CS 450: Operating Systems Lecture 8: Mutual Exclusion & Synchronization

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

Critical Sections

2

Critical Sections

- Say two threads have sections of code S_1 (in one thread) and S_2 (in the other)…

  … such that we cannot allow both S_1 and S_2 to execute concurrently.

  (All of S_1 must finish before starting S_2 and vice versa.)

- Then S_1 and S_2 are "***critical sections***" of their threads. Example: Our `x++` and `x--`.

Mutual Exclusion

- The ***mutual exclusion*** ("***mutex***") problem is the problem of avoiding concurrent execution of critical sections.
- We can generalize to > two threads.
- We can generalize to > 1 piece of code in each thread: Any identified piece of code in one thread excludes any identified piece of code in the other thread.
- We can also have > 1 mutex problem.

Wait Your Turn

`turn = …` // Either 0 or 1

// `turn` ∈ {`0`,`1`} = the thread allowed to proceed

```
/* Thread 0 */
while(turn!=0);

x++;

turn=1;
```

```
/* Thread 1 */
while(turn!=1);

x--;

turn=0;
```

Repeatedly Execute C.S.?

`turn = … //` Either 0 or 1

`// turn` ∈ {`0,1`} = the thread allowed to proceed

What if we repeatedly execute C.S.?

```
/* Thread 0 */
do {
     …
    CS0
    …
 } while (…)
```

```
/* Thread 1 */
do {
     …
    CS1
    …
 } while (…)
```

Wait Your Turn Only If We Both Want to Go

- Use an array `want[0..1]`: `want[i]` true iff thread `i` wants to access its C.S.
- If both threads want their C.S's, then `turn` ∈ {`0`,`1`} = the thread allowed to go
- We can go into our C.S. if it's our turn or if (it's not our turn but) the other thread doesn't want its C.S.
- We must wait if `want[`*other*`]` = `true` and `turn` ≠ *us*.

Peterson's Solution

Let *us* = our thread nbr (0 or 1)

Let *other* = the other thread nbr (1 or 0)

```
do {
    …
    want[us] = true;
    turn = other;
    while (want[other] && turn != us) ;
    … Our Critical Section …
    want[us] = false;
    …
} while (…);
```

Observations

- Once `turn` = *us*, it stays that way until we set `turn` = *other*.
- `want[`*us*`]` is true between our lines 3…7.
    - Only we set `want[`*us*`]`: The other thread never changes our `want[…]` flag.

Mutual Exclusion ?

- Claim: During our line 6,

  `want[`*us*`]` ∧ (`want[`*other*`]` ⇒ `turn=`*us*)

- It holds instantaneously after our line 5.
- If `want[`*other*`]` holds then the other thread is in its lines 3…7.
- The other thread set `turn=`*us* at its line 4 and `turn` can't change while we're at line 6.

10

Mutual Exclusion !

- If we're at our C.S. (line 6), then
  $\texttt{want[}us\texttt{]} \wedge (\texttt{want[}other\texttt{]} \Rightarrow \texttt{turn=}us)$
- If the other thread is at its C.S. then
  $\texttt{want[}other\texttt{]} \wedge (\texttt{want[}us\texttt{]} \Rightarrow \texttt{turn=}other)$
- For us both to be in our C.S.'s, we need
  - $\texttt{want[0]}$, $\texttt{want[1]}$, $\texttt{want[0]} \Rightarrow \texttt{turn=1}$, and $\texttt{want[1]} \Rightarrow \texttt{turn=0}$
  - These can't all be true simultaneously.

Progress & Bounded Waiting

- Peterson's solution guarantees ***progress***: If no thread is in its C.S. and a thread wants to enter its C.S., then it can, eventually.
- Also guarantees ***bounded waiting***: If a thread is blocked trying to enter its C.S., it cannot wait forever as the other thread enters its C.S. over and over.

Recall Original Wait Loop

```
x = 10;
ok_to_go = true;
```

```
/* Thread 0 */              /* Thread 1 */
while (!ok_to_go) ;         while (!ok_to_go) ;
ok_to_go = false;           ok_to_go = false;

x++;                        x--;

ok_to_go = true;            ok_to_go = true;
```


13

Test-and-Set

- The problem was with

```
while (!ok_to_go) ;
ok_to_go ← false
```

- Problem was caused by interleaving between the loop and flag assignment
- IBM 360 Test-and-set instruction
  - `TS reg, x // reg ← x and x ← 1`
- Later architectures: Compare and swap

14

Test-and-Set

- Let's paraphrase
  - `TestSet(flag)` yields the value of `flag`; it also sets `flag` ← `true`
  - Atomic operation; can't be interrupted between copying old value of `flag` and setting `flag` to `true`.

Use Test-and-Set

- (Parent initializes `busy` ← `false;`)

```
while (TestSet(busy)) ;
```
… *Critical Section* …
```
busy ← false;
```

- Doesn't guarantee bounded waiting

16

Use Test-and-Set

```
x = 10;
busy = false
```

```
/* Thread 0 */
while(testSet(busy)) ;
x++;
busy = false;
```

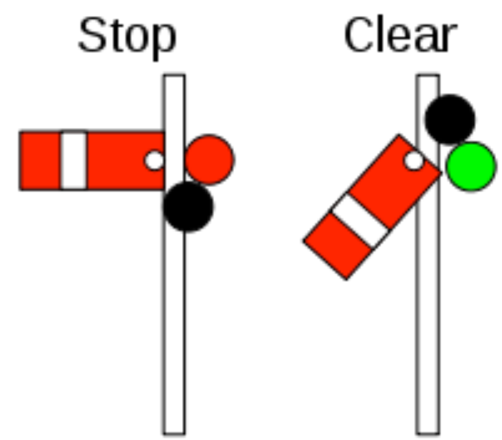
```
/* Thread 1 */
while(testSet(busy)) ;
x--;
busy = false;
```

Semaphores

18

Higher-Level Synchronization Primitive

- Spin looping $$; yield to OS instead?
- Semaphore primitive (Edsger W. Dijkstra)
- Railroad semaphore flags: (thanks, Wikipedia).



- When you see the flag, continue iff it's clear (raise flag behind you, lower it when you leave the protected area).

Binary Semaphore

- A binary semaphore has two states 0 & 1.
  - If you want to enter the C.S., wait if the semaphore is 0.
    - If it's 1, decrease it to 0, do your C.S., and then increase it to 1.
    - Increasing the semaphore causes the waiting thread to be awoken; it can enter its C.S.

20

Counting Semaphore

- Counting semaphore `s` basically an integer plus a queue. Once initialized, we can
- `s.wait()`: atomically,

  `if (--s < 0)`

  enter queue for `s` *and block.*
- `s.signal()`: atomically,

  `++s; if (`*queue not empty*`)` *remove some process from queue and awaken it*

Wait, Signal, P, V

- The original names for `wait()` and `signal()` are `P()` and `V()`.
  - P = prolaag = short for “probeer te verlagen” is Dutch for “try to reduce”.
  - V = verhogen is Dutch for “increase”
- Exist other names (acquire/release, down/up, suspend/post, …).

Value of Semaphore

- We don't get to look at value of semaphore (wouldn't necessarily help anyway).
- If `s` < `0`, then |`s`| = nbr. processes blocked.
- If `s` ≥ `0`, then `s` = nbr. waits that can be done before someone blocks.
  - `s` (if ≥ `0`) is nbr. of resources that can be obtained via `wait()`.

Mutex via Semaphores

- We can solve the mutex problem using a binary semaphore:

```
Semaphore s = 1;
```

```
/* Thread 0 */              /* Thread 1 */
...                         ...
s.wait();                   s.wait();
... C.S. ...                ... C.S. ...
s.signal();                 s.signal();
```

Producer-Consumer Problem

The Producer-Consumer Problem

- Archetypical problem in concurrency.
  - Two processes and a buffer.
  - Producer process repeatedly adds item to buffer; consumer process repeatedly removes item from buffer.
  - Consumer must wait if buffer is empty; producer must wait if buffer is full

Consumer Process

```
do {
    …
    Wait until buffer not empty;
    Get item from buffer;
    Use item;
    …
while (…);
```

Use a semaphore to wait until buffer not empty.

Consumer

- Parent:

```
Semaphore not_empty = 0;
Buffer buf;     // initially empty
```

- Consumer: (Waits until buffer nonempty)

```
...
not_empty.wait();
item = buf.get_item();
item.use();
...
```


28

Is buffer Thread-Safe?

- Can buffer routines be interleaved?
- If we try to concurrently/simultaneously execute `buf.get_item()` and `buf.add_item(item)`, can the buffer get broken?

If buffer Not Thread-Safe

- If the buffer is not thread-safe, we need a separate mutex semaphore for the buffer.
- Parent:

```
 Semaphore not_empty = 0;
 Buffer buf;    // initially empty
semaphore buf_mutex = 0;
```

Consumer's buffer mutex

- Consumer:

```
...
not_empty.wait();

buf_mutex.wait();
item = buf.get_item();
buf_mutex.signal();

item.use();
...
```

What About Producer?

- Producer is symmetric; need a `not_full` semaphore initially true
- Parent:

```
Semaphore not_empty = 0;
Semaphore not_full  = 1;
Buffer buf;    // initially empty
semaphore buf_mutex = 0;
```


32

Producer

- Producer: (Waits until buffer not full)

```
  …
item = …

not_full.wait();

buf_mutex.wait();
buf.add_item();
buf_mutex.signal();

  …
```


33

Producer and Consumer Unblock Each Other

- Once producer adds an item, it can do `non_empty.signal();` to waken consumer if necessary.
- Once consumer removes an item, it can do `non_full.signal();` to waken producer if necessary.

Full Consumer Code

- Consumer:

```
…
not_empty.wait();

buf_mutex.wait();
item = buf.get_item();
buf_mutex.signal();

not_full.signal();

item.use();
…
```


35

Full Producer Code

- Producer:

```
  …
item = …

not_full.wait();

buf_mutex.wait();
buf.add_item(item);
buf_mutex.signal();

not_empty.signal();
  …
```

Observations

- We can have multiple producers and consumers sharing the same buffer.
- Why are Producer and Consumer so similar?
  - Think of the producer as a consumer of buffer holes.

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 = 1;`
  // mutex for `read_count`
- `semaphore DB_mutex = 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();
```

Updated Feb 12

… read DB …

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