Multi-threaded Programs

Shankar

March 1, 2021

Outline

Multi-threaded programs

- Multiple threads executing concurrently in the same address space
- Threads interact by reading and writing shared memory
- Need to ensure that threads do not "interfere" with each other
- For example, given a linked list X
  - while a thread is adding an item to X, another thread should not read or write X.
  - if thread u blocks when it finds X empty, another thread should not insert an item in between u finding X empty and blocking
- Formalizing "non-interference":

  a code chunk S in a program is atomic if while a thread u is executing S, no other thread can change an intermediate state of u's execution of S.

Hypothetical synchronization construct: *await*

- *await B*: *S*, where *S* is a code chunk (*no blocking or infinite loop*) and *B* is a boolean condition (*no side effects*):
  - execute *S* only if *B* holds, all in one atomic step
  - if *B* does not hold, wait
- *atomic S*: short for *await True*: *S*
- Programming languages provide more limited constructs:
  - locks, condition variables, semaphores, ...
- Use awaits to develop program, then implement using locks, etc
  - easily doable if the code outside awaits is interference-free
- Canonical synchronization problems
  - mutual-exclusion, readers-writers, producer-consumer, ...

Outline

Locks

- Lock operations: acquire and release
- lck ← Lock()   // define a lock
- lck.acq()   // acquire the lock; blocking
  - call only if caller does not hold lck
  - returns only when no other thread holds lck
- lck.rel()   // release the lock; non-blocking
  - call only if caller holds lck
- lck.rel() does not give priority to threads blocked in lck.acq()

Condition variables

- Condition variable operations: wait, signal and signal_all
- A condition variable is associated with a lock
- cv ← Condition(lck)   // condition variable associated with lck
- cv.wait()   // wait on cv; blocking
  - call only if caller holds lck
  - atomically release lck and wait on cv
    when awakened: acquire lck and return
- cv.signal()   // signal cv; non-blocking
  - call only if caller holds lck
  - wake up a thread (if any) waiting on cv
- cv.signal_all()   // wake up all threads waiting on cv
- lck.acq() does not give priority to threads blocked in cv.wait()

Outline

Semaphores

- Semaphore: variable with a non-negative integer count
- Semaphore operations: P() and V()
- sem ← Semaphore(N) // define semaphore with count N (≥ 0)
- sem.P() // blocking
  - wait until sem.count > 0 then decrease sem.count by 1; return
  - checking sem.count > 0 and decrementing are one atomic step
- sem.V() // non-blocking
  - atomically increase sem.count by 1; return
- V() does not give priority to threads blocked in P()

Outline

Progress assumption

- For a multi-threaded program to achieve anything, we have to assume that its threads execute with non-zero speed (but otherwise arbitrarily varying)
- Making this precise is simple for non-blocking statements but not for blocking statements (eg, acquire, wait, P, await)
- A thread at an non-blocking statement T eventually gets past T
  - Achieved if every unblocked thread periodically gets cpu cycles
- A thread at a blocking statement T eventually gets past T if T is *continuously* unblocked or *repeatedly* (but not continuously) unblocked
  - Achieved in most implementations only in a probabilistic sense, not in a deterministic sense

Outline

Bounded counter

Program P0:

- x, y: global int variables; initially 0
- up(), down() // callable by multiple threads simultaneously
- up() increments x only if x < 100, and returns 2*x
- down() decrements x only if x > 0, and returns 2*x

- up():

```
int z
await (x < 100):
    x ← x+1
    z ← x
return 2*z
```

- down():

```
int z
await (x > 0):
    x ← x-1
    z ← x
return 2*z
```

Program P1:

- x, y // as in P0
- lck ← Lock()
- cvNF ← Condition(lck) // for guard (x < 100)
- cvNE ← Condition(lck) // for guard (x > 0)

- up():
```
int z
lck.acq()
while (not x < 100):
    cvNF.wait()
x ← x+1
z ← x
cvNE.signal()
lck.rel()
return 2*z
```

- down():
```
int z
lck.acq()
while (not x > 0):
    cvNE.wait()
x ← x-1
z ← x
cvNF.signal()
lck.rel()
return 2*z
```

Program P2:

- x, y // as in P0
- lck ← Lock()
- cv ← Condition(lck) // for both guards

- up():
```
int z
lck.acq()
while (not x < 100):
    cv.wait()
x ← x+1
z ← x
cv.signal_all()
lck.rel()
return 2*z
```

- down():
```
int z
lck.acq()
while (not x > 0):
    cv.wait()
x ← x-1
z ← x
cv.signal_all()
lck.rel()
return 2*z
```

Program P3:

- x, y // as in P1
- mutex ← Semaphore(1) // for lck
- gateNF ← Semaphore(0) // for cvNF
- gateNE ← Semaphore(0) // for cvNE

- up():
```
  int z
  mutex.P()
  while (not x < 100)
     mutex.V()
     gateNF.P()
     mutex.P()
  x ← x+1
  z ← x
  gateNE.V()
  mutex.V()
  return ← 2*z
```

- down():
```
  int z
  mutex.P()
  while (not x > 0)
     mutex.V()
     gateNE.P()
     mutex.P()
  x ← x-1
  z ← x
  gateNF.V()
  mutex.V()
  return ← 2*z
```

Outline

Bounded Buffer Problem

- Given BB // has no synchronization
  - buf: buffer of capacity N items
  - num: number of items in buf
  - add(x): add item x to buf; non-blocking
  - rmv(): return an item from buf; non-blocking
- Obtain enQ(x) and deQ() such that
  - callable by multiple threads simultaneously // safety
  - enQ(x) calls add(x) once, waiting if buf is full // " "
  - deQ() calls rmv() once, waiting if buf is empty // " "
  - at most one add() or rmv() ongoing at any time // " "
  - if buf not full and at least one enQ() ongoing, eventually an enQ() returns // progress
  - if buf not empty and at least one deQ() ongoing, eventually a deQ() returns // " "

Solution using Awaits

Program BB0:

- buf, num, add(x), rmv() // as in BB

- enQ(x):
```
  await (num < N):
     add(x)
  return
```

- deQ():
```
  await (num > 0):
     tmp ← rmv()
  return tmp
```

Program BB1

- buf, num, add(x), rmv() // as in BB0
- lck: lock
- cvNF, cvNE: cond vars // not-full, not-empty

- enQ(x):

```
enQ(x):
  lck.acq()
  while (num = N):
      cvNF.wait()
  add(x)
  cvNE.signal()
  if num < N:
     cvNF.signal()
  lck.rel()
  return
```

- deQ():

```
deQ():
  lck.acq()
  while (num = 0):
      cvNE.wait()
  tmp ← rmv()
  cvNF.signal()
  if num > 0:
     cvNE.signal()
  lck.rel()
  return tmp
```

- Is red code needed?

Program BB2:

- buf, num, add(x), rmv() // as in BB0
- Semaphore(1) mutex
- Semaphore(0) gateNF, gateNE
- nwNF, nwNE: initially 0

- enQ(x):

```
mutex.P()
while num = N:
  nwNF ++
  mutex.V(); gateNF.P()
  nwNF --
add(x)
if num > 0 and nwNE > 0:
  gateNE.V()
else mutex.V()
return
```

- deQ():

```
mutex.P()
while num = 0:
  nwNE ++
  mutex.V(); gateNE.P()
  nwNE --
tmp ← rmv()
if x < 100 and nwNF > 0:
  gateNF.V()
else mutex.V()
return tmp
```

Non-standard solution using sempahores

Program BB3:

- buf, num, add(x), rmv() // as in BB
- Semaphore(1) mutex
- Semaphore(N) nSpace
- Semaphore(0) nItem

- enQ(x):

```
    nSpace.P()
    mutex.P()
    add(x)
    mutex.V()
    nItem.V()
    return
```

- deQ():

```
    nItem.P()
    mutex.P()
    tmp ← rmv()
    mutex.V()
    nSpace.V()
    return tmp
```

- Cute. But not adaptable.

Bounded Buffer with variable-size items

- Like the bounded-buffer except
  - buf has a capacity of N bytes
  - num: indicates available bytes in buf
  - add(x,k): add item x of size k bytes
  - rmv(k): return an item of size k bytes

- Previous await-structured solution BB0 is easily adapted
  - enQ(x,k):
    ```
    await (num ≤ N - k)
        add(x,k)
    ```
  - deQ(k):
    ```
    await (num ≥ k)
        tmp ← rmv(k)
    return tmp
    ```

- Can transform above to using standard synch constructs
- Exercise: can you adapt program BB3 to solve this

Outline

Readers-Writers Problem

- Given non-blocking functions `read()`, `write()`
- Obtain functions `cread()`, `cwrite()` such that
  1. each is callable by multiple threads simultaneously
  2. cread() calls read() once, waits if ongoing write()
  3. cwrite calls write() once, waits if ongong write() or read()
  4. allow multiple ongoing read() calls
  5. if every read() and write() call returns then
     a. every cread() call eventually returns
     b. every cwrite() call eventually returns
- 1–4 are safety requirements
- 5 is a progress requirement

Consequence of safety requirements

- Every evolution of a solution is an alternating sequence of idle intervals and busy intervals
- An idle interval has no read or write
- A busy interval is either a *read interval* or a *write interval*
- A write interval has exactly one write
- A read interval has one or more reads
  - it starts with the first `read()` call
  - it ends when the last `read()` return

RW1: partial solution using awaits

Program RW1:

- nr ← 0 // number of ongoing reads
- nw ← 0 // number of ongoing writes

- cread():

```
r1: await (nw = 0)
        nr ++
    read()
r2: await (true)
        nr --
```

- cwrite():

```
w1: await (nw = nr = 0)
        nw ++
    write()
w2: await (true)
        nw --
```

- RW1 satisfies 5a but not 5b
  (thread stuck at w1 due to endless stream of reads)

RW2: Lock-cv version of RW1

Program RW2:

- nr, nw: initially 0 // as in RW1
- lck, cvR, cvW // lock, cv-read, cv-write

- cread():

```
  lck.acq()
  while not nw = 0:
    cvR.wait()
  nr ++
  lck.rel()
  read()
  lck.acq()
  nr --
  if nr = 0:
    cvW.signal()
  cvR.signal()
  lck.rel()
```

- cwrite():

```
  lck.acq()
  while not nw = nr = 0:
    cvW.wait()
  nw ++
  lck.rel()
  write()
  lck.acq()
  nw --
  cvW.signal()
  cvR.signal()
  lck.rel()
```

RW2a: simplified RW2

- While `write()` ongoing, no other `read()` or `write()` ongoing
- Hence can remove `lck.rel` and `lck.acq` surrounding `write()`
- Then nw is always 0, so can simplify code

Program RW2a:

- nr, lck, cvW // as in RW2; no need for nw, cvR

- cread():

```
lck.acq()
nr ++
lck.rel()
read()
lck.acq()
nr --
if (nr = 0)
   cvW.signal()
lck.rel()
```

- cwrite():

```
lck.acq()
while (not nr = 0)
   cvW.wait()
write()
cvW.signal()
lck.rel()
```

Semaphore versions of RW1

- Several ways to transform program RW1 to a semaphore program
  - apply "lock-cv $\rightarrow$ semaphore" transformation on RW2
  - apply "lock-cv $\rightarrow$ semaphore" transformation on RW2a
- Left as exercises

RW3: another partial semaphore solution – 1

- Following is the partial solution usually given in texts
- Variables
  - `Semaphore(1) wrt`: protects every busy interval
    - `wrt.P()` is done at the start of the interval
    - `wrt.V()` is done at the end of the interval
  - `int nr`: number of ongoing reads
    - for detecting the start and end of a read interval
  - `Semaphore(1) mutex`: protects nr
- Note
  - In a read interval of more than one read,
    `wrt.P()` and `wrt.V()` are done in different cread calls
  - If read threads are blocked (due to ongoing write),
    one is waiting on `wrt` and the others on `mutex`

RW3: partial solution using semaphores – 2

- cread():

```
mutex.P()
nr ++
if (nr = 1)
   wrt.P()
mutex.V()
read()
mutex.P()
nr --
if (nr = 0)
   wrt.V()
mutex.V()
```

- cwrite():

```
wrt.P()
write()
wrt.V()
```

- Cute. But not easily modified to satisfy requirement 5b.

RW4: solution using awaits – 1

- One way to satisfy requirement 5b is to impose a limit, say N, on the number of consecutive reads while a writer is waiting.
- It's simpler and adequate to impose the limit on every reading interval, whether or not a writer is waiting. That is what we do here.
- Variables
  - nr ← 0: # ongoing reads
  - nw ← 0: # ongoing writes
  - nx ← 0: # of reads in this read interval
    - incremented when a read starts
    - zeroed when read interval ends

RW4: solution using awaits – 2

- cread():

```
await (nw = 0 and nx < N):
    nr ++
    nx ++
read()
await (true)
    nr --
    if nr = 0:
      nx ← 0
```

- cwrite():

```
await (nw = nr = 0)
    nw ++
write()
await (true)
    nw --
```

- Exercise: transform to lock-cv and semaphore programs

Outline

Read-write locks

- A read-write lock can be held as a "read-lock" or as a "write-lock"
- Can view it as consisting of one write-lock and many read-locks
- At any time, [# wlocks, # rlocks] held is [0, 0], [0, >0], or [1, 0]
- Operations
  - `rwlck` ← ReadWriteLock()   // define a read-write lock
  - `rwlck.acqR()`   // acquire read-lck; blocking
  - `rwlck.relR()`   // release read-lock; non-blocking
  - `rwlck.acqW()`   // acquire write-lck; blocking
  - `rwlck.relW()`   // release write-lock; non-blocking
- Call `acqR()` or `acqW()` only if caller does not have lock
- Call `relR()` or `relW()` only if caller has the appropriate lock

Implementing read-write locks

- Any solution to the readers-writers problem yields a read-write lock

- Program readers-writers
  - variables // rwlock vars
  - cread():
    ```
    entry code  //  acqR()
    read()
    exit code   //  relR()
    ```
  - cwrite():
    ```
    entry code  //  acqW()
    write()
    exit code   //  relW()
    ```