Avevamo visto un esempio di device driver, evidenziando la funzione dei buffer. Oggi vedremo qualche dispositivo di ingresso in più e il loro effetto sui buffer.

Immaginiamo di avere CPU. Memoria RAM e Dispositivo di Input/Output.

Facendo riferimento alla architettura Amber, i registri sono da 32 bit. L’idea è di avere un sottoinsieme dei bit del registro (7) da cui leggere il codice ascii del successivo carattere messo in input sulla tastiera. Il nostro dispositivo lavorerà sotto interruzioni (che invierà dopo aver aver finito la trascrizione del carattere). Quando arriva una richiesta di interruzione, se non è mascherata, il processore parte ad eseguire l’interrupt handler, ossia la sequenza di istruzioni in RAM che gestiscono quel particolare tipo di interruzioni (tali istruzioni sono puntate da una cella presente tra le prime della RAM, perché tutti gli handler sono uno dietro l’altro e sono puntatori al codice effettivo).

Potremmo pensare di avere, oltre la tastiera, un secondo dispositivo di IO, che sia un TTY (nome convenzionale dei terminali sul sistema Unix, si trovano nel folder dev) Output (per esempio un monitor, o una stampante).   
L’organizzazione del dispositivo di output vede a sua volta un registro in cui i 7 bit meno significativi contengono la codifica di un carattere in codice ascii. Questo ovviamente non basta, perché ci vuole anche un’indicazione che indichi se il dispositivo è pronto a ricevere un nuovo carattere o no, stesso discorso vale per il dispositivo di input: serve un bit in più.

Per questioni di comodità questo bit in più è quello più significativo: in questo modo tramite un’operazione di lettura è sufficiente controllare il bit di segno. 1 vuol dire dispositivo libero e 0 significa occupato.  
Per scrivere il programma dell’interrupt handler possiamo semplificarci la vita associando il segno negativo (1) al dispositivo libero e il segno positivo (0) al dispositivo occupato.

Il gestore delle interruzioni, nel caso della tastiera, copia il carattere da questa a un registro della cpu, dopodiché lo sposta in un buffer della memoria RAM. In questo modo possiamo collezionare un carattere alla volta, fino a riempire il buffer di ingresso.   
Tipicamente le stringhe di caratteri lette dal dispositivo di ingresso dovranno poi essere consegnate a una qualche applicazione, che ha richiesto la lettura. Questa consegna avverrà tramite una copiatura dal buffer a un’altra ridotta area di memoria dedicata a quell’applicazione (perché a causa della memoria virtuale un’applicazione non può mai accedere direttamente a un buffer, quindi il sistema operativo glie ne dedica una più piccola tutta per lei).

Mentre nel caso dell’output questa copia avviene tramite il flush, nel caso dell’input si può dare un significato particolare ad alcuni caratteri: convenzionalmente, il carattere che ha questa proprietà particolare è il carattere new\_line (ottenuto premendo invio).

La system call usata dalle applicazioni per prendere l’input da tastiera è la system call READ.   
In genere per ogni applicazione in esecuzione vengono aperti tre file: stdin, stdout e stderr, che hanno diverso file descriptor (0, 1 e 2). Quindi per accedere allo standard input (che normalmente corrisponde al dispositivo di ingresso del terminale, cioè la tastiera) bisogna chiamare una system call READ che ha file descriptor 0, e che presenta l’indirizzo e le dimensioni del buffer. Questo effettua una copia dal buffer interno (a livello di sistema) al buffer applicativo. Se quando viene chiamata la READ il carattere di terminazione di riga (l’a capo) non è ancora presente nel buffer, allora la call resta in sospeso finchè non viene inserito.

Questo è il modo di fornire istruzioni al livello più basso in assoluto, noi in genere non utilizziamo questo metodo: abbiamo funzioni di libreria del C che fanno da interfacce di programmazione di livello di più alto, ossia getc, getchar, scanf, fscanf…  
Per implementare queste funzioni si utilizza un ulteriore meccanismo di bufferizzazione. A livello di OS avviene che, quando abbiamo il terminatore di riga può essere effettuata l’operazione di copia nel buffer applicativo. Dal buffer applicativo si può poi accedere a un carattere alla volta con la getc, o all’intero buffer in una volta con scanf.

Negli esempi di dispositivi visti adesso non abbiamo parlato di DMA perché dovendo effettuare i trasferimenti di dati un byte alla volta (dalla tastiera viene premuto un solo carattere per volta) è più efficiente usare direttamente il processore.

Vediamo adesso un esempio di dispositivo di IO a blocchi, come un’unità a disco, che sia quindi dotata di un sistema DMA di tipo BUS-Master. L’unità a disco viene formattata in una serie di blocchi. Essi sono la quantità minima di informazioni che possono essere lette oppure scritte. Normalmente, in un sistema reale dotato di MMU, viene utilizzato un meccanismo di Paginazione (per cui la RAM è suddivisa in pagine). Può essere molto interessante l’idea di far coincidere la dimensione delle pagine RMA con le dimensioni dei blocchi di informazioni nell’unità a disco. In questo modo a ogni blocco corrisponde una pagina; ciò rende più facile i trasferimenti.   
Questa è una semplificazione, non strettamente necessaria ma intelligente da inserire. Supponiamo di effettuare una scrittura di dati da una pagina RAM a un blocco dell’unità a disco. Per fare questo il processore usa il meccanismo del DMA contenuto nell’unità a disco. Abbiamo sempre l’idea di utilizzare dei buffer: in questo caso il buffer sarà al livello di sistema (non accessibile alle applicazioni). Si può quindi programmare il Bus-Master inserendogli l’indirizzo del buffer, la dimensione (corrispondente alla lunghezza di una pagina), l’indirizzo del blocco dell’unità a disco e dicendogli che si tratta di un’operazione di scrittura, ovviamente ammesso che il DMA sia libero (quindi si spenderà un altro bit per quello). Fatto questo il sistema operativo si disinteressa dell’operazione, l’unica cosa che deve assicurarsi è che alla pagina buffer non acceda alcuna applicazione. Alla fine dell’operazione il bit di stato del DMA passerà a “libero” e invierà un interrupt al processore, che andrà a vedere quali dispositivi possono aver chiamato l’interrupt e notare che il DMA è libero. Fatto ciò l’area di memoria RAM usata come buffer può essere usata per altri scopi.

Se io volessi effettuare un trasferimento di dati in uscita dovrei fare gli stessi passi ma al contrario (dopo la programmazione del DMA si individua un’area di memoria da usare come buffer e si copiano lì i dati dall’unità a disco). Nell’operazione di lettura dalla unità a disco inoltre la programmazione vede l’indirizzo del buffer, l’indirizzo del blocco, le dimensioni del blocco e l’ordine di lettura (se ovviamente il DMA è libero).

Anche in questo caso sarà richiesta un’operazione di copia dei buffer di sistema per renderli accessibili alle applicazioni, anche in questo caso avremo quindi le system call di write e read. La WRITE conterrà infatti il file descriptor (che individua la zona del disco su cui scrivere le informazioni), l’indirizzo del buffer a livello utente (da cui prendere le informazioni) e le dimensioni del buffer. Il buffer del sistema operativo individuato dovrà quindi essere di dimensioni maggiori o uguali di quello usato a livello utente (per non perdere informazioni). Supponendo che le pagine siano da 1024 byte, potrebbe succedere che l’applicazione richieda di scrivere un buffer di dimensioni 2000 byte. I primi 1024 byte saranno scritti in una pagina e i successivi 976 byte saranno scritti in un'altra pagina (per un totale di 2048 byte occupati a livello di sistema, ci saranno quindi 48 byte in più che assumeranno il valore 0…0 e che verranno comunque copiati nell’unità a disco). Quando verrà fatta questa operazione di copia non è dato saperlo dal punto di vista di un’applicazione, la domanda è fatta al sistema operativo e una volta che i dati sono stati scritti nei buffer di sistema sta al sistema operativo decidere Quando inviarli poi all’unità a disco. Questo rende il sistema molto più efficiente (perché così il sistema fa queste operazioni solo quando non ha altro di importante da fare), d’altra parte, il fatto che ci possa essere un ritardo anche abbastanza notevole (secondi o addirittura minuti) causa complicazioni a livello di tolleranza ai guasti: se a un certo punto si guasta qualche cosa (viene meno l’alimentazione) c’è la possibilità che non tutti i dati siano stati salvati in unità a disco e tutto quello che è rimasto nel buffer di sistema viene perso (possono essere salvati parzialmente o affatto).

Si ricorda che nel processore Amber tutte queste System Call sono implementate con la SWI.

Nel caso opposto, cioè dell’operazione di lettura, le cose sono più semplici: bisogna reperire le informazioni presenti nell’unità a disco. La prima cosa da fare è programmare il DMA, che completata l’operazione di “lettura”, ossia il trasferimento dei dati dall’unità a disco al buffer RAM è sufficiente effettuare la copia dal buffer di sistema a quello applicativo. Anche in questo caso l’utente non può controllare il numero preciso di byte da trasferire nel buffer di sistema. Se l’utente vuole leggere solo dieci byte il buffer di sistema conterrà comunque una pagina/ un blocco di memoria a disco. Solo che può convenire fare ciò e conservare i dati in più, perché potrebbe capitare che l’utente poi decida di voler utilizzare anche i byte dopo. Quindi il buffer di sistema (o meglio la pagina del buffer, se ci sono più pagine allora le pagine già lette completamente possono essere riutilizzate) viene ripulito solo DOPO che l’utente ha letto tutte le celle di tale buffer e le ha copiate nel buffer applicativo (anche se magari gli servivano solo 10 byte, in ogni casa all’interno del buffer applicativo ci dovranno passare tutti).

In genere l’operazione di lettura da memoria a disco prevede che prima il file venga “aperto” tramite una system call di tipo OPEN, ma noi non approfondiremo questo aspetto.