Synchronization, Distributed Mutual Exclusion and Leader Election

CMSC 602
Advanced Operating Systems

- Language mechanisms for synchronization
- Contention-based and token-based mutual exclusion
- Leader election algorithms

§3.5, §4.5, §4.6, and Chapter 10

Ju Wang, 2003 Fall
Virginia Commonwealth University
Synchronization Support in Programming Language

To support inter-processes synchronization, programming language must define new structs and functions:

- How to specify concurrent activities
- How to allow interprocess communication
- How to allow nondeterministic execution of processes
Synchronization mechanisms and language facilities

<table>
<thead>
<tr>
<th>methods</th>
<th>language facilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>semaphore</td>
<td>shared variable and system call</td>
</tr>
<tr>
<td>monitor</td>
<td>data type abstraction</td>
</tr>
<tr>
<td>conditional critical region</td>
<td>control structure</td>
</tr>
<tr>
<td>serialized</td>
<td>data type and control structure</td>
</tr>
<tr>
<td>path expression</td>
<td>data type and program structure</td>
</tr>
</tbody>
</table>

Table 1: Shared-Variable Synchronization

<table>
<thead>
<tr>
<th>methods</th>
<th>language facilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>communicating sequential processes</td>
<td>input and output</td>
</tr>
<tr>
<td>remote procedure call</td>
<td>procedure call</td>
</tr>
<tr>
<td>rendezvous</td>
<td>procedure call and communication</td>
</tr>
</tbody>
</table>

Table 2: Message Passing Synchronization
• Reader/Writer problem is chosen to study the capability of different synchronization implementation.

• The problem is general enough to represent the important synchronization requirements

• Several variation of reader/writer problem can be defined:
  – reader preference: an arriving writer waits until there are no running readers
    * strong reader preference: waiting reader is scheduled over the waiting writers.
    * weak read preference
  – writer preference: an arriving reader waits until there are no running and waiting writers
Shared Variable Synchronization: Semaphore

- Semaphore: a shared variable and system call approach
- Semaphore are built-in system data type, typically associated with internal lock and process waiting queue (not visible to programmer)
- Can only be accessed via two operations: P and V.
- Semaphore provide only primitive locking mechanisms, correct synchronization is left entirely to the user processes.
- Lack synchronization transparency (code Figure 3.12 at page 73), not appropriate in distributed systems.
Shared Variable Synchronization: Monitor

- An object based method, used in JAVA.

- A monitor consists of
  - a declaration for its local variables,
  - a set of allowable operations,
  - and an initialization procedure

- Difference between monitor and ordinary objects: only one monitor instance can be active at any time.

- However, the critical section is predefined in the monitor operations.
  - Advantage: more structured and improved readability, since all synchronization code could be find in monitor definition.
  - Disadvantage: not as flexible as in the case of semaphore, where critical section can be defined arbitrary using a P V pair
  - Process can suspend or resume its execution inside the monitor code.
  - Conditional variables: wait/signal

- only allow one active monitor procedure
Monitor Solution to Weak Reader Preference Problem:

```plaintext
/****************** monitor definition***************************/
rw: monitor
var rc: integer; toread, towrite: condition;
busy: boolean;

procedure startread begin
  if busy then toread.wait;
  rc:=rc+1;
  toread.signal;
end

procedure startwrite begin
  if busy or rc != 0 then towrite.wait;
  busy:= true;
  toread.signal or towrite.signal;
end

initialization:
begin
  rc:=0; busy:=false
end
```

---

read processes | write processes
---|---
 rw.startread  | rw.startwrite
 read database  | write database
 rw.endread     | rw.endwrite
Monitors in Java

- A critical section can be a block or a method and are identified with the `synchronized` keyword.

- Java associates a lock with any object. The acquisition and release of a lock is done when a `synchronized` code block is entered and exited.

- Whenever a thread enters a synchronized method, the thread that called the method locks the object whose method has been called. Other threads cannot execute a `synchronized method on the same object` until the object is unlocked.

- When the thread that holds the lock exits the synchronized method, it automatically releases the lock. One of the threads waiting for the lock on the object acquires it, and enters the synchronized method it called.
• To increase parallelism, a block of code (instead of the entire method) may be synchronized.

• Synchronized blocks also allow the programmer to explicitly specify which object’s lock should be acquired by a thread

• Monitor in Java is associated with the object, not the class. Several threads can all be executing the same method in parallel, but the receiving objects have to be different

• Is deadlock possible with monitor?
Conditional Critical Region:

- Structured version of the semaphore approach

- Critical section codes are explicitly name and expressed by **region-begin-end**

- In critical section, a condition can be tested use `when` predicate. If the condition is not met, the process will be suspend, and other process can enter the critical region

```plaintext
var db: shared;
rc: integer;

reader processes
region db begin rc:=rc+1 end
begin
read database
region db begin rc:=rc-1 end

writer processes
region db when rc = 0
begin
write database
end
```
Shared Variable Synchronization: more

Serializer:

- define a abstract data type similar to the one used in monitor
- extended monitor to allow multiple activations inside monitor procedure.
- exclusive region and hollow region:
  - a new control structure: joincrowd-then-begin-end, allow a process to release the serializer and joins a crowd of concurrent processes

Path Expression

- the order of procedure execution must satisfy the path expression
  - path 1: ([read],write) end
- need compiler to translate the path expression into low level synchronization primitives
Message Passing Synchronization

• Asynchronous message passing: nonblocking send, blocking receive
  – assume infinite buffer capacity in the shared communication channels

• Synchronous message passing: blocking send/receive
  – don’t need buffering in the channel

• Communicating sequential processes (CSP)
  – uses input/output rendezvous for synchronization
  – output command \( Q!exp \)
  – input command \( P?var \)
  – rendezvous through explicit naming of each other’s process names
  – implement nondeterministic in CSP: **alternative** commands and **guard** commands
- drawback: direct use of process names is not desired. e.g. for reader/writer problem, all process names must be known when writing the alternate block.
- no elegant solutions for reader/writer problem with CSP.

- Remote procedure call

- Ada Rendezvous: also use procedure call for remote rendezvous and implement nondeterministic language construct for concurrent processes.
  - use accept command at the server to rendezvous with client calling the remote procedure
  - select statement, similar to alternate in CSP provide nondeterministic execution at server.

```plaintext
task rw is
  entry startread; entry endread;
  entry startwrite; entry endwrite;
end
```
task body rw is
  rc=0: integer;
  busy=false: boolean;
begin
  loop select
    when busy==false ->
      accept startread do rc:=rc+1; end;
    or ->
      accept endread do rc=rc-1; end;
    or when rc=0 and busy = false ->
      accept startwrite do busy =true end;
    or ->
      accept endwrite do busy = false end;
  end loop
end;
Concurrent Programming Language

- extended from sequential language
- Occam: evolved from CSP, used in DSP system.
- SR: view concurrent program as a collection of resources.
- Linda: defined a tuple space full of (passive) data tuple and (active) process tuples
  - `in(s)` to match a tuple according to template `s`, blocking.
  - `out(s)`: put a tuple `s` to the tuple space.
Distributed Mutual Exclusion

- Mutual exclusion ensures that concurrent processes make a serialized access to shared resources or data

- Distributed mutual exclusion algorithm achieves mutual exclusion using only peer communication

- Contention-based method: competition

- Controlled methods: logical token
Contention Based Methods: Timestamp Prioritized Schemes

- Use Lamport’s logical clock to totally order the requests for entering critical section (e.g., use Lamport’s total order or two phase total order protocol)

To request the critical section:

1. Broadcasting a (Lamport timestamped REQUEST message
2. Each process maintain a queue of REQUEST messages according their timestamps.
3. On receiving a REQUEST message, a process will send back a REPLY

- A process is allowed to enter critical region only after received all REPLYS, and his request is in the top of the queue.

- When exiting the critical section, broadcast RELEASE message.
• When receiving a RELEASE, the receiver will remove the corresponding REQUEST in its queue.

• Correctness is guaranteed since requests are totally ordered

• Message complexity 3(N-1).

• A simple improvement (by Ricart): the REPLY message is delayed if is current in critical section, or had made a request with smaller timestamp. → no need to send release after completion. 2(N-1) messages per request.
Contention Based Methods: Voting Schemes

- Timestamp based method needs permission from all processors, what if one processor failed?

- When a process receives a REQUEST message, it sends a REPLY only if the process has not voted for any other candidate.

- Once voted, no more REPLY could be sent from this processor, until its vote has been returned.

- A candidate wins the election when it has received a majority of the votes.
Voting Schemes

- Deadlock: what if each candidate collect same amount of votes?
  - require global communication to break ties.

- Or allow change of vote, e.g.,
  - If received a REQUEST with smaller timestamp (compared to the current vote), the processor can retrieve its vote by sending an INQUIRE message to the candidate.
  - If not entered into the critical section yet, the candidate will reply with a RELINQUISH message
  - If already in critical section, will reply a RELEASE after finish its critical section.

- require $O(N)$ message per-entry, the only way to reduce the message complexity is to reduce the the number of votes for a winner—use quorums
Voting Scheme: Quorums

- Each process $P_i$ has an associated committee set $S_i$, and to be able to enter critical section, the process need all votes from its voting district $S_i$.

- To ensure mutual exclusion, we must define $S_i$ such a way that: for any $i$ and $j$ pair, $S_i \cap S_j \neq \text{null}$, referred as quorums.

- For performance consideration, we often want that $S_i$ is not some subset of other $S_j$, and all quorums are of the same size.

- The minimal size of quorums $O(\sqrt{N})$.
  - Let $K$ be the size of voting district
  - Let each process participate $D$ voting district
  - Assume $D = K$, the maximum number of distinguish voting district $M$ is $K(K-1)+1$
  - Let $M$ be the number of process in the group, we have $K = O(\sqrt{N})$
- How to construct a minimum voting district given a number of process?

- Maekawa’s method (§10.1.2):
  - Assume the number of process is of form $n^2$
  - Organized the process in a $nxn$ matrix, and label the process by $(r, s)$ $1 \leq i, j \leq n$
  - Then the voting district for process $(i,j)$ is $\bigcup_{1 \leq i \leq n}(i, s)) \bigcup \bigcup_{1 \leq j \leq n}(r, j))$
Token Based Methods

- There is only one token in the system to assure mutual exclusion
- Process posses the token can enter the critical section
- Explicit control token: reduce the message complexity
- Different methods focus on how token is requested and passed around.
- Three typical topologies: ring, tree, and broadcast
Token Based Methods: Ring Structure Schemes

- Processes are connected in a logical ring structure
- Token circulates in the ring
- Advantages: simple, deadlock-free, and fair.
- Disadvantages: token has to circulate in the ring even if no process wish to enter the critical section.
- The waiting time in the worst case is long.
- The idea is used in Token Bus (802.4) and Token Ring (802.5) LAN.
- Ring initialization and failure handling.
Token Based Methods: Tree Structure

- Processes are organized into a logical tree.
- Token always resides at the root of the tree.
- When a process wishes to acquire the token, a request is sent from the node along the path to the root.
- When successfully acquired, the token migrates to the requesting node, forming a new tree rooted at the new node.
- Each process maintains a FIFO request queue.
- When receiving a request, a processor might:
  - If the FIFO queue is empty and token is somewhere else, the process must request the token request, append the request to the FIFO queue.
  - FIFO queue is not empty, then an uplink request must have been made for some downlink request in the FIFO queue, nothing need to be done.
Token Based Methods: Tree Structure

- If a node has the token and is not using it, the first entry from the FIFO queue is removed and sends the token to that process.
  - It then has to make a request to the new token hold if its local FIFO queue is not empty

Figure 2: Token migration:(1)token originally at node $P_1$, (2) Node $P_4$ made request first, (3) Node $P_3$ made request after it pass token to process $P_4$. 
Token Based Methods: Broadcast Structure

- Philosophy: get ride of topology maintainable, use group communication
- Put enough global information in token for process coordination
- The control token contains: a token vector $T$ and a request queue $Q$
  - Each entry in $T$ represent the accumulated critical section performed by a node
  - The request queue in the token is a list of pending requests in FIFO order
- In each node:
  - Each node the number of request it had made, and attach this number in the REQUEST message.
  - Each node $p$ maintain a sequence vector $S$, containing the highest sequence number of every process that $p$ has heard of.
Token Based Methods: Broadcast Structure

Algorithm:

- $p_i$ broadcast a REQUEST with local sequence number $seq$,
- when receive a REQUEST, $p_j$ will update its $S_j[i] = \max(S_j[i], seq)$,
- if $p_j$ holding an idle token (Q empty), it sends the token to process $i$ if $S_j[i] = T[i] + 1$
- When a requesting process $p_i$ received the token, it can enter critical section.
  - Upon finishing, it will update token: $T[i] = S_i[i]$,
  - it also compare $S$ and $T$, and append all processes satisfying $S_i[k] = T[k]$ and $k \ni Q$ to Q.
  - $p_i$ then remove the top process Q[1] in Q, and send the token to Q[1].
Token Based Methods: Broadcast Structure

Broadcast token is deadlock-free and starvation-free

sequence vector $S_i$

```
Process 1
15  20  11  9

Process 1
14  21  10  8

Process 1
15  21  10  9
```

token vector $T$

```
1  2  3  4
15  20  10  8

1  2  3  4
15  20  11  9

1  2  3  4
15  21  10  9
```

token queue $Q$

```

1  2  3  4
3  4

1  2  3  4
4  2

1  2  3  4
2
```