SHARED MEMORY MUTUAL EXCLUSION

Shared memory model is implemented in operating systems through semaphoresmonitors and atomically executable special purpose hardware.

Lamport's bakery algorithm

- Lamport proposed the classical bakery algorithm for n-process mutual exclusion inshared memory systems.
- This algorithm satisfies the requirements of the critical section problem namelymutual exclusion, bounded waiting, and progress.
- All process threads must take a number and wait their turn to use a sharedcomputing resource or to enter their critical section.
- The number can be any of the global variables, and processes with the lowestnumber will be processed first.
- If there is a tie or similar number shared by both processes, it is managed throughtheir process ID.
- If a process terminates before its turn, it has to start over again in the processqueue.
- A process wanting to enter the critical section picks a token number that is onegreater than the elements in the array choosing [1…n].
- Processes enter the critical section in the increasing order of the token numbers.
- In case of concurrent accesses to choosing by multiple processes, the processesmay have the same token number.
- Then, a unique lexicographic order is defined on the tuple (token, pid) and thisdictates the order in which processes enter the critical section.

(shared vars)

boolean: choosing[1…n];

integer: timestamp[1….n];

repeat

(1) P_i executes the following for the entry section:

(1a) choosing[i] ← 1;

(1b) timestamp[i] ←$\max_{k\in[1...n]}$(timestamp[k]) + 1;

(1c) choosing[i] ← 0;

(1d) for count = 1 to n do

(1e) while choosing[count] do no-op;

(1f) while timestamp[count] ≠0 and (timestamp[count], count)

<(timestamp[i], i) do

(1g) no-op.

(2) P_i executes the critial section (CS) after the entry section

(3) P_i executes the following exit section after the CS:

(3a) timestamp[i] ← 0

(4) P_i executes the remainder section after the exit section until false;

until false;

Fig : Lamport's Bakery algorithm for shared memory exclusionMutual exclusion

- In the entry section, a process chooses a timestamp for itself, and resets it to 0when it leaves the exit section.
- These steps are non-atomic in the algorithm. Thus multiple processes could bechoosing timestamps in overlapping durations.
- When process i reaches line 1d, it has to check the status of each other process j, todeal with the effects of any race conditions in selecting timestamps.
- In lines 1d–1f, process i serially checks the status of each other process j.
- If j is selecting a timestamp for itself, j's selection interval may have overlappedwith that of i, leading to an unknown order of timestamp values.
- Process i needs to make sure that any other process j(j < i) that had begun to execute line 1b concurrently with itself and may still be executing line 1b does notassign itself the same timestamp.

- If this is not done mutual exclusion could be violated as i would enter the CS, andsubsequently, j, having a lower process identifier and hence a lexicographically lower time stamp, would also enter the CS.
- The i waits for j's timestamp to stabilize, i.e., choosing [j] to be set to false.
- Once j's timestamp is stabilized, i moves from line 1e to line 1f.
- Either j is not requesting or j is requesting. Line 1f determines the relative prioritybetween i and j.
- The process with a lexicographically lower timestamp has higher priority and enters the CS; the other process has to wait (line 1g).
- Thus mutual exclusion is satisfied by the algorithm.

Bounded Waiting

- Bounded waiting is satisfied because each other process j can overtake process i atmost once after i has completed choosing its timestamp.
- The second time j chooses a timestamp, the value will necessarily be larger than i's timestamp if i has not yet entered its CS.

Progress

- Progress is guaranteed because the lexicographic order is a total order and the process with the lowest timestamp at any time in the loop is guaranteed to enter theCS.

Improvements in Lamport's Bakery Algorithm

i) Space complexity

- A lower bound of n registers, specifically, the timestamp array, has been shown forthe shared memory critical section problem.

ii) Time complexity

- When the level of contention is low, the overhead of the entry section does not scale.
- This issue is addressed his concern is addressed by fast mutual exclusion with O(1).
- The limitation of this approach is that it does not guarantee bounded delay.

Lamport's WRWR mechanism and fast mutual exclusion

- This algorithm illustrates an important technique – the (W − R − W – R) sequencethat is a necessary and sufficient sequence of operations to check for contention and to ensure safety in the entry section, by employing just two registers.
- The basic sequence of operations for W(x)–R(y)–W(y)–R(x):
  1. The first operation needs to be a Write to x. If it were a Read, then all contending processes could find the value of the variable even outside the entrysection.
  2. The second operation cannot be a Write to another variable, for that could equally be combined with the first Write to a larger variable. The second operation should not be a Read of x because it follows Write of x and if there isno interleaved operation from another process, the Read does not provide anynew information. So the second operation must be a Read of another variable,say y.
  3. The sequence must also contain Read(x) and Write(y) because there is no pointin reading a variable that is not written to, or writing a variable that is never read.
  4. The last operation in the minimal sequence of the entry section must be a Read,as it will help determine whether the process can enter CS. So the last

operation should be Read(x), and the second-last operation should be the Write(y).

(shared variable among the processes)

```
integer: x, y;                          // shared register initialized
boolean b[1….n];     //flags to indicate interest in critical section
repeat
(1) Pi(1 ≤i ≤n) executes entry section:
(1a)        b[i] ← true;
(1b)        x ← i;
(1c)        if y ≠0 then
(1d)              b[i] ← false;
(1e)              await y=0;
(1f)              goto(1a);
(1g)        y ← i;
(1h)        if x ≠i then
(1i)              b[i] ← false;
(1j)              for j = 1 to n do
(1k)                    await y = 0;
(1l)              if y ≠i then
(1m)                    await y = 0;
(1n)                    goto(1a);
(2) Pi(1 ≤i ≤n) executes entry section:
(3) Pi(1 ≤i ≤n) executes exit section:
(3a)        y ← 0;
(3b)        b[i] ← false
Forever.
```

Fig : Lamport's fast mutual exclusion algorithm

Hardware Support for Mutual Exclusion

- Hardware support can allow for special instructions that perform two ormore operations atomically.
- Two such instructions, Test &Set and Swap are defined and implemented.
- The atomic execution of two actions, a Read and a Write operation can simplify a mutual exclusion algorithm.

(shared variables among the processes accessing each of the different object types)
register: Reg ← initial value; // shared register initialized
(local variables)
integer: old ← initial value; // value to be returned
(1) Test & Set(Reg) return value:
(1a) old ← Reg;
(1b) Reg ← 1;
(1c) return(old).
(2) Swap(Reg, new) return value:
(2a) old ← Reg;
(2b) Reg ← new;
(2c) return(old).

Fig : Definitions for Test&Set, Swap operations

(shared variables)
register: Reg ← false; // shared register initialized
(local variables)
integer: blocked ← 0 // variable to be checked before entering CS
repeat
(1) P_i executes the following for the entry section:
(1a) blocked ← true;
(1b) repeat
(1c) blocked ← Swap(reg, blocked);

(1d) until blocked = false;

(2) P_i executes the critical section (CS) after the entry section

(3) P_i executes the following exit section after the CS:

(3a) Reg ← false;

(4) P_i executes the remainder section after the exit section

until false;

Fig : Code for Swap operation

(shared variable)

register: Reg ← false; // shared register initialized

boolean: waiting[1…n];

(local variables)

integer: blocked ← initial value // value to be checked before // entering CS

repeat

(1) P_i executes the following for the entry section:

(1a) waiting[i] ← true;

(1b) blocked ← true;

(1c) repeat waiting[i] and blocked do

(1d) blocked ← Test&Set(Reg);

(1e) waiting[i] ← false;

(2) P_i executes the critical section (CS) after the entry section

(3) P_i executes the following exit section after the CS:

(3a) next ← (i + 1) mod n;

(3b) while next ≠1 and waiting [next] = false do

(3c) next ← (next + 1) mod n;

(3d) if next = i then

(3e) Reg ← false;

(3f) else waiting[j] ← false;

(4) P_i executes the remainder section after the exit section

until false;

Fig : Code for Test & Set operation