CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Multithreading, Semaphores, and Rendezvous

- New concurrency pattern!
  - **semaphore::wait** and **semaphore::signal** can be leveraged to support a different form of communication: **thread rendezvous**.
  - Thread rendezvous is a generalization of **thread::join**. It allows one thread to stall—via **semaphore::wait**—until another thread calls **semaphore::signal**, often because the signaling thread just prepared some data that the waiting thread needs before it can continue.

- To illustrate when thread rendezvous is useful, we'll implement a simple program without it, and see how thread rendezvous can be used to repair some of its problems.
  - The program has two meaningful threads of execution: one thread publishes content to a shared buffer, and a second reads content as it becomes available.
  - The program is a nod to the communication in place between a web server and a browser. The server publishes content over a dedicated communication channel, and the browser consumes it.
  - The program also reminds me of how two independent processes behave when one writes to a pipe, a second reads from it, and how the write and read processes behave when the pipe is full (in principle, a possibility) or empty.

Multithreading, Semaphores, and Rendezvous

- Consider the following program, where concurrency directives have been intentionally omitted. The full, very buggy example is right here.

```
static void writer(char buffer[]) {
  cout << oslock << "Writer: ready to write." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length
    char ch = prepareData();
    buffer[i % 8] = ch;
    cout << oslock << "Writer: published data packet with character '" << ch << "'." << endl
  }
}

static void reader(char buffer[]) {
  cout << oslock << "\t\tReader: ready to read." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length
    char ch = buffer[i % 8];
    processData(ch);
    cout << oslock << "\t\tReader: consumed data packet " << "with character '" << ch << "'."
  }
}

int main(int argc, const char *argv[]) {
  char buffer[8];
  thread w(writer, buffer);
  thread r(reader, buffer);
  w.join();
  r.join();
  return 0;
}
```

Multithreading, Semaphores, and Rendezvous

- Here's what works:
  - Because the **main** thread declares a circular buffer and shares it with both children, the children each agree where content is stored.
  - Think of the buffer as the state maintained by the implementation of **pipe**, or the state maintained by an internet connection between a server and a client.
  - The **writer** thread publishes content to the circular buffer, and the **reader** thread consumes that same content as it's written. Each thread cycles through the buffer the same number of times, and they both agree that **i % 8** identifies the next slot of interest.
- Here's what's broken:
  - Each thread runs more or less independently of the other, without consulting the other to see how much progress it's made.
  - In particular, there's nothing in place to inform the **reader** that the slot it wants to read from has meaningful data in it. It's possible the writer just hasn't gotten that far yet.
  - Similarly, there's nothing preventing the **writer** from advancing so far ahead that it begins to overwrite content that has yet to be consumed by the **reader**.

Multithreading, Semaphores, and Rendezvous

- One solution? Maintain two **semaphore**s.
  - One can track the number of slots that can be written to without clobbering yet-to-be-consumed data. We'll call it **emptyBuffers**, and we'll initialize it to 8.
  - A second can track the number of slots that contain yet-to-be-consumed data that can be safely read. We'll call it **fullBuffers**, and we'll initialize it to 0.
- Here's the new **main** program that declares, initializes, and shares the two **semaphore**s.

```
int main(int argc, const char *argv[]) {
  char buffer[8];
  semaphore fullBuffers, emptyBuffers(8);
  thread w(writer, buffer, ref(fullBuffers), ref(emptyBuffers));
  thread r(reader, buffer, ref(fullBuffers), ref(emptyBuffers));
  w.join();
  r.join();
  return 0;
}
```

- The **writer** thread waits until at least one buffer is empty before writing. Once it writes, it'll increment the full buffer count by one.
- The **reader** thread waits until at least one buffer is full before reading. Once it reads, it increments the empty buffer count by one.

Multithreading, Semaphores, and Rendezvous

- Here are the two new thread routines:

```
static void writer(char buffer[], semaphore& full, semaphore& empty) {
  cout << oslock << "Writer: ready to write." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length
    char ch = prepareData();
    empty.wait();   // don't try to write to a slot unless you know it's empty
    buffer[i % 8] = ch;
    full.signal();  // signal reader there's more stuff to read
    cout << oslock << "Writer: published data packet with character '" << ch << "'." << endl
  }
}

static void reader(char buffer[], semaphore& full, semaphore& empty) {
  cout << oslock << "\t\tReader: ready to read." << endl << osunlock;
  for (size_t i = 0; i < 320; i++) { // 320 is 40 cycles around the circular buffer of length
    full.wait();    // don't try to read from a slot unless you know it's full
    char ch = buffer[i % 8];
    empty.signal(); // signal writer there's a slot that can receive data
    processData(ch);
    cout << oslock << "\t\tReader: consumed data packet " << "with character '" << ch << "'."
  }
}
```

- The reader and writer rely on these **semaphore**s to inform the other how much work they can do before being necessarily forced off the CPU.
- Thought question: can we rely on just one **semaphore** instead of two? Why or why not?