UNIVERSITAT ROVIRA I VIRGILI

Fonaments de sistemes operatius. Pràctica 2.

Ferrero Ramos, Arey

Kaddouch, Hatim

Villaró Serrano, Oriol

Índex

Especificacions

Es subministra el codi d'una versió rudimentària del Joc del 'Tron'. En aquesta versió inicial, el codi és seqüencial i, per tant, hi ha un sol tron oponent.

Fase 1

Es demana modificar el codi seqüencial subministrat per a que les funcions encarregades del moviment del tron usuari i del tron oponent actuïn com a fils d'execució independents. Això ha de permetre crear un nombre determinat de trons oponents. Per a fer-ho, s'hauran d'utilitzar les eines subministrades en la biblioteca de *pthread.h*.

Fase 2

Es demana solucionar els problemes de concurrència que, inevitablement, es produiran durant l'execució de la fase 1 degut a l'accés simultani dels diferents *threads* als recursos compartits. Per a fer-ho, caldrà establir seccions crítiques en el codi de la fase anterior. Això s'aconseguirà amb la utilització dels semàfors d'exclusió mútua per a *threads*: Els *mutex*.

Fase 3

Es demana modificar el codi de la fase 1 per a que la funció encarregada del moviment del tron oponent sigui un procés diferent i la funció encarregada del moviment del tron usuari i la funció encarregada del comptatge del temps funcionin com a fils d'execució independents. Això ha de permetre un nombre determinat de trons oponent. Per a fer-ho es necessari utilitzar les funcions d'accés a memòria subministrades en el fitxer memòria.c.

Per a facilitar la correcció dels diferents aspectes que es demanaven en la fase 4 de l'enunciat de la pràctica, aquesta fase s'ha dividit en dues fases diferents:

Fase 4

Es demana solucionar els problemes de concurrència que, inevitablement, es produiran durant l'execució de la fase 3 degut a l'accés simultani dels diferents processos i *threads*

als recursos compartits. Per a fer-ho, caldrà establir seccions crítiques en el codi de la fase anterior. En els cas dels *threads*, això s'aconseguirà utilitzant semàfors d'exclusió mútua per a *threads* o *mutex*. Per als processos, s'haurà d'utilitzar un altre tipus de semàfors. Les eines per a desenvolupar seccions crítiques per a processos es subministren en el fitxer semàfor.c.

Fase 5

Es demana modificar el codi de la fase anterior per a implementar una nova modalitat de joc que incorpora els següents canvis:

- Si un tron A xoca contra un tron B, el tron A canviarà de direcció (excepte que sigui el tron usuari, que serà destruït) i el tron B s'haurà d'escriure en estat invers durant 10 segons.
- Si un tron C xoca contra la seqüència no inversa del tron B, el tron C continuarà amb la seva execució normal i el tron B serà destruït.

Per a aconseguir implementar aquesta nova modalitat de joc, serà necessari que els trons puguin interaccionar entre ells, cosa que s'aconseguirà incorporant la funcionalitat de pas de missatges. Això s'aconseguirà utilitzant bústies. Les eines per a implementar el pas de missatges es subministren en el fitxer missatge.c.

Disseny i implementació

Fase 1

Per a convertir les funcions encarregades del moviment dels trons usuari i oponents en fils d'execució independents, n'hi hagut prou amb entendre el funcionament del codi d'exemple del laboratori sobre creació de threads, com l'ús de les funcions pthread_create() i pthread_join(); i amb seguir les especificacions de l'enunciat de la pràctica que proporcionaven un procediment molt pautat i explicatiu. En conseqüència, no s'ha hagut de prendre cap decisió de disseny rellevant durant el desenvolupament d'aquesta fase.

Fase 2

En aquesta fase s'han hagut de decidir les seccions crítiques que ha de tenir el codi després la conversió de les funcions encarregades del moviment dels trons usuari i oponents. En el cas de la funció mou_usuari(), hi ha dues seccions crítiques. La primera serveix per protegir el codi de la funció win_gettec():

```
pthread_mutex_lock(&mutex);
tecla = win_gettec();
pthread_mutex_unlock(&mutex);
```

La funció win_gettec() serveix per obtenir una tecla introduïda per l'usuari. Si el codi d'aquesta funció no fos una secció crítica, el valor de la tecla introduïda per l'usuari es podria perdre.

La segona secció crítica de la funció mou_usuari() serveix per protegir el codi que s'executa des de que s'obté el caràcter de la següent posició del taulell de joc que ha d'ocupar el tron usuari amb la funció win_quincar() fins que s'escriu el següent element del rastre en dita posició amb la funció win_escricar() en cas que el caràcter anterior fos un espai en blanc:

```
pthread_mutex_lock(&mutex);
cars = win_quincar(seg.f,seg.c);
if (cars == ' ')
{
```

```
      usu.f = seg.f;
      usu.c = seg.c;
      win_escricar(usu.f,usu.c,'0',INVERS);
      pthread_mutex_unlock(&mutex);
      ...
}
else
{
      pthread_mutex_unlock(&mutex);
      ...
}
```

En la funció mou_oponent() hi ha una única secció crítica semblant a la segona secció crítica de la funció mou_usuari() però, en aquest cas, és molt més amplia ja que hi ha moltes més instruccions entre la funció win_quincar() i la funció win_escricar():

```
pthread_mutex_lock(&mutex);
cars = win_quincar(seg.f,seg.c);
if (cars != ' ')
      canvi = 1;
else
      if (varia > 0)
      {
            k = rand() % 10;
            if (k < varia) canvi = 1;
      }
if (canvi)
{
      nd = 0;
      for (k=-1; k<=1; k++)
      {
            vk = (opo[(intptr_t) index].d + k) % 4;
            if (vk < 0) vk += 4;
            seg.f = opo[(intptr_t) index].f + df[vk];
```

```
            seg.c = opo[(intptr_t) index].c + dc[vk];
            cars = win_quincar(seg.f,seg.c);
            if (cars == ' ')
            {
                  vd[nd] = vk;
                  nd++;
            }
      }
      if (nd == 0)
            fi2 = 1;
      else
      {
            if (nd == 1)
                  opo[(intptr_t) index].d = vd[0];
            else
                  opo[(intptr_t) index].d = vd[rand() % nd];
      }
}
if (fi2 == 0)
{
      opo[(intptr_t)  index].f  =  opo[(intptr_t)  index].f  +
df[opo[(intptr_t) index].d];
      opo[(intptr_t)  index].c  =  opo[(intptr_t)  index].c  +
dc[opo[(intptr_t) index].d];
      win_escricar(opo[(intptr_t)         index].f,opo[(intptr_t)
index].c,'1' + (intptr_t) index,INVERS);
      pthread_mutex_unlock(&mutex);
      ...
}
else
{
      pthread_mutex_unlock(&mutex);
      ...
}
```

Aquestes dues seccions crítiques busquen protegir el mateix tipus de codi i son importants per dos motius. Primer de tot, perquè és imprescindible que el codi de la funció win_escricar() estigui contingut en una secció crítica, ja que l'accés sense sincronisme a aquesta funció és la causa de gairebé la totalitat dels errors de concurrència que es produïen en la fase anterior. Tot i que protegint únicament el codi de la funció win_escricar() sembla que s'eliminen tots aquests errors, és necessari que la secció crítica sigui més amplia, per a evitar que s'interrompi l'execució del codi que es troba entre la funció win_quincar() i la funció win_escricar(). Així, podria passar que un dels trons detectés que la següent posició a la que es desplaçarà és un espai en blanc però després la seva execució fos interrompuda i un altre tron escrivís el seu rastre en aquesta posició, de manera que quan es retornés a l'execució del tron que s'havia interromput, aquest escriuria el seu rastre per sobre del tron que ha ocupat l'espai en blanc que ell volia ocupar abans. En conseqüència, un dels trons s'escriuria a sobre de l'altre, sense que cap dels dos fos destruït. Cal dir que aquest error és molt poc freqüent i, per tant, molt difícil de detectar amb un joc de proves convencional.

En el main() hi ha una secció crítica que serveix per protegir el codi de la funció win_escristr():

```
pthread_mutex_lock(&mutex);
win_escristr(strin);
pthread_mutex_unlock(&mutex);
```

Aquesta secció crítica té sentit a nivell teòric, ja que la funció win_escristr() és un dels recursos compartits del sistema. No obstant, a nivell pràctica és difícil que doni problemes ja que només es crida una vegada des de la funció inicialitza_joc() i després en el bucle principal del programa per a mostrar el temps de joc. Tot i així, s'ha mantingut perquè mai es pot saber quan s'haurà de cridar des d'un altre punt del programa en una modificació posterior del codi.

En aquesta fase, també s'ha demanat d'implementar uns canvis importants en la modalitat del joc: Per a guanyar la partida, el tron usuari haurà de destruir tots els trons oponents que hagin estat creats i quan un tron xoca, l'esborrat del seu rastre s'ha de produir de manera concurrent al moviment de la resta de trons.

Per a aconseguir aquesta nova modalitat de joc, primer de tots es imprescindible que la crida a la funció esborrar_posicions() estigui fora de les seccions crítiques de les funcions encarregades del moviment dels trons. Això permetrà que l'esborrat del rastre es produeixi de manera concurrent al moviment dels altres trons. No obstant, en el codi de la funció esborrar posicions hi ha una crida a la funció win_escricar() que provocarà nous errors de concurrència. Per tant, s'haurà de establir una nova secció crítica dins de la funció esborrar_posicions() per protegir el codi de la funció win_escricar():

```
pthread_mutex_lock(&mutex);
win_escricar(p_pos[i].f,p_pos[i].c,' ',NO_INV);
pthread_mutex_unlock(&mutex);
```

A més, s'hauran de convertir les variables globals p_opo i n_opo en variables locals de la funció mou_oponent i s'haurà de crear una nova variable anomenada oponents_morts, que s'incrementarà cada vegada que un tron oponent sigui destruit per l'usuari. Com que la variable oponents_morts és una variable global a la que poden accedir els diferents threads, l'acció d'incrementar la variable s'ha de protegir en una nova secció crítica:

```
pthread_mutex_lock(&mutex_morts);
oponents_morts++;
pthread_mutex_unlock(&mutex_morts);
```

A més, per a que el joc només acabi quan el tron usuari mor o ha destruit a tots els trons oponents, s'haurà de convertir la variable fi2 en una variable local de la funció mou_oponent() i redissenyar la condició del bucle principal de joc de la funció mou_oponent() i del main():

```
do
{
      //Cos del bucle
} while ((fi1) && (oponents_morts < n_oponents));
```

Fase 3

La conversió de la funció encarregada del moviment del tron usuari en un fil d'execució independent és un pas que ja s'havia realitzat en la fase 1 i la creació de un nou fil d'execució que s'encarregués del comptatge del temps de joc ha estat un pas trivial un cop s'havien adquirit els coneixements impartits en la fase 1. En conseqüència, el gruix d'aquesta fase ha estat convertir la funció encarregada del moviment dels trons oponents en processos independents. Tot i així, seguint l'exemple del laboratoris sobre creació de processos no s'han hagut de prendre gaires decisions de disseny. En essència, totes aquelles variables que en la fase 1 s'han hagut de declarar com a variable globals del codi, en aquesta fase s'han hagut de redefinir com a zones de memòria compartida. També s'ha hagut de fer el mateix amb el taulell de joc. Això es degut a que el codi dels

diferents processos està localitzat en fitxers diferents. Llavors, com cada procés a de tenir accés a aquestes zones de memòria compartida, tots els identificadors de les zones de memòria compartida que s'han definit en el programa principal s'han de passar com a paràmetres en la crida execlp():

```
sprintf(a1,"%i",id_camp);
sprintf(a2,"%i",n_fil);
sprintf(a3,"%i",n_col);
sprintf(a4,"%i",varia);
sprintf(a5,"%i",retard);
sprintf(a6,"%i",id_fi1);
sprintf(a7,"%i",id_oponents_morts);
sprintf(a8,"%i",i);
sprintf(a9,"%i",n_oponents);
execlp("./oponent3", "oponent3", a1, a2, a3, a4, a5, a6, a7, a8, a9, ll_args[3], (char *) 0);
```

Tot i que no ha sigut una decisió de disseny com ha tal, hem volgut destacar també un error que s'ha produït en aquesta fase per les grans dificultats que hem tingut per a resoldre'l. Aquest error és una altra mostra de les grans dificultats que suposa el treballar amb punters en el llenguatge de programació C. En la crida de la funció win_set(), el primer paràmetre que se li passa és la variable p_camp, la qual en la capçalera d'aquesta funció apareix acompanyada d'un *. Això ens ha fet assumir que el que rebia la funció win_set() era la referència de la funció p_camp i que, per tant, en la crida a la funció aquesta variable s'havia de passar acompanyada d'un &:

```
win_set(&p_camp, n_fil, n_col);
```

No obstant, aquesta deducció és incorrecta perquè la variable p_camp és un punter al camp de joc. En conseqüència, en la capçalera de la funció win_set(), el símbol * no representa un pas per referència sinó que la variable p_camp ha estat definida com un punter una zona de memòria compartida. Per tant, la crida correcta a la funció win_set() és la següent:

```
win_set(p_camp, n_fil, n_col);
```

Un altre dificultat que s'ha hagut de resoldre és que, a diferència de en les dues fases anteriors, els trons prenien tots la mateixa direcció inicial en cada nova execució del programa. Això es degut a que la funció rand() repeteix sempre el mateix nombre aleatori per a cada tron del joc en cada execució del programa. Per solucionar aquest problema és necessari utilitzar la funció srand(), que serveix per modificar la llavor dels nombres aleatoris. No obstant, la dificultat en aquest punt és decidir quin valor passar-li a la funció srand(), ja que ha de ser un valor diferent per a cada un dels trons però que alhora varií amb cada execució del programa:

```
srand(time(NULL) + index);
```

Com que la funció time(NULL) retorna la data i hora actuals, el seu valor serà diferent en cada execució del programa i la variable índex és diferent per a cada tron. En conseqüència, la suma de d'aquests dos valors passat com a paràmetre de la funció srand() permet a la funció rand() generar un nombre aleatori diferent per a cada un dels trons oponents en cada execució del programa. Per tant, els trons sempre tindran direccions inicials diferents.

Fase 4

La major part de les seccions crítiques definides en aquesta fase son exactament les mateixes que en la fase 2. L única diferència és que en el fitxer que conté el programa principal i els diferents *threads* s'utilitzen *mutex* com en dita fase mentre que el fitxer que conté el codi del procés que representa cada un dels oponents s'utilitzen semàfors n-aris. A més, s'afegeixen dues noves seccions crítiques. La primera serveix per protegir el codi de la funció win_update():

```
pthread_mutex_lock(&mutex);
win_update();
pthread_mutex_unlock(&mutex);
```

La funció win_update() és una funció que serveix per repintar tot el taulell de joc en un sistema en el que es treballa amb múltiples processos (a partir de la fase 3) i, per tant, la seva execució cada cert temps és imprescindible per mantenir el camp de joc actualitzat. Per tant, la seva execució no pot ser interrompuda ja que llavors s'introduiran caràcters erronis en el taulell de joc.

L'altre secció crítica serveix per protegir la crida a la funció win_escricar() en les funcions inicialitza_joc(), tant la del fitxer tron4.c i el threads com la del fitxer oponent4.c:

```
waitS(id_sem);
win_escricar(opo.f,opo.c,'1' + index,INVERS);
signalS(id_sem);
```

La secció crítica del fitxer tron4.c no es realment necessària, ja que quan s'executa la funció inicialitza_joc() del fitxer tron4.c, encara no s'ha iniciat l'execució concurrent del programa i, per tant, no hi haurà cap altre procés o thread que pugui accedir al camp de joc. Tot i així, s'ha mantingut per a recordar la importància de tractar les crides a la funció win_escricar() com a seccions crítiques per garantir l'accés correcte i ordenat dels diferents processos al camp de joc.

Finalment, donada una situació en la que s'havia de produir un empat perquè el tron usuari aconseguia destruir a tots els trons oponents però després xocava durant l'esborrat de l'últim o últims d'aquests trons, s'ha vist que quan s'acabava l'esborrat de l'últim tron oponent es parava en sec l'esborrat del tron usuari, i el programa es quedava penjat durant un temps variable abans d'acabar la seva execució correctament. Això era degut a que, una vegada tots els trons oponents havien sigut destruïts, es sortia del bucle principal del joc del main() i per tant, ja no es continuava cridant a la funció win_update() per actualitzar el camp de joc tot i que a nivell lògic es seguia esborrant el rastre del tron usuari. Per solucionar això, s'ha decidit afegir una crida a la funció win_update en el codi d'esborrar posicions. Aquesta crida està localitzada dins d'un condicional a l'interior del qual només s'accedeix si tots els trons oponents han estat destruïts. Aquesta crida està inclosa dins de la secció crítica que protegeix la funció win_escricar():

```
pthread_mutex_lock(&mutex);
win_escricar(p_pos[i].f,p_pos[i].c,' ',NO_INV);
if (*oponents_morts == n_oponents)
     win_update();
pthread_mutex_unlock(&mutex);
```

També passa el mateix quan l'últim tron oponent viu és destruït durant el procés d'esborrat del rastre del tron usuari al que ha aconseguit eliminar. No obstant, degut a que la funció win_update() només es pot cridar des de el fitxer tron4.c, aquest problema no té solució possible.

Fase 5

La primera decisió que s'ha de prendre per aconseguir implementar el pas de missatges entre els diferents trons és si crear una bústia per a cada un dels trons o si utilitzar una sola bústia compartida entre tots els trons. Les dues opcions tenen avantatges i desavantatges. La primera opció és més fàcil de dissenyar i la seva implementació tindrà menys línies de codi. A més, la càrrega de la CPU serà menor. Tot i així, nosaltres ens hem decantat pel segon sistema. Utilitzar una única bústia per a tots el trons implica que l'espai de la memòria de dades dedicat al pas de missatges pot ser fins a 10 vegades inferior al que s'ocuparia implementant l'opció de una bústia per a cada tron.

Dit això, s'ha de especificar que s'han creat dues bústies. La bústia de xocs s'encarregarà de emmagatzemar els missatges enviats per a que el tron xocat passi a escriure el seu rastre d'estat invers a estat no invers mentre que la bústia de morts servirà per emmagatzemar els missatges enviats per un tron que ha xocat amb el rastre no invers d'un altre tron que haurà de ser destruït.

La primera dificultat que s'ha de resoldre en el disseny del pas de missatges és com utilitzar la funció receiveM(). Aquesta funció bloqueja l'execució del procés o *thread* que la invoca fins que hi hagi un nou missatge a la bústia. Això implica que si s'executa des de les funcions encarregades del moviment dels trons aquestes es quedaran bloquejades a l'espera de rebre un missatge i, per tant, els trons no es mouran. La millor manera de solucionar aquest problema es que cada una de les funcions encarregades del moviment dels trons creï un nou fil d'execució que s'encarregui de cridar a la funció receiveM(). Que aquest fil d'execució estigui bloquejat no suposarà cap problema perquè les funcions encarregades del moviment dels trons podran continuar la seva execució amb normalitat. Si s'implementa aquesta solució s'ha de tenir en compte que si s'utilitza la funció pthread_join() per a recollir el *thread* i aquest *thread* està bloquejat en la funció receiveM(), això pot donar lloc a un interbloqueig. En conseqüència, no es recomanable utilitzar pthread_join() per a recollir aquests *threads*. A diferència del cas dels *threads* implementats en la fase 1, això no suposa cap problema, ja que no es necessari que el thread pare esperi a que els threads fills acabin per a continuar amb l'execució de la última part del codi. Cada un d'aquests *threads* ja es destruirà quan s'acabi l'execució de la respectiva funció encarregada del moviment de cada tron.

El principal problema d'utilitzar una única bústia per a tots els trons, és que quan un tron extreu un missatge de la bústia, el destrueix després de llegir-lo. En conseqüència si el primer tron que llegeix el missatge no és per al que anava dirigit, aquest missatge es perdrà i no es faran els canvis corresponents en el tron que l'havia de rebre. La solució que nosaltres hem donat a aquest problema és que si el missatge no correspon al tron que l'ha consumit, aquest tron senzillament tornarà a enviar el missatge a la bústia. Es mostra aquesta solució en la gestió de la bústia de xocs:

```
receiveM(id_bustia_xocs, missatge);
```

```
if (*missatge == ('1' + (intptr_t) index)) {
      invers = 0;
      win_retard(10000);
      invers = 1;
}
else
      sendM(id_bustia_xocs, missatge, 1);
```

En el cas contrari de la bústia de xocs, es desactivarà una variable de dos estats anomenada `invers` que serà l'encarregada d'indicar a la funció encarregada del moviment dels tron corresponent si el tron s'ha d'escriure de manera inversa o de manera no inversa:

```
if (invers)
      win_escricar(opo.f,opo.c,'1' + index,INVERS);
else
      win_escricar(opo.f,opo.c,'1' + index,NO_INV);
```

Ara es mostra aquesta solució en la gestió de la bústia de morts:

```
receiveM(id_bustia_morts, missatge);
if (*missatge == ('1' + (intptr_t) index))
      mort = 1;
else
      sendM(id_bustia_morts, missatge, 2);
```

En el cas contrari de la bústia de morts, s'activarà una variable de dos estats anomenada `mort` que serà l'encarregada d'indicar a la funció encarregada del moviment del tron corresponent que aquell tron ha de morir:

```
if ((fi2 == 0) && (mort == 0))
{
      opo.f = opo.f + df[opo.d];
```

```
        opo.c = opo.c + dc[opo.d];
        if (invers)
              win_escricar(opo.f,opo.c,'1' + index,INVERS);
        else
              win_escricar(opo.f,opo.c,'1' + index,NO_INV);
        signalS(id_sem);
        p_opo[n_opo].f = opo.f;
        p_opo[n_opo].c = opo.c;
        n_opo++;
}
else
{
        signalS(id_sem);
        esborrar_posicions('1'   +   index,   p_opo,   n_opo,   (void
*)(intptr_t) index);
        waitS(id_sem);
        (*oponents_morts)++;
        signalS(id_sem);
        if (fi2 == 0)
              fi2 = 1;
}
```

No es pot reutilitzar la variable fi2 per a matar el tron en aquesta situació concreta perquè si la variable fi2 fos activada pel *thread* encarregat de gestionar els missatges de la bústia de morts en algun punt del codi entre la funció esborrar_posicions() i la condició de finalització del bucle, el joc acabaria sense que es produís l'esborrat del tron que ha mort. A més, s'ha de senyalar que aquesta situació és la més probable ja que la major part del temps la CPU estarà executant la funció win_retard(), que es troba entre aquestes dos punts del codi. Per tant, si s'esborra el rastre del tron degut a l'activació de la variable `mort`, s'haurà d'activar la variable fi2 per a que es compleixi la condició de finalització del bucle.

Jocs de proves

Fase 1

Prova	Descripció	Correcte?	Observacions
1	Prova de fum.	Sí.	Degut a l'accés simultani dels diferents *threads* al taulell de joc, es produeixen tants errors de concurrència que es molt difícil apreciar si el funcionament del programa es correcta. En conseqüència, la major part de les proves que s'haurien de fer per comprovar el correcte funcionament d'aquesta fase s'integraran en el joc de proves de la fase següent. Tot i així, l'objectiu d'aquesta prova de fum és obtenir una primera impressió positiva sobre si es produeix la execució concurrent dels diferents trons creats.

Fase 2

Prova	Descripció	Correcte?	Observacions
1	Es produeix l'execució concurrent dels *threads* amb el nombre de trons oponents màxim que es pot crear (9 oponents).	Sí.	
2	Aturada del joc utilitzant la tecla return. S'imprimeix la frase 'S'ha aturat el joc amb la tecla RETURN!'.	Sí.	
3	El tron usuari es destruit al xocar contra una paret. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
4	El tron usuari es destruït al xocar contra si mateix. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
5	El tron usuari es destruït al xocar contra un tron oponent. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
6	El tron usuari aconsegueix destruir a tots els trons oponents. S'imprimeix la frase 'Ha guanyat l'usuari!'.	Sí.	
7	El tron usuari aconsegueix destruir a tots els trons oponents però xoca mentre s'està esborrant el rastre de l'últim tron oponent i,	Sí.	

	per tant, també és destruït. S'imprimeix la frase 'S'ha produït un empat!'.		
8	El tron usuari xoca i és destruït, però mentre s'està esborrant el seu rastre, els trons oponents també son destruïts. S'imprimeix la frase 'S'ha produït un empat!'.	Sí.	
9	No es produeix cap error de sincronisme amb el nombre de trons oponents màxim que es pot crear (9 oponents).	Sí.	
10	El comptatge del temps es realitza de manera correcta.	Sí.	

Fase 3

Prova	Descripció	Correcte?	Observacions
1	Es produeix l'execució concurrent dels processos i els *threads* amb el nombre de trons oponents màxim que es pot crear (9 oponents).	Sí.	
2	Aturada del joc utilitzant la tecla return. S'imprimeix la frase 'S'ha aturat el joc amb la tecla RETURN!'.	Sí.	
3	El tron usuari es destruit al xocar contra una paret. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
4	El tron usuari es destruït al xocar contra si mateix. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
5	El tron usuari es destruït al xocar contra un tron oponent. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
6	El tron usuari aconsegueix destruir a tots els trons oponents. S'imprimeix la frase 'Ha guanyat l'usuari!'.	Sí.	
7	El tron usuari aconsegueix destruir a tots els trons oponents però xoca mentre s'està esborrant el rastre de l'últim tron oponent i, per tant, també és destruït. S'imprimeix la frase 'S'ha produït un empat!'.	Sí.	
8	El tron usuari xoca i és destruït, però mentre s'està esborrant el	Sí.	

	seu rastre, els trons oponents també son destruïts. S'imprimeix la frase 'S'ha produït un empat!'.		
9	El comptatge del temps es realitza de manera correcta.	Sí.	Degut als errors de concurrència produïts per l'accés dels diferents processos i *threads* al taulell de joc, en ocasions el temps de joc no es visualitza correctament
10	Els trons es desplacen amb una direcció inicial aleatòria i diferent en cada execució del programa.	Sí.	

Fase 4

Prova	Descripció	Correcte?	Observacions
1	No es produeix cap error de sincronisme amb el nombre de trons oponents màxim que es pot crear (9 oponents).	Sí.	
2	Es produeix l'execució concurrent dels processos i *threads* amb el nombre de trons oponents màxim que es pot crear (9 oponents).	Sí.	
3	Aturada del joc utilitzant la tecla return. S'imprimeix la frase 'S'ha aturat el joc amb la tecla RETURN!'.	Sí.	
4	El tron usuari es destruit al xocar contra una paret. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
5	El tron usuari es destruït al xocar contra si mateix. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
6	El tron usuari es destruït al xocar contra un tron oponent. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
7	El tron usuari aconsegueix destruir a tots els trons oponents. S'imprimeix la frase 'Ha guanyat l'usuari!'.	Sí.	
8	El tron usuari aconsegueix destruir a tots els trons oponents però xoca mentre s'està esborrant el rastre de l'últim tron oponent i, per tant, també és destruït. S'imprimeix la frase 'S'ha produït un empat!'.	Sí.	
9	El tron usuari xoca i és destruït, però mentre s'està esborrant el seu rastre, els trons	Sí.	

	oponents també son destruïts. S'imprimeix la frase 'S'ha produït un empat!'.		
10	El comptatge del temps es realitza de manera correcta.	Sí.	
11	Els trons es desplacen amb una direcció inicial aleatòria i diferent en cada execució del programa.	Sí.	
12	Si el tron usuari aconsegueix destruir a tots els trons oponents però xoca mentre s'està esborrant el rastre de l'últim o últims trons oponents, l'esborrat del rastre del tron usuari es continua mostrant fins al final encara que s'hagi finalitzat l'esborrat del rastre dels trons oponents.	Sí.	

Fase 5

Prova	Descripció	Correcte?	Observacions
1	Si el tron usuari xoca contra el rastre invers d' un tron oponent, el tron usuari es destruït i el tron oponent s'escriu en estat no invers durant 10 segons.	Sí.	Si l'esborrat del rastre del tron usuari s'acaba abans de que hagin passat 10 segons no es tornarà a mostrar el tron oponent en estat invers perquè s'haurà acabat el joc.
2	Si un tron oponent xoca contra el rastre invers del tron usuari, el tron oponent canvia de direcció i el tron usuari s'escriu en estat no invers durant 10 segons.	Sí.	
3	Si un tron oponent xoca contra el rastre invers de un altre tron oponent, el primer tron oponent canvia de direcció i el segon tron oponent s'escriu en estat no invers durant 10 segons.	Sí.	
4	Si el tron usuari xoca contra el rastre no invers de un tron oponent, el tron oponent es destrueix i el tron usuari continua el seu recorregut en l'estat en que estava.	Sí.	
5	Si un tron oponent xoca contra el rastre no invers del tron usuari, el tron usuari es destrueix i el tron oponent continua el seu	Sí.	

	recorregut en l'estat en que estava.		
6	Si un tron oponent xoca contra el rastre no invers de un altre tron oponent, el segon tron oponent es destrueix i el primer tron oponent continua el seu recorregut en l'estat en que estava.	Sí.	
7	Si un tron oponent xoca contra si mateix, canvia de direcció però es continuarà escrivint en estat invers.	Sí.	
8	Es produeix l'execució concurrent dels *threads* amb el nombre de trons oponents màxim que es pot crear.	Sí.	
9	Aturada del joc utilitzant la tecla return. S'imprimeix la frase 'S'ha aturat el joc amb la tecla RETURN!'.	Sí.	
10	El tron usuari es destruït al xocar contra una paret. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
11	El tron usuari es destruït al xocar contra si mateix. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
12	El tron usuari es destruït al xocar contra un tron oponent. S'imprimeix la frase 'Ha guanyat l'ordinador!'.	Sí.	
13	El tron usuari aconsegueix destruir a tots els trons oponents. S'imprimeix la frase 'Ha guanyat l'usuari!'.	Sí.	
14	El tron usuari aconsegueix destruir a tots els trons oponents però xoca mentre s'està esborrant el rastre de l'últim tron oponent i, per tant, també és destruït. S'imprimeix la frase 'S'ha produït un empat!'.	Sí.	
15	El tron usuari xoca i és destruït, però mentre s'està esborrant el seu rastre, els trons oponents també son destruïts. S'imprimeix la frase 'S'ha produït un empat!'.	Sí.	

16	No es produeix cap error de sincronisme amb el nombre de trons oponents màxim que es pot crear (9 oponents).	Sí.	
17	Els trons es desplacen amb una direcció inicial aleatòria i diferent en cada execució del programa	Sí.	
18	Si el tron usuari aconsegueix destruir a tots els trons oponents però xoca mentre s'està esborrant el rastre de l'últim o últims trons oponents, l'esborrat del rastre del tron usuari es continua mostrant fins al final encara que s'hagi finalitzat l'esborrat del rastre dels trons oponents.	Sí.	

Fonts documentals

- https://campusvirtual.urv.cat/pluginfile.php/3127939/mod_resource/content/2/labs/Lab5-Threads.pdf
- https://campusvirtual.urv.cat/pluginfile.php/3127948/mod_resource/content/1/labs/Lab6-SincroThreads.pdf
- https://campusvirtual.urv.cat/pluginfile.php/3127954/mod_resource/content/1/labs/Lab7-Procs.pdf
- https://campusvirtual.urv.cat/pluginfile.php/3127958/mod_resource/content/0/labs/Lab8-SincroProcs.pdf
- http://www.chuidiang.org/clinux/funciones/rand.php