Tentamensskrivning — Realtidsprogrammering — Java
2001-08-23, kl 8.00-13.00

Anvisningar

Tillåtna hjälpmedel: inga utöver Java snabbreferens.

För godkänt betyg krävs att större delen av de 8 första uppgifterna samt en av uppgifterna 9 eller 10 behandlas nöjaktigt. För högre betyg krävs naturligtvis mer, så gör så många uppgifter du kan.

Senast 3:e arbetsdagen efter tentamen anslås på institutionens anslagstavla vilka som deltagit i tentamen men som, enligt institutionens noteringar, ännu inte redovisat övningarna, laborationerna eller projektet. Deras skrivning rättas inte (och kan komma att annulleras) såvida inte rättelse skett eller dispens erhållits från ansvarig lärare. Rättelse skall göras senast den 6:e arbetsdagen efter tentamen.

Uppgifter

1. Utöver vad som skall gälla för rent sekventiella program, vad skall gälla för att ett program med jämlöpande trådar/processer skall betraktas som korrekt? (1p)

2. Utöver vad som skall gälla för rent sekventiella program, vad skall gälla för att ett realtidsprogram skall betraktas som korrekt? (1p)

3. Varför skulle det vara fördelaktigt med automatisk minneshantering (d.v.s användande av en s.k. Garbage Collector eller GC) i realtidssystem? (1p)

4. Varför är schemaläggningsprincipen *round-robin* vanlig för interaktiva system utan speciella krav på real tid? (1p)

5. Varför är det svårt att låta operativsystemprocesser kommunicera via gemensamma variabler? Hur brukar man göra istället? (2p)

6. En vanlig användning av en semafor är att åstadkomma ömsesidig uteslutning (mutual exclusion) över en kritisk region. I detta fall är det alltid samma tråd som låser semaforen (mha `take()`) och som släpper den (mha `give()`) när den kritiska regionen passerats.

Implementera en klass `MutexSemaphore` som har två publika operationer, `give()` och `take()`. Operationerna får vara deklarerade `synchronized`, men måste inte vara det.

Klassen `MutexSemaphore` ska fungera som en vanlig binär semafor, men den ska dessutom kontrollera att det är *samma tråd* som anropar `give()` som det var som låste semaforen mha `take()`. Detta används för att garantera korrekthet hos programmet och för felsökning.

Om någon annan tråd anropar `give()` än den som tidigare anropade `take()`, eller om semaforen inte var låst när `give()` anropades, skall ett `IllegalGiveError` genereras i anropet av `give()` (görs med satsen "`throw new IllegalGiveError();`"). Klassen `IllegalGiveError` antas finnas fördefinierad och vara en subklass till standardklassen `Error`.

För att garantera portabilitet får endast Javas normala synkroniseringsmekanismer användas, dvs inte klasserna i paketet `se.lth.cs.realtime` eller liknande. Du får naturligtvis komplettera klassen med egna attribut, operationer och konstruktor efter behov.
Tips: En referens till den nu körande tråden kan erhållas genom att till exempel skriva "`Thread t = Thread.currentThread();`". (3p)

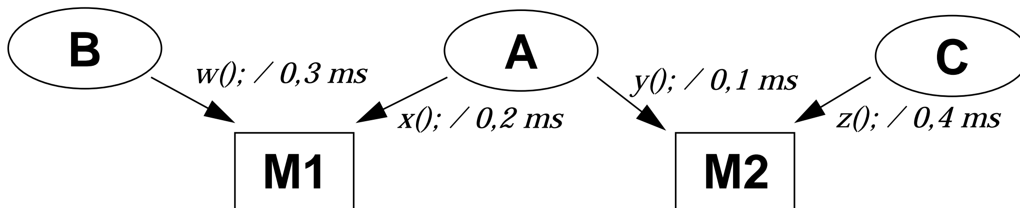
7. Betrakta ett system bestående av tre oberoende periodiskt exekverande trådar med nedanstående karaktäristika (C = värstafallsexekveringstid, D = deadline, T = period).

Tråd	*C (ms)*	*D (ms)*	*T (ms)*
A	2	6	10
B	9	25	25
C	3	7	7

a) Vad blir de maximala svarstiderna för de tre olika trådarna om man tillämpar Rate Monotonic Scheduling (RMS)? (3p)

b) Är systemet schemaläggningsbart, dvs klarar systemet alltid av alla sina deadlines? Motivera ditt svar! (1p)

8. De tre högst prioriterade trådarna, kallade A, B och C, i ett realtidssystem kommunicerar med varandra via två monitorer, M1 respektive M2, enligt följande figur:



Monitoroperationerna *w*, *x*, *y* och *z* anropas av trådarna A, B och C enligt ovanstående figur en gång varje respektive tråds period. Anropen är inte nästlade, dvs man anropar inte en monitoroperation inuti en annan monitoroperation. För varje monitoroperation anges även maximal tid det kan ta att exekvera operationen i figuren.

Tråden A har högst prioritet och tråden C har lägst prioritet av de tre trådarna. Trådarna kommunicerar inte med andra (lägre prioriterade) trådar än de som nämnts här. För övrigt gäller följande för trådarna (C = värstafallsexekveringstid, D = deadline, T = period):

Tråd	*C (ms)*	*D (ms)*	*T (ms)*
A	1,0	50	50
B	1,5	100	100
C	2,0	200	200

a) Vad blir den maximala tiden som var och en av trådarna A, B och C kan blockeras av *lägre prioriterade* trådar under en period under förutsättning att *dynamisk prioritetsärvning (basic priority inheritance)* används? Observera att det är den maximala *blockeringstiden* som söks, inte trådarnas svarstider. (2,5p)

b) I vissa situationer har man inte tillgång till en realtidskärna med prioritetsärvning. Vad blir i så fall den maximala tiden som var och en av trådarna A, B och C kan blockeras av *lägre prioriterade* trådar? (2,5p)

9. Meddelandekonvertering

Jämlöpande aktiviteter inom ett program, s.k. trådar, kommunicerar ofta via meddelandeköer, s.k. eventbuffert:ar. Något som är speciellt för fallet då de har producent-konsument-förhållande är att producentens exekveringsfrekvens kan variera. För denna vanliga situation brukar vi använda oss av klassen `RTEventBuffer`, direkt eller via klassen `RTThread`. Varje mottaget `RTEvent` kan normalt behandlas klart, och ev skickas vidare, innan nästa `RTEvent` tas emot.

Lite annorlunda blir det om flera inkommande `RTEvent` måsta tas emot innan utdata kan produceras. I följande uppgift krävs dessutom att producenten inte får fördröjas av tidskrävande operationer eller blockering. För att renodla problemet antar vi fixa buffertstorlekar och event-typer enligt följande:

- Inkommande eventobjekt som är av typen `JPEGEvent` skall samlas ihop så att det finns fyra stycken, ordnade och direkt på varandra följande, eventobjekt som sedan skall omvandlas till ett nytt objekt av typen `MPEGEvent`. Detta sker genom att anropa
  ```
  mpegEv = ImgLib.compress(jpgEv1, jpgEv2, jpgEv3, jpgEv4);
  ```
  där in- och ut-argumenten utgörs av JPEG- respektive MPEG-eventobjekten.
- `JPEGEvent` och `MPEGEvent` är båda subklasser till `RTEvent`. Vi behöver inte känna till deras innehåll i denna uppgift.
- Andra typer av event skall bara skickas vidare, d.v.s. ingen behandling krävs.
- Samtliga utgående (MPEG-event eller vidarebefodrade) event `ev` skickas till sin gemensamma destinationen genom att anropa
  ```
  Player.putEvent(ev);
  ```

Din uppgift är att utveckla en klass **`Bundler`** (med ev ytterligare klasser och trådar i den mån de behövs) enligt följande specifikationer:

1. Det skall finnas en metod `putEvent(RTEvent)` som olika producenttrådar kan anropa för att skicka sina eventobjekt.
2. Anroparen av `putEvent` (enligt punkt 1) för inte fördröjas av långa beräkningar eller blockering. Ett enkelt omhändertagande av eventobjektet är givetvis acceptabelt, men en tidskrävande uppgift som att konvertera fyra JPEG-event till ett MPEG-event är det till exempel inte.
3. Upp till 20 stycken objekt skall kunna buffras i Bundler utan att varken blockering av anroparen eller förlust av eventet sker.
4. Då produktionen och skickande av event överstiger det maximala antalet enligt punkt 3 så skall de ytterligare eventobjekten kastas.
5. Det får inte finnas bortkastade event inom perioden för en MPEG-frame. D.v.s., argumenten till `ImgLib.compress` ovan måste vara fyra på varande följande JPEG-event; om ett JPEG-event behövt kastas så skall övriga tidigare JPEG-event för det tänkta MPEG-eventet också kastas.

Till din hjälp finns bifogad infomation om klassen RTEventBuffer.

(8p)

10. Styrning av robotarm

I ett robotstyrsystem skall börvärden till reglering av robotens olika leder beräknas. Detta göres utifrån aktuell specifikation av robotens uppgift, och resulterar i beräknade börvärden som sedan skall överföras till regleringen vilken sker i en annan processor. I ett första steg har geometrisk och dynamisk planering av rörelsen skett i en tråd `Planner` som finns färdig. Baserat på utdata från `Planner` skall börvärdena till regleringen beräknas. Vår uppgift är att utveckla programvaran som hanterar denna beräkning samt överför beräknade börvärden till reglerprocessorn. Via operatören eller från inbyggda systemfunktioner kan det även komma speciella order om förändringar i regleringen. Exempelvis skall ett nödstopp först resultera i att speciella börvärden för snabbast möjliga stopp av pågående robotrörelse skickas till regleringen, och därefter skall återstart av normal rörelse förberedas.

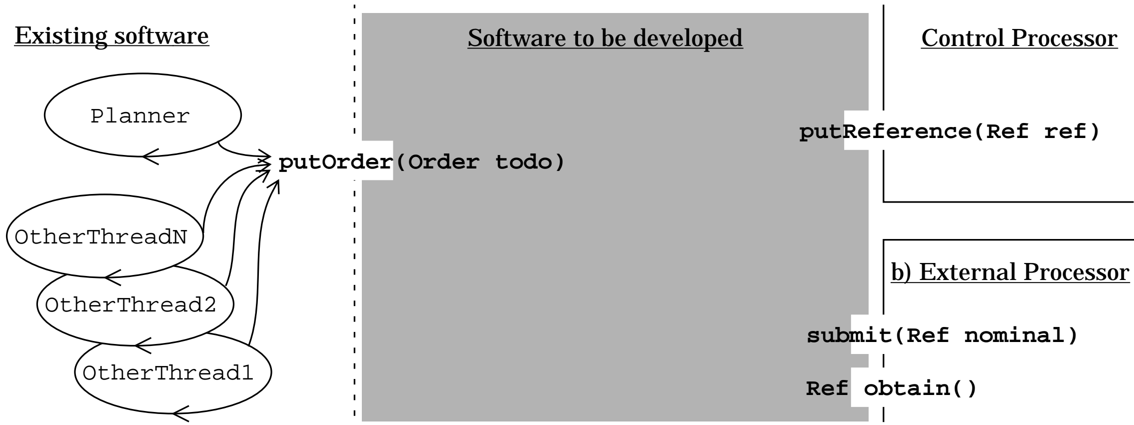
Det är ytterst viktigt att gällande realtidspecifikationer uppfylls eftersom: 1) Om beräkningen av börvärden fördröjs så kommer roboten att röra sig ryckigt på ett sätt som orsakar bristande kvalité, mekaniskt slitage samt sporadiska nödstopp då den inbyggda övervakningen detekterar felaktig rörelse. 2) De speciella order som ibland skall hanteras får varken buffras upp eller fördröjas mer än tiden det tar att hantera en rörelseorder, c:a 2 ms. I annat fall äventyras personsäkerheten vilket kan innebära livsfara för robotoperatören.

Följande gäller för det befintliga systemet:

- Tråden **`Planner`** körs med prioriteten `MAX_PRIORITY-1` och med en periodtid på 4 ms. Den får fördröjas högst 1 ms (för att tillsammans med det följande under varje 4 ms period ge minst 1 ms lucka för ordergenerering och annat för andra trådar).
- Inga andra existerande trådar i systemet har samma eller högre prioritet än `Planner`-tråden.
- Schemaläggningen sker strikt efter prioritetsordning.
- En Order definieras av en klass enlig:
```
public class Order extends RTEvent {}
```
- Planner skickar order av typen
```
public class Step extends Order {
    // Specification of motions step goes here, not for you to write.
}
```
  via metoden `putOrder` i en klass `ArmHandler` som du skall implementera. Varje order av typen `Step` anger hur ytterligare ett steg skall tas längs den för roboten specificerade banan.
- För varje mottagen order av typen `Step` skall nya börvärden beräknas genom anrop av den redan existerande (statiska) rutinen `computeRef` i klassen `MotionLib`. D.v.s. ett börvärde `ref` beräknas enligt
```
ref = MotionLib.computeRef(stepOrder);
```
  vilket tar minst 1,4 och högst 2 ms CPU-tid!
- Ett beräknat börvärde `ref` skickas till reglerprocessorn genom anropet
```
ArmControl.putReference(ref);
```
  vilket tar försumbar tid.
- En order av kommandotyp definieras av klassen
```
public class Command extends Order {
    // Specification of commands (stopping, recovery, etc.) goes here,
    // not for you to write.
}
```
- Ett `Command`-order `cmd` skall i din programvara hanteras genom att anropa den färdiga rutinen `handle` enligt
```
CommandLib.handle(cmd);
```
  vilket tar mellan 0,1 och 0,5 ms att utföra.
- Implementeringen av `putOrder` måste vara sådan att anroparen endast fördröjs försumbar tid, exempelvis för lagring av ordern för vidare bearbetning av annan tråd.

- Då ny order har givits (genom att anropa `putOrder`) måste denna först utföras klart innan hantering av ny order påbörjas (med ett undantag i uppgift b nedan).

Följande figur illustrerar systemet där den ▒▒-markerade delen skall utvecklas i form av klassen `ArmHandler` och ev andra klasser och egna trådar.



a) Utför design och implementering (inkl prioritetssättning) utifrån ovan angivna systemegenskaper och så att en kommandoorder från en tråd (någon `OtherThread` enligt figur) som ensam har prioriteten `MAX_PRIORITY-2` garanterat inom 5 ms skall både kunna ge sin order och få den utförd (senast 5 ms efter anrop av `putOrder`). (5p)

b) För att förbättra robotens förmåga att anpassa sina rörelser till en delvis okänd omgivning så vill man göra det möjligt att låta en extern processor (till vilken externa sensorer kopplats) modifiera börvärdena innan de skickas till den interna regleringen enligt uppgift a. Detta hanteras från din programvara genom att mellan anropen av `computeRef` och `putReference` enligt ovan först anropa den befintliga metoden `submit` enligt

```
ExternalControl.submit(ref);
```

och sedan erhålla ett ev. modifierat börvärde genom att anropa den befintliga metoden `obtain` enligt

```
ref = ExternalControl.obtain();
```

som returnerar sitt värde inom 1 ms efter det att `submit` anropats. Innan nytt värde finns klart så är anrop av `obtain` blockerande.

Notera att de trådar som anropar `putOrder` inte får blockeras, en order enligt anrop av `putOrder` skall vara utförd inom 5 ms, varje order skall hanteras i tur och ordning förutom att kommandoorder får utföras under tiden den externa processorn beräknar nytt returvärde till obtain.

(5p)