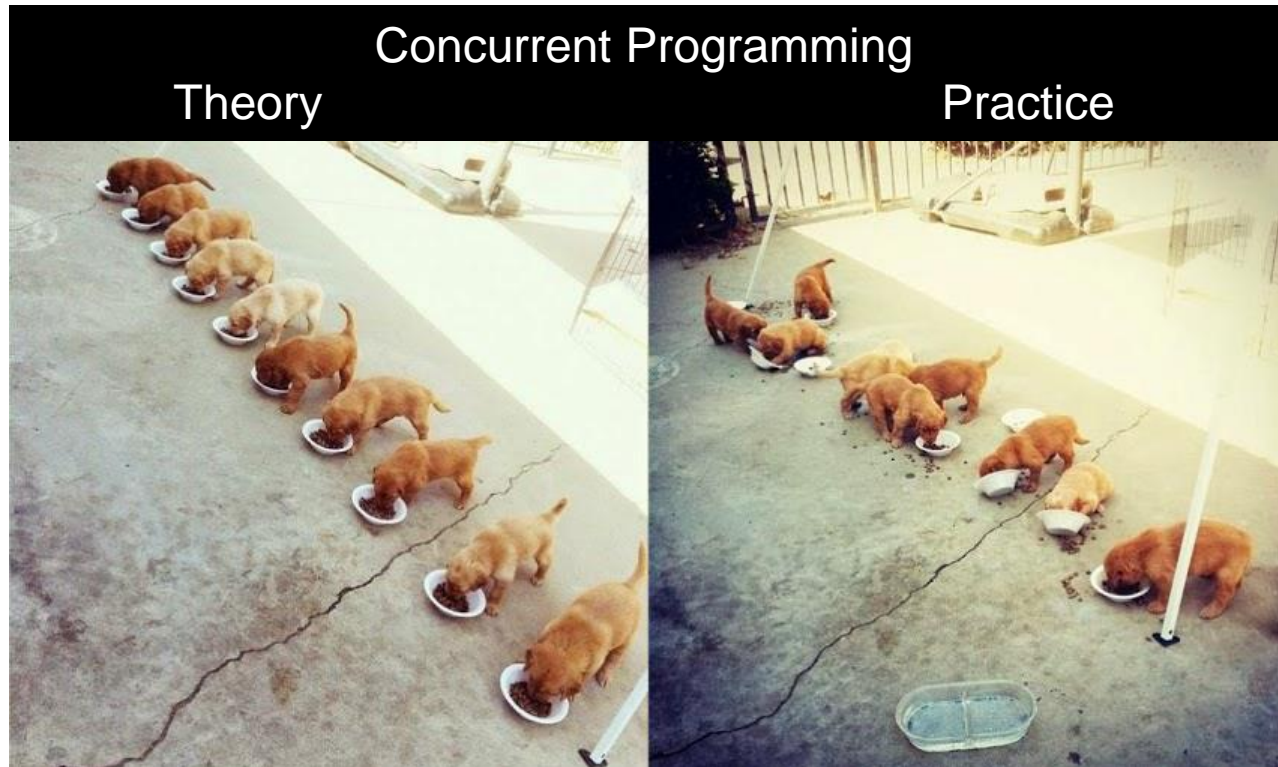


Advanced Solutions to the Critical Section Problem



Dr Amanda Coles

Chapters 6 of "Operating Systems Concepts" Silberschatz, Galvin, Gagne

Chapters 5 and 6 of "Principles of Concurrent and Distributed Programming" M. Ben-Ari

Overview

- The Critical Section Problem with n Processes;
 - Bakery Algorithm;
- Synchronisation Hardware;
- Semaphores:
 - The Producer Consumer Problem;
 - Infinite buffer;
 - Finite buffer;
 - The Dining Philosophers Problem;
- Semaphores in Java.

Advanced Algorithms

- So far all of our solutions to the critical section problem have considered two threads.
- Now we will consider algorithms that work for n -threads:
 - After all, this is more realistic.
- We will consider some primitives provided by operating systems to implement synchronisation.

Bakery Algorithm

- Our first algorithm is the Bakery Algorithm.
 - Inspired by service in bakeries (and other shops) where customers take a ticket on arrival and are called to be served in order of ticket number.
- Upon entering customers take a ticket, and when service is available the customer with the lowest ticket number is served.

The Bakery Algorithm for 2 Threads

integer np \leftarrow 0, integer nq \leftarrow 0	
p	q
<pre>loop_forever p1: non-critical section p2: np \leftarrow nq + 1 p3: await nq = 0 or np \leq nq p4: critical section p5: np \leftarrow 0</pre>	<pre>loop_forever p1: non-critical section p2: nq \leftarrow np + 1 p3: await np = 0 or nq < np p4: critical section p5: nq \leftarrow 0</pre>

- Simple version for 2 threads;
- np and nq are the ticket numbers.
 - Zero indicates thread does not want to enter its critical section.
 - A positive number indicates the process is in the queue to enter, the queue is sorted lowest number first.
- For equal ticket numbers we will arbitrarily favour p (p3 \leq , q3 <).

Proofs of Properties

- In this week's exercises!

The Bakery Algorithm for N Threads

integer array[1...N] number \leftarrow [0,...,0]

loop_forever

p1: non-critical section

p2: number[i] \leftarrow 1 + max(number)

P3: for all *other* processes j

p3: await number[j] = 0 or number[i] \ll number[j]

p4: critical section

p5: number[i] \leftarrow 0

- Each thread has a unique id, i , in the range [0..N].
- **number[i] \ll number[j]** means either:
 - number[i] < number[j] or:
 - number[i] = number[j] and $i < j$.
- For equal ticket numbers we will arbitrarily favour lower id thread.

The Bakery Algorithm for N Threads

integer array[1...N] number \leftarrow [0,...,0]

loop_forever

p1: non-critical section

p2: number[i] \leftarrow 1 + max(number)

P3: for all *other* processes j

p3: await number[j] = 0 or number[i] \ll number[j]

p4: critical section

p5: number[i] \leftarrow 0

- Pick a ticket number that is larger than any currently held ticket.
- Enter critical section when no other process is waiting to enter (value is zero) or none holds a lower numbered ticket (break ties on id);
- Set ticket number to zero (I am not waiting enter CS).

Note on Atomicity for the Astute

- There is a big assumption here:
 - Finding the max of an array is an atomic operation.
- This can be fixed, but the algorithm is more complicated.

integer array[1...N] number \leftarrow [0,...,0]

loop_forever

p1: non-critical section

P2: choosing[i] \leftarrow true

p3: number[i] \leftarrow 1 + max(number)

p4: choosing[i] \leftarrow false

p5: for all *other* processes j

p6: await choosing[j] \leftarrow false

p7: await number[j] = 0 or number[i] \ll number[j]

p8: critical section

p9: number[i] \leftarrow 0

The Good and The Bad

- Good:
 - No variable is written to by more than one process;
 - Mutual Exclusion, Freedom from Deadlock and Freedom for Starvation hold.
- Bad:
 - Unbounded ticket numbers;
 - Each process must query all others to see if it can enter CS, even if no others want to enter.
- Overall: algorithm too inefficient to use in practice.

The Fast Algorithm (Optional Reading)

Algorithm 5.6: Fast algorithm for two processes

integer gate1 \leftarrow 0, gate2 \leftarrow 0

boolean wantp \leftarrow false, wantq \leftarrow false

p

p1: gate1 \leftarrow p
 wantp \leftarrow true
p2: if gate2 \neq 0
 wantp \leftarrow false
 goto p1
p3: gate2 \leftarrow p
p4: if gate1 \neq p
 wantp \leftarrow false
 await wantq = false
p5: if gate2 \neq p goto p1
 else wantp \leftarrow true
 critical section
p6: gate2 \leftarrow 0
 wantp \leftarrow false

q

q1: gate1 \leftarrow q
 wantq \leftarrow true
q2: if gate2 \neq 0
 wantq \leftarrow false
 goto q1
q3: gate2 \leftarrow q
q4: if gate1 \neq q
 wantq \leftarrow false
 await wantp = false
q5: if gate2 \neq q goto q1
 else wantq \leftarrow true
 critical section
q6: gate2 \leftarrow 0
 wantq \leftarrow false

Overview

- The Critical Section Problem with n Processes;
 - Bakery Algorithm;
- Synchronisation Hardware;
- Semaphores:
 - The Producer Consumer Problem;
 - Infinite buffer;
 - Finite buffer;
 - The Dining Philosophers Problem;
- Semaphores in Java.

Synchronisation Hardware

- So far we have been assuming an atomic assignment operator.
 - This would, of course, have to be provided at a lower level, e.g. OS or hardware.
- Instead we can consider a more general idea:
 - A solution to the critical section problem requires a **lock**: a thread must obtain the lock in order to enter its critical section.
- In fact, hardware can provide us with primitives to allow us to do synchronisation much more easily.

Single vs Multi-Processor Solutions

- In a single processor environment we could solve the critical section problem simply by disabling interrupts whilst shared data is being modified.
- In a multi-processor environment however:
 - Message has to be passed to each processor every time a CS is entered: slows down system.
 - Also disabling interrupts affects the system clock if the clock is kept up to date by interrupts.

Test and Set

boolean lock \leftarrow false

p

loop_forever

p1: non-critical section

p2: await(!testAndSet(lock))

p3: critical section

p4: lock = false;

```
boolean testAndSet(Boolean lock){
    boolean toReturn =
        lock.getValue();
    lock.setValue(true);
    return toReturn;
}
```

- Single processor instruction that will both test the value of a variable and set it to a new value.
- Test and set will return the value of the lock (false means no one is in cs) and set the value of the lock to true;
 - Does no harm if it is already true
- If two processors attempt to execute this instruction simultaneously the hardware will ensure the instructions will be executed sequentially (in some arbitrary order);
- Works for n processes: atomic assignment no longer required.

Swap

```
boolean lock ← false
```

```
p  
boolean key ← true
```

```
loop_forever  
p1: non-critical section  
P2: key = true  
p3: while(key = true)  
    swap(lock, key);  
p4: critical section  
p5: lock = false;
```

```
void swap(Boolean lock,  
           Boolean key){  
    boolean tmp = lock.getValue();  
    lock.setValue(key.getValue());  
    key.setValue(tmp);  
}
```

- Single processor instruction that will swap the value of two booleans.
- Again hardware ensures this instruction is not simultaneously executed.
- Works for n processes: atomic assignment no longer required.
- N.B. Both of these algorithms could lead to starvation but it is unlikely (see Section 6.4 of “Operating Systems Concepts” for details of how to get around this).

Overview

- The Critical Section Problem with n Processes;
 - Bakery Algorithm;
- Synchronisation Hardware;
- **Semaphores:**
 - The Producer Consumer Problem;
 - Infinite buffer;
 - Finite buffer;
 - The Dining Philosophers Problem;
- Semaphores in Java.

Semaphores

- Application programmers do not typically reason with hardware-level instructions;
- Instead higher-level software mechanisms are generally provided that make use of the lower-level hardware instructions.
 - Less prone to error as more abstract: less complex;
 - Nicer to work with.
- Semaphores are an example of a commonly used synchronisation mechanism.
 - Provided by most operating systems.
- Proposed by Dijkstra in 1965.

What is a Semaphore (Busy-Wait)?

- A semaphore comprises:
 - An integer variable, v ;
 - (Optionally) A set of processes, blocked, (initially empty).
- A semaphore has two methods:

Naïve Version (busy wait):

```
Wait(S) {  
    while(S.v <= 0) {  
        //do nothing  
    }  
    S.v = S.v - 1;  
}
```

Naïve Version (busy wait):

```
Signal(S) {  
    S.v = S.v + 1;  
}
```

`while (S.v <= 0) { }` Also called a *spin lock*.

What is a Semaphore (Blocked-Set)?

- A semaphore comprises:
 - An integer variable, v ;
 - A set of processes, blocked, (initially empty).
- A semaphore has two methods:

```
Wait(S) {  
    S.v = S.v - 1;  
    if(S.v < 0) {  
        S.blocked = S.blocked U this;  
        this.state = blocked;  
    }  
}
```

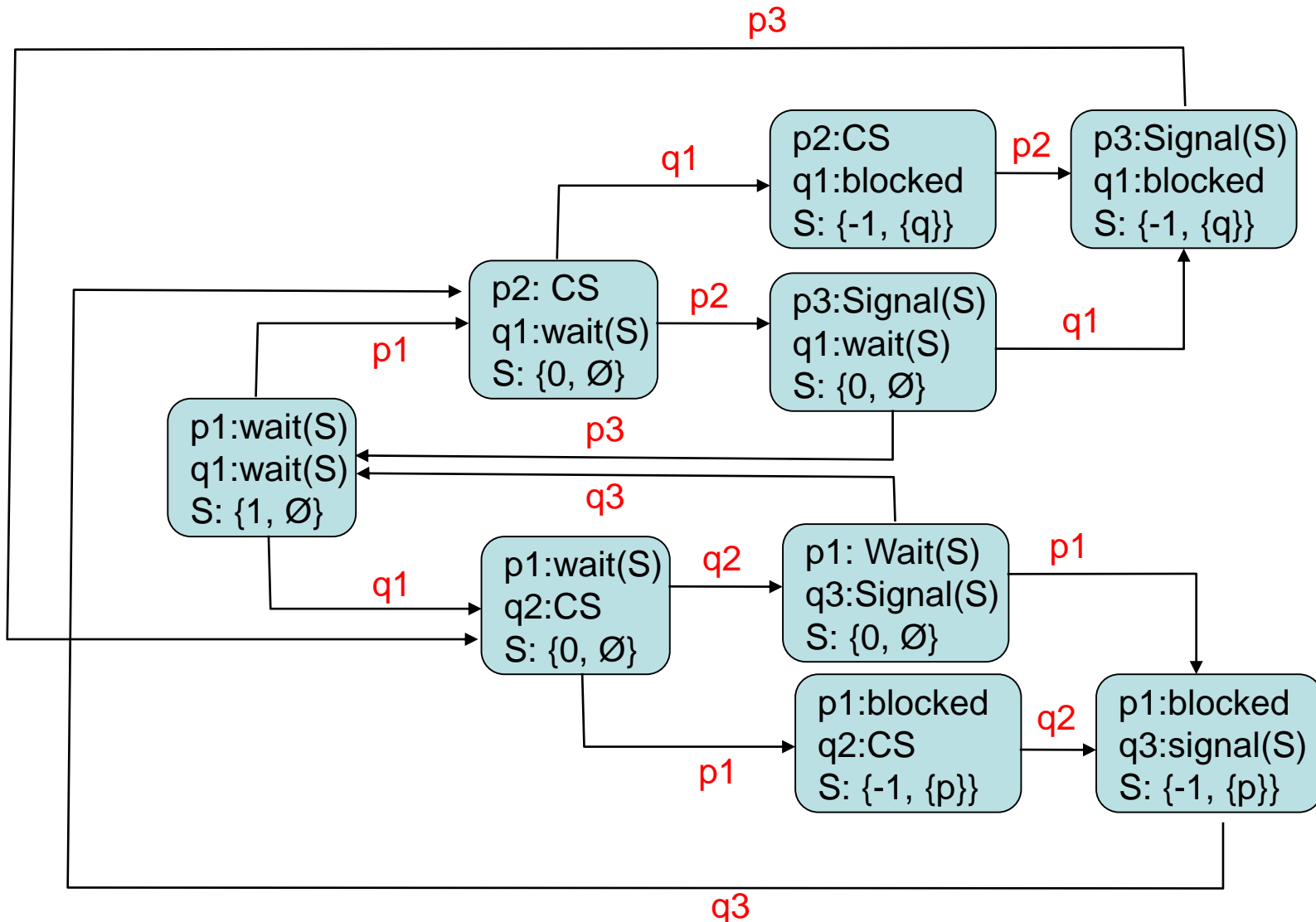
```
Signal(S) {  
    S.v = S.v + 1;  
    select one q from S.blocked:  
        q.state = runnable;  
}
```

Critical Section Problem for 2 Threads with Semaphores

Semaphore $S \leftarrow (1, \emptyset)$	
p	q
loop_forever p0: non-critical section p1: wait(S) p2: critical section p3: signal(S)	loop_forever q0: non-critical section q1: wait(S) q2: critical section q3: signal(S)

- Initially $S.v = 1$;
- Before a thread enters its critical section:
 - $S.v$ will be decremented: zero means some thread is in cs.
 - So if the other process calls $\text{wait}(S)$ $S.v = 0$ and it will be blocked.
- When a process leaves its critical section:
 - Signal adds 1 to $S.v$ so that when a process leaves its critical section $S.v = 1$ and other processes can enter.
 - If the other process is waiting it is unblocked.

State Diagram (Excl. non-cs)



No state (p2,q2,_), therefore **mutual exclusion** holds.
 No state in which both are blocked therefore **no deadlock**.

Starvation

- When a process is awoken it leaves the blocked state and continues straight into its critical section (setting $S.v$ to 1).
- The absence of starvation relies on this:
 - Otherwise if the thread wasn't scheduled before the other thread, then the other thread could be scheduled to run and complete $\text{wait}(S)$ first each time (this is unlikely but possible: as Attempt 4 in the previous lecture a continuous execution of the same interleaving).

Properties of Busy-Wait Semaphores

Semaphore $S \leftarrow (1, \emptyset)$	
p	q
<pre>loop_forever p1: non-critical section p2: wait(S) p3: critical section p4: signal(S)</pre>	<pre>loop_forever q1: non-critical section q2: wait(S) q3: critical section q4: signal(S)</pre>

- $S.v > 0$ always;
- $S.v = \text{init} + \# \text{signal}(S) - \# \text{wait}(S)$;
- $\# \text{CS} + S.v = 1$;
- $\# \text{CS} = \# \text{wait}(S) - \# \text{signal}(S)$.

Solution for N Processes

Semaphore $S \leftarrow (1, \emptyset)$
p
<pre>loop_forever p1: non-critical section p2: wait(S) p3: critical section p4: signal(S)</pre>

- Mutual Exclusion and Freedom from Deadlock hold by arguments applied to 2 processes, and properties of semaphores.
- Let's take a look at starvation.

Scenario for Starvation

Thread p	Thread q	Thread r	S
p1: wait(S)	q1: wait(S)	r1: wait(S)	(0, ∅)
p2: critical-section	q1: wait(S)	r1: wait(S)	(-1, {q})
p2: critical-section	q1: blocked	r1: wait(S)	(-2, {q, r})
p2: critical-section	q1: blocked	r1: blocked	(-2, {q, r})
p3: signal(S)	q1: blocked	r1: blocked	(-2, {q, r})
p1: wait(S)	q1: blocked	r2: critical-section	(-1, {q})
p1: wait(s)	q1: blocked	r3: signal(S)	(-2, {p, q})
p1: blocked	q1: blocked	r3: signal(S)	(-2, {p, q})
p2: critical-section	q1: blocked	r1: wait(S)	(-1, {q})
p3: signal(S)	q1: blocked	r1: wait(S)	(-2, {q, r})
p3: signal(S)	q1: blocked	r1: blocked	(-2, {q, r})

- Again requires a specific interleaving so improbable but possible;
- Can fix this by using a queue (FIFO) in the semaphore instead of a set (blocked-queue semaphore).

Waiting for Other Threads Using Semaphores

Semaphore $S1 \leftarrow (0, \emptyset)$, Semaphore $S2 \leftarrow (0, \emptyset)$		
p	q	r
p1: non-critical section p2: signal(S1)	p1: non-critical section p2: signal(S2)	r1: wait(S1) r2: wait(S2) r3: continue

- Similar to join() but r did not start p and q.
- Can be used when one thread might need to wait for another:
 - e.g. mergesort:
 - Split array in half;
 - P sorts one half, q sorts the other half
 - R does merging after completion

b. This part of the question is about Semaphores.

i. Describe the functions wait and signal in a busy-wait semaphore and explain briefly how they can be used to solve the critical section problem.

[5 marks]

ii. Why might a busy-wait semaphore be an inefficient way (in terms of CPU utilisation) to solve the critical section problem?

[2 marks]

iii. Explain how a blocked-queue semaphore differs from a busy-wait semaphore (including changes to the semaphore functions) and why these differences avoid the above efficiency problems that busy-wait semaphores suffer.

[6 marks]

Overview

- The Critical Section Problem with n Processes;
 - Bakery Algorithm;
- Synchronisation Hardware;
- **Semaphores:**
 - The Producer Consumer Problem;
 - Infinite buffer;
 - Finite buffer;
 - The Dining Philosophers Problem;
- Semaphores in Java.

The Producer Consumer Problem

- A ***producer*** process creates data;
- A ***consumer*** process takes the data and uses it;
- This pairing is common in computer science.
- For ***asynchronous*** communication we need a buffer.
- Two issues can occur:
 - Buffer is empty, consumer cannot take data;
 - Buffer is full, producer cannot add data;

Infinite Buffer Producer/Consumer

Semaphore notEmpty $\leftarrow (0, \emptyset)$, infinite queue<d> buffer \leftarrow empty	
producer	consumer
<pre>loop_forever p1: d \leftarrow produce p2: append(d,buffer) p3: signal(notEmpty)</pre>	<pre>loop_forever q1: wait(notEmpty) q2: d \leftarrow take(buffer) q3: consume(d)</pre>

- Infinite buffer case: only need to synchronise removal.
- Notice the semaphore is initialised to 0, not 1 as in previous examples;
- This is because we don't want wait to be able to consume until something has been added to the buffer;
- Here the semaphore value is 0 if the buffer is empty.

Finite Buffer Producer/Consumer

Semaphore notEmpty $\leftarrow (0, \emptyset)$, Semaphore notFull $\leftarrow (N, \emptyset)$,
size N queue<d> buffer \leftarrow empty

producer	consumer
<pre>loop_forever p1: d \leftarrow produce P2: wait(notFull) p3: append(d,buffer) p4: signal(notEmpty)</pre>	<pre>loop_forever q1: wait(notEmpty) q2: d \leftarrow take(buffer) q3: signal(notFull) q4: consume(d)</pre>

- Finite buffer case: create an analogy, producer makes non-empty buffers for the consumer; consumer makes non-full buffers for the producer.
- Notice the notFull semaphore is initialised to N, so it can be decremented N times (i.e. N things can be added to the buffer) before it reaches zero.
- This technique is called split semaphores (we also used it to wait for other threads).

Initialising Semaphores to Values Other than 1 or 0

Semaphore $S \leftarrow (N, \emptyset)$
p
<pre>loop_forever p1: non-critical section p2: wait(S) p3: critical section p4: signal(S)</pre>

- Now N threads can successfully proceed through wait, before one is blocked to call signal: e.g. $N = 3$, 3 threads can enter at once.
- Clearly this doesn't enforce mutual exclusion, but it does allow a limit on the number of threads that could be in their critical section.
- Can be useful if e.g. want to limit number of processes which can have a file open, or can connect to a server, or perhaps we have 3 instances of a resource, so 3 threads can use the resource at once.

Types of Semaphore

- ***Busy Wait Semaphore:*** the first type we saw where a process continually loops checking `s`;
- ***Blocked-Set Semaphore (weak semaphore):*** the type of semaphore we have been using, stores blocked processes in a set and wakes one arbitrarily;
- ***Blocked-Queue Semaphore (strong semaphore):*** uses a queue instead of a set.

Overview

- The Critical Section Problem with n Processes;
 - Bakery Algorithm;
- Synchronisation Hardware;
- **Semaphores:**
 - The Producer Consumer Problem;
 - Infinite buffer;
 - Finite buffer;
 - The Dining Philosophers Problem;
- Semaphores in Java.

The Dining Philosophers Problem

- Canonical problem in concurrent programming literature;
- 5 Philosophers sit around a table.
- Each alternates between thinking and eating.
- Caveat:
 - A philosopher requires two forks to eat.
 - There are only 5 forks on the table, one between each pair of philosophers.
- Not entirely a compelling problem, but illustrates some important issues.

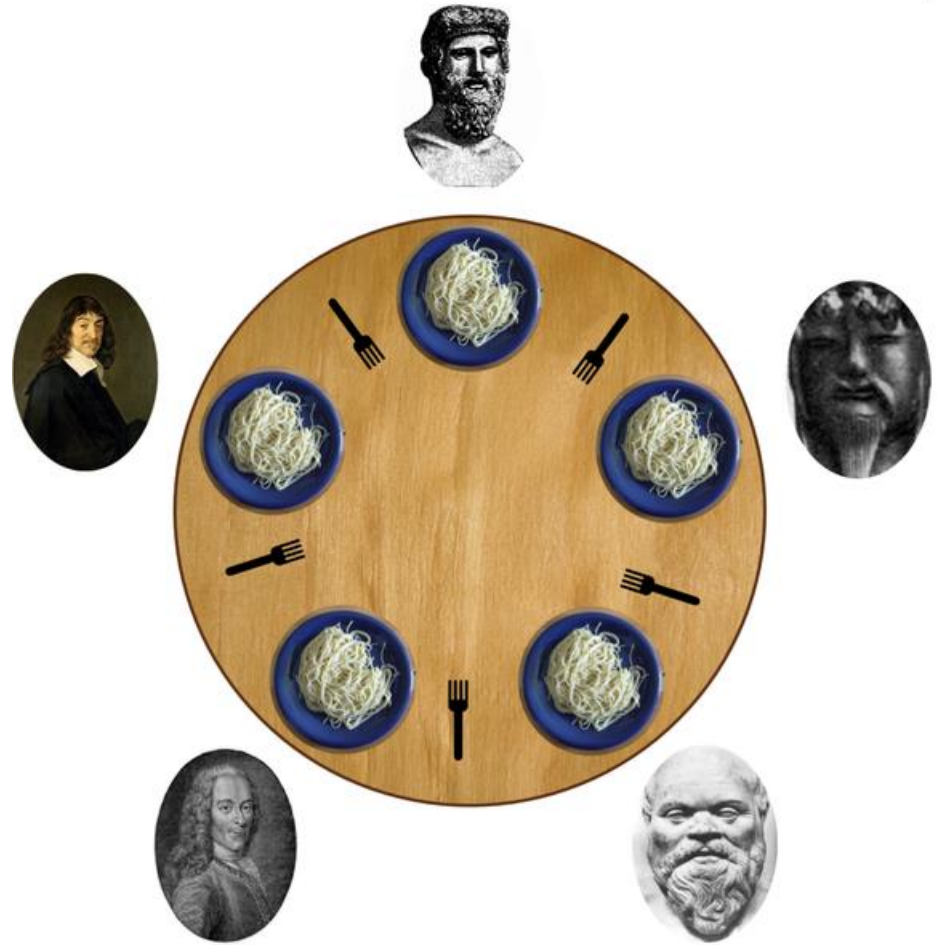


Image: Wikipedia

Philosophers as Threads

Philosopher
<code>loop_forever</code>
<code>p1: think</code>
<code>p2: preprotocol</code>
<code>p3: eat</code>
<code>P4: postprotocol</code>

- No two philosophers can hold the same fork simultaneously.
- Each philosopher can pick up the fork on his right or left but only one at a time (mutual exclusion)
- Freedom from starvation (literally)
- Freedom from deadlock.

First Proposed Solution

Semaphore array fork $\leftarrow [1,1,1,1,1]$
<pre>loop_forever p1: think p2: wait(fork[i]) //left fork p3: wait(fork[i+1]) //right fork p4: eat p5: signal(fork[i+1]) P6: signal(fork[i])</pre>

- Each fork can now only be held by one philosopher.
- What happens if the execution trace is:
 - p1,q1,r1,s1,t1,p2,q2,r2,s2,t2?

Second Attempt

Semaphore array fork $\leftarrow [1,1,1,1,1]$,
Semaphore room $\leftarrow 4$

```
loop_forever
p1:  think
P2:  wait(room)
p3:  wait(fork[i]) //left fork
p4:  wait(fork[i+1]) //right fork
p5:  eat
p6:  signal(fork[i+1])
P7:  signal(fork[i])
P8:  signal(room)
```

- Now only 4 philosophers may enter the room at once.
- Solves the problem, but not very sociable.

Freedom From Starvation

- If philosopher 1 is starved then they are waiting forever for a semaphore.
- **Case fork[1] (left fork):** this means that philosopher 0 is holding fork[0] as a right fork, i.e. also already has a left fork and is eating, therefore will eventually signal fork[1] so this philosopher can continue.
- **Case fork[2] (right fork):** this means that philosopher 2 must have taken their left fork (fork[2]) and, **if they cannot proceed**, must be unable to take their right fork, fork[3].
 - This in turn means p3 has taken fork[3] and cannot get fork[4];
 - And that p4 has taken fork[4] and cannot get fork[0];
 - And that p0 has taken fork[0] and cannot get fork[1];
 - But we know that not all philosophers are in the room due to the room semaphore so this cannot be the case.
- **Case room (out of the room):** if we ensure that room is a blocked queue semaphore then because none of the philosophers can be blocked indefinitely when in the room (see above) one will eventually complete and signal room, allowing the other to enter.

A More Sociable Solution

Semaphore array fork $\leftarrow [1,1,1,1,1]$
Philosopher 4
<pre>loop_forever p1: think p2: wait(fork[0]) //right fork p3: wait(fork[4]) //left fork p4: eat p5: signal(fork[0]) P6: signal(fork[4])</pre>

- This is an asymmetric solution: philosopher 4 picks up right then left, all others left then right.
- Exercise: show that this is starvation free.

4. a. Consider the following threads, all of which are to be executed concurrently.

Semaphore $s1 = 0$, $s2 = 0$, $s3 = 1$, $\text{int } v \leftarrow 4$		
p	q	r
p1: wait($s1$)	q1: wait($s2$)	r1: wait($s3$)
p2: $v = v/2$	q2: wait($s3$)	r2: $v = v * 3$
p3: $v = v * 4$	q3: $v = v + 4$	r3: signal($s3$)
p4: signal($s2$)	p5: signal($s2$)	r4: signal($s1$)
		r5: signal($s2$)

- i. Assuming each line of the code is executed atomically (including the division, addition and multiplication statements) what are the possible values of the variable v upon a successful complete execution of this program? Explain how the execution of the code can give rise to these values.

[7 marks]

- ii. What problem could occur if the multiplication operator was not atomic? Explain your answer.

[4 marks]

Overview

- The Critical Section Problem with n Processes;
 - Bakery Algorithm;
- Synchronisation Hardware;
- Semaphores:
 - The Producer Consumer Problem;
 - Infinite buffer;
 - Finite buffer;
 - The Dining Philosophers Problem;
- **Semaphores in Java.**

Semaphores in Java

- API Documentation:
 - <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html>
 - `import java.util.concurrent.Semaphore;`
- Constructor:
 - `Semaphore(int permits)` Creates a Semaphore with the given number of permits and nonfair fairness setting.
 - `Semaphore(int permits, boolean fair)` Creates a Semaphore with the given number of permits and the given fairness setting.
- Important Methods:
 - void `acquire()` Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted. Throws `InterruptedException`
 - void `release()` Releases a permit, returning it to the semaphore.
 - N.B. `acquire()` = wait, `release()` = signal.
 - boolean `isFair()` Returns true if this semaphore has fairness set true.

Example 1: Producer Consumer

```
public class Buffer {  
  
    LinkedList<Integer> buffer;  
    Semaphore access;  
  
    public Buffer() {  
        buffer = new LinkedList();  
        access = new Semaphore(1);  
    }  
  
    public Integer removeItem() {  
        ...  
    }  
  
    public void addItem(Integer i) {  
        ...  
    }  
}
```

- Accesses to the buffer (add/remove) are the critical sections.
- We need a semaphore to make sure that access to the buffer is atomic.
- Initialising the semaphore to 1 makes sure only one thread can modify the buffer at once.

Adding and Removing

```
public Integer removeItem(){
    try{
        access.acquire();
    } catch (Exception e){
        e.printStackTrace();
    }
    Integer toReturn =
buffer.removeFirst();
    access.release();
    System.out.println("Buffer size
" + buffer.size());
    return toReturn;
}
```

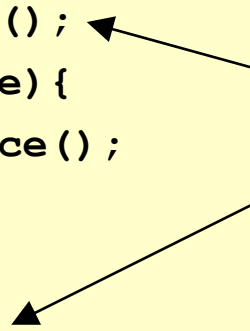
```
public void addItem(Integer i){
    try{
        access.acquire();
    } catch (Exception e){
        e.printStackTrace();
    }
    buffer.addLast(i);
    access.release();

    System.out.println("Buffer size
" + buffer.size());
}
```

- addItem and removeItem can run in parallel;
- But, the critical sections cannot as they are protected by a semaphore;
- Notice we call release as soon as we finish with the shared resource, not after printing.

```
public class Producer extends Thread {  
    Semaphore notFull;  
    Semaphore notEmpty;  
    Buffer buffer;  
  
    public Producer(Semaphore isNotFull, Semaphore isNotEmpty, Buffer  
        toUse){  
        notFull = isNotFull; notEmpty = isNotEmpty; buffer = toUse;  
    }  
  
    public void run(){  
        for(int i = 0; i < 10; ++i){  
            try{  
                notFull.acquire();  
            } catch (Exception e){  
                e.printStackTrace();  
            }  
            buffer.addItem(i);  
            notEmpty.release();  
            //waste some time  
        }  
    }  
}
```

Notice that the semaphore that is released (signalled) is not the same semaphore that is acquired (waited for).



```
public class Consumer extends Thread {
    Semaphore notFull;
    Semaphore notEmpty;
    Buffer buffer;

    public Consumer(Semaphore isNotFull, Semaphore isNotEmpty,
                    Buffer toUse){
        notFull = isNotFull; notEmpty = isNotEmpty; buffer = toUse;
    }

    public void run(){
        for(int i = 0; i < 10; ++i){
            try{
                notEmpty.acquire();
            } catch (Exception e){
                e.printStackTrace();
            }
            Integer item = buffer.removeItem();
            notFull.release();
            //waste some time (a bit more than producer)
            System.out.println("Got " + item);
        }
    }
}
```

Notice that the semaphores are now used in the opposite order.


```
public class ProducerConsumerSemaphore {  
  
    public static void main(String[] args) {  
        Semaphore notEmpty = new Semaphore(0);  
        //determines the size of the buffer  
        Semaphore notFull = new Semaphore(5);  
  
        Buffer buffer = new Buffer();  
        Producer p = new Producer(notFull, notEmpty, buffer);  
        Consumer c = new Consumer(notFull, notEmpty, buffer);  
  
        c.start();  
        p.start();  
    }  
}
```

- Notice notEmpty = 0, zero threads can be allowed to consume.
- NotFull = 5, 5 threads can produce before any need to be blocked.

Semaphore Debugging

- Don't forget to release semaphores that you acquire;
- Avoid deadlock:
 - One thread doing acquire A, acquire B and another doing acquire B, acquire A is likely to lead to this.
- Only keep the semaphore as long as you need it.
- Beware: you can call `release()` if you didn't call `acquire()` and that can mean more things than intended enter critical sections.

Example 2: Dining Philosophers

```
public class Philosopher extends Thread {  
  
    int position;  
    Semaphore[] forks;  
  
    public Philosopher (int pos, Semaphore[] forksIn){  
        position = pos;  
        forks = forksIn;  
    }  
  
    public void run () {  
        ...  
    }  
}
```

```
public void run () {  
    while(true){  
        System.out.println("Philosopher " + position + " Thinking....");  
        try{  
            Thread.sleep(100);  
            int left = position;  
            int right = position + 1;  
            //nicer would be int right = (position + 1) % 5;  
            if(right == 5){  
                right = 0;  
            }  
            forks[left].acquire();  
            forks[right].acquire();  
            System.out.println("Philosopher " + position + " Eating");  
            Thread.sleep(100);  
            forks[right].release();  
            forks[left].release();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Main Method

```
public class DiningPhilosophersSemaphore {  
  
    public static void main(String[] args) {  
        Semaphore[] forks = new Semaphore[5];  
        for(int i = 0; i < 5; ++i){  
            forks[i] = new Semaphore(1);  
        }  
        for(int i = 0; i < 5; ++i){  
            Philosopher phil = new Philosopher(i,forks);  
            phil.start();  
        }  
    }  
}
```

- Semaphores set to 1, only one philosopher can use each fork.
- Create 5 philosopher threads and start them running.

```
public void run () {  
    while(true){  
        System.out.println("Philosopher " + position + " Thinking....");  
        try{  
            Thread.sleep(100);  
            int left = position;  
            int right = position + 1;  
            if(right == 5){ //swap right and left to avoid deadlock  
                left = 0;  
                right = 4;  
            }  
            forks[left].acquire();  
            forks[right].acquire();  
            System.out.println("Philosopher " + position + " Eating");  
            Thread.sleep(100);  
            forks[right].release();  
            forks[left].release();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Summary

- The Critical Section Problem with n Processes;
 - Bakery Algorithm;
- Synchronisation Hardware;
- Semaphores:
 - The Producer Consumer Problem;
 - Infinite buffer;
 - Finite buffer;
 - The Dining Philosophers Problem;
- Semaphores in Java.