Sistemi Operativi
Compitino Advanced 18 Marzo 2005

1. Un modo per implementare le protezioni è dotare ogni processo della sua lista di abilitazioni. Come si può evitare che il processo stesso, per errore o malevolmente, cerchi di cambiarsi le proprie abilitazioni?

   È sufficiente che tali protezioni siano su una pagina, o su un segmento, non scrivibile in modalità user, ma solo dal kernel. Apposite system call permetteranno di accedere a tale segmento in modo controllato.

2. In Unix, la system call `fsync()` serve per forzare la scrittura su disco di tutti i blocchi dati e attributi modificati di un file.

   (a) Questa system call ha senso anche per file system journaled?

   Sì, perché solitamente i file system journaled mantengono sui log solo gli attributi, mentre i blocchi dati sono soggetti al normale meccanismo di caching con scrittura asincrona.

   (b) Esiste in MS/DOS una primitiva analoga? Perché?

   No, non esiste perché in MS/DOS le scritture sono sincrone (write-through).

3. Supponiamo che un processo in un sistema UNIX a memoria virtuale paginata, per quanto di piccole dimensioni, compia un intenso I/O accedendo ad una grande quantità di dati in un brevissimo arco di tempo. Pensando al meccanismo che viene utilizzato per allocare i buffer di I/O si dica che effetti può avere l'esecuzione di detto processo sulla gestione della memoria e quindi sugli altri processi.

   Può abbassare notevolmente la free list, e quindi attivare il processo di paginazione per liberare frame di memoria, e quindi agli altri processi possono essere sottratti frame che avevano a disposizione.

4. Si consideri la seguente stringa dei riferimenti: 1 6 2 3 3 3 4 2 4 6 5.

   (a) Usando il modello del working set con $\Delta = 4$ quanti page faults ci sono complessivamente in un sistema con 3 frames fisici?

   8 page faults. Basta fare la matrice:

1	6	2	3	3	3	4	2	4	6	5
1	6	2	3	3	3	4	2	4	6	5
	1	6	2	2	2	3	4	2	4	6
		1	6	6			3	3	2	4
			1							2
p	p	p	p			p	p		p	p

   (b) Quanto deve essere Δ per minimizzare i page faults?

   Gli unici page faults che si potrebbero evitare sono quando rientra la 2 e la 6. Per mantenere la 2 serve $\Delta = 6$, mentre per la 6 servirebbe $\Delta = 9$ ma con almeno 4 pagine fisiche. Quindi il valore che minimizza è $\Delta = 6$ (con 7 page faults).

5. Si consideri un disco gestito con politica LOOK. Inizialmente, la testina è posizionata sul cilindro 80, discendente; lo spostamento ad una traccia adiacente richiede 1ms. Al driver di tale disco arrivano richieste per i cilindri 27, 61, 74, 24, rispettivamente agli istanti 0ms, 20ms, 50ms, 55ms. Si trascuri il tempo di latenza.

(a) In quale ordine vengono servite le richieste?

> All'istante 0, la testina inizia a muoversi alla velocità di 1 traccia/ms verso il cilindro 27. Dopo 20ms, quando arriva la richiesta per il cilindro 61, la testina si trova già oltre, sul cilindro 60, la direzione è discendente e quindi non viene servita. Stesso discorso per l'istante 50, quando arriva la richiesta per il cilindro 74. La testina continua il movimento verso il cilindro 27, ove vi giunge all'istante $80 - 27 = 53$ms. Dopo aver servito questa richiesta, ci sono le due richieste a 61 e 74 in sospeso; la testina inverte quindi la direzione, verso la traccia 61. Quando, dopo 2ms, arriva la richiesta per il cilindro 24, è troppo tardi: la testina continua a salire fino al cilindro 61 (che viene raggiunto dopo $61-27 = 34$ms, ossia all'istante $53 + 34 = 87$ms) e poi al cilindro 74 (che viene raggiunto dopo altri $74 - 61 = 13$ms, ossia all'istante $87+13 = 100$ms). Infine, viene invertita nuovamente la direzione per raggiungere il cilindro 24 dopo $74 - 24 = 50$ms, ossia all'istante $100 + 50 = 150$ms. Quindi l'ordine è: 27, 61, 74, 24.

(b) Il *tempo di attesa* di una richiesta è il tempo che intercorre dal momento in cui è sottoposta al driver, a quando viene effettivamente servita. Qual è il tempo di attesa medio per le quattro richieste in oggetto?

> I tempi di attesa per le quattro richieste sono rispettivamente: per il cilindro 27: 53-0=53ms; per il cilindro 61: 87-20=57ms; per il cilindro 74: 100-50=50ms; per il cilindro 24: 150-55 = 95ms. La media è $\frac{53+57+50+95}{4} = 63.25$ ms.

6. Si consideri un sistema a paginazione a tabella delle pagine invertita (senza hash); spazio di indirizzamento fisico a 38 bit e logico a 64 bit, pagine da 8K, 8GB di RAM. Ogni entry nella page table occupa 64 bit. Supponiamo che un processo acceda alle seguenti pagine del suo spazio virtuale: 0xBC0E, 0xBC0F, 0x220 e 0x7FFFFFFFFFFF.

   (a) Quanti frame devono essere allocati al processo per contenere queste pagine e quanto serve (nel caso peggiore) al sistema di paginazione?

   > Visto che le pagine sono da 8KB gli ultimi 13bits di ogni indirizzo logico riferiscono all'offset e quindi le pagine logiche riferite sono 3 (0x5, 0x0 e 0x3FFFFFFFFF). Al sistema di paginazione alla peggio serviranno altri 3 frames per mantenere le relative traduzioni.

   (b) Con un tempo di accesso in cache t_c, hit ratio α, in RAM t_m, page fault rate p e tempo di recupero pagina da disco t_d qual'è il tempo di accesso medio in memoria (EAT)?

   > Abbiamo un totale di $\frac{8 \cdot 2^{30} B}{8 \cdot 2^{10} B} = 2^{20}$ entries in tabella. La tabella occuperà $\frac{2^{20} \cdot 8B}{4 \cdot 2^{10} B} = 2^{11}$ frames e quindi assumiamo di tenerla sempre in RAM. Il tempo medio di ricerca in tabella è $t_r = \frac{1}{2} 2^{20} t_m = 2^{19} t_m$.
   > Ad ogni accesso attendiamo che la cache tenti di risolvere l'indirizzo. Se va male, con frequenza $1 - \alpha$, dobbiamo accedere alla tabella per la ricerca. Ora, se va male, con frequenza p dobbiamo accedere al disco. Poi (in ogni caso) facciamo l'accesso in RAM. Quindi $t_{EAT} = t_c + (1-\alpha)(t_r + pt_d) + t_m = t_c + (1 + 2^{19}(1-\alpha))t_m + p(1-\alpha)t_d$.

7. Si consideri un disco con velocità di rotazione pari a 5400 rpm e tempo medio di seek t_s pari a 9 msec, blocchi da 4K. Ogni traccia contiene 720 blocchi. Il file system sia NTFS.

   (a) Quant'è il tempo per la lettura sequenziale completa di un file di 3.5MB allocato (per quanto possibile) tutto contiguo?

   > Il tempo di latenza medio è $t_l = \frac{1}{2} \frac{60}{5400} = 5.6ms$. Il tempo di lettura di 1 blocco è $t_b = \frac{60}{5400 \cdot 720} = 15.4\mu s$.

Il file è composto da $\lceil \frac{3.5 \cdot 2^{20} B}{4 \cdot 2^{10} B} \rceil = \lceil 3.5 \cdot 2^8 \rceil = 896$ blocchi. Per essere il più possibile contiguo il file sarà logicamente di un solo run e fisicamente spezzato in due parti, un'intera traccia da 720 blocchi più un'altra da 176.
Quindi iniziamo a leggere la MFT in un tempo di $t_s + t_l + t_b$, poi andiamo a leggere il primo pezzo in un tempo di $t_s + t_l + 720t_b$ e infine il secondo pezzo in $t_s + t_l + 176t_b$. Totale $t = 3t_s + 3t_l + 897t_b = 57.5ms$.

(b) E quanto è invece se detto file si ritrova frammentato in pezzi tutti da 16KB in giro per il disco?

Ci servono $896/4 = 224$ runs da 4 blocchi. Per ogni run impieghiamo $t_s + t_l + 4t_b$. Assumendo (ragionevolmente) che l'informazione sui 224 runs sia memorizzata nello stesso blocco della MFT, il totale è $t = 225t_s + 225t_l + 897t_b = 3.3s$ (57 volte di più).

8. (a) Quali cautele bisogna avere nel "portare" un sistema operativo scritto per monoprocessore su una macchina multiprocessore (a memoria condivisa)? Si analizzino le differenze fra il caso a microkernel e quello monolitico. Si discuta quindi sulla scalabilità dei due approcci.

Il problema che si presenta nel porting è l'accesso contemporaneo da parte di più processori alle stesse strutture dati del kernel. Per ovviare al problema in modo molto semplice basta limitare ad 1 processore per volta la possibilità di eseguire codice kernel. Con questo approccio blocchiamo però processori diversi che potrebbero voler operare su sezioni di kernel disgiunte.

- Nel caso monolitico abbiamo molte sezioni grosse e molte di queste sono fra loro disgiunte e quindi sicuramente questo approccio non è efficiente.
- Nel caso a microkernel invece le sezioni sono poche e piccole quindi l'incidenza del problema è sicuramente minore.

Per ovviare all'inefficienza in entrambe i casi bisogna ristrutturare il codice del kernel in sezioni indipendenti, ognuna ad accesso esclusivo.

- Nel caso a microkernel questa operazione è relativamente più agevole viste le ridotte dimensioni delle sezioni e il loro modesto numero.
- Nel caso monolitico sorgono sicuramente molti più problemi. Questo tra l'altro aumenta esponenzialmente la possibilità che per errore si introducano errate sequenze di mutex che possono portare a deadlock di interi processori in modalità kernel.

In conclusione per i motivi precedenti e, soprattutto, vista la percentuale di tempo che un processo passa in modalità kernel nei due casi si può concludere che l'approccio a microkernel scala sicuramente molto meglio.

(b) I processi di sistema da quale CPU vengono eseguiti nel caso a microkernel?

I processi di sistema possono essere eseguiti da una qualunque CPU. Lo scheduler del sistema operativo si occuperà di bilanciare il carico.

(c) E le chiamate di sistema nel caso monolitico vengono eseguite necessariamente dalla stessa CPU invocante?

No, le chiamate che sospendono un processo partono su un determinato processore ma dopo il risveglio possono completarsi su un'altro. Si tenga comunque presente che solitamente

c'è una preferenza per il processore originante: lo scheduler tende a riassegnare un processo allo stesso processore per sfruttare meglio le cache (principio del "letto caldo").

(d) Si possono avere deadlock dovuti all'uso di risorse del kernel? Quale soluzione per evitarli, in caso, appare più adeguata?

Sì è possibile avere deadlocks. Visto che le risorse del kernel sono ben note, e non cambiano facilmente nelle varie release del sistema operativo, la soluzione più adeguata (per evitare deadlocks) è l'ordinamento preventivo delle risorse del kernel: ogni processo può allocare tali risorse solo in ordine crescente.

9. Scrivere uno pseudo-programma C che si occupi a intervalli di 24h di registrare in un file di log la differenza (in sec) fra l'orologio di sistema e quella di un server specificato da linea di comando.

```
includere gli headings

definire le costanti

int main (unsigned argc, char **argv)
{
  int sock, log_fd;
  int count, diff;
  char inputline[LINESIZE];
  struct_per_systemclock time;
  struct_per_la_socket client, server;
  struct_per_indirizzi_IP *host;
  altre_vars;

  if (argc != 2) {
    fprintf(stderr, "usage: %s <server address>\n", argv[0]);
    exit(2);
  }

  sock = crea_socket;
  if (sock < 0) {
    perror("creating socket");
    exit(1);
  }

  client=prepariamo_struttura(costanti_client);

  if ( leghiamo_indirizzo(sock,client) < 0) {
    perror("bind failed");
    exit(1);
  }

  if ( (host=recuperiamo_indirizzo_da(argv[1]) == NULL) {
    perror("unknown host");
    exit(1);
  }

  server=prepariamo_struttura(host);
```

```
  while (TRUE) {
      /* spediamo un pacchetto dummy per svegliare il server */
      count = send_pacchetto(sock, server);

      time = leggi_systemclock();

      /* riceviamo la risposta, contenente la data locale */
      receive_pacchetto(sock, inputline, server);

      diff = calcola_differenza_orari(time,inputline);

      if ( (log_fd = open_file(nome_file_log, append)) < 0) {
        perror("cannot open log file");
        exit(3);
      }

      fprintf(log_fd,"Local and Remote Time differs by %d secs\n",
              diff);

      close(log_fd);
      sleep(costante_24h);
  }
}
```