PROGRAMMAZIONE DI SISTEMA

Esercizi proposti e svolti di teoria
Tratti da compiti di esame

Gestione file system e IO (esercizi relativi a cap 12, 13 e 14)

1. Sia dato un file system in cui è possibile l'accesso concorrente di più processi a uno stesso file.

   Quali operazioni deve svolgere il sistema operativo per realizzare una open()? E una close()?

open()
la open riceve come parametro il nome del file, deve aprirlo nella modalità richiesta e ritornare il corrispondente file descriptor (o handle/puntatore, a seconda del sistema operativo)

1) il nome del file viene cercato utilizzando la struttura dati (in memoria e/o disco) basata sui direttori
2) se ha successo, la ricerca ritorna il File Control Block (copia nella system-wide open-file table, già presente in precedenza oppure generate ora).
3) viene creata una nuova entry nella per-process open-file table (che fa riferimento al FCB nella system-wide), di cui si ritorna il puntatore/handle o indice (file descriptor)

close()
la close deve chiudere il file, eliminando le relative entry nelle tabelle dei file aperti, qualora non più utili

1) la corrispondente entry nella per-process open-file table viene cancellata
2) Si decrementa il contatore dei riferimenti nella tabella system-wide, se questo diventa 0, si cancella l'entry anche da questa

   A quali strutture dati (tabelle di gestione file) si deve fare accesso per realizzare una read() e/o una write()?

Si fa accesso a

1) per-process open-file table, dove, nella entry relativa, si trovano il puntatore alla posizione di lettura/scrittura nel file e la modalità di accesso
2) system-wide open-file table. Vi si deve accedere, tra l'altro, per fare la conversione da indirizzo logico nel file a fisico (numero di blocco e offset nel blocco)

   Che cosa sono la *system-wide open-file table* e la *per-process open-file table*? Perché in un sistema operativo possono essere necessarie entrambe anziché solo una delle due?

Le tabelle contengono tutte le informazioni necessarie in memoria per gestire correttamente i file. Sono necessarie entrambe perché un file può essere aperto contemporaneamente più volte (in modalità diverse), sia nel contesto di un singolo processo (eventualmente con più thread) che da parte di più processi.
Le informazioni comuni, nella system-wide, includono una copia del FCB ed eventuali primitive per gestire/sincronizzare accessi condivisi e altro.
Le informazioni nella per-process includono il puntatore al file e la modalità di accesso (es. lettura/scrittura)

2. Si consideri un buffer di kernel utilizzato come passaggio per i blocchi di un file in transito tra disco e memoria user: i dati che debbono essere trasferiti (ad esempio mediante una `read(fd,addr,size)`, con `addr` e `size` che determinano la destinazione in memoria user) fanno un passaggio in più: da disco a buffer kernel (per una dimensione `size`) e successivamente da buffer kernel alla vera destinazione `addr`.
   Perché può essere vantaggioso il buffer kernel, pur costringendo a un passaggio in più in RAM?

R1	*Il vantaggio principale è legato al fatto di aver disaccoppiato il lavoro sulla memoria USER rispetto all'accesso a disco. In sistemi con paginazione e/o swapping è quindi possiblle fare swap out di un intero processo utente in attesa di I/O, oppure di una pagina coinvolta in tale IO, in quanto il buffer user non viene "bloccato" dall'attesa di I/O. Il vantaggio del doppio buffer rispetto al singolo è poi quello di realizzare una concorrenza di tipo pipelining, in cui si può trasferire da memoria kernel a user e al tempo stesso da disco a memoria kernel.* *Un ulteriore vantaggio del buffer kernel può essere la funzione di cache, cioè il buffer kernel già riempito in anticipo, per evitare che il processo vada in attesa di I/O.* *ATTENZIONE: Si noti che il vantaggio NON è quello di evitare al processo user di fare l'IO. Il processo USER NON NE HA I PRIVILEGI. Sia con buffer che senza buffer, l'I/O viene effettuato da una system call, mediante un opportuno driver (di KERNEL): in un caso il driver lavora su memoria user (e la blocca) nell'altro caso su buffer kernel.* *Non ha senso neppure in questo contesto parlare di interrupt, DMA; CPU o altro. Valgono considerazioni simili alle precedenti.*

In un sistema con paginazione, il parametro `size` può essere arbitrario, oppure deve essere un multiplo della dimensione di blocco o di una pagina?

R2	Il parametro `size` è arbitrario, in quanto la `read` è una funzione al livello user, che non ha alcuna diretta dipendenza dalle strategie di paginazione. Non solo `size` può essere arbitrario, ma anche l'indirizzo di partenza `addr` non è necessariamente allineato a un inizio di pagina.

Si supponga di usare un "doppio" buffer.
[spiegazione (si spiega la tecnica per la read, la write sarà duale): mentre uno dei due buffer (detto "kernel") è coinvolto in trasferimento da disco, l'altro buffer (detto "user") può essere usato (purché caricato in precedenza da dati provenienti da disco) per trasferire dati alla destinazione in memoria user. Dopo ogni operazione si scambiano i ruoli dei due buffer].
Si dica, supponendo di voler leggere sequenzialmente un file da 200KB, quanti Byte passano complessivamente sul bus dati nei due casi *(singolo e doppio buffer)*. Per le letture da disco si consideri di usare DMA. Nel caso di doppio buffer si dimezza il numero di byte trasferiti rispetto al singolo buffer *(motivare la risposta)*?

R3	Il doppio buffer può solamente velocizzare le operazioni (grazie a un superiore livello di parallelismo), ma il numero di byte gestiti è lo stesso nei due casi (singolo e doppio buffer: semplicemente cambia la collocazione dei dati nel buffer kernel). Complessivamente, i 200KB passano una volta sul bus durante il trasferimento in DMA da disco a buffer kernel. Il passaggio da buffer kernel a memoria user è invece una copia da RAM a RAM (indirizzi sorgente e destinazione diversi. Nel caso di trasferimento gestito dalla CPU, i dati passano due volte sul bus dati (da RAM a CPU e da CPU a BUS). Si tratta quindi di leggere da RAM 200KB e di scriverne altrettanti. In totale transitano 600KB (200KB + 2*200KB). *NOTA (fuori programma): si noti che, qualora si utilizzasse il DMA per il trasferimento da RAM e RAM, si potrebbe fare il transito con un solo passaggio sul bus, ma solo a patto di utilizzare DMA controller di tipo fetch-and-deposit, che permettono (in due cicli di bus, uno di lettura e uno di scrittura) di gestire in modo unitario operazioni RAM-RAM.*

3. Si considerino i due tipi di sincronizzazione, relativi ad operazioni di I/O: sincrono e asincrono. Si spieghino le principali differenze tra gli I/O dei due tipi. Si definisca poi I/O bloccante e non bloccante: si tratta di sinonimi di asincrono e sincrono? Ci sono differenze (tra bloccante/non-bloccante e sincrono/asincrono)? (se no, perché, se si, quali?).

R1	In modo molto sintetico, si può affermare che • IO bloccante e sincrono siano equivalenti: il processo che effettua IO ne attende il completamento, in stato di wait. • IO non bloccante permette al processo di proseguire, IO asincrono è di fatto NON bloccante, con l'aggiunta di tecniche che permettano di gestire (successivamente) il completamento dell'IO. Tali tecniche sono: funzioni di `wait` (dipendono dal sistema operativo) che consentano di attendere il completamento dell'IO, oppure funzioni di tipo *"callback",* richiamate in modo automatico dal sistema, al completamento dell'IO (attenzione: sono funzioni che vanno scritte dallo user)

Un I/O sincrono può essere effettuato in polling oppure è necessario effettuarlo in interrupt? (si risponda alla stessa domanda per in caso dell'I/O asincrono).

R2	Polling e interrupt sono due modalità diverse per gestire un dispositivo di IO. Ovviamente il polling risulta meno efficiente, con rare eccezioni. Si tratta tuttavia di un problema interno ai driver (moduli del Kernel) e quindi indipendenti dal processo user. In sostanza, realizzare una system call read/write in modo sincrono o asincrono, può esser fatto con driver che lavorino sia in modo polling che interrupt,

In quale modo può un processo beneficiare di un I/O asincrono ? E' possibile, nel caso di I/O asincrono, scrivere un programma con istruzioni, successive all'I/O, che dipendano dai dati coinvolti (ad es. letti) durante l'I/O ?

R3	Il processo può beneficiare in quanto può eseguire altre istruzioni mentre l'IO è in corso. Si tratta quindi di una possibile forma di concorrenza. Il processo non può, durante l'IO, utilizzare i dati coinvolti. Se deve farlo, occorre sincronizzarsi (e aspettare) sulla relativa (dell'IO asincrono) operazione di wait.

4. Si consideri il problema della gestione di una richiesta di IO a blocchi realizzato mediante DMA. Che cosa si intende, in questo contesto, con il termine "*cycle stealing*"? Perché il trasferimento in DMA è vantaggioso rispetto all'IO programmato? Immaginando di dover trasferire 40KB di dati da disco a memoria RAM, quanti Byte transitano sul bus dati della RAM nei due casi (trasferimento in DMA e IO programmato)? (motivare la risposta). Qualora si voglia usare per l'IO un buffer in memoria Kernel, in che cosa il doppio buffer si differenzia dal singolo buffer e per quale motivo può risultare vantaggioso?

R	*(Si mettono in evidenza, in modo sintetico, gli aspetti principali che una risposta corretta dovrebbe contenere)*
	
	• Il "cycle stealing" è la sottrazione (con attesa per la CPU) di cicli di BUS alla CPU, mentre il DMA ha il controllo dei BUS di accesso alla RAM
	• Per due motivi principali:
	  ○ Perché i trasferimenti mediante DMA fanno passare i dati "direttamente" tra IO e RAM (senza passare dalla CPU, con un numero doppio di operazioni)
	  ○ Perché mentre si trasferiscono dati in DMA la CPU può fare altro (aumentando il grado di multiprogrammazione)
	• Transitano 40KB in DMA e 80KB (i 40KB passano due volte, in quanto debbono transitare nella CPU) nel caso di IO programmato.
	• Il doppio buffer è una tecnica nella quale mentre un buffer viene scritto l'altro (riempito in precedenza) può essere letto in parallelo. Permette quindi una forma di pipelining. Con iul buffer singolo uno dei due (chi scrive o chi legge) deve invece attendere il completamento dell'altra operazione. *(ATTENZIONE; si parla qui di doppio buffer di kernel, non di dualità buffer kernel e buffer user)*