# Kessels’ Algorithm in C

Demonstrated by Ýmir Þórleifsson & Bjartur Þórhallsson

## Unintentional error

The assignment of the second flag is incorrect. The way that the code is written now allows for both processes to enter the critical section. However, there is a simple fix for this error.

We simply make the following change to the code:

|  |
| --- |
| FROM flag2[id] = 1 – flag2[1 – id];  TO flag2[id] = (id+flag2[1 – id]) % 2; |

This makes sure that the while test will become true whenever there is not a process in the critical section.

## Observation of unoptimized compilation

When executing the program without any optimizations we noticed that on occasion both threads were able to access the critical section, it looks like that under some circumstances, they are not blocking each other from accessing the critical section, while they are there.

## Observation of optimized compilation

Running the program using optimization is not a good idea. Since the compiler doesn’t expect to be running a multi-threaded program, it stores the flag variables as a temp variable and uses temp to perform the test in the while loop. A tread stopped in the while loop will stay there indefinitely and never enter the critical section, since the thread never reads the flag that the other tread changes. The compiler does this because it will save time to store the data in a register instead of reading from memory every time the check is being performed. This is not good for multi-threaded programs, since they share certain variables, and they must access it from memory in order to update and read the shared data.

## Comparisions of the assembly

We noticed that there are significantly less lines of assembly in the optimized file. But this is because it assumes that it can skip parts of the code that are crucial for it to work. This is because it stores a temp value in beginning of execution, and that temp never changes. Even though it will and should not be stored as a temp and looked up every time instead.

## Kessels’ Algorithm without the unintentional error

We need to force our compiler to read the shared memory even though it might not like it. We however know better, so we use a module that allows us to do just that. The module that we use is stdatomic.h, we can use it to create atomic variables that the compiler needs to read from memory.

|  |
| --- |
| #*include* <pthread.h>  #*include* <stdio.h>  #*include* <stdbool.h>  #*include* <assert.h>  #*include* <stdatomic.h> // *Use stdatomic.h to force reading vars from memory*  // *Mark shared variables as atomic*  \_Atomic(*bool*) flag1[2] = {*false*, *false*};  \_Atomic(unsigned) flag2[2] = {0, 0};  int critical = 0;  void \**process*(void \*arg)  {  *const* int id = \*((int \*) arg);  int n = 0;  start:  /\* *Remainder of code* \*/  *fprintf*(*stdout*, "%d: Thread %d in remainder of code\n", n++, id);  flag1[id] = *true*;  flag2[id] = (id + flag2[1 - id]) % 2;  *while* (flag1[1 - id] == *true* &&  ((flag2[0] + flag2[1]) % 2) == id) {  /\* *busy waiting* \*/  }  /\* *Critical section* \*/  critical++;  *fprintf*(*stdout*, "%d: Thread %d in critical section\n", n++, id);  *assert*(critical <= 1);  *fflush*(*stdout*);  critical--;  flag1[id] = *false*;  *goto* start;  *return* *NULL*;  }  int *main*()  {  pthread\_t p1, p2;  int a1 = 0, a2 = 1;  *pthread\_create*(&p1, *NULL*, *process*, &a1);  *pthread\_create*(&p2, *NULL*, *process*, &a2);  *pthread\_join*(p2, *NULL*);  *pthread\_join*(p1, *NULL*);  *return* 0;  } |