Operating Systems

Autumn 2024

race conditions, mutual exclusion

What is a thread?

- An abstraction for a single running process
- A multi-threaded program has more than one point of execution
  - Multiple program counters, one for each thread
  - They share code, heap, and global variables
  - Each thread has its own stack
  - Each thread has its own private set of registers

Data race

- A program has a data race if it is possible for a thread to modify an addressable location at the same time that another thread is accessing the same location
- The result/correctness depends on the sequence/timing of events; i.e., how things end up being scheduled.

Shared counter

- Two threads updating a single shared variable `cnt`

```
int cnt = 0;
void *worker(void *ptr) {
    int i;
    for (i = 0; i < ITERATIONS_PER_THREAD; i++)
        cnt++;
}
```

- What happens when two threads execute concurrently?

Order of execution of threads

- Possible implementation of `cnt++`:

```
eax := cnt       // read
eax := eax + 1   // modify
cnt := eax       // write
```

- OS might decide to context switch from one thread to another at any time
- Thus the atomic actions of concurrent threads may be interleaved in any possible order

Possible interleaving

- eax_i denotes the value of register in thread i
-
```
eax_1 := cnt
eax_1 := eax_1 + 1
<switch>
eax_2 := cnt
eax_2 := eax_2 + 1
cnt := eax_2
<switch>
cnt := eax_1
```
- Result: cnt one less than correct! (lost update)

Q

- Consider 2 threads sharing a global variable `count`, initially 10:

```
// Thread A
count++;

//Thread B
count--;
```

- What are the possible values for count after both threads finish executing?

Q

```
int count = 0;
// Thread 1                          // Thread 2
for (i=0; i < 10; i++) {             for (i=0; i < 10; i++) {
    count++;                             count++;
}                                    }
```

- What is the final value of count?
  - A value between 2 and 20

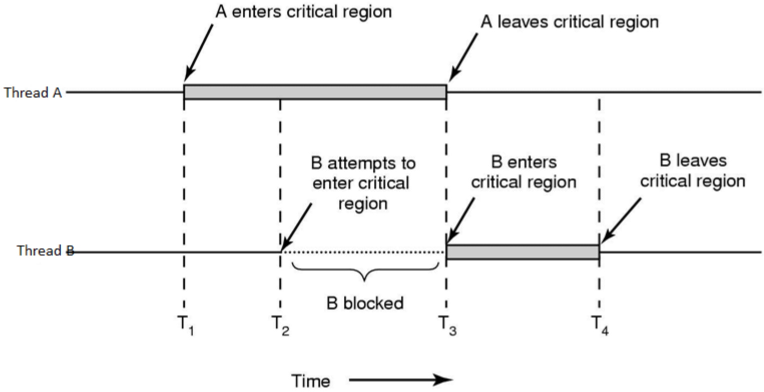
count = 2

- t1 and t2 read count
- t1 increments 9 times. so count becomes 9
- t2 increments once. so count becomes 1
- t1 reads count (value is 1)
- t2 increments 9 times. so count becomes 10
- t1 increments once. so count becomes 2

Critical section

- Critical section: code that access shared resource (e.g., variable or data structure)
- Race condition: arises when multiple threads simultaneously enter critical section leading to a non-deterministic outcome

Critical section



What basic mechanism can stop thread B from entering critical region when thread A in?

Requirements for a good solution

- Mutual exclusion (safety property)
    - At most one thread can be in CS at a time

Requirements for a good solution

- Mutual exclusion (safety property)
  - At most one thread can be in CS at a time
- Progress (liveness property)
  - If no thread is currently in CS and threads are trying to access, one should eventually be able to enter the CS

Requirements for a good solution

- Mutual exclusion (safety property)
  - At most one thread can be in CS at a time
- Progress (liveness property)
  - If no thread is currently in CS and threads are trying to access, one should eventually be able to enter the CS
- Bounded waiting (liveness property)
  - Once a thread T starts trying to enter the CS, there is a bound on the number of times other threads get in

Progress vs. Bounded waiting

- Liveness requirements are needed for a solution to be useful

Progress vs. Bounded waiting

- Liveness requirements are needed for a solution to be useful
- Progress
  - If no thread can enter CS, we don't have progress

Progress vs. Bounded waiting

- Liveness requirements are needed for a solution to be useful
- Progress
  - If no thread can enter CS, we don't have progress
- Bounded waiting
  - If thread A is waiting to enter CS while B repeatedly leaves and re-enters CS ad infinitum, we don't have bounded waiting

Mutual exclusion: legacy solutions

- Mutual exclusion algorithm solely based on read and write operations to a shared memory

Mutual exclusion: legacy solutions

- Mutual exclusion algorithm solely based on read and write operations to a shared memory
- First correct solution for two threads by Dekker in 1966
- Peterson proposed a simpler solution in 1981

Mutual exclusion: legacy solutions

Solution for 2 threads T_0 and T_1

Algorithm 1 Peterson's algorithm for thread T_i

Global Variables:

```
bool wants[2] = {false, false};
int not_turn; /* can be 0 or 1 */

enter_CS()
    wants[i] = true;
    not_turn = i;
    while wants[1-i] == true and not_turn == i do
        /* do nothing */
    end while

leave_CS()
    wants[i] = false;
```

Peterson's algorithm: a few comments

- wants: To declare that the thread wants to enter
- not_turn: To arbitrate if the 2 threads want to enter
- Line 6: "The other thread wants to access and not our turn, so loop"

Correctness (1)

Algorithm 1 Peterson's algorithm for thread T_i

```
Global Variables:
bool wants[2] = {false, false};
int not_turn; /* can be 0 or 1 */

enter_CS()
    wants[i] = true;
    not_turn = i;
    while wants[1-i] == true and not_turn == i do
        /* do nothing */
    end while

leave_CS()
    wants[i] = false;
```

- Mutual exclusion
  - Assume both threads want to enter CS, then wants[0] = wants[1] = 1
  - not_turn can only be 0 or 1
  - the condition of the while loop can only resolve to false for one of the threads

Correctness (2)

Algorithm 1 Peterson's algorithm for thread T_i

```
Global Variables:
bool wants[2] = {false, false};
int not_turn; /* can be 0 or 1 */

enter_CS()
    wants[i] = true;
    not_turn = i;
    while wants[1-i] == true and not_turn == i do
        /* do nothing */
    end while

leave_CS()
    wants[i] = false;
```

- Progress
    - If T_0 doesn't want CS, wants[0] == false, so T_1 won't loop
    - If both threads try to enter, one thread will succeed

Correctness (3)

Algorithm 1 Peterson's algorithm for thread T_i

Global Variables:

```
bool wants[2] = {false, false};
int not_turn; /* can be 0 or 1 */

enter_CS()
    wants[i] = true;
    not_turn = i;
    while wants[1-i] == true and not_turn == i do
        /* do nothing */
    end while

leave_CS()
    wants[i] = false;
```

- Bounded waiting
    - If T_1 was blocked from CS and T_0 tries to re-enter, T_0 will set not_turn $= 0$, allowing T_1 in

Q1

- Consider a modified algorithm where lines 4 and 5 are swapped
- Does this solution work?

```
Global Variables:
1: bool wants[2] = {false, false};
2: int not_turn; /* can be 0 or 1 */

3: enter CS()
4:     not_turn = i;
5:     wants[i] = true;
6:     while wants[1-i] == true and not_turn == i do
7:         /* do nothing */
8:     end while

9:  leave CS()
10:    wants[i] = false;
```

Answer

- Mutual exclusion violation
- Bad trace
  - T1 does not_turn = 1
  - T0 does not_turn = 0
  - T0 does wants[0] = true
  - T0 does while (wants[1] == true and not_turn == 0); // (false and true) $\rightarrow$ false

Answer

- Mutual exclusion violation
- Bad trace
  - T1 does not_turn = 1
  - T0 does not_turn = 0
  - T0 does wants[0] = true
  - T0 does while (wants[1] == true and not_turn == 0); // (false and true) → false
  - T0 enters the CS

Answer

- Mutual exclusion violation
- Bad trace
  - T1 does not_turn = 1
  - T0 does not_turn = 0
  - T0 does wants[0] = true
  - T0 does while (wants[1] == true and not_turn == 0); // (false and true) $\rightarrow$ false
  - T0 enters the CS
  - T1 does wants[1] = true
  - T1 does while (wants[0] == true and not_turn == 1); // (true and false) $\rightarrow$ false

Answer

- Mutual exclusion violation
- Bad trace
  - T1 does not_turn = 1
  - T0 does not_turn = 0
  - T0 does wants[0] = true
  - T0 does while (wants[1] == true and not_turn == 0); // (false and true) → false
  - T0 enters the CS
  - T1 does wants[1] = true
  - T1 does while (wants[0] == true and not_turn == 1); // (true and false) → false
  - T1 enters the CS

Q2 - take home

- Assume the `while` statement in line 6 is changed to
  - while (wants[1-i] == true and not_turn != i)
- Show a trace in which there is a mutual exclusion violation
- Show a trace in which there is NO mutual exclusion violation

Peterson's algorithm: a few more comments

- Given solution works for 2 threads
- Can be generalized to *n* threads but *n* must be known in advance
- To implement a general lock, processors provide hardware primitives