

CPU-SCHEDULUNG

- 1. Tw: il tempo che un processo attende nella coda di ready prima che venga messo in running **tempo attesa medio**
- 2. Tr: il tempo che intercorre tra la fine di un IO burst e la messa in esecuzione di un processo **tempo risposta medio**
- 3. Tt: il tempo necessario per la conclusione di un processo. **tempo esecuzione medio**

Esercizio 3

Si consideri un Sistema Operativo batch, avente una tabella di processi di dimensione 20. Si assuma che i job durino in media 10s. Con queste premesse, ogni quanto tempo il sistema puo' accettare un nuovo job senza eccedere il numero di PCB disponibili?

$$20 = \lambda \cdot 10 \rightarrow \lambda = \frac{20}{10} = 2 \text{ Hz}$$

Soluzione La Legge di Little permette di mettere in relazione dimensione della coda dei processi, tempo medio di esecuzione e frequenza media di arrivo degli stessi; tale formula e' descritta dalla relazione:

$$n = \lambda \cdot W \quad (1)$$

dove n indica la dimensione della coda, λ la frequenza di arrivo media e W il tempo di esecuzione medio.

Earliest deadline first

- Condizione necessaria per EDF scheduling: $U_{CPU} \leq 1$ con $U_{CPU} = \text{SOMMATORIA DI } (CPU \text{ CURST / PERIOD})$

$$U_{CPU} = \sum_{\text{processi}} \frac{\text{CPU BURST}}{\text{PERIOD}}$$

MEMORY

bit \times address bus \times con x : mem. fisica moltiplicabile come 2^x

popine = memoria fisica moltiplicabile come 2^x = 2^x

bit \times moltiplicare i popine = 2^x dell'operazione precedente

sezioni = s , con s : # indirizzi logici = $\frac{2^{\text{bit} \times \text{indirizzo logico}}}{\text{mem. fisica come } 2^x} = 2^s$

KiB	: 2^{10}
Mega	: 2^{20}
Giga	: 2^{30}
Tera	: 2^{40}
4	= 2^2
8	= 2^3
16	= 2^4
32	= 2^5
64	= 2^6
128	= 2^7
256	= 2^8
512	= 2^9
1024	= 2^{10}

La formula generica del *Effective Access Time* e' data dalla relazione

$$\text{EAT} = p_{hit}(T_{TLB} + T_{RAM}) + p_{fault}(2T_{TLB} + 2T_{RAM})$$

VIRTUAL MEMORY

ciclo lettura scrittura

Come si puo' implementare l'algoritmo di sostituzione delle pagine *Second Chance*?

Soluzione L'algoritmo *Second Chance* - o altresi' noto come *clock algorithm* - e' un algoritmo per la sostituzione delle pagine che usa un reference bit (implementato in hardware) per capire quale pagina pagina e' stata usata meno di recente e quindi rimpiazzare. Si tratta di una approssimazione dell'algoritmo di rimpiazzamento ottimo che dispone in anticipo della sequenza di pagine richieste. In particolare:

- se il reference bit di una pagina e' 0 allora essa viene rimpiazzata;
- se il reference bit e' 1 il bit viene settato a 0 e la pagina viene lasciata in memoria, quindi si passa alla pagina successiva - usando le stesse regole.

Un modo per implementare tale algoritmo e' attraverso una *coda circolare* con un puntatore alla pagina da rimpiazzare - che scorre nella coda finche' non trova una pagina con reference bit pari a 0. Trovata una pagina che soddisfa le richieste, viene eliminata dalla coda e rimpiazzata con la nuova. Il funzionamento dell'algoritmo e' illustrato in Figura 1.

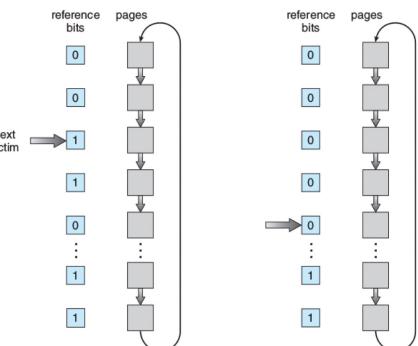


Figure 1: L'algoritmo di sostituzione delle pagine *Second Chance*. Il puntatore scorre finche' una pagina con reference bit 0 non viene trovata; quindi la nuova pagina prendera' il posto di quest'ultima nella coda circolare.

OR: toglie quello utilizzato dopo

LFU: toglie quello utilizzato più lavorato nel tempo
(quello che è inutilizzato da + tempo)

FILE SYSTEM

Sistemi Operativi

AA 2020/21

Esercitazione

19 Maggio 2021

Esercizio 1

Si consideri un file consistente di 100 blocchi e che il suo *File Control Block* (FCB) sia già in memoria. Siano dati due file-systems gestiti rispettivamente tramite allocazione a lista concatenata (*Linked List Allocation*) e allocazione indicizzata (*Indexed Allocation*). Si assuma che nel caso indicizzato il FCB sia in grado di contenere i primi 200 blocchi del file.

Domanda Si calcolino le operazioni di I/O su disco necessarie per eseguire le seguenti azioni in entrambi i file-systems:

1. Aggiunta di un blocco all'inizio del file
2. Aggiunta di un blocco a metà' del file
3. Aggiunta di un blocco alla fine del file

Soluzione Nel caso di file-system con *Indexed Allocation*, ogni file conterrà un index block, ovvero un blocco contenente i puntatori a tutti gli altri blocchi componenti il file. Cio' implica che per raggiungere un dato blocco basterà effettuare una semplice indicizzazione. Nell'implementazione con *Linked List Allocation* invece, ogni blocco contiene il riferimento al precedente ed al successivo. In questo caso, per effettuare operazioni che non siano all'inizio del file bisognerà scorrere la lista fino al blocco desiderato ed in seguito eseguire l'operazione necessaria.

Supponendo che il FileControlBlock sia già stato caricato in memoria, avremo:

1. Linked Allocation: 1 IO-ops; Indexed Allocation: 1 IO-ops
2. Linked Allocation: 52 IO-ops; Indexed Allocation: 1 IO-ops
3. Linked Allocation: 3 IO-ops; Indexed Allocation: 1 IO-ops

Per quanto riguarda Indexed Allocation, abbiamo soltanto una operazione di IO per scrivere il blocco che aggiungiamo. Per quanto riguarda Linked Allocation, quando il blocco è all'inizio del file dobbiamo fare una operazione di IO per accedervi in scrittura. Quando il blocco da aggiungere è a metà del file, 50 operazioni di IO vengono effettuate per leggere la prima metà dei blocchi del file, un'altra operazione è necessaria per accedere in scrittura al cinquantesimo blocco di cui cambiamo il successore, un'ultima viene fatta per scrivere il blocco aggiunto: il totale consta quindi di 52 operazioni di IO. Quando il blocco da aggiungere è alla fine del file, ammesso di avere il puntatore all'ultimo elemento, vi accediamo in lettura e in scrittura per modificare il puntatore al prossimo, quindi effettuiamo un'altra operazione di IO per scrivere il blocco aggiunto, totalizzando così 3 operazioni di IO.

Se invece assumiamo che il FileControlBlock non sia in memoria, allora occorre considerare due operazioni di IO in aggiunta a quelle dello schema precedente: una per caricarlo in memoria, una per scriverlo con la size aggiornata. In questo caso avremo quindi:

1. Linked Allocation: 3 IO-ops; Indexed Allocation: 3 IO-ops

2. Linked Allocation: 54 IO-ops; Indexed Allocation: 3 IO-ops

3. Linked Allocation: 5 IO-ops; Indexed Allocation: 3 IO-ops

A prescindere dal dettaglio del calcolo, che come abbiamo visto può dipendere dalle assunzioni fatte, è importante che sia chiaro il vantaggio della implementazione Indexed Allocation in caso di accessi scattered a blocchi di un file. Il numero di operazioni di IO è in questo caso indipendente dalla posizione del blocco all'interno del file, dal momento che non c'è bisogno di scorrerlo per intero fino al punto desiderato.

Esercizio 2

Che relazione c'è tra un *File Descriptor* ed una entry nella tabella globale dei file aperti del file system?

1) Soluzione Per ogni istanza del file aperto da un processo e identificata dal descrittore del file, una struct del tipo OpenFileRef viene aggiunta ad una lista accessibile nel PCB. Structs associate allo stesso file puntano alla stessa entry nella tabella globale dei file aperti, in cui abbiamo una struct del tipo OpenFileInfo per ogni file aperto.

Riassumendo, mentre possono esistere più descrittori di file per lo stesso file aperto, una e una sola entry ad esso associata esiste nella lista globale dei file aperti, alla quale tutte le OpenFileRef nella lista presente all'interno del PCB puntano.

2) Soluzione Un File Descriptor consiste in un file handler che viene restituito ad un processo in seguito ad una chiamata alla syscall `open()`. In seguito a tale chiamata, il sistema scandisce il FS in cerca del file e, una volta trovato, il FCB è copiato nella tabella globale dei file aperti. Per ogni singolo file aperto, anche se da piu' processi esiste una sola entry nella tabella globale dei file aperti.

Viene, quindi, creata una entry all'interno della tabella dei file aperti detenuta dal processo, la quale punterà alla relativa entry nella tabella globale, insieme ad altre informazione - e.g. permessi, locazione del cursore all'interno del file, ecc. La syscall `open()` restituisce per l'appunto l'entry all'interno della tabella del processo - il File Descriptor. Più `open()` su uno stesso file da parte di uno stesso processo generano descrittori diversi.

Un file descriptor (open file ref) è un file handler restituito ad un processo che apre un file tramite una syscall `open()`. In seguito a tale chiamata il sistema scandisce il File system finché non trova il file, è una volta trovato il suo FCB è copiato UNA SOLA VOLTA nella tabella globale dei file aperti, tramite una struct open File Info.

L'handler del file descriptor è salvato nel PCB del processo, e possono esisterne diversi per lo stesso file, che però puntano tutti alla STESSA entry nella tabella globale.

Esercizio 3

Cos'e' un *File Control Block* (FCB)? Cosa contiene al suo interno?

Soluzione Il FCB e' una struttura dati che contiene tutte le informazioni relative al file a cui e' associato. Esempi di informazioni possono essere: permessi, dimensione, data di creazione, numero di inode (se esiste), ecc. . Inoltre il FCB contiene informazione sulla locazione sul disco dei dati del file - ad esempio in un FS con allocazione concatenata il puntatore al primo blocco del file. In Figura 1 e' riportata una illustrazione di tale struttura.

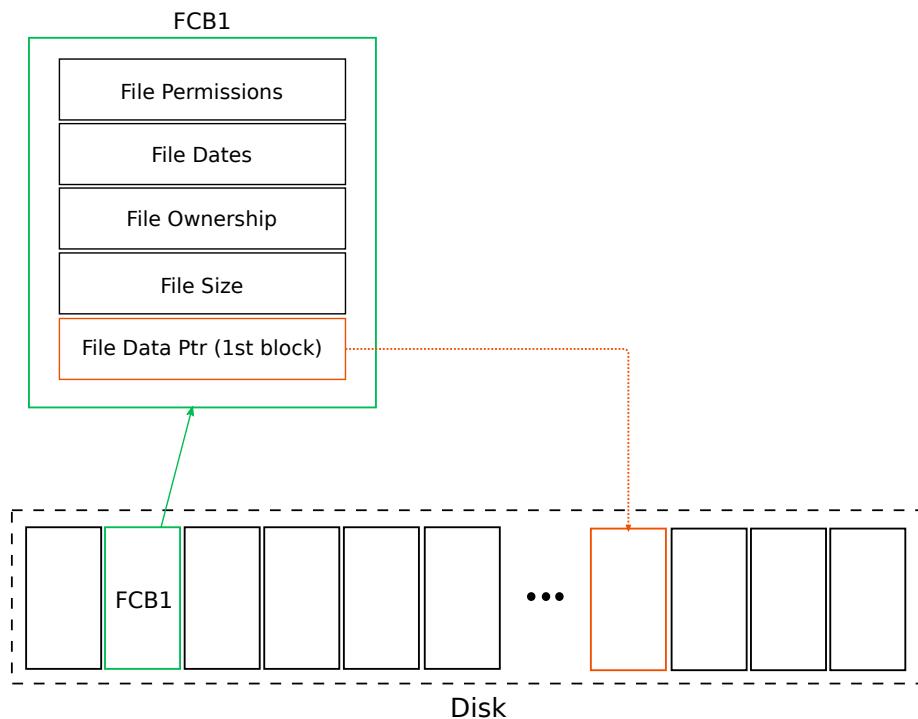


Figure 1: Esempio di FCB. La struttura contiene tutti gli attributi del file compresa la locazione dei dati - qui rappresentato dal puntatore al primo blocco della lista contenente i dati, supponendo un FS con linked allocation.

Esercizio 4

Si consideri l'implementazione di un file system con allocazione concatenata (Linked Allocation) ed un file system che invece utilizzi una allocazione ad indice (Indexed Allocation).

Domanda Illustrare brevemente i vantaggi dell'uno e dell'altro nell'eseguire le seguenti operazioni:

1. accesso sequenziale
2. accesso indicizzato
3. operazioni su file di testo

Soluzione

1. **Accesso Sequenziale:** in questo caso, il file system che usa la *lista concatenata* sarà favorito, garantendo una maggiore velocità dell'operazione. Ciò poiché non bisogna effettuare alcuna ricerca per trovare il blocco successivo, poiché esso sarà semplicemente il blocco **next** nella lista.
2. **Accesso Indicizzato:** questa operazione - contrariamente alla precedente - risulta essere molto onerosa per il file system che usa la *linked list*. Infatti, per ogni accesso, bisognerà scorrere tutta la lista finché non viene trovato il blocco desiderato. La ricerca tramite **inode** risulterà molto più efficiente.
3. **Accesso su file di testo:** per la natura del tipo di file (testo), la *linked list* risulterà più efficiente ancora una volta. Questo poiché i file di testo sono memorizzati in maniera sequenziale sul disco, riportandoci al caso 1.

auto da esami

Esercizio 7

Quali sono le principali differenze tra un indirizzo *logico* ed un indirizzo *fisico*? Da chi vengono generati?

Soluzione Un indirizzo logico non si riferisce ad un indirizzo realmente esistente in memoria. Esso e' in realta' un indirizzo astratto generato dalla CPU, che verrà poi tradotto in un indirizzo fisico tramite la Memory Management Unit (MMU). L'indirizzo fisico, quindi, si riferisce ad una locazione esistente della memoria e *non* e' generato dalla CPU, bensì dalla [MMU](#).

Esercizio 8

Spiegare brevemente la differenza tra `open(...)` e `fopen(...)`.

Soluzione `fopen(...)` una funzione di alto livello che ritorna una stream, mentre `open(...)` una syscall di basso livello che ritorna un *file descriptor*. `fopen(...)`, infatti, nasconde una chiamata alla syscall `open(...)`.

Esercizio 9

In cosa consiste l'operazione di `mount` di un file system?

Soluzione Tramite tale operazione il sistema operativo viene informato che un nuovo file system pronto per essere usato. L'operazione, quindi, provvederà ad associarlo con un dato *mount-point*, ovvero la posizione all'interno della gerarchia del file system del sistema dove il nuovo file system verrà caricato. Prima di effettuare questa operazione di *attach*, ovviamente bisognerà controllare la tipologia e l'integrità del file system. Una volta fatto ciò, il nuovo file system sarà a disposizione del sistema (e dell'utente), come raffigurato nella Figura 2.

Esercizio 10

Cosa succede durante la system call `fork()`? Illustrare in dettaglio i passi necessari alla sua esecuzione, evidenziando come vengono modificate le strutture nel kernel.

Soluzione La syscall `fork()` è usata per creare un nuovo processo - *children* - a partire da un processo *parent*. Il processo creato, avrà una copia dell'address space del parent, consentendo di comunicare facilmente tra loro. I processi, in questo caso, continueranno l'esecuzione concorrentemente. Il child, eredita anche i privilegi e gli attributi nel parent, nonché alcune risorse (quali i file aperti). Una volta creato il processo, se non viene invocata l'istruzione `exec()` il child sarà una mera copia del parent (con una propria copia dei dati); in caso contrario sarà possibile eseguire un diverso comando.

Il parent aspetta che il child completi il suo task tramite l'istruzione `wait()`; quando il child finisce la sua esecuzione (o avviene una chiamata `exit()`), il parent riprende il suo flusso, come riportato in Figura 4.

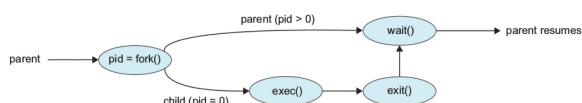


Figure 4: Creazione di un processo tramite syscall `fork()`.

Esercizio 3

Quali risorse vengono usate per creare un thread? In cosa differiscono da quelle usate per la creazione di un processo?

Soluzione Creare un thread - sia esso kernel o user - richiede l'allocazione di una data structure contenente il register set, lo stack e altre informazioni quali la priorità, come riportato in Figura 3.

Creare un nuovo processo invece, richiede l'allocazione di un nuovo PCB (una data structure molto più pesante). Il PCB contiene tutte le informazioni del processo, quali `pid`, stato del processo, informazioni sull'I/O, il Program Counter e la lista delle risorse aperte dal processo. Inoltre il PCB include anche informazioni sulla memoria allocata dal processo (tabella delle pagine, regioni mmapped etc.). Tale operazione è relativamente costosa.

In definitiva, quindi, la creazione di un thread è più leggera rispetto a quella di un nuovo processo.

Esercizio 8

Che relazione c'è tra una *syscall*, un generico *interrupt* e una *trap*? Sono la stessa cosa?

Soluzione Ovviamamente le tre cose **non** sono la stessa cosa.

Una syscall è una chiamata diretta al sistema operativo da parte di un processo user-level - e.g. quando viene invocata la funzione `printf`.

Nel caso si volesse aggiungere una nuova syscall, essa, una volta definita, va registrata nel sistema e aggiunta al vettore delle syscall del sistema operativo, specificandone il numero di argomenti (ed il loro preciso ordine).

Esercizio 8

Che cos'è una *syscall*? Come è possibile aggiungere (in linea di massima) una nuova *syscall* in un OS previsto di tale meccanismo?

Soluzione Una *syscall* è una chiamata diretta al sistema operativo da parte di un processo user-level - e.g. quando viene invocata la funzione `printf`.

Nel caso si volesse aggiungere una nuova syscall, essa, una volta definita, va registrata nel sistema e aggiunta al vettore delle syscall del sistema operativo, specificandone il numero di argomenti (ed il loro preciso ordine).

Esercizio 10

Cos'è la directory `/proc` e cosa contiene al suo interno?

Soluzione La directory `proc` è un *virtual filesystem*, poiché non contiene file reali, bensì runtime system information - e.g. memoria di sistema, configurazione hardware, etc. Molte delle utilities di sistema infatti, si riferiscono ai file contenuti in questa cartella - e.g. `lsmod` ≡ `cat /proc/modules`.

Esercizio 4

Cos'è una *Shared Memory*? Fornire un breve esempio del suo utilizzo.

Soluzione La *shared memory* è un meccanismo usato per permettere a processi diversi di comunicare tra loro (Interprocess Communication - IPC). In questo caso, viene riservata una porzione di memoria condivisa tra i vari processi, i quali potranno scambiarsi informazioni semplicemente scrivendo e leggendo in tale porzione di memoria. I processi sceglieranno la locazione di memoria ed la tipologia di dati; essi dovranno anche sincronizzarsi in modo da non operare contemporaneamente sugli stessi dati. La shared memory, per esempio, è molto utile nel caso di problemi *producer/consumer* - o analogamente *client/server*. In questo caso, infatti, sarà necessario istanziare un buffer condiviso da entrambi i processi, in modo che il produttore possa rendere disponibile ai consumatori ciò che ha prodotto.

message passing

Un altro metodo di IPC prevede l'uso di *messaggi* (Message Passing). In questo caso, i processi comuniceranno inviandosi dei messaggi che saranno gestiti tramite opportune syscall - creando quindi overhead. Questi ultimi sono da favorire nel caso in cui i dati da veicolare abbiano una dimensione ridotta o nel caso di architetture fortemente multicore - per evitare problemi di coerenza delle cache. Entrageditmbi i metodi sono riportati nella Figura 2.

Esercizio 5

Siano dati i seguenti algoritmi di *page replacement*:

- LRU
- FIFO
- Second-chance
- Optimal

Domande

(A) Ordinare gli algoritmi in base al loro *page-fault rate*.

(B) Evidenziare gli algoritmi che soffrono dell'anomalia di Belady.

Soluzione Partendo dall'algoritmo con le migliori performances in termini di *page-fault rate*, avremo:

#	Algorithm	Belady
1.	Optimal	no
2.	LRU	no
3.	Second Chance	sí*
4.	FIFO	sí*

Esercizio 7

Illustrare le differenze tra `fork()` e `vfork()`. Che cosa succede se a seguito di una `vfork` non viene fatta immediatamente una `exec()`?

Soluzione `vfork()` è progettata come variante più efficiente della `fork()`, specializzata nel caso l'istruzione immediatamente successiva alla `fork()`, nel processo figlio sia una `exec()`. A differenza della `fork()`, la `vfork()` non replica la memoria del processo padre. Se la prima istruzione eseguita è una `exec()`, l'operazione di copia non è necessaria in quanto l'immagine del processo figlio verrà sovrascritta dalla `exec()`. Non chiamare la `exec()` dopo una `vfork()` genera comportamenti non specificati, in quanto la memoria del padre non è replicata.

Esercizio 9

(A) Illustrare in modo conciso e preciso il meccanismo di *context switch*, avvalendosi di semplici illustrazioni.

(B) E' possibile implementare ugualmente un multitasking preemptive su una CPU priva di MMU? Motivare in modo sintetico la risposta.

Soluzione Per poter rimuovere dall'esecuzione un processo P_0 e quindi eseguire un nuovo processo P_1 , il SO deve salvare lo stato corrente del processo P_0 in modo da poter ripristinare l'esecuzione dello stesso in un secondo momento. Le informazioni di un processo sono contenute nel suo Process Control Block (PCB). Quindi, il *context switch* può essere riassunto visivamente nella Figura 3.

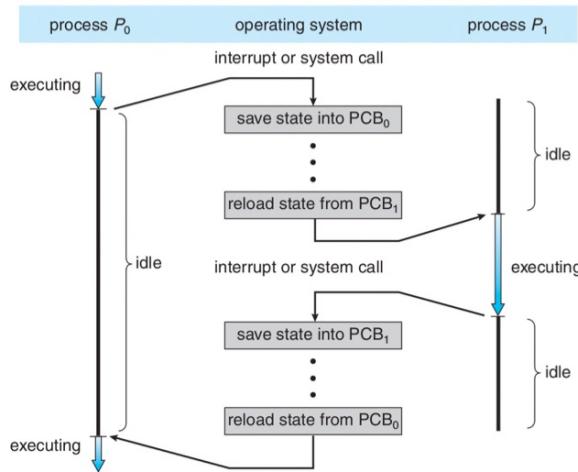


Figure 3: Esempio di *context switch*.

E' bene notare che il *context switch* è fonte di overhead a causa delle varie operazioni di preambolo e postambolo necessarie allo switch - e.g. salvare lo stato, blocco e riattivazione della pipeline di calcolo, svuotamento e ripopolamento della cache.

E' possibile implementare multitasking preemptive su una CPU priva di MMU poiché essa non è necessaria per il context switch.

Esercizio 3

A cosa serve la syscall `ioctl`? Come si puo configurare la comunicazione con devices a caratteri e seriali in Linux?

Soluzione La syscall `ioctl` permette di interagire con il driver di un device generico - e.g. una webcam. Tramite essa sarà possibile ricavare e settare i parametri di tale device - e.g. ricavare la risoluzione della webcam o settarne la tipologia di acquisizione dati.

Per configurare devices seriali a caratteri - e.g. terminali - è possibile usare le API racchiuse nella interfaccia `termios`. Tramite di essa, avremo accesso a tutte le informazioni relative al device - e.g. baudrate, echo,

Esercizio 4

Spiegare brevemente cose il Direct Memory Access (DMA) e quando viene usato.

Soluzione Alcuni device necessitano di trasferire grandi quantità di dati a frequenze elevate. Effettuare tali trasferimenti tramite il protocollo genericamente usato per altri tipi di periferiche richiederebbe l'intervento della CPU per trasferire un byte alla volta i dati - tramite un processo chiamato Programmed I/O (PIO). Ciò risulterebbe in un overhead ingestibile per l'intera macchina, consumando inutilmente la CPU. Per consentire il corretto funzionamento di tali device evitando gli svantaggi del **PIO**, tali periferiche possono avvalersi di controlleri dedicati che effettuano **DMA**, cioè andando a scrivere direttamente sul bus di memoria. La CPU sarà incaricata soltanto di "validare" tale trasferimento e poi sarà di nuovo libera di eseguire altri task.

Questo tipo di periferiche sono molto comuni ai giorni nostri e sono usate nella maggior parte dei dispositivi elettronici - pc, smartphones, servers, Esempi di periferica che si avvalgono di controller **DMA** sono videocamere, dischi, schede video, schede audio, ecc.

Esercizio 6

Si consideri un file consistente di 60 blocchi e che il suo File Control Block (FCB) sia già in memoria. Siano dati due File System (FS) gestiti rispettivamente tramite allocazione a lista concatenata - Linked List Allocation (LLA) - e allocazione contigua - Contiguous Allocation (CA). Si assuma che nel caso di CA, eventuale spazio per estendere il file sia disponibile solo alla fine dello stesso - non all'inizio.

Domanda Si calcolino le operazioni di I/O su disco necessarie per eseguire le seguenti azioni in entrambi i FS:

- Rimozione di un blocco all'inizio del file
- Rimozione di un blocco ad un terzo del file
- Rimozione di un blocco alla fine del file

Soluzione Nel caso di LLA per effettuare una operazione bisognerà scorrere la lista fino al blocco in questione. Nell'implementazione con CA i blocchi sono allocati in maniera sequenziale sul disco, quindi ogni volta bisognerà ricopiare tutti i blocchi in modo da "compattarli". Date queste premesse, i risultati sono i seguenti:

1. LLA: 1 IO-ops; CA: 118 IO-ops
2. LLA: 22 IO-ops; CA: 78 IO-ops
3. LLA: 60 IO-ops; CA: 0 IO-ops

Si noti che, nell'implementazione con CA la rimozione di un file in posizione $n = 20$ richiede di spostare tutti i blocchi posteriori a quello in questione. Si ricorda che per spostare un blocco è necessario prima leggerlo (1 IO-ops) e poi scriverlo nella posizione giusta (1 IO-ops).