

CORSO di

# SISTEMI OPERATIVI “REAL-TIME”



## GIUOCO DEL PING-PONG

GIAN MARCO ROSSETI  
LORENZO SALVADORI  
FARALLI ANDREA



ANNO ACCADEMICO 2003-04



UNIVERSITA' DEGLI STUDI DI SIENA  
INGEGNERIA DELL'AUTOMAZIONE - AREZZO

Il progetto "Ping-Pong" riprende molti aspetti del ben più noto gioco del Pong. In grafica minimalista, due cursori si devono contendere una pallina che rimbalza sulle pareti del campo di gioco, tentando di segnare un punto all'avversario. Le regole sono semplici:

- Il giocatore pilota il cursore rosso tramite il mouse.
- La pallina può solo rimbalzare su di esso con angolo dipendente dal punto d'impatto.
- La provenienza del cursore- giocatore (destra o sinistra) impone la nuova direzione della pallina dopo il contatto.
- L'avversario tenta di intercettare la traiettoria della pallina e rilanciarla al mittente.
- Se si colpisce la parete alle spalle dell'avversario si segna un punto e si sale di livello.
- Se la pallina colpisce la parete alle spalle del cursore- giocatore la partita finisce.

Come richiesto, il progetto è stato implementato utilizzando l'ambiente S.Ha.R.K. il quale ci permette di utilizzare il linguaggio di programmazione C ampliato di alcune librerie di funzioni che permettono la multiprogrammazione con esplicite funzioni di gestione temporale. S.Ha.R.K., ci consente inoltre, la possibilità di gestire i più noti algoritmi di scheduling ed implementare funzioni "semaforo" per le sincronizzazioni fra i diversi task concorrenti e gli accessi a strutture dati condivise.

L'idea principale è stata quella di realizzare un programma che, naturalmente, esegue concorrentemente due o più task. Questi task devono avere un'attivazione periodica cioè devono avvenire "richieste di esecuzione" ad intervalli regolari e terminare entro una, più o meno nota, deadline (nel nostro caso entro è la "prossima" attivazione del medesimo task). Altra specifica richiesta nella stesura del progettino è la presenza di una struttura dati condivisa alla quale cooperano i diversi task (e accedono in maniera competitiva).

Il primo task creato è stato il gestore del mouse quindi il cursore- giocatore. Il mouse deve essere inizializzato; questo è stato fatto dentro la funzione "main". I passi più importanti per quanto riguarda l'inizializzazione sono:

- La scelta del tipo di mouse attraverso la funzione *mouse\_def\_ps2* che ovviamente definisce l'utilizzo di un mouse di tipo ps2.
- La funzione *mouse\_limit* che ci ha permesso di restringere l'area di lavoro del mouse in modo da confinare il movimento del cursore lungo una linea retta. In questo modo il cursore- giocatore può solo spostarsi a destra e a sinistra.
- La funzione *mouse\_threshold* utile per definire la sensibilità del mouse.

L'aspetto più importante per quanto riguarda la gestione del cursore- giocatore è dato dalla *mouse\_hook*. Questa ha il compito di richiamare una funzione da eseguire all'interno del programma ad ogni movimento del mouse; ad ogni movimento del mouse, viene generato un interrupt, questo ha il compito a sua volta di fare una richiesta per l'esecuzione di un determinato task ad esso "agganciato". Un vantaggio di questo metodo è sicuramente quello di non caricare il sistema se non ci sono eventi sul mouse ma, per ritorno, abbiamo lo svantaggio di dover gestire un task ad attivazione sporadica, il quale va ad interferire con il lavoro dei task ad attivazione periodica, compromettendone così il tempo di esecuzione.

Fatta questa breve premessa rimane solo da spiegare cosa richiamo la *mouse\_hook*. Viene richiamata la funzione *my\_mouse\_handler*; questa è forse la funzione più veloce all'interno del programma. Questo principalmente perché, come già detto, non è periodica, quindi, per non interferire pesantemente sugli altri task, è stata ridotta al minor numero di istruzioni possibili.

#### Descrizione di *my\_mouse\_handler*:

La prima istruzione importante riguarda un semaforo di mutua esclusione (*mutex\_gi*). Questa è

stata inserita poiché si accede alla struttura condivisa cioè siamo in una *sezione critica*. Dato che stiamo per utilizzare dei dati condivisi tra i diversi task, vogliamo evitare assolutamente che, “qualcun altro” li modifichi, in modo totale o parziale, mentre vengono letti e valutati. Allo stesso modo così evitiamo anche di modificarli mentre un altro task sta cercando eventualmente di leggerli.

Descrizione delle istruzioni più importanti:

```
- if (m->x > str_cond.xattuale) str_cond.direzione_mouse =destra;  
if (m->x < str_cond.xattuale) str_cond.direzione_mouse=sinistra;
```

Indicano la provenienza del mouse in modo tale da rilanciare la pallina a sinistra se il cursore proviene da sinistra e viceversa.

```
- str_cond.xattuale=m->x;  
str_cond.yattuale=m->y;
```

Scrivono le coordinate attuali del mouse su variabili globali in modo da poter essere utilizzate anche dagli altri task (struttura condivisa).

```
-grx_box(str_cond.xvecchio,str_cond.yvecchio,str_cond.xvecchio+50,str_cond.yvecchio+15,BLACK);  
grx_box(m->x,m->y,m->x+50,m->y+15,RED);  
str_cond.xvecchio=m->x;  
str_cond.yvecchio=m->y;  
sem_post(&mutex_gi);
```

Simulo il movimento del cursore che “insegue” il puntatore del mouse.

Passando ai task periodici dobbiamo definire alcuni aspetti fondamentali come il periodo di richiesta di attivazione, la wct, e lo scheduling ma prima descriviamo il comportamento il loro comportamento.

Nota: Una breve premessa a riguardo i task periodici è che entrambi, dopo l'attivazione, entrano in un loop infinito (*while(1)*). A riguardo viene naturale chiedersi: come viene gestita la periodicità, le deadline e quant'altro se questo task non termina mai? La risposta è data dal comando *task\_endcycle()*. Quest'ultimo, infatti, indica allo scheduler che il task termina lì per poi ripartire al prossimo periodo (richiesta di attivazione del task) dal comando *while(1)*. In pratica la deadline si deve trovare temporalmente dopo l'istruzione *task\_endcycle()*.

#### Descrizione del task *avversario*:

All'interno del main viene attivato (dopo le varie inizializzazioni) come primo il task *avversario*. Lo scopo di questo è di generare un cursore- avversario gestito dal calcolatore. L'avversario deve tentare di intercettare la pallina prima che essa colpisca la parete superiore del campo di gioco. Gli aspetti principali di questo task sono:

- il ciclo *while*, tramite il quale vengono, ripetute una serie di istruzione ad intervalli di richiesta di attivazione regolari.

- il semaforo *mutex\_av*, per accedere alla struttura dati in mutua esclusione. Con questo si proteggono le “valutazioni” che vengono fatte sulle coordinate della pallina le quali potrebbero essere modificate da altri task in competizione per la struttura dati.

- il task si occupa di simulare l'inseguimento della pallina da parte del cursore- avversario. Questo aspetto è stato fatto con il blocco di istruzioni:

```
if ((str_cond.xattuale_av + 25) < str_cond.x)  
{  
    passo = passo + inerzia_av;  
    if (passo >= vmax_av) //limite velocità avversario  
        {passo=vmax_av;}  
    str_cond.xattuale_av += passo;//incremento passo avversario  
}
```

```
if ((str_cond.xattuale_av + 25) > str_cond.x)  
{  
    passo=passo - inerzia_av;
```

```

if (passo <= -vmax_av)
{
    passo=-vmax_av;
}
str_cond.xattuale_av += passo;
}

```

Come si vede è stato simulato un movimento inerziale sul cursore, il quale, arrivato alla coordinata della pallina, non riesce ad arrestarsi subito ma continua oltre il punto esatto, poi, tenterà di tornare indietro fino a raggiungere esattamente la posizione. Tutto questo viene fatto utilizzando la variabile *passo* la quale indica lo l'incremento che il cursore ha lungo l'asse x ad ogni ciclo while. *Passo* naturalmente viene modificata ad ogni ciclo (a seconda della posizione rispetto alla pallina) in modo da simulare le accelerazioni del cursore.

Come si vedrà nel listato l'accelerazione aumenta al salire del livello, cioè ogni volta che l'avversario subisce un punto in modo da rendere il livello più difficile.

Nota: la velocità del cursore ha un valore massimo.

### Descrizione del task fly:

Questo task è l'anima del programma poichè si occupa di fare gran parte del lavoro.

Innanzitutto, a dispetto del nome, il task ha la funzione principale di muovere la pallina nel campo di gioco.

Andando con un po' di ordine indichiamo gli aspetti fondamentali:

- A parte l'inizializzazione anche questo task esegue tutte le istruzioni all'interno del costrutto while in maniera periodica.

- Le coordinate della pallina vengono calcolate con le istruzioni :

```

str_cond.x = str_cond.x + dx;
str_cond.y = str_cond.y + dy;

```

dove *str\_condx* e *str\_condy* sono la posizione mentre *dx* e *dy* sono l'incremento delle coordinate rispettivamente lungo gli assi x e y.

- Inizialmente viene calcolato un angolo di partenza random della pallina (*r*), in modo che questa, partendo dal centro dello schermo, venga lanciata comunque verso il basso.

- Dato un angolo di rimbalzo, gli incrementi *dx* e *dy* sono calcolati con:

```

dx = (int)(veloc_pallina * cos(r*PI/180));
dy = (int)(veloc_pallina * sin(r*PI/180));

```

possiamo così pensare gli angoli in gradi.

- Viene calcolata la direzione di provenienza della pallina (destra o sinistra) in modo da rimbalzare in modo opportuno sul cursore- avversario.

- Game over: Se il giocatore subisce un gol la partita finisce e vengono ripristinati i valori di partenza:

```

if (str_cond.y >= YMAX-4)
{
    indice--; //do la possibilità di lanciare una nuova pallina
    /*ripristino valori iniziali*/
    inerzia_av=0.3;
    veloc_pallina = 8;
    draw_fly(ox, oy, 0);
    vmax_av=10;
    /*rilascio i semafori presi*/
    sem_post(&mutex_gi);
    sem_post(&mutex_av);
    break;
}

```

Nota: Prima di interrompere il task vengono rilasciati i semafori di mutua esclusione.

- Se è l'avversario a subire il gol si segna un punto e si aumenta di livello in modo da rendere il gioco più difficile.

```

if (str_cond.y <= YMIN+4)
{
    veloc_pallina++;
    punteggio++;
    timer=50; /*timer per pausa dopo il goal*/
    if (punteggio == 18) vmax_av = 11;
    grx_text("IL TUO PUNTEGGIO E' ", XMIN, YMENU+40, WHITE, 0);
    sprintf(s2, "%3d", punteggio);
}

```

```

grx_text(s2, XMIN +150, YMENU+40, WHITE, 0);
if (veloc_pallina >= 14) veloc_pallina=14;//limite velocità pallina
inerzia_av = inerzia_av + 0.33;
dy = -dy;
}

```

Note: La pallina aumenta di velocità fino ad un limite massimo.

Il *timer* blocca il gioco per un po' di tempo prima che l'avversario rilanci la pallina.

- Aspetto chiave del gioco sono i rimbalzi della pallina sui cursori. Quest'ultimi sono divisi in bande in modo da far rimbalzare la pallina con angolo differente a seconda del punto d'impatto. E' premura del giocatore colpire con angolo opportuno in modo da spiazzare l'avversario e segnare il goal. Vediamo un esempio di rimbalzo sul cursore\_giocatore:

```

if ((str_cond.y >= str_cond.yattuale-5) && (str_cond.y <= str_cond.yattuale+5) && (str_cond.x >= str_cond.xattuale-3)
&& (str_cond.x < str_cond.xattuale))
{
dx = (int)(veloc_pallina * cos(20*PI/180));
dy = (int)(veloc_pallina * sin(20*PI/180));
dy = -abs(dy);
if (str_cond.direzione_mouse == sinistra)
{
dx = -abs(dx);
str_cond.x += dx;
}
else {dx = abs(dx);
str_cond.x += dx;
}
str_cond.y += dy;
}

```

Siamo, come si vede, nella zona esterna sinistra (molto difficile da colpire). Si nota subito che la "zona" di rimbalzo lungo la y è una certa fascia e non un punto. Questo perché abbiamo calcolato che la pallina nelle peggiori condizioni (angolo di incidenza 45° e velocità massima) ha un passo dy dato da  $\sin(45) * \text{velocità\_massima\_pallina}$ . Non sapendo quindi dove cadrà con precisione sappiamo comunque che cadrà in questa fascia. La pallina sarà rilanciata secondo la provenienza del cursore\_giocatore.

Nota: Dato che ad un certo punto del while le coordinate del cursore\_giocatore non servono più viene rilasciato il semaforo dedicatogli (*mutex\_gi*) in modo che il mouse possa tornare a lavorare "normalmente" mentre ancora viene tenuta la mutua esclusione col task avversario.

- Il rimbalzo sull'avversario è fatto con gli stessi criteri del cursore\_giocatore. L'unica differenza sta nella direzione del rimbalzo che dipende unicamente dalla provenienza della pallina.

### Note conclusive.

Gli aspetti *real-time* di questo gioco sono stati dati dalla scelta dei parametri passati al kernel:

- Innanzitutto abbiamo fatto la scelta di utilizzare dei task di tipo hard. Forse non è la scelta migliore per questo tipo di applicazione dato la varietà dei task aperiodici presenti nel sistema come ad esempio la *mouse\_hook*. Questo però crediamo che renda meglio l'idea di *sistema real-time* in quanto il salto di dead-line ha eventi catastrofici sul sistema. E' qui che diventa veramente importante la scelta dei diversi parametri da passare allo scheduler. Per quello che riguarda l'algoritmo di scheduling abbiamo utilizzato EDF (Earliest Deadline First) il quale ordina dinamicamente le attivazioni dei vari task richiedenti per deadline crescente. Questo algoritmo ricordiamo, risulta essere sempre utilizzabile se esiste un qualsiasi tipo di algoritmo valido per il determinato gruppo di task quindi ci è sembrata la scelta più valida.

Per quanto riguarda le attivazioni dei due task periodici abbiamo scelto il valore di 40ms il quale risulta un buon valore per avere nell'occhio umano, la sensazione di un movimento fluido. Infatti a 25 Hz troviamo appunto i 40 ms di periodo.

Il punto critico è stato nella scelta dei *wcet* (worst case execution time). Infatti per la scelta abbiamo dovuto procedere un po' per tentativi. Abbiamo scelto dei valori e testato il gioco fino a riscontrare degli errori di "deadline miss". Testando il gioco su due macchine abbiamo riscontrato valori molto diversi. Ad esempio:

- su un PIII a 450MHz non possiamo scendere al di sotto di 1.2ms per la *wcet* del task *fly* e 1 ms per il task *avversario*.

-su le macchine del laboratorio (Athlon Xp 2000) otteniamo valori molto inferiori: 0.7 ms 0.2 ms rispettivamente per il task *fly* e il task *avversario*.