LINKÖPINGS UNIVERSITET

TDIU16: Process- och operativsystemprogrammering

Lab 4: Synkronisering av filsystemet

Filip Strömbäck, Klas Arvidsson, Daniel Thorén

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2023

2023-03-16

Mål

Filsystemet som finns i Pintos fungerar i nuläget bra så länge inte flera trådar (eller processer) försöker använda det samtidigt. Det är såklart väldigt vanligt att flera processer eller trådar försöker använda filsystemet samtidigt, så vi måste se till att filsystemet fungerar korrekt även i dessa fall.

Del A Övning på synkronisering

Målet med denna del är att öva på att identifiera och synkronisera delad data inför nästkommande delar. Vi gör det i ett litet program som är externt från Pintos. Programmet efterliknar hur koden i filen `src/filesys/inode.c` är uppbyggd, så din lösning till den här delen kan användas som inspiration för synkroniseringen i Pintos senare.

I mappen `src/standalone/lab4a` finns filen `data-files.c` som implementerar ett litet system för att effektivt använda ”stora” datafiler. Tanken är att det finns två datafiler som resten av programmet behöver komma åt med jämna mellanrum. När någon del av programmet behöver en av filerna så vill vi ladda in hela filen i RAM så att det går snabbt att komma åt datan i filen. I och med att filerna är stora vill vi se till att 1) vi inte laddar in varje fil mer än en gång i RAM och 2) vi vill ta bort kopian i RAM så snart vi vet att vi inte behöver den längre.

Det finns tre funktioner som är extra intressanta i implementationen:

`data_open` Öppnar datafilen med det givna indexet (0 eller 1). Om filen redan var öppen så ska den öppna filen återanvändas.

`data_reopen` Anropas ifall en användare av en datafil vill öppna samma datafil igen. Varje anrop till `data_reopen` gör därmed att ett extra anrop till `data_close` behövs för att filen ska tas bort.

`data_close` Stänger en tidigare öppen datafil. När alla som använder filen har stängt den så ska datastrukturen frigöras för att spara minne.

För att hålla koll på hur många gånger en fil har blivit öppnad används en räknare vid namn `open_count`. Den ökas varje gång en fil öppnas, och minskas varje gång en fil stängs. På det viset kan implementationen hålla koll på när det är säkert att ta bort kopian i RAM. Detta liknar hur Pintos hanterar inodes.

Implementationen innehåller för närvarande synkroniseringsfel som gör att implementationen inte fungerar som den ska. Exempelvis kan det bli så att samma fil läses in i RAM mer än en gång. Det kan också bli så att en fil tas bort innan alla som öppnat den är färdiga med att använda den. Din uppgift är att lösa de synkroniseringsproblem som finns. Sträva efter en lösning som tillåter maximal möjlig parallellism. Det vill säga: är det säkert att låta olika trådar köra exempelvis `data_reopen` samtidigt så ska din implementation tillåta detta.

Likt i uppgiften ”Övning på semaforer” så går programmet att köra i visualiseringsverktyget, men även direkt från terminalen.

A.1 Visualiseringsverktyget

På skolans datorer och ThinLinc kan du starta verktyget med kommandot `/courses/TDIU16/progvis/start` Du kan även köra verktyget på din egen dator om du vill. Instruktioner för detta finns på kurshemsidan (välj ”Visualiseringsverktyg” i vänstermenyn).

När du har startat verktyget kan du välja *File* → *Open...* och sedan välja källkodsfilen för labben. Koden kompileras automatiskt, så du behöver inte göra det manuellt. Du kan sedan välja i vilken ordning du vill att de olika trådarna ska köras. På så vis kan du se vad som händer i olika fall. Målet är att din implementation ska fungera korrekt oavsett i vilken ordning du kör trådarna. Verktyget kan också automatiskt testa alla

möjliga kombinationer åt dig om du väljer *Run* → *Look for errors*. Detta ger en indikation på om din lösning är korrekt eller inte.

Om du har gjort ändringar i källkoden kan du välja *File* → *Reload* (Ctrl+Shift+R) för att ladda in programmet igen.

A.2 I terminalen

Om du vill lösa labben med vanliga utvecklingsverktyg går även detta. Gå till mappen `src/standalone/lab4a` och skriv `make data-files` för att kompilera programmet. Detta skapar filen `exec-wait` som kan köras med `./data-files`. Du kan självklart använda GDB som vanligt: `gdb ./data-files` eller `gdb -tui ./data-files`.

A.3 Tips för implementation och testning

Som ofta är fallet är det svårt att testa ifall kod är korrekt synkroniserad. I allmänhet kräver detta att man noga funderar över vilken data som kan vara delad mellan olika trådar, och vilken kod som måste synkroniseras för att lösa de problem som kan uppkomma. I det här problemet görs detta lämpligtvis med hjälp av lås. Ett bra tillvägagångssätt är då att tänka att varje lås skyddar någon eller några specifika variabler. Sedan kan man titta igenom sitt program och se till så att tråden håller låset när den använder skyddad data.

I det här fallet kan visualiseringsverktyget hjälpa till att kontrollera att din implementation är fri från synkroniseringsproblem så som den används i testprogrammet. Alternativet *Run* → *Look for errors* ger en bra indikation på detta. Utöver det vill vi att implementationen ska garantera att det aldrig finns fler än två instanser av `struct data_file` samtidigt, och att alla instanser frigörs i slutet av programmet. Dessutom vill vi att så stora delar av programmet som möjligt ska kunna köras parallellt (dvs. vi vill inte ha ett stort lås kring all kod eftersom det är onödigt restriktivt). Visualiseringsverktyget kan inte automatiskt kontrollera om de tre sistnämnda egenskaperna är uppfyllda, utan där behöver du själv fundera på om implementationen är korrekt (eller fråga assistenten). Att inte ha några synkroniseringsproblem som upptäcks av verktyget är dock en mycket bra början!

Det är en bra idé att diskutera din lösning med assistenten när du känner dig klar. På så vis kan du undvika problem i nästa laboration.

Del B Synkronisering (99% förarbete, 1% modifiering)

Från systemanropen använder du egna datastrukturer (arrayer, listor, etc.) och anropar flera delsystem som är helt eller delvis osynkroniserade. Om processerna/trådarna råkar exekevera i ”fel” ordning, med trådbyten på ”fel” ställen så kommer implementationen att gå sönder.

Målet med den här delen är att lösa dessa problem i systemet. De delar du behöver fundera över är filsystemet (se `src/filesys/`), din processlista, din lista över fildeskriptorer och eventuella andra datastrukturer du har lagt till. Du kommer garanterat behöva synkronisera delar av filsystemet. De datastrukturer som du själv har lagt till kan också behöva synkroniseras, men exakt hur och var beror på din implementation i tidigare laborationer.

Information om hur filsystemet är uppbyggt och hur det fungerar finns under rubriken ”Filsystem” i Pintos-Wiki. Fokusera särskilt på rubriken ”Struktur” för att enklare avgöra vilka filer som är intressanta att börja undersöka. Till din hjälp har du sedan frågeställningarna nedan. De pekar på problem som kan uppstå i filsystemet innan det är synkroniserat. Frågeställningarna pekar dig till alla delar av filsystemet som behöver behandlas, men de är inte kompletta i bemärkelsen att de visar alla möjliga saker som kan gå fel (då skulle labbhandledningen bli väldigt lång...).

B.1 Vad behöver synkroniseras?

Grundregeln är att *all data som är delad mellan trådar* måste synkroniseras. Titta därför extra noga på *variabler som kan nås av flera trådar*. Det vill säga, antingen *globala variabler* eller variabler som flera trådar kan ha en *pekare* till. Identifiera sedan kritiska sektioner kring den delade datan och se till att *mutual exclusion* upprätthålls med hjälp av lås (eller semafor om så krävs).

För att identifiera delad data, och de kritiska sektionerna som finns, så behövs en god förståelse om hur koden fungerar och vad den försöker åstadkomma. Det finns alltså inga ”enkla” regler som fungerar i allmänhet. Man måste helt enkelt titta på koden, fundera på vad som kan gå fel om flera trådar gör saker samtidigt, och sedan på lämpligt sätt förhindra problemen som kan uppstå. Frågeställningarna nedan är en bra utgångspunkt för detta.

B.2 Var ska man synkronisera?

En till synes enkel lösning på problemet skulle vara att lägga all synkronisering direkt i hanteringen av systemanrop. Detta är dock inte en bra lösning av två anledningar: först och främst använder andra delar av Pintos filsystemet, så att bara synkronisera i systemanropen kommer inte garantera att endast en tråd kör koden i filsystemet, vilket i sin tur inte löser problemen. Den andra stora anledningen är att om vi har ett (eller ett fåtal) lås i systemanropen så kommer bara ett systemanrop kunna köras åt gången, även om en process skriver data till skärmen och en annan försöker starta en ny process. Detta är två orelaterade saker som utan problem kan göras parallellt, och vi vill därmed tillåta detta. Försök därför att lägga synkroniseringen så nära de resurser som är delade som möjligt. Försök också i den mån det är möjligt att lägga synkroniseringen i de funktioner som finns, så att det inte går att glömma att låsa något innan man anropar en viss funktion.

B.3 Frågeställningar

1. Katalogen är tom. Två processer lägger till filen ”kim.txt” samtidigt. Är det efteråt garanterat att katalogen innehåller endast en fil ”kim.txt”?
2. Katalogen innehåller en fil ”kim.txt”. Två processer tar bort ”kim.txt”, och en process lägger samtidigt till ”kam.txt”. Är det efteråt garanterat att katalogen innehåller endast fil ”kam.txt”?
3. Systemets globala *inode*-lista är tom. Tre processer öppnar samtidigt filen ”kim.txt”. Är det garanterat att *inode*-listan sedan innehåller endast en cachad referens till filen, med `open_cnt` lika med 3?

4. Systemets globala *inode*-lista innehåller en referens till ”kim.txt” med `open_cnt` lika med 1. En process stänger filen samtidigt som en annan process öppnar filen. Är det garanterat att *inode*-listan efteråt innehåller samma information?
5. Free-map innehåller två sekvenser med 5 lediga block. Två processer skapar samtidigt två filer som behöver 5 lediga block. Är det efteråt garanterat att filerna har fått var sin sekvens lediga block?
6. Katalogen innehåller en fil ”kim.txt”. Systemets globala *inode*-lista innehåller en referens till samma fil med `open_cnt` lika med 1. Free-map har 5 block markerade som upptagna. En process tar bort filen ”kim.txt” samtidigt som en annan process stänger filen ”kim.txt”. Är det efteråt garanterat att *inode*-listan är tom, att free-map har 5 nya lediga block, och att katalogen är tom?
7. Katalogen innehåller en fil ”kim.txt”. En process försöker öppna filen samtidigt som en annan process tar bort filen ”kim.txt” och skapar sedan en ny fil ”kam.txt”. Är det efteråt garanterat att den första processen antingen lyckades öppna filen ”kim.txt”, eller att den misslyckades? Eller kan det bli så att den råkar öppna ”kam.txt” i stället?
8. Liknande frågor skall du själv ställa dig i relation till din process-lista och till din(a) fil-list(or).

B.4 Testprogram

Filen `src/examples/pfs.c` innehåller ett testprogram som försöker stresstesta filsystemet. Det kräver att du synkroniserar funktionerna `inode_read_at` och `inode_write_at` med ett lås som du deklarerar i `struct inode` (om du inte redan har gjort det). I nästa del kommer vi att ersätta detta lås med en mer avancerad variant som tillåter mer parallellism.

Programmet fungerar på följande sätt: Programmet `pfs` skapar filerna `file.1` och `messages` i filsystemet. Sedan startar det två processer som kör `pfs_writer` och tre processer som kör `pfs_reader`. Sedan väntar det på att alla programmen kört klart (med systemanropet `wait`).

Programmet `pfs_writer` öppnar filen `file.1` och fyller hela filen med en bokstav. Detta upprepas för varje bokstav, och varje bokstav används 200 gånger.

Programmet `pfs_reader` öppnar filen `file.1` och kontrollerar med jämna mellanrum att ett anrop till `write` bara innehåller en bokstav. Det vill säga att den aldrig ser ett halvklart anrop till `write` från en `pfs_writer`-process. Om allt är bra skrivs raden `cool` till filen `messages`. Annars skrivs `INCONSISTENCY` ut både till filen och till skärmen.

Programmet kan köras med följande kommandorad:

```
pintos -v -k -T 240 --fs-disk=2 \
       -p ../examples/pfs -a pfs \
       -p ../examples/pfs_writer -a pfs_writer \
       -p ../examples/pfs_reader -a pfs_reader \
       -g messages -- -f -q -S -F=10000 run pfs
```

Här används flaggan `-g` för att hämta filen `messages` från Pintos hårddisk till ditt filsystem (givet att Pintos inte kraschade, såklart). På så sätt kan du inspektera filen `messages` och räkna antalet `cool` som finns i den:

```
grep -c cool messages
```

Du får själv fundera på hur många `cool` filen borde innehålla (600, 400, 200, eller något annat). Kontrollera också att filen inte innehåller något annat än `cool`:

```
grep -v '^cool$' messages
```

Om synkroniseringen inte fungerar som det ska kan det finnas annat i filen, eller för många `cool` i filen (hur kan det komma sig?).

Flaggan `-S` i kommandoraden ovan aktiverar ett system som gör att alla systemanrop körs långsamt för att öka sannolikheten att trådbyten sker i den känsliga koden. Detta gör dock att testet tar längre tid att köra. Flaggan `-F=10000` ökar hur ofta trådbyten sker (ökar frekvensen av timer-interrupt), vilket också ökar sannolikheten att något dåligt sker.

Dessa flaggor gör med stor sannolikhet att du kommer se antingen *INCONSISTENCY* eller något annat felmeddelande när `pfs` körs. Tänk dock på att det inte går att testa ifall multitrådade program är korrekta. Det innebär att även om testet inte rapporterar något fel så kan det fortfarande finnas synkroniseringsproblem i din kod. Du behöver därför också fundera noga på hur du har synkroniserat, och övertyga dig själv om att något dåligt inte kan hända, oavsett i vilken ordning trådarna exekeveras.

Tips: Testa även att lägga till flaggan `-L` (som i *leak check*) innan ordet `run` i kommandoraden. Detta gör att systemet kommer att visa dig minnesläckor i slutet av körningen. En korrekt implementation bör inte ha minnesläckor i det här skedet. Men, som beskrivs i avsnittet ”Felsökning” i Pintos-Wiki, finns det tillfällen då minnesläckor kan visas i acceptabla lösningar.

Del C Readers-writers lock

Att läsa och skriva filer på disk är en mycket vanlig operation som dessutom ofta tar mycket tid (disken är långsam i förhållande till resten av systemet). Målet med synkroniseringen i förra uppgiften var att allt skulle fungera korrekt, vilket i princip innebär att alla systemanrop ska se ut som de är *atomära*, dvs. ingen process ska kunna se ett systemanrop som är halvklart.

Antag exempelvis att vi har följande två processer och en fil i filsystemet som heter `test` med innehållet `aaaa`:

```
int main() {
  int fd = open("test");
  write(fd, "bbbb", 4);
  close(fd);
}
```

```
int main() {
  char buffer[4];
  int fd = open("test");
  read(fd, buffer, 4);
  close(fd);
  write(STDOUT_FILENO, buffer, 4);
}
```

I det här fallet ska den andra processen antingen se att filen innehåller `aaaa`, eller `bbbb`, men aldrig något mellanting. Exempelvis ska den aldrig kunna se `bbaa` eller någon annan variant.

Detta kan vi enkelt lösa med ett lås, men det är överdrivet restriktivt. Det är ju inga problem att låta flera processer läsa data så länge ingen process skriver! I och med att disken är långsam är det i det här fallet ännu viktigare att vi låter flera processer läsa innehållet när det är möjligt att göra på ett säkert sätt.

Vi vill alltså ersätta låset i `inode_read_at` och `inode_write_at` med ett så kallat *readers- writers lock* för att tillåta att flera processer läser från samma fil, men fortfarande förhindra att någon skriver medan andra läser, eller att flera processer skriver samtidigt.

Information om detta finns under rubriken ”Readers- writers lock” i Pintos-Wiki.

Din lösning får innehålla *starvation*. Du får däremot inte begränsa antalet processer som väntar på sin tur. Om någon process vill läsa eller skriva så ska den alltid kunna få en köplats om den inte direkt kan få tillgång till filen.

Tänk noga på var du placerar dina synkroniseringsvariabler. Använd den kunskap du har om filsystemets implementation som du skaffade dig i tidigare uppgift. Du måste se till att en fil som används samtidigt från två olika processer faktiskt också använder samma synkroniseringsvariabler, medan andra filer som används samtidigt använder andra synkroniseringsvariabler.

C.1 Testa utanför Pintos

Om du vill kan du testa din lösning utanför Pintos. Filen `src/standalone/lab4c/rwlock.c` innehåller en minimal implementation av `inode`-datatypen så att det går att testa implementationen av reader- writer låset i visualiseringsverktyget. I skelettet representeras fildatan av en filvariabel för att förenkla implementationen. Detta räcker dock för att simulera läsning och skrivning till fildatan.

Skelettet är tänkt att användas i visualiseringsverktyget (se instruktioner under del A i detta dokument), men det går också att kompilera i terminalen med `make rwlock`. Skelettet är däremot inte designat för att vara enkelt att testa i terminalen. Om du använder funktionen *Run* → *Look for errors* är det en bra idé att ändra så att looparna i `reader_thread` och `writer_thread` bara kör 2 iterationer i stället för 10. Detta gör att verifieringen går mycket snabbare.

C.2 Testprogram

Programmet `src/examples/pfs.c` som är beskrivet ovan (för del B) är även lämpligt för denna del av laborationen.