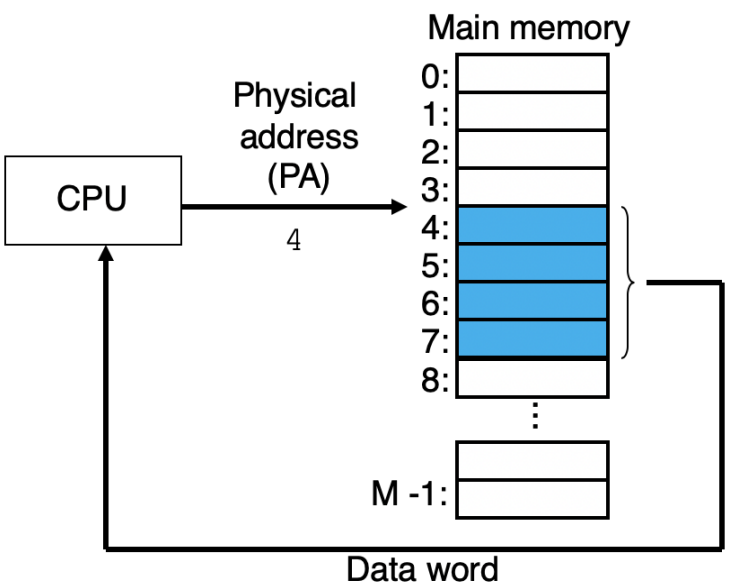
CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

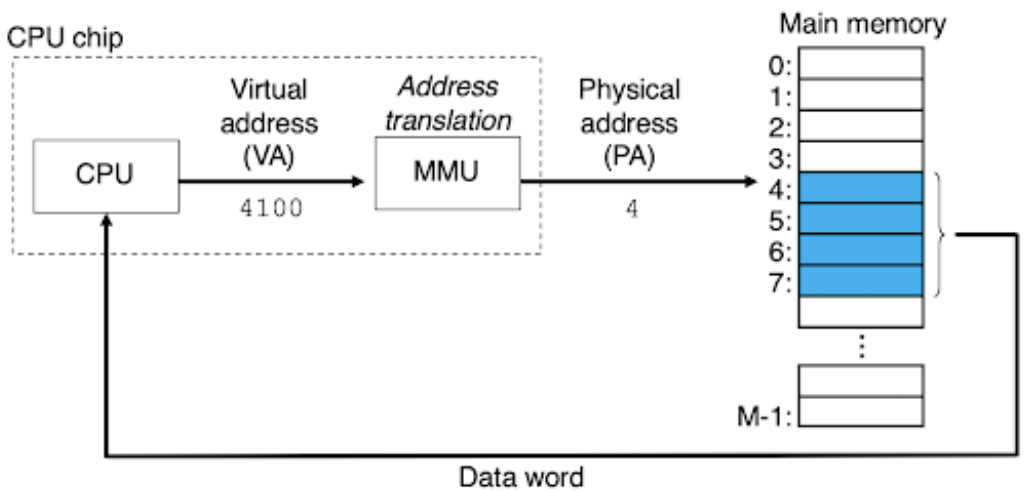
Lecture 12: Virtual Memory

- Main memory (or rather, RAM, which is short for random access memory) is organized as an array of contiguous bytes. It's physical hardware where the operating system and all active user processes store relevant code and data in order to run more efficiently.
  - Each byte has its own address, and addresses range from 0 (the zero address) through some number M - 1, where M is typically a large power of 2.
  - Older machines rely on **physical addressing**, which means addresses generated and dereferenced by the OS and user processes are trivially mapped to the data residing at that same address.
  - Physical addressing is the standard addressing scheme when operating systems only permit one process to exist at any one time.
    - Stated another way: a process more or less owns all of memory while it's running and, in principle, can read to or write from any address. That's physical addressing!



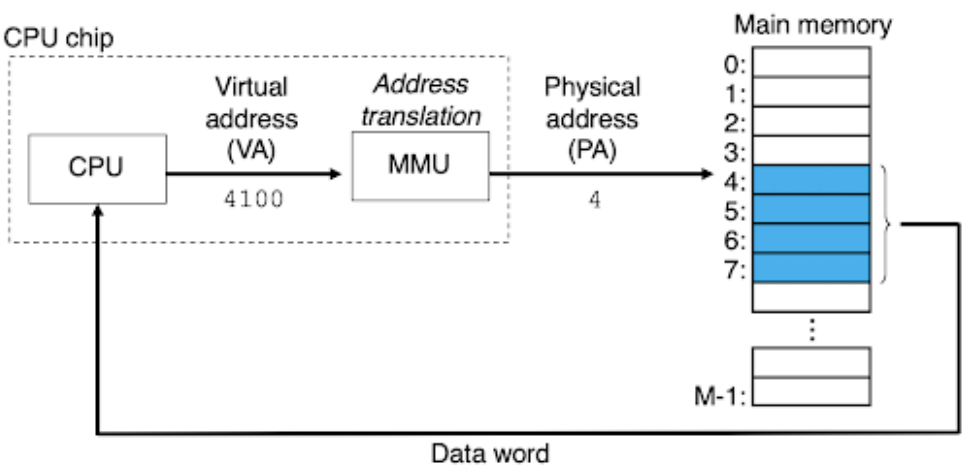
Lecture 12: Virtual Memory

- Modern devices—laptops, desktops, iPhones and Androids, etc.—allow **multiple** processes to run at the same time. If two or more processes operate as if they own all of memory, then physical addressing is simply not possible.
  - The simplest solution is to allow all processes to operate as if they own all of memory **anyway** and require the CPU and the OS to jointly treat a process's addresses as virtual, and to convert every virtual address to a physical one—a computation that's 100% invisible to the process—before accessing RAM.
  - The CPU and the OS use specialized hardware called a **memory management unit** (MMU) to manage translations between virtual addresses (the addresses computed by the process) and physical ones (as generated by the MMU).

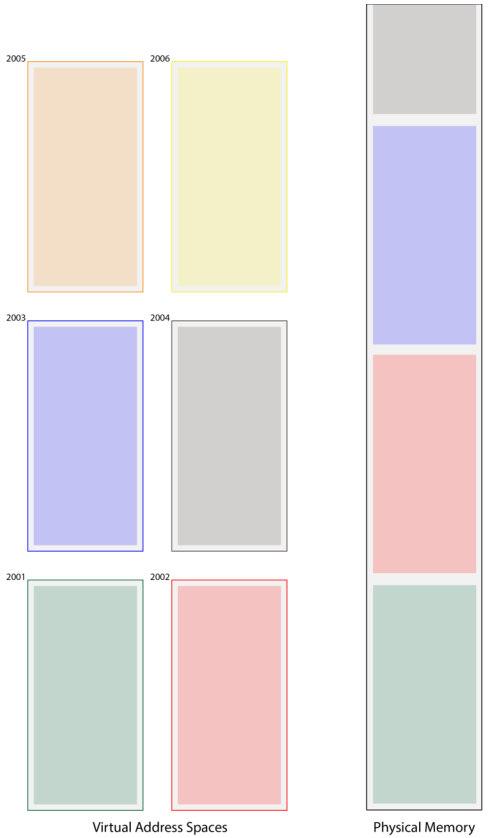


Lecture 12: Virtual Memory

- We'll define an **address space** as an ordered sequence integer addresses, starting at 0. Main memory defines a physical address space, but the address space that a process **thinks** it's accessing is virtual.
- The size of a physical address space is dictated by the hardware. If you have 16GB of RAM, physical addresses range from 0 through 2^34 - 1.
- The size of a virtual address space is dictated by the size of a pointer. The myth machines, along with virtually all modern desktops and laptops, are 64-bit, which means that pointers are 64 bits wide. That means processes can distinguish between 2^64 different addresses and their virtual address spaces are 2^64 bytes.
- The MMU converts each (virtual address, pid) pair to a physical address in main memory, and the operating system ensures that no two virtual addresses ever map to the same physical address.
- If all process addresses are virtual, the OS can map those addresses to any physical storage device.



Lecture 12: Virtual Memory



- One approach is to assume an infinite amount of physical memory, and map each virtual address space to an equally large, contiguous portion of physical memory.
  - Virtual memory addresses would be 64-bits, but physical addresses would be even wider.
  - The OS could maintain a data structure—a map or a dictionary—that maps pids to some multiple of 2^64. In the context of the diagram on the right:
    - pid 2001 maps to 0x0000
    - pid 2002 maps to 0x0001
    - pid 2003 maps to 0x0002, etc
  - The OS and the MMU would jointly convert virtual addresses to physical ones by prepending that multiple to the left, e.g. 0x7FFFFFFF80004000 in process 2004's space would be reliably and deterministically converted to 0x00037FFFFFFF80004000.
  - This translation process is delightfully simple and farcically naive.
    - We don't have an infinite amount of physical memory. In fact, no computer I know of has anywhere near 2^64 = 18.4 exabytes.
    - The vast majority of any single virtual address space is unused. The accumulation of bytes needed for all segments is typically on the order of MBs.

Lecture 12: Virtual Memory

- The OS shouldn't try to map virtual addresses that will never be used. Instead, it should only map virtual addresses that are part of segments.
- We could relax the requirement that segments within any single process be clustered in physical memory.
- Doing so would allow the OS more flexibility in how physical memory is used to back full segments in virtual memory.
- Small segments in new processes might be mapped to open pockets within physical memory that couldn't be used previously.

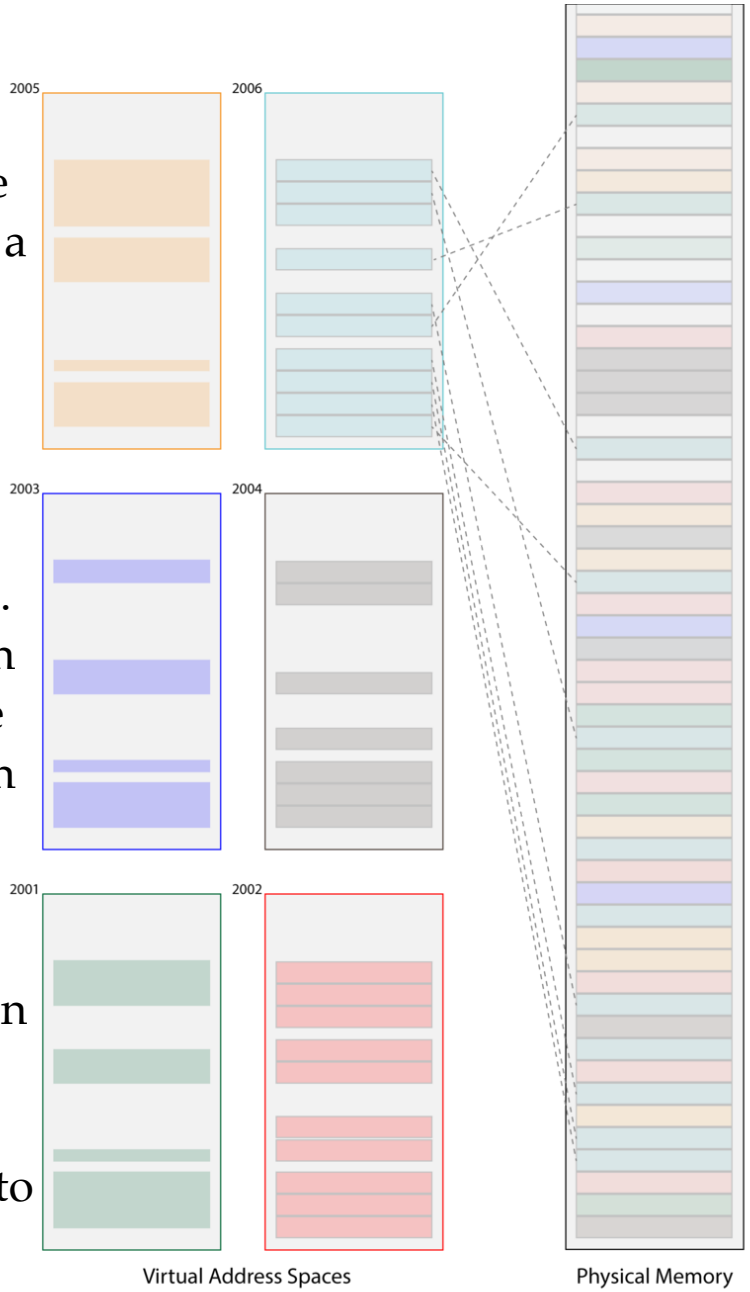
Lecture 12: Virtual Memory

- The huge win with this approach is that only virtual addresses that contribute to a process's runtime get mapped. The physical memory requirements are much smaller and more realistic.

- This approach is better, but far from perfect.
  - Address translations are more complicated and time consuming.
  - The heap and stack segments often grow over time and might need to be moved, and moved often.
  - If variably sized regions of physical memory are allocated and deallocated, over time, physical memory may become fragmented and interfere with the OS's ability to map large virtual segments to physical memory.

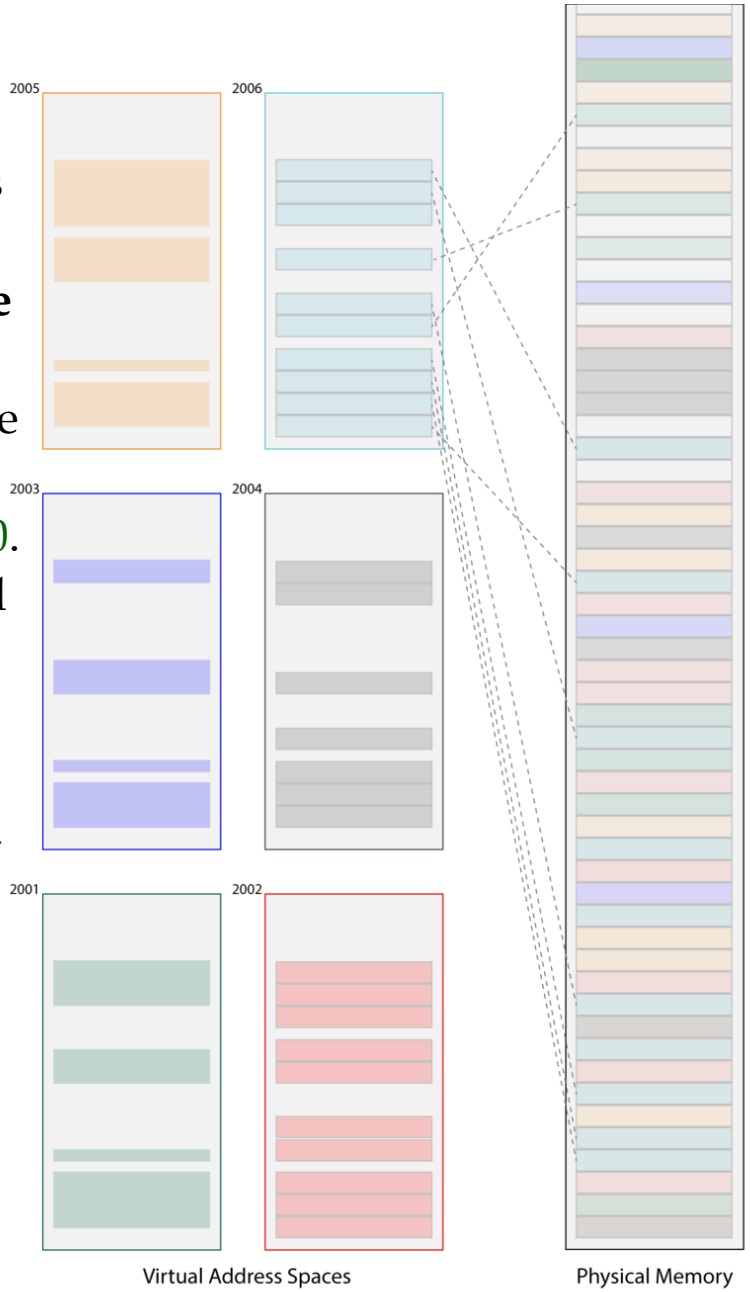
Lecture 12: Virtual Memory

- Support for virtual memory is provided through a scheme very similar to that for file systems, though the unit of storage is called a page instead of a block, and it is typically 4096 bytes (or some other power of two, but we'll assume 2^12 = 4096 here).
- Virtual memory segments are actually contiguous sequences of virtual pages. The OS maps each virtual page to a physical one. The physical pages themselves can be drawn from any portion of physical memory. Since all pages are the same size, any free page can be allocated to back a new virtual page of memory.
- The OS stores a **page table** of virtual-to-physical page mappings in the kernel portion of a process's virtual address space.
- The OS takes measures to ensure the virtual addresses used by the page table itself map to well-known regions of physical memory.



Lecture 12: Virtual Memory

- Without going into the weeds on implementation details, the page table maps virtual page addresses to physical page addresses. Each mapping is stored in a **page table entry**.
- Because all pages—virtual and physical—are aligned on 4KB boundaries, the base addresses of every single page ends in 0x000. Similarly, the offset of a byte within a virtual page is backed by a byte at the same offset within the physical page.
- If the OS maps the virtual page with base address of, say, 0x7FFFFFFF80234000 to, say, 0x835432000, the table entry would store a 52-bit 7FFFFFFF80234 as the key and the 24-bit 835432 as the value. There's no need to store the 3 trailing zeroes, because they're implied by the 4K-page alignment constraints.



Lecture 12: Virtual Memory

- The page table stored on behalf of a single process might look like this:

7FFFFFFF80234 maps to 835432
0000DDEE65111 maps to 45D834

many additional entries

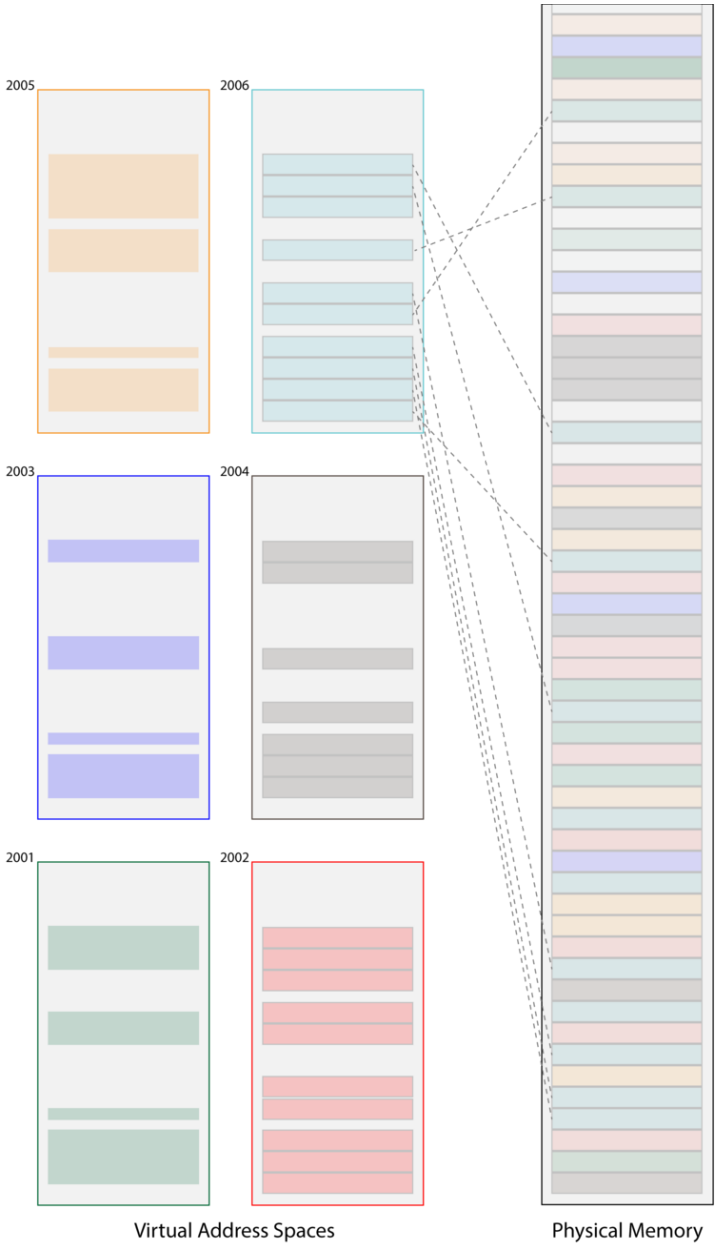
0000A9CF9AAAF maps to 12387B

- The address translation is managed by the MMU on the CPU, and the translation itself can be computed via bitwise operations like <<, &, |, and >>.

0x7FFFFFFF80234230 & 0xFFF = 0x230

0x230 | (0x835432 << 12) = 0x835432230

- The MMU is optimized to do these computations blazingly fast.



Lecture 12: Virtual Memory

- Many, including the authors of your primary textbook/reader, view the **hard drive** as the physical memory and **main memory** as a **cache** storing those pages from disk currently being accessed. This viewpoint has many advantages!
  - Executables (e.g. **ls**, **pipeline-test**, etc) are stored on disk and loaded into main memory when a process is created to run that executable.
  - As a general rule, every single virtual page of memory uniquely maps to a physical page of memory. However, Linux often ignores that rule for code and rodata segments, since code and global constants are read only.
    - In particular, multiple processes running **emacs**—there were 17 such processes running on **myth63** at the time I constructed this slide—all map their code segment to the same pages in physical memory.
  - As main memory becomes saturated, the OS will **evict** pages that have become inactive and flush them to the hard drive, often in a special file called a **swap file**.
    - For instance, at the time of this writing, **myth53** had one process running **vi** that had been idle for 5 days.
    - The VM page mappings have almost certainly been discarded and will only be remapped when the user interacts with that **vi** session again. The state of memory needed to eventually continue would have, at some point, been written to disk.

Lecture 12: Gentle Introduction to Threads

- A **thread** is an independent execution sequence within a single process.
  - Operating systems and programming languages generally allow processes to run two or more functions simultaneously via **threading**.
  - The process's stack segment is essentially subdivided into multiple miniature stacks, one for each thread.
  - The thread manager time slices and switches between threads in much the same way that the OS scheduler switches between processes. In fact, threads are often called **lightweight processes**.
  - Each thread maintains its own stack, but shares the same text, data, and heap segments with other threads within the same process.
    - Pro: it's easier to support communication between threads, because they run in the same virtual address space.
    - Con: there's no memory protection, since the virtual address space is shared. Race conditions and deadlock threats need to be mitigated, and debugging can be difficult. Many bugs are hard to reproduce, since thread scheduling isn't predictable.
    - Pro **and** con: Multiple threads can access the same global variables.
    - Pro **and** con: Each thread can share its stack space (via pointers) with its peer threads.

Lecture 12: Gentle Introduction to Threads

- Threading and Introverts
  - C++ provides support for threading and many synchronization directives.
  - Presented below is a short C++ **introverts** example we've already seen once before. The full program is online right here.
  - Once initialized, each of the six **recharge** threads is eligible for processor time.
  - These six threads compete for CPU time in much the same way that processes do, and we have very little control over what choices the scheduler makes when deciding which thread to run next.
  - Hot take: processes are are actually implemented as threads.

```
static void recharge() {
  cout << oslock << "I recharge by spending time alone." << endl << osunlock;
}

static const size_t kNumIntroverts = 6;
int main(int argc, char *argv[]) {
  cout << "Let's hear from " << kNumIntroverts << " introverts." << endl
  thread introverts[kNumIntroverts]; // declare array of empty thread handles
  for (thread& introvert: introverts)
     introvert = thread(recharge);    // move anonymous threads into empty handles
  for (thread& introvert: introverts)
     introvert.join();
  cout << "Everyone's recharged!" << endl;
  return 0;
}
```

Lecture 12: Gentle Introduction to Threads

- The primary data type is the **thread**, which is a C++ class used to manage the execution of a function within its own thread of execution.
- We *install* the **recharge** function into temporary threads objects that are then moved (via **thread**::**operator=(thread&& other)**) into a previously empty **thread** object.
  - This is a relatively new form of **operator=** that fully transplants the contents of the **thread** on the right into the **thread** on the left, leaving the **thread** on the right fully gutted, as if it were zero-arg constructed. Restated, the left and right thread objects are effectively swapped.
- The **join** method is to threads what **waitpid** is to processes.
- Because **main** calls **join** against all threads, it blocks until all child threads all exit.
- The prototype of the thread routine—in this case, **recharge**—can be anything (although the return type is always ignored, so it should generally be **void**).
- **operator<<**, unlike **printf**, isn't thread-safe.
  - I've constructed custom stream manipulators called **oslock** and **osunlock** that can be used to acquire and release exclusive access to an **ostream**.
  - These manipulators—which we get by #**include**-ing **"ostreamlock.h"**—whelp ensure only one thread gets permission to insert into a stream at any one time.

Lecture 12: Gentle Introduction to Threads

- Thread routines can accept any number of arguments using variable argument lists. (Variable argument lists—the C++ equivalent of the ellipsis in C—are supported via a recently added feature called variadic templates.)
- Here's a slightly more involved example, where **greet** threads are configured to say hello a variable number of times.

```
static void greet(size_t id) {
  for (size_t i = 0; i < id; i++) {
    cout << oslock << "Greeter #" << id << " says 'Hello!'" << endl << osunlock;
    struct timespec ts = {
      0, random() % 1000000000
    };
    nanosleep(&ts, NULL);
  }
  cout << oslock << "Greeter #" << id << " has issued all of his hellos, "
       << "so he goes home!" << endl << osunlock;
}

static const size_t kNumGreeters = 6;
int main(int argc, char *argv[]) {
  cout << "Welcome to Greetland!" << endl;
  thread greeters[kNumGreeters];
  for (size_t i = 0; i < kNumGreeters; i++) greeters[i] = thread(greet, i + 1);
  for (thread& greeter: greeters) greeter.join();
  cout << "Everyone's all greeted out!" << endl;
  return 0;
}
```