Network Algorithms

Shared Memory
Spectrum of Distributed (Computer) Systems

E.g., tiny graphical processing units (GPUs) and specialized devices, in which large arrays of **simple processors** work in lock-step (“Gleichschritt”), PRAM, ...

Multi-threaded + multi-core servers/desktops with **shared memory for communication**.

Loosely-coupled **peer-to-peer** systems with message passing communication

---

small/synchronous/... wide-area/decoupled/...
The Shared Memory Model

Shared memory consists of registers.

Shared memory consists of registers.
**Shared Memory**

A shared memory system is a system that consists of asynchronous processes that access a **common (shared)** memory. A process can atomically access a register in the shared memory through a set of **predefined operations**. An atomic modification appears to the rest of the system instantaneously. Apart from this shared memory, processes can also have some **local (private) memory**.

**Often a useful, simpler alternative model to reason about distributed systems!**
Atomic operations on \( R \)? Often advanced versions of read-modify-write.

**Examples:** (a.k.a. Data Types, Mealy Machines)

1. **Test-and-Set**(\( R \)): \( t := R; \ R := 1; \) return \( t \)

2. **Fetch-and-Add**(\( R; \ x \)): \( t := R; \ R := R + x; \) return \( t \)

3. **Compare-and-Swap**(\( R; \ x; \ y \)):
   
   if \( R = x \) then \( R := y; \) return true;
   
   else return false; endif

Shared Register \( R \)
Why Shared Memory?

- Programming a shared memory system is easier: programmers access global variables directly!

- Because of this, even message passing systems often programmed through a shared memory middleware!

- From a message passing perspective, shared memory model is like a bipartite graph:

Nodes (asynchronous, may fail)

Shared registers (no failures, no delay)
The Power of RMW

The power of a shared memory system is determined by the Consensus Number ("universality of consensus").

**Consensus Number**

The power of the RMW variant is measured by the consensus number. Consensus number $k$ defines whether one can solve consensus for $k$ processes (but not $k+1$).

Examples:

- **Test-and-Set** has consensus number 2
  (one can solve consensus with 2 processes, but not 3)
- **Compare-and-Swap** has an infinite Consensus Numbers!
Desirable Properties of Distributed Systems

Safety, Liveness

**Safety:** “Something bad will never happen”,
Examples: some invariant holds
(function never returns -1 values),
serializability for DB transactions,
linearizability

**Liveness:** “Eventually something good happens”,
“system makes progress”
An operation op1 \textit{precedes} an operation op2 iff op1 terminates before op2 starts.
An operation op2 \textit{follows} operation op1 iff op1 precedes.
Linerizability

History

A history is a sequence of invocations and responses made of an object by a set of threads.

Linearizable

A history $H$ of an object type $T$ (with a given sequential spec) is linearizable if there is an equivalent total order $O$ of operations in $H$, such that

1. $O$ respects the real time order of operations in $H$: if an operation finished before another started, it appears before in the history,

2. $O$ respects the sequential specification of $T$. 
A Classic Shared Memory Problem

Fundamental synchronization problem: access to a resource

The Mutual Exclusion Type

Each process executes the following code sections:

<Entry> ➔ <Critical Section> ➔ <Exit> ➔ <Remaining Code>

A mutual exclusion algorithm consists of code for entry and exit sections, such that the following holds:

1) Mutual Exclusion (Property?): At most one process is in the critical section.
2) No deadlock (Property?): If some process manages to get to the entry section, later some (possibly different) process will get to the critical section.

Sometimes we in addition ask for

3) No lockout (Property?): If some process manages to get to the entry section, later the same process will get to the critical section. (“Fairness”)
4) Unobstructed exit (Property?): No process can get stuck in the exit section.
A Classic Shared Memory Problem

Fundamental synchronization problem: access to a resource

Mutual Exclusion

Each process executes the following code sections:
<Entry> ➔ <Critical Section> ➔ <Exit> ➔ <Remaining Code>

A mutual exclusion algorithm consists of code for entry and exit sections, such that the following holds:

(1) **Mutual Exclusion (Safety)**: At most one process is in the critical section.
(2) **No deadlock (Liveness)**: If some process manages to get to the entry section, later some (possibly different) process will get to the critical section.

Sometimes we in addition ask for
(3) **No lockout (Liveness)**: If some process manages to get to the entry section, later the same process will get to the critical section. (“Fairness”)
(4) **Unobstructed exit (Liveness)**: No process can get stuck in the exit section.
A Classic Shared Memory Problem

Mutual Exclusion

Each process executes the following code sections:

\[
\begin{align*}
\text{Entry} & \quad \text{Critical Section} & \quad \text{Exit} & \quad \text{Remaining Code}
\end{align*}
\]

A mutual exclusion algorithm consists of code for entry and exit sections, such that the following holds:

1. **Mutual Exclusion (Safety):** At most one process is in the critical section.
2. **No deadlock (Liveness):** If some process manages to get to the entry section, later some (possibly different) process will get to the critical section.
3. **No lockout (Liveness):** If some process manages to get to the entry section, later the same process will get to the critical section. (“Fairness”)
4. **Unobstructed exit (Liveness):** No process can get stuck in the exit section.

**Comments:**
- Mutex trivial if no failures! (How?)
- And trivially linearizable: enter critical section one by one
- One way to “enforce” linearizability
  - Run (group of) operations in critical section
  - Not very efficient solution (“locking”)
- Not a total type!
  - In certain states, no response is defined (if process crashed in critical section)
  - Cannot be implemented in wait-free manner (if process fails in critical section)

Sometimes we in addition ask for

3. **No lockout (Liveness):** If some process manages to get to the entry section, later the same process will get to the critical section. (“Fairness”)
4. **Unobstructed exit (Liveness):** No process can get stuck in the exit section.
Mutex

Test-and-Set(R): t := R; R := 1; return t

How to achieve Mutex with single Test-and-Set register?
Mutex

Input: Shared register R:=0
<Entry>
Repeat:
\[ r := \text{test-and-set}(R) \]
Until r=0
<Critical Section>
…
<Exit>
R:=0
<Remaining Code>
…

Test-and-Set(R): \( t := R; R := 1; \) return t

Set register to 1, then check whether it was so already.

How to achieve Mutex with single Test-and-Set register?
Mutex

Input: Shared register $R:=0$

<Entry>

Repeat:
  $r := \text{test-and-set}(R)$

Until $r=0$

<Critical Section>

... 

<Exit>

$R:=0$

<Remaining Code>

... 

Test-and-Set($R$): $t := R; R := 1; \text{return } t$

How to achieve Mutex with single Test-and-Set register?

Set register to 1, then check whether it was so already.

Correct Mutex?

No deadlock?

No lockout?

Unobstructed exit?
Mutex

(1) Mutex: OK
(2) Deadlock free: OK
(3) Lockout: Not OK!
(4) Unobstructed exit: OK

Proof:
(1) Mutex: R initially 0. Let $p_i$ be the $i$-th process to successfully execute the test-and-set (i.e., result 0) at time $t_i$, and say at time $t'_i$ $p_i$ resets $R:=0$. Between these times, nobody else can execute CS.
(2) One of the processes waiting in the entry section will successfully test-and-set as soon as the process in the critical section exited.
(4) Since the exit section only consists of a single instruction (no potential infinite loops) we have unobstructed exit.
Lockout

May be unfair!

Mutex

Input: Shared register $R:=0$
<Entry>
Repeat:
\[ r := \text{test-and-set}(R) \]
Until $r=0$
<Critical Section>
...
<Exit>
$R:=0$
<Remaining Code>
...

Test-and-Set($R$): $t := R; R := 1; \text{return } t$

Always same process may win!
Solution: make FIFO queue…

What about weaker objects?
Can I do without atomic RMW?
Lockout

May be unfair!

Mutex

Input: Shared register $R:=0$
<Entry>
Repeat:
    $r := \text{test-and-set}(R)$
Until $r=0$
<Critical Section>
...
<Exit>
$R:=0$
<Remaining Code>
...

Test-and-Set($R$): $t := R; R := 1; \text{return } t$

Always same process may win!
Solution: make FIFO queue…

What about weaker objects?
Can I do without atomic RMW?
Yes: Peterson’s algorithm!
Peterson’s Algo

Assume: two processes only!
Need three registers (init: 0).

Peterson’s Mutex

Code for process $P_i$

<Entry>

$W_i := 1$

$\prod := 1 - i$

Loop until $\prod = i$ or $W_{1-i} = 0$

<Critical Section>

...

<Exit>

$W_i := 0$

<Remaining Code>

...

Process indicates that it wants to enter CS in “Want-Register”.
Can only do if other process does not want, or I have priority.

Spin-Lock! (Busy-wait)

Priority register used to avoid deadlock!
Peterson gives (1) mutex, (2) no deadlock, (3) no lockout, (4) unobstructed.

Proof:

(1) **Mutex**: First process at Line 2 gets priority. If both compete, only priority process can access CS.

(2) **No Deadlock**: Priority process gets direct access to the critical section.

(3) **Fairness**: Non-priority process gets priority when other process starts again! Shared variable…

(4) **Unobstructed Exit**: Exit only a single instruction.

QED
Peterson gives (1) mutex, (2) no deadlock, (3) no lockout, (4) unobstructed.

Proof:

(1) **Mutex**: First process at Line 2 gets priority. If both compete, only priority process can access CS.

(2) **No Deadlock**: Priority process gets direct access to the critical section.

(3) **Fairness**: Non-priority process gets priority when other process starts again! Shared variable…

(4) **Unobstructed Exit**: Exit only a single instruction.

QED

What about Mutex with >2 processes?
Another fundamental task: Store&Collect

Goal: Collect up-to-date info about other processes!

Two operations:
- **sop(val)**: Process $p_i$ stores val to be the latest value of its own register $R_i$. ("1:1 write", so single-writer model)
- **cop()**: Collects a "view", a function $V$ where $V(p_i)$ is the latest value stored by $p_i$, for each process $p_i$. 
Assume: registers initialized to «?».

Note: Collect has no sequential specification and cannot be linearized.

Our goal here: A collect operation \textbf{cop} should never read from the future or miss a preceding store operation \textbf{sop}.

For a collect operation \textbf{cop}, the following validity properties must hold for every process \(p_i\):

1. If \(V(p_i) = "?"\), then no store operation by \(p_i\) \textbf{precedes} \textbf{cop}.
2. If \(V(p_i) = v\), then \(v\) is the value of a \textbf{sop} operation of \(p_i\) that does not \textbf{follow} \textbf{cop}, and there is no store operation by \(p_i\) that \textbf{follows} \textbf{sop} and \textbf{precedes} \textbf{cop}.
Complexity Measure

We measure the following complexity.

**Step Complexity**

Step complexity of an operation is the number of accesses to the registers in the shared memory.

**How to implement a valid Collect() operation?**
Simple Algorithm

Step sop() & cop()

Operation \texttt{STORE(val)}, by proc. \( p_i \)

\begin{align*}
R_i & := \text{val}; \\
\text{Operation COLLECT:} \\
\text{for } i := 1 \text{ to } n \text{ do} \\
& \quad V(p_i) := R_i \quad (\ast \text{read register } R_i \ast)
\end{align*}

Works (atomic read/write). Complexity?
Simple Algorithm

**Step sop() & cop()**

**Operation STORE(val), by proc. \( p_i \)**
\[ R_i := \text{val}; \]

**Operation COLLECT:**
\[ \text{for } i := 1 \text{ to } n \text{ do } \]
\[ V(p_i) := R_i \quad (* \text{read register } R_i *) \]
\[ \text{end} \]

Works (atomic read/write).

**Complexity?**
STORE is 1 step
COLLECT is \( n \) steps
Adaptive Algorithm

If only two processes wrote some value, COLLECT is too costly! How to make an operation adaptive to the number of processes that were active in the execution?

Adaptive Operation

If up to time $t$, $k \leq n$ processes have started or finished at least one operation, an operation is called adaptive if step complexity depends on $k$ but not on $n$.

How to make our algorithms adaptive?
If only two processes wrote some value, COLLECT is too costly! How to make an operation adaptive to the number of processes that were active in the execution?

Adaptive Operation

If up to time $t$, $k \leq n$ processes have started or finished at least one operation, an operation is called adaptive if step complexity depends on $k$ but not on $n$.

How to make our algorithms adaptive? We need Splitters…
Synchronization primitive:
- Process entering it exits with stop, left or right
- If k processes enter, at most one exists with stop, and at most k-1 processes exit with left and at most k-1 processes exit with right.
- If single process enters it, stop for sure.

Not perfect balance, but there are two processes that obtain different values (stop, left, right).

How to implement splitter?
Splitter Algo

Two shared registers $X: \{?, 1, ..., n\}$, $Y: \text{bool}$
Initialization: $X=\?$, $Y=false$

**Splitter access by $pi$:**

$X:=i$
If $Y$ then *return right*
Else
  $Y:=\text{true}$
  if $X=i$ then *return stop*
  else *return left*
Splitter Algo

Splitter

Two shared registers X: {?, 1, ..., n}, Y: bool
Initialization: X=?, Y=false

Splitter access by pi:
X:=i
If Y then return right
Else
  Y:=true
  if X=i then return stop
  else return left

Why correct?

Stefan Schmid @ T-Labs Berlin, 2013/4
Correctness

A single process always stops:
- Clear: check solo-run

At most k-1 return right:
- **First process** checking Y will not return right

At most k-1 return left:
- Assume process p is **last** to set X:=i
- If p does not return right, it will find its own value later and **stop**: it does not return left!

At most one process stops:
- Assume contrary: both processes pi and pj return stop, and w.l.o.g. assume pi sets X:=i before pj sets X:=j.
- Both can only reach “else” if Y was false for both! But then, X value of pi has been overwritten in the meanwhile, and pi does not return stop!

QED

**Splitter**

Two shared registers X: {?, 1, ..., n}, Y: bool
Initialization: X=?, Y=false

**Splitter access by pi:**

X:=i
If Y then **return right**
Else
  Y:=true
  if X=i then **return stop**
  else return left

QED
Correctness

A single process always stops:
- Clear: check solo run

At most $k-1$ return right:
- First process checking $Y$ will not return right

At most $k-1$ return left:
- Assume process $p$ is last to set $X:=i$
- If $p$ does not return right, it will find its own value later and stop: it does not return left!

At most one process stops:
- Assume contrary: both processes $p_i$ and $p_j$ return stop, and w.l.o.g. assume $p_i$ sets $X:=i$ before $p_j$ sets $X:=j$.
- Both can only reach “else” if $Y$ was false for both! But then, $X$ value of $p_i$ has been overwritten in the meanwhile, and $p_i$ does not return stop!

How to realize adaptive collect now? Splitter trees!
Splitter Tree

Assume we have $2^{n-1}$ splitters, arranged in complete binary tree:

Let $S(v)$ be splitter of node $v$ in tree.
Additionally: for every splitter, shared variables $Z_S: \{?, 1, \ldots, n\}$ and boolean $M_S$. A splitter is marked if $M_S = \text{true}$. 
Assume we have $2^n - 1$ splitters, arranged in complete binary tree:

Let $S(v)$ be splitter of node $v$ in tree. Additionally: for every splitter, shared variables $Z_S: \{?, 1, \ldots, n\}$ and boolean $M_S$. A splitter is marked if $M_S = \text{true}$.
k processes traversing splitter tree:

- At most one process can stop at some given splitter
- Every process stops at some splitter at depth at most k-1. Why?
k processes traversing splitter tree:
- At most one process can stop at some given splitter
- Every process stops at some splitter at depth at most k-1. Why?

Proof by Induction:
- By definition, k processes enter the root splitter at depth 0
- If k-i processes enter splitter at root of subtree at depth i (induction hypothesis), at most k-i-1 obtain left and right.
k processes traversing splitter tree:
- At most one process can stop at some given splitter
- Every process stops at some splitter at depth at most k-1. Why?

**Proof by Induction:**
- By definition, k processes enter the root splitter at depth 0
- If k-i processes enter splitter at root of subtree at depth i (induction hypothesis), at most k-i-1 obtain left and right.

How to exploit for adaptive STORE and COLLECT?
active

check vicinity
Tree Ops

**Operation STORE\( (val) \) (by process \( p_i \)):**

1. \( R_i := val \)
2. if first STORE operation by \( p_i \) then
3. \( v := \) root node of binary tree
4. \( \alpha := \) result of entering splitter \( S(v) \);
5. \( M_{S(v)} := \) true
6. while \( \alpha \neq \) stop do
7. if \( \alpha = \) left then
8. \( v := \) left child of \( v \)
9. else
10. \( v := \) right child of \( v \)
11. end if
12. \( \alpha := \) result of entering splitter \( S(v) \);
13. \( M_{S(v)} := \) true
14. end while
15. \( Z_{S(v)} := i \)
16. end if

**Operation COLLECT:**

Traverse marked part of binary tree:

17. for all marked splitters \( S \) do
18. if \( Z_S \neq \perp \) then
19. \( i := Z_S; V(p_i) := R_i \)
20. end if
21. end for

Traverse splitter tree from top, mark traversed nodes \((M)\) and store splitter where \( p_i \) stopped.

Only collect marked parts!
Tree Ops

Operation \text{STORE}(val) \ (by\ process\ p_i) : 
1: \ R_i := val 
2: if first \text{STORE} operation by \ p_i then 
3: \ v := \text{root node of binary tree} 
4: \ \alpha := \text{result of entering splitter} \ S(v); 
5: \ M_{S(v)} := \text{true} 
6: while \alpha \neq \text{stop} do 
7: \ if \alpha = \text{left then} 
8: \ v := \text{left child of} \ v 
9: \ else 
10: \ end if 
11: \ \alpha := \text{result of entering splitter} \ S(v); 
12: \ M_{S(v)} := \text{true} 
13: \ end while 
14: \ Z_{S(v)} := i 
15: \ end if 

Operation \text{COLLECT}: 
Traverse marked part of binary tree: 
16: for all marked splitters \ S \ do 
17: if \ Z_S \neq \perp \ then 
18: \ i := Z_S; V(p_i) := R_i 
19: \ end if 
20: \ end if 
21: \ end for

Traverse splitter tree from top, mark traversed nodes (M) and store splitter where \ pi \ stopped.

Complexity of solution?

Only collect marked parts!
Adaptive Collect

Step complexity of first STORE is O(k), subsequent Ones are O(1). COLLECT has step complexity O(k).

Proof.

- First store: splitter tree traversal to find “my” location, at most at depth k
- From then on, will always store there: O(1)
- COLLECT:
  - Only need to check marked part and their neighbors
  - Marked part of the tree is connected
  - At most 2k-1 nodes are marked:
    - By induction: a single process entering a splitter will always stop
    - The right and left child of the root are subtrees too, first node will stop at first splitter.

QED
Adaptive Collect

Step complexity of first STORE is $O(k)$, subsequent Ones are $O(1)$. COLLECT has step complexity $O(k)$.

Proof.

- First store: splitter tree traversal to find “my” location, at most at depth $k$
- From then on, will always store there: $O(1)$
- COLLECT:
  - Only need to check marked part and their neighbors
  - Marked part of the tree is connected
  - At most $2k-1$ nodes are marked:
    - By induction: a single process entering a splitter will always stop
    - The right and left child of the root are subtrees too, first node will stop at first splitter.

QED

Disadvantage? Space complexity! Store $O(2^n)$ tree in memory...
Idea: Instead of arranging splitters in $2^n$ binary tree, arrange them in $n \times n$ matrix:

5x5 splitter matrix
Idea: Instead of arranging splitters in $2^n$ binary tree, arrange them in $n \times n$ matrix:

5x5 splitter matrix

Space complexity $n^2$. Step complexity?
Matrix Collect

Step complexity of first STORE is $O(k)$, subsequent ones are $O(1)$. COLLECT has step complexity $O(k^2)$.

Proof.

- Let $x_i$ be number of procs entering row $i$. By induction on $i$, $x_i \leq k-i$.
- Of course: $x_0 \leq k$
- Let $j$ be largest column s.t. at least one process visits the splitter at $(i-1,j)$.
- Not all processes go left, so $x_i \leq k-i$.
- Same for column.
- So every process stops the latest in row $k-1$ and column $k-1$.
- The number of marked splitters is at most $k^2$: complexity of COLLECT.
- The longest path in matrix is $2k$, so STORE complexity at most $O(k)$.

QED
Remarks

- Randomized algorithms can achieve binary trees of depth $O(\log n)$ only.

- $O(k)$ step complexity and $O(n^2)$ space complexity is possible for COLLECT, even deterministically.
End of Lecture