LINKÖPINGS UNIVERSITET

TDIU16: Process- och operativsystemprogrammering

Lab 1: Introduktion till Pintos och C-programmering

Filip Strömbäck, Klas Arvidsson, Daniel Thorén

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2026

2026-03-11

Mål

Denna laboration har följande mål:

1. Installera Pintos på ditt system
2. Öva på programmering i C
3. Öva på felsökning med GDB

Som en del av punkt 2 ovan kommer du att implementera en *associativ container* som vi också kommer att använda i kommande laborationer i Pintos. Felsökning med GDB är också något som underlättar felsökning av framtida laborationer.

Tanken är att laborationen ska ta ca 4 timmar att göra. Laborationen är tänkt som en introduktion till C och GDB. I övrigt introducerar den inga nya koncept.

Del A Installera Pintos

Målet med denna del är att skapa din egen kopia av Pintos-repositoryt att jobba med under kursen, och se till så att du kan kompilera och köra Pintos framöver.

För information om detta, se rubrikerna ”Introduktion”, ”Installation”, och ”Bra att känna till om C” i Pintos-Wiki.

Denna del av laborationen behöver du inte redovisa.

Del B Programmering i C och felsökning med GDB

Denna del har som mål att ge en introduktion till programmering i C, men fokus ligger på felsökning med GDB. I denna del kommer vi att arbeta utanför Pintos, vilket innebär att vi kan starta GDB som vanligt. Eftersom Pintos körs inuti en virtuell maskin krävs några extra steg för att starta GDB, men resten är detsamma. Detta är beskrivet under ”Felsökning med GDB” under ”Felsökning” i Pintos-Wiki. Det finns även en kort sammanfattning i slutet av denna del.

I mappen `standalone/lab1b/` finns tre C-program: `debug1.c`, `debug2.c`, och `debug3.c`. Dessa program innehåller fel som gör att programmen kraschar. Uppgiften är att hitta felen med hjälp av GDB, och att sedan korrigera felen på lämpligt sätt.

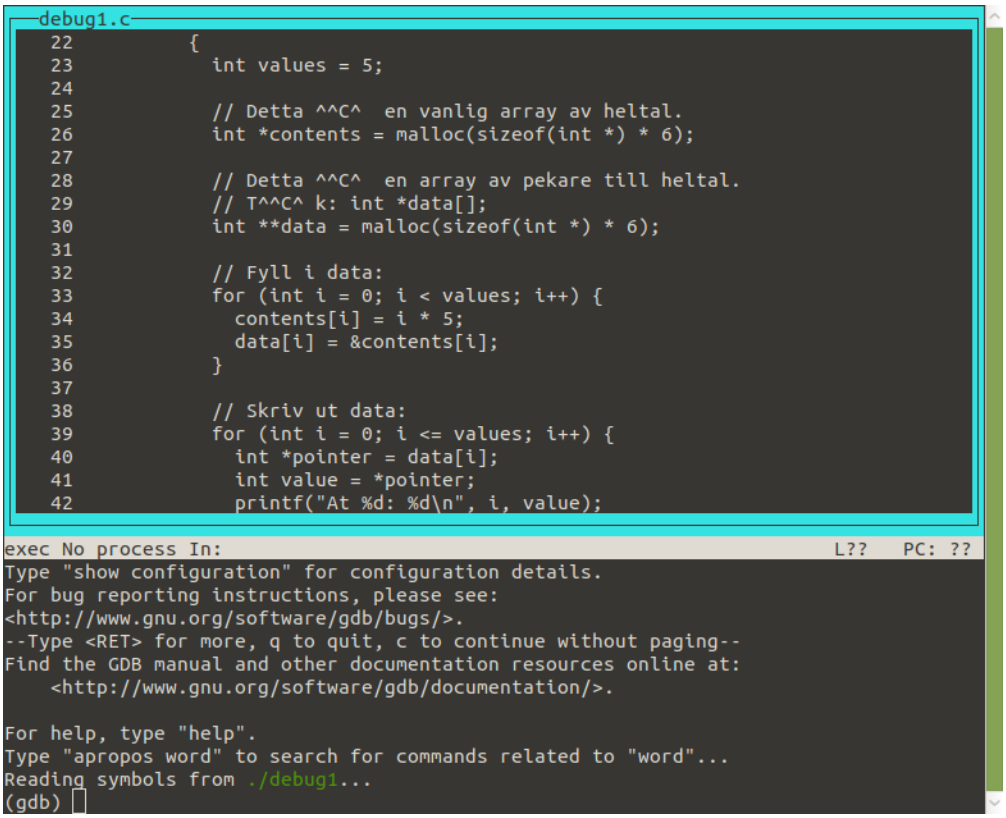
Notera: I den här delen används en version av funktionerna `malloc` och `free` som gör att programmet kraschar snabbare vid minnesfel. Detta gör felsökningen enklare i den här uppgiften. Detta betyder dock **inte** att du kan lita på att ditt program kraschar direkt ifall du råkar göra samma fel. Det kan vara så att programmet ser ut att fungera, men kraschar långt senare, när du lägger till ny men orelaterad kod.

Tips: Om du inte förstår vad som händer i någon del av programmen, så kan du använda visualiseringsverktyget som är beskrivet på kurshemsidan för att visualisera programmen om du vill. På skolans datorer och via ThinLinc kan det startas med kommandot `/courses/TDIU16/progvis/start` Instruktioner för att köra verktyget på egen dator finns på kurshemsidan. Se dock till att öva på att använda GDB också, eftersom Pintos inte går att köra i visualiseringsverktyget.

B.1 Program 1: `debug1.c`

Du kan kompilera programmet med kommandot `make debug1` Du kan sedan köra programmet med kommandot `./debug1` Du kommer då se att programmet kraschar med meddelandet `Segmentation fault (core dumped)`. Detta innebär att programmet försökte komma åt minne på en adress som av någon anledning inte var giltigt.

För att felsöka med GDB kan vi köra kommandot: `gdb -tui ./debug1` Flaggan `-tui` öppnar det som GDB kallar för ett *Text User Interface* (därav flaggan `-tui`). Glömmer man flaggan `-tui` går det även att starta TUI:t genom att skriva `tui enable` inuti GDB, eller genom att trycka Ctrl+X följt av Ctrl+A. När du har startat GDB kommer du se något liknande bilden nedan:



Om din terminal är liten kan du se ett meddelande i botten som säger: `Type <RET> for more, q to quit, c to continue without paging`. Detta sker när GDB försöker skriva ut mer text än vad som får plats i rutan i botten. Skriv helt enkelt `c` och tryck på enter för att komma till prompten `(gdb)` som i bilden.

Som man kan se i bilden består gränssnittet av två delar. Den övre delen visar källkod för vårt program. GDB kommer här att peka på vilken del av programmet som körs, vart brytpunkter är satta och dylikt. I och med att programmet inte har startat ännu så innehåller den här delen inte så mycket intressant just nu.

Nedre delen består av en ”terminal” där man kan skriva kommandon till GDB, och där GDB svarar med information om programmet. Prompten `(gdb)` innebär att GDB är redo och väntar på att ta emot kommandon. Om man stänger av TUI:t i GDB är det endast denna delen som visas.

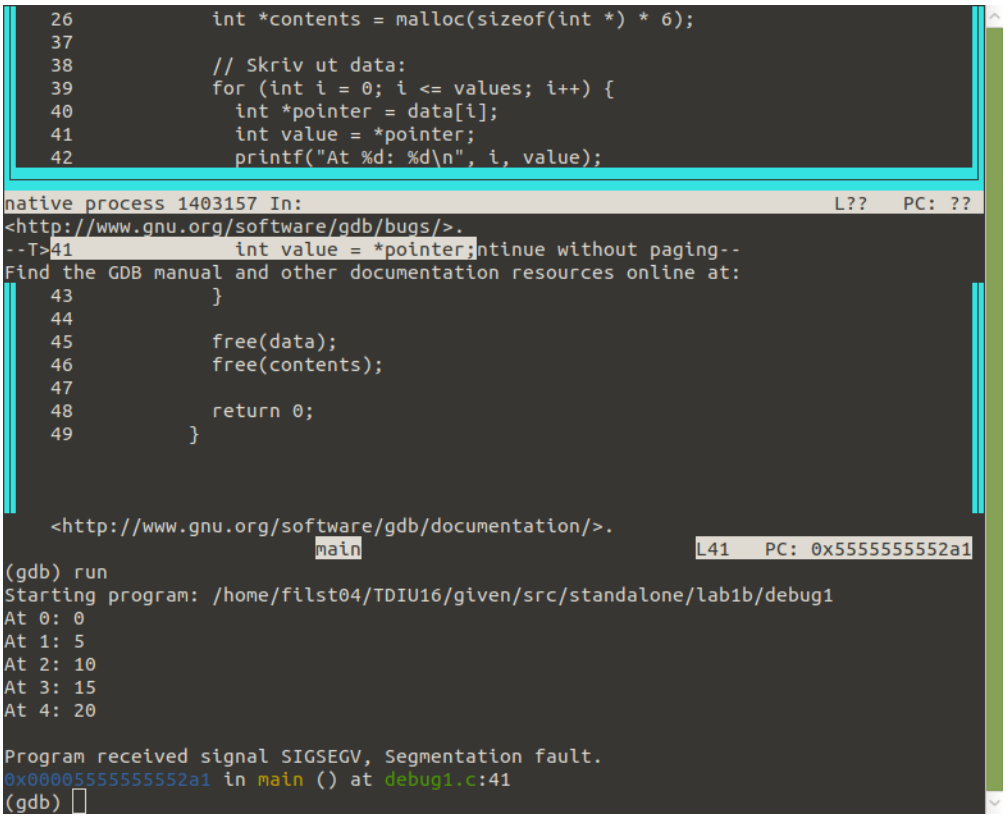
Redan här är det värt att poängtera ett par saker om TUI:t. I bilden ovan (och som standard) så är den övre delen markerad med en blå ram. Detta innebär att den har fokus. Om du trycker på pil upp och ned så kommer GDB att skrolla kodvyn. Om du trycker på Ctrl+X följt av O (dvs. tryck först Ctrl+X, släpp upp båda tangenterna, och tryck sedan O), så kommer den blåa ramen försvinna. Detta innebär att ”terminalen” i botten har fokus, och pil upp kan därmed användas för att hoppa bakåt i historiken och för att stega framåt och bakåt i din inmatning (likt hur det fungerar i en vanlig terminal). Den som är van vid Emacs kanske ser att Ctrl+X, O är samma kommando som används för att byta *window* i Emacs. Många enkla kommandon från Emacs fungerar även i GDB.

Det händer ibland att TUI:t blir ”trasigt”. Detta sker då programmet du felsöker skriver ut saker till terminalen, och ”skriver över” det som GDB tror finns där. Detta går enkelt att korrigera genom att trycka på

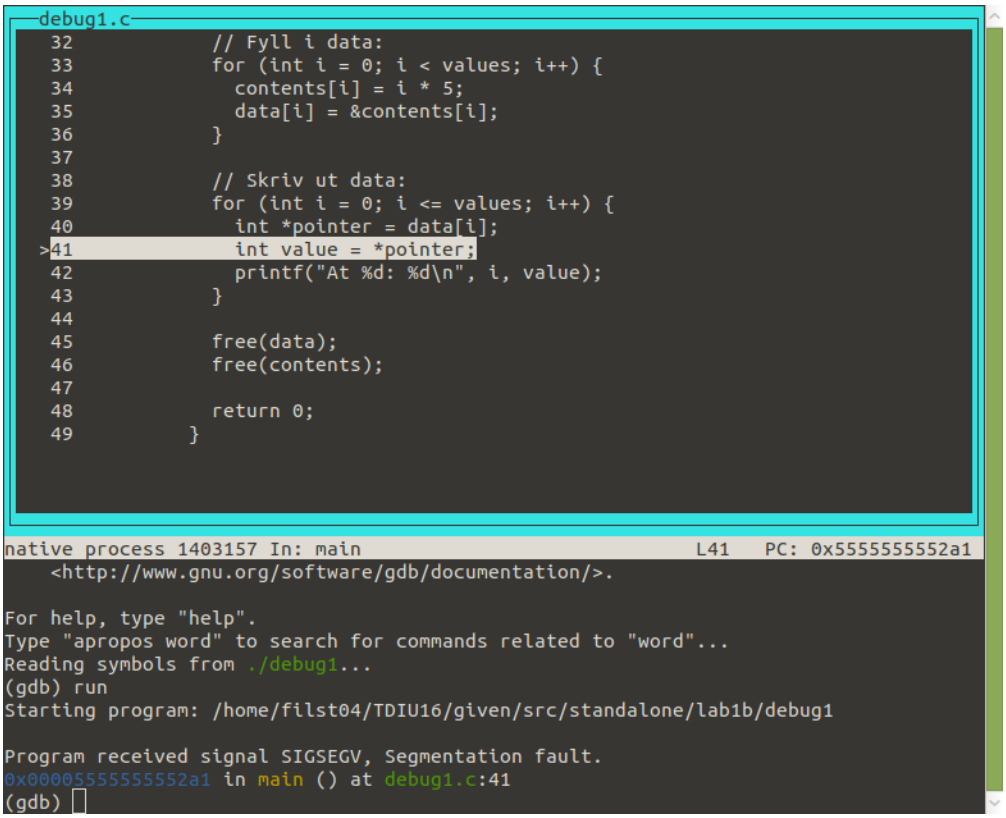
Ctrl+L. Alltså, ser något konstigt ut, testa att trycka på Ctrl+L, det skadar aldrig!

Nu är vi redo att börja felsöka. Starta programmet genom att skriva kommandot `run` och tryck på enter. Värt att notera är att (nästan) alla kommandon i GDB går att förkorta. Så länge det är tydligt för GDB vad du menar så kommer den ofta att göra rätt sak. Det går exempelvis att förkorta `run` till `r`. I den här instruktionen skrivs hela kommandona ut, och förkortningar anges inom parentes efteråt där det är relevant.

När du skrivit `run` (`r`) och tryckt på retur så kommer GDB att starta programmet och låta det köra. I det här fallet så skriver programmet ut lite saker och kraschar sedan. På mitt system ser det då ut som i bilden nedan:



Här ser GDB trasigt ut (på grund av utdatan från programmet), alltså trycker jag på Ctrl+L och får följande:



I den övre delen kan vi se att GDB har markerat raden som höll på att köras när programmet kraschade. I den nedre delen kan vi se meddelandet `Program received signal SIGSEGV, Segmentation fault`. Detta är samma information som vi såg i terminalen förut. Nästa rad säger att programmet kraschade på adress `0x00005555555552a1`. Detta är i sig inte intressant (vi vet inte vad som finns där, den varierar dessutom antagligen från körning till körning). GDB säger dock att det motsvarar rad 41 i filen `debug1.c`, och att denna rad ligger i funktionen `main`. Samma information visas också i övre halvan av fönstret på ett lite mer grafiskt sätt.

För att inse vad som gick fel kan vi nu inspektera vårt program med hjälp av kommandot `print` (`p`). Kommandot evaluerar ett C-uttryck och skriver ut resultatet. Det mesta som fungerar i C fungerar även i GDB (men inte riktigt allt). Här kan vi exempelvis se värdet av variabeln `i` genom att skriva: `print i` (eller `p i`). Vi får då följande svar:

```
$1 = 5
```

Detta innebär alltså att `i` var 5. Delen `$1 =` innebär att vi kan använda `$1` i ett senare uttryck ifall vi vill återanvända resultatet på ett smidigt sätt (exempelvis `print $1 + 2` ger 7).

Insikten att `i` var 5 gör att vi kan fokusera på element nummer 5 i arrayen. Vi kan återigen använda print-kommandot för att inspektera programmet. Antingen kan vi skriva `print data[i]`, eller så ser vi att värdet vi är ute efter redan finns i variabeln `pointer` från raden innan och skriver ut `pointer` i stället. Vi får då följande svar:

```
$2 = (int *) 0xcccccccccccccccc
```

Detta innebär att element fem i arrayet (och pekaren) har värdet `0xcccccccccccccccc`, och att typen är `int *` (pekare till `int`). I och med att vi kraschade på raden där programmet försökte avreferera pekaren så kan vi från detta misstänka att pekaren inte är giltig, men för att vara säkra kan vi fråga GDB genom att

skriva: `print *pointer` (dvs. uttrycket som vi tror krascahde). Vi får då följande svar:

```
Cannot access memory at address 0xcccccccccccccccc
```

Detta bekräftar våra misstankar om att pekaren var ogiltig. Härifrån kan vi alltså dra slutsatsen att pekaren i element nummer 5 i `data` inte är korrekt. Nästa fråga är då varför detta är fallet. Vi kan felsöka detta med GDB, exempelvis genom att skapa en så kallad *brytpunkt* (eng: *breakpoint*). Detta kan vi göra med kommandot `break` (`b`). Till kommandot `break` behöver vi också ange var brytpunkten ska vara. Man kan antingen ge den ett namn på en funktion (exempelvis `break main`) eller ett filnamn och ett radnummer (`break debug1.c:41`). I vårt fall vill vi skapa en brytpunkt på rad 35, och vi kan därmed skriva `break debug1.c:41` (eller bara `break 35` eftersom det är den filen vi ser i övre halvan). Som svar markerar GDB brytpunkten med `b+` till vänster om radnumret, och svarar med:

```
Breakpoint 1 at 0x1242: file debug1.c, line 35.
```

Vår brytpunkt fick alltså nummer 1. Om vi senare vill stänga av den kan vi skriva `disable 1` (`dis 1`). Vi kan nu starta om programmet med kommandot `run` (`r`) som förut. GDB säger att den kommer stänga av instansen som körs just nu, och frågar om det är okej. Svara ja på frågan (`y`). GDB startar då om vårt program och skriver ut följande:

```
Breakpoint 1, main () at debug1.c:41
```

Detta innebär att programmet har stoppats vid brytpunkt nummer 1 som vi skapade nyss. Vi kan också notera att `b+` till vänster om radnumret har ändrats till `B+` för att indikera att programmet har stannat vid brytpunkten åtminstone en gång. Nu kan vi likt tidigare skriva `print i` (`p i`) för att se vad loopvariabeln har för innehåll. Första gången är den (föga förvånande) 0. Härifrån kan vi nu be GDB att fortsätta köra programmet med kommandot `continue` (`c`). GDB skriver då ut:

```
Continuing.

Breakpoint 1, main () at debug1.c:41
```

Detta innebär att vi återigen står vid brytpunkt 1. Vi kan verifiera att programmet är i nästa iteration av loopen genom att skriva ut `i` igen. Detta bör ge 1 den här gången. Eftersom vi kommer vilja skriva ut värdet på `i` många gånger kan vi be GDB att skriva ut det efter varje kommando. Detta kan vi göra med kommandot `display i` (`disp i`). GDB svarar då med följande information:

```
1: i = 1
```

Detta innebär att vårt uttryck fick ID 1, och att `i` hade värdet 1. Fortsätter vi i loopen med kommandot `continue` (`c`) så kommer GDB att skriva ut något i stil med följande varje gång:

```
Continuing.

Breakpoint 1, main () at debug1.c:41
1: i = 2
```

Om vi någon gång i framtiden inte längre vill se uttrycket hela tiden kan vi skriva `undisplay 1` (där 1 är ID:t) för att sluta visa uttrycket.

Fortsätter vi stega igenom loopen med `continue` (`c`) så kommer vi se att element nummer 5 aldrig fylls i innan vi kommer till den andra loopen. Det är alltså därför loopen kraschar. Nu kan vi avsluta GDB med kommandot `quit` (`q`) och lösa problemet på lämpligt sätt.

Det finns andra sätt att stega igenom koden än att sätta brytpunkter och låta programmet köra fram till dem. Här är de mest användbara:

- `step` (`s`): Stega till nästa rad kod som körs. Om markören står på ett funktionsanrop så kommer GDB stega in i funktionen (eftersom nästa rad som körs är inuti funktionen). Kallas ibland *step into*.

- `next` (`n`): Stega till nästa rad kod som körs i den nuvarande funktionen. Hoppa alltså över eventuella funktionsanrop på nuvarande rad. Kallas ibland *step over*.
- `finish` (`fin`): Stega till slutet av funktionen som nu körs.

Till sist kan det vara värt att veta att man oftast inte behöver starta om GDB om man har gjort ändringar till sitt program. Om man kompilerar programmet i en annan terminal, och sedan ber GDB att starta om programmet (med `run`) så kommer GDB att inse att programmet har blivit ändrat, och gör sitt bästa för att bevara brytpunkter och dylikt trots ändringarna. Däremot så misslyckas den ibland, och det händer att den kraschar, så ha förväntningarna på en lagom nivå.

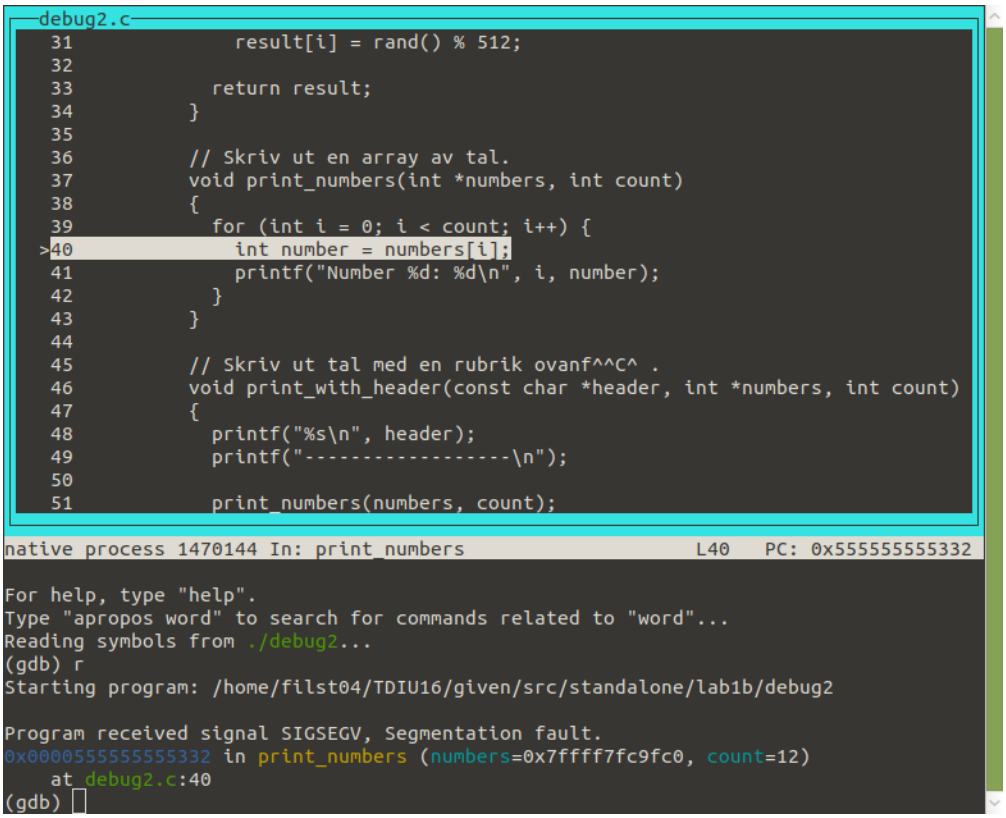
B.2 Program 2: `debug2.c`

Du kan kompilera programmet med kommandot `make debug2` Du kan sedan köra programmet med kommandot `./debug2` Du kommer då se att programmet skriver ut en lista med tal, sedan kraschar det med meddelandet `Segmentation fault` igen.

Denna gång består uppgiften av två delar:

- att lösa problemet så att programmet inte längre kraschar
- att skriva ut tabellen så att alla tal är högerjusterade (dvs. så att alla `:` är på en snygg rad, och sista siffran på varje rad är i samma kolumn)

Likt program 1 kan vi felsöka detta med GDB. Starta GDB med `gdb -tui ./debug2` och kör programmet tills det kraschar (med `run` eller `r`). Detta bör ge ett felmeddelande i stil med bilden nedan (efter att ha tryckt Ctrl+L):



Här kan vi se att GDB skriver ut parametrarna till `print_numbers`, vilket kan vara användbart. Likt tidigare kan vi skriva ut värdet på `i` med `print i` (`p i`):

```
$1 = 0
```

Då ser vi att `i` är 0. Vi kan skriva ut värdet inuti `numbers` för att se vad som står där. Gör vi det så får vi följande svar (adressen är antagligen en annan):

```
Cannot access memory at address 0x7ffff7fc9fc0
```

Detta tyder på att pekaren pekar på minne som inte är giltigt. Frågan är dock var pekaren kom ifrån. Med kommandot `backtrace` (`bt`) kan vi be GDB att visa en så kallad *stack trace*:

```
#0  0x0000555555555332 in print_numbers (numbers=0x7ffff7fc9fc0, count=12)
    at debug2.c:40
#1  0x00005555555553a0 in print_with_header (header=0x555555556032 "Second time:",
    numbers=0x7ffff7fc9fc0, count=12) at debug2.c:51
#2  0x0000555555555406 in main () at debug2.c:62
```

Här ser vi en numrerad lista med så kallade *stack frames*. Just nu är vi i frame nummer 0, eftersom vi står i `print_numbers`. Den anropades av koden i frame nummer 1, inuti `print_with_header`. Denna anropades i sin tur av `main`. Vi kan hoppa till tidigare frames med kommandot `frame` (`f`). Går vi till frame 1 (`frame 1` eller `f 1`) så påminner GDB oss om var vi är:

```
#1  0x00005555555553a0 in print_with_header (header=0x555555556032 "Second time:",
    numbers=0x7ffff7fc9fc0, count=12) at debug2.c:51
```

Här kan vi se att `numbers` kommer direkt som en parameter till `print_with_header`, och att det enda som sker med `numbers` är att den skickas vidare till `print_numbers` (vi ser också att parametern `numbers` innehåller samma pekarvärde som vi såg i `print_numbers`). Det är alltså antagligen inget fel här. Vi går vidare till frame 2 (med `frame 2` eller `f 2`):

```
#2  0x0000555555555406 in main () at debug2.c:62
```

Här kan vi se att samma `numbers` skickas som parameter till båda anrop till `print_with_header`. Vi vet från tidigare att det första anropet lyckades, så vad hände däremellan? Vi lägger en brytpunkt på rad 59 (raden med första anropet till `print_with_header`) med `break 59` (`b 59`) och startar om programmet med `run` (`r`).

Nu kan vi se vad som händer med variabeln `numbers` när vi kör kod. Eftersom vi kommer att vilja se den hela tiden kan vi köra `display numbers` för att skriva ut adressen i variabeln. För att smidigt se vad den innehåller kan vi också lägga till `display numbers[0]`. Just nu borde GDB svara med något i stil med nedan (adresser och tal kan variera):

```
1: numbers = (int *) 0x7ffff7fc9fc0
2: numbers[0] = 299
```

Vi kan nu stega framåt i programmet med kommandot `next` (`n`). När vi hamnar på raden `printf("\n");` får vi följande rapport från GDB (programmet skriver ut saker, så tryck på Ctrl+L för att se en korrekt vy):

```
1: numbers = (int *) 0x7ffff7fc9fc0
2: numbers[0] = <error: Cannot access memory at address 0x7ffff7fc9fc0>
```

Intressant. Adressen i `numbers` är densamma som tidigare (vilket är som förväntat), men vi kan inte längre läsa det första elementet. Någonting hände inuti `print_with_header`! Vi startar om programmet med `run` (`r`) för att komma tillbaka till vår brytpunkt. Denna gång stegar vi in i funktionen `print_with_header` med kommandot `step` (`s`) i stället. GDB skriver ut ett meddelande om att vi är inuti en annan funktion för att informera oss om detta. Eftersom vi nu är i en annan funktion så måste vi säga åt den att visa våra variabler igen (det kanske är något annat som är intressant i en annan funktion!). Vi skriver alltså `display numbers`

och `display numbers[0]` igen. Här borde vi se samma utdata som i `main` tidigare (men kanske med ett annat slumptal):

```
1: numbers = (int *) 0x7ffff7fc9fc0
2: numbers[0] = 299
```

Vi kan nu stega igenom hela `print_with_header` med kommandot `next` (`n`). För att slippa skriva `n` hela tiden räcker det att köra kommandot en gång. Sedan kan man helt enkelt trycka på enter utan att skriva något för att köra föregående kommando igen. På så sätt kommer man snabbt och smidigt igenom hela funktionen. Notera att eftersom funktionen skriver ut saker kan man behöva trycka Ctrl+L ibland.

När vi stegar igenom funktionen så ser vi att utskriften av `numbers[0]` är korrekt ända fram tills efter anropet av `free`. Det är alltså anropet till `free` som är problemet!

Lös nu problemet på lämpligt sätt, och modifiera funktionen `print_numbers` så att den skriver ut talen högerjusterade.

B.3 Program 3: `debug3.c`

Du kan kompilera programmet med kommandot `make debug3` Du kan sedan köra programmet med kommandot `./debug3` Du kommer då se att programmet skriver meddelandet `Strings in C are fun!` och kraschar sedan med meddelandet `Segmentation fault` igen.

Likt tidigare så kör vi programmet i GDB med `gdb -tui ./debug3` och kör det med `run` (`r`). Denna gång visas dock inte någon rad i vår källkod, utan GDB säger bara:

```
Program received signal SIGSEGV, Segmentation fault.
__strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:141
../sysdeps/x86_64/multiarch/strlen-avx2.S: No such file or directory.
```

Det verkar som att funktionen `__strlen_avx2` kraschade när den försökte göra sin uppgift (det råkar vara en implementation av `strlen` i standardbiblioteket, den räknar hur lång en sträng är). GDB säger också att den inte hittade källkoden för funktionen, vilket är varför den inte visar något i den övre delen av skärmen. Vid första anblick skulle man kunna tro att vi har hittat en bugg i standardbiblioteket. Det är dock inte nödvändigtvis fallet, eftersom det är mer sannolikt att vi har använt standardbiblioteket på ett felaktigt sätt. För att se vad som kan vara fel kör vi `backtrace` (`bt`). Då får vi följande svar från GDB:

```
#0  __strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:141
#1  0x00007ffff7df1d15 in __vfprintf_internal (s=0x7ffff7f666a0 <_IO_2_1_stdout_>,
    format=0x555555556028 "Copy:     %s\n", ap=ap@entry=0x7fffffffd5c0,
    mode_flags=mode_flags@entry=0) at vfprintf-internal.c:1688
#2  0x00007ffff7ddad3f in __printf (format=<optimized out>) at printf.c:33
#3  0x00005555555552ca in main () at debug3.c:38
```

Detta är en lista över så kallade *stack frames*. Varje frame motsvarar ett anrop till en funktion. Listan kan se lite kryptisk ut till en början. En bra idé är att leta efter funktionsnamn i raderna efter varje nummer. Här ser vi att frame 0 motsvarar anropet till funktionen `__strlen_avx2` där programmet kraschade. Frame 1 är då funktionen som anropade `__strlen_avx2`, vilket här är funktionen `_vfprintf_internal`. Frame 2 motsvarar i sin tur `__printf` (vilket råkar vara det interna namnet på `printf`), och frame 3 motsvarar `main`.

En annan observation som blir relevant i Pintos senare är att parametern till frame nummer 2 (`__printf`) bara säger `<optimized out>`. Detta händer ibland när man kör med optimeringsflaggor. Då kan kompilatorn ha beslutat att ett värde på en viss variabel inte behövs på ett visst ställe i koden, och därmed kan GDB inte visa det. Det kan också hända när man använder `print`-kommandot eller `display`-kommandot.

När man ser den här typen av fel så är det en bra idé att gå igenom frames tills man hittar kod som man själv har skrivit och börja där. Exempelvis så kan vi börja med att skriva `frame 1` (`f 1`) och se om vi känner

igen koden där. Sedan försöka med `frame 2` (`f 2`) och så vidare tills vi känner igen oss. I just det här fallet så kommer vi inte se den relevanta källkoden förrän vi kommer till `frame 3`, då vi är i `main` på raden när programmet försöker skriva ut `copy` genom att anropa `printf`.

Eftersom programmet kraschade när det försökte skriva ut `copy` kan det vara något problem där. Vi försöker att skriva ut variabeln med `print copy` (`p copy`):

```
$2 = 0x7ffff7fc9fe0 "Strings in C are fun!\314\314\314\314\314\314\314\314\314\314",
<incomplete sequence \314>
```

Här ser vi att GDB är snäll och tolkar variabeln som en C-sträng. Vi ser också att början av strängen är korrekt, men att det är skräp i slutet av strängen. Vi ser också att GDB säger `<incomplete sequence \314>` i slutet. Detta innebär är att den inte hittade tecknet 0 innan den kom fram till ogiltigt minne. Eftersom tecknet 0 används för att indikera slutet av en sträng i C betyder det antagligen att `strlen` (som anropades av `printf`) försökte leta efter tecknet 0, men kraschade innan den hittade det tecknet. Det verkar alltså som att koden i `my_strdup` har misslyckats med att lägga in ett 0-tecken.

Lös probelemet på lämpligt sätt.

B.4 GDB i Pintos

Eftersom Pintos körs i en virtuell maskin kan vi inte bara köra `gdb -tui pintos`. Det skulle instruera GDB att felsöka det program som startar Pintos i en virtuell maskin. Vi måste i stället göra på ett litet annat sätt när vi startar Pintos och GDB för att kunna felsöka det som körs *inuti* den virtuella maskinen.

Nedan följer en kort beskrivning av hur detta sker. Det är inte en del av denna laboration, men det finns här för att belysa att uppstarten är lite annorlunda. Som nämnt ovan så finns ytterligare information om hur felsökning i Pintos sker i Pintos-Wiki. Här följer en kort sammanfattning:

1. Starta GDB med kommandot `pintos-gdb -tui build/kernel.o` (givet att du står i `userprog`)
2. Starta Pintos i en annan terminal. Ersätt kommandot `pintos` i den kommandorad du körde med `debugpintos`, lämna resterande argument som de var. Om du bara vill testa att starta GDB kan du använda kommandot `debugpintos --filesys-size=2 -- -f -q`.
3. Koppla ihop GDB med Pintos genom att skriva `debugpintos` i GDB-prompten.

Värt att notera är att i steg 2 så kommer kommandot `debugpintos` starta en virtuell maskin som är redo att köra Pintos. Den kommer dock inte att börja köra Pintos förrän GDB har anslutits.

I och med att GDB inte kan starta Pintos hur som helst så fungerar inte kommandot `run` som ovan. I stället får man hjälpa GDB genom att starta om Pintos manuellt. Detta kan göras genom att koppla ifrån Pintos med kommandot `kill` (`k`) i GDB, starta om Pintos i den andra terminalen (med samma `debugpintos`-kommando), och till sist köra kommandot `debugpintos` inuti GDB igen. På så sätt kommer GDB att komma ihåg dina brytpunkter, och de variablerna du visade med `display`.

Del C Associativ databehållare

I ett operativsystem behöver man ofta hålla reda på olika resurser och ge dem ”namn” i form av heltal. Därför ska du i denna del implementera en modul som fungerar som en associativ behållare för detta ändamål. Vi kommer exempelvis att använda den för att hålla koll på fildeskriptorer i nästa laboration.

För att illustrera detta, tänk dig att vi vill lagra strängar (`const char *`) i behållaren. Vi vill då kunna göra följande:

```
int main(void) {
  struct map m;
  map_init(&m);                       // initiera behållaren
  int a = map_insert(&m, "hello");    // returnerar exempelvis 0
  int b = map_insert(&m, "world");    // returnerar exempelvis 1
  map_find(&m, a);                    // returnerar "hello"
  map_find(&m, b);                    // returnerar "world"
  map_remove(&m a);                   // returnerar "hello" och tar bort elementet
  map_find(&m, a);                    // returnerar NULL
}
```

I mappen `standalone/lab1c` finns det ett testprogram för detta, samt filer `map.h` och `map.c` för din implementation. I `map.h` är det en bra idé att inkludera `#include <stdbool.h>` för att kunna använda booleans som i C++.

Notera: Tanken med uppgiften är att öva på programmering i C utanför Pintos. Detta gör vi genom att bygga och testa en enkel datastruktur som vi sedan har nytta av i Pintos. Tanken är *inte* att datastrukturen ska vara särskilt komplicerad. Tvärt om är det en bra idé att hålla implementationen så *enkel* som möjligt. Det gör det enklare att fokusera på det som är nytt i kursen — programmering i en kernel, systemanrop, och synkronisering. Sikta på en implementation som är så enkel att du enkelt kan se att den är korrekt genom att titta på koden. En sådan implementation bör inte ta mer än ett par timmar att implementera och testa.

C.1 Funktionalitet

Din modul ska innehålla en datastruktur som heter `struct map`. Den ska ha operationerna som beskrivs nedan. Dessa operationer tar alla en pekare till en instans av datastrukturen. För enkelhets skull använder vi definitionerna `key_t` och `value_t` för att benämna nyckel- och värdetyper. Du kan dock anta att `key_t` alltid kommer vara en heltalstyp (dvs. `int`, `long`, eller liknande), och att `value_t` kommer vara någon form av pekare (ex.vis `char *` eller `struct file *`). De kan definieras som nedan:

```
typedef char* value_t;
typedef int key_t;
```

Funktionerna ska bete sig enligt följande:

- `void map_init(struct map* m);`

  Denna funktion motsvarar en konstruktor i C++. I C måste den anropas manuellt varje gång en variabel av typen `struct map` skapas. Ansvaret för detta ligger på användaren av modulen. `map_init` har därmed till uppgift att initiera alla medlemmar i `struct map`, så att den motsvarar en tom container. Kom ihåg: när variabeln skapas så finns inga garantier på innehållet i dess medlemmar. Vi kan alltså inte ens anta att alla datamedlemmar är 0.

- `key_t map_insert(struct map* m, value_t v);`

  En funktion för att sätta in ett nytt värde i behållaren. För att senare kunna hitta värdet igen ska funktionen hitta en nyckel som inte tidigare användes och returnera den. Denna nyckel ska sedan kunna användas för att hitta värdet igen.

- `value_t map_find(struct map* m, key_t k);`

  Denna funktion ska hämta ett värde som tidigare är insatt med `map_insert`. Om värdet hittas ska det returneras. Annars ska `NULL` returneras.

- `value_t map_remove(struct map* m, key_t k);`

  Denna funktion ska ta bort elementet med nyckeln k ur datastrukturen. Fungerar som `map_find` med skillnaden att den också tar bort elementet.

Utöver ovanstående funktioner kommer det att finnas behov av lite mer avancerad funktionalitet. Detta går att implementera generellt i C med hjälp av funktionspekare (jämför med lambdafunktioner eller funktionsobjekt i C++). All funktionalitet som behövs i labbserien går att implementera med hjälp av de två generella funktionerna nedan, men det går också bra att göra speciella funktioner för de specifika fall som krävs senare.

- ```
  void map_for_each(struct map* m,
                    void (*exec)(key_t k, value_t v, int aux),
                    int aux);
  ```

  Funktionen ska iterera igenom alla element i datastrukturen och anropa funktionen *exec* för varje värde som finns. Parameter 2 är en pekare till en funktion som tar 3 parametrar: en nyckel, ett värde och ett godtyckligt heltal. Den tredje parametern (`aux`) är det värde som skickas med till *exec* vid varje anrop. Exempelprogrammet använder detta för att veta vilken delmängd av nycklarna som ska skrivas ut.

  Tittar vi på andra parametern ser vi att det är en pekare till en funktion som tar tre parametrar och returnerar *void*. Just denna syntax ser lite struligt ut i C, men tänk bort parenteserna runt ordet *exec* och stjärnan framför så ser du att det är en normal funktionsdeklaration man gjort om till pekare.

- ```
  void map_remove_if(struct map* m,
                     bool (*cond)(key_t k, value_t v, int aux),
                     int aux);
  ```

  Funktionen ska fungera precis som `map_for_each`, men om *cond* returnerar *true* så ska elementet tas bort från datastrukturen. Detta är bra om endast vissa associationer ska tas bort, eller om extra steg för att exempelvis frigöra minne krävs innan borttagning. Exempelprogrammet använder detta för att frigöra minne innan elementen frigörs.

C.2 Implementation

Nedan följer två idéer för hur datastrukturen kan implementeras:

Alternativ 1

Det enklaste alternativet är att implementera `map` som en array med en förbestämd storlek. Detta gör det väldigt enkelt att hantera i C (man behöver inte tänka så mycket på minneshantering), men det har såklart problemet att den inte kan växa dynamiskt.

I och med att vi själva bestämmer vad för nyckel som insatta element får, så kan vi helt enkelt bestämma att ett element som är på plats 3 i arrayen får nyckeln 3. Detta gör uppslagning trivialt (titta på rätt plats i arrayen, med relevanta felkontroller innan). Insättning består då av att helt enkelt hitta en ledig plats i arrayen.

I C kan detta implementeras som följer:

```
/* symbolisk konstant för att enkelt kunna ändra storleken senare vid behov */
#define MAP_SIZE 32
```

```
struct map
{
  value_t content[MAP_SIZE];
}
```

Alternativ 2

Om man inte vill begränsa storleken i förväg kan man i stället använda den länkade lista som finns i Pintos (uppgift X3). Nackdelen är att man måste tänka lite mer på minneshantering i och med att alla noder allokeras dynamiskt. Detta kan implementeras enligt följande:

```
struct map_element
{
  key_t    key;  /* nyckeln */
  value_t  value; /* värdet associerat med nyckeln */
  /* list-element för att kunna sätta in i listan */
  struct list_elem elem;
}

struct map
{
  /* listan med alla lagrade element */
  struct list content;

  /* räknare för vilken nyckel som är nästa lediga */
  int next_key;
}
```

Vanliga fel och funderingar

Här finns några vanliga frågor gällande den här labben. Se också ”Vanliga fel och funderingar” i Pintos-wikin för generella fel.

Minnesläcka i den givna `main` Programmet är inte helt färdigt (i och med att `map` inte är helt klar), så titta noga på kommentarerna kring `YOUR CODE` i källkoden, och hur programmet försöker frigöra minnet i slutet av programmet. Fundera på vilken del (`map`-strukturen, eller användaren av den) av programmet som har ansvaret för att frigöra minnet.