Parallel Programming and Heterogeneous Computing

Shared-Memory: Concurrency & Synchronization

Max Plauth, *Sven Köhler*, Felix Eberhardt, Lukas Wenzel, and Andreas Polze
Operating Systems and Middleware Group

Concurrency in History

- *1961, Atlas Computer & LEO III*
  - Based on Germanium transistors, military use & accounting
  - 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*

1

Cooperating Sequential Processes

Cooperating Sequential Processes [Dijkstra1965]
A Comparator

Paper starts with a discussion of theoretical sequential machines.

Example: **Sequential** electromagnetic solution to find the index of the **largest** value in an array.

Building block: Binary comparator cell

- Current lead through magnet coil
- Switch to magnet with larger current

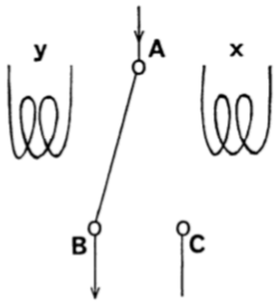


Fig. 1. $x<y$

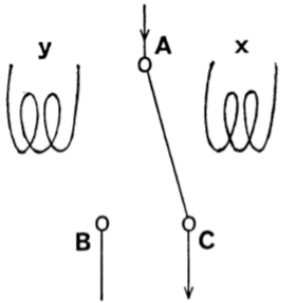
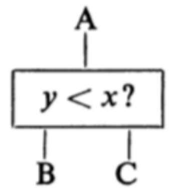


Fig. 2. $y<x$



no yes

Cooperating Sequential Processes [Dijkstra1965] Sequence of Comparators

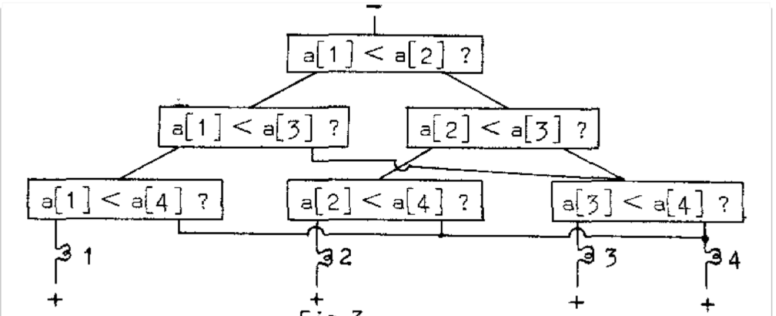
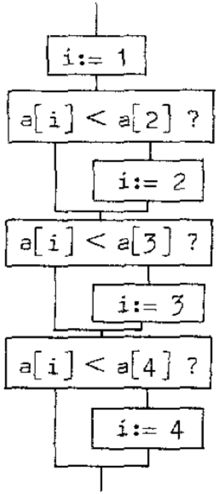


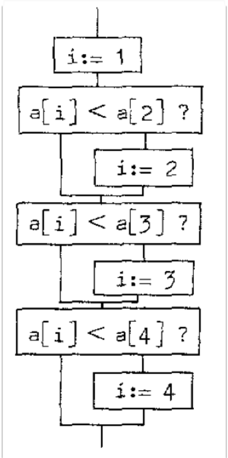
Fig.3.

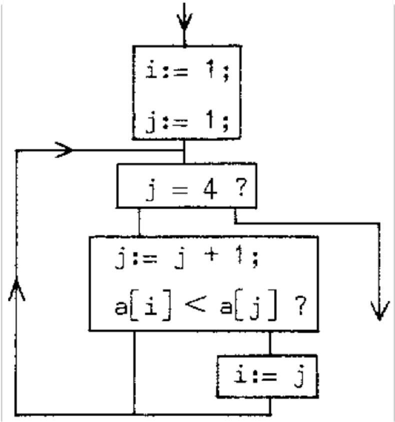


- **Progress of time** is relevant
  - After applying one step, machine needs some time to show the result
  - Same line differs only in left operand
  - Concept of a parameter that comes from past operations, leads to alternative setup for the same behavior
- Rules of behavior form a **program**

Cooperating Sequential Processes [Dijkstra1965]
Different Expressions of Sequence

- 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





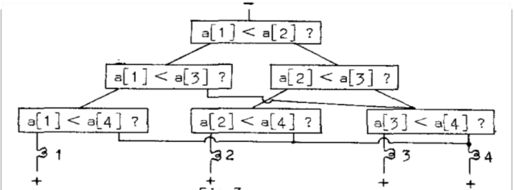


Fig.3.

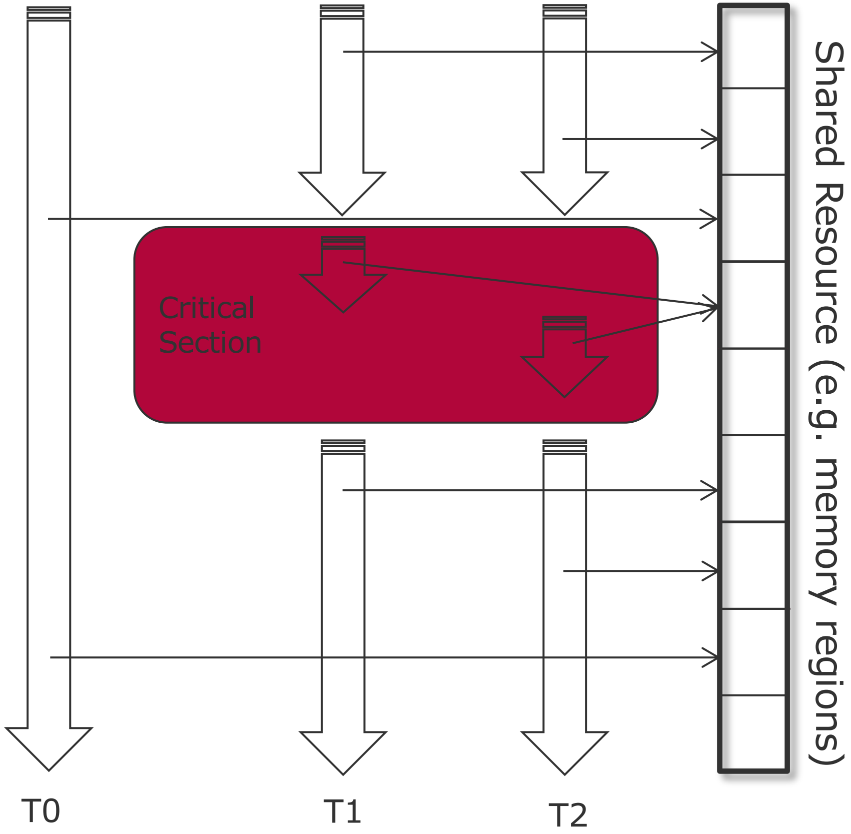
```
"       i:= 1; j:= 1;
back: if j ≠ n then
        begin j:= j + 1;
              if a[i] < a[j] then i:= j;
              goto back
        end" .
```

Cooperating Sequential **Processes** [Dijkstra1965]

: **Race condition**

- 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 speed
  - If this is not fulfilled, might result in "analogue interferences" (**race conditions**).
- Prevention: **A critical section** for two cyclic sequential processes
  - At any moment, at most one process is engaged in the section
  - Implemented through common variables
  - Implementation requires atomic read / write behavior

Critical Section



Critical Section Problem

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

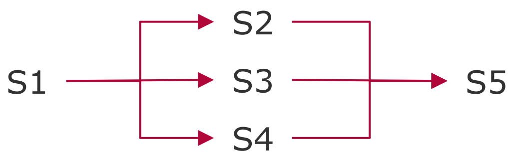
: Critical Section
: Mutual Exclusion
: Progress
: Bounded Waiting

Cooperating Sequential Processes [Dijkstra1965] Compounds and cycles

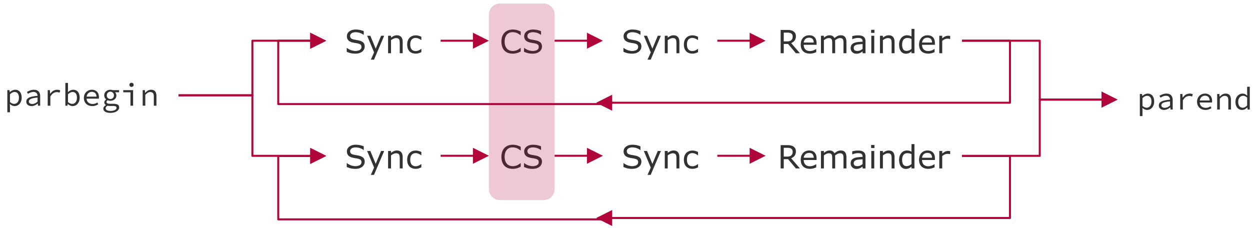
HPI Hasso Plattner Institut

- `parbegin / parend` extension to ALGOLG60 – every statement within compound block is run concurrently

```
begin S1; parbegin S2; S3; S4 parend; S5 end
```



- Assumes atomicity on statement (source code line) level
- A cycle is a repeated synchronization, critical section and non-critical remainder part of two cooperating processes.



Cooperating Sequential Processes [Dijkstra1965]
Approach #1: Turn Flag

- First approach:
  - Passing a single flag
- Discussion:
  - Too restrictive, since strictly alternating
  - One process may die or hang outside of the critical section **(no progress)**

```
"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 [Dijkstra1965] Approach #2: Two Flags

- Separate indicators for enter/ leave
- More fine-grained waiting approach
- Too optimistic, both processes may end up in the critical section **(no mutual exclusion)**

```
"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 [Dijkstra1965] Approach #3: First Raise, then Check

- First *raise the flag*, then check for the other
- Mutual exclusion works
  - If c1=0, then c2=1, and vice versa in CS
- 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" .
```

: Deadlock

Cooperating Sequential Processes [Dijkstra1965] Approach #4: Raise, Check, Lower, Repeat

- Reset locking of critical section if the other one is already in
- Problem due to assumption of relative speed
  - Can lead for one slow process to starve (**bounded waiting**)
  - or **live lock** (both spinning)

```
"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"   .
```

: Livelock

Cooperating Sequential Processes [Dijkstra1965]
Solution: Dekker got it!

- Solution: Dekker‘s algorithm, attributed by Dijkstra
  - Combination of approach #4 and a variable `turn`, which realizes mutual blocking avoidance through prioritization
  - Idea: Spin for section entry only if it is your turn

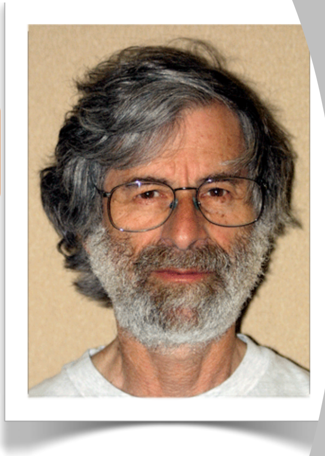
```
"begin integer c1, c2, turn;
    c1:= 1; c2:= 1; turn:= 1;
    parbegin
    process 1: begin A1: c1:= 0;
            L1: if c2 = 0 then
                    begin if turn = 1 then goto L1;
                        c1:= 1;
                    B1: if turn = 2 then goto B1;
                        goto 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" .
```

Bakery Algorithm [Lamport1974]

```
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[j]) ;
        while ((num[j] != 0) &&
         ((num[j],j) "<" (num[i],i)))
        {};}}

def unlock(i) {
   num[i] = 0; }

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



The Downside of Proposed Solutions

- Dekker provided first correct solution only based on shared memory, guarantees three major properties
  - **Mutual exclusion**
  - **Freedom from deadlock**
  - **Freedom from starvation**

```
critical section 2;
turn:= 1; c2:= 1;
remainder of cycle 2; goto  A2
```

- 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
- Situation today: Unpredictable sequential instruction stream
  - Out-of-order execution
  - Re-ordered memory access
  - Compiler optimizations

Test-and-Set Instructions

- **Test-and-set** processor instruction, wrapped by the operating system or compiler
  - 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 **(spin) lock**
- Efficient especially for short waiting periods
- For long periods try to *deactivate* your processor between loops.

```
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"

EWD123 - 27

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.

Cooperating Sequential Processes [Dijkstra1965] Binary and General Semaphores

- Find a solution to allow waiting sequential processes to *sleep*
- Special purpose integer called **semaphore**, two **atomic** operations
  - **P**-operation: Decrease value of its argument semaphore by 1, **"wait"** if the semaphore is already zero
  - **V**-operation: Increase value of its argument semaphore by 1, 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"

```
wait (S):
   while (S <= 0);
   S--;

signal (S):
   S++;
```

Cooperating Sequential Processes [Dijkstra1965] 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" .
```

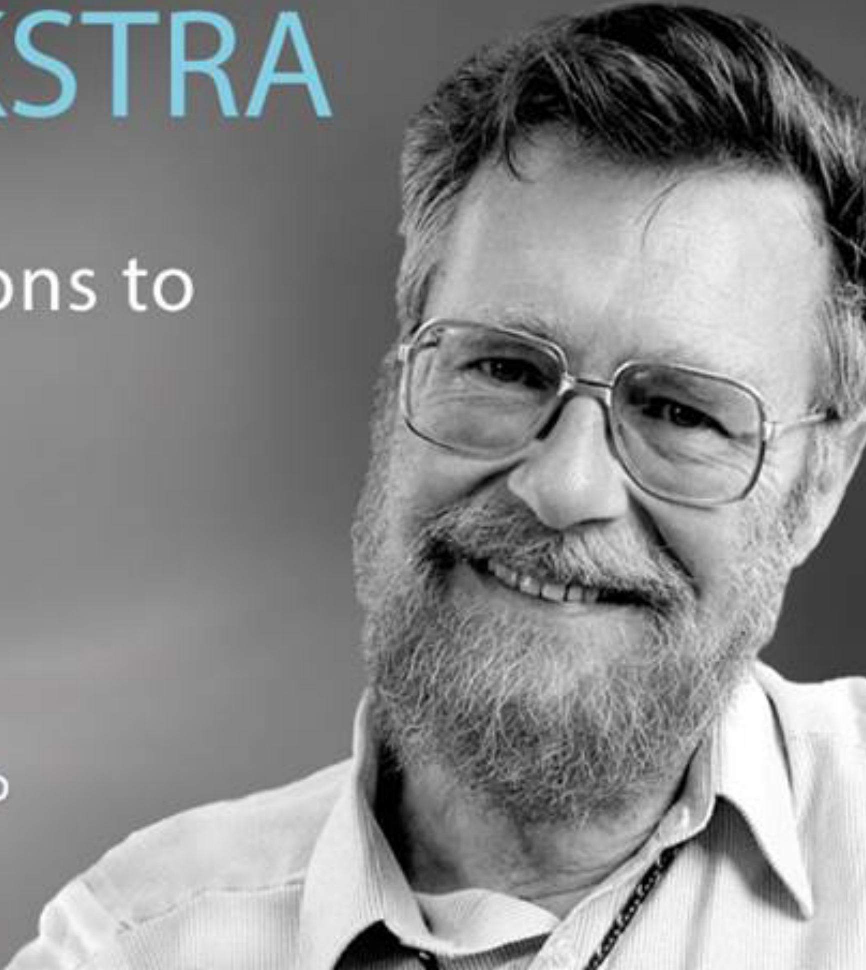
Cooperating Sequential Processes [Dijkstra1965]
Example: General (Counting) Semaphore

```
"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" .
```

EDSGER W. DIJKSTRA

Fundamental contributions to programming as a high, intellectual challenge.

A.M. TURING AWARD 1972

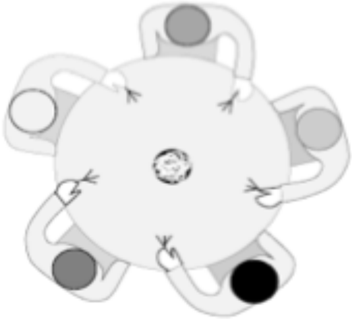


https://www.youtube.com/watch?v=6sIlKP2LzbA

Other Synchronization Primitives

Dining Philosophers Problem [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

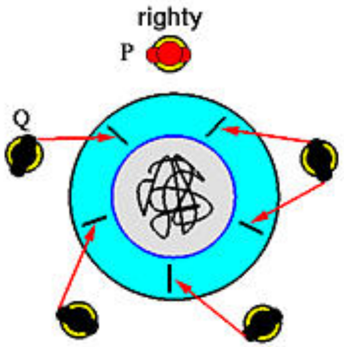


Dining Philosophers Problem [Dijkstra]

- 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

Possible 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

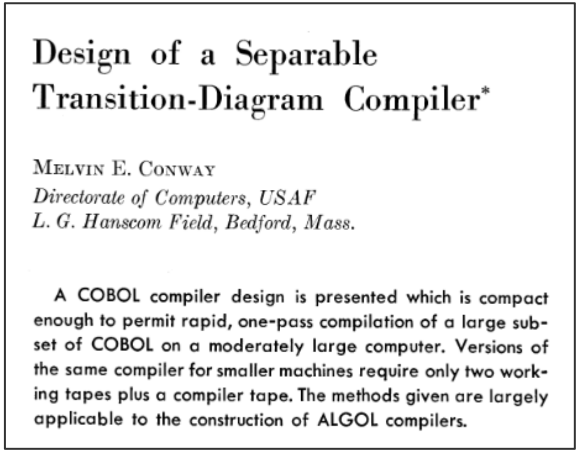


Coffman Conditions [Coffman1970]

- *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 task at a time
  - **Hold and wait condition** – Task already holding resources may attempt to hold new resources
  - **No preemption condition** – Once a task holds a resource, it must voluntarily release it on its own
  - **Circular wait condition** – Possible for a task 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

Coroutines [Conway1963]

- 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 switching
  - Portable implementations in C are difficult
  - Fiber concept in the operating system is helpful


Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

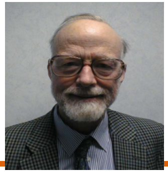
Coroutines

```
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():
    for i in range(5):
        yield i * 2

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

Monitors [Hoare1974]



- 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;
```

Monitors in Java

- Monitors are part of the Java programming language
- Add *synchronized* keyword to method, to make access exclusive.
- *Object* base class provides condition variable functionality – *Object.wait()*, *Object.notify(),* and a wait queue, callable from synchronized methods

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);
  }
}
```

Other 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“ tasks can enter the critical section at the same time, but „Writer“ task should gain exclusive access
  - Different optimizations possible:
    minimum reader delay, minimum writer delay, throughput, …

High-Level Primitives: Concurrent Collections

Datastructures with build-in concurrency support

- 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

Microsoft Parallel Patterns Library

Java 7 – java.util.concurrent

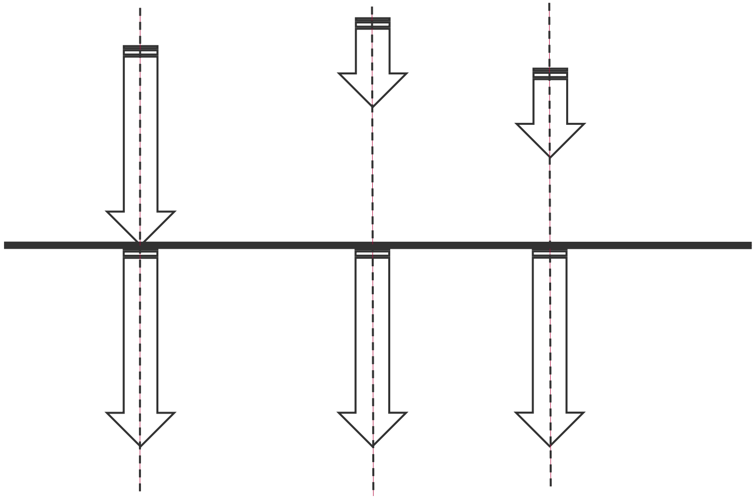
Class Summary

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 lock needs to remember the locking task(s), which increases the overhead

High-Level Primitives: Barrier



- All concurrent activities stop there and continue together
- Participants statically defined at compile- or start-time

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

```c
int x = 23, y = 0;
bool done = false;
```

```c
x = 42;
done = true;
```

```c
while (!done) {}
y = x;
```

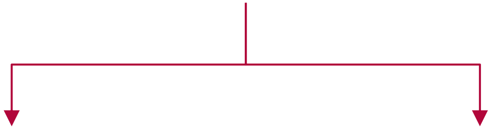
```c
printf("%d\n", y);
```

y?

Boehm, H. J., & Adve, S. V. (2012). You don't know jack about shared variables or memory models. *Communications of the ACM*, *55*(2), 48-54.

Instruction Reordering

```
int x = 0, y = 0;
```



```
x = 2000;
y = 11;
```

```
printf("%d\t", y);
printf("%d\n", x);
```

Possible Outputs:

- 0 0
- 0 2000
- 11 2000
- 11 0

When is reordering allowed (per Thread)?

Arch	LoadLoad	LoadStore	StoreLoad	StoreStore
x86, amd64			✓	
ARM, Power	✓	✓	✓	✓

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. ARM/Power)
- 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

https://en.cppreference.com/w/cpp/atomic/atomic/store

Transactional Memory [C++ JTC1/SC22 Proposal]

```
void LockFreeQueue::push(Node* newHead)
{
    atomic_noexcept
    {
        // begin tranxaction
        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 write operation encounters an invalidated cache, fail

        oldHead = newHead;
        // commit transaction, repeat on fail.
    }
}
```

Transactional Memory (POWER8)

```
tm_start:
tbegin.                  # TFHAR ← addr(tbegin+4)
  beq fail_handler       # if failure go to handler
  lwz r2, tlock          # make failure handler's lock part of transaction's load footprint
  cmpwi r2, 1            # some thread in critical section in failure handler?
  bne transaction        # no, go execute transaction
  tabort.                # abort transaction and enter failure handler
  transaction:           # begin transaction if lock not taken

  <transactional loads, stores, and other instructions>

  tsuspend.              # enter suspend region

    <Suspended state loads, stores, and other instructions>

  tresume.               # resume transaction

    <additional transactional loads, stores, and other instructions>

tend.                    # attempt to commit the transaction

....
....

fail_handler:

          <determine whether transaction retry possible>

          beq tm_start           # reattempt if it is reasonable to do so
          loop: lwz r2, tlock    # check lock
                cmpwi r2, 1      # taken already?
```

- concurrent writes detected via cache invalidation
- cpu status flag signals failed transaction
- fail handler can choose to use **lock elision**

Le, Hung Q., et al. "Transactional memory support in the IBM POWER8 processor." *IBM Journal of Research and Development* 59.1 (2015): 8-1.

8 Simple Rules For Concurrency [Breshears2009]

- *„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

Literature

[Dijkstra1965]

Dijkstra, E. W. (1965). "Cooperating sequential processes" reprinted in *The origin of concurrent programming* (pp. 65-138). Springer, New York

[Lamport1974]

Lamport, L. (1974). "A new solution of Dijkstra's concurrent programming problem". *Communications of the ACM*, 17(8), 453-455.

[Coffman1970]

Shoshani, A., & Coffman, E. G. (1970, October). "Sequencing tasks in multiprocess systems to avoid deadlocks". In *11th Annual Symposium on Switching and Automata Theory (swat 1970)* (pp. 225-235). IEEE.

[Hoare1974]

Hoare, C. A. R. (1974). "Monitors: An operating system structuring concept." reprinted in *The origin of concurrent programming* (pp. 272-294). Springer, New York

And now for a break and a cup of macchiato*.

*or beverage of your choice