Chapter 2/6

- Critical Section Problem / Mutual exclusion
  - progress, bounded wait
- Hardware Solution
  - disable interrupts
    - problems?
- Software Solution
  - busy wait?
    - Tokens
    - Bakery algorithm
    - Special instructions (atomic test-set)
  - Semaphores
  - Monitors
Other Synchronisation Problems

- Dining Philosophers
- Producer Consumer
- Readers Writers
  - reader’s priority, writer’s priority
Readers/Writers with R priority

- **Reader**
  
  ```c
  P(mutex)
  if (nr == 0) {
    nr++; P(notaccessed);
  } else
  nr++;
  V(mutex);
  
  // Read Operations
  P(mutex);
  nr --;
  if (nr == 0) V(notaccessed);
  V(mutex);
  ```

- **Writer**
  
  ```c
  P(exclw);
  P(notaccessed);
  
  // Write Operations
  V(notaccessed);
  P(exclw);
  ```
Serializers

– Monitor Problems
  • If monitor encapsulates resource, then concurrency is reduced even where it is possible
  • If resource is outside, then rouge processes can bypass the monitor.

– Serializers try to avoid this:
  • They are still an ADT with defined operations that encapsulate data, and enforce mutual exclusion.
  • Procedures ma have “hollow” regions where they may allow other processes to access the serializer.
    – join-crowd (crowdid) then body end
    – enqueue (prio,qname) until (condition)
  • all events that gain and release the serializer are totally ordered.
Serializer to solve Readers/Writers

• **Read**
  
  Enque (rq) until empty(wcrowd)
  
  Joincrowd(rc) then
    //Read operation
  
  end

• **Write**
  
  Enque (wq) until ( empty(wc) && empty(rc) && empty(rq) )
  
  Joincrowd (wc) then
    //Write Operation
  
  end
Path Expressions

- Defines possible “valid” execution histories of the operations
  - Sequencing: a;b – a precedes b, no concurrency.
  - Selection: a+b – either a or b is done, but not both and in any order.
  - Concurrency: \{a\} – any number of instances of a can be done at the same time.

- **Path** \{read\} + write **end** gives a weak reader’s priority solution.
CSP

- P2?v
  - Get the value of v from P2 as an input
- P1!10
  - Output value 10 to P1
- The input and output are synchronized if they name each other as source/destination, and the types match
- G-> CL – execute commands in list CL if guard G is true.
- Alternative command – execute one of the choices where is guard is true.
  - G1 -> CL1 o G2 -> CL2 … o … Gn -> CLn
- Repetitive Command *[Alternative] – repeat until all guards are false.
In a distributed system, a site can either be requesting CS execution, executing CS, or none of the above.

Requirements for solutions:
- Deadlock free, starvation free, Fair, Fault tolerant

Metrics of performance (loading conditions):
- # of messages needed for CS
- Synch. Delay – time between one site leaving CS and another entering.
- Response time – Time interval between CS request and end of CS
- Throughput: rate at which system executes CS.
  - \( \frac{1}{\text{synch. delay} + \text{CS execution time}} \)
Solutions

– Centralized approach: Make a single site responsible for permissions.
  • Needs only 3 messages / CS (which 3 ?)
  • Single point of failure, load on central site, 2T synch. Delay

– Lamports algorithm (non token based, FIFO delivery)
  • When Si needs CS, it sends REQ(tsi, i) to all sites in its request set., and places it in its request queue. A site Sj which receives this places it in its own queue, and sends a timestamped REPLY message
  • Si can enter CS when
    – Its request is as the top of the queue
    – It has a reply from all sites it sent a message to with timestamp > timestamp of request
  • Upon exiting CS, removes its request, and sends a release message to all sites. Each receiving site dequeues the request as well
Does it work?

– Can Prove by contradiction
  • Basically this means that a process entered CS even though a request from another process with lower timestamp was in its queue.

– Requires 3(n-1) messages / CS, sd is T

– Improvement – Ricart-Agrawala Algorithm
  • A request is sent just as in Lamport’s algo.
  • On receiving a request, a reply is sent if this site is neither executing its CS nor requesting it. Otherwise, timestamps are compared and a reply sent if the received tstamp is lower than the local tstamp. Otherwise defer.
  • Enter CS when reply received from all.
  • Upon exiting CS, send replies to deferred sites.

– Note that once I have clearance to go into CS, I can do so many times as long as I don’t send back reply.
Maekawa’s Algo.

– Each site’s request set is constructed so that
  • Intersection of request set for any pair of sites is not null
  • Each site is in its own request set
  • The request set size is $K$ for any site.
  • Each site is contained in $K$ sets ($K = \sqrt{N}$)

– To request
  • Site $S_i$ sends $REQ(i)$ to all sites in its request set.
  • On receiving the request, $S_j$ will send $REPLY(j)$ if it hasn’t sent a reply to anyone since it got the last release. Otherwise hold.

– To Execute CS
  • When you get all Replies

– To Release CS
  • Send Release($i$) to all sites in request set.
  • When $S_j$ gets release message, it sends reply to next waiting request.
– Need $3\sqrt{N}$ messages, $2T$ synch. delay.
– Problem – deadlock can occur
  • Imagine a situation with three sites each requesting CS.
– Solution – prioritize request using timestamps and do some extra processing.
  • Basically, eliminate circular wait. Site will send a failure message if it can’t honor your request.
  • If a site is locked, but receives a request from a site with higher priority, it “inquires” from the locking site to see if the lock can be released.
  • Message traffic now $5\sqrt{N}$
Token Based

– Suzuki Kasami Broadcast Algorithm:
    • Basically, need a token to get into CS. Site possessing the token can get into CS repeatedly. RN is an array of integers denoting the largest number in request sequence from a site. The token itself has an array LN containing sequence number of most recently executed request and a queue Q of requesting sites.

– Request
    • If requesting site does not have token, it increments RN[i][i] and sends REQ(i, RN[i][i]) to everyone else. When Sj receives this, it updates RN[j][i]. If it has idle token it sends it to Si

– CS is executed when token is received

– Release
    • Set LN[i] to RN[i][i]. If RN[j][j] = LN[j]+1, then Sj is appended to token Q
    • If token queue is nonempty, delete top entry and send token to that site. This makes it “non-symmetric”

– Messages is 0 or N, Snych. delay is 0 or T.
Raymond’s Tree Based Algo.

- The site with the token is the root of a tree. Each node has a variable called holder pointing to parent. Each node also has a r-q that contains requests for tokens from children.
- Request
  - To request, send request to parent if your r_q is empty and add yourself to the r_q
  - When you get a request, add to r_q and forward to parent if you have not sent a previous request.
  - When root site gets request, it sends token to requesting site and sets holder to point to that site.
  - When site gets a token, it deletes top entry from r_q, sends token and points holder. If r_q is nonempty, it sends request to holder.
- Execute
  - When get the token and your request at top of r_q
- Release
  - If r_q is nonempty, delete top entry, send token, point holder. If r_q still nonempty, send request to holder.
- Message complexity is O(logN), Synch. Delay is (T log N) /2

• Do Section 6.14