CS232 Operating Systems
Assignment 03: Concurrency and Synchronization
Due : 22nd November 2020

Name: Muhammad Munawwar Anwar
ID: ma04289

November 22, 2020

1 ohours.c

```c
/*
 * A3 Synchronization problem code
 */

#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <assert.h>
#include <stdbool.h>

/*** Constants that define parameters of the simulation ***/

#define MAX_SEATS 3           /* Number of seats in TA's office */
#define TA_LIMIT 10       /* Number of students a TA can help before he needs a break */
#define MAX_STUDENTS 1000       /* Maximum number of students in the simulation */

#define CLASS_OS "OS"
#define CLASS_PFUN "PFUN"

/* Add your synchronization variables here */

sem_t Total_Student;
sem_t chair_lock;
sem_t ta_break;
sem_t os_cv;
sem_t pfun_cv;

bool pfun_flag =false ;
bool os_flag =false;
static int pfun_num_students_waiting = 0;
static int os_num_students_waiting = 0;


/* Basic information about simulation.  They are printed/checked at the end
 * and in assert statements during execution.
 *
 * You are responsible for maintaining the integrity of these variables in the
 * code that you develop.
 */

static int students_in_office;   /* Total numbers of students currently in the office */
static int class_os_inoffice;      /* Total numbers of students from OS class currently in the office */
static int class_pfun_inoffice;      /* Total numbers of students
```

```
   from PFUN class in the office */
static int students_since_break = 0;









typedef struct {
          int arrival_time;  // time between the arrival of this
   student and the previous student
          int question_time; // time the student needs to spend with
   the TA
          char student_class [5];
          int student_id;
  } student_info;

/* Called at beginning of simulation.  Create/initialize all
   synchronization
 * variables and other global variables that you add.
 */
static int
initialize(student_info *si, char *filename) {

  students_in_office = 0;
  class_os_inoffice = 0;
  class_pfun_inoffice = 0;
    students_since_break = 0;

  /* Initialize your synchronization variables (and
        * other variables you might use) here
   */


        /* Read in the data file and initialize the student array */
        FILE *fp;

        if((fp=fopen(filename, "r")) == NULL) {
          printf("Cannot open input file %s for reading.\n", filename
   );
          exit(1);
        }
        int i =0;
        while ( (fscanf(fp, "%d%d%s\n", &(si[i].arrival_time), &(si[i
   ].question_time), si[i].student_class)!=EOF) && i < MAX_STUDENTS )
    {
              i++;
        }
        fclose(fp);
        return i;
}
```

```
96
97  /* Code executed by TA to simulate taking a break
98   * You do not need to add anything here.
99   */
100 static void
101 take_break() {
102   sleep(5);
103         printf("The TA is taking a break now.\n");
104 }
105
106 /* Code for the TA thread. This is fully implemented except for
       synchronization
107  * with the students.  See the comments within the function for
       details.
108  */
109 void *TAthread(void *junk) {
110
111         printf("The TA arrived and is starting his office hours\n");
112
113   /* Loop while waiting for students to arrive. */
114   while (1) {
115
116     /* YOUR CODE HERE. */
117     /* Add code here to handle the student's request.       */
118         /* Currently the body of the loop is empty. There's    */
119         /* no communication between TA and students, i.e. all  */
120         /* students are admitted without regard of the number  */
121       /* of available seats, which class a student is in,     */
122         /* and whether the TA needs a break.                   */
123     sem_wait(&chair_lock);
124
125
126     if (students_since_break == TA_LIMIT && students_in_office == 0)
127     {
128       take_break();
129       for(int i = 0; i <10; i++)
130       {
131         sem_post(&ta_break);
132       }
133       students_since_break = 0;
134
135     }
136
137     sem_post(&chair_lock);
138
139
140
141   }
142
143   pthread_exit(NULL);
144 }
145
146
147 /* Code executed by a OS class student to enter the office.
148  * You have to implement this.  Do not delete the assert() statements
```

```
   ,
 * but feel free to add your own.
 */
void
class_os_enter() {

  /* Request permission to enter the office.  You might also want to
   add  */
        /* synchronization for the simulations variables below
              */
        /*  YOUR CODE HERE.
                                                          */
  sem_wait(&ta_break);
  sem_wait(&Total_Student);
  sem_wait(&chair_lock);

  while (class_pfun_inoffice>0)
  {
    os_num_students_waiting +=1;
    sem_post(&chair_lock);
    sem_wait(&pfun_cv);
    sem_wait(&chair_lock);
  }
  //os_flag = true;
  students_in_office += 1;
  students_since_break += 1;
  class_os_inoffice += 1;
  sem_post(&chair_lock);

}

/* Code executed by a PFUN class student to enter the office.
 * You have to implement this.  Do not delete the assert() statements
   ,
 * but feel free to add your own.
 */
void
class_pfun_enter() {

  /* Request permission to enter the office.  You might also want to
   add  */
        /* synchronization for the simulations variables below
              */
        /*  YOUR CODE HERE.
              */


  sem_wait(&ta_break);
  sem_wait(&Total_Student);
  sem_wait(&chair_lock);
  while (class_os_inoffice>0)
  {
    pfun_num_students_waiting +=1;
    sem_post(&chair_lock);
    sem_wait(&os_cv);
```

```
197     sem_wait(&chair_lock);
198   }
199
200   pfun_flag = true;
201   students_in_office += 1;
202   students_since_break += 1;
203   class_pfun_inoffice += 1;
204   sem_post(&chair_lock);
205
206 }
207
208 /* Code executed by a student to simulate the time he spends in the
      office asking questions
209  * You do not need to add anything here.
210  */
211 static void
212 ask_questions(int t) {
213   sleep(t);
214 }
215
216
217 /* Code executed by a OS class student when leaving the office.
218  * You need to implement this.  Do not delete the assert() statements
      ,
219  * but feel free to add as many of your own as you like.
220  */
221 static void
222 class_os_leave() {
223   /*
224    *  YOUR CODE HERE.
225    */
226
227   sem_wait(&chair_lock);
228   students_in_office -= 1;
229   class_os_inoffice -= 1;
230   if (class_os_inoffice == 0 )
231   {
232     os_flag =false;
233     int i;
234     for (i = 0; i < pfun_num_students_waiting;i++)
235     {
236     sem_post(&os_cv);
237     }
238     pfun_num_students_waiting = 0;
239   }
240   sem_post(&chair_lock);
241   sem_post(&Total_Student);
242
243
244 }
245
246 /* Code executed by a PFUN class student when leaving the office.
247  * You need to implement this.  Do not delete the assert() statements
      ,
248  * but feel free to add as many of your own as you like.
```

```
249  */
250 static void class_pfun_leave() {
251   /*
252    *  YOUR CODE HERE.
253    */
254   sem_wait(&chair_lock);
255   students_in_office -= 1;
256   class_pfun_inoffice -= 1;
257   if (class_pfun_inoffice == 0)
258   {
259     pfun_flag =false;
260     int i;
261     for(i = 0; i <os_num_students_waiting;i++)
262     {
263     sem_post(&pfun_cv);
264     }
265     os_num_students_waiting = 0;
266   }
267   sem_post(&Total_Student);
268   sem_post(&chair_lock);
269
270 }
271
272 /* Main code for OS class student threads.
273  * You do not need to change anything here, but you can add
274  * debug statements to help you during development/debugging.
275  */
276 void*
277 class_os_student(void *si) {
278   student_info *s_info = (student_info*)si;
279
280   /* enter office */
281   class_os_enter();
282
283   assert(students_in_office <= MAX_SEATS && students_in_office >= 0);
284   assert(class_pfun_inoffice >= 0 && class_pfun_inoffice <= MAX_SEATS
       );
285   assert(class_os_inoffice >= 0 && class_os_inoffice <= MAX_SEATS);
286   assert((class_os_inoffice == 0 && class_pfun_inoffice>=0) || (
       class_os_inoffice >= 0 && class_pfun_inoffice==0));
287
288         /* ask questions  --- do not make changes to the 3 lines
       below*/
289         printf("Student %d from OS class starts asking questions for
       %d minutes\n", s_info->student_id, s_info->question_time);
290     ask_questions(s_info->question_time);
291         printf("Student %d from OS class finishes asking questions
       and prepares to leave\n", s_info->student_id);
292
293   /* leave office */
294   class_os_leave();
295
296   assert(students_in_office <= MAX_SEATS && students_in_office >= 0);
297   assert(class_pfun_inoffice >= 0 && class_pfun_inoffice <= MAX_SEATS
       );
```

```
298   assert(class_os_inoffice >= 0 && class_os_inoffice <= MAX_SEATS);
299   assert((class_os_inoffice == 0 && class_pfun_inoffice>=0) || (
        class_os_inoffice >= 0 && class_pfun_inoffice==0));
300
301   pthread_exit(NULL);
302 }
303
304 /* Main code for PFUN class student threads.
305  * You do not need to change anything here, but you can add
306  * debug statements to help you during development/debugging.
307  */
308 void*
309 class_pfun_student(void *si) {
310   student_info *s_info = (student_info*)si;
311
312   /* enter office */
313   class_pfun_enter();
314
315   assert(students_in_office <= MAX_SEATS && students_in_office >= 0);
316   assert(class_pfun_inoffice >= 0 && class_pfun_inoffice <= MAX_SEATS
        );
317   assert(class_os_inoffice >= 0 && class_os_inoffice <= MAX_SEATS);
318   assert((class_os_inoffice == 0 && class_pfun_inoffice>=0) || (
        class_os_inoffice >= 0 && class_pfun_inoffice==0));
319
320         printf("Student %d from PFUN class starts asking questions
        for %d minutes\n", s_info->student_id, s_info->question_time);
321   ask_questions(s_info->question_time);
322         printf("Student %d from PFUN class finishes asking questions
        and prepares to leave\n", s_info->student_id);
323
324   /* leave office */
325   class_pfun_leave();
326
327   assert(students_in_office <= MAX_SEATS && students_in_office >= 0);
328   assert(class_pfun_inoffice >= 0 && class_pfun_inoffice <= MAX_SEATS
        );
329   assert(class_os_inoffice >= 0 && class_os_inoffice <= MAX_SEATS);
330   assert((class_os_inoffice == 0 && class_pfun_inoffice>=0) || (
        class_os_inoffice >= 0 && class_pfun_inoffice==0));
331
332   pthread_exit(NULL);
333 }
334
335 /* Main function sets up simulation and prints report
336  * at the end.
337  */
338 int main(int nargs, char **args) {
339   int i;
340   int result;
341         int student_type;
342     int num_students;
343         void *status;
344         pthread_t ta_tid;
345         pthread_t student_tid[MAX_STUDENTS];
```

```
346         student_info s_info[MAX_STUDENTS];
347     sem_init(&Total_Student,0,MAX_SEATS);
348     sem_init(&chair_lock,0,1);
349     sem_init(&ta_break,0,TA_LIMIT);
350     sem_init(&pfun_cv,0,0);
351     sem_init(&os_cv,0,0);
352 
353   if (nargs != 2) {
354     printf("Usage: officehour <name of inputfile>\n");
355     return EINVAL;
356   }
357 
358   num_students = initialize(s_info, args[1]);
359   if (num_students > MAX_STUDENTS || num_students <= 0) {
360     printf("Error:  Bad number of student threads. Maybe there was a
      problem with your input file?\n");
361     return 1;
362   }
363 
364   printf("Starting officehour simulation with %d students ...\n",
365     num_students);
366 
367   result = pthread_create(&ta_tid, NULL, TAthread, NULL);
368   if (result) {
369     printf("officehour:  pthread_create failed for TA: %s\n",
      strerror(result));
370                 exit(1);
371   }
372 
373   for (i=0; i < num_students; i++) {
374 
375                 s_info[i].student_id = i;
376     sleep(s_info[i].arrival_time);
377 
378             student_type = rand() % 2;
379 
380     if (strcmp (s_info[i].student_class, CLASS_OS)==0)
381       result = pthread_create(&student_tid[i], NULL, class_os_student
      , (void *)&s_info[i]);
382     else // student_type == CLASS_PFUN. assuming input is all correct
      !
383       result = pthread_create(&student_tid[i], NULL,
      class_pfun_student, (void *)&s_info[i]);
384 
385     if (result) {
386       printf("officehour: thread_fork failed for student %d: %s\n",
387             i, strerror(result));
388                         exit(1);
389     }
390 
391   }
392 
393   /* wait for all student threads to finish */
394   for (i = 0; i < num_students; i++)
395     pthread_join(student_tid[i], &status);
```

```
396
397   /* tell the TA to finish. */
398   pthread_cancel(ta_tid);
399
400   printf("Office hour simulation done.\n");
401
402   return 0;
403 }
```

Listing 1: ohours.c

2 - Design.pdf

CS232 Operating Systems
Assignment 03: Concurrency and Synchronization
Design document

Name: Muhammad Munawwar Anwar
ID: ma04289

November 22, 2020

Task 1 - TA's office has a maximum of 3 seats

Total_Student is a semaphore which is initialised with the value MAX_SEATS. Whenever class_pfun_enter() or class_os_enter() is run, sem_wait(&Total_Students) is called which decrements the value of semaphore Total_Student by 1. Whenever class_pfun_leave() or class_os_leave() is run sem_post (&Total_Students) is called which increments the value of Total_Student by 1. When the value of Total_Student is 0, any student who wants to enter the TA's office will have to wait until the value of Total_Student is > 0 and the student thread is enqued on a queue. Thereby, ensuring that no more than 3 students can enter the TA's office at the same time. Whenever the the value of Total_Student is > 0, a student thread is dequed and the student enters the TA's office.

chair_Lock is a semaphore which is initialised with a value of 1 . The semaphore chair_lock acts as a central lock which must be acquired by a thread before it updates the values of students_inoffice, class_pfun_inoffice, class_os_inoffice and students_since_break. After updating these values, the thread releases the lock. chair_lock prevents a race condition from occuring by ensuring that only one thread enters the critical section at one time.

Task 2 - There are no PFUN and OS students in the TA's office at the same time

os_cv and pfun_cv are semaphores which are initialised with the value 0. Both os_cv and pfun_cv act as conditional variables. When OS students are in the TA's office pfun_cv causes the PFUN threads to sleep and when PFUN students are in the TA's office os_cv causes the OS threads to sleep. Therefore ensuring mutual exclusion between the OS and PFUN students threads. os_flag is True when OS students are in the TA's office and False when PFUN students are in the TA's office. pfun_flag is True when PFUN students are in the TA's office and False when OS students are in the TA's office. Both os_flag and pfun_flag are initialised as False. pfun_num_students_waiting keeps track the number of PFUN students that are waiting when the OS students are in the TA's office and os_num_students_waiting keeps the track number of OS students waiting when PFUN students are in the TA's office.

When the class_pfun_enters() runs, it initially checks that students since break is less than 10 and the number of students in office is less than 3. It then acquires chair_lock and checks if os_flag is set to True. If os_flag is False, then it sets pfun_flag to True and updates values accordingly. However, if the os_flag is True, class_pfun_enters() calls sem_wait(&os_cv) which causes PFUN student thread to sleep.But before calling sem_wait(&os_cv), class_pfun_enters() releases the chair_lock and increments the value of pfun_num_students_waiting by 1. By releasing the chair_lock,class_pfun_enters() ensures that there is no deadlock. When the class_os_leave() runs, it checks that if the class_os_in_office is equal to 0 and os_flag is set to True. If this is the case, then class_os_leave()

calls sem_post(&os_cv) pfun_num_students_waiting times to wake all the sleeping PFUN student threads. When class_pfun_enters() runs, it re-acquires the chair_lock first and then updates the values.

When the class_os_enters() runs, it initially checks that students since break is less than 10 and the number of students in office is less than 3. It then acquires chair_lock and checks if pfun_flag is set to True. If pfun_flag is False, then it sets os_flag to True and updates values accordingly. However, if the pfun_flag is True, class_os_enters() calls sem_wait(&pfun_cv) which causes OS student thread to sleep.But before calling sem_wait(&pfun_cv), class_os_enters() releases the chair_lock and increments the value of os_num_students_waiting by 1. By releasing the chair_lock,class_os_enters() ensures that there is no deadlock. When the class_pfun_leave() runs, it checks that if the class_pfun_in_office is equal to 0 and pfun_flag is set to True. If this is the case, then class_pfun_leave() calls sem_post(&pfun_cv) os_num_students_waiting times to wake all the sleeping OS student threads. When class_OS_enters() runs, it re-acquires the chair_lock first and then updates the values.

Task 3 - TA takes a break after helping 10 students

ta_break is a semaphore which is initialised with the value TA_LIMIT. Whenever class_pfun_enter() or class_os_enter() is run, sem_wait(&ta_break) is called which decrements the value of semaphore ta_break by 1. When the ta_thread() runs, it acquires chair_lock . Then it checks if students_since_break is equal to TA_LIMIT and students_in_office is 0. If this is the case, then it calls sem_post(&ta_break) TA_LIMIT times to wake all sleeping threads who were waiting while the TA was taking a break. When the value of ta_break is 0, any student who wants to enter the TA's office will have to wait until the value of ta_break is > 0 and the student thread is enqued on a queue. Thereby, ensuring that the TA takes a break after 10 students. Whenever the the value of ta_break is > 0, a student thread is dequed and the student enters the TA's office.