it is *complicated*: it is not quick-and-easy to convince yourself that these two sections of code always produce the desired behavior.

Furthermore, it is *wasteful*: Thread A goes into a loop waiting for B to release its note. This is called "busy waiting", or "spinning", and wastes CPU time and energy, when the CPU could be doing something useful.

Also, it is *asymmetric*: notice that even though both threads try to achieve the same goal, they do it in very different ways. This is a problem specially when we were to write, say, a third thread. This third thread would probably look very different than both A and B, and this type of asymmetric code does not scale very well.

Finally, this code is potentially non-portable; it doesn't use a standard library.

So the question is: how can we guarantee correctness and at the same time avoid all these drawbacks? The answer is that we can augment the programming language with high-level constructs capable of solving synchronization problems. Currently, the best known constructs used in order to deal with concurrency problems are *locks, semaphores, monitors*.

5.1 Locks/Mutexes

Locks (also known as mutexes, short for mutual exclusion locks) provide mutual exclusion to shared data inside a critical session. They are implemented by means of two atomic routines: *acquire*, which waits for a lock, and takes it when possible; and *release*, which unlocks the lock and wakes up the waiting threads. The rules for using locks/mutexes are the following:

1. only one person can have the lock;
2. locks must be acquired before accessing shared data;
3. locks must release after the critical section is done;
4. locks are initially released.

The syntax for using locks in C/C++ is the following:

```
   pthread_mutex_init(&l);
   ...
   pthread_mutex_lock(&l);
         update data    // this is the critical section
   pthread_mutex_unlock(&l)
```

Let us now try to rewrite the "Too Much Milk" problem in a simpler, cleaner, more symmetric, and portable way, using locks. In order to do so, the code for Thread A (and also for Thread B) would be the following:

```
pthread_mutex_lock(&l)
if (no milk)
   buy milk
pthread_mutex_unlock(&l)
```

This is clearly much easier to understand than the previous solutions; also, it is more scalable, since all threads are implemented in the exact same way.

5.2 Implementation of Locks/Mutexes

How could we implement locks? No matter how we choose to implement them, we *must* have some hardware support. One possibility for implementing locks on a uniprocessor machine is is to disable interrupts when testing/setting locks. With interrupts disabled on a single processor machine, the processor cannot switch processes, and so we can guarantee that only the active process will have access to the shared data. Another option would be to make use of atomic operations, such as `test_and_set`. This type of operation (which usually corresponds to a single atomic assembly instruction) behaves as if it used the following C function, atomically:

```
int test_and_set(int x)   // let x be strictly either 0 or 1.
{
  if (x) { return 1; } else { x=1; return 0; }
}
```

All this needs to be implemented atomically, in hardware. Using this type of atomic operation, one could implement `thread_lock(l)` simply as

```
while test_and_set(l) { do nothing; }  // spinlock version of thread_lock()
```

and `thread_unlock(l)` simply as

```
l = 0;    // we need this to be an atomic clear (or assign) instruction
```

The assembly instruction `test_and_set` can be made to be atomic across multiple processors. An equivalent option would be an atomic `compare_and_swap` assembly instruction.

These low-level hardware solutions are then built up into high-level functions, either built into the languages, or in libraries. In general, do not implement your own locking functions, but rather use functions from a tested library. Getting things right can be tricky, and your own solution is also likely to be non-portable.

Summary of this section:

- Communication between threads is done implicitly, via shared variables;
- Critical sections are regions of code that access shared variables;
- Critical sections must be protected by synchronization methods;
  - We need primitives that ensure *mutual exclusion*;
  - Writing "personalized" solutions to concurrency is tricky and error-prone;
  - The solution is to introduce general high-level constructs into the language, such as `pthread_mutex_lock()` and `pthread_mutex_unlock()`.

5.3 Advanced Synchronization, Part 1

Synchronization serves two purposes: 1) to ensure **safety** for updates on shared data (e.g. to avoid races conditions), and 2) to **coordinate and order** actions taken by threads (e.g. handling threads which communicate intermediate results amongst one another).

One of the most important aspects of parallel programs is that *all* their possible interleavings must be correct. One possible way to guarantee this is to simply put one lock in the beginning of each thread; however, it is also clear that we want to use as as few constraints as possible, in order to effectively exploit the available concurrency. Thus, the correct placement of locks is not always trivial.

In general, locks provide safety and correctness, while condition variable provide ordering.

5.3.1 Example

Consider the following example, of a multi-threaded program that spawns N threads; each thread performs some expensive computation, and *safely* adds its results to a global variable.

```
#include <pthread.h>
#include <stdio.h>

const int N = 8; // number of threads
pthread_t threads[N];
pthread_mutex_t myLock;
int total = 0;

void * expensiveComputation (void *x) {
  int v = *((int *) x);
  delete ((int *) x);
  int res = computeNthDigitOfPi (v);
  pthread_mutex_lock (&myLock);
  total += res;
  pthread_mutex_unlock (&myLock);
  return NULL;
}

int main()
{
  pthread_mutex_init (&myLock, NULL);

  for (int i=0; i<N; i++) {
    int * ptr_i = new int (i);
    pthread_create (&threads[i], NULL, expensiveComputation, (void *)ptr_i);
  }
  for (int i=0; i<N; i++) {
    pthread_join (threads[i], NULL);
  }
  printf("total: %d\n", total);
}
```

5.4 Thread safety

What does it mean for something to be thread-safe? By saying that X is thread-safe we mean that if multiple threads use X at the same time, we don't have to worry about concurrency problems. The STL, for instance, is **not** thread-safe; if we were to create an STL queue and have two threads to operate on it simultaneously, we would have to manually perform all locking operations. The *cout* instruction is also not thread-safe.

Suppose now we want to build a thread-safe queue; the methods we want the queue to have are *insert(item)*, *remove()* and *empty()*. The first question to be asked is what should *remove()* do when the queue is empty. One solution would be for it to return a special value (such as NULL or -1, etc) or to throw an exception. It would much more elegant and useful, however, to make that function call wait until something actually appears in the queue. By implementing this type of blocking system, we are in fact implementing part of a *producer-consumer* system.

Now let us think of how to make the function wait. We can spin, i.e. write something like *while (empty())* ;. This, however, obviously doesn't work, since the test of emptiness needs to *read shared data*; we need to put locks somewhere! And if we lock around the *while(empty());* line, the program will hang forever. Several other naïve approaches do not work (see slides for examples). The conclusion is that we need some way of going to sleep and at the same time having someone to wake us up when there's something interesting to do. Let us now present several possible implementations for this system and discuss why they do *not* work. The first possibility is:

```
dequeue()
   lock()     // needs to lock before checking if it's empty
   if (queue empty)
       sleep()
   remove_item()
   unlock()

enqueue()
   lock()
   insert_item()
   if (thread waiting)
       wake up dequeuer
   unlock()
```

One problem with this approach is that the dequeuer goes to sleep while holding the lock! How about the second possible approach:

```
dequeue()
   lock()     // need to lock before checking if it's empty
   if (queue empty) {
       unlock()
       sleep()
   }
   remove_item()
   unlock()


enqueue()
   lock()
   insert_item()
   if (thread waiting)
       wake up dequeuer
   unlock()
```

One problem here is that we are using an "if" instead of a "while" when checking whether or not the queue is empty. Consider the case where the dequeuer is sleeping, waiting for the enqueuer to insert an item. The enqueuer inserts an item, sends a wake-up signal, and the dequeuer wakes up. But then another dequeuer thread runs and removes the item. The first dequeuer thread now tries to remove an item from an empty queue. This problem can be fixed by using a "while" instead of the "if" statement, like this:

```
dequeue()
   lock()     // need to lock before checking if it's empty
   while (queue empty) {
       unlock()
       sleep()
   }
   remove_item()
   unlock()


enqueue()
   lock()
   insert_item()
   if (thread waiting)
       wake up dequeuer
   unlock()
```

This presents a more subtle and harder problem: the dequeuer might unlock, and then before it actually executes the sleep() function, the enqueuer could get the CPU back. In this case, the enqueuer would acquire the lock and insert the new item, but since there would be no one sleeping, there would be no thread to wake up, so the enqueuer would simply unlock, and never wake up the dequeuer again. Then the dequeuer would get the CPU back, finish calling sleep(), and sleep forever, even though the queue is not empty. The main issue is that another thread can run between the unlock() and the sleep() calls in dequeue().

The general solution to this type of problem is to make "unlock + sleep" atomic, which we will use to build our function called wait():

```
wait(lock l, cv c) {
    unlock_and_sleep(l,c);  // in one atomic step, unlock and sleep, waiting for
    lock(l);                // re-acquire the lock
}
```

We use *condition variables* to do exactly that: condition variables make it possible and easy to go to sleep, by atomically releasing the lock, putting the thread on the waiting queue and going to sleep. Each condition variable has a "wait queue" of threads waiting on it, managed by the threads library. There are three important functions to be used when dealing with condition variables:

- *wait(lock l, cv c)*: atomically releases the lock and goes to sleep. When calling wait, we must be holding the lock. Depending upon the threads library, wait() may or may not re-acquire the lock when awakened (pthread_cond_wait() does re-acquire the lock);
- *signal(cv c)*: wakes up one waiting thread, if there are any;
- *broadcast(cv c)*: wakes up all waiting threads.

5.4.1 Producer-consumer using condition variables

Now let us present an implementation of a producer-consumer system using condition variables. This implementation works.

```
dequeue()
   lock(A)
   while (queue empty) {
          wait(A, C)    // atomically releases lock A and sleeps, waiting for
                        // condition variable C.
                        // When the thread wakes up, it re-acquires the lock.
                        // C is the condition variable that means ``queue not emp
   }
   remove_item()
   unlock(A)
```

```
enqueue()
   lock(A)
   insert_item()
   signal(C)
   unlock(A)
```

In dequeue() above, if the thread wakes up and by chance the queue is empty, there is no problem: that's why we need the "while" loop.

5.5 Advanced Synchronization

Recall that condition variables are synchronization primitives that enable threads to wait until a particular condition occurs.

Generalizing, the combination of locks and condition variables is sometimes called a *monitor*, which is sometimes incorporated into data structures in some languages (note that this terminology is not always used in a standardized way).

In the last class we also discussed how to use condition variables and signals to implement a simple producer-consumer system. Looking at this system again, could we move the signal() call in enqueue() down below the unlock()?

```
dequeue()
   lock(A)
   while (queue empty)
         wait(A, C)
   remove_item()
   unlock(A)


enqueue()
   lock(A)
   insert_item()
                    // used to signal here, inside lock
   unlock(A)
   signal(C)        // now signal here, outside lock
```

Will this work? The answer is yes. It might not be as clean conceptually, but nothing bad will happen (verify this for yourself).

As an additional note, signals are memoryless: If no threads are waiting for them, they don't do anything.

Let us now discuss how to implement a more realistic producer-consumer, in the sense that it only has a *bounded (finite) buffer*. Assume we are dealing with a Coke machine: no one can buy a Coke while someone else is using the machine; the delivery person needs to wait until there is free space before putting more Coke(s) into the machine; and the buyer needs to wait until there is at least one Coke available. For this system, we are going to use two condition variables; the code is as follows:

```
cokebuyer()
   lock(A)
   while (cokes==0)  // machine is empty
      wait(A, coke_in_machine)
   buy_coke()        // this works: when we get out of wait while loop, we are sure th
   signal(coke_bought)
   unlock(A)


deliveryperson()
   lock(A)
   while (cokes == maxcokes)  // machine is full
       wait(A, coke_bought)
   put_coke_in_machine()      // and now we are sure there is a spot to put a new Coke
   signal(coke_in_machine)
   unlock(A)
```

5.6 A Few Networking Tidbits

There is a very small amount of networking background required for Project 2.

In Project 2, we want our computer (or our "host") to request information over the Internet from a remote webserver (another Internet "host"). On our computer, from within our Project 2 application, we will open a "socket", which on many operating systems acts like a file which you can read or write. Moving down the network stack into the operating system, a socket uses a transport protocol to communicate with the remote host. The application-layer HTTP protocol which webservers use is built on top of the transport-layer TCP protocol, which provides reliable, in-order, stream-based delivery between your socket and the remote socket. A socket also has a "port" number associated with it. When we open our socket, we will request that it opens port 80 on the remote host, since this is the port number on which almost all webservers listen for requests. Finally, when opening the socket, we need to specify which remote webserver we want to communicate with, such as www.cnn.com, etc. Moving down one more layer, every host (laptop, server, cell phone, etc) which is directly accessible on the Internet has a unique network-layer IP address, such as 128.119.240.19, and most of those have hostnames, such as www.cs.umass.edu; the Domain Name Service (DNS) translates from hostnames to IP addresses.

That was a quick summary of what our web-spider application needs to do in terms of networking: open a socket to port 80 on a given remote host, using TCP. The `clientsocket` class defined in `simplesocket.h` provides methods to do this (see also the Project 2 assignment):

```
string hostname,file;       // C++ strings
int ret,port=80,timeout=0;  // timeout=0 means: do not timeout
char buf[MAX_READ];         // used for both reading from and writing to socket
```

```
clientsocket sock(hostname.c_str(), port, timeout, false /* false: don't print d
if (sock.connect()) {
        // initialize a buffer, buf, to send a HTTP 1.0 request to remote webser
        sprintf(buf, "GET /%s HTTP/1.0\r\nHost: %s\r\n\r\n", file.c_str(),hostna
        sock.write(buf, strlen(buf));

        // read reply from remote webserver
        int size = 0;
        sock.setTimeout(5);
        while ((ret = sock.read(buf+size, MAX_READ-1-size)) > 0)
                size += ret;
} else {
        // error with sock.connect()
}
sock.close();
// do whatever needs to be done with received buffer, buf
```

Note that the `sock.read()` method is in a while-loop, since reading from a socket will return between one byte and the maximum number of bytes requested (`(MAX_READ-1-size)` in the code above). You'll need to keep repeatedly reading from the socket again until you get all the bytes you want. `sock.read()` will block until it has at least one byte to return, unless its request times out (set by `sock.setTimeout(5)` to be 5 seconds), or unless the connection is closed.

As an aside, this final point makes HTTP 1.0 a bit easier to use than HTTP 1.1: HTTP 1.0 uses so-called "non-persistent" connections, where the webserver closes the connection after its sends the webpage contents. For the purposes of this simple web-spider project, we will use HTTP 1.0 so that our code is a little simpler, since when `sock.read()` finishes fetching a particular page, the remote webserver will close the connection, and then the next `sock.read()` will return 0, breaking us out of the while-loop.

5.7 Semaphores

Synchronization can be achieved by means other than locks and condition variables. One alternative is *semaphores*. A semaphore is used to regulate traffic in a critical section. A semaphore is implemented as a non-negative integer counter with atomic increment and decrement operations, up() and down(), respectively. Whenever the decrement operation would make the counter negative, the semaphore blocks instead.

- `down(sem)`: tries to decrease the counter by one; if the counter is zero, blocks until the counter is greater than zero. down() is analogous to wait().

- `up(sem)`: increments the counter; wakes up one waiting process. up() is analogous to signal().

Semaphores in computer science were introduced by the Dutch computer scientist Edsger Dijkstra, and so down() is sometimes called P(), from the Dutch for *proberen*, roughly "try [to decrement]", and up() is sometimes called V(), from the Dutch for *verhogen*, roughly "increment".

Notice that we can use semaphores to impose both mutual exclusion and ordering constraints. In some sense semaphores, with two operations, are more elegant than monitors, with five operations (namely, lock(), unlock(), wait(), signal(), and broadcast()). Some people, however, find semaphores less intuitive than monitors. There is also an important difference between semaphores and signals: signals have no memory, whereas semaphores do have memory.

5.7.1 Semaphore Example: Ordering

For example, by initializing a semaphore to 0, threads can wait for an event to occur, and impose an ordering constraint, similar to a monitor using condition variables, but with memory:

```
semaphore sem=0;    // 0==locked for a semaphore

thread A
   down(sem)        // wait for thread B; note sem has memory!
   // do stuff

thread B
   // do stuff, then wake up A
   up(sem)
```

5.7.2 Semaphore Example: Producer-Consumer

Now let's implement bounded producer-consumer with semaphores, a Coke machine which is initially empty:

```
semaphore sem_mutex = 1  // binary semaphore for mutual exclusion, 1==unlocked
semaphore slots_left = N
semaphore cokes_left = 0

producer()
    down(slots_left)
    down(sem_mutex)
    insert Coke into machine
    up(cokes_left)
    up(sem_mutex)

consumer()
    down(cokes_left)
```

```
down(sem_mutex)
buy Coke
up(slots_left)
up(sem_mutex)
```

Note that the `down(slots_left)` needs to go before the `down(sem_mutex)`, and similarly for `down(cokes_left)` and `down(sem_mutex)`, because otherwise the system can end up waiting forever. For example, imagine an empty machine with the consumer arriving first, and calling `down(sem_mutex)` before `down(cokes_left)`.

In some sense, the consumer calling `down(cokes_left)` is a *reservation* to get a Coke in the future. Even if there are no Cokes currently in the machine, the consumer is reserving a claim for a future Coke with this action. Similarly, the producer calling `down(slots_left)` is a reservation to insert a new Coke in the future.

The invariant for this producer-consumer system is that `cokes_left` and `slots_left` are accurate counts of the number of unclaimed Cokes and unclaimed slots left in the machine. The invariant is only true when the `sem_mutex` lock is not held.

5.8 Spin Locks versus Blocking Locks

We've talked about spinlocks previously, for example,

```
void spinlock(lock &l) {
     while(test_and_set(l) == 1)
          ;
}
```

For most situations, this is extremely crude, and wastes CPU time and electrical energy which could instead be used for something useful.

A blocking lock, in contrast, returns to the scheduler to allow another thread to be scheduled when it cannot immediately acquire a lock:

```
void blocking_lock(lock &l) {
     while(test_and_set(l) == 1)
         sched_yield();           // yield to other threads/processes
}
```

A downside of a blocking lock is that it will always cause a context switch, which can be fairly expensive.

If we are pretty sure that the lock will be released quite quickly, it may be less expensive to use a spinlock than a blocking lock. In most situations, however, it is preferable to use a blocking lock, and let other threads or processes use the CPU time and energy which would otherwise be wasted spinning.

5.9 Blocking Locks and Multiple CPUs

We have talked about blocking locks, which voluntarily yield, and spinlocks, which just spin until they acquire the lock. At first, it seems like spinlocks are very wasteful, and that one should always use blocking locks. But this question is a bit more complicated than that.

In well-written code, locks are only held around *small* pieces of code, so they should not be held for very long. On a multi-processor system, it often makes sense to use spinlocks, since the thread holding the lock will likely release the lock soon, and may be running simultaneously on another processor, about to release the lock.

This question is further complicated, however, by the possibility of pagefaults (see virtual memory, later in this course). Pagefaults or similar hard-to-predict delays could make even well-written multi-threaded code hold locks for longer than anticipated.

A hybrid type of lock, called *spin-then-yield* locks, will spin for some amount of time, and if it still hasn't acquired the lock, yield. A spin-then-yield lock can be adaptive in the amount of time it spins before yielding.

5.10 Major locking errors

When programming with threads, there are three very common mistakes that programmers often make:

1. locking twice (depending on the system and type of lock, can cause crashes, hangs, or do bizarre things);
2. locking and not unlocking (i.e. failure to unlock);
3. deadlock (see next lecture).
4. Priority inversion - This is not an error per se, but an important issue that occurs

Of these problems, locking twice is probably the easiest type of error to detect. Here's one example:

```
function f() {
    lock(L);
    g();
    unlock(L);
}
```

```
function g() {
    lock(L);
    // access shared data
    unlock(L);
}
```

So-called "recursive" locks can deal with this situation correctly, though normal locks will cause this thread to wait forever when the function `g()`, when called from `f()`, then calls `lock(L)` on a previously-held lock. Dealing with this can lead to a common code pattern with certain functions designed only to be called with locks held:

```
function f() {
    lock(L);
    g_internal();
    unlock(L);
}
```

```
function g() {
    lock(L);
    g_internal();
    unlock(L);
}
```

```
function g_internal() {
    // locks must be hel
    // access shared dat
}
```

Failure to unlock is slightly more difficult to detect. It can occur, for example, if the programmer forgets to release the lock in one of the possible execution branches of the function:

```
function f() {
    lock();
    if (x==0) {
        // should also unlock here before returning!
        return;
    }
    // do something
    unlock();
    return;
}
```

One way to deal with this is just to remember to `unlock()` at each possible return. Another is to have every return path go through the same section of code (and in C, `goto` is sometimes useful for this, despite its bad reputation).

Priority inversion is when low priority tasks get scheduled more often or in preference to higher priority tasks. This would not be erroneous, but certainly has implications on system performance and thus an important issue. Low priority taks holding locks can prevent higher priority tasks from being scheduled because the higher priority tasks may be waiting on the same locks.

5.11 Increasing Concurrency: Read-Write Locks

Consider a large web-based database. In some sense, Google is sort of like this. There might be many users who want to read from the database, but only a few users who are allowed to write to the database. If we use standard locks to control access to the database, the application will be much slower than it could be with a more clever type of lock.

Suppose we have one object that is shared among several threads. Suppose also that each thread is either a reader or a writer. Readers only read data but never modify it, while writers read and modify data. If we know which threads are reading and which ones are writing, what can we do to increase concurrency?

First, we have to prevent two writers from writing at the same time. In addition, a reader cannot read while a writer is writing. There is no problem, however in allowing lots of readers to read at the same time. Read-write locks achieve this, and can greatly improve performance for this sort of application.

Note that the lock is the same for both readers and writers (called 'rw' in the slides), but the readers use an rlock() on the lock and writers use a wlock() on the same lock. Writers requesting a wlock() will have to wait until all readers and writers have released the lock. Readers requesting a rlock() can acquire it even if other readers are holding the lock. Readers however will still have to wait until any writer holding the lock releases the lock.

Programmers must be careful not to alter the data when holding the rlock. To make any changes to data (to write to it), wlock must be acquired.

Scheduling readers and writers who request lock at the same time-

1. Always favor readers - This improves concurrency. However, if there is a steady stream of readers, writers never get to write anything (called *writer starvation*)

2. Always favor writers - This reduces concurrency

3. Alternate between readers and writers, or similar versions of this policy - Usually fair, works well, avoids starvation while maintaining good amount of concurrency.

5.12 Deadlocks

At the end of this lecture, and into the next lecture, we will discuss the last major type of logical error that can occur when programming with threads. A *deadlock* occurs when two things (threads, processes, etc) wait on each other.

One example of a deadlock is known as the *Dining Philosophers* problem. In this abstract problem, philosophers alternate between thinking and eating, and the dining table has as many forks as philosophers. Each philosopher needs two forks to eat with. The problem which can occur is if each philosopher gets one fork, and will not let go of it. Then no philosopher can get two forks, which he or she needs in order to eat. In this situation, we have a *deadlock*, and the philosophers will **starve**!

5.13 Deadlocks

Here is another example of a program which can deadlock:

```
Thread A
    lock(L1);
    lock(L2);

Thread B
    lock(L2);
    lock(L1);
```

If Thread A acquires lock L1, and then the scheduler switches to Thread B, which acquires lock L2, then this program will deadlock. Neither thread releases the lock it holds, and each is trying to acquire the lock which the other holds. In fact, without recursive locks, this simple example will also deadlock:

```
Thread A
    lock(L);
    lock(L);
```

5.13.1 Necessary Conditions for Deadlock

Here are the conditions necessary for a deadlock to occur; note that *all* of them are necessary, and none is sufficient:

1. *finite resources*: the resources are held in a mutually-exclusive way, and they can be exhausted, causing waiting.

2. *hold-and-wait*: each thread holds one resource while waiting for another.

3. *no preemption*: threads only release resources voluntarily. No other thread (or the OS) can force the thread to release its resources.

4. *circular wait*: we have a circular chain of waiting threads.

5.13.2 Deadlock Detection

Deadlocks can be detected while the program is running, by running cycle detection algorithms on the graph that defines the current use of resources.

Define this graph as follows: it has one vertex for each resource ($r_1, \ldots, r_m$) and one vertex for each thread ($t_1, \ldots, t_n$). If a resource r_i is held by thread t_j, then we add an edge from vertex r_i to vertex t_j. If a thread t_k is trying to acquire resource r_ℓ, then we add an edge from vertex t_k to vertex r_ℓ.

Given this graph, we can run a cycle detection algorithm. If a cycle is found, there is a deadlock.

Detecting a deadlock is much easier than recovering from a deadlock. Several possible approaches might be to kill all of the threads in the cycle, or kill the threads one at a time, forcing them to release resources, and hope that this will break the deadlock.

While this may sound easy, it is not. Killing threads generally doesn't release their resources cleanly (locks, memory, files, etc). This is essentially the *rollback* problem, to back out all the actions of a thread. Databases usually include rollback mechanisms, which can be quite complicated, and it is not always possible to roll back all the actions of a thread (consider a thread which outputs hard-copy printed pages).

As a general guideline, do not use functions like `pthread_cancel()`, which kills a thread, unless you *really* know what you are doing.

5.13.3 Deadlock Prevention

While it is hard to resolve a deadlock which has been detected, fortunately it is fairly easy to prevent deadlocks from ever happening.

The key is that the conditions above for deadlock are *all* necessary. We just have to ensure that at least one condition cannot be true. We definitely cannot get rid of the problem of finite resources; that is something we have to live with. And in many cases it can be difficult to get rid of the conditions of hold-and-wait and lack of preemption.

But we can eliminate circular waiting. One standard way of doing this is to have a so-called *canonical ordering* of the locks. This means that the programmer must always acquire the locks in a specified order, such as first lock1, then lock2, then lock3, etc. By doing so, we will ensure a system whose thread/resource graph is cycle-free, and therefore the system will be deadlock-free.

Chapter 6

Virtual Memory

In modern operating systems, applications *do not* directly access the physical memory. Instead, they use so-called *virtual memory*, where each virtual address is translated to a physical address.

Why would one do this? Here are some of the reasons that virtual memory is useful:

- To isolate and protect processes from each other,
- To manage the limited physical memory efficiently,
- To give each process the illusion of having the whole address space for itself.

Since each program thinks it has the whole memory to itself, programs can use *a lot* of virtual memory. In fact, a computer might use huge amounts of virtual memory, much more than the amount of actual physical memory available. Most of the time this works fairly well, since processes typically only need to use a small amount of their virtual memory at once, and often a lot can be shared among processes (such as with shared libraries). But if the system gets into a situation where the active virtual memory demands exceed the available physical memory, we will need to use another mechanism, such as "swapping" memory pages to disk.

Virtual memory uses *one level of indirection* in the translation from virtual addresses to physical addresses.

6.1 Memory Pages

Applications allocate memory in terms of the number of bytes that they need, but this level of granularity is too fine-grained for the operating system to manage the system memory in this way. Instead, the OS manages the memory in *groups of bytes* called pages. Pages are typically of 4kb or 8kb. Suppose we decide to use 4kb pages; in a 32-bit machine, it is possible to address at most 2^{32} different addresses; if we divide that maximum amount of memory by the size of the page, we see that there can be at most $\frac{2^{32}}{2^{12}} = 2^{20}$ 4kb pages.

Using pages makes it easier to manage the whole memory, avoiding excessive fragmentation and waste. As an analogy, think of Tetris, but with only square blocks: it is relatively easy to avoid fragmentation in this case. Both the virtual and the physical memory are divided into pages. As an aside, we often refer to physical memory pages as *frames*.

Every virtual page consists of a range of virtual addresses. Virtual pages are then mapped to physical pages in the physical memory. When an application needs to access a virtual page, the corresponding virtual address is translated into a "real" physical address, and then the actual data might be read and/or written. As a consequence, note that although arrays are contiguous in virtual memory, they may be non-contiguous in physical memory. Note also that some virtual pages do not map to any actual physical memory address, because not all virtual pages are necessarily being used at any given time. The unused virtual pages are called *invalid*. If a program ever tries to access an invalid page, it pagefaults.

A typical virtual memory layout includes a stack, a mmap region, a heap, and the code of the program (the "text" section). Since mmap and heap grow towards each other, in the worst case they could smash into each other. The same could happen with the stack: if a program recurses for a large number of times, the stack could grow over the mmap region ("stack overflow") and in this case something bad and usually unpredictable will happen.

6.1.1 The Memory Management Unit

When a program issues a memory load or store operation, the virtual addresses (VAs) used in those operations have to be translated into "real" physical memory addresses (PAs). This translation is the main task of the MMU (Memory Management Unit). The MMU maintains a *page table* (a big hash table) that maps virtual pages to physical pages. Since memory accesses are happening all the time, the MMU needs to be extremely fast and implemented correctly. For these reasons, MMUs are almost always implemented in hardware. Note that we can't map every single byte of virtual memory to a physical address; that would require a huge page table. Instead, the MMU maps virtual pages to physical pages. Also, since we want to isolate each program's address space from the address spaces of other applications, the MMU must keep a separate page table for each process; this implies that different processes could use the same virtual address to refer to different data, and that would not be a problem since these virtual addresses would be mapped into different physical addresses. The page table also marks all virtual pages that are allocated (and therefore are being used), so that it is possible to know which virtual pages have valid mappings into physical pages. All virtual pages that are not being mapped to a physical page are marked as *invalid*; segfaults occur when a program tries to reference or access a virtual address that is not valid. In addition to valid bits, page table entries (PTEs) also store a lot of other information, such as "read" and "write" bits to indicate which pages can be read/written.

6.1.2 Virtual addresses

Virtual addresses are made up of two parts: the first part is the page number, and the second part is an offset inside that page. Suppose our pages are 4kb ($4096 = 2^{12}$ bytes) long, and that our machine uses 32-bit addresses. Then we can have at most 2^{32} addressable bytes of memory; therefore, we could fit at most $\frac{2^{32}}{2^{12}} = 2^{20}$ pages. This means that we need 20 bits to address any page. So, the page number in the virtual address is stored in 20 bits, and the offset is stored in the remaining 12 bits.

Now suppose that we have one such page table per process. A page table with 2^{20} entries, each entry with, say, 4 bytes, would require 4Mb of memory! This is somehow disturbing because a machine with 80 processes would need more than 300 megabytes just for storing page tables! The solution to this dilemma is to use *multi-level page tables*. This approach allows page tables to point to other page tables, and so on. Consider a 1-level system. In this case, each virtual address can be divided into an offset (10 bits), a level-1 page table entry (12 bits), and a level-0 page table entry (10 bits). Then if we read the 10 most significant bits of a virtual address, we obtain an entry index in the level-0 page; if we follow the pointer given by that entry, we get a pointer to a level-1 page table. The entry to be accessed in this page table is given by the next 12 bits of the virtual address. We can again follow the pointer specified on that level-1 page table entry, and finally arrive at a physical page. The last 10 bits of the VA address will give us the offset within that PA page.

A drawback of using this hierarchical approach is that for every load or store instruction we have to perform several indirections, which of course makes everything slower. One way to minimize this problem is to use something called Translation Lookaside Buffer (TLB); the TLB is a fast, fully associative memory that caches page table entries. Typically, TLBs can cache from 8 to 2048 page table entries.

Finally, notice that if the total virtual memory in use (i.e. the sum of the virtual memory used by all processes) is larger than the physical memory, we could start using RAM as a cache for disk. In this case, disk could used to store memory pages that are not being used or that had to be removed from RAM to free space for some other needed page, which itself had been moved to the disk sometime in the past. This approach obviously requires locality, in the sense that the whole set of working pages must fit in RAM. If it does not, then we will incur in a lot of disk accesses; in the worst case, these accesses could cause thrashing, i.e. the system doing nothing except a lot of disk reads and writes.

6.2 Page Tables

As mentioned above, *page tables*, are lookup tables mapping a process' virtual pages to physical pages in RAM. How would one implement these page tables?

6.2.1 Single-Level Page Tables

The most straightforward approach would simply have a single linear array of page-table entries (PTEs). Each PTE contains information about the page, such as its physical page number ("frame" number) as well as status bits, such as whether or not the page is valid, and other bits to be discussed later.

If we have a 32-bit architecture with 4k pages, then we have 2^{20} pages, as discussed in the last lecture. If each PTE is 4 bytes, then each page table requires 4 Mbytes of memory. And remember that each process needs its own page table, and there may be on the order of 100 processes running on a typical personal computer. This would require on the order of 400 Mbytes of RAM just to hold the page tables on a typical desktop!

Furthermore, many programs have a very sparse virtual address space. The vast majority of their PTEs would simply be marked invalid.

Clearly, we need a better solution than single-level page tables.

6.2.2 Multi-Level Page Tables

Multi-level page tables are tree-like structures to hold page tables. As an example, consider a two-level page table, again on a 32-bit architecture with $2^{12} = 4$ kbyte pages. Now, we can divide the virtual address into three parts: say 10 bits for the level-0 index, 10 bits for the level-1 index, and again 12 bits for the offset within a page.

The entries of the level-0 page table are pointers to a level-1 page table, and the entries of the level-1 page table are PTEs as described above in the single-level page table section. Note that on a 32-bit architecture, pointers are 4 bytes (32 bits), and PTEs are typically 4 bytes.

So, if we have one valid page in our process, now our two-level page table only consumes

$$(2^{10}\text{ level-0 entries})\cdot(2^{2}\text{ bytes/entry})+1\cdot(2^{10}\text{ level-1 entries})\cdot(2^{2}\text{ bytes/entry}) = 2\cdot 2^{12}\text{ bytes} = 8\text{ kbytes}.$$

For processes with sparse virtual memory maps, this is clearly a huge savings, made possible by the additional layer of indirection.

Note that for a process which uses its full memory map, that this two-level page table would use slightly more memory than the single-level page table (4k+4M versus 4M). The worst-case memory usage, in terms of efficiency, is when all 2^{10} level-1 page tables are required, but each one only has a single valid entry.

In practice, most page tables are 3-level or 4-level tables. The size of the indices for the different levels are optimized empirically by the hardware designers, then these sizes are permanently set in hardware for a given architecture.

6.2.3 Page-Table Lookups

How exactly is a page table used to look up an address?

The CPU has a page table base register (PTBR) which points to the base (entry 0) of the level-0 page table. Each process has its own page table, and so in a context switch, the PTBR is updated along with the other context registers. The PTBR contains a physical address, not a virtual address.

When the MMU receives a virtual address which it needs to translate to a physical address, it uses the PTBR to go to the the level-0 page table. Then it uses the level-0 index from the most-significant bits (MSBs) of the virtual address to find the appropriate table entry, which contains a pointer to the base address of the appropriate level-1 page table. Then, from that base address, it uses the level-1 index to find the appropriate entry. In a 2-level page table, the level-1 entry is a PTE, and points to the physical page itself. In a 3-level (or higher) page table, there would be more steps: there are N memory accesses for an N-level page table.

This sounds pretty slow: N page table lookups for *every* memory access. But is it necessarily slow? A special cache called a TLB[1] caches the PTEs from recent lookups, and so if a page's PTE is in the TLB cache, this improves a multi-level page table access time down to the access time for a single-level page table.

When a scheduler switches processes, it invalidates all the TLB entries. The new process then starts with a "cold cache" for its TLB, and takes a while for the TLB to "warm up". The scheduler therefore should not switch too frequently between processes, since a "warm" TLB is critical to making memory accesses fast. This is one reason that *threads* are so useful: switching threads within a process does not require the TLB to be invalidated; switching to a new thread within the same process lets it start up with a "warm" TLB cache right away.

So what are the drawbacks of TLBs? The main drawback is that they need to be extremely fast, fully associative caches. Therefore TLBs are very expensive in terms of power consumption, and have an impact on chip real estate, and increasing chip real estate drives up price dramatically. The TLB can account a significant fraction of the total power consumed by a microprocessor, on the order of 10% or more. TLBs are therefore kept relatively small, and typical sizes are between 8 and 2048 entries.

An additional point is that for TLBs to work well, memory accesses have to show *locality*, i.e. accesses aren't made randomly all over the address map, but rather tend to be close to other addresses which were recently accessed.

6.2.4 Introduction to Paging

We will briefly introduce *paging* to finish off this lecture. When a process is loaded, not all of the pages are immediately loaded, since it's possible that they will not all be needed, and memory is a scarce resource. The process' virtual memory address space is broken up into pages, and pages which are valid addresses are either loaded or not loaded. When the MMU encounters a virtual address which is valid but not loaded (or "resident") in memory, then the MMU issues a *pagefault* to the operating system. The OS will then load the page from disk into RAM, and then let the MMU continue with its address translation. Since access times for RAM are measured in nanoseconds, and access times for disks are measured in milliseconds, excessive pagefaults will clearly hurt performance a lot.

[1] TLB stands for "translation lookaside buffer", which is not a very enlightening name for this subsystem.

Chapter 7

Paging

In recent lectures, we have been discussing virtual memory. The valid addresses in a process' virtual address space correspond to actual data or code somewhere in the system, either in physical memory or on the disk. Since physical memory is fast and is a limited resource, we use the physical memory as a cache for the disk (another way of saying this is that the physical memory is "backed by" the disk, just as the L_1 cache is "backed by" the L_2 cache).

Just as with any cache, we need to specify our policies for when to read a page into physical memory, when to *evict* a page from physical memory, and when to write a page from physical memory back to the disk.

7.1 Reading Pages into Physical Memory

For reading, most operating systems use *demand paging*. This means that pages are only read from the disk into physical memory when they are needed. In the page table, there is a *resident* status bit, which says whether or not a valid page resides in physical memory. If the MMU tries to get a physical page number for a valid page which is not resident in physical memory, it issues a *pagefault* to the operating system. The OS then loads that page from disk, and then returns to the MMU to finish the translation.[1]

In addition, many operating systems make some use of *pre-fetching*, which is called pre-paging when used for pages. The OS guesses which page will be needed next, and begins loading it in the background, to avoid future pagefaults. This depends heavily on locality of accesses, namely that future accesses will be near recent accesses, and this often true.

[1]In contrast, if the MMU issues a pagefault for an *invalid* virtual address, then the OS will issue a *segfault* to the process, which usually terminates the process. Segfault is an old term for "segmentation fault", or "segmentation violation", leading to the name of the corresponding Unix signal: SIGSEGV.

7.2 Evicting and Writing Pages from Physical Memory

When do we write a page from physical memory back to the disk?

In general, caches have two broad types of writing policies. One approach is a *write-through* cache. In this case, when a value in the cache is written, it is immediately written to the backing store as well (in this case, the disk). The cache and backing store are always synchronized in this case, but this can be very slow. The other main approach is a *write-back* cache. In this case, the backing store and the cache are sometimes out of sync, but this approach is much faster. This is what is used with paging, for obvious speed reasons.

When a page is loaded from the disk to physical memory, it is initially *clean*, i.e. the copy in physical memory matches the copy on disk. If the copy in memory is ever changed, then its page-table entry is marked *dirty*, and it will need to be written back to the disk later.

When physical memory fills up, and a non-resident page is requested, then the OS needs to select a page to *evict*, to make room for the new page. The evicted page is called the *victim*, and is saved to the so-called "swap" space.[2] The swap space is a separate region of the disk from the file system, and the size of the swap space limits the total virtual address space of all programs put together (though in practice, there is a lot of memory shared between processes, for instance shared libraries).

There are a variety different strategies for choosing which page to evict, with tradeoffs for each strategy. These strategies will be discussed later. One thing to note is that evicting a clean page is fast, since it doesn't need to be written back to the disk. A second note is that to speed up the process of evicting pages, the OS can write dirty pages back to disk as a background task. In this way, more pages will be clean and can therefore be evicted a lot more quickly, when it is time to do so.

7.2.1 `mmap()`

In the last section, we discussed how the swap space on disk is a backing store for physical memory, and that the swap space is an area on disk distinct from the filesystem. Since the disk's filesystem already stores files, it would be redundant to store files both on the filesystem and also in the swap space. It would be very convenient if we could use the filesystem itself as an additional backing store for different parts of physical memory. Doing this is very common, and uses the `mmap()` function.

The `mmap()` function maps a file into virtual memory as a big array. For instance, a word-processing document file could be mapped into memory, making it much easier for the programmer to support skipping forward and back in the file, editing it, and saving it back to disk, compared with if the programmer had to rely exclusively on file I/O stream functions. One of the main reasons we need 64-bit architectures is to be able to `mmap()` extremely large files, such as

[2]The term "swap space" is an old term from batch-processing days, when entire processes were swapped back and forth between the physical memory and the backing store. Generally, "paging" is a more accurate and more modern term than "swapping" is.

databases.

7.3 A Day in the Life of a page

Suppose your process starts up, and allocates some memory with `malloc()`. The allocator will then give part of a memory page to your process. The OS then updates the corresponding page-table entry (PTE), marking the virtual page as *valid*. If you then try to modify part of that page, only then will the OS actually allocate a physical page, and so the PTE will now be marked as *resident*. In addition, since the memory has been modified, the PTE will be marked *dirty*.

If the OS ever needs to evict this page, then, since it is dirty, the OS has to copy that page to disk (i.e. it *swaps* it, or performs *paging*). The PTE for the page is still marked valid but is now marked non-resident.

And, if we ever touch that page again (i.e. if we try to read or write it), the OS may have to evict some other page in order to bring our page back into physical memory.

One obvious implication of this is that pagefaults are slow to resolve, since disk accesses are performed. Therefore one possible optimization is for the OS to write dirty pages to disk constantly as an idle background task. Then, when it is time to evict those pages, they will be clean, and the OS won't have to write them to disk. This makes the eviction process much faster.

7.4 Sharing Memory

As we have discussed, page tables map virtual page addresses to physical page addresses. One of the advantages of using virtual addresses is that we can achieve complete separation between processes, in terms of address spaces. One drawback to this is that it is convenient to be able to *share* some things, for example library code which would otherwise be replicated wastefully by many different programs. We don't want to have to load exactly the same library code into every process' address space; we'd prefer to map the library code to its own pages, and let all processes share those pages. This will usually reduce the memory requirements for the system.

In order to do this, we need to do some common tricks with page tables.

The first important memory-sharing concept is known as *Copy-On-Write*, or COW. COW shares pages by default, whenever sharing is still possible. Whenever a new process is created, for example by `fork()`, we "clone" an old process by making a copy of its page tables and marking all referenced pages as read-only.

Then whenever either of the processes (the original one, or the clone) tries to write to one of the pages, the two processes will differ, and sharing is no longer possible. The OS allocates a new page and changes the mapping in one of the page tables. If neither of the processes ever tries to modify a memory location, however, the processes will share the same (read-only) pages forever! COW *tries to maximize the amount of sharing at all times*.

There are a number of status bits in page tables in addition to valid, resident, and dirty, including *readable*, *writable*, and *executable*. Shared libraries are readable and executable (because they include code), but are not writable. In this way, processes can share libraries without worrying that another process can corrupt their address space, because the pages used for shared libraries are not writable.

As a different type of example, interprocess communication (IPC) can use shared memory which is readable and writable as a wide and high-speed communication path between processes.

COW follows the principle of delaying work until it needs to be done, because often it turns out that, due to an error or similar condition, the work won't have to be done after all.

7.5 Allocating new pages

Processes have valid and invalid entries on their page tables. The valid entries all point to somewhere "real" (e.g. a physical page, or some portion of disk in case of non-resident pages, etc). The entries that don't point anywhere are the entries that we will use when allocating a new page.

The allocation of new pages can be done in two ways: either via `sbrk()`, or via `mmap()`. If you want to increase the size of the heap (i.e. the number of valid pages), you can use `sbrk()`. Using `mmap()`, on the other hand, maps a file into a process' virtual address space. In the allocator you implemented, for example, you used `mmap()` to map memory addresses to the file `/dev/zero`. This makes it seem like you were allocating space from `/dev/zero` each time you called `mmap()`. (Remember that whenever you read something from `/dev/zero`, you get only zeroes for as long as you want to keep reading.) But, since `/dev/zero` is a read-only file and we usually call `mmap()` using the `MAP_PRIVATE` flag, we follow the COW rules. When you actually try to write to the memory `mmap()`'d from `/dev/zero`, the OS intervenes and clones the corresponding page. So, instead of actually writing to `/dev/zero`, you end up writing to a new memory page.

Now suppose you mmap 3 pages to /dev/zero. Right after you do this, the process' page table contains three mappings to /dev/zero. These are all COW mappings to a same single page in memory, which itself maps to /dev/zero[3]. However, the first time some of these pages is modified, a *new page* is created, and the corresponding mapping in one of the page tables is modified. Notice that we could have used mmap with any other file instead of /dev/zero; say, an MP3 file. In this case, whenever we mmap'd, we would be actually mapping memory addresses to portions of the MP3 file. If we then tried to write to those areas of memory, we would be indirectly overwriting the file! Notice, however, that we could be careful enough and used the mmap parameter MAP_PRIVATE; then, we would still be able to read from the MP3 file, but all writings to it would be done using Copy On Write.

[3] This implies that if you were to mmap 6 gigabytes, but never touch those, all you'd have really allocated is 4kb – the 4kb corresponding to the single shared page on memory that maps to /dev/zero.

7.6 Cache Replacement Policies

The physical memory acts as a cache backed by the disk. When the physical memory is full, and we want to read in another page from disk, we have to evict a page from physical memory. How do we choose which page to evict? This is determined by the cache replacement policy.

Note that in a theoretical algorithms course, you are generally interested in the worst-case asymptotic performance of an algorithm. In building systems, worst-case performance can be important (for instance, in systems with real-time requirements), but often we can most about the algorithm's performance for the most common case. If a particular algorithm has good worst-case performance, but bad common-case performance, it is a bad algorithm.

7.6.1 Optimal Replacement Policy

The optimal replacement policy, called OPT, is to evict the page which will be accessed farthest into the future. Since we can't predict the future precisely, we can't implement OPT in any real system. We can, however, run OPT on a saved trace of memory accesses, to measure how our real algorithm compares with OPT for a given trace.

7.6.2 Least-Recently Used (LRU)

LRU evicts the page which was last accessed the farthest into the past of any page resident in physical memory, i.e. the least-recently used page. LRU approximates OPT when the recent past is a good prediction of the future, which is often true in real programs. Variants of the LRU policy are currently used in all real operating systems. We will discuss LRU further below.

For certain patterns of memory accesses, however, LRU can be quite bad. For example, the worst case for LRU is a loop which is larger than the cache size (physical memory size in this case). In this situation, LRU will miss, or page fault, on *every* page access. So perhaps there are alternatives which are better than LRU?

7.6.3 Most-Recently Used (MRU) and FIFO

Evicting the most-recently used (MRU) page does very well on LRU's worst case. In general, however, MRU is a bad idea, since many programs exhibit temporal locality in their memory accesses, and MRU is effectively assuming that memory accesses will *not* exhibit locality.

Another possibility is First-In, First-Out, or FIFO. At first, this seems like it would be competitive with LRU, but it is not, since it also ignores temporal locality. For example, with a cache size of three, and the access pattern ABCAD, A would be evicted when D is loaded into the cache, which generally is not a good idea. (FIFO also has an unusual property called Belady's Anomaly in which for contrived examples, means that adding more memory to a system can cause more paging. So adding more physical memory to a computer using FIFO replacement could in theory make it slower. This cannot happen with LRU replacement.)

7.7 Implementation of LRU

How do we actually implement a LRU? Here are several ideas:

1. On every access, mark the page with a timestamp. Whenever we need to evict a page, we search through memory for the oldest page, the least-recently used page. But we need memory accesses to be fast. With this approach, on every memory access we would also need to read(load) a clock (or counter) and perform a store.

2. Maintain a queue of pages. Every time we touch a page, we move that page to the beginning of the queue. When we need to evict a page, we evict the last element of the queue. Again, the pointer manipulation to do this would slow down the memory accesses, which we need to be fast on average.

3. Use a hash table rather than a queue. In this case, we have to compute a hash address on every memory access. Bad idea.

4. *Approximate* LRU. We can do this by maintaining *reference bits* for every page. On each memory access, the MMU hardware sets the page's reference bit to 1. Periodically, the OS tells the MMU to reset all the reference bits. Whenever we need to evict a page, we select one that has a reference bit not set. (So eviction may require an $\mathcal{O}(n)$ search through page tables to find a page with reference bit not set, but the common case, cache hits, are very fast, done in hardware.) This algorithm considers any page which is zeroed to be "old enough". Then, although we can't know exactly what is the *least*-recently used, we do know that whatever is marked is a least *more* recent than something not marked. In practice, this approach works pretty well.

LRU is already only an approximation to OPT. In practice, we also approximate LRU itself. Our LRU approximation *optimizes for the common case*, which is cache hits, rather than cache misses. (If the common case is not cache hits, then your cache is not helping you.)

7.7.1 LRU Implementation Details

Now, let us discuss two related algorithms for deciding which pages to evict. The *clock algorithm* is one of the most popular choices. It works by keeping frames in a circular structure. (Note that this circle may be very large: a 32-bit machine with 4k pages can have up to $2^{20} \approx 1$ million frames.) When a page fault occurs, it checks the reference bit of the *next frame*. If that bit is zero, it evicts that page and sets its bit to 1; if the reference bit is 1, the algorithm sets the bit to 0 and advances the pointer to next frame. For more details, please refer to `http://en.wikipedia.org/wiki/Page_replacement_algorithm`.

An implementation which is often used in real operating systems is a *segmented queue*. This type of algorithm divides all existing pages into two sets of pages. The most-active one-third of all pages use a clock algorithm. After that, pages that are evicted out of the clock are moved to a "uncommon" linked list, in which we use exact LRU. This way, we approximate LRU for the frequently-referenced pages (1/3 of the page frames - fast clock algorithm), and at the same time use exact LRU on the infrequently accessed pages (2/3 of all page frames). Since the pages in the uncommon list are not accessed very frequently, it is okay if their accesses are a little slower due to some pointer manipulation.

7.7.2 Drawbacks

The question of *fairness*, regarding page eviction, is a hard one. How do we decide what is fair?

Many operating systems use *global LRU*, where pages from all processes are managed together using the approximate LRU algorithms described above. This is easy to implement, but has a number of problems.

For instance, with global LRU, there's no isolation between processes, and greedy or badly-written programs will push other programs out of physical memory. In addition, the priority which one gives a program is generally priority for the scheduler, not priority for physical memory (though there are approaches to try to deal with this).

There's also the "sleepyhead" problem, which is where an important but rarely-used program will get paged out, and then start up slowly. For instance, the `ntpd` (network time protocol) program doesn't run often, but when it does run, it needs to make precise timing measurements, which would be impacted by pagefaults. `ntpd` and similar programs can get around this problem by "pinning" their pages, meaning that their physical pages are not allowed to be paged out to disk.

In general, the problem of fairness is that processes that are greedy (or wasteful) are rewarded; processes with poor locality end up squeezing out those with good locality, because they "steal" memory. There is no simple solution to this problem.

Chapter 8

Dynamic Memory Management

Usually memory is allocated from a large pool of unused memory area called the heap. In C++, dynamic allocation/deallocation must be manually performed using commands like *malloc, free, new and delete*. *Malloc* allocates space of a given size and gives a pointer back to the programmer. The programmer then can do whatever he or she wants with it. The *new* command, on the other hand, allocates a specific object of a given size. The general way in which dynamic allocation is done is that the program asks the memory manager to allocate or free objects (or multiple pages); then, the memory manager asks the OS to allocate/free pages (or multiple pages). Notice, however, that the allocator *does not* give the whole allocated page back to the program; it just gives what it asked for. The rest of the page (i.e. the parts not given to the program) is saved for future memory requests.

8.1 Allocation techniques

Since most of the programs require a lot of memory allocation/deallocation, we expect the memory management to be fast, to have low fragmentation, make good use of locality, and be scalable to multiple processors. Newer issues include security from attacks (e.g. use randomization of locations to minimize effect of buffer-overflow attacks) as well as reliability in the face of errors (e.g. detecting double `free()'s` and buffer overruns).

In analyzing the speed of an allocator, the key components are the cost of malloc, the cost of free, and the cost of getting the size of an object (for realloc). In certain types of programs, for instance those with a lot of graphics manipulation, allocators can become the performance bottleneck, and so speed is important.

We say that *fragmentation* occurs when the heap, due to characteristics of the allocation algorithm, gets "broken up" into several unusable spaces, or when the overall utilization of the memory is compromised. External fragmentation happens when there is waste of space outside (i.e. in between) allocated objects; internal fragmentation happens when there is waste of space inside an allocated area.

Remember that the allocator might at any single time have several pages with unused space, from which it could draw pieces of memory to give to requesting programs. There are several ways to decide what are the good spots, among those with free memory, from which to take memory. The *first-fit* method finds the first chunk of desired size and returns it; it is generally considered to be very fast; *best-fit* finds the chunk that wastes the least of space; and *worst-fit* takes memory from the largest existent spot, and as a consequence maximizes the free space available. As a general rule, first-fit is faster, but increases fragmentation. One useful technique that can be applied with these methods is always to coalesce free adjacent objects into one big free object.

8.2 Keeping track of free memory

How do we keep track of where the free pieces of memory are? One idea is to maintain a set of linked-lists of free space; each linked-list will store free chunks of some given size (say, one list for chunks of 4 bytes, one for chunks of 8, 16, etc). This approach resembles the best-fit algorithm, since we can easily find the list with chunks of memory that are the closest to what we need; the difference, of course, is that usually the linked lists store not objects of *arbitrary* size, but only powers of two. Also, we can never coalesce adjacent elements of the lists, or split elements, since then the very idea of segregating by size classes wouldn't make sense anymore.

Another possible technique to keep track of free memory is called Big Bag of Pages (BiBOP). In this technique, a bunch of pages, usually segregated by size classes, is maintained such that there is a header on each page; on this header, among other things, we can find a pointer to the next page of objects of that given size, and also a pointer to the first free "slot" inside that page. Using the right optimizations we can make BiBOP find and manage free slots of memory in O(1).

8.3 Custom memory allocation

Some people write custom memory allocators to meet their specific needs. Although this is not needed for most of the applications, it is also not uncommon. The goal, of course, is to replace the OS allocator in an optimized version that deals more efficiently with the most frequently used data structures of some given application. Ideally, we expect the new allocator to require reduced runtime, provide expanded functionality (sometimes) and run on reduced space (rarely). Examples of applications that use custom allocators are the Apache webserver, GCC, the STL and most database servers. Unfortunately, there are also some drawbacks regarding the use of custom memory allocators, such as the increased burden on programmers, as well as the fact that they seldom allow the use of standard memory debuggers.

8.4 Kinds of allocators

We can imagine several types of memory allocators. *Per-class allocators*, for instance, are useful when we know that we're going to allocate the same type of object again and again. Then, instead of segregating by object sizes (i.e. maintaining a list for all 8, 16, ... byte objects), we can create a list of memory slots for objects of, say, 18 bytes. This can be seen as a big pool of free memory chunks used by some commonly occurring structure/object. Per-class allocators are typically very fast and simple, but might also be space-inefficient. An example is the "slab allocator" in the Linux kernel.

Custom patterns allocators, on the other hand, are useful whenever we know we're going to have a specific type of allocation dynamics. For instance, suppose we know that we'll always have to allocate a lot of objects, and then we'll delete them all, but we will never allocate anything after we started deleting (e.g. alloc, alloc, delete, delete, delete). In this case, a custom pattern allocator could simply use a stack and a pointer to the last occupied position of that stack; then, when allocating, we can just request the size of the object from the memory pointed to by our pointer. Allocation and deallocation are O(1). In fact, this type of pattern allocator can be implement using pointer-bumping (i.e. by "just moving pointers", instead of performing crazy linked-list operations). Notice that if we use this type allocator (i.e. one that assumes a specific pattern of allocation) on some program that *does not* exhibit that pattern, we can end up with tons of unused memory, since a free slot n of memory cannot be reclaimed until we free the very last piece of occupied memory following n.

Allocation can also be done using *regions*. The idea in this case is to create a big blob of free memory, called a region. We then define that deletion in that region will only occur "en masse", that is, for all objects inside the region. This is interesting if we are dealing with modular applications, e.g. a web browser that has to load a plug-in and then free all the memory used by that plug-in. In this case, we don't have to dealloc each individual object of the plug-in; in fact *we don't care* what are the objects used internally by the plug-in. We can simply delete all the region of memory used by it. Although very fast when dealing with these specific scenarios, allocation by regions is very prone to programmer errors and are significantly nontrivial (i.e. a pain) to maintain.

As a final remark, notice that there is no single best allocator; different types of programs perform wildly different under different types of allocators, because of the ways they use memory, their patterns of allocation, etc.

8.5 Review of Custom Allocators

We reviewed per-class allocators, custom pattern allocators, and region allocators from last lecture.

8.6 DieHard Allocator

Allocators can also be used to avoid problems with unsafe languages. C and C++ are pervasive, with huge amounts of existing code. They are also memory-unsafe languages, in that they allow many errors and security vulnerabilities. Some examples include double `free()`, invalid `free()`, uninitialized reads, dangling pointers, and buffer overflows in both stack and heap buffers.

DieHard is an allocator developed at UMass which provides (or at least improves) soundness for erroneous programs.

There are several hardware trends which are occurring: multicore processors are becoming the norm, physical memory is relatively inexpensive, and 64-bit architectures are increasingly common, with huge virtual address spaces. Meanwhile, most programs have trouble making full use of multiple processors. The net result is that there may soon be unused processing power and enormous virtual address spaces.

If you had an infinite address space, you wouldn't have to worry about freeing objects. That would mostly eliminate the double `free()`, invalid `free()`, and dangling pointer bugs. And if your heap objects were infinitely far apart in memory, you wouldn't need to worry about buffer overflows in heap objects.

DieHard tries to provide something along these lines, within the constraints of finite physical memory. It uses randomized heap allocation, so objects are not necessarily contiguous in virtual memory. Since the address space is actually finite, objects won't actually be infinitely far apart, and buffer overruns might actually cause collisions between heap objects. But this is where the multicore processors come in: With the unused processor cores, run multiple copies of the application, say three copies, each allocating into their own randomized heap. So the heap errors are independent among the three copies of the application. All copies get the same input, and the output is the result of voting among the three copies of the program. If one instance of the application disagrees with the other two, it is killed, since there was likely a collision between heap objects in that one. Similarly, if one instance dies with a segfault or other error, the others remain running. Surviving copies can be forked to replace copies which were killed off, though this reduces the independence among copies.

This is transparent to correct applications, and allows erroneous applications to survive longer, which will make their users happier. Furthermore, when a copy dies or is killed off, the error can be noted and sent to the software maintainers automatically.

DieHard increases the chances of turning problematic errors into benign errors. It trades memory space for robustness, which is probably fine as memory prices continue to drop. It uses randomization and replication to provide fault-tolerance.

Chapter 9

Garbage Collection

The dynamic memory allocator is a layer between the application and the OS, managing heap objects. When a program requests memory from the allocator (via `malloc()`, for instance), the allocator will return a pointer (or reference) to a piece of memory of the appropriate size. When the program is done with the memory, the memory should be released back to the allocator. Languages such as C and C++ leave this job to the programmer to perform manually, for example by using `free()`. On the other hand, languages such as Java, python, etc automatically manage dynamically-allocated memory, which makes the programmer's life easier, and can eliminate entire classes of memory management bugs.

Although using `free()` and `delete` is relatively simple, it can be tricky to get them right. A significant fraction of bugs in C and C++ programs are related to manual memory management. If we forget to free objects, we end up with memory leaks; if we free memory too soon, we end up with "dangling pointers"; also, we can try to do weird things, like performing double frees, etc. Therefore, a process that manages memory automatically is clearly useful. The most important concept for correctly implementing a garbage collector is that of *live objects*: a live object is any object that can still be reached through one (or more) pointers.

Let's analyze the following C++ code:

```
node x = new node ("happy");  // note: class node implicitly is a pointer
node ptr = x;
delete x;
node y = new node ("sad");    // can allocate into same space x pointed to!
cout << ptr->data << endl;    // can print "sad"
```

Even after the call to `delete`, `x` is still alive, because it can be reached through `ptr`. But C++ doesn't keep track of that. After the call to `delete`, the allocator considers memory area that used to belong to `x` as free. So, when new space is allocated to `y`, the space which `x` used to point to (and which `ptr` still points to) could be reused. This type of program can lead to insidious, hard-to-find bugs.

9.1 Garbage Collection Algorithms

Any garbage-collection algorithm has to reclaim all objects that are in memory but that can *no longer* be reached through any pointers. These objects are clearly useless, and their memory areas can be returned to the system. Even though garbage collection can be very helpful from a programmer's point of view, it can be slow and can degrade cache performance and page locality.

The main question for all garbage collection algorithms is: "how can we know whether or not something is still reachable?" Some of the classical algorithms for determining this are the mark-sweep algorithm, reference counting, and semispace.

9.2 Mark-sweep

The objects that a program can access directly are those objects which are referenced by local variables on the processor stack, or by any global/static variables that refer to objects, or by variables in CPU registers. In the context of garbage collection, these variables are called the *roots*. An object is indirectly accessible if it is referenced by a field in some other (directly or indirectly) accessible object. An accessible object is said to be *live*. Conversely, an object which is not live is garbage. Note that heap objects which are live are indirectly accessible from the roots or other heap objects.

The idea of mark-sweep is relatively straightforward. We start at the roots, and recursively visit every object accessible through pointers, marking them as live. At the end of the process, everything not marked is considered garbage and will be deleted. Notice that mark-sweep can perform *lazy garbage collection*, in the sense that it does not necessarily need to remove the garbage immediately.

Note that mark-sweep does not clean up memory which is allocated, but simply never used. Also, periodically we have to visit all objects recursively, starting from the roots. For a large program, this will be slow. This is a problem with the traditional mark-sweep algorithm.

9.3 Reference counting

The idea of reference counting is to maintain, for every object, the total number of references to that object, i.e. the number of "incoming" pointers. Whenever the number of references is zero we know that the object is not accessible through any pointers, and thus it is garbage. Also, whenever we choose to delete some useless object, we have to recursively check that object to see if it contains pointers to other objects; if so, we have decrement the reference counters for those objects as well.

One problem of reference counting, though, is how to deal with cycles. Suppose object A points to B, and B points to A. If these objects *can only be reached through each other*, we have to free them too, even though their counters are not zero! Note that cycles can common in many data structures; for example, consider a doubly-linked list. The commonly adopted solution to this problem is to run mark-sweep now and then, in order to remove cyclic references, and then run normal reference counting on the rest of the time.

Reference counting spreads out the overhead of garbage collection. Traditional mark-sweep does all of the garbage collection operations in one large pass, whereas with reference counting, every reference operation will have some additional overhead, spreading the garbage collection costs out in time. Relative to traditional mark-sweep, reference counting scales better even for large programs.

9.4 Semispace GC

Semispace works by maintaining two disjoint areas from which memory can be allocated. These areas are called the *from-space* and the *to-space*. At first, the algorithm allocates memory only from the from-space, without ever worrying about garbage collection. Allocation then is typically performed using simple pointer bumping, which simplifies the whole process a lot. When we finally run out of space in the from-space, we sweep through all allocated objects that can be somehow reached; those are the ones that are still live. We then move each of these live objects to the to-space, taking care to update all pointers to the live objects to point to its new location in the to-space. Hence semispace is called a *copying* collector. After having done all this moving, only live objects are in the to-space. From this moment on, we can start using the to-space to perform allocation. The process is repeated again when the to-space eventually runs out of space. This method has as advantages the fact that it might "compact" objects in the memory, thus increasing locality and minimizing fragmentation. Also, when performing allocation from one of the spaces, it can use simple pointer bumping, which is very fast. However, this approach doubles the memory requirements.

9.5 Generational GC

Generational GC is an optimization used by copying collectors. Its key assumption is that *most objects die young* (this is called the *generational hypothesis*). When this assumption holds, we can optimize the GC for the common case. To do so, we first allocate objects into a *nursery*, which is a small area of memory. We can collect in that area very frequently, since (according to the generational hypothesis), most of them will be garbage. Also notice that, since most of them will indeed be garbage, it will be very fast to traverse them using, e.g. Mark-Sweep. After performing GC in the nursery, we copy out all the survivors into the *mature space*; by doing so, we are assuming that all objects that survived the nursery will most likely live for a long time. Note that we still have to collect the mature space; however, since those objects are not likely to die soon, we don't have to garbage-collect that area as frequently as we garbage-collect in the nursery. A key idea is that we need to keep track of pointers from the mature space into the nursery.

9.6 Conservative GC

Conservative GC can be used for languages such as C and C++, which were not explicitly designed for garbage collection. This is a non-copying technique. A conservative garbage collector is one that, instead of knowing the exact location of each object, discovers the set of live objects by scanning a region of memory, looking for areas that may be objects. Because C and C++ allow casting, anything that can hold a pointer could conceivably be a pointer, such as an `unsigned long` which is cast as a pointer, or a floating-point type which is cast as a pointer, etc. Conservative GC more or less uses the "duck test", paraphrased here as "if it looks like a pointer and acts like a pointer, it's probably a pointer". Starting from the roots, we can find all objects which look like pointers, and the objects to which they would point if they were in fact pointers, and mark those objects as live. By doing this, we can provide garbage collection to languages which otherwise would not support it.

The possible objects found may or may not actually be objects, but we are ensured that all live objects referred to from that particular area are found by the garbage collector. Since we must discover the areas that might be objects, we also have to know how to identify pointers; usually, if something looks like a pointer, we assume it is a pointer. Then, conservative GC traces through those "pointers", marking everything that is still live.

Some of the drawbacks of this method are: *1)* areas of memory that look like pointers to objects, but aren't, cause garbage objects to be retained as long as the fake pointer exists; this increases memory usage of the garbage collector, and can cause other limited resources to be exhausted; and *2)* the methods of finding exactly which areas to trace aren't always portable.

9.7 Comparing GC to malloc

Garbage collectors trade space for time. If we collect all the time (which requires a lot of processing time), the GC allocator will use the least memory possible. On the other hand, if we have a lot of space to use, we can collect very infrequently and thus be very fast. In general, with enough extra memory, GC can take almost no extra time extra, when compared to explicit memory management. For example, if we are willing to spend 4x more space than usual, garbage collection can run as fast as explicit memory management; if, on the other hand, we don't give it too much space, it will have to run very frequently and thus will demand much more processing time than explicit memory management.

Faced with this tradeoff, when should we use a garbage collector instead of manual memory management? Generally, use a garbage collector *1)* if you have lots of memory; and *2)* if we *really* want to avoid bugs or extra programming effort related to memory management. If, however, hardware resources are the limiting factor and programmer resources are less of a limiting factor, then manual memory management is probably a better idea.

Chapter 10

Building Concurrent Applications

In building concurrent systems, we want to overlap access to different resources (e.g. CPU, disk, network) in order to hide latency and maximize parallelism and throughput. Furthermore, we want our systems to be relatively simple to build and maintain. There are several standard threading models, or patterns, for building concurrent applications, including thread pools, producer-consumer, bag-of-tasks, and work queues. We discuss these below.

10.1 Basic Server Model

Suppose we want to program a Web server; we could easily do so without using concurrency, as follows:

```
while(true)
   wait connection
   read from socket and parse url
   look up url contents in cache
   if (!in cache)
        fetch from disk
        put in cache
   send data to client
```

How do we make this code work for lots and lots of clients? Since there are a lot of slow operations being performed (e.g. connecting to the networking, accessing the disk, etc), processing only one client at a time clearly is not scalable. Our goal then is to build a concurrent application that both minimizes latency and maximizes parallelism.

10.2 New Thread for each Client

We could improve the model above using our previous knowledge of programming with threads, and just spawn a new thread for each new connection to the server. This would work, but with the drawback of requiring lots of thread creation operations, which can be relatively expensive.

10.3 Thread Pools

Another solution then would be to keep a *pool of threads*. Whenever a new task arrives, the system would simply get a thread from pool and set that thread to work on the given task. Whenever the pool is empty, the system blocks until another thread completes its task.

One natural question now is: how many threads do we "pre-create" and add to the pool? Ideally, we would want to create an infinite (or large enough) amount of threads; however, this is clearly not possible, since each thread has a descriptor, a stack, etc, but our main memory is finite. Also, beyond a certain point, creating more threads does not help anymore, and the server starts to slow down again due to things like paging or lock contention. For example, if we only have 4 CPUs, it doesn't help to try to run a billion things at the same time. In practice, the fine-tuning of how many thread to add to the pool for any given application is set empirically, or perhaps adaptively.

10.4 Producer-consumer

The general idea of a producer-consumer architecture is related to *building a pipeline of threads*, similar to the web spider from Project 2. Each step of the processing will now be done by a specialized threads; whenever one thread is done with its part of the processing, it forwards the data to the next stage of the pipeline, like a factory assembly line. In a producer-consumer system, we guarantee that each stage of the pipeline signals the next stage of the pipeline when it completes its task, and blocks when there's nothing else for it to do, etc.

If we were to implement our web server in a producer-consumer fashion, we could simply create a pair of threads: one for reading the URL requested by the client, and another one for writing the answer back to the network. We could also create yet another intermediate thread, which would first look for the URL in a cache. This intermediate thread might help the overall average response time of the system, if we can implement the *shortest time to completion first* policy, i.e. finish the easy things first, which leads to the lowest average response time.

As with the thread pool approach, it is not always clear how many threads we need for each of stage of the pipeline; the exact amount depends on the application and is usually fine-tuned manually. Since each thread has a specialized task, if the number of threads for each task is not well-tuned, then many threads might be idle, wasting resources.

In general, the producer-consumer approach works well if the producer and the consumer are symmetric, i.e. if they proceed roughly at same rate. On the other hand, if the order of processing doesn't matter, it doesn't make sense to use a producer-consumer architecture, with its ordered pipeline of processing.

10.5 Bag of tasks

The producer-consumer model above has a standard human analogy of an assembly line. Humans have specialized skills, however, whereas threads running the same program do not need to be specialized: every thread has access to the same code and data. So any thread could do the work of any other thread.

In the *bag of tasks* approach, we have a collection of mostly independent "worker threads"; we no longer need to have different types of threads (one for the parser, one for the workers, etc). A bag of tasks system is very simple: every time a worker doesn't have anything to do, it goes to the bag of tasks and gets something to work on; in our web server example, this bag of tasks could contain new connections to be processed.

The advantage of the bag of tasks approach is that we can imagine all workers as being homogeneously implemented, each one running a specific piece of code depending on the type of the task it has to process at that time (imagine each worker as being implemented with a big switch statement).

Let us now present a way to convert the pseudo-code for the web server, given in the first section, to a Bag of Tasks style. Assume we have an *addWork* thread and a bunch of worker threads.

```
addWork thread():   // only one thread of this type
  while(true)
    wait connection
    bag.put(url)

worker thread():    /// there will be a bunch of these threads
  while(true)
    url = bag.poll()
    look up url contents in cache
    if (!in cache)
        fetch from disk
        put in cache
    send data to client
```

The bag of tasks approach seems to exploit the available parallelism well, and it is easier to program and to extend. It has one major performance drawback, however, namely that there is only *one* lock protecting the bag; this implies contention. Also, imagine for instance that some of the workers are pretty quick; it could be the case that they acquire the lock, finish processing, release the lock, re-acquire it, etc. The problem now is that each worker would be, itself, a new bottleneck. One possible solution to this problem is to have *several bags*; this is the approach that we discuss next.

10.6 Work queues

In this type of concurrent architecture, each thread has its own work queue of jobs. Now, contrary to the Bag of Tasks architecture, we have no single point of contention; surely there is still contention for the lock of each of these queues, but at least there is *less* contention than before.

Although the existence of several "bags" eases the contention problem, we now have yet another drawback: suppose that one thread finishes all its jobs and yet there is still more work to be done on some other queues; this should never happen, given that there exists an idle thread. The solution is that the idle thread voluntarily *steals* work from another (random) thread. This type of behavior can actually be shown to be optimal in terms of load balancing, and works great for heterogeneous tasks. This is also a flexible model for programming concurrent applications, as it allows us to easily re-define tasks, performs automatic load balancing, separates thread logic from functionality, etc. Overall, the use of work queues is very popular when structuring servers nowadays.

Chapter 11

Networking and Distributed Systems

Networks and operating systems have a lot of overlap. In this course, however, we are not going to discuss networking in detail, but rather focus on the *networking abstractions* provided by operating systems. These abstractions are necessary because in real-life networks we have to deal with a large variety of hardware and protocols. It would be almost impossible to implement practical networking solutions if we had to actually take all these factors into account when implementing, say, an instant messenger. When designing application-layer software, we we want to focus solely on the logic necessary to solve that *particular* problem; we don't want to deal with the details of routing, lost messages, etc. Furthermore, since networks are slow, we will need to use our previous work on threading and concurrency to hide latency and maximize parallelism.

What abstractions can the OS provide us so that we don't have to worry about all the nitty-gritty details of networking?

11.1 OS abstractions

When we download a movie from the Internet, we don't care about, say, packet sizes. However, even if we are not aware of how low-level details of networks are implemented, our data is actually being sent as lots of small packets, some of which might be lost or arrive out of order. Therefore, it is important that the protocols we use guarantee that we won't have to deal with any of these problems. In other words, all we want is a simple interface that solves our problem; this interface for network programming will be provided by the operating system. Some of these abstractions are:

- process-to-process communication
- remote function calls
- security
- hosts have human-readable names

- ordered, reliable communication
- streams of bytes, instead of discrete packets
- support for large messages
- routing to final destination
- independence from the low-level network and hardware details

11.2 Protocols

Modern networks are not implemented as a single piece of software; that would render the task of dealing with multiple technologies and manufacturers virtually impossible. The solution for this problem is to structure computer networks as stacks of different *protocols*. A protocol can be seen as a "language" in which two entities (computers, routers, etc) communicate with each other. Typically, one protocol is created to solve a predetermined set of networking tasks, such as being able to send fixed-length messages, or guaranteeing message order, etc. The capability of a protocol to solve these tasks makes it a "black box" that can be used by protocols on the level immediately above; similarly, the protocol itself can make use of the capabilities provided by the protocol on the level immediately below. Thus, all modern networks are implemented as a stack of abstractions, all of which are implemented as composable protocols; we say that the layer N provides services to layer N+1 and uses the services from layer N-1.

As an example, consider the stack presented in Figure 11.1; in this example we show how the HTTP protocol (or, similarly, SMTP and ssh) can be implemented using the services provided by the TCP protocol. Also, the TCP protocol is implemented using the services of the IP layer, which itself uses the services provided by the Ethernet layer. Notice that one advantage of this type of architecture is that if we were to build a new functionality (say, wireless connections, instead of Ethernet), it would suffice for us to implement this new layer and to guarantee that it provides, to the IP level, the same services that were once provided by the Ethernet level. Nowhere would we need to care about the details of TCP, HTTP, or, for that matter, any of the top-level protocols.

11.3 Some transport protocols: UDP and TCP

Most real-life applications are built on top of the UDP and TCP transport protocols.

UDP, which stands for *User Datagram Protocol*, provides the capability of delivering individual messages from one computer to another. However, these messages might be lost or can arrive out of order. Why would someone want to use such a protocol? One example application might be video streaming, in which we don't care that much if a packet was lost, and if the video has kept playing, we're not going to go back and show an old frame which has arrived late.

While UDP has its place for some specific applications, by far the most used transport protocol is TCP (*Transmission Control Protocol*). TCP provides a connection with reliable, in-order delivery of a

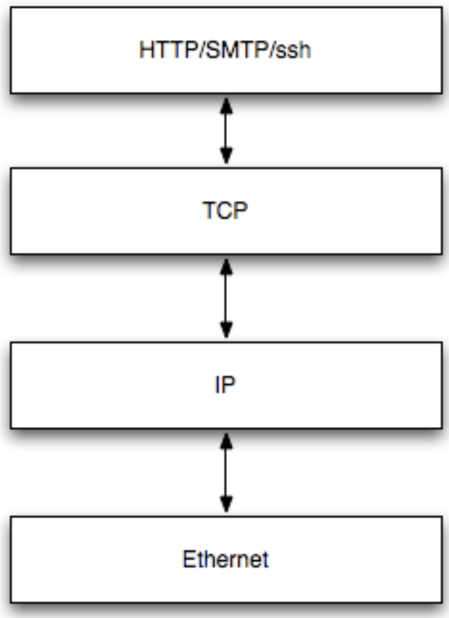

Figure 11.1: Example of a stack of protocols.

stream of bytes, so the application programmer doesn't have to worry very much about dropped packets, packets which arrive out of order, or the details of network message boundaries.

Notice that maybe the most important feature provided by transport protocols is the possibility of performing *process-to-process communication*. This capability means that not only can we send messages from one machine to another, but from a specific process in the origin machine to a specific process in the destination machine. In addition to the IP addresses, we also use the *port* numbers, which are endpoint sub-addresses, visible to the application layer. If we think of IP addresses as analogous to "building numbers", we can interpret ports as analogous to "apartment numbers".

11.4 Sockets

When programming a server or a client, we have to deal with port numbers and IP addresses, but we usually do this through an abstraction called a *socket*. Sockets are the standard API for network programming; they are the abstraction which we use to describe a connection, i.e. a duplex communication. The typical way of creating and using sockets, in C, is the following:

```
// Client side ------------------------------------------------------------

   struct addrinfo *addr;
```

```
    int fd = socket(AF_INET, SOCK_STREAM, 0);        // fd is the socket's filehandle
    getaddrinfo("www.cnn.com", "http", NULL, &addr); // converts hostname to IP addres
    connect(fd, addr->ai_addr, addr->ai_addrlen);    // connect to remote host
    send(fd, send_buf, len, 0);                      // now send then receive data
    ...
    recv(fd, recv_buf, len, 0);

// Server side --------------------------------------------------------------

    struct addrinfo *addr, *remote_addr;
    int fd = socket(AF_INET, SOCK_STREAM, 0);     // fd is the socket's filehandle
    getaddrinfo(NULL, "http", NULL, &addr);       // convert service to port number ("ht
    bind(fd, addr->ai_addr, addr->ai_addrlen);    // bind socket to a particular port nu
    listen(fd, 1); // listen for connections. Second argument is # of allowable waitin

    // Now accept connection on socket fd; when a connection comes in, we'll move it to
    int newfd = accept(fd, remote_addr->ai_addr, &(remote_addr->ai_addrlen));

    // we process the accepted connection on newfd, with recv() and send(),
    // and at the same time, we'll keep using fd to accept other incoming connections
```

11.5 Distributed systems – implementation patterns

There are two dominant forms of building distributed systems: *client-server* systems, i.e. systems in which one machine acts as a server and act as clients; and *peer-to-peer* systems, where each host is both a server and a client. Standard web access follows the client-server model, and services such as BitTorrent follow the peer-to-peer model. However, in real life these are not completely distinct; for instance, internally Google applies some of each of these methodologies. Actually, client-server system are sometimes implemented as a *chain* of clients and servers, even though from the point of view of the client there's only one server.

Another possible building pattern is that in which the client communicates directly with only a server, but in the background its requests are actually processed by a *cloud of machines*, which actually resembles the way Google works internally. In this case, the cloud of machines implements a system with lots of redundancy (e.g. via a big distributed file system). Some of the machines in the cloud hold data, and other machines hold meta-data. When some client requests, for instance, a file, the meta-data servers first identify which other machines hold each piece of that file. This type of architecture represents the current trend being followed when creating large scale systems. So, in practice, we see that current successful large scale systems are a mix between client-server architectures and peer-to-peer systems, implemented with lots of storage and computational redundancy.

11.6 Distributed systems

Distributed systems are composed of a number of physically separate machines connected by one or more communication links. Unlike parallel systems, there's no shared clock or memory. This means that distributed systems are very loosely coupled; for example, if your web browser crashes while visiting cnn.com, cnn.com itself will not crash. In general, we can identify several advantages in building distributed systems, such as:

- Shared resources do not need to be unnecessarily replicated. For example, you don't need to store all your files in all Edlab machines, but rather a distributed network filesystem lets it appear that your Edlab files are accessible on any of the Edlab machines. At the same time, you generally do not want to store all your files in just *one* machine, and so a distributed filesystem could use replication across multiple machines to provide fault tolerance.
- Expensive (or scarce) resources can be shared. As an example, a poster-size color laser printer can be shared via a local network.
- It is possible to present the same working environment to users, by keeping their files and profiles on remote servers. For instance, today, as opposed to during the 1970s, each one of us owns or uses several computers, and we do not just use one big computer. Unfortunately, this is also a problem, since, say, data on our cellphone can be out of sync with data on our laptop. The good news is that distributed systems help to standardize these environments; one possible solution to this problem, for instance, is to implement thin clients, in which the user might owns only one fast machine but lots of very simple "keyboard-screen" machines.

11.7 Building distributed systems

Unfortunately, building real-life distributed systems is not easy. It is hard, for instance, to implement instructions such as "send this data structure to be processed on that computer." To enable this, we have to worry about details of the different hardware architectures and operating systems involved, etc. The good news is that there are ways to abstract away some of the details of this process. RPC (*Remote Procedure Call*), for example, implements the idea of calling functions which will be executed in a remote machine. When a RPC function is called, what really happens is that the RPC library performs data marshalling on the function parameters and sends them to the remote computer; that computer, after having executed the function itself, marshalls and returns the results back to the caller. Nowadays, several distributed systems are being built using technologies derived from RPC; among these technologies are RMI(Java), CORBA and DCOM.

11.8 Ajax

Ajax is a relatively new set of techniques that, in a sense, implement some of the functionalities originally provided by RPC. Ajax is primarily used to build web applications with good performance and interactive responsiveness. The name comes from “Asynchronous JavaScript and XML” (though the general technique doesn’t necessarily need to use JavaScript or XML).

The idea is to use JavaScript (or the equivalent) to run a web application in your browser, and do as much as possible on your computer, minimizing data exchanges over the network. Typically, Ajax applications are designed so that most data exchanges can be asynchronous and in the background. This means that you will be able to keep doing useful work with the application even on a relatively slow network connection.

Ajax resembles RPC because most of the processing can be done “remotely” on the client, instead of on the server. A good example of this technology is Google Maps. One reason why Google Maps works so well is because it explicitly uses the fact that most users own a fairly fast computer, and thus lots of the actual processing can be done locally. The user just needs to fetch asynchronously each piece (tile) of the map, and, when these are received, he or she can process them locally. If, on the other hand, we were to send every mouse movement to Google, and expect all processing to be performed remotely, we would end up with a very slow and impractical system. Notice that because Ajax is asynchronous, it scales very well with slow connections. Ajax does not block waiting for pieces of data to arrive before making the system usable. Instead (in the case of Google Maps), we notice that the user is allowed to manipulate the map (panning around, zooming in, etc) even when the map itself has not yet been completely received.

Appendix A

Introduction to C++

A.1 Introduction

C++ is an object-oriented language and is one of the most frequently used languages for development due to its efficiency, relative portability, and its large number of libraries. C++ was created to add OO programming to another language, C. C++ is almost a strict superset of C and C programs can be compiled by a C++ compiler.[1] In fact, any of the C++ assignments in 377 can be completed strictly using C, but you will probably be more comfortable using C++.

When comparing C++ and Java, there are more similarities than differences. This document will point out the major differences to get you programming in C++ as quickly as possible. The highlights to look at are C++'s handling of pointers and references; heap and stack memory allocation; and the Standard Template Library (STL).

A.2 Basic Structure

First, we compare two simple programs written in Java and C++.

Java Hello World (filename: `HelloWorldApp.java`):

```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello␣World!");
    }
}
```

C++ Hello World (filename: `HelloWorld.cpp`):

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

int main (){
    cout << "Hello␣World!" << endl;
    return 0;
}
```

In Java, the entry point for your program is a method named `main` inside a class. You run that

[1]There are a few things that you can do in C that you can't do in C++, but they are obscure constructions that you will not run into.

particular `main` by invoking the interpreter with that class name. In C++, there is only ONE `main` function. It must be named `main`. It cannot reside within a class. `main` should return an integer – 0 when exiting normally. This points out one major difference, which is C++ can have functions that are not object methods. Main is one of those functions.

The *using* and *include* lines in the C++ program allow us to use external code. This is similar to *import* in Java. In 377, we will generally tell you the necessary include files. For 377, always use the *std* namespace.

We will explain `cout` later.

A.3 Compiling and Running

Java:	**C++:**
`javac HelloWorldApp.java`	`g++ -o HelloWorld HelloWorld.cpp`
`java HelloWorldApp`	`./HelloWorld`

In Java, you produce bytecode using `javac` and run it using the Java just-in-time compiler, `java`. C++ uses a compiler, which creates a stand-alone executable. This executable can only be run on the same platform as it was compiled for. For instance, if you compile HelloWorld for an x86 Linux machine, it will not run on a Mac.

The ./ at the beginning of run line says to run the HelloWorld that is in the current directory. This has nothing to do with C++, but many Unix/Linux systems do not include the current directory in the search path. Also, it makes sure you get the *right* HelloWorld. For instance, there is a Unix program called *test* that can be easily confused with your own program named *test*.

A.4 Intrinsic Types

Java:	C++:
`byte myByte;`	`char myByte;`
`short myShort;`	`short myShort;`
`int myInteger;`	`int myInteger;`
`long myLong;`	`long myLong;`
`float myFloat;`	`float myFloat;`
`double myDouble;`	`double myDouble;`
`char myChar;`	`char myChar;`
`boolean myBoolean;`	`bool myBoolean;`

So the differences between Java and C++ intrinsic types are: C++ does not have a separate type for bytes, just use char; and the boolean type is named bool.

A.5 Conditionals

Java:

```java
boolean temp = true;
boolean temp2 = false;

if (temp)
    System.out.println("Hello␣World!");

if (temp == true)
    System.out.println("Hello␣World!");

if (temp = true) // Assigns temp to be true
    System.out.println("Hello␣World!");
```

C++:

```cpp
bool temp = true;
int i = 1;

if (temp)
    cout << "Hello␣World!" << endl;

if (temp == true)
    cout << "Hello␣World!" << endl;

if (i)
    cout << "Hello␣World!" << endl;
```

Conditionals are almost exactly the same in Java and C++. In C++, conditionals can be applied to integers—anything that is non-zero is true and anything that is zero is false.

NOTE: Be very careful in both C++ and Java with = and ==. In both languages, = does an assignment and == does not – it tests equality. In C++, you can now do this with integers. For instance:

```cpp
if (x = 0){
}
```

sets x to be 0 and evaluates to false. Unless you know what you are doing, always use == in conditionals.

In Java, the double form of the operators gives short-circuiting: && and ||. In C++, it does the same thing.

A.6 Other control flow

For loops, while, and do-while are the same syntax. C++ also has switch statements with the same syntax. Remember break.

A.7 Pointers and Reference Variables

Pointers and reference variables are the most difficult concept in C++ when moving from Java. Every object (and simple data types), in C++ and Java reside in the memory of the machine. The location in memory is called an address. In Java you have no access to that address. You cannot read or change the address of objects. In C++ you can.

In C++, a pointer contains the address of an object or data type. Pointers point to a specified type and are denoted with *.

```
int *ptr;
```

The variable `ptr` is a pointer to an integer. At the moment ptr does not contain anything, it is uninitialized. However, we can find out the address of some integer, using &, and store it in ptr. For instance:

```
int *ptr, *ptr2;
int x = 5;
int y = 4;

ptr = &x;
ptr2 = &y;
```

At the end of that example, ptr contains the address of x, and ptr2 contains the address of y. Additionally we can *dereference* the get the value of what ptr points to:

```
int *ptr;
int x = 5;

ptr = &x;

cout << *ptr << endl;  //prints 5
```

There are other tricky things you can do with pointers, but that is all you should need for 377. It you want another reference on the subject, take a look at: `http://www.codeproject.com/cpp/pointers.asp`.

There is something else in C++ called a reference variable. These are most useful in function arguments discussed later.

A.7.1 Assignment

In C++, the assignment operator works a little different than in Java. For simple data types, it works exactly the same. However, for objects it works differently.

In Java, assignment copies a reference to the object. In C++, it COPIES the object. If you want to copy a reference, then use pointers. For instance, in C++:

```
SomeClass x, y;
SomeClass *a;

x=y; // This copies object y to x.  Modifying x does NOT modify y.

a=&x;  // This copies a reference to x.
       // Modifying the object a points to modifies x.
```

A.7.2 Object Instantiation

In Java, if you declare and instantiate an object like this:

```
SomeClass x;

x = new SomeClass();
```

In C++, if you declare an object, it instantiates it for you:

```
SomeClass x;
```

A.7.3 The $->$ Operator

For pointers to objects you can call methods and modify variables like so:

```
SomeClass x;
SomeClass *a;

a=&x;
(*a).SomeMethod();
```

So we have dereferenced a and called its SomeMethod. This is ugly. We can use the $->$ operator to do the same thing:

```
SomeClass x;
SomeClass *a;

a=&x;
a->SomeMethod();
```

Pretty, huh?

A.8 Global Variables, Functions and Parameters

In C++, you can have variables that are not members of objects. That is called a global variable. That variable is accessible from any function or object in the same source file. **NOTE:** Global variables are generally considered poor programming form. However, in this class I will grant you some leeway in using them. Do not abuse this. I consider making loop control variables global a serious abuse.

Similar to global variables, there are functions that are not methods. For instance main is not a method of any object. Similarly you can create new functions that are not contained in objects. These functions can take parameters just like methods. For instance:

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

void foo(int i){
  cout << i << endl; // Prints 1
}
```

```
int main (){
  foo (1);

  return 0;
}
```

Similar to Java, C++ functions can return nothing (void), simple data types, or objects. If you return an object in C++, you are returning a COPY of that object, not a reference. You can return a pointer to an object, if you want to return a reference.

But here is where Java and C++ have one major difference: parameters. In Java simple data types are passed by value (a copy), and objects are passed by reference. In C++ both simple data types and objects are passed by value![2] However, functions would be fairly useless in C++ if we couldn't change what we passed. That is where pointers come in. Quite simply:

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

void foo(int *i){
  *i = 6;
}

void bar(int i){
  i = 10;
}

int main (){
  int i = 0;

  foo (&i);
  cout << i << endl;  // prints 6

  bar (i);
  cout << i << endl;  // prints 6

  bar (&i);  // WOULD NOT COMPILE

  return 0;
}
```

In the above example, we have a function foo that takes a pointer to an integer as a parameter. When we call foo, we have to pass it the address of an integer. So foo is modifying the same i as main. The function bar takes i by value and any changes made are lost. The last call to bar attempts to call bar with an address to an integer, not an integer, and would not compile.

Ugly, huh? In C++, there is a slightly simpler way of accomplishing the exact same thing:

```
#include <iostream>
```

[2] This is more consistent than Java, and one of the things that C# fixes. C# passes EVERYTHING by reference including simple data types.

```
using namespace std;

void foo(int &i){
  i = 6;
}

int main (){
  int i = 0;

  foo (i);
  cout << i << endl;  // prints 6.

  return 0;
}
```

The integer i in foo is a reference variable. It takes care of passing the address, and dereferencing i when it is used in foo. This is cleaner and easier to understand, but both are valid. However, you cannot avoid the uglier form. If I give you a function to call then it will be of the first form.[3]

A.9 Arrays

Java and C++ both have arrays. Indexes start at 0, and you index into them using [ ]. However, arrays behave a little differently in C++. First, the elements of the array do not need to be allocated with new, C++ allocates them for you. For instance, if you create an array of Objects, it is an array of Objects, not an array of references like in Java. Second, C++ arrays do not know how large they are. You have to keep track of this yourself. C++ will not stop you from going past the end or beginning of the array. This is a programming error and will often crash your program. Third, if you refer to an array without a subscript, C++ treats this is a pointer to the first element. For instance, in this program we pass a pointer to qux to foo, which modifies the contents of the first element.

```
#include <iostream>

using namespace std;

void foo (int bar[]){
  bar[0] = 5;
}

int main(){

  int qux[10];

  qux[0] = 10;
  foo (qux);
```

[3]This maintains compatibility with C.

```
  cout << qux[0] << endl;  // Prints 5
}
```

You can also create an array of pointers.

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

void foo (int* bar[]){
  *(bar[0]) = 5;
}

int main(){
  int* qux[10];

  qux[0] = new int;
  *(qux[0]) = 10;
  foo(qux);

  cout << *(qux[0]) << endl;  // Prints 5
}
```

A.10 Structs and Classes

C++ has something called a struct. Think of it as a class with no methods, only public member variables. You can refer to the variables using '.'. If you hold a pointer to a struct, you can use the $->$ operator. For instance:

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

struct foo{
  int a;
};

int main (){
  foo b, *c; // b is a struct, c points to a struct

  b.a = 6;
  c = &b;
  c->a = 7; // Remember c points to the struct b!

  cout << b.a << endl;  // prints 7.

  return 0;
}
```

As for classes, the syntax is a little different, but many things are the same.

In Java, you have:

```
public class IntCell
{
  public IntCell( )
     { this(0); }

  public IntCell( int initialValue )
     { storedValue = initialValue; }

  public int getValue( )
     { return storedValue; }

  public int setValue( int val )
     { storedValue = val; }

  private int storedValue;
}
```

and in C++, you have:

```
class IntCell
{
  public:

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

  int getValue( )
     { return storedValue; }

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

  private:

  int storedValue;
};
```

The end of a C++ class has a semicolon. DO NOT FORGET IT. The compilation errors will not tell you it is missing. The compiler will give you strange errors you don't understand.

In C++, there is no visibility specifier for the class.[4]

In C++, public and private are applied to sections inside the class.[5]. In Java, one constructor can call another. You can't do this in C++. The above syntax says that the default value is 0 if one is not passed. This must be a fixed, compile time value.

There is also a slightly nicer way to declare classes in C++. This is exactly the same as the previous definition, we have just split the interface and the implementation.

[4] This only applies to a top-level class. See next section on inheritance.
[5] C++ also has protected, see the next section on inheritance

```
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;
}
```

A.11 Operator Overloading, Inheritance

Yep, C++ has both. Do you need them for 377? Nope.

A.12 Stack and Heap Memory Allocation

This is the second most difficult thing to get a handle on besides pointers.

A.12.1 Stack Memory

In a C++ function, local variables and parameters are allocated on the stack. The contents of the stack is FREED (read destroyed) when the function ends. For instance in this example:

```
int foo (int a){
  int b = 10;

  return 0;
}
```

After foo ends you can't access a or b anymore. This shouldn't surprise you at all, Java works the same way. However, now that we have pointers you can do something really bad:

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

// BAD BAD BAD

int* foo (){
  int b = 10;

  return &b;
}

int main(){
  int *a;

  a = foo();

  cout << *a << endl;  // Print out 10?

  return 0;
}
```

In this example, we have a function foo that returns a pointer to a stack allocated variable b. However, remember when foo returns, the memory location of b is freed. We try to dereference the pointer to b in the cout statement. Dereferencing a *stale* pointer to freed memory, makes your program die a horrible nasty death.[6]

A.12.2 Heap Memory

However, what if we want a variable to live on after the end of a function? There are two ways to do it. First is to make it global. Then it will live for the entire lifetime of the program. This is poor programming practice as well as not being dynamic. (How many objects do you need?)

Instead we allocate from the heap using new, just like in Java. However, new returns a pointer (a reference, just like in Java!). For instance:

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

// GOOD GOOD GOOD

int* foo (){
  int *b;

  b = new (int);
```

[6] Actually if you compile this program and run it, it will probably work. However, in a more complex program it will NOT

```
  *b = 10;

  return b;
}

int main(){

  int *a;

  a = foo();

  cout << *a << endl;  // Print out 10!

  return 0;
}
```

Now b is allocated on the heap and lives forever! Great, but there is one slight problem in C++. Java knows when you are no longer using an object and frees it from the heap[7]. C++ does not so you have to tell it when you are done. For instance in Java this is legal:

```
  bar qux;

  for (int i=0; i < 1000000; i++)
      qux = new bar();
```

Well it is legal in C++ too, but you have what is called a *memory leak*. You are allocating objects and never freeing them. In C++ you must do this:

```
  bar *qux;

  for (int i=0; i < 1000000; i++){
      qux = new (bar);
      delete (qux);
  }
```

Easy, right? Ok, just remember that for **EVERY** new, you must do a delete exactly once. If you delete memory that has already been deleted once, you are deleting a stale pointer. Again, your program will die a horrible, nasty death. One strategy to diagnose this is to comment out your calls to delete and see if your program still works (but sucks up memory). Another good idea is to set pointers to NULL after you delete them and always check that the pointer is NOT NULL before calling delete.

[7]This is called garbage collection

A.13 Input, Output, Command Line Parameters

A.13.1 Input

Say you want to read something from a file.

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

int main(){

  int a;
  ifstream input;

  input.open("myfile");
  while (input >> a);
  input.close();
  input.clear();

  return 0;
}
```

This allows you to read a bunch of integers from myfile. You only need clear if you intend to use the input object again. We will give you most of the code you need for input.

A.13.2 Output

Say you want to print two integers. Very easy:

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

int main(){

  int a = 5, b = 6;

  cout << a << "␣" << b << endl; // Prints 5 6

  return 0;
}
```

The endl is an end-of-line character.

A.13.3 Command line parameters

Just follow the example:

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

int main(int argc, char * argv[]){

  int a;
  ifstream input;

  if (argc != 3){ // One for the program name and two parameters
     cout << "Invalid␣usage" << endl;
     return -1;
  }

  int a = atoi (argv[1]);  // First parameter is a number
                           // atoi converts from characters to int

  input.open(argv[2]);  /// Second parameter is the name of a file
  input.close();

  return 0;
}
```

A.14 The Standard Template Library

As you know already, it is tough to program without good data structures. Fortunately, most C++ implementations have them built in! 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. I will make this clearer in class. 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. If you know, you are a C++ guru![8]

A.15 Crashes, Array Bounds, Debugging

C++ programs can crash for a lot of reasons. You will run your program and it will print out some strange message, such as "segfault" or "bus error". This means you have done something wrong in your program. The three most common errors are: accessing an array out of bounds, dereferencing a stale pointer, and memory allocation errors (such as deleting a stale pointer).

There is a tool named `gdb` that can help you diagnose some of these problems. The TAs will show you how to use it in office hours.

A.16 Misc

A.16.1 Assert

The assert function is a handy call in C++. If you assert something that is true, nothing will happen. If you assert something that is false your program will exit on that line and tell you it wasn't true.

```
int *ptr = NULL;

assert(ptr != NULL); // Program exits
```

A.16.2 Typedef

Typedef is just some way of defining one type as another.

```
typedef int* p_int;

p_int a;  // a is pointer to an integer
```

[8]Ok, maybe not a guru, but you can probably pass 377.

A.16.3 Exceptions

Yes, C++ has exceptions kind of like Java. Do you need them in 377? Nope.

A.16.4 Comments

Same syntax as Java.

A.16.5 #define

You can can globally replace a set of characters with another[9].

```
#define MAX 100

int baz[MAX]; // baz is an array of 100 integers
```

A.16.6 static

The keyword static means something in Java classes. It means the same thing in C++ (with some differences). Don't worry about using it in classes, you won't need to in 377.

However, it also means several other things in C++. If you put static before a global variable or function, it means it is only visible in this file. You will only be writing programs with one file, but you will be including files the instructors wrote. To ensure there are no names that conflict, we ask that you declare all your functions (except main) and global variables with static.

If you declare a variable inside a function to be static, it is just like a global variable, but only visible inside the function. In other words it keeps its value between calls to that function.[10]

Here is an example of both:

```
static int b;

static void foo(int a){
  static int x;
}
```

A.16.7 unsigned variables

There are other datatypes, such as "unsigned int". It is a purely positive integer (i.e. no twos-complement). Don't do this:

[9] This is done before compilation by the preprocessor

[10] This is because static variables are allocated in the data segment, not the stack.

```
void foo(unsigned int a){
  cout << a << endl;
}

int main(){

  int a = -1;

  foo(a);
}
```

Actually the compiler will stop you from doing this. You are trying to give it the wrong type.

A.16.8 typecasting

However, there is something you can do about it. You can change any type into any other type. As you can tell this is more dangerous than running with scissors. For instance, this WILL compile:

```
void foo(unsigned int a){
  cout << a << endl;
}

int main(){

  int a = -1;

  foo((unsigned int)a);
}
```

The int in parentheses says make this an unsigned int.

You can also do this:

```
int main(){

  int *a;
  float b;

  a = (int *)&b;  // Take the address of b and pretend it is a pointer to an int
}
```

Don't.