Linköpings universitet
Institutionen för datavetenskap 2018-05-30

Tentamen i
TDIU16 Process- och operativsystemprogrammering

Datum 2018-05-30

Tid 14-18

Institution IDA

Kurskod TDIU16

Provkod TEN1

Examinator

Klas Arvidsson (klas.arvidsson@liu.se)

Administratör

Madeleine Häger Dahlqvist

Jourhavande lärare

Klas Arvidsson (013-28 21 46)
Filip Strömbäck (013-28 27 52)

Tillåtna hjälpmedel

Inga hjälpmedel.

Instruktioner

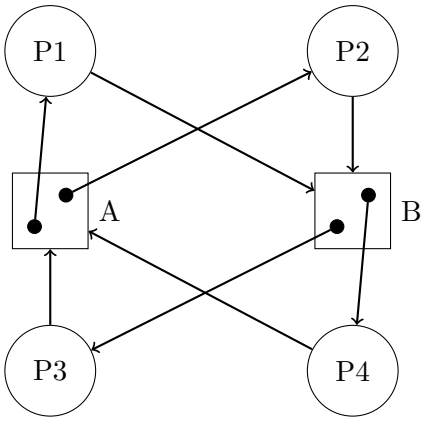
- Ange din tolkning av frågan och alla antaganden du gör.
- Var precis i dina påståenden. Bevisa ditt resonemang när möjligt. Svävande eller tvetydiga formuleringar leder till poängavdrag.
- Motivera tydligt och djupgående alla påståenden och resonemang. Förklara uträkningar och lösningsmetoder.
- Sträva alltid efter maximal parallellism när du synkroniserar kod. Om du anser att lösningen med maximal parallellism är sämre än en annan lösning, motivera varför.
- Frågor om tentan ställs genom tentaklienten.
- Tentamen har 7 uppgifter på 8 sidor (inklusive denna sida).
- Tentamen är på 30 poäng och betygsätts U, 3, 4, 5 (prel. gränser: 16p, 22p, 26p). Poäng ges för motiveringar, förklaringar och resonemang. Enbart rätt svar ger inte (full) poäng.
- Lycka till!

Inlämning och givna filer

- Skriv dina lösningar i en textfil (`.txt`), som källkod (`.c`) eller i OpenOffice (`.odt`). Namnge filerna utefter vad som anges i uppgifterna.
- Filerna skickas in med hjälp av tentaklienten. Gör en inskickning per numrerad uppgift (kom ihåg att välja rätt uppgiftsnummer när du skickar in). Systemet svarar sedan med ett meddelande om allt gick bra. Om du kommer på att du gjort fel vid inskickningen, eller vill ändra dina svar kan du skicka in samma uppgift igen, så rättas den senaste inskickningen.
- Tentan rättas i efterhand. Ni kommer alltså inte få svar på inlämnade uppgifter under tentans gång.
- Inlämnad kod behöver **inte** kompilera för att ge poäng. Det viktiga är att vi förstår vad ni försöker åstadkomma. Det är alltså okej att fylla i eventuella detaljer ni inte kommer ihåg, eller inte får att kompilera med pseudokod och/eller beskrivande kommentarer.
- Kopiera de givna filerna till er hemmapp innan ni försöker redigera dem. Gör det med kommandot `cp -r given_files/* ~/`
- I slutet av de givna filerna finns ett testprogram som visar hur koden kan användas. Testprogrammet är **inte** en del av uppgiften och kommer därför inte att rättas. Eventuella modifikationer ni gör till testprogrammet kommer alltså att ignoreras, så det är ingen vits att synkronisera där.
- Testprogrammen i den givna koden finns bara där för att det ska gå att kompilera och köra koden i uppgifterna. De är inte gjorda för att påvisa eventuella synkroniseringsfel i koden. Det är såklart tillåtet att modifiera testprogrammen ifall ni vill försöka hitta problem, men det är dock **inte** en del av tentan och kommer inte påverka bedömningen av uppgiften.
- Testprogrammen kan kompileras med hjälp av den givna makefilen. Kommandot `make <filnamn>` kompilerar filen `<filnamn>.c` och skapar en körbar fil `<filnamn>`. För att kompilera filen `uppgift3.c` kör du alltså `make uppgift3`. Se till att kopiera `Makefile` från `given_files` ifall kommandot misslyckas!

1. I ett system körs fyra processer, *P1, P2, P3* och *P4*. Figur 1 innehåller en resursallokeringsgraf för systemet. Är systemet i deadlock? Motivera ditt svar! [2p]

   Lämna in som `uppgift1.txt` *eller* `uppgift1.odt`*.*



Figur 1: Resursallokeringsgraf för ett system.

2. I ett system körs tre processer, *P1, P2* och *P3*. I systemet finns det också tre resurser, *A, B* och *C*. Det finns 4 st av resurserna *A, B* och *C*. Tabell 1 visar hur många resurser varje process har för närvarande, och Tabell 2 och det maximala resursbehovet för varje process. Systemet är i ett säkert läge.

   Lämna in svar på (a) och (b) som `uppgift2.txt` *eller* `uppgift2.odt`*.*

	A	B	C
P1	0	1	1
P2	1	0	2
P3	1	2	0

Tabell 1: Nuvarande resursanvändning.

	A	B	C
P1	4	3	2
P2	2	0	3
P3	2	3	1

Tabell 2: Maximal resursanvändning.

   (a) Process *P2* begär en extra resurs av typ *A*. Ska begäran tillåtas enligt Banker's algoritm? [2p]

   (b) Om ett system enligt Banker's algoritm *inte* är i ett säkert läge, vad innebär det? Välj bland alternativen nedan. Motivera ditt svar! [1p]

      1. Deadlock har uppstått.
      2. Deadlock *kommer* att uppstå i framtiden.
      3. Deadlock *kan* uppstå i framtiden.
      4. Någon process har exekeverat för snabbt så att ett synkroniseringsfel har uppstått.

Bakgrund till uppgift 3-5

Erik är, som ni kanske vet, mycket bra på att hitta på roliga idéer till tentauppgifter. Om man frågar honom varifrån han får alla bra idéer brukar han säga något i stil med: ”Jag sparar alla idéer jag kommer på i min bullshit-buffer, sedan är det bara att plocka därifrån!” Denna buffer är alltså en mycket viktig resurs för den grupp som Erik tillhör, eftersom den är ansvarig för innehållet i de flesta tentauppgifter (inklusive denna).

För att kunna göra intressanta tentauppgifter när Erik är upptagen med annat (exempelvis snurra på stolen, sjunga eller klippa ut tokens till brädspel) har vi implementerat vår egen idébuffer på filen `exam_ideas.c` (finns i `given_files`). Datatypen `idea_buffer` representerar en sådan idébuffer. Idébuffern har tre operationer:

- `idea_init`: Initierar idébuffern.
- `idea_add`: Lägger till en idé (en sträng) i buffern. Om buffern är full ska `idea_add` returnera `false`.
- `idea_get`: Väljer och returnerar en slumpmässig idé från buffern. Den valda idén tas sedan bort ur buffern så att den inte används till fler än en tentauppgift. Om buffern är tom så ska `idea_get` vänta på att en ny idé läggs till med `idea_add`.

Inför tentaperioder kommer funktionerna `idea_add` och `idea_get` att användas flitigt, och då är det viktigt att de kan användas från flera trådar samtidigt.

I slutet av filen `exam_ideas.c` finns också ett litet testprogram som visar hur idébuffern kan användas. Detta testprogram (dvs. funktionerna `main` och `worker`) är **inte** en del av uppgiften, och kommer därför inte rättas. Eventuell synkronisering som görs där kommer alltså inte räknas som en lösning på problemet.

Du kan kompilera koden med kommandot `make exam_ideas`. Se till att du har kopierat `Makefile` om det inte fungerar.

3. *Svara på del (a), (b) och (c) genom att modifiera en kopia av `exam_ideas.c`. Skicka sedan in den modifierade filen.*

   (a) Används *busy-wait* någon stans i koden? [1p]

   Kopiera de ställen i koden där busy-wait förekommer till kommentaren i början av filen. Om ingen busy-wait finns, skriv i så fall det i stället.

   (b) Varför bör *busy-wait* undvikas? [1p]

   Skriv ditt svar i kommentaren i början av `exam_ideas.c`.

   (c) Använd lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 7 för att eliminera den *busy-wait* som du hittat. [2p]

   Ändra i din kopia av `exam_ideas.c`. Se till att svaren från (a) och (b) finns kvar.

4. Efter att ha använt programmet under en tentamensperiod märker vi att samma idé har använts till flera uppgifter på olika tentor (det vill säga: flera anrop till `idea_get` har returnerat samma idé). Dessutom har Erik noterat att hans idéer ibland har försvunnit från buffern, trots att `idea_add` har returnerat `true`.

   Utgå gärna från din lösning av uppgift 3 när du löser den här uppgiften.

   Svara på del (a), (b) och (c) genom att modifiera en kopia av ***`exam_ideas.c`****. Skicka sedan in den modifierade filen.*

   (a) Förklara med exempel vad som kan ha hänt: [2p]

   1. Om samma idé har använts till flera uppgifter.
   2. Om buffern har ”tappat bort” en eller flera idéer.

   Skriv dina svar i kommentaren i början av ***`exam_ideas.c`****.*

   (b) Markera alla kritiska sektioner som finns i funktionerna `idea_add` och `idea_get` som i exemplet nedan. För varje kritisk sektion, skriv också vilken resurs som är kritisk. [2p]

```
// kritisk sektion börjar, resurs: x
if (x > 10)
  x = 10;
// kritisk sektion slutar, resurs: x
```

   Markera de kritiska sektionerna i din kopia av ***`exam_ideas.c`****, fortsätt sedan med (c).*

   (c) Använd lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 7 för att synkronisera koden utefter de kritiska sektioner du hittade. [4p]

   Notera: För mycket omotiverad låsning kan ge poängavdrag.

   Ändra i din kopia av ***`exam_ideas.c`****. Se till att svaren från (a) och (b) finns kvar.*

5. Använd de atomiska operationerna i kodlistning 2 på sida 8 i stället för de synkroniseringsprimitiver du använde i uppgift 4. Du kan utgå från din lösning från uppgift 3 om du vill, men du kan också börja om från början. [4p]

   Lämna in din modifierade version av ***`exam_ideas.c`*** *i sin helhet.*

6. Du har nyligen öppnat en liten affär med en egen liten nisch: alla kunder måste alltid köpa exakt två olika saker varje gång de handlar hos dig. Din affärsidé blir snabbt en stor succé, och du bestämmer dig för att anställa en medhjälpare. Utöver det så inser du att du behöver ett system för att hålla reda på hur mycket av varje produkt du har i lagret. Din implementation av lagersystemet finns på filen `store.c` (finns i `given_files`).

   Det viktiga i systemet är funktionen `buy`, som anropas när en kund vill köpa saker i din affär. Funktionen ser först till så att båda varorna finns i lager, och uppdaterar sedan lagerstatusen. Gick allt bra returneras `true`. Programmet ser därmed också till att du inte råkar bryta din affärsidé: Om en av varorna inte finns i lager så tillåts inte köpet, eftersom kunden då bara skulle få en vara!

   I och med att du nyligen har anställt ytterligare personal så måste du se till att `buy`-funktionen i ditt system är trådsäker. Du har börjat med att låsa delar av funktionen, vilket kan ses i `store.c`. Däremot har du stött på ett problem: ibland låser sig programmet helt. Du befarar att det finns ett deadlock någonstans!

   Du kan kompilera koden med kommandot `make store`. Se till att du har kopierat `Makefile` om det inte fungerar.

   Svara på del (a) och (b) genom att modifiera din kopia av `store.c`. Skicka sedan in `store.c`.

   (a) Visa med hjälp av ett exempel hur ett deadlock kan uppstå i `buy`-funktionen. Beskriv kort de fyra villkoren för deadlock, och visa att ditt exempel uppfyller alla villkoren. [4p]

   Skriv ditt svar i kommentaren i början av `store.c`.

   (b) Modifiera synkroniseringen så att deadlock inte längre kan uppstå. Motivera också vilket av villkoren för deadlock din lösning bryter. Koden måste såklart fortfarande vara korrekt synkroniserad. [2p]

   Ändra i din kopia av `store.c`. Se till att ditt svar från (a) finns kvar.

7. På filen `batch.c` (finns i `given_files`) finns ett program som summerar de 100 första heltalen i en fil. I och med att dessa filer ofta finns på långsamma diskar (exempelvis disketter) så har programmeraren valt att göra inläsningen och summeringen i en separat tråd så att programmet kan göra andra saker under tiden.

   Funktionen `start_sum` startar en tråd som summerar talen i filen. Det värde som returneras därifrån kan sedan skickas till `wait_sum` för att vänta på att summeringen blir klar och hämta resultatet. Se exempel på användning i `main`.

   Den givna koden kan kompileras med kommandot `make batch`. Se till att du har kopierat `Makefile` om det inte fungerar.

   Svara på del (a) och (b) genom att modifiera din kopia av `batch.c`. Skicka sedan in `batch.c`.

   (a) Efter att ha använt programmet ett tag märker du att programmet ger fel resultat ibland. I sällsynta fall har det varit så illa att programmet har kraschat. I och med att det händer så sällan så misstänker du synkroniseringsfel. Visa hur ett olägligt trådbyte kan ha orsakat något av dessa problem. [2p]

   Skriv ditt svar i kommentaren i början av `batch.c`.

   (b) Lös problemet genom att modifiera koden. [1p]

   Ändra i din kopia av `batch.c`. Se till att ditt svar från (a) finns kvar.

Tillgängliga synkroniseringsprimitiver

```
1  struct semaphore {
2      os_sema_t os;
3  };
4
5  void sema_init(struct semaphore *sema, unsigned value);
6  void sema_destroy(struct semaphore *sema);
7  void sema_down(struct semaphore *sema);
8  void sema_up(struct semaphore *sema);
9
10 struct lock {
11     os_lock_t os;
12 };
13
14 void lock_init(struct lock *lock);
15 void lock_destroy(struct lock *lock);
16 void lock_acquire(struct lock *lock);
17 void lock_release(struct lock *lock);
18
19 struct condition {
20     os_cond_t os;
21 };
22
23 void cond_init(struct condition *cond);
24 void cond_destroy(struct condition *cond);
25 void cond_wait(struct condition *cond, struct lock *lock);
26 void cond_signal(struct condition *cond, struct lock *lock);
27 void cond_broadcast(struct condition *cond, struct lock *lock);
```

Kodlistning 1: Synkroniseringsprimitiver

Finns även i `given_files/wrap/synch.h`.

Tillgängliga atomiska operationer

Atomiska operationer ekvivalenta med följande kod finns implementerade i filen `atomics.h` i `given_files/wrap/`.

```
const char *atomic_swap(const char **value, const char *replace) {
    void *old = *value;
    *value = replace;
    return old;
}

const char *compare_and_swap(const char **value, const char *compare,
                             const char *swap) {
    void *old = *value;
    if (old == compare)
        *value = swap;
    return old;
}

int atomic_add(int *value, int add) {
    int old = *value;
    *value += add;
    return old;
}

int atomic_sub(int *value, int add) {
    int old = *value;
    *value -= add;
    return old;
}
```

Kodlistning 2: Atomiska operationer