Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 9: Semaphores

Prof. Ryan Huang

Review: writing concurrent programs

Multiple threads work on shared data

- Must synchronize to ensure correct results

Synchronization

- Two types: mutual exclusion, ordering
- Difficult with low-level mechanism (e.g., atomic load, atomic store)

Monitors

- Locks for mutual exclusion
- Condition variables for ordering constraints

Semaphores

An abstract "integer" (initialized to some value *N*)

- Proposed by Dijkstra in the "THE" system in 1968
- Generalized lock

Provides two operations

- `down()`: wait for semaphore value to become positive, then decrement it by 1
  - Originally called `P()`, also `wait()`
- `up()`: increment semaphore value by 1
  - Originally called `V()`, also `signal()`

Atomic

```
while (1) {
  if (value > 0) {
    value--
    break
  }
}
```

Atomic

```
value++
```

No other operations – not even just reading its value!

Semaphores (cont'd)

Represent a resource with *N* available units or a resource that allows concurrent accesses

- E.g., a parking lot with *N* available spots

`down()` **will return only *N* more times than** `up()`

Special case: binary semaphore

- Semaphore value is 0 or 1
- `up()` atomically sets value to 1
- Versus counting semaphore

Is binary semaphore equivalent to lock?

Semaphores can enforce mutual exclusion <u>and</u> ordering

Mutual exclusion

```
semaphore sem(1)
sem.down()
critical section
sem.up()
```

Ordering constraints

- Example: ensure that Thread A's task is done before Thread B's task

```
semaphore sem(0)
```

Thread A	**Thread B**
	sem.down()
task A	task B
sem.up()	

Recap: producer-consumer

Producers fill a shared buffer; consumers empty it

Coke machine problem

- Coke machine can hold at most MAX cokes
- Delivery person (producer) adds one coke to machine
- Consumer buys one coke

Producer-consumer with semaphores

As always, think about shared data, mutual exclusion, and before-after constraints

Semaphores:

- *mtx*: for exclusive access to coke machine
- *fullSlots*: counts number of cokes in machine
- *emptySlots*: counts number of empty slots in machine

Before-after constraints

- Consumer must wait if no cokes in machine
- Producer must wait if machine is full

Producer-consumer with semaphores

```
semaphore mtx(1);             // mutual exclusion to shared data
semaphore fullSlots(0);       // number of full slots
semaphore emptySlots(MAX);    // number of empty slots
```

```
producer {
  mtx.down()

  emptySlots.down()

  // add coke to machine

  fullSlots.up()

  mtx.up()

}
```

```
consumer {
  mtx.down()

  fullSlots.down()

  // take coke out of machine

  emptySlots.up()

  mtx.up()

}
```

Producer-consumer with semaphores

Why do we need different semaphores for fullSlots and emptySlots?

What if there's 1 full slot, and multiple consumers call down() at the same time?

Producer-consumer with semaphores

```
semaphore mtx(1);              // mutual exclusion to shared data
semaphore fullSlots(0);        // number of full slots
semaphore emptySlots(MAX);     // number of empty slots
```

```
producer {
  mtx.down()

  emptySlots.down()

  // add coke to machine

  fullSlots.up()

  mtx.up()

}
```

```
consumer {
  mtx.down()

  fullSlots.down()

  // take coke out of machine

  emptySlots.up()

  mtx.up()

}
```

What's wrong with this solution?

Producer-consumer with semaphores

```
semaphore mtx(1);            // mutual exclusion to shared data
semaphore fullSlots(0);      // number of full slots
semaphore emptySlots(MAX);   // number of empty slots
```

```
producer {
   mtx.down()

   emptySlots.down()

   // add coke to machine

   fullSlots.up()

   mtx.up()

}
```

```
consumer {
   mtx.down()

   fullSlots.down()

   // take coke out of machine

   emptySlots.up()

   mtx.up()

}
```

Any concerns?

Producer-consumer with semaphores

What about the state of the system between between emptySlots.down() and mtx.down()?

Does the order of up() matter?

What if producer called fullSlots.up() earlier?

```
semaphore mtx(1);              // mutual exclusion to shared data
semaphore fullSlots(0);        // number of full slots
semaphore emptySlots(MAX);     // number of empty slots
```

```
producer {
  emptySlots.down()

  mtx.down()

  // add coke to machine

  mtx.up()

  fullSlots.up()

}
```

```
consumer {
  fullSlots.down()

  mtx.down()

  // take coke out of machine

  mtx.up()

  emptySlots.up()

}
```

Monitors vs. semaphores

Monitors

- Locks for mutual exclusion
- Condition variables for ordering

Semaphores: one mechanism for both mutual exclusion and ordering

- Elegantly minimal
- Can be difficult to use

Mutual exclusion (locks vs. semaphores)

lock = binary semaphore (initialized to 1)

`lock()` = `down()`

`unlock()` = `up()`

Mutex	**Semaphore**

`mutex m`
 	
`m.lock()`
`<critical section>`
`m.unlock()`
	`semaphore m(1)`
	
	`m.down()`
	`<critical section>`
	`m.up()`

Ordering (condition variables vs. semaphores)

Condition variable	Semaphore
`while (condition) { wait(m)}`	`down()`
Waiting condition is in user code and uses user variables (more flexible)	Waiting condition is in semaphore code and uses semaphore's value (wait if v==0)
User variables are protected by mutex	Semaphore's value is protected by semaphore's implementation of `up()`, `down()`
Must hold lock when calling wait	Must not hold "lock" (semaphore) when calling `down()` for ordering

Ordering (condition variables vs. semaphores)

Condition variable	Semaphore
No memory of past signals: <u>Thread A</u> `wait()` <u>Thread B</u> `signal()` What happens if wait(), then signal()? What happens if signal(), then wait()? Why is this ok?	Remembers past `up()` calls <u>Thread A</u> `down()` <u>Thread B</u> `up()` What happens if down(), then up()? What happens if up(), then down()?

Implementing custom waiting condition with semaphores

Semaphores work best if the shared integer and waiting condition (value==0) map naturally to problem domain

How to implement custom waiting condition with semaphores?

What's a condition variable?

A place for threads to wait

"A place for threads"

- A set of waiting threads

"to wait"

- To wait, create a semaphore(0), add the semaphore to the waiting set, then call down() on it
- To signal, call up() on the waiting thread's semaphore

```
mutex cokeLock
cv waitingProducers
cv waitingConsumers
```

Producer

```
cokeLock.lock()

while (numCokes == MAX) {
   waitingProducers.wait(cokeLock)
}

numCokes++

waitingConsumers.signal()

cokeLock.unlock()
```

Consumer

```
cokeLock.lock()

while (numCokes == 0) {
   waitingConsumers.wait(cokeLock)
}

numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

```
sem cokeLock(1)
set<sem*> waitingConsumers
set<sem*> waitingProducers
```

Producer

```
cokeLock.down()
while (numCokes == MAX) {
    waitingProducers.wait(cokeLock)
    sem *s = new sem(0)
    waitingProducers.insert(s)
    cokeLock.up()
    s->down()
    cokeLock.down()
}
numCokes++
waitingConsumers.signal()
if (!waitingConsumers.empty()) {
    waitingConsumers.begin()->up()
    waitingConsumers.erase(
          waitingConsumers.begin())
}
cokeLock.up()
```

Consumer

```
cokeLock.down()
while (numCokes == 0) {
    waitingConsumers.wait(cokeLock)
}

numCokes--

waitingProducers.signal()

cokeLock.up()
```

```
sem cokeLock(1)
set<sem*> waitingConsumers
set<sem*> waitingProducers
```

Producer

```
cokeLock.down()
while (numCokes == MAX) {
    waitingProducers.wait(cokeLock)
    sem *s = new sem(0)
    waitingProducers.insert(s)
    cokeLock.up()
    s->down()
    cokeLock.down()
}
numCokes++
waitingConsumers.signal()
if (!waitingConsumers.empty()) {
    waitingConsumers.begin()->up()
    waitingConsumers.erase(
          waitingConsumers.begin())
}
cokeLock.up()
```

Consumer

```
cokeLock.down()
while (numCokes == 0) {
    waitingConsumers.wait(cokeLock)
    sem *s = new sem(0)
    waitingConsumers.insert(s)
    cokeLock.up()
    s->down()
    cokeLock.down()
}
numCokes--
waitingProducers.signal()
if (!waitingProducers.empty()) {
     waitingProducers.begin()->up()
     waitingProducers.erase(
          waitingProducers.begin())
}
cokeLock.up()
```

```
sem cokeLock(1)
set<sem*> waitingConsumers
set<sem*> waitingProducers
```

Producer

```
cokeLock.down()
while (numCokes == MAX) {
   sem *s = new sem(0)
   waitingProducers.insert(s)
   cokeLock.up()
   s->down()
   cokeLock.down()
}
numCokes++
if (!waitingConsumers.empty()) {
   waitingConsumers.begin()->up()
   waitingConsumers.erase(
         waitingConsumers.begin())
}
cokeLock.up()
```

Consumer

```
cokeLock.down()
while (numCokes == 0) {
   sem *s = new sem(0)
   waitingConsumers.insert(s)
   cokeLock.up()
   s->down()
   cokeLock.down()
}
numCokes--
if (!waitingProducers.empty()) {
    waitingProducers.begin()->up()
    waitingProducers.erase(
         waitingProducers.begin())
}
cokeLock.up()
```

What happens if we get interrupted after releasing the cokeLock but before downing the semaphore?