Linköpings universitet
Institutionen för datavetenskap 2019-05-29

Tentamen i
TDIU16 Process- och operativsystemprogrammering

Datum 2019-05-29

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.

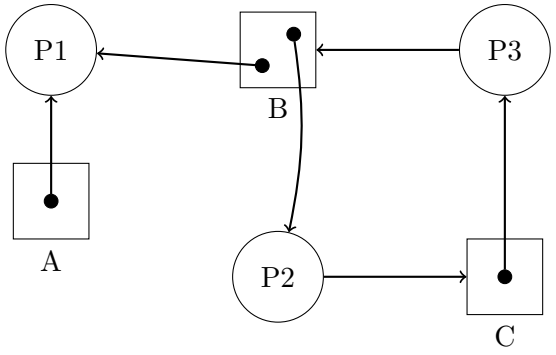
Instruktioner

- 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 6 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 (att meddelandet är mottaget). 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 ~/Desktop/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 tre processer, *P1*, *P2* och *P3*. Det finns också tre typer av resurser, *A*, *B* och *C*. Processerna har för närvarande allokerat resurser enligt fig. 1.



Figur 1: Systemets nuvarande resursanvändning.

   (a) Namnge och beskriv kort de fyra villkoren för deadlock. [2p]

   (b) Är systemet i deadlock? Motivera ditt svar med de fyra villkoren för deadlock. [2p]

   Lämna in svar på (a) och (b) som ***uppgift1.txt*** *eller* ***uppgift1.odt****.*

2. I ett system körs tre processer, *P1, P2* och *P3*. I systemet finns det också tre typer av resurser, *A, B* och *C*, och 5 instanser av vardera. Tabell 1 visar hur systemets resurser är allokerade i nuläget och det maximala resursbehovet för varje process.

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

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

Maximal resursanvändning

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

Nuvarande resursanvändning

Tabell 1: Resurser i systemet.

   (a) Är systemet i ett säkert läge enligt Banker's algorithm? Redovisa dina beräkningar. [2p]

   (b) Process *P3* begär en extra resurs av typ *A*. Ska begäran tillåtas enligt Banker's algoritm? Redovisa dina beräkningar. [2p]

   (c) Vad ska göras då en begäran inte tillåts enligt Banker's algorithm? Motivera ditt svar. [1p]

      1. Processen som begärde en extra resurs måste avslutas – det kommer **garanterat** bli deadlock om den fortsätter köra.
      2. Processen som begärde en extra resurs får vänta – det kommer **garanterat** bli deadlock om den fortsätter köra.
      3. Processen som begärde en extra resurs får vänta – det **kan** bli deadlock om den fortsätter köra.
      4. Processen som begärde en extra resurs får fortsätta – det är först vid nästa resursförfrågan som problem kan uppstå, och vi kan lösa problemen då.

3. I smurfarnas lilla by finns en populär bar där invånarna brukar spendera fredagkvällarna med att avnjuta de underbara mjölkdrinkar som erbjuds där. Favoriten i byn är en drink bestående av mjölk och blåbär. Dessvärre fungerar inte det tidigare systemet med att invånarna turas om att arbeta i baren eftersom alla är trötta efter en hård vecka med Pintoslaborationer. Efter mycket funderande har invånarna kommit fram till att bygga två robotar som kan arbeta som bartenders och servera drinkar, så att alla kan avnjuta fredagkvällen utan arbete.

   Robotarna styrs av en dator som kör programmet som finns givet i filen `bar_bot.c`. För att kunna hantera båda robotarna samtidigt (smurfarna gillar inte att vänta för länge), har en tråd för varje robot skapats så att de kan arbeta parallellt. Programmet håller reda på vad för ingredienser som finns tillgängliga i baren (datatypen `ingredient` och variabeln `supply`), samt recept för de drinkar som erbjuds (datatypen `recipe` och variabeln `recipes`). Som systemet fungerar i nuläget initieras de tillgängliga ingredienserna och recepten en gång innan robotarna startas (alltså, bara från en tråd). Funktionen `make_drink` anropas dock av trådarna som styr robotarna för att se om det finns tillräckligt med ingredienser för att göra en viss drink, och för att uppdatera mängden ingredienser som finns tillgängliga.

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

   Skriv dina svar i filen `bar_bot.c` och skicka in den modifierade filen.

   (a) Efter att ha kört systemet några kvällar inser invånarna ett problem. Ibland, ofta nära stängning då det är ont om ingredienser, gör robotarna drinkar trots att det inte finns tillräckligt mycket ingredienser kvar. Detta trots att programmet kontrollerar om det finns tillräckligt mycket kvar innan varje drink tillreds! Robotarna borde i stället informera om problemet och inte göra en halvfärdig drink (`make_drink` borde alltså returnera `false`). Programmerarna är mycket konfunderade och ber dig se om du kan förstå vad som har hänt! [1p]

   Förklara med ett exempel vad som kan ha gått fel då en robot serverar en drink utan att det finns tillräckligt av någon ingrediens. D.v.s. `make_drink` returnerar `true` trots att det inte finns tillräckligt av båda ingredienserna som krävs.

   Skriv ditt svar i kommentaren i början av filen `bar_bot.c`.

   (b) Vilka kritiska sektioner finns i `make_drink`-funktionen? För varje kritisk sektion, notera också vilken eller vilka resurser som är kritiska. [2p]

   *Markera de kritiska sektionerna med kommentarer i filen `bar_bot.c`. Exempelvis som nedan. Använd **inte** radnummer, de blir felaktiga när du svarar på frågorna.*

```
1 // Kritisk sektion 1 börjar. Resurs: foo
2 // Kritisk sektion 1 slutar.
```

   (c) Lös problemet du hittade i (b) med hjälp av lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 7. [2p]

   Notera: Försök maximera den teoretiska parallellismen i din lösning. Notera gärna ifall du tror att din lösning är ineffektiv i praktiken trots att den har hög teoretisk parallellism.

   Modifiera koden i `bar_bot.c`.

   (d) Finns det risk att deadlock uppstår i din lösning i (c)? Motivera ditt svar med hjälp av de fyra villkoren för deadlock. [2p]

   Skriv ditt svar i kommentaren i början av filen `bar_bot.c`.

4. Bertram tycker mycket om att programmera, och letar ständigt efter sätt att skriva fin och lättläslig kod. Eftersom den första implementationen av systemanropet `wait` i Pintos inte blev tillräckligt bra började Bertram leta efter andra bra abstraktioner som kunde förenkla implementationen. Efter mycket funderande kom Bertram fram till att en *future* antagligen kan användas för ändamålet och började implementera en sådan.

   Bertrams implementation finns i filen `future.c`, och är något förenklad jämfört med andra implementationer. Det finns en datatyp, `future`, som representerar ett värde (ett heltal) som kommer att skapas någon gång i framtiden. Detta är exempelvis användbart när man vill beräkna något i en separat tråd, men inte vill hantera synkroniseringen explicit. Då kan man i stället skapa en future och låta den separata tråden spara sitt resultat i den när den är klar. Originaltråden kan sedan hämta värdet från future-objektet när det behövs. Det är precis det här som den givna koden i botten av `future.c` gör för att illustrera hur implementationen ska fungera. Detta är dock inte en del av uppgiften, vilket kommentaren i filen också påpekar.

   Future-objektet som är implementerat i `future.c` har följande operationer:

   `future_init` Initierar ett future-objekt. Vi antar att varje objekt bara initieras en gång, och att det inte används någon annanstans förrän det har initierats ordentligt.

   `future_set` Sparar ett värde (ett heltal) i future-objektet som sedan kan hämtas med hjälp av `future_get`. Vi antar att `future_set` anropas exakt en gång för varje future-objekt.

   `future_get` Försöker hämta värdet som finns i future-objektet. Om det inte finns något värde ännu ska `future_get` vänta på att en annan tråd sparar ett värde i future-objektet med `future_set`. I slutändan ska värdet som future-objektet innehåller returneras. Denna operation ska gå att anropa flera gånger på samma objekt till skillnad från `future_set`.

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

   Skriv dina svar i filen `future.c` och skicka slutligen in den modifierade filen.

   (a) Vad är *busy-wait*, och varför bör det undvikas? [1p]

   Skriv ditt svar i kommentaren i början av filen `future.c`.

   (b) Förekommer *busy-wait* någonstans i koden? Markera i så fall alla ställen där det förekommer med en kommentar. Ange även vad som väntas på. [1p]

   *Markera de ställen i koden där busy-wait förekommer med kommentarer i filen `future.c`. Exempelvis som nedan. Använd **inte** radnummer, de blir felaktiga när du svarara på frågorna.*

```
// Början av busy wait. Vi väntar på att X ska bli 0.
...
// Slut av busy wait
```

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

   Modifiera koden i `future.c`.

   (d) Finns det några fler saker i implementationen som behöver synkroniseras? Beskriv i så fall hur de kan åtgärdas. Motivera annars varför implementationen är trådsäker givet din lösning i (c). [1p]

   Skriv ditt svar i kommentaren i början av filen `future.c`.

5. Du håller på att implementera ett program som ritar grafer som du hoppas att du kan få ha som hjälpmedel på en kommande mattetenta. I och med att tentatiden är begränsad är varje sekund av tentatiden dyrbar, så du har verkligen inte tid att vänta på ditt program i onödan! Efter att ha funderat lite inser du att det som tar mest tid är att beräkna funktionen du vill rita ut för alla värden på x-axeln, och du har därmed beslutat dig för att parallellisera beräkningen på dina 4 CPU-kärnor.

   Din lösning så här långt finns i filen `parallelize.c`. Du har refaktorerat ditt program så att alla tunga beräkningar sker i funktionen `run_in_serial`. Den funktionen får en funktion och en array av x-värden som parametrar, beräknar $f(x)$ för alla värden som finns i `input`-arrayen och lägger resultaten `output`-arrayen. För att göra alla beräkningar parallellt har du också implementerat `run_in_parallel`, som ska göra samma sak med skillnaden att den fördelar arbetet på flera trådar.

   Notera: Funktionen `run_in_parallel` ska kunna anropas från flera trådar samtidigt.

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

   Skriv dina svar i filen `parallelize.c` och skicka slutligen in den modifierade filen.

   (a) Tyvärr märker du att `run_in_parallel` inte fungerar som den ska. Ibland verkar det inte som att alla värden beräknas, och ibland kraschar till och med programmet helt och hållet. [1p]

   Beskriv med ett exempel vad som kan ha gått fel då programmet kraschar eller då alla värden inte har hunnit beräknas när `run_in_parallel` returnerar.

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

   (b) Använd lämpliga synkroniseringsprimitiver från kodlistning 1 på sidan 7 för att lösa problemet du hittade i (a). [2p]

   Modifiera koden i `parallelize.c`.

   (c) Efter ett tag märker du också att implementationen ibland inte beräknar alla x-värden som finns med i `input`-arrayen. Förklara med ett exempel hur ett olägligt trådbyte kan få detta att inträffa! [1p]

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

   (d) Använd lämpliga synkroniseringsprimitiver från kodlistning 1 på sidan 7 för att lösa problemet du hittade i (c). [2p]

   Modifiera koden i `parallelize.c`.

6. Lös de problem som beskrivs i fråga 5, d.v.s. både (a) och (c), med hjälp av atomiska operationer i stället för de ”vanliga” synkroniseringsprimitiverna. Använd de atomiska operationer som finns i kodlistning 2 på sidan 8. [3p]

   Notera: Det är okej om din lösning innehåller *busy-wait* – det går inte att eliminera *busy-wait* med bara atomiska operationer.

   Gör en ny kopia av den givna filen `parallelize.c`, och spara den som exempelvis `atomics.c`. Skriv sedan dina svar i den filen och skicka in den.

Tillgängliga synkroniseringsprimitiver

```
1  struct semaphore {
2      // Privat data. Använd funktionerna nedan för att manipulera semaforen.
3      os_sema_t os;
4  };
5
6  void sema_init(struct semaphore *sema, unsigned value);
7  void sema_destroy(struct semaphore *sema);
8  void sema_down(struct semaphore *sema);
9  void sema_up(struct semaphore *sema);
10
11 struct lock {
12     // Privat data. Använd funktionerna nedan för att manipulera låset.
13     os_lock_t os;
14 };
15
16 void lock_init(struct lock *lock);
17 void lock_destroy(struct lock *lock);
18 void lock_acquire(struct lock *lock);
19 void lock_release(struct lock *lock);
20
21 struct condition {
22     // Privat data. Använd funktionerna nedan för att manipulera
23     // condition-variabeln.
24     os_cond_t os;
25 };
26
27 void cond_init(struct condition *cond);
28 void cond_destroy(struct condition *cond);
29 void cond_wait(struct condition *cond, struct lock *lock);
30 void cond_signal(struct condition *cond, struct lock *lock);
31 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/`.

```
1  int test_and_set(int *value) {
2      int old = *value;
3      *value = 1;
4      return old;
5  }
6
7  int atomic_swap(int *value, int replace) {
8      int old = *value;
9      *value = replace;
10     return old;
11 }
12
13 int compare_and_swap(int *value, int compare, int swap) {
14     int old = *value;
15     if (old == compare)
16         *value = swap;
17     return old;
18 }
19
20 int atomic_add(int *value, int add) {
21     int old = *value;
22     *value += add;
23     return old;
24 }
25
26 int atomic_sub(int *value, int add) {
27     int old = *value;
28     *value -= add;
29     return old;
30 }
```

Kodlistning 2: Atomiska operationer