Last week: synchronisation, but lacking the simplicity

- we looked at various solutions to the **mutual exclusion problem**

- algorithms were limited to the simplest tools - **atomic read** and **write** to shared memory

  => difficult to implement; complex
  => reliance on busy waiting
  => no encapsulation of synchronisation variables
Short diversion: hot desks
Short diversion: hot desks

desk please!
Short diversion: hot desks

merci!
Short diversion: hot desks

merci!
Short diversion: hot desks
Short diversion: hot desks

desk please!
Short diversion: hot desks

super duper!
Short diversion: hot desks

super duper!
Short diversion: hot desks
Short diversion: hot desks

gotta wait!
Short diversion: hot desks

- all done!
- gotta wait!
Short diversion: hot desks

gotta wait!

all done!
Short diversion: hot desks

gotta wait!

1

all done!
Short diversion: hot desks

gotta wait!
Short diversion: hot desks

woohoo!
Short diversion: hot desks

woohoo!
Short diversion: hot desks

a semaphore
Today’s lecture: semaphores

• we will discuss semaphores, an important synchronisation primitive

• conceptually simple, although their implementations require stronger atomic operations

• widespread use in operating systems

• invented by Dijkstra in 1965
Next on the agenda

1. general and binary semaphores
2. implementing semaphores
3. beyond the mutual exclusion problem
4. simulating general semaphores
General semaphores
(aka “counting semaphores”)

• a general semaphore is an object consisting of:

  (1) an integer variable count such that count ≥ 0

  (2) two atomic operations: down and up

if a process calls down when count > 0, then count is decremented by 1 (otherwise it first waits)

if a process calls up, then count is incremented by 1
General semaphores
(in Eiffel-like pseudocode)

class SEMAPHORE
feature
  count : INTEGER

  down
    do-atomic
      await count > 0
      count := count - 1
    end

  up
    do-atomic
      count := count + 1
    end
end
General semaphores
(in Eiffel-like pseudocode)

class SEMAPHORE
feature
count : INTEGER

  down
  do-atomic
  await count > 0
  count := count - 1
  end

  up
  do-atomic
  count := count + 1
  end
end

will discuss how to ensure atomicity and how to avoid busy wait later!
Mutual exclusion for two processes

- create a semaphore `s` and initialise `s.count` to 1; then:

  ```
  s.down
  critical section
  s.up
  ```
Mutual exclusion for two processes

• create a semaphore \( s \) and initialise \( s.count \) to 1; then:

\[ s.down \]
\[ critical\ section \]
\[ s.up \]

one process at a time; or one hot desk!
### Mutual exclusion for two processes

- or in the style of last week’s mutual exclusion problems:

<table>
<thead>
<tr>
<th>count := 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
</tbody>
</table>
Mutual exclusion for **two** processes

- **mutual exclusion** and **deadlock freedom** can be proven
  
  \( \Rightarrow \) remember the atomicity of *down* and *up*!

- **solution does not** satisfy **starvation freedom**

  \( \Rightarrow \) a different implementation later will fix this
The general semaphore invariant

- general semaphores are characterised by the following invariant -- important for proofs!

- given some semaphore, let:

\[ = k \text{ denote its initial value with } k \geq 0 \]
\[ = \text{count} \text{ denote its current value} \]
\[ = \#\text{down} \text{ denote the number of completed down operations} \]
\[ = \#\text{up} \text{ denote the number of completed up operations} \]

- then the following equations are invariant:

\[(1) \text{ count } \geq 0 \]
\[(2) \text{ count } = k + \#up - \#down \]
Binary semaphores

• in the previous example, \( s.count \) is always either 0 or 1

• such a semaphore is called a binary semaphore and can be implemented using a Boolean variable

\[
b : \text{BOOLEAN}
\]

\[
down
\]
\[
do-atomic
\]
\[
\text{await} \ b
\]
\[
b := \text{false}
\]
\[
end
\]

\[
up
\]
\[
do-atomic
\]
\[
b := \text{true}
\]
\[
end
\]
Binary semaphores

• in the previous example, $s.count$ is always either 0 or 1

• such a semaphore is called a binary semaphore and can be implemented using a Boolean variable

\begin{verbatim}
  b : BOOLEAN

down
  do-atomic
  await b
  b := false
end

up
  do-atomic
  b := true
end
\end{verbatim}

This is deceptively similar to the previous weeks’ early, and wrong attempts at providing mutual exclusion. What’s different?
Next on the agenda

1. general and binary semaphores

2. implementing semaphores

3. beyond the mutual exclusion problem

4. simulating general semaphores
Avoiding busy waiting

• **busy-wait semaphores** are not ideal
  
  => they are not starvation free
  => inefficient in the context of multitasking

• more preferable would be for processes to **block themselves** when having to wait
  
  => thus freeing processing resources as early as possible

• idea: keep track of blocked processes, “waking them” upon **up** calls on the semaphore
Avoiding busy waiting

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.

Avoiding busy waiting
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.

Avoiding busy waiting

```python
if s.count < 1
```

![State diagram](image)
Implementing the scheme

• to avoid starvation, we will track blocked processes in a collection *blocked*

• we equip *blocked* with the following operations, which will be integrated into *down* and *up*

  => add(P) -- insert process P into collection
  => remove -- select, remove, and return an item from the collection
  => is_empty -- true if collection empty; false otherwise

• if *blocked* is implemented as a *set*, we call the semaphore *weak*; if as a *FIFO queue*, then *strong*
Weak semaphore

• a weak semaphore is a blocking semaphore in which
the collection blocked is implemented as a set

=> remove will pick and remove a random element

\[
\begin{align*}
\text{down} & \\
\text{do-atomic} & \\
\text{if } \text{count} > 0 \text{ then} & \\
& \quad \text{count} := \text{count} - 1 \\
\text{else} & \\
& \quad \text{blocked.add}(P) \\
& \quad P.\text{state} := \text{blocked} \\
\text{end} & \\
\text{end} & \\
\text{up} & \\
\text{do-atomic} & \\
\text{if } \text{blocked.is_empty} \text{ then} & \\
& \quad \text{count} := \text{count} + 1 \\
\text{else} & \\
& \quad Q := \text{blocked.remove} \\
& \quad Q.\text{state} := \text{ready} \\
\text{end} & \\
\text{end} &
\end{align*}
\]
Weak semaphore

- a **weak semaphore** is a blocking semaphore in which the collection `blocked` is implemented as a set

=> *remove* will pick and remove a **random** element

```plaintext
\[
\begin{align*}
\text{down} & \\
\text{do-atomic} & \\
\text{if} & \quad \text{count} > 0 \text{ then} \\
& \quad \text{count} := \text{count} - 1 \\
\text{else} & \\
& \quad \text{blocked.add}(P) \\
& \quad P.\text{state} := \text{blocked} \\
\text{end} & \\
\text{end} & \\
\text{up} & \\
\text{do-atomic} & \\
\text{if} & \quad \text{blocked.is_empty} \text{ then} \\
& \quad \text{count} := \text{count} + 1 \\
\text{else} & \\
& \quad Q := \text{blocked.remove} \\
& \quad Q.\text{state} := \text{ready} \\
\text{end} & \\
\text{end} & \\
\end{align*}
\]

\text{– add current process P to blocked} \\
\text{– block P (instead of busy wait)}
```
Weak semaphore

- a weak semaphore is a blocking semaphore in which the collection *blocked* is implemented as a set

=> remove will pick and remove a random element

```plaintext
down
do-atomic
   if count > 0 then
      count := count - 1
   else
      blocked.add(P)
      P.state := blocked
   end
end

up
do-atomic
   if blocked.is_empty then
      count := count + 1
   else
      Q := blocked.remove
      Q.state := ready
   end
end
```

- select and remove some process Q from blocked
- unblock Q so that it can access the resource
(Question: why is count left unchanged?)
Mutual exclusion for two processes

- weak semaphores provide starvation-freedom in the two process scenario

=> why?

- what about mutual exclusion for n processes?
Mutual exclusion for \( n \) processes

- create a semaphore \( s \) and initialise \( s.count \) to 1; then:
  
  - \( s\text{.down} \)
  - \( s\text{.critical section} \)
  - \( s\text{.up} \)

- starvation is possible for \( n > 2 \) with weak semaphores because we select a process from blocked at random

- solution is to use a strong semaphore, in which \( blocked \) is implemented as a FIFO queue
Strong semaphores provide a solution to the mutual exclusion problem with \( n \) processes (how to prove)

- **mutual exclusion** -- show that

\[
\#cs + \text{count} = 1
\]

where \#cs is the number of processes in critical sections

- **starvation freedom** -- apply *proof by contradiction*

  \[=> \text{begin by assuming that a process in } \text{blocked} \text{ is starved}\]
A note on atomicity

• operations down and up are typically built in software using lower-level primitives (e.g. synchronisation algorithms)

• alternatively:

  => using test-and-set instructions (atomic read and write)
  => disabling interrupts (only realistic on a single processing unit)
A note on Java

- `java.util.concurrent.Semaphore`

  [http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html)

- **constructors**
  
  => `Semaphore(int k)`  -- a weak semaphore
  
  => `Semaphore(int k, boolean b)`  -- a strong semaphore if `b` true

- **operations**
  
  => `acquire()`  -- corresponds to `down`
  
  => `release()`  -- corresponds to `up`
Next on the agenda

1. general and binary semaphores
2. implementing semaphores
3. beyond the mutual exclusion problem
4. simulating general semaphores
The *k*-exclusion problem

- in the *k*-exclusion problem, we allow up to *k* processes to simultaneously be in their critical sections
  
  \[ \Rightarrow \text{mutual exclusion is the } k = 1 \text{ instance} \]
  
- use a general semaphore corresponding to the number of processes allowed to be in their critical sections

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P&lt;sub&gt;i&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>while true loop</td>
</tr>
<tr>
<td>2</td>
<td>s.down</td>
</tr>
<tr>
<td>3</td>
<td>critical section</td>
</tr>
<tr>
<td>4</td>
<td>s.up</td>
</tr>
<tr>
<td>4</td>
<td>non-critical section</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
The \( k \)-exclusion problem

• in the \( k \)-exclusion problem, we allow up to \( k \) processes to simultaneously be in their critical sections

\[ \Rightarrow \text{mutual exclusion is the } k = 1 \text{ instance} \]

• use a general semaphore corresponding to the number of processes allowed to be in their critical sections

\[
\begin{array}{c|c}
\text{s.count} & \equiv k \\
pi & \hline \\
1 & \text{while true loop} \\
 & \text{s.down} \\
2 & \text{critical section} \\
3 & \text{s.up} \\
4 & \text{non-critical section} \\
\end{array}
\]

1 \quad 2 \quad \ldots \quad k
Barriers

• semaphores can be used to control the ordering of events in a system

• a barrier is a form of synchronisation that determines a point in a program’s execution that all processes in a group have to reach before any of them may move on

=> important for concurrent iterative algorithms
Barriers

- Semaphores can be used to control the ordering of events in a system.

- A barrier is a form of synchronisation that determines a point in a program’s execution that all processes in a group have to reach before any of them may move on.

=> important for concurrent iterative algorithms

<table>
<thead>
<tr>
<th>s1.count := 0</th>
<th>s2.count := 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>code before the barrier</td>
<td>code before the barrier</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>s1.up</td>
<td>s2.up</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>s2.down</td>
<td>s1.down</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>code after the barrier</td>
<td>code after the barrier</td>
</tr>
</tbody>
</table>
Barriers

- Semaphores can be used to control the ordering of events in a system.

- A barrier is a form of synchronisation that determines a point in a program’s execution that all processes in a group have to reach before any of them may move on.

=> important for concurrent iterative algorithms

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>code before the barrier</td>
</tr>
<tr>
<td>2</td>
<td>s1.up</td>
</tr>
<tr>
<td>3</td>
<td>s2.down</td>
</tr>
<tr>
<td>4</td>
<td>code after the barrier</td>
</tr>
<tr>
<td>s1.count := 0</td>
<td>s2.count := 0</td>
</tr>
</tbody>
</table>

s1 is the barrier for P2; s2 is the barrier for P1 -- why are they initialised to 0?
The producer-consumer problem

Producers

store (buffer, int)

Buffer

243
46
71
97

consume (buffer)

Consumers
The producer-consumer problem

Producers

store (buffer, int)

Buffer

243
46
71
97

Consumers

consume (buffer)

require buffer.not_full

require buffer.not_empty
The producer-consumer problem

• a good solution would:

  => ensure that every data item produced is eventually consumed
  => be deadlock-free
  => be starvation-free

• need a semaphore for mutual exclusion (the buffer)

• but additional semaphore(s) for condition synchronisation

  => e.g. consumer should block until the buffer is non-empty
Solution for an **unbounded buffer**

```plaintext
mutex.count := 1
not_empty.count := 0

Producer_i

<table>
<thead>
<tr>
<th>1</th>
<th><strong>while true loop</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>d := produce</td>
</tr>
<tr>
<td>3</td>
<td>mutex.down</td>
</tr>
<tr>
<td>4</td>
<td>b.append(d)</td>
</tr>
<tr>
<td>5</td>
<td>mutex.up</td>
</tr>
<tr>
<td></td>
<td><strong>not_empty.up</strong></td>
</tr>
<tr>
<td></td>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>

Consumer_i

<table>
<thead>
<tr>
<th>1</th>
<th><strong>while true loop</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td><strong>not_empty.down</strong></td>
</tr>
<tr>
<td>3</td>
<td>mutex.down</td>
</tr>
<tr>
<td>4</td>
<td>d := b.remove</td>
</tr>
<tr>
<td>5</td>
<td>mutex.up</td>
</tr>
<tr>
<td></td>
<td>consume(d)</td>
</tr>
<tr>
<td></td>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>
```
Solution for an **unbounded buffer**

<table>
<thead>
<tr>
<th>Producer$_i$</th>
<th>Consumer$_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>while true loop</strong></td>
</tr>
<tr>
<td>2</td>
<td>d := produce</td>
</tr>
<tr>
<td>3</td>
<td>mutex.down</td>
</tr>
<tr>
<td>4</td>
<td>b.append(d)</td>
</tr>
<tr>
<td>5</td>
<td>mutex.up</td>
</tr>
<tr>
<td>6</td>
<td>not_empty.up</td>
</tr>
<tr>
<td>7</td>
<td>end</td>
</tr>
<tr>
<td>1</td>
<td><strong>while true loop</strong></td>
</tr>
<tr>
<td>2</td>
<td>not_empty.down</td>
</tr>
<tr>
<td>3</td>
<td>mutex.down</td>
</tr>
<tr>
<td>4</td>
<td>d := b.remove</td>
</tr>
<tr>
<td>5</td>
<td>mutex.up</td>
</tr>
<tr>
<td>6</td>
<td>consume(d)</td>
</tr>
<tr>
<td>7</td>
<td>end</td>
</tr>
</tbody>
</table>

observe that not_empty.count = #items_in_buffer
Solution for an unbounded buffer

<table>
<thead>
<tr>
<th>Producer_i</th>
<th>Consumer_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>while true loop</td>
</tr>
<tr>
<td></td>
<td>d := produce</td>
</tr>
<tr>
<td>2</td>
<td>b.append(d)</td>
</tr>
<tr>
<td>3</td>
<td>mutex.up</td>
</tr>
<tr>
<td>4</td>
<td>not_empty.up</td>
</tr>
<tr>
<td>5</td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>while true loop</td>
</tr>
<tr>
<td></td>
<td>not_empty.down</td>
</tr>
<tr>
<td></td>
<td>mutex.down</td>
</tr>
<tr>
<td></td>
<td>d := b.remove</td>
</tr>
<tr>
<td></td>
<td>mutex.up</td>
</tr>
<tr>
<td></td>
<td>consume(d)</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>

observe that not_empty.count = #items_in_buffer

mutex.count := 1
not_empty.count := 0

blocks until not_empty.count > 0
Solution for a bounded buffer

\[
\begin{align*}
\text{mutex}.\text{count} & := 1 \\
\text{not}_\text{empty}.\text{count} & := 0 \\
\text{not}_\text{full}.\text{count} & := k
\end{align*}
\]

where \(k\) is the size of the buffer

<table>
<thead>
<tr>
<th>Producer(_i)</th>
<th>Consumer(_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> while true loop</td>
<td><strong>1</strong> while true loop</td>
</tr>
<tr>
<td>(d := \text{produce})</td>
<td>(\text{not}_\text{empty}.\text{down})</td>
</tr>
<tr>
<td>(\text{not}_\text{full}.\text{down})</td>
<td>(\text{mutex}.\text{down})</td>
</tr>
<tr>
<td>(\text{mutex}.\text{down})</td>
<td>(d := \text{b.remove})</td>
</tr>
<tr>
<td>(\text{b.append}(d))</td>
<td>(\text{mutex}.\text{up})</td>
</tr>
<tr>
<td>(\text{mutex}.\up)</td>
<td>(\text{not}_\text{full}.\up)</td>
</tr>
<tr>
<td>(\text{not}_\text{empty}.\up)</td>
<td>(\text{consume}(d))</td>
</tr>
<tr>
<td><strong>end</strong></td>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>
Dining philosophers problem
(a solution that can deadlock)

• multiple semaphores must be used with care -- they are prone to deadlock!

```
s[1].count := 1, ..., s[n].count := 1

<table>
<thead>
<tr>
<th>Philosopher_i</th>
<th>while true loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>think</td>
</tr>
<tr>
<td>2</td>
<td>s[i].down</td>
</tr>
<tr>
<td>3</td>
<td>s[(i mod n) + 1].down</td>
</tr>
<tr>
<td>4</td>
<td>eat</td>
</tr>
<tr>
<td>5</td>
<td>s[(i mod n) + 1].up</td>
</tr>
<tr>
<td>6</td>
<td>s[i].up</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
</tbody>
</table>
```
Dining philosophers problem
(a solution that can deadlock)

- multiple semaphores must be used with care -- they are prone to deadlock!

\[
\begin{array}{|c|}
\hline
\text{s[1].count := 1, ..., s[n].count := 1} \\
\hline
\end{array}
\]

| Philosopher, \( i \) | \begin{align*}
\text{while true loop} \\
1 & \quad \text{think} \\
2 & \quad \text{s[i].down} \\
3 & \quad \text{s[(i mod n) + 1].down} \\
4 & \quad \text{eat} \\
5 & \quad \text{s[(i mod n) + 1].up} \\
6 & \quad \text{s[i].up} \\
\text{end}
\end{align*} |
|---|---|

\textit{circular waiting!}
Dining philosophers problem
(a fix!)

• assume that philosopher $n$ picks up the left fork before the right fork

• this breaks the circle of resource requests; there will always be one philosopher who can acquire both forks and release them again

<table>
<thead>
<tr>
<th>Philosopher$_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>while true loop</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>
Next on the agenda

1. general and binary semaphores
2. implementing semaphores
3. beyond the mutual exclusion problem
4. simulating general semaphores
General semaphores are superfluous

• while conceptually useful, general semaphores (theoretically) are not necessary -- they can be implemented through binary semaphores alone
General semaphores are superfluous

```plaintext
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
  end

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

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

count := k

**value of the general semaphore**

general_down

do

delay.down
mutex.down

count := count − 1

if count > 0 then

delay.up

end

general_up

do

mutex.down

count := count + 1

if count = 1 then

delay.up

end

mutex.up

end

**protects count**

**not called when count = 0**
Next on the agenda

1. general and binary semaphores
2. implementing semaphores
3. beyond the mutual exclusion problem
4. simulating general semaphores
Summary

• semaphores are conceptually simple but powerful tools for solving synchronisation problems

• choice of implementation can affect starvation-freedom

• applications beyond mutual exclusion: k-exclusion, barriers, condition synchronisation

but: correct usage is still far from trivial

• essential reading: Chapter 4 of the CCC textbook