6.852 Lecture 14 (continued)

• Mutual exclusion with read/write memory (continued)
  – Burns' algorithm
  – lower bound on number of registers

• Algorithms with read-modify-write operations
  – test-and-set locks; queue locks
  – pragmatic issues: contention, caching
  – practical algorithms (to be continued)

• Reading:
  – Chapter 10
  – Mellor-Crummey and Scott paper (Dijkstra prize winner)
  – Magnussen, Ladin, Hagersten paper
Next time

• Continue practical mutual exclusion algorithms
• Generalized resource allocation/exclusion problems
• Reading: Chapter 11
Space/memory considerations

• All previous algorithms use more than n variables
  – Bakery could use just n variables (why?)
• All but Bakery use multiwriter variables
  – these can be expensive to implement
• Bakery algorithm uses infinite-size variables
  – difficult to adapt to use finite-size variables
• Can we do better?
Burns' algorithm

- Uses $n$ single-writer binary variables
- Simple
- Guarantees safety (mutual exclusion) and progress
  - but not starvation-freedom!
Burns' algorithm

try_i

L: for j = 1 to i-1 do
    if flag(j) = 1 then goto L
flag(i) := 1
for j = 1 to i-1 do
    if flag(j) = 1 then
        flag(i) := 0
        goto L
exit_i
flag(i) := 0
rem_i

M: for j = i+1 to n do
    if flag(j) = 1 then goto M
crit_i

minor change from book
Burns' algorithm

• Mutual exclusion:
  – if two processes in critical section simultaneously, who set flag to 1 (for the last time) first?

• Progress:
  – assume fair execution (everyone trying keeps taking steps)
  – if someone trying but no one is ever subsequently critical, someone eventually reaches M (why?)
  – anyone reaching M never falls back
  – someone who reaches M eventually becomes critical (why?)
Lower bound on registers

• Can we use fewer than n registers?
  – not if single-writer (why?)
  – not even if multiwriter!
Lower bound on registers

- Need at least 2 registers (if n > 1): by contradiction
  - before entering C, a process must write shared register
    * otherwise, no one else would know it entered C
  - run one process solo until just before it writes shared register
    * process covers the register
  - run second process until it enters C
    * can do so because it can't tell first process has run at all
  - continue first process, overwriting shared register
    * no more evidence of second process in C
    * first process enters C (contradicting mutual exclusion!)
Lower bound on registers

• Need at least 3 registers (if n > 2)?
  – run first process solo until just before it writes a register (x)
  – run second process until just before it writes other register (y)
    • must do so, or else run till enter C, then run first process, as before
  – run third process until it enters C...
Lower bound on registers

• Need at least 3 registers (if n > 2)?
  – run first process solo until just before it writes a register (x)
  – run second process until just before it writes other register (y)
    • must do so, or else run till enter C, then run first process, as before
  – run third process until it enters C...
  may see that second process wrote x, and so not enter C
Lower bound on registers

- Need at least 3 registers (if n > 2)?
  - run first process solo until just before it writes a register (x)
  - run second process until just before it writes other register (y)
    - must do so, or else run till enter C, then run first process, as before
  - run third process until it enters C...

  may see that second process wrote x, and so not enter C

Need some way to get two processes to cover both registers in a state indistinguishable from an idle state to a third process
Lower bound on registers

- Idea: one process acquires lock three times
  - at least two times, first register (x) written is the same
  - use first time to get second process to cover other register (y)
  - then acquire lock and return to apparently idle state
  - then cover x again

```
1 first covers a register (x)  
1 covers a register (y?)  
1 covers a register (x?)  
1st covers C,E,R,T\[\text{C,E,R,T}\]  
2 first covers y  
```
Lower bound on registers

- Idea: one process acquires lock three times
  - at least two times, first register (x) written is the same
  - use first time to get second process to cover other register (y)
  - then acquire lock and return to apparently idle state
  - then cover x again
Lower bound on registers

- **Lemma 1:** Process i can reach C from any (reachable) idle state s (and any states indistinguishable to i) without any steps by other process.
  - by progress condition

- **Lemma 2:** If execution fragment $\alpha$ has only steps of i and i starts in R and ends in C, then i writes some shared register not covered by any other process.
  - otherwise other processes can eliminate any evidence of i
  - one of them must enter C (by progress)
  - contradicts mutual exclusion (because i also in C)
Lower bound on registers

• Defn: s' is **k-reachable** from s if there is an exec frag from s to s' involving only steps by procs 1 to k.

• Lemma 3: For any $k \in [1,n-1]$ and from any idle state, there is a k-reachable state in which procs 1 to k cover k distinct shared registers and that is indistinguishable to procs k+1 to n from some k-reachable idle state.
  
  – By induction on k.
  – Base case (k=1):
    • run proc 1 until just before it writes first shared register
Lower bound on registers

- Lemma 3: For any $k \in [1, n-1]$ and from any idle state, there is a $k$-reachable state in which procs 1 to $k$ cover $k$ distinct shared registers and that is indistinguishable to procs $k+1$ to $n$ from some $k$-reachable idle state.

  - Inductive step: Assume lemma for $k < n-1$; prove for $k+1$.
    - Let $t_1$ be state guaranteed by inductive hypothesis.
    - Let each process from 1 to $k$ take a step, overwriting covered register.
    - Run all processes 1 to $k$ until each is in $R$; resulting state $u_1$ is idle.
    - Repeat, generating $t_2$, $u_2$, $t_3$, $u_3$, etc., until we get $t_i$ and $t_j$ ($i < j$) that cover same set $X$ of registers (why is this guaranteed to terminate?)
    - Run $k+1$ alone from $t_i$ until just before it writes a register not in $X$.
    - Run all processes 1 to $k$ as if from $t_i$ to $t_j$ (they can't tell the difference)
    - Result indistinguishable from $t_j$ (and thus the idle state) to procs $k+2$ to $n$. 
Lower bound on registers

- **Lemma 1:** Process i can reach C from any (reachable) idle state s (and any states indistinguishable to i) without any steps by other process.

- **Lemma 2:** If execution fragment has only steps of i and i starts in R and ends in C, then i writes some shared register not covered by any other process.

- **Lemma 3:** For any $k \in [1,n-1]$ and from any idle state, there is a k-reachable state in which procs 1 to k cover k distinct shared registers and that is indistinguishable to procs k+1 to n from some k-reachable idle state.

- **Theorem:** Any algorithm that solves n-process mutual exclusion with only read/write shared registers needs at least n of them.
  
  - By Lemma 3 from initial state, get state in which n-1 registers are covered and is indistinguishable from idle state to n.
  
  - By Lemma 1, n can reach C from this state (in which n is in R).
  
  - By Lemma 2, n must write some register not covered.
What lower bounds are good for

- At Bell Labs (several years ago), Gadi Taubenfeld found out Unix group was trying to develop an asynchronous mutual exclusion algorithm that used only a few r/w shared registers. He told them it was impossible.

- New research direction: Develop “space-adaptive” algorithms that potentially use many variables, but use few if only few processes are active (or “contend”).

- Also “time-adaptive” algorithms.

- In practice, this often means you can get much better performance/lower overhead.
Mutual exclusion with RMW

• Stronger memory primitives
  – all modern architectures provide one or more of these
    • called “synchronization primitives” or “atomic primitives”
    • typically expensive compared to reads and writes
      – but atomic reads and writes are also expensive
    • variables can also be read and written
  – not all the same strength: we'll come back to this in 2 weeks
  – does it enable better algorithms?
Mutual exclusion with RMW

- Test-and-set algorithm (trivial)
  - test-and-set: sets value to 1, returns previous value
    - usually on binary variables
  - one variable, 0 when unlocked (initial state), 1 when locked
  - to acquire lock, repeatedly test-and-set until get 0
  - to release lock, set variable to 0
  - no fairness

\[
\begin{align*}
\text{try}_i & \quad \text{exit}_i \\
\text{waitfor}(\text{test-and-set}(x) = 0) & \quad x := 0 \\
\text{crit}_i & \quad \text{rem}_i
\end{align*}
\]
Mutual exclusion with RMW

• Queue lock
  – shared variable: Q: a FIFO queue
    • supports enqueue, dequeue, head operations
    • very big variable!
  – to acquire lock, add self to queue, wait until you're at head
  – to release lock, remove self from queue
  – guarantees bounded bypass (indeed, no bypass)

\[
\text{try}_i \\
\quad \text{enqueue}(Q, i) \\
\quad \text{waitfor}(\text{head}(Q) = i) \\
\text{crit}_i \\
\text{exit}_i \\
\quad \text{dequeue}(Q) \\
\quad \text{rem}_i
\]
Mutual exclusion with RMW

- Ticket lock
  - like Bakery algorithm: get a number, wait till it's your turn
    - guarantees bounded bypass (indeed, no bypass)
  - shared variables: next, granted: integers, initially 0
    - supports fetch-and-increment (f&i)
  - to acquire lock, increment next, wait till granted
  - to release lock, increment granted

\[
\begin{align*}
\text{try}_i & \quad \text{exit}_i \\
\text{ticket} & := f\&i(\text{next}) \quad \text{f&i}(\text{granted}) \\
\text{waitfor}(\text{granted} = \text{ticket}) & \quad \text{rem}_i \\
\text{crit}_i & \\
\end{align*}
\]
Mutual exclusion with RMW

• Ticket lock
  – like Bakery algorithm: get a number, wait till it's your turn
    • guarantees bounded bypass (indeed, no bypass)
  – shared variables: next, granted: integers, initially 0
    • can we make these bounded in size? what bound?

\[
\begin{align*}
\text{try}_i & \\
\text{ticket} & := f\&i(\text{next}) \\
\text{waitfor}(\text{granted} = \text{ticket}) & \\
\text{crit}_i & \\
\text{exit}_i & \\
f\&i(\text{granted}) & \\
\text{rem}_i & 
\end{align*}
\]
Mutual exclusion with RMW

• How small can we make the RMW variable?
  - one bit if only require progress (test-and-set algorithm)
  - $\Theta(n)$ values ($\Theta(\log n)$ bits) for bounded bypass
    • actually we know at least $n$ values; can do in $n+k$ for small $k$
  - for starvation-freedom, it's harder:
    • lower bound of about $\sqrt{n}$
    • algorithm for $n/2 + k$, for small $k$

In practice, on a real shared-memory multiprocessor, we want few variables of size $O(\log n)$. So ticket algorithm is pretty good (in terms of space).