Sistemes Operatius II
Avaluació parcial – Parcial part 2 – 20 de desembre del 2018

Nom i Cognoms: _____

En total hi ha 3 problemes. Podeu fer servir apunts o portàtil per fer consultes a les transparències però no es pot programar cap codi. Només cal respondre breument a les preguntes. No es puntuaran preguntes no raonades.

Problema 1 (concurrència, dificultat baixa, 2.5 punts). Suposeu el següent codi

```
01    variables globals:
02          pthread_mutex_t mutex1, mutex2;
03
04    funcio1:
05          lock(&mutex1);
06          /* funcio 1, part 1 */
07          lock(&mutex2);
08          /* funcio 1, part 2 */
09          unlock(&mutex2);
10          /* funcio 1, part 3 */
11          unlock(&mutex1);
12
13    funcio2:
14          lock(&mutex2);
15          /* funcio 2, part 1 */
16          lock(&mutex1);
17          /* funcio 2, part 2 */
18          unlock(&mutex1);
19          /* funcio 2, part 3 */
20          unlock(&mutex2);
```

Es demana

1. En aquest codi les variables mutex1 i mutex2 són variables globals. Fa falta que siguin globals? No poden ser locals a cadascuna de les dues funcions, funcio1 i funcio2? Raona la resposta.

Les variables mutex1 i mutex2 serveixen protegir a variables compartides entre les seccions crítiques corresponents a funcio1 i funcio2. En cas que les variables mutex1 i mutex2 siguin locals, cada funció tindrà la seva pròpia còpia de les variables i, per tant, no seran visibles per a l'altre funció. Cal que siguin globals perquè les funcions sàpiguen quan l'altre funció ha entrat a la secció crítica protegides per mutex1 i mutex2.

2. Indica quin problema es pot produir en aquest codi i per què. Raona la resposta indicant els canvis de context que s'han de produir perquè es produeixi el problema.

En aquest cas tenim seccions crítiques niades. No hi ha problema si les seccions crítiques estan niades entre sí. El que cal anar amb compte és l'ordre amb què s'agafen les claus. En aquest cas particular, es pot produir un deadlock. Es pot produir, per exemple, així: la funcio1 agafa la clau mutex1 i es produeix un canvi de context. A continuació la funcio2 agafa la clau mutex2. Així que funcio2 intenti agafar mutex1, no podrà fer-ho ja que la té funcio1. De la mateixa forma, funcio1 no podrà agafar mutex2 ja que la té funcio2. Les dues funcions es quedaran doncs bloquejats de forma

indefinida.

3. Comenta (breument) com solucionar el problema que es dóna al punt 2. Indica com ha de ser el codi de funcio1 i funcio2 perquè el codi funcioni correctament.

Es proposen aquí dues solucions. La primera és fer que funcio1 i funcio2 agafin les claus en el mateix ordre. Així:

```
funcio1:
      lock(&mutex1);
      lock(&mutex2);
      /* funcio 1, part 1 */
      /* funcio 1, part 2 */
      /* funcio 1, part 3 */
      unlock(&mutex2);
      unlock(&mutex1);

funcio2:
      lock(&mutex1);
      lock(&mutex2);
      /* funcio 2, part 1 */
      /* funcio 2, part 2 */
      /* funcio 2, part 3 */
      unlock(&mutex2);
      unlock(&mutex1);
```

Una segona solució es basa en introduir una nova clau, mutex3, que proteixi la secció crítica de les funcions. Així no fa falta canviar l'ordre en què s'agafen les claus:

```
funcio1:
      lock(&mutex3);
      lock(&mutex1);
      /* funcio 1, part 1 */
      lock(&mutex2);
      /* funcio 1, part 2 */
      unlock(&mutex2);
      /* funcio 1, part 3 */
      unlock(&mutex1);
      unlock(&mutex3);

funcio2:
      lock(&mutex3);
      lock(&mutex2);
      /* funcio 2, part 1 */
      lock(&mutex1);
      /* funcio 2, part 2 */
      unlock(&mutex1);
      /* funcio 2, part 3 */
      unlock(&mutex2);
      unlock(&mutex3);
```

Problema 2 (semàfors i monitors, dificultat mitjana, 3.5 punts). A la biblioteca de la Facultat de Matemàtiques i Informàtica hi ha ordinadors disponibles per als alumnes de grau. En total hi ha M ordinadors disponibles i un estudiant qualsevol pot agafar un ordinador sempre que n'hi hagi un de lliure. Un cop l'ha fet servir el torna a la biblioteca.

Disposem d'una variable global que ens indica si l'ordinador està lliure (disponible) o no. El codi de la declaració, així com el d'inicialització, agafar un ordinador i retornar-lo es mostra a continuació.

```
01    variables globals:
02          int M = ...; // un valor sencer, pex 10
03          boolean lliure[M];
04
05    inicialitzacio:
06          int i;
07          for(i = 0; i < M; i++)
08                lliure[i] = true;
09
10    agafar_ordinador:
11          int i;
12          for(i = 0; i < M; i++)
13                if (lliure[i]) {
14                      lliure[i] = false;
15                      return i;
16                }
17          exit(ERROR);
18
19    retornar_ordinador(int i):
20          lliure[i] = true;
21
22    prestec_ordinador:
23          int i;
24          i = agafar_ordinador;
25          /* L'estudiant treballa amb l'ordinador */
26          retornar_ordinador(i);
```

Suposem que s'ha executat la funció d'inicialització per indicar que tots els ordinadors estan disponibles. Respecte la darrera funció, prestec_ordinador, es demana el següent

1. Comenta el problema que es pot produir en cas que múltiples estudiants vulguin fer un préstec d'un ordinador. Indica quina(es) funció(ns) cal protegir, a la funció prestec_ordinador, perquè no es produeixin problemes. Raona la resposta!

Quan hi ha múltiples fils cal protegir les variables globals que comparteixen. En aquest cas es tracta del vector lliure. En particular, a la funció agafar_ordinador pot donar-se el cas en què múltiples estudiants vulguin agafar el mateix ordinador provocant una condició de carrera.

Per altra banda, també es convenient protegir la funció retornar_ordinador, en especial en sistemes muliprocessador. A simple vista pot semblar que no és necessari, atès que a cada estudiant se li assignarà un ordinador diferent. Per tant, la funció retornar_ordinador escriurà en posicions diferents del vector per a estudiants diferents. El problema rau en el fet que, a nivell de maquinari, l'escriptura és realizarà a la memòria caché del processador, i l'escriptura a memòria RAM es realizarà més endavant. Les funcions de bloqueig com unlock contenen el codi màquina necessari per buidar la memòria cau i escriure-la a memòria RAM, de forma que els altres estudiants puguin veure que s'ha modificat el contingut del vector lliure.

En aquesta prova parcial no s'ha penalitzat el fet de no protegir la funció retornar_ordinador i s'han acceptat les dues solucions: protegir o no protegir-la. Es vol insistir en la importància de protegir les variables globals compartides entre fils. En llenguatge C l'escriptura es realitza a memòria caché i després a memòria RAM (automàticament pel maquinari o bé manualment per la funció unlock). En Java, en canvi, l'actualització de les variables compartides es realitza un cop es crida a la funció unlock: és la màquina virtual la que realitza aquesta actualització.

El que segueix a continuació suposa que s'han de bloquejar totes dues funcions, agafar_ordinador i retornar_ordinador.

2. Indica i comenta com s'ha de modificar el codi de prestec_ordinador perquè no es produeixin problemes en cas que múltiples estudiants vulguin fer un préstec d'un ordinador al mateix temps. S'han de fer servir semàfors.

Es pot fer fent servir un semàfor binari global, s, inicialitzat a 1.

```
prestec_ordinador:
      int i;
      sem_wait(&s);
      i = agafar_ordinador;
      sem_post(&s);
      /* L'estudiant treballa amb l'ordinador */
      sem_wait(&s);
      retornar_ordinador(i);
      sem_post(&s);
```

3. Atès el fet que només hi ha M ordinador disponibles, comenta com s'ha de modificar el codi de prestec_ordinador per assegurar que l'estudiant trobi un ordinador lliure sempre que en vulgui agafar un (és a dir, que no s'executi la línia 17 del codi). S'han de fer servir semàfors.

Mitjançant el codi anterior bloquegem l'accés a les funcions agafar_ordinador i retornar_ordinador. El que es vol fer ara és, a més, limitar els recursos disponibles. Això es pot fer amb un semafor general, que anomenarem disponibles, inicialitzat a M.

```
prestec_ordinador:
      int i;
      sem_wait(&disponibles);
      sem_wait(&s);
      i = agafar_ordinador;
      sem_post(&s);
      /* L'estudiant treballa amb l'ordinador */
      sem_wait(&s);
      retornar_ordinador(i);
      sem_post(&s);
      sem_post(&disponibles);
```

4. En cas que es vulguin fer servir monitors comenta breument com solucionar els punts 2 i 3. Pots fer referència a una transparència directament si així ho desitges.

Per solucionar el problema fent servir monitors: per al punt 2 es fan servir semàfors binaris i per tant es poden fer servir les funcions lock/unlock. Per al semàfor general del punt 3, es pot fer servir la implementació dels semàfors mitjançant monitors indicada a la transparència 12 dels monitors.

Problema 3 (monitors, dificultat alta als darrers exercicis, 4 punts). A continuació es mostra el següent algorisme, vist a classe de teoria, dels lectors i escriptors justos per a monitors.

```
01    variables globals:
02          int nr = 0; boolean w = false;
03          pthread_mutex_t mutex; pthread_cond_t cond;
04
05    read_lock:
06          lock(&mutex);
07          while (w) {
08                wait(&cond, &mutex);
09          }
10          nr++;
11          unlock(&mutex);
12
13    read_unlock:
14          lock(&mutex);
15          nr--;
16          if (nr == 0) broadcast(&cond);
17          unlock(&mutex);
18
19    write_lock:
20          lock(&mutex);
21          while (w) {
22                wait(&cond, &mutex);
23          }
24          w = true;
25          while (nr != 0) {
26                wait(&cond, &mutex);
27          }
28          unlock(&mutex);
29
30    write_unlock:
31          lock(&mutex);
32          w = false;
33          broadcast(&cond);
34          unlock(&mutex);
```

Es demana contestar a les següents preguntes breument:

1. Què és el que fa que aquest algorisme sigui considerat un algorisme just? En altres paraules, com s'assegura que els escriptors impedeixin l'entrada als lectors a la seva zona crítica?

L'algorisme és just gràcies a la forma amb què es fa servir la variable w. Suposem que hi ha lectors llegint. En cas que arribi un escriptor, aquest posarà la variable w a true. Tots els lectors que arribin a continuació es quedaran adormits a la línia 08. Els lectors aniran sortint un a un fent servir la funció read_unlock. Un cop hagin sortit tots els lectors, l'escriptor hi podrà entrar. Els escriptors no s'hauran d'esperar de forma indefinida per poder escriures com pot passar amb l'altre algorisme. Un cop un escriptor notifica que vol escriure, no s'admet l'entrada de nous lectors a la secció crítica.

2. Suposem que hi ha un escriptor escrivint i que hi ha lectors adormits al wait de la línia 8 així com escriptors adormits al wait de la línia 22. En sortir l'escriptor de la seva secció crítica crida la funció write_unlock, la qual crida al broadcast de la línia 33. Se sap qui dels dos, lectors o escriptors, podrà entrar a la secció crítica? Comenta breument la resposta.

Atès que nr == 0 i w == false, no se sap quin dels dos, lectors o escriptors, podrà agafar la clau per poder entrar a la secció crítica. Tant les lectors com els escriptors competiran per poder-la agafar i qualsevol dels dos la pot obenir.

3. (Difícil) Es proposa modificar el codi per donar preferència als escriptors en cas que es doni el cas descrit a la pregunta anterior. Indiqueu clarament les modificacions a realitzar al codi. Les podeu realitzar directament sobre el codi d'aquesta pàgina.

Es mostra una possible solució al problema (n'hi han d'altres vàlides):

```
01    variables globals:
02          int nr = 0; int nw = 0; boolean w = false;
03          pthread_mutex_t mutex; pthread_cond_t cond;
04
05    read_lock:
06          lock(&mutex);
07          while (nw > 0) {
08                wait(&cond, &mutex);
09          }
10          nr++;
11          unlock(&mutex);
12
13    read_unlock:
14          lock(&mutex);
15          nr--;
16          if (nr == 0) broadcast(&cond);
17          unlock(&mutex);
18
19    write_lock:
20          lock(&mutex);
            nw++;
21          while (w) {
22                wait(&cond, &mutex);
23          }
24          w = true;
25          while (nr != 0) {
26                wait(&cond, &mutex);
27          }
28          unlock(&mutex);
29
30    write_unlock:
31          lock(&mutex);
            nw--;
32          w = false;
33          broadcast(&cond);
34          unlock(&mutex);
```

4. (Difícil) Comenteu per què les modificacions proposades donen preferència als escriptors.

Al codi anterior es mostren les modificacions a realitzar per donar preferència als escriptors. En concret, es fa servir una variable nw per saber quants escriptors hi ha esperant per poder escriure. Si n'hi ha algun que vulgui escriure, els lectors s'esperaran fins que no en quedi cap. Observar que el codi compleix amb els requisits que ha de complir el paradigma dels lectors-escriptors: hi pot haver múltiples lectors llegint a la vegada, però només hi pot haver un escriptor escrivint en un moment determinat.