Lösningsförslag ID2206/ID2200/IS1350 Operativsystem
Torsdagen 2013-05-23 kl 1400-1800

SCS/ICT/KTH

Examinator: Robert Rönngren

Hjälpmedel: Inga
Tentamensfrågorna behöver inte återlämnas efter avslutad tentamen.

Ange på omslaget vilken kurs du tenterar och vilken termin du läste kursen.

Varje inlämnat blad skall förses med följande information:
- Namn och personnummer
- Nummer för behandlade uppgifter
- Sidnummer

ID2200: Studenter som tenterar ID2200 besvarar inte frågor markerade med *****
IS1350: Studenter som tenterar IS1350 besvarar inte frågor markerade med **¤**

Rättning:
- ***Alla svar*, även på delfrågor, måste ha åtminstone en kortfattad motivering för att ge poäng**
- Resultat beräknas finnas rapporterat i LADOK 13/6

Betygsgränser:
- Godkänt (E) garanteras från 50% av den maximala möjliga poängen för respektive kurs

Lösningsförslag: anslås på kursens webbsida efter tentamen (antagligen inte förrän 27/5)

---

1) Antag att du skall utveckla en applikation i C som skall gå att köra både på BSD Unix och Linux plattformar. Vad bör du tänka på för att applikationen skall bli så portabel som möjligt och vad är det minsta du måste göra för att flytta den från den ena plattformen till den andra (förutom att föra över filer). (1p)

   Man bör hålla sig till en standard för C (t.ex ANSI, C99) som stöds på båda plattformarna, vilket t.ex innebär att man inte kan anta att tal representeras annat än som standarden för språket föreskriver och man bör välja en standard för systemanropen (t.ex POSIX) som stöds på båda plattformarna. Applikationen måste kompileras om på respektive plattform.

2) Kalle skall skriva ett program som skall utföra motsvarande följande kommando till kommandotolken: **`cat`** ***`filnamn`*** **`|`** **`sort`** **`|`** **`less`** där *`filnamn`* ges som ett kommandoradsargument till programmet. Kalle har föreslagit följande struktur för sitt program:

   1) Skapa en pipe "P"
   2) Skapa en process "A" som vars stdout kopplas till pipen "P"s skrivände och som kör "cat *filnamn*"
   3) Vänta på att "A" terminerar korrekt - om "A" terminerar med fel så avbryts hela exekveringen
   4) Starta en process "B" vars stdin kopplas till pipen "P"s läsände, stdout kopplas till "P"s skrivände och som kör "sort"
   5) Vänta på att "B" terminerar korrekt - om "B" terminerar med fel så avbryts hela exekveringen
   6) Starta en process "C" vars stdin kopplas till pipen "P"s läsände och som kör "less"
   7) Vänta på att "C" terminerar
   8) Stäng pipen "P"s båda ändar
   9) Terminera

   Analysera Kalles lösning med avseende på styrkor och svagheter! (2p)

   Kalles lösningsförslag har minst tre fel: Man bör stänga alla oanvända ändar på pipen i alla processer eftersom man annars riskerar att processer aldrig kommer att läsa EOF då någon som inte skriver till pipen ändå har den öppen. Man kan inte starta en process (A) som skriver till pipen och invänta att den terminerar utan att starta en process som läser från pipen eftersom den skrivande processen kan fylla pipen och om pipen blir full kan den skrivande processen (A) inte fortsätta skriva. Det skulle kunna leda till deadlock eftersom skrivning till en full pipe blockerar. Man måste också skapa en separat pipe för varje par av processer att kommunicera via, dvs en separat pipe för varje pipetecken (|) i kommandot eftersom man inte kan blanda kommunikation från flera källor till flera mottagare i en pipe. I Kalles lösning kommer process B att ha pipen öppen för både läsning och skrivning det gör att den aldrig kommer att läsa filslut (EOF) (eftersom den själv kan skriva till pipen) och därför aldrig kommer att kunna upptäcka att den fått all indata - dvs. den kommer att låsa sig.

3) Förklara hur OS:et hanterar vart och ett av nedanstående för en process då den termineras till följd av ett exekveringsfel, som t.ex. ett SEGMENTATION FAULT, i ett UNIX/LINUX system:
   i) Virtuell adressrymd (0.5p)
   ii) Fysisk adressrymd (0.5p)

iii) Öppna filer (0.5p)
iv) Post i processtabellen (0.5p)

Fundera över hur varje del implementeras:
i) Den försvinner när processen termineras vilket innebär att om den har en egen sidtabell så tas den bort, alternativt om man har en gemensam sidtabell så markerar systemet på något sätt att det här process id är ogiltigt och ev. block allokerade för data på swaparean frigörs.
ii) Eventuella ramar som allokerats för processen markeras som lediga i ramtabellen.
iii) Öppna filer "stängs" och ev data i in- och utmatningsbuffertar försvinner när buffertarna frigörs.
iv) Posten i processtabellen ligger kvar till dess att någon process läst av den via wait(2), den uppdateras också så att statusinformationen indikerar varför processen terminerades.

4) Förklara varför man kan få en uppsättning processer, att totalt sett bli klara snabbare om man, på en dator med en-kärnig processor, kör dem pseudo-parallellt än om man skulle köra varje process i sekvens. (1p)

Det generella svaret är att om man kan överlappa beräkningar med I/O kan den totala tiden (wall-clock) reduceras genom att processerna i praktiken kommer att köras parallellt på CPU och av i/o enheterna.

5) Många OS använder eller har använt sig av pre-emptiv (avbrytande) schemaläggning med fixa time-slices. Diskutera vad man bör ta hänsyn till när man bestämmer längden på en time-slice (antag att man använder en och samma längd på time-slice för alla processer). (1p)

Time-slice bör inte vara för lång eftersom systemet kan upplevas ha dålig responstid (i extremfallet skulle time-slice kunna sättas till en längd som skulle kunna upplevas som oändlig - dvs. om en process med hög prioritet inte utför något avbrytande systemanrop skulle den i praktiken kunna vara den enda som får exekvera) Den bör heller inte vara så kort att tiden för att växla mellan processer (context-switch) blir för dominerande (dvs. att för stor andel av tiden skulle ägnas åt context-switching istället för att exekvera processerna)

OBS! om en process blir klar innan den utnyttjat hela sin time-slice så kommer den inte att blockera processorn till dess att time-slicen tagit slut (jfr. fråga 4)

6) Copy-on-write är en viktig teknik i många OS. Förklara vad den går ut på och exemplifiera genom att förklara hur den används i samband med systemanropet `fork()` i moderna LINUX/UNIX system. (1p)

Det innebär att två eller flera processer delar dataareor men om någon av de delade dataareorna uppdateras (skrivs till) så skapas privata kopior av den uppdaterade data-arean för varje process. Tekniken används ofta i samband med systemanropet fork() så att barn och förälder delar sidor men om någon av dem skriver till en sida så skapas privata kopior av sidan. Det gör att man kan spara tid och minne jämfört med att kopiera förälderns adressrymd till barnet när barnet skapas. Speciellt som man kan anta att barnet i de flesta fall kommer att exekvera ett annat program än föräldraprocessen. Observera att om förälder och barn delar öppna filer så kopieras inte dessa vid skrivning!

7) Förklara vad som händer och varför det händer när man exekverar följande program: (1p)

```
static int main(int argc, char **argv)
{
   execlp("ls", "ls", "-la", NULL);
}
```

Det kommer inte att exekveras eftersom main() är deklarerad som static och borttaget ur symboltabellen. Det medför att run-time systemet (os:et) inte kan hitta var exekveringen skall starta.

8) a) Förklara begreppen trådar på användarnivå (user-level threads) och kärntrådar (kernel threads) (1p)
b) Diskutera de främsta för- och nackdelarna med att använda trådar på användarnivå respektive kärntrådar (2p)

i) trådar på användarnivå implementeras helt och hållet inom en process och är inte kända, och kan därför inte heller schemaläggas, av operativsystemet. Kärntrådar implementeras/stöds av operativsystemet och kan därför schemaläggas av OS:et (OBS! användaren kan normalt skapa kärntrådar pss. som han/hon kan skapa processer i sina applikationer).
ii) Den enda egentliga fördelen med att använda trådar på användarnivå är att applikationerna kan vara lättare att flytta mellan olika system om trådbiblioteket enkelt kan flyttas.
Nackdelarna är att om en tråd inom processen blockerar på t.ex I/O kan inte OS:et schemalägga någon annat tråd inom processen utan blockerar hela processen. Eftersom OS:et inte känner till trådarna kan det heller inte schemalägga dem på olika processorkärnor om man har en flerkärnig maskin. En annan ***stor*** nackdel är att programmeraren normalt själv får sköta schemaläggningen genom att anropa funktioner för att växla mellan/invänta trådar.

För kärntrådar gäller att OS:et känner till trådarna och sköter schemaläggningen. Det betyder att om en tråd blockerar så behöver inte alla trådar i processen, dvs hela processen blockeras och man behöver inte som användare/programmerare explicit anropa schemaläggaren för att växla mellan trådar. Kärnan bör också kunna utnyttja om det finns fler processorkärnor så att trådarna schemaläggs på olika kärnor (sann parallellitet). Den enda egentliga nackdelen är att det kan vara mer komplicerat att flytta applikationer mellan olika plattformar om man inte följer en standard för trådningen.

9) a) Förklara begreppen swaping och paging. (1p)
b) Behöver man implementera båda mekanismerna på ett system med virtuellt sidindelat minne? (1p)

a) swaping innebär att hela processer flyttas mellan primärminne och backing-store (swaparean - typiskt disk) det är en typ av *schemaläggningsåtgärd* då processer som finns i primärminnet också kan schemaläggas för exekvering medan processer på backing-store inte kan schemaläggas för exekvering. Paging innebär att man använder sidindelat minne och att man flyttar sidor till/från primärminne och disk(page-in/page-out).
b) Normalt sett ja. Vid överlast vill man kunna minska antalet aktiva processer. Det kan ske genom att man pagear ut all sidor för dessa processer - men enklare är att ha en swap mekanism.

10) a) Beskriv vad en TLBär. (1p)
b) Beskriv vad som händer vid en TLB-miss. (2p)'

a) En Translation Lookaside Buffer är en del av MMUn och fungerar som en cache för de mest frekventa översättningarna från <PID, sidnr> till <ramnr>. Dvs den cachar del av processernas sidtabeller. Avsikten är att snabba upp adressöversättningen från virtuell adress till fysisk adress. Den implementeras oftast med hjälp av associativt minne i vilket man kan göra en sökning efter innehållet på en (eller ett fåtal) clockcykler.
b) TLB-miss innebär att den sökta översättningen från PID,sidnr till ramnr inte finns cachad i TLBn. Normalt görs ett TRAP till OS:et som slår upp översättningen i sidtabellen (det kan innebära att man tvingas hantera ett sidfel) och som uppdaterar informationen i TLBn.

11) Förklara begreppen minor pagefault (kallas ibland också soft miss) och major pagefault (kallas ibland också hard miss) och när de kan uppstå. (2p)

Vi antar att det handlar om att vi försöker accessa en sida som vi har rätt att accessa. Sidfel uppstår när informationen i sidtabellen indikerar att sidan inte finns inladdad i primärminnet, dvs. inte finns i en ram. När man hanterar ett sidfel måste man alltså se till att den sida som accessas kommer att finnas tillgänglig i primärminnet i en ram och att sidtabellen uppdateras. Vid ett major pagefault så måste den eftersökta sidan läsas in från backing store (swap arean på disken) till promärminnet, eventuellt måste man också frigöra en ram genom att skriva ut den sidan (om sidan är modifierad) som ligger i ramen till backing store (swap arean). Ett minor pagefault involverar inte någon diskaktivitet. Det kan ske vid ett av följande tillfällen: i) paging demonen har tagit en ram från en process för att lägga till listan med lediga ramar som används för att snabbt behandla sidfel. Om den ramen inte använts för att lägga in en annan sida i, dvs. ramen finns i listan med lediga ramar, så finns den sökta sidan kvar i ramen och processen kan få tillbaka ramen med sidan i utan någon diskaktivitet. ii) Om en process behöver en ny nollställd sida, t.ex för att utvidga någon dataarea eller stacken, och det finns en ledig ram eller man kan hitta en ren victim-page, så kan man hantera sidfelet utan diskaktivitet.

12) Om exekveringen fortsätter med följande referenssträng: 7, 2, 1, 2, 0, 3, 1, 4, 2 Vilken av sidorna i nedanstående tabell, som visar innehållet i de fyra tillgängliga ramarna, kommer i så fall att bytas ut först för sidutbytesalgoritmerna:
i) FIFO iii) NRU (0.5p 0.5p)
ii) LRU iv) OPT (0.5p 0.5p)

Sidnr	Tid då sidan laddades in	Tid för senaste referens	R	M
0	245	351	1	1
1	197	267	0	1
2	203	255	0	0
3	262	312	1	0

För att få poäng måste man i svaret förklara varför de olika sidutbytesalgoritmerna väljer en specifik sida som victim page. i) 1, ii) 2, iii) 2, iv) 3

13) I tabellen nedan finns ett program och resultat från två körningar på ett 64-bitars LINUX system. Förklara:
i) värdena på den lokala variabeln som skrivs ut (1p)
ii) adresserna för den lokala variabeln som skrivs ut (1p)

```
void fun(void)
{ int local;
  printf("&local %u, local %u\n", &local, local);
  local = 0;}

int main(int argc, char **argv)
{ fun();
  fun();}
```

```
[rron@subway tmp]$ ./a.out
&local 2213170780, local 58
&local 2213170780, local 0
[rron@subway tmp]$ ./a.out
&local 859630924, local 58
&local 859630924, local 0
```

i) lokala variabler i C initieras inte. Det betyder att första gången fun() anropas så ligger det ett värde i minnet på den position på stacken som används för att lagra variabeln local som är 58 (i vårt fall antagligen en rest från vad som skett vid uppstart av processen innan main() anropats. Värdet i de byten på stacken sätts till 0. Vid nästa anrop till fun() hamnar aktiveringsposten för fun() på exakt samma positioner på stacken (i den virtuella adressrymden) som vid det första anropet med den skillnaden att de bytes som används för den lokala variabeln local nu har värdet 0.
ii) Adresserna som skrivs ut är virtuella adresser (och har alltså inget att göra med i vilka ramar processens sidor är inladdade) I en så stor adressrymd som en 64-bitars adressrymd är det inte rimligt att anta annat än att en process bara kommer att utnyttja en mindre del av adressrymden på en vanlig dator. Linux slumpar därför ut var man lägger olika dataareor som stack och heap för att göra det svårare för malware (virus etc.) att hitta dessa och kunna utnyttja dem vid olika typer av attacker. Det är alltså en säkerhetsmekanism som gör att stacken hamnar på olika adresser i den virtuella dressrymden i olika exekveringar.

14) Antag att du har ett datorsystem där man använder register för att styra I/O. Resonera kring om det är rimligt att dessa register finns inmappade i adressrymden åtkomliga från user-mode respektive kernel-mode. (1p)

Registren för att styra I/O, dvs data- och kontrollregister på I/O enheterna, måste finnas inmappade i OS:ets (kärnans) adressrymd för att man skall kunna styra I/O från kärnan (normalt drivrutinerna i kärnan). De får inte finnas inmappade i användarens (user mode) adressrymd eftersom en användarpocess då direkt skulle kunna göra I/O utan de säkerhetskontroller som finns implementerade i operativsystem och filsystem. Kontroller som sker t.ex vid access av en fil om man använder systemanrop för att läsa/skriva filen. Man kan också få problem med att I/O-operationer som utförs av olika processer inte kan samordnas.

15) Du skall skriva ett flertrådat program där trådarna kan behöva ha tillgång till en eller flera kritiska sektioner samtidigt. De kritiska sektionerna skyddas genom ömsesidig uteslutning implementerat med ett binärt lås för varje kritisk sektion. Vad bör du tänka på när du skriver koden för att undvika deadlock? (1p)

Man bör säkerställa att man bryter minst ett av Coffmans villkor för att garantera att man inte hamnar i deadlock. i det här fallet bör man lämpligen numrera alla kritiska sektioner så att var och en får ett unikt nummer. Om en process/tråd behöver accessa flera kritiska sektioner samtidigt måste dessa låsas i stigande nummerordning. På så sätt bryter man villkoret att det måste kunna uppstå cirkulär väntan.

16) Antag att du startar upp applikationen XYZ. Vilka rättigheter har applikationen när den exekveras? (1p)

Om inte applikationen har getts andra rättigheter (via t.ex setuid() som gör att man kan sätta med vilken användaridentitet applikationen skall exekvera oavsett vilken användare som kör den) så kommer applikationen att exekvera med exakt samma rättigheter som den användare som startade den. Skälet är enkelt - det en användare kan göra på en dator är att starta olika typer av applikationer som kommandotolk och andra program. Om man som användare har rättigheter i systemet så är det alltså dessa rättigheter man skall använda när man startar upp applikationer.

17) Beskriv vilka operationer som måste göras internt i ett UNIX -filsystem när man tar bort en fil (hårdlänk räknaren för filen går ner till noll) (1.5p)

i) ta bort posten (entryt) i katalogen som namnger filen och pekar ut dess inod
ii) lämna tillbaka alla diskblock som använts för filen till filsystemet och lägga in den i listan med lediga block
iii) avallokera inoden och lämna tillbaka den till filsystemet

18) ¤Förklara vad sessionssemantik innebär i ett distribuerat filsystem? (1p)

Det innebär att ändringar som gjorts i en lokal kopia av en fil bara skrivs tillbaka då filen stängs. (kallas också upload/download semantik)

19) *Realtidssystem görs ofta resurstillräckliga, dvs. jobb kan alltid få de resurser de behöver för sin exekvering. Är inte detta ett slöseri med resurser? Förklara varför man ändå gör så här i många fall. (1p)

Om det inte finns tillräckligt med resurser i ett system måste de delar som blir kritiska sektioner, dvs. resurser som kan accessas och uppdateras av flera jobb (processer) samtidigt, skyddas som kritiska sektioner. Man vet att schemaläggningen i ett system där jobben synkroniserar via semaforer (lås) är ett NP-komplett problem i det allmänna fallet, dvs. det kan göra det praktiskt omöjligt att hitta en schemaläggning som gör att alla jobb alltid kommer att klara sina deadlines även om en sådan skulle existera.

20) *a) Förklara Jackssons regel för schemaläggning av realtidsystem på en processorkärna och vad den garanterar. (1p)
*b) Vad är motsvarande regel på multi-core system? (1p)

a) En schemaläggning enligt Jacksons regel, dvs. att man har en avbrytande schemaläggning där den process som har närmast till sin deadline får högst prioritet, är optimal. (Dvs om systemet går att schemalägga så går det också att schemalägga enligt Jacksons regel).
b) Jacksons regel ger inte garanterat optimala schemaläggningar på multi-core system. Det finns heller ingen generell regel som garanterar det på multi-core system.

21) I moderna OS finns ofta möjligheten att ha sidor i den virtuella adressrymden i flera olika tillstånd som brukar motsvara: i) Ännu ej inmappad, ii) inmappad eller iii) reserverad för framtida bruk.
a) Vad händer om man försöker accessa en ännu ej inmappad sida? (0.5p)
b) Ge minst ett exempel på varför man kan vilja reservera sidor för framtida bruk. (0.5p)

Jämför med hur t.ex sidor hanteras i Mocrosofts OS (inmappad = comitted).
a) Det ger ett SEGMENTATION FAULT eftersom processen ännu inte har rättigheter att accessa sidan. (Jfr. t.ex hur malloc() fungerar då man behöver utöka storleken av heapen)
b) Ett exempel kan vara att man vill reservera utrymme för stackar för trådar som kommer att skapas i framtiden

"Kreativitet är smittsamt!."
A. Einstein