Mutual Exclusion

P. J. Denning

For CS471/CS571

Mutual Exclusion Problem

- Programming requirement: prevent two processes from executing code in a "critical section" at the same time.
  - CS manipulates a share resource
  - Simultaneous use enables interleaving of CS actions, creating a race that can leave the shared resource in an inconsistent state and cause an error for the processes.

- Mutual exclusion means that the processes execute the CS in time intervals that do not overlap.
- The ME requirement is one of the most basic in an operating system; without it the OS cannot share resources safely.

Examples of Unwanted Races

- Processes sharing a ticket counter that they update with a statement like

  mynumber = ticketcount++

- Processes getting shared units from a resource pool, returning them later
- Multiple CPUs accessing the ready list
- Joint account holders accessing their account simultaneously from different ATMs

Locking Protocol

Lock

CRITICAL SECTION

unlock

How to set up the lock so that only one process at a time can pass it?

Don't want races on the lock!

Solution Approaches

- **Software:** specify lock and unlock protocols as software.
  - Busy waiting a likely problem
- **Hardware:** provide machine instruction to replace software lock and unlock
- **Hybrid:** using semaphores (or equivalent) the lock protocol suspends a process that encounters locked lock
  - Avoids busy waiting

Software Solutions

- First: Edsger Dijkstra, 1965
- Second: Donald Knuth, 1966
- Third: Leslie Lamport, 1974
- Major requirements:
  - safe: impossible for two processes to be in CS simultaneously
  - live: impossible for any process to be waiting when no other process is contending or using CS.

Dijkstra's Solution

- N computers operating in a shared memory.
- Common variables:
  - b[1..N], c[1..N] booleans
  - integer k
- Integer k satisfies $1 \le k \le N$
- b[i] and c[i] can be set only by process i

Dijkstra's Protocol

```
    b[i]=0
L1: if k≠i then {
        c[i]=1
        if b[k] then k=i
        goto L1
    }
    else {
        c[i]=0
        for j=1 to N do { if j≠i and not c[j] then goto L1 }
    }
    CRITICAL SECTION
    c[i]=1
    b[i]=1
```

NOTES:

b[i]=1 only when process i in the protocol
c[i]=1 only when process i is trying to get k to point to itself

Opening part can yield 2 or more processes who executed k=i simultaneously.

The if statement just before CS will send all but at most 1 process back to the beginning; only the one with k=i can get back to the if statement.

The process inside the CS need not be the one that won the race k=i

Knuth's Solution

- Knuth observed that it is possible for an unlucky process to wait forever in Dijkstra's protocol.
- He redesigned it to put an upper limit on the wait time of a process to enter the CS.
- Common variables:
  - control[1..N] -- not boolean (values 0, 1, 2)
  - k ($1 \le k \le N$)

```
L0: control[i]=1
L1: for j=k downto 1, N downto 1 do {
      if j=i then break
      if control[j]≠0 then goto L1
    }
    control[i]=2
    for j=N downto 1 do {
       if j≠i and control[j]=2 then goto L0
    }
    k=i
    CRITICAL SECTION
    k = if i=1 then N else i-1
    control[i]=0
```

NOTES:

Search order for other processes (k, k-1, ..., 1, N, N-1, ..., k+1) --
the first computer in this ordering can enter the unused CS

Value k changed to i-1 (mod N) when process i exits CS, giving process i lowest priority for the CS in the next round.

- Knuth showed that the upper limit is $2^{N-1}-1$ passages of other processes through the CS before process i enters CS.

Lamport's Solution

- Avoid a common store so that failure of the memory containing it does not crash the system
- Simulate the ticket counter in a bakery
- Allow processes to read (but not write) each other's memory; they can write their own memories.

```
choosing[i]=1
n[i] = max(n[1],…,n[k])+1
choosing[i]=0
for j=1 to k do {
     while choosing[j] do { }
     while n[j]!=0 and (n[j],j) < (n[i],i) do { }
}
 CRITICAL SECTION
 n[i]=0
```

NOTES:

Process takes number with the 1+max statement.

Two processes may take the same number by computing 1+max together before either stores the result in its local memory.

If two processes have the same number, the one with smaller process index goes first.

Notation (a,b) < (c,d) means: a<c or (a=c and b<d).

The variable choosing[i] tells a process in the while loops that process i will shortly register a number in n[i]. If the while loop were to proceed, it might choose a process with n[j]>n[i], which would subsequently let both i and j in the CS together.

NOTES:

Cannot use ticket counter variable instead of 1+max because of possible time delay between computing 1+ticketcount and storing the new value of tickecount -- if process i delays between the two events, others can raise the ticketcount, which will then be reset to a lower value when process i stores it. This could lead to mulitple processes in the CS.

Hacker Modification

- Attempt to modify Lamport's solution by eliminating busy waiting.
- Function YIELD puts process in wait state if its wakeup waiting switch is off, and leaves switch off when it returns.
- Function NOTIFY(j) awakens j or sets its wakeup waiting switch to on.

```
n[i] = max(n[1],...,n[k])+1
for j=1 to k do
  { if n[j]!=0 and (n[j],j) < (n[i],i) then YIELD }
CRITICAL SECTION
wj=0
m=n[i]+1
for j=k downto 1 do
  { if j=i then no-op
    else if n[j]=n[i] then {wj=j; m=n[i]}
    else if n[j]=m    then {wj=j}
   }
NOTIFY(wj)
n[i]=0
```

NOTES:

Apparently simple modification fails.

Not safe: processes 1 and 2 come simultaneously when system is empty; both compute ticket number 1; process 1 pauses; process 2 enters the CS because it sees n[1]=0. Then process 1 awakens and also enters the CS because it sees n[1]=n[2] and its own index 1 is smaller than the other's 2.

Not live: processes 1 and 2 come simultaneously when system is empty; both compute ticket number 1; process 2 pauses; process 1 enters the CS and then pauses just after NOTIFY(0); now process 2 wakes up and sees (1,1)=(n[1],1)<(n[2],2)=(1,2) and yields; finally process 2 awakens, sets n[1]=0, and leaves. Now process 2 is waiting on YIELD and no other process is present to awaken it.

Hardware Solution

- Test-and-set-lock instruction
  - TSL x returns value of x and sets x=0 in one memory cycle.
  - Since memory allows only one CPU to access a location in the same memory cycle, TSL x is mutually excluded from all other executions of TSL x.
- The protocol (x=1 means unlocked):

  ```
  while (TSL x) do { }
  CRITICAL SECTION
  x=1
  ```

- Uses busy waiting

Hybrid Solution

- Uses semaphore to avoid busy waiting
- The protocol (Initial count of mutex = 1):

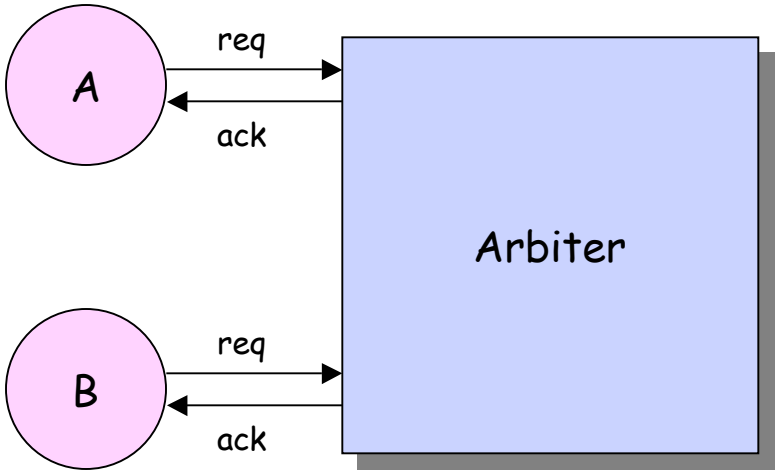
  wait(mutex)

  CRITICAL SECTION

  signal(mutex)

The Arbitration Problem

- What if we took away the assumption that the memory mutually excludes accesses to the same location in the same memory cycle?
- Can we still implement mutual exclusion?

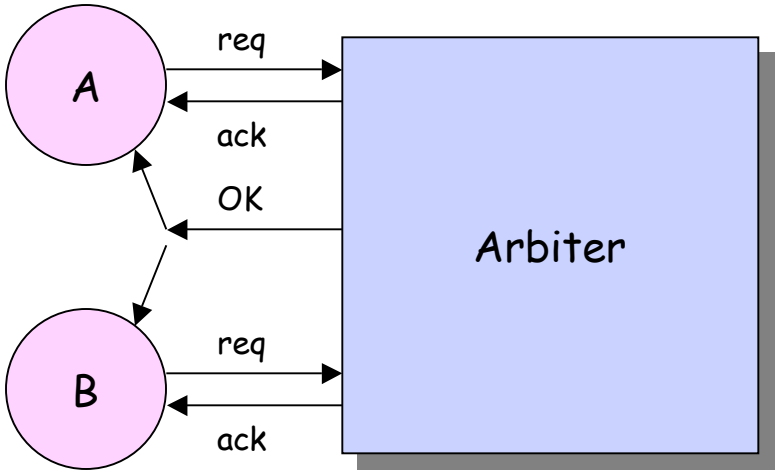


A (and B):
- noncritical section
- req
- wait for ack
- CRITICAL SECTION
- req
- wait for ack
- repeat

Arbiter:
- when one or more req occurs, choose one
- send it ack
- wait for its req
- repeat

- The statement “when one or more event occurs, choose one” is a problem.
- If two events occur within the switching time of the circuit, the arbiter will enter an ill-defined meta-unstable state.
- While it is in the meta-unstable state, its outputs are meaningless and may not even be measurable signals.
- After an indeterminate time t, the arbiter will exit the meta-unstable state and re-enter a stable state. $P[t>T] = 1-e^{-at}$, where $1/a$ is the circuit’s mean switching time .

- With the set up sketched earlier, the Arbiter can misbehave if both A and B send requests at (nearly) the same time -- for example,
  - neither is given the go-ahead
  - both are given the go-ahead
- If first happens, system can hang up; if second, processes proceed in error, leading to later crash.
- Need to modify the setup so that A and B stop and wait until the Arbiter has entered a stable state.



A (and B):
- noncritical section
- req
- wait for OK
- wait for ack
- CRITICAL SECTION
- req
- wait for OK
- wait for ack
- repeat

Arbiter:
- when events occur,
  - set OK=0, seek to move to stable state in which one is chosen
- when state stable, set OK=1.
- send chosen input ack
- repeat

- To have safe arbiter, must permit indefinite waiting time while arbiter makes a decision. (Mean = 1/a, but occasionally decision times can be much longer.)
- If you insist on sampling arbiter output after T seconds, there is a probability of failure ($1=e^{-aT}$). With events occurring at computer speed, mean time to failure may be only a few hours or days.