CS241 Lawrence Angrave L13 – CSP Solutions. Condition Variables.

Name these concepts:

"Only one process(/thread) can be in the CS at a time"

"If waiting, then another process can only enter the CS a finite number of times"

"If no other process is in the Critical Section then the process can immediately enter the CS"

**Candidate** #4

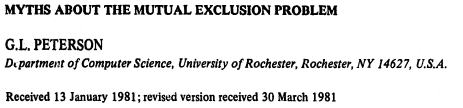
|  |  |
| --- | --- |
| raise my flag  if your flag is raised, wait until my turn  // *Do Critical Section stuff*  turn = *yourid*  lower my flag | raise my flag  if your flag is raised, wait until my turn  // *Do Critical Section stuff*  turn = *yourid*  lower my flag |

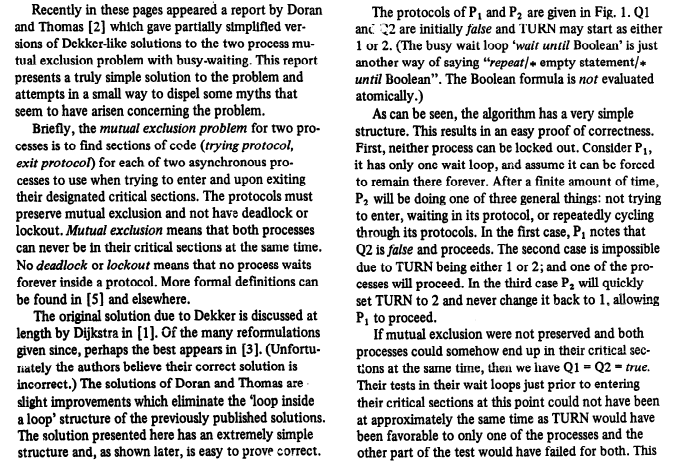
// Threads do other stuff and then will repeat in the future

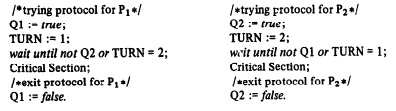
Problems with 4?

**Candidate** #5

|  |  |
| --- | --- |
| raise my flag  while(your flag is raised) :  if it's your turn to win :  lower my flag  wait while your turn  raise my flag  // *Do Critical Section stuff*  set your turn to win  lower my flag | raise my flag  while(your flag is raised) :  if it's your turn to win :  lower my flag  wait while your turn  raise my flag  // *Do Critical Section stuff*  set your turn to win  lower my flag |







**4 Give two reasons why even implementing a 'correct solution' in C might still fail.**

3 What are condition variables? How do you use them? What is *Spurious Wakeup*?

Use a condition variable to wait for a data structure to have at least one item. There is one thread that might be calling *pushdata* or delete several times. Another thread that might call *getLast* several times

Plan:

Write a busy wait.

Add mutex  
 Add condition variable.

|  |  |
| --- | --- |
| pthread\_mutex\_t m;  pthread\_condition\_t cv;  float myarray[10];  int count ;  void init() {  pthread\_mutex\_init(& m, NULL);  pthread\_condition\_init( & cv, NULL);  }  int pushdata(float v) {  myarray[count ++] = v;  }  void delete() {  count --;  } | float getLast() {      // 'result' must be valid!  float result = myarray[count];  return result;  } |

7. Use a CV to implement a simple version of a *counting* *semaphore*

Note a real semaphore might implement a queue of waiting threads to ensure fairness (and avoid *starvation*).

|  |  |
| --- | --- |
| sem\_init(sem\_t \*s, int shared, int value) {  } | typedef struct sem\_t {  } sem\_t; |
| sem\_post(sem\_t\*s) {  } | sem\_wait(sem\_t\*s) {  } |

|  |  |
| --- | --- |
| pthread\_mutex\_t m;  pthread\_condition\_t cv;  float myarray[10];  int count ;  void init() {  pthread\_mutex\_init(& m, NULL);  pthread\_condition\_init( & cv, NULL);  }  int pushdata(float v) {  **p\_m\_lock(&m)**  myarray[count ++] = v;  **ring\_bell(cv)=signal**  **p\_m\_unlock(&m)**  }  void delete() {  **p\_m\_lock(&m)**  count --;  **p\_m\_unlock(&m) // what would happen if we forget this one?**  } | float getLast() {    **p\_m\_lock(&m)**  **while( count == 0 ) {**  **unlock(&m)**  **meditate(&cv)**  **lock(&m) // == cond\_wait(&cv,&m)**  **}**  // 'result' must be valid!  float result = myarray[count];  **p\_m\_unlock(&m);**  return result;  } |
| 1. Add busy loop. 2. "Fix" mutual exclusion 3. Add pseudo cv | 4. what if we wrote unlock before result = ??  5. What if we forgot an unlock?  6 How many threads can be inside the CS? Why? |
| sem\_init(sem\_t \*s, int shared, int value) {  s->count = value;  pthread\_mutex\_init(& s->m, NULL);  pthread\_condition\_init( & s->cv, NULL);  } | typedef struct sem\_t {  int count;  pthread\_mutex\_t m;  pthread\_condition\_t cv;  } sem\_t; |
| sem\_post(sem\_t\*s) {  pthread\_mutex\_lock(& s->m);  s->count ++;  if(s->count ==1) /\* Wake up one waiting thread!\*/  pthread\_mutex\_signal( s->cv);  pthread\_mutex\_unlock(& s->m);  } | sem\_wait(sem\_t\*s) {  pthread\_mutex\_lock(& s->m);  while(s->count <=0) {  pthread\_cond\_wait( &s->cv, &s->m);  }  s->count --;  pthread\_mutex\_unlock(& s->m);  } |