Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Shared-Memory Concurrency

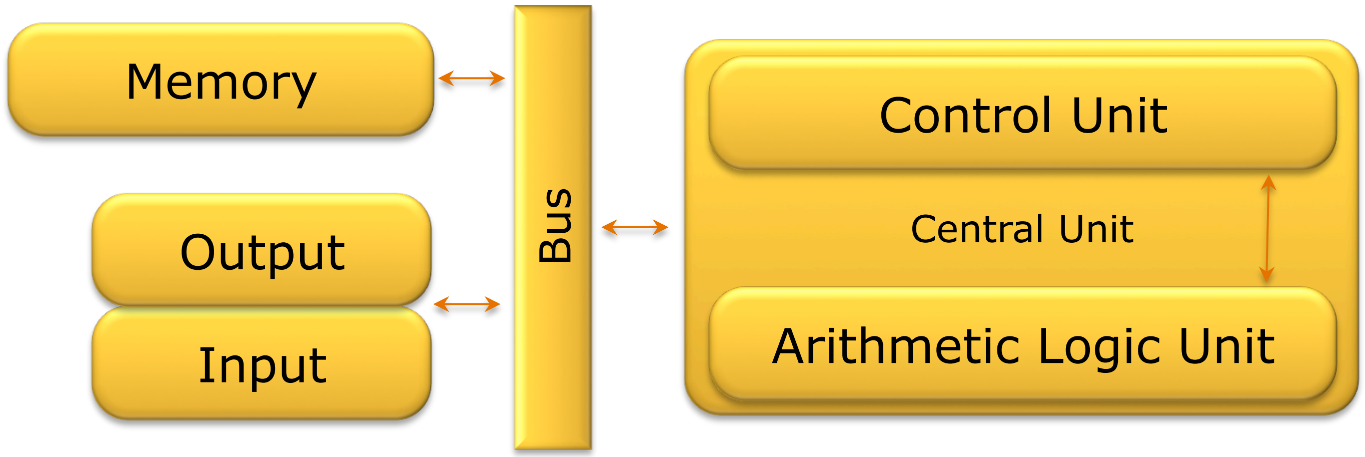
Programmierung Paralleler und Verteilter Systeme (PPV)

Sommer 2015

Frank Feinbube, M.Sc., Felix Eberhardt, M.Sc.,
Prof. Dr. Andreas Polze

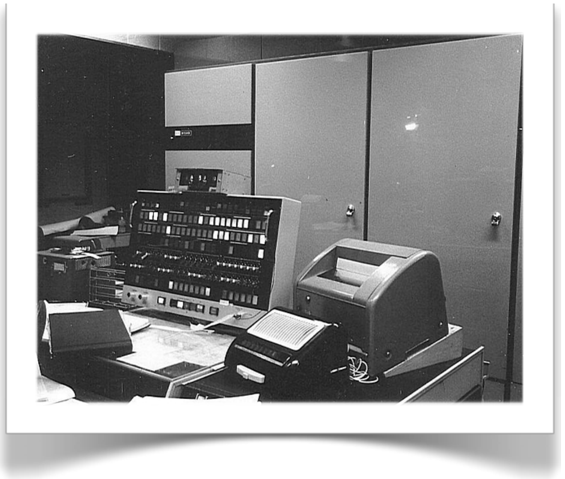
Von Neumann Model

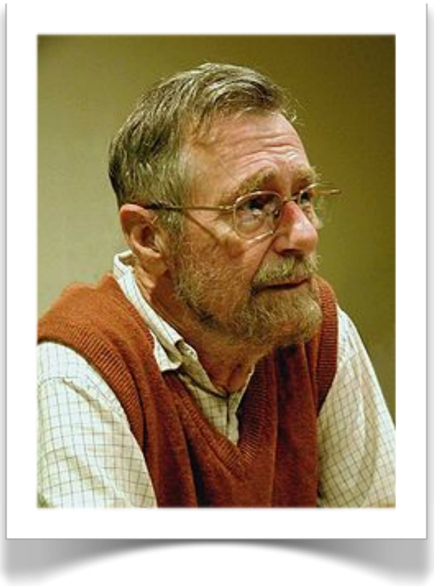
- Processor executes a sequence of instructions
  - Arithmetic operations
  - Memory to be read / written
  - Address of next instruction
- Software layering tackles complexity of instruction stream
- Parallelism adds coordination problem between multiple instruction streams being executed



Concurrency in History

- *1961, Atlas Computer, Kilburn & Howarth*
  - Based on Germanium transistors, assembler only
  - First use of interrupts to simulate concurrent execution of multiple programs - *multiprogramming*
- 60's and 70's: Foundations for concurrent software developed
  - *1965, Cooperating Sequential Processes, E.W.Dijkstra*
    - First principles of concurrent programming
    - Basic concepts: *Critical section, mutual exclusion, fairness, speed independence*





Cooperating Sequential Processes [Dijkstra]

- *1965, Cooperating Sequential Processes, E.W.Dijkstra*
  - Comparison of sequential and non-sequential machine
  - Example: Electromagnetic solution to find the **largest** value in an array
    - Current lead through magnet coil
    - Switch to magnet with larger current
  - Progress of time is relevant

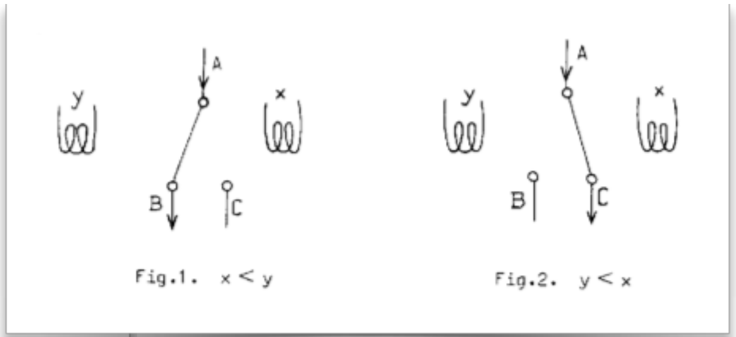


Fig.1. $x < y$ Fig.2. $y < x$

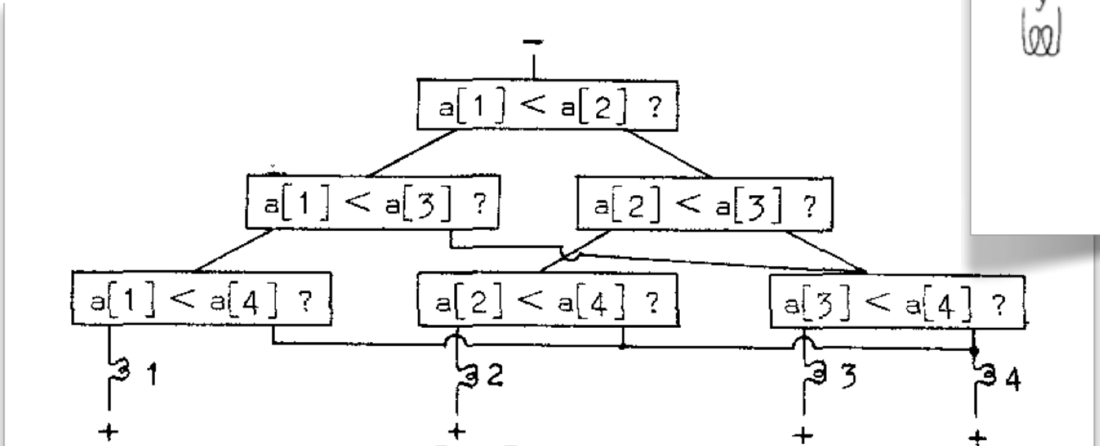


Fig.3.

Cooperating Sequential Processes [Dijkstra]

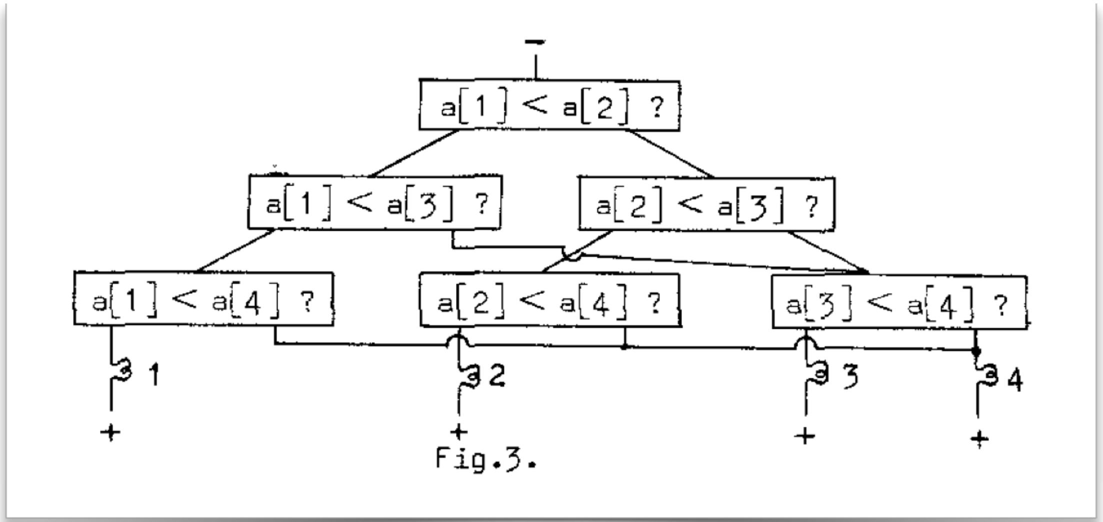
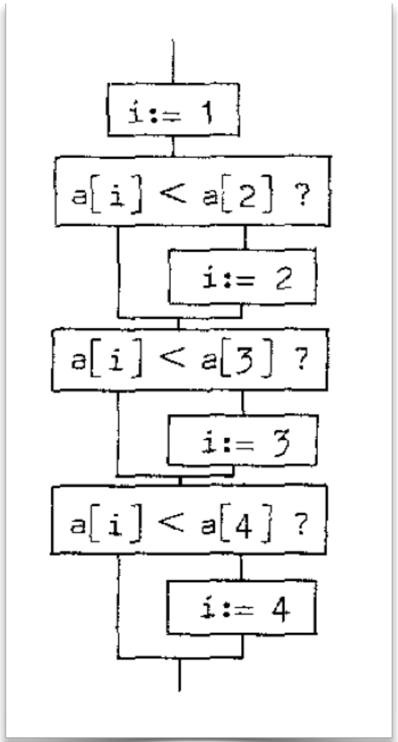


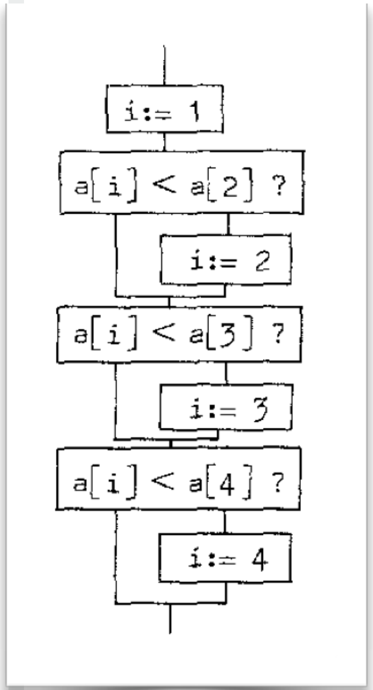
Fig.3.

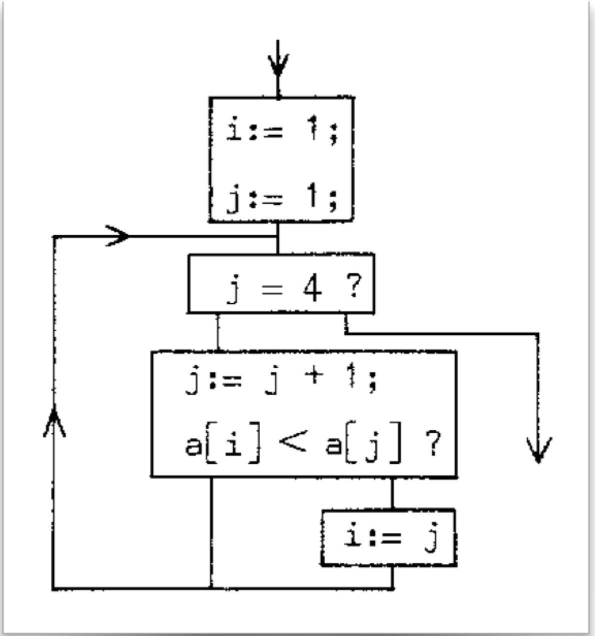


- **Progress of time** is relevant
  - After applying one step, machine needs some time to show the result
  - Same line differs only in left operand
  - Concept of a parameter that comes from history, leads to alternative setup for the same behavior
- Rules of behavior form a **program**

Cooperating Sequential Processes [Dijkstra]

- Idea: Many programs for expressing the same intent
  - Example: Consider repetitive nature of the problem
  - Invest in a variable j
    → generalize the solution for any number of items





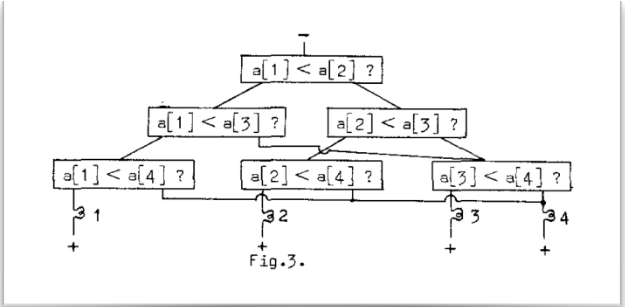
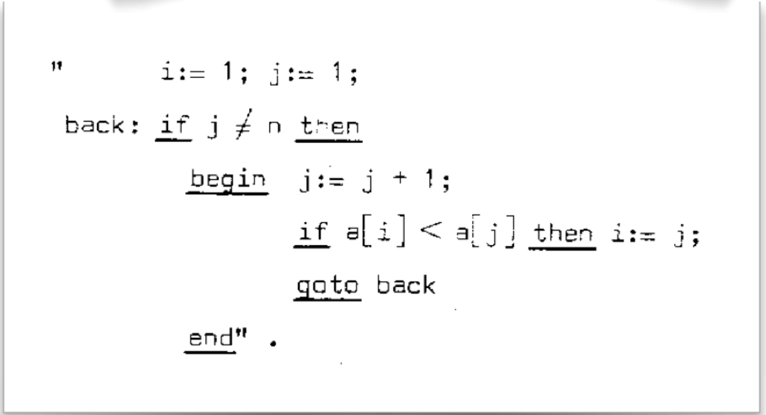


Fig.3.

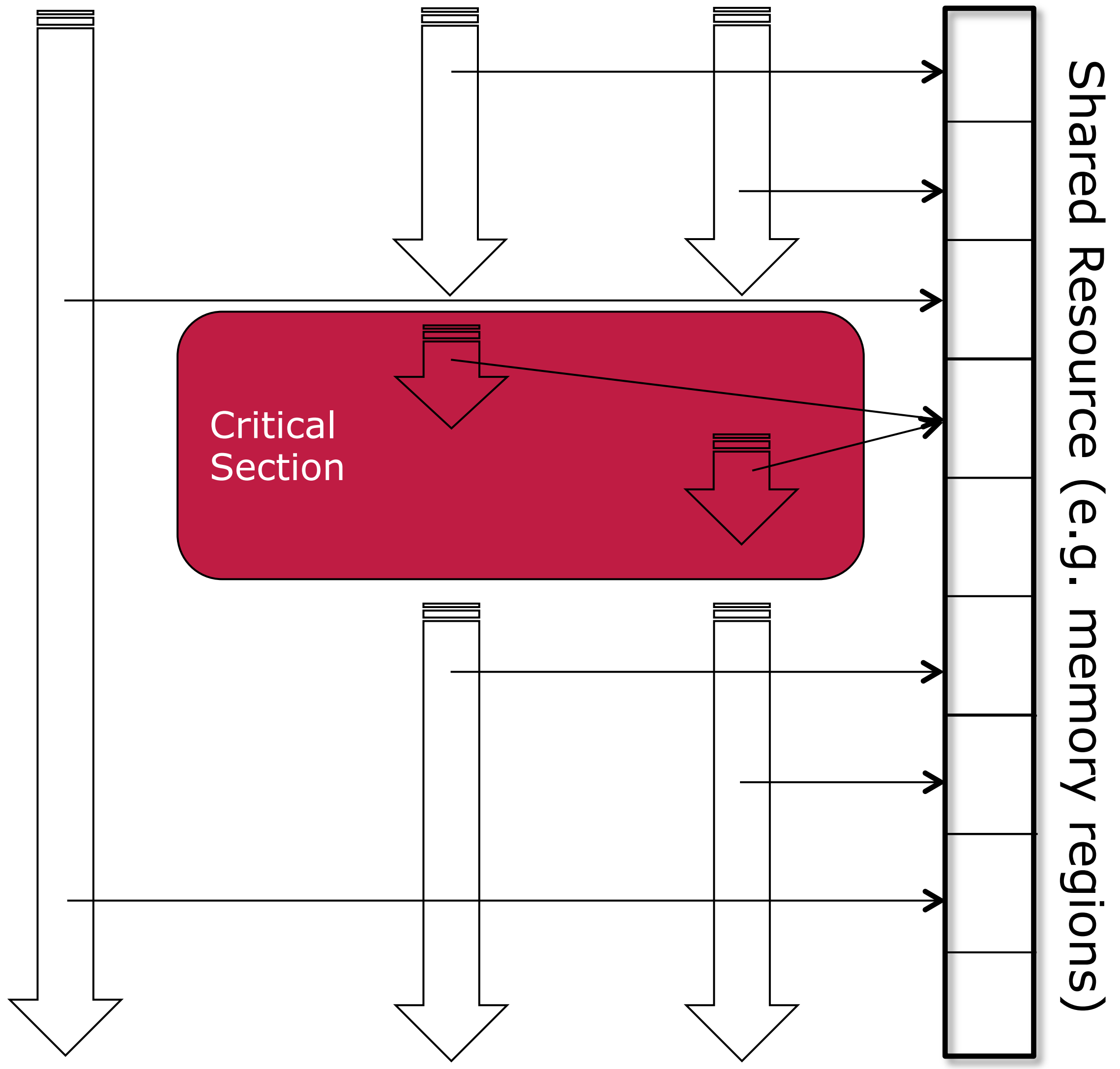


```
"       i:= 1; j:= 1;
back: if j ≠ n then
        begin  j:= j + 1;
               if a[i] < a[j] then i:= j;
               goto back
        end" .
```

Cooperating Sequential Processes [Dijkstra]

- Assume we have multiple of these sequential programs
- How about the cooperation between such, maybe loosely coupled, sequential processes ?
  - Beside rare moments of communication, processes run autonomously
- Disallow any assumption about the relative speed
  - Aligns to understanding of sequential process, which is not affected in its correctness by execution time
  - If this is not fulfilled, it might bring „analogue interferences“
- Note: Dijkstra already identified the **„race condition“** problem
- Idea of **a critical section** for two cyclic sequential processes
  - At any moment, at most one process is engaged in the section
  - Implemented through common variables
  - Demands atomic read / write behavior

Critical Section

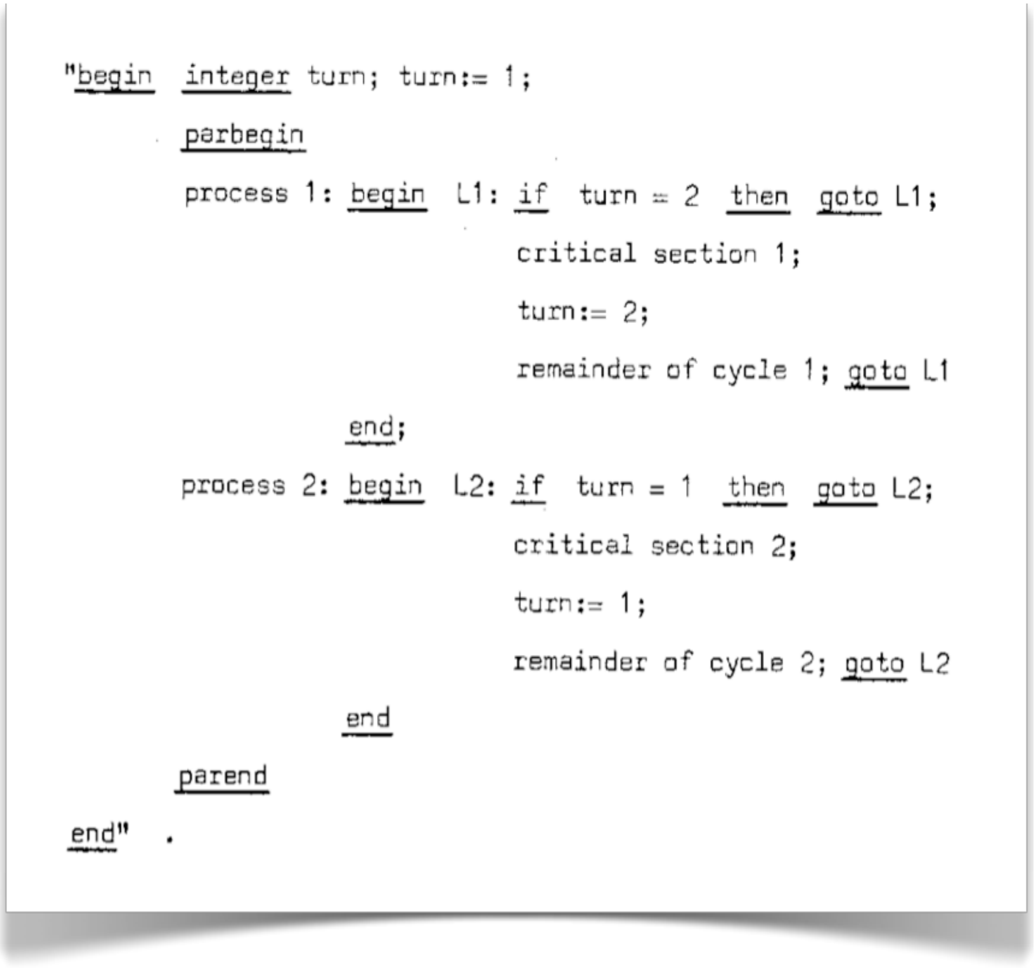


Critical Section

- N threads has some code - **critical section** - with shared data access
- **Mutual Exclusion** requirement
  - Only one thread at a time is allowed into its critical section, among all threads that have critical sections for the same resource.
- **Progress** requirement
  - If no other thread is in the critical section, the decision for entering should not be postponed indefinitely. Only threads that wait for entering the critical section are allowed to participate in decisions.
- **Bounded Waiting** requirement
  - It must not be possible for a thread requiring access to a critical section to be delayed indefinitely by other threads entering the section (**starvation problem**)

Cooperating Sequential Processes [Dijkstra]

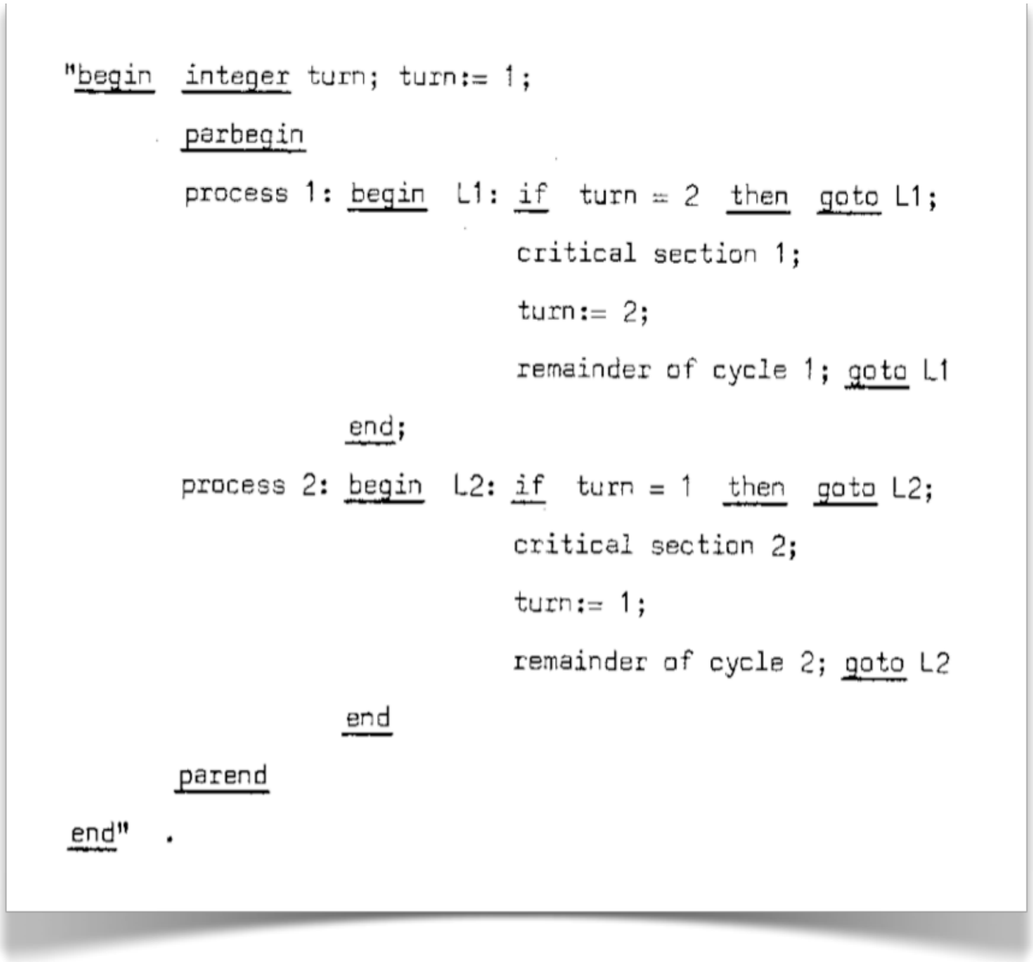
- Attempt to develop a critical section concept in ALGOL60
  - `parbegin / parend` extension
  - Atomicity on source code line level



```
"begin  integer turn; turn:= 1;
        parbegin
        process 1: begin  L1: if  turn = 2  then  goto L1;
                              critical section 1;
                              turn:= 2;
                              remainder of cycle 1; goto L1
                   end;
        process 2: begin  L2: if  turn = 1  then  goto L2;
                              critical section 2;
                              turn:= 1;
                              remainder of cycle 2; goto L2
                   end
        parend
end"  .
```

Cooperating Sequential Processes [Dijkstra]

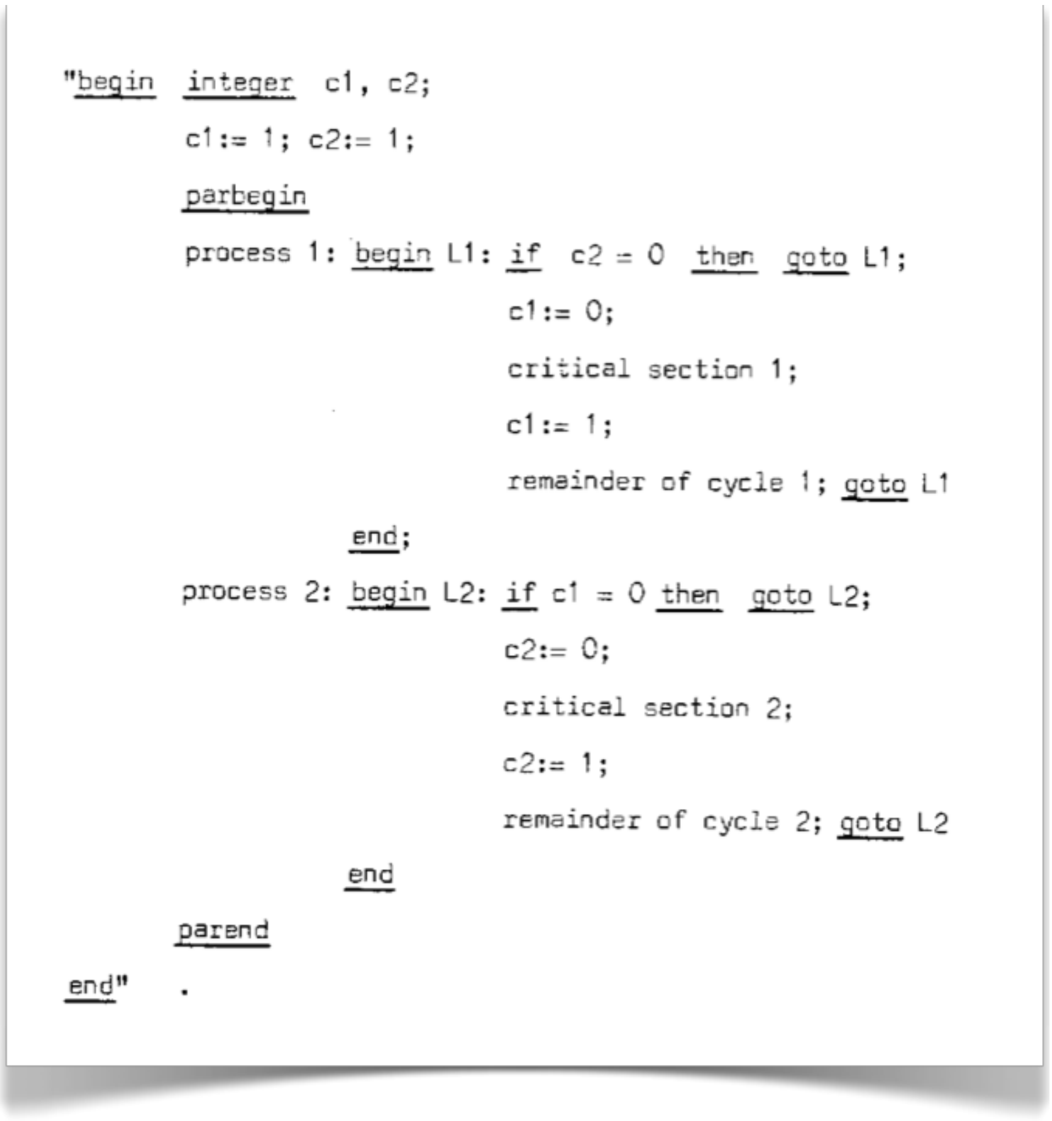
- Attempt to develop a critical section concept in ALGOL60
  - `parbegin / parend` extension
  - Atomicity on source code line level
- First approach
  - Too restrictive, since strictly alternating
  - One process may die or hang outside of the critical section



```
"begin integer turn; turn:= 1;
    parbegin
    process 1: begin L1: if turn = 2 then goto L1;
                         critical section 1;
                         turn:= 2;
                         remainder of cycle 1; goto L1
               end;
    process 2: begin L2: if turn = 1 then goto L2;
                         critical section 2;
                         turn:= 1;
                         remainder of cycle 2; goto L2
               end
    parend
end" .
```

Cooperating Sequential Processes [Dijkstra]

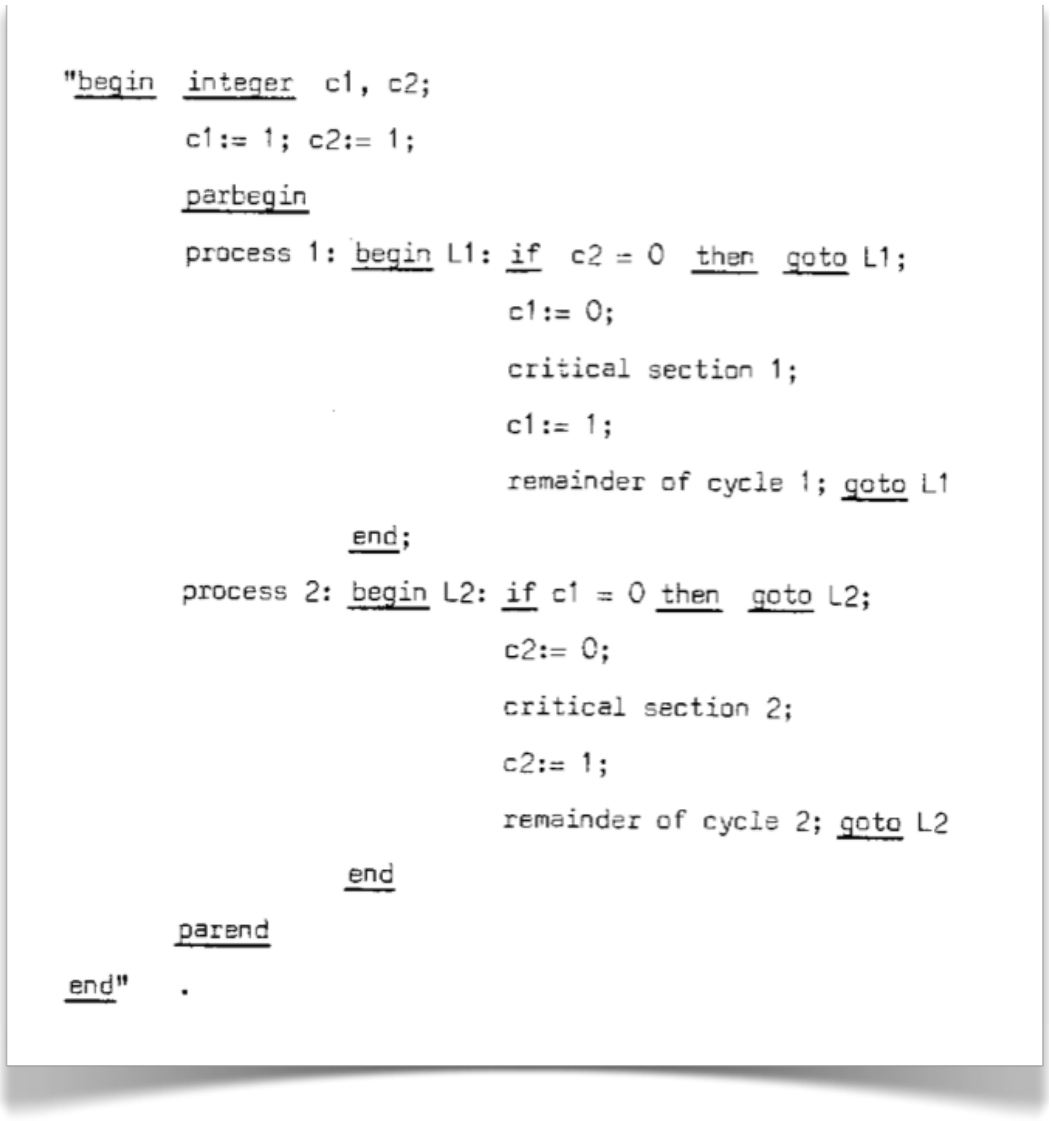
- Separate indicators for enter/ leave
- More fine-grained waiting approach



```
"begin  integer  c1, c2;
        c1:= 1; c2:= 1;
        parbegin
        process 1: begin L1: if  c2 = 0  then  goto L1;
                             c1:= 0;
                             critical section 1;
                             c1:= 1;
                             remainder of cycle 1; goto L1
                   end;
        process 2: begin L2: if c1 = 0 then  goto L2;
                             c2:= 0;
                             critical section 2;
                             c2:= 1;
                             remainder of cycle 2; goto L2
                   end
        parend
end"   .
```

Cooperating Sequential Processes [Dijkstra]

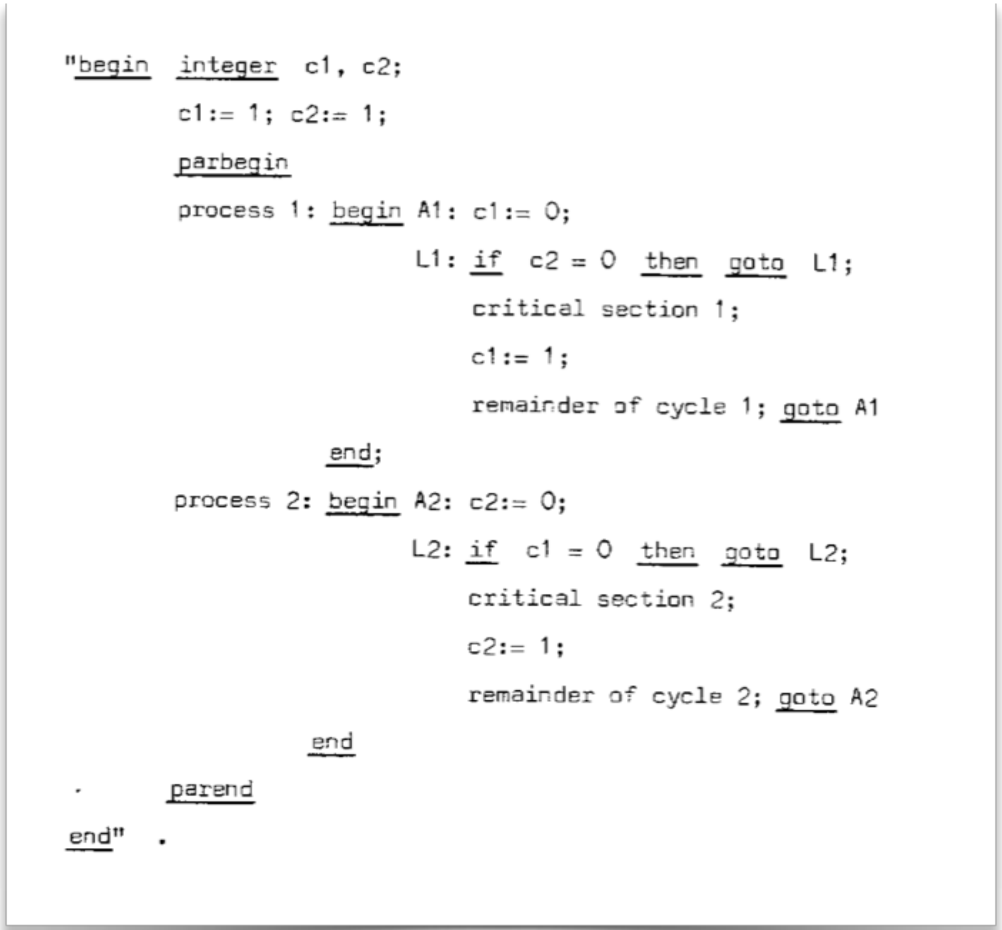
- Separate indicators for enter/ leave
- More fine-grained waiting approach
- Too optimistic, both processes may end up in the critical section



```
"begin  integer  c1, c2;
        c1:= 1; c2:= 1;
        parbegin
        process 1: begin L1: if  c2 = 0  then  goto L1;
                             c1:= 0;
                             critical section 1;
                             c1:= 1;
                             remainder of cycle 1; goto L1
                   end;
        process 2: begin L2: if c1 = 0 then  goto L2;
                             c2:= 0;
                             critical section 2;
                             c2:= 1;
                             remainder of cycle 2; goto L2
                   end
        parend
end"    .
```

Cooperating Sequential Processes [Dijkstra]

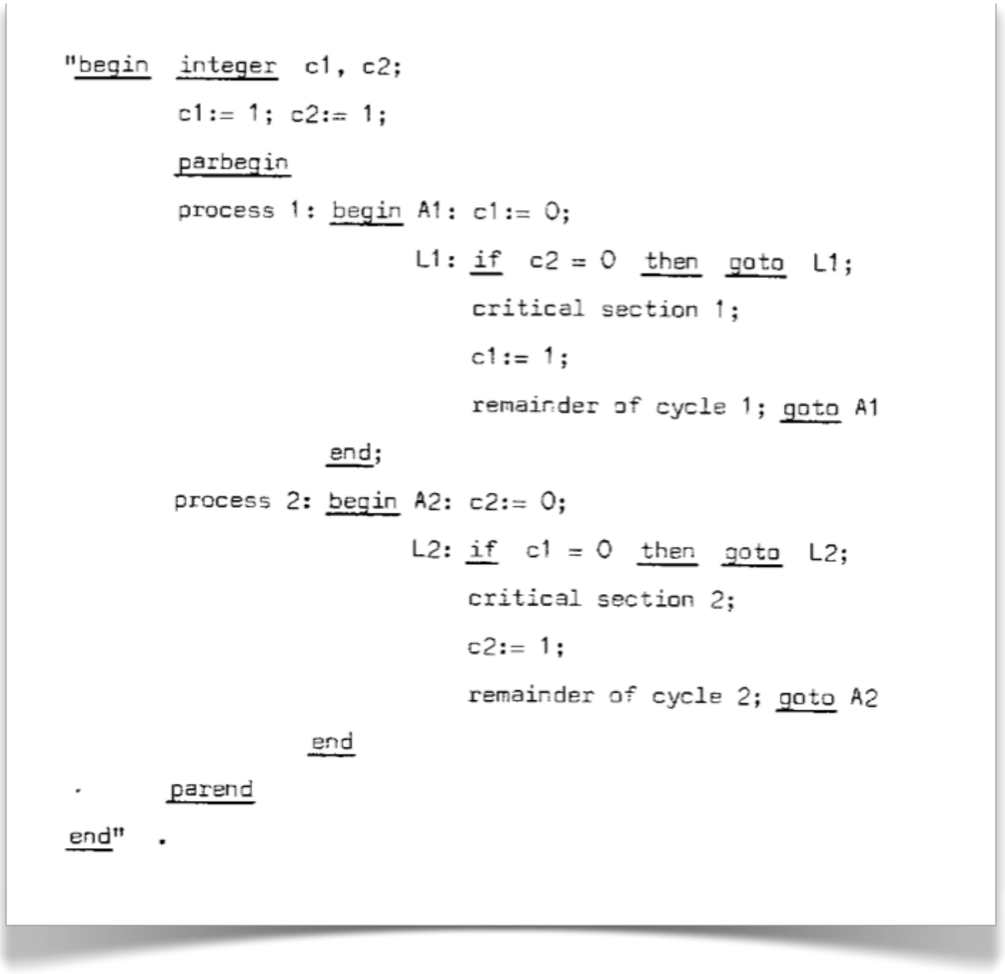
- First ‚raise the flag‘, then check for the other
- Concept of a selfish process



```
"begin  integer  c1, c2;
        c1:= 1; c2:= 1;
        parbegin
        process 1: begin A1: c1:= 0;
                         L1: if  c2 = 0  then  goto  L1;
                             critical section 1;
                             c1:= 1;
                             remainder of cycle 1; goto A1
                   end;
        process 2: begin A2: c2:= 0;
                         L2: if  c1 = 0  then  goto  L2;
                             critical section 2;
                             c2:= 1;
                             remainder of cycle 2; goto A2
                   end
        parend
end"  .
```

Cooperating Sequential Processes [Dijkstra]

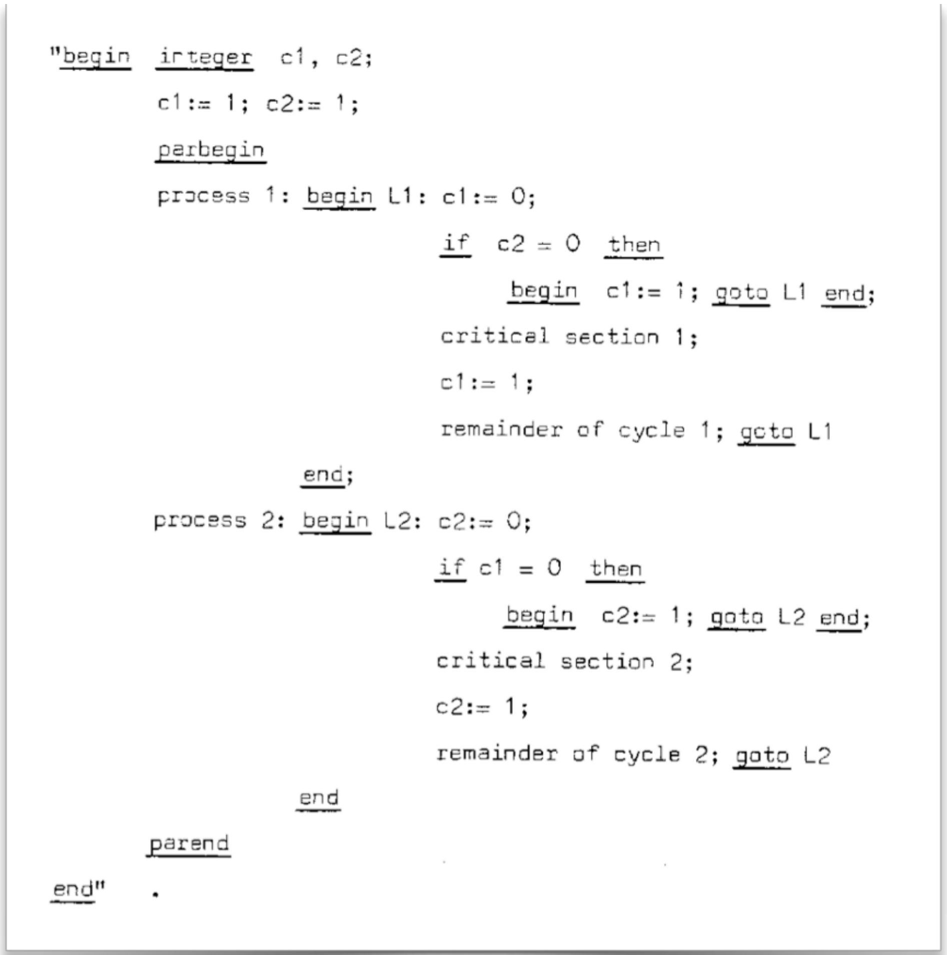
- First ‚raise the flag ‘, then check for the other
- Concept of a selfish process
- Mutual exclusion works
  - If c1=0, then c2=1, and vice versa
- Variables change outside of the critical section only
  - Danger of mutual blocking (**deadlock**)



```
"begin  integer  c1, c2;
        c1:= 1; c2:= 1;
        parbegin
        process 1: begin A1: c1:= 0;
                         L1: if  c2 = 0  then  goto  L1;
                             critical section 1;
                             c1:= 1;
                             remainder of cycle 1; goto A1
                   end;
        process 2: begin A2: c2:= 0;
                         L2: if  c1 = 0  then  goto  L2;
                             critical section 2;
                             c2:= 1;
                             remainder of cycle 2; goto A2
                   end
        parend
end"  .
```

Cooperating Sequential Processes [Dijkstra]

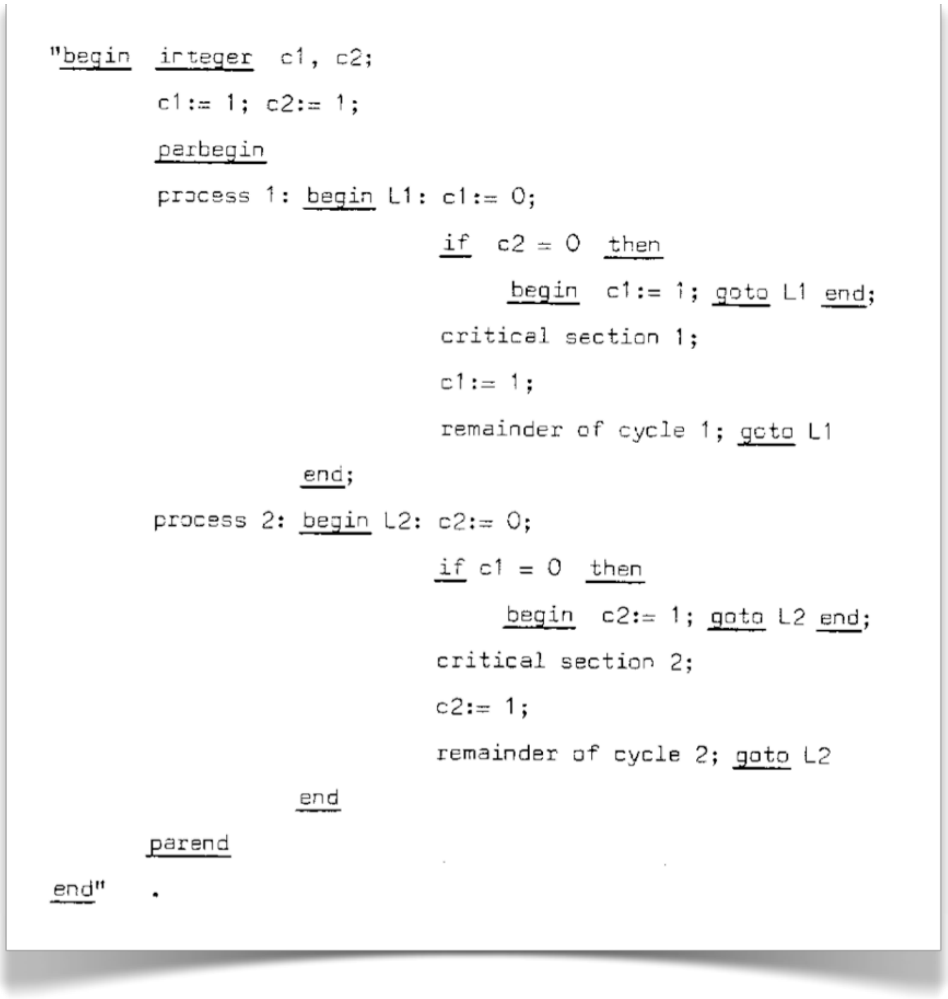
- Reset locking of critical section if the other one is already in



```
"begin integer c1, c2;
         c1:= 1; c2:= 1;
         parbegin
         process 1: begin L1: c1:= 0;
                              if  c2 = 0  then
                                  begin  c1:= 1; goto L1  end;
                              critical section 1;
                              c1:= 1;
                              remainder of cycle 1; goto L1
                    end;
         process 2: begin L2: c2:= 0;
                              if c1 = 0  then
                                  begin  c2:= 1; goto L2 end;
                              critical section 2;
                              c2:= 1;
                              remainder of cycle 2; goto L2
                    end
         parend
end"   .
```

Cooperating Sequential Processes [Dijkstra]

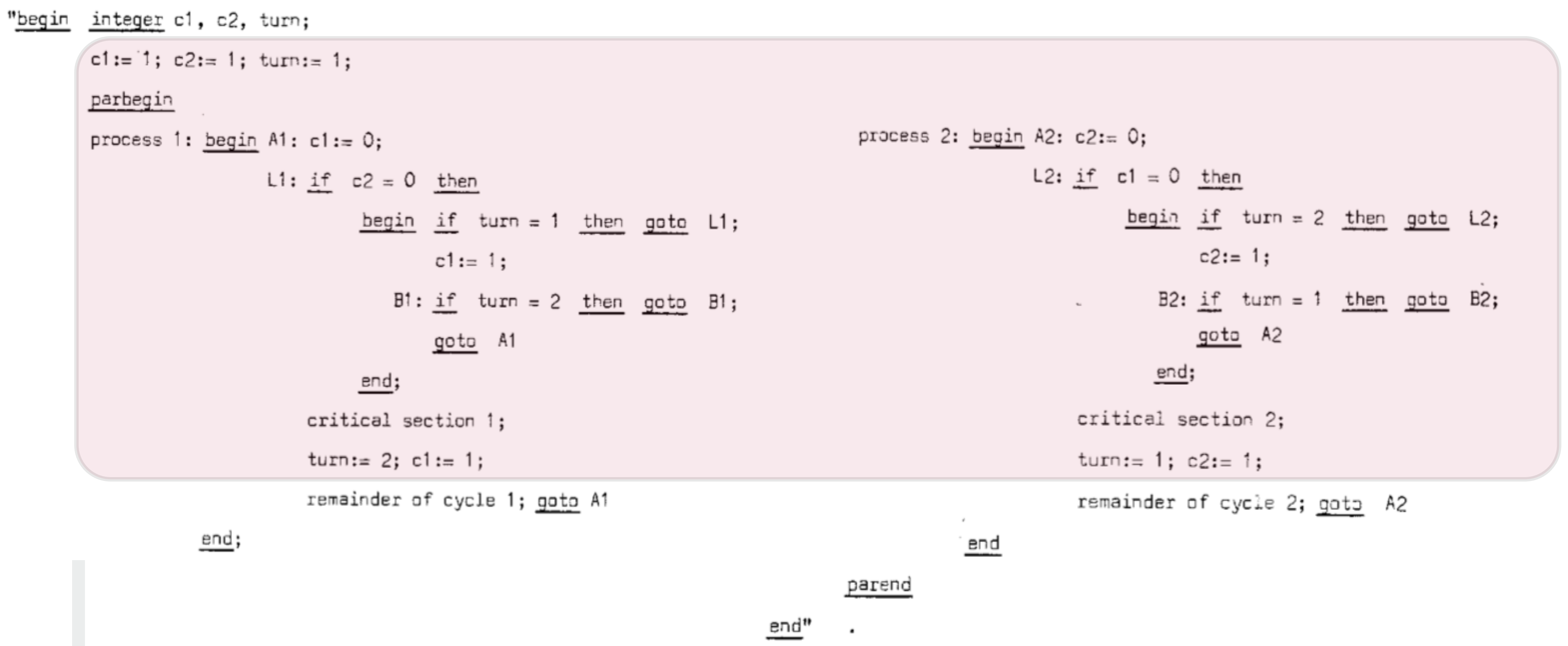
- Reset locking of critical section if the other one is already in
- Problem due to assumption of relative speed
  - Process 1 may run much faster, always hits the point in time were c2=1
  - Can lead for one process to ‚wait forever' without any progress



```
"begin  integer  c1, c2;
        c1:= 1; c2:= 1;
        parbegin
        process 1: begin L1: c1:= 0;
                             if  c2 = 0  then
                                  begin  c1:= 1; goto L1 end;
                             critical section 1;
                             c1:= 1;
                             remainder of cycle 1; goto L1
                   end;
        process 2: begin L2: c2:= 0;
                             if c1 = 0  then
                                  begin  c2:= 1; goto L2 end;
                             critical section 2;
                             c2:= 1;
                             remainder of cycle 2; goto L2
                   end
        parend
end"   .
```

Cooperating Sequential Processes [Dijkstra]

- Solution: Dekker's algorithm, referenced by Dijkstra
  - Combination of fourth approach and turn ‚variable', which realizes mutual blocking avoidance through prioritization
  - Idea: Spin for section entry only if it is your turn



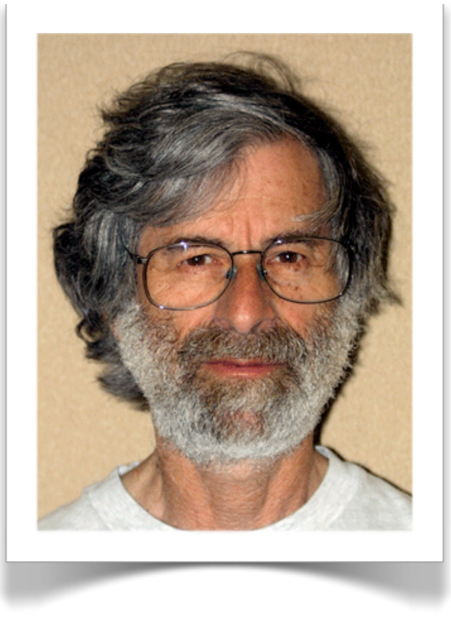
```
"begin integer c1, c2, turn;
c1:= 1; c2:= 1; turn:= 1;
parbegin
process 1: begin A1: c1:= 0;
           L1: if c2 = 0 then
                   begin if turn = 1 then goto L1;
                         c1:= 1;
                      B1: if turn = 2 then goto B1;
                         goto A1
                   end;
              critical section 1;
              turn:= 2; c1:= 1;
              remainder of cycle 1; goto A1
           end;
process 2: begin A2: c2:= 0;
           L2: if c1 = 0 then
                   begin if turn = 2 then goto L2;
                         c2:= 1;
                      B2: if turn = 1 then goto B2;
                         goto A2
                   end;
              critical section 2;
              turn:= 1; c2:= 1;
              remainder of cycle 2; goto A2
           end
parend
end" .
```

Critical Sections

- Dekker provided first correct solution only based on shared memory, guarantees three major properties
  - **Mutual exclusion**
  - **Freedom from deadlock**
  - **Freedom from starvation**
- Generalization by Lamport with the **Bakery algorithm**
  - Relies only on memory access atomicity
- Both solutions assume atomicity and predictable sequential execution on machine code level
- Hardware today: Unpredictable sequential instruction stream
    - Out-of-order execution
    - Re-ordered memory access
    - Compiler optimizations

Bakery Algorithm [Lamport]

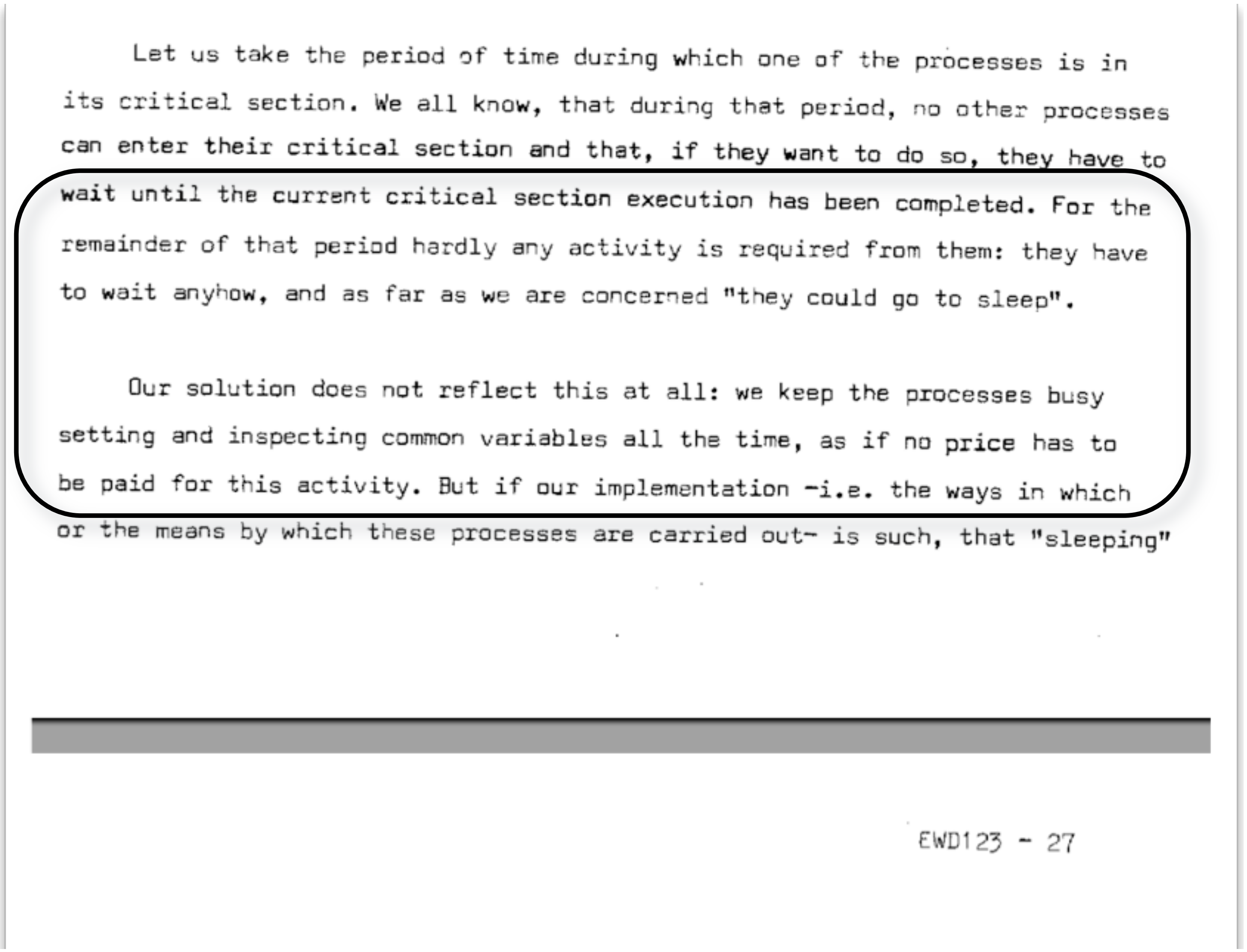
```
def lock(i) { # wait until we have the smallest num
    choosing[i] = True;
    num[i] = max(num[0],num[1] ...,num[n-1]) + 1;
    choosing[i] = False;
    for (j = 0; j < n; j++) {
      while (choosing[i]) ;
      while ((num[j] != 0) &&
       ((num[j],j) "<" (num[i],i))) {};}}
def unlock(i) {
    num[i] = 0; }

lock(i)
… critical section …
unlock(i)
```



Test-and-Set

- **Test-and-set** processor instruction, wrapped by the operating system
  - Write to a memory location and return its old value as atomic step
  - Also known as **compare-and-swap (CAS)** or **read-modify-write**
- Idea: Spin in writing 1 to a memory cell, until the old value was 0
  - Between writing and test, no other operation can modify the value
- Busy waiting for acquiring a lock
- Efficient especially for short waiting periods

```
function Lock(boolean *lock) {
    while (test_and_set (lock))
      ;
}

#define LOCKED 1
 int TestAndSet(int* lockPtr) {
     int oldValue;
     oldValue = SwapAtomic(lockPtr, LOCKED);
     return oldValue == LOCKED;
 }
```



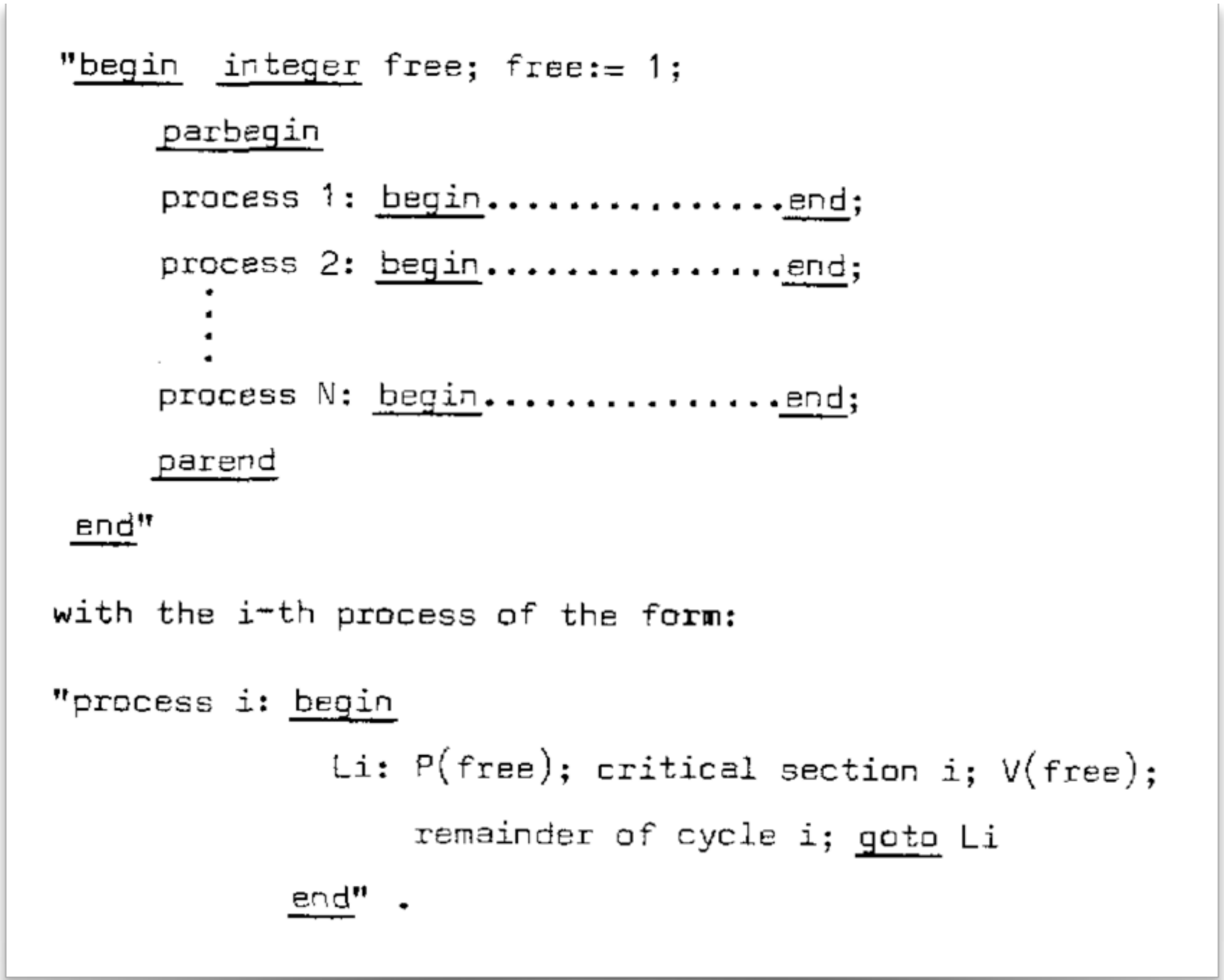
Let us take the period of time during which one of the processes is in its critical section. We all know, that during that period, no other processes can enter their critical section and that, if they want to do so, they have to wait until the current critical section execution has been completed. For the remainder of that period hardly any activity is required from them: they have to wait anyhow, and as far as we are concerned "they could go to sleep".

Our solution does not reflect this at all: we keep the processes busy setting and inspecting common variables all the time, as if no price has to be paid for this activity. But if our implementation -i.e. the ways in which or the means by which these processes are carried out- is such, that "sleeping"

Binary and General Semaphores [Dijkstra]

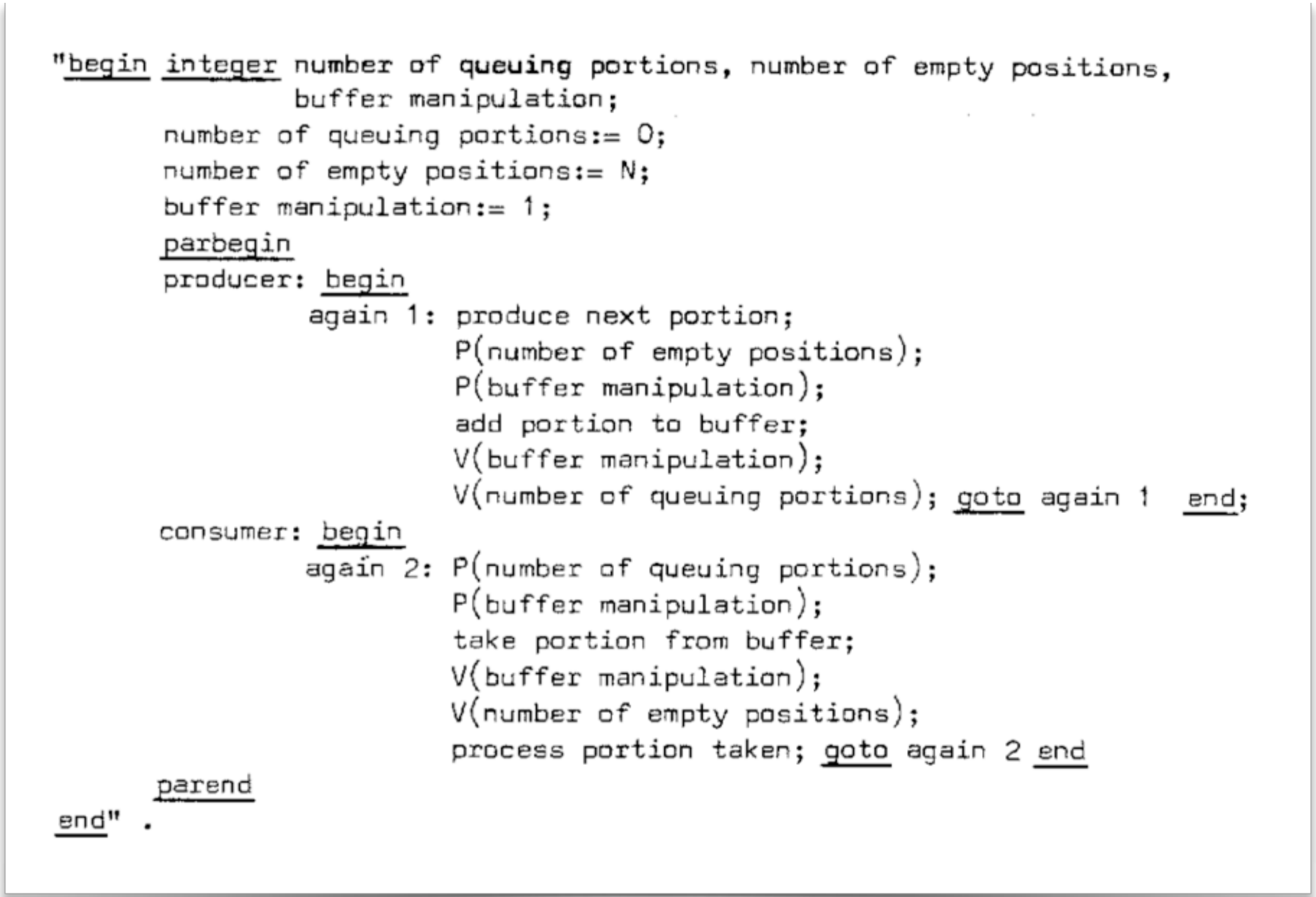
- Find a solution to allow waiting sequential processes to ‚sleep'
- Special purpose integer called **semaphore**
    - **P**-operation: Decrease value of its argument semaphore by 1 as atomic step, **"wait"** if the semaphore is already zero
    - **V**-operation: Increase value of its argument semaphore by 1 as atomic step, useful as **„signal"** operation
- Solution for critical section shared between N processes
- Original proposal by Dijkstra did not mandate any wakeup order
    - Later debated from operating system point of view
    - „Bottom layer should not bother with macroscopic considerations"

Example: Binary Semaphore



```
"begin  integer free; free:= 1;
      parbegin
      process 1: begin...............end;
      process 2: begin...............end;
        .
        .
        .
      process N: begin...............end;
      parend
 end"

with the i-th process of the form:

"process i: begin
               Li: P(free); critical section i; V(free);
                   remainder of cycle i; goto Li
            end" .
```

Example: General Semaphore



```
"begin integer number of queuing portions, number of empty positions,
                buffer manipulation;
      number of queuing portions:= 0;
      number of empty positions:= N;
      buffer manipulation:= 1;
      parbegin
      producer: begin
              again 1: produce next portion;
                       P(number of empty positions);
                       P(buffer manipulation);
                       add portion to buffer;
                       V(buffer manipulation);
                       V(number of queuing portions); goto again 1  end;
      consumer: begin
              again 2: P(number of queuing portions);
                       P(buffer manipulation);
                       take portion from buffer;
                       V(buffer manipulation);
                       V(number of empty positions);
                       process portion taken; goto again 2 end
      parend
end" .
```

Coroutines

- Conway, Melvin E. (1963).
  "Design of a Separable Transition-Diagram Compiler".
- Generalization of the subroutine concept
  - Explicit language primitive to indicate transfer of control flow
  - Leads to multiple entry points in the routine
- Routines can suspend (yield) and resume in their execution
- Co-routines may always yield new results -> generators
  - Less flexible version of a coroutine, since yield always returns to caller
- Good for concurrent, not for parallel programming
- Foundation for other concurrency concepts
  - Exceptions, iterators, pipes, …
- Implementation demands stack handling and context switch
  - Portable implementations in C are difficult
  - Fiber concept in the operating system is helpful

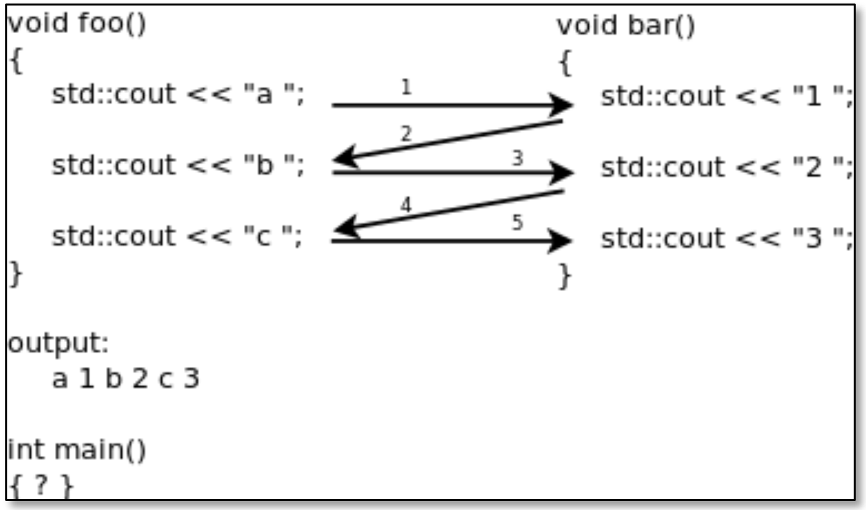
Coroutines

```
var q := new queue
coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume
coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```

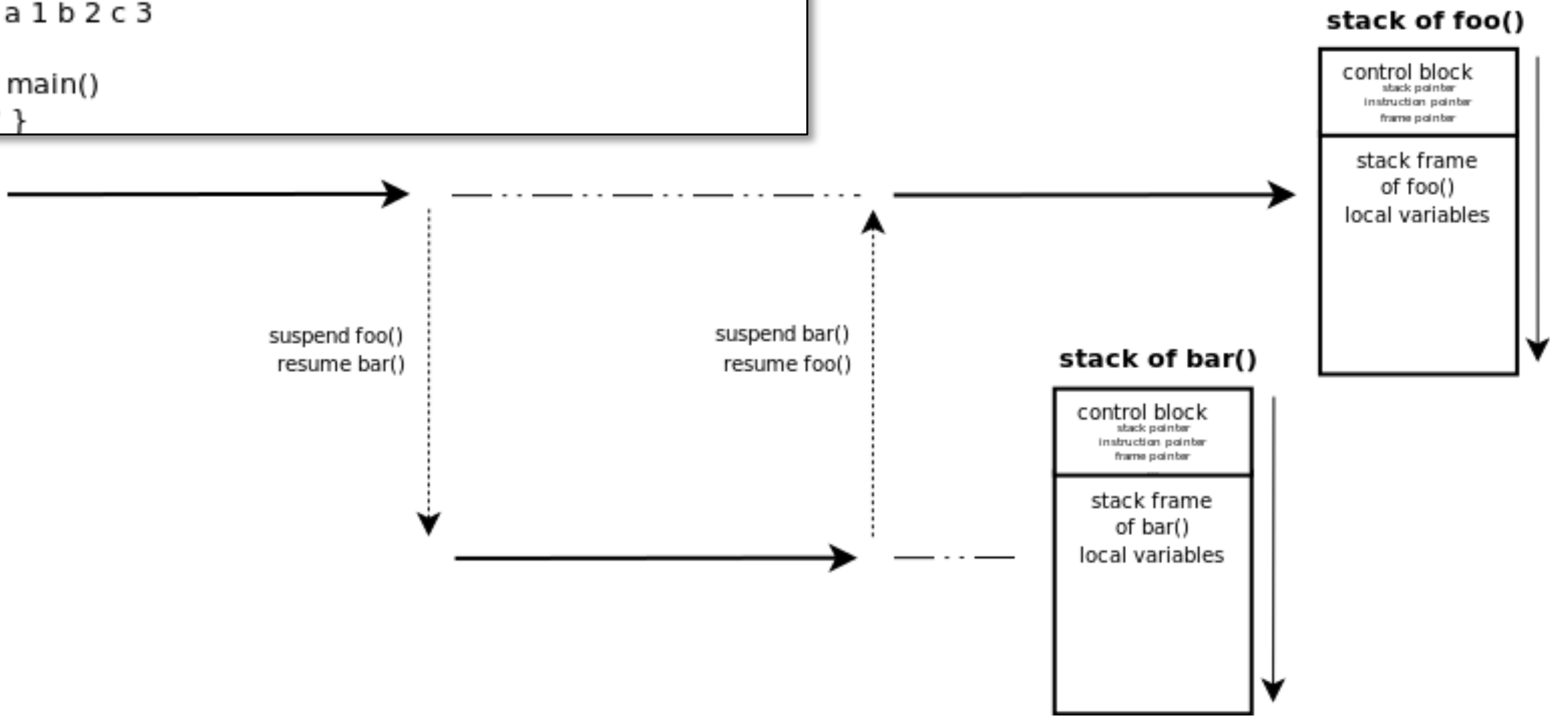
```
def generator():
        n=range(5)
        for i in n:
                yield i

for item in generator():
        print item
```

Coroutines

[boost.org/docs]

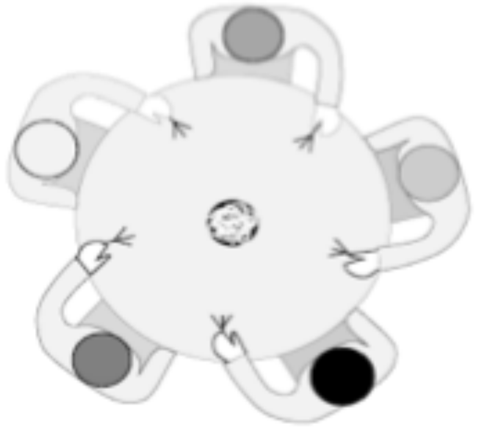


```
void foo()                              void bar()
{                                       {
    std::cout << "a ";                      std::cout << "1 ";
    std::cout << "b ";                      std::cout << "2 ";
    std::cout << "c ";                      std::cout << "3 ";
}                                       }

output:
    a 1 b 2 c 3

int main()
{ ? }
```



Dining Philosophers [Dijkstra]

- Five philosophers work in a college, each philosopher has a room for thinking
- Common dining room, furnished with a circular table, surrounded by five labeled chairs
- In the center stood a large bowl of spaghetti, which was constantly replenished
- When a philosopher gets hungry:
  - Sits on his chair
  - Picks up his own fork on the left and plunges it in the spaghetti, then picks up the right fork
  - When finished he put down both forks and gets up
  - May wait for the availability of the second fork

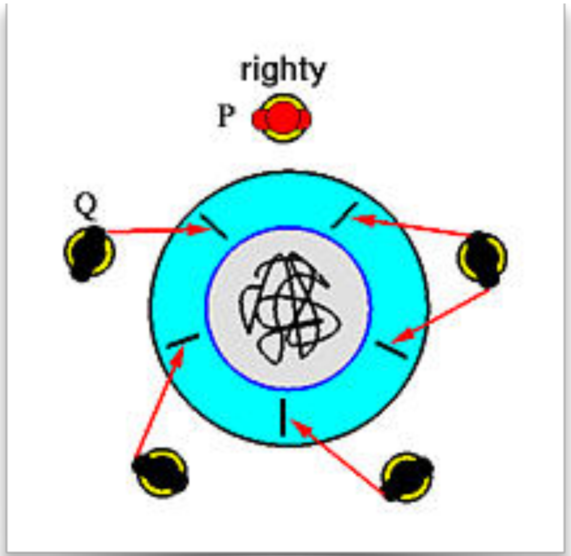


Dining Philosophers [Dijkstra]

- Idea: Shared memory synchronization has different standard issues
- Philosophers as tasks, forks as shared resource
- Explanation of the *deadly embrace (deadlock)* and *starvation*
- How can a deadlock happen ?
  - All pick the left fork first and wait for the right
- How can a live-lock (starvation) happen ?
  - Two fast eaters, sitting in front of each other
- Ideas for solutions
  - Waiter solution (central arbitration)
  - Lefty-righty approach

One Solution: Lefty-Righty-Approach

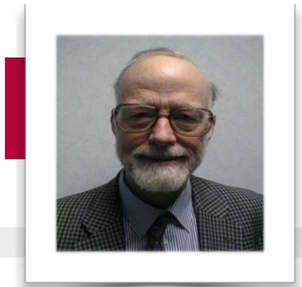
- PHIL$_n$ is a righty (is the only one starting with the right fork)
  - Case 1: Has right fork, but left fork is held by left neighbor
    - Left neighbor will put down both forks when finished, so there is a chance
    - PHIL$_n$ might always be interrupted before eating (starvation), but no deadlock of all participants occurs
  - Case 2: Has no fork
    - Right fork is captured by right neighbor
    - In worst case, lock spreads to all but one righty
  - ...
- Proof by Dijkstra shows deadlock freedom, but still starvation problem



Coffman Conditions

- *1970. E.G. Coffman and A. Shoshani.*
  Sequencing tasks in multiprocess systems to avoid deadlocks.
  - All conditions must be fulfilled to allow a deadlock to happen
  - **Mutual exclusion condition** - Individual resources are available or held by no more than one thread at a time
  - **Hold and wait condition** – Threads already holding resources may attempt to hold new resources
  - **No preemption condition** – Once a thread holds a resource, it must voluntarily release it on its own
  - **Circular wait condition** – Possible for a thread to wait for a resource held by the next thread in the chain
- Avoiding circular wait turned out to be the easiest solution for deadlock avoidance
- Avoiding mutual exclusion leads to **non-blocking synchronization**
  - These algorithms no longer have a critical section

Monitors

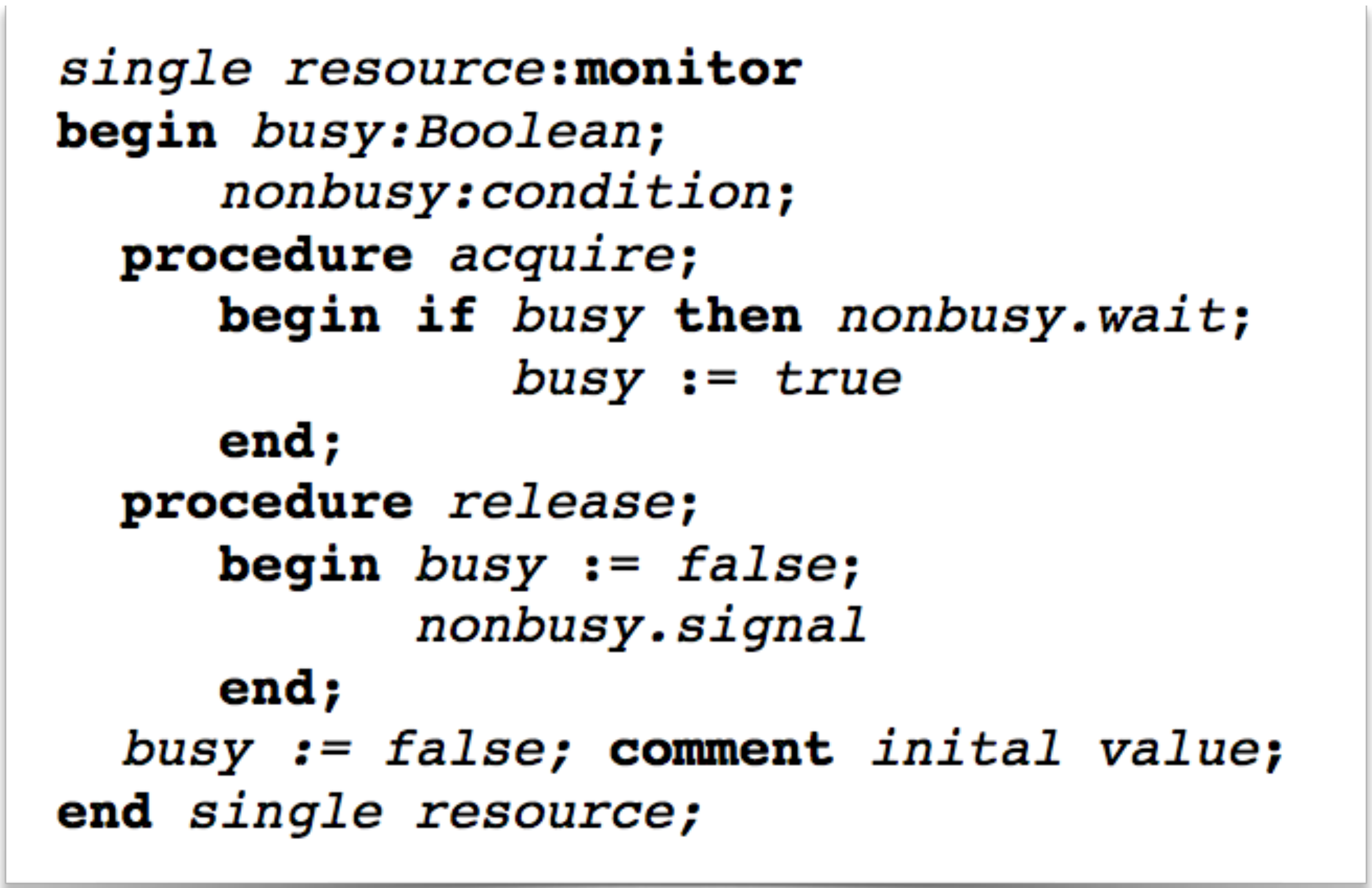


- *1974, Monitors: An Operating System Structuring Concept, C.A.R. Hoare*
  - First formal description of monitor concept, originally invented by Brinch Hansen in 1972 as part of an OS project
  - Operating system has to schedule requests for various resources, separate schedulers per resource necessary
  - Each contains local administrative data, and functions used by requestors
  - Collection of associated data and functionality: **monitor**
    - Note: The paper mentions Simula 67 classes (1972)
    - Functions are the same for all instances, but invocations should be mutually exclusive
    - Function execution is the **occupation of the monitor**
    - Easily implementable with semaphores

Condition Variables

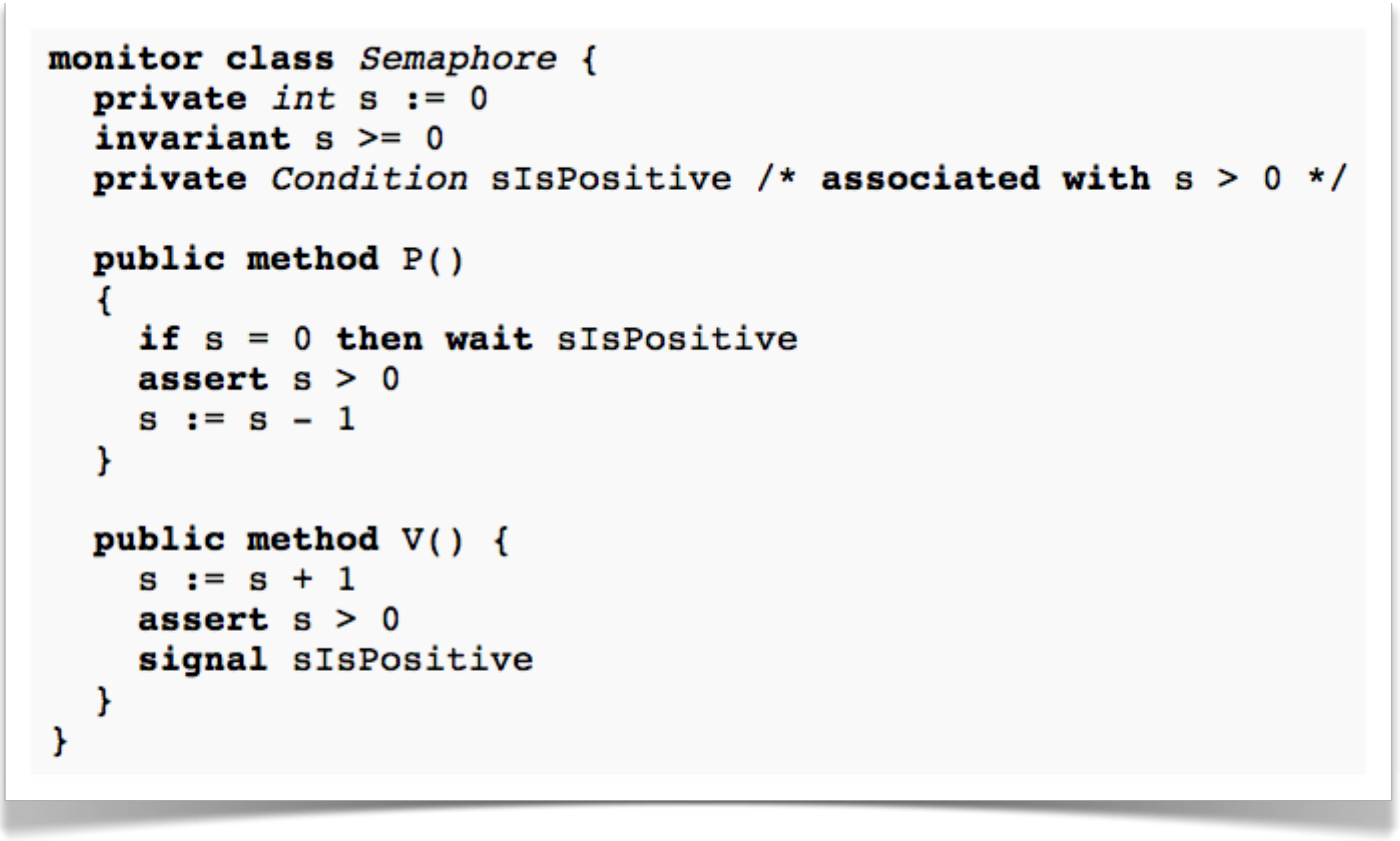
- Function implementation itself might need to wait at some point
  - **Monitor wait()** operation: Issued inside the monitor, causes the caller to wait and temporarily release the monitor while waiting for some assertion
  - **Monitor signal()** operation:
    Resume one of the waiting callers
- Might be more than one reason for waiting inside the function
  - Variable of type **condition** in the monitor, one for each waiting reason
  - Delay operations relate to some specific condition variable: *condvar.wait()*, *condvar.signal()*
  - Programs are signaled for the condition they are waiting for
  - Hidden implementation as queue of waiting processes

Single Resource Monitor



```
single resource:monitor
begin busy:Boolean;
      nonbusy:condition;
  procedure acquire;
      begin if busy then nonbusy.wait;
               busy := true
      end;
  procedure release;
      begin busy := false;
            nonbusy.signal
      end;
  busy := false; comment inital value;
end single resource;
```

Implementing a Semaphore with a Monitor



```
monitor class Semaphore {
  private int s := 0
  invariant s >= 0
  private Condition sIsPositive /* associated with s > 0 */

  public method P()
  {
    if s = 0 then wait sIsPositive
    assert s > 0
    s := s - 1
  }

  public method V() {
    s := s + 1
    assert s > 0
    signal sIsPositive
  }
}
```

Monitors - Example

- Monitors are part of the Java programming language
- Each class can be as monitor
  - Mutual exclusion of method calls by *synchronized* keyword
  - *Object* base class provides condition variable functionality – *Object.wait()*, *Object.notify(),* and a wait queue
  - Both functions are only callable from *synchronized* methods (otherwise *IllegalMonitorStateException*)
  - Monitor code can use arbitrary objects as condition variables
- At runtime
  - By calling *XXX.wait()*, a thread gives up ownership of the monitor and blocks in the call
  - Monitor is also given up by leaving the *synchronized* method
  - Other threads call *XXX.notify()* to signal waiters, but still must give up the ownership of the monitor

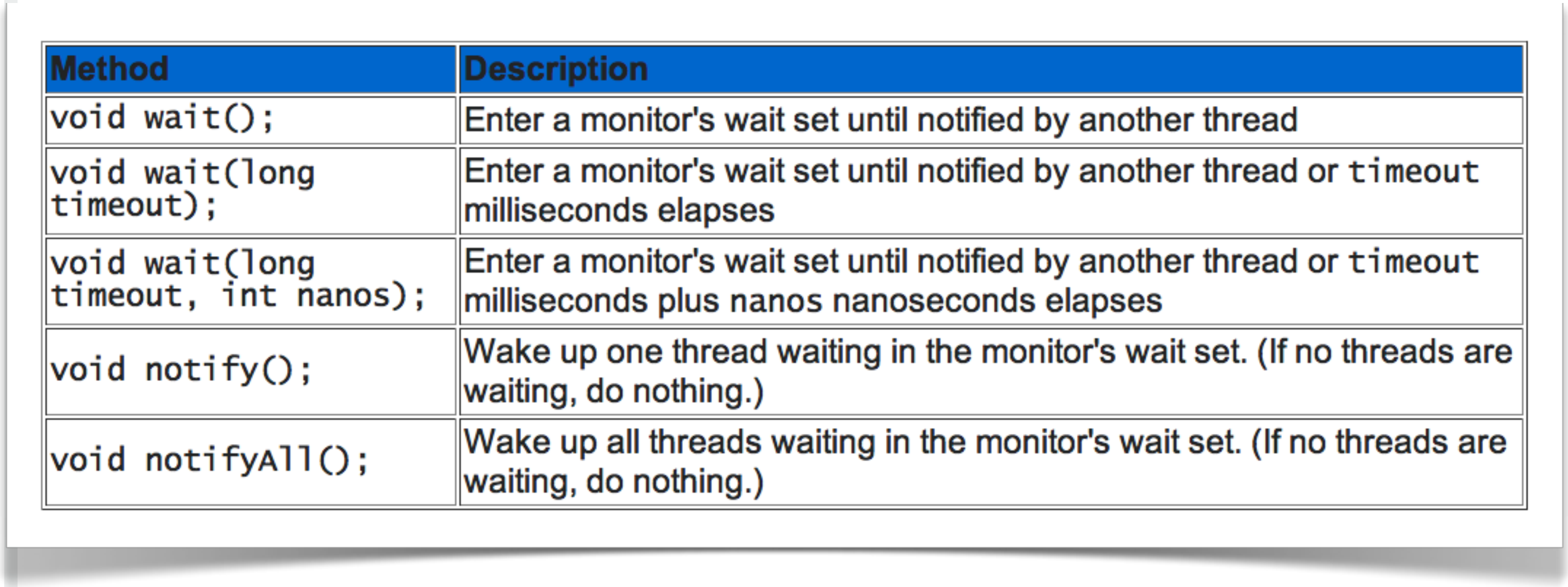
Java Example

```
class Queue {
  int n;
  boolean valueSet = false;
  synchronized int get() {
    while(!valueSet)
      try { this.wait(); }
      catch(InterruptedException e) { ... }
    valueSet = false;
    this.notify();
    return n;
  }
  synchronized void put(int n) {
    while(valueSet)
      try { this.wait(); }
      catch(InterruptedException e) { ... }
    this.n = n;
    valueSet = true;
    this.notify();
  }
}
```

```
class Producer implements Runnable {
  Queue q;
  Producer(Queue q) {
    this.q = q;
    new Thread(this, "Producer").start(); }
  public void run() {
    int i = 0;
    while(true) { q.put(i++); }
}}

class Consumer implements Runnable { ... }

class App {
  public static void main(String args[]) {
    Queue q = new Q();
    new Producer(q);
    new Consumer(q);
  }
}
```

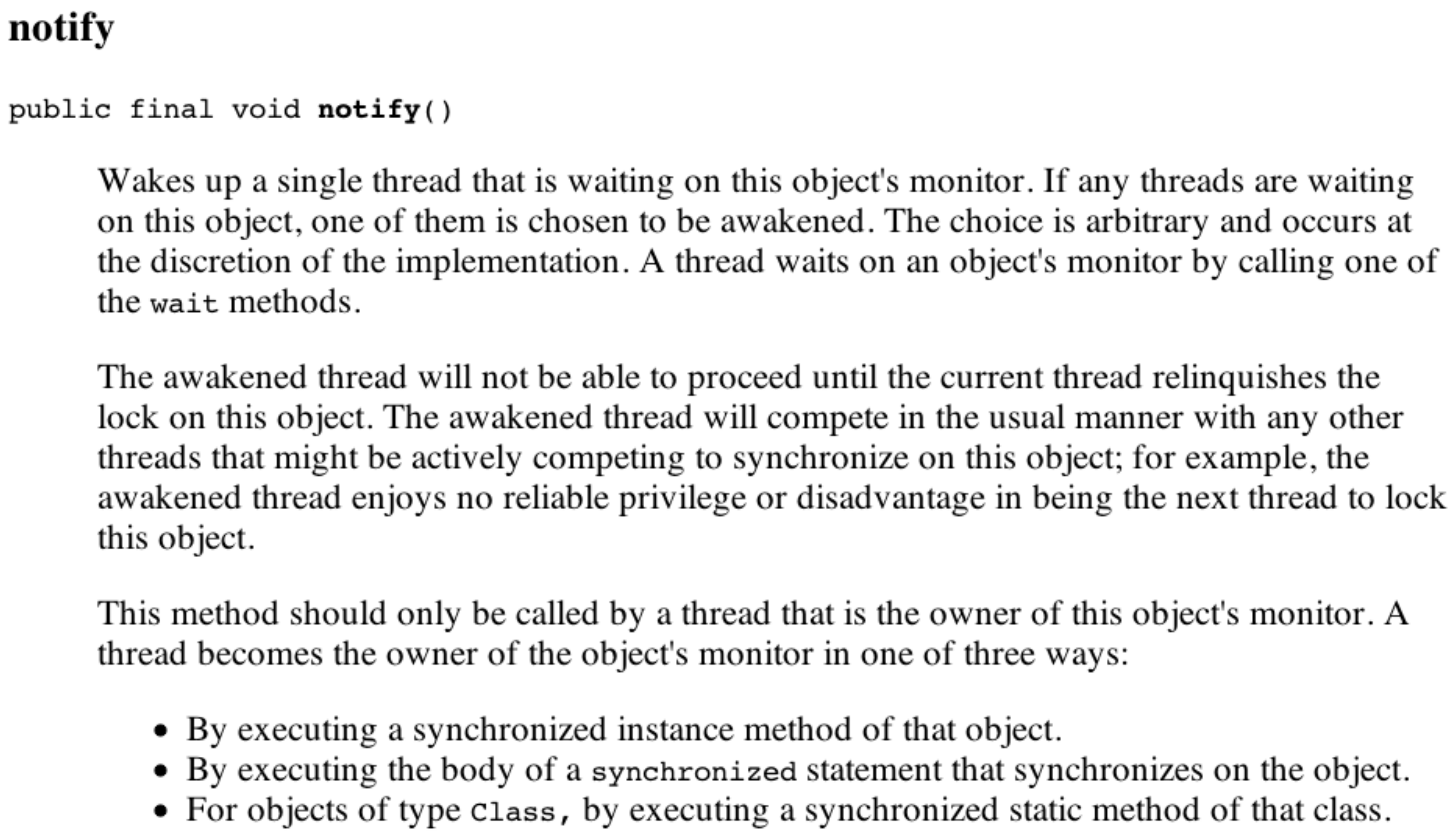
Monitors - Java

- Since the operating system gives boost for threads being waked up, the signaled thread is likely to be scheduled as next
- Also adopted in other languages



Method	Description
`void wait();`	Enter a monitor's wait set until notified by another thread
`void wait(long timeout);`	Enter a monitor's wait set until notified by another thread or `timeout` milliseconds elapses
`void wait(long timeout, int nanos);`	Enter a monitor's wait set until notified by another thread or `timeout` milliseconds plus nanos nanoseconds elapses
`void notify();`	Wake up one thread waiting in the monitor's wait set. (If no threads are waiting, do nothing.)
`void notifyAll();`	Wake up all threads waiting in the monitor's wait set. (If no threads are waiting, do nothing.)

Notify Semantics



notify

```
public final void notify()
```

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the `wait` methods.

The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:

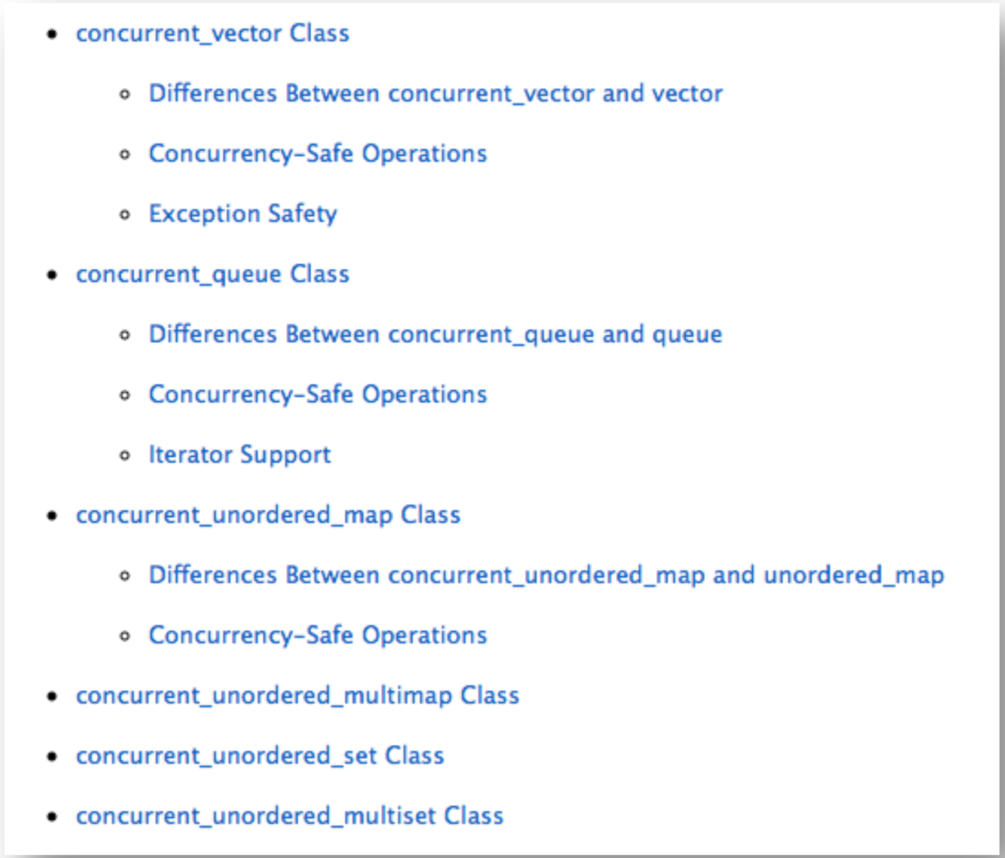
- By executing a synchronized instance method of that object.
- By executing the body of a `synchronized` statement that synchronizes on the object.
- For objects of type `Class,` by executing a synchronized static method of that class.

High-Level Primitives

- Today: Multitude of high-level synchronization primitives
- **Spinlock**
  - Perform busy waiting, lowest overhead for short locks
- **Reader / Writer Lock**
  - Special case of mutual exclusion through semaphores
  - Multiple „Reader“ processes can enter the critical section at the same time, but „Writer“ process should gain exclusive access
  - Different optimizations possible:
    minimum reader delay, minimum writer delay, throughput, …
- **Mutex**
  - Semaphore that works amongst operating system processes
- **Concurrent Collections**
  - Blocking queues and key-value maps with concurrency support

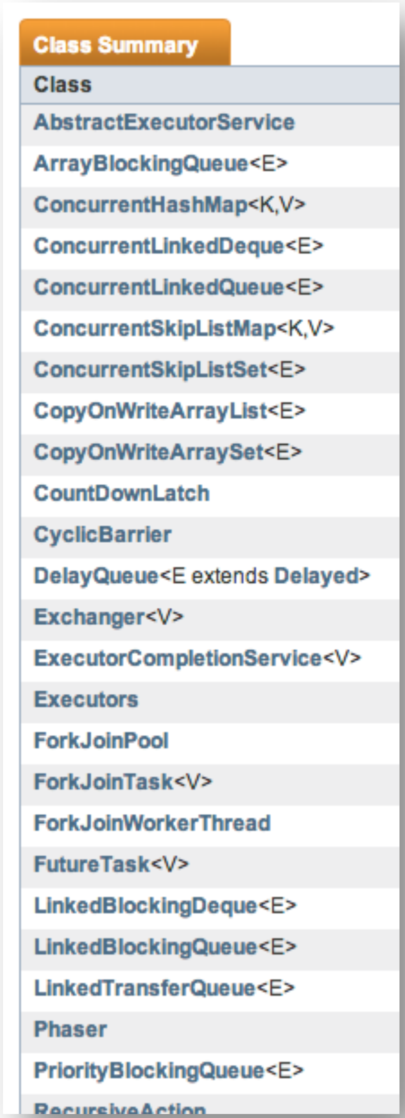
Conccurent Collections

Microsoft Parallel Patterns Library



- concurrent_vector Class
  - Differences Between concurrent_vector and vector
  - Concurrency-Safe Operations
  - Exception Safety
- concurrent_queue Class
  - Differences Between concurrent_queue and queue
  - Concurrency-Safe Operations
  - Iterator Support
- concurrent_unordered_map Class
  - Differences Between concurrent_unordered_map and unordered_map
  - Concurrency-Safe Operations
- concurrent_unordered_multimap Class
- concurrent_unordered_set Class
- concurrent_unordered_multiset Class

Java 7 – java.util.concurrent

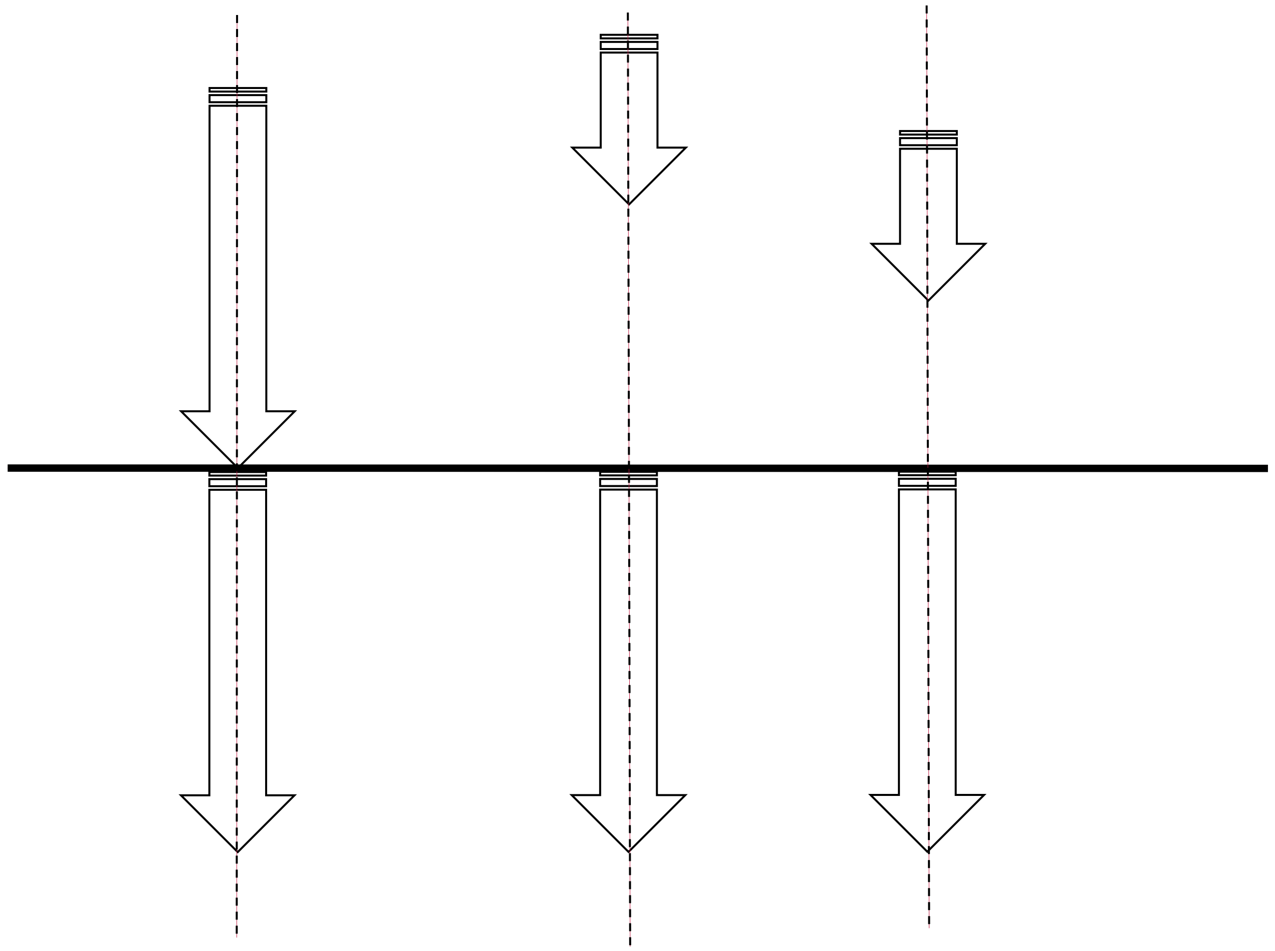


Class Summary
Class
AbstractExecutorService
ArrayBlockingQueue<E>
ConcurrentHashMap<K,V>
ConcurrentLinkedDeque<E>
ConcurrentLinkedQueue<E>
ConcurrentSkipListMap<K,V>
ConcurrentSkipListSet<E>
CopyOnWriteArrayList<E>
CopyOnWriteArraySet<E>
CountDownLatch
CyclicBarrier
DelayQueue<E extends Delayed>
Exchanger<V>
ExecutorCompletionService<V>
Executors
ForkJoinPool
ForkJoinTask<V>
ForkJoinWorkerThread
FutureTask<V>
LinkedBlockingDeque<E>
LinkedBlockingQueue<E>
LinkedTransferQueue<E>
Phaser
PriorityBlockingQueue<E>
RecursiveAction

High-Level Primitives

- **Reentrant Lock**
  - Lock can be obtained several times without locking on itself
  - Useful for cyclic algorithms (e.g. graph traversal) and problems were lock bookkeeping is very expensive
  - Reentrant mutex needs to remember the locking thread(s), which increases the overhead
- **Barriers**
  - All concurrent activities stop there and continue together
  - Participants statically defined at compile- or start-time
  - Newer **dynamic barrier** concept allows late binding of participants (e.g. X10 clocks, Java phasers)
  - **Memory barrier** or **memory fence** enforce separation of memory operations before and after the barrier
    - Needed for low-level synchronization implementation

Barrier



Lock-Free Programming

- Lock-free programming as a way of sharing data without maintaining locks
  - Prevents deadlock and live-lock conditions
  - Goal:
    Suspension of one thread never prevents another thread from making progress (e.g. synchronized shared queue)
  - Blocking by design does not disqualify the lock-free realization
- Algorithms rely on hardware support for atomic operations
  - *Read-Modify-Write (RMW)* operations
  - *Compare-And-Swap (CAS)* operations
- These operations are typically mapped in operating system API

Lock-Free Programming

```
void LockFreeQueue::push(Node* newHead)
{
    for (;;)
    {
        // Copy a shared variable (m_Head) to a local.
        Node* oldHead = m_Head;
        // Do some speculative work, not yet visible to other threads.
        newHead->next = oldHead;
        // Next, attempt to publish our changes to the shared variable.
        // If the shared variable hasn't changed, the CAS succeeds and we return.
        // Otherwise, repeat.
        if (_InterlockedCompareExchange(&m_Head, newHead, oldHead) == oldHead)
            return;
    }
}
```

Sequential Consistency

- Consistency model where the order of memory operations is consistent with the source code
  - Important for lock-free algorithm semantic
  - Not guaranteed by some processor architectures (e.g. PowerPC)
- Java and C++ support the enforcement of sequential consistency
  - Compiler generates additional *memory fences* and *RMW* operations
  - Still does not prevent from memory re-ordering due to instruction re-ordering by the compiler itself

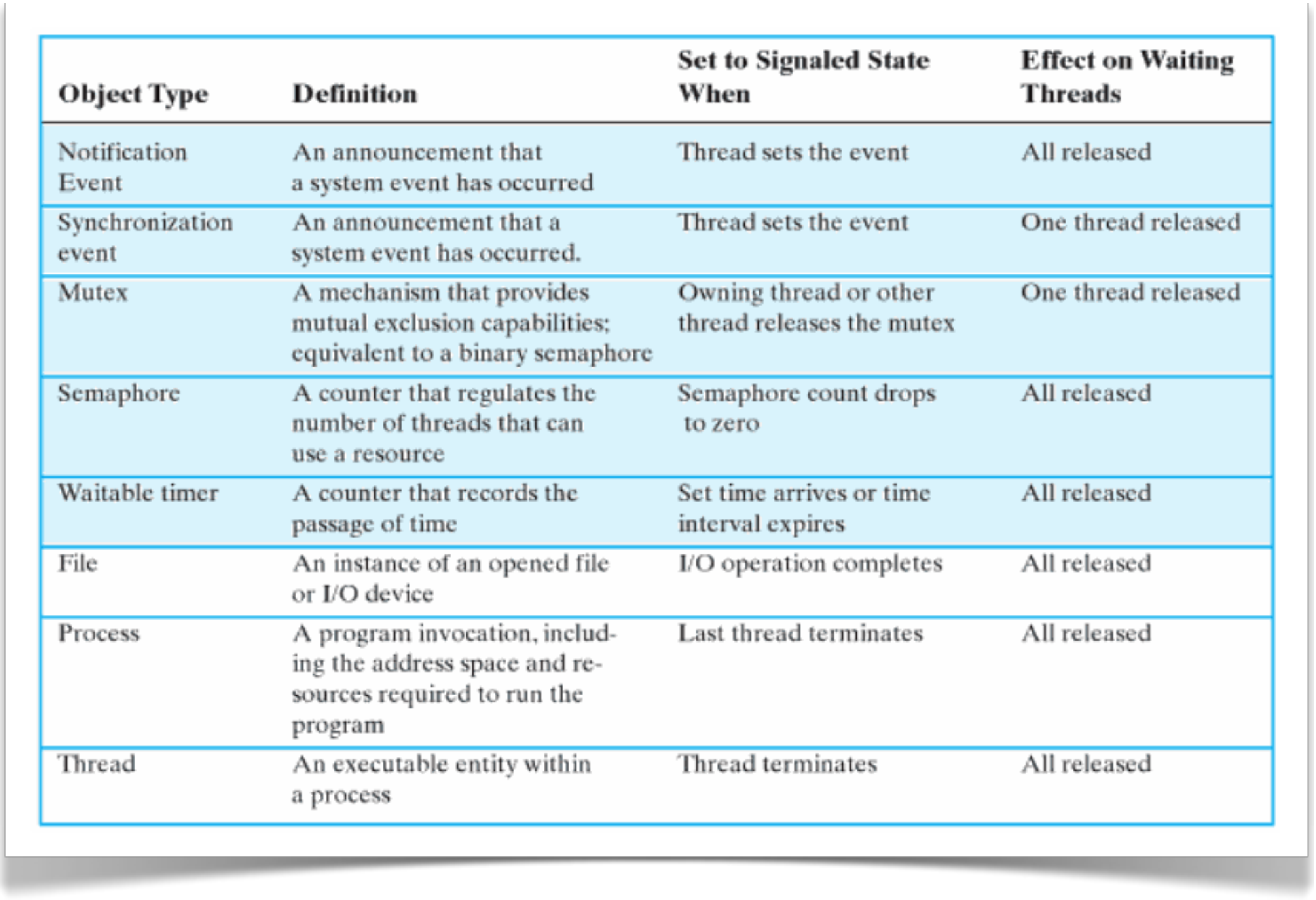
```
std::atomic<int> X(0), Y(0);
int r1, r2; void thread1()
{
    X.store(1);
    r1 = Y.load();
} void thread2()
{
    Y.store(1);
    r2 = X.load();
}
```

r1 and r2 never become zero at the same time

Example: Modern Operating Systems

- Mutual exclusion of access necessary whenever a resource …
  - ... does not support shared access by itself
  - ... sharing could lead to unpredictable outcome
- Examples: Memory locations, stateful devices
- Traditional OS architecture approaches
  - Disable all interrupts before entering a critical section
  - Mask interrupts that have handlers accessing the same resource (e.g. Windows dispatcher database)
  - Both do not work for true SMP systems
- Hardware-supported synchronization primitives are now part of every modern OS kernel – semaphores, spinlocks, …
- Also exported to user space, since it faces the same problem

Windows Synchronization Objects [Stallings]



Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification Event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

Example: Modern Operating Systems

- Linux
  - Kernel disables interrupts for synchronizing access to global data on uniprocessor systems
  - Uses spin-locks for multiprocessor synchronization
  - Uses semaphores and readers-writers locks when longer sections of code need access to data
  - Implements POSIX synchronization primitives to support multitasking, multithreading (including real-time threads), and multiprocessing.

8 Simple Rules For Concurrency [Breshears]

- *„Concurrency is still more art than science"*
  - Identify truly independent computations
  - Implement concurrency at the highest level possible
  - Plan early for scalability
  - Code re-use through libraries
  - Use the right threading model
  - Never assume a particular order of execution
  - Use thread-local storage if possible, apply locks to specific data
  - Don't change the algorithm for better concurrency

Example: Eve Online

- MMORPG space game, client applications, server cluster
- Single shard server instance
- 120.000 active players, > 24.000 concurrent users
- Relies on Stackless Python
    - Thread of execution is a **tasklet** (no preemption, no OS thread)
    - **Channels** as task rendevous and context switch point
    - No C stack per tasklet, on small Python datastructure
    - Leads to support for **co-operative multitasking**
    - Almost no race condition