Operativsystem ID2200/06

tentamen och omtentamen

2017-01-14 08:00-12:00

Namn: _____

Instruktioner

- Du får, förutom skrivmateriel, endast ha med dig en egenhändigt handskriven A4 med anteckningar. Mobiler etc skall lämnas till tentamensvakterna.
- Svaren skall lämnas på dessa sidor, använd det utrymme som finns under varje uppgift för att skriva ner ditt svar.
- Svar skall skrivas på svenska eller engelska.
- Du skall lämna in hela denna tentamen.
- Inga ytterligare sidor skall lämnas in.

Versioner

Om detta är din ordinarie tentamen i ID2200/06 så skall frågorna 1-9 besvaras men inte fråga 10.

Om detta är en omtentamen för den version av kursen som gick före HT2016 så gäller följande:

- För de som läste ID2200 VT2016 är tentamen på 6hp och frågorna 1-9 skall besvaras.
- För de som läste ID2200 före VT2016 är tentamen på 3.8hp och endast frågorna 1 till 8 skall besvaras.
- För omtentander på ID2200 så gäller även att: om det är så att du inte har fått godkänt på de laborationsmoment som kursen innehöll kan du välja att besvara frågorna under 9 och då få det momentet tillgodoräknat.
- För de som läste kursen ID2206 före HT2016 är tentamen på 4.5hp och frågorna 1 till och med 8 och fråga 10 skall besvaras.

- För omtentander på ID2206 så gäller även att: om det är så att du inte har fått godkänt på de laborationsmoment som kursen innehöll kan du välja att besvara frågorna under 9 och då få det momentet delvis tillgodoräknat. Frågorna under 9 motsvar ca: 2.2hp.

Betyg för 6hp

Tentamen har ett antal uppgifter där några är lite svårare än andra. De svårare uppgifterna är markerade med en stjärna, *poäng**, och ger poäng för de högre betygen. Vi delar alltså upp tentamen i grundpoäng och högre poäng. Se först och främst till att klara grundpoängen innan du ger dig i kast med de högre poängen.

Notera att det av de 40 grundpoängen räknas bara som högst 36 och, att högre poäng inte kompenserar för avsaknad av grundpoäng. Gränserna för betyg är som följer:

- Fx: 21 grundpoäng
- E: 23 grundpoäng
- D: 28 grundpoäng
- C: 32 grundpoäng
- B: 36 grundpoäng och 12 högre poäng
- A: 36 grundpoäng och 18 högre poäng

Gränserna kan komma att justeras nedåt men inte uppåt.

Gränsen för E är för tentamen på 4.5hp 18 poäng och för 3.8hp tentamen 16 poäng.

Erhållna poäng

Skriv inte här, detta är för rättningen.

Uppgift	1	2	3	4	5	6	7	8	9
Max G/H	4/0	2/2	4/2	4/2	4/2	4/2	4/2	2/2	12/8
G/H									

Totalt antal poäng:

Namn:_____

1 Operativsystem

1.1 filer och rättigheter [2 poäng]

I ett Unix-operativsystem så är filer och mappar (directories) skyddade så att rättigheterna för att använda ett objekt är begränsat. Dessa rättigheter kan vi se då vi listar innehållet i en mapp. Beskriv vilka rättigheter som gäller för filen `foo` nedan.

```
> ls -l foo
> -rwxr-x--- 1 kalle trusted 234 dec 26 13:18 foo
```

Svar: Ägaren `kalle` har läs-, skriv- och exekveringsrättigheter. Medlemmar i gruppen `trusted` kan läsa och exekvera. Övriga användare har inga rättigheter

1.2 kommandon i ett shell [2 poäng]

Ge en kort beskrivning av vad kommandona nedan gör.

- `wc`
- `grep`
- `mkdir`
- `pwd`

Svar: Slå upp deras betydelse med hjälp av `man`.

2 Processer

2.1 fork() [2 poäng]

Vad skrivs ut när vi kör programmet nedan och varför får vi detta resultat?

```
#include <unistd.h>
#include <stdio.h>

int x = 42;
```

Namn:_____

```
int main() {

  if(fork() == 0) {
    x++;
    printf("x = %d\n", x);
  } else {
    x++;
    printf("x = %d\n", x);
  }
  return 0;
}
```

Svar: Det kommer att skrivas `x = 43` och `x = 43` efter varandra. De två processerna kommer få var sin kopia av den globala datastrukturen `x` som då har värdet 42. Eftersom uppdateringarna är oberoende av varandra har båda processerna värdet 43 vid utskrift.

2.2 IDT [2 poäng*]

Vad fyller den så kallade IDT:n (Interrupt Descriptor Table) för funktion och vad händer när en process i *user mode* exekverar instruktionen `INT` (x86 assembler). Ge en kortfattad beskrivning.

Svar: IDT:n sätts upp av kärnan och har pekare till procedurer som skall exekveras vid olika avbrott (interrupt). När en användarprocess exekverar exempelvis `INT 80` så går processorn över i *kernel mode* och hoppar till den procedur som anges för position 80 (hex).

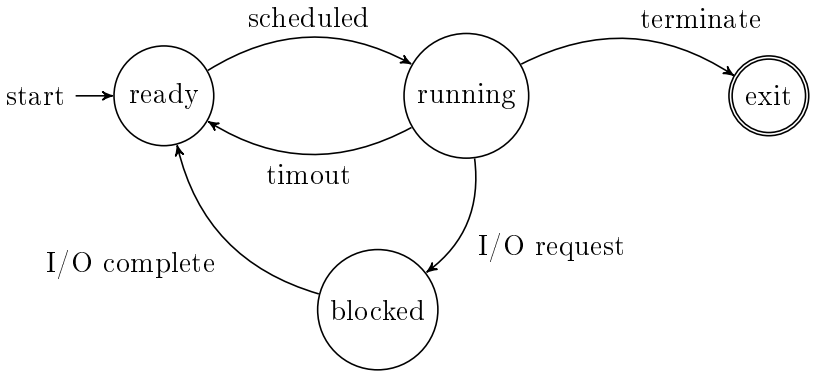
3 Schemaläggning

3.1 tillståndsdiagram [2 poäng]

Här följer ett tillståndsdiagram för processer vid schemaläggning. Fyll i de markerad delarna så att man förstår vad tillstånden betyder och när en process förs mellan olika tillstånd.

Svar:

Namn:_____



3.2 shortest job first [2 poäng]

Antag att vi har en schemaläggare enligt principen "shortest job first" dvs. utan att kunna avbryta pågående jobb. Om vi har tre jobb som tar 10ms, 20ms och 30ms så är det en bättre stategi än att ta jobben i slumpmässig ordning, visa varför.

Svar: Om vi tar dem i SJF-ordning så kommer vi få en genomsnittlig omloppstid på $(10 + 30 + 60)/3 = 33$ ms. Om vi råkar ta dem i "fel" ordning så blir om loppstiden $(30 + 50 + 60)/3 = 47$ ms.

3.3 rättvis schemaläggare [2 poäng*]

Det finns en grupp med schemaläggningmetod där varje process får en "rättvis" del av beräkningskraften. Några av dessa är baserade på ett "lotteri" men vi kan också implementera en helt deterministisk algoritm. Vad kallas denna metod? Ge en kortfattat beskrivning hur den fungerar.

Svar: Den kallas "steg-schemaläggare" (stride scheduler). Alla processer har ett värde och en steglängd. Processen med minst värde blir schemalagd men får sedan sitt stegvärde (stride value) adderat till sitt värde.

4 Virtuellt minne

4.1 basregister [2 poäng]

Varför vill man inte implementerar ett segmenterat minne genom att endast använda ett *basregister* (base register) för att beskriva ett segment?

Namn:_____

Svar: En process skulle kunna använda ett offset som adresserar utanför det segment den blivit tilldelad. För att förhindra detta används ett register som sätter en gräns s.k. *bound*.

4.2 ett träd [2 poäng]

Vid representation av en *sidtabell* (page table) så används en trädstruktur. Varför har man en trädstruktur, det skulle gå betydligt snabbare att slå upp ett värde om man representerade tabellen som en *array* med direkt åtkomst till elementen. Ett träd ger oss bara en eller flera indirektionssteg så varför använda ett träd?

Svar: Vi behöver inte representera hela tabellen utan bara den bråkdel av den virtuella adressrymden som används. Detta ger oss en betydligt mindre datastruktur att hantera - viktigt för en 32-bitars adressrymd och helt avgörande för en rymd på 64-bitar.

4.3 segmenterat minne x86 [2 poäng*]

När används segmenterat minne i ett Linuxsystem på en x86-arkitektur?

Svar: Det används för implementering av minne som hör till en specifik tråd eller kärna.

5 Minneshantering

5.1 parkeringsplatser [2 poäng]

När det skulle ordnas parkeringsplatser längs med Sveavägen så fanns två förslag: 1/ ha målade parkeringsrutor som var 6m långa eller 2/ låt bilar parkera med 25 cm mellanrum, utan att begränsas av målade rutor. Om vi för enkelhetens skull antar att bilar är mellan 4.0 och 5.5 meter och att alla kan parkera en bil i en lucka med en halv meter till godo, vad är då problemet med respektive lösning?

Svar: I första alternativet så kommer vi ha en intern fragmentering då vi i snitt förlorar 75 cm per plats. I fall två kan vi få extern fragmentering eftersom luckor kan bli för små för att användas.

Namn:_____

5.2 lista med friblock [2 poäng]

Om vi vid implementering av `malloc()` och `free()` väljer att spara de fria blocken i en länkad lista som är ordnad i adressordning, så har vi en viss fördel. När vi gör `free()` på ett block och lägger in det i listan så kan vi utföra en operation som minskar den externa fragmenteringen. Vad är det vi kan göra och varför är det en fördel att ha listan ordnad i adressordning?

Svar: Vi kan omedelbart avgöra om de närmaste blocken är i direkt anslutning till det nya blocket och i så fall slå ihop blocken till ett större block. Utan ordningen skulle vi var tvungna att söka igenom alla fria block.

5.3 buddy-allokering [2 poäng*]

Den så kallade "buddy-algoritmen" för att hantera ledigt minne har en klar föredel men också en klar nackdel. Antag att vi har ett minne som är indelat i 4K-block och att vi har 16K som är fria i block `0b10100`. Vad gör vi när vi får 8K ledigt i block `0b11000`? Hur illustrerar detta nackdelen?

Svar: De två blocken ligger bredvid varandra men kan inte slås ihop till ett block på 24K. Det 8K block som blev ledigt skulle möjligtvis kunna slås ihop med ett 8K block på position `0b11010`.

6 Flertrådad programmering

6.1 count [2 poäng]

Om vi exekverar proceduren `hello()` nedan samtidigt i två trådar så kan resultatet bli att `count` får värdet ...- vilka värden kan `count` anta efter det att båda trådarna exekverat klart? Hur kan det komma sig?

```
int loop = 10;
int count = 0;

void *hello() {
  for(int i = 0; i < loop; i++) {
      count++;
  }
}
```

Svar: Variabeln `count` kan ha ett slutgiltiga värde i intervallet $[2, 20]$. Detta eftersom operationen `count++` inte är atomär utan består av en läsoperation, en addering och en skrivoperation.

Namn:_____

6.2 egen stack [2 poäng]

Om vi delar upp en process i två trådar så kan de båda trådarna läsa från varandras stackar - sant eller falskt? Motivera ditt svar.

Svar: Sant - de två trådarna delar adressrymd och har samma läs och skrivrättigheter. De har var sin stack och därmed varsin stackpekare men det är ingenting som hindrar (fast inte så klokt) att man läser eller skriver från varandras stackar.

6.3 total store order [2 poäng*]

Hur skiljer sig *total store order* från sekventiell konsistens (sequential consistency)?

Svar: Vid sekvensiell konsistens utförs alla operationer i en total ordning som överensstämmer med ordningen för varje process. Vid *total store order* gäller detta endast för skrivoperationer, läsoperationer tvingas inte att utföras i den ordning som processen utför dem. En läsoperation kan utföras även om en tidigare skrivoperation till en annan adress inte har utförts.

7 Filsystem

7.1 zombies [2 poäng]

Om man råkar ta bort den sista länken till en fil så blir filen en så kallad *zombie-fil* - sant eller falskt? Motivera ditt svar.

Svar: Falskt, en *zombie* är en process som terminerat där moderprocessen ännu inte har tagit del av dess resultat. En fil tas bort från filsystemet när sista länken tas bort.

7.2 bitmappar [2 poäng]

Vid implementation av ett filsystem så kan vi dela in en disk i: super-block, bitmappar, inoder och datablock. Vad används bitmapparna till?

Svar: Bitmapparna beskriver vilka inoder och vilka datablock som är lediga.

Namn:_____

7.3 en map [2 poäng*]

Vilken information innehåller en mapp (directory) och hur representeras den?. Beskriv i termer av inod:er och datablock, vilken information finns var? Rita en enkel bild som beskriver strukturen.

Svar: En mapp representerar av en inode. Noden innehåller generell information: när skapad, senaste ändrar, ägare, antal block etc. Inoden har även en, eller flera, pekare till datablock som innehåller en mappning från namn till typ och inoder. Varje element är en länk till en: fil, sub-map, symbolisk länk mm.

8 Virtualisering

8.1 user mode [2 poäng]

Vid virtualisering av ett helt operativsystem kommer det virtualiserade operativsystemet att köra i *user mode* - sant eller falsk? Motivera ditt svar.

Svar: Sant, operativsystemet kör i *user mode* så att alla avbrott kommer att kontrolleras av den *hypervisor* som övervakar de virtualiserade operativsystemen.

8.2 emulatorer [2 poäng*]

En vanlig *hypervisor* används för att virtualisera ett eller flera operativsystem så att alla kan köras på samma hårdvara. Så kallade *emulatorer*, som exempelvis QEMU, har en ytterligare fördel - vilken?

Svar: De kan virtualisera en annan hårdvara än den hårdvar som emulatorn kör på.

9 Implementering

9.1 memory map [2 poäng]

Nedan följer en, något förkortad, utskrift av en minnesmappning av en körande process. Beskriv kortfattat vad varje segment markerat med `???` fyller för roll.

Namn:_____

```
> cat /proc/13896/maps

00400000-00401000 r-xp 00000000 08:01 1723260            .../gurka ???
00600000-00601000 r--p 00000000 08:01 1723260            .../gurka ???
00601000-00602000 rw-p 00001000 08:01 1723260            .../gurka ???
022fa000-0231b000 rw-p 00000000 00:00 0                  [???]
7f6683423000-7f66835e2000 r-xp 00000000 08:01 3149003    .../libc-2.23.so ???
    :
7ffd60600000-7ffd60621000 rw-p 00000000 00:00 0          [???]
7ffd60648000-7ffd6064a000 r--p 00000000 00:00 0          [vvar]
7ffd6064a000-7ffd6064c000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

Svar: De första tre segmenten är: kod, read-only data och global data för processen `gurka`. Efter det har vi ett segment för processens *heap*. Segmentet markerat med `libc-2.23.so` är ett delat bibliotek. I den övre regionen hittar vi processens stack.

9.2 execlp() [2 poäng*]

I programmet nedan anropar vi biblioteksproceduren `execlp()`. Vad gör denna rutin och när och i vilken ordning kommer "Humle" och "Dumle" skrivas ut? Motivera ditt svar.

```
int main() {

  int pid = fork();

  if(pid == 0) {
    execlp("gurka", "gurka", NULL);
    printf("Humle\n");
  } else {
    wait(NULL);
    printf("Dumle\n");
  }
  return 0;
}
```

Svar: Proceduren `execlp()` kommer att ersätta processens minnessegment med kod och data från programmet `gurka`. "Humle" kommer aldrig att skrivas ut (om vi antar att `execlp()` lyckas) medan "Dumle" skrivs ut först när programmet `gurka` har terminerat.

Namn:_____

9.3 modifiera kärnan [2 poäng]

Om vi skall lägga till en funktionalitet i en Linuxkärna så måste vi kompilera om kärnan och starta om systemet - sant eller falskt. Motivera varför detta är nödvändigt eller varför det inte är nödvändigt.

Svar: Falskt - man kan ladda en kärnmodul i den körand kärnan till exempel genom att använda kommandot `insmod`.

9.4 proc_create() [2 poäng*]

Det finns flera sätt på vilket man kan kommunicera med en kärnmodul, koden nedan visar på ett sätt. Här initialiserar vi vissa datastrukturer som sedan gör det möjligt för andra processer att på ett enkelt sätt anropa vår modul. Vilket är det synliga *interface* som processer kommer att använda? Ge en kort förklaring.

```
static int __init skynet_init(void) {
  proc_create("skynet", 0, NULL, &skynet_fops);
  printk(KERN_INFO "Skynet in control\n");
  return 0;
}

static void __exit skynet_cleanup(void) {
  remove_proc_entry("skynet", NULL);
  printk(KERN_INFO "I'll be back!\n");
}
```

Svar: Vi registrerar modulen som en fil under `/proc`. Processer kan nu använda vanliga filoperationer för att kommunicera med modulen.

9.5 en graf [2 poäng]

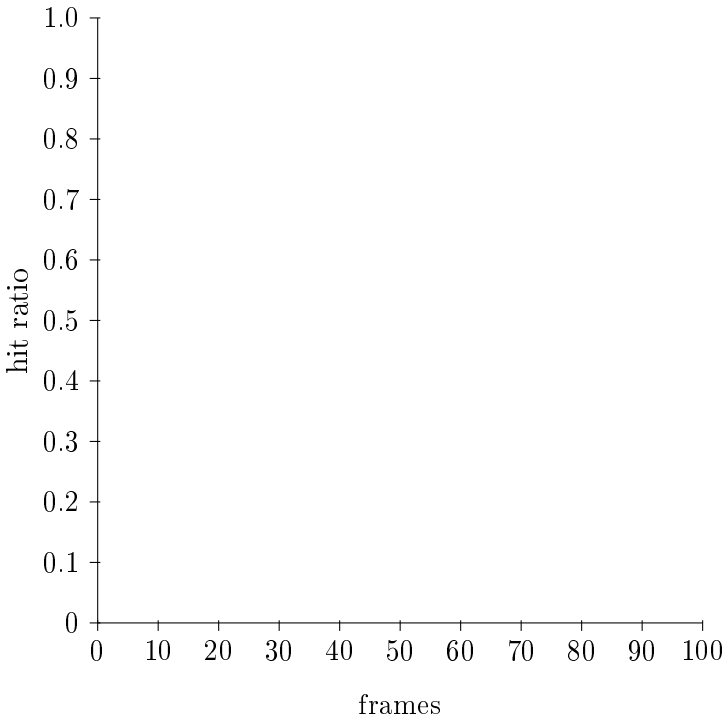
Antag att vi gör ett experiment där vi jämför tre metoder att implementera en *swapping-algoritm* dvs att kasta ut sidor från minnet när vi behöver ytterligare ramar. I vårt experiment har vi ett virtuellt minne på 100 sidor (pages) och simulerar ett minne med upp till 100 ramar (frames). Experimentet simulerar en sekvens av minnesoperationer med temporal lokalitet (temporal locality).

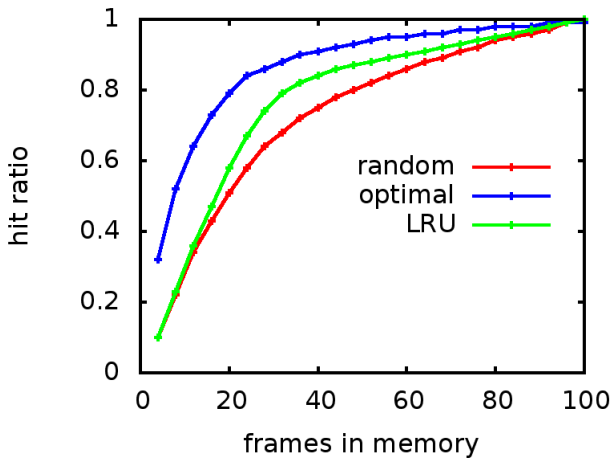
I diagrammet nedan skall du skriva in försvarbara mätserier för följande tre strategier:

- RND: välj en sida slumpmässigt

Namn:_____

- OPT: välj den sida som inte kommer att användas på länge
- LRU: välj en sida som inte har använts på länge





Svar:

Namn:_____

9.6 en dyr operation [2 poäng]

Nedan är ett utsnitt från ett program som implementerar *Least Recently Used* (LRU). Koden visar på varför LRU är dyr att implementera och att man kanske istället väljer att approximera denna strategi. Vad är det koden gör och när används den?

```
    :
if(entry->present == 1) {

    if(entry->next != NULL) {

      if(first == entry) {
        first = entry->next;
      } else {
        entry->prev->next = entry->next;
      }
      entry->next->prev = entry->prev;

      entry->prev = last;
      entry->next = NULL;

      last->next = entry;
      last = entry;
    }
} else {

  :
}
```

Svar: Koden länkar ut ett entry och lägger det sist i en länkad lista som skall vara uppdaterad med de minst använda sidorna först. Operationen måste göras varje gång en sida refereras.

9.7 tick-tack [2 poäng*]

Vid implementering av *klock-algoritmen* så räcker det med att ha den cirkulära listan av element enkellänkade dvs vi behöver inte ha dubbellänkar. Varför klara vi oss bra med en enkellänkad struktur?

Svar: Vi plockar aldrig bort element slumpmässigt utan alltid från visarens nuvarande position. Vi kan enkelt hålla reda på dess position och dess omedelbara föregångare vilket är det enda vi behöver för att länka ur ett element.

Namn:_____

9.8 lock() [2 poäng]

Antag att vi har definierar `try()` enligt koden som följer. Proceduren `__sync_val_compare_and_swap(int *loc, int old, int new)` kommer att i en atomär operation jämföra värdet för `*loc` och om det är lika med `old` skriva dit `new`. Proceduren returnerar värdet för `*loc` före operationen.

```
int try(volatile int *mutex) {
  return __sync_val_compare_and_swap(mutex, 0, 1);
}
```

Hur kan vi implementera ett enkelt *spin-lock*? Utgå från koden nedan och antag att vi anropar `lock()` och `unlock()` med en pekare till ett globalt lås som innehåller `0` i sitt olåsta läge.

```
void lock(volatile int *mutex) {
  :
  :
  :
}

void unlock(volatile int *mutex) {
  :
  :
  :
}
```

Svar:

```
void lock(volatile int *mutex) {
  while(try(mutex) != 0);
}

void unlock(volatile int *mutex) {
  *mutex = 0;
}
```

9.9 futex_wait() [2 poäng]

Vi kan förbättra egenskaperna, eller rättare sagt lindra problemen, med ett *spin-lock* genom att använda s.k. *futex*. Koden som följer implementerar två procedurer `futex_wait()` och `futex_wake()`.

```
int futex_wait(volatile int *futexp) {
  return syscall(SYS_futex, futexp, FUTEX_WAIT, 1, NULL, NULL, 0);
```

Namn:_____

```
}

void futex_wake(volatile int *futexp) {
  syscall(SYS_futex, futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
}
```

Vad gör dessa procedurera och hur kan det användas?

Svar: Proceduren `futex_wait()` kommer att suspenderar den körande processen om låset fortfarande är låst. Proceduren `futex_wake()` kommer att väcka en suspenderar process (om det finns någon suspenderad).

9.10 updatering av en lista [2 poäng*]

Antag att vi har en enkellänkad sorterad lista som vi vill uppdatera genom att ta bort eller lägga till element i. Vi vill att flera trådar skall kunna utföra operationer parallellt och implementerar därför en lista där varje cell är skyddat med ett lås. Operationerna kräver naturligtvis att vi håller två lås när vi tar bort ett element men det enkelt att implementera.

Vi kan välja på att ha använda `pthread_mutex_lock()` för att implementera låsoperationerna men vi skulle också kunna använda enkla *spin-locks*. Vad är fördelen respektive nackdelen med att använda *spin-locks* och hur förändras förutsättningarna om vi har fler trådar och färre kärnor?

Svar: Fördelen med *spin-locks* skulle vara att låsen hålls för mycket kort period och att ett taget lås med stor sannolikhet släpps inom några få klockcykler. Om vi spinner så kan vi undvika att suspendera en process. Nackdelen är förstås om processen som håller låset är suspenderad. Om vi ökar antalet trådar eller minskar antalet kärnor så ökar chansen att en process suspenderas vilket skulle tala emot att använda ett *spin-lock*

Namn:_____

10 Bara för omtentamen i ID2206

10.1 saltade lösenord [2 poäng]

När lösenord lagras i krypterad form på en server så används ofta ett så kallat *salt*. Vad har saltet för funktion?

Svar: Saltet är ett unikt värde som lagras tillsammans med det krypterade summan av lösenordet och saltet. Detta förhindrar att en angripare kan jämföra krypterade lösenord mot en databas med färdigkrypterade ord.

10.2 Rate Monotonic Scheduling [2 poäng]

En realtidsschemaläggare som baserar sig på "Rate Monotonic Scheduling" (fast prioritet där prioriteten bestäms av periodiciteten) är en relativt enkel schemaläggare. Har vi några garantier för att den kommer att fungera dvs att inga deadlines kommer att missas?

Svar: Ja, RMS kommer alltid att fungera om lasten är under $n * (2^{1/n} - 1)$ där n är antalet processer. Den kan fungera vid högre last men vi har inga garantier.

10.3 multiprocessor [2 poäng*]

I ett operativsystem för en multiprocessor så har vi frågan om en process som har kört på en processor skall schemaläggas på en annan processor. Vad är fördelen respektive nackdelen med att låta processer schemaläggas på olika processorer?

Svar: Fördelen är att vi får en bättre möjlighet att lastbalansera systemet så att alla processer samma tillgång till processorkraft. Nackdelen är att vi vid ett byte av processor kommer till en processor där vi inte kommer ha några data strukturer processorns cache.

10.4 NFS [2 poäng*]

Vid implementering av NFS så har servern ett tillstånd för varje öppnad fil som bland annat innehåller processens aktuella skriv- och läsposition i filen - sant eller falsk? Motivera ditt svar.

Svar: Falskt - servern är tillståndslös. Den aktuella positionen finns hos klienten.