# Operativsystem ID1200/06

## Tentamen

## 2018-01-12 14:00-18:00

**Instruktioner**

- Du får, förutom skrivmateriel, endast ha med dig en egenhändigt handskriven A4 med anteckningar.
- 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.

**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.

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:_____

# 1 Processer

## 1.1 vad är problemet? [2 poäng]

Koden nedan kanske fungerar att kompilera men vi gör ett allvarligt misstag. Vilket fel gör vi och vad skulle kunna hända?

```
#include <stdlib.h>

#define SOME 42  // should be  2..47

int *some_fibs() {

  int buffer[SOME];

  buffer[0] = 0;
  buffer[1] = 1;

  for(int i = 2; i < SOME; i++) {
     buffer[i] = buffer[i-1] + buffer[i-2];
  }
  // buffer contains SOME Fibonacci numbers
  return buffer;
}
```

**Svar:** Arrayen `buffer` är allokerad på stacken och kommer troligtvis att blir överskriven redan vid nästa proceduranrop.

Namn:_____ Persnr:_____

## 1.2 minnesmappning [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.

```
> 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.

Namn:_____ Persnr:_____

### 1.3 Arghhh! [2 poäng]

Antag att vi har ett program `boba` som skriver "Don't get in my way" på `stdout`. Vad kommer resultatet bli om vi kör programmet nedan och varför blir det så? (proceduren `dprintf()` tar en fildescriptor som argument)

```
int main() {

  int fd = open("quotes.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);

  int pid = fork();

  if(pid == 0) {
    dup2(fd, 1);
    close(fd);
    execl("boba", "boba", NULL);
  } else {
    dprintf(fd, "Arghhh!");
    close(fd);
  }
  return 0;
}
```

**Svar:** I `dup2(fd,1)` sätter vi om `stdout` till den öppnade filen. Boba kommer då att skriva sin rad till filen `quotes.txt`. Samtidigt kommer moderprocessen att skriva "Arghhh!" på samma fil. De två processerna kommer dock att dela på samma representation av den öppnade filen, dela position och samsas om att skriva. Vi kommer därför se en blandning av de båda texterna i filen `quotes.txt` i.e. den ena texten kommer inte skriva över den andra.

Namn:_____ Persnr:_____

### 1.4 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? Visa med en ritning vilken information som används och hur operationen skulle utföras.

**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.

Det finns två, icke uteslutande, fall som vi måste hålla reda på 1/ adressen på ett block i listan plus dess storlek är lika ned de friade blockets storlek 2/ det friade blockets storlek plus dess storlek är lika med nästa blocks adress.

Namn:_____ Persnr:_____

### 1.5 intern paging [2 poäng*]

När vi implementerar minneshantering internt för en process (till exempel med malloc()) så använder vi en form av segmentering. Det är därför vi kan få problem med extern fragmentering. Om det är bättre med så kallad paging, varför använder vi inte paging då vi implementerar intern minneshantering?

**Svar:** Vi skulle behöva implementera en adressöversättare som i varje referens delade upp en adress i sida och offset. Adressen till sidan skulle sen behöva omvandlas till en ram-adress med hjälp av en omvandlingtabell. Vi skulle med all säkerhet behöva arbete med väldigt små sidor för att inte få för stor intern fragmentering. Detta skulle ge en stor omvandlingtabell som skulle vara svår att hantera. Att göra detta i mjukvara utan stöd från hårdvara i processorn är alldeles för kostsamt.

Ett alternativ skulle vara att bara dela ut block av en mycket liten storlek, säg 16 byte (stor nog för två pekare), och låta processen representera alla objekt med hjälp av dess. Inte omöjligt och det är nästan så en del listbaserad programmeringsspråk hanterar sitt minne.

Namn:_____ Persnr:_____

### 1.6 context [2 poäng*]

Med hjälp av bibliotektsanropet `getcontext()` kan en process spara undan sitt eget så kallade *context*. Vi skulle kunna bygga upp ett bibliotek som lät oss skapa nya exekverande trådar och växla mellan dessa manuellt genom att en exekverande tråd anropade en schemaläggare.

Varför skulle vi vilja bygga upp ett liknande bibliotek, finns det några fördelar? Vad skulle nackdelarna vara?

**Svar:** Vi skulle får ett trådbibliotek där trådarna hanterades av processen själv. Ett byte mellan två trådar skulle ta mindre tid än det skulle ta om vi lät operativsystemet växla mellan trådarna (som sker i pthread-bibioteket). Vi skulle kunna undvika många synkroniseringsproblem eftersom vi skulle veta att bara en tråd exekverade åt gången.

En nackdel skulle vara att vi inte skulle kunna utnyttja en processor med flera kärnor. Vi skulle också bli helt blockerade om en av våra trådar gjorden en I/O-operation.

Namn:_____ Persnr:_____

# 2 Kommunikation

## 2.1 count [2 poäng]

Vad kommer skrivas ut om vi exekverar proceduren `hello()` nedan samtidigt i två trådar? Motivera svaret.

```
int loop = 10;

void *hello() {
  int count = 0;

  for(int i = 0; i < loop; i++) {
      count++;
  }
  printf("the count is %d\n", count);
}
```

**Svar:** Varje tråd har sin version av `count` och de två trådarna kommer alltså inte att störa varandra. Varje tråd kommer därför att skriva ut the count is 10".

Namn:_____ Persnr:_____

## 2.2 pipes [2 poäng]

Om vi har två processer, en producent och en konsument, som kommunicerar via en s.k. *pipe*. Hur kan vi då förhindra att producenten skickar mer information än vad konsumenten kan ta emot och därmed får systemet att krascha?

**Svar:** *Pipes* har en inbyggd flödesontroll. Om konsumenten inte väljer att läsa kommer producenten att bli suspenderad när den försöker skriva.

Namn:_____ Persnr:_____

### 2.3 namnrymd [2 poäng*]

Nedan är kod där vi öppnar en socket och använder namnrymden `AF_INET`. Vi kan då adressera en server med hjälp av portnummer och IP-adress. Det finns en annan namnrymd som vi kan använda när vi arbetar med socket. Nämn en och beskriv vilka för och nackdelar den kan ha.

```
struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_port = htons(SERVER_PORT);
server.sin_addr.s_addr = inet_addr(SERVER_IP);
```

**Svar:** Vi kan använda filnamn (`AF_UNIX`) som namnrymd. Detta har nackdelen att vi inte kan nås från andra noder i ett nätverk utan är begränsade till den nod vi kör på. Det kan naturligtvis också vara en fördel då vi slipper exponera oss för omvärden. Implementeringen kan också vara mer effektiv eftersom vi inte behöver använda oss av t.ex. TCP.

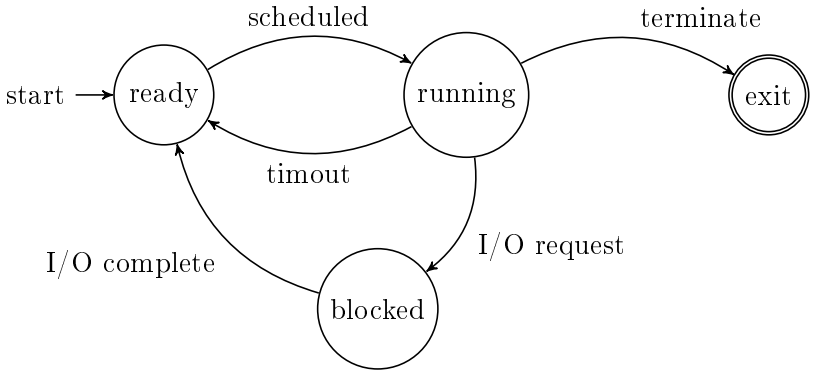
Namn:_____ Persnr:_____

# 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:**




## 3.2 reaktionstiden [2 poäng]

När vi vill minska reaktionstiden så kan vill vi helst kunna avbryta jobba även om de inte är klara. Om vi gör detta så har vi en parameter som vi kan välja, genom att anpassa denna så kan vi förbättra reaktionstiden. Vad är det för parameter som vi kan sätta? Hur skall vi förändra den och vilka oönskade konsekvenser kan det få?

**Svar:** Vi kan minska på den tid som varje jobb tillåts köra innan vi byter till nästa jobb. Detta medför att jobb som är klara att köra mycket snabbt blir schemalagda. Nackdelen är att vi förlänger den genomsnittliga omloppstiden och i värsta fall spenderar en stor del av tiden på att växla mellan processer.

Namn:_____ Persnr:_____

### 3.3 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. Om vi antar att deadline är lika med fulla perioden hur kan vi då beskriva ett systems belastning?

Har vi några garantier för att schemaläggningen kommer att fungera dvs att inga deadlines kommer att missas?

**Svar:** Ett systems belastning är summan av $e_i/p_i$. Där $e_i$ är ett jobbs exekveringstid och $p_i$ dess periodicitet. Schemaläggaren kommer garnterat att fungera om lasten är under $n * (2^{1/n} - 1)$ där $n$ är antalet processer (ca: 69% belastning för stor $n$). Den kan fungera vid högre last men vi har inga garantier.

Namn:_____ Persnr:_____

# 4 Virtuellt minne

## 4.1 segmentering [2 poäng]

När man använder segmentering för att hantera fysiskt minne så kan man få problem med extern fragmentering. Detta undviks om man istället använder s.k. paging. Varför kan vi undvika extern fragmentering med hjälp av paging? Är det något vi riskerar?

**Svar:** Eftersom alla ramar är lika stora och en process kan tilldelas vilken ram som helst så kan en ram alltid användas. Det finns med andra ord inga luckor som är för små för att användas. Om sidstorleken är för stor och begärda segement är små riskerar vi att öka intern fragmentering.

Namn:_____ Persnr:_____

## 4.2 nästan rätt [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. Hur skulle vi kunna aproximera algoritmen och vad skulle det ha för konsekvenser? Kan delar av algoritmen implementeras i hårdvara?

```
    :
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. Om vi nöjer oss med att enbart registrera om en sida har använts eller inte sedan vi senast kontrollerade så kan detta hanteras med en bit i ett sidtabellspost (page table entry). Detta kan skötas, och sköts, i hårdvara. När det är dags att välja en sida som kan plockas bort går man igenom en cirkulär struktur och letar efter en sida vars värde är noll. Sidtabellsvärden som är satt till ett, sätts till noll. Detta kommer ge sidorna en ändra chans".

Namn:_____ Persnr:_____

### 4.3 x86_64 adressering [2 poäng*]

I en x86-processor i 64-bitarsmode så innehåller ett PTE en ramadress på 40 bitar. Denna kombineras med den virtuella adressens offset på 12 bitar till en fysisk adress. Detta blir 52 bitar men en process har endast 48-bitars virtuellt minne. Vilken fördel får vi genom att ha en 52-bitars fysisk adress?

**Svar:** En enskild process kan visserligen inte adressera mer än 48-bitars adressrymd men vi kan ha flera processer i minnet samtidigt. Om vi var begränsade till 48-bitars fysiskt minne skulle vi inte kunna ha mer än 64 Gibyte i RAM.

Namn:_____ Persnr:_____

# 5 Filsystem och lagring

## 5.1 lista innehållet i en map [2 poäng]

Om vi vill lista innehållet i en map så kan vi använda oss av biblioteksrutinen `opendir()`. Vilken information kan vi direkt få ut av strukturen som pekas ut av `entry` i koden nedan? Beskriv tre viktiga egenskaper. Vilka egenskaper av en **fil** kan vi inte hitta direkt från strukturen och var kan vi hitta dessa?

```
int main(int argc, char *argv[]) {

  char *path = argv[1];

  DIR *dirp = opendir(path);

  struct dirent *entry;

  while((entry = readdir(dirp)) != NULL) {

    //  what information do we have?

  }
}
```

**Svar:** Det vi kan hitta direkt är: namn, typ och inod-nummer. Typen kan vara: fil, map, mjuk länk mm. Alla egenskaper (storlek, skapad, ändrad, ägare etc) för en fil måste vi hämta in filens inod.

## 5.2 ta bort en fil [2 poäng]

Om vi använder kommandot `rm` så tar vi inte bort en fil utan bara en hård länk till en fil. När försvinner själva filen? Hur hanteras detta?

**Svar:** Varje inode inhehåller information om hur många hårda länkar som finns till filen. När vi tar bort den sista länken så kommer filen att att tas bort (dess data kommer ligga kvar på disk men är inte åtkomligt via filsystemet).

Namn:_____ Persnr:_____

### 5.3 loggbaserade fs [2 poäng*]

I ett loggbaserar filsystem skriver vi alla förändringar i en kontinuerlig logg utan att göra förandringar i de redan existerande block som en fil har. Vad är poängen med att hela tiden skriva nya modifierade kopior av datablock istället för att gå in och göra de små förändringar som vi vill göra? Om det är bättre, är det något som blir sämre?

**Svar:** Genom att hela tiden skriva i slutet på loggen behöver vi inte röra skrivhuvudet. Om vi skall skriva på alla enskilda block så måste vi föra skrivhuvudet fram och tillbaks vilket kommer vara en stor nackdel. Vi betalar priset när vi skall läsa en fil som i värsta fall nu är utspridd över hela disken.