Programmazione di Sistema
11 settembre 2018 (teoria)

Si prega di rispondere in maniera leggibile, descrivendo i passaggi e i risultati intermedi. Non è possibile consultare alcun materiale. Durata della prova: 70 minuti. Sufficienza con punteggio >= 8. Le due parti possono essere sostenute in appelli diversi. La presenza a una delle due parti annulla automaticamente un'eventuale sufficienza già ottenuta (per la stessa parte): viene intesa come rifiuto del voto precedente.

1. *(4 punti)* Si consideri la seguente sequenza di riferimenti in memoria nel caso di un programma di 4K parole in cui, per ogni accesso (indirizzi in esadecimale), si indica se si tratta di lettura (R) o scrittura (W): W 3A1, R 3F5, R A64, W BD3, W 57E, R A08, R B85, W 3A0, R A1A, W A36, R B20, R 734, R AB8, R C4E, W B64.
   Si determini la stringa dei riferimenti a pagine, supponendo che la loro dimensione sia di 512 parole. Si utilizzi un algoritmo di sostituzione pagine di tipo Enhanced Second-Chance, per il quale, al bit di riferimento (da inizializzare a 0 in corrispondenza al primo accesso a una nuova pagina dopo il relativo page fault), si unisce il bit di modifica (modify bit). Si assuma che una pagina venga sempre modificata in corrispondenza a una scrittura (write), che siano disponibili 3 frame e che l'algoritmo operi con il criterio seguente: dato il puntatore alla pagina corrente (secondo la strategia FIFO) si fa un primo giro, senza modificare il reference bit, sulle pagine per localizzare la vittima (l'ordine di priorità è (reference,modify): (0,0), (0,1), (1,0), (1,1)); una volta determinata la vittima, si fa un secondo giro per azzerare i reference bit delle pagine "salvate" (comprese tra la posizione di partenza e la vittima).

   Determinare quali e quanti page fault (accessi a pagine non presenti nel resident set) si verificheranno. Si richiede la visualizzazione (dopo ogni accesso) del resident set, indicando per ogni frame i bit di riferimento e modifica. Si numerino le pagine a partire da 0.

   Utilizzare, per questa domanda, lo schema seguente per svolgere l'esercizio, indicando nella prima riga la stringa dei riferimenti a pagine (rappresentate a scelta in esadecimale o decimale), nella seconda Read o Write, nelle tre successive (che rappresentano i 3 frame del resident set), le pagine allocate nei corrispondenti frame, indicando per ognuna i bit (reference,modify). Indicare inoltre (sottolineandola, circolettandola o ponendo una freccia), quale pagina si trova in testa al FIFO.
   Nell'ultima riga si indichi la presenza o meno di un Page Fault.

   R. Si noti che ogni cifra esadecimale rappresenta 4 bit. Due cifre sono 8 bit. La divisione per 512 (2^9) si ottiene quindi eliminando le due cifre esadecimali meno significative e dividendo ulteriormente per 2: in pratica si divide per 2 la prima delle 3 cifre esadecimali.

Riferimenti	1	1	5	5	2	5	5	1	5	5	5	3	5	6	5
Read/write	W	R	R	W	W	R	R	W	R	W	R	R	R	R	W
Resident Set	1_{01}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{11}	1_{01}	1_{01}	1_{01}	1_{01}
			5_{00}	5_{11}	5_{11}	5_{11}	5_{11}	5_{11}	5_{11}	5_{11}	5_{11}	5_{01}	5_{11}	5_{01}	5_{11}
					2_{01}	2_{01}	2_{01}	2_{01}	2_{01}	2_{01}	2_{01}	3_{00}	3_{00}	6_{00}	6_{00}
Page Fault	X		X		X							X		X	

   Numero totale di page fault: ……5…….

2. *(3 punti)* Sia dato un file system Unix, basato su inode aventi 13 puntatori (10 diretti, 1 indiretto singolo 1 doppio e 1 triplo). I puntatori hanno dimensione di 32 bit e i blocchi hanno dimensione 2KB. Si sa che il file system contiene 1000 file di dimensione media 15MB e che la frammentazione interna totale è di 1MB, si calcoli il massimo numero possibile di file con indice indiretto triplo e con indice indiretto doppio che possono essere presenti nel file system.

R1	***Osservazioni*** *Un blocco indice contiene 2KB/4B = 512 puntatori.* *La dimensione (netta, indipendente dal tipo di file system) complessiva dei file è 15MB * 1000 = 15000MB (*= 15GB) *L'occupazione (questa dipende dal file system e tiene conto dei blocchi effettivamente usati), tenendo conto della frammentazione interna, è 15001MB = 15001*512 blocchi*

	Calcolo massimo per N2/N3 *(numero file* con indice indiretto doppio/triplo) Per calcolare il numero massimo occorre considerare l'occupazione minima. *Si indicano con MIN_2 e MIN_3 le occupazioni minime dei due tipi di file:* MIN_2 = (10 + 512 + 1) blocchi MIN_3 = (10 + 512 + 512^2 +1) blocchi N2 = $\lfloor$15001 * 512 / 523$\rfloor$ = 14685 N3 = $\lfloor$15001 * 512 / (523 + 512^2)$\rfloor$ = 29

Quale è il massimo numero possibile di file privi di frammentazione interna *(motivare la risposta)*?

R2	***Osservazioni*** *La frammentazione interna complessiva è nota (1MB). Il massimo numero di file a frammentazione interna nulla ($N0_{MAX}$) si ottiene ripartendo la frammentazione sul minimo numero N_{min} di file, aventi quindi frammentazione massima di 1 blocco – 1 Byte ciascuno* ***Calcolo*** *N_{min} = $\lceil$1MB / (2KB-1B)$\rceil$ = $\lceil$1MB / 2047$\rceil$= 513* *$N0_{MAX}$ = 1000 – 513 = 487*

Si sa che per un file "a.dat" si utilizzano 2000 blocchi di indice. Quanti blocchi di dato occupa e quale dimensione ha il file "a.dat" *(qualora non siano possibili valori univoci, si determinino valori minimi e massimi)*?

R3	***Osservazioni*** *Si osservi che il numero di blocchi di indice per un file che occupa totalmente il secondo livello è 1 (singolo) + 1 (doppio primo liv.) + 512 (doppio secondo liv).* *Per 2000 blocchi indice occorre quindi indice indiretto triplo, tale da utilizzare 2000 – 514 = 1486 blocchi indice* ***Calcolo*** *Occorre calcolare quanti blocchi indice triplo a secondo e terzo livello (usati solo in parte) ci sono (a primo livello ce n'è uno solo):* *$\lceil$1486/512$\rceil$= 3 => 1 (triplo, I liv.) + 3 (triplo, II liv.) + 1482 (triplo, III liv.)* ***Occupazione:*** *numero blocchi dato NBL (i valori minimo e massimo dipendono da quanti indici ci sono nell'ultimo blocco indici (min 1 – max 512)):* *NBL_{min} = 10 (diretti) + 512 (singolo) + 512*512 (doppio) + 1481*512 + 1 (triplo) = 1020939* *NBL_{MAX} = NBL_{min} + 511 = 1021450* ***Dimensione:*** *La minima si ottiene da assumendo frammentazione interna massima (2047 byte) – La massima si ottiene da assumendo frammentazione 0.* *DIM_{min} = NBL_{min} * 2KB – 2047B* *DIM_{MAX} = NBL_{MAX} * 2KB*

3. *(3 punti) (le riposte si/no vanno motivate)* 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 e buffer 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.*

4. *(4 punti)* Sia dato un sistema operativo OS161.

4.a) Si riporta in figura il codice relativo alle funzioni `thread_exit` e `thread_destroy` (incompleto):

```
thread_exit(void) {
  struct thread *cur = curthread;

  /* Detach from our process.
     You might need to move this action
     around, depending on how
     your wait/exit works. */
  proc_remthread(cur);
  /* Make sure we *are* detached
     (move this only if you're sure!) */
  KASSERT(cur->t_proc == NULL);
  /* Check the stack guard band. */
  thread_checkstack(cur);
  /* Interrupts off on this processor */
  splhigh();
  thread_switch(S_ZOMBIE, NULL, NULL);

  panic("braaaaaiiiiiiiiiinsssss\n");
}
```

```
thread_destroy(struct thread *thread) {
  KASSERT(thread != curthread);
  KASSERT(thread->t_state != S_RUN);

  /* If you add things to struct thread,
     be sure to clean them up
     either here or in thread_exit().
     (And not both...) */

  /* Thread subsystem fields */
  KASSERT(thread->t_proc == NULL);
  if (thread->t_stack != NULL) {
    kfree(thread->t_stack);
  }
  ...
  kfree(thread->t_name);
  kfree(thread);
}
```

Si risponda alle seguenti domande *(tutte le eventuali risposte si/no vanno motivate):*

- Perché la `thread_exit` non ha parametri mentre la `thread_destroy` si?

R	*La* `thread_exit` *fa terminare in thread corrente, mantre la* `thread_destroy` *termina (deallocandone la struttura dati) un altro thread, di cui deve ricevere il puntatore.*

- E' possibile una delle seguenti sequenze di istruzioni?

`thread_exit();` `thread_destroy(thread);`	`thread_destroy(thread);` `thread_exit();`

R	*La prima non è possibile in quanto dopo `thread_exit` non può essere eseguito nulla: il thread termina. La seconda è possibile, a patto che, ovviamente, `thread` (parametro della `thread_destroy`) non sia il thread corrente: il thread corrente, quindi, ne distrugge un altro e poi termina.*

- Perché la `thread_exit` termina con una chiamata a `panic`, mentre la `thread_destroy` no?

R	*La risposta è strettamente connessa alla precedente: `thread_exit` fa terminare il thread portandolo mediante `thread_switch`, nello stato `S_ZOMBIE`. Arrivare ad eseguire l'istruzione `panic`, significherebbe un evidente errore (da parte del programmatore!). La `thread_destroy` invece, terminando la struttura dati di un altro thread, può uscire regolarmente dalla funzione e continuare l'esecuzione.*

- Perché è possibile *(si veda il commento)* dover spostare la chiamata a `proc_remthread(cur)`, effettuata in `thread_exit`, a seconda di come si realizza la `wait/exit`?

R	*Perché le system call `exit` e `wait` debbono chiudere un processo, liberandone l'address space, ad esempio chiamando `proc_destroy`. Ciò richiede che preventivamente tutti i thread siano "staccati" dal loro processo. Siccome l'azione di staccare un thread dal relativo processo, così come fatto nella `thread_exit`, impedisce che vengano eseguite altre operazioni (rilascio di address space o segnalazione a processo in wait) dopo (perché il thread muore), è possibile che la `thread_exit` vada modificata (ad esempio accettando l'exit di un thread già staccato dal relativo processo).*

- Perché si fa l'asserzione `KASSERT(thread->t_state != S_RUN)`? Quale dovrebbe essere lo stato di `thread`?

R	*Perché non si può fare thread_destroy su un processo in esecuzione (si tratterebbe di un errore del programmatore). Lo stato dovrebbe essere `S_ZOMBIE`.*

- L'istruzione `kfree(thread->t_stack)` restituisce la RAM allocata per lo stack dell'address space associato al thread?

R	*NO. Restituisce l'stack di kernel del thread. Il rilascio dello stack* dell'address space (non fatto nella versione base dumbvm) andrebbe fatto in ogni caso nella `as_destroy` (per l'intero processo) e non nella `thread_destroy`.

4.b) Si vuol realizzare il supporto per la system call `read`, di cui si fornisce il prototipo:

```
ssize_t read(int filehandle, void *buf, size_t size);
```

Si scriva a tale scopo (e se ne spieghi in breve il contenuto) una funzione `sys_read`, richiamabile dalla `syscall`. Si richiede, in particolare, che la funzione permetta input da un file arbitrario (QUINDI NON LIMITATA a stdin). Una eventuale funzione di accesso a tabella dei file aperti può essere semplicemente chiamata (senza doverla realizzare).

R	*Ci si limita ad una versione estremamente semplificata, che affronta il problema del input da file generico (non stdin, si faccia riferimento all'ultimo laboratorio). La versione proposta, ad esempio, acquisisce direttamente in memoria user (senza usare buffer kernel e funzione `copyout`). Si utilizzano due funzioni che andrebbero realizzate: una per reperire il puntatore a `vnode` a partire dal file descriptor (un intero) e una per unizializzare UIO a livello user. Si indica poi (senza realizzarla) l'eventualita' che l'IO sia da console (il caso dovrebbe essere limitato a stdin, a patto che non sia stato re-diretto su file).*

```
int sys_read(int filehandle, userptr_t buf, size_t size) {
  struct vnode *v;
  int result = 0;

  // la funzione va implementata gestendo opportunamente in open/close
  // una tabella (un vettore) di puntatori a vnode per i file aperti
```

```
  v = getVnodeFromFileDescriptor(fd);
  if (/* è la console */) {
    // gestione standard input, ad esempio mediante kgets
  }
  else {
    struct iovec iov;
    struct uio u;
    // funzione da realizzare (simile a uio_kinit, ma per spazio user)
    uio_uinit(&iov, &u, buf, size, 0, UIO_READ);
    result = VOP_READ(v, &u);
  }
  return result;
}
```