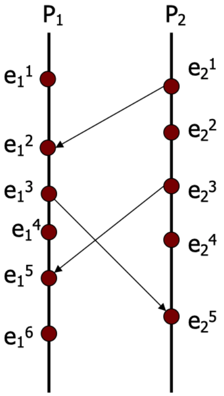
1 Concurrency Errors

1.1 Sequential Consistency

- Instructions are executed by the order they appear in the program.
- Memory behaves as a shared array. (Reads and writes are effective immediately)
- This is naturally true for a sequential program, but it is not true for concurrent program!

1.2 How to see correct states?

First we go the maximum we can go in one side, until we find a precedence. Next we advance the minimum from the other side until our precedence is fulfilled.



1.3 Common Concurrency Errors

- Data Races (atomicity violations)
- Ordering violation
- Unintended sharing
- High-level atomicity violations
- Deadlocks
- Livelocks

1.4 Data Race Detection

1.4.1 Static Program Analysis

Advantages:

- Reason about all inputs/interleavings
- No run-time overhead
- Adapt well-understood static-analysis techniques
- Possibly with annotations to document concurrency invariants

Disadvantages:

- Tools produce "false positivesänd/or "false negatives"
- May be slow, require program annotations
- May be hard to interpret results
- May not scale to large or complex programs

1.4.2 Dynamic Program Analysis

Advantages:

- **Soundness** - Every actual data race is reported
- **Completeness** - All reported warnings are actually races (avoid "False Positives")

Disadvantages:

- Run-time overhead (5-20x for best tools)
- Memory overhead for analysis state
- Reason only about observed executions

Types of algorithms:

- Lock-set Algorithm
- Happens-Before
- Noise-Injection

Happens-Before : Defines a partial order for events in a set of concurrent threads. We can stablish a happen-before if we have locks (lock in one thread, unlock in order thread over the same lock).

Lock-set Algorithm

Two data structs (LockHeld(x) , LockSet(x))

When thread 't' acquires lock 'l' = $LocksHeld(t) = LocksHeld(t) \cup l$

When thread 't' releases lock 'l' = $LocksHeld(t) = LocksHeld(t) \setminus t$

When thread 't' accesses location 'x' = $LocksSet(x) = LocksHeld(t) \cap LocksSet(t)$

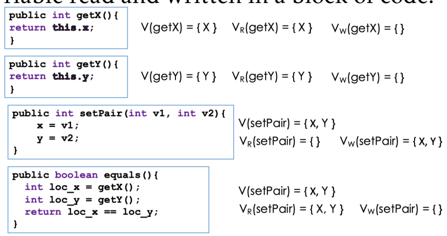
"Data race"warning if Lockset(x) becomes empty.

Locks Held start empty and LockSet Start with all locks.

No warnings is equivalent to no data races on the current execution, but wanings does not imply data race.

1.4.3 High Level Data Races

View Analysis is composed by every variable read and written in a block of code.



```
public int getX(){
return this.x;
}
```

V(getX) = { X }   V_R(getX) = { X }   V_W(getX) = { }

```
public int getY(){
return this.y;
}
```

V(getY) = { Y }   V_R(getY) = { Y }   V_W(getY) = { }

```
public int setPair(int v1, int v2){
    x = v1;
    y = v2;
}
```

V(setPair) = { X, Y }
V_R(setPair) = { }   V_W(setPair) = { X, Y }

```
public boolean equals(){
  int loc_x = getX();
  int loc_y = getY();
  return loc_x == loc_y;
}
```

V(setPair) = { X, Y }
V_R(setPair) = { X, Y }   V_W(setPair) = { }

Casual Dependencies Graph= Data Dependencies + Control Dependencies.

There is a direct correlation between a read variable 'x' and a written variable 'y' if in the dependency graph D there is a path from 'x' to 'y'

There is a Common Correlation between read variable 'x' and 'y' if in a dependency graph d, there is a write variable 'z' such that: 'z != x' and 'z != y' and there are paths from 'x' to 'z' and from 'y' to 'z'

Test for HLDR

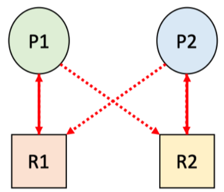
T1 runs V1 = A,B,C and V2 = A,B,C,D
T2 runs V3 = A,B,E and V2 = B,C,F
IS THERE A HLDR?
V2 is maximal in T1
$V2 \cap V3 = A, B$
$V2 \cap V4 = B, C$
$A, B \subseteq B, C or B, C \subseteq A, B$ No!
Common-Correlation(A,C)?
- YES! high Level Data Race
- No! No High Level Data Race

1.5 Deadlock Detection

A set of two or more processes are deadlocked if:

- They are blocked (i.e in the waiting state)
- Each is holding a resource
- Each is waiting to acquire a resource held by another process in the set.

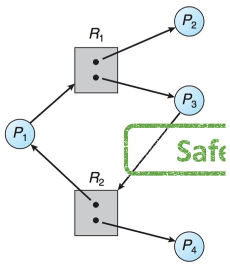
[]



Condicions Necessary for Deadlock

- **mutual Exclusion** only one process can use a resource at a time
- **hold and wait** a process holding at leadt one resource is waiting to acquire additional resources which are currently held by other processes.
- **no preemption** a resource can only be released voluntarily by the process holding it.
- **cirular wait** a cycle of processses requests exists ($P_0 -> P_1 -> P_2 -> ... -> P_{n-1} -> P_0$)

1.6 Resource Allocation Graph



If a graph contains no cycles -> No deadlock
If a graph contains a cycle ->
-if onyly one instace per resource type, then deadlock
- if serveral instaces per resouce type, possibility of deadlock.

1.7 How to Deal with deadlocks

- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery
- Ignore the issue!

1.8 Deadlock Prevention

Restrict the way requests can be made...

Mutual Exclusion

Hold and Wait

-must guarantee what whenever a process requests a resource, it does not hold any other resources.
-require process to request and allocate all its resources before it begins execution.
-low resource utilization; starvation possible

No Preemption

- if a process that is holding some resources requests another resource that cannot be immedialy allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait

- impose a total ordering of all resources types, and require that each process requests reources in an increasing order of enumeration.

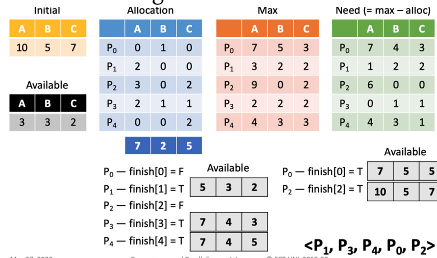
1.9 Deadlock Avoidance

Requires that the system has some aditional a priori information available.
- Requires that each process declares the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition. - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

System is in safe state if there exists a sequence of ALL the processes in the system such that for each Pi, the resources that Pi can be satisfied by currently available resource + resource's held by all the Pj, with j < i
If a system is in safe state -> no deadlocks
If a system is in unsafe state -> possibility of deadlock
Avoidance -> ensures that a system will never enter an unsafe state.

1.10 Avoidance Algorithms

Single instance of a resource type, use a resource-allocation graph
Multiple instances of a resource type, use the bankers algorithm



Initial

A	B	C
10	5	7

Available

A	B	C
3	3	2

Allocation

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2
	7	2	5

Max

	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Need (= max – alloc)

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

P_0 — finish[0] = F
P_1 — finish[1] = T   Available: 5 3 2
P_2 — finish[2] = F
P_3 — finish[3] = T   7 4 3
P_4 — finish[4] = T   7 4 5

P_0 — finish[0] = T   Available: 7 5 5
P_2 — finish[2] = T   10 5 7

$\langle P_1, P_3, P_4, P_0, P_2 \rangle$

2 Parallel Architectures

Flynn's Taxonomy

- Single Instruction (SI) - System in which all processors execute the same instruction
- Multiple Instruction (MI) - System in which different processors execute different instructions
- Single Data (SD) - System in which all processors operate on the same data
- Multiple Dara (MD) - System in which different processors may operate on different data.
- SISD - Classic von Neumann architecture; serial computer
- MIMD - Collection of autonomous processors that can execute multiple independent programs; each of which can have its own data stream
- SIMD - Data is devided among the processors and each data item is subjected to the same sequnce of instructions; GPUs; Advanced Vector Extensions (AVX)
- MISD - Very rare;

MIMD Architectures (Shared Memory)

Uniform Memory Access (UMA) - Each cpu uses same memory.

Non-Uniform Memory Access (NUMA) - Each set of CPU uses a part of the memory.

Distributed Memory

Hybrid Memory

3 Performance

Efficacy - Computational requirements (what needs to be done?)

Efficiency - Computing resources (how much will it cost?)

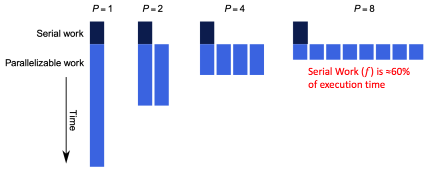
Embarrassingly Parallel Computation - is one that can be obviously divided into completely independent parts that can be executed simultaneously. (Trully - there is no interaction, nearly - result must be collected in some way)

4 Performance Metrics and Formulas

$S(p) = \frac{T_1}{T_p}$, $E(p) = \frac{S_p}{p, Cost(p)=p*T_p}$

5 Amddahl's Law

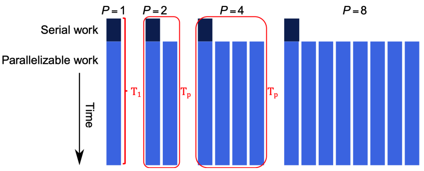
Interested in solving problems faster.
- Time: wall clock time



$S_p <= \frac{1}{f + \frac{(1-f)}{p}}$, $S_{p->inf} <= \frac{1}{f}$

6 Gustafson-Barsis Law

- Often interested in large problems when scaling
- Time: CPU time



$S <= \alpha + P.(1 - \alpha)$

6.1 Work-Span

Work Law: $T_p \geq \frac{T_1}{P}$
Span Law: $T_p \geq T_\infty$

Speedup = $\frac{T_1}{T_p}$ on P Processors

If $T_1/T_p = P$,we have (perfect) linear speedup.
If $T1/Tp > P$, we have superlinear speedup, which is not possible in this performance model, beacuse of the work Law.
We can write a work-span formula to derive a lower bound on T_p : $Max(T_1/P, T_\infty) \leq T_p$
Brent's Lemma derives an upper bound: Captures the additional cost executing the other tasks not on the critical path, assume can do so without overhead.
$T_p \leq (T_1 - T_\infty)/P + T_\infty$

7 Parallel Patterns

- Nesting Pattern

7.1 Serial Control Patterns

sequence,selection,iteration,recursion

7.2 Parallel Control Pattern

fork-join,map,stencil,reduction,scan, recurrence

7.3 Serial DATA management Patterns

random read and write, stack allocation, heap allocation, objects.

7.4 Parallel Data Management Patterns

pack, pipeline, geometric decomposition, gather, scatter.

8 Depencencias

Sequential Consistency in Parallel Exectuion, Statments exectuion does not interfere with each other, Computations result equal to either A B or B A
True Dependency - Read After Write δ
Anti-Dependence - Write After Read δ^{-1}
Output Dependence - Write After Write δ^0
Some dependences can be removed by modifying the program. - Rearranging statements and or Eliminating statements.
Loop Carried dependence is a dependence between two statements instances in two different iterations of a loop. Otherwise, it is loop-independent.

9 Map Reduce

a program model and a an associated implementation for processing large databasets.
Batch processing: - All input in know when the computation starts, the complete computation is executed, No interaction with the user.

1. Record Reader
2. Map
3. Combiner (Optional)
4. Partioner
5. Shuffle and Sort
6. Reduce

10 Synchronization

Sequencial Algorithm -> formal description of a behavior of a sequential state machine. When written in a specific programming language. an algorithm is called a program. A process is an instance of an algorithm.
Competition : To access to the disk resource
Cooperation : Barrier, producer-consumer

10.1 Solving Mutual Exclusion

Atomic Register

```
operation acquire_mutex(i) is
    FLAG[i] ← up;
    AFTER_YOU ← i;
    wait ((FLAG[j] = down) ∨ (AFTER_YOU ≠ i));
    return()
end operation.

operation release_mutex(i) is FLAG[i] ← down; return() end operation.
```

Special hardware primitives
test&set / reset
swap
compare&swap
fetch&add

11 Locking Strategies

11.1 Coarse-Grained Synchronization

Use a single lock. Methods executed in mutual exclusion.
Eliminates all the concurrency within the object.

11.2 Fine-Grained Synchronization

(Hand-over hand locking, linked list)

Split object intomultiple independenty-synchronized components.

Methods conflict when they access the same component at the same time.

11.3 Optimistic Synchronization

Check if the operation can be done if so, lock and check again.

Traverse the list without locking until location is found. Lock nodes and transverse again to confirm that the locked nodes are still in the list.

11.4 Lazy Synchronization

Pospone Hard Work (Logical removal, Physical removal containers are wait-free)

11.5 Lock-Free Synchronization

Compare and Set reference and deleted bit at the some time. (In java use,)