Parallel Programming Concepts

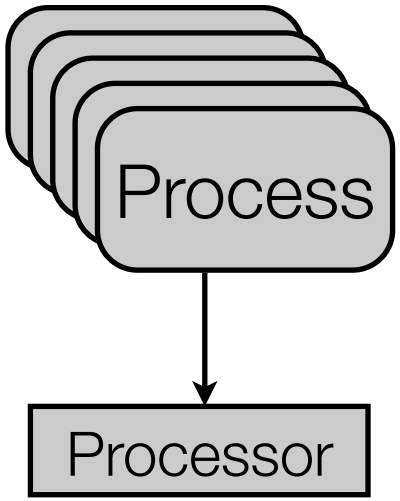
Theory of Concurrency - Shared Memory

Peter Tröger

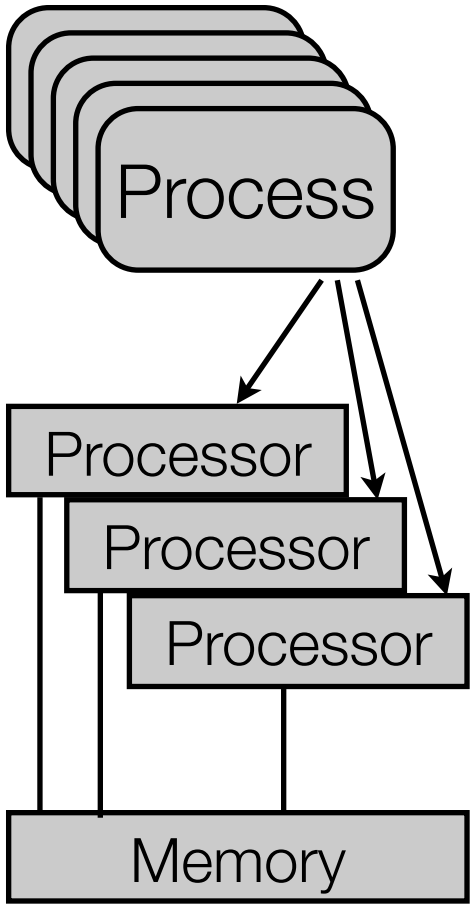
Sources:
Clay Breshears: The Art of Concurrency, Chapter 3
Dijkstra, Edsger W.: Cooperating sequential processes. / Hierarchical ordering of sequential processes
C.A.R. Hoare: Monitors - An Operating System Structuring Concept

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

Von Neumann Model



- Pipelining
- Super-scalar
- VLIW
- Branch prediction
- ...



- 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**
- **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 developed all foundations for concurrent software, mainly for operating system reliability
- *1965, Cooperating Sequential Processes, E.W.Dijkstra*
  - Foundation of abstract 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 parallelism
  - Design principles for parallel programming languages
    - Time-related interference control at compile time
    - Disjoint processes without common variables
    - Explicit declaration of shared resources, critical region support
    - Conditional critical regions
      („with *resource* when *expression* do *critical region*“)

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: Atomic statements are eventually included in the stream (fairness)

Cooperating Sequential Processes [Dijkstra]

- *1965, Cooperating Sequential Processes, E.W.Dijkstra*
  - Starts with comparison of sequential and non-sequential machine
    - Example: Determine largest out of four values

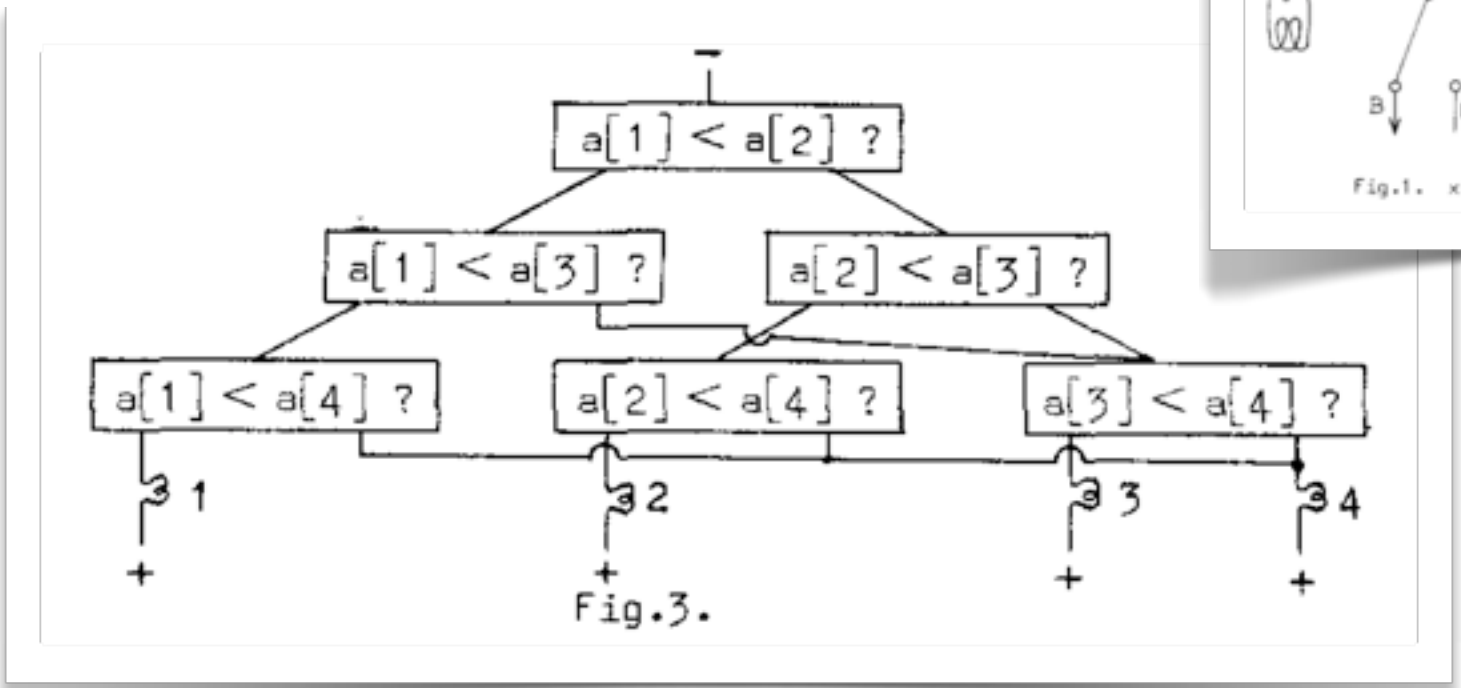


Fig.3.

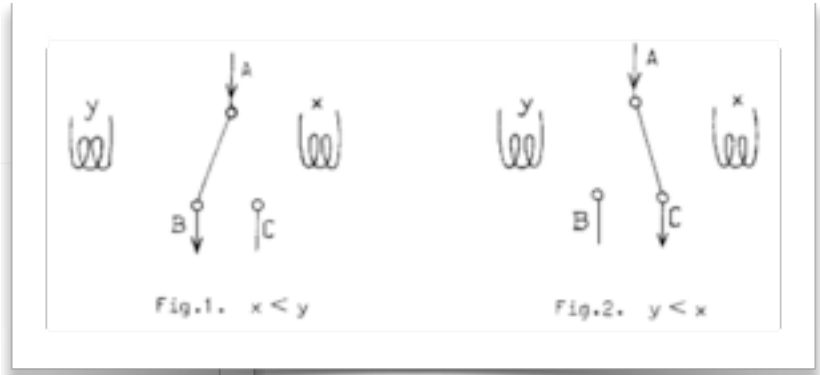
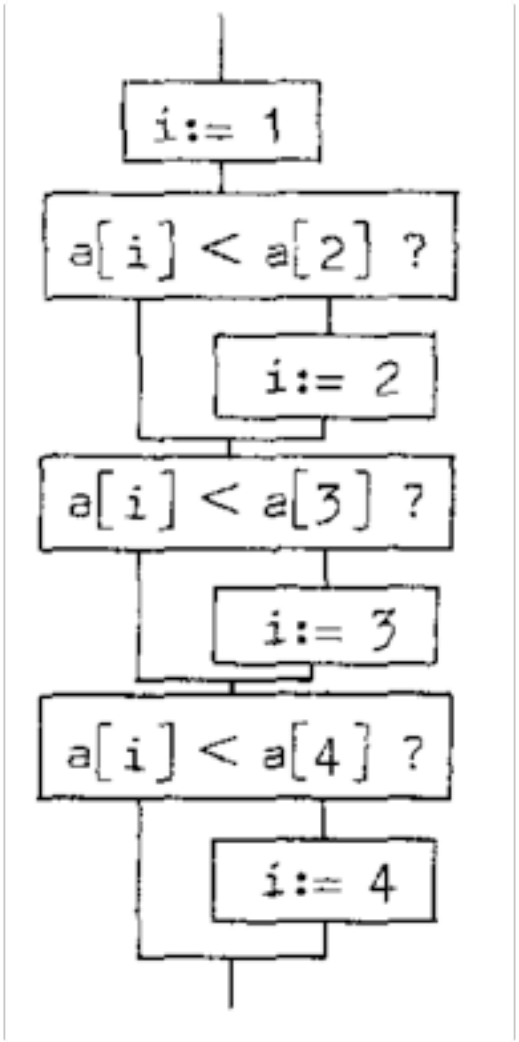


Fig.1. x < y Fig.2. y < x

Cooperating Sequential Processes [Dijkstra]

- Progress of time is relevant for this machine
  - After applying the currents, machine needs some time to show result
- Interpretation in space vs. behavioral interpretation
  - Questions on the same line differs only in left operand - a parameter, based on history -> *variable*
  - Six comparators working simultaneously vs. three comparisons being evaluated in sequence
  - Rules of behavior form a *program*



Cooperating Sequential Processes [Dijkstra]

- Cooperation between loosely coupled sequential processes
  - Beside rare communication moments, processes run independently
  - Disallow any assumption about the relative speed
    - Aligns to understanding of sequential process, which is not affected in its correctness by execution time
      - Examples: Manual input station, stoppable systems
    - Could be regarded as implicit inter-communication
    - Might bring „analogue interferences“ on verification attempt
- Note: Dijkstra already identified crucial issue called **„race condition“** today

Cooperating Sequential Processes [Dijkstra]

- Concept of **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 other flag
  - Mutual exclusion guaranteed
    - If c2=1, then c1=0 already, 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"  .
```

Deadlock ?

- *1970. E.G. Coffman and A. Shoshani. Sequencing tasks in multiprocess systems to avoid deadlocks.*
  - Four necessary conditions required to allow deadlock
    - **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

Cooperating Sequential Processes [Dijkstra]

- Fourth approach
  - Reset flag for critical section entering if the other one has that too
- Rejected due to relative speed assumption
  - Machines running the processes might be the same !
  - 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
  - Combination of fourth approach and turn ‚variable‘, which realizes mutual blocking avoidance through prioritization
  - Spin for section entry only if it is your turn

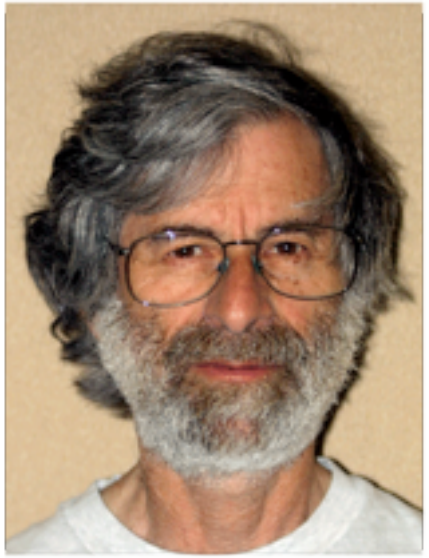
```
"begin  integer c1, c2, turn;
        c1:= 1; c2:= 1; turn:= 1;
        parbegin
        process 1: begin A1: c1:= 0;
                         L1: if  c2 = 0  then
                                 begin  if  turn = 1  then  goto  L1;
                                        c1:= 1;
                                    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"  .
```

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);
```



Test-and-Set

- Problems today: Out-of-order execution, re-ordered memory access, compiler optimizations changing semantics
- **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 could 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

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

- 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 - „Reserve“ operation
  - **V**-operation: Increase value of its argument semaphore by 1 as atomic step
- Solution for critical section shared between N processes
- Original proposal by Dijkstra did not mandate any wakeup order, only *progress*
  - Longest waiting time assumption by Hoare, later debated from operating system implementation point of view
    - „Bottom layer should not bother with macroscoping 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" .
```

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 might need to delay a caller in some step
  - **wait** operation: Issued inside the monitor, causes the caller to wait and temporarly release the monitor while waiting for some assertion
  - **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 wait to be signaled for the condition they are waiting for
  - Hidden implementation of condition, 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 method exclusion 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 by calling *wait()* or leaving the *synchronized* method
    - Threads calling *notify()* are still continued, so data still might change until they 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() {
    if(!valueSet)
      try { this.wait(); }
      catch(InterruptedException e) { ... }
    valueSet = false;
    this.notify();
    return n;
  }
  synchronized void put(int n) {
    if(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);
  }
}
```

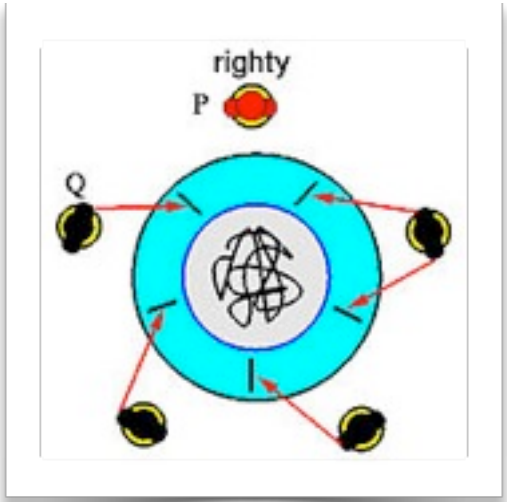
Dining Philosophers [Dijkstra]

- Shared memory synchronization has different standard issues
- Explanation of the *deadly embrace (deadlock)* and the *starvation problem*
- Five philosophers, eating spaghetti or thinking, never speak to each other
- Only five forks, need two to eat
  - No two neighbors may eat at the same time
  - Philosophers as tasks,
    forks as shared resource
  - How can a deadlock happen - lefty / righty ?
  - How can a live-lock (starvation) happen ?

(C) Wikipedia

One Solution: Lefty-Righty-Approach

- $PHIL_n$ is a righty (ist 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 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



Reader / Writer Locks

- Multitude of high-level synchronization primitives, based on initial mutual exclusion and critical section concepts
- *1971, Concurrent Control with „Readers“ and „Writers“. Courtois et al.*
  - Special case of mutual exclusion through semaphores
    - „Reader“ processes are allowed to 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`

Windows Dispatcher Object

Windows Research Kernel

Filter: Dispatch

Start | DISPATCHER_HEADER | KiInitSpinLocks | _DISPATCHER_HEADER

Functions (12/4367)

Types (7/1957)

- _DISPATCHER_CONTEXT
- _DISPATCHER_HEADER
- _DISPATCHER_HEADER32
- _DISPATCHER_HEADER64
- _FAST_IO_DISPATCH
- _LSA_DISPATCH_TABLE
- _PM_DISPATCH_TABLE

base\ntos\inc\ntosdef.h

_DISPATCHER_HEADER

Source Code

Hide Line Numbers

```
566 struct _DISPATCHER_HEADER {
567     union {
568         struct {
569             UCHAR Type;
570             union {
571                 UCHAR Absolute;
572                 UCHAR NpxIrql;
573             };
574
575             union {
576                 UCHAR Size;
577                 UCHAR Hand;
578             };
579
580             union {
581                 UCHAR Inserted;
582                 BOOLEAN DebugActive;
583             };
584         };
585
586         volatile LONG Lock;
587     };
588
589     LONG SignalState;
590     LIST_ENTRY WaitListHead;
591 };
```

Windows Semaphore Object

Windows Research Kernel

Filter: SEMAPHORE

ASSERT_SEMAPHORE | KSEMAPHORE | SEMAPHOR...ORMATION | SEMAPHOR...ON_CLASS

- Functions (9/4167)
- Types (3/1957)
- Typedefs (6/5432)

KSEMAPHORE
PKSEMAPHORE
PRKSEMAPHORE
PSEMAPHORE_BASIC_INFORMATION
SEMAPHORE_BASIC_INFORMATION
SEMAPHORE_INFORMATION_CLASS

base\ntos\inc\ke.h

KSEMAPHORE

Read Accesses (rvalue) [3]

Write Accesses (lvalue) [2]

Field Statistics

Source Code

Hide Line Numbers

```
838 typedef struct _KSEMAPHORE {
839     DISPATCHER_HEADER Header;
840     LONG Limit;
841 } KSEMAPHORE, *PKSEMAPHORE, *PRKSEMAPHORE;
```

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.