Operativsystem ID1200/06

(ID2200/06 6hp)

Tentamen

2019-01-11 14:00-18:00

Instruktioner

- Du får, förutom skrivmateriel, endast ha med dig en egenhändigt handskriven A4 med anteckningar. Anteckningarna lämnas in och kan inte återanvändas.
- 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 och den handskrivna sidan med anteckningar. Inga ytterligare sidor skall lämnas in.

Betyg

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.

För frågor med flera delfrågor ges normalt 2p för alla rätt och 1p för ett fel.

Notera att det av de 24 grundpoängen räknas bara som högst 22 och, att högre poäng inte kompenserar för avsaknad av grundpoäng. Gränserna för betyg är som följer:

- Fx: 12 grundpoäng
- E: 13 grundpoäng
- D: 16 grundpoäng
- C: 20 grundpoäng
- B: 22 grundpoäng och 6 högre poäng
- A: 22 grundpoäng och 10 högre poäng

Gränserna kan komma att justeras nedåt men inte uppåt.

Namn:_____ Persnr:_____

Svar: De angivna svaren är inte nödvändigtvis svar som skulle ge full poäng, längre förklaringar kan vara nödvändigt.

1 Processer

1.1 stack eller heap [2 poäng]

Vad gör proceduren nedan och var skall `gurka` allokeras, på stacken eller *heapen*? Varför? Skriv färdigt koden så att gurka blir allokerat utrymme.

Svar: Proceduren byter värde på de två argumenten. Variabeln skall allokeras på stacken eftersom den har känd storlek och inte behövs efter anropet till *tomat()*. Detta görs genom en lokal deklaration `int gurka;`.

```
void tomat(int *a, int *b) {
  // allocate room for gurka


  gurka = *a;
  *a = *b;
  *b = gurka;
}
```

1.2 fork() [2 poäng]

Vad skrivs ut när vi kör programmet nedan, vilka alternativ finns och varför får vi detta resultat?

```
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

void foo(int *x) {
  (*x)++;
  }

int main() {
  int global = 17;
  int pid = fork();
  if(pid == 0) {
    foo(&global);
  } else {
    foo(&global);
```

Namn:_____ Persnr:_____

```
    wait(NULL);
    printf("global = %d \n", global);
  }
  return 0;
}
```

Svar: Det kommer att skrivas `global = 18` en gång. De två processerna kommer få var sin kopia av stacken och därmed `global`. Eftersom uppdateringarna är oberoende av varandra får båda processerna värdet 18. Det är endast moderprocessen som skriver ut resultatet.

Namn:_____ Persnr:_____

1.3 a stack, a bottle and ... [2 poäng]

Du har skrivit programmet nedan för att undersöka vad som ligger på stacken.

```
void zot(unsigned long *stop ) {
  unsigned long r = 0x3;
  unsigned long *i;
  for(i = &r;  i <= stop; i++){ printf("%p          0x%lx\n", i, *i);  }
}

void foo(unsigned long *stop ) {
  unsigned long q = 0x2;
  zot(stop);
}

int main() {
  unsigned long p = 0x1;
  foo(&p);
 back:
  printf("  p: %p \n", &p);
  printf("  back: %p \n", &&back);
  return 0;
}
```

Detta är utskriften. Förklara vad som finns på de positioner som pekas ut.

Svar: Föregående stackbaspekare (EBP) och returadressen från zot.

```
0x7ffca03d1748      0x3
0x7ffca03d1750      0x7ffca03d1750
0x7ffca03d1758      0xb93d7906926a7d00
0x7ffca03d1760      0x7ffca03d1790        <-------
0x7ffca03d1768      0x55cdac31d78c        <-------
0x7ffca03d1770      0x7ffca03d17d8
0x7ffca03d1778      0x7ffca03d17b0
0x7ffca03d1780      0x1
0x7ffca03d1788      0x2
0x7ffca03d1790      0x7ffca03d17c0
0x7ffca03d1798      0x55cdac31d7c2
0x7ffca03d17a0      0x55cdac31d810
0x7ffca03d17a8      0x12acac31d5f0
0x7ffca03d17b0      0x1
  p: 0x7ffca03d17b0
  back: 0x55cdac31d7c2
```

Namn:_____ Persnr:_____

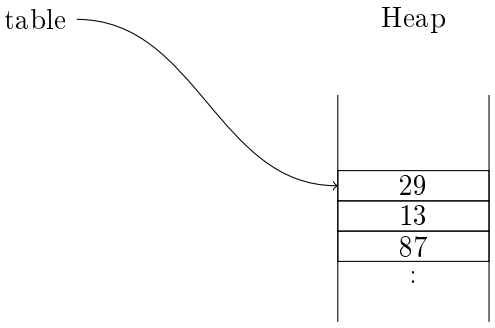
1.4 storleken på blocket? [2 poäng]

Vid implementering av *free()* måste vi veta hur stort block det är som skall frigöras. Hur vet vi storleken? Rita och förklara i figuren nedan hur en implementation skulle kunna se ut.

Svar: Man kan gömma en *header* som ligger innan det utdelade blocket där storleken på blocket skrivas.

```
int *new_table(int elements) {
  return (int*)malloc(sizeof(int)*elements);
}

int main() {)
  int *table = new_table(24);
    :
   table[0] = 29;
   table[1] = 13;
   table[2] = 87;
    :
  free(table);
  return 0;
}
```



Namn:_____ Persnr:_____

1.5 IDTR [2 poäng*]

När man exekverar instruktionen `INT` så sker ett hopp till angiven position i IDT:n. Samtidigt med detta hopp sker en övergång från *user mode* till *kernel mode*. Detta för att operativsystemet skall få full access till dess datastrukturer. Vad förhindrar en användare från att ändra IDTR så att det pekare på en tabell som den själv kontrollerar.

Svar: Att ändra IDTR är en priviligierad instruktion (LIDT) som endast får utföras i *kernel mode*. Om man utför instruktionen i *user mode* så får vi ett avbrott som operativsystemet fångar (via ett hopp i IDT:n). Att operativsystemet har lagt tabellen i *kernel space* skyddar själva tabellen men inte registret i sig.

1.6 biblioteksanrop vs systemanrop [2 poäng*]

Ett operativsystem som implementerar POSIX skall erbjuda viss funktionalitet för en användarprocess. Tillhandahålls detta genom systemanrop, genom bilbliteksrutiner, eller genom en kombination av de båda. Förklara skillnaden mellan systemanrop och biblioteksrutiner och vad som tillhör operativsystemet.

Svar: POSIX tillhandahålls delvis genom systemanrop och delvis genom biblioteksrutiner och tillhör båda operativsystemet. Bibliteksrutiner exekveras i *user space* och kan då arbeta utan ett kostsamt byte av kontext. Eftersom biblioteksrutiner ar snabbare så kan med fördel så mycket som möjligt implementeras där. Bibliteksrutinerna är dock begränsade i det att de inte har tillgång till operativsystemets globala datastrukturer och kan därför inte erbjuda all funktionalitet. Även om inte allt kan hanteras i *user space* så är det, som i fallet malloc/free, en fördel att sköta merparten av arbetet där.

Namn:_____ Persnr:_____

2 Kommunikation

2.1 en buffer [2 poäng]

Vi implementerar en buffer med ett element enligt nedan (get() definierad på liknande sätt). Vi kommer ha flera trådar som producerar och konsumerar från bufferten. Bufferten skyddas med ett lås och trådarna synkroniserar med hjälp av en villkorsvariabel (conditional variable). Programmet nedan fungerar inte (de anrop vi gör till pthread biblioteket är inte ens giltiga), varför fungerar det inte (även om det skulle vara giltiga anrop)?

Svar: Problemet är att vi släpper på låset och därefter suspenderar på villkorsvariablen i två operationer. Om processen blir avbruten mellan dessa operationer kan en annan process hinna: anropa get(), ta låset, plock bort elementet från bufferten och signalera på villkorsvariabeln. Om vi nu suspendera på villkoret så kommer vi aldrig att vakna.

Koden nedan har även ett problem i det att vi inte kommer ha låset om vi vaknar. Vi har även ett problem om vi har flera producenter och konsumenter i det att en signal från en producent väcker en producent iställlet för en konsument.

```
#define TRUE 1
#define FALSE 0

volatile int buffer = 0;
volatile int empty = TRUE;

pthread_mutex_t lock;
pthread_cond_t signal;

void put(int value) {
  pthread_mutex_lock(&lock);
  while(TRUE) {
    if(empty) {
      buffer = value;
      empty = FALSE;
      pthread_cond_signal(&signal);
      pthread_mutex_unlock(&lock);
      break;
    } else {
      pthread_mutex_unlock(&lock);
      pthread_cond_wait(&signal);
    }
```

Namn:_____ Persnr:_____

```
    }
}
```

Namn:_____ Persnr:_____

2.2 pipes [2 poäng]

Programmet nedan öppnar en *pipe* och itererar ett antal omgångar (ITERATIONS) där varje omgång skickar iväg ett antal (BURST) meddelanden ("0123456789"). Vi måste dock hantera situationen där processen som läsar medellanden inte hinner med; hur implementerar vi flödeskontroll så att den buffer vi har inte flödar över?

Svar: Pipes har inbyggs flodeskontroll, vi behöver inte göra någonting.

```
int main() {
  int mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
  mkfifo("sesame", mode);
  // add flow control


  int flag = O_WRONLY;
  int pipe = open("sesame", flag);

  /* produce quickly */
  for(int i = 0; i < ITERATIONS; i++) {
    for(int j = 0; j < BURST; j++) {
      write(pipe, "0123456789", 10);
      // add flow control


    }
    printf("producer burst %d done\n", i);
  }
  printf("producer done\n");
}
```

2.3 SOCK_WHAT [2 poäng*]

När man skapa en *socket* så kan man välja på om det skall vara en *SOCK_STREAM* eller *SOCK_DGRAM*. Vilka egenskaper skiljer dessa och när är det fördelaktigt att välja den ena eller andra?

Svar: Den stora skillnaden är att *SOCK_STREAM* är en dubbelriktad förbindelse av en sekvens av bytes medan *SOCK_DGRAM* är en enkelriktad kanal för meddelanden av begränsad längd.

Fördelen med *SOCK_DGRAM* är att mottagaren läser ett helt meddelande åt gången och behöver inte själv fundera på var det första meddelandet

slutar och nästa börjar. Om meddelanden inte är för stora så är nästan alltid *SOCK_DGRAM* att föredra. Ordningen är dock inte garanterad och inte heller att meddelanden kommer fram. Om detta är viktigt måste man själv ordna ett protokoll för att hålla ordning på meddelanden och begära omsändning av förlorade meddelanden.

Namn:_____ Persnr:_____

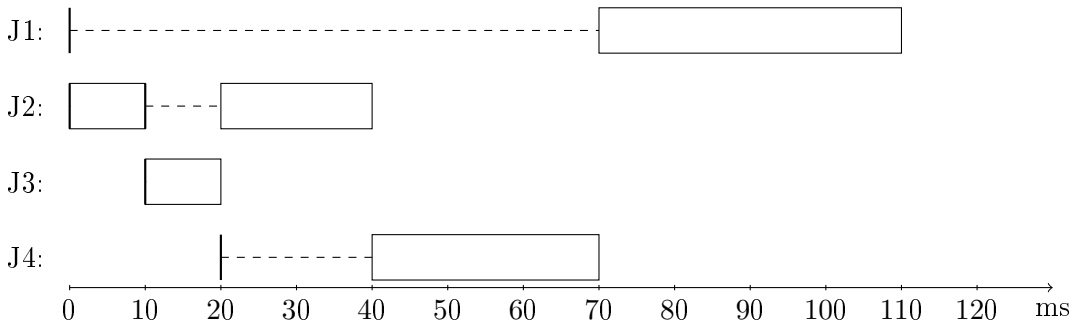
3 Schemaläggning

3.1 Bonnie Tylor [2 poäng]

Antag att vi har en schemaläggare som använder *shortest time to completion first* eller som det också kallas *preemptive shortest job first*. Vi har fyra jobb som nedan anges med ⟨*anländer vid*, *exekveringtid*⟩ i ms. Rita upp ett tidsdiagram över exekveringen och ange omloppstiden för vart och ett av jobben.

Svar:

- J1 : ⟨0,40⟩ 110 ms
- J2 : ⟨0,30⟩ 40 ms
- J3 : ⟨10,10⟩ 10 ms
- J4 : ⟨20,30⟩ 50s

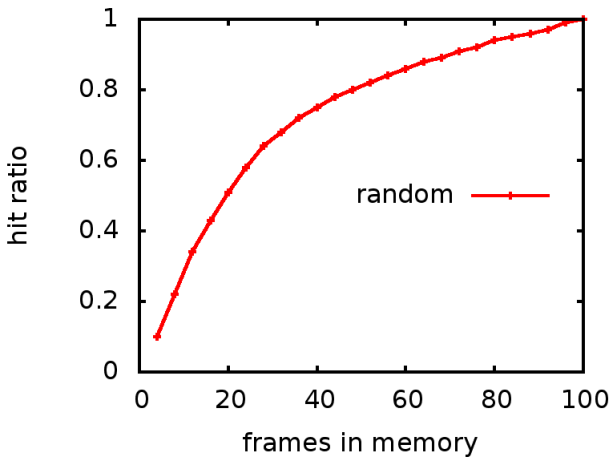


Namn:_____ Persnr:_____

3.2 slumpen inte helt fel [2 poäng]

Vid testkörning av en implementation för swapping så fann man att en helt slumpmässig algoritm fungerade bättre än väntat. I grafen nedan ses kvoten mellan träffar och sidreferenser när man varierar storleken på minnet. Processen som kör använder 100 sidor och de olika körningarna visar träff-kvoten för olika storlekar på minnet.

Man skulle naturligtvis förvänta sig 50% träff när antalet sidor i minnet var hälften av antalet sidor som processen använde men träff-kvoten är bättre än så. Vad skulle en trolig anledning till den bättre träff-kvoten vara?



Svar: EN trolig förklaring är att processen inte refererar till sidorna helt slumpmässigt. Det kan vara så att några sidor refereras oftare eller att sidor ofta refereras upprepade gånger i tätt följd (tidslokalitet).

Namn:_____ Persnr:_____

3.3 omvänd prioritet [2 poäng*]

När man schemalägger processer strikt efter prioritet och samtidigt har lås som processer kan ta, så kan det bli problem. Beskriv hur en högprioriterad process kan få vänta i evighet på lägre prioriterad process fast schemaläggaren garanterar att processer med högre prioritet alltid körs före lägre prioriterade processer.

Svar: Om en process med lägsta prioritet tar ett lås men sedan blir avbruten av processer med högre prioritet som också vill ha låset så kommer den högprioriterade processen vara blockerad i väntan på låset, Så länge det finns processer med prioritet högre än den lågprioriterade så kommer dessa få köra, även om processen med högsta prioritet väntar på låset.

Namn:_____ Persnr:_____

4 Virtuellt minne

4.1 aproximativ vadå? [2 poäng]

Klockalgoritmen som används för att kast ut sidor ur minnet beskrivs som en *approximativ* algoritm. Vad är det den approximera? Beskriv ett scenario där den kanske inte gör det bästa valet, på grund av att den approximerar

Svar: Den approximera LRU (*least recently used*). Den noterar bara att en sida har använts sedan förra vändan, inte hur många gånger eller i vilken ordning. Om det är dags att slänga ut en sida och alla sidor är markerade som använda kommer algoritmen att nolla marköreran en efter en tills den gått ett helt varv runt och sen slänga ut första sidan. Det kan var så att det var den sida som refererades senast och då borde vara den sista som skulle slängas ut.

4.2 implementerad buddy-algoritmen [2 poäng]

Antag att du skall implementera buddy-algoritmen för minnesallokering. Till din hjälp har du en funktion som givet ett block av storlek k hittar dess buddy. Antag att alla block är markerade som antingen fria eller tagna och har ett fält som ger storleken. De fria blocken har även två pekare som länkar in det i en dubbellänkad lista för fria block av samma storlek.

Antag att du skall återlämna ett block (*free*) och hittar dess buddy - vad måste du först förvissa dig om innan du kan slå ihop det återlämnade blocket med dess buddy? Om blocken kan slås ihop, vilka operationer skall du då göra?

Svar: Man måste kontrollera om buddy är **fri** och att det är av **samma storlek**. Om det är fallet så kan blocken slås ihop. Det görs genom att **länka ur buddy**, ta reda på vilket av blocket som är huvudblock och ändra storlek på detta till det dubbla. Man måste sen frigöra det sammanslagna blocket **rekursivt**. När man inte hittar en fri buddy så **länkar man in** det fria blocket först i listan för dess storlek.

Namn:_____ Persnr:_____

4.3 x86_32 sidor på 512 byte [2 poäng*]

Antag att någon av vill skapa en x86-arkitektur för inbyggda system där vi inte behöver en så stor virtuell adressrymd. Du är ombedd att föreslå en arkitektur för det virtuella minnet.

Ordlängden skall var 32 bitar och den skall naturligtvis ha ett sidindelat virtuellt minne. Vi har som krav att sidstorleken skall var 512 bytes så det sätter lite begränsningar på din design.

Hur man skall dela upp en virtuell adress så att man kan använda en hierarkisk sidtabell byggd på sidor om 512 byte? Ge ett förslag på hur en virtuell adress skall avkodas; den virtuella adressrymden måste inte vara fullt 32 bitar.

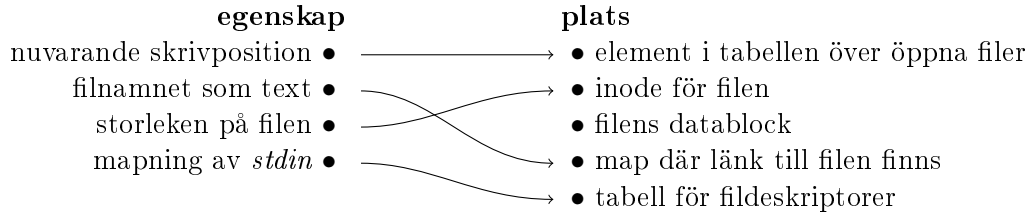
Svar: Ett förslag är att använda ett offset på 9 bitar och sen ha tre nivåer i trädet med index om 7 bitar för varje nivå. Nio bitar som offset är för att kunna adressera en sida på 512 byte. De sju bitarna som index skulle ge 128 element i varje tabell och om vi har fyra byte per element så ger det en tabellstorlek på 512 byte vilket är perfekt. Den virtuella minnesrymden är då $3 * 7 + 9 = 30$ bitar.

Namn:_____ Persnr:_____

5 Filsystem och lagring

5.1 vad ligger var [2 poäng]

En fil har många egenskaper; var återfinns nedan listade egenskaper? Förbind de fyra egenskaperna till vänster med korrekt positionerna till höger. Flera egenskaper kan återfinnas på samma plats men varje egenskap återfinns endast på en plats.



Svar: Som pilarna ovan visar.

5.2 olika typer [2 poäng]

En map kan innehålla länkar av olika typer, beskriv vad dessa fem länkar är av för typ:

```
drwxrwxr-x 2 johanmon johanmon 4096 dec 21 17:43 bar.doc
-rwxrwxr-x 1 johanmon johanmon 8464 dec 21 22:25 cave
lrwxrwxrwx 1 johanmon johanmon    7 dec 21 17:43 foo.pdf -> ./gurka
-rw-rw-r-- 1 johanmon johanmon    7 dec 21 17:42 gurka
prw-r--r-- 1 johanmon johanmon    0 dec 21 22:25 sesame
```

Svar: *bar.doc* är en map, *cave* är en exekverbar fil, *foo.pdf* är en symbolisk (mjuk) länk, *gurka* är en vanlig fil och *sesame* är en *pipe*.

Namn:_____ Persnr:_____

5.3 journalbaserat fs [2 poäng*]

Antag att vi har ett journalbaserat filsystem. Vilket/vilka av följande påståenden nedan är riktiga och vilka är felaktiga - förklara varför.

- Det är viktigt att transaktioner genomförs (checkpoint) i samma ordning som de skapades.
- En transaktion är att betraktad som giltig i den stund den skrivs i fullo i journalen.
- I en återstart efter en crash måste vi vara noga så att vi inte genomför (checkpoint) en transaktion två gånger.

Svar:

- Riktigt: om flera transaktioner rör samma block är resultatet den sist genomförda.
- Riktigt: en halvt skriven transaktion är inte giltig, en fullständig transaktion är giltig. Detta är oavsett om vi har uppdaterat (ceckpoint) de verkliga sektorerna på hårddisken.
- Felaktigt: eftersom operationeran är idempotenta, gör det ingenting om vi utför en transaktion en extra gång (huvudsaken är att de genomförs och att de genomförs i rätt ordning).