Linköpings universitet
Institutionen för datavetenskap 2023-05-31

Tentamen i
TDIU16 Process- och operativsystemprogrammering

Datum 2023-05-31

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. 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 5 uppgifter på 10 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.
- 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 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. 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. Systemet är i ett säkert läge.

   Uppdatera

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

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

Maximal resursanvändning

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

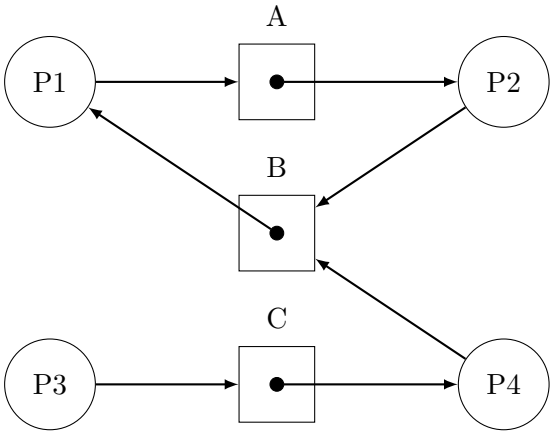
Nuvarande resursanvändning

Tabell 1: Resurser i systemet.

   (a) Process *P2* begär en extra resurs av typ *B*. Vad ska hända med processen enligt Banker's Algorithm? Redovisa dina beräkningar. [2p]

   (b) Process *P1* begär sedan en extra resurs av typ *A*. Vad ska hända med processen enligt Banker's algorithm? Redovisa dina beräkningar. [2p]

2. Nedan finns en resursallokeringsgraf för ett system med fyra processer och tre resurser. [2p]



   Är systemet som beskrivs av grafen i deadlock? Motivera ditt svar.

   Lämna in svar som `uppgift2.txt`*,* `uppgift2.odt` *eller* `uppgift2.pdf`*.*

3. En av dina vänner har nyligen sommarjobbat på startupen SJ (Snålköpings Järnvägar). Som en del av sommarjobbet har din vän fått utveckla ett system för att hålla koll på platsbokningar på tågen. För att maximera kapaciteten på tågen ska systemet inte bara hålla koll på vilken passagerare som har vilken plats, utan även vilken delsträcka som passageraren är bokad på. Exempelvis, om ett tåg går från Snålköpings Central via Uppsala med slutstation Stockholm ska det vara möjligt för en passagerare att boka plats nummer 1 på delsträckan Snålköping C – Uppsala, samtidigt som en annan passagerare ska kunna boka samma plats på delsträckan Uppsala – Stockholm. En passagerare ska aldrig behöva byta plats under sin resa.

   För att hålla reda på detta finns en datatyp `seat` som representerar bokningen av en plats på ett tåg. Datatypen håller koll på vilken passagerare som har bokat platsen under vilka delsträckor. För enkelhets skull antar vi att stationerna är numrerade från 0 och uppåt. I exemplet ovan är alltså Snålköping C nummer 0, Uppsala nummer 1, och Stockholm nummer 2. Datastrukturen lagrar då passageraren på delsträckan Snålköping C – Uppsala i plats nummer 0 i arrayen, och passageraren på delsträckan Uppsala – Stockholm på plats nummer 1 i arrayen. För enkelhets skull lagras bokningarna som kundnummer (0 och uppåt). Talet -1 används för att indikera att platsen är ledig på den delsträckan.

   Utöver detta så finns datatypen `train`, som representerar alla platser i ett helt tåg. Utöver funktioner för att skapa och frigöra ett tåg finns två funktioner:

   `train_book`: Letar efter en plats som är ledig under den angivna delsträckan. Delsträckor anges som startstation och slutstation. Exempelvis: 0, 2 motsvarar sträckan Snålköping C – Stockholm i exemplet ovan. Om en ledig plats hittas så bokas platsen och platsens nummer returneras. Annars returneras -1.

   `train_print`: Skriver ut en översikt över vilka platser som är bokade under tågets resa. Varje rad i utskriften representerar en delsträcka, och varje kolumn motsvarar en plats.

   Du kan anta att innehållet i både `train` och i `seat` endast modifieras med hjälp av funktionerna som hör till datatypen (dvs. de är privata). Du kan också anta att funktionerna som börjar med `seat_` endast anropas av funktionerna som börjar på `train_`.

   Efter mycket testande har din kompis insett att det finns problem med implementationen. När SJ började köra bokningar från flera trådar samtidigt (för att maximera bokningskapaciteten, och därmed vinsten) så började kunder klaga på att platser ibland blev dubbelbokade under delar av resan. Dessutom har det hänt att en kund med en giltig biljett har fått sin plats avbokad. Efter mycket letande har de kommit fram till att problemet ligger i koden som din vän utvecklade under sommaren. Eftersom du nyligen har läst en kurs i parallellprogrammering har din vän nu kommit för att be dig att lösa problemet!

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

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

(fortsättning på nästa sida)

(a) Beskriv med ett exempel vad som kan ha gått fel när en plats har blivit dubbelbokad på en (eller flera) delsträckor. [1p]

Skriv ditt svar i kommentaren i början av filen.

(b) Använd lämpliga synkroniseringsprimitiver från kodlistning 1 på sidan 9 för att lösa problemet du identifierade ovan. [4p]

För full poäng ska din lösning tillåta att bokning av oberoende delsträckor kan köras parallellt.

Om funktionen `train_book` tar parametrarna `train_book(customer, from, to)`, så ska alltså anropen `train_book(1, 2, 4)` och `train_book(2, 5, 7)` gå att köra parallellt, utan att det ena behöver vänta på den andra.

Notera att även `train_print`-funktionen kan köras när som helst i bokningsprocessen. Lös även eventuella synkroniseringsfel som finns i samband med `train_print`.

Modifiera koden i filen.

(c) Vilka är de fyra villkoren för deadlock? Namnge villkoren och ge en kort beskrivning. [2p]

Skriv ditt svar i kommentaren i början av filen.

(d) Innehåller din lösning deadlocks? Använd de fyra villkoren för deadlock för att motivera ditt svar. [2p]

Skriv ditt svar i kommentaren i början av filen.

(e) Antag att ett tåg är helt fullbokat, förutom plats nummer 1. Plats nummer 1 är bara ledig mellan station 1 och station 2. [2p]

I den här situationen anropar tråd A `train_book(1, 1, 2)` och samtidigt anropar tråd B `train_book(2, 1, 4)`. Tråd A försöker alltså boka en plats mellan station 1 och station 2, medan tråd B försöker boka en plats mellan station 1 och station 4.

Eftersom det inte finns några lediga platser efter hållplats 2 så kommer bokningen som tråd B försöker göra inte att lyckas, oavsett om den hinner före tråd A eller inte.

Garanterar din implementation att bokningen som tråd A gör kommer att lyckas i den här situationen? Eller är det möjligt att bokningen som tråd B försöker göra kan få tråd A:s bokning att misslyckas? Motivera ditt svar.

Skriv ditt svar i kommentaren i början av filen.

(f) Antag att en tråd anropar funktionen `train_print` samtidigt som en annan tråd anropar `train_book` för att boka en plats på 4 delsträckor. Är det då i din implementation garanterat att utskriften innehåller antingen 0 av dessa eller alla 4? Det vill säga, vi kommer aldrig att se endast 1–3 av delsträckorna bokade? [2p]

Skriv ditt svar i kommentaren i början av filen.

4. Fraktaler har fascinerat många genom åren. En av det mest kända fraktalerna är antagligen Mandelbrot-fraktalen. Trots att idén och matematiken bakom fraktalen är väldigt enkel så innehåller den många komplexa detaljer. Idén är att vi har en funktion, $f(z)$, som tar emot ett komplext tal, och producerar ett annat komplext tal. Vi tänker oss dessa komplexa tal som punkter i 2 dimensioner. För att bestämma vilken färg en punkt p i vår bild ska ha sätter vi först $z = 0$, sedan beräknar vi $z = f(z) + p$ om och om igen tills dess att $|z|$ (beloppet av z, avståndet från origo) är tillräckligt stort. Antalet iterationer som behövs innan $|z|$ blir tillräckligt stort bestämmer sedan färgen av punkten. Ett fåtal iterationer ger en ljus färg, och många iterationer ger en mörk färg. För Mandelbrot-fraktalen använder vi $f(z) = z^2$.

   Självklart är detta opraktiskt att göra manuellt. Filen `fractal.c` innehåller därför ett program som utför ovanstående beräkning. Programmet gör helt enkelt processen ovan många gånger för olika punkter för att producera en bild. Detta kräver dock en hel del datorkraft, speciellt för högupplösta bilder. Programmet använder därför flera trådar (definierat av `THREAD_COUNT`) för att snabba upp beräkningen.

   Funktionen `fractal_compute_point` implementerar processen att beräkna antalet iterationer som krävs i en specifik punkt. Funktionen `fractal_iter_to_char` översätter sedan detta till ett motsvarande tecken. Dessa funktioner fungerar som de ska.

   Programmet innehåller också funktionen `fractal_print`. Denna funktionen startar ett antal trådar som var och en kör funktionen `fractal_worker`. Tråden som kör `fractal_print` ber sedan trådarna som kör `fractal_worker` att beräkna alla tecken i den första raden. När trådarna är klara så skriver `fractal_print`-tråden ut raden. Efter det ber den `fractal_worker`-trådarna att beräkna nästa rad, och så vidare.

   Tyvärr är koden i `fractal.c` inte korrekt synkroniserad. Problemen är så pass stora att det endast kommer ut en enstaka rad, om något alls. Det verkar som att antingen `fractal_print`-tråden och/eller `fractal_worker`-trådarna fastnar någonstans. De få gångerna det fungerar som det ska ser man också ibland att vissa tecken inte har beräknats på nytt, och innehåller samma tecken som i raden ovanför.

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

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

(fortsättning på nästa sida)

(a) Beskriv med exempel vad som kan ha hänt då: [1p]

- ...programmet fastnar.
- ...programmet inte har uppdaterat en eller flera pixlar från en tidigare rad.

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. Du kan antingen kopiera de delar som innehåller busy-wait till kommentaren i början eller markera dem i källkoden (exempel nedan). Använd **inte** radnummer, de blir felaktiga när du svarar på resten av frågorna.*

```
1 // Busy-wait på denna rad.
```

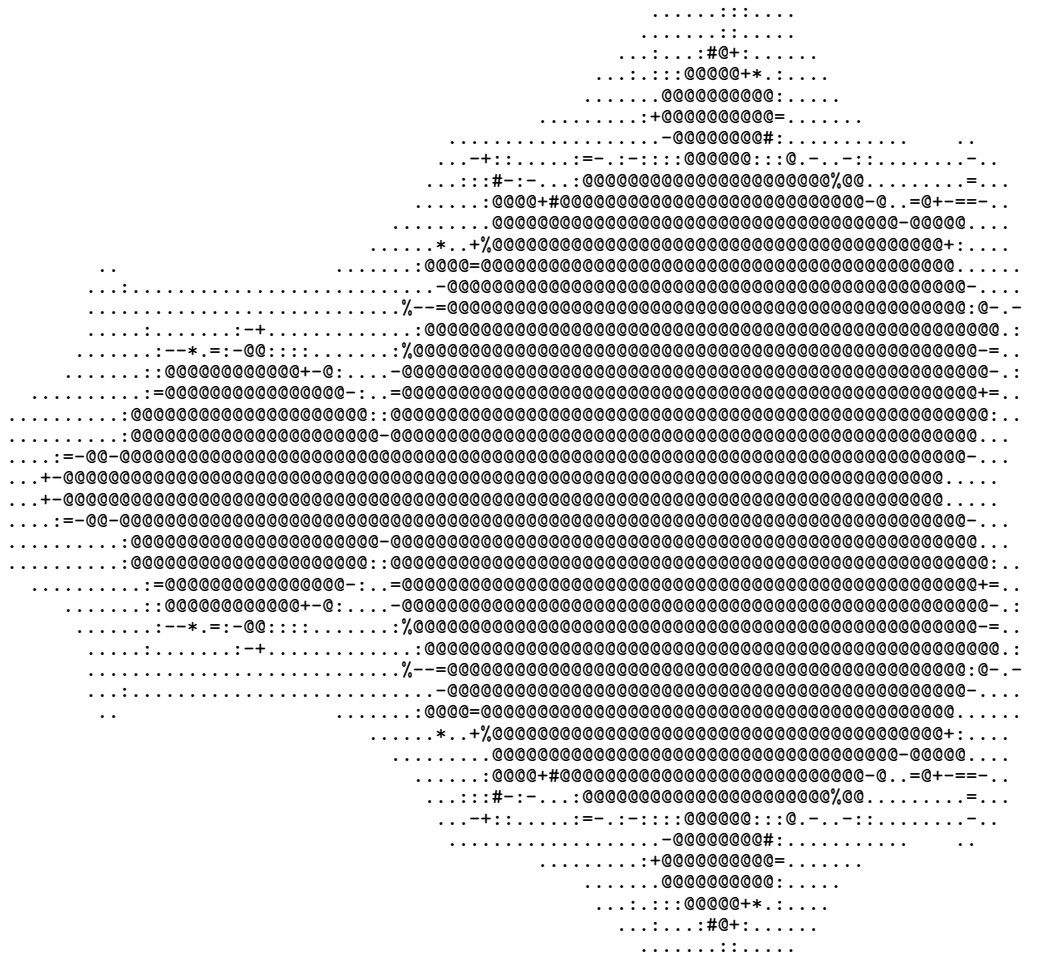
(c) Lös de *busy-wait* du identifierade i förra uppgiften med hjälp av lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 9. [4p]

Modifiera koden i filen.

(d) Lös problemen du identifierade i (a) med hjälp av lämpliga synkroniseringsprimitiver från kodlistning 1 på sida 9. [2p]

Modifiera koden i filen.

Nedan finns den ASCII-version av Mandelbrot-fraktalen som programmet skriver ut när det fungerar korrekt:



5. Lös de problem som beskrivs i fråga 4 enbart med hjälp av de atomiska operationerna som finns i kodlistning 2 på sidan 10. 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 `fractal.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