Class 15
Today’s plan

- Pragmatic issues for shared-memory multiprocessors
- Practical mutual exclusion algorithms
  - Test-and-set locks
  - Ticket locks
  - Queue locks
- Generalized exclusion/resource allocation problems
- Reading:
  - Herlihy, Shavit, Chapter 7
  - Mellor-Crummey, Scott paper (Dijkstra prize winner)
  - Magnussen, Landin, Hagersten paper
  - Lynch, Chapter 11
- Next:
  - Consensus
  - Lynch, Chapter 12
Last time

- Mutual exclusion algorithms using read/write shared memory:
  - Dijkstra, Peterson, Lamport Bakery, Burns
- Mutual exclusion algorithms using read/modify/write (RMW) shared memory:
  - Trivial 1-bit Test-and-Set algorithm, Queue algorithm, Ticket algorithm
- Single-level shared memory
- But modern shared-memory multiprocessors are somewhat different.
- The difference affects the design of practical mutex algorithms.
Shared-memory multiprocessors

P1

P2

P3

P4

P5

Shared memory
Shared-memory multiprocessors

P₁ P₂ P₃ P₄ P₅

Network (bus)

Mem Mem Mem Mem Mem

Shared memory
Shared-memory multiprocessors
Shared-memory multiprocessors
Costs for shared-memory multiprocessors

- Memory access costs are non-uniform:
  - Next-level cache access is ~10x more expensive (time delay).
- Remote memory access produces network traffic.
  - Network bandwidth can be a bottleneck.
- Writes invalidate cache entries.
  - A process that wants to read must request again.
- Reads typically don’t invalidate cache entries.
  - Processes can share read access to an item.
- All memory supports multiple writers, but most is reserved for individual processes.
Memory operations

- Modern shared-memory multiprocessors provide stronger operations than just reads and writes.
- “Atomic” operations:
  - Test&Set: Write 1 to the variable, return the previous value.
  - Fetch & Increment: Increment the variable, return the previous value.
  - Swap: Write the submitted value to the variable, return the previous value.
  - Compare&Swap (CAS): If the variable’s value is equal to the first submitted value, then reset it to the second submitted value; return the previous value. (Alternatively, return T/F indicating whether the swap succeeded.)
  - Load-link (LL) and Store-conditional (SC): LL returns current value; SC stores a new value only iff no updates have occurred since the last LL.
Mutual exclusion in practice

• Uses strong, “atomic” operations, not just reads and writes:
  – Test&Set, Fetch&Increment, Swap, Compare&Swap (CAS), LL/SC

• Examples:
  – One-variable Test&Set algorithm
  – Ticket lock algorithm: Two Fetch&Increment variables.
  – Queue lock algorithms:
    • One queue with enqueue, dequeue and head.
    • Since multiprocessors do not support queues in hardware, implement this using Fetch&Increment, Swap, CAS.

• Terminology: Critical section called a “Lock”.
Spinning vs. blocking

- What happens when a process wants a lock (critical section) that is currently taken? Two possibilities:
  - Spinning:
    - The process keeps performing the trying protocol.
    - Our theoretical algorithms do this.
    - In practice, often keep retesting certain variables, waiting for some “condition” to become true.
    - Good if waiting time is expected to be short.
  - Blocking:
    - The process deschedules itself (yields the processor)
    - OS reschedules it later, e.g., when some condition is satisfied.
    - Monitors, conditions (See HS, Chapter 8).
    - Better than spinning if waiting time is long.

- Choice of spinning vs. blocking applies to other synchronization constructs besides locks, e.g., producer-consumer synchronization, barrier synchronization.
Our assumptions

- Spinning, not blocking.
  - Spin locks are commonly used, e.g., in OS kernels.
  - Assume critical sections are very short.
  - Processes usually hold only one lock at a time.
- No multiprogramming (one process per processor).
  - Processes are not “swapped out” while in the critical region, or while executing trying/exit code.
- Performance is critical.
  - Must consider caching and contention effects.
  - Unknown set of participants (adaptive).
Spin locks

• Test&Set locks
• Ticket lock
• Queue locks
  – Anderson
  – Graunke/Thakkar
  – Mellor-Crummey/Scott (MCS)
  – Craig-Landin-Hagersten (CLH)
• Adding other features
  – Timeout
  – Hierarchical locks
  – Reader-writer locks
• Note: No formal complexity analysis here!
Test&Set Locks

- Simple T&S lock, widely used in practice.
- Test-and-Test&Set lock, reduces contention.
- T&S with backoff.
Simple Test&Set lock

\textbf{lock}: \{0, 1\}; initially 0

\begin{align*}
\text{try}_i & \quad \text{exit}_i \\
\text{waitfor}(\text{test}\&\text{set}((\text{lock}) = 0) & \quad \text{lock} := 0 \\
\text{crit}_i & \quad \text{rem}_i
\end{align*}

- Simple.
- Low space cost (1 bit).
- But lots of network traffic if highly contended.

Many processes waiting for lock to become free.
Simple test&set lock

Network (bus)

P₁  P₂  P₃  P₄  P₅

1   -   -   -   -

Mem  Mem  Mem  Mem
Simple test&set lock

![Diagram of a simple test&set lock system with processes P1 to P5 connected to a network (bus) and memories Mem1 to Mem5.](image-url)
Simple test&set lock

Network (bus)

P1  P2  P3  P4  P5

Mem  Mem  Mem  Mem
Simple test&set lock

Network (bus)

P₁  P₂  P₃  P₄  P₅

reqX

Mem  Mem  Mem  Mem  Mem
Simple test&set lock

```
 P1  P2  P3  P4  P5
 --- --- --- --- ---
-    -    -    -    -
 1    -    -    -    -
--- --- --- --- ---
 Network (bus) 
--- --- --- --- ---
 Mem Mem Mem Mem Mem
```
Simple test&set lock

Network (bus)

\[
\begin{align*}
P_1 &\quad - \\
P_2 &\quad - \\
P_3 &\quad 1 \\
P_4 &\quad - \\
P_5 &\quad - \\
\end{align*}
\]

Mem Mem Mem Mem Mem
Simple test&set lock

- $P_1$
- $P_2$
- $P_3$
- $P_4$
- $P_5$

Network (bus)

- Mem
- Mem
- Mem
- Mem
Simple test&set lock

Diagram showing connections between processes and memory units:

- Process $P_1$ connects to memory
- Process $P_2$ connects to memory
- Process $P_3$ connects to memory
- Process $P_4$ connects to memory
- Process $P_5$ connects to memory

Connections labeled with 't&s' indicate test&set operations.
Simple test&set lock

\[ P_1 \quad P_2 \quad P_3 \quad P_4 \quad P_5 \]

Network (bus)

Mem

Mem

Mem

Mem

Mem
Simple test&set lock

Network (bus)

P₁ P₂ P₃ P₄ P₅

Mem Mem Mem Mem Mem
Simple test&set lock

P1 P2 P3 P4 P5

Network (bus)

Mem Mem Mem Mem Mem

reqX reqX
Simple test&set lock

P₁ P₂ P₃ P₄ P₅

Network (bus)

Mem Mem Mem Mem Mem

reqX reqX
Simple test&set lock

Network (bus)

P1  P2  P3  P4  P5

Mem  Mem  Mem  Mem

reqX
Simple test&set lock

Network (bus)

P1

P2

t&s

P3

P4

1

reqX

P5

Mem

Mem

Mem

Mem

Mem
Simple test&set lock

Network (bus)

P1  P2  P3  P4  P5

-  -  -  1  -

Mem  Mem  Mem  Mem

reqX

t&s  t&s  t&s
Simple test&set lock

Network (bus)

P1 → Mem
P2 ← t&s
P3 ← t&s
P4 → 1
P5 ← reqX
Simple test&set lock

Network (bus)

P₁

P₂

t&s

P₃

t&s

P₄

1
reqX

P₅

Mem

Mem

Mem

Mem

Mem
Simple test&set lock

Network (bus)

- t&s
- t&s
- 1

P₁
P₂
P₃
P₄
P₅

Mem
Mem
Mem
Mem
Mem
Simple test&set lock

Network (bus)

P₁  P₂  P₃  P₄  P₅

-    1    -    -    -

Mem  Mem  Mem  Mem
Simple test&set lock

Network (bus)

P₁  P₂  P₃  P₄  P₅

Mem  Mem  Mem  Mem
Simple test&set lock

Diagram:

- $P_1$
- $P_2$
- $P_3$
- $P_4$
- $P_5$

Network (bus)

- Mem
- Mem
- Mem
- Mem
- Mem
Simple test&set lock

Network (bus)

P_1  P_2  P_3  P_4  P_5

Mem  Mem  Mem  Mem
Simple test&set lock

P1  P2  P3  P4  P5

-    -    -    1    -

Network (bus)

Mem  Mem  Mem  Mem  Mem
Simple test&set lock

P₁ P₂ P₃ P₄ P₅

Network (bus)

Mem Mem Mem Mem Mem
Simple test&set lock

Network (bus)

P₁  P₂  P₃  P₄  P₅

w(0) - 1 - - -

Mem Mem Mem Mem Mem
Simple test&set lock

Network (bus)

P_1  P_2  P_3  P_4  P_5

Mem  Mem  Mem  Mem  Mem

reqX

1
Simple test&set lock

Network (bus)

P₁

P₂

P₃

P₄

P₅

0

Mem

Mem

Mem

Mem
Simple test&set lock

Network (bus)
Test-and-test\&set lock

- To help cope with high contention.
- Test-and-test\&set:
  - First “test” (read).
  - Then, if the value is favorable (0), attempt test\&set.
- Reduces network traffic (but it's still high!).
Test-and-test&set lock

\[ \begin{array}{ccccc}
P_1 & P_2 & P_3 & P_4 & P_5 \\
1 & 1 & 1 & 1 & 1 \\
\hline \\
Mem & Mem & Mem & Mem & Mem
\end{array} \]

Network (bus)
Test-and-test&set lock

P1  P2  P3  P4  P5

w(0)

1  1  1  1  1

Network (bus)

Mem  Mem  Mem  Mem  Mem
Test-and-test&set lock

Diagram showing a network (bus) with processes $P_1$ to $P_5$ connected to memory devices (Mem).
Test-and-test&set lock

P1

0

read

read

read

read

Network (bus)

Mem

Mem

Mem

Mem

Mem
Test-and-test&set lock

P1 P2 P3 P4 P5
0 0 0 0 0

Network (bus)

Mem Mem Mem Mem Mem
Simple Test&Set lock with backoff

• More help coping with high contention.
• Recall: Test-and-test&set
  – Read before attempting Test&Set
  – Reduces network traffic.
  – But it’s still high—especially when a cascade of requests arrives just after the lock is released.
• Test&Set with backoff
  – If Test&Set “fails” (returns 1), wait before trying again.
    • Makes success more likely.
    • Reduces network traffic (both read and write).
  – Exponential backoff seems to work best.
  – Obviates need for Test-and-test&set.
Ticket lock

\textbf{next}: integer; initially 0
\textbf{granted}: integer; initially 0

\texttt{try}_i\texttt{ exit}_i
\begin{align*}
\text{ticket} & := \texttt{f&i} (\textbf{next}) \\
\text{waitfor} (\textbf{granted} = \text{ticket}) & \quad \text{f&i} (\textbf{granted}) \\
\text{crit}_i & \\
\end{align*}

- Simple, low space cost, no bypass.
- Network traffic similar to Test-and-test&set (why?)
  - Not quite as bad, though.
- Can augment with backoff.
  - Proportional backoff seems best: delay depends on difference between ticket and granted.
  - Could introduce extra delays.
Queue Locks

• Processes form a FIFO queue.
  – Provides first-come first-serve fairness.
• Each process learns if its turn has arrived by checking whether its predecessor has finished.
  – Predecessor can notify the process when to check.
  – Improves utilization of the critical section.
• Each process spins on a different location.
  – Reduces invalidation traffic.
Several queue locks

• Array-based:
  – Anderson’s lock.
  – Graunke and Thakkar’s lock (skip this).

• Link-list-based:
  – Mellor-Crummey and Scott
  – Craig, Landin, Hagensten
Anderson’s array lock

slots: array[0..N-1] of { front, not_front };
    initially (front, not_front, not_front,..., not_front)
next_slot: integer; initially 0

try_i
    my_slot := f&i(next_slot)
    waitfor(slots[my_slot] = front)
crit_i

exit_i
    slots[my_slot] := not_front
    slots[my_slot+1] := front
rem_i

• Entries are either “front” or “not-front” (of queue).
  – Exactly one “front” (except for short interval in exit region).
• Tail of queue indicated by next_slot.
  – Queue is empty if next_slot contains front.
• Each process spins on its own slot, reducing invalidation traffic.
Anderson’s array lock

slots: array[0..N-1] of { front, not_front };
    initially (front, not_front, not_front,..., not_front)
next_slot: integer; initially 0

try_i
    my_slot := f&i(next_slot)
    waitfor(slots[my_slot] = front)
exit_i
    slots[my_slot] := not_front
    slots[my_slot+1] := front
rem_i

• Each process spins on its own slot, reducing invalidation traffic.
• Technicality: Separate slots should use different cache lines, to avoid “false sharing”.
• This code allows only N competitors ever. But Anderson allows wraparound:
Andersson's array lock

**slots**: array[0..N-1] of { front, not_front };
  initially (front, not_front, not_front,..., not_front)

**next_slot**: integer; initially 0

try_i
  my_slot := f&i(next_slot)
  if my_slot mod N = 0
    atomic_add(next_slot, -N)
  my_slot := my_slot mod N
  waitfor(slots[my_slot] = front)

exit_i
  slots[my_slot] := not_front
  slots[my_slot+1 mod N] := front
rem_i

- Wraps around to allow reuse of array entries.
- Still only N of competing processes at one time.
- High space cost: One location per lock per process.
Mellor-Crummey/Scott queue lock

- “...probably the most influential practical mutual exclusion algorithm of all time.” ---2006 Dijkstra Prize citation
- Each process has its own “node”.
  - Spins only on its own node, locally.
  - Others may write its node.
- Small space requirements.
  - Can “reuse” nodes for different locks.
  - Space overhead: $O(L+N)$, for $L$ locks and $N$ processes, assuming each process accesses only one lock at a time.
  - Can allocate nodes as needed (typically upon process creation).
- May spin on exit.
**Mellor-Crummey/Scott lock**

**node**: array[1..N] of [next: 0..N, wait: Boolean]; initially arbitrary

**tail**: 0..N; initially 0

try _i_

  **node**[i].next := 0
  pred := swap( **tail**, i)
  if pred ≠ 0
    **node**[i].wait := true
    **node**[pred].next := i
    waitfor(¬ **node**[i].wait)

crit _i_

exit _i_

  if **node**[i].next = 0
    if CAS( **tail**, i, 0) return
    waitfor( **node**[i].next ≠ 0)
  **node**[ **node**[i].next].wait := false
  rem _i_

- Use array to model nodes.
- CAS: Change value, return true if expected value found.
Mellor-Crummey/Scott lock

\[
\text{try}_i \quad \text{node}[i].\text{next} := 0 \\
\text{pred} := \text{swap}(\text{tail}, i) \\
\text{if pred} \neq 0 \\
\text{node}[i].\text{wait} := \text{true} \\
\text{node}[\text{pred}].\text{next} := i \\
\text{waitfor}(\neg \text{node}[i].\text{wait})
\]

\[
\text{exit}_i \quad \text{if node}[i].\text{next} = 0 \\
\quad \text{if CAS(\text{tail}, i, 0) return} \\
\quad \text{waitfor(node}[i].\text{next} \neq 0) \\
\quad \text{node[node}[i].\text{next]].\text{wait} := \text{false} \\
\text{rem}_i
\]
Mellor-Crummey/Scott lock

\( \text{try}_i \)

\[
\begin{align*}
\text{node}[i].\text{next} & := 0 \\
\text{pred} & := \text{swap}(\text{tail},i) \\
\text{if} \ pred \neq 0 \\
\text{node}[i].\text{wait} & := \text{true} \\
\text{node}[\text{pred}].\text{next} & := i \\
\text{waitfor}(\neg\text{node}[i].\text{wait})
\end{align*}
\]

\( \text{crit}_i \)

\( \text{tail} \)

\[
\text{node}[1]
\]

\( \text{exit}_i \)

\[
\begin{align*}
\text{if} \ \text{node}[i].\text{next} & = 0 \\
\text{if} \ \text{CAS}(\text{tail},i,0) \ \text{return} \\
\text{waitfor}(\text{node}[i].\text{next} \neq 0) \\
\text{node}[\text{node}[i].\text{next}].\text{wait} & := \text{false}
\end{align*}
\]

\( \text{rem}_i \)
Mellor-Crummey/Scott lock

\[
\text{try}_i \\
\text{node}[i].\text{next} := 0 \\
\text{pred} := \text{swap}(\text{tail},i) \\
\text{if} \ \text{pred} \neq 0 \\
\text{node}[i].\text{wait} := \text{true} \\
\text{node}[\text{pred}].\text{next} := i \\
\text{waitfor}(\neg \text{node}[i].\text{wait})
\]

\[
\text{exit}_i \\
\text{if} \ \text{node}[i].\text{next} = 0 \\
\text{if CAS}(\text{tail},i,0) \ \text{return} \\
\text{waitfor}(\text{node}[i].\text{next} \neq 0) \\
\text{node}[\text{node}[i].\text{next}].\text{wait} := \text{false}
\]

\[
\text{rem}_i
\]

\[
\text{crit}_i \\
\text{tail}
\]

\[
\text{node}[1]
\]

? \[/]
Mellor-Crummey/Scott lock

try\textsubscript{i}
\begin{align*}
\text{node}[i].\text{next} & := 0 \\
\text{pred} & := \text{swap(tail}, i) \\
\text{if pred} & \neq 0 \\
\text{node}[i].\text{wait} & := \text{true} \\
\text{node}[\text{pred}].\text{next} & := i \\
\text{waitFor(}\neg\text{node}[i].\text{wait})
\end{align*}

exit\textsubscript{i}
\begin{align*}
\text{if node}[i].\text{next} & = 0 \\
\text{if CAS(tail}, i, 0) \text{ return} \\
\text{waitFor(node}[i].\text{next} \neq 0) \\
\text{node[node}[i].\text{next}].\text{wait} & := \text{false} \\
\text{rem}_i
\end{align*}
Mellor-Crummey/Scott lock

\[
\begin{align*}
\text{try}_i & & \\
\text{node}[i].next & := 0 & \\
\text{pred} & := \text{swap}(\text{tail},i) & \\
\text{if pred} & \neq 0 & \\
\text{node}[i].wait & := \text{true} & \\
\text{node}[\text{pred}].next & := i & \\
\text{waitfor}(\neg \text{node}[i].wait) & & \\
\end{align*}
\]

\[
\begin{align*}
\text{exit}_i & & \\
\text{if node}[i].next & = 0 & \\
\text{if CAS(\text{tail},i,0) return} & \\
\text{waitfor(node}[i].next\neq 0) & \\
\text{node[node}[i].next].wait & := \text{false} & \\
\text{rem}_i & & \\
\end{align*}
\]
try\textsubscript{i}:
\begin{align*}
& \text{node}[i].next := 0 \\
& \text{pred} := \text{swap}(\text{tail}, i) \\
& \text{if pred} \neq 0 \\
& \quad \text{node}[i].wait := \text{true} \\
& \quad \text{node}[\text{pred}].next := i \\
& \quad \text{waitfor}(\neg \text{node}[i].wait)
\end{align*}

exit\textsubscript{i}:
\begin{align*}
& \text{if node}[i].next = 0 \\
& \quad \text{if CAS(tail, i, 0) return} \\
& \quad \text{waitfor(node}[i].next \neq 0) \\
& \quad \text{node[node}[i].next].wait := \text{false}
\end{align*}

rem\textsubscript{i}:

P\textsubscript{1} in C
Mellor-Crummey/Scott lock

try\textsubscript{i}
\textbf{node}[i].next := 0
pred := swap(tail, i)
if pred \neq 0
\textbf{node}[i].wait := true
\textbf{node}[pred].next := i
waitfor(\neg \textbf{node}[i].wait)

exit\textsubscript{i}
if \textbf{node}[i].next = 0
if CAS(tail, i, 0) return
waitfor(\textbf{node}[i].next \neq 0)
\textbf{node}[\textbf{node}[i].next].wait := false
rem\textsubscript{i}

P_1 \text{ in } C
try$_i$

\begin{align*}
\text{node}[i].next & := 0 \\
\text{pred} & := \text{swap}(\text{tail},i) \\
\text{if pred} & \neq 0 \\
\text{node}[i].wait & := \text{true} \\
\text{node}[\text{pred}].next & := i \\
\text{waitfor}(\neg \text{node}[i].wait)
\end{align*}

exit$_i$

\begin{align*}
\text{if node}[i].next & = 0 \\
\text{if CAS(tail},i,0) & \text{ return} \\
\text{waitfor}(\text{node}[i].next \neq 0) \\
\text{node[node}[i].next].wait & := \text{false} \\
\text{rem}_i
\end{align*}
Mellor-Crummey/Scott lock

try\textsubscript{i}
\begin{align*}
\text{node}[i].next &:= 0 \\
\text{pred} &:= \text{swap} \left( \text{tail}, i \right) \\
\text{if pred} &\neq 0 \\
\text{node}[i].wait &:= \text{true} \\
\text{node}[\text{pred}].next &:= i \\
\text{waitfor} \left( \neg \text{node}[i].wait \right)
\end{align*}

exit\textsubscript{i}
\begin{align*}
\text{if node}[i].next & = 0 \\
\text{if CAS} \left( \text{tail}, i, 0 \right) &\text{ return} \\
\text{waitfor} \left( \text{node}[i].next \neq 0 \right) \\
\text{node}[\text{node}[i].next].wait &:= \text{false} \\
\text{rem}_i
\end{align*}
Mellor-Crummey/Scott lock

\[
\text{try}_i
\]
\[
\text{node}[i].next := 0
\]
\[
pred := \text{swap}(\text{tail},i)
\]
\[
\text{if pred} \neq 0
\]
\[
\text{node}[i].wait := \text{true}
\]
\[
\text{node}[\text{pred}].next := i
\]
\[
\text{waitfor}(\neg \text{node}[i].wait)
\]

\[
\text{exit}_i
\]
\[
\text{if node}[i].next = 0
\]
\[
\text{if CAS}(\text{tail},i,0) \text{ return}
\]
\[
\text{waitfor}(\text{node}[i].next \neq 0)
\]
\[
\text{node}[\text{node}[i].next].wait := \text{false}
\]
\[
\text{rem}_i
\]
try_i
    node[i].next := 0
    pred := swap(tail,i)
    if pred ≠ 0
        node[i].wait := true
        node[pred].next := i
        waitfor(¬node[i].wait)

exit_i
    if node[i].next = 0
        if CAS(tail,i,0) return
        waitfor(node[i].next ≠ 0)
        node[node[i].next].wait := false

rem_i

P_1 in C    P_4 waiting

Mellor-Crummey/Scott lock
try \(_i\)
\[
\text{node}[i].\text{next} := 0 \\
\text{pred} := \text{swap}(\text{tail},i) \\
\text{if pred} \neq 0 \\
\quad \text{node}[i].\text{wait} := \text{true} \\
\quad \text{node}[\text{pred}].\text{next} := i \\
\quad \text{waitfor}(\neg \text{node}[i].\text{wait})
\]

exit \(_i\)
\[
\text{if node}[i].\text{next} = 0 \\
\quad \text{if CAS(}\text{tail},i,0) \text{ return} \\
\quad \text{waitfor}(\text{node}[i].\text{next} \neq 0) \\
\text{node[}\text{node}[i].\text{next}].\text{wait} := \text{false}
\]

rem \(_i\)
\[
\text{node}[1] \rightarrow \text{node}[4] \rightarrow \text{node}[3]
\]

P\(_1\) in C  
P\(_4\) waiting  
P\(_3\) waiting
Mellor-Crummey/Scott lock

try

node[i].next := 0
pred := swap(tail, i)
if pred ≠ 0
  node[i].wait := true
  node[pred].next := i
  waitfor(¬node[i].wait)

exit

if node[i].next = 0
  if CAS(tail, i, 0) return
  waitfor(node[i].next ≠ 0)
  node[node[i].next].wait := false
rem

P₄ waiting   P3 waiting
Mellor-Crummey/Scott lock

\[ \text{try}_i \]
\[
\text{node}[i].next := 0
\]
\[
\text{pred} := \text{swap(tail,} i)\]
\[
\text{if pred} \neq 0
\]
\[
\text{node}[i].wait := \text{true}
\]
\[
\text{node[pred].next} := i
\]
\[
\text{waitfor}(\neg \text{node}[i].\text{wait})
\]

\[ \text{exit}_i \]
\[
\text{if node}[i].\text{next} = 0
\]
\[
\text{if CAS(tail,} i, 0) \text{ return}
\]
\[
\text{waitfor(node}[i].\text{next} \neq 0)
\]
\[
\text{node[node}[i].\text{next}].\text{wait} := \text{false}
\]

\[ \text{rem}_i \]

\text{crit}_i

\text{tail}

\text{node[1]} \quad \text{node[4]} \quad \text{node[3]}

? \quad F \quad T

P4 waiting \quad P3 waiting
try\textsubscript{i} 
\begin{align*}
\text{node}[i].next & := 0 \\
\text{pred} & := \text{swap}(\text{tail}, \text{i}) \\
\text{if} \ \text{pred} & \neq 0 \\
\text{node}[i].\text{wait} & := \text{true} \\
\text{node}[\text{pred}].\text{next} & := \text{i} \\
\text{waitfor}(\neg \text{node}[i].\text{wait})
\end{align*}

exit\textsubscript{i} 
\begin{align*}
\text{if} \ \text{node}[i].\text{next} & = 0 \\
\text{if} \ \text{CAS}(\text{tail}, \text{i}, 0) \ \text{return} \\
\text{waitfor}(\text{node}[i].\text{next} \neq 0) \\
\text{node}[\text{node}[i].\text{next}].\text{wait} & := \text{false} \\
\text{rem}_i
\end{align*}
Mellor-Crummey/Scott lock

try_i
node[i].next := 0
pred := swap(tail,i)
if pred ≠ 0
    node[i].wait := true
    node[pred].next := i
    waitfor(¬node[i].wait)

exit_i
if node[i].next = 0
    if CAS(tail,i,0) return
    waitfor(node[i].next ≠ 0)
    node[node[i].next].wait := false

rem_i

Diagram:
- tail
- node[1]: ?
- node[4]: F
- node[3]: T

P_4 in C
P_3 waiting
Craig/Landin/Hagersten lock

**node**: array[0..N] of \{wait,done\}; initially all done  
**tail**: 0..N; initially 0

local to i: my_node: 0..N; initially i  

\[
\begin{align*}
\text{try}_i & \quad \text{exit}_i \\
\text{node}[\text{my_node}] := \text{wait} & \quad \text{node}[\text{my_node}] := \text{done} \\
\text{pred} := & \quad \text{my_node} := \text{pred} \\
\text{swap}(\text{tail},\text{my_node}) & \quad \text{rem}_i \\
\text{waitfor}(\text{node}[\text{pred}] = \text{done}) & \\
\text{crit}_i & \\
\end{align*}
\]

- Even simpler than MCS.  
- Has same nice properties, plus eliminates spinning on exit.  
- Not as good on cacheless architectures, since nodes spin on locations that could be remote.
Craig/Landin/Hagersten lock

**node**: array[0..N] of {wait, done}; initially all done

**tail**: 0..N; initially 0

local to i: my_node: 0..N; initially i

try
  node[my_node] := wait
  pred :=
  swap(tail, my_node)
  waitfor(node[pred] = done)
exit

• Queue structure information now distributed, not in shared memory.
• List is linked implicitly, via local pred pointers.
• Upon exit, processes acquire new node id (specifically, from predecessor).
Craig/Landin/Hagersten lock

**node**: array[0..N] of \{wait, done\}; initially all done

**tail**: 0..N; initially 0

local to i: my_node: 0..N; initially i

\[
\text{try}_i
\]

\[
\text{node}[\text{my\_node}] := \text{wait}
\]

\[
\text{pred} :=
\]

\[
\text{swap}(\text{tail}, \text{my\_node})
\]

\[
\text{waitfor}(\text{node}[\text{pred}] = \text{done})
\]

\[
\text{crit}_i
\]

\[
\text{exit}_i
\]

\[
\text{node}[\text{my\_node}] := \text{done}
\]

\[
\text{my\_node} := \text{pred}
\]

\[
\text{rem}_i
\]

**Diagram**

- Node [0] (d)
-尾节点 (tail)
- 状态为等待 (wait)
Craig/Landin/Hagersten lock

\textbf{node}: array[0..N] of \{wait,done\}; initially all done
\textbf{tail}: 0..N; initially 0

local to \textit{i}: my\_node: 0..N; initially \textit{i}

\begin{verbatim}
try_i
  node[my\_node] := wait
  pred :=
  swap(tail,my\_node)
  waitfor(node[pred] = done)
\end{verbatim}

\begin{verbatim}
exit_i
  node[my\_node] := done
  my\_node := pred
\end{verbatim}

\begin{verbatim}
rem_i
\end{verbatim}
Craig/Landin/Hagersten lock

\textbf{node}: array[0..N] of \{wait,done\}; initially all done
\textbf{tail}: 0..N; initially 0

local to i: \textbf{my\_node}: 0..N; initially i

\begin{align*}
\text{try}_i & \quad \textbf{node}[\text{my\_node}] := \text{wait} \\
\text{pred} & := \\
\text{swap}(\text{tail,my\_node}) \\
\text{waitfor}(\textbf{node}[\text{pred}] = \text{done}) \\
\text{crit}_i
\end{align*}

\begin{align*}
\text{exit}_i & \quad \textbf{node}[\text{my\_node}] := \text{done} \\
\text{my\_node} & := \text{pred} \\
\text{rem}_i
\end{align*}

\begin{tikzpicture}[scale=0.5]
  \node (node0) at (0,0) {node[0]};
  \node (node1) at (2,0) {node[1]};
  \node (tail) at (1,1) {tail};
  \draw[->] (node0) -- (tail);
  \draw[->] (node1) -- (tail);
  \node (d) at (0,-2) {d};
  \node (w) at (2,-2) {w};
\end{tikzpicture}
Craig/Landin/Hagersten lock

\textbf{node:} array[0..N] of \{wait, done\}; initially all done
\textbf{tail:} 0..N; initially 0

local to i: my_node: 0..N; initially i

\texttt{try_i, node[my_node] := wait}
\texttt{pred := swap(tail, my_node)}
\texttt{waitfor(node[pred] = done)}
\texttt{crit_i}

\texttt{exit_i, node[my_node] := done}
\texttt{my_node := pred}
\texttt{rem_i}
Craig/Landin/Hagersten lock

**node**: array[0..N] of {wait, done}; initially all done  
**tail**: 0..N; initially 0 

local to i: my_node: 0..N; initially i

\[
\begin{align*}
\text{try}_i & \\
\text{node}[\text{my_node}] & := \text{wait} \\
\text{pred} & := \\
\text{swap}(& \text{tail}, \text{my_node}) \\
\text{waitfor}(\text{node}[\text{pred}] = \text{done}) \\
\text{crit}_i & \end{align*}
\]

\[
\begin{align*}
\text{exit}_i & \\
\text{node}[\text{my_node}] & := \text{done} \\
\text{my_node} & := \text{pred} \\
\text{rem}_i & 
\end{align*}
\]

Graph:

- **node[0]**: d  
- **node[1]**: w  
- Pred: \text{pred}_1  
- Tail: \text{tail}  
- P_1 in C
Craig/Landin/Hagersten lock

\textbf{node}: array[0..N] of \{wait, done\}; initially all done
\textbf{tail}: 0..N; initially 0

local to i: my_node: 0..N; initially i

\textbf{try}_i

node[my_node] := wait
pred :=
swap(tail, my_node)
waitfor(node[pred] = done)

\textbf{exit}_i

node[my_node] := done
my_node := pred

rem_i

\begin{itemize}
  \item node[0]
  \begin{itemize}
    \item d
    \item pred_1
  \end{itemize}
  \item node[1]
  \begin{itemize}
    \item w
    \item Pred_4
  \end{itemize}
  \item node[4]
  \begin{itemize}
    \item w
    \item P_4 \text{ waiting}
  \end{itemize}
\end{itemize}

P_1 \text{ in C}
Craig/Landin/Hagersten lock

**node**: array[0..N] of {wait, done}; initially all done  
**tail**: 0..N; initially 0

local to i: **my_node**: 0..N; initially i

try i

\[
\text{node}[\text{my}_\text{node}] := \text{wait}
\]

\[
\text{pred} := \text{swap}(\text{tail}, \text{my}_\text{node})
\]

\[
\text{waitfor} (\text{node}[\text{pred}] = \text{done})
\]

exit i

\[
\text{node}[\text{my}_\text{node}] := \text{done}
\]

\[
\text{my}_\text{node} := \text{pred}
\]

rem i

\[
\text{pred} := \text{pred} \rightarrow 1
\]

\[
\text{crit}_i
\]

- **node[0]**: d  
  - pred<sub>1</sub>: d  
  - tail

- **node[1]**: d  
  - pred<sub>4</sub>: w

- **node[4]**: w

P<sub>4</sub> waiting
Craig/Landin/Hagersten lock

**node**: array[0..N] of {wait,done}; initially all done

**tail**: 0..N; initially 0

local to i: my_node: 0..N; initially i

**try**

- **node[my_node]** := wait
- **pred** :=
- **swap(tail,my_node)**
- **waitfor(node[pred] = done)**

**exit**

- **node[my_node]** := done
- **my_node := pred**

**rem**

- **rem**

![Diagram showing the state of the lock with nodes and their connections.](image-url)
Craig/Landin/Hagersten lock

node: array[0..N] of {wait, done}; initially all done
tail: 0..N; initially 0

local to i: my_node: 0..N; initially i

try_i
  node[my_node] := wait
  pred :=
  swap(tail, my_node)
  waitFor(node[pred] = done)
exit_i
node[my_node] := done
my_node := pred
rem_i
Craig/Landin/Hagersten lock

\textbf{node}: array[0..N] of \{wait, done\}; initially all done
\textbf{tail}: 0..N; initially 0

local to \(i\): my\_node: 0..N; initially \(i\)

\begin{align*}
\text{try}_i \\
\text{node}[\text{my\_node}] & := \text{wait} \\
\text{pred} & := \\
\text{swap}(\text{tail}, \text{my\_node}) \\
\text{waitfor}(\text{node}[\text{pred}] = \text{done}) \\
\text{crit}_i
\end{align*}

\begin{align*}
\text{exit}_i \\
\text{node}[\text{my\_node}] & := \text{done} \\
\text{my\_node} & := \text{pred} \\
\text{rem}_i
\end{align*}

\[P_4 \text{ in } C\]
Craig/Landin/Hagersten lock

**node**: array[0..N] of {wait, done}; initially all done

**tail**: 0..N; initially 0

local to i: my_node: 0..N; initially i

try$_i$

node[my_node] := wait

pred :=

swap(tail, my_node)

waitfor(node[pred] = done)

crit$_i$

exit$_i$

node[my_node] := done

my_node := pred

rem$_i$

---

**Diagram**

- **P$_1$ using node[0]**
- **P$_4$ in C**
- **P$_1$ waiting**
- **node[1]**
- **node[4]**
- **node[0]**
- **tail**

- d  → pred$_4$
- w  → pred$_1$
Additional lock features

- Timeout (of waiting for lock)
  - Well-formedness implies you are stuck once you start trying.
  - May want to bow out (to reduce contention?) if taking too long.
  - How could we do this?
    - Easy for test&set locks; harder for queue locks (and ticket lock).

- Hierarchical locks
  - If machine is hierarchical, and critical section protects data, it may be better to schedule “nearby” processes consecutively.

- Reader/writer locks
  - Readers don't conflict, so many readers can be “critical” together
  - Especially important for “long” critical sections.
Generalized Resource Allocation

• A very quick tour
• Lynch, Chapter 11
Generalized resource allocation

- Mutual exclusion: Problem of allocating a single non-sharable resource.
- Can generalize to more resources, some sharing.
- Exclusion specification $E$ (for a given set of users):
  - Any collection of sets of users, closed under superset.
  - Expresses which users are incompatible, can’t coexist in the critical section.

- Example: k-exclusion (any k users are okay, but not k+1)
  $E = \{ E : |E| > k \}$

- Example: Reader-writer locks
  - Relies on classification of users as readers vs. writers.
  $E = \{ E : |E| > 1 \text{ and } E \text{ contains a writer} \}$

- Example: Dining Philosophers (Dijkstra)
  $E = \{ E : E \text{ includes a pair of neighbors} \}$
Resource specifications

• Some exclusion specs can be described conveniently in terms of requirements for concrete resources.

• Resource spec: Different users need different subsets of resources
  – Can't share: Users with intersecting sets exclude each other.

• Example: Dining Philosophers (Dijkstra)
  \[ E = \{ E : E \text{ includes a pair of neighbors} \} \]
  Forks (resources) between adjacent philosophers; each needs both adjacent forks in order to eat.
  Only one can hold a particular fork at a time, so adjacent philosophers must exclude each other.

• Not every exclusion problem can be expressed in this way.
  – k-exclusion cannot.
Resource allocation problem, for a given exclusion spec $E$

- Same shared-memory architecture as for mutual exclusion (processes and shared variables, no buses, no caches).
- Well-formedness, as before.
- Exclusion: No reachable state in which the set of users in $C$ is a set in $E$.
- Progress: As before.
- Lockout-freedom: As before.
- But these don’t capture concurrency requirements.
  - Any lockout-free mutual exclusion algorithm also satisfies $E$ (provided that $E$ doesn’t contain any singleton sets).
- Can add concurrency conditions, e.g.:
  - Independent progress: If $i \in T$ and every $j$ that could conflict with $i$ remains in $R$, then eventually $i \rightarrow C$. (LTTR)
  - Time bound: Obtain better bounds from $i \rightarrow T$ to $i \rightarrow C$, even in the presence of conflicts, than we can for mutual exclusion.
Dining Philosophers

• Dijkstra’s paper posed the problem, gave a solution using strong shared-memory model.
  – Globally-shared variables, atomic access to all of shared memory.
  – Not very distributed.
• More distributed version: Assume the only shared variables are on the edges between adjacent philosophers.
  – Correspond to forks.
  – Use RMW shared variables.
• **Impossibility result**: If all processes are identical and refer to forks by local names “left” and “right”, and all shared variables have the same initial values, then we can’t guarantee DP exclusion + progress.
• **Proof**: Show we can’t break symmetry:
  – Consider subset of executions that work in synchronous rounds, prove by induction on rounds that symmetry is preserved. Then by progress, someone $\rightarrow C$. So all do, violating DP exclusion.
Dining Philosophers

- **Example:** Simple symmetric algorithm where all wait for R fork first, then L fork.
  - Guarantees DP exclusion, because processes wait for both forks.
  - But progress fails---all might get R, then deadlock.

- So we need something to break symmetry.

- **Solutions:**
  - Number forks around the table, pick up smaller numbered fork first.
  - Right/left algorithm (Burns):
    - Classify processes as R or L (need at least one of each).
    - R processes pick up right fork first, L processes pick up left fork first.
    - Yields DP exclusion, progress, lockout freedom, independent progress, and good time bound (constant, for alternating R and L).

- Generalize to solve any resource problem
  - Nodes represent resources.
  - Edge between resources if some user needs both.
  - Color graph; order colors.
  - All processes acquire resources in order of colors.
Next time

- Impossibility of consensus in the presence of failures.
- Reading: Lynch, Chapter 12
6.852J / 18.437J Distributed Algorithms
Fall 2009

For information about citing these materials or our Terms of Use, visit: [http://ocw.mit.edu/terms.](http://ocw.mit.edu/terms)