CS 450: Operating Systems Lecture 9: Concurrency Problems

Spring 2014, J. Sasaki
Dept of Computer Science
Illinois Institute of Technology

Reader-Writer Problem

The Reader-Writer Problem

- The Reader-Writer problem studies a resource with different categories of use that have different exclusion needs.
- Database shared by reader and writer threads.
  - Multiple threads can read concurrently.
  - Writer threads can't write concurrently.
  - If a writer is writing, no reader can read.
- Pedestrian crossing problem (pedestrians vs cars)

Reader-Writer Solution

- `int read_count = 0;` // nbr readers
- `semaphore RC_mutex = new Semaphore(1);`
  // mutex for `read_count`
- `semaphore DB_mutex = new Semaphore(1);`
  // mutex for database access

Writer Process

- Writers are straightforward:

```
do {
   DB_mutex.wait();
   … perform write …
   DB_mutex.signal();
} while(…);
```

Reader Process

- First reader has to wait for database.
  - Other readers wait for first reader to get DB (by waiting to update `read_count`)
- Each finishing reader decreases `read_count`
- Last finishing reader releases DB.

```
// Reader (embedded in do-while loop)
  RC_mutex.wait();
  ++read_count;
  if (read_count == 1) {
     DB_mutex.wait();
  }
  RC_mutex.signal();
```

... read DB ...

```
  RC_mutex.wait();
  --read_count;
  if (read_count == 0) {
     DB_mutex.signal();
  }
  RC_mutex.signal();
```

Problem Variations

- First variation: A reader is kept waiting only if a writer has the database.
    - If the readers have the DB, then other readers can appear and get the DB.
    - Writers can starve.
- Second variation: A writer is kept waiting only if another writer has the database.
    - Readers may starve

Problem Variations

- Resource with multiple levels of user
  - Users of different levels are mutually excluded.
  - Are services mutually exclusive within the same level?
- Variation 1: Reader users have higher priority than writer users.
- Variation 2: Writer users have higher priority than reader users.

Fairness?

- Which thread does semaphore signal unblock?
  — Unpredictable
- Can't guarantee bounded waiting.
  - Queueing the blocked threads would help.
- Can we attach a queue to a semaphore?

Semaphores in Python

Python Semaphores

```
import threading
from threading import Semaphore
from threading import BoundedSemaphore;

# Plain semaphore can become larger than initial value
# Note: acquire() = P(), release() = V()
#
>>> s1 = Semaphore(1) # max value
>>> s1.release()      # ok

# Bounded semaphore can’t become larger than
# initial value
#
>>> s2 = BoundedSemaphore(1)
>>> s2.release() # causes runtime error
```

Mutex Program

- Small python program runs two threads in parallel.
- Thread *n*'s critical section prints `(`*n n n*`)`.
- To simulate unpredictable speed of execution, we sleep random amounts of time before printing `(`*n*, *n*, and *n*`)` and outside the critical section.
- If a `(1 1 1)` and `(2 2 2)` are interleaved, then the threads were both in their critical sections, concurrently.
- If `safe` is `True`, we use a mutex semaphore and treat the printing as a critical section (and avoid interleaving).

mutex.py

```
from threading import Thread, BoundedSemaphore
import random, time

def main(safe = True, trials = 5):
   global mutex
   mutex = BoundedSemaphore(1)
   p_args = {'safe': safe, 'trials': trials}
   p1 = Thread(target = thread, args=(1,), kwargs=p_args)
   p2 = Thread(target = thread, args=(2,), kwargs=p_args)
   p1.start()
   p2.start()
   p1.join()
   p2.join()
   print()
```

```
def thread(pnbr, safe = False, trials = 5):
   global mutex
   for i in range(trials):
      if safe:
         mutex.acquire()
      trace(pnbr, '({}')
      trace(pnbr, '{}')
      trace(pnbr, '{})')
      if safe:
         mutex.release()
      time.sleep(random.randint(0,2))

def trace(pnbr, tag):
   print((tag + ' ').format(pnbr), \
      end = '', flush = True )
   time.sleep(random.randint(0,2))
```

```
> python3 -i mutex.py
>>> main()
(1 1 1) (2 2 2) (1 1 1) (1 1 1) (1 1 1)
(1 1 1) (2 2 2) (2 2 2) (2 2 2) (2 2 2)
>>> main(safe = False)
(1 (2 2 2) (2 1 1) (1 2 2) 1 1) (1 (2 1
2 1) 2) (1 (2 2 2) 1 (2 1) 2 2) (1 1 1)
>>>
```

Reader-Writer Program

- Writer thread `1` will get DB mutex, print `(w1 w1 w1)` respectively (peppered with random sleeps), then release DB mutex. (Writer 2 similar.)
- Reader thread `1` prints `(r1 RC` and `r1 RC)` around each RC mutex get & release; it prints `[r1 DB` and `r1 DB]` around each DB mutex get & release. It prints `r1` between the two RC mutex gets & releases, to indicate it's reading the DB. (Reader 2 is similar.)

```
from threading import Thread, BoundedSemaphore
import random, time

def main(safe = True, trials = 5):
   global RC_mutex, DB_mutex, read_count
   read_count = 0;   # nbr readers
   RC_mutex = BoundedSemaphore(1);
   DB_mutex = BoundedSemaphore(1);
   p_args = {'safe': safe, 'trials': trials}
   r1 = Thread(target = reader, args=(1,), kwargs=p_args)
   r2 = Thread(target = reader, args=(2,), kwargs=p_args)
   w1 = Thread(target = writer, args=(1,), kwargs=p_args)
   w2 = Thread(target = writer, args=(2,), kwargs=p_args)
   [r1.start(), r2.start(), w1.start(), w2.start()]
   [r1.join(), r2.join(), w1.join(), w2.join()]
   print()
```

```
def writer(id, safe = True, trials = 5):
   global DB_mutex
   for _ in range(trials):
      if safe:
         DB_mutex.acquire()
      trace_write(id, '(w{}')
      trace_write(id, 'w{}')
      trace_write(id, 'w{})')
      if safe:
         DB_mutex.release()
      time.sleep(random.randint(0,2))

def trace_write(id, tag):
   print((tag + ' ').format(id), flush=True, end='')
   time.sleep(random.randint(0,2))
```

```
def reader(id, safe = True, trials = 5):
   global RC_mutex, DB_mutex, read_count
   for _ in range(trials):
      if safe:
         RC_mutex.acquire()
         trace_read(id, '(r{} RC', end='')
      read_count = read_count + 1
      if safe:
         if read_count == 1:
               DB_mutex.acquire()
               trace_read(id, '[r{} DB')
         RC_mutex.release()
         trace_read(id, 'r{} RC)')

      trace_read(id, 'r{}')
```

… continued on next slide …

```
     if safe:
        RC_mutex.acquire()
        trace_read(id, '(r{} RC', end='')
     read_count = read_count - 1
     if safe:
        if read_count == 0:
              DB_mutex.release()
              trace_read(id, 'r{} DB]')
        RC_mutex.release()
        trace_read(id, 'r{} RC)')
     time.sleep(random.randint(0,2))

def trace_read(id, tag, end='\n'):
   print((tag + (' ' if end == '' else '')).\
      format(id), flush=True, end=end )
   time.sleep(random.randint(0,2))
```

```
> python3 -i readwrite.py
>>> main(trials=2)
(r1 RC [r1 DB       r1 incr’s read count to 1 & gets DB for the readers
r1 RC)              r1 releases read count
(r2 RC r2 RC)       r2 gets read count, increments it to 2
r1                  r1 reads
r2                  r1 reads
(r2 RC r2 RC)       r2 decrements read count to 1
(r1 RC r1 DB]       r1 decrements read count to 0 & releases DB
(w1 w1 r1 RC)       w1 gets DB & writes before r1 finishes decrement
w1) (r2 RC (w1 w1 w1) (w2 w2 w2) [r2 DB
                           w1 finishes writing, gives up DB, but before
                           r2 can get DB, w1 jumps in again & prints, then
                           w2 jumps in and prints, then r2 gets the DB
r2 RC)              r2 releases read count it held as w1 and w2 printed
(r1 RC r1 RC)       r1 increments read count to 2
r2                  r2 reads
r1                  r1 reads
(r2 RC r2 RC)       r2 decrements read count to 1
(r1 RC r1 DB]       r1 decrements read count to 0 & releases DB
(w2 w2 w2) r1 RC)      w2 gets DB & writes before r1 finishes decrement
```

- With unsafe execution
  - `w2` starts writing before `w1` finishes
  - `r2` reads before `w1` finishes its 2nd write
  - `w2` starts writing before `w1` finishes 2nd write

```
>>> main(safe = False, trials = 2)
r1
r1
r2
(w1 w1 (w2 w1) w2 w2) (w1 w1 r2
(w2 w1) w2 w2)
```

FIFO Queue Semaphore

FIFO Queue of Blocked Threads

- Given: Thread-unsafe queue mechanism.
- Want to attach a thread-safe queue of blocked threads to a semaphore.
  - If blocking is necessary, then wait() adds you to the end of the queue and blocks
  - If the queue is nonempty, signal() will unblock the head of the queue.

FIFO Semaphore Approach

- Use lots of semaphores
  - One mutex semaphore to protect queue.
  - One semaphore that we conceptually wait/ signal
  - To block on the FIFO semaphore, a thread will create a thread-local semaphore, enqueue it, and block on it.
  - signal() dequeues/unblocks semaphore for the next thread to awaken.

Pseudocode

```
FIFO_Sem class:
  int value;  // value of semaphore
  Semaphore mutex;  // for queue
  Queue queue;


new FIFO_Sem(initial_value):
  self.value = initial_value
  self.mutex = new Semaphore(1);
  self.queue = new Queue();
```

Pseudocode

```
wait():
  Semaphore barrier = new Semaphore(0)
  boolean block = false;

  self.mutex.wait();

  if (--self.value < 0) {
     self.queue.enqueue(barrier);
     block = true;
  }

  self.mutex.signal();

  if (block)
      barrier.wait();
```

Pseudocode

```
signal():
  self.mutex.wait();
  self.value++;
  if (!self.queue.is_empty()) {
     Semaphore barrier
        = self.queue.dequeue();
    barrier.signal();
  }
  self.mutex.signal();
```