ETH Zürich

Chair of Software Engineering

Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 4: Semaphores

Today's lecture

In this lecture you will learn about:

- the type of semaphores, an important synchronization primitive,
- implementation variants of semaphores, in particular weak and strong semaphores,
- uses of semaphores, in particular solutions to problems involving mutual exclusion, condition synchronization (the producer-consumer problem), and barriers.

The type of semaphores

The need for a new synchronization primitive

- The synchronization algorithms can provide process synchronization using atomic read and write only
- As a low-level synchronization primitive, they also have a number of disadvantages
    - they rely on busy waiting (inefficient for multitasking)
    - their synchronization variables are freely accessible within the program (no encapsulation)
    - they can become very complex (difficult to implement)

Semaphores

- Semaphores: a higher-level synchronization primitive (not really high-level though) that alleviates some of the problems of synchronization algorithms
- A very important primitive, widely implemented and with many uses
- This comes at a price: the implementation of semaphores needs stronger atomic operations
- Invented by E.W. Dijkstra in 1965
- In other contexts, "semaphore" means traffic signal, e.g. to keep rail tracks free in railroad traffic control

General semaphores

- A *general semaphore* is an object that consists of a variable count and two operations down and up:
    - if a process calls down where count > 0, then count is decremented; otherwise the process waits until count is positive.
    - if a process calls up then count is incremented.
- Atomicity requirements: testing and decrementing, as well as incrementing have to be atomic
- A general semaphore is sometimes also called a *counting semaphore*
- *Value of a semaphore*: value of its count variable

Simple implementation of a general semaphore

```
class SEMAPHORE
feature
    count : INTEGER
    down
        do
            await count > 0
            count := count - 1
        end
    up
        do
            count := count + 1
        end
end
```

Comments on the simple implementation

- We have used the **await** statement: this is busy waiting – we'll get rid of it in more refined implementations
- We use object-oriented / Eiffel-like syntax, but in pseudo-code style
- We will also write for a semaphore s
  - s.count -- value of variable count of s
  - s.down, s.up -- calls to routines of s
- Of course, when semaphores were invented, object-orientation was not yet around

Mutual exclusion for two processes (1)

- Providing mutual exclusion with semaphores: initialize s.count to 1, and enclose the critical section as follows

  s.down

  critical section

  s.up

- Presented in the style of the mutual exclusion problem:

count := 1			
P1		P2	
	while true loop		**while true loop**
1	**await** count > 0	1	**await** count > 0
	count := count – 1		count := count – 1
2	critical section	2	critical section
3	count := count + 1	3	count := count + 1
4	non-critical section	4	non-critical section
	end		**end**

Mutual exclusion for two processes (2)

- Mutual exclusion and deadlock-freedom are easy to prove
- Remember atomicity of test/decrement and increment
- Starvation-freedom is not satisfied, however we will see later how a different implementation fixes this problem

Binary semaphores

- A *binary semaphore* is a semaphore whose value is 0 or 1
- Implementation using a boolean variable is possible

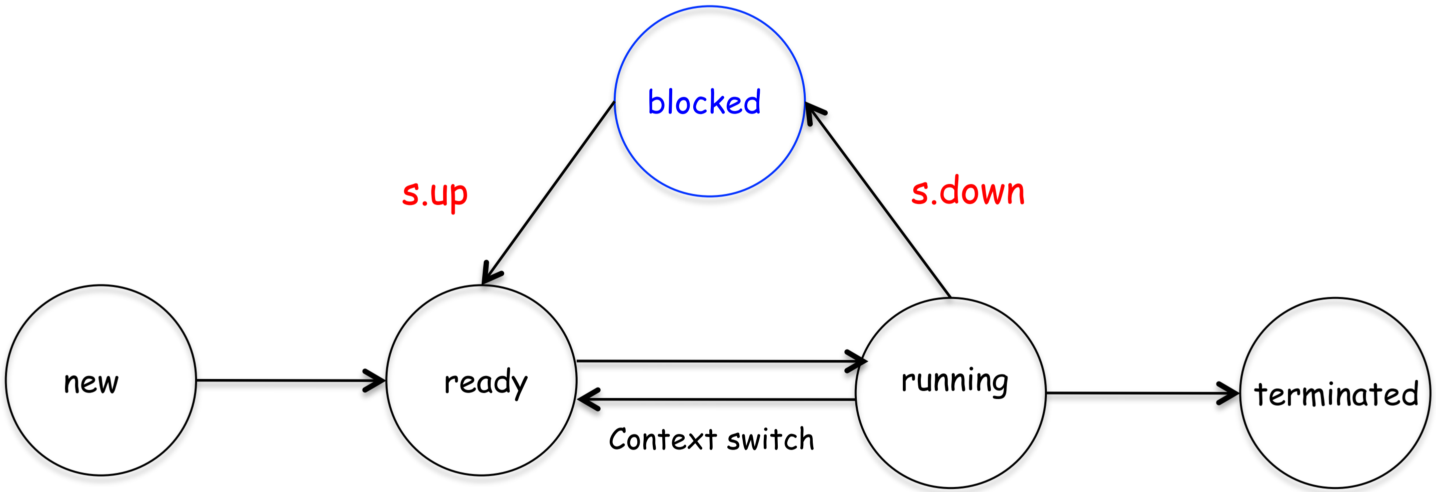
```
b : BOOLEAN
down
    do
        await b
        b := false
    end
up
    do
        b := true
    end
```

Implementation of semaphores

Avoiding busy waiting

- Busy-wait semaphores are unsatisfactory:
  - not starvation-free
  - inefficient when multitasking
- Instead we want a solution where processes block themselves when having to wait, thus freeing processing resources as early as possible

Efficiency: blocking of processes



- A process can be in the following states:
  - *new:* being created.
  - *running*: instructions are being executed.
  - *blocked: currently waiting for an event.*
  - *ready:* ready to be executed, but not been assigned a processor yet.
  - *terminated*: finished executing.

Starvation-freedom: process collections

- In order to avoid starvation, blocked processes are kept in a collection blocked with the following operations:
  - add(P) inserts a process P into the collection
  - remove selects and removes an item from the collection, and returns it
  - is_empty determines whether the collection is empty
- A semaphore where blocked is implemented as a set is called a *weak semaphore*
- Assume for now a weak semaphore

Semaphore implementation (1)

```
count : INTEGER
blocked: CONTAINER
down
   do
      if count > 0 then
         count := count – 1
      else
         blocked.add(P)        -- P is the current process
         P.state := blocked    -- block process P
      end
   end
up
   do
      if blocked.is_empty then
         count := count + 1
      else
         Q := blocked.remove -- select some process Q
         Q.state := ready      -- unblock process Q
      end
   end
```

Semaphore implementation (2)

- Note the differences to the simple implementation:
  - Blocking instead of busy waiting
  - Increment only if there are no blocked processes
- Mutual exclusion and deadlock-freedom preserved
- Starvation-freedom in the *two process* scenario:
  - Assume P1 is blocked
  - When P2 exits the critical section, it unblocks P1 but does not increment the variable count
  - As the value of the semaphore remains 0, process P2 cannot enter before process P1

The semaphore invariant (1)

- We make the following assumptions:
  - $k \geq 0$: the initial value of the semaphore
  - count: current value of the semaphore
  - #down: number of completed *down* operations
  - #up: number of completed *up* operations

The semaphore invariant (2)

- *A semaphore satisfies the following invariants:*

  (1) count $\geq$ 0

  (2) count = k + #up – #down

Proof. (1) easy. (2) is preserved by all operations:

down:

- if count > 0 then #down is incremented and count decremented
- if count $\leq$ 0 then down does not complete and count is unchanged

up:

- if blocked is empty then #up and count are incremented;
- if blocked is not empty then #up and #down are incremented and count is unchanged

Mutual exclusion for n processes

- The mutual exclusion problem for n processes is solved like the one for two processes: initialize count to 1, protect critical sections with down and up
- Starvation is possible in the case of weak semaphores: the reason is that we select a process from blocked at random
- A semaphore where blocked is implemented as a queue is called a *strong semaphore*
- Using a strong semaphore we have a first-come-first-served solution to the mutual exclusion problem for n processes

Solution of the mutual exclusion problem

- *The strong semaphore provides a solution to the mutual exclusion problem for n processes*

Proof. Mutual exclusion:

- Let #cs be the number of processes in critical sections
- Show that #cs + count = 1 is an invariant [...]
- Since count ≥ 0, we have #cs ≤ 1

Starvation-freedom:

- Assume a process is starved with *i* processes ahead of it and argue that in this case count = 0 [...]
- Hence there must be a process in the critical section by the above invariant
- This process must eventually unblock one of the *i* processes
- The result follows by induction on *i*.

Ensuring atomicity of the semaphore operations

- How is the atomicity of *down* and *up* ensured?
- Typically *down* and *up* are not provided by hardware, they must be built in software from lower-level primitives
- We could use synchronization algorithms
- If we have a single processing unit, we may just disable all interrupts; then the scheduler cannot remove the process from the processing unit
- This does not work on multiprocessors: disabling all interrupts on all processing units is too expensive
- Instead use test-and-set: for each semaphore, keep also a test-and-set integer

Side remark: Semaphores in Java

- Java Threads offers semaphores as part of the **java.util.concurrent.Semaphore** package
- Constructors:
    - Semaphore(int k), a weak semaphore
    - Semaphore(int k, boolean b), a strong semaphore if b is set true
- Operations:
    - acquire(), corresponds to down

      -> throws InterruptedException
    - release(), corresponds to up

Uses of semaphores

Uses of semaphores

- Semaphores are a very versatile mechanism, and can be used not only for mutual exclusion
- In the following we give examples of such uses

The k-exclusion problem

- In the *k-exclusion problem*, we allow up to k processes to be in their critical sections at the same time
- A solution is easily obtained with general semaphores
- The value of a semaphore corresponds intuitively to the number of processes that are still allowed to proceed into a critical section

```
s.count := k
```

P_i

```
while true loop
   s.down
   critical section
   s.up
   non-critical section
end
```

Barriers (1)

- A *barrier* is a form of synchronization that determines a point in the execution of a program which all processes in a group have to reach before any of them may move on.
- Barriers are important for iterative algorithms:
    - in each iteration processes work on different parts of the problem
    - before starting the new iteration, all processes need to have finished (e.g. to combine an intermediate result)

Barriers (2)

- A simple barrier for two processes:

s1.count := 0 s2.count := 0			
P1		P2	
1 2 3 4	code before the barrier s1.up s2.down code after the barrier	1 2 3 4	code before the barrier s2.up s1.down code after the barrier

- Semaphore s1 provides the barrier for P2, and semaphore s2 provides the barrier for P1

The producer-consumer problem

- Consider two types of looping processes:
  - *Producer:* At each loop iteration, produces a data item for consumption by a consumer
  - *Consumer:* At each loop iteration, consumes a data item produced by a producer
- Producers and consumers communicate via a shared buffer implementing a queue
- Producers append data items to the back of the queue and consumers remove data items from the front
- The problem consists in writing code for producers and consumers such that the following conditions are satisfied:
  - Every data item produced is eventually consumed
  - The solution is deadlock-free
  - The solution is starvation-free

The producer-consumer problem: background

- The producer-consumer problem corresponds to issues found in many variations on concrete systems
- *Producers*: devices and programs such as keyboards, word processors produce data items such as characters or files to print
- *Consumers*: the operating system and printers are the consumers of these data items
- It has to be ensured that these different entities can communicate with each other appropriately, such that no data items get lost or the system enters a deadlock

The producer-consumer problem: variants

- There are two variants of the producer-consumer problem:
    - the shared buffer is assumed to be unbounded
    - the shared buffer is assumed to be bounded
- We will work on the problem with unbounded buffers first

Condition synchronization

- In the producer-consumer problem, we have to ensure that processes access the buffer properly
    - Consumers have to wait if the buffer is empty
    - Producers have to wait if the buffer is full (in the bounded buffer version of the problem)
- *Condition synchronization* is a form of synchronization where processes are delayed until a certain condition is true
- In the producer consumer problem we have to use two forms of synchronization
    - Mutual exclusion: to prevent races on the buffer
    - Condition synchronization: to prevent improper access of the buffer (as described above)

Solution of the producer-consumer problem (1)

- Two semaphores needed:
  - mutex: to ensure mutual exclusion
  - not_empty:
    - if not_empty.count = 0, then the buffer is empty
    - if not_empty.count = k > 0 then the buffer contains k items
- **Idea:**
  - Once a producer inserts an item, it executes not_empty.up to wake up any blocked consumers or to set the count right
  - Consumers may block on not_empty.down before accessing the buffer

Solution of the producer-consumer problem (2)

mutex.count := 1 not_empty.count := 0	
Producer$_i$	Consumer$_i$

while true loop
    d := produce
    mutex.down
    b.append(d)
    mutex.up
    not_empty.up
end
	while true loop
	    not_empty.down
	    mutex.down
	    d := b.remove
	    mutex.up
	    consume(d)
	end

- To see that the algorithm is correct, prove that *not_empty.count = #items_in_buffer* is an invariant that holds at the beginning and end of each loop
- Deadlock-freedom is also satisfied, and with a strong semaphore also starvation-freedom

Solution for bounded buffers

mutex.count := 1 not_empty.count := 0 not_full.count := k			
Producer$_i$		Consumer$_i$	
	while true loop		**while true loop**
1	d := produce	1	not_empty.down
2	not_full.down	2	mutex.down
3	mutex.down	3	d := b.remove
4	b.append(d)	4	mutex.up
5	mutex.up	5	not_full.up
	not_empty.up		consume(d)
	end		**end**

- To take care of the case that the buffer can also be completely filled, a semaphore not_full is introduced, making the solution more symmetric

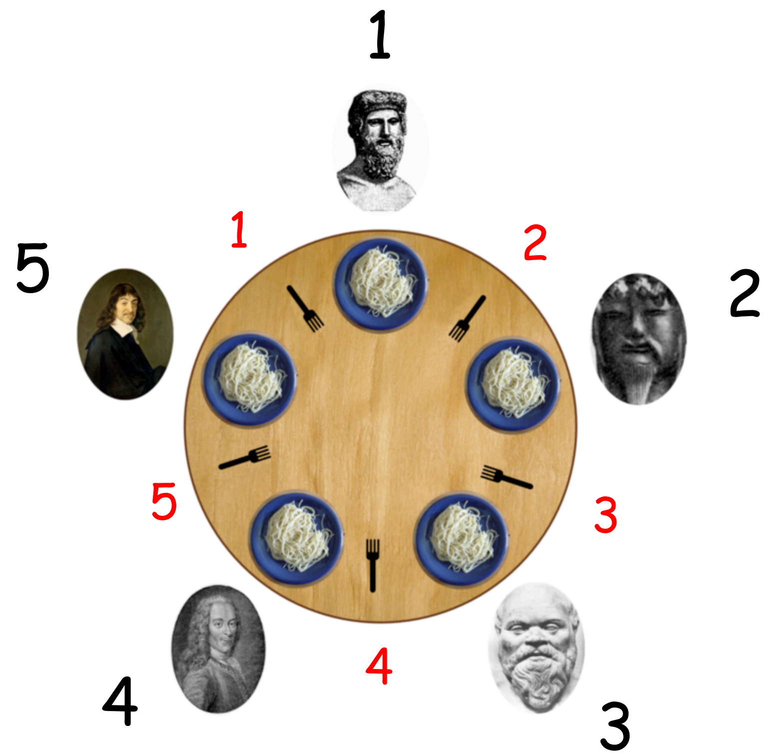
Naming semaphores

- It is good practice to name a semaphore used for condition synchronization after the condition one wants to be true:
    - not_empty: "wait until the buffer is not empty" and "signal processes when the buffer is not empty"

Dining philosophers problem: solution attempt

- Dining philosophers problem: n philosophers
- Solution attempt:

```
s[1].count := 1, ..., s[n].count := 1
-------------------------------------
Philosopher_i
-------------------------------------
while true loop
    think
    s[i].down
    s[(i mod n) + 1].down
    eat
    s[(i mod n) + 1].up
    s[i].up
end
```



- Semaphore s[i] corresponds to the availability of the *i*th fork
- Problem?

Dining philosophers problem: a fix

- Asymmetric solution: one philosopher picks up forks in a different order

$Philosopher_n$

```
while true loop
    think
    s[1].down
    s[n].down
    eat
    s[n].up
    s[1].up
end
```

- Hence the *circular wait* condition (Coffman) is broken: no deadlock

Simulating general semaphores

General semaphores are superfluous

- We have distinguished binary semaphores from general (counting) semaphores
- Having general semaphores is beneficial, it allows us to solve problems like the k-exclusion problem effortlessly
- However, from a theoretical perspective they are not needed: we can implement general semaphores with binary semaphores

Implementing general semaphores by binary ones

```
mutex.count := 1  -- binary semaphore
delay.count := 1    -- binary semaphore
count := k

general_down
   do
      delay.down
      mutex.down
      count := count – 1
      if count > 0 then
         delay.up
      end
      mutex.up
   end

general_up
   do
      mutex.down
      count := count + 1
      if count = 1 then
         delay.up
      end
      mutex.up
   end
```

Correctness idea for the simulation

- The variable count represents the value of the general semaphore
- The binary semaphore mutex protects modifications on count
- The first k – 1 processes executing general_down will also execute delay.up, but not the *k*th process
- Hence further processes have to wait at the entry to general_down
- In this case count = 0, and the first process to execute general_up will execute delay.up