CS 318 Principles of Operating Systems

Fall 2021

Lecture 8: Synchronization Exercises

Prof. Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Administrivia

In-class Quiz next Tuesday (09/28)

- For Lecture 3 and 4
- Similar format as Quiz 1, bring a computer

Using Semaphores

We've looked at a simple example for using synchronization

- Mutual exclusion while accessing a bank account

Now let's use semaphores to look at more interesting examples

- Readers/Writers
- Bounded Buffers
- Building H2O

Readers/Writers Problem

Readers/Writers Problem:

- An object is shared among several threads
- Some threads only read the object, others only write it
- We can allow multiple readers but only one writer
  - Let $\#r$ be the number of readers, $\#w$ be the number of writers
  - Safety: $(\#r \geq 0) \wedge (0 \leq \#w \leq 1) \wedge ((\#r > 0) \Rightarrow (\#w = 0))$

How can we use semaphores to implement this protocol?

Use three variables

- `int readcount` – number of threads reading object
- Semaphore `mutex` – control access to `readcount`
- Semaphore `w_or_r` – exclusive writing or reading

Readers/Writers

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex(1);
// exclusive writer or reader
Semaphore w_or_r(1);

writer() {
    wait(&w_or_r);  // lock out readers
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex);     // lock readcount
    readcount += 1;   // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex);    // unlock readcount
    Read;
    wait(&mutex);      // lock readcount
    readcount -= 1;    // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex);     // unlock readcount
}
```

Readers/Writers

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex(1);
// exclusive writer or reader
Semaphore w_or_r(1);

writer() {
    wait(&w_or_r);   // lock out readers
    Write;
    signal(&w_or_r);// up for grabs
}
```

```
reader() {
    wait(&mutex);     // lock readcount
    readcount += 1;   // one more reader
    if (readcount == 1)
        wait(&w_or_r);// synch w/ writers
    signal(&mutex);    // unlock readcount
    Read;
    wait(&mutex);      // lock readcount
    readcount -= 1;    // one less reader
    if (readcount == 0)
        signal(&w_or_r); // up for grabs
    signal(&mutex);     // unlock readcount
}
```

Readers/Writers Notes

`w_or_r` **provides mutex between readers and writers**

- writer wait/signal, reader wait/signal when `readcount` goes from 0 to 1 or from 1 to 0.

If a writer is writing, where will readers be waiting?

Once a writer exits, all readers can fall through

- Which reader gets to go first?
- Is it guaranteed that all readers will fall through?

If readers and writers are waiting, and a writer exits, who goes first?

Why do readers use `mutex`?

Why don't writers use `mutex`?

What if the signal is above "`if (readcount == 1)`"?

Semaphores in Pintos

```
reader() {
    wait(&mutex); // sema_down
    …
    signal(&mutex); // sema_up
    Read;
    wait(&mutex);
    …
    signal(&mutex);
}
```

```
void sema_down(struct semaphore *sema)
{
  enum intr_level old_level;
  old_level = intr_disable();
  while (sema->value == 0) {
    list_push_back(&sema->waiters,
          &thread_current()->elem);
    thread_block();
  }
  sema->value--;
  intr_set_level(old_level);
}
```

```
void sema_up(struct semaphore *sema)
{
  enum intr_level old_level;
  old_level = intr_disable();
  if (!list_empty (&sema->waiters))
    thread_unblock(list_entry(
      list_pop_front(&sema->waiters),…));
  sema->value++;
  intr_set_level(old_level);
}
```

Bounded Buffer

Problem: a set of buffers shared by producer and consumer threads

- **Producer** inserts resources into the buffer set
  - Output, disk blocks, memory pages, processes, etc.
- **Consumer** removes resources from the buffer set
- Whatever is generated by the producer

Producer and consumer execute at different rates

- No serialization of one behind the other
- Tasks are independent (easier to think about)
- The buffer set allows each to run without explicit handoff

Safety:

- Sequence of consumed values is prefix of sequence of produced values
- If nc is number consumed, np number produced, and N the size of the buffer, then $0 \leq np - nc \leq N$

Bounded Buffer (2)

$$0 \le np - nc \le N \quad \Leftrightarrow \quad 0 \le (nc - np) + N \le N$$

Use three semaphores:

- empty – number of empty buffers
  - Counting semaphore
  - empty = $(nc - np) + N$
- full – number of full buffers
  - Counting semaphore
  - full = $np - nc$
- mutex – mutual exclusion to shared set of buffers
  - Binary semaphore

Bounded Buffer (3)

```
Semaphore mutex(1);  // mutual exclusion to shared set of buffers
Semaphore empty(N);  // count of empty buffers (all empty to start)
Semaphore full(0);   // count of full buffers (none full to start)
```

```
producer() {
  while (1) {
    Produce new resource;
    wait(&empty); // wait for empty buffer
    wait(&mutex); // lock buffer list
    Add resource to an empty buffer;
    signal(&mutex); // unlock buffer list
    signal(&full);  // note a full buffer
  }
}
```

```
consumer() {
  while (1) {
    wait(&full);  // wait for a full buffer
    wait(&mutex); // lock buffer list
    Remove resource from a full buffer;
    signal(&mutex); // unlock buffer list
    signal(&empty); // note an empty buffer
    Consume resource;
  }
}
```

Bounded Buffer (4)

Why need the mutex at all?

Where are the critical sections?

What has to hold for deadlock to occur?

- $empty = 0$ and $full = 0$
- $(nc - np) + N = 0$ and $np - nc = 0$
- $N = 0$

What happens if operations on mutex and full/empty are switched around?

- The pattern of signal/wait on full/empty is a common construct often called an interlock

Readers/Writers and Bounded Buffer are classic sync. problems

Monitor Readers and Writers

Using Mesa monitor semantics.

Will have four methods: `StartRead`, `StartWrite`, `EndRead` **and** `EndWrite`

Monitored data: `nr` **(# of readers) and** `nw` **(# of writers) with monitor invariant**

$$(nr \geq 0) \land (0 \leq nw \leq 1) \land ((nr > 0) \Rightarrow (nw = 0))$$

Two conditions:

- `canRead`: $nw = 0$
- `canWrite`: $(nr = 0) \land (nw = 0)$

Monitor Readers and Writers

Write with just `wait()`

- Will be safe, maybe not live – why?

```
Monitor RW {
  int nr = 0, nw = 0;
  Condition canRead, canWrite;

  void StartRead () {
    while (nw != 0) wait(canRead);
    nr++;
  }

  void EndRead () {
    nr--;
  }
```

```
  void StartWrite {
    while (nr != 0 || nw != 0) wait(canWrite);
    nw++;
  }

  void EndWrite () {
    nw--;
  }
} // end monitor
```

Monitor Readers and Writers

add `signal()` and `broadcast()`

```
Monitor RW {
  int nr = 0, nw = 0;
  Condition canRead, canWrite;

  void StartRead () {
    while (nw != 0) wait(canRead);
    nr++;
  }                          can we put a signal here?

  void EndRead () {
    nr--;
    if (nr == 0) signal(canWrite);
  }
```

```
  void StartWrite () {
    while (nr != 0 || nw != 0) wait(canWrite);
    nw++;
  }                          can we put a signal here?

  void EndWrite () {
    nw--;
    broadcast(canRead);
    signal(canWrite);
  }
} // end monitor
```

Monitor Readers and Writers

Is there any priority between readers and writers?

What if you wanted to ensure that a waiting writer would have priority over new readers?

Monitor Bounded Buffer

```
Monitor bounded_buffer {
  Resource buffer[N];
  // Variables for indexing buffer
  // monitor invariant involves these vars
  Condition not_full; // space in buffer
  Condition not_empty; // value in buffer

  void put_resource (Resource R) {
    while (buffer array is full)
        wait(not_full);
    Add R to buffer array;
    signal(not_empty);
  }
```

```
  Resource get_resource() {
    while (buffer array is empty)
        wait(not_empty);
    Get resource R from buffer array;
    signal(not_full);
    return R;
  }
} // end monitor
```

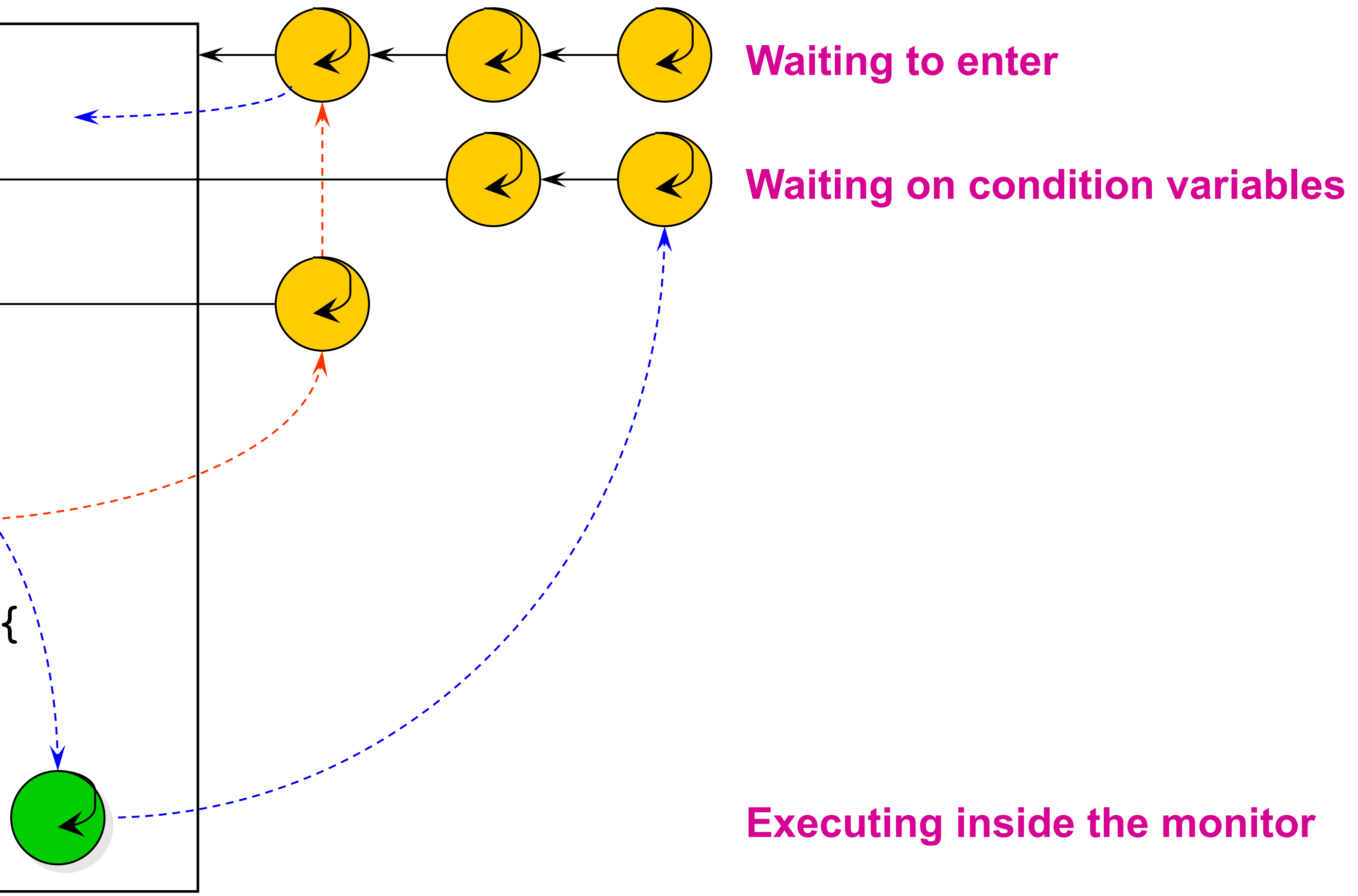
- What happens if no threads are waiting when signal is called?

Monitor Queues

```
Monitor bounded_buffer {

  Condition not_full;
  …other variables…
  Condition not_empty;

  void put_resource() {
    …wait(not_full)…
    …signal(not_empty)…
  }
  Resource get_resource() {
    …
  }
}
```



The H2O Problem

Setup:

- You have been hired by a company to do climate modelling of oceans.
- The program matches atoms of different types as they form molecules.
- In an excessive reliance on threads, each atom is represented by a thread.

Requirements

- Write code to form water out of two hydrogen threads and one oxygen thread (H2O)
- Two procedures: `HArrives()` and `OArrives()`
    - A water molecule forms when two H threads are present and one O thread.
    - Otherwise, the atoms must wait.
    - Once all three are present, one of the threads calls `MakeWater()` and only then, all three depart.

Description from "Operating Systems: Principles and Practice"

Define Variables

Data Structure

```
Status {
    bool ready;
    Condition cv;
};
```

Key Variables

- `int numH` – number of hydrogen threads waiting
- `int numO` – number of oxygen threads waiting
- `Semaphore mutex` – control access to `numH` and `numO`

- `List<Status *> waitingH` – hydrogen threads waiting queue
- `List<Status *> waitingO` – oxygen threads waiting queue

Building H2O

```
int numH = 0; // number of hydrogen threads waiting
int numO = 0; // number of oxygen threads waiting
Semaphore mutex(1);  // mutual exclusion
List<Status *> waitingH; // hydrogen threads waiting queue
List<Status *> waitingO; // oxygen threads waiting queue
```

```
HArrives() {
  wait(&mutex);
  numH++;
  if (numH == 2 && numO >= 1) {
    h = waitingH.remove();
    o = waitingO.remove();
    h->ready = true;
    o->ready = true;
    cond_signal(&h->cv);
    cond_signal(&o->cv);
    numH -= 2;
    numO -= 1;
    MakeWater();
  }
```

```
  else {
    h = new Status;
    waitingH.add(h);
    while (!h->ready)
      cond_wait(&h->cv, &mutex);
    delete h;
  }
  signal(&mutex);
}
```

Building H2O

```
int numH = 0; // number of hydrogen threads waiting
int numO = 0; // number of oxygen threads waiting
Semaphore mutex(1);  // mutual exclusion
List<Status *> waitingH; // hydrogen threads waiting queue
List<Status *> waitingO; // oxygen threads waiting queue
```

```
OArrives() {
  wait(&mutex);
  numO++;
  if (numH >= 2) {
    h1 = waitingH.remove();
    h2 = waitingH.remove();
    h1->ready = true;
    h2->ready = true;
    cond_signal(&h1->cv);
    cond_signal(&h2->cv);
    numH -= 2;
    numO -= 1;
    MakeWater();
  }
```

```
  else {
    o = new Status;
    waitingO.add(o);
    while (!o->ready)
      cond_wait(&o->cv, &mutex);
    delete o;
  }
  signal(&mutex);
}
```

Next Time…

Read Chapter 32