Mutual Exclusion Problem

• There are \( n \) concurrent processes \( \{P_0, P_1, \ldots, P_n\} \).

• Each process accesses one common resource, but cannot be allowed to access it at the same time
  – Serialize the access to the shared resource

• Shared resource could be
  – Change a common variable
  – Update a table, write to a file
  – Access a printer
Illustration: Producer-Consumer

While (count == BUF); // do nothing
buffer[in] = item;
in = (in+1) mod BUF
++count;

While (count == 0); // do nothing
item = buffer[out];
out = (out+1) mod BUF
--count;

Producer

reg₁ = count;
reg₁ = reg₁ + 1
count = reg₁

Consumer

reg₂ = count;
reg₂ = reg₂ - 1
count = reg₂

T0: prod: reg₁ = count [reg₁ = 5]
T1: prod: reg₁ = reg₁ + 1 [reg₁ = 6]
T2: cons: reg₂ = count [reg₂ = 5]
T3: cons: reg₂ = reg₂ - 1 [reg₂ = 4]
T4: prod: count = reg₁ [count = 6]
T5: cons: count = reg₂ [count = 4]

In order to get the correct result, one process must manipulate the shared variable at a time ➔ need some form of synchronization among the processes
Model Assumptions

• Concurrent systems can communicate using:
  – Shared memory (shared variables)
  – Message passing

• Type of shared variable defines the operations that can be performed atomically
  – Read/write shared variables:
    • Read: Process i reads register x and uses x to modify the process state
    • Write: process i writes a value to register x
  – Read-modify-write shared variables:
    • Read x ;
    • computation, possibly using x to determine new value of x;
    • Write the new value to x

• The events are:
  – Asynchronous ; No Failure
Mutual Exclusion (mutex) Problem

- Each processor's code is divided into four sections:
  - **entry**: synchronize with others to ensure mutually exclusive access to the resource
  - **critical**: use some resource; when done, enter the...
  - **exit**: clean up; when done, enter the...
  - **remainder**: not interested in using the resource
Mutual Exclusion Algorithm

• ME algorithm specifies code for entry and exit sections to ensure:
  – Mutual exclusion ➔ at most one process is in its critical section at any time
  – progress conditions are maintained in fair execution
    • No deadlock: if a process is in its entry section, then later some process is in its critical section
    • No lockout: if a process is in its entry section, then later the same process is in its critical section
    • Bounded waiting: no lockout + while processor is waiting, others enter the critical section only a bounded number of times
Outline: Mutex Algorithms

• Mutex algorithms using multi-writer/multi-reader shared registers
  – Dijkstra’s Algorithm
  – Peterson’s Algorithm
    • 2-process
    • N-process
    • Tournament

• Mutex algorithms using “single-writer/multiple reader shared registers
  – Lamport’s Bakery algorithm
Dijkstra’s Mutex Algorithm

**Shared variables:**

- \(\text{turn} \in \{1, \ldots, n\}\), initially arbitrary, writable and readable by all processes
- For every \(i, 1 \leq i \leq n\):
  - \(\text{flag}(i) \in \{0, 1, 2\}\), initially 0, writable by process \(i\) and readable by all processes

**Process \(i\):**
// remainder R region

\[T_i: \]

\[L: \text{flag}(i) = 1\]
- while \(\text{turn} \neq i\) do
  - if \(\text{flag}(\text{turn}) = 0\), then \(\text{turn} = i\);
  - \(\text{flag}(i) = 2\)
- for \(j \neq i\) do
  - if \(\text{flag}(j) = 2\), then goto \(L\)

**Crit\(_i\)**

**Exit\(_i\)**

\(\text{flag}(i) = 0\)

**rem\(_i\)**
Dijkstra’s Mutex Algorithm

• Stage 1:
  – Set flag = 1 \(\rightarrow\) intent to execute critical section
  – Check if turn = i; if not, then set turn = i, o/w Keep checking
    • Turns current owner is observed to be inactive

• Stage 2:
  – Set flag = 2;
  – Check that no other process has flag = 2
  – If check completes successfully, move to C; o/w back to Stage 1

• Exit (E) phase:
  – flag = 0;
Observations

• Are there multiple steps hidden in each pseudo-code instruction
  – Testing flag(turn)
  – Checking for all other processes’ flag ... keeping track of the processes already checked

• Does it work?
  – Prove that algorithm satisfies mutual exclusion
  – Prove that algorithm guarantees progress

• Upper bound in time: time from any point in execution when some process is in $T$ and no one in $C$, until someone enters $C$
  – Assume upper bound $k$ on time between successive steps in process
  – Assume upper bound $c$ on max time any process spends in $C$
  – Some user enters $C$ within time $O(kn)$
Limitation of Dijkstra’s Algorithm

• Is critical region access granted fairly to all users?
  – Fairness: no process is starved, or locked out
  – What if contention is rare?

• It uses multi-writer/multi-reader register (turn)
  – Difficult to implement in practice in multi-processor systems
Peterson’s 2P algorithm

Shared variables:
\( turn \in \{0, 1\} \), initially arbitrary, writable and readable by all processes for every \( i \in \{0, 1\} \):
\( flag(i) \in \{0, 1\} \), initially 0, writable by \( i \) and readable by \( \overline{i} \)

Process \( i \):
// remainder R region

Try\(_i\) T\(_i\):
\( flag(i) = 1 \)
\( turn = i; \)
wait-for flag \((j) = 0 \) or turn \( \neq i \)

Crit\(_i\)

Exit\(_i\)
\( flag(i) = 0 \)
rem\(_i\)
Observation on Petersons’ 2P

• Try block
  – Sets $\text{flag}(i) := 1$, sets $\text{turn} := i$.
  – Waits for either $\text{flag}(1-i) = 0$ or $\text{turn} \neq i$.
    
    Other process not active.  Other process has the $\text{turn}$ variable.

• Is it lockout free ?
  – Bounded waiting for a process waiting to enter C
Proof Sketch on No-Lockout

**Note:** $P_1 : \text{turn} = 1$, $P_2 : \text{turn} = 0$

L1: $P_1$ is in $T$ after setting flag(1) = 1.

L2: Assume $P_2$ enters $C$ **three** times

L3: On second and third times in order to enter $C$,
      $P_2$ must set "turn = 0", and then wait-for (and must get) "turn = 1"

L4: flag(1) = 0 is not possible because of L1

L5: L3 & L4 $\Rightarrow$ $P_1$ has set "turn = 1" **twice** (since $P_2$ never sets turn to 1)

L6: But $P_1$ has set "turn = 1" only once while entering $T$ region.

L7: Hence this is a contradiction.

**Note:** Allows Bounded-bypass
Petersons’ n-Process Algorithm

• Use the 2P algorithm iteratively to determine a winner among n process
  – series of n-1 competitions at levels 1, 2, ..., n-1

• At each step, ensure at least one loser
• At level 1, there are n-1 winners
• At level k, there are n-k winners
• At level n-1, only 1 process can win
Tournament Algorithm

- Use the 2-process algorithm as building block by organizing the n processes in a tree.
- Each process is at leaf.
- $\log_2 n$ competitions for each proc.
- Each process i has
  - Unique competition at each level.
Bakery Algorithm

- Uses single-writer/multi reader shared registers
- The algorithm works even if a read overlaps a write (concurrent read/write), and gets arbitrary value
  - e.g. while write is changing a value from 0 to 1, a read may get a value 3456.
Bakery Algorithm

Shared variables:
For every $i$, $1 \leq i \leq n$:
- $\text{choosing}(i) \in \{0,1\}$, init 0, writable by proc $i$, readable by all proc $j \neq i$
- $\text{number}(i) \in \mathbb{N}$, init 0, writable by $i$ and readable by all proc $j \neq i$

Process $i$:
// remainder R region

Try$_i$ $T_i$:
- $\text{choosing}(i) = 1$
- $\text{number (i)} = 1 + \max_{j \neq i} \text{number}(j)$
- $\text{choosing}(i) = 0$
- for $j \neq i$ do
  - wait-for $\text{choosing}(j) = 0$
  - wait-for $\text{number}(j) = 0$ or $(\text{number}(i),i) < (\text{number}(j),j)$

Crit$_i$

Exit$_i$
- $\text{number}(i) = 0$

rem$_i$
Explanation & Observations

Works like taking tickets in a bakery (or bank)

- choosing[i] = 1: signal to other processes that processor i is choosing a number
- number[i] = 0 implies that processor i is in remainder code
- number[i] != 0 implies that processor i is either in critical section or at the entry point

Can two processes choose the same number? break the tie in favor of process with smaller index

- Guarantees high level fairness ➔ if process i completes the choosing phase (called doorway) before process j enters T, then j cannot enter C before i does.
- Does it guarantee FIFO processing with respect to time of entry? NO
Number of Registers: ME algos

- Dijkstra: n+1
- Peterson 2P: 2 + 1
- Peterson NP: n + 1
- Bakery: 2n

What is the lower bound on the number of registers to solve ME?

**Theorem:** If algorithm A solves the ME problem for $n \geq 2$ processes, using only read-write shared variables, then A must use at least n shared variables.
**ME using Read-modify-write (RMW)**

- Supports the following general operation:
  
  \[
  \text{RMW}(V, f):
  \]
  
  \[
  \begin{align*}
  \text{temp} &= V; \\
  V &= f(V); \\
  \text{return temp;}
  \end{align*}
  \]

- Mutual exclusion can be achieved with **one** RMW variable
- Conceptually, the list of waiting processors is stored in a circular queue of length “n”
- Each waiting processor remembers in its local state:
  - Its own location in the queue
  - Does not need a shared variable for this purpose
Illustration

The RMW shared object just contains these two "pointers"

Counting semaphore ?
ENTRY code:
// increment last to enqueue self
position := rmw ( V, (V.first, V.last+1)
// wait until first equals this value
repeat
    queue := rmw(V,V)
until (queue.first = position.last)

EXIT code:
// dequeue self
rmw (V, (V.first+1, V.last))