Semaphores

Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

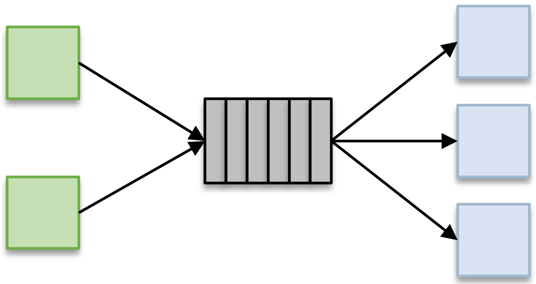
Systems Software &
Architecture Lab.

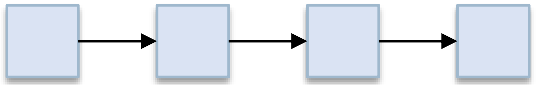
Seoul National University

Spring 2024

Synchronization Types

- Mutual exclusion
  - Only one thread in a critical section at a time
- Waiting for events
  - One thread waits for another to complete some action before it continues
  - Producer/consumer
    - Multiple producers, multiple consumers
  - Pipeline
    - A series of producer and consumer





Higher-level Synchronization

- Spinlocks and disabling interrupts are not enough
  - Useful only for very short and simple critical sections
  - Need to block threads when lock is held by others (mutexes)
  - Need to block threads until a certain condition is met

- Higher-level synchronization mechanisms
  - Semaphores
  - Monitors
  - Mutexes and condition variables (used in Pthreads)

Semaphores

- A synchronization primitive higher level than locks
  - Invented by Dijkstra in 1968, as part of the THE OS
  - Does not require busy waiting
  - A semaphore is an object with an integer value (state)
  - State cannot be directly accessed by user program, but it determines the behavior of semaphore operations

- Manipulated atomically through two operations
  - `Wait()`: decrement the value, and wait until the value is >= 0:
    Also called as `P()` (after Dutch word for test), `down()`, or `sem_wait()`
  - `Signal()`: increment the value, then wake up a single waiter:
    Also called as `V()` (after Dutch word for increment), `up()`, or `sem_post()`

Image from https://commons.wikimedia.org/wiki/File:Semaphore_signals,_Harrogate_railway_station_%2819th_April_2019%29_001.jpg

Implementing Semaphores

```
typedef struct {
  int value;
  struct process *Q;
} semaphore;
```

```
void wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S->Q;
    block();
  }
}

void signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->Q;
    wakeup(P);
  }
}
```

wait() / signal() are critical sections! Hence, they must be executed atomically with respect to each other.

HOW??

Types of Semaphores

- Binary semaphore (≈ mutex)
  - Semaphore value is initialized to 1
  - Guarantees mutually exclusive access to resource
  - Only one thread allowed entry at a time

- ____________ semaphore
  - Semaphore value is initialized to N
  - Represents a resource with many units available
  - Allows threads to enter as long as more units are available

Bounded Buffer Problem (1)

- Producer/consumer problem
  - There is a set of resource buffers shared by producers and consumers
  - Producer inserts resources into the buffer
    - Output, disk blocks, memory pages, etc.
  - Consumer removes resources from the buffer
    - Whatever is generated by the producer
  - Producer and consumer execute at different rates
    - No serialization of one behind the other
    - Tasks are independent
    - The buffer allows each to run without explicit handoff
  - pipe: single producer, single consumer

Bounded Buffer Problem (2)

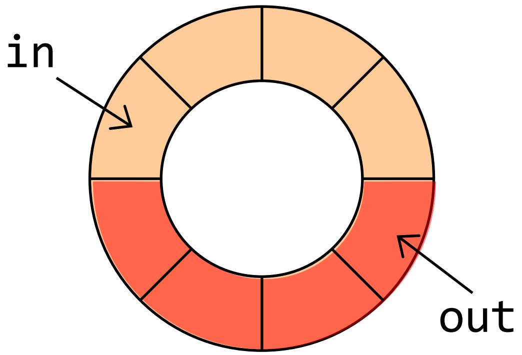
- No synchronization

Producer

```
void produce(data)
{

  while (count==N);
  buffer[in] = data;
  in = (in+1) % N;
  count++;

}
```

```
int count;
```

```
struct item buffer[N];
int in, out;
```



Consumer

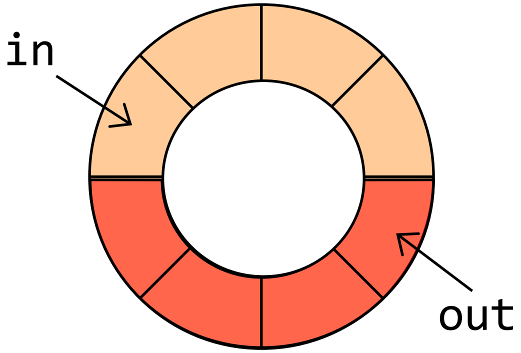
```
void consume(&data)
{

  while (count==0);
  *data = buffer[out];
  out = (out+1) % N;
  count--;

}
```

Bounded Buffer Problem (3)

- Implementation with semaphores

```
Semaphore
      mutex = 1;
      empty = N;
      full = 0;
```

```
struct item buffer[N];
int in, out;
```



Producer

```
void produce(data)
{

  wait(&empty);
  wait(&mutex);
  buffer[in] = data;
  in = (in+1) % N;
  signal(&mutex);
  signal(&full);

}
```

Consumer

```
void consume(&data)
{

  wait(&full);
  wait(&mutex);
  *data = buffer[out];
  out = (out+1) % N;
  signal(&mutex);
  signal(&empty);

}
```

Readers-Writers Problem (1)

- Sharing resource among multiple readers and writers
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers at a time
  - We can only allow one writer at a time

- Implementation with semaphores
  - `readcount`: # of threads reading object
  - `mutex`: control access to `readcount`
  - `rw`: exclusive writing or reading

Readers-Writers Problem (2)

```
// number of readers
int readcount = 0;

// mutex for readcount
Semaphore mutex = 1;

// mutex for reading/writing
Semaphore rw = 1;

void Writer()
{
  wait(&rw);
  ...
  // Write
  ...
  signal(&rw);
}
```

```
void Reader()
{
  wait(&mutex);
  readcount++;
  if (readcount == 1)
     wait(&rw);
  signal(&mutex);
  ...

  // Read


  ...
  wait(&mutex);
  readcount--;
  if (readcount == 0)
     signal(&rw);
  signal(&mutex);
}
```

Readers-Writers Problem (3)

- If there is a writer
  - The first reader blocks on `rw`
  - All other readers will then block on `mutex`
- Once a writer exits, all readers can fall through
  - Which reader gets to go first?
- The last reader to exit signals waiting writer
  - Can new readers get in while writer is waiting?
- When a writer exits, if there is both a reader and writer waiting, which one goes next is up to scheduler

Dining Philosophers Problem (1)

- A classic synchronization problem by Dijkstra, 1965
- Modeled after the lives of five philosophers sitting around a round table

- Each philosopher repeats forever:
  - Thinking
  - Pick up two forks
  - Eating
  - Put down two forks
- Pick one fork at a time

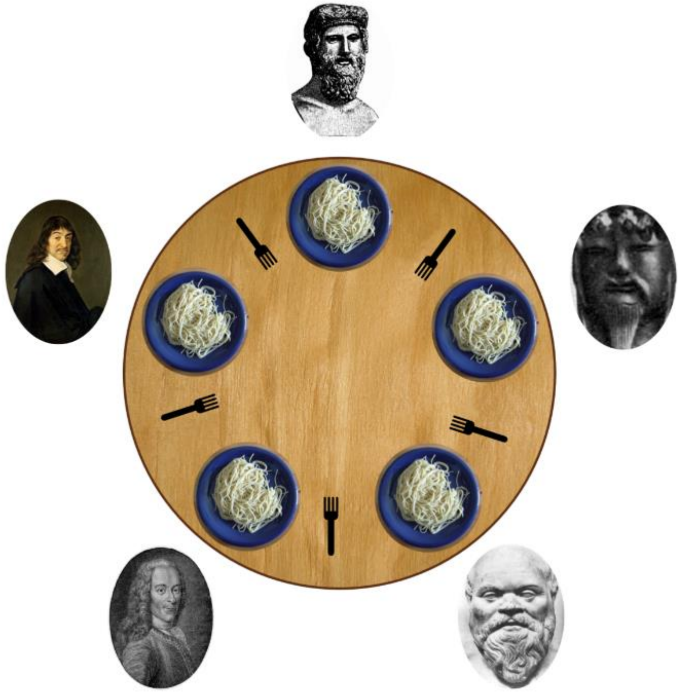


Image from https://en.wikipedia.org/wiki/Dining_philosophers_problem

Dining Philosophers Problem (2)

- A simple solution

```c
// initialized to 1
Semaphore forks[N];

#define L(i)  (i)
#define R(i)  ((i + 1) % N)

void philosopher(int i)
{
  while (1) {
    think();
    pickup(i);
    eat();
    putdown(i);
  }
}
```

```c
void pickup(int i) {
  wait(&forks[L(i)]);
  wait(&forks[R(i)]);
}

void putdown(int i) {
  signal(&forks[L(i)]);
  signal(&forks[R(i)]);
}
```

Dining Philosophers Problem (3)

- A deadlock-free solution

```
// initialized to 1
Semaphore forks[N];

#define L(i)  (i)
#define R(i)  ((i + 1) % N)

void philosopher(int i)
{
  while (1) {
    think();
    pickup(i);
    eat();
    putdown(i);
  }
}
```

```
void pickup(int i) {
  if (i == (N-1)) {
    wait(&forks[R(i)]);
    wait(&forks[L(i)]);
  } else {
    wait(&forks[L(i)]);
    wait(&forks[R(i)]);
  }
}

void putdown(int i) {
  signal(&forks[L(i)]);
  signal(&forks[R(i)]);
}
```

Summary

- Pros
  - Simple, yet powerful
  - Same primitive can be used for both critical sections (mutual exclusion) and coordination among threads (scheduling)

- Cons
  - They are essentially shared global variables; can be accessed from anywhere (bad software engineering)
  - There is no connection between the semaphore and the data being controlled by it
  - No control over their use, no guarantee of proper usage
  - Hard to program with and prone to bugs