Sistemi operativi 2/ed
Paolo Ancilotti, Maurelio Boari, Anna Ciampolini, Giuseppe Lipari
Copyright © 2008 – The McGraw-Hill Companies srl

SOLUZIONI DEI PROBLEMI DEL CAPITOLO 5

1. Nel caso di gestione sincrona, visto nel testo, la funzione di lettura

```
int read (int disp, char *pbuf, int cont)
```

attiva il dispositivo e blocca il processo chiamante in attesa della terminazione del trasferimento. Per questo motivo sono necessari tre parametri: il dispositivo (`disp`) da attivare, il puntatore al buffer (`pbuf`) in memoria nel quale trasferire i dati letti, e il numero di dati (`cont`) da leggere. Inoltre, la funzione, una volta terminata l'esecuzione, e cioè dopo che il processo chiamante è stato risvegliato alla fine del trasferimento, restituisce un intero corrispondente a reale numero di dati letti, se l'operazione è terminata correttamente, oppure il valore -1 in caso di errore. Volendo adesso realizzare una gestione asincrona, è necessario che le operazioni precedentemente svolte dalla sola funzione `read` vengano separate nelle due nuove funzioni `start_input` e `wait_input` relative, l'una all'attivazione del trasferimento e l'altra all'eventuale blocco del processo chiamante e alla restituzione del valore relativo all'esito del trasferimento stesso.
Per questo motivo la funzione `start_input` ha bisogno esattamente degli stessi parametri visti per la `read`. La funzione però, una volta inizializzati i campi del descrittore del dispositivo e una volta attivato il dispositivo stesso, termina senza sospendere il processo chiamante che quindi può proseguire in modo asincrono e concorrentemente con l'esecuzione del trasferimento di dati. Per questo motivo, la funzione non restituisce nessun valore:

```
void start_input(int disp, char * pbuf, int cont){
    descrittore[disp].contatore = cont;
    descrittore[disp].puntatore = pbuf;
    < attivazione del dispositivo> ;  // bit di start=1
{
```

Una volta attivato il dispositivo, alla fine della lettura di ciascun dato, la CPU viene interrotta ed è quindi necessario gestire le generiche interruzioni esattamente come visto nel caso sincrono e segnalare tramite il semaforo `dato_disponibile` presente nel descrittore del dispositivo, che l'operazione è terminata soltanto in corrispondenza dell'arrivo dell'ultima interruzione.

```
void inth() {  // funzione di risposta alle interruzioni
 char b;
 <legge il valore del registro di stato del controllore>;
 if (<bit di errore = = 0>)  // assenza di errori
 { <trasferimento del valore del registro dati nella variabile locale b>;
   *(descrittore[disp].puntatore) = b;
   descrittore[disp].puntatore ++;
   descrittore[disp].contatore --;
   if(descrittore[disp].contatore!=0)<riattivazione dispositivo>;
   else
      {descrittore[disp].esito=<codice di terminazione corretta>;
      <disattivazione del dispositivo>;
      descrittore[disp].dato_disponibile.signal();
      }
 }
 else   // presenza  di errori
   {   <routine di gestione errore>;
```

Copyright © 2008 – The McGraw-Hill Companies srl

```
            if (<errore non recuperabile>)
            {descrittore[disp].esito=<codice di terminazione anomala>;
            descrittore[disp].dato_disponibile.signal();
            }
        }
      return ; // ritorno da interruzione
    }
```

Quando il processo ha necessità conoscere l'esito del trasferimento invoca la funzione che dovrà bloccare il processo se il trasferimento non è terminato, e restituire l'esito del trasferimento una volta terminato. La funzione restituisce quindi un intero:

```
int start_input () {
    descrittore[disp].dato_disponibile.wait();
    if (descrittore[disp].esito == <codice di errore>)
          return(-1); // in caso di errore restituisce –1
    return(cont - descrittore[disp].contatore);
                        // altrimenti restituisce il numero di dati letti
}
```

Se, quando il processo chiama questa funzione, il trasferimento non è terminato il processo si sospende perché, non essendo ancora arrivata l'ultima interruzione, sul semaforo `dato_disponibile` non è stata ancora eseguita la `signal`, altrimenti il processo completa l'esecuzione della funzione ottenendo come risultato il numero di dati letti o il valore -1 in caso di errore.

2. Il tempo richiesto per leggere i primi 160 record allocati negli ultimi 160 settori della traccia N. 3 è pari al tempo di *seek*, più la *rotational latency*, più il tempo richiesto per il trasferimento dei 160 settori pari a 3 millisecondi in quanto questi settori corrispondono alla metà dei settori presenti sulla traccia e quindi necessitano di un tempo di trasferimento pari alla metà del tempo di rotazione del disco:

$$5.2 + 3 + 3 = 11.2 \text{ millisecondi}$$

Per leggere i successivi 320 settori della traccia N. 4 è necessario il tempo di *seek*, che però adesso si riduce al minimo per il passaggio da una traccia alla successiva, più la *rotational latency*, più il tempo necessario per il trasferimento dei 320 settori, che ora corrisponde al'intero tempo di rotazione del disco:

$$0.6 + 3 + 6 = 9.6 \text{ millisecondi}$$

Lo stesso tempo è necessario per il trasferimento dei 320 settori presenti sulla traccia successiva (la N. 5):

$$0.6 + 3 + 6 = 9.6 \text{ millisecondi}$$

Infine, per trasferire gli ultimi 160 record presenti sulla traccia N. 6, è necessario il tempo minimo di *seek* e la *rotational latency* come nei due casi precedenti, più il tempo di trasferimento dei record, che adesso corrisponde, come per i primi 160 record, alla metà del tempo di rotazione del disco:

$$0.6 + 3 + 3 = 6.6 \text{ millisecondi}$$

Concludendo, il tempo necessario per leggere l'intero file è pari a:

$$11.2 + 9.6 + 9.6 + 6.6 = 37 \text{ millisecondi}$$

Copyright © 2008 – The McGraw-Hill Companies srl

3. Se i 960 record sono allocati su disco in modo casuale, il tempo necessario per trasferire ciascun settore corrisponde al tempo di *seek* (corrispondente per tutti allo stesso valore relativo al tempo medio di *seek*) più la *rotational latency,* più il tempo richiesto per il trasferimento di un singolo settore che, come indicato nella tabella 5.2 riportata nel testo, è di 0.019 milisecondi. Quindi il tempo necessario per trasferire un solo record è di:

$$5.2 + 3 + 0.019 = 8.419 \text{ millisecondi}$$

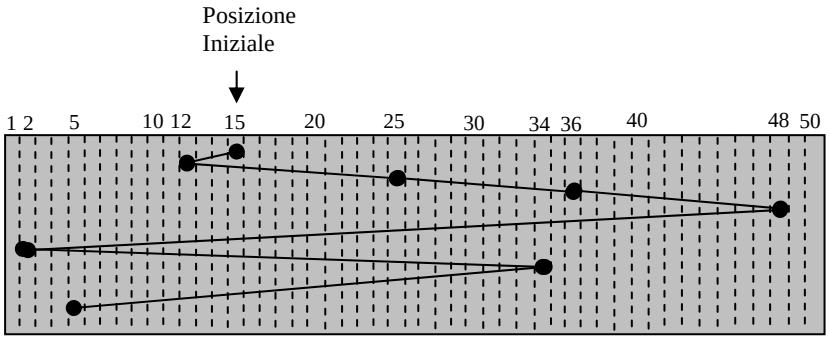
Per cui il tempo complessivo per trasferire l'intero file è di:

$$960*8.419 = 8082.24 \text{ millisecondi}$$

e cioè oltre 8 secondi.

4. Algoritmo FCFS

Di seguito viene riportato lo schema che illustra quali sono i movimenti della testina se, come indicato dal problema, la posizione iniziale della testina è sulla traccia 15 e le richieste pendenti sono, nell'ordine, per le tracce 12, 25, 36, 48, 2, 34, 5 e nell'ipotesi di servire tali richieste con l'algoritmo FCFS (nella figura è stato supposto che 50 sia il numero totale delle tracce del disco):

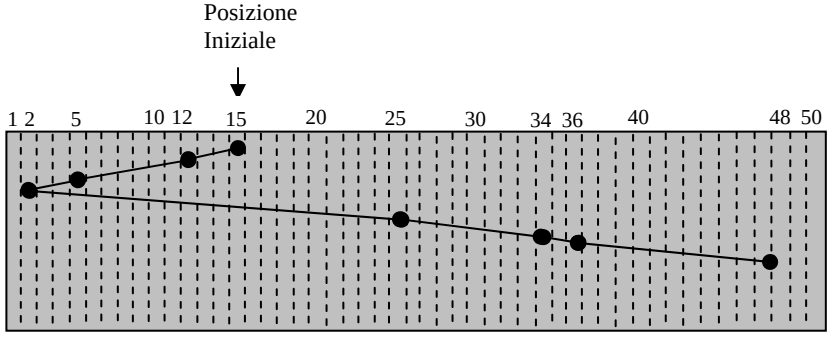


Il numero totale di tracce su cui è necessario spostare la testine è quindi:

$$3+13+11+12+46+32+29=146$$

Algoritmo SSTF

Di seguito viene riportato lo stesso schema ma nell'ipotesi di utilizzare l'algoritmo SSTF:



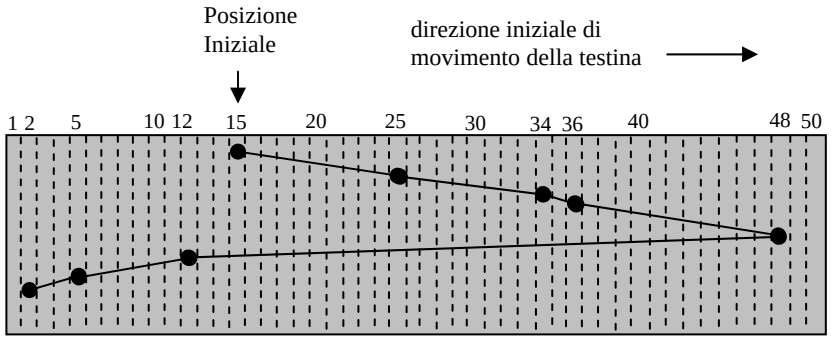
Copyright © 2008 – The McGraw-Hill Companies srl

Il numero totale di tracce su cui è necessario spostare la testina è quindi:

$$3+7+3+23+9+2+12=59$$

Algoritmo SCAN

Nel caso dell'algoritmo SCAN è necessario anche specificare, nel momento in cui devono essere servite le richieste pendenti, qual'è la direzione di movimento della testina, se dalla traccia più interna (traccia 1) alla più esterna (traccia 50) o viceversa.
Di seguito viene riportato lo schema nella prima ipotesi di movimento della testina dalla traccia 1 alla traccia 50:



Il numero totale di tracce su cui è necessario spostare la testina è quindi:

$$10+9+2+12+36+7+3=79$$

Se la direzione di movimento della testina fosse quella dalla traccia 50 alla traccia 1, l'ordine con cui verrebbero servite le richieste pendenti, per questo esempio, corrisponderebbe a quello relativo all'algoritmo SSTF come può essere osservato dallo schema corrispondente**.**