CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Multithreading and Dining Philosophers

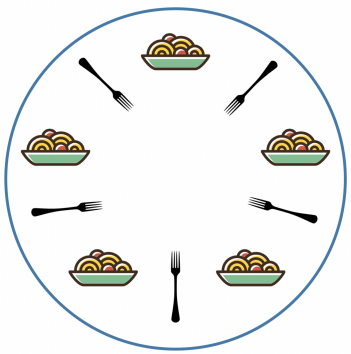
- The Dining Philosophers Problem
  - This is a canonical multithreading example used to illustrate the potential for deadlock and how to avoid it.
    - Five philosophers sit around a table, each in front of a big plate of spaghetti.
    - A single fork (utensil, not system call) is placed between neighboring philosophers.
    - Each philosopher comes to the table to think, eat, think, eat, think, and eat. That's three square meals of spaghetti after three extended think sessions.
    - Each philosopher keeps to himself as he thinks. Sometime he thinks for a long time, and sometimes he barely thinks at all.
    - After each philosopher has thought for a while, he proceeds to eat one of his three daily meals. In order to eat, he must grab hold of two forks—one on his left, then one on his right. With two forks in hand, he chows on spaghetti to nourish his big, philosophizing brain. When he's full, he puts down the forks in the same order he picked them up and returns to thinking for a while.
  - The next two slides present the core of our first stab at the program that codes to this problem description. (The full program is right here.)

Multithreading and Dining Philosophers

- The Dining Philosophers Problem
  - The program models each of the forks as a **mutex**, and each philosopher either holds a fork or doesn't. By modeling the fork as a **mutex**, we can rely on **mutex::lock** to model a thread-safe grab and **mutex::unlock** to model a thread-safe release.

```
static void philosopher(size_t id, mutex& left, mutex& right) {
  for (size_t i = 0; i < 3; i++) {
    think(id);
    eat(id, left, right);
  }
}

int main(int argc, const char *argv[]) {
  mutex forks[5];
  thread philosophers[5];
  for (size_t i = 0; i < 5; i++) {
    mutex& left = forks[i], & right = forks[(i + 1) % 5];
    philosophers[i] = thread(philosopher, i, ref(left), ref(right));
  }
  for (thread& p: philosophers) p.join();
  return 0;
}
```



images courtesy of Roz Cyrus

Multithreading and Dining Philosophers

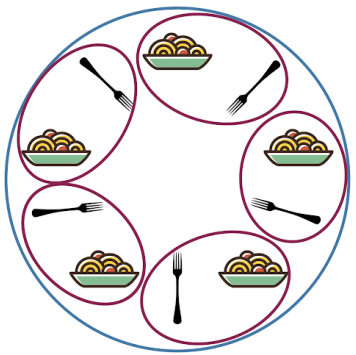
- The Dining Philosophers Problem
  - The implementation of **think** is straightforward. It's designed to emulate the time a philosopher spends thinking without interacting with forks or other philosophers.
  - The implementation of **eat** is almost as straightforward, provided you understand the thread subroutine is being fed references to the two forks he needs to acquire if he's permitted to eat.

```
static void think(size_t id) {
  cout << oslock << id << " starts thinking." << endl << osunlock;
  sleep_for(getThinkTime());
  cout << oslock << id << " all done thinking. " << endl << osunlock;
}

static void eat(size_t id, mutex& left, mutex& right) {
  left.lock();
  right.lock();
  cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
  sleep_for(getEatTime());
  cout << oslock << id << " all done eating." << endl << osunlock;
  left.unlock();
  right.unlock();
}
```

Multithreading, Dining Philosophers, Deadlock

- The program appears to work well (we'll run it several times), but it doesn't guard against this: each philosopher emerges from deep thought, successfully grabs the fork to his left, and is then forced off the processor because his time slice is up.
- If all five philosopher threads are subjected to the same scheduling pattern, each would be stuck waiting for a second fork to become available. That's a real deadlock threat.
- Deadlock is more or less guaranteed if we insert a **sleep_for** call in between the two calls to **lock**, as we have in the version of **eat** presented below.
  - *We should be able to insert a **sleep_for** call anywhere in a thread routine and not introduce deadlock. If a gratuitous call to **sleep_for** introduces any form of deadlock, then you have a problem to be solved.*



```
static void eat(size_t id, mutex& left, mutex& right) {
  left.lock();
  sleep_for(5000);  // artificially force off the processor
  right.lock();
  cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
  sleep_for(getEatTime());
  cout << oslock << id << " all done eating." << endl << osunlock;
  left.unlock();
  right.unlock();
}
```

Multithreading, Dining Philosophers, Deadlock

- When coding with threads, you need to ensure that:
  - there are no race conditions, even if they rarely cause problems, and
  - there's zero threat of deadlock, lest a subset of threads forever starve for processor time.
- **mutex**es are generally the solution to race conditions, as we've seen with the ticket agent example. We can use them to mark the boundaries of critical regions and limit the number of threads present within them to be at most one.
- Deadlock can be programmatically prevented by implanting directives to limit the number of threads competing for a shared resource, like, you know, utensils.
  - We could, for instance, recognize that it's impossible for three philosophers to be eating at the same time. That means we could limit the number of philosophers who have permission to grab forks to a mere 2.
  - We could also argue it's okay to let four—though certainly not all five—philosophers grab forks, knowing that at least one will successfully grab both.
    - My personal preference? Impose a limit of four.
    - My rationale? Implant the **minimal** amount of bottlenecking needed to remove the threat of deadlock, and trust the OS and thread manager to otherwise make good choices.

Multithreading, Dining Philosophers, Deadlock Removal

- Here's the core of a program that limits the number of philosophers grabbing forks to four. (The full program can be found right here.)
  - I impose this limit by introducing the notion of a permission slip / permit. Before grabbing forks, a philosopher must first acquire one of four permission slips.
  - These permission slips need to be acquired and released without race condition.
  - For now, I'll model a permit using a counter—I call it **permits**—and a companion **mutex**—I call it **permitsLock**—that must be acquired before examining or changing **permits**.

```
int main(int argc, const char *argv[]) {
  size_t permits = 4;
  mutex forks[5], permitsLock;
  thread philosophers[5];
  for (size_t i = 0; i < 5; i++) {
    mutex& left = forks[i],
         & right = forks[(i + 1) % 5];
    philosophers[i] =
        thread(philosopher, i, ref(left), ref(right), ref(permits), ref(permitsLock));
  }
  for (thread& p: philosophers) p.join();
  return 0;
}
```

Multithreading, Dining Philosophers, Deadlock Removal

- The implementation of **think** is the same, so I don't present it again.
- The implementation of **eat**, however, changes a bit.
  - It accepts two additional references: one to the number of available **permits**, and a second to the **mutex** used to guard against simultaneous access to **permits**.

```
static void eat(size_t id, mutex& left, mutex& right, size_t& permits, mutex& permitsLock) {
  waitForPermission(permits, permitsLock); // on next slide
  left.lock(); right.lock();
  cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
  sleep_for(getEatTime());
  cout << oslock << id << " all done eating." << endl << osunlock;
  grantPermission(permits, permitsLock); // on next slide
  left.unlock(); right.unlock();
}

static void philosopher(size_t id, mutex& left, mutex& right,
                        size_t& permits, mutex& permitsLock) {
  for (size_t i = 0; i < kNumMeals; i++) {
    think(id);
    eat(id, left, right, permits, permitsLock);
  }
}
```

Multithreading, Dining Philosophers, Deadlock Removal

- The implementation of **eat** on the prior slide deck introduces calls to **waitForPermission** and **grantPermission**.
  - The implementation of **grantPermission** is certainly the easier of the two to understand: transactionally increment the number of **permits** by one.
  - The implementation of **waitForPermission** is less obvious. Because we don't know what else to do (yet!), we busy wait with short naps until **permits** goes positive. Once that happens, we consume a permit and return.

```
static void waitForPermission(size_t& permits, mutex& permitsLock) {
  while (true) {
    permitsLock.lock();
    if (permits > 0) break;
    permitsLock.unlock();
    sleep_for(10);
  }
  permits--;
  permitsLock.unlock();
}

static void grantPermission(size_t& permits, mutex& permitsLock) {
  permitsLock.lock();
  permits++;
  permitsLock.unlock();
}
```

Dining Philosophers, Deadlock Removal, Busy Waiting

- The second version of the program works, in the sense that it never deadlocks.
  - It does, however, suffer from busy waiting, which the systems programmer gospel says is verboten unless there are no other options.
- A better solution? If a philosopher doesn't have permission to advance, then that thread should sleep until another thread sees reason to wake it up. In this example, another philosopher thread, after it increments **permits** within **grantPermission**, could notify the sleeping thread that a permit just became available.
- Implementing this idea requires a more sophisticated concurrency directive that supports a different form of thread communication—one akin to the use of signals and **sigwait** to support communication between processes. Fortunately, C++ provides a standard directive called the **condition_variable_any** to do exactly this.

```
class condition_variable_any {
public:
   void wait(mutex& m);
   template <typename Pred> void wait(mutex& m, Pred pred);
   void notify_one();
   void notify_all();
};
```

Multithreading and Condition Variables

- Here's the **main** thread routine that introduces a **condition_variable_any** to support the notification model we'll use in place of busy waiting. (Full program: here)

```
int main(int argc, const char *argv[]) {
  size_t permits = 4;
  mutex forks[5], m;
  condition_variable_any cv;
  thread philosophers[5];
  for (size_t i = 0; i < 5; i++) {
    mutex& left = forks[i], & right = forks[(i + 1) % 5];
    philosophers[i] =
       thread(philosopher, i, ref(left), ref(right), ref(permits), ref(cv), ref(m));
  }
  for (thread& p: philosophers) p.join();
  return 0;
}
```

  - All of the variables needed to foster inter-thread communication are defined across the first three lines of **main**.
  - The **philosopher** thread routine and the **eat** thread subroutine accept references to **permits**, **cv**, and **m**, because references to all three need to be passed on to **waitForPermission** and **grantPermission**.
  - I go with the shorter name **m** instead of **permitsLock** for reasons I'll soon get to.

Multithreading and Condition Variables

- Here's the **main** thread routine that introduces a **condition_variable_any** to support the notification model we'll use in place of busy waiting. (Full program: here)

```
int main(int argc, const char *argv[]) {
  size_t permits = 4;
  mutex forks[5], m;
  condition_variable_any cv;
  thread philosophers[5];
  for (size_t i = 0; i < 5; i++) {
    mutex& left = forks[i], & right = forks[(i + 1) % 5];
    philosophers[i] =
       thread(philosopher, i, ref(left), ref(right), ref(permits), ref(cv), ref(m));
  }
  for (thread& p: philosophers) p.join();
  return 0;
}
```

  - All of the variables needed to foster inter-thread communication are defined across the first three lines of **main**.
  - The **philosopher** thread routine and the **eat** thread subroutine accept references to **permits**, **cv**, and **m**, because references to all three need to be passed on to **waitForPermission** and **grantPermission**.
  - I go with the shorter name **m** instead of **permitsLock** for reasons I'll soon get to.

Multithreading and Condition Variables

- The new implementations of **waitForPermission** and **grantPermission** are below:

```
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  while (permits == 0) cv.wait(m);
  permits--;
}

static void grantPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  permits++;
  if (permits == 1) cv.notify_all();
}
```

  - The **lock_guard** is a convenience class whose constructor calls **lock** on the supplied **mutex** and whose destructor calls **unlock** on the same **mutex**. It's a convenience class used to ensure the lock on a **mutex** is released no matter how the function exits (early return, standard return at end, exception thrown, etc.)
  - **grantPermission** is a straightforward thread-safe increment, save for the fact that if **permits** just went from 0 to 1, it's possible other threads are waiting for a permit to become available. That's why the conditional call to **cv.notify_all()** is there.

Multithreading and Condition Variables

- Implementations of **waitForPermission** and **grantPermission**. Still right here:

```
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  while (permits == 0) cv.wait(m);
  permits--;
}

static void grantPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  permits++;
  if (permits == 1) cv.notify_all();
}
```

- The implementation of **waitForPermission** will eventually grant a permit to the calling thread, though it may need to wait a while for one to become available.
  - Yes, **waitForPermission** requires a **while** loop instead an **if** test. Why? It's possible the permit that just became available is immediately consumed by the thread that just returned it. Unlikely, but technically possible.
  - When **cv** is notified within **grantPermission**, the thread manager wakes the sleeping thread, but mandates it reacquire the lock on **m** (very much needed to properly re-evaluate **permits == 0**) before returning from **cv.wait(m)**.
  - If there aren't any permits, the thread is forced to sleep via **cv.wait(m)**. The thread manager releases the lock on **m** just as it's putting the thread to sleep.

Multithreading and Condition Variables

- The Dining Philosophers Problem, continued
  - **while** loops around **cv.wait(m)** calls are so common that the **condition_variable_any** class exports a second, two-argument version of **wait** whose implementation is a **while** loop around the first. That second version looks like this:

```
template <Predicate pred>
void condition_variable_any::wait(mutex& m, Pred pred) {
  while (!pred()) wait(m);
}
```

  - It's a template method, because the second argument supplied via **pred** can be anything capable of standing in for a zero-argument, **bool**-returning function.
  - The first **waitForPermissions** can be rewritten to rely on this, as with:

```
static void waitForPermission(size_t& permits, condition_variable_any& cv, mutex& m) {
  lock_guard<mutex> lg(m);
  cv.wait(m, [&permits] { return permits > 0; });
  permits--;
}
```

Multithreading and Condition Variables

- Fundamentally, the **size_t**, **condition_variable_any**, and **mutex** are collectively working together to track a resource count—in this case, four permission slips.
  - They provide thread-safe increment in **grantPermission** and thread-safe decrement in **waitForPermission**.
  - They work to ensure that a thread blocked on zero permission slips goes to sleep until further notice, and that it remains asleep until another thread returns one.
- In our latest **dining-philosopher** example, we relied on these three variables to collectively manage a thread-safe accounting of four permission slips. However!
  - There is little about the implementation that requires the original number be four. Had we gone with 20 philosophers and and 19 permission slips, **waitForPermission** and **grantPermission** would still work as is.
  - The idea of maintaining a thread-safe, generalized counter is so useful that most programming languages include more generic support for it. That support normally comes under the name of a **semaphore**.
  - For reason that aren't entirely clear to me, standard C++ omits the **semaphore** from its standard libraries. My guess as to why? It's easily built in terms of other constructs, so it was deemed unnecessary to provide official support for it. (Update, C++20 is including a **counting_semaphore**, but our versions clang++ and of **g++** don't support it yet. Still, it should have been part of C++11.)

Multithreading and Semaphores

- The **semaphore** constructor is so short that it's inlined right in the declaration of the **semaphore** class, e.g. **semaphore::semaphore(int val): value(val) {}**
- **semaphore::wait** is our generalization of **waitForPermission**.

```
void semaphore::wait() {
  lock_guard<mutex> lg(m);
  cv.wait(m, [this] { return value > 0; })
  value--;
}
```

- Why does the capture clause include the **this** keyword?
  - Because the anonymous predicate function passed to **cv.wait** is just that—a regular *function*. Since functions aren't normally entitled to examine the **private** state of an object, the capture clause includes **this** to effectively convert the **bool**-returning function into a **bool**-returning **semaphore** method.
- **semaphore::signal** is our generalization of **grantPermission**.

```
void semaphore::signal() {
  lock_guard<mutex> lg(m);
  value++;
  if (value == 1) cv.notify_all();
}
```

Multithreading and Semaphores

- Here's our final version of the **dining-philosophers**.

```
static void philosopher(size_t id, mutex& left, mutex& right, semaphore& permits) {
  for (size_t i = 0; i < 3; i++) {
    think(id);
    eat(id, left, right, permits);
  }
}

int main(int argc, const char *argv[]) {
  semaphore permits(4);
  mutex forks[5];
  thread philosophers[5];
  for (size_t i = 0; i < 5; i++) {
    mutex& left = forks[i], & right = forks[(i + 1) % 5];
    philosophers[i] = thread(philosopher, i, ref(left), ref(right), ref(permits));
  }
  for (thread& p: philosophers) p.join();
  return 0;
}
```

  - It strips out the exposed **size_t**, **mutex**, and **condition_variable_any** and replaces them with a single **semaphore**.
  - It updates the thread constructors to accept a single reference to that **semaphore**.

Multithreading and Semaphores

- **eat** now relies on that **semaphore** to play the roles previously played by **waitForPermission** and **grantPermission**.

```
static void eat(size_t id, mutex& left, mutex& right, semaphore& permits) {
  permits.wait();
  left.lock();
  right.lock();
  cout << oslock << id << " starts eating om nom nom nom." << endl << osunlock;
  sleep_for(getEatTime());
  cout << oslock << id << " all done eating." << endl << osunlock;
  permits.signal();
  left.unlock();
  right.unlock();
}
```

  - We could switch the order of the last two lines, so that **right.unlock()** precedes **left.unlock()**. Is the switch a good idea? a bad one? or is it really just arbitrary?
  - A former student suggested we use a **mutex** to bundle the calls to **left.lock()** and **right.lock()** into a critical region. Is this a solution to the deadlock problem?
  - We could lift the **permits.signal()** call up to appear in between **right.lock()** and the first **cout** statement. Is that valid? Why or why not?