
Tentamen i TDIU16 Process- och operativsystemprogrammering

Datum 2024-05-29

Examinator

Tid 08–12

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

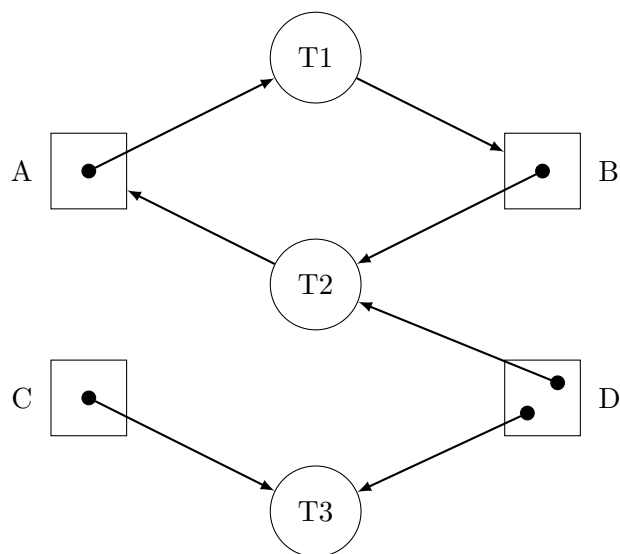
- Ange din tolkning av frågan och alla antaganden du gör.
- Var precis i dina påståenden. Visa ditt resonemang när möjligt. Svävande eller tvetydiga formuleringar leder till poängavdrag.
- Motivera tydligt och djupgående alla påståenden och resonemang. Förklara uträkningar och lösningsmetoder.
- Sträva 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 9 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 tre trådar och fyra 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. Du håller på att utveckla ett spel. En viktig del av spelet är att hålla koll på alla animationer som pågår. För att hålla reda på alla animationerna har du byggt ett en datastruktur **animations**. Datastrukturen innehåller en samling *händelser* (**event**). Varje händelse innehåller en funktion som ska anropas för varje bildruta (eng. *frame*) i spelet. För att maximera prestandan i ditt spel vill du självklart köra olika händelser på flera trådar parallellt.

Datastrukturen finns i filen **animations.c**. Datastrukturen har följande operationer:

animations_create Skapar en ny **animations**-instans. Som en del av detta startas även ett förbestämt antal trådar som sedan används för att anropa funktionerna i de händelser som läggs till.

animations_destroy Förstör en **animations**-instans. Som en del av detta stoppas trådarna som skapades av **animations_create**. Funktionen antar att ingen annan tråd använder **animations**-instansen som ska förstöras.

animations_add Lägger till en händelse till ett **animations**-objekt. Detta ska kunna ske ifrån godtyckliga trådar, även under tiden som **animations_run_step** körs.

animations_run_step Kör alla händelser som har lagts till på de skapade trådarna och väntar tills alla händelser har körts klart. Om nya händelser läggs till under tiden ska de också köras.

I **animations.c** finns också ett enkelt testprogram som illustrerar hur datastrukturen är tänkt att fungera. Testprogrammet skapar händelser som skriver ut enskilda bokstäver till terminalen. Det anropar sedan **animations_run_step** 10 gånger och skriver ut en rad med **-**-tecken mellan varje anrop. Utdata ska alltså innehålla rader av bokstäver avskiljda av rader med **-**-tecken. Det är **inte** viktigt att bokstäverna är i samma ordning i varje rad. Det är dock viktigt att alla bokstäver ska finnas med i varje rad. Testprogrammet lägger också till en ny utskrifts-händelse i varje iteration, så antal bokstäver ska öka med 1 varje gång. Ett exempel på hur de första raderna kan se ut finns nedan:

```
-----
acbdefhgijA
-----
abcedfghijAB
-----
abcdegfhiAjBC
-----
abcedfghjiABDC
-----
...
```

Efter att ha testkört din implementation inser du snabbt att datastrukturen inte fungerar som den ska. I och med att den beter sig olika varje gång misstänker du trådproblem. Mer specifikt har du märkt följande:

- Ibland körs inte alla händelser (alla tecken skrivs inte ut).
- Ibland körs samma händelse mer än en gång (dubletter av tecken).
- Om två händelser läggs till samtidigt så tappas ibland en av dem bort.
- Ibland kraschar programmet precis efter anropet till **animations_destroy**.
- Implementationen verkar använda mer CPU-tid än nödvändigt.

(fortsättning på nästa sida)

Du kan kompilera programmet med kommandot `make animations` i terminalen. Kör sedan programmet genom att skriva `./animations`. Notera att din kod **inte** behöver kompilera när du skickar in den.

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

- (a) Beskriv med exempel (ett för varje punkt) vad som kan ha gått fel när: [4p]

- 1: ...en händelse körs mer än en gång (ett tecken skrivs ut mer än en gång)
- 2: ...en händelse tappas bort då `animations_add` anropas av flera trådar samtidigt
- 3: ...en händelse körs efter att `animations_run_step` returnerat (ett tecken kommer ut för sent)
- 4: ...programmet kraschar precis efter anropet till `animations_destroy`

Skriv ditt svar i kommentaren i början av filen.

- (b) Identifiera de instanser av *busy-wait* som finns i koden. [1p]

*Skriv ditt svar i kommentaren i början av filen. Kopera antingen de delar som innehåller `busy-wait` till kommentaren, eller markera dem i källkoden. Använd **inte** radnummer. De blir felaktiga när du svarar på resten av frågorna.*

- (c) Åtgärda de problem med *busy-wait* du hittade i (b), samt problem 3 och 4 ifrån (a). [4p]
Använd lämpliga synkroniseringsprimitiver från kodlistning 1 på sidan 8.

Modifiera koden i filen.

- (d) Lös problem 1 och 2 du hittade i (a) med hjälp av lämpliga synkroniseringsprimitiver från kodlistning 1 på sidan 8. Datastrukturen ska efter detta fungera enligt specifikationen på förra sidan. Specifikationen finns också i källkoden, innan respektive funktion. [4p]

Modifiera koden i filen.

3. En typ av problem som förekommer ofta inom datavetenskap är att söka efter ett element som uppfyller vissa egenskaper. Problem av den här typen kallas ibland för *sökproblem*. En trevlig egenskap hos sökproblem är att de ofta är enkla att parallellisera eftersom det oftast går att testa olika lösningar parallellt.

Ett exempel på ett sökproblem är att kontrollera om ett tal, n , är ett primtal eller inte. Talet n är ett primtal om det inte finns något annat heltal, d så att $\frac{n}{d}$ är ett heltal (vi säger då att d är en *delare* till n). Vi kan uttrycka detta som ett sökproblem som följer: Finns det ett heltal d i intervallet $2 \leq d < n$ så att n är delbart med d ?

Testprogrammet i filen `multiresult.c` implementerar en lösning till problemet. Det startar 8 trådar som tillsammans itererar igenom alla tal från 2 till n , och testar om något av dem är en delare till n . För att koordinera arbetet mellan trådarna använder testprogrammet en datastruktur `multiresult`. Datastrukturen lagrar den första delaren som hittas av trådarna. I och med att vi bara är intresserade av att hitta *en* delare kan resterande trådar avbrytas så snart en delare har hittats. Detta sköts också av datastrukturen `multiresult`.

Datastrukturen innehåller följande operationer:

multi_init Initierar en instans av datastrukturen.

multi_post_nothing Anropas av en av trådarna som söker efter delare för att meddela att den är klar med sin sökning, men att den inte hittade en delare.

multi_post_result Anropas av en av trådarna som söker efter delare för att meddela att den hittade ett resultat och att den därmed är klar med sin sökning. Det första resultatet som hittas ska lagras i datastrukturen. Eventuella ytterligare resultat ignoreras.

multi_has_result Kontrollerar om datastrukturen innehåller ett resultat.

multi_wait Vänta på att det specificerade antalet trådar har blivit klara och har anropat antingen `multi_post_nothing` eller `multi_post_result`. Funktionen ska också meddela om ett resultat har hittats eller inte (detta beskrivs i mer detalj i kommentaren i källkoden). När `multi_wait` är klar är det inte giltigt att anropa `multi_post_nothing` eller `multi_post_result` längre, så vi antar att det inte sker. Det går alltså inte att använda samma instans mer än en gång.

Testprogrammet är korrekt. Dock fungerar det inte som det ska på grund av problem i implementationen av datastrukturen `multiresult`:

- Ibland väntar `multi_wait`, trots att alla trådarna har blivit klara.
- Ibland lagras inte resultatet av det första anropet till `multi_post_result`, utan det blir överskrivet av ett anrop ifrån en annan tråd.
- Implementationen verkar använda mer CPU-tid än nödvändigt.

Du kan kompilera programmet med kommandot `make multiresult` i terminalen. Kör sedan programmet genom att skriva `./multiresult`. Notera att din kod **inte** behöver kompilera när du skickar in den.

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

(fortsättning på nästa sida)

- (a) Beskriv med exempel (ett för varje punkt) vad som kan ha gått fel när: [2p]

1: ...`multi_wait` väntar trots att alla trådarna har blivit klara.

2: ...ett resultat lagrat av `multi_post_result` blir överskrivet av ett senare resultat.

Skriv ditt svar i kommentaren i början av filen.

- (b) Identifiera de instanser av *busy-wait* som finns i koden. [1p]

*Skriv ditt svar i kommentaren i början av filen. Kopera antingen de delar som innehåller busy-wait till kommentaren, eller markera dem i källkoden. Använd **inte** radnummer. De blir felaktiga när du svarar på resten av frågorna.*

- (c) Åtgärda de problem med *busy-wait* du hittade i (b). Använd lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 8. [2p]

Modifiera koden i filen.

- (d) Lös problemen du hittade i (a) med lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 8. [2p]

Modifiera koden i filen.

- (e) Finns det några ytterligare variabler som behöver skyddas i din implementation? Motivera ditt svar, och skydda eventuella oskyddade variabler med lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 8. [1p]

Skriv din motivation i kommentaren i början av filen, samt modifiera koden vid behov.

4. Den här uppgiften utgår ifrån samma problem som används i fråga 3. Gör därför en ny kopia av den givna filen `multiresult.c` som du använder för att lösa den här uppgiften. Döp denna kopia till `multiresult_atomics.c` och skicka in den filen när du är klar.

Likt i fråga 3 kan du kompilera programmet med kommandot `make multiresult_atomics` i terminalen. Kör sedan programmet genom att skriva `./multiresult_atomics`. Notera att din kod **inte** behöver kompilera när du skickar in den.

*Skriv dina svar i **multiresults__atomics.c** och skicka in den modifierade filen.*

- (a) Lös de problem som beskrevs i fråga 3 enbart med hjälp av de atomiska operationer som finns i kodlistning 2 på sidan 9. Du behöver dock **inte** lösa problemen med *busy-wait*. [3p]

Modifiera koden i filen.

- (b) Varför är det inte möjligt att lösa problemen med *busy-wait* enbart med hjälp av atomiska operationer? [2p]

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