LINKÖPINGS UNIVERSITET

TDIU16: Process- och operativsystemprogrammering

Readers-writers lock

Klas Arvidsson, Daniel Thorén, Filip Strömbäck

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2018

2020-03-06

1 Mål

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 vill ska den andra processen antingen ska 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.

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

2 Uppgift

Implementera synkronisering av läsning och skrivning av filer så att inga fel kan uppstå. En bra början är att se till ingen kan läsa eller skriva förrän tidigare läsningar eller skrivningar slutförts helt och hållet. Detta är dessutom enkelt att implementera.

Utöka sedan din lösning så att *flera kan läsa samtidigt*, d.v.s. innan andra som läser är helt klara. Det får fortfarande inte vara möjligt att någon skriver samtidigt som någon annan läser eller skriver. D.v.s senaste skrivna data i filen skall fortfarande synas i sin helhet, aldrig delvis. I din lösning får *starvation* vara möjligt. Du får däremot inte begränsa antalet processer som väntar på sin tur, vill någon process läsa eller skriva skall den alltid få en ”köplats” om den inte kan få direkt tillgång till filen. Antalet processer som får läsa samtidigt är obegränsat. Skillnaden från enkel synkronisering är att endast en av de som läser samtidigt behöver markera att filen är upptagen (den markeringen gäller gemensamt för samtliga som läser just då). Frågan är bara vem som skall markera att filen är upptagen, och vem som markerar att den är ledig igen? Och hur skall du hålla reda på det?

Tänk noga på var du placerar dina synkroniseringsvariabler. Använd den kunskap du 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.

3 Valfritt: Testa din implementation utanför Pintos

Om du vill experimentera utanför Pintos så finns ett minimalt skelett i filen `src/standalone/lab13/rwlock.c`. Detta är helt fristående från Pintos och innehåller en mycket minimal implementation av `inode`-datatypen. Datatypen i skelettet innehåller en heltalsvariabel som representerar läsning och skrivning till disk i stället för den riktiga logiken som finns i Pintos.

Skelettet är tänkt för att köra i visualiseringsverktyget som finns på kurshemsidan. Med hjälp av det kan ni se hur er implementation beter sig i olika scenarier. Det går också att kompilera och köra koden utan verktyget med hjälp av kommandot `make`. Däremot är det inte enkelt att testa implementationen som den är just nu. I så fall behöver du utöka det huvudprogram som finns där, eller stega i koden med GDB.

Notera 2021: Skelettet fanns sannolikt inte när ni klonade ert repository, så ni kan antingen köra `git pull origin master` i ert repo, eller hämta filerna från `https://gitlab.liu.se/tdiu16-material/pintos/-/tree/master/src/standalone/lab13`

4 Testa din implementation i Pintos

Ett speciellt testprogram finns i examples-katalogen. Det körs på följande sätt:

```
pintos -v -k -T 120 --fs-disk=2 --qemu \
       -p ../examples/pfs -a pfs \
       -p ../examples/pfs_writer -a pfs_writer \
       -p ../examples/pfs_reader -a pfs_reader \
       -g messages -- -f -q -F=2000 run pfs
```

Programmet `pfs_reader` är det som kontrollerar att senast skrivna ”papper” alltid syns i sin helhet. Om det är så skrivs `cool` i filen `messages`. Annars skrivs `INCONSISTENCY`. **Tre** `pfs_reader`-processer startas, och **varje** `pfs_reader` utför kontrollen **500** gånger (samtidigt som `pfs_writer` skriver till filen). När allt är klart skall du kontrollera hur många `cool` som skrivits till filen:

```
grep -c cool messages
```

Du får själv fundera på vilken siffra som är rätt. 1500 eller 500? Hur kan det komma sig?

Du skall även kontrollera att resultatet inte innehåller något annat än cool:

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

Finns det något annat fungerar inte din synkronisering perfekt. Då kan det även finnas lite för många `cool` i filen. Hur kan det komma sig att det blir för många `cool`?

Tips: Hur håller filen reda på aktuell läs- och skrivposition? I vilken datastruktur lagras det?