Linköpings universitet
Institutionen för datavetenskap 2022-06-01

Tentamen i
TDIU16 Process- och operativsystemprogrammering

Datum 2022-06-01

Tid 08–12

Institution IDA

Kurskod TDIU16

Provkod DAT1

Examinator

Filip Strömbäck (filip.stromback@liu.se)

Administratör

Annelie Almquist

Jourhavande lärare

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 5 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`) eller i OpenOffice-format (`.odt`). Filnamn anges i uppgifterna.
- 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. 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 ifall ni vill testa något, men uppgifterna ska vara korrekta utan extra synkronisering 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 du har kopierat alla givna filer ifall det inte fungerar.

1. I ett system körs tre processer, *P1*, *P2* och *P3*. I systemet finns det tre typer av resurser, *A*, *B* och *C*. Det finns 5 instanser av vardera resurs. 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 `uppgift1.txt` *eller* `uppgift1.odt`*.*

	A	B	C
P1	3	2	2
P2	3	3	5
P3	2	2	3

Maximal resursanvändning

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

Nuvarande resursanvändning

Tabell 1: Resurser i systemet.

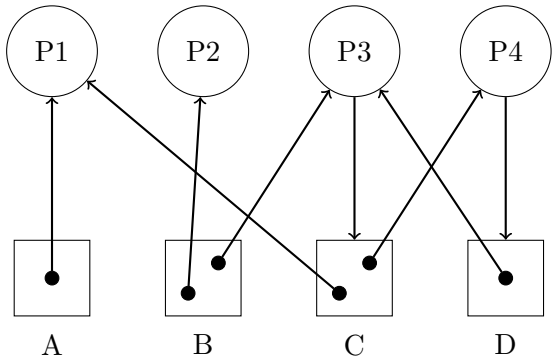
   (a) Är systemet i ett säkert läge enligt Banker's algorithm? Redovisa dina beräkningar. [2p]

   (b) Process *P2* 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 avslutas.
      2. Processen som begärde en extra resurs får vänta tills alla andra processer har kört klart.
      3. Processen som begärde en extra resurs får vänta tills någon process har lämnat tillbaka en instans av den efterfrågade resursen.
      4. Processen som begärde en extra resurs får vänta. Varje gång någon process har lämnat tillbaka en resurs kör vi Banker's igen och testar ifall processen får fortsätta.

2. Nedan finns en resursallokeringsgraf för ett system med fyra processer och fyra resurser.



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

   (a) Vilka av de fyra processerna väntar på något? [1p]

   (b) Är systemet i deadlock? I så fall, vilka processer är i deadlock? Motivera ditt svar. [1p]

3. Din kompis Kim har skrivit ett program som hanterar tåg. Programmet finns i filen `trains.c` och innehåller en datastruktur, `track`, som representerar ett enkelriktat spår mellan två städer. Alla tåg på spåret åker alltså åt samma håll. För att tillåta flera tåg att åka på samma spår på ett säkert sätt är spåren uppdelade i ett antal segment. Styrsystemet för spåret ser sedan till att det bara finns ett tåg i varje segment.

   Målet med datastrukturen `track` är alltså att hålla reda på vilka tåg som är på vilka segment. Den ska också se till att det maximalt är ett tåg i varje segment genom att låta tågen vänta när det behövs. Datastrukturen består av följande funktioner:

   `track_create` Skapar ett spår som består av ett givet antal segment. Parametern `direction` anger i vilken riktning tågen åker (antingen från låga index till höga index, eller vice versa).

   `track_enter` Anropas för att sätta in ett tåg i början av spåret. Väntar tills det första segmentet på spåret är ledigt.

   `track_move` Anropas för att flytta ett tåg från ett segment till nästa. Väntar tills nästa segment är ledigt.

   `track_remove` Anropas när ett tåg lämnar det sista segmentet av spåret.

   `track_draw` Ritar ut spåret till terminalen.

   `track_free` Frigör ett spår.

   Programmet innehåller också kod som skapar två spår. Ett med tåg som går åt ena hållet, och ett åt andra. Programmet startar sedan 12 trådar som var och en är ansvariga för att köra ett tåg på ett av spåren och ritar ut spåren tills alla tåg har kört klart. Detta sköts av funktionerna `main` och `run_train`. Du behöver dock inte modifiera dessa funktioner, utan du kan anta att de fungerar korrekt.

   Tyvärr verkar det som att datastrukturen `track` inte fungerar som den ska. Trots att 12 tåg skapas så syns inte alla i terminalen. Ibland ser det till och med ut som att ett tåg åker in i ett annat så att bara det ena av tågen finns kvar!

   Det finns också ett problem med utritningen. Tanken är att varje tåg ska visas på exakt ett ställe så länge det är på ett spår. Däremot så verkar det inte som att så är fallet. Ibland visas samma tåg på två segment bredvid varandra i en kort stund.

   Du kan kompilera koden med kommandot `make trains`. Se till att du har kopierat `Makefile` om det inte fungerar. Du kan sedan köra programmet med `./trains`.

   Skriv dina svar i din kopia av `trains.c` och skicka in den modifierade filen.

(fortsättning på nästa sida)

(a) Beskriv med ett exempel vad som kan ha skett då ett eller flera tåg på samma spår försvinner. [1p]

Skriv ditt svar i kommentaren i början av filen.

(b) Beskriv med ett exempel vad som kan ha skett då samma tåg (samma siffra) visas i fler än ett segment i utskriften från programmet. [1p]

Skriv ditt svar i kommentaren i början av filen.

(c) Finns det några instanser av busy-wait i koden? [1p]

*Markera de busy-wait som finns i filen, exempelvis som nedan. Använd **inte** radnummer, de blir felaktiga när du svarar på resten av frågorna.*

```
1 // Busy wait på denna rad
```

(d) Markera de kritiska sektioner som finns i koden. För varje kritisk sektion, markera också vilken eller vilka resurser som är kritiska. [2p]

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

```
1 // Kritisk sektion 1 börjar. Resurs: data.z
2 // Kritisk sektion 1 slutar.
```

(e) Lös de synkroniseringsproblem du hittade i (c) och (d) med hjälp av lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 8. [4p]

Notera: 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 filen.

(f) Vilka är de fyra villkoren för deadlock? Beskriv varje villkor kort (med en mening). [2p]

Skriv ditt svar i kommentaren i början av filen.

(g) Innehåller din lösning deadlock? Använd de fyra villkoren för deadlock för att motivera ditt svar. [1p]

Skriv ditt svar i kommentaren i början av filen.

4. Du har hittat ett bibliotek som hanterar vektorgrafik. I biblioteket representeras bilder som en uppsättning linjer (linjesegment). Biblioteket finns i filen `picture.h`. För att kunna visa bilderna på skärmen måste de konverteras till pixlar (detta kallas *rasterisering*). För att göra detta innehåller biblioteket en funktion (`picture_compute_row`) som skapar en rad av pixlar för en specifik y-koordinat. Din plan är därför att anropa denna funktionen för alla y-koordinater från 0 till höjden av bilden. Detta fungerar bra, men är långsamt. Du inser att du kan konvertera flera rader parallellt på olika trådar för att snabba upp processen.

   Din lösning på problemet finns i filen `draw.c`. I filen finns din funktion `draw_picture` som ska använda ett visst antal trådar för att konvertera bilden och rita ut den till skärmen. Funktionen startar trådar som kör funktionen `draw_worker`. Dessa trådar hittar nästa rad som ska konverteras, gör konverteringsarbetet, och skriver sedan ut raden till terminalen när det är dags. Detta fungerar dock inte som det ska. Ibland kommer raderna ut i fel ordning. Du misstänker därmed någon form av synkroniseringsfel.

   Datastrukturerna som används (främst `struct picture`) finns definierade i `picture.h`. Du ska inte behöva modifiera något inuti `picture.h` för att lösa uppgiften.

   Du kan kompilera koden med kommandot `make draw`. Se till att du har kopierat `Makefile` om det inte fungerar. Du kan sedan köra programmet med `./draw <filnamn>` där `<filnamn>` är namnet på en `.txt`-fil som innehåller en bild. Exempelvis `./draw cross.txt` eller `./draw pintos.txt`.

   Skriv dina svar i din kopia av `draw.c` och skicka in den modifierade filen.

   Tips: Fokusera på funktionerna `draw_picture` och `draw_worker`. Du kan anta att datan i bilden inte modifieras under tiden som den ritas ut.

   (a) Ge ett exempel på vad som kan ha hänt när två (eller flera) rader har skrivits ut i fel ordning (dvs. i vilken ordning har olika satser körts för att det ska bli fel?). [1p]

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

   (b) Finns det några instanser av busy-wait i `draw_picture` eller `draw_worker`? [1p]

   *Markera de busy-wait som finns i filen `draw.c`. Exempelvis som nedan. Använd **inte** radnummer, de blir felaktiga när du svarar på resten av frågorna.*

   ```
   // Busy wait på denna rad
   ```

   (c) Vilka kritiska sektioner finns i `draw_picture` och `draw_worker`-funktionerna? För varje kritisk sektion, markera också vilken eller vilka resurser som är kritiska. [2p]

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

   ```
   // Kritisk sektion 1 börjar. Resurs: data.z
   // Kritisk sektion 1 slutar.
   ```

   (d) Lös de synkroniseringsproblem du hittade i (b) och (c) med hjälp av lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 8. [4p]

   Notera: 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 `draw.c`.

5. Lös de problem som beskrivs i fråga 4 enbart med hjälp av de atomiska operationerna som finns i kodlistning 2 på sidan 9. Du kan självklart utgå från din lösning till fråga 4 om du vill, men all synkronisering ska göras med atomiska operationer. [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 `draw.c`*, och spara den som exempelvis* `atomics.c`*. Skriv sedan dina svar i den filen och skicka in den.*

Tillgängliga synkroniseringsprimitiver

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

```
struct semaphore {
    // ...
};

void sema_init(struct semaphore *sema, unsigned value);
void sema_destroy(struct semaphore *sema);
void sema_down(struct semaphore *sema);
void sema_up(struct semaphore *sema);

struct lock {
    // ...
};

void lock_init(struct lock *lock);
void lock_destroy(struct lock *lock);
void lock_acquire(struct lock *lock);
void lock_release(struct lock *lock);

struct condition {
    // ...
};

void cond_init(struct condition *cond);
void cond_destroy(struct condition *cond);
void cond_wait(struct condition *cond, struct lock *lock);
void cond_signal(struct condition *cond, struct lock *lock);
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.

```c
int test_and_set(int *value) {
    int old = *value;
    *value = 1;
    return old;
}

int atomic_swap(int *value, int replace) {
    int old = *value;
    *value = replace;
    return old;
}

int compare_and_swap(int *value, int compare, int swap) {
    int 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;
}

int atomic_read(int *value) {
    return *value;
}

void atomic_write(int *value, int write) {
    *value = write;
}
```

Kodlistning 2: Atomiska operationer