Buffering dell’I-O

Buffering dell’I/O

Una delle tecniche più usate dagli SO per poter fare I/O nella maniera più efficiente è quella del **buffering** (indica una certa zona di memoria temporanea utilizzata per completare una certa operazione).

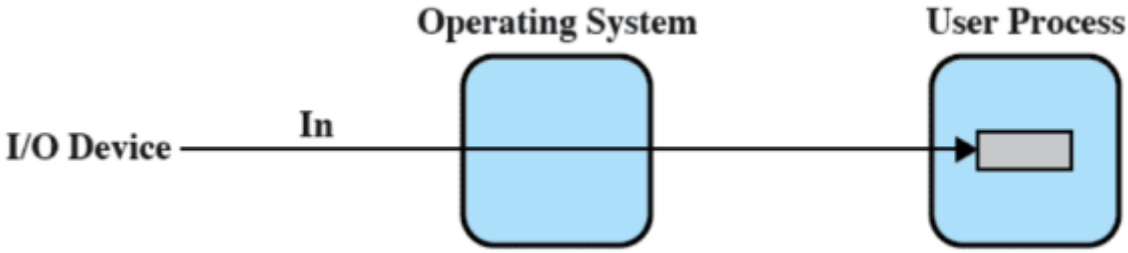
> **Info**
>
> Nella gestione della memoria (RAM) abbiamo visto che è presente il problema di dover spostare le pagine da memoria virtuale a disco (come effetto collaterale di usare la memoria virtuale ci sta quello di gestire l’I/O).

L’idea è quella che un processo potrebbe richiedere un I/O su una certa sua zona di memoria (gestito dal DMA), ma viene subito swappato (quindi sospeso). In questo caso si generebbe una contraddizione in quanto, per poter usare il DMA, il processo (o una sua parte) si deve trovare in memoria ma ciò non è possibile se questo deve essere swappato.
Questo problema potrebbe essere risolvibile attraverso il *frame blocking* (evitare che le pagine usate dal DMA siano swappate), tuttavia se si inizia a fare quest’operazione in maniera eccessiva limito le pagine swappabili e di conseguenza anche il numero di processi presenti in RAM.

Per risolvere questo problema è quindi risolvibile attraverso il **buffering**, ovvero, come per il prepaging, facendo in anticipo trasferimenti di input e in ritardo trasferimenti di output rispetto a quando arriva la richiesta

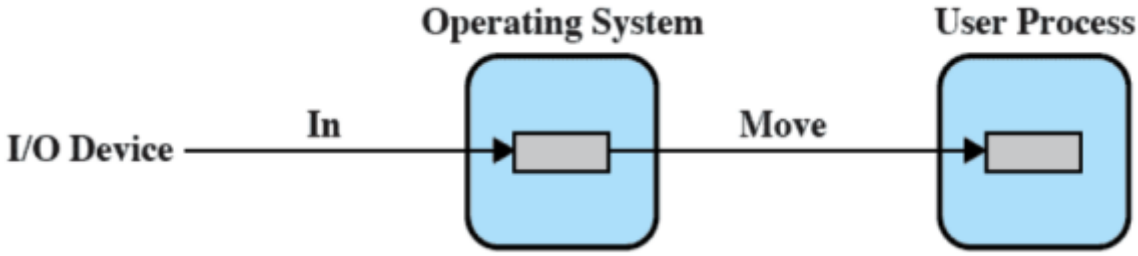
Senza buffer

Se non ci fosse il buffering il SO accede al dispositivo nel momento in cui ne ha necessità



Buffer singolo

Con il buffer viene messa una porzione di memoria ausiliaria nel mezzo gestita dal SO all'interno del kernel (spesso statica, altre volte dinamica). Quindi quando arriva la richiesta di I/O viene letta e scritta nel sistema operativo, e in un secondo momento viene passata al processo utente. In questo modo è quindi prevedibile evitando il contatto diretto tra I/O e RAM



Lettura e scrittura nel buffer sono separate e sequenziali

Buffer singolo orientato a blocchi

Questo tipo di buffering riguarda i dispositivi orientati a blocchi (es. dischi)

Con buffer singolo i trasferimenti di input sono fatti al buffer in system memory. Il blocco viene poi mandato nello spazio utente solo quando necessario.
A questo punto, nonostante non sia arrivata nessun'altra richiesta di **input**, il blocco successivo viene comunque letto nel buffer (input **anticipato**) in quanto i dati sono solitamente acceduti sequenzialmente e dunque ci sta una buona possibilità che servirà, e sarà già stato letto (questa cosa funziona molto bene per quanto riguarda i dischi)

L'**output** invece viene **posticipato**. Per questo motivo è necessaria la system call `flush` quando si fa debugging in C (un errore potrebbe verificarsi ben dopo un print, ma non si ha nulla a schermo a causa dell'output posticipato, uso quindi `flush` per svuotare il buffer)

Buffer singolo orientato agli stream

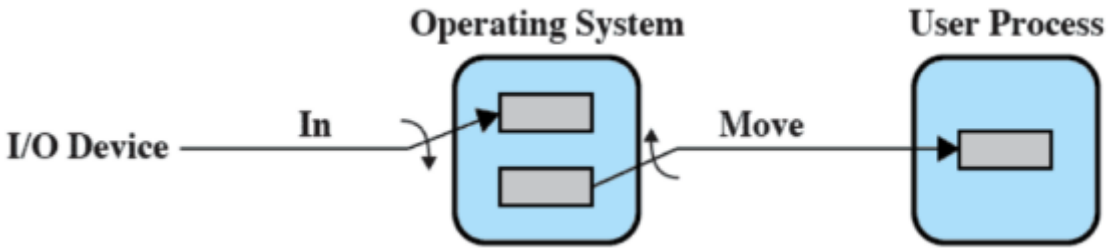
Questo tipo di buffering riguarda i dispositivi orientati agli stream (es. mouse, tastiera, schede di rete)

Qui invece di avere un blocco grande come capita per il disco, la bufferizzazione riguarda molto spesso un'intera linea di input o output (es. una riga intera del prompt) Viene invece bufferizzato un byte alla volta per i dispositivi in cui un singolo carattere premuto va gestito (in pratica è un'istanza di un ben noto problema di concorrenza, il produttore/consumatore)

Buffer doppio

Essendo particolarmente piccolo il buffer all'interno del SO se lo utilizzo per gestire tutti i dispositivi di I/O, questo si riempie. Per ovviare a questo problema mi è utile utilizzare un buffer multiplo

L'idea è quindi che un processo può trasferire dati da o a uno dei buffer, mentre il SO svuota o riempie l'altro



Lettura e scrittura nel buffer sono parallele: un buffer letto, l'altro scritto

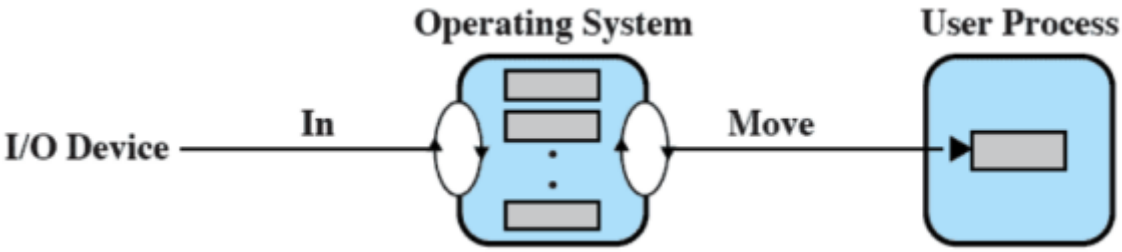
---

Buffer circolare

Quando sono presenti più di due buffer, si parla di buffer circolare

In questo caso ciascun buffer è un'unità del buffer circolare e viene utilizzato quando l'operazione di I/O deve tenere il passo del processo

> **Hint**
>
> Anche qui riguarda il problema del produttore/consumatore



In questo caso i buffer vengono riempiti e svuotati in ordine

---

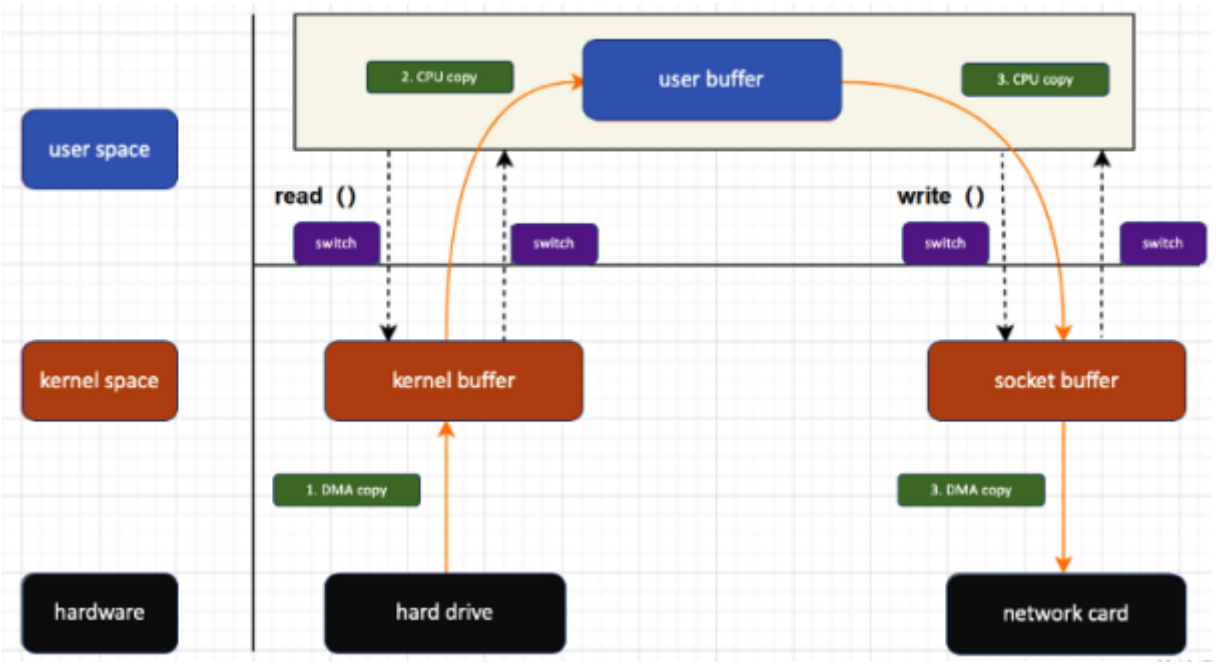
Buffer: pro e contro

Permette di mantenere il processore non idle anche quando sono presenti numerose processe di I/O, ma se la domanda è molta, i buffer si riempiranno e il vantaggio si perde
Utile soprattutto quando ci sono molti e vari dispositivi di I/O da gestire, migliora infatti l'efficienza dei processi e del SO

I pro sono maggiori dei contro

Overhead

Introduce overhead a causa della copia intermedia in kernel memory



Zero copy

Evita inutili copie intermedie, facendo un trasferimento da un kernel buffer ad un altro