TTK4235: Debugging via GDB og Valgrind

Kolbjørn Austreng

Vår 2020

Om øvingen

Denne øvingen handler om strukturert debugging ved bruke av GDB og et profileringsverktøy som heter Valgrind. GDB er en debugger som lar dere gå gjennom koden deres, stoppe vilkårlige steder, inspisere variabler, sette verdier, med mer. Alt dette uten å måtte opprette et mylder av "`printf`s" liberalt strødd rundt om i programmet deres.

Valgrind, derimot, lar dere profilere minnebruken deres - som kan hjelpe dere med å fjerne minnelekkasjer eller oppdage når dere bruker minne dere ikke lenger har allokert (en såkalt "use after free"-feil).

Godkjenning gjøres på Sanntidssalen, og må gjøres innen slutten av uke 7 (14. Februar).

Som tidligere øvinger er ikke terskelen for å få godkjent øvingen i seg selv veldig høy, men jeg kommer til å forvente å se GDB i flittig bruk i løpet av heisprosjektet - så det er lurt å gjøre en god innsats, i tillegg til å referere tilbake til øvingen i løpet av heisprosjektet.

GDB for mikrokontrollere

Det ligger ved et appendiks om debugging av mikrokontrollere. Dette trenger dere ikke tenke på før mikrokontrollerlaben, men er greit å se tilbake på når den tiden kommer.

Til alle som ikke trenger debugging:

Riiiiiight :)

1 Intro

GDB står for **G**NU **Deb**ugger og er et veldig slagkraftig verktøy når til kommer til programvare som ikke fungerer som den skal. Formålet med denne øvingen er å introdusere GDB til særs enkle ting, slik at dere blir eksponert til systematisk debugging før dere virkelig trenger det; på et tidspunkt er `printf` rett og slett ikke tilstrekkelig lenger.

GDB kan først virke litt skummelt, fordi det er et verktøy som i utgangspunktet ikke har noe pent grafisk grensesnitt. Det er derimot mange *wrappere* som gir dette - mest kjent er nok programvaren DDD.
For å ikke kaste dere ut på veldig ukjent farvann, skal vi bruke en "semi-grafisk" visning som kommer med GDB, kalt **T**ext **U**ser **I**nterface. Denne visningen kan startes ved å starte `gdb` med flagget `-tui`, eller ved å trykke `Ctrl + X, A` fra en vanlig `gdb`-instans.

Til slutt skal vi røre så vidt ved et minneprofileringsverktøy kalt Valgrind, som er det beste alternativet for å luke vekk minnerelaterte feil som GDB ikke nødvendigvis finner så lett.

1.1 Oppsett

GDB fungerer ved å kjøre maskinkoden til et kompilert program i bakgrunnen, og vise hvilke kodelinjer dette svarer til i forgrunnen. Ved vanlig kompilering og lenking kastes den såkalte *symboltabellen* bort, så vi trenger en kompilator som kan ivareta denne under kompilering.

Kompilatoren `gcc` har denne muligheten. Deretter trenger vi debuggeren selv, nemlig `gdb`. Begge disse skal være installert på Sanntidssalen. Installasjon av både `gcc` og `gdb` er rett fram på Linux og Mac. På Windows er MinGW[1] den enkleste måten å kjøre programvaren.

2 Oppgave

Gjennom øvingen skal dere kompilere feilaktig kode og debugge den med `gdb`. Ofte vil feilene være så banale at dere rett og slett ser dem, men motstå allikevel fristelsen til å skippe over. Øvingen skal tross alt forberede dere til den vakre dagen dere virkelig trenger en systematisk debugger, og da er det greit å ha basisen under kontroll.

[1] www.mingw.org

2.1 Breakpoints

Breakpoints er sannsynligvis den mest brukte-, og en av de nyttigste egenskapene til en debugger. De tillater deg å midlertidig stoppe kodekjøringen for å inspisere variabler, og liste opp hvordan du kom til linjen du stoppet på.

GDB opererer med tre forskjellige typer breakpoints:

- *Breakpoint*: Stopper kodekjøringen når du kommer til en spesifikk linje.
- *Watchpoint*: Stopper kodekjøringen dersom en spesifisert variabel eller minneadresse endre verdi.
- *Catchpoint*: Stopper kodekjøringen dersom en definert hendelse inntreffer. Hendelsen kan for eksempel være et kall til `exec`, et `syscall`, en `assert`, eller en `exception`.

Av disse er vanlige breakpoints desidert av mest nytte. Selv om watch- og catchpoints av og til kan være praktiske å ha, blir de fort ubrukelige dersom du debugger kode med flere tråder.

For å demonstrere hvordan man vanligvis bruker breakpoints skal vi bruke programmet `break`. Kall `make break` for å kompilere, og `gdb -tui break` for å starte GDB med programmet vårt.
Breakpoints kan settes på forskjellige måter. Skriv nå `break local_call` for å sette et breakpoint på starten av funksjonen `local_call`. Kall deretter `break 9` for å sette et breakpoint til linje 9. Kall så `run` for å starte utføringen av programmet. Dere vil få noe tilsvarende dette:

```
(gdb) break local_call
Breakpoint 1 at 0x64e: file break.c, line 5.
(gdb) break 9
Breakpoint 2 at 0x665: file break.c, line 9.
(gdb) run
Starting program: /home/austreng/tilpdat/assign/debug/break

Breakpoint 2, main () at break.c:11
```

Hva? Hvorfor stoppet vi først på linje 11, når vi klart satte et breakpoint til linje 9? Grunnen er at vi ikke gjør noe spennende på linje 9; vi oppretter kun en variabel, men vi gir den ikke noen verdi. GDB opererer egentlig ikke med linjene kode du ser på toppen av skjermen, men med maskininstruksjonene de har kompilert til. Det kan derfor fint hende at linje 9 ikke svarer til en instruksjon GDB kan stoppe ved. Dette er verdt å ha i bakhodet dersom GDB tilsynelatende ikke gjør det du sier.

GDB har også betingede breakpoints. Altså breakpoints som kun aktiveres dersom en betingelse er oppfylt. Skriv `break 14 if i > 60` for å sette et breakpoint midt i forløkken, som kun skal aktiveres dersom tellervariabelen er større enn 60. Kall deretter `continue` for å gå videre fra linje 11. Dere vil nå stoppe på linje 14, og om dere skriver `print i` vil dere se at tellervariabelen `i` har verdi 61. Altså ble ikke breakpointet aktivert før vi ønsket.

Sett derimot nå at vi i utgangspunktet tok feil; vi ønsket ikke å stoppe når `i` gikk over 60, men 80. Om vi skriver `info break` får vi opp en oversikt over hvilke breakpoints vi har:

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000055555555464e in
↪  local_call
                                                   at
                                                   ↪  break.c:5
2       breakpoint     keep y   0x0000555555554665 in main at
↪  break.c:9
        breakpoint already hit 1 time
3       breakpoint     keep y   0x0000555555554675 in main at
↪  break.c:14
        stop only if i > 60
        breakpoint already hit 1 time
```

Vi kan nå kalle `condition 3 i > 80`, for å endre betingelsen til breakpoint nummer 3 til å kun aktiveres dersom `i` er større enn 80. Nå kan dere kalle `continue`, etterfulgt av `print i`, som vil demonstrere at vi faktisk bare stopper når vi vil:

```
(gdb) condition 3 i > 80
(gdb) continue
Continuing.

Breakpoint 3, main () at break.c:14
(gdb) print i
$2 = 81
```

Vi kan fjerne betingelsen fra breakpoint 3 ved å simpelthen kalle `condition 3`. Om vi nå kaller `continue` vil vi derimot stoppe i forløkken hver gang, fordi `i` nå alltid er større enn 80. Om vi er ferdige med breakpoint 3, kan vi skru det av ved å kalle `disable 3`. Dette vil endre `Enb`-feltet (enable) fra `y` til `n`:

```
(gdb) disable 3
(gdb) info break
```

```
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000055555555464e in
↪  local_call
                                                   at
                                                   ↪  break.c:5
2       breakpoint     keep y   0x0000555555554665 in main at
↪  break.c:9
        breakpoint already hit 1 time
3       breakpoint     keep n   0x0000555555554675 in main at
↪  break.c:14
        breakpoint already hit 2 times
```

Vi kan også permanent slette breakpoint 3 ved å kalle `delete 3`, om vi vet vi ikke har behov for det igjen.

GDB kan også sett breakpoints i andre filer, som for eksempel filen `break2.c`, ved å bruke en `<fil>:<linje eller funksjon>`-syntaks:

```
(gdb) break break2.c:library_call
Breakpoint 4 at 0x55555555469a: file break2.c, line 4.
```

Om man kun trenger å stoppe et sted én gang, så har GDB midlertidige breakpoints, som på samme måte som vanlige breakpoints, men med kodeordet `tbreak` istedenfor `break`:

```
(gdb) tbreak library_call
Note: breakpoint 4 also set at pc 0x55555555469a.
Temporary breakpoint 5 at 0x55555555469a: file break2.c, line
↪  4.
```

Om du nå kaller `info break` vil du se at `Disp`-feltet (disposition) er forskjellig fra vanlige breakpoints:

```
4       breakpoint     keep y   0x000055555555469a in
↪  library_call
                                                   at break2.c:4
5       breakpoint     del  y   0x000055555555469a in
↪  library_call
                                                   at break2.c:4
```

Her betyr `keep` at breakpointet vil beholdes, mens `del` indikerer at et breakpoint vil slettes etter at det blir truffet.

En siste ting som er greit å vite om breakpoints er at du kan tilknytte vilkårlige kommandoer til hvert enkelt breakpoint, som vil kjøre hver gang breakpointet aktiveres. Om du for eksempel ønsker å legge inn en `printf`, uten å måtte endre koden og gjenkompilere, kan du gjøre slik:

```
(gdb) break 14
Breakpoint 8 at 0x555555554675: file break.c, line 14.
(gdb) command 8
Type commands for breakpoint(s) 8, one per line.
End with a line saying just "end".
>printf "Value of i is %d\n", i
>end
```

Dette vil skrive ut verdien til `i` på samme måte som "gammeldags printf-debugging" gjør, men du slipper altså å tukle med koden og kompilere på nytt. GDB kan faktisk kall en hvilken som helst funksjon - også bibliotek-funksjoner - via breakpointkommandoer. Det skal ikke mye fantasi til for å se at dette er en uhyre kraftig egenskap!

2.2 Inspeksjon og endring av variabler

GDB kan inspisere og endre variabler under kjøring. For å illustrere hvordan dette kan gjøres, skal vi bruke programmet `inspect`. Kall `make inspect`, og deretter `gdb -tui inspect` for å starte GDB.

Sett opp et breakpoint i slutten av `main` ved å kalle `break 21`. Kall deretter `run`. Når GDB stopper vil det være like før programmet returnerer. Kall nå `print stat_array` og `print dyn_array`:

```
Breakpoint 1, main () at inspect.c:21
(gdb) print stat_array
$1 = {1, 2, 3, 4, 5}
(gdb) print dyn_array
$2 = (int *) 0x555555756260
```

Som dere ser av koden, inneholder `stat_array` og `dyn_array` den samme informasjonen, men de er allokert henholdsvis statisk og dynamisk. Når vi printer den dynamiske tabellen, får vi bare pekeren. Dette skjer fordi GDB ikke kan være sikker på nøyaktig hva som ligger bak pekeren.

Vi er derimot sikre på hva pekeren skjuler, så vi kan utnytte en spesiell syntaks for å skrive den ut som et array:

```
(gdb) print *dyn_array@5
$3 = {1, 2, 3, 4, 5}
```

Dette betyr bare at vi skal dereferere pekeren, og ta 5 elementer, og skrive dem ut som en tabell. Det er ikke veldig ofte man trenger denne syntaksen, men det er greit å vite at den finnes.

GDB kan også skrive ut structs, som dere vil se om dere kaller `print sleipner`:

```
(gdb) print sleipner
$4 = {legs = 8, can_fly = 1}
```

GBD kan også endre variabler underveis, ved å bruke kodeordet `set`:

```
(gdb) print some_number
$5 = 1
(gdb) set some_number = 2
(gdb) print some_number
$6 = 2
```

Om dere nå lurer på hva $-tegnet betyr, så er det simpelthen en referanse til GDB sin variabelhistorie. Tallet bak $ vil ganske enkelt øke med én for hver gang dere skriver ut noe.

2.2.1 Display

GDB har også en kommando kalt `display`. Den fungerer på samme måte som `print`, men vil automatisk skrive ut en variabel hver gang et breakpoint aktiveres, så lenge variabelen finnes på øverste stackframe. For å stoppe en aktiv `display` brukes naturlig nok kommandoen `undisplay` med IDen til `display`et.

For å prøve ut `display` kan dere kalle `break 14` for å opprette et breakpoint ved linja `dyn_array[i] = i + 1;`, deretter `display *dyn_array@5`, og til slutt `run` for å starte programmet på nytt. Hver gang dere treffer breakpointet vil dere se at ett nytt element er blitt satt i `dyn_array`. For å slippe å skrive `continue` mange ganger kan dere forresten bare trykke enter for å repetere siste kommando som ble kjørt.

2.3 Step og Next

GDB kan, som dere sikkert forventer, gå gjennom kode linje for linje. Kompiler og debug programmet `step` ved å kalle `make step` etterfulgt av `gdb -tui step`. I GDB kaller dere `break main`, fulgt av `run`. Programmet vil nå stoppe på linjen `int prod1 = multiply(2, 6);`. Kall nå kommandoen `step` fire-fem ganger og legg merke til hva som skjer.

Deretter starter dere programmet på nytt ved å kalle `run` igjen. Denne gangen kaller dere `next` istedenfor `step` et par ganger. Som dere ser, fungerer de to kommandoene litt annerledes:

- `step` vil gå en kodelinje videre, og for hvert funksjonskall den møter på, vil `step` gå inn i implementasjonen.
- `next` vil gjøre det samme som `step`, men vil glatt kjøre alle funksjonskall den møter i bakgrunnen - uten å gå inn i implementasjonen.

Når du kaller `step` vil den kun gå inn i en implementasjon dersom det finnes et symbol for funksjonen. Dermed vil den gå inn i all kode kompilert med flagget `-g`, men hoppe over ting den ikke kan lese. Dermed vil ikke `step` prøve å gå inn i implementasjonen til for eksempel `printf`.

2.4 Until og Finish

Om du kommer inn i en løkke, eller havner i en funksjon du er sikker på er feilfri, så er det greit å kunne hoppe ut av den. For dette formålet har GDB kommandoene `until` og `finish`. Kall først `make until`, og deretter `gdb -tui until`. Inne i GDB oppretter dere et breakpoint for `main` ved å kalle `break main`.

Kall så `run` for å starte utføringen av programmet. Dere vil nå stoppe i toppen av forløkken, fordi dette er den første interessante linjen kode i main. Vi ønsker å hoppe over løkken, men fortsette rett etter den. Kall `until` tre ganger. Dere vil se at GDB nå stopper på linje 21 - rett før `pointless_function` kalles.

Kommandoen `until` skal hoppe til kodelinjen som er "1 større enn nåværende kodelinje". Grunnen til at dere måtte kalle `until` tre ganger istedenfor én, er at GDB som sagt arbeider med maskinkoden til programmet. I C-koden ser det ut som om vi gjør en test i starten av løkken - men i den ferdigkompilerte koden skjer faktisk testen i bunnen av løkken. Dette er ikke veldig viktig å huske på, men det er greit å vite dersom en kommando tilsynelatende oppfører seg litt rart.

Når dere så har kommet til `pointless_function`, kaller dere `step` for å gå inn i implementasjonen. Sett nå at dere har gjort dere ferdig inne i denne funksjonen og ønsker å komme tilbake til `main`. Dere kunne selvsagt brukt midlertidige breakpoints og `continue`, men dette er tungvint. GDB definerer kommandoen `finish`, som dere heller kan bruke. Denne kommandoen vil fullføre et funksjonskall og stoppe kodeutføringen rett etter funksjonen returnerer.

2.5 Callstack

Ofte er tilfellet at en variabel uventet skifter verdi, eller at en funksjon kalles ut av det blå. Gjerne skjer dette fordi man har en lang linje av funksjoner etter hverandre, der én av dem er feil. Om man forsøker å finne en slik bug ved hjelp av `printf`, så har man virkelig sysselsatt seg selv i lang tid fremover. For å gjøre jobben enklere, kan GDB skrive ut kallstacken, eller hver enkelt *stackframe*.

Kall først `make trace` for å kompilere programmet `trace`, og så `gdb -tui trace` for å debugge det. Fra GDB kaller dere så `watch global_value` for å stoppe når den globale variabelen `global_value` på mystisk vis skifter verdi. Deretter kaller dere `run` for å kjøre programmet. Dere vil få en output som ligner på denne:

```
(gdb) watch global_value
Hardware watchpoint 1: global_value
(gdb) run
Starting program: trace

Hardware watchpoint 1: global_value

Old value = 0
New value = 1
myst_func_2 (level=0) at trace.c:63
```

Variabelen skifter altså verdi fra 0 til 1, og i dette tilfellet skjedde det før linje 63. Vi kan ikke være helt sikre på nøyaktig hvilken linje endringen skjedde, nok en gang fordi GDB egentlig jobber med maskinkoden til programmet, og prøver deretter så godt det kan å koble maskinkode til C-kode. Heldigvis er det ikke så veldig interessant hvilken linje endringen skjedde på, så lenge vi iallefall får en anelse.

Kall nå `backtrace`. Dette vil skrive ut kallstacken til programmet - altså hvilke funksjonskall som har hendt til nå. Dere vil få noe som ser slik ut:

```
(gdb) backtrace
#0  myst_func_2 (level=0) at trace.c:63
#1  0x00005555555548bb in myst_func_3 (level=1) at trace.c:76
#2  0x00005555555548ca in myst_func_3 (level=2) at trace.c:79
#3  0x0000555555554957 in myst_func_4 (level=3) at trace.c:102
#4  0x00005555555547bf in myst_func_1 (level=4) at trace.c:36
#5  0x00005555555548ac in myst_func_3 (level=5) at trace.c:73
#6  0x00005555555548ca in myst_func_3 (level=6) at trace.c:79
#7  0x00005555555547b0 in myst_func_1 (level=7) at trace.c:33
#8  0x0000555555554939 in myst_func_4 (level=8) at trace.c:96
#9  0x00005555555548d9 in myst_func_3 (level=9) at trace.c:82
#10 0x00005555555547b0 in myst_func_1 (level=10) at trace.c:33
#11 0x0000555555554989 in main () at trace.c:113
```

Helt på bunnen vil dere se at utgangspunktet vårt var `main`, som så kalte `myst_func_1` med `level`parameter lik 10. Deretter ser vi at en rekke kall til forskjellige `myst_func`-funksjoner blir gjort, før vi kommer til `myst_func_2`, som må ha endret `global_value`. En slik utskrift av kallstacken kan gi en

veldig god indikasjon på hvor ting går galt.

GDB kan også skrive ut en spesifikk *stackframe*. En stackframe er ett nivå på kallstacken, og inneholder informasjon om hvilken funksjon som ble kalt, og hvilke argumenter som ble gitt. Kall `frame` for å skrive ut øverste stackframe, eller for eksempel `frame 8` for å skrive ut en stackframe 8 nivåer ned:

```
(gdb) frame
#6  0x00005555555548ca in myst_func_3 (level=6) at trace.c:79
79                                          myst_func_3(level -
↪  1);
(gdb) frame 8
#8  0x0000555555554939 in myst_func_4 (level=8) at trace.c:96
96                                  myst_func_1(level - 1);
```

2.6 Segfault

Kompiler og kjør programmet `seg1`, ved å kalle `make seg1`, etterfulgt av `./seg1`. Dere vil få en segmentation fault - altså forsøker programmet å tukle med minne det ikke eier.

For å undersøke hva som er feil, kjører vi det under GDB ved å kalle `gdb -tui seg1`. Forsøk først å kjøre programmet ved å kalle `run`. GDB vil skrike noe i duren av:

```
Program received signal SIGSEGV, Segmentation fault.
0x000055555555461e in main () at seg1.c:7
```

Dette forteller oss at vi gjorde en ulovlig minneoperasjon i seg1.c, på linje 7. Vi har allerede bare fra dette mye å gå på. Kall deretter `print i`, for å skrive ut verdien av tellervariabelen. Den eksakte verdien er avhengig av arkitektur, men den er i alle fall negativ. Dette er spennende, men la oss late som om vi fortatt ikke er helt sikre på hvorfor.

Kall deretter `break 7`, for å sette opp et breakpoint på linje 7. Så kaller dere `info break` for å få en liste med informasjon om breakpointene dere har i programmet. Dere vil ha noe som ser slik ut:

```
(gdb) break 7
Breakpoint 1 at 0x555555554607: file seg1.c, line 7.
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555554607 in main at
↪  seg1.c:7
```

Vi skal nå knytte kommandoer til breakpointet vår, som skal kjøre hver gang vi kommer til det. Kall `command 1` for å komme inn i kommandomodus for breakpoint 1. Deretter skriver dere `silent` på en linje, `info local` på neste linje, og til slutt `end` på siste linje - slik:

```
(gdb) command 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>silent
>info local
>end
```

Om dere nå kaller `info break` igjen vil dere se at breakpoint 1 har fått to kommandoer knyttet til seg:

```
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555554607 in main at
↪  seg1.c:7
        silent
        info local
```

Her betyr `silent` at GDB ikke vil skrive ut informasjon om breakpointet hver gang det aktiverer, og `info local` betyr at vi ønsker å skrive ut de lokale variablene som finnes til skjermen for inspeksjon. Start deretter programmet på nytt ved å kalle `run`.

GDB vil nå stoppe på linje 7, og skrive ut `i = 0`. Kall deretter `continue` for å gå til neste løkkeiterasjon. Trykk så enter et par ganger - dette vil repetere siste kommando, altså `continue`:

```
i = 0
(gdb) continue
Continuing.
i = -1
Continuing.
i = -2
Continuing.
i = -3
Continuing.
i = -4
```

Det er nå åpenbart hva som foregår - vi teller nedover, når vi skulle ha telt oppover. Dette kommer jo selvsagt av at vi har skrevet `i--` i løkken, mens vi mente `i++`.

2.7 Segfault, nei... wait what?

Kompiler programmet `seg2` ved å kalle `make seg2`. Kjør det deretter ved å kalle `./seg2`. Ta deretter en titt på koden i `seg2.c`. Burde dere ikke fått en segfault her?[2]

Det er helt klart hva programmet vårt gjør: Det allokerer ved *compile time* et array av heltall, med størrelse 200. Deretter skriver vi over 240 av disse elementene. Altså skriver vi til 40 minnesegmenter som vi ikke egentlig eier. Til slutt skiver vi ut "Exit normally", om vi ikke har fått en segfault før nå.

Moralen bak dette programmet er: selv om et program kjører uten segfaults, kan du ikke være sikker på at alt du gjør med minnet er lovlig.

2.7.1 Forklaring

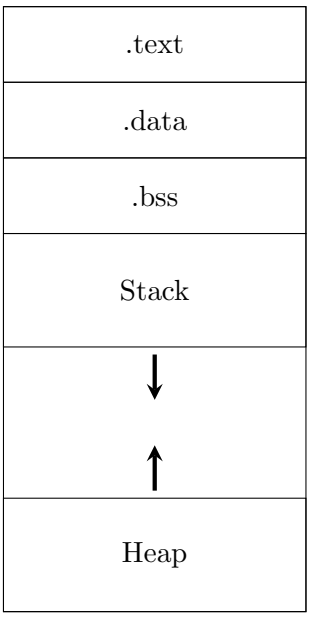

Figure 1: Vanlig minneutseende for et program.

Når dere kompilerer et program vil dere til slutt ende opp med noe som ligner på figur 1. Programmet er delt opp i områder med forskjellige rettigheter. Feltet `.text` er minneområdet som inneholder maskinkoden som faktisk kjører, samt statisk lenkede biblioteker. Dette området krever åpenbart lese- og kjørerettigheter, men vi har virkelig ikke lyst til å gi området skriverettigheter - fordi programmet da potensielt kunne ha skrevet om seg

[2] Avhenging av hvilken arkitektur dere kjører, kan dere faktisk få en segfault, men det er helt opp til hvilken type prosessor dere har.

selv.

Feltet `.data` og `.bss` inneholder henholdsvis *compile time* initialisert- og allokert, men uinitialisert minne. Disse feltene er dermed for globale variabler.

Heapfeltet er området minne kall til `malloc` og `free` vil virke på. Heapen er for dynamisk allokert minne, og vil vokse oppover dersom et kall til `malloc` bruker opp allokert heapstørrelse.

Ovenfor heapen ligger området for stacken. Denne vil bestå av såkalte stackframes med lokale variabler. Om du gjør uendelig rekursjon i et språk som ikke støtter *tail recursion*, så vil stacken vokse nedover til den til slutt kolliderer med heapen. Dette er det som kalles en "stack overflow".

Nøyaktig hvor dynamisk lenkede biblioteker ligger, og hvor kopier av eksterne variabler finnes, er implementasjonsspesifikt - men de generelle trekkene i figur 1 stemmer som regel.

Når dere kjører et program vil dere som regel ikke jobbe med sekundærminne som en harddisk direkte. Grunnen er at dette ville vært svært ineffektivt - så datamaskinen vil laste inn deler av sekundærminne til primærminne, altså RAM. Disse delene kalles *memory pages*, og størrelsen på disse er selvsagt avhengig av arkitektur - fordi ingenting kan være enkelt. Poenget er allikevel at operativsystemet vil gi hver slik *page* diverse flagg for å holde styr på hva du kan gjøre med minnet.

Uheldigvis er det ganske vanlig at du har et array som ikke tar en hel *page*. Dermed kan du gjøre ulovlige minneoperasjoner så lenge du er "heldig" å treffer i en *page* hvor du har aksessrettigheter. En *out-of-bounds* på ett element vil derfor ikke alltid gi en segfault, som det burde.

Simpelthen fordi et program ikke genererer en segfault betyr dermed ikke at det gjør ting det ikke burde med minne.

2.8 Minnelekkasje

Det er alltid litt skummelt å bruke `malloc` i C, eller tilsvarende `new` i C++. Begge disse vil allokere minne fra *heapen*, som ikke frigjøres før du eksplisitt kaller `free` eller `delete`. Om du har et program som glemmer å frigjøre minne, og det kjører over lang nok tid - så kommer nok ting til å bryte sammen til slutt. Dette fenomenet kalles minnelekkasje.

Et av de mest effektive verktøyene for å bekjempe denne utingen kalles

`Valgrind`. Valgrind emulerer en virtuell CPU og kjører programmet ditt på denne. Det vil naturlig nok gjøre at ting kjører smertefullt sakte, men til gjengjeld vil du ha full kontroll over hva som allokeres av minne - og hva som gis eller ikke gis tilbake.

Kall `make leak` fra kommandolinjen for å kompilere programmet `leak`. Kjør det deretter ved å kalle `./leak 12`. Programmet vil nå skrive ut de 12 første Fibonaccitallene - og om du ikke vet hvordan koden ser ut, har du ingen grunn til å mistenke ugler i mosen (vel, det at programmet heter `leak` gir kanskje noen indikasjoner).

Kjør deretter kommandoen `valgrind --leak-check=yes ./leak 12`. Nå vil Valgrind kjøre programmet, men holde styr på hvor mye minne som allokeres og frigjøres. Dere vil få en output som minner om denne:

```
==6232==
==6232== HEAP SUMMARY:
==6232==     in use at exit: 48 bytes in 1 blocks
==6232==   total heap usage: 2 allocs, 1 frees, 1,072 bytes
↪  allocated
==6232==
==6232== 48 bytes in 1 blocks are definitely lost in loss
↪  record 1 of 1
==6232==    at 0x4C2CEDF: malloc (vg_replace_malloc.c:299)
==6232==    by 0x108745: create_array (leak.c:5)
==6232==    by 0x10881A: main (leak.c:23)
==6232==
==6232== LEAK SUMMARY:
==6232==    definitely lost: 48 bytes in 1 blocks
==6232==    indirectly lost: 0 bytes in 0 blocks
==6232==      possibly lost: 0 bytes in 0 blocks
==6232==    still reachable: 0 bytes in 0 blocks
==6232==         suppressed: 0 bytes in 0 blocks
==6232==
==6232== For counts of detected and suppressed errors, rerun
↪  with: -v
==6232== ERROR SUMMARY: 1 errors from 1 contexts (suppressed:
↪  0 from 0)
```

Dette er en oppsummering av heapforbruk i løpet av programmets levetid. Som dere ser, mister vi 48 byte fordi vi ikke frigjør allokeringen vi gjør. Valgrind hjelper oss også med å se hvor lekkasjen skjer; vi kan se av *stacktracen* at vi fra `main` kaller `create_array`, som deretter kaller `malloc` uten at `free` kalles noe sted.

Valgrind har ganske mange finurlige funksjoner, men `--leak-check=yes` er uten tvil den mest brukte.

2.8.1 Trivia

Valgrind uttales slik du ville gjort det på Norsk - altså ikke "Wall-grind". Navnet har sitt opphav i Norrøn mytologi, der Valgrind er navnet på hovedporten til Valhall; drikkehallen til Odins Einherjere.

Skaperene av Valgrind ville faktisk opprinnelig kalle programvaren "Heimdall", etter æsen som vokter Bifrost; reinbuebroen inn i Åsgard. Dette navnet var derimot allerede tatt av en sikkerhetsprogramvare.

2.9 Hva nå?

Som dere sikkert begynner å innse er GDB er latterlig kraftig verktøy som gjør debugging mye enklere enn å legge til og fjerne `printf` fra et utall steder i koden. Faktisk er GDB så kraftig at du kan kaste ut C og programmere nesten utelukkende i GDB direkte. Helt sant. Om du av en eller annen grunn ønsket å gjøre det, ville du brukt noe GDB kaller *convenience variables* for å lagre data, breakpoint commands for å utføre operasjoner, og en hel del stygge hacks med *.gdbinit*-filen. Før du forsøker, tenk deg derimot godt om; hva ville du da brukt for å debugge feilaktig GDB-programvare?

Når du føler deg komfortabel med en kommando, kan du tillate deg å være litt slepphendt med hva du skriver. Så lenge det GDB leser inn fra brukeren ikke er tvetydig, godtas overraskende mye vrøvl. For eksempel kan du skrive "`p i`" istedenfor "`print i`", eller "`i lo`" istedenfor "`info locals`". Om du bruker entall eller flertall i en rekke kommandoer (e.g. "breakpoint" vs "breakpoints") er også ofte revnende likegyldig.

Om du noen gang skulle trenge hjelp, har GDB en grei `help <topic>` kommando som enkelte ganger kan være kjekk å ha. Stort sett er derimot StackOverflow enklere.

A GDB og mikrokontrollere

Det er helt mulig å bruke GDB for å debugge mikrokontrollere, så lenge du har en måte å kommunisere debuggingssignaler til hardwaret. Hardwaret må også støtte debugging, men dette er stort sett garantert i nye mikrokontrollere. Det er umulig å fortelle hvordan man går frem for enhver tenkelig plattform, med ulike debuggingsgrensesnitt - men her er en rask oppskrift som viser hvordan man kan gå frem for å bruke SWD (single wire debugging) for å debugge micro:biten som er brukt i faget. For alle andre plattformer dere måtte komme i kontakt med, er det nok enklest å google "how to set up GDB server on *xxx*".

A.1 Server

Det første vi trenger i det generelle tilfellet er et serverprogram som kjører på målhardwaret vi ønsker å debugge. Dette vil tillate en GDB-klient å koble seg til over en seriell linje, eller over en TCP/IP-forbindelse. Fordi vårt hardware (micro:biten) allerede støtter SWD (single wire debugging) over JLink, trenger vi derimot ikke å kompilere serveren selv. Istedenfor kan vi utnytte andre verktøy som er i stand til å bruke SWD, og så koble GDB-klienten til dette verktøyet. Dette er forsøkt illustrert i figur 2.

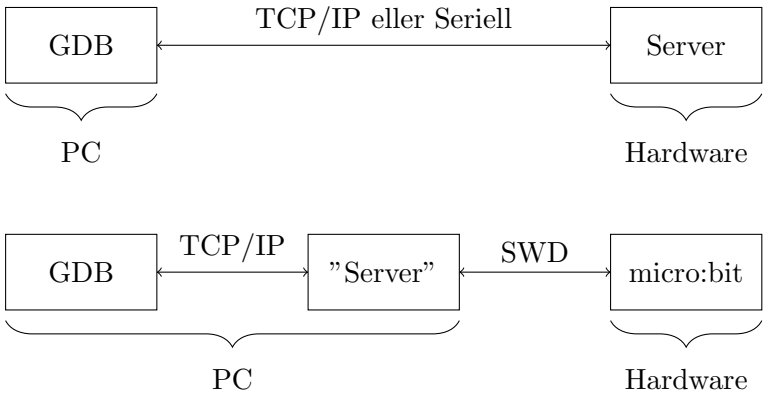

Figure 2: Skjematisk illustrasjon av GDB sitt grensesitt mot hardware.

Her tar "Server"-blokken seg av kommunikasjon med både GDB, og hardware, ved å abstrahere bort hvilken type forbindelse vi har mellom datamaskinen og plattformen vi debugger.

I prinsippet kan "Server" være hva som helst, så lenge det er et program som støtter kommandoer fra GDB, og er i stand til å kommunisere med

målhardwaret vårt. OpenOCD (Open On-Chip-Debugger) er et godt alternativ for dette, og er programvaren vi vil bruke i dette eksempelet.

Før vi kommer til OpenOCD må vi derimot flashe hardwaret med kode. I utgangspunktet kan GDB flashe kode for oss, men dette er dessverre ikke støttet av JLink firmwaren vi kjører på micro:biten.

A.2 Flash koden vi ønsker å debugge

I eksempelmappen ("`example_microbit`") ligger det litt forholdsvis triviell kode som dere kan kompilere og flashe ved å kalle `make; make flash`. Dere vil da se at LED-matrisen på micro:biten skrur seg på, men virker litt dimmet - dette er fordi vi toggler hver rad så fort mikrokontrolleren kan.

A.3 OpenOCD

OpenOCD står som sagt for Open On-Chip-Debugger, og lar oss spesifisere hvordan vi skal prate med hardware over en form for kobling som støtter debugging - eksempelvis JTAG, CMSIS-DAP, USB-Blaster, eller i vårt tilfelle; SWD.

Før vi kan bruke OpenOCD, må det naturlig nok være installert. Det kan dere gjøre ved å kalle `sudo apt install openocd` på datamaskinene på Sanntidssalen - eller tilsvarende kommandoer på andre varianter av Linux.

OpenOCD kan startes på to måter. Dere kan enten spesifisere alle parametrene nødvendig for å definere kommunikasjonsprotokollen via kommandolinjen, eller dere kan opprette en fil kalt `openocd.cfg` i mappen dere har prosjektet deres. For gjenbrukbarhetens skyld er det bedre å benytte seg av denne configfilen.

I vårt tilfelle skal `openocd.cfg` se slik ut:

```
interface jlink
transport select swd

source [find target/nrf51.cfg]

gdb_memory_map enable
```

Deretter starter dere koblingen mellom datamaskin og micro:bit ved å kalle `sudo openocd` fra samme mappe som configfilen ligger. Dere vil da få en output som ligner denne:

```
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
```

```
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
cortex_m reset_config sysresetreq
adapter speed: 1000 kHz
Info : No device selected, using first device.
Info : J-Link OB-BBC-microbit compiled Mar 24 2017 09:33:30
Info : Hardware version: 1.00
Info : VTarget = 3.300 V
Info : clock speed 1000 kHz
Info : SWD DPIDR 0x0bb11477
Info : nrf51.cpu: hardware has 4 breakpoints, 2 watchpoints
```

Dette lover kjempebra. OpenOCD er nå klar for å motta forespørsler over TCP/IP, som GDB kan koble seg til via. Om dere ikke spesifiserer noe annet, vil OpenOCD lytte på port 3333.

Med litt rolig port-forwarding kunne dere nå debugget over internett - limitless possibilities! Vi skal derimot beholde beina på bakken, og kun gjøre noen enkle greier lokalt.

Dere kan ikke bruke `nrfjprog` og `openocd` samtidig. Dere må altså lukke `openocd` før dere kaller `make flash`. Dette er igjen grunnet begrensninger i JLink-firmwaren på micro:biten.

A.4 GDB

Nå kan dere fra samme mappe starte GDB, ved å kalle `arm-none-eabi-gdb build/main.elf -iex "target remote localhost:3333" -tui`. Grunnen til at vi trenger `arm-none-eabi`-prefikset, er at dette er GDB bygget for ARM-baserte arkitekturer, i motsetning til x86-64, som vi ellers kjører på.

Fra GDB kan dere nå kalle `monitor reset halt`, etterfulgt av `break main`, og deretter `continue`. Programmet på mikrokontrolleren vil nå kjøre gjennom oppstartskoden og stoppe ved første instruksjon i `main`; `for`-løkken som konfigurerer matrisen på micro:biten.

Dere kan nå kalle `until` et par ganger, til dere stopper på første linje i `while`-løkka. Hver kommando i GDB kan "lagge" litt, fordi micro:biten ikke akkurat er definisjonen av kraftig hardware, og dermed faktisk bruker litt tid på å utføre operasjoner. Når dere er kommet ned til `GPIO->OUTSET...`, kan dere bruke `next` til å steppe gjennom hver operasjon og se at matrisen

på micro:biten skrur seg på- og av én rad av gangen.

Stort sett er alle operasjonene dere kan gjøre i "vanlig" GDB nå tilgjengelige for dere på mikrokontrolleren. I tillegg til disse, har dere også en del hardwarespesifikke kommandoer dere kan bruke; disse starter med `monitor [...]`, eksempelvis kommandoen `monitor reset halt`, som vil starte målet på nytt, og med en gang fryse programflyten.