On Multi-Threaded Programming

Shankar

February 18, 2018

Multi-threaded programs

- Multiple threads executing concurrently in the same address space
- Threads interact by reading and writing shared memory
  - eg: threads u and v read/write a structure (memory area) x
- Requires synchronization of threads
  - u should wait to access x while v is writing x
  - u should wait to "add" to x while x is "full"
- Canonical synchronization problems
  - mutual-exclusion, readers-writers, producer-consumer, ...
- Standard synchronization constructs
  - locks, conditions, semaphores, ...
- Goal: solve synchro problems using standard synchro constructs

Outline

Locks, condition variables, semaphores

Await-structured program

Achieving priority for waiting threads

Bounded Buffer

Readers-Writers

Read-write Locks

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
- Weak lock: lck.acq() returns if lock is continuously free
- Strong lock: lck.acq() returns if lock is repeatedly free
  // even if only intermittently free

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 coming from cv.wait()

Semaphores

- Semaphore: variable with a non-negative integer count
- Semaphore operations: P() and V()
- sem $\leftarrow$ 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 waiting threads
- Semaphore can be strong or weak (just like a lock)

Outline

Locks, condition variables, semaphores

Await-structured program

Achieving priority for waiting threads

Bounded Buffer

Readers-Writers

Read-write Locks

Awaits

- Standard synchro constructs (ie, lock, cv, sem) are low level
- High-level construct: await (B) $\{S\}$ // await B: S
  - if B holds execute S, all in one atomic step
  - if B does not hold, wait
  - B has no side effect
- Weak await: does S if B holds continuously
- Strong await: does S if B holds repeatedly // even if intermittent
- atomic $\{S\}$ // short for await (true) $\{S\}$
- A program using awaits is
  - easier to understand than one using std synchro constructs
  - can be transformed to one using std synchro constructs
  - often provides a convenient intermediate program

Await-structured program

- We say code chunks S and T in a program conflict if
  - a thread can write to a memory area
  - another thread can simultaneously read/write the same area
- This is a dynamic (not textual) notion
  - S and T can update the same location but be conflict-free if two threads cannot execute them simultaneously
- S and T can be the same code chunk
  - S conflicts with itself if it writes to a global location x and two threads can execute S simultaneously
- Await-structured program:
  - awaits are the only synchronization constructs
  - all the code outside the awaits is conflict-free

Example await-structured program

Program P0:

- x, y: global int variables; initially 0
- up(), down() // callable by multiple threads simultaneously

- 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

Locks, condition variables, semaphores

Await-structured program

Achieving priority for waiting threads

Bounded Buffer

Readers-Writers

Read-write Locks

Method: Await → Sem priority for waiting

- Await-structured program with distinct await guards $B_1, \cdots, B_N$
- Want an equivalent semaphore program such that processes stuck in an await have higher priority than processes arriving freshly to the await
- Solution:
  - Semaphores *mutex* and $gate_1, \cdots, gate_N$ // as before
  - After executing the update of an await
    - do *mutex*.V() if no B_i holds and has waiting processes
    - o/w select one such B_i and do $gate_i.V()$ (do not *mutex*.V())

Method: Await → Sem priority for waiting

- Await-structured program with distinct await guards $B_1, \cdots, B_N$
- *mutex* ← Semaphore(1)
- For every B_i
  - $gate_i$ ← Semaphore(0) // to wait for B_i
  - nw_i ← 0 // number of processes waiting at $gate_i$
- Replace each await (B_i) S_i by

```
mutex.P()
if (not B_i)
    nw_i++; mutex.V(); gate_i.P(); nw_i--
S_i
for k in 1, ···, N
    if (B_k and nw_k > 0)
        gate_k.V()
        return
mutex.V()
```

Applying method to program P0

Program P4:

- x, y, mutex, gateNF, gateNE // as in P2
- nwNF , nwNE: initially 0 // # waiting on gateNF, gateNE

- up():

```
int z
mutex.P()
if (not x < 100)
   nwNF ++
  mutex.V(); gateNF.P()
   nwNF --
x ← x+1
z ← x
if x>0 and nwNE>0
   gateNE.V()
else
   mutex.V()
return 2*z
```

- down():

```
int z
mutex.P()
if (not x > 0)
   nwNE ++
  mutex.V(); gateNE.P()
   nwNE --
x ← x-1
z ← x
if x<100 and nwNF>0
   gateNF.V()
else
   mutex.V()
return 2*z
```

Outline

Locks, condition variables, semaphores

Await-structured program

Achieving priority for waiting threads

Bounded Buffer

Readers-Writers

Read-write Locks

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
```

- awaits with weak progress adequate to achieve desired progress (do not require progress for every waiting enQ or deQ)

Program BB1

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

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

- 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 --
```

- Weak awaits: RW1 does not satisfy requirement 5 (eg, thread stuck at r1 due to endless stream of reads/writes)
- Strong awaits: 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
  - apply "await $\rightarrow$ semaphore with awakened priority" on RW1
- 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.

- Variables
  - nR ← 0: # ongoing reads
  - nW ← 0: # ongoing writes
  - ncR ← 0: # of reads since last write
    - incremented when a read starts
    - zeroed when a write starts
  - nwW ← 0: number of waiting writes
    - incremented when a thread enters cwrite
    - decremented when the thread starts to write

RW4: solution using awaits – 2

- cread():

```
  await (nW = 0 and
          (ncR < N or nwW = 0)
        )
      nR ++
      ncR ++
  read()
  await (true)
      nR --
```

- cwrite():

```
  await (true)
      nwW ++
  await (nW = nR = 0)
      nW ++
      nwW --
      ncR ← 0
  write()
  await (true)
      nW --
```

- Exercise: transform to lock-cv and semaphore programs

Outline

Read-write lock

- 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

- Weak lock: `acqX()` returns if lock is continuously free
- Strong lock: `acqX()` returns if lock is repeatedly free

  // even if only intermittently free

Implementing read-write locks

- Any readers-writers solution yields a read-write lock
- Weak or strong depending on readers-writers solution

Program readers-writers

- variables
- cread():

```
entry code   // acqR()
read()
exit code    // relR()
```

- cwrite():

```
entry code   // acqW()
write()
exit code    // relW()
```