Pràctica 2.1

Objectiu

L'objectiu d'aquesta pràctica és aprendre conceptes bàsics relacionats amb la programació amb múltiples fils d'execució (*multithreading*).

Grups de pràctiques

Els grups de pràctiques seran de dos o tres alumnes.

Entorn de treball

Les implementacions s'han de poder executar sobre un ordinador del laboratori i/o un servidor *bsd*.

Lliuraments

La practica2 tindrà dos lliuraments. La primera part és la que us donem en aquest document. Un alumn@ del grup haurà de lliurar el fitxer font a través de l'aplicació MOODLE. Al segon lliurament s'haurà de posar-hi la documentació i tots els fitxers. La data límit del lliuraments està fixada en el moodle.

Correcció

La correcció de la pràctica 2.1 es realitzarà, en primera convocatòria, durant la sessió del laboratori 7, mitjançant una entrevista amb un professor i tots els membres del grup de pràctiques. L'entrevista tindrà una durada aproximada de 10 minuts. En una segona convocatòria l'entrevista es farà conjuntament amb la pràctica 2.2, i tindrà una durada major. Les dates d'entrevista s'especificaran a la pàgina Moodle de l'assignatura, així com la manera de concertar hora. A més, **es realitzarà una verificació automàtica de còpies entre tots els codis presentats**. Si es detecta alguna còpia, tots els alumnes involucrats (els qui han copiat i els que s'han deixat copiar) tindran la pràctica suspesa i, per extensió, l'assignatura suspesa.

El dia de l'entrevista cal portar un informe escrit on es documenti la pràctica de la manera habitual, és a dir: *Especificacions, Disseny, Implementació* i *Joc de proves*. En l'apartat de joc de proves NO s'han d'incloure "pantallazos" del programa, ja que la impressió sobre paper dels resultats no demostra que la implementació presentada realment funcioni. El que cal fer és enumerar (redactar) tots els casos possibles que el programa és capaç de tractar. Les proves presentades i altres afegides pel professor, s'executaran al bsd o a un ordinador del deim5, durant l'entrevista. El professor realitzarà preguntes sobre el contingut de l'informe i el funcionament dels programes presentats. Cada membre del grup tindrà una nota individual, que dependrà del nivell de coneixements demostrat durant l'entrevista.

Enunciat

Es tracta d'implementar una versió bàsica del joc **Arkanoid** o Breakout. Es genera un camp de joc rectangular amb una porteria/sortida, una paleta que es mou amb el teclat per a cobrir la sortida i una pilota que rebotarà contra les parets del camp, contra la paleta i contra alguns blocs. Hi ha 3 tipus de blocs: uns blocs que es trenquen i desapareixen (A), uns blocs que es trenquen, desapareixen i creen una nova pilota (B) i uns blocs que no es trenquen i simplement fan rebotar la pilota. En el cas de tenir varies pilotes, al xocar entre elles hauran de rebotar. Quan una de les pilotes surti per la porteria, desapareixerà.
El programa acabarà la seva execució quan no hi hagi cap pilota al camp, quan no quedin blocs dels que es trenquen o quan l'usuari premi la tecla RETURN. El joc es guanya si s'han trencat tots els blocs trencables, i hi ha com a mínim una pilota al camp de joc.

En el moodle us proporcionem una versió inicial del programa '`mur0.c`' on de manera seqüència es controla una pilota i la paleta d'usuari. Per tal d'aprendre programació *multithreading*, es demana modificar aquesta versió inicial de forma que l'ordinador pugui moure més d'una pilota simultàniament. La idea principal és que les funcions que controlen el moviment de la pilota i de la paleta s'adaptin per a ser executar com un fil independent o *thread*. Així, en l'evolució del joc només caldrà crear tants fils d'execució com pilotes s'hagin de crear en trencar-se els blocs fins a un màxim de 9 pilotes.

Fases de la pràctica

Per tal de facilitar el disseny de la pràctica es proposen 4 fases:

0. Programa seqüencial (`'mur0.c'`):

Versió seqüencial amb una sola pilota i una paleta. Aquesta versió us la proporcionen en el laboratori 5.

1. Creació de threads (`'mur1.c'`):

Partint de l'exemple anterior, modifiqueu les funcions que controlen el moviment de la paleta i la pilota per a què puguin actuar com a fils d'execució independents. Així el programa principal crearà un fil pel jugador (paleta) i un fil per a la primera pilota. Després s'afegirà l'aparició de noves pilotes quan una destrueixi algun dels blocs (B). Aquesta fase segur que tindrà deficiències que es solucionaran a la següent.

2. Sincronització de threads (`'mur2.c'`):

Completeu la fase anterior de manera que es sincronitzi l'execució dels fils a l'hora d'accedir als recursos compartits (per exemple, l'entorn de dibuix,

rutines de les curses, variables globals necessàries). D'aquesta manera s'han de solucionar els problemes de visualització que presenta la versió del mur1, els quals es fan més evidents quan treballem en un multiprocessador o en un servidor.

Fase 0

Per tal de generar la visualització del joc es proporcionen una sèrie de rutines dins del fitxer 'winsuport.c', que utilitzen una llibreria d'UNIX anomenada '*curses*'. Aquesta llibreria permet escriure caràcters ASCII en posicions específiques de la pantalla (fila, columna). El fitxer 'winsuport.h' inclou les definicions i les capçaleres de les rutines implementades, així com la descripció del seu funcionament:

```
int  win_ini(int *fil, int *col, char creq, unsigned int inv);
void win_fi();
void win_escricar(int f, int c, char car, unsigned int invers);
char win_quincar(int f, int c);
void win_escristr(char *str);
int  win_gettec(void);
void win_retard(int ms);
```

- Cal invocar la funció `win_ini` abans d'utilitzar qualsevol de les altres funcions, indicant el número de files i columnes requerits de la finestra de dibuix, així com el caràcter per emmarcar el camp de joc i si aquest aquest caràcter s'ha de visualitzar en atribut invers (definicions `INVERS` o `NOINV`). Els paràmetres de files i columnes es passen per referència perquè, si al invocar a la funció valen zero, la finestra ocuparà tota la pantalla del terminal i les variables referenciades contindran el número de files i columnes obtingut per la funció d'inicialització. L'última fila de la finestra de dibuix no formarà part del camp de joc, ja que es reserva per escriure missatges.
- En acabar el programa, cal invocar la funció `win_fi`.
- Per escriure un caràcter en una fila i columna determinades cal utilitzar la funció `win_escricar`. El paràmetre `invers` permet ressaltar el caràcter amb els atributs de vídeo invertits (definicions `INVERS` o `NOINV`). El codi ASCII servirà per identificar el tipus d'element del camp de joc (espai lliure `' '`, paret `'+'`, paleta `'0'`, pilota `'1'`).
- La funció `win_quincar` permet consultar el codi ASCII d'una posició determinada.
- La funció `win_escristr` serveix per escriure un missatge en l'última línia de la finestra de dibuix.
- Per llegir una tecla es pot invocar la funció `win_gettec`. Es recomana utilitzar les tecles definides en el fitxer de capçalera 'winsuport.h' per especificar el moviment del jugador.
- La funció `win_retard` permet aturar l'execució d'un procés o d'un fil durant el número de mil·lisegons especificat per paràmetre.

El programa d'exemple 'mur0.c' utilitza les funcions anteriors per dibuixar el camp de joc i per controlar el moviment de la paleta i de la pilota. Per a generar l'executable primer cal compilar els fitxers fonts i ensamblar-los:

```
$     make mur0
```

Ara ja es pot executar el programa. Abans però, s'han de conèixer els arguments que cal passar-li per línia de comandes. El primer argument serveix per indicar un fitxer de text que contindrà la configuració de la zona de joc i de la posició i velocitat inicials de la pilota. L'estructura del fitxer serà la següent:

```
num_files  num_columnes  mida_porteria
pos_fila pos_columna vel_fila vel_columna
```

on la primera línia contindrà números enters i la segona números reals. Per exemple, el fitxer `'prova1.txt'` conté la següent informació:

```
$     more params.txt
      25 40 20
      21.0 20.0 -1.0 0.3
```

El fitxer indica una zona de dibuix de 25 files i 40 columnes[1] (camp de joc de 24x40) i una sortida de 20 espais. La posició inicial de la pilota serà (21, 20) i la velocitat inicial serà de -1.0 punts per files (cap amunt) i 0.3 punts per columna (a la dreta).

A més, es podrà afegir un segon argument opcional per a indicar un retard general del moviment dels objectes (en ms). El valor per defecte d'aquest paràmetre es de 100 (1 dècima de segon).

El nom de l'executable serà `'mur0'`, i la manera usual d'invocar-lo és la següent:

```
$     ./mur0 fit_config [retard]
```

Així, la comanda per executar el programa amb la configuració del fitxer `'params.txt'` i 50 mil·lisegons de retard de moviment és la següent:

```
$     ./mur0 params.txt 50
```

Finalment, si s'executa sense arguments es mostra una ajuda sobre el funcionament del programa.

1 Redimensioneu el terminal amb prou files i columnes.

Fase 1

En aquesta fase cal modificar les funcions anteriors de moviment de la paleta i de la pilota per tal que puguin funcionar com a fils d'execució independents. Les funcions bàsiques de creació (i unió) de *threads* s'expliquen a la sessió 5 dels laboratoris d'FSO. A més, cal fer una sèrie de canvis addicionals per a què tot funcioni correctament. A continuació es proposen una sèrie de passos que divideixen la feina a fer en petites tasques. Abans de fer cap modificació, es suggereix fer una còpia del programa original (i canviar els comentaris de la capçalera):

```
$     cp mur0.c mur1.c
```

Pas 1.1: Primera lectura del programa

Abans que res, cal donar-li una ullada general a la versió inicial del programa `'mur0.c'`, per tal d'entendre la seva estructura.

Primer cal observar les variables globals més importants, és a dir:

```
int n_fil, n_col;       /* numero de files i columnes del taulell */
int m_por;              /* mida de la porteria (en caracters) */
int f_pal, c_pal;       /* posicio del primer caracter de la paleta */
int f_pil, c_pil;       /* posicio de la pilota, en valor enter */
float pos_f, pos_c;     /* posicio de la pilota, en valor real */
float vel_f, vel_c;     /* velocitat de la pilota, en valor real */
```

Després observem que el programa està organitzat en les següents funcions:

```
int carrega_configuracio(FILE *fit);
int inicialitza_joc(void);
int mou_pilota(void);
int mou_paleta(void);
```

Finalment, ens fixarem en l'estructura bàsica del programa principal, el qual utilitza les funcions anteriors per inicialitzar la finestra de dibuix i, si tot ha funcionat bé, executeu el bucle principal, el qual activa periòdicament les funcions per moure paleta i moure pilota. El programa acaba quan s'ha premut la tecla `RETURN` o quan la pilota surt per la porteria (`'fi1'` o `'fi2'` diferent de zero):

```
int main(int n_args, char *ll_args[])
{
  :
  if (carrega_configuracio(fit_conf) !=0)/* llegir dades del fitxer  */
     exit(3);     /* aborta si hi ha algun problema en el fitxer */
  :
  if (inicialitza_joc() !=0)  /* intenta crear el taulell de joc */
     exit(4);     /* aborta si hi ha algun problema amb taulell */
  do              /********** bucle principal del joc **********/
  {   fi1 = mou_paleta();
      fi2 = mou_pilota();
      win_retard(retard);           /* retard del joc */
  } while (!fi1 && !fi2);

  win_fi();                   /* tanca les curses */
}
```

Pas 1.2: Modificar la funció de moviment de la pilota

Com que hem d'adaptar la funció **mou_pilota** per a què es pugui executar com un fil d'execució independent, el primer que cal fer és canviar la capçalera:

```
void * mou_pilota(void * index)
```

El valor que es passa pel paràmetre `index` serà un enter que indicarà l'ordre de creació de les pilotes (0 -> primera, 1 -> segona, etc.). Aquest paràmetre servirà per accedir a la taula global d'informació de les pilotes, així com per escriure el caràcter corresponent (identificador '1' per la primera, '2' per la segona, etc.). De moment, però, no utilitzarem el paràmetre, ja que farem una primera versió de la funció per manegar una única pilota com un fil d'execució independent.

Al contrari que en la fase 0, la nova funció de moviment s'ha d'executar de manera independent al bucle principal del programa. Això implica que cal implementar el seu propi bucle per generar el moviment dels objectes. Per finalitzar l'execució d'aquest bucle de moviment, es pot consultar les variables `'fi1'` i `'fi2'`, que indiquen alguna condició de final de joc (tecla `RETURN` o pilota surt per porteria). Per tant, aquestes variables ara han de ser globals (no locals al `main`), i s'actualitzaran directament des de dins de les funcions de `mou_paleta` i `mou_pilota`.

Pas 1.3: Modificar la funció de moviment de l'usuari

Igual que en el cas anterior, s'ha d'adaptar la funció **mou_paleta** per a que es pugui executar com un fil d'execució independent. La nova capçalera serà la següent, on el paràmetre `nul` no conté cap informació:

```
void * mou_paleta(void * nul)
```

Una altra vegada, caldrà crear un bucle independent de moviment per a la paleta, amb les mateixes condicions d'acabament generals.

Pas 1.4: Modificar la funció principal del programa 

Després cal modificar la funció **main**, creant els dos fils corresponents a la primera pilota i el fil corresponent a la paleta. El seu bucle principal ara NO ha d'invocar a les funcions de moviment de paleta ni el de la pilota, donat que ja s'executen de manera concurrent, sinó que tan sols controlarà el temps de joc (`minuts:segons`) i el mostrarà per la línia de missatges, fins que es doni alguna condició de finalització. L'última acció del programa principal serà la d'escriure el temps total de joc per la sortida estàndard, juntament amb els missatges de finalització proposats en la versió d'exemple.

Arribats a aquest punt ja podem provar els canvis efectuats, encara que de moment només es mourà una sola pilota. Genereu el fitxer executable `'mur1'`:

```
$     make mur1
```

ⓘ *ATENCIÓ: podria ser que la pilota es mogués molt ràpid i no la vegessiu.*

Pas 1.5: Acabar la funció de moviment de les pilotes.

Modifiqueu la funció `comprovar_bloc` per a que si la pilota que es mou toca un bloc del tipus (B) es crei una nova pilota utilitzant un *thread,* al qual se li passarà per paràmetre l'identificador de la nova pilota.

Ara la funció **`mou_pilota`** ha de poder gestionar les dades de la pilota que se li assignarà a través del paràmetre `index`. Per això cada pilota haurà de disposar de la seva pròpia informació. Així doncs, s'han de convertir les variables globals `'f_pil'`, `'c_pil'`, `'pos_f'`, `'pos_c'`, `'vel_f'` i `'vel_c'` en vectors de màxim 9 posicions.

ⓘ *ATENCIÓ: la implementació de la fase 1 no realitza cap mena de sincronisme entre els fils. Per tant, és possible que apareguin caràcters erronis ocasionals a causa de l'execució concurrent i descontrolada d'aquests fils. Això no obstant, el propòsit d'aquesta fase és veure que s'executen tots els fils requerits en el fitxer de configuració especificat com a argument del programa.*

Fase 2.

En aquesta fase cal establir seccions crítiques que evitin els problemes de concurrència de la fase anterior. Per fer això caldrà incloure semàfors per fer l'exclusió mútua entre els fils. Les funcions bàsiques de manipulació de semàfors per sincronitzar els *threads* s'expliquen a la sessió 6 dels laboratoris de FSO.

Les seccions crítiques han de servir per impedir l'accés concurrent a determinats recursos per part dels diferents fils d'execució. En el nostre cas, cal establir seccions crítiques per accedir a l'entorn de dibuix i a algunes variables global que s'hagués de protegir contra els accessos concurrents desincronitzats.

El fitxer executable s'anomenarà `'mur2'`.

Per tal d'agilitzar el procés de desenvolupament de la pràctica 2.1 (i també de la 2.2), es recomana la utilització d'un fitxer de text anomenat `'Makefile'`, el qual ha d'estar ubicat en el mateix directori que els fitxers font a compilar. Per permetre la compilació de la nova fase, es poden afegir les següents línies:

```
mur2 : mur2.c winsuport.o winsuport.h
      gcc -Wall mur2.c winsuport.o -o mur2 -lcurses -lpthread
```

D'aquesta manera s'estableixen les dependències entre els fitxers de sortida (en aquest cas `'`**`mur2`**`'`) i els fitxers d'entrada dels quals depenen, en aquest cas: ***`mur2.c`***, ***`winsuport.o`*** i ***`winsuport.h`***, a més d'especificar la comanda que s'ha d'invocar en cas que la data de modificació d'algun dels fitxers d'entrada sigui posterior a la data de l'última modificació del fitxer de sortida. Aquest control el realitza la comanda **`make`**. Per exemple, podem realitzar les següents peticions:

```
$     make winsuport.o
$     make mur2
$     make
```

Si no s'especifica cap paràmetre, `make` verificarà les dependències del primer fitxer de sortida que trobi dins del fitxer `'Makefile'` del directori de treball actual.