Parallel Programming Concepts

Theory of Concurrency - Shared Memory

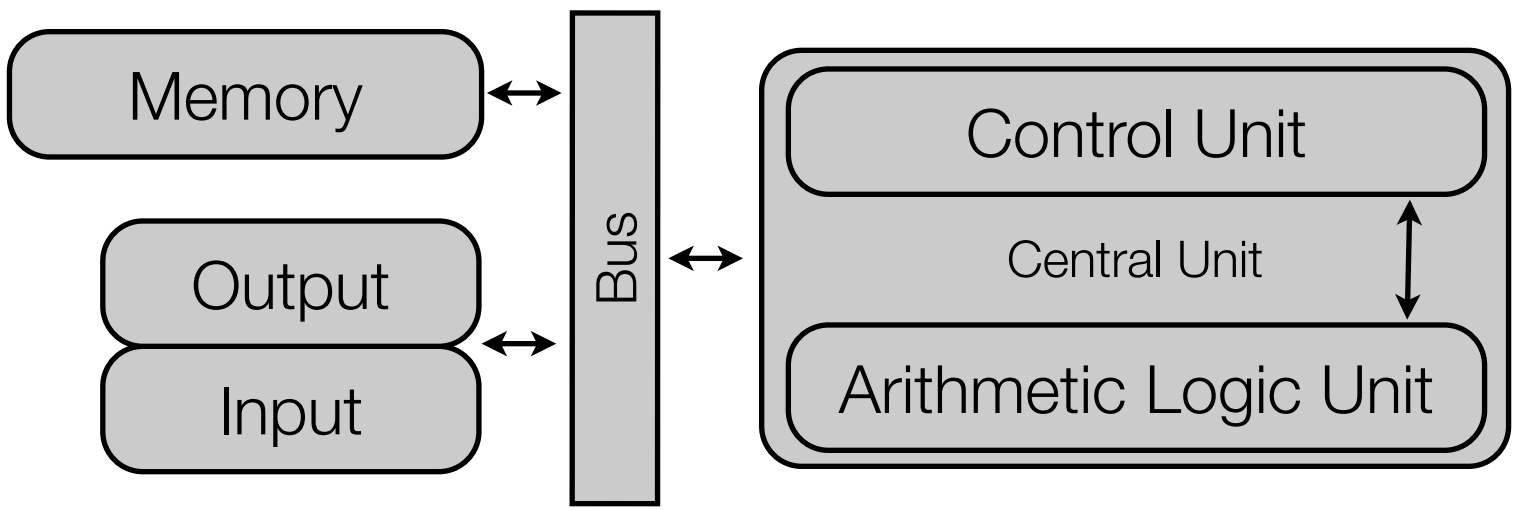
Peter Tröger

Sources:
Clay Breshears: The Art of Concurrency, Chapter 3
Dijkstra, Edsger W.: Cooperating sequential processes. / Hierarchical ordering of sequential processes
C.A.R. Hoare: Monitors - An Operating System Structuring Concept

„An ounce of prevention equals a pound of cure.“

Von Neumann Model

- Processor executes a sequence of instructions, which specify
  - Arithmetic operation
  - 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



Terminology

- **Concurrency**
  - Supported to have two or more actions *in progress* at the same time
  - Classical operating system responsibility
    (resource sharing for better utilization of CPU, memory, network, ...)
  - Demands **scheduling** and **synchronization** - interference control
- **Parallelism**
  - Supported to have two or more actions executing *simultaneously*
  - Demands **parallel hardware**, **concurrency support**, (and **communication**)
  - Programming model relates to chosen hardware / communication approach
- Examples: Windows 3.1, threads, signal handlers, shared memory

History

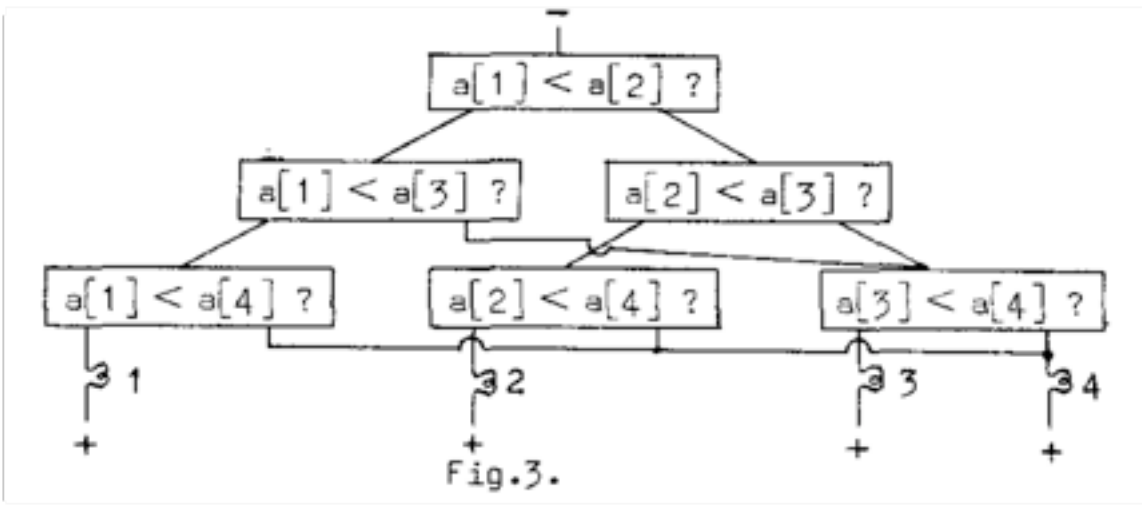
- *1961, Atlas Computer, Kilburn & Howarth*
  - Based on Germanium transistors, assembler only
  - First use of interrupts to simulate concurrent execution of multiple programs - *multiprogramming*
- 60‘s and 70‘s: Foundations for concurrent software developed (operating system reliability)
- *1965,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

History

- *1971, Towards a Theory of Parallel Programming, C. A. R. Hoare*
  - First notable attempt to extend programming languages with a first-class concept of parallelism
  - Design principles for parallel programming languages
    - Time-related interference control at compile time
    - Disjoint processes without common variables
    - Explicit declaration of shared resources and critical regions
    - Conditional critical regions („with *resource* when *expression* do *critical region*“)
  - Novel approach at a time were everything was related to variable checking
  - Mostly known for CSP / Occam work (next lecture)

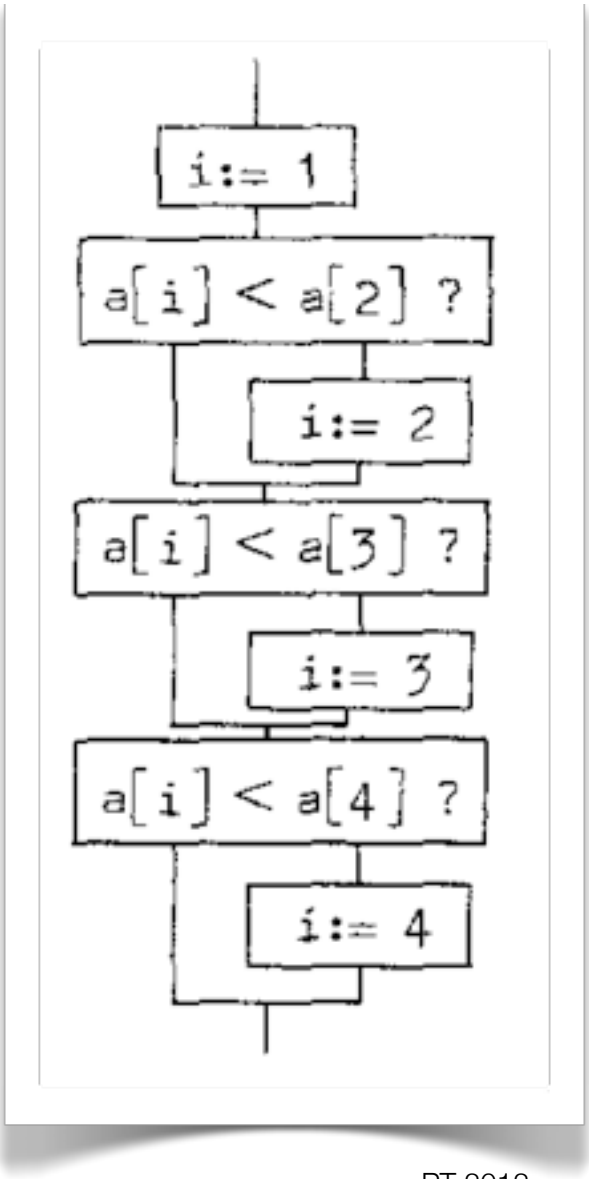
Cooperating Sequential Processes [Dijkstra]

- *1965, Cooperating Sequential Processes, E.W.Dijkstra*
  - Paper starts with comparison of sequential and non-sequential machine
  - Example: Electromagnetic solution to find a **largest** value in an array
    - Current lead though magnet coil
    - Switch to magnet with larger current



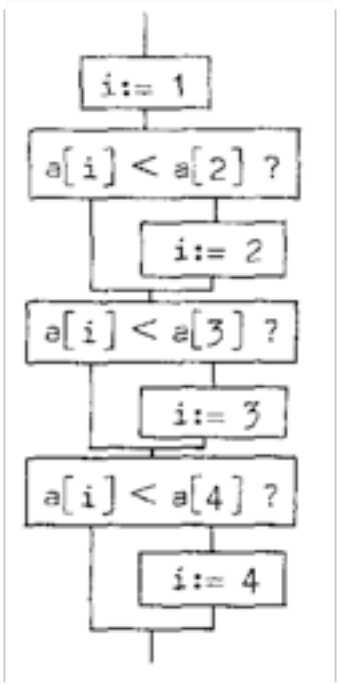
Cooperating Sequential Processes [Dijkstra]

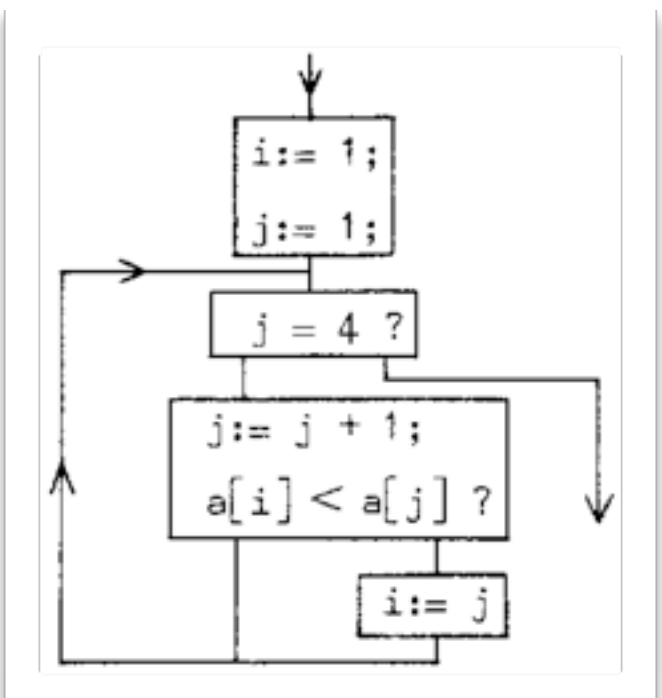
- **Progress of time** is relevant for this machine
  - After applying the currents in a step, machine needs some time to show the result
- Interpretation of first approach
  - Same line differs only in left operand
  - Concept of a parameter that comes from history -> *variable*
- Different setup for same behavior - six simultaneous comparators (last slide) vs. three comparisons evaluated in sequence
- Rules of behavior form a *program*



Cooperating Sequential Processes [Dijkstra]

- Idea: Multiple ways of expressing the program intent
  - Example: Consider repetitive nature of the problem
  - Invest in a variable j -> generalize the solution for any number of items





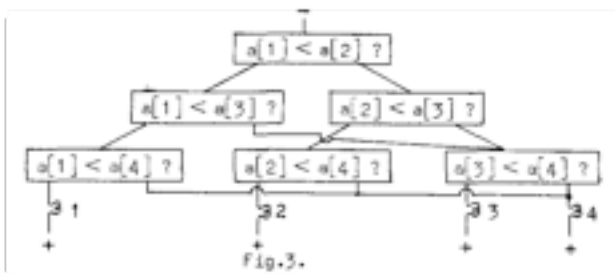


Fig.3.

```
"       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 / *processes*
- How about the cooperation between loosely coupled sequential processes ?
  - Beside rare communication moments, 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 not fulfilled, might bring „analogue interferences“ on verification attempt
  - Interleaved instructions result in a different result then the single versions
- Note: Dijkstra already identified the concept of a **„race condition“** here

Race Condition

```
void echo() {
  char_in = getchar();
  char_out = char_in;
  putchar(char_out);
}
```

- Executed by two threads on uniprocessor
- Executed by two threads on multiprocessor
- What happens ?

Cooperating Sequential Processes [Dijkstra]

- Idea of a **critical section**
  - Two cyclic sequential processes
  - At any moment, at most one process is engaged in its critical section
  - Use common variables with atomic read / write behavior
- First approach
  - Too restrictive solution

```
"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" .
```

- Note: Atomicity concept on the scope of source code lines

Cooperating Sequential Processes [Dijkstra]

- Second approach
  - Separate indicators for entering / leaving the critical section
  - More fine-grained waiting approach
  - Too optimistic solution, might violate critical section property

```
"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]

- Third approach
  - First ‚raise the flag‘, the check for the other flag
  - Mutual exclusion guaranteed
    - If c1=0, then c2=1, and vice versa
    - Variables only change outside of the critical section
  - 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]

- Fourth approach
  - Reset locking of critical section if the other one is already in
- Problem due to assumption of relative speed
  - Maybe no parallel hardware - only concurrent execution
  - Can lead for one process to ‚wait forever' without any progress

```
"begin  integer  c1, c2;
        c1:= 1; c2:= 1;
        cobegin
        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
        coend
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, otherwise wait for 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
```

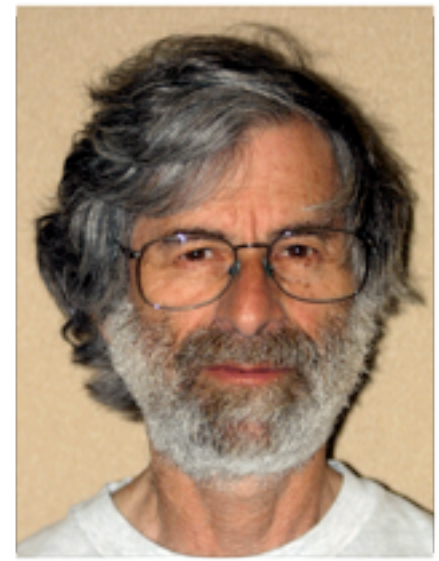
```
        parend
end"    .
```

Mutual Exclusion Today

- Dekker provided first correct solution only based on shared memory
- Guarantees three major properties
  - Mutual exclusion
  - Freedom from deadlock
  - Freedom from starvation
- Generalization for n processes by Lamport with the **Bakery algorithm**
  - Relies only on memory access atomicity

Bakery Algorithm [Lamport]

```
do {
  choosing[i] = 1;
  number[i] = max(number[0],number[1] ...,number[n-1]) + 1;
  choosing[i] = 0;
  for (j = 0; j < n; j++) {
     while (choosing[j] == 1) ;
     while ((number[j] != 0) &&
      ((number[j],j) ‘’<‘’ (number[i],i)));
  }
  critical section
  number[i] = 0;
  remainder section
} while (1);
```



Critical Section

- n threads all competing to use a shared resource (i.e.; shared data, spaghetti forks)
- Each thread has some code - **critical section** - in which the shared data is accessed
- **Mutual Exclusion** demand
  - Only one thread at a time is allowed into its critical section, among all threads that have critical sections for the same resource.
- **Progress** demand
  - 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** demand
  - 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)

Test-and-Set

- Reality today is much worse:
  Out-of-order execution, re-ordered memory access, compiler optimizations
- **Test-and-set** processor hardware feature, wrapped by the operating system
  - Write to a memory location and return its old value as atomic operation
  - 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
  - Can be implemented with atomic swap (or any other *read-modify-write*) hardware operation
  - Typical example for a **spin-lock** approach
    - Busy waiting for acquiring a lock
    - Efficient for short periods, since no overhead of context switching

```
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;
 }
```

Abstraction of Concurrency [Breshears]

- Programs are the execution of atomic statements
  - „Atomic“ can be defined on different granularity levels, e.g. source code line

    -> Concurrency should be treated as abstract concept
- Concurrent execution is the interleaving of atomic statements from multiple sequential processes
  - Scheduling is (typically) non-deterministic
  - Unpredictable execution sequence of atomic instructions
- Concurrent algorithm should maintain properties for all possible inter-leavings
  - Example: All atomic statements are eventually included (fairness)

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"

is a less expensive activity than this busy way of waiting, then we are fully justified (now also from an economic point of view) to call our solution misleading.

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
    - Blocks if the semaphore is already zero - **„wait“** operation
  - **V**-operation: Increase value of its argument semaphore by 1 as atomic step
    - **„signal“** operation
- Solution for critical section shared between N processes
- Original proposal by Dijkstra did not mandate any wakeup order, only *progress*
  - Later debated from operating system implementation 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" .
```

Deadlock ?

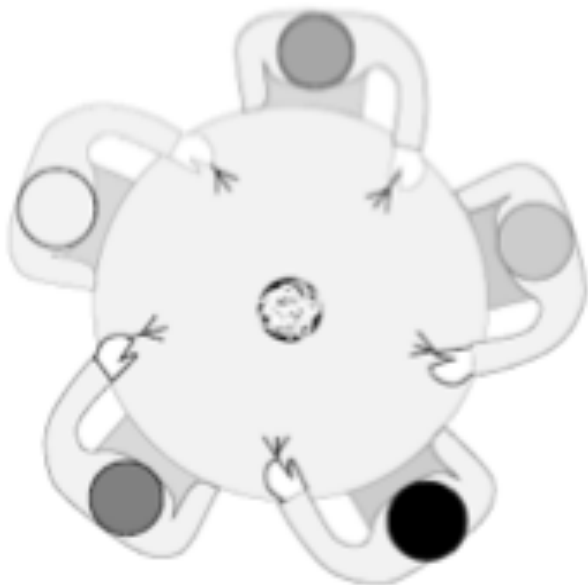
- *1970. E.G. Coffman and A. Shoshani.*
  Sequencing tasks in multiprocess systems to avoid deadlocks.

  - **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
  - All conditions must be fulfilled to allow a deadlock to happen

```
"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" .
```

Dining Philosophers (E.W.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



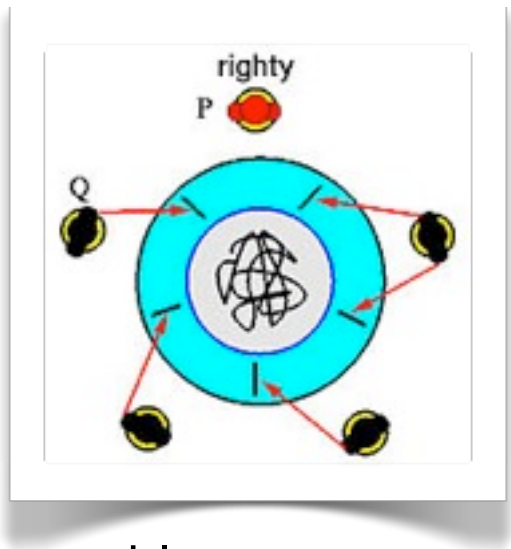
Dining Philosophers (E.W.Dijkstra)

- Idea: Shared memory synchronization has different standard issues
- Explanation of the *deadly embrace (deadlock)* and the *starvation problem*
  - Forks taken one after the other, released together
    - No two neighbors may eat at the same time
- Philosophers as tasks, forks as shared resource
  - 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
- One possibility: Waiter solution (central arbitration)

(C) Wikipedia

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



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 operating system 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 procedures: **monitor**
    - Note: The paper mentions the class concept from Simula 67 (1972)
  - Procedures are common between all instances, but calls should be mutually exclusive (local state + resource state) - **occupation of the monitor**

Monitors and Condition Variables

- Simple implementation of method protection by semaphores
- Method 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: Resumes one of the waiting callers
- Might be more than one reason for waiting inside the monitor
  - Variable of type **condition** in the monitor, one for each wait reason
  - Delay operations relate to condition variable: *condvar.wait*, *condvar.signal*
  - Programs expect to be 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

- Java programming language
  - Each class might act as monitor, mutual exclusion of method calls by *synchronized* keyword
  - One single wait queue per object, no need for extra condition variables
    - Each Java object can act as condition variable - *Object.wait()* and *Object.notify()*
    - Threads give up monitor ownership and blocks by calling *wait()*
    - Threads also give up ownership by leaving the *synchronized* method
    - Threads calling *notify()* are still continued, so data still might change until they ultimately give up the ownership -> signaling acts only as ‚hint' to the waiting thread
- Coordination functions in *Object* only callable from *synchronized* methods

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.)

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);
  }
}
```

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.

Reader / Writer Locks

- Today: Multitude of high-level synchronization primitives,
  based on initial mutual exclusion and critical section concepts
- Example: *1971, Concurrent Control with „Readers“ and „Writers“. Courtois et al.*
  - Special case of mutual exclusion through semaphores
    - Multiple „reader“ processes can enter the critical section at the same time
    - „writer“ process should gain exclusive access
  - Different optimizations: minimum reader delay, minimum writer delay
  - Problem 1: No reader should wait for a writer that waits for a reader
  - Problem 2: Fast write when ready

Example: Modern Operating Systems

- Mutual exclusion of access necessary whenever the resource ...
  - ... does not support shared access by itself
  - ... sharing could lead to unpredictable outcome
- Examples: Memory locations, stateful devices
- Code sections accessing the non-sharable resource form a *critical section*
- 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

Modern Operating Systems

- User-mode software has the same problem
  - Also needs reliable multi-processor synchronization
  - Spin locks not appropriate - kernel needs to provide synchronization primitives that put the waiting thread to sleep
- Windows NT kernel:
  - Spin-locks protecting global data structures in the kernel (e.g. DPC queue)
  - User-mode synchronization primitives mapped to kernel-level *Dispatcher Object*
    - Can be in *signaled* or *non-signaled* state
    - *WaitForSingleObject(), WaitForMultipleObjects()*

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

Windows Synchronization Functions

- Condition variables and reader/writer lock function
  - AcquireSRWLockExclusive, AcquireSRWLockShared, InitializeConditionVariable, InitializeSRWLock, ReleaseSRWLockExclusive, ReleaseSRWLockShared, SleepConditionVariableCS, SleepConditionVariableSRW, TryAcquireSRWLockExclusive, TryAcquireSRWLockShared, WakeAllConditionVariable, WakeConditionVariable
- Critical section functions
  - DeleteCriticalSection, EnterCriticalSection, InitializeCriticalSection, InitializeCriticalSectionAndSpinCount, InitializeCriticalSectionEx, LeaveCriticalSection, SetCriticalSectionSpinCount, TryEnterCriticalSection
- Event functions

  CreateEvent, CreateEventEx, OpenEvent, PulseEvent, ResetEvent, SetEvent
- One-time initialization functions

  InitOnceBeginInitialize, InitOnceComplete, InitOnceExecuteOnce, InitOnceInitialize

Windows Synchronization Functions

- Interlocked functions
- Mutex functions
- Semaphore functions
- Linked list, timer queue, waitable timers ...
- Wait functions

`MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, RegisterWaitForSingleObject, SignalObjectAndWait, UnregisterWait, UnregisterWaitEx, WaitForMultipleObjects, WaitForMultipleObjectsEx, WaitForSingleObject, WaitForSingleObjectEx, WaitOrTimerCallback`

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