III. Basic Interaction and Cooperation Mechanisms

Basis:

(c.f. Definition of Distributed Systems)

- *Hardware:* Distributed, heterogenous compute nodes connected by various types of networks.
- *Software:* Application and operating system processes (or threads) in a mix of *sequential, nondeterministic interleaving or truly parallel* execution on the same or different nodes.

Target Distributed Systems Model:

▶ System(s) of **active processes** get some *common* work done

$\Longrightarrow$ *Information exchange through interaction mechanisms*

▶ **Interaction** defines two principal **Roles**: '*provider*' vs. '*recipient*'

- Shared Memory: 'writer' vs. 'reader'
- Distributed Memory: 'sender' vs. 'receiver'

$\Longrightarrow$ *Information exchange introduces a* **causal dependency** *where the recipient depends on the provider.*

Preview: Dependency degrees and coupling

1. **No** direct or in-direct **interaction** $\Longrightarrow$ *no coupling*
   asynchronous processes with isolated local states only
2. **Signal**: **existence** (pure) of information representation
   'pure' synchronization $\Longrightarrow$ *very tight coupling*
   Example: P_r `waits` for `signal` from P_p; check for non-`null` ref.
3. **Data: existence plus content** $\Longrightarrow$ *tight coupling*
   Example: P_r reads content of data or message provided by P_p
4. **Task**: **existence plus content plus interpretation**
   of information representation $\Longrightarrow$ *moderate coupling*
   Expl.: Server P_r gets method call and parameters from client P_p
5. **Self-contained**: **existence plus content plus procedure**
   to deal with content $\Longrightarrow$ *loose coupling*
   Expl.: P_r gets method call, parameters and executable from P_p

III.1 Process System Model

1. **Static Structure** prescribes the scope for behavior
   - a finite set of **States** Q with a specific initial state $q_0 \in Q$ and a (possibly empty) set of final states Q_F
   - a finite set of **Rules** R describing all system steps permitted, i.e., all allowed state changes $\rightarrow \in Q \times Q$
2. **Dynamic Behavior** results from possibly infinite sequences of state changes, so-called **processes** that adhere to all rules of R.

Definition III.1: (Process)

Process *is a* **sequence of steps** $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \ldots$ *where*

- *each step is performed by an* **action** a_{i+1} *that results in a state change* $q_i \rightarrow q_{i+1}$ *(*$i \in [0 : \infty]$*)*
- *all resulting states* q_i *of a process are permissible, i.e.,* $q_i \in Q$
- *the action for each step is permitted by some rule in* R
- *each single action is assumed to be* **atomic**, *i.e., indivisible.* ♦

Processes as Execution Sequences

Two dual aspects of a process are equivalent:

- *passive view* as a sequence (trace) of states $q_0, q_1, \ldots$
- *active view* as a sequence (trace) of actions $a_1, a_2, \ldots$

Note: '*Trace*' is often used for single execution sequences

Sequential vs. Parallel Execution

- A **single process** as a sequence of steps is always **sequential**
  - the steps of the same process are executed in a *linear total order*
  - for any two actions a and b in a process holds: $(a \sqsubset b)$ or $(b \sqsubset a)$
- **Parallelism** is made possible by potential and execution means:
  - Execution of several processes in different components that are (more or less) *independent* from each other (*partial order*).
  - Combining more than one subsystem to a new system that allows for more than one execution step at the same time.

Definition III.2: (Process System Model)

1. *A* **process system** PS *consists of a set* $\{P_1, \ldots, P_n\}$ *of sequential processes* $P_1, \ldots, P_n$.
2. *The* **Action Execution Order** $\sqsubset_{PS}$ **of** PS *describes the admissible overall execution orders of all processes of* PS*:*
   (a) *For each* $P \in PS$ *all actions are in a linear, sequential order.*
   (b) *Actions from different processes* $P_a \neq P_b$ *are not ordered and said to be independent and* **concurrent**.
   (c) $\sqsubset_{PS}$ *is an irreflexive, asymmetric and transitive* **partial order** *relation among the actions of all processes of* PS ♦

- The starting point of any PS is an almost unordered system where each process adheres to its own linear order but all actions from different processes are completely independent from each other.
- Introducing interaction among processes defines additional dependencies and restricts the overall execution space.

Definition III.3: (Properties of Processes and Systems)

1. P **terminates:** *P is in a state $q \in Q$ where the rule R permits no more actions* $\Longrightarrow q \in Q_F$ *is a final state.*
2. PS *is* **deterministic** $:\Longleftrightarrow$
   $\forall\ q \in Q\ \exists$ **exactly one** *process P starting with q.*
3. PS *is* **determinate** $:\Longleftrightarrow \forall\ q \in Q$ *holds: All terminating processes starting with $q \in Q$ also end in the same finite state q'.*

♦

- A non-terminating process has no well-defined final state, hence, Def.III.3 (3) does not apply.
- **Deterministic vs. Determinate Systems**:
  - $*$ a deterministic system is always also determinate.
  - $*$ a determinate system is not required to be deterministic.

  Determinacy is more relaxed but ensures the correct result.

Two Aspects of Non-Determinism

◀ **Don't Know**-Non-Determinism as a *modeling mechanism*

- complex systems with high spectrum of possible behavior
- not every detail can be captured in a model of reasonable size
- worst-case-assumptions are needed to avoid system malfunctions

Examples: securing a user interface
integration of a black-box device in a software system

▶ **Don't Care**-Non-Determinism as an *abstraction mechanism*

- problem allows for different approaches to find 'a' solution
- system allows for tolerance in execution orders due to independent 'concurrent' system parts
- different execution orders end up with the same results

Examples: non-deterministic automata models
evaluation orders for expressions, functional languages

Specification Level vs. Execution Level

Programs specify the rules R that restrict the **processes** of a system.

1. A **sequential** program specifies exactly one process P.
2. A **non-deterministic** program specifies an arbitrary process P_i of a possibly infinite set $\{ P_1, P_2, \ldots \}$ of processes.
3. A **concurrent** program specifies a set $\{ P_1, P_2, \ldots\}$ of more or less independent processes with partially ordered actions.

$\Longrightarrow$ *'Concurrency' is a specification level property*.

Program execution uses the potential of relaxed strictness tailored to the underlying hardware:

- Availability of more than one processor allows for *true parallelism*.
- Otherwise, *non-deterministic interleaving* (pseudo-parallel execution) by choosing an overall total execution order $\sqsubset_{ex}$ that respects the partial order $\sqsubset_{PS}$ ($\sqsubset_{PS} \subseteq \sqsubset_{ex}$) becomes possible.

$\Longrightarrow$ *'Parallelism' is an execution level property*.

Example: Concurrency, Parallelism and Atomicity

- Concurrent program does not imply an order on a_1 and a_2

  $\texttt{i := 5; parbegin \{ }\underbrace{\texttt{i++}}_{a_1}\texttt{ \} } \| \texttt{ \{ }\underbrace{\texttt{i++}}_{a_2}\texttt{ \} parend; y := i; end;}$

- 'Three' admissible execution orders:
  - quasi-parallel execution orders $a_1; a_2$ or $a_2; a_1$ $\Longrightarrow$ `y` $\approx$ 7
  - truly parallel execution $a_1 \| a_2$ $\Longrightarrow$ `y` $\approx$ 6

    actions read simultaneously 5, increment by 1 and write 6 back

  $\Longrightarrow$ *Pseudo-Parallelism does not simulate true parallelism.*

- On execution level there are not only 2, but 'at least' 6 actions:

  `i++` $\approx$ `i=i+1` $\approx$ `LD R1,i; INC R1; ST R1,i`

  - `LoaD` $\approx$ MEM$\rightarrow$Register / `STore` $\approx$ Register$\rightarrow$MEM
  - OS-scheduling may interrupt processes at all times
  - Example: `LD_a1; INC_a1; LD_a2; ST_a1; INC_a2; ST_a2;`

  - more than three admissible execution orders on machine level

  $\Longrightarrow$ *Atomicity must not be assumed at program level.*

III.2 Coordination, Interaction and Causality

- **Coordination** is needed to guide the work of different processes running in a distributed system towards a common goal, e.g.,
  - ∗ organizing a parallel search in a bunch of documents
  - ∗ reaching a common agreement concerning a trusted entity
- **Interaction** is required to implement coordination efforts.
  - a process **publishes** (part of) it's state, e.g.,
    - ∗ handing a file descriptor to another process, e.g. OS
    - ∗ forwarding a search result for filtering
  - a process **re-acts** based on the state of another process, e.g.,
    - ∗ Printing a file when signalled as 'completed'
    - ∗ Continue with a calculation when all partial results are available

Interaction is implemented based on the underlying paradigm:

- SMS: Writing and reading the same data
- DMS: Sending and receiving messages

Dependencies, Causality and Execution Orders

- ▶ Interaction defines two principal **Roles**: '*provider*' vs. '*recipient*' c.f. pg. III-1
- ◀ Interaction introduces **dependencies** between actors of these roles:
  - ∗ SMS: 'Reader' depends on 'Writer' to provide current data
  - ∗ DMS: 'Receiver' depends on 'Sender' for sending message

Principle of Causality: *'Cause and Effect'*

- Information is only accessible *after* it has been provided.
- The *execution order* between different parts of an interaction is essential for the overall result, e.g.,
  - ◀ A Reader may read outdated (current) data when the read access is executed before (after) the corresponding write actions, which may violate determinacy.
  - ◀ A Receiver may be blocked from further execution while waiting for a message that does not arrive which results in a non-terminating 'blocked' system.

Example: One-way Interaction and Dependency

(i) Shared-Memory-Model

```
g=0; parbegin {g=F(3); a=2*g} // {x=3*g} parend
```

∗ P2 reads variable written by P1

$\Longrightarrow$ **Execution order is important for P2, but not specified**

SMS *requires additional restriction on execution order* $\sqsubset_{PS}$

(ii) Message-Passing-Model: two alternative specifications (A/B)

```
(A)     parbegin  {g=0; g=F(3); snd(g,P2); a=2*g}
            //    {g=0;                   rcv(g,P1); x=3*g}   parend

(B)     parbegin  {g=0; snd(g,P2); g=F(3);  a=2*g}
           //     {g=0;           rcv(g,P1); x=3*g}           parend
```

∗ P2 receives and uses ('old' vs. 'new') value from sender P1

$\Longrightarrow$ **Execution order is important and specified via** `snd/rcv`

DMS: *Access and order restriction in a single statement*

Example: Mutual Interaction and Dependency

(i) Shared-Memory-Model

```
g = 0; h=0;  parbegin {g=h+1} // {h=g+2} parend;
```

- Processes P1 and P2 read and write the same variable:
  P1 before P2: $g \leftarrow 1$ and $h \leftarrow 3$
  P2 before P1: $g \leftarrow 3$ and $h \leftarrow 2$

$\Longrightarrow$ **Execution order relevant for both processes.**

Additional Problem: true parallelism and atomicity

P1 and P2 in parallel: $g \leftarrow 1$ and $h \leftarrow 2$

c.f. pg. III-9

$\Longrightarrow$ **Execution sequences without interrupts are needed**, i.e., guaranteeing atomicity for sequences consisting of many steps.

(ii) Message-Passing-Model: no simple one-to-one correspondence,

but sending and receiving of data among more than two processes causes similar effects and problems.

A Dependency Model for SMS and DMS – 1

Abstract Model independent of underlying programming model

- Information exchange is implemented by data exchange.
- Supplying data by a *provider role* is done by **Write**-Actions.
- Using data by a *recipient role* is done by **Read**-Actions.

Definition III.4: (Basic Interaction Primitives)

Let PS be a process system, P a process and a an action of P.

1. *The effect of an action a on the state of a process P is characterized by the following sets of data used:*

   $Read(a)$ *denotes all* **data read in** a

   $Write(a)$ *denotes all* **data written in** a

   $Data(a) := Read(a) \cup Write(a)$ *denotes all data* **used** *by* a.
2. *The overall effect of P is defined by it's overall data usage*

   $Read(P) := \bigcup_{a \in P} Read(a)$; $Write(P)$, $Data(P)$ *resp.* ♦

A Dependency Model for SMS and DMS – 2

- The **basic inter-action step** between two processes P_1 and P_2 is executed by actions $a_1 \in P_1$ and $a_2 \in P_2$ such that $Data(a_1) \cap Data(a_2) \neq \emptyset$.
- The detailed structure of $Data(a_1) \cap Data(a_2)$ defines the type of dependency between the actions and, hence, the processes.

Definition III.5: (Different Types of Dependencies)

Let PS be a process system and $a_1 \in P_1$; $a_2 \in P_2$ actions.

1. **One-way Read-Write-Dependency**:
   $a_1 \xrightarrow{rw} a_2 \;:\Longleftrightarrow\; Write(a_2) \cap Read(a_1) \neq \emptyset$
2. **Mutual Read-Write-Dependency**:
   $a_1 \xleftrightarrow{mu} a_2 \;:\Longleftrightarrow\; (a_1 \xrightarrow{rw} a_2) \wedge (a_2 \xrightarrow{rw} a_1)$
3. **Mutual Write-Write Conflict**:
   $a_1 \xleftrightarrow{ww} a_2 \;:\Longleftrightarrow\; Write(a_1) \cap Write(a_2) \neq \emptyset$ ♦

Example: Write-Write Dependencies

($\overset{ww}{\longleftrightarrow}$):

$$\texttt{x := 1; z := x; a := 3;}$$
$$\texttt{parbegin } \underbrace{\texttt{y := x+z}}_{a_1} \parallel \underbrace{\texttt{y := 2*a;}}_{a_2} \texttt{ parend;}$$
$$\texttt{a := x+y;}$$

$a_1 \overset{ww}{\longleftrightarrow} a_2$, because $Write(a_1) \cap Write(a_2) = \{y\} \neq \emptyset$

- ▷ a_1 executed before a_2 $\Longrightarrow$ a $\approx$ 7
- ▷ a_2 executed before a_1 $\Longrightarrow$ a $\approx$ 3
- ▷ $a_1 \parallel a_2$ executed truly parallel $\Longrightarrow Value(y) \in \{2, 6\}$ undefined

Note: *Dependency relation is not always transitive!*

... parbegin ... x := y+1 ... $\parallel$... y := 2*z ... $\parallel$... z := a+b ... parend; ...

- $a_1 \overset{rw}{\longrightarrow} a_2$ caused by y and $a_2 \overset{rw}{\longrightarrow} a_3$ caused by z
- a_1 and a_3 with disjoint $Read/Write$-sets are 'independent'.

Problem: if a_1 is executed after a_2 $\Longrightarrow$ the order between a_2 and a_3 matters for a_1 (*debugging nightmare*)

Dependencies, Concurrency and Non-Determinism

▶ *Imperative, sequential programming* is built upon MEM accesses, i.e., '*assignments*', and dependencies but can rely on a fixed total linear execution order.

◀ *Imperative, concurrent programming* utilizes the potential of non-determinism and partial execution orders:

- Different executions of the same system in different orders may lead to different results, so-called **Race-Conditions**.
- The causal order implied by dependencies has to be respected.
  $\Longrightarrow$ otherwise, the result is an **in-determinate process system**.

▶ *Imperative, concurrent programming* makes use of *synchronization* to restrict the permissible execution orders such that

∗ dependencies are respected as much as needed in order to guarantee determinacy, and the

∗ potential of non-determinism is preserved as much as possible.

Definition III.6: (Synchronization)

Let $Prog$ be a concurrent program that describes the process system PS and it's action execution order $\sqsubset_{PS}$. The extension of $\sqsubset_{PS}$ by means of additional **order restrictions** *for ensuring determinacy of PS is called* **synchronization**.

Statements used in the modified program $Prog'$ in order to specify synchronization are called **synchronization mechanisms**. ♦

- ▷ *Shared Memory* uses special mechanisms that guard parallel access to data or block processes until current data are available.
- ▷ *Distributed Memory* uses `snd` and `rcv` which are inherently causally ordered because a message can only be received if it has been sent before, i.e., $snd \sqsubset_{PS} rcv$.

Caution: Synchronization should be used with great care!

- ▶ actions waiting for predecessors that are not executed are **blocked**.
- ▶ cycles of order restrictions among processes may lead to **deadlocks**.

Example: Synchronizing One-Way Dependencies

1. **One-way dependency** $a_1 \xrightarrow{rw} a_2 \implies \sqsubset_{PS} := \sqsubset_{PS} \cup a_2 \sqsubset a_1$ **is respected using one-way synchronization**
   - a priori known order $\implies$ fixed **statically** in program code
     Example: P2 requires result from P1
   - **Simple Mechanism**: $signal(k)/wait(k)$ where $k \in \mathbb{N}$
     Semantics: $\forall\ k \in \mathbb{N}$ holds $signal(k) \sqsubset wait(k)$

Example: (left side without – right side using synchronization) c.f. pg. III-12

```
g=0; parbegin
      {g=F(3); a=2*g}
     //
      {x=3*g}
     parend
```

```
g=0; parbegin
      {g=F(3); signal(S); a=2*g}
     //
      {            wait(S);   x=3*g}
     parend
```

∗ Undesirable order no longer admissible: system is determinate right side

∗ Message Passing uses `snd/rcv` with 'implicit' `signal/wait`

Example: Synchronizing Mutual Dependencies

2. **Mutual Dependency** $a_1 \overset{mu,ww}{\longleftrightarrow} a_2 \implies (a_2 \sqsubset a_1)$ **or** $(a_1 \sqsubset a_2)$ **has to be respected using multi-way synchronization**

◀ **Problem:** often no real preference for a specific order
$\implies$ static decision for one-way synchronization makes no sense.

▶ **Typical Scenario:** lots of processes access common data

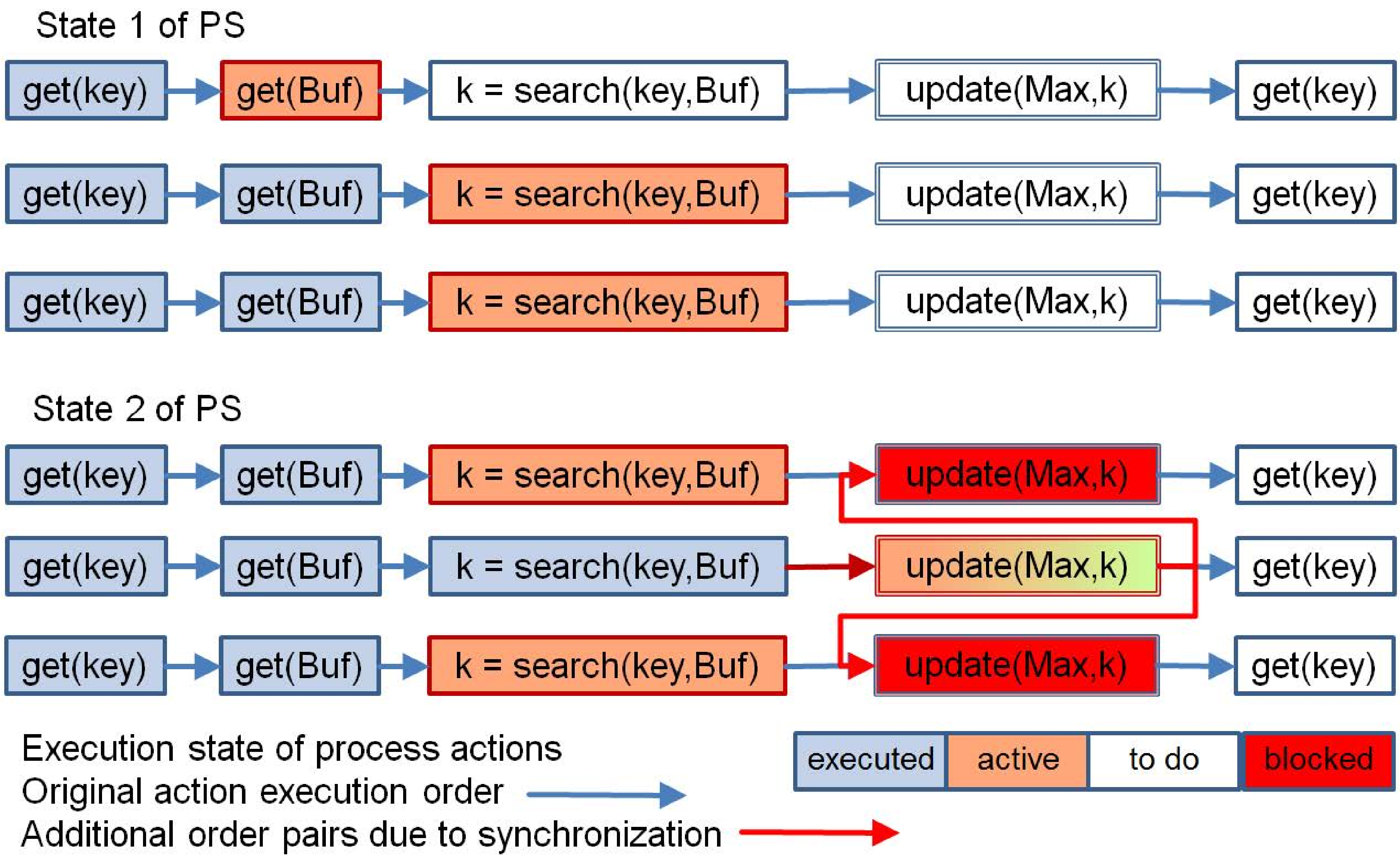
- ∗ actual execution order is insignificant, but
- ∗ specific Read and Writes should be executed without interrupt i.e., *Atomicity for a sequence of actions* is required

$\implies$ **dynamic mutual exclusion synchronization mechanism**

Example:

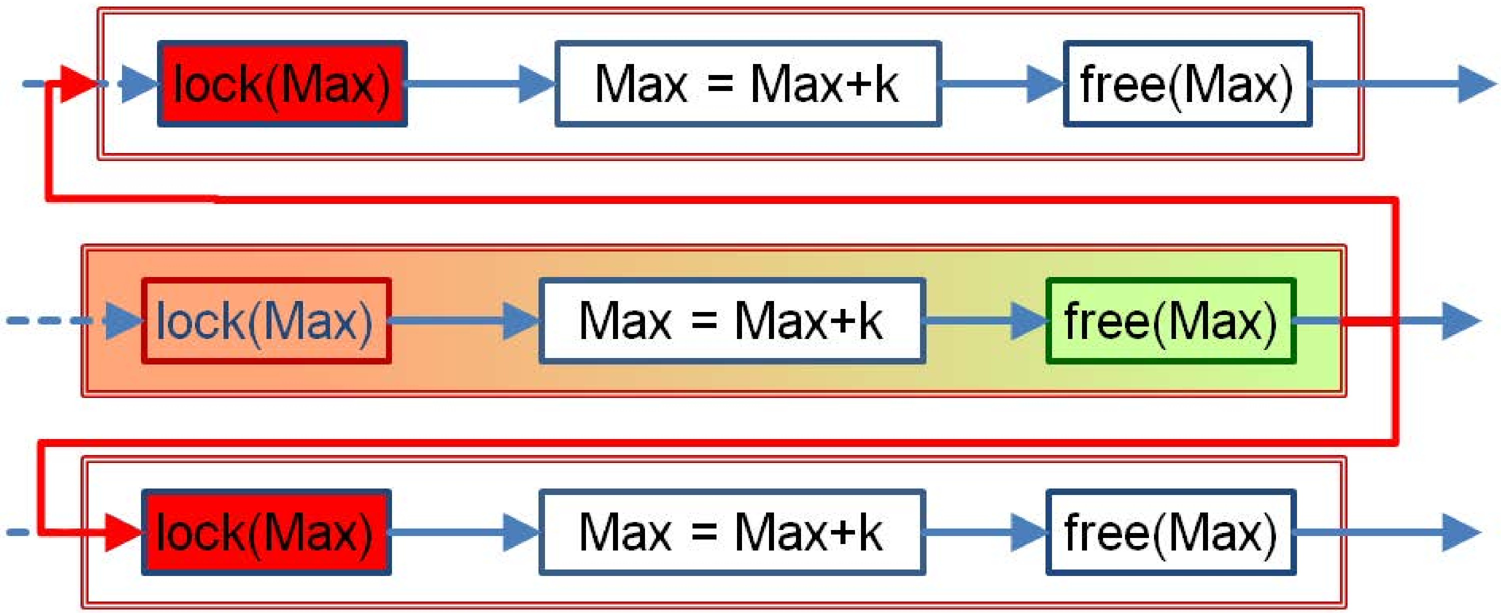
- calculation of the frequency of `key` occurrences in text buffers `Buf` c.f. pg. III-21
- single processes search *locally* for a single key in a single buffer
- *global* `updates` add local frequencies up to a global result

Example: 3 Processes with local orders plus update



- `Max` used by all processes in `update` $\Longrightarrow$ all processes are mutual dependent w.r.t. `Max`, i.e., $\overset{mu}{\longleftrightarrow}$ and $\overset{ww}{\longleftrightarrow}$

Example: Internal Mechanism for Synchronization



Programming language construct: lock/free for variables

- Detailed execution order does not matter for overall result.
- 'First' successful `lock(Max)` introduces new $\sqsubset_{PS}$ pairs between `free(max)` at the end of the active update and *all other* `lock(Max)` actions yet to be executed.
- Executing corresponding `free` supersedes new pairs from $\sqsubset_{PS}$
- Reading and writing `Max` is done without any other process accessing `Max` during this sequence of steps.

Definition III.7: (Critical Section and Mutual Exclusion)

Let $K \subseteq PS$ be a set of processes from PS that are mutual dependent and let M be the data this dependency is based upon, i.e., M is part of the intersection of data sets from all processes.

1. *A sequence of actions $cs = a_l a_{l+1} \ldots a_f$ of $P \in K$ that causes the dependency is called* **critical section** *w.r.t. M $:\Longleftrightarrow$*
   If cs is **active**, *the execution order $\sqsubset_{PS}$ of PS ensures that no other process from PS executes an action a which uses data from M, i.e., $Data(a) \cap M \neq \emptyset$.*
2. *The class $CS(M)$ of all cs from K is* **strictly mutual exclusive** *w.r.t. data M.* ♦

Remark: A finite sequence of actions $a_l a_{l+1} \ldots a_f$ is **active** in a state of PS if at least a_l has been executed and a_f has not yet finished its execution.

A weaker Notion of Critical Sections

▷ A **critical section** of **Order** k ($k > 0$) allows for a **maximum** of k active critical sections cs in parallel (at the same time).

▷ Strict mutual exclusion is the special case of a cs of order $k = 1$

Example: Reader-Writer Problems

- k Reader processes $R_1, R_2, \ldots R_k$ use data M
- m Writer processes $W_1, \ldots, W_m$ write data M
- Restrictions on parallel access:
  1. Only one Writer is allowed at a specific time ($\xleftrightarrow{ww}$)
  2. Writers and Readers are mutual exclusive ($\xrightarrow{rw}$)
  3. A maximum of $k = 3$ Readers are allowed for parallel access, e.g., caused by insufficient resources

c.f. pg. III-43

Definition III.8: (Blocking, Deadlocks, Starvation)

Let PS be a process system with processes $\{P_1, \ldots, P_n\}$.

1. *PS is* **safe** *w.r.t. critical sections* $:\Longleftrightarrow$ *Definition* III.7 *holds for all critical sections $CS(M)$ of PS.*
2. *A process $P \in PS$ is* **permanently blocked** *if the rule R does prevent P permanently from the execution of it's next action because P has to wait for actions of other processes from PS.*
3. *A subset $D \subseteq PS$ of processes is called* **Deadlock** *if all processes of D are permanently blocked due to a cyclic wait for actions from other processes from D caused by $\sqsubset_{PS}$.*
4. *A process $P \in PS$ is treated* **unfair**, *i.e., is subject to* **Starvation** *if the process P cannot proceed for an infinite amount of steps, although the next action of P is allowed by the rule R.* ♦

- *Blocking* and *Deadlock* are caused by the specification level $\sqsubset_{PS}$.
- *Starvation* is caused by the execution environment, typically by keeping a process from resource access for an infinite long 'time'.

Cooperation, Coordination and Interaction

- ▷ In order to get common work done, processes have to *cooperate*.
- ▷ Cooperation is made possible by *coordination* using *interaction*.
- ▷ *Interaction* implies interaction roles and causes *dependencies*.
- ▷ Combining dependencies with non-deterministic specifications of execution orders may lead to *race conditions* and, hence, indeterminate systems.
- ▷ In order to *ensure determinacy*, additional restrictions on execution orders and, thus, reduced potential for parallel execution may be required.
- ▷ *Synchronization* is used to implement execution order restrictions.
- ▷ *Synchronization mechanisms* are dependent from the underlying interaction model.

Cooperation: *Coordinated Interaction ruled by means of synchronization mechanisms to ensure determinate systems in order to get some common work done.*

SMS vs. DMS Synchronization Specification

Shared-Memory paradigm:

c.f. pg. III-10

- Interaction $\approx$ Read/Write of common variables, data blocks or files
- Order restrictions among processes:
  - Imperative programming languages use dedicated statements for synchronization, e.g., `signal/wait` or `locks`.
    $\Longrightarrow$ **Synchronization and interaction in separate constructs**
  - Object-oriented languages combine synchronization with attributes or methods (code blocks) of classes.
    $\Longrightarrow$ **Synchronization and interaction object-based.**

Message-Passing paradigm:

- Interaction by means of sending/receiving messages
- Order restrictions are built into Message-Passing constructs

$\Longrightarrow$ **Synchronization and Interaction in the same construct.**

III.3 Implementing Synchronization for SMS

Important Facts:

- ▶ Atomicity on single nodes is an indispensable prerequisite for synchronization among different nodes.
- ▶ On a single node, hardware-based atomic actions are required.
- ◁ Synchronizing $n > 2$ processes is much harder than synchronizing only 2 processes, esp. due to *fairness* considerations.

Different Levels and Efficiency Considerations:

- Low-level atomicity mechanisms have to be based on processes *actively polling* for access until granted.
  $\Longrightarrow$ **spin locks** cost CPU time (*active waiting*)
- Higher levels of atomicity provided by the operating system *deschedule* processes to `WAIT`-Queues when access is not granted.
  $\Longrightarrow$ **sleep locks** cost dispatch overhead (*passive waiting*)

Example: No synchronization without atomic basis

Objective: Data M should be used **mutual exclusive** by P_1 and P_2

Attempt: M '*guarded*' by `int d`: M free $\iff$ `d==0`; d initial 0

- processes check actively whether $(d == 0)$ before entering cs
- processes assign $d = 1$ when entering and $d = 0$ before leaving cs

$$\ldots \overbrace{\texttt{while (d} \neq \texttt{0) \{ \}; d=1}}^{\text{Entry Prologue for } P_1} \ldots\ cs_1\ \ldots\ \overbrace{\texttt{d=0;}}^{\text{Epilogue}}$$

$$\ldots \underbrace{\texttt{while (d} \neq \texttt{0) \{ \}; d=1}}_{\text{Entry Prologue for } P_2} \ldots\ cs_2\ \ldots\ \underbrace{\texttt{d=0;}}_{\text{Epilogue}}$$

Problem: Reading a value and writing afterwards is **not atomic**

◀ P_1 and P_2 read `d` in state 0; both assign $d = 1$; both enter cs
$\Longrightarrow$ process system is *not safe* w.r.t. M.

◁ Scheduling may allow P_1 access $n > 1$ times, even if P_2 is also trying to get access to M $\Longrightarrow$ process system is *not fair*.

Hardware Basis for Atomic Actions

Atomicity basis depends on underlying hardware model:

▷ based on **Processor**s:

- **Single cores/processors:** masking interrupts for some steps
- **Multi-cores/processors:** locking access to bus/interconnect

▶ based on **Memory** access

- ▷ locking memory regions for single process access
  requires special memory HW support, e.g., SMS supercomputers
- ▶ special hardware support for **atomic read-and-write**
  - ▷ easy to implement efficient operating system level primitives
  - ◁ not always available and not portable across architectures
- ▶ serialized MEM support for **atomic reads or writes**
  - ◁ algorithms a bit more 'tricky' to ensure atomicity
  - ▷ work on all machines due to lower demands
    Note: restricted to *basic data types* like `boolean` or `int`

Atomic read-and-write for synchronizing 2 processes

Assembler construct: `test_and_set(x,R)`

- Parameters: *local* register `R`; `boolean` variable `x` from MEM
- Semantics: **uninterrupted** execution of a 3 statement sequence

```
begin R = x; if (R == 0) then x=1 fi; end
```

Implementing mutual exclusion: initial state (`bolt == 0`)

$$\texttt{x1=1; } \ldots \overbrace{\texttt{while (x1==1) test\_and\_set(bolt,x1)}}^{\text{Entry Prologue for } P_1} \ldots\ cs_1\ \ldots \overbrace{\texttt{bolt=0;}}^{\text{Epilogue}}$$

$$\texttt{x2=1; } \ldots \underbrace{\texttt{while (x2==1) test\_and\_set(bolt,x2)}}_{\text{Entry Prologue for } P_2} \ldots\ cs_2\ \ldots \underbrace{\texttt{bolt=0;}}_{\text{Epilogue}}$$

- **Interpretation:** cs is free, i.e., `bolt` is open $\Longleftrightarrow$ (`bolt == 0`)
- Epilogue is not critical as it is only a single write operation
- **Additional assumptions:** cs_1/cs_2 last only finite time
  `bolt` is only used in prologue/epilogue

Atomic read-and-write for $n > 2$ processes

Basic Concept: *Epilogue hands cs over to 'next' waiting process*

- `boolean Wait[0:n-1]` initial $[0, \ldots, 0]$ registers waiting processes
- Prologue$_i$ $\approx$ register in `Wait`$[i]$ and set flag X_i (1)
- Condition for waiting: P_i is registered **and** flag X_i is set (2)
- End of Prologue$_i$: unregister via (`Wait[i] = 0`) (3)
- Epilogue$_i$: iff no process is waiting $\Longrightarrow$ cs is freed: `bolt=0`; (4-6)
  otherwise free next process P_j by (`Wait`$[j]$ `= 0`) (7)

▷ Prologue$_i$:

```
(1)     Wait[i]=1; Xi=1;
(2)     while (Wait[i]==1) & (Xi==1) test_and_set(bolt,Xi);
(3)     Wait[i]=0;
```

▷ Epilogue$_i$:

```
(4)     j = (i+1) mod n;                      /* next process in array */
(5)     while (j <> i) and (Wait[j] == 0) do j = (j+1) mod n; od;
(6)     if (j == i) then  bolt=0;
(7)                 else  Wait[j]=0;
```

A two-process solution using atomic read-or-write

Characteristics:

- **Weak assumption:** only a single Read **or** Write is atomic
- Processes synchronize based on active waits, i.e., spin locks

Techniques to ensure safeness under these conditions:

1. Always register first before trying to enter a critical section in order to ensure safeness.
   Counterexample: Naive Algorithm with parallel access to `bolt` c.f. pg. III-29
2. Avoid cyclic waits and deadlocks by introducing additional decision criteria that respect fairness in case of conflict.
3. Avoid starvation by means of introducing a round-robin mechanism among all waiting processes.
4. Fairness considerations are only meaningful if there is more than one waiting process.

Unsuccessful Attempts using atomic read-or-write

1. **separate global variables**; register first, then test

   `x1=0; x2=0;`

   $$P_1 \quad \ldots\ldots \overbrace{\texttt{x1=1; while (x2==1) \{ \};}}^{\text{Entry Prologue for } P_1} \ldots\ cs_1\ \ldots\ \overbrace{\texttt{x1=0;}}^{\text{Epilogue}}\ \ldots\ldots$$

   $$P_2 \quad \ldots\ldots \underbrace{\texttt{x2=1; while (x1==1) \{ \};}}_{\text{Entry Prologue for } P_2} \ldots\ cs_2\ \ldots\ \underbrace{\texttt{x2=0;}}_{\text{Epilogue}}\ \ldots\ldots$$

   $\Longrightarrow$ safe, but system may end up in **Deadlock**

2. **same variable but processes use distinct values** (round-robin)

   `turn` $\in$ { 1, 2 } ;

   $$P_1 \quad \ldots\ldots \overbrace{\texttt{while (turn==2) \{ \};}}^{\text{Entry Prologue for } P_1} \ldots\ cs_1\ \ldots \overbrace{\texttt{turn=2;}}^{\text{Epilogue}} \ldots\ldots$$

   $$P_2 \quad \ldots\ldots \underbrace{\texttt{while (turn==1) \{ \};}}_{\text{Entry Prologue for } P_2} \ldots\ cs_2\ \ldots \underbrace{\texttt{turn=1;}}_{\text{Epilogue}} \ldots\ldots$$

   $\Longrightarrow$ safe, no deadlock, fair, but prone to **permanent blocking**

Peterson-Algorithm synchronizing 2 prozesses (1981)

`x1=0;` $\{0,1\} \approx P_1$ {not registered, registered}

`x2=0;` $\{0,1\} \approx P_2$ {not registered, registered}

`turn=1;` $\{1,2\} \approx$ decides in case of conflict

Process P_1

$Prologue_1$ `x1=1; turn=1;`

```
while (x2==1) and (turn==1) { };
```

$critical\ section\ cs_1$

$Epilogue_1$ `x1=0;`

Process P_2

$Prologue_2$ `x2=1; turn=2;`

```
while (x1==1) and (turn==2) {};
```

$critical\ section\ cs_2$

$Epilogue_2$ `x2=0;`

Case-based Analysis of Peterson-Algorithm – 1

- Processes register using private variable **before** cs $\Longrightarrow$ **Safeness**
- '`turn`' ensures round-robin access $1{-}2{-}1{-}2{-}1{-}\ldots$ $\Longrightarrow$ **Fairness**
- '`turn`' influences execution in case of conflict only (and)
  $\Longrightarrow$ no **permanent blocking**

1. **Safeness:** P_i in cs_i $\Longrightarrow$
   (a) $(x_{3-i} == 0) \vee (turn \neq i)$ due to `while`-Condition **and**
   (b) $(x_i == 1)$, due to registration; reset at the end of cs_i only.

   Case Analysis for P_i in cs_i:

   I: $(x_{3-i} == 0) \Longrightarrow P_{3-i}$ not in cs_{3-i} due to (b)

   II: $(x_{3-i} == 1) \Longrightarrow (turn \neq i)$ due to (a) $\Longrightarrow (turn == 3 - i)$
   $\Longrightarrow P_{3-i}$ waits in Prologue$_{3-i}$ $\Longrightarrow P_{3-i}$ not in cs_{3-i}

Result: always at most one of the processes is in critical section

Case-based Analysis of Peterson-Algorithm – 2

2. P_i **permanently blocked?** Prologue$_i$ is only critical sequence
   $\Longrightarrow$ P_i **permanently** in Prologue$_i$
   $\Longrightarrow$ Condition $(x_{3-i} == 1) \wedge (turn == i)$ holds permanently
   $\Longrightarrow$ P_{3-i} registered & $(turn = 3 - i)$ **before** P_i
   $\Longrightarrow$ P_{3-i} is allowed to enter critical section cs_{3-i}
   As cs_{3-i} is finite $\Longrightarrow$ P_{3-i} executes Epilogue$_{3-i}$, esp., $x_{3-i} = 0$
   $\Longrightarrow$ Condition for P_i does not hold anymore (Contradiction)

3. **Starvation for P_i?** Reason: P_{3-i} **'passes'** P_i $n > 1$ times
   P_i waits in Prologue$_i$; P_{3-i} in cs_{3-i}
   As cs_{3-i} is finite $\Longrightarrow$ P_{3-i} executes $x_{3-i} = 0$ in Epilogue$_{3-i}$
   But: P_{3-i} re-enters Prologue$_{3-i}$ **before** P_i executes its test anew
   P_{3-i} executes $(x_{3-i} = 1)$ $\Longrightarrow$ P_i still has to wait
   P_{3-i} executes $(turn = 3-i)$ $\Longrightarrow$ P_{3-i} blocks itself in Prologue$_{3-i}$
   $\Longrightarrow$ P_i evaluates $(turn \neq i)$ and leaves Prologue$_i$(Contradiction)

Alternative Analysis – State-Space Exploration

Graphical Model: Algorithm is ruled by three variables

▷ States $Q \approx < x_1, x_2, turn > \in \{0,1\} \times \{0,1\} \times \{-,1,2\}$

▷ Initial state $q_0 < 0, 0, - >$ (initial value of `turn` is irrelevant)

▷ **Actions:** Prologue-Steps, cs_i; cs_{3-i}, Epilogue-Steps

▷ only finite set of states $\Longrightarrow$ finite diagram by using loops

Properties of Algorithm defined in terms of diagram states:

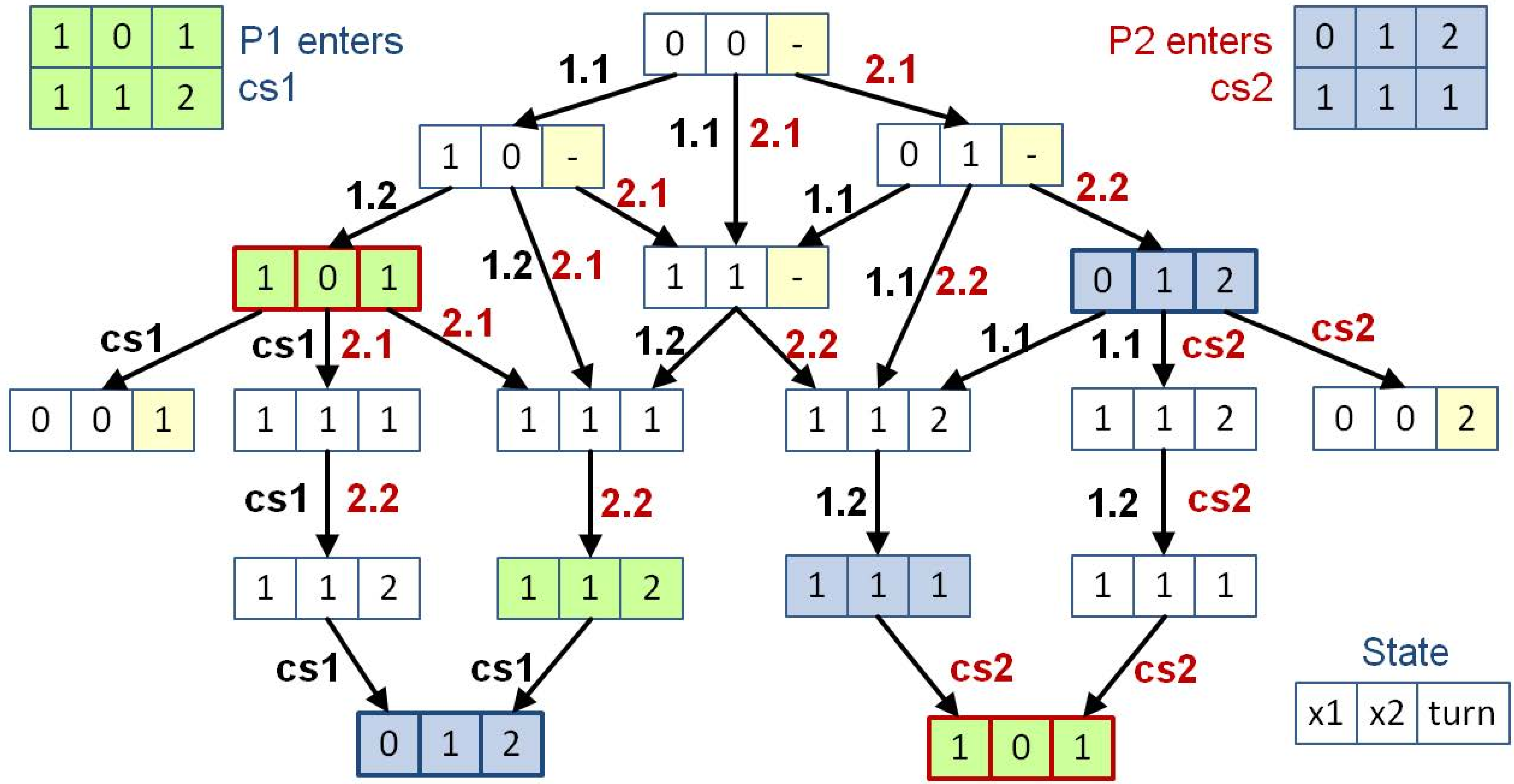
1. If there is no state s.t. cs_1 **and** cs_2 can be entered $\Longrightarrow$ **safe**
2. If there is no state s.t. **only** Prologue-Loops are executed in both processes $\Longrightarrow$ **no permanent blocking**
3. If in each cyclic, i.e., possibly infinite run cs_1 **and** cs_2 are allowed in a round-robin fashion iff both are registered $\Longrightarrow$ **no starvation**

Pros and Cons for these kinds of models:

▷ Case Analysis is prone to the same errors as the algorithm.

◁ State-Space tends to get huge for complicated systems.

State Space Analysis - Peterson Algorithm




```
x1=1; turn=1; while (x2==1) and (turn==1) { }; ... cs1 ... x1=0;
x2=1; turn=2; while (x1==1) and (turn==2) { }; ... cs2 ... x2=0;
```

Assessment of low-level Synchronization Solutions

▶ *Indispensable basis for any kind of synchronization*

- Assembler level is not the ultimate basis for a solution.
- Similar problems in electronics due to varying signal propagation times $\Longrightarrow$ *glitches* caused by *hazards*, i.e., race conditions

◀ *Missing abstraction*: Low level results in ill-structured code.

◀ *Wasted CPU-time*: Basic level has to rest on possibly long-time *active waiting* $\Longrightarrow$ costly in multi-process environments.

Solution: Multi-layered SW Architecture w.r.t. synchronization

c.f. pg. III-28

- HW-supported *spin locks* for 'short' critical sections on OS level
- Higher-level `lock`-constructs based on *sleep locks* due to OS scheduling and `WAIT`-Queues in programming languages.
- Well-structured *object-oriented* and *pattern-based* techniques embedded in languages and libraries for parallel programming.

III.4 Higher level SMS Synchronization

▷ Generalized high-level concepts

- ▷ **Semaphores** as an abstraction from active wait — Dijkstra
- ▶ **Monitors** as the basis for object-oriented synchronization

▶ Standard data structures with synchronization properties

Definition III.9: (Semaphore – User View)

A Semaphore is a shared integer variable that is only manipulated by the following three operations:

- `init(sem,n)`: *initializes a semaphore* `sem` *where* `n` $\in \mathbb{N}_0$.
- `P(sem)`: *decrements* `sem` *by* 1
- `V(sem)`: *increments* `sem` *by* 1

A semaphore `sem` *respects always the following two rules:*

1. *All operations on* `sem` *are* **mutually exclusive** *actions.*
2. *A process trying a* `P()` *operation that would render the value of* `sem` **negative** *is blocked.* ♦

Using simple Semaphores for Synchronization

1. **One-sided synchronization:** (a_2 before a_1) c.f. pg. III-19
   Example: P_1 needs result in a_1 that is provided by P_2 in a_2
   Initialization: `Semaphore s; init(s,0);`
   P_1: .... `P(s);` a_1; ...
   P_2: .... a_2; `V(s);` ....
   $\Longrightarrow$ Semaphore implements a `signal/wait` mechanism.

2. **Mutual exclusion:** A specific semaphore for **each** data structure c.f. pg. III-13
   Example: global g is updated in $n > 1$ processes by $g = g + N$
   Initialization: `Semaphore s; init(s,1);`
   P_i: ...... `P(s);` cs_i; `V(s);` ...
   $\Longrightarrow$ Semaphore implements a `lock/release` mechanism.

Alternative Variants for Semaphores

- ▷ **boolean** Semaphore uses only $\{0, 1\}$ as its value domain
- ▷ **integer** Semaphore uses value domain $\mathbb{N}_0$ (**counting** Semaphore)
- ▶ **additive** Semaphore: Integer-Sem. with comfortable operations
  - `P(sem,k)`: decrements `sem` by $\mathtt{k} \in \mathbb{N}$
  - `V(sem,k)`: increments `sem` by $\mathtt{k} \in \mathbb{N}$

  Note: `P(sem,k)` blocks if `sem` $<$ `k`

Application for an additive semaphore:

3. **Reader/Writer** system allowing 3 Readers maximum in parallel c.f. pg. III-24

   Initialization: `Semaphore rw; init(rw,3)`

   Reader: `... P(rw,1); READ; V(rw,1); ...`

   Writer: `... P(rw,3); WRITE; V(rw,3); ...`

   Writer is blocked by trying `P(rw,3)` if a Reader is currently active.

Definition III.10: (Semaphore – Implementation View)

A **Semaphore** *is a data structure built from an integer* **counter variable** `sem` *and a* **waiting room** *for processes (PCBs)* `susp`*. The semaphore implements the following* **mutual exclusive** *operations:*

1. `init(s,n) :: sem = n; susp = PCBQueue();` *where* $\mathtt{n} \in \mathbb{N}_0$
2. `P(s) :: sem = sem-1;`
```
            if (sem < 0)
              { susp.EnQueue(myself.PCB()); myself.suspend(); }
```
3. `V(s) :: sem = sem+1;`
```
            if (sem <= 0)
              { proc = susp.DeQueue(); proc.resume();}
```
♦

Note:

- |s.sem| yields the number of blocked processes for `integer` sem.
- **Fairness** for a *single* semaphore is warranted if the `PCBQueue` is a FIFO-Queue.

Monitors for Object-Based Synchronization

Brinch-Hansen 1973
Hoare 1974

- Monitors explicitly connect the shared data to be guarded from parallel access and the operations used to do so.
- Monitor concept incorporates an explicit concept of **interface**.

Definition III.11: (Monitor (naive))
A **Monitor** *is a* **data structure** D *consisting of*

1. *a set of* **shared variables** *and*
2. *a set of allowed* **access operations** $OP = \{ op_1, \ \dots \ , op_n \}$.

OP *constitutes a class of critical sections* $CS(D)$ *of order 1.* ♦

◀ Naive monitor concept is much too restrictive:
- No parallel actions on D permitted at all
- Lots of meaningful interaction techniques are not implementable

- Relaxed monitor concept with additional functionality needed.

Example: Bounded-Buffer using Event-Variables

Motivation: $op_i \in OP$ only meaningful after $op_{j \neq i}$ changes data

e.g., 'get' ('put') makes no sense if buffer is empty (full) $\Longrightarrow$ process has to release monitor in order to proceed eventually.

```
TYPE buffer = MONITOR                              (* BOUNDED BUFFER EXAMPLE *)
VAR count: INTEGER;
VAR not_full, not_empty: EVENT;
  PROCEDURE put_buf(r: DATA)
    BEGIN  IF is_full? THEN wait(not_full); ... signal(not_empty)  END;
  PROCEDURE get_buf(VAR r: DATA)
    BEGIN  IF is_empty? THEN wait(not_empty); ... signal(not_full) END;
  FUNCTION is_full?  BEGIN is_full? := (count = 100) END;
  FUNCTION is_empty? BEGIN is_empty? := (count = 0)  END;
BEGIN count := 0  END;          (* Initialization: Number of Elements *)
```

Language Construct: Monitor enhanced by *Event Variable* e

- `wait(e)` blocks calling process until `signal(e)` is issued
- `signal` has no effect if waiting queue for e is empty($\neq$ Semaphore)
- `signal(e)` re-activates one of the waiting processes: *Which one?*

Problem: Strategies for Passing Monitor Access

Process finishes monitor op**:** How to hand over monitor access?

- 'global' waiting queue for monitor may hold $n > 1$ processes
- 'local' queues for several events may also hold $n > 1$ processes

Application semantics is crucial: *No 'generic best solution'!*

1. P is granted `mutex` by op_i and blocks immediately due to `wait(e)`
   P *did not alter state* $\Longrightarrow$
   processes P' waiting internally are not re-activatable $\Longrightarrow$
   release for processes waiting in 'global' queue outside monitor.

2. P holds `mutex` via op_i and *alters state of data* in monitor
   $\Longrightarrow$ processes from all `wait`-queues may be allowed to proceed
   $\Longrightarrow$ **Competition** among local and global `wait`-Queues

Note: *Fairness* considerations are in favor of processes waiting internally w.r.t. events.

Case Study: Synchronization in Java – 1

▷ Access to basic 32-Bit data is *atomic*, but

- ◀ 64/128-Bit require $n > 1$ accesses and may be interrupted.
- ◀ Internal optimization like *caching* prevents timely propagation of shared values between different threads on the same `JVM`.

$\Longrightarrow$ Use `volatile` in order to eliminate both problems.

▶ All `objects` are *monitors* and support *locking* by use of

- `synchronized`: entire objects, methods or critical code blocks
- Set of waiting threads *(WaitSet)* that is manipulated by
  - ∗ Threads trying to acquire a 'used' `lock` end up in *WaitSet*
  - ∗ Threads explicitly `waiting` (using timeouts) to release `lock`
  - ∗ Thread interrupts and end of timeouts to quit waiting
  - ∗ Object `notify` to re-activate a *random* single thread
  - ∗ Object `notifyAll` to re-activate *all* waiting threads

Problem: *Fairness* is **not** ensured, esp. with `notify`!

Case Study: Synchronization in Java – 2

◀ Controlling threads and critical accesses explicitly using interrupts, `wait`-conditions or scheduling constructs, e.g., `sleep`, in code.

▷ `java.util.concurrent.Semaphore`: `acquire/release`

  ∗ constructed as boolean/additive as well as fair/unfair variant

  ∗ `acquire` interruptible, non-interruptible or non-blocking ('try')

▷ Atomic *Objects* for basic data structures supporting atomic `compareAndSet`, `getAndAdd` . . . more efficiently than using locks.

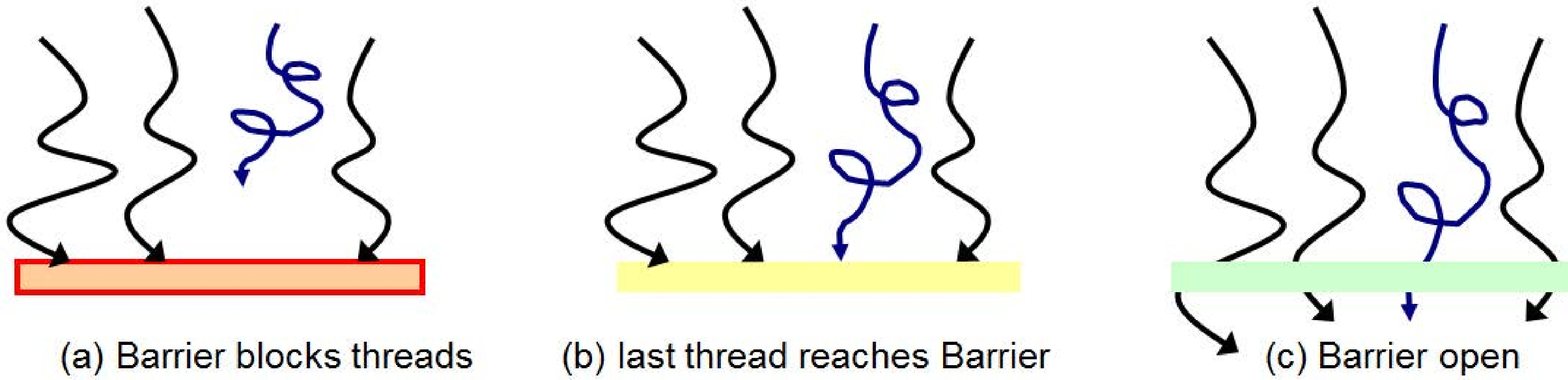
▶ Queues, linked lists, hash maps etc. as high-level data structures supporting synchronized access efficiently.

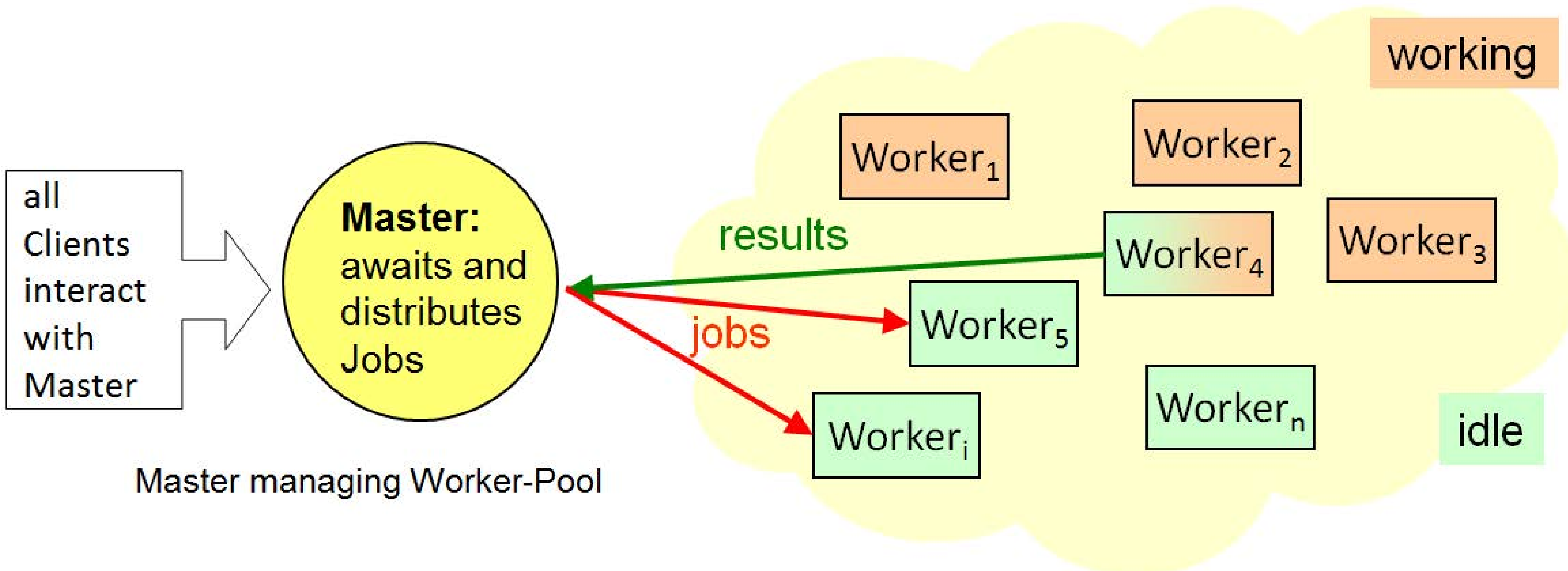
▶ High-level thread control: `CountDownLatches` or `Barriers` facilitating work distribution and master-worker architectures.

c.f. pg. III-50

▶ High-level execution control: `Executors` and `Futures` in order to implement 'asynchronous systems' using postponed or *delayed executions* based on application logic.

Example: Barrier and Master-Worker Paradigm





Master managing Worker-Pool

III.5 Message Passing Interaction

Constructs: P_1: `SND(exp,...)` $\longrightarrow$ ...

P_2: ... $\longrightarrow$ `RCV(var,...)`

Effect ($P_1 \mapsto P_2$): '$\texttt{var}_{P_2}$:= $\texttt{VAL(exp)}_{P_1}$'

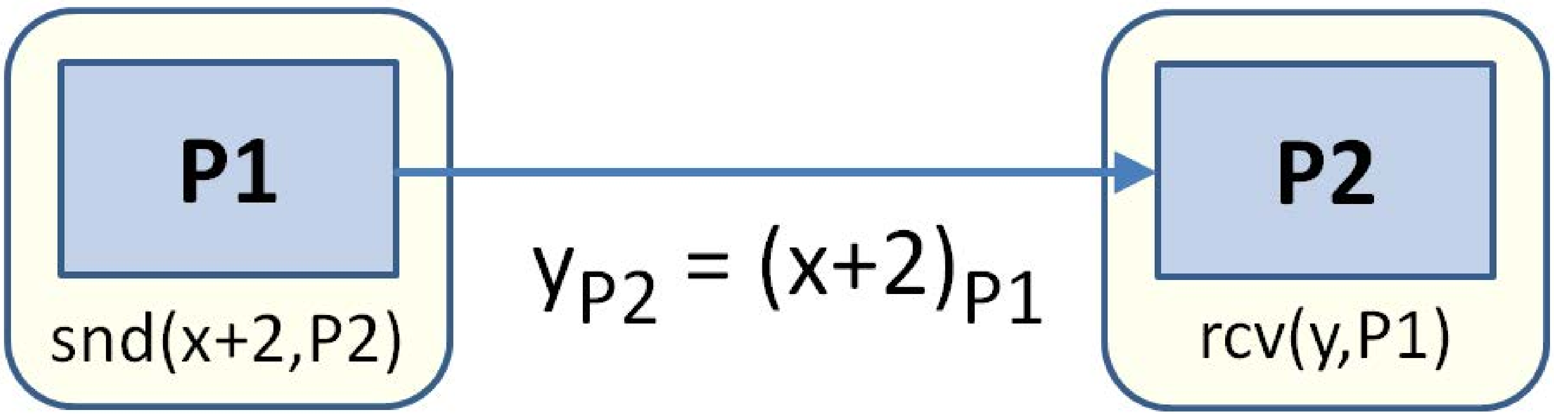
Causality: Message transport requires time, i.e., `SND` $\sqsubset_{PS}$ `RCV`

Characteristics of different Message-Passing Paradigms:

1. **Message payload specification:** c.f. II-11/ – II-13
   - Result of expression evaluation: `int`, reference to a linked DS
   - Externalization and marshalling of *message* content
2. **Destination address(es) specification**
   Note: this is not needed in a SMS paradigm
3. **Different levels of synchronization and coupling** between sending and receiving processes beyond `SND` $\sqsubset_{PS}$ `RCV`.
4. **Transient** vs. **Persistent** Communication ranging from no buffering, short-term buffering of msgs, ..., `message queueing`

Identification of Message Destination - 1

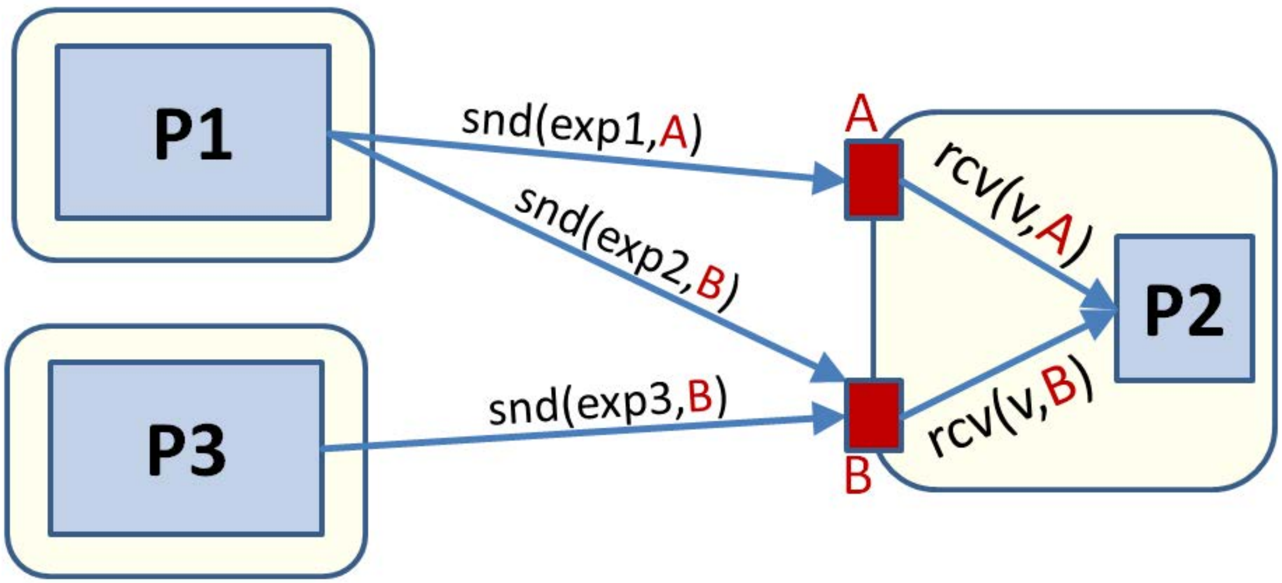
1. **Direct explicit Naming**: Usage of 'process names'
   - ◀ fixed identifier in code $\Longrightarrow$ barely usable
   - ▷ Flexible, internal logical identifier management
     - ∗ standard in parallel programming environments, e.g., `MPI`
     - ∗ process identifiers as `return` values from process start managed using logical abstractions, e.g., 'neighborhood'



◁ Modularization and Encapsulation only sufficient when using well-documented, fixed naming conventions.

Identification of Message Destination - 2

2. **Indirect Naming using Ports** yields useful abstraction
   - Sender and receiver process remain anonymous
   - A single process may use different Ports
   - Different processes may use the same port
   - typically no (or only restricted) buffering at ports

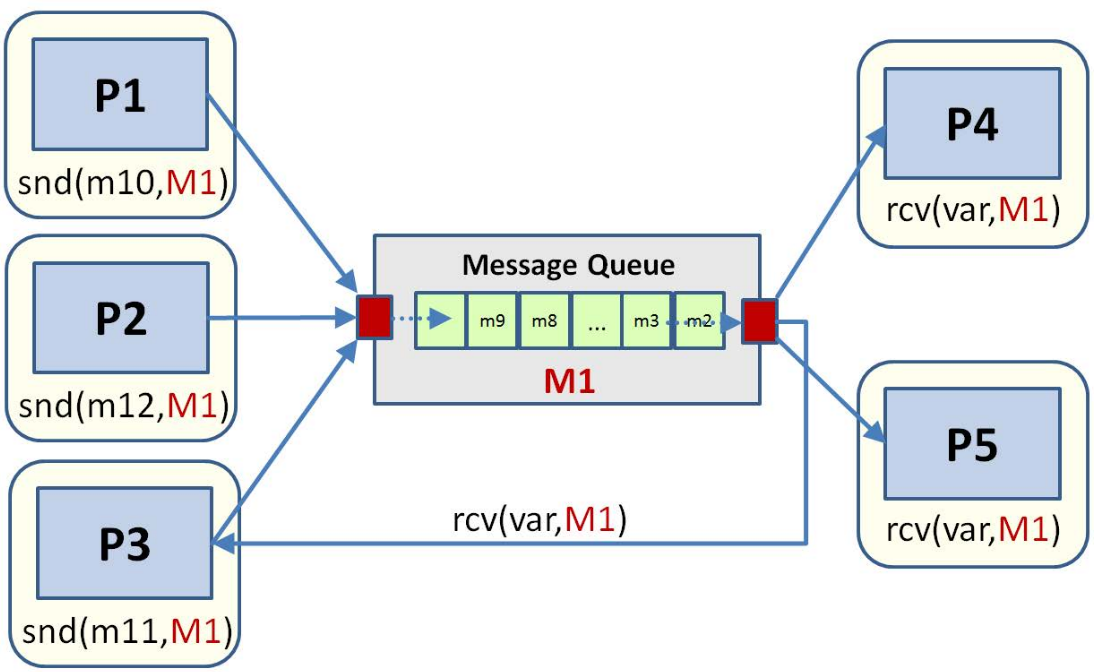


permits n–1 and 1–n interaction
Ports not part of processes but at OS or NW layers
e.g. `UDP sockets`

▷ Easy to re-use by parameterizing applications with logical port Identifiers, e.g., in program libraries.

Identification of Message Destination - 3

3. **Indirect Naming using Channels**
   - Connect logical ports by means of **Channels**
   - Definition/set-up by infrastructure specification or during process creation, e.g., `TCP sockets`
4. **Mailbox**: connections act as buffered channels, e.g., `JMS`



▷ high level of de-coupling w.r.t. Identification **and** Time

◁ copying needed

▷ m–n-communication

▷ broadcast/multicast via non-destructive message receive

Synchronization among Communicating Processes

$$P^1 : q_0^1 \xrightarrow{a_1^1} q_1^1 \xrightarrow{a_2^1} \ldots\ldots\ldots q_k^1 \xrightarrow{\mathbf{snd(P_2)}} q_{k+1}^1 \xrightarrow{a_{k+2}^1} \ldots$$

$$P^2 : q_0^2 \xrightarrow{a_1^2} \ldots q_m^2 \xrightarrow{\mathbf{rcv(P_1)}} q_{m+1}^2 \xrightarrow{a_{m+2}^2} \ldots$$

c.f. pg. III-56

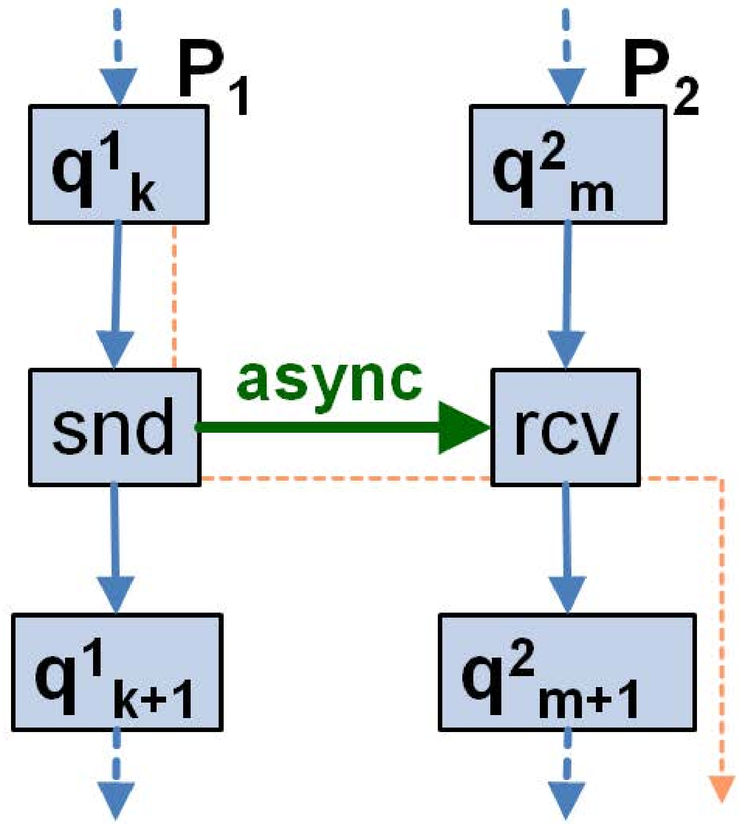
- **Asynchronous Communication:** inevitable effect `snd` $\sqsubset_{PS}$ `rcv`
  - q_{m+1}^2 in P_2 only reachable if P_1 in q_{k+1}^1
  - $\xrightarrow{a_{k+2}}$ executable in P_1 even if P_2 not in q_m^2

  $\Longrightarrow$ Only P_1 affects execution of P_2 (**one-sided effect**)

- **Synchronous Communication:** maximum effect in both procs.
  `rcv` **after** `snd` $\wedge$ `snd` **after** `rcv` $\Longrightarrow$ **coincident** *as a single action!*
  - q_{m+1}^2 in P_2 only reachable if P_1 in q_{k+1}^1
  - q_{k+1}^1 in P_1 only reachable if P_2 in q_{m+1}^2
  - $\xrightarrow{snd(P_2)}$ in P_1 ends if P_2 is in q_{m+1}^2

  $\Longrightarrow$ **Symmetrical effect** for executing both, P_1 and P_2

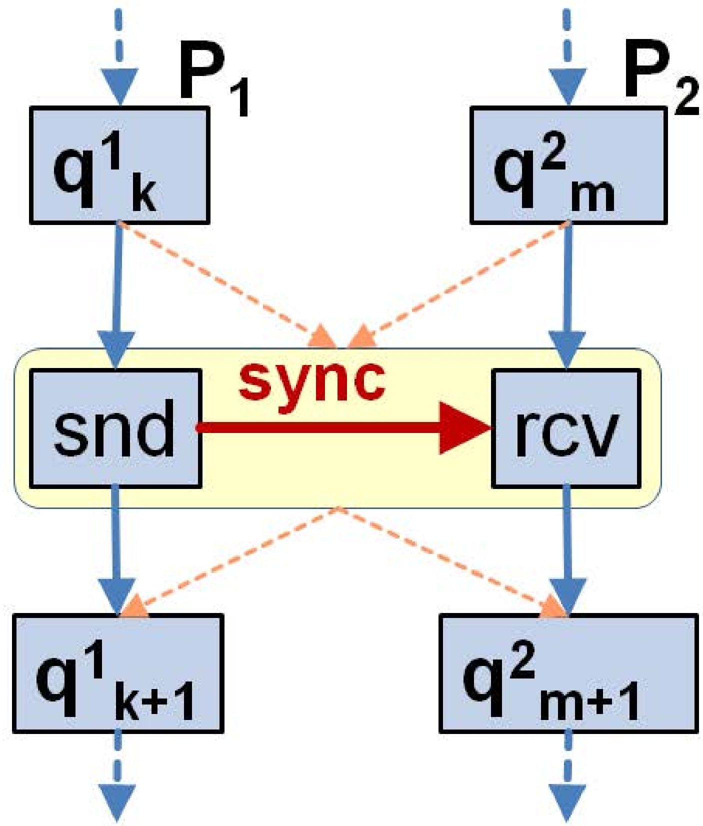
Asynchronous vs. Synchronous Communication



- ▷ easier in programming
- ▷ simulates sync (*handshake*)

  `snd(P₂); rcv(P₂)`

  $\|$ `rcv(P₁); snd(P₁)`
- ◁ but: limits to asynchrony



- ▷ easier to implement
- ◁ 'easier' to *Deadlock*, e.g.,

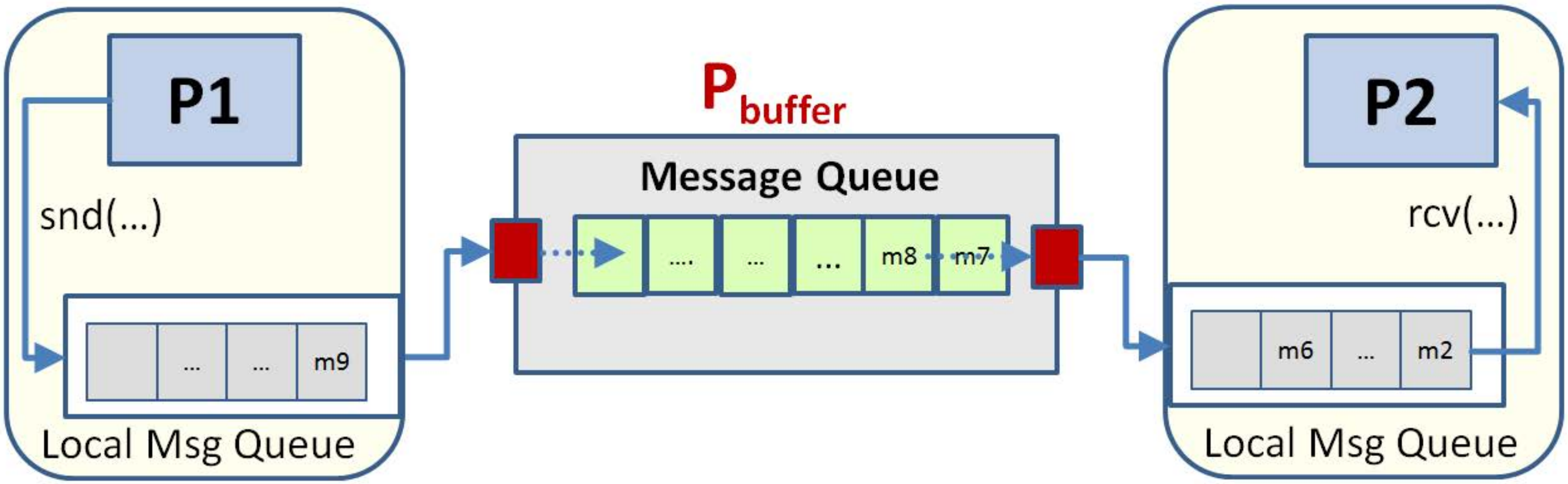
  `snd(P₂); rcv(P₂)`

  $\|$ `snd(P₁); rcv(P₁)`
- ◁ prone to programming errors

Limits in Implementing Asynchrony

Problem: P_1 sends *unbounded many* message to P_2
P_2 fails to execute $\texttt{rcv(P}_1\texttt{)}$ frequently

$\Longrightarrow$ *How to build a buffer for infinite many messages?*



Different Locations for Buffering:

- ▷ local buffer for sending process (OS-I/O-Queue)
- ▷ special buffer process P_{buffer} like, e.g., in a mailbox system
- ▷ local buffer for receiving process (OS-I/O-Queue)

Theory: Unfeasible as any buffer has only finite capacity

Approximating Asynchronous Semantics

Concept: Sending process acts like asynchronous communication is possible as long as there is sufficient buffer space; otherwise one of several possible programming models is used: *If buffer is full*

◀ **Buffer-blocking** sender semantics
- $\Longrightarrow$ block sender as in synchronous communication
- $\Longrightarrow$ $\exists$ maximum **synchronous distance** caused by buffer size
- $\Longrightarrow$ Communication becomes synchronous if buffer is full

◁ **Lost messages** semantics
- $\Longrightarrow$ sender is not blocked but keeps sending
- $\Longrightarrow$ drop oldest messages $\Longrightarrow$ Non-blocking bounded queue.

Problem: *Semantics is dependent from actual data.*

▶ **Raise an** `exception`, i.e., the programmer is handed the opportunity to wait, to drop messages (FIFO, LIFO, attributes) or to block the sender until some buffer space becomes available.

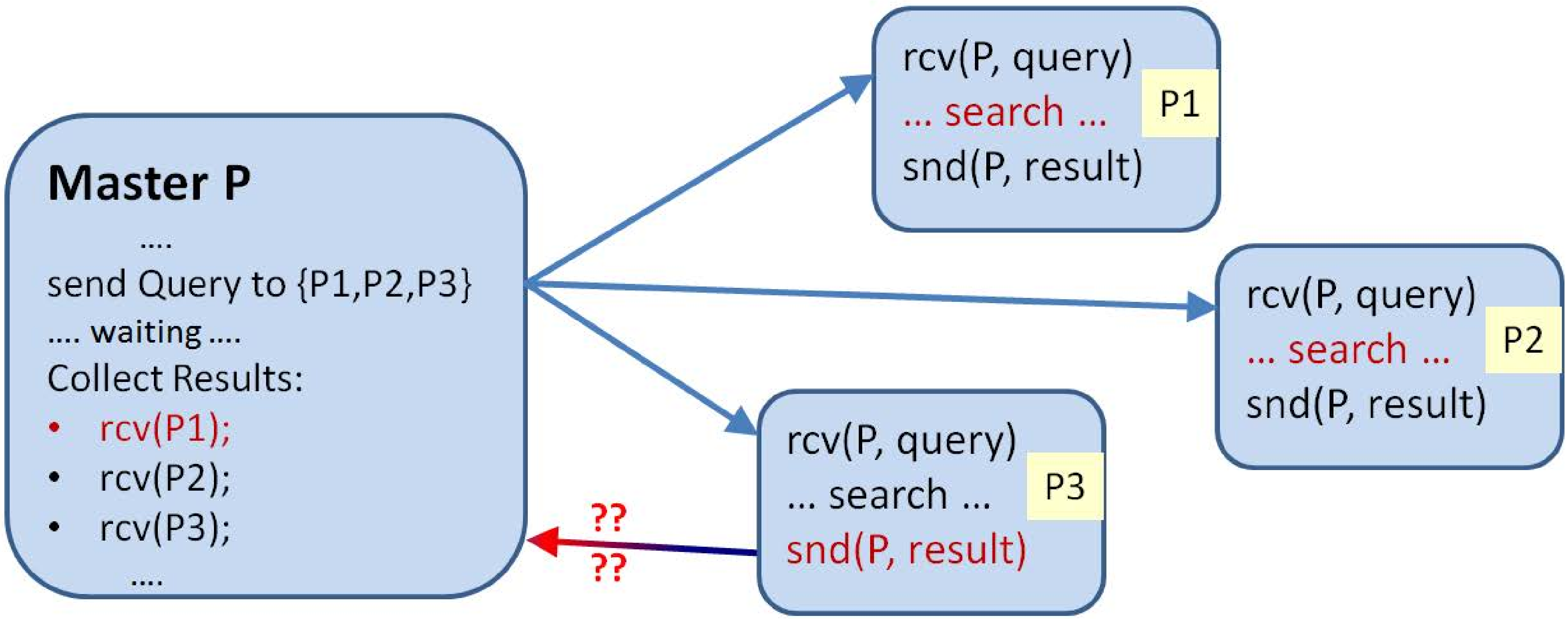
De-Coupling using Selective Receives – 1

Setting: $P \longleftrightarrow \{P_1, P_2, P_3\}$ interacts with $n > 1$ processes.

$\lhd$ Behavior of computation process is not predictable in detail

$\lhd$ no a priori known order of exchanges among processes known

$\Longrightarrow$ programming a fixed order of `rcvs` is likely to block processes



$\Longrightarrow$ P is blocked by `rcv(P`$_1$`)` although P_3 is able to send a result.

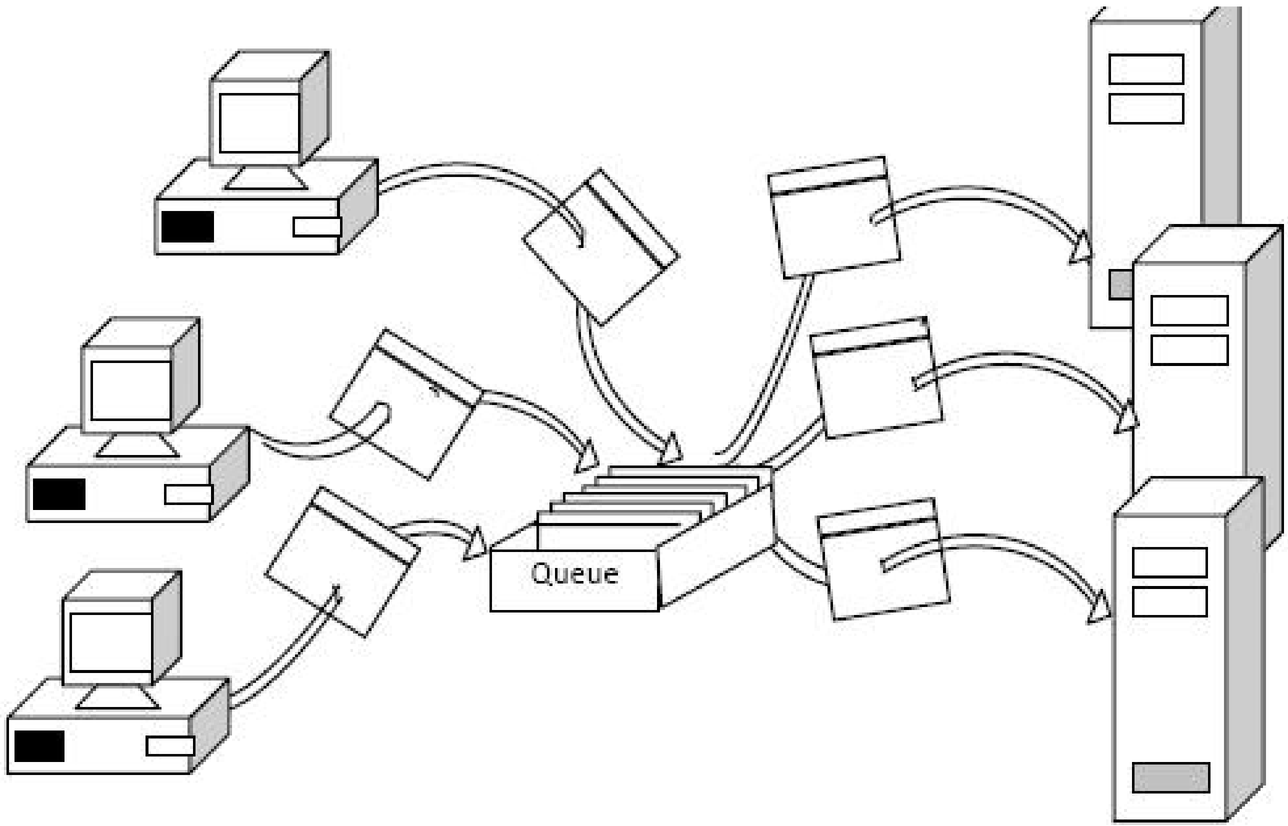
De-Coupling using Selective Receives – 2

Design Space for Programming Language Solutions:

1. **Thread-per-port**: Each sender is served by a thread of its own c.f. II-18
   $\Longrightarrow$ check for messages in a round-robin fashion
   $\Longrightarrow$ computation process is not blocked.
2. **rcv-from-any**: Use the same **port** for all sending processes
   $\Longrightarrow$ blocks only if no message is found at the port at all
   $\Longrightarrow$ blocking is no problem as there is no work to be done.
3. **Time-outs**: use receive constructs that allow for timeout settings
   Problem: program tends to react differently due to variable loads and response times.
4. **probe** as a non-blocking test whether there are messages or not.
5. **Snd/Rcv** as **guarded commands**, e.g., in `CSP` or `select` in `ADA` Dijkstra, Hoare
   $\texttt{DO guard}_1 \longrightarrow \texttt{stmt}_1 \mid \ldots \mid \texttt{guard}_n \longrightarrow \texttt{stmt}_n \mid \texttt{ELSE stmt; OD;}$
   $Distributed\ termination$ iff no more communication is possible.

Preview: Decoupling using Message Queueing

c.f. IV.



- ▶ *De-couples systems w.r.t. time and space*
- ▶ bridges heterogeneity through standardized message formats
- ▷ standard part of (almost) all middleware systems, e.g., `Java JMS`
- ◁ administration and server overhead only pays off in large systems

Overview: Java Message Passing Using Sockets

Two principal types of sockets are of interest here:

1. **TCP:** `Stream sockets`: are used in a *Client/Server* fashion
   * `ServerSockets` wait for `accepting` connection requests
   * Client `Sockets` request connections
   * successful connections work as two-way *streams*
2. **UDP:** `Datagram sockets`: similar to traditional message passing
   * `DatagramPackets`: payload plus sender/receiver addresses
   * `DatagramSockets` are bound to ports for packet `send`/`receive`

Addressing uses (IP-Address, Port-Address) based on `java.net`

`Java` *message passing using UDP sockets will be introduced in the exercises for the first assignment. Message passing using TCP sockets is – for example – one of the topics of DSG-PKS-B 'Programmierung komplexer Systeme'.*

III.6 Message Passing Synchronization

1. **One-way synchronization:** implement `signal/wait` by `snd/rcv` asynchronous `snd`; blocking `rcv` c.f. pg. III-19
2. **Mutual Exclusion** for 'Shared data' may be implemented by
   (a) **Centralized Server:** hosts data and is known to all processes
   (b) **Migrating Server:** migrates data to exactly one requesting process at a time
   (c) **Distributed Agreement:** allows for many copies of data spread among processes

   Safeness: It has to be guaranteed that at any 'point in time' only one single process has write access to 'the data'

Additional Problems:

◀ unreliable message transport $\Longrightarrow$ message loss and re-ordering

◀ no global time to 'order' concurrent requests for ensuring fairness

How to Overcome Unreliable Transport – 1

◀ Ordering messages or arbitrary events for $n > 2$ processes is done using so-called '*logical time*' maintained by algorithms like Lamport-Time or Vector-Time (see chapter VI)

▶ Ordering messages among *pairs of processes* (P_s, P_r) is easy.

1. **Use `counters` in order to detect lost messages**:
   - $\forall$ process pairs (P_s, P_r) *counters* $S_{s,r}$ and $R_{s,r}$ are introduced.
     - each sender P_s uses $S_{s,r}$ to count messages sent to P_r
     - each receiver P_r uses $R_{s,r}$ to count messages received from P_s
     - all counters in all processes are initialized by 0
     - for each `snd` in P_s, the local counter $S_{s,r}^{P_s}$ is incremented
     - for each `rcv` in P_r, the local counter $R_{s,r}^{P_r}$ is incremented
     - up-to-date counters are part of each message: *piggy backing*
   - When receiving a message, P_r checks whether $(S_{s,r}^{P_s} \neq R_{s,r}^{P_r} + 1)$ holds in order to expose a missing or untimely message from P_S.

How to Overcome Unreliable Transport – 2

2. **Storing messages and `Ackn/Resend` control messages based on 1. implement reliable messaging in case of 'finite' errors.**
   - P_s stores all messages sent to other processes including the corresponding counter as long as there is no `Ackn` from P_r.
   - P_r sends a receipt `Ackn` iff everything is ok, otherwise a `Resend` message is sent from P_r to P_s.
   - P_s re-sends missing messages as long as no receipt has been received; afterwards the messages delivered are dropped.

   $Timeouts$ may be used to overcome permanently crashed processes.

3. **Message ordering between exactly two processes**:
   - If P_r notices that ($S_{s,r}^{P_s} == R_{s,r}^{P_r} + k$ where $k > 1$), the message is *buffered* at P_r until the missing messages have been received.
   - After receiving a timely message, P_r checks its local buffer for messages that may be ready to be consumed now.

Centralized Server for Shared Data

- **Global Server:** specific $P \in PS$ for all critical sections $cs(D)$

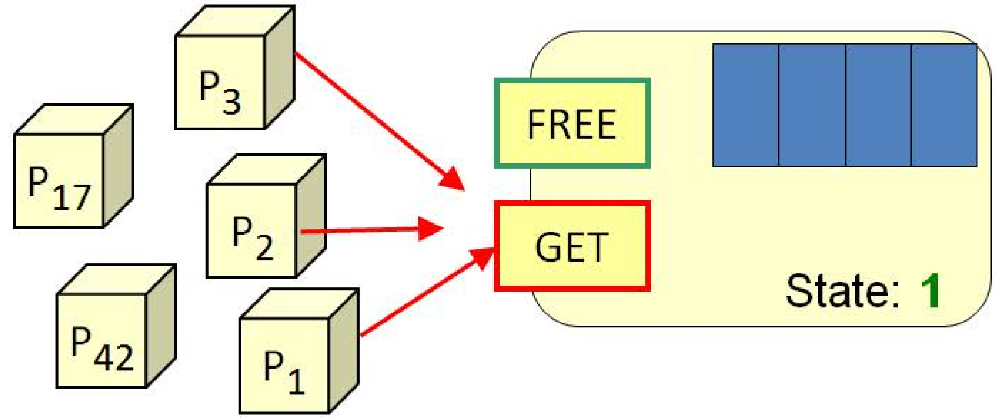
```
FORALL D DO csd.state = 1; csd.Queue.create(); OD;
WAIT FOR                              /* Server-Loop as a Guarded Command */
   rcv(P_i,csd,GET)  ==>
      IF (csd.state==1) THEN csd.state = 0; snd(P_i,csd,OK);
      ELSE csd.Queue.enqueue(P_i); snd(P_i,csd,WAIT);      FI;
   rcv(P_i,csd,FREE)  ==>
      IF (csd.Queue.isempty()) THEN csd.state = 1;
      ELSE P' = csd.Queue.frontdequeue(); snd(P',csd,OK); FI;
END WAIT;
```

- Arbitrary **Client**: process $P_i \in PS \setminus P$

```
snd(P,csd, GET); rcv(P,csd, X);        /* Prologue with blocking rcv */
IF (X==WAIT) THEN rcv(P,csd, Y) FI;          /* any more waiting? */
csd_Pi;
snd(P,csd, FREE);                                    /* Epilogue */
```

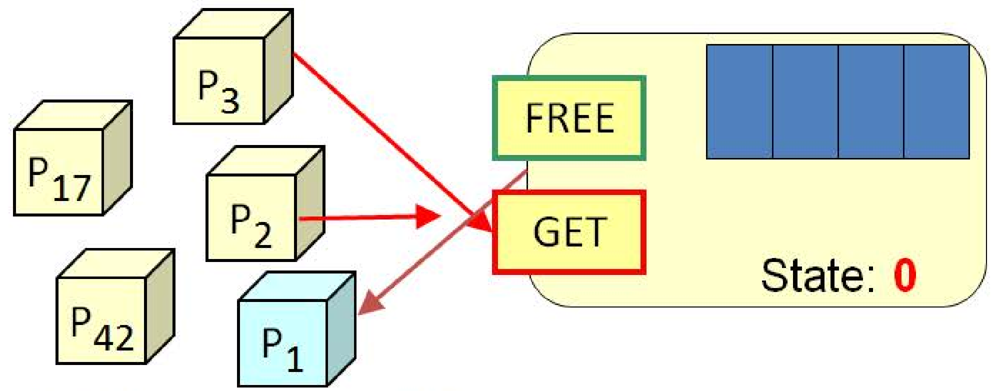
Example: Typical Run with Centralized Server - 1

(1) P1, P2 and P3 starting requests for cs;
P1 request arrives at server

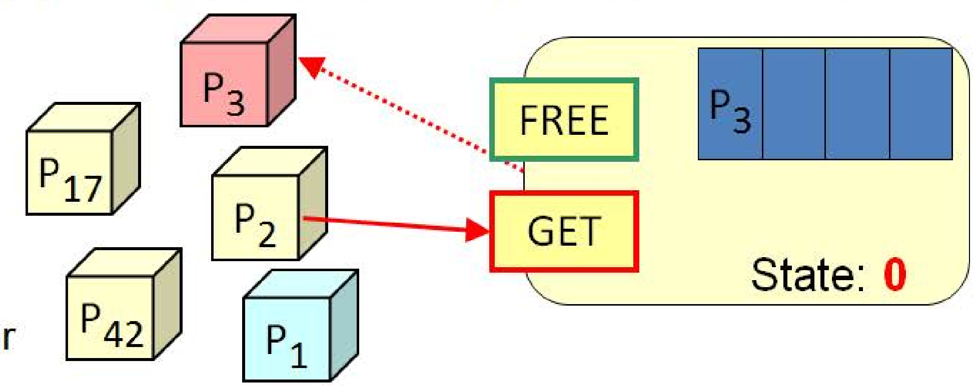


Legend:

- Client States:
  - no request or no answer
  - WAIT from server received
  - active in critical section
- Different messages:
  GET OK WAIT FREE



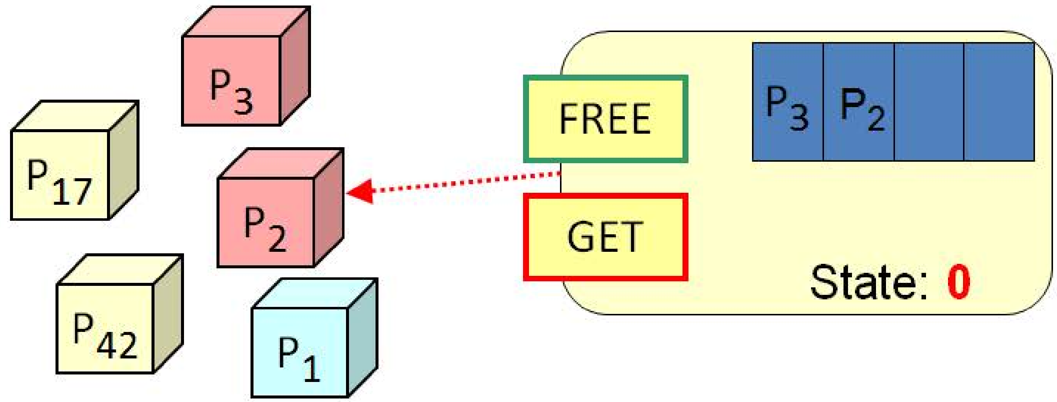
(2) P1 is granted cs; P3 request arrives at server



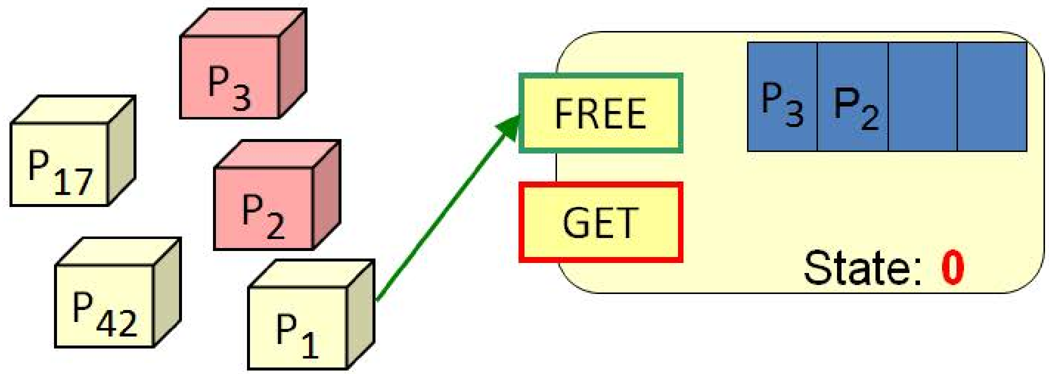
(3) P3 receives ‚WAIT' and is stored in queue
P2 request arrives at server

Example: Typical Run with Centralized Server - 2

(4) P2 receives ‚WAIT‘
two processes wait in
server Queue now

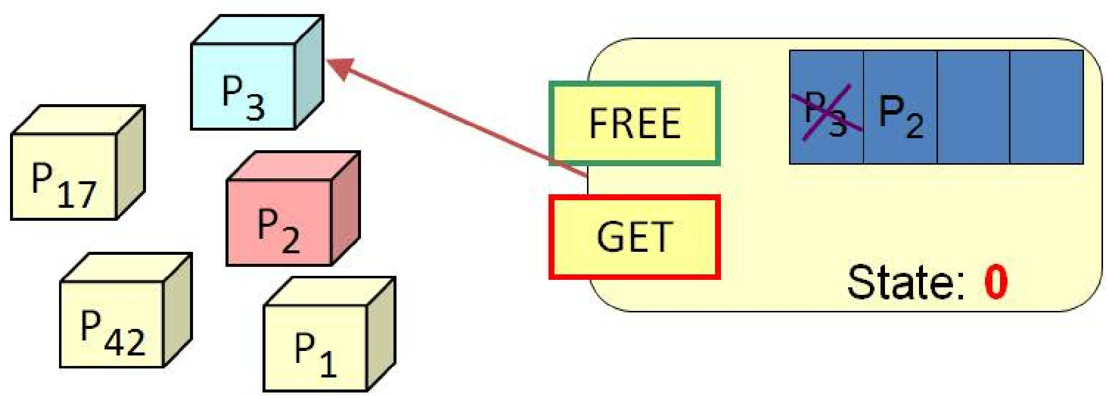


(5) P1 leaves cs and ‚FREE‘
message from P1 arrives
at server



(6) P3 is removed from Queue
and gets ‚OK‘ message to
enter cs;
P2 wait still in Queue

… etc. …



Assessment of Centralized Server Solution – 1

▶ P grants at most a single `OK` at a time $\Longrightarrow$ *System is* **safe**.

▷ Local FIFO Queue inside P is fair, but what about messaging?

- ∗ Server is only able to treat *known requests* fair.
- ∗ If messages from a process or subsystem are arbitrarily delayed or dropped, the system cannot be fair.

$\Longrightarrow$ **Starvation** *depends on communication system properties*.

◁ **Overhead:** maximum of 3 messages for a single access to `csd`.

◀ **Errors** render entire system useless:

- ◁ **Lost** `GET/WAIT/OK` **messages** block Client process indefinitely
- ◁ **Lost** `FREE` **messages** from Client block server side and `csd`
- ◁ **Crash** of server process P or process P_i currently holding `csd` block entire system.

$\Longrightarrow$ *Server process as a single point of failure combined with Client processes as multiple single points of failure.*

Assessment of Centralized Server Solution – 2

Pragmatism: Centralized Algorithm works fine iff

1. Message transport is reliable and to some extent fair.
2. Server runs on reliable hardware, e.g., virtual servers with backup.
3. Additional Measures:
   - Use distinct server processes for managing different data sets D
   - Replicate server for the same data set D on different hardware and interact among replicated servers in order to ensure consistent queues and preserve safeness.
   - Use **time-out**s if waiting for control messages lasts too long:
     - Clients should take actions if servers are down.
     - Server should take actions if `FREE`s from clients are overdue.

$\Longrightarrow$ *Centralized Server works only in a reliable environment.*

Note: *Migrating servers work similarly and exhibit almost the same properties.*

Preview: Truly distributed synchronization?

Two principal classes of really distributed algorithms:

1. **Negotiation based on local conditions in client processes**
   - P_i sends `requests` to P_js asking for permission
   - Many algorithms using different *request* and *inform* sets
     * a process has to ask all processes from its *request* set
     * a process has to answer all requests from its *inform* set
   - Sets have to be constructed such that always a **majority** of all processes is asked for permission in order to ensure safeness.
2. **Access based on exclusive ownership of a** `control token`
   - Methods vary w.r.t. how token is acquired from other processes
   - Simple case: in a *ring* of processes, token is passed around.

Problems are similar to those detected for the centralized version

◀ *single point-of-failure* $\longrightarrow$ *multiple single points-of-failure*

◀ algorithms do not work with unreliable communication

Conclusion: Basic Interaction Mechanisms

- ▶ Sufficient abstractions from hardware and network in order to implement portable distributed systems.
- ◀ Insufficient abstraction from underlying system to program on a really comfortable level (apart from msg queueing).
- ◀ Only (very) tightly coupled systems implementable by directly interacting based on *Read/Write sets* of data, no matter whether interaction is based on `read/write` or simple `snd/rcv`. c.f. pg. III-2

Higher levels of abstraction and de-coupling:

- ▶ *Message Queueing* is a first step from basics to better de-coupling via more advanced middleware technology.
- ▷ *Client-Server* is a more abstract paradigm based on remote procedures, objects and services suitable for typical settings in an internet-based environment. End of III