Parallel Programming and High-Performance Computing

Part 3: Foundations

Dr. Ralf-Peter Mundani
CeSIM / IGSSE / CiE
Technische Universität München

3 Foundations

Overview

- terms and definitions
- process interaction for MemMS
- process interaction for MesMS
- example of a parallel program

A distributed system is the one
that prevents you from working because of the failure
of a machine that you had never heard of.
—Leslie Lamport

Terms and Definitions

- sequential vs. parallel: an algorithmic analysis
  - sequential algorithms are characterised that way
    - all instructions *U* are processed in a certain sequence
    - this sequence is given due to the causal ordering of *U*, i. e. the causal dependencies from another instructions' results
  - hence, for the set *U* a partial order $\leq$ can be declared
    - $x \leq y$ for $x, y \in U$
    - $\leq$ representing a reflexive, antisymmetric, transitive relation
  - example (*a*, *b* of type integer)

```
I1:  a ← a − b
I2:  b ← b + a
I3:  a ← b − a
```

partial order: $I1 \leq I2 \leq I3$

Terms and Definitions

- sequential vs. parallel: an algorithmic analysis (cont'd)
  - often, for $(U, \leq)$ more than one sequence can be found so that all computations (on the monoprocessor) are executed correctly
  - example

```
I1:  x ← a + b
I2:  y ← c * c
I3:  z ← x - y
```

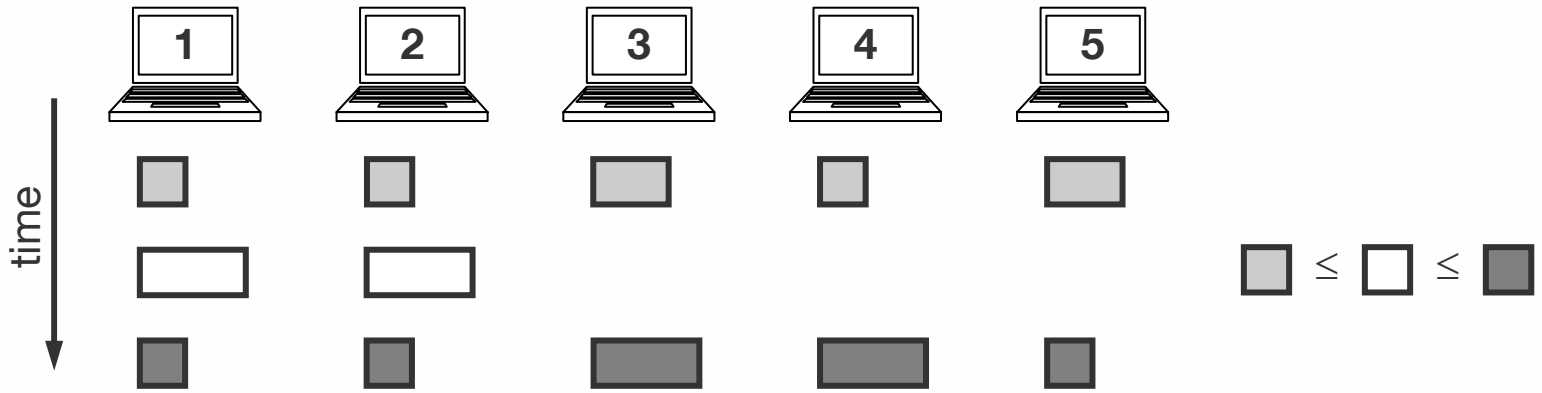
partial order: $I1, I2 \leq I3$

```
I2:  y ← c * c
I1:  x ← a + b
I3:  z ← x - y
```

  - more general

Terms and Definitions

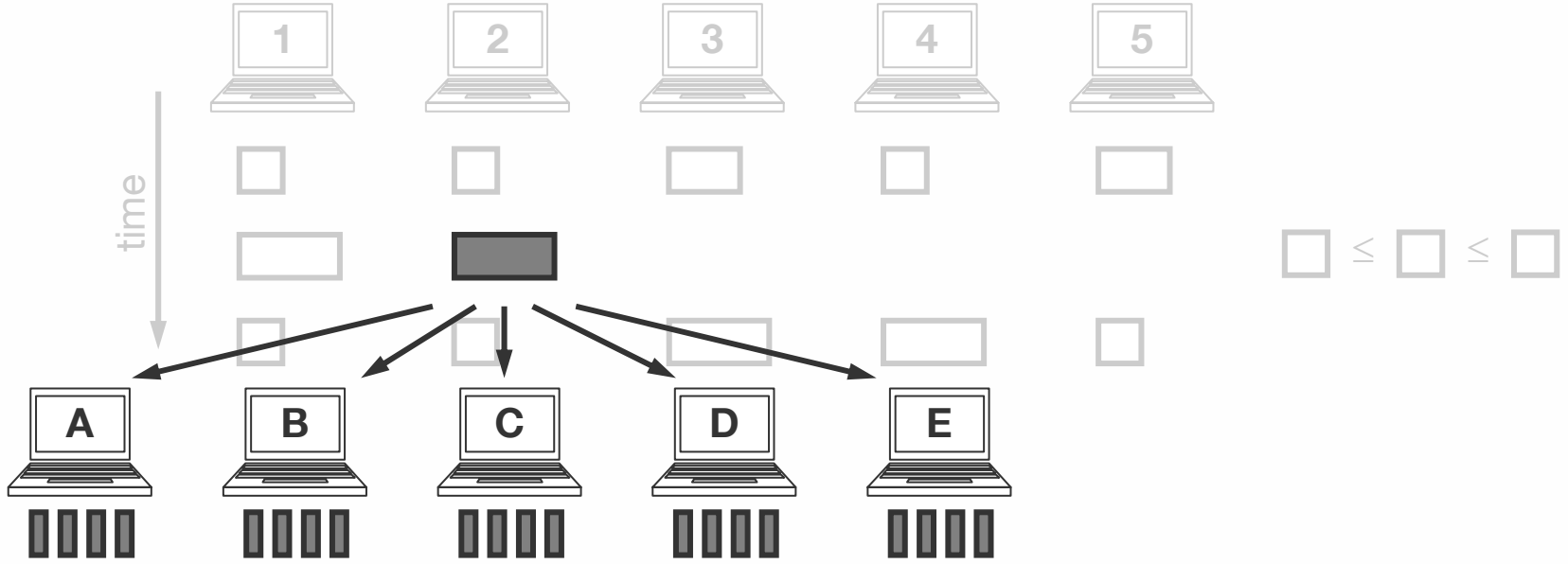
- sequential vs. parallel: an algorithmic analysis (cont'd)
  - first step towards a parallel program: concurrency
    - via $(U, \leq)$ identification of independent blocks (of instructions)
    - simple parallel processing of independent blocks possible (due to *none* or only *few* communication / synchronisation points)



  - suited for both parallel processing (multiprocessor) and distributed processing (metacomputer, grid)

Terms and Definitions

- sequential vs. parallel: an algorithmic analysis (cont'd)
  - further parallelisation of sequential blocks
    - subdivision of larger blocks for parallel processing
    - here, communication / synchronisation indispensable



Terms and Definitions

- general design questions
  - several considerations have to be taken into account for writing a parallel program (either from scratch or based on an existing sequential program)
  - standard questions comprise
    - which part of the (sequential) program can be done in parallel
    - which programming model to be used
    - what kind of structure to be used for parallelisation
    - what kind of compiler to be used
    - what about load balancing strategies
    - what kind of architecture is the target machine
    - …

Terms and Definitions

- dependence analysis
  - processes / (blocks of) instructions cannot be executed simultaneously if there exist dependencies between them
  - hence, a dependence analysis of a given algorithm is necessary
  - example

```
for_all_processes i ← 0 to N do
    a[i] ← i + 1
od
```

  - what about the following code

```
for_all_processes i ← 1 to N do
    x ← i – 2*i + i*i
    a[i] ← a[x]
od
```

  - as it is not always obvious, an algorithmic way of recognising dependencies (via the compiler, e. g.) would preferable

Terms and Definitions

- dependence analysis (cont'd)
  - BERNSTEIN (1966) established a set of conditions, sufficient for determining whether two processes can be executed in parallel
  - definitions
    - I_i (input): set of memory locations read by process P_i
    - O_i (output): set of memory locations written by process P_i
  - BERNSTEIN's conditions

$$I_1 \cap O_2 = \varnothing \qquad I_2 \cap O_1 = \varnothing \qquad O_1 \cap O_2 = \varnothing$$

  - example

P_1: $a \leftarrow x + y$ $\qquad\qquad$ P_2: $b \leftarrow x + z$

$I_1 = \{x, y\}$, $O_1 = \{a\}$, $I_2 = \{x, z\}$, $O_2 = \{b\}$ ➔ all conditions fulfilled

Terms and Definitions

- dependence analysis (cont'd)
  - further example

P_1: $a \leftarrow x + y$ $\qquad$ P_2: $b \leftarrow a + b$

$I_1 = \{x, y\}, O_1 = \{a\}, I_2 = \{a, b\}, O_2 = \{b\}$ ➔ $\boldsymbol{I_2 \cap O_1 \neq \varnothing}$

  - BERNSTEIN's conditions help to identify instruction-level parallelism or coarser parallelism (loops, e. g.)
  - hence, sometimes dependencies within loops can be solved
  - example: two loops with dependencies – which to be solved

loop *A*:

```
for i ← 2 to 100 do
    a[i] ← a[i–1] + 4
od
```

loop *B*:

```
for i ← 2 to 100 do
    a[i] ← a[i–2] + 4
od
```

Terms and Definitions

- dependence analysis (cont'd)
  - expansion of loop *B*

```
a[2] ← a[0] + 4        a[3] ← a[1] + 4
a[4] ← a[2] + 4        a[5] ← a[3] + 4
a[6] ← a[4] + 4        a[7] ← a[5] + 4
     ⋮                      ⋮
```

  - hence, *a*[3] can only be computed after *a*[1], *a*[4] after *a*[2], …
    ➔ computation can be split into two independent loops

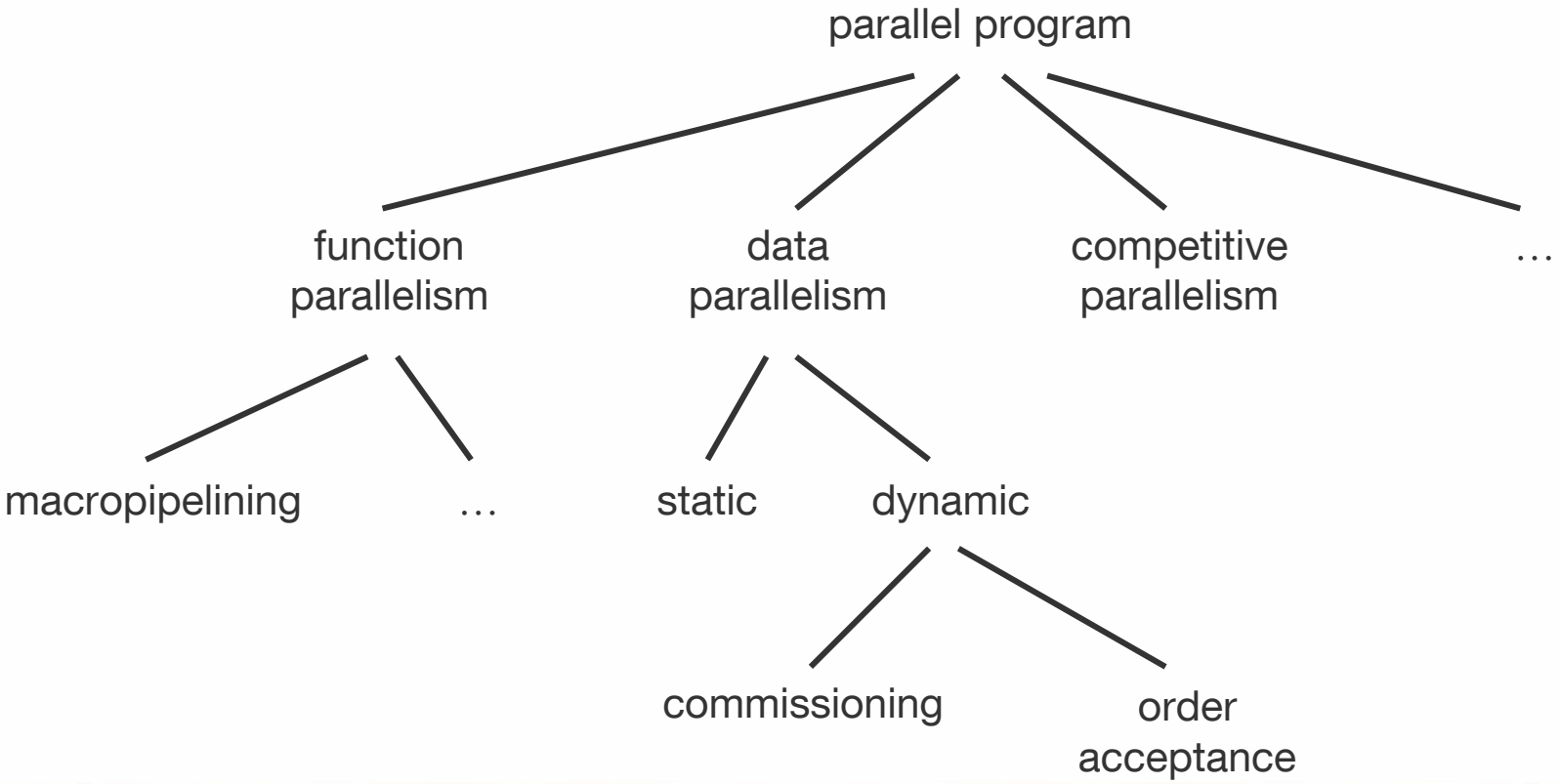
```
a[0] ← …
for i ← 1 to 50 do
    j ← 2*i
    a[j] ← a[j–2] + 4
od
```

```
a[1] ← …
for i ← 1 to 50 do
    j ← 2*i + 1
    a[j] ← a[j–2] + 4
od
```

  - many other techniques for recognising / creating parallelism exist (see also Chapter 4: Dependence Analysis)

Terms and Definitions

- structures of parallel programs
  - examples of structures



Terms and Definitions

- function parallelism
  - parallel execution (on different processors) of components such as functions, procedures, or blocks of instructions
  - drawback
    - separate program for each processor necessary
    - limited degree of parallelism ➔ limited scalability
  - macropipelining for data transfer between single components
    - overlapping parallelism similar to pipelining in processors
    - one component (producer) hands its processed data to the next one (consumer) ➔ stream of results
    - components should be of same complexity (➔ idle times)
    - data transfer can either be synchronous (all components communicate simultaneously) or asynchronous (buffered)

Terms and Definitions

- data parallelism
  - parallel execution of same instructions (functions or even programs) on different parts of the data (SIMD)
  - advantages
    - only one program for all processors necessary
    - in most cases ideal scalability
  - drawback: explicit distribution of data necessary (MesMS)
  - structuring of data parallel programs
    - *static*: compiler decides about parallel and sequential processing of concurrent parts
    - *dynamic*: decision about parallel processing at run time, i. e. dynamic structure allows for load balancing (at the expenses of higher organisation / synchronisation costs)

Terms and Definitions

- data parallelism (cont'd)
  - dynamic structuring
    - commissioning (*master-slave*)
      - one master process assigns data to slave processes
      - both master and slave program necessary
      - master becomes potential bottleneck in case of too much slaves (➔ hierarchical organisation)
    - order polling (*bag-of-tasks*)
      - processes pick the next part of available data "from a bag" as soon as they have finished their computations
      - mostly suitable for MemMS as bag has to be accessible from all processes (➔ communication overhead for MesMS)

Terms and Definitions

- competitive parallelism
  - parallel execution of different processes (based on different algorithms or strategies) all solving the same problem
  - advantages
    - as soon as first process found the solution, computations of all subsequent processes are allowed to stop
    - on average, superlinear speed-up possible
  - drawback
    - lots of different programs necessary
  - nevertheless, rare case of parallel programs
  - examples
    - sorting algorithms
    - theorem proving within computational semantics

Terms and Definitions

- parallel programming languages
  - explicit parallelism
    - parallel programming interfaces
      - extension of sequential languages (C, Fortran, e. g.) by additional parallel language constructs
      - implementation via procedure calls from respective libraries
      - example: MPI, PVM, Linda
    - parallel programming environments
      - parallel programming interface plus additional tools such as compiler, libraries, debugger, …
      - most (machine dependent) environments come along with a parallel computer
      - example: MPICH

Terms and Definitions

- parallel programming languages (cont'd)
  - implicit parallelism
    - mapping of programs (written in a sequential language) to the parallel computer via compiler directives
    - primarily for the parallelisation of loops
    - only minor modifications of source code necessary
    - level of parallelism
      - block level for parallelising compilers (➔ threads)
      - instruction / sub-instruction level for vectorising compilers
    - example: OpenMP (parallelising), Intel compiler (vectorising)

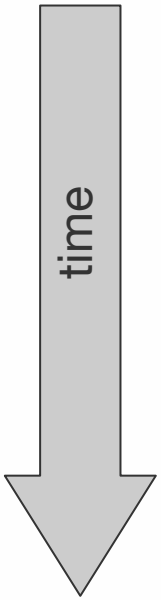
3 Foundations

Overview

- terms and definitions
- process interaction for MemMS
- process interaction for MesMS
- example of a parallel program

Process Interaction for MemMS

- problem: ATM race condition with two withdraw threads



thread 1	thread 2	balance
(withdraw $50)	(withdraw $50)	
read balance: $125		$ 125
	read balance: $125	$ 125
	set balance: $(125–50)	$ 75
set balance: $(125–50)		$ 75
give out cash: $50		$ 75
	give out cash: $50	$ 75

Process Interaction for MemMS

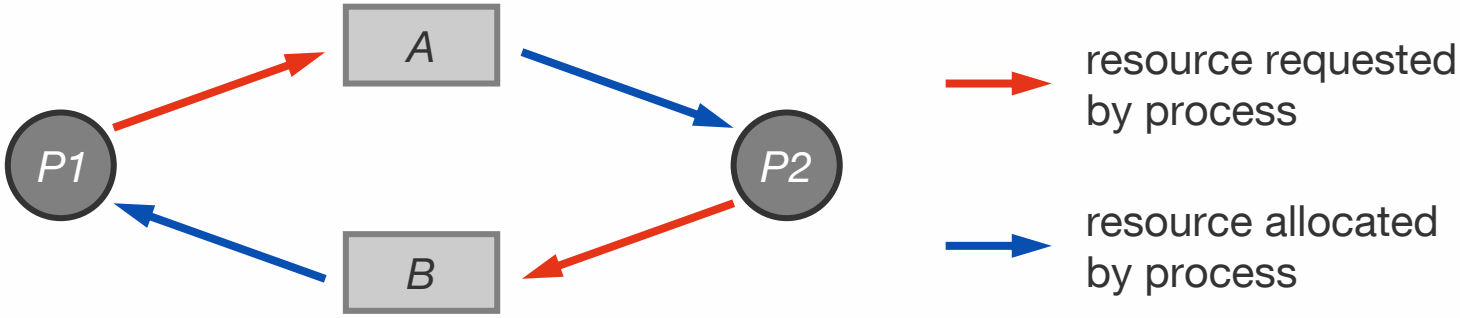
- principles
  - processes depend from each other if they have to be executed in a certain order; this can have two reasons
    - *cooperation*: processes execute parts of a common task
      - producer/consumer: one process generates data to be processed by another one
      - client/server: same as above, but second process also returns some data (result of a computation, e. g.)
      - …
    - *competition*: activities of one process hinder other processes
  - synchronisation: management of cooperation / competition of processes ➔ ordering of processes' activities
  - communication: data exchange among processes
  - MemMS: realised via shared variables with read / write access

Process Interaction for MemMS

- synchronisation
  - two types of synchronisation can be distinguished
    - *unilateral*: if activity *A2* depends on the results of activity *A1* then *A1* has to be executed before *A2* (i. e. *A2* has to wait until *A1* finishes); synchronisation does not affect *A1*
    - *multilateral*: order of execution of *A1* and *A2* does not matter, but *A1* and *A2* are not allowed to be executed in parallel (due to write / write or read / write conflicts, e. g.)
  - activities affected by multilateral synchronisation are *mutual exclusive*, i. e. they cannot be executed in parallel and act to each other atomically (no activity can interrupt another one)
  - instructions requiring mutual exclusion are called *critical sections*
  - synchronisation might lead to deadlocks (mutual blocking) or lockout ("*starvation*") of processes, i. e. indefinable long delay

Process Interaction for MemMS

- synchronisation (cont'd)
  - necessary and sufficient constraints for deadlocks
    - resources are only exclusively useable
    - resources cannot be withdrawn from a process
    - processes do not release assigned resources while waiting for the allocation of other resources
    - there exists a cyclic chain of processes that use at least one resource needed by the next processes within the chain



Process Interaction for MemMS

- synchronisation (cont'd)
  - possibilities to handle deadlocks
    - deadlock detection
      - techniques to detect deadlocks (identification of cycles in waiting graphs, e. g.) and measures to eliminate them (rollback, e. g.)
    - deadlock avoidance
      - by rules: paying attention that at least one of the four constraints for deadlocks is not fulfilled
      - by requirements analysis: analysing future resource allocations of processes and forbidding states that could lead to deadlocks (HABERMANN's / banker's algorithm well known from OS, e. g.)

Process Interaction for MemMS

- methods of synchronisation
  - lock variable / mutex
  - semaphore
  - monitor
  - barrier

Process Interaction for MemMS

- lock variable / mutex
  - used to control the access to critical sections
  - when entering a critical section a process
    - has to wait until the respective lock is open
    - enters and closes the lock, thus no other process can follow
    - opens the lock and leaves when finished
    - lock / unlock have to be executed from the same process
  - lock variables are abstract data types consisting of
    - a boolean variable of type mutex
    - at least two functions lock and unlock
    - further functions (Pthreads): init, destroy, trylock, …
  - function lock consists of two operations “test” and “set” which together form a non interruptible (i. e. atomic) activity

Process Interaction for MemMS

- semaphore
  - abstract data type consisting of
    - nonnegative variable of type integer (semaphore counter)
    - two atomic operations *P (“passeeren”)* and *V (“vrijgeven”)*
  - after initialisation of semaphore S the counter can only be manipulated with the operations $P(S)$ and $V(S)$
    - $P(S)$: if $S > 0$ then $S \leftarrow S - 1$
      else the processes executing $P(S)$ will be suspended
    - $V(S)$: $S \leftarrow S + 1$
  - after a *V*-operation any suspended process is reactivated (busy waiting); alternatives: always next process in queue
  - *binary semaphore*: has only values “0” and “1” (similar to lock variable, but *P* and *V* can be executed by different processes)
  - *general semaphore*: has any nonnegative number

Process Interaction for MemMS

- semaphore (cont'd)
  - initial value of semaphore counter defines the maximum amount of processes that can enter a critical section simultaneously
  - critical section enclosed by operations *P* and *V*

```
# mutual exclusion
(binary) semaphore s; s ← 1
execute p1() and p2() in parallel

begin procedure p1()
    while (true) do
        P(s)
        enter_crit_section()
        V(s)
    od
end
```

```
begin procedure p2()
    while (true) do
        P(s)
        enter_crit_section()
        V(s)
    od
end
```

Process Interaction for MemMS

- semaphore (cont'd)
  - consumer/producer-problem: semaphore indicates difference between produced and consumed elements
  - assumption: unlimited buffer, atomic operations store and remove

```
# consumer/producer
(general) semaphore s; s ← 0
execute producer() and consumer() in parallel

begin procedure producer()
    while (true) do
        produce X
        store X
        V(s)
    od
end
```

```
begin procedure consumer()
    while (true) do
        P(s)
        remove X
        consume X
    od
end
```

Process Interaction for MemMS

- monitor
  - semaphores solve synchronisation on a very low level ➔ already one wrong semaphore operation might cause the breakdown of the entire system
  - better: synchronisation on a higher level with monitors
    - abstract data type with implicit synchronisation mechanism, i. e. implementation details (such as access to shared data or mutual exclusion) are hidden from the user
    - all access operations are mutual exclusive, thus all resources (controlled by the monitor) are only exclusively useable
  - monitors consist of
    - several monitor variables and monitor procedures
    - a monitor body (instructions executed after program start for initialisation of the monitor variables)

Process Interaction for MemMS

- monitor (cont'd)
  - only access to monitor-bound variables via monitor procedures, direct access from outside the monitor is not possible
  - only one process can enter a monitor at each point in time, all others are suspended and have to wait outside the monitor
  - synchronisation via condition variables (based on mutex)
    - *wait(c)*: calling process is blocked and appended to an internal queue of processes also blocked due to condition *c*
    - *signal(c)*: if queue for condition *c* is not empty, the process at the queue's head is reactivated (and also preferred to processes waiting outside for entering the monitor)
  - condition variables are monitor-bound and only accessible via operations wait and signal (➔ no manipulation from outside)

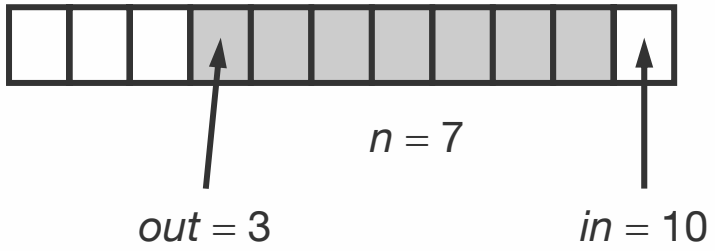
Process Interaction for MemMS

- monitor (cont'd)
  - consumer/producer-problem with limited buffer (1)

```
# consumer/producer
const size ← ...

monitor limitedbuffer
    buffer[size] of integer
    in, out: integer
    n: integer
    notempty, notfull: condition

begin procedure store(X)
    if n = size then wait(notfull) fi
    buffer[in] ← X; in ← in + 1
    if in = size then in ← 0 fi
    n ← n + 1
    signal(notempty)
end
```



Process Interaction for MemMS

- monitor (cont'd)
  - consumer/producer-problem with limited buffer (2)

```
begin procedure remove(X)
    if n = 0 then wait(notempty) fi
    X ← buffer[out]; out ← out + 1
    if out = size then out ← 0 fi
    n ← n – 1
    signal(notfull)
end

monitor body: in ← 0; out ← 0; n ← 0

begin procedure producer()
    while (true) do
        produce X
        store(X)
    od
end

begin procedure consumer()
    while (true) do
        remove(X)
        consume X
    od
end
```

Process Interaction for MemMS

- monitor (cont'd)
  - some remarks
    - compared to semaphores, after once correctly programmed monitors cannot be disturbed by adding further processes
    - semaphores can be implemented via monitors and vice versa
    - nowadays, multithreaded OS use lock variables and condition variables instead of monitors
    - nevertheless, monitors are used within Java for synchronisation of threads

Process Interaction for MemMS

- barrier
  - synchronisation point for several processes, i. e. each process has to wait until the last one also arrived
  - initialisation of counter C before usage with the amount of processes that should wait (init-barrier operation)
  - each process executes a wait-barrier operation
    - counter C is decremented by one
    - process is suspended if $C > 0$, otherwise all processes are reactivated and the counter C is set back to the initial value
  - useful for setting all processes (after independent processing steps) into the same state and for debugging purposes

Process Interaction for MemMS

- simple case study

“Program testing can be used to show the presence of bugs, but never to show their absence.”

E.W. Dijkstra

Process Interaction for MemMS

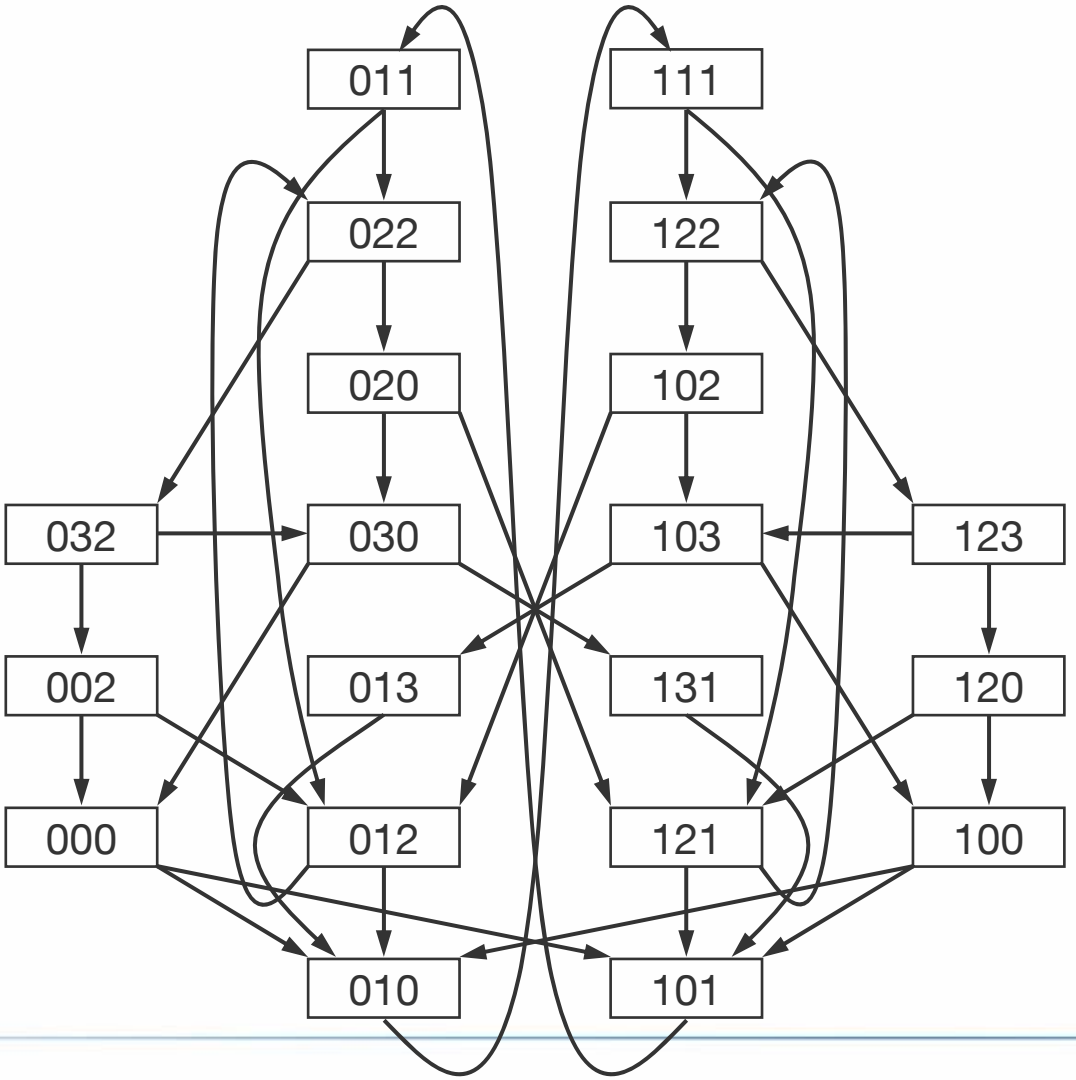
- simple case study (cont'd)
  - test case: reader-writer-problem
  - to be examined
    - *deadlock*: program will always continue
    - *mutual exclusion*: resource is never used by both processes at any time
    - *liveness*: resource will be used by any process
  - status variables
    - x, pc_0, pc_1
    - hence, 32 states possible (10 not reachable)

```
     boolean x ← 0

     proc rw0 {
        while (true) {
0:         x ← 0
1:         sync ( )
2:         if (x = 0)
3:            use_resource
        }
     }

     proc rw1 {
        while (true) {
0:         x ← 1
1:         sync ( )
2:         if (x = 1)
3:            use_resource
        }
     }
```

Process Interaction for MemMS

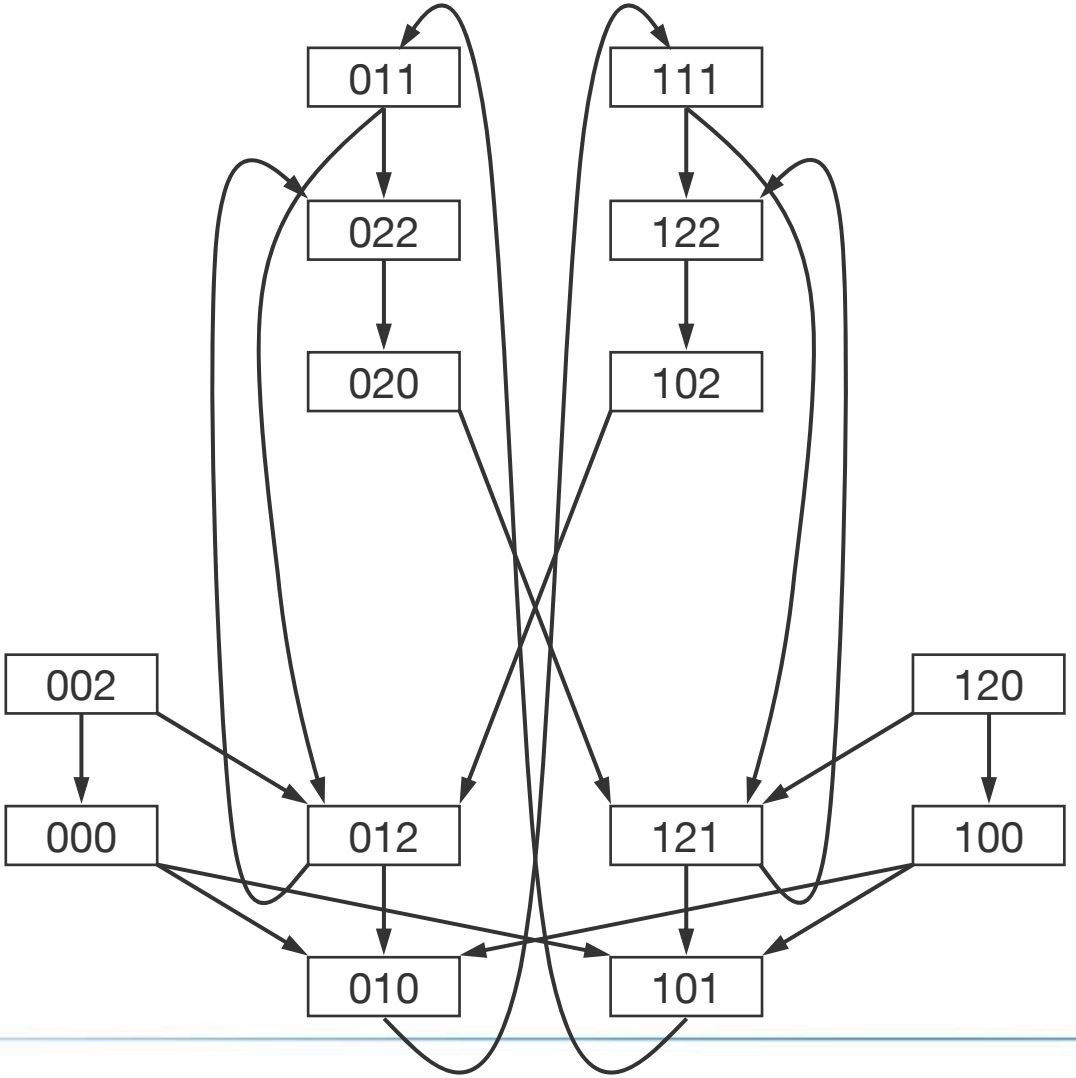



```
        boolean x ← 0

        proc rw0 {
            while (true) {
0:              x ← 0
1:              sync ( )
2:              if (x = 0)
3:                  use_resource
            }
        }

        proc rw1 {
            while (true) {
0:              x ← 1
1:              sync ( )
2:              if (x = 1)
3:                  use_resource
            }
        }
```

Process Interaction for MemMS

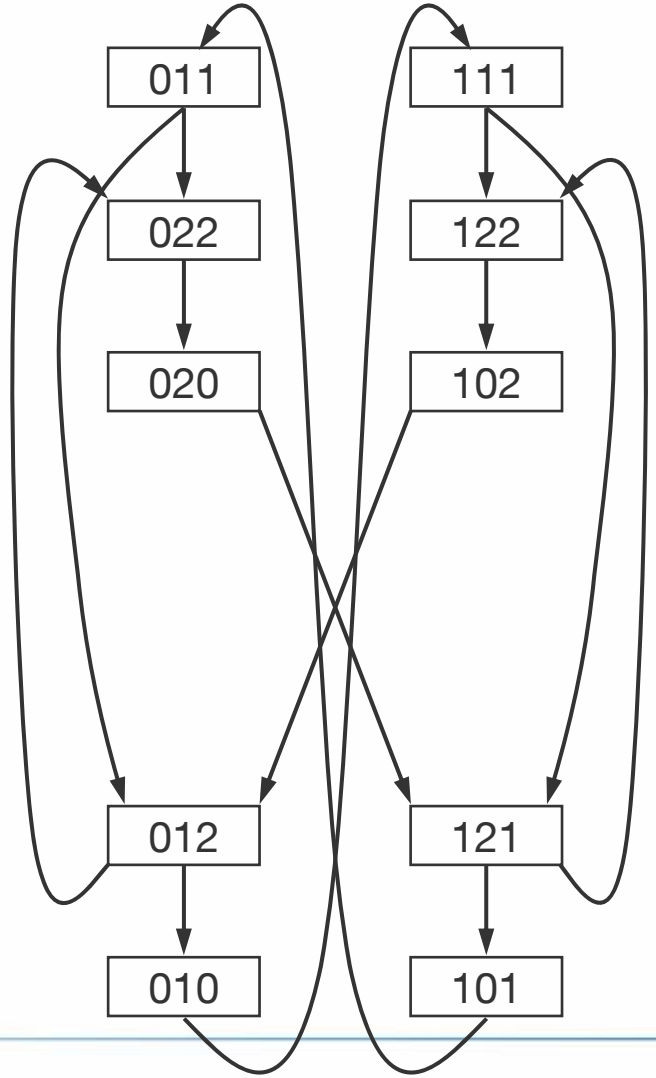



```
        boolean x ← 0

        proc rw0 {
            while (true) {
0:              x ← 0
1:              sync ( )
2:              if (x = 0)
3:                  use_resource
            }
        }

        proc rw1 {
            while (true) {
0:              x ← 1
1:              sync ( )
2:              if (x = 1)
3:                  use_resource
            }
        }
```

Process Interaction for MemMS

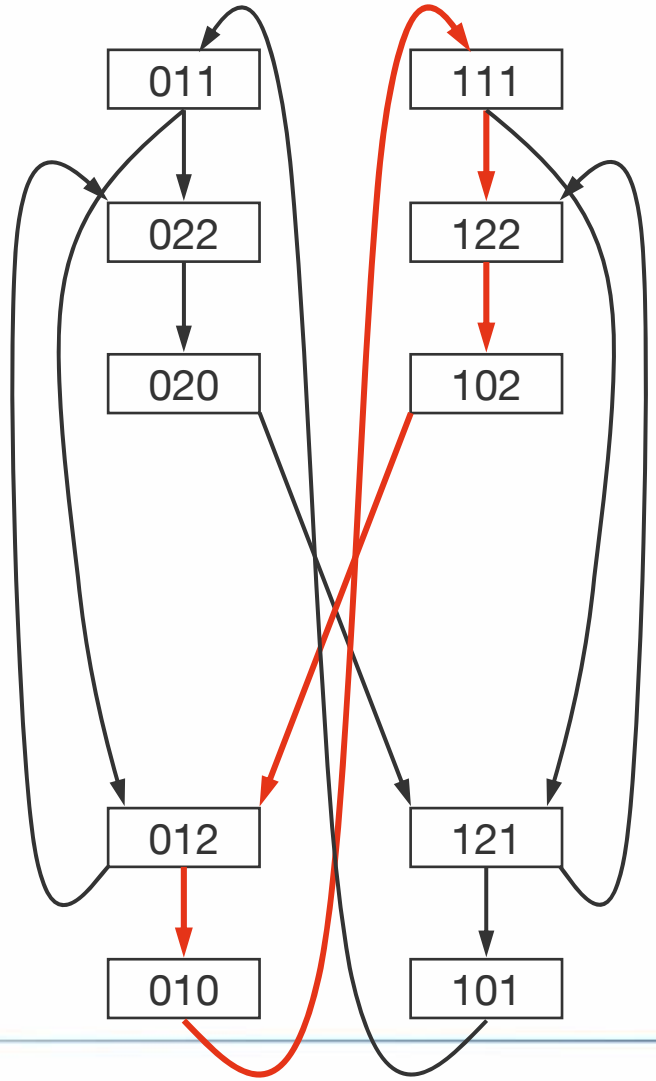



```
        boolean x ← 0

        proc rw0 {
            while (true) {
0:              x ← 0
1:              sync ( )
2:              if (x = 0)
3:                  use_resource
            }
        }

        proc rw1 {
            while (true) {
0:              x ← 1
1:              sync ( )
2:              if (x = 1)
3:                  use_resource
            }
        }
```

Process Interaction for MemMS

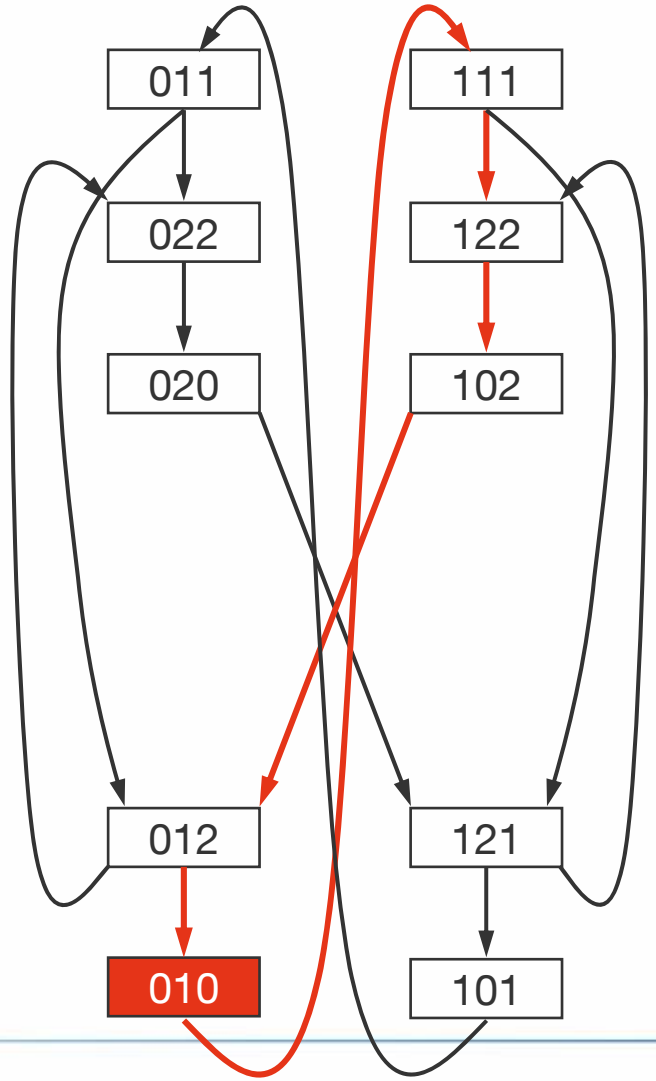



```
boolean x ← 0

proc rw0 {
    while (true) {
0:      x ← 0
1:      sync ( )
2:      if (x = 0)
3:          use_resource
    }
}

proc rw1 {
    while (true) {
0:      x ← 1
1:      sync ( )
2:      if (x = 1)
3:          use_resource
    }
}
```

Process Interaction for MemMS

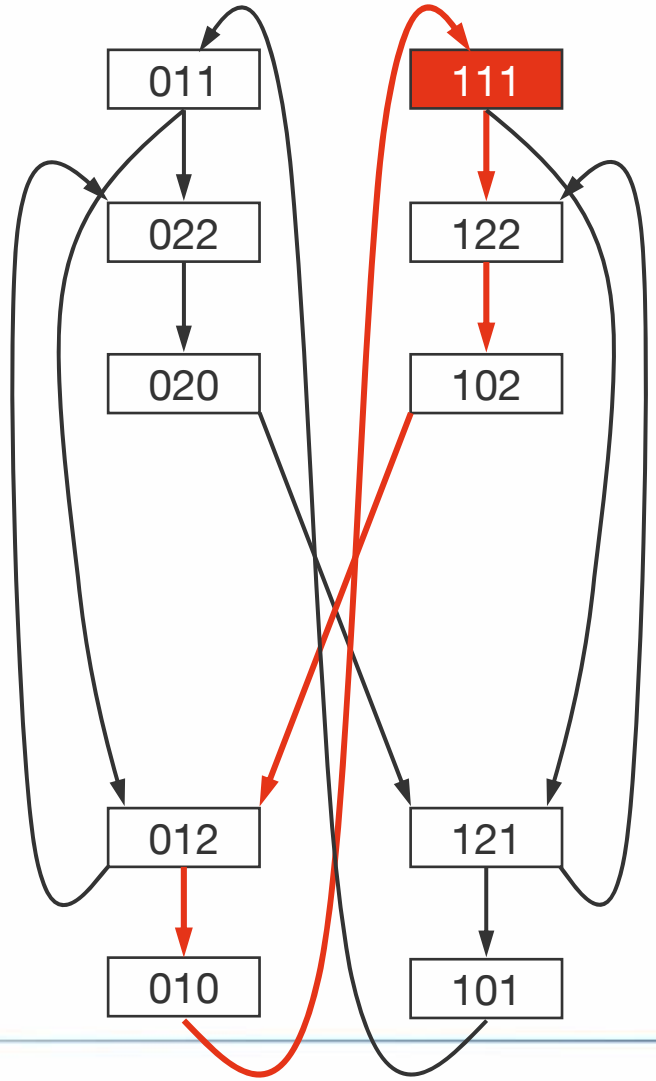



```
boolean x ← 0

proc rw0 {
    while (true) {
0:      x ← 0
1:      sync ( )
2:      if (x = 0)
3:          use_resource
    }
}

proc rw1 {
    while (true) {
0:      x ← 1
1:      sync ( )
2:      if (x = 1)
3:          use_resource
    }
}
```

Process Interaction for MemMS

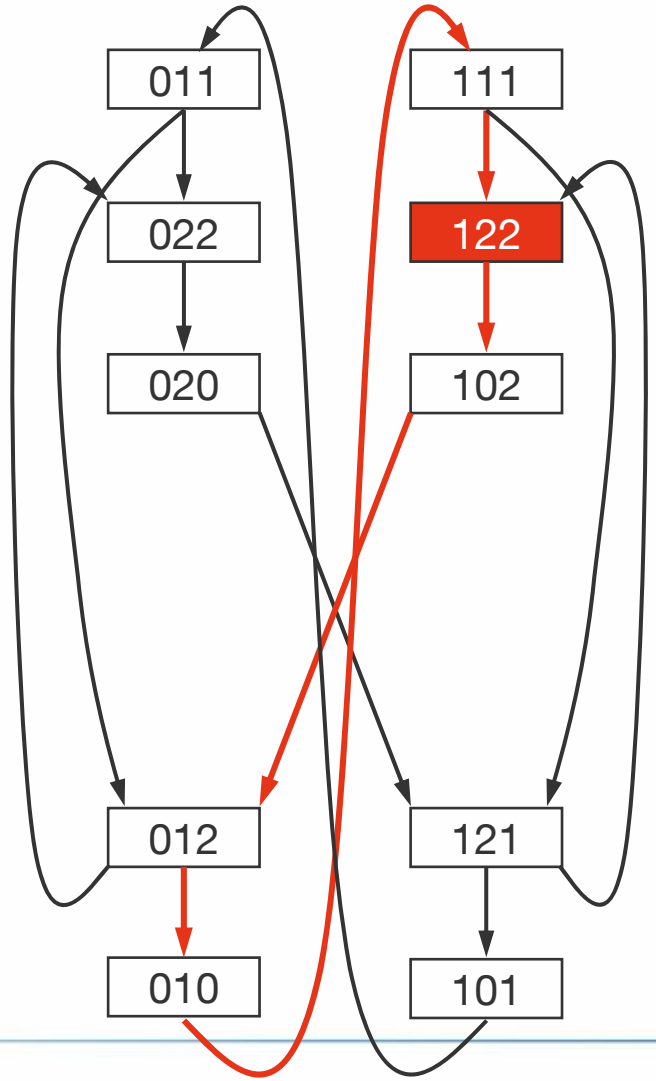



```
boolean x ← 0

proc rw0 {
    while (true) {
0:      x ← 0
1:      sync ( )
2:      if (x = 0)
3:          use_resource
    }
}

proc rw1 {
    while (true) {
0:      x ← 1
1:      sync ( )
2:      if (x = 1)
3:          use_resource
    }
}
```

Process Interaction for MemMS

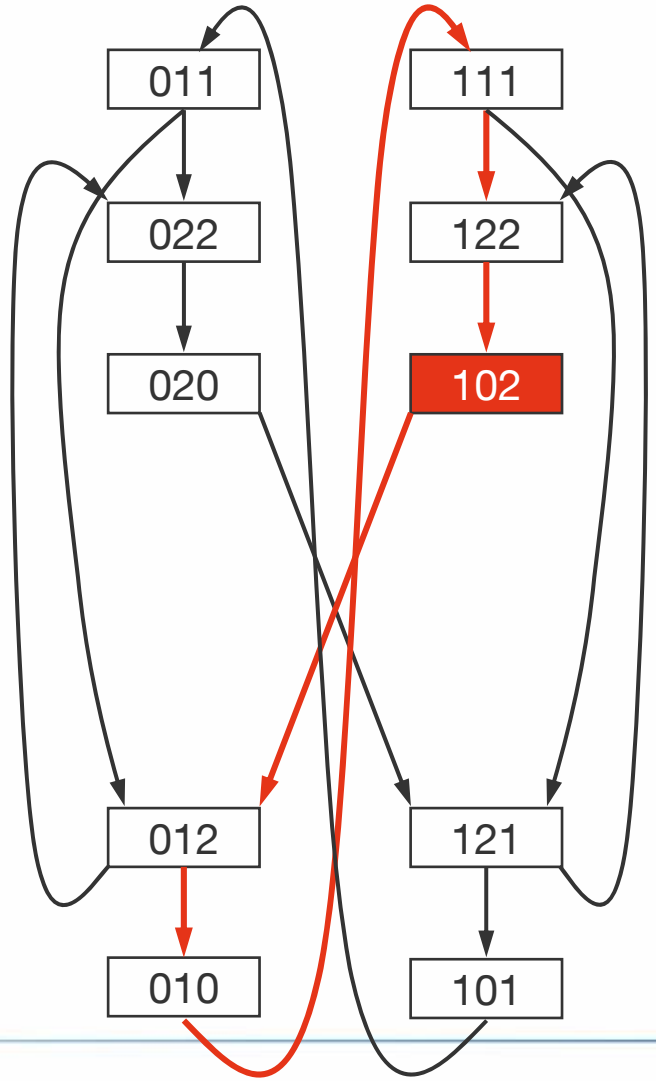



```
boolean x ← 0

proc rw0 {
    while (true) {
0:      x ← 0
1:      sync ( )
2:      if (x = 0)
3:          use_resource
    }
}

proc rw1 {
    while (true) {
0:      x ← 1
1:      sync ( )
2:      if (x = 1)
3:          use_resource
    }
}
```

Process Interaction for MemMS

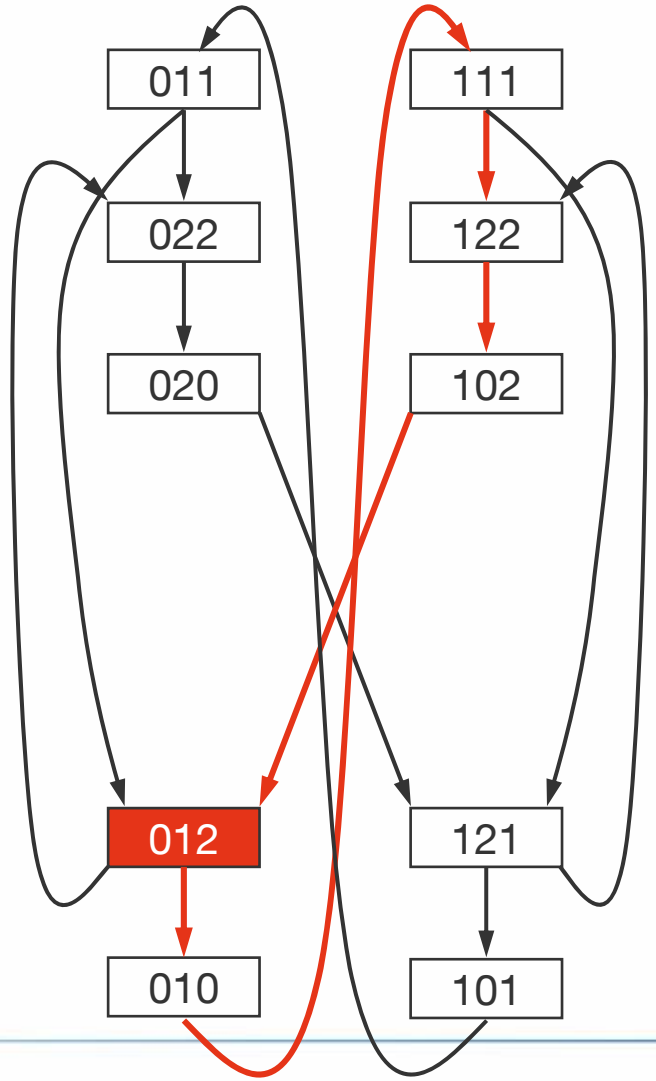



```
boolean x ← 0

proc rw0 {
    while (true) {
0:      x ← 0
1:      sync ( )
2:      if (x = 0)
3:          use_resource
    }
}

proc rw1 {
    while (true) {
0:      x ← 1
1:      sync ( )
2:      if (x = 1)
3:          use_resource
    }
}
```

Process Interaction for MemMS




```
boolean x ← 0

proc rw0 {
    while (true) {
0:      x ← 0
1:      sync ( )
2:      if (x = 0)
3:          use_resource
    }
}

proc rw1 {
    while (true) {
0:      x ← 1
1:      sync ( )
2:      if (x = 1)
3:          use_resource
    }
}
```

3 Foundations

Overview

- terms and definitions
- process interaction for MemMS
- process interaction for MesMS
- example of a parallel program

Process Interaction for MesMS

- message passing paradigm
  - no shared memory for synchronisation and communication
  - hence, transfer mechanism for information interchange necessary
  - message passing
    - messages: data units transferred between processes
    - send / receive operations instead of read / write operations
  - implicit (sequential) order during send / receive-stage
    - a message can only be received after a prior send
    - communication via message passing (independent from the transferred data) leads to an implicit synchronisation
    - synchronisation due to availability / unavailability of messages
    - messages are resources that don't exist before the send and in general also after the receive operation

Process Interaction for MesMS

- messages
  - created whenever a process performs a send
  - necessary information to be provided from the sender
    - destination (process, node, communication channel, e. g.)
    - unique identifier of message (number, e. g.)
    - memory (so called send buffer) containing the data to be transferred with the message
    - data type and amount of elements within the send buffer
  - data type and amount of elements have to match for the receiver, otherwise a correct interpretation of the data is in doubt

Process Interaction for MesMS

- sending messages
  - send operations can be
    - *synchronous / asynchronous*: a process performing a send is either blocked (synchronous) or not (asynchronous) until a respective receive operation is executed
    - *buffered / unbuffered*: a send operation may first copy the data from the send buffer to a system buffer (buffered) or directly perform the transfer (unbuffered ➔ faster, but risk of overwriting the send buffer due to parallel execution of transfer (NIC) and next send operation (CPU) possible)
    - *blocking / non-blocking*: a send operation can either be blocked until the send buffer has been emptied (blocking) or immediately give control to the next instruction of the sending process (non-blocking ➔ risk of overwriting send buffer)

Process Interaction for MesMS

- receiving messages
  - a process has to specify which message to receive (via message identifier or wildcard) and where to store the data (so called receive buffer)
  - receive operations can be
    - *destructive / non-destructive*: a receive operation copies the data transferred with the message into the receive buffer and either destroys the message (destructive) or keeps it for later usage (non-destructive)
    - *synchronous / asynchronous*: a process performing a receive is either blocked until a message has arrived (synchronous) or not (asynchronous), thus it can continue with its execution and check again at a later point in time

Process Interaction for MesMS

- synchronisation characteristics
  - synchronous message passing
    - both sender and receiver use synchronous operations
    - content and destination of a message are known on both sides at the same time
    - hence, the message can directly be transferred from the send to the receive buffer
  - asynchronous message passing
    - at least sender or receiver uses asynchronous operations (typically the sender)
    - as not both processes are available for communication at the same time, some buffer for the message transfer from sender to receiver is necessary

Process Interaction for MesMS

- addressing modes
  - different addressing modes can be distinguished
    - *direct naming*: process identifiers are used for sender and receiver ➔ identifiers have to be known during development
    - *mailbox*: global memory where processes can store (send) and remove (receive) messages (used in the Distributed Execution and Communication Kernel (DECK), e. g.)
    - *port*: a port is bound to one process and can be used in one direction only, i. e. either for sending or receiving messages
    - *connection / channel*: for the usage of ports the setup of connections or channels is required, i. e. the connection of a send port of one process with the receive port of another process ➔ data written from the sender to its port can be read from the receiver on the other port

3 Foundations

Overview

- terms and definitions
- process interaction for MemMS
- process interaction for MesMS
- example of a parallel program

Example of a Parallel Program

- labyrinth
  - given: map of some labyrinth that contains
    - one entrance
    - one exit
    - no cycles

  - task: determine if there exists a way from the entrance through the labyrinth to the exit (not the way itself)

Example of a Parallel Program

- labyrinth (cont'd)
  - map stored as matrix of size $M \times N$

Example of a Parallel Program

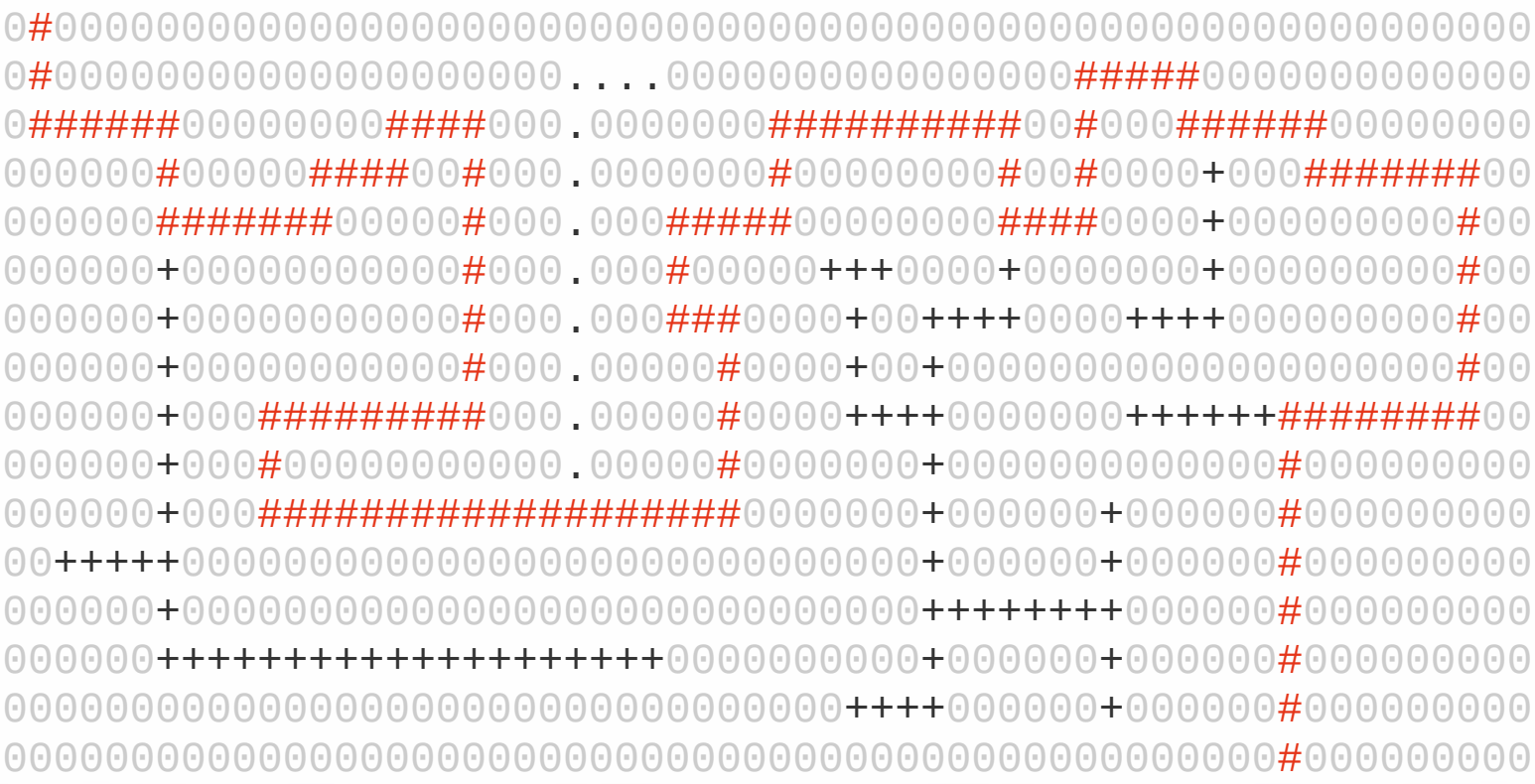
- labyrinth (cont'd)
  - sequential algorithm (1)

```
position ← entrance

while (true) do
    switch (position) do
        case 'crossroads':  position ← TURN_RIGHT()
        case 'dead end':    position ← TURN_BACK()
        case 'exit':        halt(OKAY)
        case 'entrance':    halt(ERROR)
    od
od
```

Example of a Parallel Program

- labyrinth (cont'd)
  - sequential algorithm (2)



Example of a Parallel Program

- labyrinth (cont'd)
  - parallelisation
    - basic questions
      - which parts to be parallelised
      - which structure: function or data parallelism
      - which model: shared or distributed memory

Example of a Parallel Program

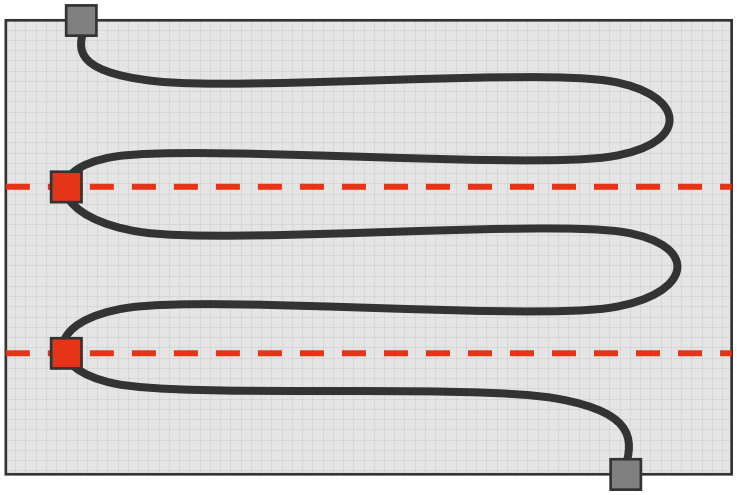
- labyrinth (cont'd)
  - variant A: *function parallelism*
    - start new processes at crossroads
    - terminate processes at dead ends
    - halt in case
      - one process has reached the exit
      - all processes have been terminated
    - questions
      - shared or distributed memory
      - total number of processes ➔ queue of tasks
      - drawbacks

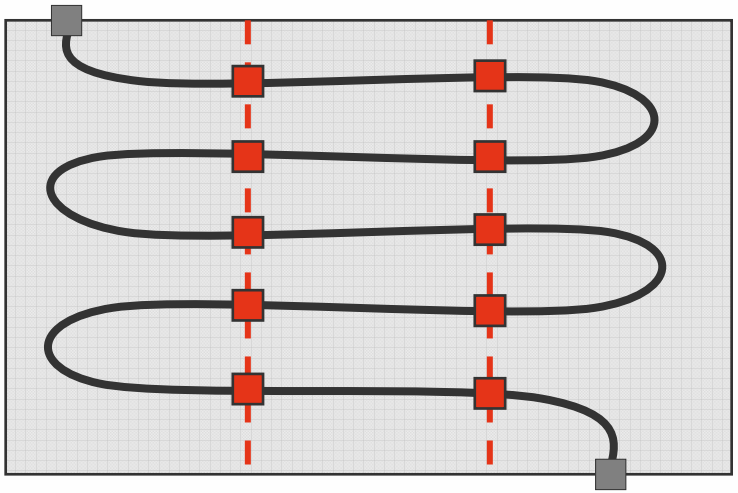
Example of a Parallel Program

- labyrinth (cont'd)
  - variant B: *competitive parallelism*
    - start *N* processes following *N* different algorithms
    - first process reaching exit or entrance tells other processes to stop
    - possible algorithms
      - always go left instead of going right
      - start from the exit and try to reach the entrance
      - randomly walk around and remember all paths that have already been examined
      - ...
    - questions
      - shared or distributed memory
      - drawbacks

Example of a Parallel Program

- labyrinth (cont'd)
  - variant C: *data parallelism* (1)
    - naïve approach
      - cut matrix into *N* parts
      - solve corresponding subproblems and check if there exist matching entrance-exit pairs ➔ communication overhead





Example of a Parallel Program

- labyrinth (cont'd)
  - variant C: *data parallelism* (2)
    - "better" approach
      - cut matrix A into parts A_i and distribute among processes
      - flood fill each part A_i with different "colors"
      - collect boundaries of all A_i
      - check if (color) transition from entrance to exit is possible
    - questions
      - shared or distributed memory
      - drawbacks