

---

# Tentamen i **TDIU16** Process- och operativsystemprogrammering

---

**Datum** 2025-06-02

**Examinator**

**Tid** 8–12

Filip Strömbäck (filip.stromback@liu.se)

**Institution** IDA

**Administratör**

Annelie Almquist

**Kurskod** TDIU16

**Jourhavande lärare**

**Provkod** DAT1

Filip Strömbäck (013-28 27 52)

## Tillåtna hjälpmedel

Inga hjälpmedel.

## Instruktioner

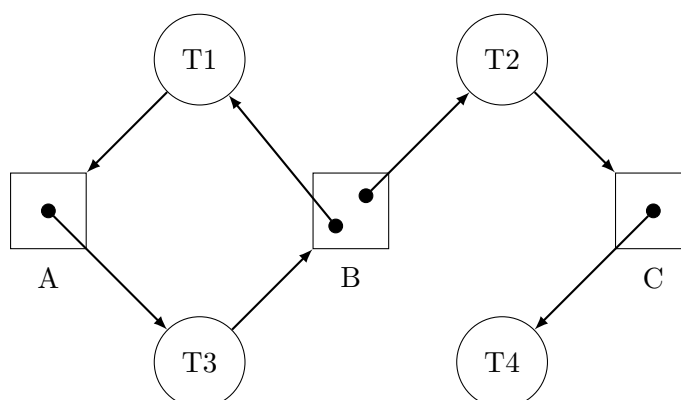
- Ange din tolkning av frågan och alla antaganden du gör.
- Var precis i dina påståenden. Visa ditt resonemang när möjligt. Svävande eller tvetydiga formuleringar leder till poängavdrag.
- Motivera tydligt och djupgående alla påståenden och resonemang. Förklara uträkningar och lösningsmetoder.
- Sträva 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 4 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

- Uppgifter lämnas in via tentaklienten. Gör en inskickning per numrerad uppgift (kom ihåg att välja rätt uppgiftsnummer när du skickar in). Systemet svarar sedan med ett meddelande om att begäran är mottagen om allt gick bra.
- Om du tror dig ha lämnat in felaktiga filer på någon uppgift kan du helt enkelt skicka in uppgiften igen. Vi kommer bara bedöma den sista inlämningen av varje uppgift.
- Tentan bedöms i efterhand. Ni kommer alltså inte få svar på inlämnade uppgifter under tentans gång.
- För varje uppgift anges i vilket format den uppgiften lämnas in. Koduppgifter lämnas in som källkod (.c). Övriga uppgifter lämnas in som en textfil (.txt), i LibreOffice-format (.odt), eller som PDF (.pdf). Filnamn är i allmänhet inte viktiga, men exempel anges i uppgifterna.
- Inlämnad kod behöver **inte** kompilera för att ge poäng. Det viktiga är att vi entydigt kan utläsa vad ni menar. Det är alltså okej att använda pseudokod för detaljer ni inte kommer ihåg, eller om det är något ni inte får att kompilera. Fördelen med kompilerande kod är att det är väldefinierat vad den betyder, och att det går att göra en "sanity check" innan man lämnar in.
- Kopiera de givna filerna till er hemmapp innan ni försöker redigera dem. Detta kan exempelvis göras med kommandot `cp -r ~/Desktop/given_files/* ~/` eller via den grafiska miljön.
- I slutet av de givna filerna finns ett testprogram som visar hur koden kan användas. Testprogrammet är **inte** en del av uppgiften, och kommer därmed inte att bedömmas. Det är helt okej att modifiera testprogrammet för att testa resten av programmet. Resten av koden ska dock fungera korrekt utan extra synkronisering i testprogrammet.
- 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 alla potentiella synkroniseringsfel i koden.
- 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 du har kopierat alla givna filer ifall det inte fungerar.

1. Nedan finns en resursallokeringsgraf för ett system med fyra trådar och tre resurser.

Lämna in svar på frågorna nedan som *uppgift1.txt*, *uppgift1.odt*, eller *uppgift1.pdf*.



- (a) Vilka är de fyra villkoren för deadlock? Namnge villkoren och ge även en kort beskrivning av varje villkor (ca 1 mening per villkor). [2p]
- (b) Är några trådar i systemet ovan i deadlock? I så fall vilka? Motivera ditt svar med hjälp av de fyra villkoren för deadlock från del (a). [2p]
2. En bekant till dig har hakat på trenden att revolutionera IT-system genom att tillhandahålla dem som tjänster. Det senaste projektet är att skapa en ny marknad inom BaaS (Banking as a Service). Din vän har därför börjat implementera datastrukturer i C för att hantera konton i banker. För maximal prestanda måste dessa datastrukturer fungera korrekt när de används av flera trådar samtidigt, och samtidigt tillåta så mycket parallellism som möjligt.
- Filen `bank.c` innehåller implementationen. Den består av `struct bank` som representerar en bank. En bank innehåller upp till `MAX_ACCOUNTS` konton som vart och ett representeras av `struct account`. Varje konto innehåller i sin tur ett namn och ett saldo. Varje konto identifieras av sitt ID, som motsvarar kontots plats i arrayen i `struct bank` (dvs. kontot med ID 1 är på index 1 i arrayen). Datastrukturen `struct bank` har följande operationer:
- bank\_create** Skapar och returnerar en ny `struct bank`.
- bank\_free** Frigör en `struct bank` som skapats med `bank_create`. Funktionen antar att ingen annan tråd använder den instans som frigörs.
- bank\_new\_account** Skapar ett nytt konto i en bank. Funktionen måste därmed hitta ett konto-ID som inte är använt, skapa kontot på den platsen i banken, och returnera ID:t på det skapade kontot. Om banken är full returneras i stället -1.
- bank\_close\_account** Stänger (= tar bort) ett konto i banken. Eventuellt kvarvarande saldo på kontot överförs till ett annat konto innan stängningen sker. Returnerar `true` om allt gick bra, eller `false` om något gick fel.
- bank\_print** Skriver ut en lista över alla konton i banken (id, namn, och saldo) samt totala mängden pengar i banken.

(fortsättning på nästa sida)

Tyvärr fungerar inte implementationen som den ska. Din bekanta har hittat tre fel med hjälp av det testprogram som finns i botten av filen. Programmet går att köra i 3 lägen för att se felen enklare (fel 1 syns enklast i läge 1 och så vidare). Felen är följande:

1. Om flera trådar skapar nya konton med `bank_new_account` samtidigt så "försvinner" ibland konton. Detta syns exempelvis genom att totala mängden pengar på banken är mindre än 20 000 kr.
2. Om flera trådar stänger konton med `bank_close_account` som i testprogrammet så borde resultatet vara att alla pengar finns på antingen ett eller två konton. Ibland försvinner dock alla konton och därmed alla pengar. Notera att `bank_close_account` ska kunna anropas samtidigt som `bank_new_account`, även om det inte testas uttryckligen.
3. Om en tråd skriver ut en lista över alla konton med `bank_print` samtidigt som en annan tråd stänger konton med `bank_close_account` så blir saldot ibland för högt (dvs. över 20 000 kr).

Du kan kompilera programmet med `make bank`. Kör sedan programmet med `./bank`. Läget för testprogrammet kan justeras med variabeln `test_mode`. Det är helt okej att modifiera testprogrammet. Testprogrammet är dock *inte* en del av uppgiften och eventuell synkronisering där kommer att ignoreras vid bedömning.

*Skriv dina svar i en kopia av `bank.c` och skicka in den modifierade filen.*

- (a) Beskriv med exempel (ett för varje fel) vad som kan ha hänt för att de tre felen ovan (numrerade 1–3) ska uppstå. Varje exempel ska bestå av ett konkret scenario med trådbyten mellan två eller flera trådar som visar hur felet uppstår. [3p]

*Skriv dina svar i kommentaren i början av filen.*

- (b) Använd lämpliga synkroniseringsprimitiver från kodlistning 1 på sidan 7 för att lösa problemen i (a). [6p]

**Notera:** Se resterande deluppgifter för ytterligare egenskaper din lösning ska ha för att få full poäng. En lösning som fungerar korrekt och inte har *data races* ger åtminstone delpoäng.

*Modifiera koden i filen.*

- (c) Finns det risk att din lösning i (b) leder till deadlock? Motivera ditt svar med hjälp av de fyra villkoren för deadlock (exempelvis genom att nämna att ett av villkoren inte uppfylls, tillsammans med motivering). [1p]

**Notera:** För full poäng i (b) bör svaret här vara "nej". Notera att testprogrammet bara är ett exempel på hur datastrukturen kan användas, deadlock ska aldrig uppstå om datastrukturen används enligt specifikation.

*Skriv ditt svar i kommentaren i början av filen.*

- (d) I en bank `b` så finns 5 konton (0–4) med 100 kronor vardera. En tråd kör `bank_close_account(b, 1, 3)` samtidigt som en annan tråd kör `bank_close_account(b, 2, 4)`. Garanterar din lösning i (b) att trådarna aldrig behöver vänta på varandra? [1p]

**Notera:** För full poäng i (b) bör svaret här vara "ja".

*Skriv ditt svar i kommentaren i början av filen.*

3. Den här uppgiften använder koden i `bank_atomics.c`. Koden är nästan identisk med koden i `bank.c` som användes i uppgift 2. Skillnaderna finns i funktionen `bank_print` och är markerade i källkoden. Skillnaderna är dock bara relevanta för del (b).

*Skriv dina svar i en kopia av `bank_atomics.c` och skicka in den modifierade filen.*

- (a) Lös de problem du hittade i funktionen `bank_new_account` i uppgift 2 (dvs. fel nummer 1) enbart med hjälp av de atomiska operationer som finns i kodlistning 2 på sidan 7. Du behöver alltså bara ta hänsyn till `bank_new_account`, och kan ignorera `bank_close_account` och `bank_print` för den här deluppgiften.

[3p]

**Notera:** Din lösning ska *inte* introducera extra variabler i varken `struct bank` eller `struct account`. Nya lokala variabler inuti `bank_new_account` är OK.

*Modifiera koden i `bank_atomics.c`.*

- (b) Funktionen `bank_print` i filen `bank_atomics.c` använder `atomic_read` för att läsa `name` och `balance` från `struct account`. Kommer detta göra att `bank_print` fungerar korrekt tillsammans med din implementation av `bank_new_account`?

[1p]

Exempelvis, antag att en tråd anropar `bank_new_account(b, "New", 100)` samtidigt som en annan tråd anropar `bank_print(b)`. Är det då garanterat att utskriften från `bank_print` antingen inte innehåller kontot "New" alls, eller att kontot "New" finns med och innehåller korrekt information? Om inte, motivera ditt svar med ett exempel.

**Notera:** Du behöver *inte* lösa detta problem för full poäng i (a), jag tror till och med inte att det går att lösa enligt kraven i (a).

*Skriv ditt svar i kommentaren i början av filen.*

4. Filen `smudge.c`<sup>1</sup> innehåller ett program som gör en svartvit bild oskarp genom att "smeta ut" innehållet i bilden horisontellt. Programmet innehåller datatypen `struct picture` som representerar en svartvit bild. Det centrala i programmet är funktionen `picture_smudge` som tar emot tre parametrar: en bild, hur mycket bilden ska smetas ut, och hur många trådar som ska användas. Funktionen startar sedan det specificerade antalet trådar. Alla dessa kör funktionen `smudge_worker`. Funktionen `smudge_worker` anropar i sin tur funktionen `smudge_pixel` för att beräkna utsmetade versioner av alla pixlar i den nya bilden.

I `smudge.c` finns också en `main`-funktion som gör det möjligt att testa `picture_smudge` från kommandoraden. Programmet läser först in en bild från disk och visar den i terminalen. Sedan skapar den en utsmetad version av bilden med `picture_smudge`. Den utsmetade bilden visas sedan i terminalen och sparas på disk. `main`-funktionen är inte en del av uppgiften, och eventuella ändringar där ignoreras vid bedömning av uppgiften. Det är dock OK att ändra `main` för att exempelvis testa med fler eller färre trådar.

Du kan kompilera programmet med `make smudge`, och sedan köra det med `./smudge`. Som standard läser programmet bilden `pintos.ppm`, visar den i original och utsmetad version, och sparar den som `output.ppm`. Du kan ange en annan bild som kommandoradsargument, exempelvis `./smudge flower.ppm`. Vill du titta på den sparade bilden kan du köra `xdg-open output.ppm`, eller öppna filen i `emacs`.

*(fortsättning på nästa sida)*

---

<sup>1</sup>Tillsammans med hjälpfunktioner i `picture_utils.h`, men de funktionerna är inte en del av uppgiften.



Figur 1: Bilder från körning av `./smudge flower.ppm`. Originalbilden visas till vänster. Den utsmetade bilden visas till höger.

Figur 1 visar bilderna från en körning av `./smudge flower.ppm`. Som du ser så fungerar inte `picture_smudge` som den ska. Två fel syns i bilden till höger: dels så finns det rader som programmet verkar ha hoppat över nästan helt (de svarta raderna). Utöver det så finns också situationer där programmet har hoppat över enskilda pixlar (exempelvis ganska långt ner i mitten i bilden till höger).

Ett problem som inte syns ovan är att `picture_smudge` ibland returnerar den nya bilden innan trådarna är helt klara. Detta visar sig som svarta pixlar i nedre högra hörnet. Det är mest tydligt när många trådar används. Dock är det svårt att se eftersom många trådar gör att programmet kraschar i de flesta fall.

*Skriv dina svar i en kopia av `smudge.c` och skicka in den modifierade filen.*

- (a) Beskriv med exempel (ett för varje situation) vad som kan ha hänt om... [3p]

1. ...programmet hoppade över en enskild pixel?
2. ...programmet hoppade över majoriteten av en hel rad?
3. ...`picture_smudge` returnerar innan alla trådar är klara?

*Skriv ditt svar i kommentaren i början av filen.*

- (b) Identifiera det eller de ställen i `smudge.c` där `busy-wait` kan ske. För varje ställe du hittar, skriv också vad koden väntar på. [1p]

*Skriv ditt svar i kommentaren i början av filen. Kopiera de delar som använder `busy-wait` till kommentaren, eller markera dem i källkoden med en kommentar. Använd **inte** radnummer. De blir fel när du svarar på resten av frågorna.*

- (c) Använd lämpliga synkroniseringsprimitiver (från kodlistning 1 på sidan 7) för att åtgärda de `busy-wait` du hittade i (b), samt problem 3 i (a). [3p]

*Modifiera koden i filen.*

- (d) Använd lämpliga synkroniseringsprimitiver (från kodlistning 1 på sidan 7) för att åtgärda problem 1 och 2 i (a). [3p]

*Modifiera koden i filen.*

- (e) Tillåter din lösning att flera trådar kör beräkningarna i `smudge_pixel` samtidigt? [1p]

**Notera:** för full poäng på (c) och (d) bör ditt svar här vara ”ja”.

*Skriv ditt svar i kommentaren i början av filen.*

## Tillgängliga synkroniseringsprimitiver

Dessa finns även i filen `given_files/wrap/synch.h`

---

```
1 struct semaphore {
2     // ...
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     // ...
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     // ...
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

## Tillgängliga atomiska operationer

Atomiska operationer ekvivalenta med följande kod finns implementerade i filen `atomics.h` i `given_files/wrap/`. Dessa funktioner fungerar på godtyckliga heltalsdatatyper och pekare.

---

```
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 }
31
32 int atomic_read(int *value) {
33     return *value;
34 }
35
36 void atomic_write(int *value, int write) {
37     *value = write;
38 }
```

---

Kodlistning 2: Atomiska operationer