6.852 Lecture 15

- Pragmatic issues for shared-memory multiprocessors
- Practical mutual exclusion algorithms
  - test-and-set locks
  - queue locks
- Generalized exclusion/resource allocation problems
- Reading:
  - Mellor-Crummey and Scott paper (Dijkstra prize winner)
  - Magnussen, Landin, Hagersten paper
  - Chapter 11
Next time

• Consensus
• Reading: Chapter 12
Mutual exclusion with RMW

• Quick review
  – shared-memory multiprocessors provide “atomic operations”
    • test&set, fetch&increment, swap, compare&swap (CAS), LL/SC
  – in practice, all mutual exclusion algorithms use these operations
    • one-variable test&set algorithm
    • queue lock: one queue with enqueue, dequeue and head
      – multiprocessors do **not** support queues in hardware
    • ticket lock algorithm: two fetch&inc variables
A note on terminology

- Different usage in “systems” and “theory” communities
  - blocking: yields processor
  - atomic operation: some kind of read-modify-write operation
  - implement: provide specified functionality??
  - simulation: experiment, or running on a (hardware) simulator
  - process vs thread
  - locks vs mutual exclusion

- Different emphasis and concerns
  - mechanism vs. abstraction: processors, locks, blocking
  - performance issues: caching, contention, etc.
Mutual exclusion in practice

- What to do when lock is taken
  - “block”: deschedule process (yield processor)
    - OS reschedules it in future, often when some condition is satisfied
  - busy-wait/spin
    - don't yield process: repeatedly test for some condition
    - should be used only if waiting is expected to be very short

The choice of blocking vs spinning applies to other synchronization constructs, such as producer-consumer and barriers.
Mutual exclusion in practice

- Spin locks are very important
  - used in OS kernels
- Assume critical sections are very short
  - typically not nested (hold only one lock at a time)
- Performance is critical
  - must consider caching and contention effects
  - adaptive requirements/performance
Shared-memory multiprocessors

P₁  P₂  P₃  P₄  P₅

Shared memory
Shared-memory multiprocessors

Network (bus)

Mem | Mem | Mem | Mem

$ | $ | $ | $ | $
Shared-memory multiprocessors

Network (bus)
Shared-memory multiprocessors
Shared-memory multiprocessors

- Memory access does not have uniform cost
  - next-level cache access is ~10x more expensive
  - remote-memory access produces network traffic
    - network bandwidth can be bottleneck
  - writes invalidate caches
    - every processor that wants to read must request again
    - can typically share read access
  - all memory is multiwriter, but most is reserved for a process
Mutual exclusion in practice

- Critical sections are very short
  - typically hold only one lock at a time
  - critical processes are not swapped out
    - assume no multiprogramming for now (one thread per processor)
- Caching and contention are important
Practical 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
Simple test&set lock

\( \text{lock}: \{0,1\}; \text{initially } 0 \)

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

- Simple
- Low space cost
- But lots of network traffic if highly contended

many processes waiting for lock to become free
Simple test&set lock

- P1
- P2
- P3
- P4
- P5

1
-
-
-
-

1
-
-
-
-

Network (bus)

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

Network (bus)

P₁  P₂  P₃  P₄  P₅

1  -  -  -  -

reqX  Mem  Mem  Mem  Mem
Simple test&set lock

P_1

- 1 -

Network (bus)

Mem Mem Mem Mem
Simple test&set lock

Network (bus)

- 1 - -

Mem Mem Mem Mem
Simple test&set lock

Network (bus)

P₁  P₂  P₃  P₄  P₅

-  1  -  -  -

reqX  reqX  reqX

Mem  Mem  Mem  Mem
Simple test&set lock

P₁, P₂, P₃, P₄, P₅

Network (bus)

Mem, Mem, Mem, Mem
Simple test&set lock

Network (bus)

P_1 - Mem

P_2 - Mem

P_3 - Mem

P_4 1 Mem

P_5 - Mem

reqX
Simple test&set lock

Network (bus)

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

Network (bus)

P₁

-

Mem

P₂
t&s

- Mem

P₃
t&s

- Mem

P₄

- Mem

P₅

- 1 Mem
Simple test&set lock

- Network (bus)
  Mem
  Mem
  Mem
  Mem

P₁
-  P₂
  1  P₃
-  P₄
-  P₅
Simple test&set lock

Network (bus)

P₁  P₂  P₃  P₄  P₅

w(0)

Mem  Mem  Mem  Mem
Simple test&set lock

Network (bus)

- Mem
- Mem
- Mem
- Mem

reqX

- 1

P_1
P_2
P_3
P_4
P_5
Simple test&set lock

P1 -> 0
P2
P3
P4
P5

Network (bus)

Mem
Mem
Mem
Mem
Simple test&set lock

Mem

$P_1$

Mem

$P_2$

Mem

$P_3$

Mem

$P_4$

Mem

$P_5$

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

- dealing with high contention
  - test-and-test\&set
    - read before attempting test\&set
    - reduces network traffic (but it's still high!)
Test-and-test&set lock

Network (bus)

P_1 \quad P_2 \quad P_3 \quad P_4 \quad P_5

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

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

\[
\begin{align*}
\text{Mem} & \quad \text{Mem} & \quad \text{Mem} & \quad \text{Mem} & \quad \text{Mem} \\
\end{align*}
\]

Network (bus)

w(0)
Test-and-test&set lock

P_1
P_2
P_3
P_4
P_5
0
-
-
-
-

Network (bus)

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

Network (bus)

P₁

P₂

P₃

P₄

P₅

Mem

Mem

Mem

Mem

Mem

Mem

Mem

Mem

Mem

0

read

read

read

read

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

Network (bus)

Mem
Mem
Mem
Mem
Simple test&set lock with backoff

• dealing with high contention
  – test-and-test&set
    • read before attempting test&set
    • reduces network traffic (but it's still high!)
  – test&set with backoff
    • if test&set “fails” (returns 1), wait before trying again
      – reduces network traffic (both read and write)
    • exponential backoff seems to work best
    • obviates need for test-and-test&set
Ticket lock

next: integer; initially 0
granted: integer; initially 0

try
  ticket := f&i(next)
  waitfor(granted = ticket)
exit

- simple, low space cost, no bypass
- network traffic similar to test-and-test&set (why?)
  - not quite as bad though
- can use backoff: but delay potentially more costly
  - proportional backoff seems best
    - delay depends on difference between ticket and granted
Array-based queue locks

• Each process spins on a different location
  – reduces invalidation traffic
    • each entry in array must be in separate cache line
  – high space cost: one location (cache line) per lock per process
    • not adaptive
Anderson lock

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

**next_slot**: integer; initially 0

\[
\begin{align*}
\text{try}_i & : \quad \text{my_slot} := f&i(\text{next_slot}) \\
\text{waitfor}(\text{slots}[\text{my_slot}] = \text{front}) & \\
\text{crit}_i & \\
\text{exit}_i & : \quad \text{slots}[\text{my_slot}] := \text{not_front} \\
& \text{slots}[\text{my_slot}+1] := \text{front} \\
\text{rem}_i & 
\end{align*}
\]

- entries 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
Anderson lock

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

try
    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
    slots[my_slot] := not_front
    slots[my_slot+1 mod N] := front
rem

Graunke/Thakkar lock

**lockval**: array[1..N] of {0,1}; initially all 1
**tail**: (1..N, {0,1}); initially (X,0) (X means “don't care”)

```
try_i
(pred,locked) := swap(tail,(i,lockval[i]))
waitfor(lockval[pred] ≠ locked)
crit_i
```

```
exit_i
lockval[i] := 1-lockval[i]
rem_i
```

- each entry belongs to some process (single-writer)
  - contains a bit indicating whether in T or C, or done
  - meaning of bit toggles
- tail contains last process in queue and meaning of bit
  - could use pointer instead of process name for linked list
  - but can't use “node” for other purposes (why?)
Mellor-Crummey/Scott lock

“probably the most influential practical mutual exclusion algorithm of all time.” -- 2006 Dijkstra Prize citation

- each process has its own “node”
  - but others may write its node
  - spin only on local node (good for “cacheless” architectures)
- can “reuse” node for different locks (or free space)
  - space overhead: $O(L+N)$ or $O(L+kN)$, $k = \#\text{locks held at once}$
  - can allocate nodes as needed (typically thread creation)
- can 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)
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
```

- as with GT, use array to model nodes
- CAS: change value, return true if expected value found
  - alternatively, return value seen regardless
try\_i

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

exit\_i

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

\[
\begin{align*}
\text{try}_i & \quad \text{exit}_i \\
\text{node}[i].\text{next} & := 0 \\
\text{pred} & := \text{swap}(\text{tail},i) \\
\text{if pred} & \neq 0 \\
\text{node}[i].\text{wait} & := \text{true} \\
\text{node}[\text{pred}].\text{next} & := i \\
\text{waitfor}(\neg\text{node}[i].\text{wait}) & \\
\text{crit}_i & \\
\text{waitfor}(\neg\text{node}[i].\text{next} & \neq 0) & \\
\text{node}[\text{node}[i].\text{next}].\text{wait} & := \text{false} \\
\text{rem}_i & \\
\end{align*}
\]
Mellor-Crummey/Scott lock

try$_i$

node[i].next := \(0\)

pred := swap(tail,i)

if pred \(\neq 0\)

node[i].wait := true

node[pred].next := i

waitfor(\neg node[i].wait)

crit$_i$

exit$_i$

if node[i].next = \(0\)

if CAS(tail,i,0) return

waitfor(node[i].next \(\neq 0\))

node[node[i].next].wait := false

rem$_i$

P$_1$ in C
Mellor-Crummey/Scott lock

try\_i

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

crit\_i

tail

\begin{itemize}
\item \textbf{node}[1]
\item \textbf{node}[4]
\item P\_1 \text{ in } C
\item pred\_4
\end{itemize}

exit\_i

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

rem\_i
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*}$

crit\_i$

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

rem\_i
Mellor-Crummey/Scott lock

try_i

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

crit_i

tail

exit_i

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

P_1 \text{ in } C

pred_4
Mellor-Crummey/Scott lock

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

crit\(_i\):

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

\text{P}_1 \text{ in C} \quad \text{P}_4 \text{ waiting}
Mellor-Crummey/Scott lock

try\_i

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

crit\_i

tail

\textbf{node}[1] \rightarrow \textbf{node}[4] \rightarrow \textbf{node}[3]

P\_1 \text{ in } C \quad P\_4 \text{ waiting} \quad P\_3 \text{ waiting}
try\textsubscript{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].\text{wait})
\end{align*}

crit\textsubscript{i}:

exit\textsubscript{i}:

\begin{align*}
\text{if node}[i].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
\end{align*}

\text{tail}

\begin{tikzpicture}
\node[node] (node1) at (0,0) {node[1]};
\node[node] (nodeF) at (2,0) {node[4]};
\node[node] (nodeT) at (4,0) {node[3]};
\node[void] (tail) at (0,-1) {tail};
\draw (node1) -- (nodeF);
\draw (nodeF) -- (nodeT);
\end{tikzpicture}

P\textsubscript{4} waiting  P3 waiting
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*}
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} \\
\text{rem}_i
\end{align*}
crit$_i$

\begin{itemize}
\item \textbf{tail}
\end{itemize}
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  \hspace{1cm} \text{exit}_i
  pred := swap(tail,my_node)  \hspace{1cm} \text{node}[my_node] := done
  waitFor(\text{node}[\text{pred}] = \text{done})  \hspace{1cm} \text{my_node} := \text{pred}

\text{crit}_i

\text{rem}_i

- eliminates spinning on exit by looking at pred node
  - list is linked “backwards”  (only implicitly via local pred)
  - needs one node always at lock; take predecessor on exit
  - not good on cacheless architectures
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

1. \( \text{node}[\text{my_node}] := \text{wait} \)
2. pred := swap(tail, my_node)
3. waitfor(\( \text{node}[\text{pred}] = \text{done} \))

exit

1. \( \text{node}[\text{my_node}] := \text{done} \)
2. my_node := pred

rem

1. \( \text{node}[\text{0}] \)

2. tail

3. d
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

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

\texttt{try}_i

\hspace{1cm} \textbf{node}[\textbf{my\_node}] := \texttt{wait}
\hspace{1cm} \texttt{pred} := \texttt{swap(\texttt{tail}, \textbf{my\_node})}
\hspace{1cm} \texttt{waitfor(\textbf{node}[\texttt{pred}] = \texttt{done})}

\texttt{crit}_i

\texttt{exit}_i

\hspace{1cm} \textbf{node}[\textbf{my\_node}] := \texttt{done}
\hspace{1cm} \textbf{my\_node} := \texttt{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

\[
\text{try}_i
\]

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

\[
\text{crit}_i
\]

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

\[
\text{rem}_i
\]

- `node[0]`: d
- `node[1]`: w
- `pred`: pred₁
- `tail`
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

\[
\begin{align*}
\text{try}_i & \quad \text{exit}_i \\
\text{node}[\text{my\_node}] & := \text{wait} \\
\text{pred} & := \text{swap} (\text{tail}, \text{my\_node}) \\
\text{waitfor} & (\text{node}[\text{pred}] = \text{done}) \\
\text{crit}_i & \\
\text{rem}_i & \end{align*}
\]
Craig/Landin/Hagersten lock

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

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

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

\text{node[0]} \quad \text{node[1]} \quad \text{node[4]}

d \quad \text{pred}_1 \quad \text{pred}_4 \quad \text{w} \quad \text{P}_1 \text{ in C} \quad \text{P}_4 \text{ 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_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

```
d       pred_1  d       pred_4  w
P_4 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_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

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

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

\[
\text{try}_i \\
\quad \text{node}[\text{my_node}] := \text{wait} \\
\quad \text{pred} := \text{swap}(\text{tail}, \text{my_node}) \\
\quad \text{waitfor} (\text{node}[\text{pred}] = \text{done}) \\
\text{crit}_i \\
\]

\[
\text{exit}_i \\
\quad \text{node}[\text{my_node}] := \text{done} \\
\quad \text{my_node} := \text{pred} \\
\quad \text{rem}_i \\
\]

\[
\begin{align*}
\text{node}[1] & \quad \text{node}[4] \\
\text{d} & \quad \text{w} \\
\end{align*}
\]

P₄ 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\_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})

crit\_i

exit\_i

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

my_node := \text{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*}
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 can we do this?
    - easy for test&set locks; harder for queue locks (including 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

- Two ways to generalize mutual exclusion
  - resource spec: different users need different subsets of resources
    - can't share: users with intersecting sets exclude each other
  - exclusion spec: incompatible sets of users
    - more general (any resource spec can be written as exclusion spec)

- Sample problems
  - Dining Philosophers (Dijkstra)
  - k-exclusion (any k users okay, but not k+1)
  - reader/writer locks
    - need further generalization: distinguish different user operations
Generalized resource allocation

• Dining Philosophers
  – neighboring philosophers share a fork
  – need fork on both sides to eat
  – no one should starve
  – can't solve without some symmetry breaking (why?)
  – solutions:
    • number forks around the table; get “smaller” fork first
    • left-right algorithm

• Generalize to solve any resource allocation problem
  – nodes represent resources
  – edge between resources if some user needs both
  – color graph; order colors