Programmazione di Sistema
25 giugno 2018 (teoria)

Nome: …….. Matricola: ……..

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. La risposta alla domanda 1 va scritta su questo foglio.

1. *(3 punti)* Si consideri la seguente sequenza di riferimenti a pagine in memoria: 577517111434123.
Si utilizzi un algoritmo di sostituzione pagine di tipo working-set (versione esatta) con finestra Δ = 3, assumendo che siano disponibili al massimo 3 frame. Determinare quali e quanti page fault (accessi a pagine non presenti nel resident set) e page out (rimozioni di pagine dal resident set) si verificheranno. Si richiede la visualizzazione (dopo ogni accesso) del resident set.

Si vuole inoltre definire una misura di località del programma svolto, basato sulla "Reuse Distance". La Reuse Distance al tempo T_i (RD_i), in cui si accede alla pagina P_i, viene definita come il numero di pagine (distinte e diverse da P_i) a cui si è fatto accesso a partire dal precedente accesso a P_i (assumendo, convenzionalmente, per il primo accesso a una pagina, il numero totale di pagine cui si è fatto accesso sino a quell'istante). Ad esempio, al tempo 3, in cui si fa il secondo accesso alla pagina 5, RD_3 = 1, in quanto tra i due accessi alla pagina 5 si è fatto accesso, due volte, a una sola pagina (7). Dati i vari RD_i, se ne calcoli il valor medio RD_{avg}. La località del programma svolto viene definita come L = 1 / (1+ RD_{avg}). Si calcolino i valori RD_i, RD_{avg} e L.

Utilizzare, per questa domanda, lo schema seguente per svolgere l'esercizio. Si sono già indicati i riferimenti e la Reuse Distance ai tempi 0 e 3.

R. Si noti che sono disponibili al massimo 3 frame, quindi la finestra Δ = 3 comprende l'accesso corrente al tempo T_i. La scelta del frame nel resident set è arbitraria, sono quindi possibili altre soluzioni, con frame permutati.

Riferimenti	5	7	7	5	1	7	1	1	1	4	3	4	1	2	3
Resident Set	5	5	5	5	5	5				4	4	4	4	4	3
		7	7	7	7	7	7	7			3	3	3	2	2
					1	1	1	1	1	1	1		1	1	1
Page Fault	x	x			x					x	x		x	x	x
Page Out							x		x			x		x	x
RD	0	1	0	1	2	2	1	0	0	3	4	1	2	5	3

Numero totale di page fault: 8….. di page out: 5… RD_{avg}: 25/15=5/3=1,67 L: 1/(1+5/3)=3/8=0,375

2. *(4 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 1KB. Si sa che il file system contiene N file con indice indiretto triplo, 10N con indice indiretto doppio e 100N altri (con indice indiretto singolo o diretto). Sapendo che i file occupano complessivamente (i blocchi di indice sono esclusi dal computo) 256 GB, si calcoli (non essendo nota la dimensione di ogni file) il massimo numero di file che possono essere presenti nel file system. E' possibile calcolare anche il numero minimo di file presenti? (Si motivi la risposta e, se possibile, si calcoli tale minimo). Sia dato, in questo file system, un file "d.txt" contenente 100000 record di dimensione variabile compresi tra 50 e 500 Byte (estremi inclusi). Si rappresenti l'organizzazione del file, calcolando quanti blocchi di dato e di indice sono necessari e/o sufficienti per rappresentarlo. Quale è la frammentazione interna di "d.txt"? (qualora non sia possibile calcolarla in modo esatto, si calcolino i valori minimo e massimo.

Si danno sia risposte esatte che approssimate (che comportano leggeri errori rispetto alla versione esatta).

R1	***Osservazioni*** *Un blocco indice contiene 1KB/4B = 256 puntatori.* *L'occupazione di 256GB include la frammentazione interna: il fatto che sia detto esplicitamente che si escludono i blocchi di indice, pone l'informazione di occupazione a livello di gestione dei blocchi da parte del file system. Questo permette, tra l'altro, di semplificare i conteggi, utilizzando come unità di misura i*

	blocchi invece che i Byte. Nella correzione del compito, si considerano corrette le soluzioni che intendono l'occupazione a livello di dimensione originale dei file (escludendo quindi la frammentazione interna). ***Calcolo massimo e minimo per N*** *Si indicano con M1, M2 e M3 le dimensioni medie dei tre tipi di file:* M1: dimensione media file con indice indiretto singolo o diretto M2: dimensione media file con indice indiretto doppio M3: dimensione media file con indice indiretto doppio $1 \text{ blocco} \leq M1 \leq (10 + 256) \text{ blocchi}$ $(10 + 256 + 1) \text{ blocchi} \leq M2 \leq (10 + 256 + 256^2) \text{ blocchi}$ $(10 + 256 + 256^2 + 1) \text{ blocchi} \leq M3 \leq (10 + 256 + 256^2 + 256^3) \text{ blocchi}$ $100N * M1 + 10N * M2 + N * M3 = 256M$ (misure in blocchi: 256GB/1KB = 256M) **Massimo**: per calcolare il massimo valore possibile per N si usano le dimensioni minime per i file. $100N * 1 + 10N * 267 + N * (267 + 256^2) = 256M$ $(3037 + 256^2) N = 256M$ $N = 3914,6$ N deve esser intero (non ci sono frazioni di file) Si arrotonda all'intero inferiore: **N = 3914**. Se si arrotondasse per eccesso l'occupazione totale sarebbe > 256M blocchi anche nel caso di file di dimensione minima. **Il numero massimo di file è quindi N+10N+100N = 434454.** **Minimo (è possibile calcolarlo in modo duale al massimo)**: per calcolare il minimo valore possibile per N si usano le dimensioni massime per i file. $100N * 266 + 10N * (266 + 256^2) + N * (266 + 256^2 + 256^3) = 256M$ *trascurando i termini meno significativi* $10N * 256^2 + N * (256^2 + 256^3) = 256M$ $10N + (1+256)N = 4K$ $N = 4K/267 = 15,34$ *Usando tutti i termini si ottiene* 15,315 *Si noti che, osservando che il rapporto tra la dimensione massima e la minima, per ognuno dei 3 tipi di file, è circa 256, si può rapidamente calcolare N dal valore calcolato in precedenza (per il valore massimo) diviso per 256:* $N = 3914/256 = 15,289$ In tutti i casi, il valore minimo va approssimato per eccesso: **N=16**. Se si utilizzasse N=15, anche con file tutti di dimensione massima, non si potrebbe raggiungere l'occupazione data. **Il numero minimo di file è quindi N+10N+100N = 1776.**

R1b	*Soluzione semplificata con calcoli approssimati* *Un blocco indice contiene 1KB/4B = 256 puntatori.* ***Calcolo valore massimo e minimo per N*** M1: dimensione media file con indice indiretto singolo o diretto M2: dimensione media file con indice indiretto doppio M3: dimensione media file con indice indiretto doppio $1 \text{ blocco} \leq M1 \leq 256 \text{ blocchi}$ $256 \text{ blocchi} \leq M2 \leq 256^2 \text{ blocchi}$ $256^2 \text{ blocchi} \leq M3 \leq 256^3 \text{ blocchi}$ $100N * M1 + 10N * M2 + N * M3 = 256M$ (misure in blocchi: 256GB/1KB = 256M) **Massimo**: per calcolare il massimo valore possibile per N si usano le dimensioni minime per i file.

	$100N * 1 + 10N * 256 + N * 256^2 = 256M$ $(2660 + 256^2) N = 256M$ $(10 + 256) N = 1M$ $N = 3942$ **Il numero massimo di file è quindi N+10N+100N = 437562.** **Minimo** Notando che il rapporto tra la dimensione massima e la minima è circa 256 **N** = $\lceil 3942/256 \rceil = \lceil 15,4 \rceil$ = **16.** **Il numero minimo di file è quindi N+10N+100N = 1776.**

R2	Dimensioni in Byte $5*10^6B \leq |d.txt| \leq 50*10^6B$ Dimensioni in blocchi di dato $|d.txt|_{min} = \lceil 5*10^6B/1024B \rceil = \lceil 4882,8 \rceil = 4883$ $|d.txt|_{max} = \lceil 50*10^6B/1024B \rceil = \lceil 48828,1 \rceil = 48829$ ***Rappresentazione*** *E' sufficiente schematizzare o descrivere a parole un file che, mediante iNode, utilizzi il livello di indirizzamento indiretto doppio, in quanto le dimensioni sono comprese nell'intervallo tra 267 e (266+256^2) blocchi.* ***Calcolo numero di blocchi di indice:*** Oltre all'iNode, serve 1 blocco di indice singolo, 1 doppio di primo livello. Il numero di blocchi di secondo livello è compreso tra $\lceil (4883-267)/256 \rceil$=19 e $\lceil (48829-267)/256 \rceil$=190. Complessivamente il numero di blocchi indice va da 21 a 192. **La frammentazione interna** non è calcolabile in modo esatto: a seconda della dimensione reale del file, va da un valore minimo di 0 a un valore massimo di 1023B (1 blocco meno 1 byte).

R2b	*Soluzione approssimata considerando 1K ≈10^3* Dimensioni in Byte 5MB ≤ |*d.txt*| ≤ 50MB Dimensioni in blocchi di dato $|d.txt|_{min}$ = 5MB/1KB = 5K $|d.txt|_{max}$ = 50MB/1KB = 50K ***Rappresentazione*** *E' sufficiente schematizzare o descrivere a parole un file che, mediante iNode, utilizzi il livello di indirizzamento indiretto doppio, in quanto le dimensioni sono comprese nell'intervallo tra 267 e (266+256^2) blocchi.* ***Calcolo numero di blocchi di indice:*** Oltre all'iNode, serve 1 blocco di indice singolo, 1 doppio di primo livello. Il numero di blocchi di secondo livello è compreso tra (5K-267)/256 = 20-1 = 19 e 50K-267)/256⌉=199. Complessivamente il numero di blocchi indice va da 21 a 201. **La frammentazione interna** non è calcolabile in modo esatto: a seconda della dimensione reale del file, va da un valore minimo di 0 a un valore massimo di 1023B (1 blocco meno 1 byte).

3. *(3 punti)* Si descrivano brevemente vantaggi e svantaggi di una inverted page table (IPT), rispetto a una tabella delle pagine standard (eventualmente gerarchica). Sia dato un processo avente spazio di indirizzamento virtuale di 32 GB, dotato di 8GB di RAM, su una architettura a 64 bit (in cui si indirizza il Byte), con gestione della memoria paginata (pagine/frame da 1KB). Si vogliono confrontare una soluzione basata su tabella delle pagine standard (una tabella per ogni processo) e una basata su IPT. Si calcolino le dimensioni della tabella delle pagine (ad un solo livello) per il processo e della IPT. Si ipotizzi che il `pid` di un processo possa essere rappresentato su 16 bit. Si utilizzino 32 bit per gli indici di pagina e/o di frame. Si dica infine, utilizzando la IPT proposta (32 bit per un indice di pagina/frame), quale è la dimensione massima possibile per lo spazio di indirizzamento virtuale di un processo.

R	***Vantaggi*** • Risparmio di memoria: la tabella ha la dimensione della RAM fisica invece che quella dello spazio di indirizzamento virtuale • C'e una unica tabella per tutti i processi (ogni frame è associato a un solo processo) **Svantaggi** • Il vero svantaggio è la lentezza, occorre cercare una pagina, anziché accedervi in modo diretto. Per questo di solito le IPT sono associate a tabelle di hash. **Per i calcoli delle dimensioni, si trascurano eventuali bit di validità/modifica.** **Page Table standard:** N pagine = 32GB/1KB = 32M (corrisponde al numero di indici di frame nella page table) |Page Table| = 32M*4B = 128MB **IPT** La tabella, unica per tutti i processi, ha dimensione fissa, in quanto contiene un indice di pagina per ogni frame in RAM (si consideri, per semplicità, di gestire tutta la RAM, compresa quella assegnata al sistema operativo). Ogni riga della IPT contiene un indice di pagina (4B) e il pid del processo (2B) N frame = 8GB/1KB = 8M |IPT| = 8M*(4B+2B) = 48MB **Spazio di indirizzamento virtuale** Il numero massimo di pagine virtuali di un processo è limitato dalla dimensione degli indici di pagina (32 bit): tale numero è quindi 4G. Siccome ogni pagina ha dimensione 1KB, lo spazio di indirizzamento virtuale ha dimensione massima 4G*1KB = 4TB.

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

A.

Si riportano in figura parti delle funzioni `uio_kinit`, `load_elf` e `load_segment`.

```
void uio_kinit (struct iovec *iov, struct uio *u,
                void *kbuf, size_t len,
                 off_t pos, enum uio_rw rw) {
  iov->iov_kbase = kbuf;
  iov->iov_len = len;
  u->uio_iov = iov;
  u->uio_iovcnt = 1;
  u->uio_offset = pos;
  u->uio_resid = len;
  u->uio_segflg = UIO_SYSSPACE;
  u->uio_rw = rw;
  u->uio_space = NULL;
}
int load_elf (struct vnode *v, vaddr_t *entrypoint) {
  Elf_Ehdr eh;   /* Executable header */
  Elf_Phdr ph;  /* "Program header" = segment header */
  int result;
  struct iovec iov;
  struct uio ku;
  ...
  uio_kinit(&iov, &ku, &eh, sizeof(eh), 0, UIO_READ);
  result = VOP_READ(v, &ku);
  ...
}
```

```
load_segment (struct addrspace *as, struct vnode *v,
    off_t offset, vaddr_t vaddr,
    size_t memsize, size_t filesize,
    int is_executable) {

  struct iovec iov;
  struct uio u;
  int result;

  iov.iov_ubase = (userptr_t)vaddr;
  iov.iov_len = memsize;  // length of the memory space
  u.uio_iov = &iov;
  u.uio_iovcnt = 1;
  u.uio_resid = filesize; // amount to read from the file
  u.uio_offset = offset;
  u.uio_segflg = is_executable ? UIO_USERISPACE :
                                 UIO_USERSPACE;
  u.uio_rw = UIO_READ;
  u.uio_space = as;

  result = VOP_READ(v, &u);
  ...
}
```

Si spieghi brevemente il ruolo della "`struct iovec`" e della "`struct uio`", In relazione alla successiva VOP_READ.

R	La struct iovec contiene il puntatore all'area di memoria destinazione della read e la relativa dimensione: &eh e sizeof(eh) nella load_elf, vaddr e memsize nella load_segment. La struct uio contiene tutte le informazioni necessarie per l'IO: • Il puntatore a una (o eventualmente un vettore di) stuct iovec • L'offset e il numero di byte da leggere nel file • Le informazioni sullo spazio virtuale (kernel/user) e tipo di I/O (R/W) da effettuare Prima di effettuare un IO in spazio kernel, è sufficiente chiamare la uio_kinit, per predisporre e collegare le due struct, prima di un IO in spazio user, le due strutture vanno caricate in forma esplicita, in quanto non c'è una funzione equivalente alla uio_kinit per lo spazio user.

Perché `load_segment` utilizza UIO_USERISPACE/UIO_USERSPACE, mentre nella parte iniziale della `load_elf` si usa (tramite `uio_kinit`) UIO_SYSSPACE?

R	Perché la prima parte di load_elf acquisisce dal file elf, in una variabile locale in memoria kernel, l'header del file elf: si tratta quindi di un IO di tipo UIO_SYSSPACE. La load_segment invece, deve acquisire i segmenti veri e propri dal file elf alle partizioni di memoria user appena allocate per il processo: l'IO è quindi di tipo UIO_USERISPACE per il codice (istruzioni) e UIO_USERSPACE per i dati.

Perché al campo `u->uio_space` in un caso viene assegnato NULL, mentre nell'altro si assegna `as`? *(a cosa serve questa assegnazione?)*

R	L'assegnazione serve per fornire le informazioni necessarie alla traduzione tra indirizzi logici a fisici. Per lo spazio kernel non serve nulla (quindi si lascia un puntatore NULL) in quanto la traduzione consiste semplicemente nel sommare/sottrarre MIPS_KSEG0. Per lo spazio user serve invece il puntatore alla struct addrspace del processo, in cui sono definite le mappature logico-fisiche dei due segmenti e dello stack.

B.

Si vuol realizzare il supporto (parziale, in quanto non si gestisce il parametro `flags`) per la system call `waitpid`, di cui si fornisce il prototipo:

```
pid_t waitpid (pid_t pid, int *returncode, int flags);
```

Si è già realizzata una funzione `proc_wait`, avente prototipo:

```
int proc_wait(proc_t *proc);
```

che dato il puntatore al descrittore di un processo, ne attende la fine (del processo) e ne ritorna lo stato di terminazione. Si realizzi una funzione `sys_waitpid`, che possa essere richiamata nella `syscall` mediante:

```
case SYS_waitpid:
  retval = sys_waitpid(tf->tf_a0, (userptr_t)tf->tf_a1);
  break;
```

ATTENZIONE: Di una eventuale tabella dei processi, è sufficiente fornire la dichiarazione e l'uso all'interno di `sys_waitpid`, nonché una breve descrizione a parole. Non è necessaria la gestione della tabella al boot e alla creazione/distruzione di un processo.

R

```
// FILE proc.c

// Si assuma che il tipo proc_t sia equivalente a struct proc
// la tabella dei processi contiene nella riga i il puntatore alla struct proc (proc_t) del processo con pid == i.
// la tabella va opportunamente aggiornata alla creazione e distruzione di un processo
// esiste una funzione, esportata da proc.c, che ritorna il puntatore a una struct proc a partire dal pid
// (leggendo la tabella): proc_from_pid

#if TABELLA_DINAMICA
// se si realizza una tabella dinamica
proc_t **tabellaProcessi; // da allocare in fase di boot
int nProc=0; // da incrementare opportunamente al boot o riallocazione
#else
// oppure, con allocazione statica, basata sulla costante MAX_PROC, da definire
proc_t *tabellaProcessi[MAX_PROC];
int nProc = MAX_PROC;
#endif

proc_t *proc_from_pid(pid_t pid) {
 return tabellaProcessi[pid];
}

// File proc_syscalls.c (o equivalente)
// NOTA: si suppone che la discruzione del processo sia effettuata nella proc_wait

int sys_waitpid (pid_t pid, userptr_t returncodeP) {
  proc_t *proc = proc_from_pid(pid); // passa da pid a puntatore
  if (proc==NULL) return -1; // o altro codice per indicare errore
    *returncodeP = proc_wait(proc); // aspetta e assegna il valore ritornato
  return (int)pid; // oppure altro codice per indicare successo
}
```