Operating System Concepts

Lecture 16: Concurrency Problems

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Today's class

- Race condition
  - examples
- Critical section
- Example: too much milk!
  - three attempts to solve the problem

Recap

- Cooperating processes (or threads) share data
  - sharing eliminates the need to copy data (enhancing performance)
  - processes share data via IPC; threads share global variables and dynamically allocated variables
- Concurrent access to shared data can lead to data inconsistency
  - programmers are responsible for synchronizing access to shared data
  - maintaining consistency requires mechanisms to ensure orderly execution of cooperating processes

Sharing data among threads can increase performance but lead to problems…

- Recall that in the concurrent (web) server example, each request is handled by a different thread
- Suppose there's a **global variable** that keeps track of the number of hits the index page of `cs.ualberta.ca` gets in a day
  - each thread will increment this shared data if the request is to access the index page: `hits = hits + 1`
  - but incrementing `hits` is not a single operation; it involves reading `hits` from memory into a register (loading), incrementing the register, and storing the register back to memory:

```
register1 <- hits
register1 <- register1 + 1
hits <- register1
```

  - thread execution can be interleaved because of time-slicing

Race condition

- When the outcome depends on the order in which threads accessing shared data run
  - the order of thread execution is not deterministic; the last thread that changes a value wins
- Consider the following example
  - suppose `n = ARRAY_SIZE -1` before the two threads start and it is a global variable

Thread A runs:

```
if (n == ARRAY_SIZE)   // A1
    return -1;         // A2
array[n] = valueA;     // A3
n = n + 1;             // A4
```

Thread B runs:

```
if (n == ARRAY_SIZE)   // B1
    return -1;         // B2
array[n] = valueB;     // B3
n = n + 1;             // B4
```

- Below are three possible scheduling orders:
  - A1,A3,A4,B1,B2 —> thread B does not get to write valueB (thread A wins)
  - A1,B1,A3,B3,A4,B4 —> thread B overwrites the value written by thread A (thread B wins)
  - A1,B1,A3,A4,B3,B4 —> thread B attempts to write the value at `array[ARRAY_SIZE]` (causing **overflow**)

Why's it called a "race" condition?

Thread A runs:

```
int x = 0;
int y = 0;
...

funcA() {
    x = y + 1;      // A1
}
```

Thread B runs:

```
int x = 0;
int y = 0;
...

funcB(){
    y = 2;          // B1
    y = y * 2;      // B2
}
```

- Below are three possible scheduling orders:
  - A1, B1, B2 —> `x = 1`
  - B1, B2, A1 —> `x = 5`
  - B1, A1, B2 —> `x = 3`
  - we say that thread A **races against** thread B
- Bugs are intermittent and difficult to catch: a small change in this code (e.g., adding a print statement) can hide the bug
  - the correct result is produced only for a certain interleaving

Race condition in the producer-consumer problem

Process A runs:

```
while(1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++; // num items in the buffer
}
```

Process B runs:

```
while(1) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
       counter--; // num items in the buffer
}
```

- Let's assume that the two processes use **shared memory** for data exchange
  - `counter` is a variable stored in a shared memory object
  - one item is produced and one time is consumed so we expected to have `counter = 5`
- Consider this execution interleaving with `counter = 5` initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6}
S5: consumer execute `counter = register2` {counter = 4}

Why do race conditions exist?

- The order of thread execution is non-deterministic because of
  - multiprocessing: a system may contain multiple processors: cooperating threads/processes can execute simultaneously
  - multiprogramming: thread/process execution can be interleaved because of time-slicing
- Operations are not typically atomic
  - for example x = x + 1 is not a single operation
- **Goal:** ensure that a concurrent program produces the correct output for all possible interleavings
  - an operation must run to completion or not run at all (atomicity), and an instruction sequence must be guaranteed to execute indivisibly
  - one thread executes an instruction sequence at a time

Critical section

- Critical section is a block of code (a number of consecutive instructions) that cannot be executed in parallel by multiple threads (**to avoid a race condition**)
  - one process/thread running a critical section excludes the other ones (i.e., **mutual exclusion**). For example, two threads adding a node to a linked list can corrupt it
  - it is typically the code that accesses and/or modifies the values of shared data (files, data structures, etc.)
- There might be multiple critical sections in a program
- Synchronization primitives, such as **locks**, are required to ensure that only one thread/process runs in the critical section at a time

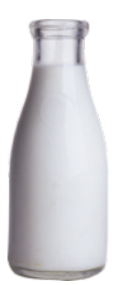
```
do {
        entry section      <- lock before entering
                              other threads will have to wait if locked
            critical section
        exit section       <- unlock when leaving
            remainder section
} while (true);
```

Requirements

- A critical section implementation must be:
  - correct: the system would behave as if only one thread was in the critical section at any given time
    - safety (mutual exclusion), liveness (progress), and bounded waiting — will be discussed in the next lecture
  - efficient: getting in and out of a critical section must be fast; critical sections should be as short as possible
  - flexible: must have as few restrictions as practically possible
  - supporting high concurrency: allows maximum concurrency while preserving correctness

A real life example

	Roommate A	Roommate B
3:00	Arrive home: no milk in fridge	
3:05	Leave for store	
3:10	Arrive at store	Arrive home: no milk in fridge
3:15	Leave store	Leave for store
3:20	Return home, put milk away	Arrive at store
3:25		Leave store
3:30		Return home: too much milk!



milk is the shared data

Requirements (correctness properties):

1. someone should buy milk if there is no milk in the fridge to avoid starvation (liveness)
2. only one person buys milk, otherwise it spoils (safety)

First attempt: just leave a note!

```
while(1){
   if(milk == 0) {
      if(note == 0) {
         note = 1;
         buy_milk();
         note = 0;
      }
   }
}
```

each roommate should check if there is a note before buying milk **(waiting)**

each roommate should leave a note when going out to buy milk **(locking)**

add some value to `milk`

each roommate should remove the note after buying milk **(unlocking)**

Assumption: load and store operations are atomic
no hardware support is required

Failed attempt — threads can get context-switched at any time

Thread A | **Thread B**

time

```
if(milk == 0) {
  if(note == 0) {
                                  if(milk == 0) {
                                    if(note == 0) {
                                      note = 1;
                                      buy_milk();
                                      note = 0;
                                    }
                                  }
    note = 1;
    buy_milk();
    note = 0;
  }
}
```

still too much milk

Can we fix this (change the order of checking the note and checking on milk)?

Thread A

```
if(note == 0) {
  if(milk == 0) {
    buy_milk();
  }
  note = 1;
}
```

Thread B

```
if(note == 1) {
  if(milk == 0) {
    buy_milk();
  }
  note = 0;
}
```

but what if the second thread doesn't run at all?

Second attempt

Thread A

```
note[0] = 1;
if(note[1] == 0) {
  if(milk == 0) {
    buy_milk();
  }
}
note[0] = 0;
```

Thread B

```
note[1] = 1;
if(note[0] == 0) {
  if(milk == 0) {
    buy_milk();
  }
}
note[1] = 0;
```

each roommate will leave a labelled note before looking in fridge

`boolean note[2];`

Failed attempt

Thread A

```
note[0] = 1;

if(note[1] == 0) {
   ...
}

note[0] = 0;
```

Thread B

```
note[1] = 1;

if(note[0] == 0) {
   ...
}
note[1] = 0;
```

this time we got no milk (starvation)

Third attempt

Thread A

```
note[0] = 1;
if(note[1] == 0) {
  if(milk == 0) {
    buy_milk();
  }
}
note[0] = 0;
```

Thread B

```
note[1] = 1;
while(note[0] == 1) {
  ; // spin
}
if(milk == 0) {
  buy_milk();
}
note[1] = 0;
```

Third attempt - scenario 1

Thread A

```
note[0] = 1;

if(note[1] == 0) {
  if(milk == 0) {
    buy_milk();
  }
}
note[0] = 0;
```

Thread B

```
note[1] = 1;

while(note[0] == 1) {
  ; // spin
}

if(milk == 0)
  buy_milk();
note[1] = 0;
```

only Thread B will execute `buy_milk( )`

Third attempt - scenario 2

Thread A

```
note[0] = 1;
if(note[1] == 0) {




  if(milk == 0) {
    buy_milk();
  }
}
note[0] = 0;
```

only Thread A will execute
`buy_milk( )`

Thread B

```
note[1] = 1;
while(note[0] == 1) {
  ; // spin
}





if(milk == 0)
  buy_milk();
note[1] = 0;
```

Third attempt - scenario 3

Thread A

```
note[0] = 1;
if(note[1] == 0) {
  if(milk == 0) {
    buy_milk();
  }
}
note[0] = 0;
```

only Thread B will execute
`buy_milk( )`

Thread B

```
note[1] = 1;
while(note[0] == 1) {
  ; // spin
}
if(milk == 0)
  buy_milk();
note[1] = 0;
```

Correctness of the third attempt

Thread A

```
   note[0] = 1;
X: if(note[1] == 0) {
     if(milk == 0) {
       buy_milk();
     }
   }
   note[0] = 0;
```

Thread B

```
   note[1] = 1;
Y: while(note[0] == 1) {
     ; // spin
   }
   if(milk == 0) {
     buy_milk();
   }
   note[1] = 0;
```

- at point X either there is a note left by Thread B or not
  - if there is a note, then B is either checking the amount of milk and buying more if needed, or is waiting for A to remove the note. So in both cases, A must remove its note ASAP
  - if not, B has either bought milk or hasn't started yet. In both cases, A can safely check if milk is needed and buy

Correctness of the third attempt

Thread A

```
   note[0] = 1;
X: if(note[1] == 0) {
     if(milk == 0) {
       buy_milk();
     }
   }
   note[0] = 0;
```

Thread B

```
   note[1] = 1;
Y: while(note[0] == 1) {
     ; // spin
   }
   if(milk == 0) {
     buy_milk();
   }
   note[1] = 0;
```

- at point Y either there is a note left by Thread A or not
  - if there is a note, then A must be checking B's note or buying milk as needed. So B has to wait until there is no longer a note left by A; Once this happens, B either finds milk that A bought or buys it if needed
  - if not, it is safe for B to buy milk if needed since A has not yet started or has quit

Is this a good solution though?

- Relies on load and store operations being atomic ✔
- It’s too complicated — it was hard to convince ourselves that this solution actually works
- It’s asymmetrical — operations executed by Threads A and B are different
    - adding more threads would require different code/logic for each new thread and modifications to existing threads
- It requires **busy waiting** — Thread B is consuming resources (CPU) despite the fact that it is not doing any useful work

Homework

- Write a program that finds the minimum value in a list of integers using K threads
    - assume that the list is partitioned equally among the threads and the minimum value found so far is stored in a global variable
- Does it have a race condition?