# Concurrency Problems

Thierry Sans

# (recap) Lock

**A lock** is an object in memory providing two atomic operations `acquire` and `release`

- ➡ We designed a blocking lock in lecture 3 (threads and synchronization)
  - ✓ **Prevent starvation** by providing a fair access to the lock (thread queue)
  - ✓ **No busy waiting** (no waste of CPU time)

# (recap) Semaphore

**A semaphore** is an object in memory providing two atomic operations `sem_wait` and `sem_signal`

- **Binary semaphore** (a.k.a *mutex*)
  controls access to a single resource (behave like a lock)
- **General semaphore** (a.k.a counting semaphore)
  controls access to a finite number of resources `n` available

✓ Same properties as the lock : no starvation, no busy waiting

# Lock vs Semaphore

**In the literature**, they are semantically different

- A lock can only be released by the owner who acquired it
- A semaphore does not have the concept of owner

**In practice**, a binary semaphore (with initial value 1) behave as a lock

➡ A lock can be used to implement a semaphore and vice versa

# The producer consumer problem

```
void producer () {
  while(1){
     item := produce()
     write(buffer, item)
  }
}
```

Critical Section!

```
void consumer () {
  while(1){
     item := read(buffer)
     consume(item)
  }
}
```

# (Bad) Producer consumer **using locks**

```
void producer () {
  while(1){
     item := produce()
     acquire(lock)
     write(buffer, item)
     release(lock)
  }
}
```

```
void consumer () {
  while(1){
     acquire(lock)
     item := read(buffer)
     release(lock)
     consume(item)
  }
}
```

- The producer might write into a full buffer
- The consumer might read from an empty buffer

# (Good) Producer consumer **using locks**

```
void producer () {
  while(1){
     item := produce()
     acquire(lock)
     while(!emtpy(buffer)){
          release(lock);
          yield();
          acquire(lock);
     }
     write(buffer, item)
     release(lock)
  }
}
```

```
void consumer () {
  while(1){
     acquire(lock)
     while(emtpy(buffer)){
          release(lock);
          yield();
          acquire(lock);
     }
     item := read(buffer)
     release(lock)
     consume(item)
  }
}
```

# (Good) Producer consumer **using semaphores**

```
sem_init(not_full, n)
sem_init(not_empty, 0)
```

```
void producer () {
  while(1){
     item := produce()
     sem_wait(not_full)
     write(buffer, item)
     sem_signal(not_empty)
  }
}
```

```
void consumer () {
  while(1){
     sem_wait(not_empty)
     item := read(buffer)
     sem_signal(not_full)
     consume(item)
  }
}
```

➡ What if we have multiple consumers and/or producers?

# (Bad) Producers consumers

```
sem_init(not_full, n)
sem_init(not_empty, 0)
sem_init(mutex, 1)
```

```
void producer () {
  while(1){
     item := produce()
     sem_wait(mutex)
     sem_wait(not_full)
     write(buffer, item)
     sem_signal(not_empty)
     sem_signal(mutex)
  }
}
```

```
void consumer () {
  while(1){
     sem_wait(mutex)
     sem_wait(not_empty)
     item := read(buffer)
     sem_signal(not_full)
     sem_signal(mutex)
     consume(item)
  }
}
```

- **Deadlock** : the producer waits for the consumer to release `mutex` while the consumer waits for producer to release `not_empty` (or vice versa)

# Deadlock

**Deadlock** when one thread tries to access a resource that a second  holds, and vice-versa

- They can never make progress

```
void thread1 (){
      ...
      sem_wait(sem1)
      sem_wait(sem2)
      /* critical section */
      sem_signal(sem2)
      sem_ignal(sem1)
      ...
}
```

```
void thread2(){
      ...
      sem_wait(sem2)
      sem_wait(sem1)
      /* critical section */
      sem_signal(sem1)
      sem_signal(sem2)
...
}
```

# (Good) Producers consumers

```
sem_init(not_full, n)
sem_init(not_empty, 0)
sem_init(mutex, 1)
```

```
void producer () {
  while(1){
     item := produce()
     sem_wait(not_full)
     sem_wait(mutex)
     write(buffer, item)
     sem_signal(mutex)
     sem_signal(not_empty)
  }
}
```

```
void consumer () {
  while(1){
     sem_wait(not_empty)
     sem_wait(mutex)
     item := read(buffer)
     sem_signal(mutex)
     sem_signal(not_full)
     consume(item)
  }
}
```

# How to avoid deadlocks

**Avoiding deadlock** using primitive synchronization mechanisms (locks and semaphores) **is hard** (cf chapter 32)

➡ Need higher abstraction synchronization mechanisms

- Condition variable
- Monitor

# Condition Variable

**A condition variable** supports three operations

- **`cond_wait(cond, mutex)`**
  unlock the `mutex` lock and sleep until `cond` is signaled then re-acquire `mutex` before resuming execution
- **`cond_signal(cond)`**
  signal the condition `cond` by waking up the next thread
- **`cond_broadcast(cond)`**
  signal the condition `cond` by waking up all threads

# Producers consumers **using a condition variable**

```
cond_init(not_full)
cond_init(not_empty)
```

```
void producer () {
 while(1){
  item := produce()
  acquire(mutex)
  while(!empty(buffer))
     cond_wait(not_full, mutex)
  write(buffer, item)
  cond_signal(not_empty)
  release(mutex)
 }
}
```

```
void consumer () {
 while(1){
  acquire(mutex)
  while(empty(buffer))
     cond_wait(not_empty, mutex)
  item := read(buffer)
  cond_signal(not_full)
  release(mutex)
  consume(item)
 }
}
```

# Monitor

**A monitor** is a programming language construct that encapsulates

- shared data structures
- procedures that operate on the shared data structures
- synchronization between concurrent threads that invoke the procedures

➡ A monitor guarantees mutual exclusion

- only one thread at a time can execute a monitor procedure
- all others are blocked in a queue

# Producers consumers **using a monitor**

```
Monitor sync_buffer {

    item buffer[N]

    void produce(item){
         write(buffer, item)
    }

    item consume(void){
         return read(buffer)
    }
}
```

# Other interesting problems

# Readers Writers

➡ allow multiple readers but only one writer in the critical section

```
void writer () {
  while(1){
     write(file, data);
  }
}
```

```
void reader () {
  while(1){
     data:= read(file);
  }
}
```

# Solution

1. **`readcount`** (variable) to keep track of the number of readers currently reading

2. **`mutex`** (binary semaphore) to synchronize the access to `readcount`

3. **`writer_or_readers`** (binary semaphore) to provide exclusive access to each writer or all readers

   - writer should wait before writing and signal after
   - readers should wait when `readcount` goes from 0 to 1 and signal when `readcount` goes from 1 to 0

# Readers Writers

```
readcount = 0
sem_init(mutex, 1)
sem_init(writer_or_readers, 1)
```

```
void writer () {
  while(1){
    sem_wait(writer_or_readers)
    write(file, data)
    sem_signal(writer_or_readers)
  }
}
```

```
void reader () {
 while(1){
   sem_wait(mutex)
   readcount += 1;
   if (readcount == 1)
     sem_wait(writer_or_readers)
   sem_signal(mutex)
   data:=read(file)
   sem_wait(mutex)
   readcount -= 1;
   if (readcount == 0)
     sem_signal(writer_or_readers)
   sem_signal(mutex)
  }
}
```

- **Writers starvation!**

# Readers Writers

```
readcount = 0
sem_init(mutex, 1)
sem_init(writer_or_readers, 1)
sem_init(service, 1)
```

```
void writer () {
  while(1){
    sem_wait(service)
    sem_wait(writer_or_readers)
    sem_signal(service)
    write(file, data)
    sem_signal(writer_or_readers)
  }
}
```

```
void reader () {
 while(1){
  sem_wait(service)
  sem_wait(mutex)
  readcount += 1;
  if (readcount == 1)
    sem_wait(writer_or_readers)
  sem_signal(service)
  sem_signal(mutex)
  data:=read(file)
  sem_wait(mutex)
  readcount -= 1;
  if (readcount == 0)
    sem_signal(writer_or_readers)
  sem_signal(mutex)
```

# Dining Philosophers

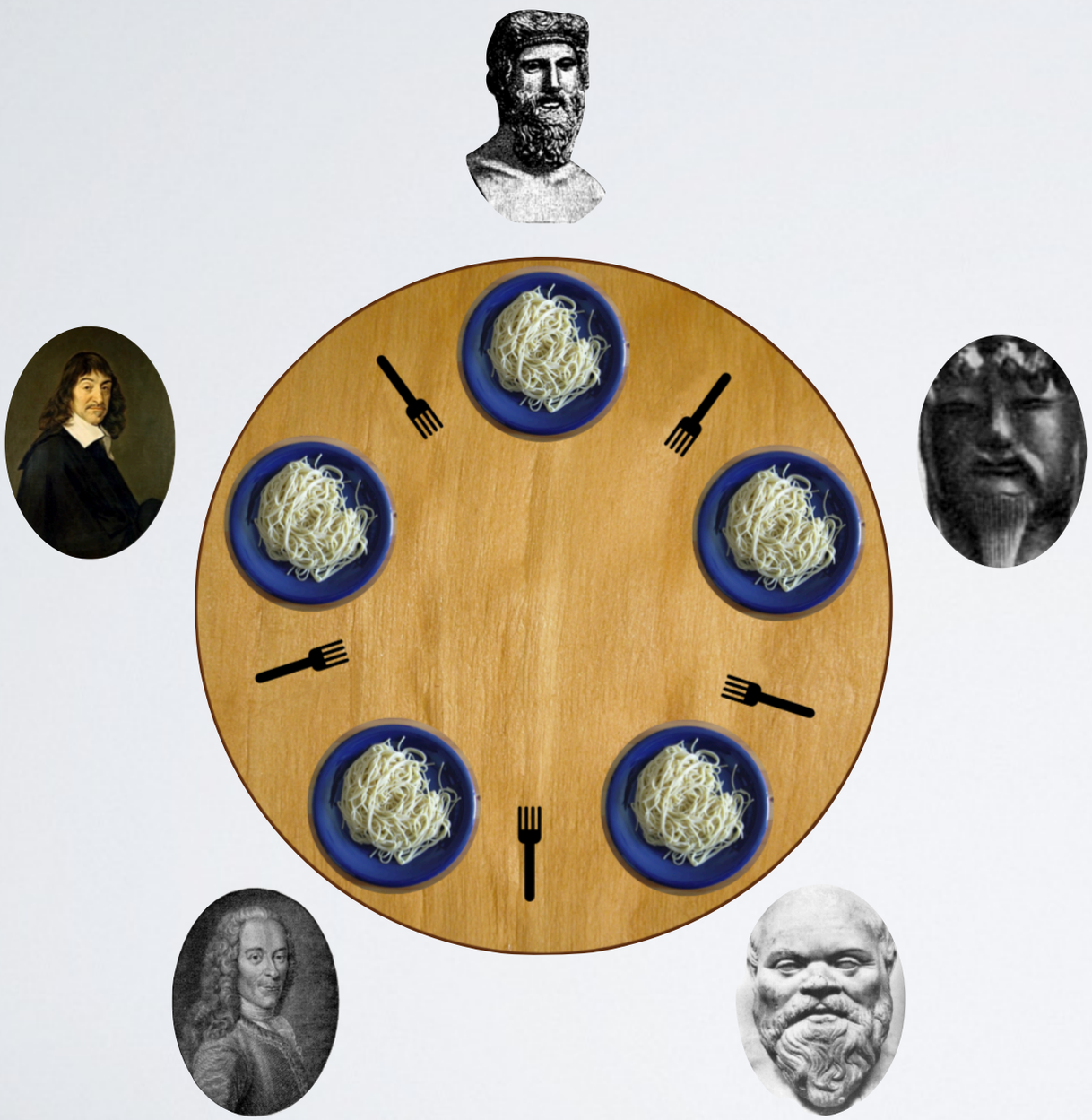

image from *wikipedia*

```
void philosopher (i, n) {
  while(1){
     grab_fork(i)
     grab_fork((i + 1)% n)
     eat & think
     drop_fork(i)
     drop_fork((i + 1)% n)
   }
}
```

# (Bad) Dining Philosophers

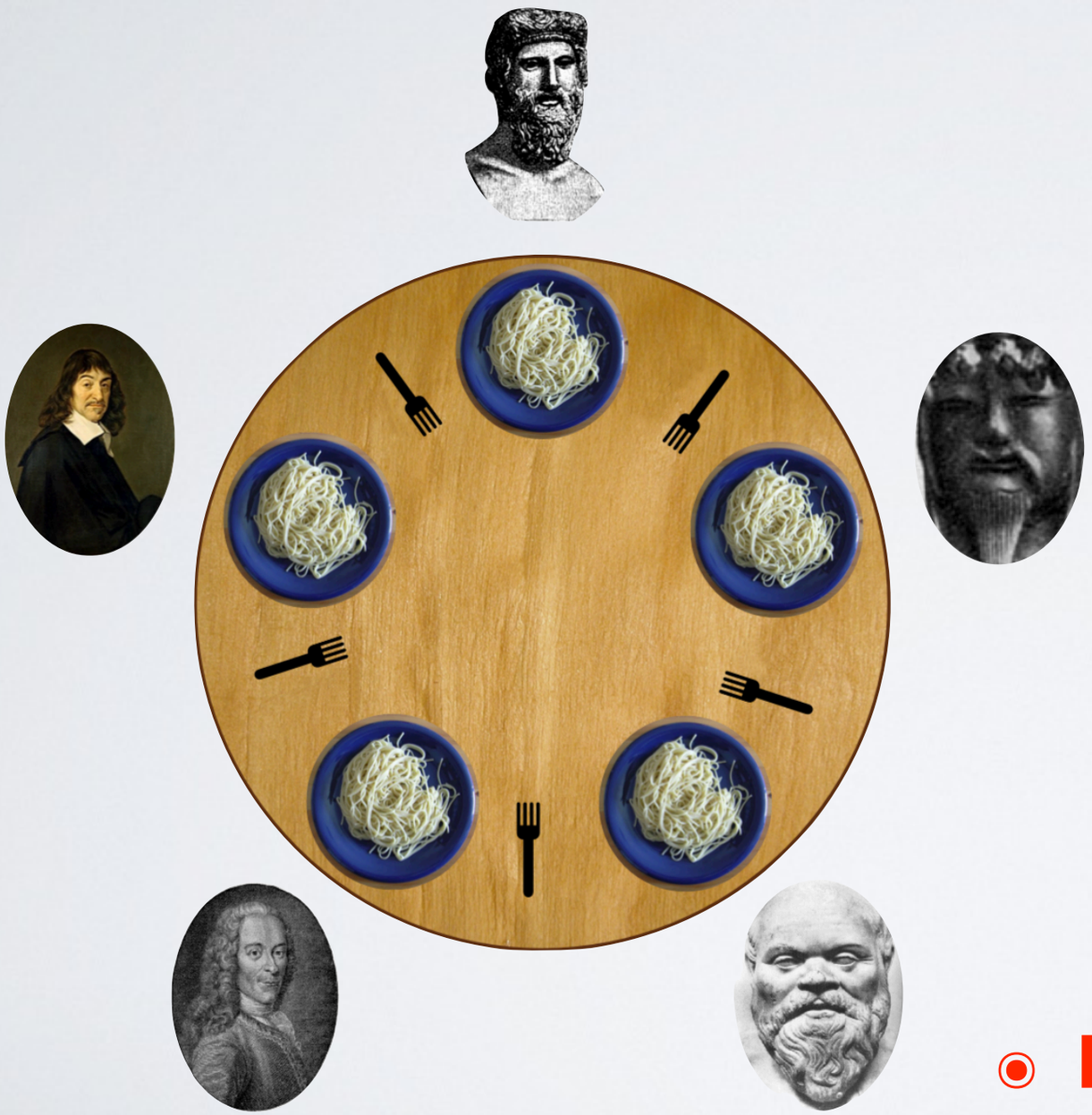

image from *wikipedia*

```
for(i=0, i<n, i++){
   sem_init(fork[i], 1)
}
```

```
void philosopher (i, n) {
  while(1){
     sem_wait(fork[i])
     sem_wait(fork[(i + 1)% n])
     eat & think
     sem_signal(fork[i])
     sem_signal(fork[(i + 1)% n])
   }
}
```

- **Deadlock** when each philosopher take the first fork "at the same time"

# (Good) Dining Philosophers

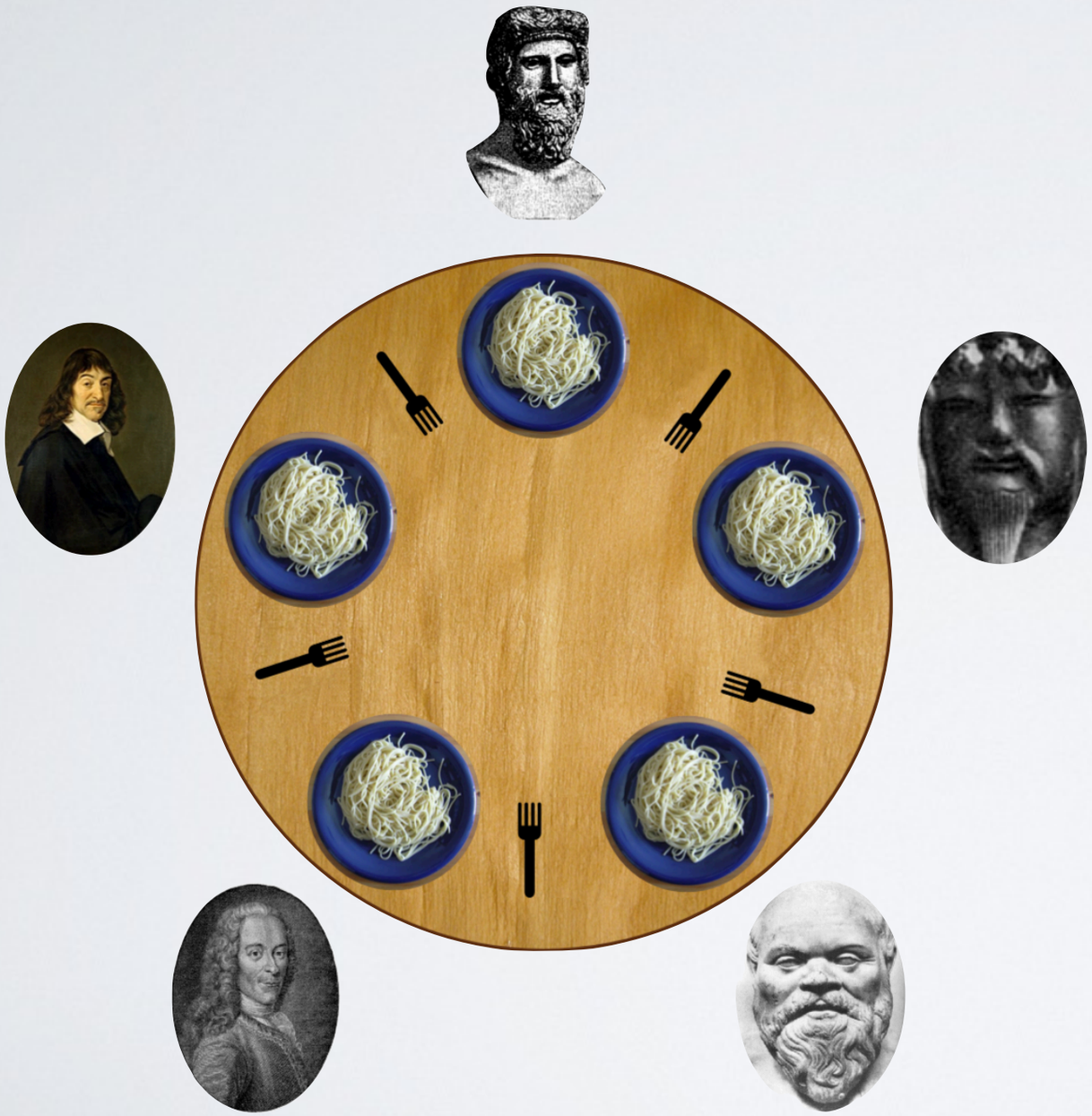

image from *wikipedia*

```
for(i=0, i<n, i++){
   init(fork[i], 1)
}
```

```
void philosopher (i, n) {
  while(1){
    if ((i+1) == n){
      sem_wait(fork[(i + 1)% n])
      sem_wait(fork[i])
    }else{
      sem_wait(fork[i])
      sem_wait(fork[(i + 1)% n])
    }
    eat & think
    sem_signal(fork[i])
    sem_signal(fork[(i + 1)% n])
  }
}
```

# More problems

# The cigarette smokers problem

Assume a cigarette requires three ingredients to make and smoke: tobacco, paper, and matches. There are three smokers around a table, each of whom has an infinite supply of one of the three ingredients — one smoker has an infinite supply of tobacco, another has paper, and the third has matches.

There is also a non-smoking agent who enables the smokers to make their cigarettes by arbitrarily (non-deterministically) selecting two of the supplies to place on the table. The smoker who has the third supply should remove the two items from the table, using them (along with their own supply) to make a cigarette, which they smoke for a while. Once the smoker has finished his cigarette, the agent places two new random items on the table.

# The barbershop problem

A barbershop consists of a waiting room with 3 chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

# The river crossing problem

Somewhere near Redmond, Washington there is a boat that is used by both Linux hackers and Microsoft employees to cross a river. The boat can hold exactly 4 people and it won't leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker with 3 employees, or to put one employee with 3 hackers. Any other combination is safe.

# The unisex bathroom

A startup company has 2 bathrooms in its building, one for the men and one for the women. Due to maintenance work, one of the bathroom is closed for a month. During that period, the employees have to use the same bathroom. Indeed, there cannot be men and women in the bathroom at the same time and there should never be more than 5 employees squandering company time in the bathroom.

# The room party problem

Often, the students organize parties in the student lounge. The Dean is allowed to enter the student lounge in 2 cases: 1) when there are no students in the room to do some security checks or 2) when there are more than 50 students in the room to break up the party. In both cases, no additional students can come in while the Dean is in the room. In the second case, the Dean leaves the room only when all students have left.