Pràctica 2.1

Objectiu.

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

Grups de pràctiques.

Els grups de pràctiques seran de dos o tres alumnes.

Entorn de treball.

Les implementacions s'han d'executar sobre el servidor *bsd* i en la distribució Linux dels laboratoris de l'assignatura.

Lliuraments.

Es faran dos lliuraments de la pràctica al moodle. El primer correspondrà a les fases 1 i 2, i el segon a les fases 3 i 4. Heu de seguir estrictament totes les instruccions donades especificades a les tasques del lliurament del moodle. La data límit de lliurament en primera convocatòria és la mateixa per a tots els grups de laboratori (veure l'entrada Distribució de classes dins l'espai Moodle de l'assignatura).

Correcció.

La revisió i correcció de la pràctica 2 es realitzarà mitjançant una entrevista entre el professor i tots els membres del grup de pràctiques. L'entrevista final tindrà una durada aproximada de 20 minuts. Les dates d'entrevista s'especificaran a l'espai moodle de l'assignatura, així com la manera de concertar hora.

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 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 sobre un ordinador com els dels laboratoris durant l'entrevista.

El professor realitzarà preguntes sobre el contingut de l'informe, del codi que presenteu i del funcionament dels programes. Cada membre del grup tindrà una nota individual, que dependrà del nivell de coneixements demostrat durant l'entrevista.

A més, **es realitzarà una verificació automàtica de copies entre tots els programes 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.

Enunciat.

Es tracta d'implementar el joc del frontó amb múltiples pilotes; cal generar un camp de joc rectangular amb una porteria, una paleta que s'ha de moure amb el teclat per a cobrir la porteria, i vàries pilotes que rebotaran contra les parets del camp, contra la paleta i també entre elles mateixes. Quan una de les pilotes surti per la porteria, el programa acabarà la seva execució i mostrarà el temps total que s'han mantingut totes les pilotes en moviment. El joc consisteix en aguantar totes les pilotes el màxim temps possible.

En el moodle es proporciona una versió inicial del programa '**fronton0.c**' on només es controla una pilota. Per tal d'aprendre programació *multithreading*, es proposa modificar aquesta versió inicial de forma que l'ordinador pugui moure més d'una pilota simultàniament. La idea principal és que la funció que controla el moviment de la pilota s'adapti per a ser executar com un fil independent o *thread*. Així, des del programa principal només caldrà crear tants fils d'execució com pilotes es vulguin (fins a 9).

Fases.

Per tal de facilitar el disseny de la pràctica es proposen 5 fases:

0. Programa seqüencial (`'fronton0.c'`):

> Es tracta d'implementar el joc bàsic, és a dir, una única pilota. Aquesta versió es proporciona com a exemple de programa escrit en llenguatge C, dins el paquet "`Material.tgz`" que hi ha a l'espai moodle de l'assignatura.

1. Creació de threads (`'fronton1.c'`):

> Partint de l'exemple anterior, modificar les funcions que controlen el moviment de la pilota i de la paleta per a que puguin actuar com a fils d'execució independents. Així el programa principal ha de crear un fil per a la paleta de l'usuari i varis fils per a les pilotes.

2. Sincronització de threads (`'fronton2.c'`):

> Completar la fase anterior de manera que l'execució dels fils a l'hora d'accedir als recursos compartits (entorn de dibuix, variables globals, etc.) es sincronitzi. D'aquesta manera s'han d'evitar els problemes d'accés concurrent al mateix recurs per part de diversos fils d'execució. També afegirem una finalització del joc després d'un nombre concret de rebots de les pilotes amb la paleta.

3. Creació de processos (`'fronton3.c'`/`'pilota3.c'`):

> Partint del codi de la fase anterior, realitzar les modificacions necessàries per convertir els *threads* que controlen les pilotes en processos independents. El control de la paleta de l'usuari continuarà com a *threads* del procés pare. Així, el programa

principal crearà un *procés* per a cada pilota, i un *thread* per a la paleta. En aquesta fase no es demana cap mena de sincronització ni comunicació entre els processos que intervenen en el joc (la sincronització entre *threads* no funciona entre processos).

4. Sincronització de processos (`fronton4.c' /pilota4.c'`):

Completar la fase anterior de manera que l'execució dels processos es sincronitzi a l'hora d'accedir als recursos compartits (per exemple, l'entorn de dibuix). D'aquesta manera s'han de solucionar els problemes de visualització que presenta la fase anterior, els quals es fan patents quan s'intenta executar el programa sobre el servidor remot *bsd*. En aquesta fase també s'ha d'implementar la interacció entre en rebot de les pilotes.

Fase 0.

Dins del directori "`exemples_P2`" estan disponibles (entre d'altres) els següents fitxers:

`prova0.txt, prova1.txt, prova2.txt, prova3.txt, fronton0.c, winsuport.c i winsuport.h`

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 n_fil, int n_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 de la finestra de dibuix, així com el caràcter per emmarcar el camp de joc. 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 `'+'`, pilota `'.'`, paleta usuari `'0'`, primera paleta ordinador `'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`, utilitzant les tecles definides en el fitxer de capçalera '`winsuport.h`' per especificar el moviment de la paleta de l'usuari.
- Finalment, la rutina `win_retard` permet aturar l'execució d'un procés o d'un fil d'execució durant el número de mil·lisegons especificat per paràmetre.

El programa d'exemple `fronton0.c'` utilitza les funcions anteriors per dibuixar el camp de joc i controlar el moviment de les paletes i de la pilota. Per generar l'executable primer cal compilar el fitxer '`winsuport.c`', per tal d'obtenir un fitxer objecte '`winsuport.o`' que contindrà les instruccions en codi màquina de les rutines de dibuix:

```
$  gcc –c winsuport.c –o winsuport.o
```

La comanda anterior només cal invocar-la una vegada per obtenir el fitxer '`winsuport.o`' corresponent a la màquina on s'han d'executar les rutines de dibuix. Després ja podem generar el fitxer executable '`fronton0`' corresponent al joc del tennis bàsic. La següent comanda s'encarrega de fer això a partir del fitxer font '`fronton0.c`', el fitxer objecte anterior i de la llibreria '`curses`':

```
$   gcc fronton0.c winsuport.o –o fronton0 –lcurses
```

Ara ja es pot executar el programa. Abans però, s'han de conèixer els arguments que cal passar-li. 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 '`camp1.txt`' conté la següent informació::

```
$   cat prova1.txt
            20 70 15
            10.0 35.5 0.4 –1.0
```

El fitxer indica una zona de dibuix de 20 files i 70 columnes (camp de joc de 19x70) i una porteria de 15 espais. La posició inicial de la pilota serà (10, 35.5) i la velocitat inicial serà de 0.4 punts per files i –1.0 punts per columna. Consultar l'annex d'aquest document per entendre què signifiquen els valors de posició i velocitat de la pilota.

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 `fronton0'` , i la manera usual d'invocar-lo és la següent:

```
$   ./fronton0 fit_config [retard]
```

Així, la comanda per executar el programa amb la configuració del fitxer '`prova1.txt`' i 50 mil·lisegons de retard de moviment és la següent:

```
$   ./fronton0 prova1.txt 50
```

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

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 L5 dels laboratoris. 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 copia del programa original (i canviar els comentaris de la capçalera):

```
$ cp fronton0.c fronton1.c
```

Pas 1.1: Primera lectura del programa

Abans que res, cal donar-li una ullada general a la versió inicial del programa `fronton0.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, executar 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 ('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 fils d'execució pertinents un cop s'hagi inicialitzat correctament l'entorn del joc. 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 ha d'executar una tasca independent fins que es doni alguna condició de finalització. De moment, però, el bucle principal no farà res; simplement esperarà a què tots els fils creats acabin la seva execució, per després destruir l'entorn de dibuix.

Arribats a aquest punt ja podem provar els canvis efectuats, encara que de moment només es mourà una sola pilota. El fitxer executable s'anomenarà `fronton1'`. Per a generar-lo, s'ha d'invocar la següent comanda, la qual crida al compilador de C passant-li la referència de la llibreria que conté el codi de les funcions per treballar amb multithreading (`pthread'`):

```
$ gcc -Wall fronton1.c winsuport.o -o fronton1 -lcurses -lpthread
```

Pas 1.5: Modificar les variables globals

Per controlar les múltiples pilotes, s'ha de convertir les variables globals `f_pil'` , `c_pil'` , `pos_f'` , `pos_c'` , `vel_f'` i `vel_c'` en vectors de màxim 9 posicions (es recomana fer servir una constant per fixar aquest límit). Així, cada pilota podrà disposar de la seva pròpia informació.

Pas 1.6: Modificar la càrrega de la configuració del joc

Després cal modificar la funció **`carrega_configuracio`** per a què pugui llegir més d'una línia de posició i velocitat de cada pilota. És a dir, el fitxer de configuració del joc admetrà la definició de vàries pilotes (màxim 9). La seva estructura serà la següent:

```
num_files  num_columnes  mida_porteria
pos1_fila pos1_columna vel1_fila vel1_columna
pos2_fila pos2_columna vel2_fila vel2_columna
etc…
```

on la primera línia contindrà les mides del tauler de joc (valors enters) i la resta de línies contindran la posició i velocitats inicials de cada pilota (valors reals). Cal crear un variable global `n_pil` que contindrà el número total de pilotes en joc.

Pas 1.7: Modificar la inicialització del joc

Després cal modificar la funció **`inicialitza_joc`** per a què col·loqui totes les pilotes a les posicions inicials a més de mostrar-les per primera vegada en pantalla.

Pas 1.8: Acabar la funció de moviment de les pilotes.

Un cop ja tenim carregats els vectors amb la informació de cada pilota, cal completar la funció **`mou_pilota`** per a què pugui gestionar les dades de la pilota que se li assignarà a través del paràmetre `index`. Vegeu l'annex d'aquest document per entendre com funciona l'algorisme de moviment de les pilotes.

Pas 1.9: Acabar la funció principal del programa.

Finalment, cal completar la funció **`main`** per arrancar els múltiples fils corresponents a totes les pilotes, a més del fil corresponent a la paleta. A més, cal modificar el bucle principal del programa per tal que realitzi una tasca independent, que en el nostre cas consistirà en controlar el temps de joc (`minuts:segons`) mostrant-lo per la línia de missatges i també afegirem una finalització del joc després d'un nombre concret de rebots de les pilotes amb la paleta. 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 que especifiquen el perquè ha acabat el joc.

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 problemes de concurrència entre els múltiples fils d'execució. Per fer això caldrà incloure semàfors per tal d'establir seccions crítiques en l'accés a l'entorn de dibuix i algunes variables globals compartides. Les funcions bàsiques de manipulació de semàfors per sincronitzar els *threads* s'expliquen a la sessió L6 dels laboratoris.

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 alguna variable global que s'hagués de protegir contra els accessos concurrents desincronitzats.

També es vol afegir el comptatge total dels rebots de les pilotes amb la paleta d'usuari. Quan el número total de rebots superi un cert valor màxim, el programa ha d'acabar la seva execució. En la línia de missatges del taulell de joc s'ha de visualitzar el número de rebots realitzats i el nombre de reboots que falten per a què s'acabi el joc.

El fitxer executable s'anomenarà '`fronton2`'. El número màxim de rebots es passarà com a segon argument de línia de comandes (després del nom de fitxer de configuració del taulell). El tercer argument, opcional, serà el valor del retard (si no s'especifica, retard=100). Per exemple, per carregar el fitxer '`prova3.txt`' amb un màxim de 100 moviments i un retard de 50 mil·lisegons, hauriem d'invocar el programa de la següent forma:

```
$   ./fronton2 prova3.txt 100 50
```

Per validar el seu funcionament, caldrà executar-lo sobre el *bsd*. Donat que el retard de xarxa evidencia més els problemes de sincronisme.

Per tal d'agilitzar el procés de desenvolupament de la pràctica 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 linies:

```
fronton2 : fronton2.c winsuport.o winsuport.h
      gcc fronton2.c winsuport.o –o fronto2 –lcurses –lpthread
winsuport.o : winsuport.c winsuport.h
      gcc –c winsuport.c –o winsuport.o
```

D'aquesta manera s'estableixen les dependències entre els fitxers de sortida (en aquest cas '`fronton2`' i '`winsuport.o`') i els fitxers font dels quals depenen, a més d'especificar la comanda que s'ha d'invocar en cas que la data de modificació d'algun dels fitxers font 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 fronton2
$ 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.

Fase 3.

L'objectiu d'aquesta fase és aprendre conceptes bàsics relacionats amb la programació amb múltiples processos (multiprocés), per tant, es proposa modificar el joc desenvolupat a la fase 2 de forma que ara s'utilitzin processos en lloc de *threads*. Concretament, cada pilota serà controlada per un procés fill. El codi d'aquests processos fill haurà de residir en un fitxer executable diferent del fitxer executable del programa principal (també anomenat programa pare).

Les funcions bàsiques de creació de processos, més la gestió de zones de memòria compartida, s'expliquen a la sessió L7 dels laboratoris.

A més del fitxer font principal `fronton3.c'` (procés pare), caldrà crear un altre fitxer font `pilota3.c'` que contindrà el codi que han d'executar els processos fill per controlar les pilotes.

Per accedir a pantalla es disposa d'un nou grup rutines definides a `winsuport2.h'`. Aquestes rutines estan adaptades a l'ús d'una mateixa finestra des de diferents processos. El seu funcionament és diferent de l'anterior versió de `winsuport` per a *threads*. Es disposa d'un exemple d'utilització als fitxers `multiproc2.c'` i '`mp_car2.c'`.

A grans trets, la utilització de `winsuport2` ha de ser el següent:

- **Procés pare**:
  - invoca a `win_ini()`: la nova implementació de la funció retorna la mida en bytes que cal reservar per emmagatzemar el contingut del camp de joc.
  - crea una zona de memòria compartida de mida igual a la retornada per `win_ini()`.
  - invoca a `win_set()`: aquesta funció inicialitza el contingut de la finestra de dibuix i permet l'accés al procés pare.
  - crea els processos fill passant-los com argument la referència (identificador IPC) de la memòria compartida del camp de joc, a més de les dimensions (files, columnes) i altres informacions necessàries.
  - executa un bucle principal on, periòdicament (p. ex. cada 100 mil·lisegons) actualitza per pantalla el contingut del camp de joc, invocant la funció `win_update()`.
  - un cop acabada l'execució del programa (inclosos tots els processos), invoca la funció `win_fi()` i destrueix la zona de memòria del camp de joc.
- **Processos fill**:
  - mapejen la referència de memòria compartida que conté el camp de joc, utilitzant la referència (identificador) que s'ha passat per argument des del procés pare.

- invoquen la funció `win_set()` amb l'adreça de la zona de memòria mapejada anteriorment i les dimensions (files, columnes) que s'han passat per argument.
- utilitzen totes les funcions d'escriptura i consulta del camp de joc `win_escricar()`, `win_escristr()` i `win_quincar()`.

Els processos fill, però, no poden fer servir la funció `win_gettec()`, la qual només és accessible al procés que ha inicialitzat l'entorn de CURSES (el procés pare).

El fitxer executable principal s'anomenarà `fronton3'` i el de les paletes d'ordinador s'anomenarà `pilota3'`. Per generar-los, s'han d'utilitzar les següents comandes:

```
$ gcc fronton3.c winsuport2.o –o fronton3 –lcurses –lpthread
$ gcc fronton3.c winsuport2.o –o fronton3 –lcurses
```

L'execució es farà de manera similar a la pràctica anterior, invocant l'executable `fronton3'` amb els paràmetres corresponents al fitxer de configuració del joc (una fila per les dimensions més una fila per la paleta més una fila per cada pilota), el nombre màxim de rebots de les pilotes i un valor opcional de retard.

Fase 4.

En aquesta fase cal establir seccions crítiques que evitin els problemes de concurrència de la fase anterior. També cal implementar la funcionalitat de comunicació entre els diversos processos fill (i potser també amb el procés pare), per practicar amb el mecanisme de comunicació entre processos. Per establir seccions crítiques cal utilitzar semàfors. La comunicació entre processos es realitzarà mitjançant bústies de missatges. Les funcions bàsiques de manipulació de semàfors i de bústies per sincronitzar i comunicar els processos s'expliquen a la sessió 8 dels laboratoris de FSO.

En la comunicació entre processos, les pilotes que xoquin s'han d'intercanviar la seva velocitat en una o totes dues direccions. En resum, el comportament ha de ser el següent:

- si una pilota, al intentar avançar en una determinada direcció, detecta un obstacle que resulta ser una altra pilota, el procés que controla la primera pilota li enviarà un missatge al procés que controla la segona pilota indicant-li si el rebot és vertical, horitzontal o diagonal, juntament amb la velocitat vertical o horitzontal (o totes dues velocitats) de la primera pilota, segons la direcció de xoc,
- en rebre el missatge, el procés que controla la segona pilota ha de respondre indicant la seva velocitat vertical o horitzontal (o totes dues velocitats), segons la direcció de xoc,
- al final, les pilotes s'han d'intercanviar la velocitat corresponent a la direcció de xoc, és a dir, si per exemple la velocitat horitzontal de la pilota 1 era de 0.6 i la velocitat horitzontal de la pilota 2 era de -0.3, després d'un xoc horitzontal la velocitat de la pilota 1 serà de -0.3 i la velocitat de la pilota 2 serà de 0.6.

Els rebots contra la paret o contra la paleta tindran el mateix efecte que en la fase anterior, és a dir, simplement canviaran el signe de la velocitat de la pilota. Altres casos que cal preveure és el xoc "simultani" de més de dues pilotes (múltiples intercanvis de velocitat).

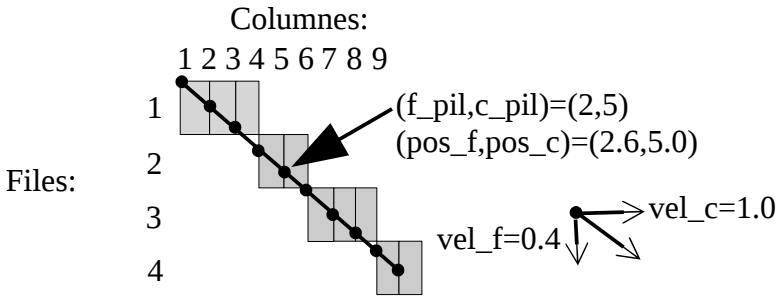
El fitxer executable principal s'anomenarà `fronton4'` i el que controla les pilotes `pilota4'` . Per validar el seu funcionament, caldrà executar-lo sobre l'entorn del servidor bsd. La utilització d'un fitxer `Makefile'` us facilitarà el desenvolupament de la pràctica.

Annex

L'algorisme proposat per controlar el moviment de la pilota manté la posició actual en dues variables globals de tipus real (`pos_f, pos_c`), i l'actualitza d'acord amb la velocitat actual representada per dues variables més de tipus real (`vel_f, vel_c`). Els rangs d'aquestes variables seran els següents:

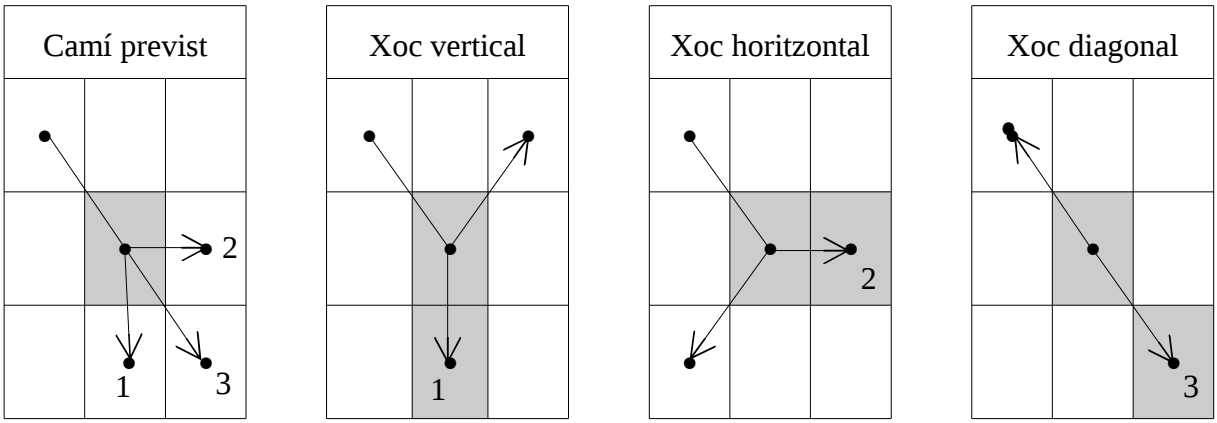
```
pos_f ∈ [1..nf-3]; pos_c ∈ [1..nc-2]; vel_f ∈ [-1..1]; vel_c ∈ [-1..1];
```

on `nf` és el número de files i `nc` és el número de columnes de la finestra de dibuix. Com que les coordenades de dibuix són enteres, cal memoritzar també els valors enters de la posició de la pilota (`f_pil, c_pil`). El gràfic següent mostra com evolucionaria una pilota amb velocitat de fila = 0.4 i velocitat de columna = 1.0:

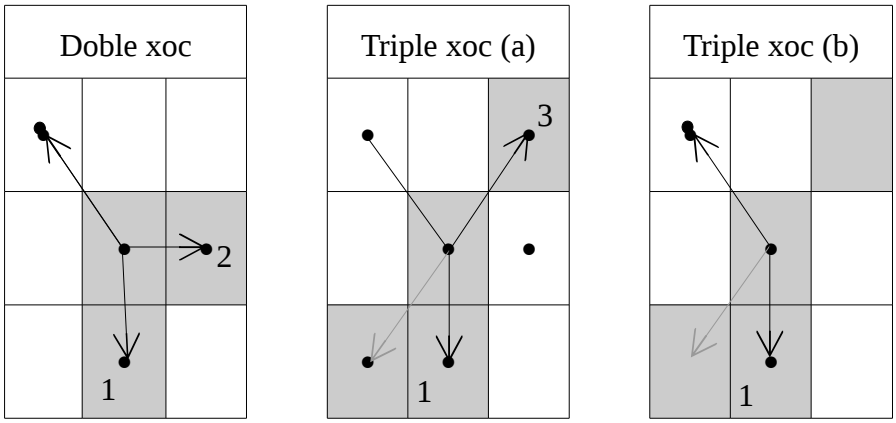


En el gràfic anterior, els punts simbolitzen la posició real, les línies gruixudes representen el moviment, i els rectangles són les posicions de pantalla que s'activarien. Per tant, en pantalla es veuria un bloc que anirà baixant una fila cada 3 i cada 2 columnes, alternativament. Tot i que el moviment en pantalla pot semblar una mica estrany, a nivell intern es controla la posició real amb molta precisió. Una pilota es dibuixarà com un dígit (del '1' al '9') invertit.

Per provocar un rebot quan hi ha un obstacle, l'algorisme calcula la nova posició hipotètica de la pilota, en coordenades enteres. Si aquesta és diferent de la posició entera actual, es passa a avaluar els possibles rebots. Això es realitza amb tres condicionals que avaluen el rebot vertical, horitzontal i diagonal, en aquest ordre. Si hi ha algun caràcter diferent de blanc en una posició de rebot, es canvia de signe la velocitat i s'actualitza la posició hipotètica. Els gràfics següents mostren exemples de desviaments bàsics de la pilota:



En els gràfics anteriors es representa l'ordre de consulta de cada direcció de rebot (1, 2 i 3) i el desviament provocat per un únic rebot en cada una d'aquestes direccions. Cal tenir en compte, però, que es pot donar més d'un rebot en un sol moviment. A continuació es mostren exemples de rebot sobre dos d'aquests casos:



En el cas del doble xoc, el primer rebot desvia la velocitat vertical (cap amunt), mentre que el segon rebot desvia la velocitat horitzontal (cap a l'esquerra). El resultat és equivalent al rebot diagonal que es produiria si hi hagués un obstacle en la direcció de moviment inicial de la pilota. En el cas del triple xoc (a), el rebot 1 desvia la velocitat en vertical (cap amunt), mentre que el rebot 3 desvia la velocitat en diagonal (cap a l'esquerra i cap avall). Aquest cas es produiria si no hi hagués rebot horitzontal. Com que la posició final està ocupada per un altre obstacle (fletxa gris), l'última condició del algorisme impedeix el moviment final. La següent invocació de la funció (figura b del triple xoc) farà que la pilota reboti un altre cop contra el primer obstacle (rebot 1) i continuï en direcció contrària a la d'arribada.