Programmazione di Sistema
6 settembre 2016 (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. Prima e seconda parte 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)* Siano dati i frammenti di programma (A e B) rappresentati, che calcolano nel vettore V, rispettivamente, le medie per riga dei dati nella matrice M (il programma B, inoltre, normalizza la matrice M):

A	B

`float M[1024][1024],V[1024];`
`...`
`for (i=0; i<1024; i++) {`
`  V[i] = 0.0;`
`  for (j=0; j<1024; j++) {`
`  V[i] += M[i][j];`
`  }`
`  V[i] = V[i]/1024;`
`}`
`...`
	`float M[1024][1024],V[1024];`
	`...`
	`for (i=1023; i>=0; i-- {`
	`  V[i] = 0.0;`
	`  for (j=1023; j>=0; j--) {`
	`  V[i] += M[i][j];`
	`  }`
	`  V[i] = V[i]/1024;`
	`  for (j=0; j<1024; j++) {`
	`  M[i][j] -= V[i];`
	`  }`
	`}`
	`...`

I programmi eseguibili generati da tali sorgenti sono eseguiti in un sistema con memoria virtuale gestita mediante paginazione, con **pagine di 2Kbyte**, utilizzando come politica di sostituzione pagine la **LRU** (Least Recently Used). Si sa che un `float` ha dimensione **32 bit** e che le istruzioni in codice macchina, corrispondenti al frammento di programma visualizzato, sono contenute in una sola pagina. Si supponga i che `M` e `V` siano allocati ad indirizzi logici contigui (prima `M`, poi `V`), a partire dall'indirizzo logico D45047400. La matrice `M` è allocata secondo la strategia "row major", cioè per righe (prima riga, seguita da seconda riga, …).

- Quante pagine (e frame) sono necessarie per contenere la matrice e il vettore ?

R	V: 1024*4B / 2KB = 2 pagine M: 1024*1024*4B / 2KB = 2K pagine Tenendo conto che le strutture dati iniziano a un indirizzo interno a una pagina, occorre complessivamente una pagina in più. M viene allocata su 2k+1 pagine, V su 3 pagine, di cui la prima comune a M.

- Ipotizzando che le variabili i, j siano allocate in registri della CPU, quanti accessi in memoria (in lettura e scrittura) fa il programma proposto, per accedere a dati (non vanno conteggiati gli accessi a istruzioni) ?

R	A	`V[i] = 0.0;  // 1 scrittura(W)`
		`V[i] += M[i][j];  // 2 letture(R), 1 scrittura(W)`
		`V[i] = V[i]/1024; // 1 lettura(R), 1 scrittura(W)`
		Totale:
		1024 * (1W + 1024 * (2R+1W) + 1R + 1W) ≈ 2M R + 1M W = 3M accessi

	B	`V[i] = 0.0;  // 1 scrittura(W)`
		`V[i] += M[i][j];  // 2 letture(R), 1 scrittura(W)`
		`V[i] = V[i]/1024; // 1 lettura(R), 1 scrittura(W)`
		`M[i][j] -= V[i];  // 2 letture(R), 1 scrittura(W)`
		Totale:
		1024 * (1W + 1024 * (2R+1W) + 1R + 1W + 1024 * (2R+1W)) ≈ 4M R + 2M W = 6M accessi

- Detto N_T il numero totale di accessi a dati in memoria, ed N_L il numero di accessi a dati nella stessa pagina di uno dei precedenti 2 accessi, si definisca come **località** del programma per i dati il numero $L = N_L/N_T$. Si calcoli la località del programma proposto.

R1	E' sufficiente considerare le istruzioni all'interno dei loop annidati, eseguite 1024*1024 volte e quindi dominanti, per 3 ordini di grandezza, rispetto alle altre, eseguite 1024 volte (si veda il punto precedente).	
	A	$N_T/1024 = 1024*3$ Si ricorda che ci sono 3 accessi per iterazione, due a V e uno a M. Gli accessi a V (nell'ambito di una riga) sono sempre nella stessa pagina del precedente (eccetto due volte, ma trascurabili, al livello di loop esterno=. Quelli a M sono sempre diversi, in quanto i 2 precedenti sono su V). $N_L/1024 = 1024*3-1024$ $N_L/N_T = 2/3$
	B	$N_T/1024 = 1024*6$ Gli accessi a M sono diversi da uno degli ultimi 2 1024 volte nel primo for (come nel caso A) e 1 nel secondo for: infatti ognuno dei due accessi a M inizia dalla stessa pagina in cui termina il precedente. Quelli a V sono sempre nella stessa pagina, nel primo for, in una pagina diversa nel secondo for, in quanto i 2 precedenti sono su M. $N_L/1024 = 1024*6-1-1024-1024$ $N_L/N_T = 2/3$

R2	Si è ritenuta accettabile una seconda interpretazione, nella quale gli *"ultimi 2 riferimenti"* sono intesi come relativi ad M (quando si considera lettura/scrittura su M) e a V.	
	A	$N_T/1024 = 1024*3$ Gli accessi a M sono diversi da uno degli ultimi 2 accessi a M solo 2 volte per riga. Quelli a V sono sempre nella stessa pagina del precedente. $N_L/1024 = 1024*3-2$ $N_L/N_T = 3070/3072 \approx 0{,}9993$
	B	$N_T/1024 = 1024*6$ Gli accessi a M sono diversi da uno degli ultimi 2 accessi a M solo 2 volte per riga (vedere R1). Quelli a V sono sempre nella stessa pagina del precedente. $N_L/1024 = 1024*6-2$ $N_L/N_T = 6142/6144 \approx 0{,}9997$

R3	Si è accettata (pur se scorretta) una ulteriore interpretazione, in cui si considera un solo accesso a M e uno a V per ogni iterazione.	
	A	$N_T/1024 = 1024*2$ Gli accessi a M sono diversi da uno degli ultimi 2 accessi a M solo 2 volte per riga. Quelli a V sono sempre nella stessa pagina del precedente. $N_L/1024 = 1024*2-2$ $N_L/N_T = 2046/2048 \approx 0{,}998$
	B	$N_T/1024 = 1024*4$ Gli accessi a M sono diversi da uno degli ultimi 2 accessi a M solo 2 volte per riga (vedere R1). Quelli a V sono sempre nella stessa pagina del precedente. $N_L/1024 = 1024*4-2$ $N_L/N_T = 4094/4094 \approx 0{,}999$

- Calcolare il numero di page fault generati dal programma proposto, supponendo che siano allocati per esso **8 frame**, di cui uno utilizzato (già all'inizio dell'esecuzione) per le istruzioni. (motivare la risposta)

R	Sia per il caso A che per B, si può osservare che una pagina non viene mai caricata in un frame due volte. Per cui i PF sono 2K+2 (2K+3 se si tiene conto dell'indirizzo di inizio) sia per A che per B.

(I programmi A e B vanno trattati come casi separati, cioè si tratta di due esecuzioni/esperimenti distinte/i)

2. *(3 punti)* Sia dato un file system su una partizione di dimensione complessiva 400GB. Il file system utilizza inode aventi 13 indici (10 diretti, 1 indiretto singolo, 1 doppio e 1 triplo). Gli indici (nei blocchi indice) hanno dimensione 32 bit e i blocchi su disco 8 KB. I blocchi liberi sono gestiti mediante bitmap, contenente 1 bit per ognuno dei blocchi (liberi o occupati, denotati rispettivamente, da 0 e 1) su disco. I 400GB della partizione contengono, oltre a blocchi dato e indice, la bitmap e una struttura dati per direttori e FCB, alla quale sono riservati 4GB.

- Quanti blocchi sono necessari per la bitmap ?

R	Si ipotizza di utilizzare la bitmap per rappresentare uno spazio di 400GB-4GB (si è considerata accettabile la bitmap per tutti i 400GB, oppure soluzioni, come pure lo scorporo dello spazio per la bitmap stessa) Num max di blocchi utilizzabili = (400GB-4GB)/8KB = 49.5M La bitmap ha 1 bit per ogni blocco su disco. |bitmap| = 49.5Mbit = 49.5M/(8*8K) blocchi = 49.5 * 2^4 = 792 blocchi (800 se non si fossero sottratti i 4GB)

- Si calcolino il numero massimo di file che utilizzano indirizzamento indiretto doppio, che possono essere contemporaneamente presenti nel file system. Si tenga conto della dimensione effettiva disponibile su disco, nonché nel fatto che ogni file necessita di blocchi di dato e di indice.

R	Il numero massimo di file con indirizzamento indiretto doppio si ottiene in corrispondenza alla dimensione minima per tali file (cioè la dimensione massima per indici indiretti singoli), tenendo conto che 1 blocco contiene 8KB/4B = 2K indici $\|F_2\|_{min}$ = 10 bl. diretti + 1 bl. indice sing. + 2K bl. indiretti sing. + 2 bl. indice doppio + 1 bl. indiretto doppio $\|F_2\|_{min}$ = 2K + 14 = 2062 blocchi $N_{F2.max}$ = 49.5M / 2062 = 25171 Si sono accettate soluzioni che approssimano la dimensione minima a 2K blocchi e/o lo spazio a 50M blocchi, con $N_{F2.max}$ = 25K (o 24K, a seconda dei casi).

- Sia dato un file di 12 blocchi. Supponendo che i blocchi (per dati e indici) nella partizione, siano numerati a partire da 0, i blocchi di dato nel file sono, nell'ordine, 5, 7, 8, 4, 2, 10, 12, 24, 30, 25, 27, 15. Il blocco di indici di primo livello utilizzato è il 28. Si rappresentino (in binario ed esadecimale) i primi 4 byte della bitmap, nell'ipotesi che nessun altro blocco di indice < 32 sia allocato ad altri file.

R	La bitmap, (partendo dal bit 0) è: 00101101 10101001 00000000 11011010 In esadecimale: 2DA900DA Si potrebbe rappresentare la stessa informazione con il bit 0 a destra: 01011011 00000000 10010101 10110100 In esadecimale: 5B0095B4

3. *(3 punti)* Si descriva, nel contesto della gestione dei dispositivi di I/O, la differenze tra I/O sincrono e asincrono, bloccante e non bloccante. I/O non bloccante e asincrono sono equivalenti?

R	• Sincrono: il processo attende che l'operazione di I/O sia completata • Asincrono: il processo va avanti mentre l' I/O è in esecuzione. Esistono , necessario avere dei meccanismi di sincronizzazione fra processo e sottosistema di I/O, che permettono di sincronizzare eventuali operazioni per le quali è necessario che l'IO sia completato. Tali meccanismi possono basarsi su primitive di sincronizzazione che permettano al processo di attendere la notifica di terminazione o sull'utilizzo di callback. La callback è una funzione che il processo passa al sottosistema di I/O e che dovrà essere eseguita al termine dell'operazione di I/O. • Bloccante: il processo è sospeso fino a che l' I/O non è completato. In pratica in questo contesto è sinonimo di sincrono • Non Bloccante: la chiamata all' I/O ritorna senza attenderne il completamento. Si noti che una chiamata asincrona è una chiamata non bloccante con in più l'aggiunta dei suddetti meccanismi di sincronizzazione. I/O non bloccante e asincrono non sono equivalenti in quanto l'asincrono è non bloccante e in più prevede meccanismi di sincronizzazione e/o esecuzione di operazioni al termine dell'IO.

Siano date due funzioni di scrittura su dispositivi di I/O, una di tipo sincrono (`writeSynch`) e una di tipo asincrono (`writeAsynch`). Nel seguito sono proposte alcune chiamate alle funzioni:

```
buf_t *buf1;
…
for (i=0; i<n; i++) {
  writeAsynch(fp1, &buf1[i], sizeof(buf_t),…);
}
…
for (i=n-1; i>=0; i--) {
  buf1[i] = f(buf1[i]);
}
…
for (i=0; i<n; i++) {
  writeSynch(fp1, &buf1[i], sizeof(buf_t),…);
}
…
```

Il programma non è completo (si sono utilizzati i "…" per indicare parti mancanti). Le funzioni `writeSynch` e `writeAsynch` realizzano, rispettivamente, scritture sincrone/asincrone tra un file ed un buffer, di cui si forniscono il puntatore e la dimensione. Si dica se il programma può funzionare, nella forma proposta, e/o come va modificato, o a quali condizioni lo si può utilizzare, affinché le chiamate ad I/O funzionino correttamente. Si chiede, in particolare, di discutere la consistenza dei dati scritti su file. Se si scambiassero, nel programma, `writeSynch` e `writeAsynch` (ponendole, rispettivamente nel primo e nel terzo ciclo for), il programma sarebbe corretto (motivare la risposta) ?

R	Il programma nella forma proposta funziona correttamente solo se fra le chiamate asincrone e la lettura dal buffer (secondo ciclo for) si è sicuri che tutte le scritture asincrone siano terminate, il che può essere garantito solo utilizzando meccanismi di sincronizzazione: tra il primo e il secondo for vanno cioè inserite una o più chiamate a funzioni di attesa (che gli IO siano terminati. In caso contrario i dati potrebbero essere modificati nel secondo ciclo for prima di esser scritti su file, risultando quindi inconsistenti. Scambiando le funzioni writeSynch e writeAsynch l'esecuzione risulterebbe corretta, per la parte visualizzata. Questo a patto che, dopo la scrittura asincrona non ci siano altre operazioni di I/O che operino su tali dati prima che tutte le scritture asincrone siano complete.

4. *(4 punti)* Facendo riferimento al sistema operativo OS161:

a) Si descrivano le caratteristiche di semafori e locks evidenziandone le differenze.
Cosa sono i wait_channel? Per cosa sono utilizzati all'interno del sistema operativo?

Si riporta la funzione wchan_wakeone:

```
wchan_wakeone(struct wchan *wc, struct spinlock *lk) {
  struct thread *target;
  KASSERT(spinlock_do_i_hold(lk));
  target = threadlist_remhead(&wc->wc_threads);
  if (target == NULL)  { return; }
  thread_make_runnable(target, false);
}
```

Si motivi l'istruzione KASSERT(spinlock_do_i_hold(lk));
Perché si rimuove un thread dalla lista wc->wc_threads ?

R	• Locks e semafori sono meccanismi di sincronizzazione utilizzati per limitare l'accesso a risorse condivise. Un semaforo è caratterizzato da un numero intero che rappresenta la quantità di risorse disponibili, quando vale 0 non vi sono risorse disponibili. Attraverso le primitive di sincronizzazione wait e signal tali risorse sono assegnate o sottratte ad un processo e il numero viene adeguatamente incrementato o decrementato.Un lock può essere visto come un mutex, o caso particolare di semaforo in cui il numero massimo di risorse è pari ad 1 (semaforo binario). • Waitchannel: è una struttura del kernel su cui un thread è in attesa per ottenere l'accesso ad una determinate risorsa. Sono utilizzati per la realizzazione di lock, semafori e condition variable. È stato introdotto in OS161 2.0 per gestire la concorrenza. Il wchan può essere utilizzato solo dal thread che ha acquisito lo spinlock • L'istruzione KASSERT(…) non è strettamente necessaria per l'esecuzione del programma ma serve ad individuare errori del programmatore in quanto quando viene invocata la funzione in questione lo spinlock deve essere tassativamente posseduto • Per il comportamento della `wchan_wakeone` solo un thread deve essere svegliato. La lista serve in quanto più thread possono essere in attesa. Nel caso specifico si rimuove il primo della lista

b) Che differenza c'è in OS161 tra un indirizzo logico USER e uno KERNEL ? E' possibile che, in momenti diversi, un indirizzo logico KERNEL e un successivo indirizzo USER possano corrispondere allo stesso indirizzo fisico in RAM ? (rispondere, e motivare la risposta, sia per il caso dubmvm, versione base, che per il caso di modifica del kernel, tale da gestire la "restituzione" di memoria)

R	• L'indirizzamento logico USER e KERNEL differiscono per il range di indirizzi utilizzati. In particolare Gli indirizzi logici USER sono quelli che partono da 0X000000000 fino a 0x80000000 escluso (i primi 2GB di RAM). Gli indirizzi KERNEL sono i seguenti 2GB di RAM ossia da 0x80000000 a 0xffffffff. • Nel caso base dubmvm no perché la memoria non viene mai rilasciata. Nel caso di modifica al kernel, dipende dall'implementazione degli allocatori. In particolar,e se è stato implementato un allocatore comune (per vm e kmalloc), allora si può avere sovrapposizione (in quanto la memoria viene rilasciata e quindi lo spazio viene reso disponibile per una eventuale riassegnazione). In caso contrario, se sono implementati due allocatori differenti, che si suddividono a priori la memoria fisica disponibile e che gestiscono quindi spazi di indirizzamento fisico separati, la sovrapposizione non è possibile.

c) Perché in OS161, per gestire gli argomenti al main, è necessario creare una copia di argv e argc ? Dove va creata tale copia ? Perché non sono sufficienti puntatori alle copie originali di argv e argc, create dal monitor di sistema, a partire da una stringa di comando ?

R	• Perché le copie originali argc e argv create dal monitor di sistema sono allocate in memoria kernel mentre il processo che li utilizzerà risiede in memoria user • Le copie sono create in memoria user • Perché i puntatori punterebbero alla memoria kernel e il processo non ha diritto di accesso

(Solo per studenti con frequenza 2014/2015 o anteriore, in alternativa (se ritenuto opportuno) alla domanda 4.a su wait_channel)

4.a.bis) Sia dato il gestore di memoria dumbvm. Si dica quale ruolo svolgono le funzioni as_create, as_define_region e as_prepare_load. Quale di esse alloca la memoria fisica chiamando getppages ?

R	• as_create: alloca e inizializza la struttura dati addrspace • as_define_region: allinea l' indirizzo virtuale base e lo imposta all'interno della struttura address_space. Imposta anche il numero di pagine necessarie • as_preapare_load: utilizza la funzione getppages() per allocare la memoria fisica delle regioni. La quantità di memoria allocata dipende dal numero di pagine impostato con la funzione as_define_region(). Alloca anche la memoria per lo stack ed inizializza la memoria fisica