Semaphores in Java

IN5170/IN9170 – Models of concurrency

Semaphores in Java, lecture 3

Fall 2022

7. September, 2022



Repetition: semaphores

- Semaphore: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two atomic operations:

P and V

- **P:** (Passeren) Wait for signal – want to pass
  - effect: wait until the value is greater than zero, and decrease the value by one
- **V:** (Vrijgeven) Signal an event – release
  - effect: increase the value by one

Basic methods of semaphores in Java

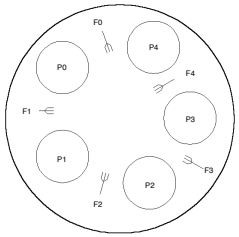
- `Semaphore(int n)` [1]
  - constructor for semaphores
  - initializes value with integer `n`; is called "set of permits"
- `acquire()`
  - corresponds to P operation
  - tries to reduce number of permits by 1
  - blocks, if that is not possible and waits, until semaphore gives permit
- `release()`
  - corresponds to V operation
  - increases number of permits by 1

[1] see `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html`

Repetition: dining philosophers

- philosophers: need two forks to eat
- forks as semaphores

```
process Philosopher [i = 0 to 4] {
  while true {
    think;
    acquire forks;
    eat;
    release forks;
  }
}
```



TODO: better picture

Dining philosophers: naïve solution in Java (I)

- forks are binary semaphores
- Philosopher is a class with
  - two attributes of type Semaphore (`leftFork` and `rightFork`)
  - the functions `eat()`, `think()` and `run()`
- creating forks = array of binary semaphores:

```
Semaphore[] forks = new Semaphore[numberOfPhilosophers];
for (int i=0; i <forks.length; i++){
  forks[i] = new Semaphore(1);
}
```

- create philosophers = array of `Philosopher` objects

```
philosophers = new Philosopher[numberOfPhilosophers];
for (int i=0; i < philosophers.length; i++) {
  philosophers[i] =
  new Philosopher(forks[i], forks[(i+1) % forks.length]);
}
```

Dining philosophers: naïve solution in Java (II)

- philosopher runs

```
while(true) {
  think();                          // think some time
  leftFork.acquire();               // acquire forks
  rightFork.acquire();
  eat();                            // eat some time
  leftFork.release();               // release forks
  rightFork.release();
}
```

- What is the problem with this solution?
  - can run into deadlocks
  - starvation can occur

Fair semaphores in Java

Fair semaphores in Java give permits in the order in which the corresponding acquires where received (FIFO behavior).

- semaphores in Java are in general not fair
- fairness leads to minor performance reduction
- in most cases: fair semaphores desired
- special constructor `Semaphore(int n, boolean fair)`:
  - `n`: number of permits
  - `fair=true` $\Longleftrightarrow$ semaphore is fair
- method `isFair()` can be used to determine if a semaphore is fair

Dining philosophers 2nd attempt (idea)

breaking the symmetry

To avoid deadlock, let 1 philosopher (say 4) grab the right fork first

```
process Philosopher [i = 0 to 3] {
  while true {
    think;
    P(fork[i]);
    P(fork[(i+1)%5]);
    eat;
    V(fork[i]);
    V(fork[(i+1)%5]);
  }
}
```

Dining philosophers 2nd attempt (idea)

breaking the symmetry

To avoid deadlock, let 1 philosopher (say 4) grab the right fork first

```
process Philosopher [i = 0 to 3] {
  while true {
    think;
    P(fork[i]);
    P(fork[(i+1)%5]);
    eat;
    V(fork[i]);
    V(fork[(i+1)%5]);
  }
}
```

```
process Philosopher4 {
  while true {
    think;
    P(fork[0]);
    P(fork[4]);
    eat;
    V(fork[4]);
    V(fork[0]);
  }
}
```

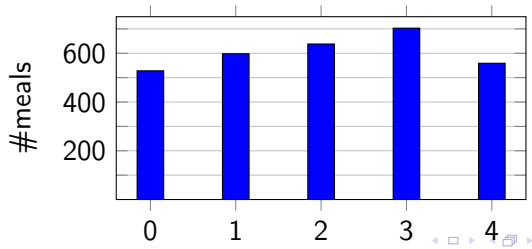
Dining philosophers 2nd attempt (implementation)

- new subclass: "inverted philosopher": picks right fork first

```
while(true) {
  think();
  rightFork.acquire();
  leftFork.acquire();
  eat();
  leftFork.release();
  rightFork.release();
}
```

- What is the problem with this solution?

successful attempts to eat per philosopher (experiment: 10s)



Dining philosophers 3rd approach (idea)

- Observation: no deadlock if at most 4 philosophers try to access forks at the same time
- Idea: restrict number of available chairs to 4
- implemented by one counting semaphore that organizes permits to use access forks, i.e. to use a chair
- needs to be fair to guaranty fairness
- each philosopher gets this semaphore as an additional attribute

Dining philosophers 3rd approach (implementation)

- define semaphore and create philosophers accordingly

```
// one less chair than there are people
Semaphore chairs =
        new Semaphore(philosophers.length - 1, true);
for (int i=0; i < philosophers.length; i++) {
  philosophers[i] =
        new Philosopher(forks[i],
                        forks[(i+1) % forks.length],
                        chairs);
}
```

- philosopher runs

```
while(true) {
  think();
  chairs.acquire();              // get a chair
  leftFork.acquire();            // follow naive approach
  rightFork.acquire();
  eat();
  leftFork.release();
  rightFork.release();
  chairs.release();              // release the chair
}
```

Comparison of solutions

approach	pro	con
naïve		deadlock possible starvation possible
asymmetric	no deadlock no starvation[2]	avg. number of meals not equal
sitting restriction	no deadlock no starvation[2]	central instance required

- there exist many more approaches for this problem[3]

[2] if semaphores are fair

[3] see e.g. https://en.wikipedia.org/wiki/Dining_philosophers_problem#Solutions

Conclusion

- `Java` provides a class `Semaphore`
- methods `acquire()` and `release()` correspond to P and V operation
- can be used as counting and binary semaphores
- we have seen three approaches to solve the dining philosophers problem

References I