Concepts of Concurrent Programming

Summary of the course in spring 2011 by Bertrand Meyer and Sebastian Nanz

Stefan Heule

2011-05-28

Contents

1 Introduction

1.1 Ambdahl's Law

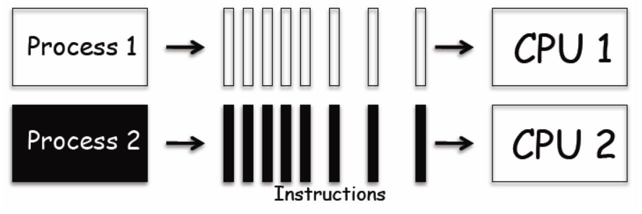
In a program that is run in parallel on n computational units, and where fraction p of the overall execution can exploit parallelism, the speedup (compared to a sequential execution) is

$$\text{speedup} = \frac{1}{(1-p) - \frac{p}{n}}$$

1.2 Basic Notions

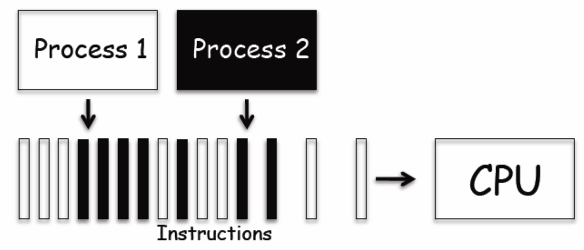
1.2.1 Multiprocessing

- Multiprocessing is the use of more than one processing unit in a system.
- Execution of processes is said to be *parallel*, as they are running at the same time.



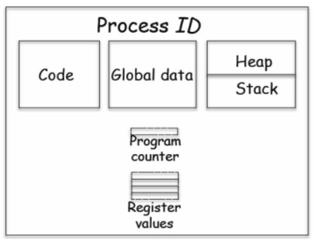
1.2.2 Multitasking

- Even on systems with a single processing unit, it appears as if programs run in parallel.
- This is achieved by the operating system through multitasking: it switches between the execution of different tasks.
- Execution of processes is said to be *interleaved*, as all are in progress, but only one at a time is running.



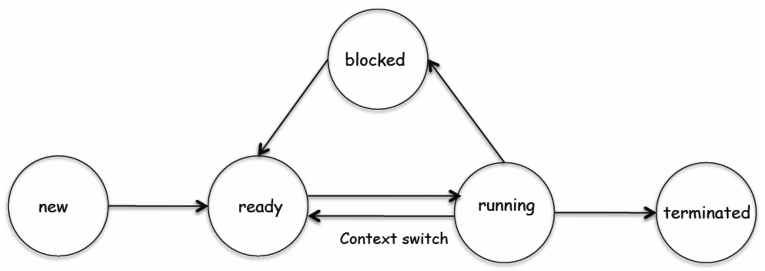
1.2.3 Definitions

- A (sequential) program is a set of instructions.
- Structure of a typical process
    - Process identifier: unique ID of a process.
    - Process state: current activity of a process.
    - Process context: program counter, register values.
    - Memory: program text, global data, stack and heap.

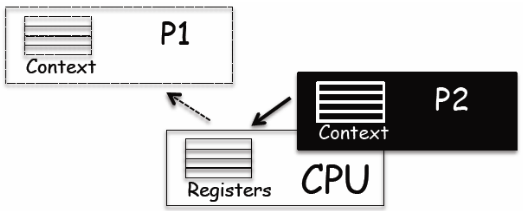


- Concurrency: Both multiprocessing and multitasking are examples of a concurrent computation.
- A system program called the *scheduler* controls which processes are running; it sets the process states:

    - *new*: being created.
    - *running*: instructions are being executed.
    - *ready*: ready to be executed, but not assigned a processor yet.
    - *terminated*: finished executing.



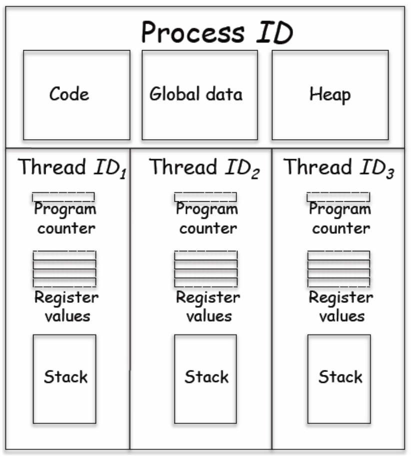
    - A program can get into the state blocked by executing special program instructions (so-called synchronization primitives).
    - When *blocked*, a process cannot be selected for execution.
    - A process gets unblocked by external events which set its state to *ready* again.
- The swapping of processes on a processing unit by the scheduler is called a *context switch*.



    - Scheduler actions when switching processes P1 and P2:

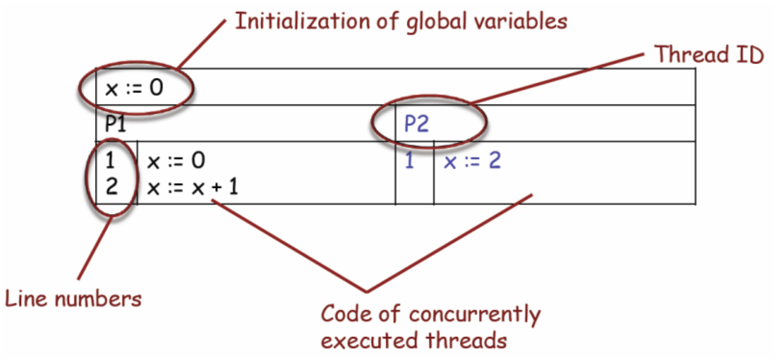
```
P1.state := ready
// save register values as P1's context in memory
// use context of P2 to set register values
P2.state := running
```

- Programs can be made concurrent by associating them with threads (which is part of an operating system process)
    - Components private to each thread
        - Thread identifier
        - Thread state
        - Thread context
        - Memory: only stack
    - Components shared with other threads
        - Program text
        - Global data
        - Heap

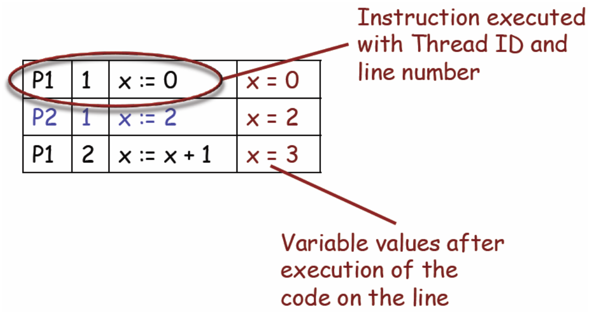


1.2.4 The Interleaving Semantics

- A program which at runtime gives rise to a process containing multiple threads is called a *parallel program*.
- We use an abstract notation for concurrent programs:


Initialization of global variables
Thread ID
x := 0
P1
P2
1 x := 0
2 x := x + 1
1 x := 2
Line numbers
Code of concurrently executed threads

- Executions give rise to *execution sequences*. For instance



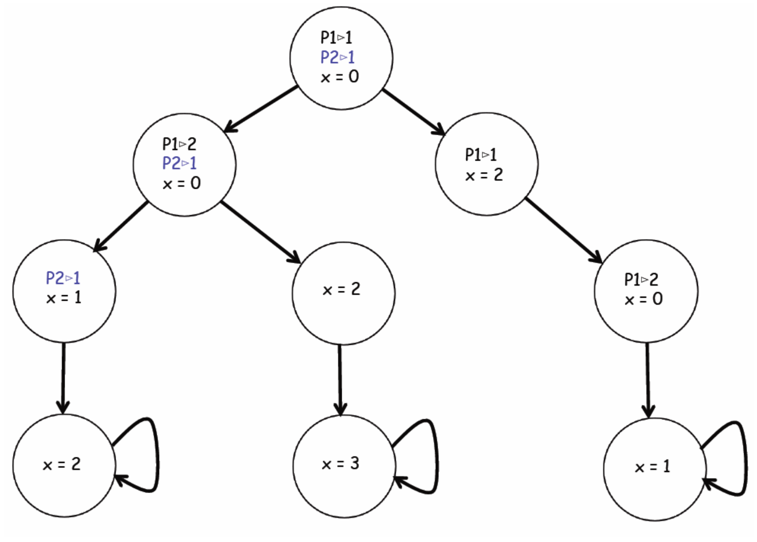
P1	1	x := 0	x = 0
P2	1	x := 2	x = 2
P1	2	x := x + 1	x = 3

Instruction executed with Thread ID and line number
Variable values after execution of the code on the line

- An instruction is *atomic* if its execution cannot be interleaved with other instructions before its completion. There are several choices for the level of atomicity, and by convention every numbered line in our programs can be executed atomically.
- To describe the concurrent behaviour, we need a model:
  - *True-concurrency semantics*: assumption that true parallel behaviours exist.
  - *Interleaving semantics*: assumption that all parallel behaviour can be represented by the set of all non-deterministic interleavings of atomic instructions. This is a good model for concurrent programs, in particular it can describe:
    - Multitasking: the interleaving is performed by the scheduler.
    - Multiprocessing: the interleaving is performed by the hardware.
  - By considering all possible interleavings, we can ensure that a program runs correctly in all possible scenarios. However, the number of possible interleavings grows exponentially with the number of concurrent processes. This is the so-called *state space explosion*.

1.3 Transition Systems and LTL

- A formal model that allows us to express concurrent computation are *transition systems*, which consist of states and transitions between them.
  - A state is labelled with *atomic propositions*, which express concepts such as
    - $P_2 \rhd 2$ (the program pointer of P_2 points to 2
    - $x = 6$ (the value of variable x is 6)
  - There is a transition between two states if one state can be reached from the other by execution an atomic instruction. For instance:



x := 0			
P1		P2	
1	x := 0	1	x := 2
2	x := x + 1		

- More formally, we define transition systems as follows:
  - Let A be a set of atomic propositions. Then, a transition system T is a triple $(S, \rightarrow, L)$ where
    - S is the set of states,
    - $\rightarrow \subseteq S \times S$ is the *transition relation*, and
    - $L: S \rightarrow \mathcal{P}(A)$ is the *labelling function*.
  - The transition relation has the additional property that for every $s \in S$ there is an $s' \in S$ such that $s \rightarrow s'$.
  - A path is an infinite sequence of states: $\pi = s_1, s_2, s_3, \ldots$
  - We write $\pi[i]$ for the (infinite) subsequence $s_i, s_{i+1}, s_{i+2}, \ldots$
- For any concurrent program, its transition system represents all of its behaviour. However, we are typically interested in specific aspects of this behaviour, e.g.,
  - "the value of variable x will never be negative"
  - "the program pointer of P_2 will eventually point to 9"
- Temporal logics allow us to express such properties formally, and we will study *linear-time temporal logic* (LTL).

1.3.1 Syntax and Semantics of Linear-Time Temporal Logic

- Syntax of LTL is given by the following grammar:

$$\Phi ::= T \mid p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \boldsymbol{G}\,\Phi \mid \boldsymbol{F}\,\Phi \mid \Phi\,\boldsymbol{U}\,\Phi \mid \boldsymbol{X}\,\Phi$$

- The following temporal operators exist:
  - $\boldsymbol{G}\,\Phi$: Globally (in all future states) Φ holds
  - $\boldsymbol{F}\,\Phi$: In some future state Φ holds
  - $\Phi_1\,\boldsymbol{U}\,\Phi_2$: In some future state Φ_2 holds, and at least until then, Φ_1 holds.
- The meaning of formulae is defined by the satisfaction relation $\vDash$ for a path $\pi = s_1, s_2, s_3, \ldots$

  - $\pi \vDash T$
  - $\pi \vDash p$ iff $p \in L(s_1)$
  - $\pi \vDash \neg\Phi$ iff $\pi \vDash \Phi$ does not hold
  - $\pi \vDash \Phi_1 \wedge \Phi_2$ iff $\pi \vDash \Phi_1$ and $\pi \vDash \Phi_2$
  - $\pi \vDash \boldsymbol{G}\,\Phi$ iff for all $i \geq 1, \pi[i] \vDash \Phi$
  - $\pi \vDash \boldsymbol{F}\,\Phi$ iff there exists $i \geq 1$, s.t. $\pi[i] \vDash \Phi$
  - $\pi \vDash \Phi_1\, \boldsymbol{U}\, \Phi_2$ iff exists $i \geq 1$, s.t. $\pi[i] \vDash \Phi_2$ and for all $1 \leq j < i, \pi[j] \vDash \Phi_1$
  - $\pi \vDash \boldsymbol{X}\,\Phi$ iff $\pi[2] \vDash \Phi$
- For simplicity, we also write $s \vDash \Phi$ when we mean that for every path π starting in s we have $\pi \vDash \Phi$.
- We say two formulae Φ and Ψ are equivalent, if for all transistion systems and all paths π we have

$$\pi \vDash \Phi \quad \text{iff} \quad \pi \vDash \Psi$$

  - For instance, we have
    - $\boldsymbol{F}\,\Phi \equiv T\,\boldsymbol{U}\,\Phi$
    - $\boldsymbol{G}\,\Phi \equiv \neg\,\boldsymbol{F}\neg\Phi$

1.3.2 Safety and Liveness Properties

- There are two types of formal properties in asynchronous computations:
  - *Safety properties* are properties of the form “something bad never happens”.
  - *Liveness properties* are properties of the form “something good eventually happens”.

1.4 Concurrency Challenges

- The situation that the result of a concurrent execution is dependent on the non-deterministic interleaving is called a *race condition* or *data race*. Such problems can stay hidden for a long time and are difficult to find by testing.
- In order to solve the problem of data races, processes have to *synchronize* with each other. *Synchronization* describes the idea that processes communicate with each other in order to agree on a sequence of actions.
- There are two main means of process communication:
  - *Shared memory*: processes communicate by reading and writing to shared sections of memory.
  - *Message-passing*: processes communicate by sending messages to each other.
- The predominant technique is shared memory communication.
- The ability to hold resources exclusively is central to providing process synchronization for resource access. However, this brings other problems:
  - A *deadlock* is the situation where a group of processes blocks forever because each of the processes is waiting for resources which are held by another process in the group.
- There are a number of necessary conditions for a deadlock to occur (*Coffman conditions*):
  - Mutual exclusion: processes have exclusive control of the resources they require.
  - Hold and wait: processes already holding resources may request new resources.
  - No pre-emption: resources cannot be forcibly removed from a process holding it.

    - Circular-wait: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds.
- The situation that processes are perpetually denied access to resources is called *starvation*. Starvation-free solutions often require some form of *fairness*:
    - *Weak fairness*: if an action is *continuously enables*, i.e. never temporarily disabled, then it has to be executed infinitely often.
    - *Strong fairness*: if an activity is *infinitely often enables*, but not necessarily always, then it has to be executed infinitely often.

2 Synchronization Algorithms

2.1 The mutual exclusion problem

- Race conditions can corrupt the result of a concurrent computation if processes are not properly synchronized. *Mutual exclusion* is a form of synchronization to avoid simultaneous use of a shared resource.
- We call the part of a program that accesses a shared resource a *critical section*.
- The mutual exclusion problem can then be described as n processes of the following form:

```
while true loop
   entry protocol
   critical section
   exit protocol
   non-critical section
end
```

- The entry and exit protocols should be designed to ensure
    - *Mutual exclusion*: At any time, at most one process may be in its critical section.
    - *Freedom of deadlock*: If two or more processes are trying to enter their critical sections, one of them will eventually succeed.
    - *Freedom from starvation*: If a process is trying to enter its critical section, it will eventually succeed.
- Further important conditions:
    - Processes can communicate with each other only via atomic read and write operations.
    - If a process enters its critical section, it will eventually exit from it.
    - A process may loop forever, or terminate while being in its non-critical section.
    - The memory locations accessed by the protocols may not be accessed outside of them.

2.2 Peterson's Algorithm

enter1 := false
enter2 := false
turn := 1 *or* turn := 2			

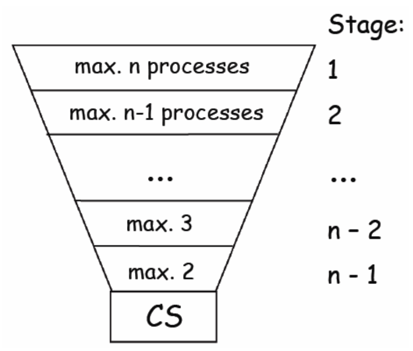
P1		P2	
	while true loop		**while true loop**
1	enter1 := **true**	1	enter2 := **true**
2	turn := 2	2	turn := 1
3	**await not** enter2 **or** turn = 1	3	**await not** enter1 **or** turn = 2
4	critical section	4	critical section
5	enter1 := **false**	5	enter2 := **false**
6	non-critical section	6	non-critical section
	end		**end**

- Peterson's algorithm satisfies mutual exclusion and is starvation-free. It can also be rather easily generalized to n processes.

```
enter[1] := 0; ...; enter[n] := 0
turn[1] := 0; ...; turn[n - 1] := 0
```

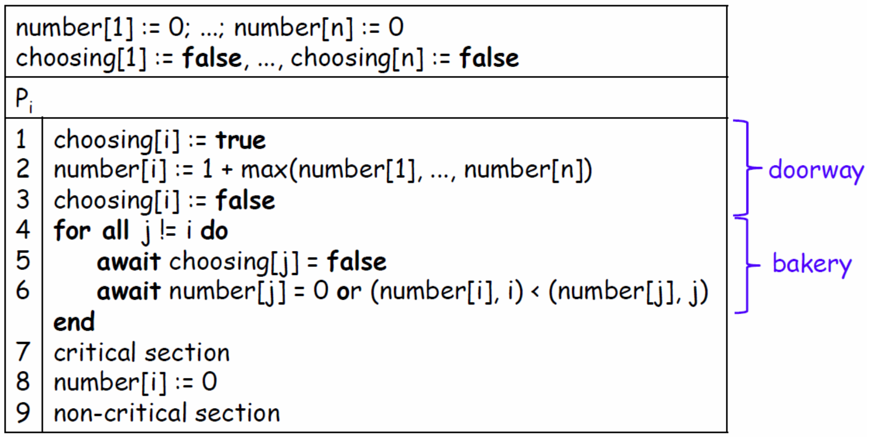
```
P_i
1  for j = 1 to n - 1 do
2     enter[i] := j
3     turn[j] := i
4     await (for all k != i : enter[k] < j) or turn[j] != i
   end
5  critical section
6  enter[i] := 0
7  non-critical section
```

- In this generalization, every process has to go though $n - 1$ stages to reach the critical section; variable j indicates the stage.
    - `enter[i]`: stage the process P_i is currently in.
    - `turn[j]`: which process entered stage j last.
    - Waiting: P_i waits if there are still processes at higher stages, or if there are processes at the same stage, unless P_i is no longer the last process to have entered this stage.
    - Idea for mutual exclusion proof: At most $n - j$ processes can have passed stage j.



2.3 The Bakery Algorithm

- Freedom of starvation still allows that processes may enter their critical sections before a certain, already waiting process is allowed access. We study an algorithm that has very strong fairness guarantees.
- More fairness notions:
    - *Bounded waiting*: If a process is trying to enter its critical section, then there is a bound on the number of times any other process can enter its critical section before the given process does so.
    - *r-bounded waiting*: If a process is trying to enter its critical section, then it will be able to enter before any other process is able to enter its critical section $r + 1$ times.
    - *First-come-first-served*: 0-bounded waiting.
- Relations between the definitions
    - Starvation-freedom implies deadlock-freedom.
    - Starvation-freedom does not imply bounded waiting.
    - Bounded waiting does not imply starvation-freedom.
    - Bounded waiting and deadlock-freedom imply starvation-freedom.



```
number[1] := 0; ...; number[n] := 0
choosing[1] := false, ..., choosing[n] := false

P_i

choosing[i] := true
number[i] := 1 + max(number[1], ..., number[n])
choosing[i] := false
for all j != i do
    await choosing[j] = false
    await number[j] = 0 or (number[i], i) < (number[j], j)
end
critical section
number[i] := 0
non-critical section
```

doorway
bakery

- The bakery algorithm is first-come-first-served. However, one drawback is that the values of tickets can grow unboundedly.

2.4 Space Bounds for Synchronization Algorithms

- A solution for the mutual exclusion problem for n processes that satisfies global progress needs to use n shared one-bit registers. This bound is tight, as Lamport’s one-bit algorithm shows.

3 Semaphores

- In Chapter 2 we have seen how synchronization algorithms can be implemented using only atomic read and write. However, these algorithms have several drawbacks:
    - They rely on busy waiting which is very inefficient for multitasking.
    - Their synchronization variables are freely accessible within the program (no encapsulation).
    - They can become very complex and difficult to implement.
- We introduce *semaphores*, a higher-level synchronization primitive that alleviates some of the problems of synchronization algorithms. They are a very important primitive, widely implemented and with many uses.
- They have been invented by E.W. Dijkstra in 1965 and rely on stronger atomic operations than only atomic read/write.

3.1 General and Binary Semaphores

- A *general semaphore* is an object that consists of a variable `count` and two operations `up` and `down`. Such a semaphore is sometimes also called *counting semaphore*.
    - 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.
- For the implementation we require testing and decrementing, as well as incrementing to be atomic.
- A simple implementation based on busy-waiting looks as follows:

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

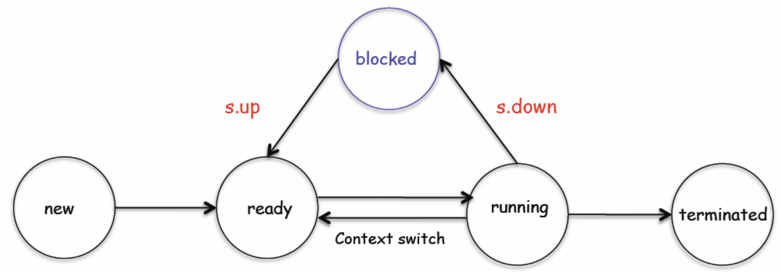
- Providing mutual exclusion with semaphores is easy: we can initialize `s.count` to 1, and enclose the critical section as follows

```
s.down
// critical section
s.up
```

- We can also implement a *binary semaphore*, whose value is either 0 or 1. It is possible to implement this using a Boolean variable.

3.2 Efficient Implementation

- To avoid busy-waiting, we use a solution where processes block themselves when having to wait, thus freeing processing resources as early as possible.



- 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*. When implemented as a queue, we call the semaphore a *strong semaphore*. This gives us a first-come-first-served solution for the mutual exclusion problem of n processes.
- An implementation could look as follows:

```
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
```

3.3 General Remarks

3.3.1 The Semaphore Invariant

- When we make the following assumption, we can express a semaphore invariant:
    - `k≥0`: the initial value of the semaphore.
    - `count`: the current value of the semaphore.
    - `#down`: number of completed `down` operations.
    - `#up`: number of completed `up` operations.
- The semaphore invariant consists of two parts:

$$\text{count} \geq 0$$
$$\text{count} = \text{k} + \#\text{up} - \#\text{down}$$

3.3.2 Ensuring Atomicity of the Semaphore Operations

- To ensure the atomicity of the sempaphore operations, one has typically built them in software, as the hardware does not provide up and down directly. However, this is possible using test-and-set.

3.3.3 Semaphores in Java

- Java Threads offers semaphores as part of the `java.util.concurrent.Semaphore` package.
    - `Semaphore(int k)` gives a weak semaphore, and
    - `Semaphore(int k, bool b)` gives a strong semaphore if `b` is set to true.
- The operations have slightly different names:
    - `acquire()` corresponds to `down` (and might throw an `InterruptedException`).
    - `release()` corresponds to `up`.

3.4 Uses of Semaphores

3.4.1 The *k*-Exclusion Problem

- In the k-exlucsion problem, we allow up to k processes to be in their critical sections at the same time. A solution can be achieved very easily using a general semaphore, where the value of the semaphore intuitively corresponds to the number of processes 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
```

3.4.2 Barriers

- 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 every iteration processes work on different parts of the problem
    - Before starting a new iteration, all processes need to have finished (e.g., to combine an intermediate result).
- For two processes, we can use a semaphore s_1 to provide the barrier for P_2 and another s_2 for P_1.

s1.count := 0
s2.count := 0			

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

3.4.3 The Producer-Consumer Problem

- We consider two types of looping processes:
    - A *producer* produces at every loop iteration a data item for consumption by a consumer.
    - A *consumer* consumes such a data item at every loop iteration.
- Producers and consumer communicate via a shared buffer implementing a queue, where the producers append data items to the back of the queue, and consumers remove data items from the front.
- The problem consists of 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.
- This abstract description of the problem is found in many variations in concrete systems, e.g., producers could be devices and programs such as keyboards or word processers that produce data items such as characters or files to print. The consumers could then be the operating system or a printer.
- There are two variants of the problem, one where the shared buffer is assumed to be *unbounded*, and one where the buffer has only a *bounded* capacity.
- Condition synchronization
    - In the producer-consumer problem we have to ensure that processes access the buffer correctly.
        - Consumers have to wait if the buffer is empty.
        - Producers have to wait if the buffer is full (in the bounded version).
    - *Condition synchronization* is a form of synchronization where processes are delayed until a certain condition is met.
- In the producer-consumer problem we have to use two forms of synchronization:
    - Mutual exclusion to prevent races on the buffer, and
    - condition synchronization to prevent improper access of the buffer (as described above).

- We use several semaphores to solve the problem.
    - `mutex` to ensure mutual exclusion, and
    - `not_empty` to count the number of items in the buffer, and
    - `not_full` to count the remaining space in the buffer (optional).

```
mutex.count := 1
not_empty.count := 0
not_full.count := k
```

```
Producer_i
while true loop
    d := produce
    not_full.down
    mutex.down
    b.append(d)
    mutex.up
    not_empty.up
end
```

```
Consumer_i
while true loop
    not_empty.down
    mutex.down
    d := b.remove
    mutex.up
    not_full.up
    consume(d)
end
```

- Side remark: It is good practice to name a semaphore used for condition synchronization after the condition one wants to be true. For instance, we used `non_empty` to indicate that one has to wait until the buffer is not empty.

3.4.4 Dining Philosophers

- We can use semaphores to solve the dining philosophers' problem for n philosophers. To ensure deadlock-freedom, we have to break the circular wait condition.

```
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
```

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

3.4.5 Simulating General Semaphores

- We can use two binary semaphores to implement a general (counting) semaphore.

```
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
```

4 Monitors

- Semaphores provide a simple yet powerful synchronization primitive: they are conceptually simple, efficient, and versatile. However, one can argue that semaphores provide “too much” flexibility:
    - We cannot determine the correct use of a semaphore from the piece of code where it occurs; potentially the whole program needs to be considered.
    - Forgetting or misplacing a `down` or `up` operation compromises correctness.
    - It is easy to introduce deadlocks into programs.
- We would like an approach that supports programmers better in these respects, enabling them to apply synchronization in a more *structured* manner.

4.1 The Monitor Type

- Monitors are an approach to provide synchronization that is based in object-oriented principles, especially the notions of *class* and *encapsulation*. A *monitor* class fulfils the following conditions:
    - All its attributes are private.
    - Its routines execute with mutual exclusion.
- A *monitor* is an object instantiation of a monitor class. Intuition:
    - Attributes correspond to shared variables, i.e., threads can only access them via the monitor.
    - Routine bodies correspond to critical sections, as at most one routine is active inside a monitor at any time.
- To ensure that at most one routine is active inside a monitor is ensured by the monitor implementation (not burdened on the programmer). One possibility is to use semaphores to implement this; we use a strong semaphore `entry` that is associated as the monitor’s lock and initialized to 1. Any monitor routine must acquire the semaphore before executing its body.
- To solve the mutual exclusion problem, we can use a monitor with n methods called `critical_1` up to `critical_n`. Then, the processes look as follows:

```
create cs.make
```

P_i

```
while true loop
   cs.critical_i
   non-critical section
end
```

4.2 Condition Variables

- To achieve condition synchronization, monitors provide *condition variables*. Although they can be compared to semaphores, their semantics is much different and deeply intertwined with the monitor concept. A *condition variable* consists of a queue `blocked` and three (atomic) operations:
    - `wait` releases the lock on the monitor, block the executing thread and appends it to `blocked`.

    - `singal` has no effect if `blocked` is empty; otherwise it unblocks a thread, but can have other side effects that depend on the signalling discipline used.
    - `is_empty` returns true if `blocked` is empty, and false otherwise.
- The operations `wait` and `signal` can only be called from the body of a monitor routine.

```
class CONDITION_VARIABLE
  feature
    blocked: QUEUE
    wait
      do
        entry.up -- release the lock on the monitor
        blocked.add(P) -- P is the current process
        P.state := blocked -- block process P
      end
    signal -- behaviour depends on signalling discipline
      deferred end
    is_empty: BOOLEAN
      do
        result := blocked.is_empty
      end
end
```

4.2.1 Signalling Disciplines

- When a process signals on a condition variable, it still executes inside the monitor. As only one process may execute within a monitor at any time, an unblocked process cannot enter the monitor immediately. There are two main choices for continuation:
    - The signalling process continues, and the signalled process is moved to the entry of the monitor (*signal and continue*).

```
signal
  do
    if not blocked.is_empty then
      Q := blocked.remove
      entry.blocked.add(Q)
  end
end
```

    - The signalling process leaves the monitor, and lets the signalling process continue (*signal and wait*). In this case, the monitor's lock is silently passed on.

```
signal
  do
    if not blocked.is_empty then
      entry.blocked.add(P) -- P is the current process
      Q := blocked.remove
      Q.state := ready -- unblock process Q
      P.state := blocked -- block process P
  end
end
```

- Comparison of “signal and continue” (SC) with “signal and wait” (SW)
  - If a thread executes a SW `signal` to indicate that a certain condition is true, this condition will be true for the signalled process.
  - This is not the case for SC, where the signal is only a hint that the condition might be true now; other threads might enter the monitor beforehand, and make the condition false again.
  - In monitors with a SC `signal`, also an operation `signal_all` usually exists to wake all waiting processes. That is

    ```
    while not blocked.is_empty do signal end
    ```

  - However, `signal_all` is typically inefficient; for many threads the signalled condition might not be true any more.
- Other signalling disciplines:
  - Urgent signal and continue: special case of SC, where a thread unblocked by a signal operation is given priority over threads already waiting in the entry queue.
  - Signal and urgent wait: special case of SW, where signaller is given priority over threads already waiting in the entry queue.
  - To implement these disciplines, a queue `urgent_entry` can be introduced which has priority over the standard queue `entry`.
- We can classify three sets of threads:
  - S as the signalling threads,
  - U as the threads that have been unblocked on the condition, and
  - B as the threads that are blocked on the entry.
- We can express the signalling disciplines concisely as follows, where $A > B$ means that the threads in A have priority over threads in B.

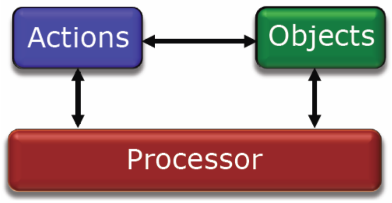
Signal and continue	$S > U = B$
Urgent signal and continue	$S > U > B$
Signal and wait	$U > S = B$
Signal and urgent wait	$U > S > B$

4.3 Summary

- Benefits of monitors
    - *Structured approach*: programmer does not have to remember to follow a wait with a signal just to implement mutual exclusion.
    - *Separation of concerns*: mutual exclusion for free, and we can use condition variables to achieve condition synchronization.
- Problems of monitors
    - *Performance concerns*: trade-off between programmer support and performance.
    - *Signalling disciplines*: source of confusion; SC problematic as condition can change before a waiting process enters the monitor.
    - *Nested monitor calls*: Consider that routine r_1 of monitor M_1 makes a call to routine r_2 of monitor M_2. If routine r_2 contains a wait operation, should mutual exclusion be released for both M_1 and M_2, or only M_2?

5 SCOOP

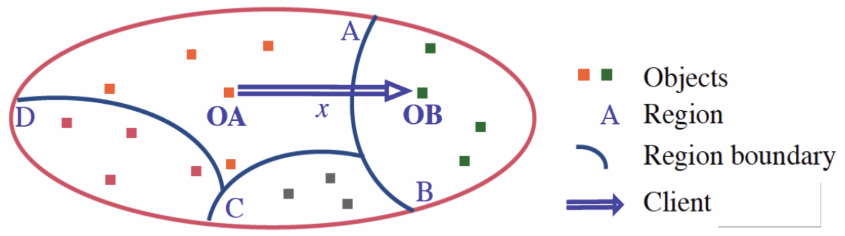
- In SCOOP, a *processor* is the thread of control supporting sequential execution of instructions on one or more objects. This can be implemented as a CPU, a process or a thread. It will be mapped to a computational resource.



- The computational model of SCOOP relies on the following fundamental rule:

> **Handler rule:** All calls targeted to a given object are performed by a single processor, called the object's *handler*.

- A call is "targeted" to an object in the sense of object-oriented programming; the call `x.r` applies the routine `r` to the *target* object identified by `x`.
- The set of objects handled by a given processor is called a *region*. The handler rule implies a one-to-one correspondence between processors and regions.



- SCOOP introduces the keyword **`separate`**, which is a type modifier. If `x` is declared **`separate`** `T` for some type `T`, then the associated object might be handled by a different processor. Note: it is not required that the object resides on a different processor.
  - For instance, if a processor p executes a call `x.r`, and `x` is handled by processor q (rather than p itself) will execute `r`. We call such a call *separate call*.

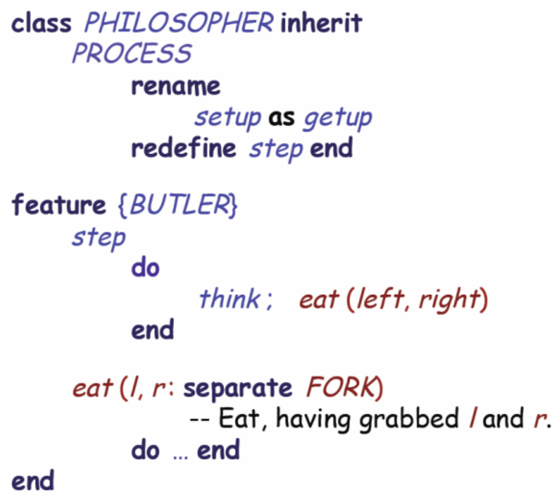
5.1 The Basic Principle

- Separate calls to *commands* are executed *asynchronously*.
  - A client executing a separate call `x.r(a)` logs the call with the handler of `x` (who will execute it)
  - The client can proceed executing the next instruction without waiting.
- A separate call to *queries* will not proceed until the result of the query has been computed. This is also called *wait by necessity*.
- For non-separate calls, the semantics is the same as in a sequential setting; the client waits for the call to finish (synchronously).
- The introduction of asynchrony highlights a difference between two notions:
  - A *routine call*, such as `x.r` executed by a certain processor p.

    - A *routine application*, which – following a call – is the execution of the routine `r` by a processor q.
- While the distinction exists in sequential programming, it is especially important in SCOOP, as processors p and q might be different.

5.2 Mutual exclusion in SCOOP

- SCOOP has a simple way to express mutually exclusive access to objects by way of message passing.
    - The SCOOP runtime system makes sure that the application of a call `x.r(a1,a2,…)` will wait until it has been able to *lock all separate objects* associated with the arguments `a1`, `a2`, etc.
    - Within the routine body, the access to the separate objects associated with the arguments `a1`, `a2`, … is thus mutually exclusive. This allows one to very easily lock a *group of objects*.
- Consider the dining philosophers example:



```
class PHILOSOPHER inherit
    PROCESS
        rename
            setup as getup
        redefine step end

feature {BUTLER}
    step
        do
            think ;   eat (left, right)
        end

    eat (l, r: separate FORK)
            -- Eat, having grabbed l and r.
        do … end
end
```

- Argument passing is *enforced* in SCOOP to protect modifications on separate objects. The following rule expresses this:

> **Separate argument rule:** The target of a separate call must be an argument of the enclosing routine.

5.3 Condition Synchronization

- Condition synchronization is provided in SCOOP by reinterpreting routine preconditions as *wait conditions*.
    - The execution of the body of a routine is delayed until its *separate preconditions* (preconditions that involve a call to a separate target) are satisfied.

Wait rule: A call with separate arguments waits until the corresponding objects' handlers are all available, and the separate conditions are all satisfied. It reserves the handlers for the duration of the routine's execution.

5.4 The SCOOP Runtime System

- When a processor makes a separate feature call, it sends a *feature request*. Each processor has a request queue to keep track of these feature requests.
- Before a processor can process a feature request, it must
  - Obtain the necessary locks, and
  - satisfy the precondition.
- The processor sends a *locking request* to a scheduler. The scheduler keeps track of the locking requests and approved the requests according to a scheduling algorithm. Several possibilities, e.g., centralized vs. decentralized or different levels of fairness.
- *Separate callbacks* are cases where a method a calls a method b, which in turn calls (back) to a.
  - This can lead to a deadlock. The solution is to interrupt processors from waiting and ask the other processor to execute the feature request right away.

5.5 The SCOOP Type System

- A *traitor* is an entity that statically is declared as non-separate, but during execution can become attached to a separate object.

5.5.1 Consistency Rules

Separate consistency rule (1): If the source of an attachment (assignment or argument passing) is separate, its target must also be separate.

```
r (buf: separate BUFFER [T]; x: T)
  local
    buf1: separate BUFFER [T]
    buf2: BUFFER [T]
    x2: separate T
  do
    buf1 := buf -- Valid
    buf2 := buf1 -- Invalid
    r (buf1, x2) -- Invalid
  end
```

Separate consistency rule (2): If an actual argument of a separate call is of reference type, the corresponding formal argument must be declared as separate.

```
  -- In class BUFFER [G]:
put (element: separate G)

  -- In another class:
store (buf: separate BUFFER [T]; x: T)
  do
    buf.put (x)
  end
```

Separate consistency rule (3): If the source of an attachment is the result of a separate call to a query returning a reference type, the target must be declared as separate.

```
  -- In class BUFFER [G]:
item: G

  -- In another class:
consume (buf: separate BUFFER [T])
  local
    element: separate T
  do
    element := buf.item
  end
```

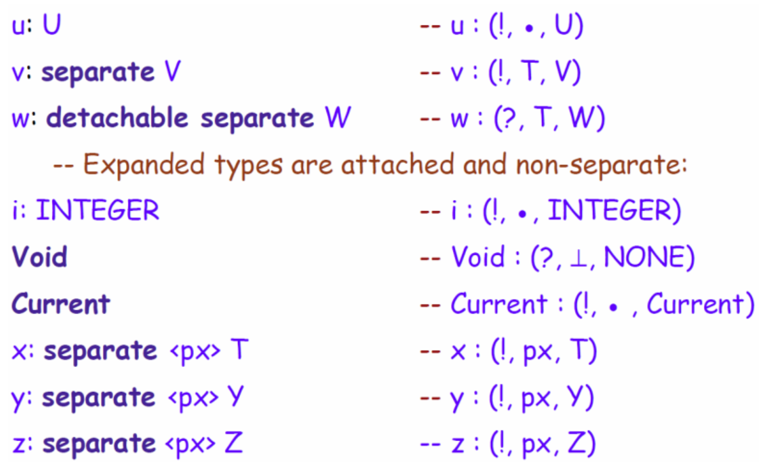
Separate consistency rule (4): If an actual argument or result of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

```
  -- In class BUFFER [G]:
put (element: G)
  -- G not declared separate

  -- In another class:
store (buf: separate BUFFER [E]; x: E)
  do
    buf l.put (x)
    -- E must be "fully expanded"
  end
```

- The rules
    - Prevent almost all traitors and are easy to understand.
    - Are too restrictive, don't support agents and there is no soundness proof.

5.5.2 The Type System of SCOOP

- We use the following notation: $\Gamma \vdash x: (\gamma, \alpha, C)$, where
    - Attachable/detachable, $\gamma \in \{!, ?\}$
    - Processor tag, $\alpha \in \{\cdot, T, \bot, \langle p \rangle, \langle a.\mathrm{handler} \rangle\}$
        - $\cdot$ is the current processor
        - T is some processor (top)
        - $\bot$ is no processor (bottom)
    - Ordinary (class) type C
- Examples



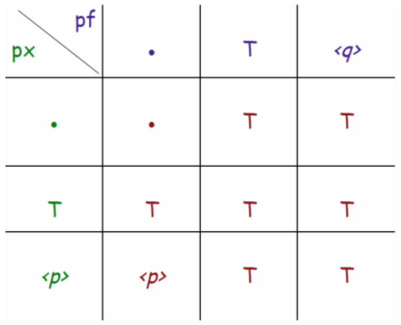
```
u: U                                   -- u : (!, •, U)
v: separate V                          -- v : (!, T, V)
w: detachable separate W               -- w : (?, T, W)
    -- Expanded types are attached and non-separate:
i: INTEGER                             -- i : (!, •, INTEGER)
Void                                   -- Void : (?, ⊥, NONE)
Current                                -- Current : (!, • , Current)
x: separate <px> T                     -- x : (!, px, T)
y: separate <px> Y                     -- y : (!, px, Y)
z: separate <px> Z                     -- z : (!, px, Z)
```

- We have the following subtyping rules
    - Conformance on class types are like in Eiffel, essentially based on inheritance:
$$D \leq_{\mathrm{Eiffel}} D \quad \Leftrightarrow \quad (\gamma, \alpha, D) \leq (\gamma, \alpha, C)$$
    - Attached ≤ detachable:
$$(!, \alpha, C) \leq (?, \alpha, C)$$
    - Any processor tag ≤ T:
$$(\gamma, \alpha, C) \leq (\gamma, T, C)$$
    - In particular, non-separate ≤ T:
$$(\gamma, \cdot, C) \leq (\gamma, T, C)$$
    - $\bot$ ≤ any processor tag:
$$(\gamma, \bot, C) \leq (\gamma, \alpha, C)$$
- Feature call rule
    - An expression `exp` of type (d, p, C) is called *controlled* if and only if `exp` is attached and satisfies *any* of the following conditions:
        - `exp` is non-separate, i.e., $p = \cdot$
        - `exp` appears in a routine `r` that has an attached formal argument a with the same handler as `exp`, i.e., $p = \mathrm{a.handler}$.

- A call `x.f(a)` appearing in the context of a routine `r` in class `C` is valid if and only if *both*:
  - `x` is controlled, and
  - `x`’s base class exports feature `f` to `C`, and the actual arguments conform in number and type to formal arguments of `f`.

- Result type combinator: What is the type T_{result} of a query call `x.f(…)`?

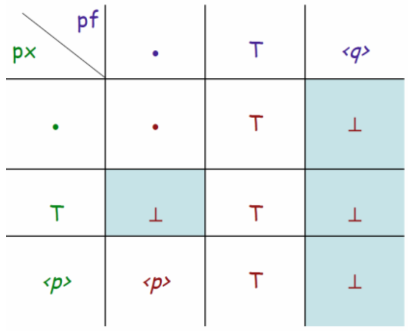
$$T_{\text{result}} = T_{\text{target}} * T_{\text{f}} = (\alpha_{\text{x}}, p_{\text{x}}, T_{\text{x}}) * (\alpha_{\text{f}}, p_{\text{f}}, T_{\text{f}}) = (\alpha_{\text{f}}, p_r, T_{\text{f}})$$



px \ pf	•	⊤	‹q›
•	•	⊤	⊤
⊤	⊤	⊤	⊤
‹p›	‹p›	⊤	⊤

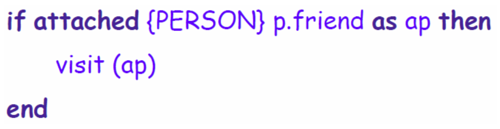
- Argument type combinator: What is the expected actual argument type in `x.f(a)`?

$$T_{\text{actual}} = T_{\text{target}} \otimes T_{\text{formal}} = (\alpha_{\text{x}}, p_{\text{x}}, T_{\text{x}}) \otimes (\alpha_{\text{f}}, p_{\text{f}}, T_{\text{f}}) = (\alpha_{\text{f}}, p_{\text{a}}, T_{\text{f}})$$



px \ pf	•	⊤	‹q›
•	•	⊤	⊥
⊤	⊥	⊤	⊥
‹p›	‹p›	⊤	⊥

- Expanded objects are always attached and non-separate, and both type combinators preserve expanded types.
- False traitors can be handled by object tests. An object test succeeds if the run-time type of its source conforms in all of
  - detachability,
  - locality, and
  - class type to the type of its target.

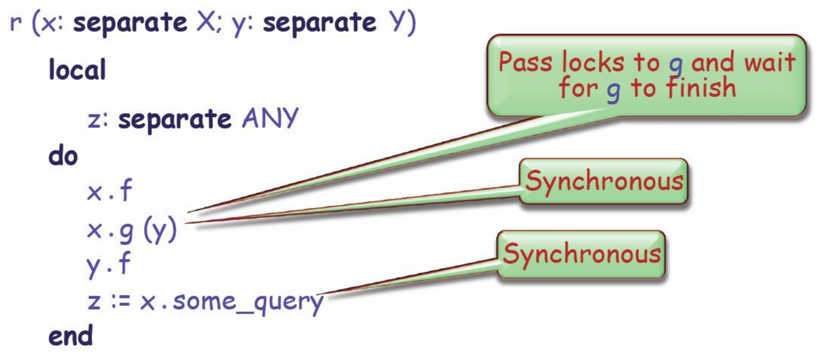


```
if attached {PERSON} p.friend as ap then
    visit (ap)
end
```

- Genericity
  - Entities of generic types may be separate, e.g. **`separate`** `LIST[BOOK]`
  - Actual generic parameters may be separate, e.g. `LIST[`**`separate`** `BOOK]`
  - All combinations are meaningful and useful.

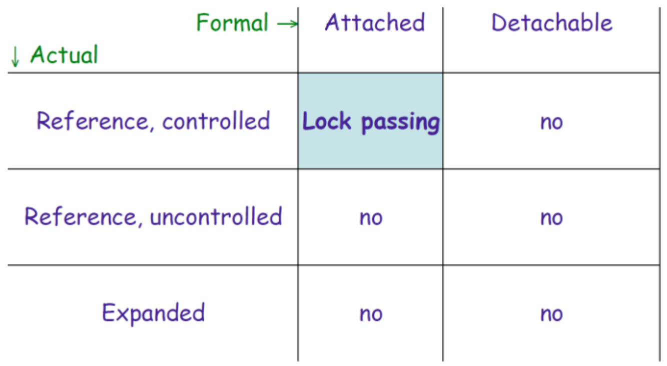
5.6 Lock Passing

- If a call `x.f(a1,…,an)` occurs in a routine `r` where one or more `ai` are controlled, the client's handler (the processor executing `r`) passes all currently held locks to the handler of `x`, and waits until `f` terminates (synchronous).



```
r (x: separate X; y: separate Y)
    local
        z: separate ANY
    do
        x.f
        x.g (y)
        y.f
        z := x.some_query
    end
```

- Lock passing combinations



Formal → / ↓ Actual	Attached	Detachable
Reference, controlled	**Lock passing**	no
Reference, uncontrolled	no	no
Expanded	no	no

5.7 Contracts

- Precondition
  - We have already seen that preconditions express the necessary requirement for a correct feature application, and that it is viewed as a synchronization mechanism:
    - A called feature cannot be executed unless the precondition holds.
    - A violated precondition delays the feature's execution.
  - The guarantees given to the supplier is exactly the same as with the traditional semantics.
- A postcondition describes the result of a feature's application. Postconditions are evaluated asynchronously; wait by necessity does not apply.
  - After returning from the call the client can only assume the controlled postcondition clauses.

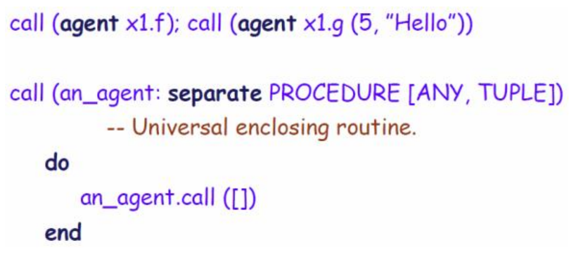
5.8 Inheritance

- SCOOP provides full support for inheritance (including multiple inheritance).
- Contracts
  - Preconditions may be kept or weakened (less waiting)
  - Postconditions may be kept or strengthened (more guarantees to the client)

    - Invariants may be kept or strengthened (more consistency conditions)
- Type redeclaration
    - Result types may be redefined covariantly for functions. For attributes the result type may not be redefined.
    - Formal argument types may be redefined contravariantly with regard to the processor tag.
    - Formal argument types may be redefined contravariantly with regard to detachable tags.

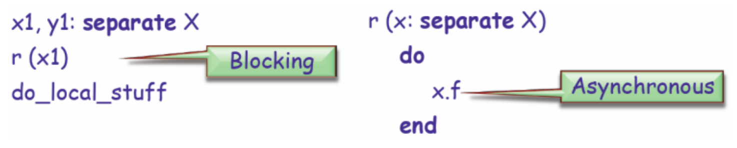
5.9 Agents

- There are no special type rules for separate agents. Semantic rule: an agent is created on its target's processor.
- Benefits
    - Convenience: we can have one universal enclosing routine:



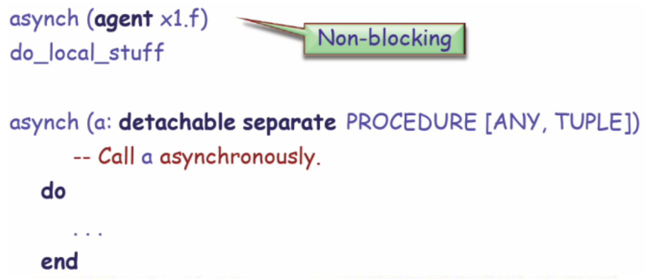
```
call (agent x1.f); call (agent x1.g (5, "Hello"))

call (an_agent: separate PROCEDURE [ANY, TUPLE])
        -- Universal enclosing routine.
    do
        an_agent.call ([])
    end
```

    - Full asynchrony: without agents, full asynchrony cannot be achieved.

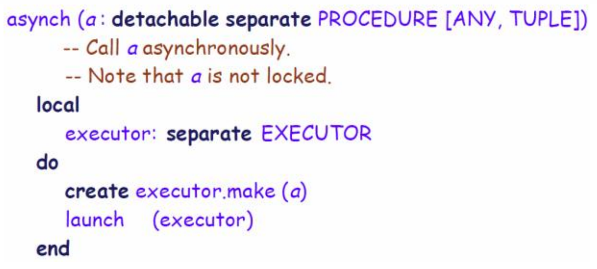


```
x1, y1: separate X              r (x: separate X)
r (x1)                              do
do_local_stuff                          x.f
                                    end
```

    - Full asynchrony is possible with agents.

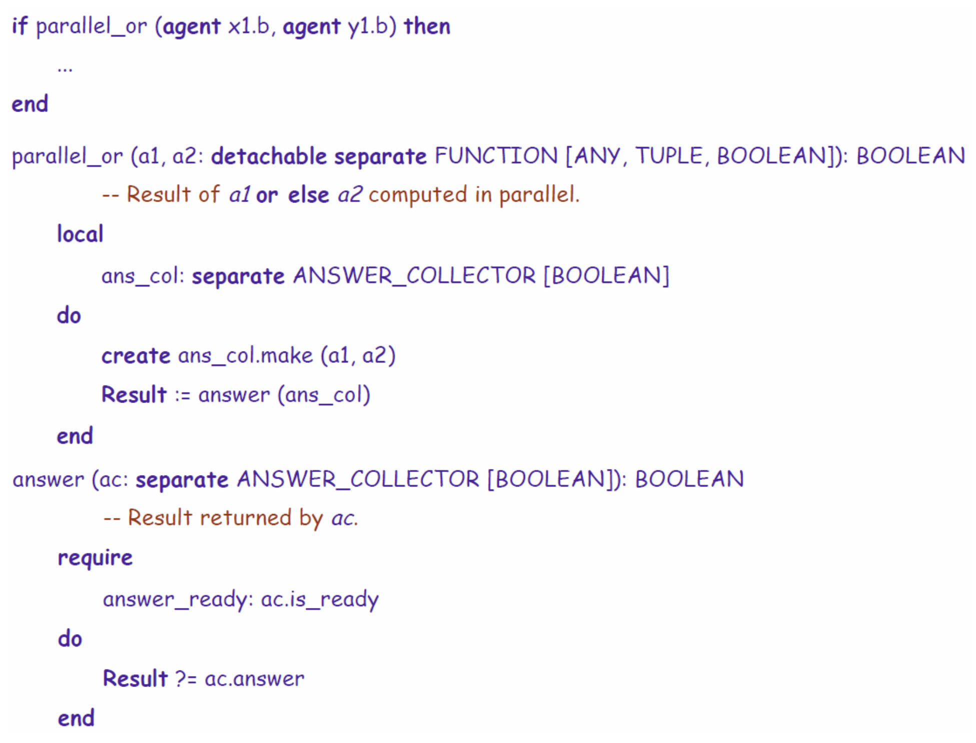


```
asynch (agent x1.f)
do_local_stuff

asynch (a: detachable separate PROCEDURE [ANY, TUPLE])
        -- Call a asynchronously.
    do
        . . .
    end
```



```
asynch (a : detachable separate PROCEDURE [ANY, TUPLE])
        -- Call a asynchronously.
        -- Note that a is not locked.
    local
        executor: separate EXECUTOR
    do
        create executor.make (a)
        launch    (executor)
    end
```

    - Waiting faster: What if we want to wait for one of two results?



```
if parallel_or (agent x1.b, agent y1.b) then
    ...
end

parallel_or (a1, a2: detachable separate FUNCTION [ANY, TUPLE, BOOLEAN]): BOOLEAN
        -- Result of a1 or else a2 computed in parallel.
    local
        ans_col: separate ANSWER_COLLECTOR [BOOLEAN]
    do
        create ans_col.make (a1, a2)
        Result := answer (ans_col)
    end
answer (ac: separate ANSWER_COLLECTOR [BOOLEAN]): BOOLEAN
        -- Result returned by ac.
    require
        answer_ready: ac.is_ready
    do
        Result ?= ac.answer
    end
```

5.10 Once Functions

- Separate functions are once-per-system.
- Non-separate functions are once-per-processor.

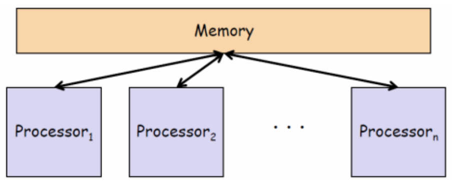
6 Review of Concurrent Languages

6.1 Computer Architectures for Concurrency

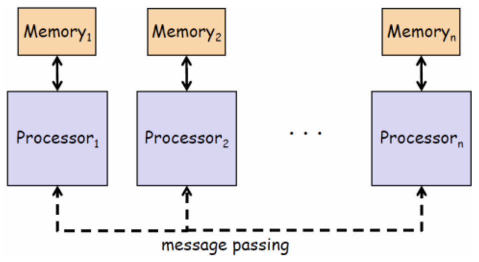
- We can classify computer architectures according to Flynn's taxonomy:

	Single Instruction	Multiple Instruction
Single Data	SISD	(uncommon)
Multiple Data	SIMD	MIMD

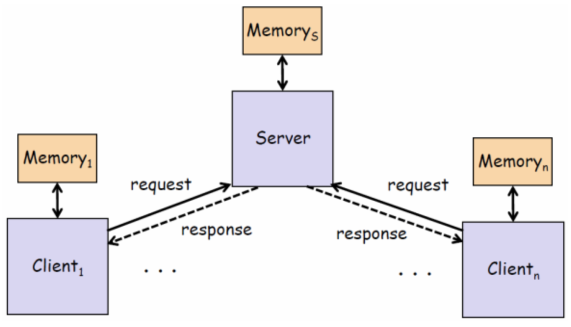
  - SISD: no parallelism (uniprocessor)
  - SIMD vector processor, GPU
  - MIMD: multiprocessing (predominant today)
- MIMD can be sublassified
  - SPMD (single program multiple data): All processors run the same program, but at independent speeds; no lockstep as in SIMD.
  - MPMD (multiple program multiple data): Often manager/worker strategy: manager distributes tasks, workers return result to the manager.
- Memory classification
  - Shared memory model
    - All processors share a common memory
    - Processes communicate by reading and writing shared variables (*shared memory communication*)



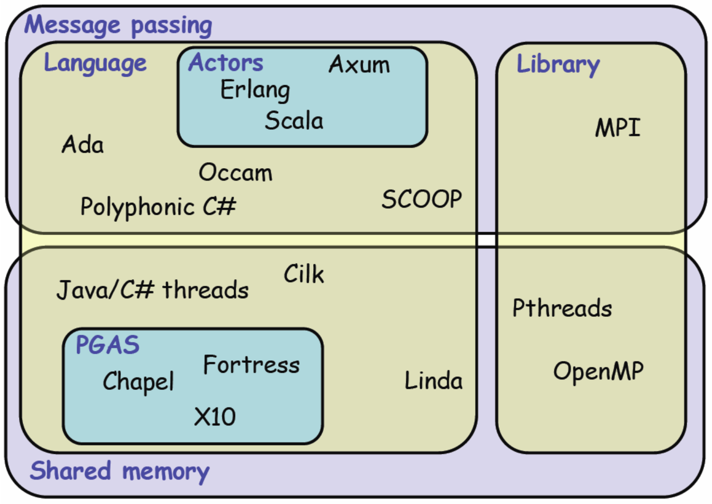
  - Distributed memory model
    - Every processor has its own local memory, which is inaccessible to others.
    - Processes communicate by sending messages (*message-passing communication*)



  - Client-server model
    - Describes a specific case of the distributed model



6.2 Classifying Approaches to Concurrency



6.3 The Actor Model in Erlang

- Process communication through asynchronous message passing; no shared state between actors.
- An *actor* is an entity which in response to a message it receives can
  - Send finitely many messages to other actors.
  - Determine new behaviour for messages it received in the future.
  - Create a finite set of new actors.
- Recipients of messages are identified by addresses; hence an actor can only communicate with actors whose addresses it has.
- A *message* consists of
  - the target to whom the communication is addressed, and
  - the content of the message.
- In Erlang, processes (i.e., actors) are created using `spawn`, which gives them a unique process identifier:

```
PID = spawn(module,function,arguments)
```

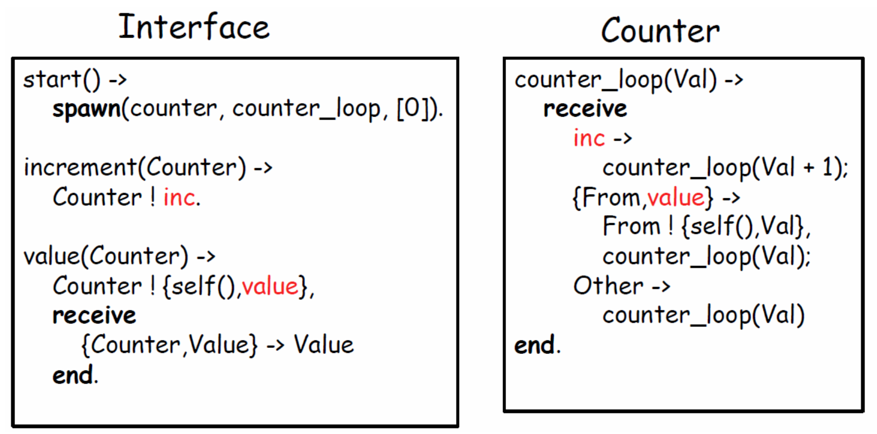
- Messages are sent by passing tuples to a PID with the `!` syntax.

```
PID ! {message}
```

- Messages are retrieved from the mailbox using the `receive` function with pattern matching.

```
receive
  message1 -> actions1;
  message2 -> actions2;
  …
end
```

- Example:



Interface

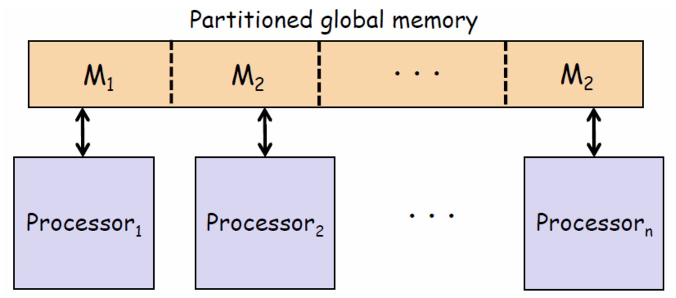
```
start() ->
   spawn(counter, counter_loop, [0]).

increment(Counter) ->
   Counter ! inc.

value(Counter) ->
   Counter ! {self(),value},
   receive
      {Counter,Value} -> Value
   end.
```

Counter

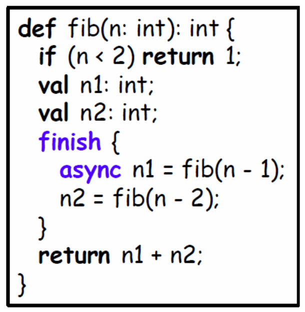
```
counter_loop(Val) ->
   receive
      inc ->
         counter_loop(Val + 1);
      {From,value} ->
         From ! {self(),Val},
         counter_loop(Val);
      Other ->
         counter_loop(Val)
end.
```

6.4 Partitioned Global Address Space (PGAS) Model and X10

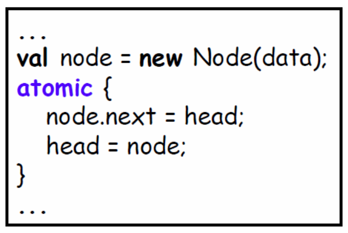
- Each processor has its own local memory, but the address space is unified. This allows processes on other processors to access remote data via simple assignment or dereference operations.



- X10 is an object-oriented language based on the PGAS model. New threads can be spawned asynchronously; asynchronous PGAS model
  - A memory partition and the threads operating on it are called a *place*.
- X10 operations
  - **async** S
    - Asynchronously spawns a new child thread executing S and returns immediately.
  - **finish** S
    - Executes S and waits until all asynchronously spawned child threads have terminated.
  - **atomic** S
    - Execute S atomically. S must be non-blocking, sequential and only access local data.



```
def fib(n: int): int {
  if (n < 2) return 1;
  val n1: int;
  val n2: int;
  finish {
    async n1 = fib(n - 1);
    n2 = fib(n - 2);
  }
  return n1 + n2;
}
```



```
...
val node = new Node(data);
atomic {
   node.next = head;
   head = node;
}
...
```

  - **when** (E) S

        - Conditional critical section: suspends the thread until `E` is true, then executes `S` atomically. `E` must be non-blocking, sequential, only access local data, and be side-effect free.

```
...
when (!buffer.full) {
   buffer.insert(item);
}
...
```

- **`at`** `(p) S`
    - Execute `S` at place `p`. This blocks the current thread until completion of `S`.

```
class C {
   var x: int;
   def this(n: int) { x = n; }
}

def increment(c: GlobalRef[C]) {
   at (c.home) c().x++;
}
```

7 Lock-Free Approaches

7.1 Problems with Locks

- Lock-based approaches in a shared memory system are error-prone, because it is easy to …
  - … *forget a lock*: danger of data races.
  - … take *too many locks*: danger of deadlock.
  - … take *locks in the wrong order*: danger of deadlock.
  - … take *the wrong lock*: the relation between the lock and the data it protects is not explicit in the program.
- Locks cause blocking:
  - Danger of *priority inversion*: if a lower-priority thread is pre-empted while holding a lock, higher-priority threads cannot proceed.
  - Danger of *convoying*: other threads queue up waiting while a thread holding a lock is blocked.
- Two concepts related to locks:
  - *Lock overhead* (increases with more locks): time for acquiring and releasing locks, and other resources.
  - *Lock contention* (decreases with more locks): the situation that multiple processes wait for the same lock.
- For performance, the developer has to carefully choose the granularity of locking; both lock overhead and contention need to be small.
- Locks are also problematic for designing fault-tolerant systems. If a faulty process halts while holding a lock, no other process can obtain the lock.
- Locks are also *not composable* in general, i.e., they don't support modular programming.
- One alternative to locking is a pure message-passing approach:
  - Since no data is shared, there is no need for locks.
  - Of course message-passing approaches have their own drawbacks, for example
    - Potentially large overhead of messaging.
    - The need to copy data which has to be shared.
    - Potentially slower access to data, e.g., to read-only data structures which need to be shared.
- If a shared-memory approach is preferred, the only alternative is to make the implementation of a concurrent program *lock-free*.
  - Lock-free programming using *atomic read-modify-write primitives*, such as compare and swap.
  - *Software transactional memory* (STM), a programming model based on the idea of database transactions.

7.2 Lock-Free Programming

- *Lock-free programming* is the idea to write shared-memory concurrent programs that don't use locks but can still ensure thread-safety.

- Instead of locks, use stronger atomic operations such as compare-and-swap. These primitives typically have to be provided by hardware.
- Coming up with general lock-free algorithms is hard, therefore one usually focuses on developing lock-free data structures like stacks or buffers.

- For lock-free algorithms, one typically distinguishes the following two classes:
  - Lock-free: some process completes in a finite number of steps (freedom from deadlock).
  - Wait-free: all processes complete in a finite number of stops (freedom from starvation).
- *Compare-and-swap* (CAS) takes three parameters; the address of a memory location, an old and a new value. The new value is atomically written to the memory location if the content of the location agrees with the old value:

```
CAS (x, old, new)
  do
    if *x = old then
      *x := new;
      result := true
    else
      result := false
    end
  end
```

7.2.1 Lock-Free Stack

- A common pattern in lock-free algorithms is to read a value from the current state, compute an update based on the value just read, and then atomically update the state by swapping the new for the old value.

```
class Node {
   Node* next;
   int item;
}

Node* head; // top of the stack
```

```
void push (int value) {
   Node* oldHead;
   Node* newHead := new Node();
   node.item := value;
   do {
      oldHead := head;
      newHead.next := head;
   } while (!CAS(&head, oldHead, newHead));
}
```

```
int pop () {
   Node* oldHead;
   Node* newHead;
   do {
      oldHead := head;
      if(oldHead = null) return EMPTY;
      newHead := oldHead.next;
   } while(!CAS(&head, oldHead, newHead));
   return oldHead.item;
}
```

7.2.2 The ABA problem

- The *ABA* problem can be described as follows:
  - A value is read from state A.
  - The state is changed to state B.
  - The CAS operation does not distinguish between A and B, so it assumes that it is still A.

- The problem is be avoided in the simple stack implementation (above) as `push` always puts a *new* node, and the old node's memory location is not free yet (if the memory address would be reused).

7.2.3 Hierarchy of Atomic Primitives

- Atomic primitives can be ranked according to their *consensus number*. This is the maximum number of processes for which the primitive can implement a consensus protocol (a group of processes agree on a common value).
- In a system of n or more processes, it is impossible to provide a wait-free implementation of a primitive with consensus number of n from a primitive with lower consensus number.

#	Primitive
1	Read/write register
2	Test-and-set, fetch-and-add, queue, stack, …
…	…
2n-2	n-register assignment
…	…
∞	Memory-to-memory move and swap, compare-and-swap, load-link/store-conditional

- Another primitive is *load-link/store-conditional*, a pair of instructions which together implement an atomic read-modify-write operation.
    - Load-link returns the current value of a memory location.
    - A subsequent store-conditional to the same location will store a new value only if no updates have occurred to that location since the load-link.
    - The store-conditional is guaranteed to fail if no updates have occurred, even if the value read by the load-link has since been restored.

7.3 Linearizability

- Linearizability provides a *correctness condition* for concurrent objects. Intuition:
- A call of an operation is split into two events:
    - Invocation: `[A q.op(a1,…,an)]`
    - Response: `[A q:Ok(r)]`
    - Notation:
        - `A`: thread ID
        - `q`: object
        - `op(a1,…,an)`: invocation of call with arguments
        - `Ok(r)`: successful response of call with result `r`
- A *history* is a sequence of invocation and response events.
    - Histories can be projected on objects (written as `H|q`) and on threads (denoted by `H|A`).
    - A response *matches* an invocation if their object and their thread names agree.
    - A history is *sequential* if it starts with an invocation and each invocation, except possibly the last, is immediately followed by a matching response.

    - A sequential history is *legal* if it agrees with the sequential specification of each object.
- A call op_1 *precedes* another call op_2 (written as $\mathrm{op}_1 \to \mathrm{op}_2$), if op_1's response event occurs before op_2's invocation event. We write $\to_H$ for the precedence relation induced by H.
    - An invocation is *pending* if it has no matching response.
    - A history is *complete* if it does not have any pending invocations.
    - $\mathrm{complete}(H)$ is the subhistory of H with all pending invocations removed.
- Two histories H and G are *equivalent* if $H|A = G|A$ for all threads A.
- A history H is linearizable if it can be extended by appending zero or more response events to a history G such that
    - $\mathrm{complete}(G)$ is equivalent to a legal sequential history S, and
    - $\to_H \subseteq \to_S$
- A history H is sequentially consistent if it can be extended by appending zero or more response events to a history G such that
    - $\mathrm{complete}(G)$ is equivalent to a legal sequential history S, and
- Intuition: Calls from a particular thread appear to take place in program order.
- Compositionality
    - Every linearizable history is also sequentially consistent.
    - Linearizability is *compositional*: H is linearizable if and only if for every object x the history $H|x$ is linearizable. Sequential consistency on the other hand is not compositional.

8 Calculus of Communicating Systems (CCS)

- CCS has been introduced by Milner in 1980 with the following model in mind:
  - A concurrent system is a collection of processes.
  - A process is an independent agent that may perform internal activities in isolation or may interact with the environment to perform shared activities.
- Milner's insight: Concurrent processes have an algebraic structure.
- This is why a process calculus is sometimes called a process algebra.

8.1 Syntax of CCS

- Terminal process
  - The simples possible behaviour is *no behaviour*, which is modelled by the terminal process 0 (pronounced "nil").
  - 0 is the only atomic process of CCS.
- Names and actions
  - We assume an infinite set $\mathcal{A}$ of *port names*, and a set $\bar{\mathcal{A}} = \{\bar{a} \mid a \in \mathcal{A}\}$ of *complementary port names*.
  - When modelling we use a name a to denote an *input action*, i.e., the receiving of input from the associated port a.
  - We use a co-name $\bar{a}$ to denote an output action, i.e., the sending of output to the associated port a.
  - We use τ to denote the distinguished internal action.
  - The set of actions Act is therefore given as $\text{Act} = \mathcal{A} \cup \bar{\mathcal{A}} \cup \{\tau\}$.
- Action prefix
  - The simplest actual behaviour is sequential behaviour. If P is a process, we write $\alpha.P$ to denote the *prefixing* of P with an *action* α.
  - $\alpha.P$ models a system that is ready to perform the action α, and then behave as P, i.e.,

$$\alpha.P \xrightarrow{\alpha} P$$

- Process interfaces
  - The set of *input* and *output actions* that a process P may perform in isolation constitutes the interface of P.
  - The interface enumerates the ports that P may use to interact with the environment. For instance, the interface of a coffee and tea machine might be

$$\text{tea}, \text{coffee}, \text{coin}, \overline{\text{cup\_of\_tea}}, \overline{\text{cup\_of\_coffee}}$$

- Non-deterministic choice
  - A more advanced sequential behaviour is that of alternative behaviours.
  - If P and Q are processes then we write $P + Q$ to denote the *non-deterministic choice* between P and Q. This process can either behave as P (discarding Q), or as Q (discarding P).
- Process constants and recursion
  - The most advanced sequential behaviour is recursive behaviour.

    - A process may be the invocation of a *process constant* $K \in \mathcal{K}$. This is only meaningful if K is defined beforehand.
    - If K is a process constant and P a process, we write $K \stackrel{\text{def}}{=} P$ to give a *recursive definition* of K (recursive if P involves K).
- Parallel composition
    - If P and Q are processes, we write $P|Q$ to denote the parallel composition of P and Q.
    - $P|Q$ models a process that behaves like P and Q in parallel:
        - Each may proceed independently.
        - If P is ready to perform an action a and Q is ready to perform the complementary action $\bar{a}$, they may interact.
    - Example:

$$\begin{array}{rl}
 & (\mathit{tea}.\mathit{coin}.\overline{\mathit{cup\_of\_tea}}.\mathrm{CTM} + \mathit{coffee}.\mathit{coin}.\mathit{coin}.\overline{\mathit{cup\_of\_coffee}}.\mathrm{CTM}) \\
| & (\overline{\mathit{tea}}.\overline{\mathit{coin}}.\mathit{cup\_of\_tea}.\overline{\mathit{teach}}.\mathrm{CS} + \overline{\mathit{coffee}}.\overline{\mathit{coin}}.\overline{\mathit{coin}}.\mathit{cup\_of\_coffee}.\overline{\mathit{publish}}.\mathrm{CS}) \\
\xrightarrow{\tau} & (\mathit{coin}.\overline{\mathit{cup\_of\_tea}}.\mathrm{CTM}) \,|\, (\overline{\mathit{coin}}.\mathit{cup\_of\_tea}.\overline{\mathit{teach}}.\mathrm{CS}) \\
\xrightarrow{\tau} & (\overline{\mathit{cup\_of\_tea}}.\mathrm{CTM}) \,|\, (\mathit{cup\_of\_tea}.\overline{\mathit{teach}}.\mathrm{CS}) \\
\xrightarrow{\tau} & \mathrm{CTM} \,|\, (\overline{\mathit{teach}}.\mathrm{CS}) \\
\xrightarrow{\overline{\mathit{teach}}} & \mathrm{CTM} \,|\, \mathrm{CS}
\end{array}$$

    - Note that the synchronization of actions such as tea/$\overline{\text{tea}}$ is expressed by a τ-action (i.e., regarded as an internal step).
- Restriction
    - We control unwanted interactions with the environment by restricting the scope of port names.
    - If P is a process and A is a set of port names we write $P \backslash A$ for the *restriction* of the scope of each name in A to P.
    - This removes each name $a \in A$ and the corresponding co-name $\bar{a}$ from the interface of P.
    - In particular, this makes each name $a \in A$ and the corresponding co-name $\bar{a}$ inaccessible to the environment.
    - For instance, we can restrict the coffee-button on the coffee and tea machine, which means CS can only teach, and never publish (i.e., in $(\mathrm{CTM} \setminus \{\mathrm{coffee}\})|CS$)
- Summary

$$\begin{array}{lll}
P ::= & K & | \quad \text{process constants } (K \in \mathcal{K}) \\
 & \alpha.P & | \quad \text{prefixing } (\alpha \in Act) \\
 & \sum_{i \in I} P_i & | \quad \text{summation } (I \text{ is an arbitrary index set}) \\
 & P_1 | P_2 & | \quad \text{parallel composition} \\
 & P \setminus L & \quad\;\; \text{restriction } (L \subseteq \mathcal{A})
\end{array}$$

- Notation
    - We use $P_1 + P_2 = \sum_{i \in \{1,2\}} P_i$ and $0 = \sum_{i \in \emptyset} P_i$

8.2 Operational Semantics of CCS

- We formalize the execution of a CCS process with labelled transition systems (LTS).

- A *labelled transition system* (LTS) is a triple $\left(\mathrm{Proc}, \mathrm{Act}, \left\{\xrightarrow{\alpha} \mid \alpha \in \mathrm{Act}\right\}\right)$ where
  - Proc is a set of *processes* (the *states*),
  - Act is a set of *actions* (the *labels*), and
  - for every $\alpha \in \mathrm{Act}$, $\xrightarrow{\alpha} \subseteq \mathrm{Proc} \times \mathrm{Proc}$ is a binary relation on processes called the *transition relation*.
  - It is customary to distinguish the initial process.
- A finite LTS can be drawn where processes are nodes, and transitions are edges. Informally, the semantics of an LTS are as follows:
  - Terminal process: $0 \nrightarrow$
  - Action prefixing: $\alpha . P \xrightarrow{\alpha} P$
  - Non-deterministic choice: $P \xleftarrow{\alpha} \alpha . P + \beta . Q \xrightarrow{\beta} Q$
  - Recursion: $X \overset{\text{def}}{=} \cdots . \alpha . X$

$$\begin{tikzcd}
{\color{red}X} \arrow[r, bend left, "\cdots"] & \alpha.X \arrow[l, bend left, "\alpha"]
\end{tikzcd}$$

  - Parallel composition: $\alpha . P | \beta . Q$ (two possibilities)

$$\begin{tikzcd}
& P \mid \beta.Q \arrow[dr, "\beta"] & \\
{\color{red}\alpha.P \mid \beta.Q} \arrow[ur, "\alpha"] \arrow[dr, "\beta"'] & & P \mid Q \\
& \alpha.P \mid Q \arrow[ur, "\alpha"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
& P \mid \bar{a}.Q \arrow[dr, "\bar{a}"] & \\
{\color{red}a.P \mid \bar{a}.Q} \arrow[ur, "a"] \arrow[rr, "\tau"] \arrow[dr, "\bar{a}"'] & & P \mid Q \\
& a.P \mid Q \arrow[ur, "a"'] &
\end{tikzcd}$$

- Formally, we can give small step operational semantics for CCS

$$\text{ACT}\quad \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad\qquad \text{SUM}_j\quad \frac{P_j \xrightarrow{\alpha} P_j'}{\sum_{i\in I} P_i \xrightarrow{\alpha} P_j'}\quad j \in I$$

$$\text{COM1}\quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad\qquad \text{COM2}\quad \frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$$

$$\text{COM3}\quad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\bar{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\text{RES}\quad \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\quad \alpha, \overline{\alpha} \notin L \qquad\qquad \text{CON}\quad \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'}\quad K \overset{\text{def}}{=} P$$

8.3 Behavioural Equivalence

- We would like to express that two concurrent systems “behave in the same way”. We are not interested in syntactical equivalence, but only in the fact that the processes have the same behaviour.

- The main idea is that two processes are behaviourally equivalent if and only if an *external observer* cannot tell them apart. This is formulated by bisimulation.

8.3.1 Strong Bisimilarity and Bisimulation

- Let $\left(\mathrm{Proc}, \mathrm{Act}, \left\{\xrightarrow{\alpha} \mid \alpha \in \mathrm{Act}\right\}\right)$ be an LTS.
- A binary relation $R \subseteq \mathrm{Proc} \times \mathrm{Proc}$ is a *strong bisimulation* if and only if whenever $(P, Q) \in R$, then for each $\alpha \in \mathrm{Act}$:
  - If $P \xrightarrow{\alpha} P'$ then $Q \xrightarrow{\alpha} Q'$ for some Q' such that $(P', Q') \in R$
  - If $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha} P'$ for some P' such that $(P', Q') \in R$
- Two processes $P_1, P_2 \in \mathrm{Proc}$ are *strongly bisimilar* ($P_1 \sim P_2$) if and only if there exists a strong bisumulation R such that $(P_1, P_2) \in R$

$$\sim \; = \bigcup_{R, R \text{ is a strong bisimulation}} R$$

8.3.2 Weak Bisumulation

- We would like to state that two processes Spec and Imp behave the same, where Imp specifies the computation in greater detail. Strong bisimulaitons force every action to have an equivalent in the other process. Therefore, we abstract from the *internal* actions.
- We write $p \overset{\alpha}{\Longrightarrow} q$ if p can reach q via an α-transition, preceded and followed by zero or more τ-transitions. Furthermore, $p \overset{\alpha}{\Longrightarrow} q$ holds if $p = q$.
  - This definition allows us to "erase" sequences of τ-transitions in a new definition of behavioural equivalence.
- Let $\left(\mathrm{Proc}, \mathrm{Act}, \left\{\xrightarrow{\alpha} \mid \alpha \in \mathrm{Act}\right\}\right)$ be an LTS.
- A binary relation $R \subseteq \mathrm{Proc} \times \mathrm{Proc}$ is a *weak bisimulation* if $(P, Q) \in R$ implies for each $\alpha \in \mathrm{Act}$:
  - If $P \xrightarrow{\alpha} P'$ then $Q \overset{\alpha}{\Longrightarrow} Q'$ for some Q' such that $(P', Q') \in R$
  - If $Q \xrightarrow{\alpha} Q'$ then $P \overset{\alpha}{\Longrightarrow} P'$ for some P' such that $(P', Q') \in R$
- Two processes $P_1, P_2 \in \mathrm{Proc}$ are *weakly bisimilar* ($P_1 \approx P_2$) if and only if there exists a strong bisumulation R such that $(P_1, P_2) \in R$

8.4 π-calculus

- In CCS, all communication links are static. For instance, a server might increment a value it receives.
- To remove this restriction, the π-calculus allows values to include channel names. A server that increments values can be programmed as

$$S = a(x, b).\overline{b}\langle x + 1\rangle.0 \mid S$$

  - The angle brackets are used to denote output tuples.