106119029 , OS Lab 8

Dipesh Kafle

Peterson's Algorithm

- I have implemented in such a way that the function to be executed will be passed as an argument and will be executed in the critical section of the algorithm. It will only work for syncronizing two processes although it is possible to do it for more than two as well by changing some code.
- This algorithm assumes that the changes in variables `flags` and `turn` are atomic i.e they will be propagated without any delay and no any interruption can happen when its being changed. Since normal data type don't have that guarantee, I needed to use `atomic_int` and `atomic_bool` types in C to have the algorithm working properly.
- To show its working, I have instantiated two threads that increment the same counter variable and I have used peterson's algorithm to ensure mutual exclusion. And in order to show its benifits, I have also done the same thing on two other threads without any sort of data race prevention.

Code

```c
#include <assert.h>
#include <pthread.h>
#include <stdatomic.h>
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

typedef struct {
  atomic_bool flags[2];
  atomic_int turn;
} petersons_lock_t;

typedef struct {
  int *counter;
  petersons_lock_t *lock;
  int p; // process number basically (can be 0 or 1 only)
} thread_func_arg;

void petersons_lock_init(petersons_lock_t *lock) {
  lock->flags[0] = FALSE;
  lock->flags[1] = FALSE;
  lock->turn = 0;
}
```

```c
// This is similar to given in wikipedia
// The p value controls whether this is process 0 or process 1
void peterson_exec(petersons_lock_t *lock, int p, void (*func)(void *),
                   void *arg) {
  assert(p == 0 || p == 1);
  lock->flags[p] = TRUE;
  lock->turn = !p;
  while (lock->flags[!p] == TRUE && lock->turn == (!p)) {
  }
  // Critical section
  // Will executer without disturbance
  func(arg);
  // End of critical section
  lock->flags[p] = FALSE;
}

void increment(void *num) { (*(int *)num)++; }

void *count(void *param) {
  thread_func_arg *arg = param;
  for (int i = 0; i < 1000000; i++) {
    // this will ensure perfect syncronization between two threads that will be
    // incrementing the same arg->counter variable
    peterson_exec(arg->lock, arg->p, increment, arg->counter);
  }
  return NULL;
}
void *count_without_lock(void *arg) {
  for (int i = 0; i < 1000000; i++) {
    (*(int *)arg)++;
  }
  return NULL;
}
```

```c
int main() {

  pthread_t p1, p2, p3, p4;
  petersons_lock_t lock;
  petersons_lock_init(&lock);
  int counter = 0, counter2 = 0;
  thread_func_arg arg1 = {.counter = &counter, .lock = &lock, .p = 0};
  thread_func_arg arg2 = {.counter = &counter, .lock = &lock, .p = 1};
  pthread_create(&p1, NULL, count, &arg1);
  pthread_create(&p2, NULL, count, &arg2);
  pthread_create(&p3, NULL, count_without_lock, &counter2);
  pthread_create(&p4, NULL, count_without_lock, &counter2);
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  pthread_join(p3, NULL);
  pthread_join(p4, NULL);
  printf("With peterson :%d\n", counter);
  printf("Without peterson :%d\n", counter2);
}
```

Output

```
λ ~/Acads/Sem4/CSLR42-OSLab/Lab8 → ./petersons
With peterson :2000000
Without peterson :2000000
λ ~/Acads/Sem4/CSLR42-OSLab/Lab8 → ./petersons
With peterson :2000000
Without peterson :1309699
λ ~/Acads/Sem4/CSLR42-OSLab/Lab8 → ./petersons
With peterson :2000000
Without peterson :1496433
λ ~/Acads/Sem4/CSLR42-OSLab/Lab8 → ./petersons
With peterson :2000000
Without peterson :2000000
λ ~/Acads/Sem4/CSLR42-OSLab/Lab8 → ./petersons
With peterson :2000000
Without peterson :1228753
λ ~/Acads/Sem4/CSLR42-OSLab/Lab8 →
```

Dekker's Algorithm

- I have again implemented in such a way that the function to be executed will be passed as an argument and will be executed in the critical section of the algorithm.
- To show its working, I have again (like Peterson's) instantiated two threads that increment the same counter variable and I have used `dekker's algorithm` to ensure mutual exclusion. And in order to show its benifits, I have also done the same thing on two other threads without any sort of data race prevention.
- This algorithm assumes that the changes in variables `wants_to_enter` and `turn` are atomic i.e they will be propagated without any delay and no any interruption can happen when its being changed. Since normal data type

don't have that guarantee, I needed to use `atomic_int` and `atomic_bool` types in C to have the algorithm working properly.

Code

```c
#include <assert.h>
#include <pthread.h>
#include <stdatomic.h>
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0

typedef struct {
  atomic_bool wants_to_enter[2];
  atomic_int turn;
} dekker_lock_t;

typedef struct {
  int *counter;
  dekker_lock_t *lock;
  int p; // process number basically (can be 0 or 1 only)
} thread_func_arg;

void lock_init(dekker_lock_t *lock) {
  lock->wants_to_enter[0] = FALSE;
  lock->wants_to_enter[1] = FALSE;
  lock->turn = 0;
}

// Dekker algorithm, for p=0, it acts as process 0 and for p=1,as process 1
//
void dekker_exec(dekker_lock_t *lock, int p, void (*func)(void *), void *arg) {
  assert(p == 1 || p == 0); // ensures the process count is binary
  lock->wants_to_enter[p] = TRUE;
  while (lock->wants_to_enter[!p]) {
    if (lock->turn != p) {
      lock->wants_to_enter[p] = FALSE;
      while (lock->turn != p) {
      }
      lock->wants_to_enter[p] = TRUE;
    }
  }
  // Will be executed exclusively
  // Critical Section
  func(arg);
  // Critical section ens
  lock->turn = !p;
  lock->wants_to_enter[p] = FALSE;
}
void increment(void *num) { (*(int *)num)++; }
void *count(void *param) {
  thread_func_arg *arg = (thread_func_arg *)param;
  for (int i = 0; i < 1000000; i++) {
    dekker_exec(arg->lock, arg->p, increment, arg->counter);
  }
  return NULL;
}

void *count_without_lock(void *arg) {
  for (int i = 0; i < 1000000; i++) {
    (*(int *)arg)++;
  }
  return NULL;
}
```

```c
int main() {

  pthread_t p1, p2, p3, p4;
  dekker_lock_t lock;
  lock_init(&lock);
  int counter = 0, counter2 = 0;
  thread_func_arg arg1 = {.counter = &counter, .lock = &lock, .p = 0};
  thread_func_arg arg2 = {.counter = &counter, .lock = &lock, .p = 1};

  pthread_create(&p1, NULL, count, &arg1);
  pthread_create(&p2, NULL, count, &arg2);
  pthread_create(&p3, NULL, count_without_lock, &counter2);
  pthread_create(&p4, NULL, count_without_lock, &counter2);
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  pthread_join(p3, NULL);
  pthread_join(p4, NULL);
  printf("With dekker: %d\n", counter);
  printf("Without dekker: %d\n", counter2);
}
```

Output