Deluppgift 20 Synkronisering (99% kodförståelse, 1% modifiering)

Saker som kan vara bra att känna till

PintOS filsystem är uppbyggt av flera olika programmoduler.

En översiksbild av vilka funktioner som finns i varje modul och hur de anropar varandra finns på `www.ida.liu.se/~TDDI04/filesys.png`. Vilka globala variabler som finns står också. För att läsa och bläddra i koden bör du använda `emacs` med `M-.` och `M-*` för att automatiskt kunna följa funktionsanrop i koden. Hur du genererar en `TAGS`-fil beskrevs i uppgift 9.

`disk.c, disk.h`
: Här finns funktioner för att läsa och skriva sektorer från disken. Läsning och skrivning till enskilda sektorer från och till disken ***är synkroniserad***. Denna synkronisering hindrar att en tråd använder disken innan föregående tråd är helt klar med läsning eller skrivning till disk.

`free_map.c, free_map.h`
: Håller reda på vilka sektorer (eller block) på disken som är upptagna och vilka som är lediga. Informationen sparas i en bitmap som hålls i minnet. Assemblerinstruktioner `or` och `and` används föra att markera respektive avmarkera en bit. Dessa är atomära på bitnivå, ***men inte på högre nivå***. På disk lagras denna som en `inode`. Fungerar det om två trådar samtidigt allokerar filer med flera block i varje?

`inode.c, inode.h`
: Definierar informationen för att hålla reda på en `inode` på disken och i cachat minnet. En `inode` är en sekvens byte som är lagrat på disk. Inoder representerar både filer och kataloger. Data som PintOS lagrar på disk för varje `ionde` är sektor på disk och storlek. I minnet lagras dessutom hur många processer som använder filen (har den öppen) och om den skall tas bort när den stängs. Ytterligare en flagga som lagras håller reda på om filen är read-only medan den är öppen (t.ex. körbara programfiler). De inoder som cachas i minnet lagras i en global lista. ***Ingen del av koden är synkroniserad.*** Denna kod är en mycket central del av Pintos filsystem och används av nästan alla moduler i denna lista.

`directory.c, directory.h`
: Innehåller funktioner för att öppna och söka efter filer i kataloger. Pintos kan bara ha en katalog med max 16 filer i nuvarande version. Denna modul använder endast funktioner i `inode`-modulen för att utföra sina uppgifter. ***Den är inte synkroniserad.*** Fungerar det om två trådar lägger till filer samtidigt?

`filesys.c, filesys.h`
: Detta är det yttersta API:et för att skapa filer, öppna filer, och ta bort filer. Internt används huvudsakligen `directory`-modulen, och `free_map`-modulen, men även enstaka funktioner från `file`-modulen (`file_open`) och `inode`-modulen (`inode_init` och `inode_create`).

`file.c, file.h`
: Detta är det yttersta API:et för att hantera `struct file*` som erhålls från `filesys_open`. Dessa funktioner använder internt endast `inode`-modulen.

Några frågeställningar

1) Katalogen är tom. Två processer lägger till filen “kalle.txt” samtidigt. Är det efteråt garanterat att katalogen innehåller endast en fil “kalle.txt”?
2) Katalogen innehåller en fil “kalle.txt”. Två processer tar bort “kalle.txt”, och en process lägger samtidigt till “nisse.txt”. Är det efteråt garanterat att katalogen innehåller endast fil “nisse.txt”?
3) Systemets globala `inode`-lista är tom. Tre processer öppnar samtidigt filen “kalle.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 “kalle.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 “kalle.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 “kalle.txt” samtidigt som en annan process stänger filen “kalle.txt”. Är det efteråt garanterat att `inode`-listan är tom, att free-map har 5 nya lediga block, och att katalogen är tom?
7) Liknande frågor skall du själv ställa dig i relation till din process-lista och till din(a) fil-list(or).

Uppgift

Från dina systemanrop använder du egna datastrukturer (arrayer och/eller listor), och anropar flera delsystem, som är helt eller delvis osynkroniserade. Om processerna/trådarna råkar exekvera i “fel” ordning, med trådbyten på “fel” ställen kommer din implementering att gå sönder.

Gå igenom frågeställningarna ovan och se vilken kod som kommer att exekveras i de olika fallen. Notera vilka funktioner som används och vilka variabler och datastrukturer som kommer att användas samtidigt. ***Synkronisera sedan koden för filsystemet.*** Du kan utifrån beskrivningen ovan avgöra vilka filer som är intressanta. Det är viktigt du tänker efter först för att få så enkel lösning som möjligt. Det finns flera godkända lösningar av varierande kvalité.

Synkronisera även alla egna datastrukturer och funktioner där så krävs.

I denna uppgift ska du titta speciellt noga på *variabler som kan nås av flera trådar* samtidigt, d.v.s. antingen *globala eller pekare* som delas mellan flera trådar. Identifiera kritiska sektioner och synkronisera med lås (eller semafor om så krävs). Conditions behöver du bara om du vill vänta på att en synkroniserad (låst) variabel skall uppfylla ett visst villkor. Tänk på att globalt placerade lås skapar onödiga köer för alla trådar som använder dem. Om det är möjligt, använd ett lås per resurs som delas.

Synkronisering av läsning och skrivning till filer skall du lämna till nästa uppgift, alternativt tillfälligt använda ett enkelt lås i `inode_read_at` och `inode_write_at`. Ett enkelt lås är hursomhelst en bra start på nästa uppgift.

Testa din implementering

Efter denna laboration skall alla inbyggda tester fungera alternativt inte fungera konsekvent vid alla körningar. Antingen fungerar ett visst test alltid, eller så fungerar det aldrig. Om något fungerar bara ibland är synkroniseringen fel. Det krävs att alla testerna körs igenom många gånger. Se uppgift 18 om du glömt hur du startar testningen.

I `examples/pfs.c` finns ytterligare ett testprogram. Det skall fungera att köra utan att det kraschar, men skall skriva ut felmeddelanden i form av ordet `INCONSISTENCY`. Du startar testet med följande kommandorad:

```
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 run pfs
```

Om inga `INCONSISTENCY` skrivs ut har du antagligen varit lite för duktig på att synkronisera koden (bra!). I senare uppgift skall du lösa problemet med inconsistency med *ett speciellt lås* som tillåter att filer läses samtidigt. Se till att filer kan skrivas och läsas samtidigt så du ser att inconsistency uppstår och därmed kan se när nästa uppgift är löst (ingen inconsistency uppstår).