LINKÖPINGS UNIVERSITET

TDIU16: Process- och operativsystemprogrammering

Pintos Wiki

Klas Arvidsson, Daniel Thorén, Filip Strömbäck

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2023

2023-03-16

Innehåll

1 Introduktion

Det första du behöver veta om Pintos är att det är ett riktigt operativsystem, om än med mycket begränsad funktionalitet. Det är inte ett lab-skelett i vanlig mening (även om vi då och då försöker förtydliga kod och kommentarer). Varje rad i Pintos har ett funktionellt syfte även i versionen du startar med, och alla variabler används till något.

Ofta får vi, när vi ber studenter motivera en lösning, kommentaren att ”det såg ut som det var förberett för vår kod här”. När vi sedan frågar hur de funktioner studenterna modifierat och de variabler de återanvänt används av Pintos har de ingen aning. De har alltså lagt ned massor av tid och jobb på en lösning som med all sannolikhet saboterar den befintliga koden så att något annat går sönder. Vilket kostar dem ännu mer tid i felsökning. Det går inte att ta en snabb titt på koden och dra slutsatsen ”det är nog här vi skall lägga till vår kod”. Ta istället en längre titt på koden, ta reda på hur den fungerar och används, och utnyttja den sedan på rätt sätt till din lösning. Det finns en specifik lösning som det är tänkt att ni skall kunna komma på och implementera inom rimlig tid, men tänk på att det finns många andra lösningar som fungerar, som är rätt, och som alltså godkänns. Smarta genomtänkta lösningar välkomnas. De använder ofta lite mer avancerad kod, men i gengäld mindre mängd kod. Spendera hellre 4h att planera en lösning och 1h att implementera den än tvärtom, det lönar sig.

Meningen med laborationerna är att du skall komplettera Pintos med kod som gör att en uppsättning vanliga systemanrop fungerar, samt korrigera en del kod som är avsiktligt dåligt implementerad (den implementation som finns från start är på vissa punkter endast till för att det skall gå att alls starta Pintos). Den kod du skriver kommer till större delen att vara rena tillägg till den befintliga koden, och måste samarbeta med resten av systemet. För att din kod skall samarbeta bra med resten av systemet krävs att du förstår hur de centrala modulerna i Pintos fungerar. Dessa finns beskrivna i Stanfords originaldokumentationen till Pintos, Appendix A, Reference Guide. Appendix A är rekommenderad läsning. Även Introduction, Project 1 och Project 2 innehåller bitar du kan ha nytta av. De viktigaste funktionerna finns även beskrivna i detta dokument.

Genom åren har vi naturligtvis haft många studenter som ignorerat allt vad dokumentation heter och gissat sig fram till en lösning. Gemensamt för dem är att de fått lägga ned mycket tid på buggar och felsökning eller ännu inte är klara. Om du använder befintliga funktioner fel, eller inte försår hur t.ex. trådsystemet fungerar, blir det ofta jobbiga, tidskrävande fel. Gissa inte.

Pintos är skrivet modulärt. Varje fil innehåller funktioner som implementerar ett delsystem, och koden är relativt välkommenterad. Ytterst är det kommentarerna och koden som är den fullständigt korrekta dokumentationen av hur det verkligen fungerar. Bläddra i koden för att hitta de funktioner som du behöver för att lösa uppgifterna, och läs på hur den kod du tänker dig att använda fungerar. Målsättningen är att allt du behöver skall finnas beskrivet i denna instruktion till slut, men dit har vi ännu inte kommit. Du kan behöva komplettera med att läsa i Stanfords dokumentation, eller direkt i koden. Är något oklart får du gärna höra av dig till oss så att vi kan förbättra detta dokument.

1.1 Tillåtna och otillåtna lösningar

Som vanligt när du löser laborationer finns det lösningar som inte godkänns. Det är självklart lösningar som kan leda till fel resultat, lösningar som är gravt ineffektiva eller klumpiga, lösningar som läcker minne, samt lösningar som på ett eller annat sätt inte uppfyller god programmeringssed. I Pintos tillåter vi dock att du gör en del lösningar som inte skulle vara godtagbara i ett riktigt system. Anledningen till detta är att det är första gången du kodar ett operativsystem, du är ovan att programmera i C, samt att vi har begränsat med tid. Därför håller vi oss till enklare lösningar:

- Begränsad storlek på datastrukturer: Du behöver hålla reda på öppna filer och startade processer m.m., det är då OK att sätta en övre gräns på hur många, vilket gör att enkla arrayer kan användas. Det skall dock vara lätt att öka eller minska begränsningen vid omkompilering.

- Globala variabler: Operativsystemet kontrollerar systemets globala resurser, och måste ofta använda globala datastruktuer för att hålla reda på dem. Du får alltså lägga datastrukturer globalt, men du måste kunna motivera det, och det måste fungera med godtyckligt antal aktiva trådar. Det är mycket viktigt att initiera all data till kända värden, inte minst för att kunna felsöka.

Några ”nya” saker som inte är acceptabla:

- Stänga av interrupts är inte tillåtet: Du skall lösa synkronisering med hjälp av lås och semaforer. De tar i sin tur hand om att stänga av och sätta på interrupts på rätt sätt. Det är bland annat därför dessa abstraktioner finns. Den enda gången lås inte kan användas är då synkronisering skall ske mellan en tråd och ett externt (asynkront) interrupt. Du kommer inte råka ut för någon sådan situation, men du kan ju fundera på varför lås inte fungerar i det fallet.
- Busy-wait är inte tillåtet. Då en tråd behöver vänta på att något villkor skall uppfyllas skall detta lösas genom att tråden placeras på en vänte-kö och låter andra trådar exekveras under tiden. Semaforer och Conditions existerar för att utföra detta på rätt sätt. Använd dem.
- Synkroniseringsproblem och ”races” är inte tillåtna. Om det går att konstruera två exekveringssekvenser (via trådbyten), oavsett hur långsökta, där samma uppgift under samma initiala förutsättningar kan leda till två olika resultat så är koden felaktig.
- Minnesläckor är absolut förbjudna. Använder du *malloc* eller *palloc_get_page* för att reservera minne är det absolut nödvändigt att du använder *free* resp. *palloc_free_page* när du inte behöver minnet mera. Läcker en process minne kan (skall!) operativsystemet ta tillbaka minnet när processen avslutar. Men om operativsystemet läcker minne slutar det med att datorn måste startas om dagligen eller oftare för att få tillbaka minnet. Det är inte acceptabelt.

Känner du dig osäker på om något är acceptabelt eller inte, rådgör med assistent i *förväg*.

Oavsett vilken lösning du väljer skall slutresultatet vara att alla systemanrop fungerar stabilt under en längre tids körning. Det som räknas från kursens sida är inte bara att eventuella test fungerar, utan även att koden är skriven på ett tydligt och korrekt sätt. Testfallen kan fungera trots gravt felaktig kod. Korrekt kod kommer alltid klara alla testfall. Du skall lösa problemet generellt, skriva kod för generella fallet, och sedan testa genom stickprov. Om du börjar med att titta på testfallen och sedan skriver kod som löser just testfallen (stickproven) blir det nästan säkert så att din kod inte löst hela problemet.

1.2 Missuppfattningar och misstag vid kodning i C

Många av de problem föregående års studenter råkat ut för härstammar från misstag och bristande förståelse av programspråket C. Det vanligaste tankesättet är ”det kompilerar, alltså är det rätt” eller ”det fungerar, alltså är det rätt”. Tyvärr är inget av detta sant, och många gånger blir den upptäckten som en kniv i ryggen efter många timmars felsökning.

1.2.1 Missuppfattningen ”det kompilerar, alltså är det rätt”

C är ett språk som tillåter dig som programmerare att göra i princip vilka knasigheter som helst utan att generera fel vid kompilering. Om du som programmerare skrivit koden implicit (t.ex. en implicit pekaromvandling) så genererar kompilatorn en varning (om du har tur), men det är i regel allt du kan hoppas på. Skriver du koden explicit, t.ex. med explicita (kanske felaktiga) typomvandlingar får du inte ens en varning då kompilatorn utgår från att du vet vad du gör. Du är **GUD** över koden och därmed kompilatorn. Du bestämmer vad som är rätt. Är du minsta osäker på någon detalj gäller det alltså att kontrollera hur det skall vara. Bättre spendera en timme med att ta reda på hur det skall vara, än att senare spendera åtta timmar med att felsöka konstiga fel. Det gäller även att vara mycket uppmärksam på kompileringsvarningar. Ofta är de oviktiga, men lika ofta signalerar de direkta felaktigeter. Grundregeln är att om du inte förstår

varför en varning uppstår skall du ta reda på det och rätta koden. Rekommendationen är att koda bort alla varningar. Annars är det lätt hänt att missa en viktig varning bland en lång lista på oviktiga.

1.2.2 Missuppfattningen ”det fungerar, alltså är det rätt”

Att något fungerar betyder faktiskt bara att det fungerar för precis det du testat. Så om du inte testar **ALLA** (vilket som regel är omöjligt) möjliga trådbytessekvenser och indatasekvenser kan du inte veta om något är rätt genom att bara testa.

När du skriver vanliga program fungerar operativsystemet i viss mån som en sandlåda, men tillåter fortfarande att programmet gör fel internt. Du har inte tillgång till privilegierade instruktioner. Du kan inte läsa och skriva var du vill i minnet. Om du gör grova fel med någon pekare får du ”segmentation fault”. Men gör du bara ”små” (men allvarliga) fel, t.ex. indexerar utanför en array eller använder avallokerat minne så kommer programmet verka fungera tills minnet du använt fel (t.ex. precis efter arrayen) används till det som det är avsett för. Symtomen på att det är fel kan alltså visa sig långt senare trots att det ser ut fungera till en början.

I Pintos skriver du operativsystemet. Där har du tillgång till allt (existerande) minne. Skriver du till ”fel” adress är alltså chansen större att allt kommer verka fungera. Tills det som den adressen egentligen används till behövs, vilket sker några dagar senare när du börjar testa andra saker. Om du inte helt förstår hur koden du skriver samarbetar med resten av Pintos är risken att allt till en början verkar fungera. Men ett par dagar senare, när du skrivit flera dussin nya rader och startar ett nytt testfall, då krashar Pintos. Och inte på grund av de två dussin nya rader du just skrev, utan på grund av det du skev i förrgår och trodde var rätt. Du behöver alltså felsöka förutsättingslöst, eller åtminstone verifiera att allt du förutsätter verkligen är sant.

2 Installation

Denna sektion går igenom hur du installerar och kör pintos samt lite information om hur du navigerar källkoden med emacs.

2.1 Förberedelser för installation på egen dator (Ej Linux)

Installera först VirtualBox, sedan en virtuell maskin (i VirtualBox) med Ubuntu Mate. Därpå installerar du enligt ”Intallation av Pintos”. Se separata instruktioner på kurshemsidan för hur du installerar VirtualBox och installerar Ubuntu Mate i den. Använd alternativt ThinLinc, dock finns det risk för prestandaproblem då många använder ThinLinc samtidigt.

2.2 Förberedelser för installation på egen dator (Linux)

I och med att Linux-system ser lite olika ut, kan du behöva anpassa kommandona lite för att fungera på ditt system. Detta fungerar på Ubuntu/Debian-system:

```
sudo apt install gcc gcc-multilib qemu git
```

Felsökning: Om du får ett felmeddelande som säger något i stil med ”qemu not found” beror det antagligen på att din distribution (exempelvis senaste Ubuntu) har delat in qemu i flera bitar. Ersätt då `qemu` med `qemu-system-x86` i kommandot ovan.

Felsökning: Om du får problem efter att ha uppdaterat QEMU kan det vara så att två olika versioner är installerade. Om så är fallet får du meddelanden i stil med följande när du försöker köra Pintos:

```
Failed to initialize module: /usr/bin/...
Note: only modules from the same build can be loaded.
```

Detta kan lösas genom att avinstallera QEMU med `sudo apt remove qemu` följt av `sudo apt autoremove` för att ta bort ytterligare paket som används av QEMU. Installera sedan QEMU igen med sudo apt install qemu.

2.3 Installation av Pintos

Den här sektionen förutsätter att du arbetar i en Linuxmiljö. Om du inte gör det, följ först för dig relevanta anvisningar ”förberedelser för installation” ovan.

Öppna ett terminalfönster och gå igenom följande punkter. Om du har en labpartner gör ni detta tillsammans på ett konto om det inte står annat. **OBS!** LIU-ID i detta avsnitt avser alltid **ditt** LiU-ID.

Vi börjar med att hämta källkoden för Pintos genom att klona kursens repository:

```
git clone https://gitlab.liu.se/tdiu16-material/pintos.git ~/tdiu16-labs
echo "export PATH=\$PATH:$HOME/tdiu16-labs/src/utils" >> ~/.bashrc
source ~/.bashrc
```

Nu är du egentligen redo för avsnittet ”Kompilera och kör Pintos”. För kunna samarbeta med din labpartner och kunna dela kod med kurspersonalen behöver du dock skapa ett eget privat projekt på gitlab.liu.se. Gå till `https://gitlab.liu.se/projects/new` och fyll i `tdiu16-labs`. Kom också ihåg att klicka ur rutan *Initialize repository with a README*, annars får du problem senare. Notera URL-en till ditt projekt på den sida du skickas till efter att du skapat projektet. URL-en ser ut så här om du följt instruktionerna exakt: `git@gitlab.liu.se:LiU-ID/tdiu16-labs.git`

Om du har skapat en ssh-nyckel sedan tidigare användning av `gitlab.liu.se` på ditt konto kan du gå vidare till nästa steg, annars finns instruktioner för att skapa en nyckel på `https://gitlab.liu.se/help/user/ssh.md`. Om din labpartner ska ha tillgång till ditt repository behöver hen också en nyckel på sitt konto. När du skapar din nyckel i terminalen ställs en rad frågor. Standardsvaret på alla frågor är bra nog i normalfallet, så tryck bara enter.

Leta upp sidan för att administrera medlemmar i `tdiu16-labs`-projektet och lägg till kurspersonalen (år 2024: filst04, klaar36, dagjo87, tobel81, malni83, matbu80, matka12) med minst rättighetsnivå `Reporter`. Troligen vill du lägga till din labpartner med som minst `Developer` också.

Nu är det dags att gå tillbaka till en terminal för att ställa in att dina ändringar av källkoden i fortsättningen ska hamna i ditt repository (annars kommer `git push` försöka använda kursens repository, och det går inte):

```
cd ~/tdiu16-labs
git remote add student git@gitlab.liu.se:LIU-ID/tdiu16-labs.git
git push -u student master
```

(Det går lägga till flera remotes, t.ex. om du och din labpartner vill ha var sitt repository på gitlab. Men då får ni själva se till att lära er var er kod tar vägen när ni pushar.)

Nu är det dags för din labpartner att installera Pintos på sitt konto. Det gör hen enklast genom att öppna en terminal och klona ditt repository:

```
git clone git@gitlab.liu.se:LIU-ID/tdiu16-labs.git ~/tdiu16-labs
cd ~/tdiu16-labs
echo "export PATH=$PATH:$HOME/tdiu16-labs/src/utils" >> ~/.bashrc
source ~/.bashrc
```

Nu är både du och din labpartner redo för avsnittet ”Kompilera och kör Pintos”. Kontrollera var för sig att Pintos fungerar på ditt konto genom att gå igenom det avsnittet.

Felsökning: Om `gitlab.liu.se` frågar efter lösenord så är det något problem med ssh-nyckeln. Prova köra `ssh-add` i en terminal (starta ev `ssh-agent`). Kontrollera att `~/.ssh/id_rsa.pub` motsvarar den nyckel

som du lagt upp på ditt konto på `gitlab.liu.se`. Försök komma ihåg om du angav något lösenord när du skapade din nyckel, om du gjorde det, skapa en nyckel utan lösenord.

2.4 Arbetssätt med Git

Det här gäller både dig och din labpartner. Innan du börjar arbeta måste du tanka hem allt som eventuellt ändrats (av din labpartner, eller av dig själv på en annan dator).

```
cd ~/tdiu16-labs
git pull
```

När du vill spara dina ändringar av källkoden kör du exempelvis nedan kommandon. Först väljs vilka filer som ska sparas (här: alla uppdaterade), sedan sparas ändringarna lokalt med en bra kommentar, och slutligen pushas allt till remote repository. Ofta är även `git status` bra för att få veta mer om vikla filer som ändrats.

```
git add -u
git commit -m "Nu fungerar systemanropet xyz!!!! Jay!"
git push
```

2.5 Bläddra i Pintos källkod bekvämt med Emacs

För att enkelt kunna använda emacs och bläddra i koden kan du skapa en så kallad TAGS-fil i src-katalogen som berättar för emacs var olika deklarationer och definitioner finns i koden. Det innebär att du genom att placera markören vid ett funktionsanrop i koden och trycka M-. (Meta-punkt) i emacs kan hoppa direkt till definitionen av motsvarande funktion. M-* hoppar tillbaka. Första gången du använder detta måste du hjälpa emacs att lokalisera TAGS-filen. TAGS-filen skapas eller uppdateras från din src katalog med kommandona:

```
cd ~/tdiu16-labs/src
make TAGS
```

2.6 Kompilera och kör Pintos

Följande kommandon utgår från att du följde ovan instruktioner till punkt och pricka. Om du inte gjorde det kan du behöva justera kommandon eller sökvägar nedan. I resten av instruktionen kommer alla sökvägar att anges relativt src-katalogen i Pintos. Det testprogram du skall använda i exemplen nedan finns i `examples/sumargv.c`. Lokalisera programmets källkod och kontrollera varför det (ibland) avslutar med kod 111.

Använd sedan följande kommandon för att kompilera Pintos och testköra programmet.

```
cd ~/tdiu16-labs/src/userprog
make -j8 -C ../examples
make -j8
pintos -p ../examples/sumargv -a sumargv -v -k --fs-disk=2 -- -f -q run sumargv
```

När du kör Pintos enligt ovan kommer mycket status-information att skrivas ut. Var alltid mycket uppmärksam på eventuella felmeddelanden. Nedan är denna information uppbruten i delar med några korta kommentarer (inklusive hur du stoppar programkörningen med Ctrl-a x!).

Följande text beskriver status för uppsättningen av emulator och initial disk:

```
Copying ../examples/sumargv into /tmp/MAdPEpgFX5.dsk...
Writing command line to /tmp/Jh32hYvaz0.dsk...
qemu -hda /tmp/Jh32hYvaz0.dsk -hdb /tmp/InW2t5E1LN.dsk -hdc /tmp/MAdPEpgFX5.dsk
     -p 1234 -m 4 -net none -monitor null -nographic
```

Därefter kommer Pintos boot-meddelanden:

```
Kernel command line: -f -q put sumargv run sumargv
Pintos booting with 4,096 kB RAM...
375 pages available in kernel pool.
374 pages available in user pool.
# main#1: thread_create("idle", ...) RETURNS 2
Calibrating timer...  16,460,800 loops/s.
hd0:0: detected 129 sector (64 kB) disk, model "QEMU HARDDISK", serial "QM00001"
hd0:1: detected 4,032 sector (1 MB) disk, model "QEMU HARDDISK", serial "QM00002"
hd1:0: detected 81 sector (40 kB) disk, model "QEMU HARDDISK", serial "QM00003"
```

Rader som startar med tecknet # är debug-meddelanden för att lättare kunna följa exekveringen av några centrala och viktiga funktioner.

Filsystemet formateras när flaggan `-f` anges, och filer kopieras in då flaggor `-p` och `-a` anges:

```
Formatting file system...done.
Boot complete.
Putting 'sumargv' into the file system...
```

Så följer starten av den första processen (`run sumargv`). Eftersom Pintos inte är fullt funktionellt ännu kommer inte så kommer några felutskrifter och programmet avslutar till följd av ett okänt(ej implementerat) systemanrop:

```
Executing 'sumargv':
# main#1: process_execute("sumargv") ENTERED
# main#1: thread_create("sumargv", ...) RETURNS 3
ERROR: Main about to poweroff with 2 threads still running!
ERROR: Check your process_execute() and process_wait().
# sumargv#3: start_process("sumargv") ENTERED
# sumargv#3: start_process(...): load returned 1
# sumargv#3: start_process("sumargv") DONE
Executed an unknown system call!
Stack top + 0: 1
Stack top + 1: 111
# sumargv#3: process_cleanup() ENTERED
sumargv: exit(-1)
# sumargv#3: process_cleanup() DONE with status -1
```

Felutskrifterna beror på att varken funktionen `process_execute` eller funktionen `process_wait` är korrekt implementerad. Det kommer du att göra i en senare uppgift. Nuvarande funktioner har bara grundfunktionalitet för att du skall kunna komma igång med systemanrops-implementationen.

Funktionen `process_execute` stänger av datorn (men borde returnera den nya processens id), och funktionen `process_wait` är bara implementerad som en stub som returnerar `-1` (minus ett) direkt. Funktionen `process_wait` anropas för att Pintos skall vänta tills `sumargv` (som är den första processen i detta fall) blir klar. När första processen är klar avslutar Pintos. Om `process_wait` returnerar för tidigt så kommer operativsystemet avsluta, och kanske stänga av datorn (om flagga `-q` angavs) medan jobb fortfarande finns kvar att utföra. I nuläget kommer inte exekveringen så långt eftersom den ofärdiga `process_execute` hinner stänga av datorn med `power_off` först.

Avstängningskoden har i vår version av Pintos felhanteringskod tillagd som gör att operativsystemet skriver ut ett fel men ändå väntar tills alla trådar är klara. Detta är praktiskt att ha medan du arbetar med Pintos, och är det enda som gör att det alls går att starta en process innan `process_execute` och `process_wait` korrigerats av dig. Kom ihåg: I vissa lägen kommer Pintos ändå att ”låsa sig” utan att stänga av. Tryck då `Ctrl-a` och sedan `x` för att avsluta emulatorn QEMU. Detta bekräftas med följande meddelande:

```
QEMU: Terminated
```

Fungerar inte det brukar ett bra trick vara att öppna en ny terminal och i den köra:

```
killall pintos
```

Programmet sumargv du körde på förra sidan skriver ut de två översta värdena på user-stacken (fetstilat). Kan du lista ut var det andra värdet kommer från? Kan du lista ut vad det första är? Studera koden för sumargv och fundera på vilket systemanrop som utförs då main returnerar. Med kännedom om hur systemanrop går till och anropas (lib/user/syscall.) och numreras (lib/syscall-nr.h), stackens utseende från "Systemanrop i Pintos", samt programmets kod (examples/sumargv.c) bör du kunna klura ut det. Att ta reda på detta ger dig kunskaper du behöver när du implementerar systemanropen.*

När Pintos sedan avslutar skrivs lite statistik ut:

```
Timer: 54 ticks
Thread: 0 idle ticks, 52 kernel ticks, 2 user ticks
hd0:0: 0 reads, 0 writes
hd0:1: 53 reads, 170 writes
hd1:0: 81 reads, 0 writes
Console: 1302 characters output
Keyboard: 0 keys pressed
Exception: 0 page faults
```

Om flagga `-q` angavs då Pintos startade kommer operativsystemet även stänga av emulatorn (datorn). Om trådar fortfarande kör kommer debug-kod att generera en felutskrift och vänta på att dessa avslutar. När emulatorn stängs av ser du följande meddelande:

```
Powering off...
```

Vi har nu gått igenom en hel programkörning med Pintos, från start av operativsystemet (boot-sekvens), via exekvering av ett program, till hur det (så småningom) skall se ut när Pintos avslutar. Kommandoraden du angav för att starta Pintos innehöll många flaggor och argument. I nästa avsnitt där felsökning introduceras används ett alternativt sätt att skriva kommandoraden, med minimalt antal flaggor. För att få mer information om vilka flaggor som kan användas och hur de fungerar kan du skriva:

```
pintos --help
```

2.6.1 Kort om x86 emulatorn QEMU

Det är även användbart att känna till lite om emulatorn `qemu` som är den emulator vi använder. Här följer några tangentbordskombinationer som kan vara användbara. Ytterligare information kan den intresserade hitta på `http://www.qemu.org/`.

```
<Ctrl-Alt> : i det grafiska fönstret tar eller släpper kontroll
             över tangentbord och mus.
<Ctrl-a x> : Avslutar emulatorn. Släpp control innan du trycker på x.
             Kan behöva tryckas flitigt för att ge effekt.
```

3 Bra att känna till om C

3.1 Parametrar till `main` i C

I C (och C++) har funktionen *main* enligt standard två inparametrar:

```
int main(int argc, char* argv[]);
```

Den första är ett heltal, och den andra en array där varje position är en adress till tecken (en C teckensträng, en pekare till en sekvens med tecken). Detta kommer man enklast fram till genom att läsa deklarationen baklänges. Arrayen skall enligt C-standard avslutas med en `NULL`-pekare. Detta gör att första parametern som anger längden på arrayen (exklusive avslutande `NULL`-pekare) inte nödvändigtvis behöver användas för att stega igenom hela arrayen, utan det går att stega tills `NULL` påträffas. Innehållet i arrayen bestäms av vad som skrivs på kommandoraden i terminalen. Varje ord, inklusive namnet på programmet, kommer ta upp ett index i arrayen. Detta betyder att:

`argv[0]` Anger alltid namnet på programmet (om möjligt)

`argv[argc]` Är alltid `NULL`

Enligt C standard skall `main` också returnera ett heltal. Heltalet noll (0) signalerar att main avslutade normalt utan fel. Övriga returvärden är upp till programmeraren att ge en betydelse. För att från terminalen se vilket varde som returnerades från main (eller gavs som parameter till `exit` (`man -s3 exit`)) skriver du kommandot `echo $?`. Observera att inga andra kommandon kan köras emellan, eftersom miljövariablen `$?` endast visar returvärdet från senast körda kommando. Observera också att returvärdet representeras med endast 8 bitar (0-255) i terminalen.

3.2 Compilera C-program

För att kompilera använder du `gcc` med följande flaggor:

```
gcc -Wall -Wextra -std=c99 -pedantic -g assignment1.c
```

Programmet kommer heta `a.out` om du inte döper om det med flaggan `-o`

Ofta behöver du också lägga till flaggan `-m32`, för att kompilera i 32-bitarsläge. I och med att Pintos är ett 32-bitars operativsystem är vissa exempel designade för ett 32-bitarssystem. Behövs denna flaggan står det i kommentaren överst i programfilen.

3.3 Stränghantering i C

Det finns en stor uppsättning funktioner för att hantera C-teckensträngar i standardbiblioteket. Titta på följande manualsidor:

- `man -s3 printf`
- `man -s3 strlen`
- `man -s3 strlcpy`
- `man -s3 strtok_r`

Dessa funktioner finns även tillgängliga i Pintos.

4 Utveckling av Pintos

4.1 Hur källkoden är strukturerad

I mappen `src/` i Pintos-repositoryt finns all källkod. Den är strukturerad i en samling undermappar, var och en med en specifik funktion. Nedan följer en lista över mapparna, ordnade baserat på vilket läge den exekeveras i (kernel-mode eller user-mode):

Kernel-mode:

`devices/` Här finns all kod som pratar med hårdvaran i Pintos. När vi kör Pintos i QEMU kommer QEMU att emulera ett komplett system åt oss, med bland annat en disk, tangentbord och skärm. Pintos

måste då prata med dessa enheter på samma sätt som man skulle ha gjort på riktigt. Detta finns implementerat här.

`filesys/` Här finns all kod som hanterar filsystemet i Pintos, det vill säga hur och var filer lagras på disk, samt åtkomst till filerna.

`threads/` Här finns all hantering av trådar i Pintos. Bland annat finns här den scheduler som används i Pintos, logik för att byta vilken tråd som är aktiv, samt synkroniseringsprimitiver.

`userprog/` Här finns all hantering av user-mode-program, eller processer. I och med att Pintos bara tillåter varje process att ha en tråd blir distinktionen mellan tråd och process inte alltid helt tydlig. Tumregeln är att allt som har med att köra program att göra hör hemma i `userprog/`, medan det som är meningsfullt att använda utan att starta program hör hemma i `threads/`.

`vm/` Här finns allt som har att göra med hantering av virtuellt minne i Pintos. Detta innefattar allt från att allokera hela pages med minne till att implementera exempelvis `malloc` och `free`.

User-mode:

`examples/` Innehåller en samling program som går att köra inuti Pintos. Varje program består i allmänhet av en `c`-fil. Vill du göra egna program kan du skapa en fil här och lägga till den i `Makefile` för att kompilera den. Notera att programmen i den här mappen **inte** går att köra direkt i Linux eftersom Linux och Pintos inte fungerar på samma sätt.

`tests/` Innehåller en uppsättning testprogram som körs automatiskt av `make check` (se detaljer under ”Automatiska tester”). Dessa program är i allmänhet som de som finns i `examples/`, med skillnaden att det också finns ett script som kontrollerar om utdatan ser korrekt ut eller inte.

`lib/user/` Implementation av diverse hjälpfunktioner som används för att implementera C-standardbiblioteket. Här implementeras även usermode-biten av alla systemanrop i Pintos (i `syscall.c`).

Annat:

`lib/` Här finns den implementation av C-biblioteket som används i Pintos. Detta innefattar funktioner såsom `printf`, `strlen` och så vidare. De filer som ligger direkt i `lib/` körs både som kernel- och usermode medan undermapparna `kernel/` och `user/` innehåller de delar som skiljer sig mellan user- och kernel-mode.

`standalone/` En samling program som är gjorda för att byggas och köras under Linux för att implementera funktionalitet till Pintos utanför Pintos, så att vanliga utvecklingsverktyg kan användas för felsökning.

`utils/` En samling script som hjälper dig att köra och felsöka Pintos. Dessa behöver du om du kör Pintos på din egna dator. På skolans system finns en central kopia av scripten som du kan lägga till med en modul. Se ”Installera Pintos” för detaljer.

4.2 Kompilera Pintos

Under rubriken ”Installera Pintos” såg vi en hel del kommandon för att kompilera Pintos. Här kommer vi att titta närmare på vad de gör.

Pintos kompileras med hjälp av GNU Make, det vill säga kommandot `make`. För att se hur Pintos kompileras räcker det alltså med att läsa de `Makefile`-filer som finns. I och med att det inte alltid är uppenbart vad som händer i en makefil kommer här en sammanfattning som är enklare att förstå.

En första iakttagelse är att de makefiler som finns beter sig olika beroende på vilken mapp de körs från. Får man underliga fel kan det alltså vara värt att dubbelkolla att man står i rätt mapp när man kör `make`. Följande mappar är relevanta för den här kursen:

userprog/ Bygger Pintos-kärnan med stöd för att köra usermode-program (kör man `make` i mappen `threads/` kommer inte stöd för usermode-program följa med, vilket kan leda till problem).

examples/ Bygger alla usermode-program i `examples/`-mappen.

Det enklaste sättet att berätta för `make` vad man vill göra är att helt enkelt `cd`:a till rätt mapp och köra `make` därifrån. Det är dock vanligt att man vill bygga om både kärnan och usermode-programmen exempelvis, och då är det bökigt att `cd`:a fram och tillbaka hela tiden. Då kan man i stället använda flaggan `-C` (stora C) till `make`, exempelvis `make -C ../examples`, vilket instruerar `make` att själv göra `cd` till den indikerade mappen innan den börjar arbeta. I och med att det är `make` som ändrar vilken mapp den står i så påverkas inte din terminal.

En annan flagga som är bra att känna till är flaggan `-j`. Den instruerar `make` att det är okej att köra flera kompileringsprocesser samtidigt i den mån det är möjligt. I och med att de flesta datorer numera har flera kärnor så snabbar detta upp kompileringsprocessen oerhört. Nackdelen är att det inte alltid är uppenbart vilket kommando som orsakade vilket felmeddelande i och med att flera kompileringsprocesser körs samtidigt. På IDA:s system är det lämpligt att skriva `-j8`, vilket instruerar `make` att köra maximalt 8 processer samtidigt. Att bara skriva `-j` är också möjligt, men **inte** en bra idé, eftersom man då tillåter `make` att köra så många processer som möjligt samtidigt. Detta kan leda till att man kör så pass många processer att man får slut på minne (framförallt när du senare kör testerna).

I övrigt kan det vara bra att känna till att `make clean` finns och fungerar som förväntat.

Om du vill lägga till ett nytt program i `examples/` modifieras `examples/Makefile` enligt följande. Antag att vi vill lägga till programmet `my_test` i filen `my_test.c`:

1. Lägg till `my_test` i `PROGS` listan.
2. Lägg till raden: `my_test_SRC = my_test.c`
3. Skriv `make` i examples-katalogen för att kompilera.

4.3 Att köra Pintos

Under rubriken ”Installera Pintos” så togs många kommandon för att köra Pintos upp utan särskilt ingående förklaring. Här kommer vi att titta närmare på vad de olika flaggorna gör, så att du förstår vad de ”magiska” kommandona gör.

I och med att Pintos är ett operativsystem, så måste det köras i en virtuell maskin. I den här kursen använder vi QEMU, och för att förenkla hanteringen av detta finns ett script som heter `pintos` som hjälper oss med det (skrivet i Perl). Precis som vid kompilation av Pintos spelar det roll vilken mapp man står i när man kör `pintos`-kommandot. Eftersom vi huvudsakligen arbetar med usermode-program vill vi i den här kursen stå i mappen `userprog/`, annars kommer den klaga på att den inte hittar diskavbilden den ska köra.

Alla parametrar finns beskrivna i hjälptexten för `pintos`-kommandot (kör `pintos --help`). Här tas bara de viktigaste parametrarna upp.

Parametrarna som skickas till `pintos`-kommandot är uppdelade i två delar: en del som hanteras av ditt Linux-system (värdstystemet), och en del som skickas mer eller mindre direkt till Pintos (gästsystemet). Dessa är enkla att se på kommandoraden eftersom de separeras av ett dubbelt bindestreck (`--`). Allt före separatorn hanteras av värdsystemet, resten skickas till Pintos.

Parametrar till värdsystemet

Här specifieras alla parametrar som har att göra med hur QEMU ska startas, och alla parametrar som har att göra med filer som finns i värdsystemet (alltså allt du kan komma åt via terminalen, eller den grafiska filvisaren i Linux). Intressant för oss är följande:

`-v` Kör utan VGA-utmatning. Om flaggan inte finns med så öppnas ett fönster där Pintos skärm visas. Eftersom Pintos skriver ut all data till terminalen som Pintos startades i också är skärmen i många fall onödig och bara ivägen.

`-k` Stänger av maskinen när Pintos har kraschat. Detta sker då scriptet ser att Pintos har skrivit ut texten `PANIC` (med stora bokstäver), så se till att inte skriva ut `PANIC` under vanlig körning.

`--fs-disk=n` Instruerar scriptet att skapa en hårddisk som Pintos kan använda att lagra filer på. Disken ska vara `n` MB stor. Disken tas bort när Pintos avslutas. Om flaggan inte specifieras används filen `fs.dsk`, vilken måste skapas manuellt i så fall.

`-p <fil> -a <namn>` Kopiera filen `<fil>` från värdsystemet till Pintos, och ge den namnet `<namn>` inuti Pintos.

`-g <fil> -a <namn>` Kopiera filen `<fil>` från Pintos till värdsystemet innan Pintos stängs av. Filen får namnet `<namn>` i värdsystemets filsystem.

Parametrar till Pintos

Pintos tolkar sina parametrar i filen `src/threads/run.c` ifall du vill se hur de hanteras, och exakt vilka kommandon som finns tillgängliga.

Parametrarna till Pintos kan man se som en sekvens av kommandon som utförs ett i taget tills alla är klara.

`-f` Formatera filsystemsdisken. Om du använder `--fs-disk=n` som parameter till värdsystemet vill du alltid säga till Pintos att formattera disken. Annars kommer Pintos inte lyckas läsa disken och avsluta med ett fel innan något annat händer.

`-q` Stäng av maskinen när alla kommandon har kört färdigt. Utan denna flaggan fortsätter Pintos att köras, och du måste avsluta QEMU på annat sätt (exempelvis Ctrl+C, eller `pkill pintos` i en annan terminal).

`run '<kommando>'` Kör ett program som finns i Pintos filsystem. `<kommando>` är en sträng likt kommandon i Bash, som börjar med namnet på programmet och kan innehålla en eller flera parametrar som skickas till programmet.

Exempel

För att illustrera hur en kommandorad kan se ut kommer här ett exempel på hur man kan köra ett program i Pintos. Kommandot antar att du står i `userprog/` mappen och har kört `make` i både `userprog/` och `examples/`:

```
pintos -v -k --fs-disk=2 -p ../examples/sumargv -a sumargv -- -f -q run 'sumargv 1 2'
```

Detta kommando startar Pintos utan VGA (`-v`), stänger av allt vid krasch (`-k`), skapar en temporär disk till Pintos filsystem (`--fs-disk=2`) och kopierar filen `../examples/sumargv` (`-p`) till Pintos med namnet `sumargv` (`-a`). Pintos ser sedan instruktioner om att formattera sin disk (`-f`), stänga av sig när den är klar (`-q`) och sedan köra `sumargv` med parametrarna `1` och `2`.

5 Felsökning

Att felsöka är utmanande och krävande. Ibland är det ens väldigt svårt att acceptera att det är fel. Ibland verkar det som om det som händer omöjligt KAN hända. Trots att det händer.

När du felsöker måste du vara ”open-minded” och struktureread. Att det fungerade i förrgår betyder inte att det var rätt då. Försök ta tillvara på det som faktiskt händer och spåra felet bakåt. Har en variable fel värde? Ta reda på alla ställen där den variabeln tilldelas. Kontrollera att den alltid initieras. Ta reda på

vilken tilldelning som ger variabeln fel värde. Den fick fel värde från någon av de variabler som förekommer i tilldelningsuttrycket. Vilken? **Skriv ut tydlig spårinformation.**

Fortsätt på samma sätt med nästa variabel som är fel, och nästa, tills du förstår hur felet uppstod. Ibland kör du fast ändå. Fråga då en assistent. Tydlig information är felaktigheters största fiende. Att bara lägga in en spårutskrift ”Nu är jag här” och sedan en ”Nu är jag här 2” räcker inte på långa vägar. Ibland måste du även lägga in kod som bara skriver ut spårinformation ”vid rätt tillfälle” för att du inte skall drunkna i den.

Se till att testa din kod ofta och noga. Försök skapa testfall för varje möjlig exekveringssekvens. Ibland är det lättare att testa ett delsystem utanför Pintos med ett eget testprogram.

Felsök systematiskt. Skriv ut värdet på alla variabler som kan påverka, även de som du ”vet är rätt” (går något fel så är ju något av det du ”vet” fel, eller hur?). Skriv noggranna felmeddelanden med mycket information, ”kalle1”, ”kalle2” osv är inte bra felutskrifter. Det bör framgå av varje utskrift vilken tråd som exekverar, vilken källkodfil, funktion och rad det är, samt namn och värde på de variabler som funktionen använder. Det gäller att hitta den information som inte är som förväntat. Var förutsättningslös. Blir det fel, oavsett hur omöjligt felet är, så är det fel, och det är i din kod felet finns även om symptomen uppstår djupt inne i någon befintlig funktion. Använd backtrace för att spåra sådant tillbaka till din kod.

5.1 Spårutskrifter

För att lätt kunna sätta på och stänga av olika utskrifter, t.ex. debug-utskrifter, är det bra att definiera ett makro som ersätter `printf`:

```
#define DBG(format, ...) printf(format "\n", ##__VA_ARGS__)
```

Antag sedan att du vill göra följande debugutskrift:

```
printf("# exekverade rad %d i filen %s", __LINE__, __FILE__);
```

Med hjälp av makrot kan du istället skriva:

```
DBG("# exekverade rad %d i filen %s", __LINE__, __FILE__);
```

Finessen är att du senare kan stänga av alla utskrifter som gjorts via `DBG` genom att ändra makrot till:

```
#define DBG(format, ...)
```

Konstanterna `__LINE__` och `__FILE__` som används ovan byts automatiskt ut mod aktuell rad och aktuellt filnamn.

OBS! Notera att utskrifterna ovan startar med `"# "`. Detta gör att dessa rader senare ignoreras då de automatiska testprogrammen körs. Rader utan denna inledning kommer tolkas som att testet gick fel. Det finns redan ett makro `debug` i `lib/debug.h` som lägger till dessa tecken automatiskt, men det kan vara intressant att ha olika namn på `debug` i olika filer för att lättare kunna sätta på och stänga av olika meddelanden (`syscall_debug`, `process_debug` etc.).

5.2 Hitta minnesläckor

Pintos innehåller ett enklare system för att hitta minnesläckor i systemet (lite likt vad Valgrind kan göra för C och C++-program). Det är inte aktivt som standard, utan behöver slås på genom att skicka flaggan `-L` till Pintos vid uppstart. Denna flaggan läggs alltså till mellan `--` och `run` på kommandoraden.

Det systemet gör för att hitta minnesläckor är att det håller koll på alla anrop till `malloc` och `free`, och ser till så att allt minne som är allokerat med `malloc` har blivit frigjort när Pintos stängs av. Om det finns minne som inte har frigjorts när Pintos stängs av så skriver systemet ut en lista på de allokeringar som är kvar, tillsammans med namnet på funktionen som anropade `malloc`.

Detta sätt att hitta minnesläckor på liknar hur Valgrind gör. Det finns dock ett par saker att tänka på:

- Detektionen av minnesläckor är, likt Valgrind, inte intelligent. Det räknar helt enkelt anrop till `malloc` och ser till att matchande anrop till `free` finns. Detta innebär att om systemet kraschar eller stängs av i förtid (exempelvis med systemanropet `halt`) så kan ”falska” minnesläckor rapporteras i och med att koden som städar upp inte har ”hunnit” köras ännu.
- Om du allokerar minne med `malloc` och lägger i globala pekarvariabler så kommer de rapporteras som minnesläckor. Detta är inte nödvändigtvis ett problem, eftersom minnet fortfarande används av systemet. Dock vet inte detektionen av minnesläckor detta, och kommer att anta att det är en minnesläcka (detta är saker som Valgrind rapporterar som ”Possible leaks”). Problemet kan lösas genom att allokera hela arrayen statiskt (ex.vis som en array i stället för en pekare). Alternativt kan kod för att städa upp läggas till i funktionen `hard_power_off` i `threads/init.c`. En sista utväg är att markera enstaka allokeringar som icke-problematiska med funktionen `not_a_leak`. Detta ska dock göras sparsamt, eftersom det eliminerar vitsen med kontroll av minnesläckor.
- Om systemet hittar en minnesläcka så skriver den ut namnet på *funktionen som anropade* `malloc`. Detta betyder inte nödvändigtvis att det är i den funktionen eller datastrukturen som felet ligger. I många fall så kommer systemet peka på en funktion som allokerar en datastruktur (exempelvis `filesys_open`), men felet ligger i att användaren av funktionen misslyckades med att avallokera minnet i något fall (med `filesys_close` i det här fallet). Används fel avallokeringsfunktion (ex.vis `free` i stället för `filesys_close`) så kan ibland interna datastrukturer visas som läckor också. Exempelvis kommer det bli en läcka från `inode_open` ifall man använder `free` i stället för att anropa `filesys_close`.

Notera att detektion av minnesläckor är ny för 2024. Den fungerar bra nog för att vi ska kunna köra vår referenslösning utan minnesläckor (förutom för testet `halt`, som faller under den översta punkten ovan). Det kan däremot finnas acceptabla lösningar som kan betraktas som att de läcker minne. Kravet i kursen är att det inte ska finnas några uppenbara minnesläckor när vi går igenom koden. Har du alltid ett konstant antal allokeringar som inte frigörs, oavsett vilket program som körs är det antagligen OK. Fråga din assistent om du är osäker!

5.3 Felsökning med GDB

Notera att debuggern inte klarar hantera brytpunkter i implementationsfiler vars namn innehåller mellanslag. Byt ut mellanslag mot understrykningstecken (_). En gammal programmerare använder bara engelska bokstäver, siffror, vanligt bindestreck och understrykningstecken i filnamn.

I första laborationen finns en mer detaljerad genomgång av hur man kan hitta fel med GDB.

Tutorial Programmet `debug.c` finns i Pintos-repositoryt i mappen `src/standalone/labx4`. Programmet innehåller dock ett allvarligt fel du skall felsöka. *Felsökning kommer bli en stor del av Pintos-laborationerna.* Börja med att kompilera programmet:

```
gcc -Wall -Wextra -std=c99 -pedantic -g debug.c
```

Starta sedan programmet med hjälp av debuggern `gdb`. Det finns en grafisk front-end `ddd`, men effektivast är att använda text-mode `gdb`, eller det text-ui som finns i GDB:

```
gdb a.out
```

Eller med text-ui:

```
gdb -tui a.out
```

Debuggern startar, läser in programmet `a.out` och presenterar en kommandoprompt `(gdb)`. Skriv `run` (istället för `a.out`) för att starta programmet. (Om main förväntar sig argument på kommandoraden kan de anges ”som vanligt” efter `run`.) Programmet kommer nu att krascha eftersom det innehåller ett fel. Skriv `bt` eller

`backtrace` för att se hela kedjan av funktionsanrop som ledde fram till krashen. Med större program är denna information ovärderlig, men i detta fall finns bara main. Vill du nu undersöka värdet på olika variabler kan du använda kommandot display. Några exempel:

```
(gdb) display *bufi
1: *bufi = 0x0
(gdb) display bufi
2: bufi = (char **) 0xffbfe25c
(gdb) display bufend
3: bufend = (char **) 0xffbfe25c
(gdb) display *(bufend-1)
4: *(bufend - 1) = 0xffbfe268 "sihtgubed"
```

Som du ser kan man skriva komplicerade uttryck (C-kod) för att undersöka innehållet av pekare. Skriv nu kommandot `break 12` för att stoppa programmet på rad 12 vid nästa körning. Starta om programmet genom att skriva `run` och bekräfta. Programmet stannar vid rad 12, och visar alla displayuttryck. Skriv `undisplay` för att inte visa uttrycken igen. För att fortsätta program-körningen kan du nu skriva `next`, `nexti`, `step`, `stepi` eller `continue`. Om du bara trycker enter upprepas föregående kommando. Använd `help` för att se skillnaden på de olika stegningarna (`help next`).

5.4 Felsökning av Pintos

Att använda debugger kräver i princip samma kommandon och flaggor som att köra utan, samt en *extra terminal för debuggern*. Dock passar vi här på att gå igenom ett alternativt sätt att skapa Pintos disk, och skippar några flaggor vi kan klara oss utan. Detta gör kommandoraden lite kortare, men det blir ibland lite svårare att nyttja samma kommandorad igen genom att bara trycka upp-pil i terminalen. Kör följande kommandon:

```
cd ~/tdiu16-labs/src/userprog
debugpintos --fs-disk=2 -p ../examples/sumargv -a sumargv -- -f run sumargv
```

Det som är nytt är att vi använder `debugpintos` istället för `pintos`, för att debuggern senare skall kunna ansluta. Pintos kommer nu vänta på att en debugger skall ansluta. **I en annan terminal** *startar du debuggern:*

```
cd ~/tdiu16-labs/src/userprog
pintos-gdb build/kernel.o
```

eller

```
cd ~/tdiu16-labs/src/userprog
pintos-gdb -tui build/kernel.o
```

Du får upp en hel del text när debuggern startar, följt av en prompt där du skriver `debugpintos`. Det du skall skriva är i kursiv stil nedan.

```
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:.
Find the GDB manual and documentation resources online at:.
```

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from build/kernel.o...done.

(gdb) debugpintos
0x0000fff0 in ?? ()
```

Debuggern skriver ut lite frågetecken. Det är OK, då har den anslutit till emulatorn. Skriv `break main` för att skapa en initial brytpunkt:

```
(gdb) break main
Breakpoint 1 at 0xc0100008: file ../../threads/init.c, line 68.
```

Debuggern bekräftar att den är med på noterna. Fortsätt programkörningen med `continue`:

```
(gdb) continue
Continuing.
Breakpoint 1, main () at ../../threads/init.c:72
72      {
```

Pintos kommer nu köras fram till brytpunkten vid `main`. Nu kan du sätta valfria brytpunker eller använda valfria kommandon i debuggern för att utföra felsökning. I detta exempel nöjer vi oss med att skapa ytterligare en brytpunkt vid `process_execute`, och sedan fortsätta (`continue`) tills Pintos kommer dit.

```
(gdb) break process_execute
Breakpoint 2 at 0xc0108367: file ../../userprog/process.c, line 147.
(gdb) continue
Continuing.Breakpoint 2, process_execute (command_line=0xc0007d91 "sumargv") at
../../userprog/process.c:147
147       int command_line_size = strlen(command_line) + 1;
```

Använd sedan kommandot `next` för att stega en rad i programmet:

```
(gdb) next
166       debug("%s#%d: process_execute('%s') ENTERED\n",
```

Nästa rad att exekvera skrivs ut. Om du inte skriver något kommando i debuggern, utan bara trycker *Enter* så kommer föregående kommando att upprepas. Prova:

```
(gdb)
179       arguments.command_line = malloc(command_line_size);
(gdb)
180       strlcpy(arguments.command_line, command_line, command_line_size);
(gdb)
200       strlcpy_first_word (debug_name, command_line, 64);
(gdb)
204       thread_id = thread_create (debug_name, PRI_DEFAULT,
(gdb)
256       power_off();
```

Prova nu kommandot `backtrace`. Det ger information om hur programstacken ser ut, vilka funktioner som ledde till raden som exekveras:

```
(gdb) backtrace
#0  process_execute (command_line=0xc0007d91 "sumargv") at ../../userprog/process.c:256
#1  0xc0100559 in run_task (argv=0xc010f44c) at ../../threads/init.c:278
#2  0xc01005fd in run_actions (argv=0xc010f444) at ../../threads/init.c:330
```

```
#3  0xc01000bd in main () at ../../threads/init.c:126
```

Stega vidare så att även `power_off` exekveras. Pintos avslutas. Sedan avslutar du debuggern:

```
(gdb) next
Watchdog has expired.  Target detached.
(gdb) quit
```

Du kan själv undersöka vilka andra felsökningsmöjligheter som finns. Här följer några av de mest vanliga kommandon du kan använda i debuggern. Kommandot `backtrace` är kanske det mest användbara. Mer ovanliga kommandon inkluderar kommandon för att skriva ut minnesinnehåll eller köra disassembler på funktioner. Sådant är kanske mest användbart för den som felsöker en kompilator eller assembler.

```
help
help bt
help next
backtrace
bt
next
nexti
step
stepi
break
clear
delete
display
undisplay
```

Om du tycker det är svårt att tyda de ibland kryptiska meddelandena från GDB så kan det vara värt att känna till att GDB har ett terminalbaserat UI som kan startas genom att lägga till flaggan `-tui` till kommandot pintos-gdb.

6 Inbyggda strukturer i Pintos

Det finns ett antal inbyggda datastrukturer i pintos som kan vara användbara. Detta kapitel beskriver dessa strukturer i detalj under respektive rubrik.

6.1 Länkad lista

I Pintos finns en länkad lista (src/lib/kernel/list.h) som är lite speciell. I grunden är det en dubbellänkad lista. För att undvika specialfall i början används ett ”dummy” header-helement, och för att undvika motsvarande specialfall i slutet används ett ”dummy” tail-element. Funktioner finns färdiga för att initiera listan, sätta in element först, sist, sorterat eller unikt, iterera genom listan, ta bort element osv. Både headerfilen och implementationsfilen innehåller användbara kommentarer. Det som är speciellt är listelementen som bygger upp listan. De är deklarerade som:

```
struct list_elem
  {
    struct list_elem *prev;     /* Previous list element. */
    struct list_elem *next;     /* Next list element. */
  };
```

Som du ser finns alltså en next-pekare och en prev-pekare. Men det finns ingen plats för data. Listan lagrar ingenting. Nu tycker du kanske det är ganska dumt att implementera en lista som inte lagrar någon data. I

själva verket är det ett genidrag. Listan är nu helt generell. Istället för att bara kunna lagra ett slags data, av förutbestämnd datatyp, kan den nu lagra vad som helst. Funktionerna som hanterar lista, t.ex. för insättning och borttagning behöver aldrig anpassas till datatypen som lagras, de vet överhuvudtaget inte om att någon data lagras.

För att lagra något i listan gör du ”tvärtom” vad som är ”normalt”. Istället för att lagra data i varje listelement skall du lagra listelementet tillsammans med din data. Det du sätter in i och får ut från listan är *alltid* listelementet. När du får ut ett listelement **VET** du dock att det ligger tillsammans med din data i minnet eftersom du skapade det så, och kan använda det speciella makrot `list_entry` för att få en pekare till ditt data. Detta fungerar enligt följande exempel.

6.1.1 Exempel på användning av listan med C-teckensträngar

1. Deklaration och initiering som krävs för att skapa och initiera en lista:

   ```
   struct list my_list;
   list_init(&my_list);
   ```

2. För att lagra C-teckensträngar i listan behövs en datatyp som lagrar en C-teckensträng tillsammans med ett listelement. Den datatypen skapas som en struct. Observera variabelnamnet på listelementet, i det här fallet “elem” då det behövs senare.

   ```
   struct string_elem
   {
     char* string;
     struct list_elem elem;
   };
   ```

3. Följande kod allokerar och initierar ett listelement av datatypen från föregående steg.

   ```
   struct string_elem* my_string_elem
     = (struct string_elem*)malloc(sizeof(struct string_elem));

   my_string_elem->string= "I rule the Pintos list"
   // my_string_elem->elem is initialized when inserted in list
   ```

4. För att sätta in `my_string_elem` i listan skall du ange adressen till ditt `list_elem`, den medlemsvariabel som angavs i fet stil förut. Du måste även ange adressen till den lista du vill sätta in i.

   ```
   list_push_front(&my_list, &my_string_elem->elem);
   ```

5. När du hämtar ut data från en lista kommer du att få exakt den adressen som du satte in, dvs en pekare till ditt `list_elem`. Denna kan konverteras till den datatyp som du skapade tidigare (i detta exempel `struct string_elem`) med hjälp av det funktionslika makrot `list_entry`. Makrot `list_entry` tar emot den pekare till `struct list_elem` som du fick ut från listan, och den datatyp som det skall konverteras till (`struct string_elem`). För att utföra adressberäkningarna korrekt behöver makrot även veta hur långt in i din struct som listelementet ligger. Till det behövs återigen variabelnamnet på listelementet vilket i det här fallet är “elem”. (Om listelementet ligger t.ex. 8 byte in i din struct, och om listelementet ligger på adressen 864 så startar din struct på 864-8=856.) Kompilatorn ser automatiskt till att beräkningen blir rätt utifrån de argument som skickas till makrot, men givetvis måste **du** veta vilken struct listelementet du får ut från listan är medlemsvariabel i för att kunna ange rätt argument. Följande kodsekvens plockar ut ett listelement och konverterar:

   ```
   struct list_elem* have = NULL;
   struct string_elem* want = NULL;
   have = list_pop_front(&my_list); /* get and remove from list */
   want = list_entry(have, struct string_elem, elem);
   ```

```
/* ... */

free(want); /* 'want' was allocated in step 3 */
```

6. När du tar bort saker ur listan, var noga med att de är bortlänkade ur listan innan du avallokerar minne. Om du tar bort saker inuti en loop som itererar över listan, var noga med att ta tillvara på resultatet från `list_remove`, nästa iteration måste utgå från det värdet eftersom det borttagna inte är giltigt att använda i `list_next`. Följande kod tar bort vissa element ur en lista:

```
for (e = list_begin(&my_list); e != list_end(&my_list); )
{
  struct string_elem *s;
  s = list_entry (e, struct string_elem, elem);

  if (strcmp(s->string, "to remove") == 0)
  {
    e = list_remove(e); // remove and go to next
    free(s); // needed if 's' was allocated before insertion
  }
  else
  {
    // e must NOT be removed when you do list_next(e)
    e = list_next(e); // keep and go to next
  }
}
```

7. Listan har en funktion för att sätta in i en sorterad lista. Efterågon data (se tidigare steg) så behövs en funktion som avgör om ett element i listan är mindre än ett annat. Den skall ta in de två listelement som skall jämföras och returnera `true` om det första argumentet är mindre än den andra. En tredje parameter som sällan behövs tillåter att eventuell annan data som behövs i jämförelsen kan skickas med (t.ex. om vi vill sortera en lista med koordinater utifrån varje koordinats avstånd till en fix punkt, där den fixa punkten måste skickas med som extra (`aux`) data). En pekare till jämförelsefunktionen skickas med till insättningsfunktionen genom att helt enkelt ange namnet på jämförelsefunktionenfunktionen. Så här ser jämförelsefunktionen ut för listan i detta exempel (`aux` används ej, så den sätts till `NULL` i anropet):

```
bool less(const struct list_elem* a,
          const struct list_elem* b,
          void* aux)
{
  struct string_elem* sa;
  struct string_elem* sb;
  sa = list_entry (a, struct string_elem, elem);
  sb = list_entry (b, struct string_elem, elem);

  return (strcmp(sa->string, sb->string) < 0);
}

list_insert_ordered(&my_list, &to_insert->elem, less, NULL);
```

6.2 Synkroniseringsmekanismer i Pintos

Pintos har ett antal synkroniseringsmekanismer tillgängliga. Dessa är deklarerade i `threads/synch.h` och definierade i `threads/synch.c`. Samtliga funktioner tar emot en pekare till en struct. Minne för structen

måste i vanlig ordning vara skapat innan anrop, antingen genom att som argument ge adressen till en vanlig variabel (*rekommenderas*):

```
struct semaphore binary_semaphore_variable;
sema_init(&binary_semaphore_variable, 1);
```

eller genom att som argument ge en pekare till dynamiskt minne allokerat med `malloc` (måste alltid avallokeras med `free` när det inte behövs mer):

```
struct semaphore* binary_semaphore_pointer = malloc(...);
sema_init(binary_semaphore_pointer, 1);
...
free(binary_semaphore_pointer);
```

Här följer beskrivning av de viktigaste funktionerna som du kan behöva för att göra din kod tråd-säker. Samtliga av dessa är naturligtvis tråd-säkra själva, dvs. de kan anropas ”samtidigt” från flera trådar utan problem.

```
void sema_init (struct semaphore *sema, unsigned value);
```

Används *alltid* för att initiera en nyligen deklarerad eller allokerad semafor *en gång*. Initierar semafor `sema` till att starta med `value` lediga resurser och en tom vänte-kö. Initiering till 1 ger en binär semafor som fungerar som ett lås, men utan felkontroller. Initiering till 0 ger en semafor där ingen resurs finns tillgänglig från början.

```
void sema_down (struct semaphore *sema);
```

Förbrukar *alltid* en resurs från `sema`. Finns ingen resurs tillgänglig så placeras anropande tråd på semaforens vänte-kö tills en resurs blir tillgänglig (tråden som anropar tar alltså paus). När funktionen returnerar är det *garanterat* att tråden fått tillgång till en resurs (att `value` plus antal exekverade `sema_up` minus antalet exekverade `sema_down` är positivt eller noll).

```
void sema_up (struct semaphore *sema);
```

Gör en *alltid* en resurs tillgänglig i `sema`. Om det finns någon tråd på `sema`’s vänte-kön kommer en av dessa att väckas och få tillgång till den nya resursen. Det *garanteras* att denna funktion aldrig kommer att vänta på något. Anropande tråd får fortsätta direkt.

```
void lock_init (struct lock *lck);
```

Initierar ett lås `lck` till att vara ledigt och med tom vänte-kö. Används *alltid* för att initiera ett nyligen deklarerat eller allokerat lås *en gång*.

```
void lock_acquire (struct lock *lck);
```

Om låset `lck` är ledigt markeras det som upptaget av anropande tråd. Om det är upptaget placeras anropande tråd på en vänte-kö tills låset blir ledigt. Det är *garanterat* att låset är markerat som upptaget av anropande tråd när funktionen returnerar. Det är *förbjudet* att anropa denna funktion när anropande tråd *själv* redan markerat låset som upptaget (”håller låset”).

```
void lock_release (struct lock *lck);
```

Markerar låset `lck` som ledigt (”släpper låset”) om exekverande tråd är den som håller låset. Om någon tråd finns på vänte-kön kommer den att väckas och få tillgång till låset. Det är *förbjudet* att anropa denna funktion om anropande tråd *inte* är den som håller låset.

```
void cond_init (struct condition *cond);
```

Initierar `cond` till att ha tom vänte-kö. Används *alltid* för att initiera ett nyligen deklarerat eller allokerat condition *en gång*.

`void cond_wait (struct condition *cond, struct lock *lck);`

Placerar *alltid* anropande tråd på `cond`'s vänte-kö och *släpper* sedan låset `lck`. Används för att vänta på att ett *externt* villkor skall uppfyllas "medan" anropande tråd hållet ett lås. Det är *garanterat* att anropande tråd inte blockerar låset *medan* den ligger på vänte-kön. Det är *garanterat* att anropande tråd "fortfarande" håller låset när funktionen returnerar. Det är **INTE** garanterat att det externa villkoret fortfarande är uppfyllt när funktionen returnerar. Det externa villkoret är något du testar för att avgöra om du behöver vänta på något. Det görs innan du anropar denna funktion.

`void cond_signal (struct condition *cond, struct lock *lck);`

Väcker en tråd från `cond`'s vänte-kö. Om vänte-kön är tom väcks *ingen* tråd. Anropande tråd skall ha gjort så resultatet av det externa villkoret förändrats innan anrop. Det är *förbjudet* att anropa denna funktion utan att hålla låset `lck`. (Låset används normalt för att skydda variabler som ingår i det externa villkoret.)

7 Pintos uppstart

Ett operativsystem startar med något som brukar kallas boot-sekvens. Det är en benämning på allt som händer när operativsystemet startar. Kod för operativsystemest skall läsas från disk och startas, systemets olika moduler skall initieras (trådhantering, minneshantering, diskhantering etc.) och tillgänglig hårdvara skall detekteras och initieras. När du vill lägga till egna datastrukturer eller medlemmar i befintliga datastrukturer är det bra att känna till var och i vilken ordning olika moduler initieras. Du kan till exempel inte allokera minne med `malloc` förrän minneshanteringsmodulen är initierad.

När systemet är initierat är det dags att starta den första processen. Detta sker genom anrop av funktionen `process_execute` som finns i filen `userprog/process.c`. Denna fil innehåller allt som har med processhantering att göra. I Pintos är en process nästan synonym med en kernel-tråd, beroende av att varje process "drivs" av en kernel-tråd. Varje process har alltså exakt en kernel-tråd. Däremot är inte alla kernel-trådar processer. Funktioner för trådhantering finns i filen `threads/thread.c` och motsvarande header-fil. Trådmodulen är den viktigaste delen i Pintos.

7.1 Pintos från start till slut, grov översikt

Detta är huvuddragen av hur Pintos exekverar i ordning från start till slut. Mycket är naturligtvis utelämnat. Pintos huvudprogram (main) finns i `threads/init.c`.

```
thread_init ();  /* initiera trådsystem */
malloc_init ();  /* dynamisk minnesallokering */
timer_init ();   /* generera regelbundna avbrott */
intr_init ();    /* avbrottshanterare */
kbd_init ();     /* tangentbord */
syscall_init (); /* systemanrop */
thread_start (); /* skapa idle-tråd */
disk_init ();    /* diskhantering */

/* starta och vänta på första processen
 * den ny processen läggs på ready-kön
 * kan börja exekvera vilket ögonblick som helst
 * kan också dröja länge innan den startar exekveringen
 */
pid = process_execute (task);

/* vänta på att ovan process blir klar
```

```
 * om detta inte fungerar går Pintos direkt vidare och avslutar
 */
process_wait (pid);

/* om flagga -q användes, stäng av datorn (emulatorn) */
if (power_off_when_done)
    power_off ();

/* avsluta huvudtråden
 * (om poweroff kördes kommer vi naturligtvis inte hit)
 * andra trådar kan fortfarande finnas kvar och köra klart
 * till slut finns endast idle-tråden kvar
 */
thread_exit ();
```

8 Systemanrop i Pintos

Användarprogram kör med begränsad, ”OS-kontrollerad”, tillgång till minne och instruktioner. Ett systemanrop är en begäran från en process, ett användarprogram, till OS att få tillgång till en OS-kontrollerad resurs (filer, minne, etc.).

Ett systemanrop startar alltid i användarprogrammet genom att en instruktion för interrupt exekveras. Pintos tillhandahåller en program-modul som specificerar ett antal systemanrop, och hur de skall gå till (från användarprogrammets kod) i form av vanliga funktioner som användarprogrammet kan anropa. Ett användarprogram får tillgång till dessa genom att inkludera `lib/user/syscall.h`. Implementationen av dessa finns i `lib/user/syscall.c`. Denna implementation konverterar bara det vanliga funktions-anropet till ett systemanrop genom att arrangera user-stacken och ”trigga” ett mjukvaruinterrupt.

Stacken organiseras som för ett vanligt funktionsanrop, men returadressen byts mot ett nummer som identifierar systemanropet som skall utföras. Dvs, först (högst upp i minnet) placeras parametrarna till systemanropet (om några) och därefter ett heltal (istället för återhoppsadress). Systemanropet `exit(111)` skulle till exempel se ut som följer (vi antar att stackpekaren startade på `BFFFFFC0`):

```
BFFFFFBC 111    (int) första parameter
BFFFFFB8 1      (int) systemanropsnummer
```

Symboliska konstanter för systemanropsnummer finns definierade som uppräkningstypen `enum` i `lib/syscall-nr.h`. Detta gör att det går att skriva koden lite tydligare genom att skriva `SYS_HALT` istället för motsvarande heltal i koden.

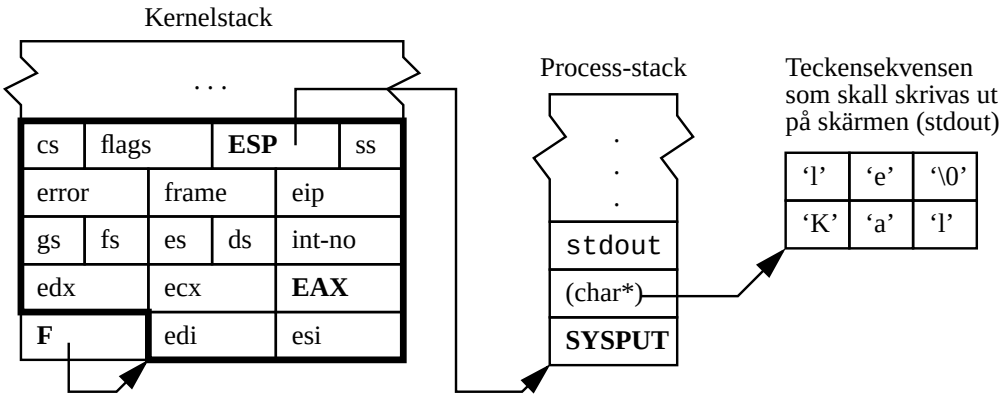
Mjukvaru-interruptet får till följd att processorn (CPU) byter till privilegierat läge (kernel-mode) och samtidigt byter stack så exekveringen i fortsättningen använder kernel-stacken. Nu sparas först alla register på kernel-stacken, så CPU när allt är klart kan återställas till det läge användarprogrammet, processen, hade innan interruptet. Koden för detta finns i `threads/intr-stubs.S` (assembler). Om du är intresserad kan du läsa koden, annars klarar du dig ändå.

När ovan CPU-context är sparat på kernel-stacken anropas en central interrupt-hanterare, `intr-handler`, som återfinns i `threads/interrupt.c`. Denna undersöker var interruptet kom ifrån och anropar en specifik interrupthanterare. En parameter skickas med. Det är en pekare till de data som lagrats på kernel-stacken (processorns register *som de såg ut* precis innan interruptet och *som de kommer återställas* när interruptet är klart). Pekaren är i form av en `struct intr_frame` som beskriver varje data. Det gör access av individuella delar enklare. Strukturen finns beskriven i `threads/interrupt.h`, och det är intressant för dig att titta vad som finns i den. Av speciellt intresse är medlemsvariabeln `esp` som kommer peka på user-stacken, samt `eax` som förväntas innehålla funktioners returvärden.

Den specifika hanterare som anropas vid mjukvaruinterrupt (för att göra systemanrop) heter `syscall_handler` och är definerad i `userprog/syscall.c`. Den nuvarande versionen skriver dock endast ut de två översta talen på user-stacken, samt avslutar tråden. Detta sista gör att systemanropet i nuläget inte kommer att returnera. Ett trådavslut innebär att OS byter till en ny tråd som helt enkelt tar bort tråden som skulle avslutas. Då kan den aldrig fortsätta exekverara. Därmed kommer användarprogramet som gör systemanropet att terminera (för tidigt).

Då du ordnat så `syscall_handler` returnerar med rätt resultat kommer vi så småningom att exekvera hela vägen tillbaka till `threads/intr-stubs.S` som ser till att återställa processorns register och återvända till ”OS-kontrollerat” läge (user-mode). Därmed är systemanropet färdigt.

Stacken vid ett imaginärt systemanrop **SYSPUT** som skriver ut en sträng på `stdout`
som det ser ut precis efter hopp till `syscall_handler(struct intr_frame *F)`



`struct intr_frame` inom fet ram
pekare **F** är parameter till `syscall_handler`

Följande systemanrop beskrivs i mer detalj i avsnitt 10.2.

- `void halt (void) NO_RETURN;`
- `void exit(int status) NO_RETURN;`
- `pid_t exec (const char *command_line);`
- `void sleep(int millis);`
- `void plist (void);`
- `void exit (int status) NO_RETURN;`
- `int wait(pid_t id);`

Följande systemanrop beskrivs i mer detalj i avsnitt 9.2.

- `int open (const char *file);`

- `int read (int fd, void *buffer, unsigned length;)`
- `int write (int fd, const void *buffer, unsigned length);`
- `void close (int fd);`
- `bool remove (const char *file);`
- `bool create (const char *file, unsigned initial_size);`
- `void seek (int fd, unsigned position);`
- `unsigned tell (int fd);`
- `int filesize (int fd);`

8.1 Lägga till ett systemanrop

För att lägga till ett systemanrop i Pintos behöver fyra filer modifieras.

1. Öppna `lib/syscall-nr.h` och lägg till ett symboliskt namn för anropet sist i enum-listan. Det tilldelas nu automatiskt ett nummer vid kompilering.
2. Öppna `lib/user/syscall.h` och lägg till en funktions-deklaration av systemanropet så som användarprogram skall kunna anropa det.
3. Öppna `lib/user/syscall.c` och lägg till en implementation av ovan deklaration genom att använda ett av de fördefinerade makron som finns beroende på antalet parametrar (se hur de andra anropen är gjorda).
4. Öppna `userprog/syscall.c` och lägg till kernel-sidans implementation av systemanropet (precis som förut).

9 Filsystem

Funktioner för att manipulera katalogträdet (listan på filnamn lagrad på disk) återfinns i `filesys/filesys.h`. Funktioner finns för att *lokalisera* (söka, öppna), *skapa* (lägga till), och *ta bort* filer i katalogträdet. Pintos implementation av filsystem är begränsad till max 16 filer, inga kataloger, och korta filnamn utan konstiga tecken. Filstorleken går inte att utöka, utan måste anges då filen skapas.

Funktioner för att hantera filer som redan är öppnade återfinns i `filesys/file.h`. Funktioner finns för att *läsa*, *skriva*, *ta reda på läs/skrivposition*, *ange läs/skriv position*, *ta reda på filens storlek*, samt *stänga* filen och frigöra de resurser Pintos allokerat för filen i fråga.

Borttagning av filer hanteras speciellt i den befintliga implementationen. När filen tas bort försvinner den direkt ur katalogen, så den inte går att öppna mer, men avallokering av diskblock sker inte direkt, utan först när alla som använder samma fil stängt sin fil. Den sista som stänger kommer att markera filens block som lediga igen. Detta gör att borttagning av filer som är öppna och används av andra processer sker korrekt.

Alla filer som öppnats av en viss process skall *alltid* stängas då *den* och *endast den* processen avslutar. Detta för att undvika minnesläckage i operativsystemet. En process kan avsluta genom att returnera från main, via systemanropet exit, eller genom att Pintos dödar processen till följd av något programmeringsfel (i processen). I alla tre fall skall kvarvarande öppna filer stängas.

9.1 Struktur

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 sidan 32. 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 beskrivs under ”Installation”.

Nedan följer en beskrivning av de olika modulerna som finns i filsystemet, och vilka abstraktioner de implementerar. Att förstå dessa abstraktioner underlättar arbetet med att synkronisera filsystemet oerhört.

Disk: `disk.h` och `disk.c`

> Här finns funktioner för att läsa och skriva sektorer från disken. Läsning och skrivning till enskilda sektorer är **synkroniserad**. Denna synkronisering hindrar att en tråd använder disken innan föregående tråd är helt klar med sin operation.
>
> Denna modul tillhandahåller alltså abstraktionen att en disk är en samling sektorer som resten av systemet kan läsa från och skriva till.

Free map: `free_map.h` och `free_map.c`

> Denna modul implementerar en så kallad bitmap som håller reda på vilka sektorer på disken som är lediga och vilka som är upptagna. Denna information lagras i en inod på disken, och läses in i RAM när disken monteras (vilket sker när Pintos startar). När innehållet i bitmappen har ändrats så skrivs den också till disk, så att det alltid finns en uppdaterad version på disk ifall Pintos kraschar. I och med att Pintos bara har stöd för att montera en disk så lagras bitmappen globalt.
>
> Implementationen av bitmapen (i `bitmap.c`) garanterar att ändring av enskilda bitar är atomära. Däremot är operationer som involverar mer än en bit inte atomära. Fungerar implementationen ändå korrekt utan ytterligare synkronisering?
>
> Denna modul håller alltså koll på vilka delar av disken som används, och vilka som är lediga. Den tillåter därmed att andra delar av systemet reserverar plats med funktionen `free_map_allocate`, och frigör plats med `free_map_release`.

Inode: `inode.h` och `inode.c`

> Denna modul tillhandahåller inoder som kan lagras på disken. En inod är en behållare för godtycklig binär data av en fördefinierad längd. Man kan alltså tänka på en inod som en fil utan namn. För att referera till en inod används i stället ett sektornummer. Man kan därmed be systemet att öppna inoden som finns i sektor nummer fem till exempel. Free-map ovan lagras exempelvis i inoden som finns i sektor nummer 0 (see `src/filesys/filesys.h`).
>
> Likt filer så innehåller `inode.h` funktioner för att öppna inoder (baserat på sektornummer), stänga inoder, läsa från dem, och skriva till dem. Utöver det finns funktioner för att skapa och ta bort inoder från disk. Värt att notera är att datatypen `struct inode` är deklarerad i `.c`-filen, och den går därför bara att använda i `inode.c`. Detta är ett medvetet val för att göra det enklare att resonera kring koden. I och med detta vet vi att all kod som modifierar medlemmar i `struct inode` måste finnas i `inode.c`. Vi kan alltså i stor utsträckning behandla `inode.c` som en isolerad modul när vi funderar kring synkronisering.
>
> En intressant detalj är att inode-modulen ser till att varje inode bara finns i RAM en gång. Det vill säga, om `inode_open(1)` anropas mer än en gång, så kommer båda anropen att returnera en pekare till samma `struct inode` (givet att ingen stängde inoden mellan anropen). Detta gör det möjligt att hålla reda på hur många som just nu har en viss inod öppen, och gör det möjligt att synkronisera åtkomst till data på disk. Det kräver dock en del logik i `inode_open`.
>
> Ingen del av koden i denna modul är synkroniserad i den givna koden. Det är därmed en bra idé att synkronisera denna modul så att den fungerar som den ska oavsett hur många trådar som använder

den samtidigt. De flesta andra moduler i filsystemet beror på inode-modulen, så om vi kan anta att inode-modulen är synkroniserad kan vi enklare resonera om beteendet i resten av modulerna.

Directory: `directory.h` och `directory.c`

Directory-model innehåller funktioner för att spara en lista av filer inuti en inod. Detta gör vi så att vi kan ge namn till våra inoder, så att program i sin tur kan öppna filer genom att ange ett namn i stället för ett sektornummer.

Implementationen i Pintos har bara stöd för en katalog. Katalogen kan därmed bara innehålla filer. Katalogen representeras som en array av `struct dir_entry`. Dessa lagras inte i RAM, utan de lagras i inoden som finns i sektor nummer 1. Alltså, för att öppna en fil med namnet `program`, så måste directory-modulen först anropas för att slå upp vilken sektor filen lagras i. För att göra det öppnar directory-model inoden i sektor 1 och itererar igenom de element som finns där. Om den hittar ett element med namnet `program`, så finns i elementet vilken sektor som filens inod finns i. Om filen hittades kan alltså sedan rätt inod öppnas genom att anropa inode-modulen.

En intressant observation från processen ovan är att directory-modulen inte behöver komma åt disken direkt, utan den arbetar helt och hållet med inode-abstraktionen för att lagra data på disk.

Likt inode-modulen är denna modul inte synkroniserad i den givna koden. Vad händer om flera filer läggs till samtidigt? Behöver implementationen synkroniseras även om den bara använder inode-modulen som vi kommer synkronisera?

File: `file.h` och `file.c`

Denna modul introducerar abstraktionen `struct file`, som representerar en öppnad fil. En öppnad fil består helt enkelt av en pekare till den inode som innehåller filens data, och ett heltal som håller koll på positionen i filen. Positionen behövs exempelvis så att man ska kunna anropa `file_read` flera gånger i rad och få olika data varje gång.

I och med att vi *inte* vill att varje process som har öppnat filen ska ha sin egen pekare till filens position så kommer `file_open` att skapa en ny `struct file` varje gång den anropas, även om flera processer öppnar samma fil. Detta skiljer sig alltså från hur `inode_open` fungerar. Det har också stor inverkan på vad som behöver synkroniseras i den här modulen.

I övrigt så använder denna modulen uteslutande funktionalitet från inode-modulen.

Det finns ingen uttrycklig synkronisering i den här modulen. Kommer `struct file`-instanser att vara delade mellan processer? Vems uppgift är det att se till så att inte filpositionen ändras på felaktiga sätt ifall en `struct file` delas? Behöver implementationen synkroniseras även om den bara använder inode-modulen som vi kommer synkronisera?

Filesys: `filesys.h` och `filesys.c`

Denna modul introducerar högnivåfunktioner för att skapa filer, ta bort filer, och öppna filer via namn. Den sköter alltså allt som har med namn på filer att göra, och sköter interaktionen med directory-modulen.

Modulen använder alltså huvudsakligen directory-modulen, men även enstaka funktioner från inode-modulen och file-modulen.

Det finns ingen uttrycklig synkronisering i den här modulen. Kommer synkroniseringen från resterande moduler att räcka, eller behövs mer synkronisering?

9.2 Systemanrop

Systemanropen för filhantering är deklarerade i `lib/user/syscall.h` och används av användarprogrammen därefter:

- `int open (const char *file);`

  Skall söka i katalogträdet efter en fil med namnet angivet i `file`. Om filen finns hämtas referensen till filen (`struct inode*`) och lagras i systemets inodelista. Om filen redan öppnats av någon process hämtas referens till filen direkt från inodelistan. Referensen till filen (`struct inode*`) lagras tillsammans med information om läs och skrivposition i `struct file*` för snabb och effektiv åtkomst vid senare läsningar eller skrivningar till filen. Notera att det kan komma skapas många öppna filer av typ `struct file*` för varje filreferens av typ `struct inode*`. *Läs och förstå var det som beskrivs ovan sker i den givna koden innan du skriver kod för delar som redan är implementerade.*

  För att dölja all kernel-data från användar-programmet knyts informationen (`struct file*`) till ett heltal som benämns fildeskriptor (`fd`). Det är detta heltal som returneras till användarprogrammet. Minus ett (-1) returneras om angiven fil inte finns. Några fd är reserverade för tangentbord och skärm. Du måste hålla reda på vilken fil som är knuten till vilken fildeskriptor för varje filöppning, samt vilken process som gjorde filöppningen. Alla `fd` en process fått via anrop av `open` måste till slut stängas oavsett om processen anropar `close` eller avslutar utan anrop av `close`, oavsett hur processen avslutas.

- `int read (int fd, void *buffer, unsigned length);`

  Läs `length` tecken från resursen `fd` och lagra i `buffer`. Returnera antal lästa tecken. `fd` kan vara `STDIN_FILENO`(tangentbord) eller en `fd` som processen tidigare fått i retur från `open`, men ännu inte skickat som argument till `close`. `STDOUT_FILENO` är ett ogiltigt `fd` vid anrop till `read`.

- `int write (int fd, const void *buffer, unsigned length);`

  Skriv `length` tecken lästa från `buffer` till resursen `fd`. Returnera antal skrivna tecken. `fd` kan vara `STDOUT_FILENO` (skärm) eller en `fd` som processen tidigare fått i retur från `open`, men ännu inte skickat som argument till `close`. `STDIN_FILENO` är ett ogiltigt `fd` vid anrop till `write`.

- `void close (int fd);`

  Stäng (avallokera kernelresurser) den fil som identifieras av `fd`. Minne för `struct file*` och ibland `struct inode*` skall återlämnas. Studera hur detta redan fungerar. Ett giltigt `fd` måste ha erhållits som returvärde vid tidigare anrop av `open` och ännu inte skickats som argument till `close`. Alla `fd` en process fått via anrop av `open` måste till slut stängas oavsett om processen anropar `close` eller avslutar utan anrop av `close`, oavsett hur processen avslutar. En process får bara lov att stänga filer som den själv har öppnat.

- `bool remove (const char *file);`

  Sök i katalogträdet efter en fil med namnet angivet i `file`. Om sådan finns raderas namnet och referensen till filen i katalogträdet, men inga diskblock avallokeras förrän alla som har filen öppen har stängt filen. *Ovan löses i befintliga funktioner.* Returnerar `true` om filnamnet raderades från katalogträdet.

- `bool create (const char *file, unsigned initial_size);`

  Skall reservera diskblock för en fil med storlek `initial_size` och lägga till en referens till filen i katalogträdet under namnet angivet i `file`. Returnera `true` om operationen lyckas.

- `void seek (int fd, unsigned position);`

  Anger (byte) `position` var i filen `fd` nästa läsning eller skrivning skall ske. Fildeskriptorn `fd` måste ange en öppen fil och `position` måste ange en position i filen.

- `unsigned tell (int fd);`

  Returnerar (byte) `position` var i filen `fd` nästa läsning eller skrivning skall ske. Fildeskriptorn `fd` måste ange en öppen fil. Det är OK om du returnerar -1 när fel uppstår trots att returvärdet är unsigned.

- `int filesize (int fd);`

  Returnerar storleken på filen identifierad av `fd`. Fildeskriptorn `fd` måste ange en öppen fil. Returvärdet är -1 om begäran inte kan utföras.

9.3 Read och Write

Systemanropet `read` används för att läsa från både tangentbortet och från en öppen fil. Systemanropet `write` används både för att skriva till skärmen och till en öppen fil. För att avgöra var läsning/skrivning skall ske används en fildeskriptor, `fd` (första parametern). Det är ett heltal som representerar en öppen fil i kernel. För skärm och tangentbord reserveras två speciella `fd`. De är definerade enligt föjlande i filen `lib/stdio.h`

```
#define STDIN_FILENO 0
#define STDOUT_FILENO 1
```

Om `fd` är lika med `STDOUT_FILENO` skall anrop av `write` skriva till skärmen. Till detta kan du använda funktionen `putbuf` som finns deklarerad i `lib/kernel/stdio.h` och implementerad i `lib/kernel/console.c`. Om `STDOUT_FILENO` anges vid anrop av `read` hanteras det som ett fel genom att returnera felkoden minus ett (-1). Notera att `printf` inte är lämplig att använda i det här fallet i och med att den arbetar med nullterminerade strängar och inte en minnesbuffer[1] med fix storlek. Om `STDOUT_FILENO` anges vid anrop av `read` hanteras det som ett fel genom att returnera felkoden minus ett (-1).

Om `fd` är lika med `STDIN_FILENO` skall anrop av `read` alltså läsa från tangentbordet. Till detta kan du använda funktionen `input_getc` som läser ett tecken från tangentbordet. Den finns deklarerad i `devices/input.h` och är implementerad i motsvarande c-fil. Tänk på att användaren normalt brukar vilja se varje inmatat tecken på skärmen medan denne skriver. Om `STDIN_FILENO` anges vid anrop av `write` hanteras det som ett fel genom att returnera felkoden minus ett (-1).

Systemanropet read skall placera de lästa tecknen i buffern tills den är full. Det är viktigt att aldrig skriva fler tecken till buffern än storleken anger (så klart). Systemanropet write fungerar tvärtom, läser tecken att skriva ut från angiven buffer. Det är viktigt att skriva ut exakt alla tecken i buffern (så klart). Både read och write returnerar antalet behandlade tecken. Ofta är returvärdet för read och write identiskt med tredje inparametern, om inte något fel uppstod.

Att läsa inmatning från användaren brukar vara lite speciellt. Normalt kan användaren skriva indata till programmmet, och ångra sig och ändra så länge inmatningen inte bekräftats med ”Return” eller ”Enter”. Först när bekräftelsen kommer får programmet tillgång till den rad som matats in. Dessutom brukar programmet då inte vänta på fler tecken (för att fylla hela buffern) utan nöja sig med det som matats in. Slutligen krävs det i C att text-stängar avslutas med ett noll-tecken. Tillsammans skulle detta innebära att tredje parametern anger hur många tecken i buffern som `maximalt` får användas, inklusive avslutande noll-tecken (`'\0'`). I Pintos antar vi att dessa speciella detaljer tas om hand av en getline-funktion i ett programbibliotek, och inte av operativsystemets systemanrop. **Systemanropet skall alltså läsa *exakt* så många tecken som anges av andra parametern. Avslutande noll-tecken är inte heller systemanropets bekymmer i vårt fall.**

I Pintos finns ytterligare en speciell sak. Denna skall systemanropet lösa. Om användaren matar in ett carriage-return tecken (`'\r'`) skall detta betraktas som, och ersättas med, ett nyradstecken (`'\n'`). Detta för att ”Enter” eller ”Return” i Pintos endast ger carriage-return tecken, medan vi när vi programmerar i UNIX normalt förväntar oss ett nyradstecken i detta läge.

[1] En ”buffer” är en array med en viss storlek att lagra eller hämta data ur. Oftast representerad som en pekare och en storlek i C.

9.4 Readers-writers lock

9.4.1 Bakgrund

När en process läser eller skriver till en fil samtidigt som en annan process skriver till samma fil, eller när en process skriver till en fil samtidigt som en annan process läser från samma fil, så kan det uppstå fel genom att processen som läser kanske läser en del av filen som den såg ut innan den ändrades, och en annan del som den såg ut efter ändring. Det kan också hända att ändringarna, om bägge processerna skriver till filen, blandas ihop om vartannat. Naturligtvis blir det ännu mer fel om fler än två processer är inblandade.

Det är inte acceptabelt att det kan bli fel. En process som läser från en fil medan en annan process skriver skall se all data som läses i ett systemanrop antingen helt och hållet som det såg ut innan uppdatering, eller helt och hållet så som det ser ut efter uppdatering. Vid skrivning gäller att allt som skrivs inom ett systemanrop skall lagras på filen i sin helhet. Det skall finnas en (kort) tidsperiod precis efter systemanropet då allt som skrivits syns i filen, även om någon samtidigt försöker skriva över vissa delar.

Det som speciellt kan observeras är att det inte kan bli några fel om två eller flera processer läser från en fil samtidigt (så länge ingen skriver till filen). Det är alltså möjligt att låta läsning ske samtidigt, parallellt. Detta kan öka prestanda genom att undvika att processer stoppar för att vänta på varandra, speciellt i situationer när en fil uppdateras sällan, men läses ofta (sidor på en webbserver t.ex.).

9.4.2 Liknelse för tillåtna fall

Det kan vara lite svårt att se hur skrivning skall fungera så här kommer en liknelse.

Tänk dig ett tomt litet bord som rymmer ungefär 4 papper i storlek A4 (om de läggs kant i kant). Ett dussin personer runt bordet håller var och en i ett papper i storlek A4. Varje papper har olika färg. Deras uppgift är att placera papperet någonstans (var och hur som helst) på bordet vid given signal. Hur ser bordet ut när alla är klara? Jo, den sista personens papper kommer alltid synas i sin helhet. Tar vi bort det papperet kommer den näst sista personens papper att synas i sin helhet. Och så vidare. *Detta är rätt resultat. Den senaste operationen skall alltid synas i sin helhet.*

Tänk dig nu att varje persons papper är delat i 8 delar. De har samma uppgift: hela papperet, alla 8 delar, skall placeras på bordet vid given signal. Hur ser bordet ut när alla är klara? Jo, förmodligen kommer inga av de först placerade pappersdelarna att synas, utan endast de sist placerade delarna av varje papper. Alla papper kommer ha någon del som ligger under de sista delarna från ett annat papper. Detta är fel resultat. Det som syns överst på bordet kommer vara en blandning av pappersdelar från olika personer, inget papper kommer synas i sin helhet. Det är inconsistency.

Rätt resultat får man när de 8 delarna behandlas som om de fortfarande satt ihop i ett papper, d.v.s. ingen får placera någon av sina delar medan någon annan håller på placera sina.

Operationen att lägga hela sitt papper (alla delar) på bordet motsvaras av att anropa systemanropet *write*. Buffern som skickas med *write* är papperet (alla delar) och det som ligger på bordet är innehållet i filen. I ett systemanrop kan ”papperet” vara uppdelat i väldigt många delar. Trots det skall det senaste ”papperet” alltid synas i sin helhet när filen läses, eller när den skrives. Filen skall alltså vid alla tidpunkter se ut som om *alla* haft *hela* papper när de placerade papperet på bordet.

9.4.3 Liknelse för algoritm

Föreställ dig en offentlig toalett med en toalettstol och en (oändligt lång) pissoar. Rummet har en ingång, men dörrlåset är tyvärr trasigt, så dörren går inte att låsa. Istället har någon satt upp en skylt med ordet ”ledigt” på ena sidan, och ordet upptaget på andra sidan. Skylten går att vända. Det finns även en tavla på dörren där man antingen kan dra ett streck, eller sudda ett streck. Något annat kan inte ritas på tavlan.

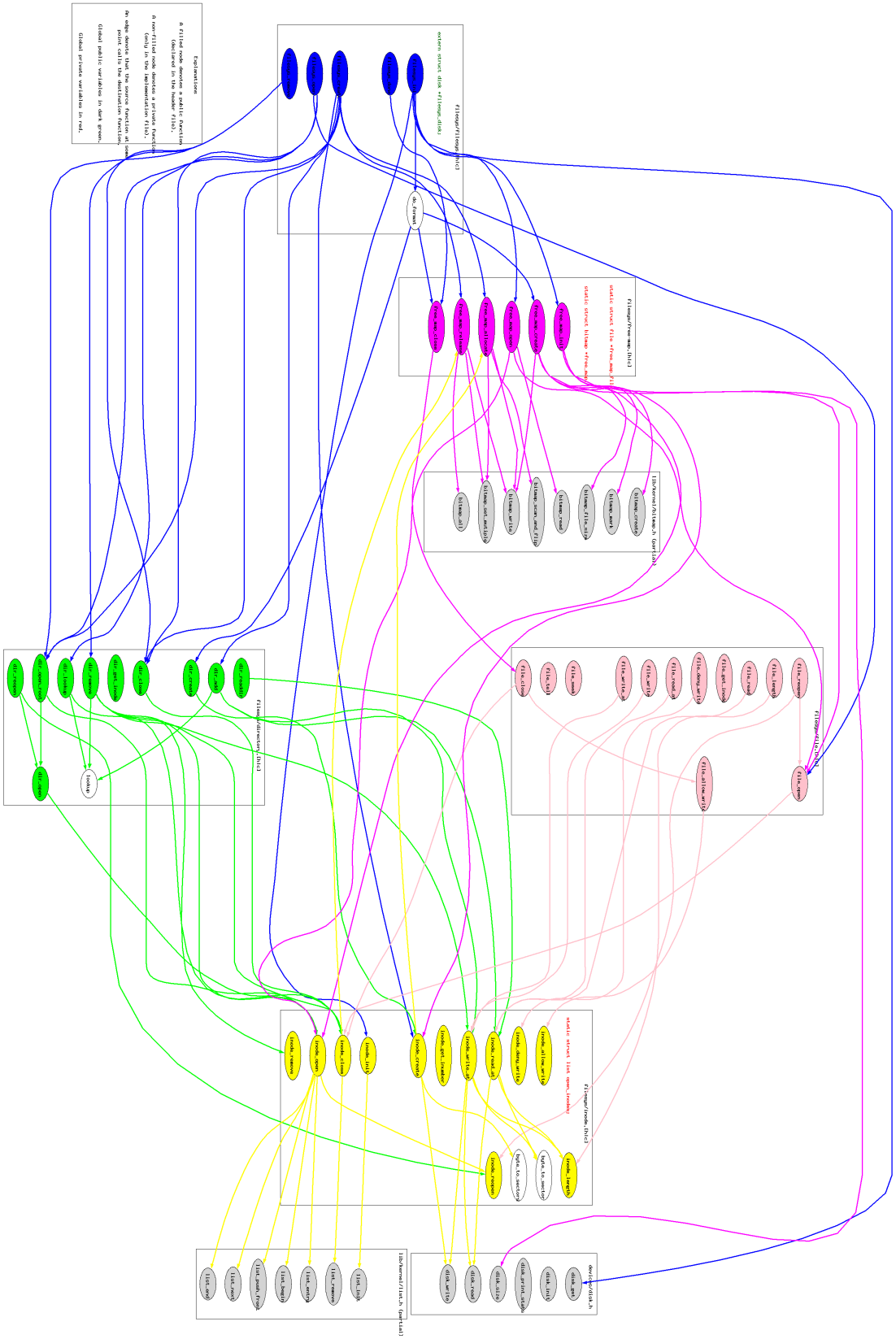
Nu är det så att herrar endast behöver nyttja pissoaren (bortse från andra behov), och damer endast behöver toalettstolen. Tiden besöket tar varierar slumpmässigt för olika personer. Kommer flera in samtidigt är det

alltså inte nödvändigtvis den som är först in som är först (eller sist) ut. En allvarlig begränsning är att ingen kommunikation mellan olika personer kan förekomma, annat än genom skylten och tavlan.

Att åstadkomma enskild tillgång till rummet för var och en är enkelt. Vilket protokoll skall alla följa då? Jo, titta först på skylten. Ledigt? Vänd skylten och stig in. Upptaget? Vänta på att skylten visar ledigt. När du går ut vänder du skylten till ledigt. Detta protokoll är enkelt men skapar lätt en kö om många behöver nyttja faciliteten.

Utgå från att det är möjligt att flera herrar nyttjar rummet samtidigt, medan damer skall få enskild tillgång till rummet. Din uppgift är att uppfinna ett protokoll för att tillåta flera herrar samtidigt.

Det som är intressant då är hur herrar respektive damer skall bete sig när de kommer och när de går. Mest intressant är kanske hur skylten och tavlan hanteras av herrarna, så att skylten alltid visar ledigt när rummet är tomt. Vem vänder fram ”upptaget”-sidan? Vem vänder tillbaka ”ledigt”-sidan? Om det är upptaget, hur vet ankommande herrar om de skall vänta eller stiga på?



10 Tråd- och processhantering i Pintos

Denna sektion beskriver hur trådar och processer hanteras i pintos, hur de startas, hur de avslutas och vilka systemanrop som berör dem.

10.1 Interna funktioner

Följande funktioner för tråd- och processhantering är centrala i Pintos. Du kommer behöva ändra alla process-funktioner någon gång. Funktionernas ansvarsfördelning som du förväntas följa:

`process_execute()`

Skapar en tråd med uppgift att läsa in och starta exekveringen av en ny process. Ser till att all data associerad med en process är korrekt uppsatt. Returnerar ett unikt id som identifierar processen i framtiden *men endast om tråden lyckades starta den nya processen.* Annars returneras -1.

`start_process()`

En hjälpfunktion till `process_execute` som gör större delen av jobbet i en egen tråd. Läser in processen från disk, allokerar all information relevant för hantering och spårning av en process, sätter upp processens initiala stack och startar processen i user-mode. Det är här vi upptäcker om en process kan startas eller ej.

`process_cleanup()`

Avallokerar en process alla resurser som kernel reserverade under exekveringen av `start_process` och `process_execute`, samt resurser som allokerats via systemanrop. *Denna funktion anropas automatiskt av thread_exit så att trådar som är processer kan avallokera sina data.* Denna funktion i sig avslutar ingen tråd eller process, den avallokerar bara data, såsom processens minnesrymd. Följaktligen är det `thread_exit` som måste anropas för att stoppa exekveringen av en process.

`process_wait()`

Denna funktion tar emot ett id för en skapad process och pausar exekvering av anropande tråd tills processen med det id-numret har avslutat. Om Pintos startas med flaggan -q är det av central betydelse att denna funktion fungerar korrekt (eller åtminstone hjälpligt) eftersom den används för att vänta på att första processen skall exekvera färdigt. Om den returnerar innan första processen avslutat kommer Pintos avslutas, och när flaggan -q används emulatorn att stängas av, med påföljd att de processer man tänkt sig köra kanske inte hinner bli klara eller ens starta.

`process_exit()`

Denna funktion är inte implementerad. Eftersom varken `thread_exit` eller `process_cleanup` kan ta emot en `exit_status` (trådar har ingen exit-status) är det tänkt att denna funktion implementeras av dig för att göra det som behövs för att ta emot `exit_status` och avsluta processen. Läs om `thread_exit` och övriga funktioner som beskrivs här innan du skriver någon kod.

`thread_create()`

Skapar en kernel-tråd som dediceras till att exekvera en viss funktion (i kernel-mode). Allt som har med processer att göra lämnas till funktionerna i filen `process.c`. Kan alltså användas för att skapa generella trådar, men skapar inga processer. Observera särskilt att denna funktion endast skapar tråden genom att allokera en stack, en trådstruktur och placera den i operativsystemets kö över trådar som ar redo att exekvera. Exakt när tråden börjar exekvera är odefinerat, och kan ske när som helst efter skapandet - ibland omedelbart, ibland efter lång tid.

`thread_current()`

Returnera en pekare till trådinformation (se `threads/thread.h`) om den tråd som exekverar just nu. *Denna behöver du använda ofta.*

Det finns även funktioner som direkt tar fram specifika delar ur den struct som representerar en tråd, men via denna funktion kan du komma åt allt. *Titta vilken information som finns i thread.h.*

`thread_exit()`

Anropas av en tråd som vill avsluta sig själv. Denna funktion returnerar inte, utan sätter en terminate-flagga och byter till nästa tråd, varpå nästa tråd raderar den avslutande tråden från systemet. (En tråd kan inte ta bort sig själv. Trådens data behövs när operativsystemed växlar till en ny tråd. Den kan inte såga av grenen den sitter på.)

10.2 Systemanrop

För att få ett praktiskt användbart operativsystem behövs stöd för att starta flera program. Då behövs minst två systemanrop för att hantera processer. En process måste kunna avsluta, och en process måste kunna starta nya processer. Dessutom behöver operativsystemet ett sätt att hålla reda på aktiva processer och visa dem för användaren (jämför aktivitetshanteraren i Windows eller skalkommandot `top` i Linux).

Systemanropen ska vara deklarerade i `lib/user/syscall.h` och används av användar-programmen därefter:

- `void halt (void) NO_RETURN;`

  Skall stänga av *datorn* (emulatorn) med omedelbar effekt. Processen kommer inte fortsätta sin exekvering eftersom datorn stängs av (därav är funktionen markerad med `NO_RETURN`).

- `void exit(int status) NO_RETURN;`

  Avsluta processen. Processen skall inte fortsätta sin exekvering (därav är funktionen markerad med `NO_RETURN`). Om processen lyckades med det den programmerades för (process-specifikt) anropas `exit` traditionellt med `status` satt till 0. Misslyckas något väljer programmeraren av processen traditionellt något annat värde på `status` (processspecifikt) för att kunna särskilja olika fel. Värdet på `status` är intressant främst för den som startade processen (förälder), och en mekanism för föräldern att ta reda på `status` skall finnas. Normalt skickas returvärdet från `main` in som `status` (se ”Parameteröverföring vid funktionsanrop”).

Exit ser även till att processer som inte längre behövs i processlistan rensas bort så att nya processer får plats. Eventuellt allokerat minne måste frigöras. Detta implementeras i `process_cleanup` så att det utförs vid alla anrop som leder till att en tråd avslutas. Avslut kan ske av kernel när något allvarligt fel uppstår i en process, och inte bara av systemanrop.

För att förbereda för enkel delning av data mellan en process och dess förälder skall en process tas bort ur listan exakt när `både` processen *och* dess förälder har avslutat. En process kan alltså finnas kvar i listan ända tills även dess förälder avslutat (*OBS! även om barn till processen fortfarande exekverar kan processen tas bort ur listan. Processen delar endast sina data med sin förälder, ej med sina barn!*). Detta tillväga-gångssätt garanterar att processens plats i listan alltid är giltigt oavsett om processen eller föräldern försöker använda den. Tänk noga igenom vilka processer som kan tas bort och när det skall göras.

Parametern `status` kan du som förut bara skriva ut, alternativt spara värdet med övrig information på processens plats i listan.

```
pid_t exec (const char *command_line);
```

Anropar endast `process_execute`. Men nu skall den nya processen läggas till i en lista över aktiva processer. Ett lämpligt process-id är nu processens index i listan, men det går att lösa på många sätt (valfritt). För att kunna hålla reda på relationen mellan processer behöver varje process hålla reda på sitt namn, sitt eget process-id, sin förälders process-id, samt eventuell extra data som en process behöver skicka tillbaka till sin förälderprocess eller tvärtom (jämför med vad som görs i uppgiften ”Den första processen”).

```
void sleep(int millis);
```

Anrop av denna funktion ”pausar” processen i ett antal millisekunder. Funktioner för att låta en tråd ”sova” i Pintos finns i `devices/timer.[h/c]`. Den befintliga implementationen av `timer_sleep` använder en implementation som hela tiden byter till en annan tråd. Detta klassas som ”busy wait”. En bra implementation placerar processen i en väntekö under hela denna tid så processen inte förbrukar onödig datorkraft på att hela tiden kontrollera om det är dags att fortsätta. Det räcker att du implementerar systemanropet utan att rätta till ”busy-wait”.

`void plist (void);`

Skriver ut hela listan med processinformation (id, namn, förälder-id, exit-status) på ett snyggt och städat format. Denna funktion kan vara bra att ha i debug-syfte, så gör utskriften tydlig och komplett.

`int wait(pid_t id);`

Tar emot ett process-id och kontrollerar att det är en barnprocess till processen som kör. Väntar därefter tills sagda barn-process har avslutat och returnerar dess `status` (se systemanropet `exit`). Avlutningsvis ska barn-processen tas bort från listan över aktiva processer eftersom den inte behövs mer. Det skall endast gå att vänta på en process en gång. Systemanropet returnerar -1 om det inte finns någon barn-process `id` som föräldern inte redan har anropat `wait` på.

10.3 Starta en process

Funktionerna `process_execute` och `start_process` är ansvariga för att starta en ny process. Den initiala versionen av dessa funktioner har flera problem och flera ”hack” för att det ändå skall fungera hjälpligt. Följande beskriver hur implementationen *borde* (skall) fungera. Det är ganska nära hur det faktiskt fungerar, men inte riktigt.

Detta beskriver vad `process_execute` skall göra. Fetstilat är sådant som inte ännu sker:

1. En kommandorad tas emot som parameter
2. En tråd skapas (placeras på ready-kön) med uppgift att exekvera den nya processen. Som första argument anges ett namn på tråden, detta är första ordet på kommandoraden.Andra argument är trådens prioritet, `PRI_DEFAULT`.Tredje argument är en pekare till den funktion tråden skall exekvera.Fjärde argument är den pekare som skickas med när tråden startar funktionen i tredje argument.Fjärde argumentet är lämpligen en pekare till en struct som kan användas för att kommunicera olika data mellan föräldertråden och den nya tråden eller vice versa.
3. **Vänta på att tråden skall nå punkt e) nedan**. Tråden skall starta, läsa in processen (`load`), och göra den klar för start. Dessa tre steg kan gå galet på flera sätt, t.ex. att programfilen är korrupt eller att minnet tar slut. Resultatet hanteras i nästa punkt.
4. **Ta emot resultatet från nya tråden.** Om allt gick bra skall ett nummer som unikt identifierar processen returneras. **Gick något galet skall -1 returneras**. (-1 kan således inte användas för att identifiera en giltig process.)

Detta beskriver vad `start_process` skall göra. Fetstilat är sådant som inte ännu sker korrekt. Under varje punkt visas relevant rad kod som referens (om det finns implementerat):

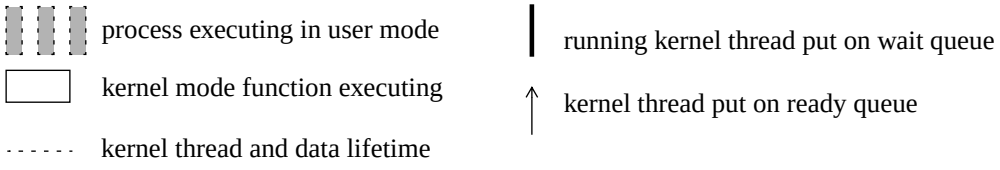
a. Ta emot pekare till data från `process_execute( struct process_arguments* pa = (struct process_arguments*) )`
b. Extrahera namnet på programmet som skall köras `( strlcpy_first_word (file_name, pa->command_line, 64); )`
c. Läsa in programmet och allokera nödvändigt minne och stack `( success = load (file_name, &if_.eip, &if_.esp); )`
d. Placera kommandoraden på processens stack som parametrar till main: `if_.esp = setup_main_stack_asm(...)`

e. **Skicka resultatet av punkter a) - d) (allt gick bra eller något gick galet)** till punkt 4) ovan

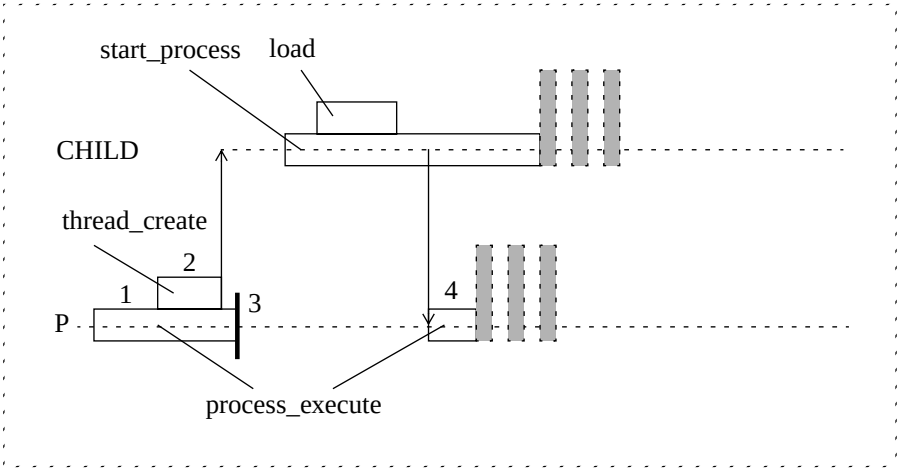
f. Starta exekveringen av processen (läs kommentaren i koden, ej viktigt att förstå exakt)`( asm volatile (̈movl %0, %%esp; jmp intr_exit¨ : g̈(&if_) : m̈emory); )`

I `process_execute` utför nuvarande lösning inte hela punkt 3) och 4), istället stängs datorn av. I punkt 4) returneras just nu alltid att allt gick bra, men när något av stegen i 3) går galet skall det synas i punkt 4). Dessutom finns ett ”race” eftersom kommandoraden kan hinna bli ogiltig innan `start_process` (som använder kommandoraden) exekverar (kommentera ut raden som gör `power_off` och se vad som händer). En korrekt lösning låter `process_execute` vänta `precis` tills `start_process` är klar med kommandoraden *och* har utfört punkt e). Lösningen i `start_process` utför inte punkt e) alls, men resterande punkter är lösta.

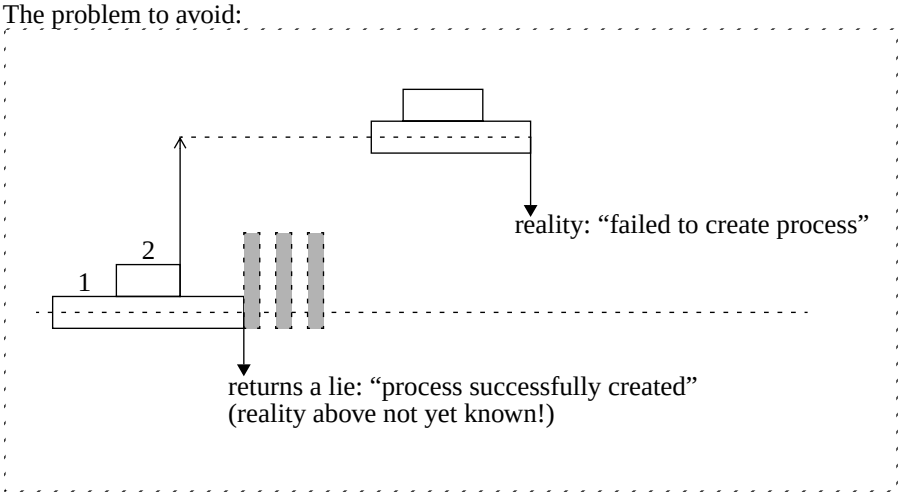
Figuren nedan sammanfattar problemet grafiskt.



How it shall work:



The problem to avoid:



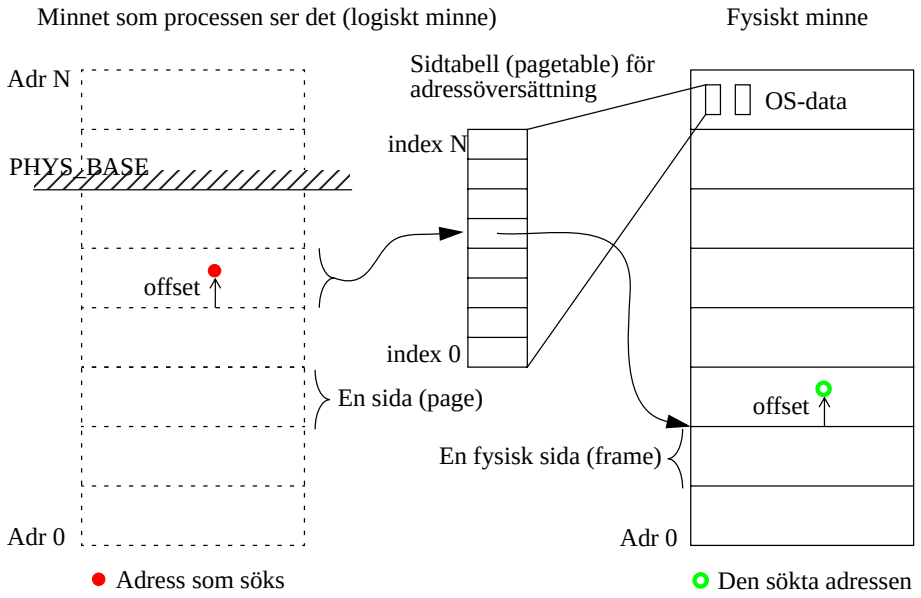
11 Minne

Som du vet så tilldelar operativsystemet en egen minnesrymd till varje användarprogram. Detta är för att ett program inte skall kunna mixtra och trixa med minne som tillhör andra program eller rent av tillhör

operativsystemet. Det är operativsystemets uppgift att säkra systemet så inga möjligheter finns att kringgå minnesskyddet. Det är mycket viktigt att operativsystemet utför kontroller så att användarprogrammen inte förstör för varandra, eller ännu värre, förstör för operativsystemet.

11.1 Minnesaddressering (Paging)

De flesta operativsystem, t.ex. Pintos, använder *paging* för att tilldela minne till processer på ett flexibelt sätt. Då ett program har placerats i minnet och börjat exekvera kallas det enligt gängse terminologi för en process, en term som används i fortsättningen. Med minnesmodellen *paging* översätter CPU alla adresser processen specificerar (user-adresser i processens logiska minnesrymd) till riktiga adresser (kernel-adresser i systemets fysiska minnesrymd). Detta görs genom att OS/CPU delar in allt minne i en uppsättning lika stora sidor (pages). Genom att titta på vilken sida en logisk adress finns i kan motsvarande frame i fysiskt minne slås upp i en tabell, och genom att titta på hur långt från starten på sidan adressen finns (offset) kan rätt adress inom den fysiska framen enkelt räknas fram.



Givet en adress USER_ADR i processen kan den fysiska adressen PHYS_ADR räknas ut enligt:

```
PAGE_NUMBER = USER_ADR / PAGE_SIZE; // heltalsdivision
PAGE_OFFSET = USER_ADR % PAGE_SIZE; // resten vid divisionen
FRAME_NUMBER = PAGETABLE[PAGE_NUMBER];
PHYS_ADR = FRAME_NUMBER + PAGE_OFFSET;
```

Dessa beräkningar är särskilt enkla och effektiva då sidstorleken är en heltalsmultipel av talsystemets bas eftersom man vid divisionen och rest-beräkningen helt enkelt kan stryka respektive behålla de N sista siffrorna, och vid additionen helt enkelt kan skriva ihop talen. Eftersom datorn räknar binärt kommer därför sidstorleken alltid vara en jämn multipel av 2 i ett riktigt system, d.v.s. 2N.

Figuren ovan visar en översiktlig bild av hur det ser ut. Flera detaljer som inte är viktiga i denna uppgift har utelämnats. Normalt använder processen endast en liten del av adressrymden den tilldelats. I typfallet placeras programkod, gobala variabler, och konstanter i sidorna längst ned i processens minne (nära address 0), och processens exekveringsstack placeras i sidor högst upp i processens minne. Däremellan brukar många sidor vara helt oanvända. För varje sida som processen behöver i logiska minnet reserveras en frame i fysiskt

minne och information om vilken frame som reserverades placeras på sidans plats i sidtabellen. Oanvända sidor markeras i sidtabellen som ogiltiga, och har ingen frame i fysiskt minne knuten till sig.

Ett problem som kan uppstå med denna minnesmodell är att processen kan råka försöka använda en adress där innehållet i översättningstabellen är ogiltigt. Detta händer ofta om processen är felaktigt skriven (t.ex. råkar använda en oinitierad pekare). Om detta fel uppstår då processen exekverar är det ingen större fara: CPU ger ifrån sig ett interrupt (pagefault) när uppslagningen misslyckas, interruptet hanteras av operativsystemet som då kan terminera processen med ett felmeddelande som brukar lyda något i stil med ”page fault”, ”segmentation fault” eller ”bus error”. Om det däremot händer då OS exekverar är det allvarligt. Det betyder ju då att OS försöker använda en adress som ”inte finns”, och OS terminerar sig själv, d.v.s. kraschar. Det får inte hända. OS får aldrig krascha!

Tyvärr är det nu så att när en process behöver en tjänst från OS måste OS få information om vad saken gäller. Denna information måste processen lagra i sitt eget minne (andras minne har den ju inte tillgång till). OS får reda på adressen dit, och OS använder adressen för att läsa och ev. modifiera minnet innan processen slutligen kan hitta resultatet av tjänsten där. Vad händer nu om processen avsiktligt eller oavsiktligt ger OS en ogiltig adress? Antingen kraschar OS, eller ännu värre, så lyckas OS (som ju har full access till allt minne) läsa/skriva adressen, vilket kan förstöra data för en annan process eller för OS självt.

För att hindra att något sådant kan uppstå måste operativsystem-programmeraren verifiera att alla adresser som kommer från en process är giltiga innan de används av OS. För att kontrollera att en address är giltig slår OS upp adressen i översättningstabellen för att se om det går bra. Hela poängen är alltså att utföra kontrollen *innan* läsning/skrivning sker till adresserna, så att OS kan undvika att krascha p.g.a. en ogiltig adress. En sådan uppslagning är dock långsam, och det är ofta *många* adresser att kontrollera då t.ex. läsning och skrivning från resp. till en fil kan behöva många megabyte (varje byte har en adress!) för att lagra all data. Det gäller därför att kontrollera precis så mycket (lite) som behövs. *Det går här att utnyttja att det som egentligen kontrolleras är innehållet i översättningstabellen. Det innehållet är identiskt för* alla *adresser inom en given sida, eftersom* alla *adresser inom en sida ger samma index i tabellen.* Resultatet för en av adresserna i en sida är alltså giltigt för alla adresser inom samma sida. Utnyttjas detta kan prestanda ökas avsevärt. En annan metod som ofta görs i praktiken är att nyttja processorns automatiska uppslagning som både är implementerad i hårdvara och (förmodligen) nyttjar en TLB.

När adresser ändå kontrolleras finns (senare och i Pintos) ytterligare två saker att kontrollera. Processer får på inga villkor använda adresser större än PHYS_BASE, då minnet ovanför den adressen är reserverat för OS kernel. PHYS_BASE är en konstant som bestämts av operativsystem-programmeraren. Slutligen är alla adresser som är NULL ogiltiga.

11.1.1 Paging i Pintos

Pintos använder *paging* i två nivåer. Varje tråd (se `threads/thread.h`) i kernel håller reda på en så kallad `pagedir`, som innehåller pekare till `pagetables`. Pagedir är alltså en pagetable för en större sidindelad pagetable. Filerna `threads/vaddr.h` och `userprog/pagedir.h` innehåller deklarationer för att hantera pages och pagetable. Observera att en manuell uppslagning i pagetable med `pagedir_get_page` är långsam jämfört med processorns automatiska uppslagning. Antalet uppslagningar skall alltså minimeras. Lyckligtvis räcker det med att slå upp *en* adress inom en sida för att kontrollera om *alla* adresser inom sidan är giltiga. Är sidan markerad som ogiltig i pagetable skulle ju uppslagningen misslyckas på varje adress inom den sidan.

Den logiska minnesrymd som via paging tilldelas en tråd (process) i Pintos är (som processen ser det) sekventiell och uppdelad i två delar. Användarminnet finns från adress 0 upp till adressen `PHYS_BASE (0xc0000000)`. Resterande minne, inklusive `PHYS_BASE`, tillhör kernel. Processen få endast använda användarminnet. Observera speciellt att processens stack är placerad så den växer från `PHYS_BASE` och nedåt mot lägre logisk adress, samt att processens kod är placerad med start på adress `0x08048000`. Det ger ett spann av en storlek om ungefär 3GB, där alla adresser däremellan knappast är knutna till fysiska adresser. Du måste alltså förutsätta att det mellan två giltiga adresser kan finnas sidor med adresser som är ogiltiga.

Reglerna för om en viss adress går att läsa på ett säkert sätt är som följer:

- Adress 0 (`NULL`) är aldrig giltig. Notera att detta ofta inte behöver hanteras som ett specialfall eftersom page 0 inte finns, och således kommer `pagedir_get_page(NULL)` returnera `NULL`.
- Data som återfinns på en (logisk) adress större än eller lika med PHYS_BASE får inte läsas eller skrivas av processen, det är reserverat för kernel. Eftersom vi fortfarande pratar om logiska adresser i processens logiska (sekventiella) minnesrymd räcker det naturligtvis att testa den högsta adressen av dem som skall användas. Är den åtkomlig är naturligtvis även alla lägre adresser åtkomliga.
- Data som är lagrat på en logisk adress som återfinns inom en ogiltig sida i pagetable är förbjuden att läsa då det skulle ge upphov till pagefault. Om `pagedir_get_page` returnerar `NULL` är samtliga adresser inom samma sida ogiltiga. Annars är samtliga adresser inom samma sida giltiga. Huruvida en logisk adress högre eller lägre än den testade är giltig går inte att avgöra, om den inte infaller inom samma page som den testade. Kontroll av denna punkt givet en sekvens adresser har du löst i lab 14.

11.2 Accesskontroll

Ett möjligt sätt för användarprogram att komma runt minnesskyddet är att låta operativsystemet självt utföra de förbjudna åtgärderna (det har ju tillgång och tillåtelse till allt). Detta kan göras genom att utnyttja parametrarna till systemanrop. Alla systemanrop använder en pekare (minnesadress!) till användarprogrammets stack för att komma åt parametrarna till systemanropet. Genom att ange en falsk stackpekare innan systemanrop kan man alltså få operativsystemet att läsa förbjudna adresser.

Vidare så anger vissa parametrar till vissa systemanrop adressen där filnamn, kommando eller data startar, eller adressen där operativsystemet skall placera returnerad data. Om operativsystemet okritiskt läser eller skriver till någon av dessa pekare kan ett användarprogram alltså komma åt förbjudet minne genom att skicka in falska pekare som parametrar till systemanropen. Framförallt är det systemanropen `read` och `write` som är farliga, men även ett par andra.

Som exempel kan du tänka dig att vi vill ändra på uppgifterna lagrade på en godtycklig (förbjuden) adress X och framåt. Först öppnar vi en fil för att temporärt lagra dessa uppgifter. Därefter anropar vi systemanropet `write` för att skriva från godtycklig adress till filen. Nu kan de förbjudna uppgifterna läsas från filen som vanligt, ändras och skrivas tillbaka till filen. Därefter anropas systemanropet read för att läsa från filen till en godtycklig adress (i detta fall tillbaka till X). Pseudokod följer:

```
/* arrange temporary storage for the data */
char data[SIZE];
int fd = open("mem.dump");

/* save content of forbidden address X to file */
write(fd, X, SIZE);

/* read it to our memory space */
read(fd, data, SIZE);

/* do something with data */


/* save the data back to our file */
write(fd, data, SIZE);

/* read it to the forbidden memory address X */
read(fd, X, SIZE);
```

Varken detta eller något annat scenario skall vara möjligt när du säkrat din implementering.

Det gäller att noga utföra alla kontroller du kan komma på för varje parameter till ett systemanrop. Det får aldrig bli fel i operativsystemet för att ett program skickar konstiga data till systemanrop. Alla data måste kontrolleras så att operativsystemet kan använda dem utan att fel uppstår i operativsystemet.

Om användarprogrammet däremot råkar skicka en felaktig pekare till ett systemanrop som gör att operativsystemet skriver över viktig data i användarprogrammets *eget* minne så får användarprogrammets programmerare skylla sig själv. *Operativsystemet skyddar bara sitt eget och andra processers minne, det finns inget sätt för operativsystemet att veta vad användarprogrammet anser vara viktigt.*

Saker som är bra att kontrollera är stackpekaren (att inga data läses utanför stacken), systemanropsnummer, alla parametrar som är pekare, pekare till strängar (alla adresser inom strängen), pekare till buffer (alla adresser i buffern), fildeskriptorer (fd), och processid (pid).

11.3 Parameteröverföring vid funktionsanrop

När ett program anropar en funktion sker ofta överföring av information till och från funktionen, så kallad parameteröverföring samt returvärde. Parametrar lagras vanligen på en stack. Varje process (eller tråd om processen är flertrådad) har en stack. I Pintos finns stacken i ett minnesutrymme som startar på adressen `PHYS_BASE` och växer nedåt (mot lägre minnesadress) varefter mer plats behövs. En pekare, stackpekaren, håller reda på var i minnet stacken slutar just nu. I Pintos finns den i (x86) processorregistret `ESP`. Den pekaren anger alltså toppen av stacken. Returvärden från funktioner placeras i (x86) processorns `EAX` register. (Kompilatorn ser till att `EAX` är ledigt när funktionen anropas.) Antag till exempel att vi har funktionen:

```
int exempel(int a, int b, int c)
{
        return a + b + c;
}
```

Antag vidare att följande anrop existerar:

```
int r = exempel(1, 2, 3);
```

Ett funktionsanrop går nu (i korthet) till enligt följande:

1. Stackpekaren räknas ned för att göra plats för sista argumentet (3).
2. Sista argumentet (3) placeras på adressen stackpekaren anger.
3. Stackpekaren räknas ned för att göra plats för näst sista argumentet (2).
4. Näst sista argumentet (2) placeras på adressen stackpekaren anger.
5. Stackpekaren räknas ned för att göra plats för första argumentet (1).
6. Första argumentet (1) placeras på adressen stackpekaren anger.
7. Stackpekaren räknas ned för att göra plats för adressen till nästa instruktion (efter funktionen).
8. Adressen till nästa instruktion (återhoppsadressen) kopieras till stacken.
9. Funktionen exekverar och räknar ut resultatet av a + b + c.
10. Resultatet (som blir 6) placeras i processorns register EAX.
11. Funktionen returnerar genom att processorn fortsätter exekvera från återhoppsadressen.
12. Stackpekaren räknas upp för att komma tillbaka till ursprungsläget (innan punkt 1).

Argumenten utvärderas och kopieras alltså till stacken i omvänd ordning, sista argumentet först, vilket gör att de hamnar i rätt ordning på stacken, första på lägst adress, nästa på följande adress osv.

Denna procedur gäller för alla funktionsanrop. Även start av programmet (anrop av main) och systemanrop har denna uppbyggnad av stacken, men platsen för återhoppsadressen används ibland till annat (systemanropsnumret i fallet med systemanrop). I Pintos startas ett program genom att programmets ”startfunktion” `_start` anropas. Startfunktionen läggs till automatiskt av kompilartorn och fungerar som en ”wrapper” till `main`. Den ser ut som följer (i Pintos, se `lib/user/entry.c`):

```
_start(int argc, char* argv[])
{
  exit(main(argc, argv));
}
```

Observera två saker:

- Inne i `_start` kommer de båda parametrarna till `_start`, `argc` och `argv`, **alltid** att läsas från stacken för att kopieras som argument till `main`. De kopieras alltså från stacken till ny position på stacken. Detta sker oavsett om `main` använder sina parametrar eller ej. Om stacken är tom när detta sker kommer processen försöka läsa parametrarna övanför(utanför) sin stack och ett pagefault erhålls.
- `_start` kommer *aldrig* att returnera, eftersom systemanropet `exit` alltid körs innan dess. Detta betyder att återhoppsadressen normalt aldrig kommer att användas. *Dock är det bra om returadressen är NULL ur felsäkerhetssynpunkt* (ett specialdesignat eller felaktigt program skulle kunna nå dit).

Funktionen `_start` är speciell på så sätt att den startas av operativsystemet. Eftersom `_start` tar emot två argument (samma som `main`) måste operativsystemet se till att dessa är korrekt initierade på stacken. Detta är lite knepigare än det först verkar. Den andra parametern till `main`, ofta kallad `argv`, är en pekare till en sekvens med teckenpekare. Det är alltså pekare i flera led! `argv` måste naturligtvis peka till en giltig sekvens av teckenpekare, som vardera i sin tur måste peka på en giltig sekvens med tecken.

Alla dessa teckenpekare och tecken måste lagras någonstans. De kan teoretiskt sett lagras var som helst i processens minne, men mest praktiskt brukar vara att lagra dem på stacken, eftersom det är det enda minnet vi har att tillgå omedelbart.

12 Automatiska tester

I mappen `tests/` finns en samling automatiska tester för olika saker i Pintos. I slutet av kursen ska alla dessa tester vara lyckade. Testerna körs med kommandot `make check` i `userprog/`-mappen. I och med att testerna tar ett tag att köras används lämpligtvis flaggan `-j8` för att snabba på processen något. Utdatan från `make check` slutar med en sammanfattning av vilka tester som lyckades och vilka som misslyckades. Se nedan för en sammanfattning av när testerna bör fungera.

OBS! Pintos egna tester ställer tre viktiga krav på din implementation:

1. Det får *inte* förekomma några debug-utskrifter som *inte* startar med fyrkant+mellanslag (`"# "`).
2. Implementationen av wait *måste* fungera korrekt.
3. När en process avslutar *måste* den skriva ut sin *exit_status* (parametern *status* till systemanropet *exit* om processen avslutar via ett systemanrop, -1 om processen avslutas av kernel till följa av något fel.) Den `printf` som behövs finns i `process_cleanup` men du måste *se till att variabeln status har rätt värde innan utskrivten sker.*

   ```
   printf("%s: exit(%d)\n", thread_name(), status);
   ```

4. Om en process kraschar av någon anledning förväntar sig testerna att dess status sätts till -1. Dvs. om en föräldraprocess kör `wait` på processen ska `wait` returnera -1.

Alla test som körs av `make check` består av att ett usermode-program körs i Pintos, och att utdatan från Pintos matchas mot en eller flera förväntade utdatamönster för att se om allt gick bra eller inte. Källkoden för de program som körs finns i mappen `tests/`. Namnet på testet motsvarar den fil som kördes. Misslyckades exempelvis testet `tests/userprog/write-zero`, så finns källkoden till testet i `tests/userprog/write-zero.c`.

Ibland är det inte enkelt att inse vad som gjorde att testet misslyckades, trots att man lusläser källkoden och funderar på vad den kan orsaka för konstigt. Lyckligtvis sparar `make check` all utdata från testerna i mappen `src/userprog/build/tests/`, återigen med namn motsvarande testets namn. Följande filer är intressanta att inspektera:

- `.allput` - Innehåller all utdata från Pintos när testprogrammet kördes.
- `.output` - Innehåller den utdata som analyseras av testet. Mer specifikt så har alla rader som börjar med `#` (fyrkant följt av mellanrum) tagits bort.
- `.result` - Innehåller ett meddelande som försöker förklara vad som var fel. Ofta i form av en diff mellan förväntad utdata och faktisk utdata. Rader med ett plus (+) framför är rader som skrevs ut av din implementation, men som var fel. Rader med minus (-) framför är rader som *inte* skrevs ut men som *borde* ha skrivits ut för att testet skulle få godkänt. Logiken är: plus = ”överflödig utdata” och minus = ”saknad utdata”. Ibland finns det flera möjliga godkända utdata, och i så fall visas skillnaden mot alla möjligheter.

Har exempelvis testet `tests/userprog/write-zero.c` misslyckats kan det vara bra att börja felsökningen med att läsa testets källkod, sedan `userprog/build/tests/userprog/write-zero.result` för att få en uppfattning om vad som har gått snett. Efter det kan det vara värt att titta på `userprog/build/tests/-userprog/write-zero.output` för att se ifall Pintos kraschade. I så fall finns en `backtrace`-rad där som kan underlätta felsökningen oerhört. Här kan man också se ifall någon av sina spårutskrifter är kvar och påverkar testen.

Det är också värt att notera att en del av testerna försöker testa att olika delar av systemet är korrekt synkroniserade. Det är därför normalt att dessa tester lyckas ibland och misslyckas ibland. Ser man det beteendet så är det sannolikt ett synkroniseringsfel någonstans.

Vill man försöka testa att synkroniseringen korrekt är det därför lämpligt att köra testerna många gånger för att se om de lyckas varje gång. Till hjälp för detta finns scriptet `pintos-check-forever`. Det kör helt enkelt `make check` gång på gång tills något test misslyckas.

Till slut ett par saker som kan vara värda att veta. Om man bara vill köra de test som misslyckades tidigare kan man köra `make recheck`, vilket ofta går snabbare än att köra alla tester från början. Har man inte ändrat något kan man se sammanfattningen igen genom att helt enkelt köra `make check` igen. Har inget ändrats inser Make det och gör inget. Vill man i stället köra testen från början kan man köra `make clean` följt av `make -j8 check` igen.

Utöver detta finns ett script, `pintos-single-test` som enkelt låter dig köra ett enskilt test. Det körs exempelvis så här: `pintos-single-test tests/userprog/write-zero`.

12.1 Vanliga fel som är svåra att felsöka

- Vissa rader verkar försvinna från utdatan i `output`-filen. När jag tittar i `allput`-filen så verkar olika rader blanda sig på konstiga sätt.

  Testerna i Pintos antar att utmatningen via `printf` är synkroniserad. Detta gäller för anrop till `printf` i kärnan, men i och med att `printf` i usermode anropar `write`, så måste du se till att din implementation skriver ut hela buffern som en operation så att den inte blir avbruten. Funktionerna för att skriva ut

en char-array som finns ser till att detta görs, så se till att din implementation inte anropar `write` flera gånger.

- Vissa rader i slutet av utdatan saknas i `output`-filen, men de finns i `allput`-filen.

  Testerna tittar bara på den utdata som skrivs ut efter `Executing '<program>'` och innan `Execution of '<program>' complete.` Om rader som testet letar efter skrivs ut efter raden med `complete` kommer de inte ses av testet. Detta beror ofta på att din implementation av `process_cleanup` meddelar föräldraprocessen att den är klar innan den gör sin utskrift.

12.2 Skapa egna testfall i Pintos (endast för referens)

Kopiera och modifiera ett befintligt testfall (*.c och *.ck) i katalogen `tests/userprog` eller `tests/filesys/base` (beroende på test). Lägg sedan till det i `Make.tests` genom att ange värden för *_TESTS*, *_SRC*, och *_PUTFILES*.

12.3 När ska testerna fungera?

Nedan finns en sammanfattning av när olika tester bör fungera. Notera, att en väl genomtänkt implementation av tidigare laborationer mycket väl kan göra att några testen fungerar tidigare. Vissa test försöker testa att synkroniseringen är korrekt gjord, så det kan fungera sporadiskt även innan labben är genomförd.

Notera att inga tester fungerar förrän labben ”Processhantering” (inkluderat `wait`) är färdig.

Testnamn	När testet bör fungera
`tests/klaar/read-bad-buf`	15. Säkra systemanropen
`tests/klaar/low-mem`	08. Den första processen
`tests/klaar/exec-corrupt`	08. Den första processen
`tests/klaar/pfs`	13. Readers-writers lock
`tests/filst/sc-bad-write`	15. Säkra systemanropen
`tests/filst/sc-bad-close`	15. Säkra systemanropen
`tests/filst/sc-bad-nr-1`	15. Säkra systemanropen
`tests/filst/sc-bad-nr-2`	15. Säkra systemanropen
`tests/filst/sc-bad-nr-3`	15. Säkra systemanropen
`tests/filst/sc-bad-align-1`	15. Säkra systemanropen
`tests/filst/sc-bad-align-2`	15. Säkra systemanropen
`tests/filst/sc-bad-exit`	15. Säkra systemanropen
`tests/filst/sc-bad-buf`	15. Säkra systemanropen
`tests/userprog/args-none`	09. Processhantering
`tests/userprog/args-single`	09. Processhantering
`tests/userprog/args-multiple`	09. Processhantering
`tests/userprog/args-many`	09. Processhantering
`tests/userprog/args-dbl-space`	09. Processhantering
`tests/userprog/sc-bad-sp`	15. Säkra systemanropen
`tests/userprog/sc-bad-arg`	15. Säkra systemanropen
`tests/userprog/sc-boundary`	15. Säkra systemanropen
`tests/userprog/sc-boundary-2`	15. Säkra systemanropen
`tests/userprog/halt`	Systemanropen `halt` och `exit`
`tests/userprog/exit`	Systemanropen `halt` och `exit`
`tests/userprog/create-normal`	06. Systemanrop för filhantering
`tests/userprog/create-empty`	06. Systemanrop för filhantering
`tests/userprog/create-null`	15. Säkra systemanropen
`tests/userprog/create-bad-ptr`	15. Säkra systemanropen
`tests/userprog/create-long`	06. Systemanrop för filhantering
`tests/userprog/create-exists`	06. Systemanrop för filhantering
`tests/userprog/create-bound`	15. Säkra systemanropen
`tests/userprog/open-normal`	06. Systemanrop för filhantering
`tests/userprog/open-missing`	06. Systemanrop för filhantering
`tests/userprog/open-boundary`	15. Säkra systemanropen
`tests/userprog/open-empty`	06. Systemanrop för filhantering
`tests/userprog/open-null`	15. Säkra systemanropen
`tests/userprog/open-bad-ptr`	15. Säkra systemanropen
`tests/userprog/open-twice`	06. Systemanrop för filhantering
`tests/userprog/close-normal`	06. Systemanrop för filhantering
`tests/userprog/close-twice`	06. Systemanrop för filhantering
`tests/userprog/close-stdin`	06. Systemanrop för filhantering
`tests/userprog/close-stdout`	06. Systemanrop för filhantering
`tests/userprog/close-bad-fd`	15. Säkra systemanropen

Testnamn	När testet bör fungera
`tests/userprog/read-normal`	06. Systemanrop för filhantering
`tests/userprog/read-bad-ptr`	15. Säkra systemanropen
`tests/userprog/read-boundary`	15. Säkra systemanropen
`tests/userprog/read-zero`	06. Systemanrop för filhantering
`tests/userprog/read-stdout`	06. Systemanrop för filhantering
`tests/userprog/read-bad-fd`	15. Säkra systemanropen
`tests/userprog/write-normal`	06. Systemanrop för filhantering
`tests/userprog/write-bad-ptr`	15. Säkra systemanropen
`tests/userprog/write-boundary`	15. Säkra systemanropen
`tests/userprog/write-zero`	06. Systemanrop för filhantering
`tests/userprog/write-stdin`	06. Systemanrop för filhantering
`tests/userprog/write-bad-fd`	15. Säkra systemanropen
`tests/userprog/exec-once`	10. Processhantering: `wait`
`tests/userprog/exec-arg`	09. Processhantering
`tests/userprog/exec-multiple`	10. Processhantering: `wait`
`tests/userprog/exec-missing`	08. Den första processen
`tests/userprog/exec-bad-ptr`	15. Säkra systemanropen
`tests/userprog/wait-simple`	10. Processhantering: `wait`
`tests/userprog/wait-twice`	10. Processhantering: `wait`
`tests/userprog/wait-killed`	15. Säkra systemanropen
`tests/userprog/wait-bad-pid`	15. Säkra systemanropen
`tests/userprog/multi-recurse`	10. Processhantering: `wait`
`tests/userprog/multi-child-fd`	10. Processhantering: `wait`
`tests/filesys/base/lg-create`	15. Säkra systemanropen
`tests/filesys/base/lg-full`	15. Säkra systemanropen
`tests/filesys/base/lg-random`	15. Säkra systemanropen
`tests/filesys/base/lg-seq-block`	15. Säkra systemanropen
`tests/filesys/base/lg-seq-random`	15. Säkra systemanropen
`tests/filesys/base/sm-create`	15. Säkra systemanropen
`tests/filesys/base/sm-full`	15. Säkra systemanropen
`tests/filesys/base/sm-random`	15. Säkra systemanropen
`tests/filesys/base/sm-seq-block`	15. Säkra systemanropen
`tests/filesys/base/sm-seq-random`	15. Säkra systemanropen
`tests/filesys/base/syn-read`	13. Readers-writers lock
`tests/filesys/base/syn-remove`	13. Readers-writers lock
`tests/filesys/base/syn-write`	13. Readers-writers lock

13 Vanliga fel och funderingar

Här finns några vanliga fel och funderingar som ni kan stöta på under laborationerna och lösningar på dessa.

Felmeddelande: `Storage size of struct map unknown` Detta innebär att kompilatorn inte har sett definitionen av en datatyp (exempelvis `struct inode`) där kompilatorn behöver veta hur stor den är (se var i din kod felmeddelandet pekar). Detta beror ofta på att du har definierat strukturen i en `c`-fil. Det är inte nödvändigtvis fel, utan det är ett vanligt sätt för att göra privata datastrukturer i C (och görs i Pintos på olika ställen). Det finns två lösningar:

- Om datatypen ska vara privat: använd bara pekare till datatypen i alla andra filer än den filen där datatypen är definierad, och anropa bara de funktionerna som finns definierade i `h`-filen för att manipulera datatypen. Skapa nya hjälpfunktioner vid behov.
- Om datatypen inte behöver vara privat: flytta definitionen av datatypen till tillhörande `h`-fil, och

dubbelkolla att du har inkluderat den `h`-filen där felet uppkom.

Varning: `Function declaration isn't a prototype` Detta innebär att du någonstans har deklarerat en funktion så här: `void minfunktion()`. I C betyder det att funktionen kan ta godtyckligt antal parametrar. Vill du att funktionen inte ska ta några parametrar ska du i stället skriva `void minfunktion(void)`.

Varning: `No previous prototype for function` Detta innebär att du har definierat en funktion utan att deklarera den först. Exempelvis om du i en `c`-fil har skrivit:

```
void minfunktion(void) {
    printf("Test\n");
}
```

Varningen försöker påminna dig om att du har glömt att deklarera funktionen i en headerfil. Så om du vill att funktionen ska vara tillgänglig från andra `c`-filer vill du lägga till en deklaration i en headerfil och se till att du har inkluderat den:

```
void minfunktion(void);
```

Alternativt, om du i stället tänkte dig att funktionen skulle vara privat i den `c`-fil du deklarerade den i så vill du märka den som `static` så att den inte riskerar att krocka med en annan funktion med samma namn i en annan `c`-fil:

```
static void minfunktion(void) {
    printf("Test\n");
}
```

Hur fungerar `sizeof`? `sizeof` är en operator som under **kompileringen** av ditt program tittar på datatypen av det du ger den, och räknar hur många bytes som krävs för att lagra den. Värt att notera är att C-arrayer lätt tappar sin storlek:

```
int a[5]; // => sizeof(a) == 5*sizeof(int)
int foo(int a[]) // => sizeof(a) == sizeof(int *)
```

Det går att räkna ut antalet element i en C-array med följande trick:

```
int array[10];
int elements = sizeof(array) / sizeof(*array);
```

I och med att detta är något som händer under kompileringen av ditt program går det inte att ta reda på hur många element som fanns i ett dynamiskt allokerat array, eller i en godtycklig pekare.

Varning: `malloc` eller `free` är inte definierad Detta beror på att du inte har inkluderat deklarationen av `malloc` (eller `free`) i den fil du arbetar med. I felmeddelandet brukar kompilatorn numera försöka vara snäll och be dig inkludera `#include <stdlib.h>`. **Detta stämmer dock inte i Pintos**. I Pintos så finns `malloc` deklarerad i filen `#include "threads/malloc.h"` i stället. Tyvärr vet kompilatorn inte om detta och ger felaktig information i det här fallet.

Skriver du program som ska köras i usermode i Pintos så är situationen lite värre. Här finns inte `malloc` implementerad alls (det är därför den inte ligger i `stdlib.h` som den brukar). Du behöver troligtvis inte `malloc` i usermode-program i den här labbserien, så det ska inte vara något problem.

Hur löser jag felhantering? Detta är inte alltid enkelt att få till rätt och snyggt (varken i C eller i andra språk). Här är några förslag som fungerar i olika situationer, fundera på vilken som passar bäst till de olika funktionerna ovan:

- Avsluta programmet med en felutskrift (likt `PANIC`). Detta är inte en bra idé för minsta lilla fel (speciellt inte när vi skriver ett operativsystem). Men det är bättre att få reda på vad som har

gått fel än att inte få reda på felen och sedan sitta och felsöka i timmar i blindo.

- Returnera en felkod. Det som är viktigt här är att värdet som betyder ”fel” inte kan förekomma i ett normalt fall. Ett exempel som har misslyckats med detta är C-funktionen `atoi`, som konverterar en sträng till ett heltal. Den returnerar 0 om något gick fel, men 0 är också ett giltigt heltal, så det är svårt att veta om konverteringen gick fel eller om det var talet 0 som blev konverterat. Ett bra exempel är systemanropet `read`. Det returnerar antalet bytes som har lästs in i en buffer. Det returnerar ett negativt tal om något har gått fel. Det fungerar bra här eftersom vi aldrig kan läsa ett negativt antal bytes.
- Modifiera gränssnittet så att felkoden får en egen kanal. Exempelvis kan vi låta funktionen returnera en bool som indikerar om funktionen lyckades eller inte, och returnera resultatet via en pekarparameter.

Oavsett vilket fall du hamnar i är det viktigt att den koden som använder funktioner som kan misslyckas faktiskt kontrollerar om något gick fel och hanterar det på lämpligt sätt.