CS4231: 01-03

01. Mutual Exclusion

properties of a mutex algorithm

- **mutual exclusion** → no more than one process in the critical section
- **no starvation** → if a process wants to enter, it eventually can always enter
  - **progress** → if one or more process wants to enter and if no one is in the critical section, then one of them can eventually enter the critical section
  - no starvation ⇒ progress
- proofs by contradiction
  - mutual exclusion: suppose both processes are in the critical section. WLOG, ...
  - progress: suppose both processes are waiting and consider the value of ...
  - no starvation: suppose process is waiting to enter the CS

peterson's algorithm

- 2 processes only
  - indicate that you wantCS and set turn to the other process
  - spin until the other process does not wantCS OR turn == yours
- if a process wants to reenter after exiting the CS, it will let the other process (if exists) go first

```
RequestCS(0) {
  wantCS[0] = true;
  turn = 1;
  while (wantCS[1] && turn == 1) {};
}
ReleaseCS(0) {
  wantCS[0] = false;
}
```

lamport's bakery algorithm

- for n processes: get a number first, then get served when all lower numbers have been served

hardware solutions

- disable interrupts (no context switching in CS)
- machine-level instructions executed atomically

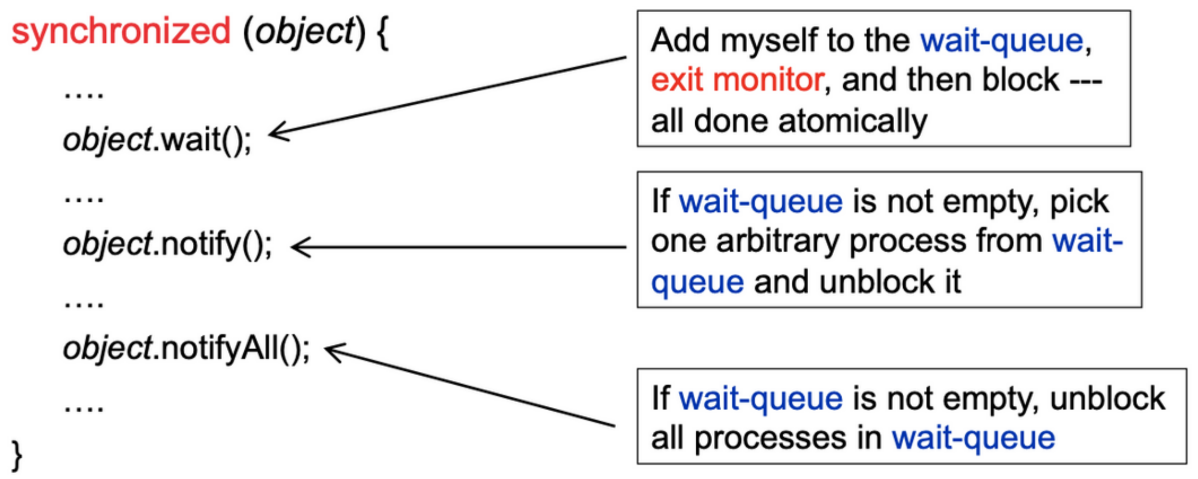
02. Synchronisation Primitives

semaphores

- no busy wait
- wait `P()` : if !value, add self to the queue and block
- signal `V()` - set value=true, wake up one arbitrary process
- avoiding deadlock
  - avoid cycles (if "wait-for" graph has a cycle, then the system has a deadlock)
  - have a total ordering

monitor

- higher-level and easier to use than semaphores
- monitor-queue:
  - enterMonitor()
    - enter monitor if no one is in
    - otherwise add self to the `monitor-queue` and block
  - exitMonitor()
    - pick one arbitrary process from `monitor-queue` and unblock
- wait-queue (within monitor):



- Hoare vs Java style monitor
  - **Hoare**: notify() immediately switches from the caller to a waiting thread
    - waiting condition guaranteed to hold when waiter executes
  - **Java**: notify() places a waiter on the ready thread, but signaler continues inside monitor
    - waiting condition not necessarily true when waiter executes
- caution: nested monitor

03. Consistency Conditions

definitions

- "**consistent**" = satisfies the specification
- **sequential consistency** → equivalent to some legal sequential history that preserves process order
  - **process order** → partial order among events (same order within same process)
  - **legal** → response is same as with only one process
    - sequential $\not\Rightarrow$ legal
  - **sequential** → invocation is *immediately* followed by response (no interleaving)
    - else *concurrent*

      Sequential:
      inv(p, read, X) resp(p, read, X, 0) inv(q, write, X, 1) resp(q, write, X, OK)

      concurrent:
      inv(p, read, X) inv(q, write, X, 1) resp(p, read, X, 0) resp(q, write, X, OK)
    - subhistory is always sequential
- **linearisability** → sequentially consistent and preserves external order
  - **external order** (<): $o1 < o2 \iff$ response of o1 is before invocation of o2
- (local property) H is linearisable $\iff \forall x,\ H|x$ is linearisable
  - × sequential consistency is NOT a local property

proof

- using directed graph: a directed edge is created from o1 to o2 if
  - o1→o2 due to obj
  - o1<o2 in external order
- any topological sorting of the graph gives us a legal sequential history S
  - no cycles in the graph
- any cycle must be composed of
  - edges to some object x (≈ 1 edge since H|x is equivalent S with a total order)
  - edges due to some H (≈ 1 edge since partial order induced by H is transitive)
  - edges due to some object y (or one edge)
  - edges due to some H (or one edge)

CS4231: 03-06

04. Models & Clocks

- visible orderings: happened-before relation (denoted $e \to f$)
  - process order, send-receive order, transitivity
- concurrent-with relation: $e||f \iff \neg(e \to f) \wedge \neg(f \to e)$
- $e \prec f$ denotes e occurred before f in the *same process*
- $s_i \rightsquigarrow r_i$ denotes s_i is the send event corresponding to receive event r_i

logical clocks

- each process has a local counter C
  - local computation: increment C
  - send event: increment C; attach C as logical clock value V
  - receive event: $C = \max(C, V) + 1$
- $s \to t \Rightarrow C_s < C_t$ (but $C_s < C_t \not\Rightarrow s \to t$)

vector clock

- each event has a vector of n integers (vector clock value v)
  - $v1 = v2$ if all *n* fields are the same
  - $v1 \leq v2$ if all corresponding fields ≤
  - $v1 < v2$ if $v1 \leq v2$ and $v1 \neq v2$
    - < is NOT a total order here!
- each process i has local vector C
  - local computation: C[i]++
  - send event: C[i]++, attach C as V
  - receive event: $C = \text{pairwise-max}(C, V)$; C[i]++
- $s \to t \iff C_s < C_t$
- proofs
  - by enumeration of all possible cases (3 happened-before cases) (same process vs diff process)
  - by contradiction

matrix clocks

- each event has n vector clocks
  - principal vector (i^{th} vector): same as vector clock
  - non-principal vector: piggybacked onto messages

05. Global Snapshot

- snapshot of local states on *n* process that could have happened sometime in the past
- global snapshot → if e2 is in the set and e1 is before e2 **in process order**, then e1 is in the set
- consistent GS → GS + any receive event has its corresponding send event in the set
  - can have outgoing (L→R) arrows, but can't have incoming (R→L) arrows
- capturing a CGS
  - special message used for snapshot protocol
  - after taking snapshot, immediately send message to other processes (total $n * (n-1)$ messages)
- on-the-fly messages: sent before sender's local snapshot, received after receiver's local snapshot

03. Consistency Conditions (cont.)

registers: atomic ⇒ regular ⇒ safe

- **atomic** register → ensures linearisability of history
- **regular** register → when a read
  - does not overlap with any write: returns the value written by ***one of the most recent writes***
  - overlaps: returns the value written by **one of the most recent writes** OR one of the ***overlapping writes***
- **safe** register → read
  - no overlap: returns one of the most recent write values
  - overlaps: return anything
- implies nothing about sequential consistency

06. Message Ordering

- causal order → if s1 happened before s2, and r1 and r2 are on the same process, then r1 must be before r2
  - $s_1 \to s_2 \Rightarrow \neg(r_2 \prec r_1)$
- FIFO → any 2 messages from process P_i to P_j are received in the same order as they are sent
  - $s_i \prec s_j \Rightarrow \neg(r_j \prec r_i)$

protocol to maintain causal ordering

- each process maintains a $n \times n$ matrix M
  - $M[i,j]$ = # of messages sent from i to j, as known by local (current) process
  - send event (i to j): M[i, j]++; attach M as T
  - receive event:
    - if $\begin{cases} T[k,j] \leq M[k,j] & \forall k \neq i \\ T[i,j] = M[i,j] + 1 \end{cases}$, then set $M = \text{pairwise-max}(M, T)$
      - M has more knowledge about receive events at j, and exactly one message is pending from i to j
    - else, delay the message
- proofs: consider column j of matrix M and the relative values of each [k, j]

causal ordering in broadcast messages

- total ordering/atomic broadcast → all messages delivered to all processes in exactly the same order
- **coordinator protocol**
  - one process is assigned as the coordinator
  - to broadcast: send message to coordinator; coordinator assigns a seqnum and forwards msg to all processes; messages delivered according to seqnum
  - ☹ coordinator has too much control

skeen's algorithm for total order broadcast

- each process maintains logical clock + buffer for undelivered messages
  1. process broadcasts a message
  2. on receive: put the message in buffer and ACK with current logical clock value
  3. sending process picks the max clock value as message number and notifies (broadcasts) message number
- deliver message if
  - all messages in the buffer have been assigned numbers
  - the message has the smallest number

proofs

- include proof that all messages will be delivered/assigned seqnums/etc