Workshop 1

Host OS

Vi benytter os primært af operativsystemet Linux Mint fordi alle i gruppen har det installeret i en virtuel maskine, men programmet er også blevet kørt og testet i Arch Linux, hvor det også virker fint. Linux Mint har systemkravene:

- 2GB RAM (4GB anbefalet for behageligt brug)
- 20GB Lagerplads (100GB anbefalet)
- 1024x768 resolution (På mindre resolution kan man holde ALT knappen inde for at trække i vinduer der ikke passer på skærmen)

Kompilering

Siden opgaven er kodet i C, så kræver det at der laves makefiler til at kompilere filerne. Her defineres flag for gcc kommandoen i command line således at der kan kompileres med –c og –Wall bare ved brug af $(CFLAGS) variablen. Dette minder om high level programmeringssprog som vi kender det.
En anden del af kompileringen som er vigtigt er funktionerne der bliver brugt. Der kan gøres brug af en funktion til at kompilere alle filer til .o filer (objekt filer) ved brug af –c flaget, der kan bruges til at linke flere filer sammen når man bruger -o flaget til at kompilere til en binær fil.
I sidste ende kommer vores makefile til at se ud som på Listing 1:

```
1  CC= gcc
2  CFLAGS = -c -Wall
3  OFLAGS = -lm
4
5  all: TempStub main
6
7  TempStub : TempStub.o
8    $(CC) TempStub.o $(OFLAGS) -o $@
9
10 TempStub.o : TempStub.c
11   @echo "compile $<"
12   $(CC) $(CFLAGS) $<
13
14 main : main.o
15   $(CC) main.o -pthread -o $@
16
17 main.o : main.c
18   @echo "Compile $<"
19   $(CC) $(CFLAGS) $<
20
21
22 clean:
23   @echo "sletter nu"
24   rm -fr *.o
```

Listing 1: Makefile

Processer og threads

Temperaturstub

Tempraturstub er en proces der genererer temperatur data hvert sekund og en gennemsnit af temperaturen over det sidste minut.

Temperatur Generation

Tempraturstub starter med at lave en tempartur float og tilfældig giver den en værdi mellem 0 og 30. Derefter bliver der genereret to tilfældige float værdier mellem -1 og 1 ved brug af:

$$2\left(\frac{rand()}{RAND\_MAX}\right) - 1 \tag{1}$$

Hvor rand() er en funktion i standard C library, som generer et ligefordelt nummer mellem 0 og RAND_MAX. Værdien RAND_MAX er det højeste tal rand() genererer.

```
/* The largest number rand will return (same as INT_MAX).  */
#define RAND_MAX 2147483647
```

Listing 2: RAND_MAX definition i stdlib.h

Ved at dividere rand() med RAND_MAX, så bliver der genereret en værdi mellem 0 og 1. Denne værdi bliver ganget med 2, og så bliver 1 trukket fra den for at få en float værdi mellem -1 og 1.
De to tilfældige værdier bliver brugt til en Marsaglia polar method algoritm til at generere to normalfordelt værdier.[1]

```
// Algorithm to create randomly distributed numbers using Marsaglia
    Polar Method
do
{
    // Generating numbers between -1 and 1
    x = 2*((double) rand()/(RAND_MAX))-1;
    y = 2*((double) rand()/(RAND_MAX))-1;
    // Checking if the random values lie in the unit circle
}while(s > 1 || s == 0);
// Computing the two normally distributed random numbers
S = sqrt((-2*log(s))/s);
R1 = (x*S)/4;
R2 = (y*S)/4;
```

Listing 3: Kode til Marsaglia Polar Method Algoritm

De to nummer R1 og R2 der bliver genereret bliver brugt til at opdaterer temperaturen for de næste to sekunder. Dette sker ved at tilføje R1 eller R2 til temperaturen til at opdaterer det, eller trække dem fra temperaturen hvis de ville gør at temperaturen bliver større end 30 eller mindre end 0.
Derefter tilføjes den opdateret temperatur til et array af størrelse 60. Dette array, som hedder TempArr, indeholder de sidste 60 temperatur værdier, og bliver brugt til at regne gennemsnit af temperaturen fra det sidste minut.

[1] https://en.wikipedia.org/wiki/Marsaglia_polar_method

```
// Update Temperature using first random number
sleep(1);
if ((Temp + R1) > 30 || (Temp + R1) < 0)
{
  Temp -= R1;
}
else
{
  Temp += R1;
}
TempArr[Index] = Temp;
Index = (Index + 1) % ARRSIZE;
Sum = 0;
for(int i = 0; i < ARRSIZE; i++)
{
    Sum = Sum + TempArr[i]
}
Avg = Sum/ARRSIZE;
.
.
.
// Same as above with second random number
```

Listing 4: Kode til at opdatere temperatur og regne gennemsnit

Inter Process Communication (IPC)

Disse temperaturværdier skal passes til læse/skrive processen igennem en Inter Process Communication (IPC) metode. Der blev valgt at bruge named pipes (FIFO filer) til IPC. En named pipe er en pipe som er repræsenteret med en fil i filsystemet. Denne fil er brugt kun som en reference til den named pipe, og der bliver ikke skrevet noget i den. En pipe er en envejs IPC kanal, hvor der er en skriver ende og en læser ende.
Når en process skal arbejde med den named pipe, så skal processen åbne den i write only eller read only mode.

```
// Create FIFO (Named pipe) file and open it
char * myFIFO = "/tmp/myfifo";
mkfifo(myFIFO, 0666);
FD = open(myFIFO, O_WRONLY);
if (FD == NULL)
{
  fprintf(stderr, "Value of errno: %d\n", errno);
  fprintf(stderr, "Error opening file: %s\n", strerror(errno));
  exit(EXIT_FAILURE);
}
```

Listing 5: Kode til at lave og åben en named pipe

Den named pipe bliver lavet i /tmp/ directory og kaldt myfifo, derefter bliver den åbnet med Write Only permissions.

```
// Try and write the data in TempStr to the pipe
if (write(FD, TempStr, strlen(TempStr) + 1) == -1)
{
  fprintf(stderr, "Value of errno: %d\n", errno);
```

```
5   fprintf(stderr, "Error closing file: %s\n", strerror(errno));
6   exit(EXIT_FAILURE);
7 }
```

Listing 6: Kode til at skrive til named pipe

() bliver brugt til at skrive igennem den named pipe. Write() tager den fil der skal skrives i, en char array som skal skrives i filen og længden af den string der skal skrives som argumenter.

String Format

I temperatur stub bliver den streng der skal sendes igennem pipen formateret sådan:

dag/måned/år timer:minuter:sekunder temperatur gennemsnit

Listing 7: Eksampel af temperatur stub data

```
11/11/2021      09:34:55        10.429107       10.575894
11/11/2021      09:34:56        10.338940       10.571903
11/11/2021      09:34:57        10.092039       10.563797
11/11/2021      09:34:58        10.436171       10.561427
11/11/2021      09:34:59        10.371044       10.557971
11/11/2021      09:35:00        10.225376       10.552088
11/11/2021      09:35:01        10.022800       10.542828
11/11/2021      09:35:02        10.160806       10.535869
11/11/2021      09:35:03        10.398770       10.532875
11/11/2021      09:35:04        9.955246        10.522490
```

Dette string format er opnået ved at bruge time.h biblioteket til at få den nuværende tid. Derefter bliver strengen printet ud og sat ind i et array der bruges til at write til pipen.

```
1 // Get current time
2 T = time(NULL);
3 tm = *localtime(&T);
4
5 // Print the string that is to be sent to the pipe then set it in
      TempStr array
6 printf("%02d/%02d/%04d\t%02d:%02d:%02d\t%f\t%f\n", tm.tm_mday, tm.
      tm_mon + 1, tm.tm_year + 1900, tm.tm_hour, tm.tm_min, tm.tm_sec,
      Temp, Avg);
7 sprintf(TempStr, "%02d/%02d/%04d\t%02d:%02d:%02d\t%f\t%f", tm.tm_mday,
       tm.tm_mon + 1, tm.tm_year + 1900, tm.tm_hour, tm.tm_min, tm.tm_sec
      , Temp, Avg);
```

Listing 8: Kode til at formatere string

Signal håndtering

Til sidst har programmet en funktion som vil køre hvis en signal interrupt(Ctrl+C) er sendt igennem. Først bliver signal() funktionen kørt, som tager 2 argumenter:

Første er hvilket signal den skal håndtere, og anden er den funktion som signalet skal kører.

```
static void Sig_Handler(int _)
{
  (void)_;
  printf("\nCaught Interrupt Signal. Exiting Program.\n");
  if (close(FD) == -1)
  {
    fprintf(stderr, "Value of errno: %d\n", errno);
    fprintf(stderr, "Error closing file: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
  }
  else
  {
    exit(EXIT_SUCCESS);
  }
}

int main (int argc, char *argv[])
{
    .
    .
    .
    // Specify function to run when Interrupt Signal (Ctrl+C) is
   caught
    signal(SIGINT, Sig_Handler);
    .
    .
    .
}
```

Listing 9: Signal håndteringskode

Når en SigInt bliver fanget, så kører Sig_Handler funktionen, som lukker named pipe filen før programmet bliver termineret.

Hovedprogram

I dette program skal der hentes data fra vores IPC metode - named pipe. Dette gør vi ved at have en reader thread som gemmer data i en global variabel og en writer thread der tager den data der er i den globale variabel og gemmer i en log fil. Log filen er formateret som vist i Listing 7.

```
char current[200000000];
char buffer[255];
char laststr[50];

void *myFileReader(){
    FILE* myfile = fopen("data", "r");
    //reads a line from the file every loop
    while (fgets(buffer, sizeof(buffer), myfile))
    {
        //concatenates the string buffer to string current (appends a
   line to the end of current)
        strcat(current, buffer);
    }
```

```
13      fclose(myfile);
14      return 0;
15  }
16
17  void *myFileWriter(char *data){
18      FILE* myfile = fopen("data", "w");
19      //writes the inputted data to the data file
20      fwrite(data, sizeof(data[0]), strlen(data), myfile);
21      fclose(myfile);
22      return 0;
23  }
24
25  void *writeThread(void *vargp){
26      myFileReader();
27      while(1){
28          if (laststr != TempStr){
29              sleep(1);
30              printf("writing new data to file\n");
31              strcat(TempStr, "\n");
32              myFileWriter(strcat(current, TempStr));
33              strcpy(laststr, TempStr);
34          }
35      }
36  }
```

Listing 10: data fil handler

Selve skrivningen til filen sker i metoden "myFileWriter"hvor vi som input kombinerer to strings(current og TempStr fra Listing 12). Current er en variabel der bliver skrevet til af myFileReader metoden, og denne funktion henter den nuværende data i ascii filen og derefter bliver der appended data på current arrayet. Current arrayet har allokeret 400.000.000 bytes i ram, da en char er 2 bytes større, for at være sikker på at der kan være tilstrækkelig data i arrayet. Metoden "writeThread"er en af de to threads i programmet, den sørger for at checke efter nyt data - og hvis der er noget, bruge myFileWriter til at skrive denne til data filen.

```
1  void *readPipe(void *vargp){
2      //reading from pipe
3      mypipe();
4  }
```

Listing 11: readpipe metoden

Den næste metode læser fra pipen på samme måde som der bliver skrevet til den i listing 5. Selve metoden er vist i følgende listing 12:

```
1  char TempStr[50];
2
3  int mypipe()
4  {
5      printf("Started program\n");
6
7      signal(SIGINT, Sig_Handler);
8
9      char * myFIFO = "/tmp/myfifo";
10     FD = open(myFIFO, O_RDONLY);
11     if (FD == NULL)
```

```
12      {
13          fprintf(stderr, "Value of errno: %d\n", errno);
14          fprintf(stderr, "Error opening file: %s\n", strerror(errno));
15          exit(EXIT_FAILURE);
16      }
17      int NoDataCnt = 0;
18      sleep(0.5);
19      while(1)
20      {
21          sleep(1);
22          int R = read(FD, TempStr, 50);
23          if (R == -1)
24          {
25              fprintf(stderr, "Value of errno: %d\n", errno);
26              fprintf(stderr, "Error closing file: %s\n", strerror(errno
    ));
27              ExitProg();
28          }
29          else if (R == 0)
30          {
31              if(NoDataCnt > 5)
32              {
33                  ExitProg();
34              }
35              else
36              {
37                  NoDataCnt++;
38                  sprintf(TempStr, "NULL");
39              }
40          }
41          else
42          {
43              NoDataCnt = 0;
44          }
45          printf("From FIFO read:\n%s\n", TempStr);
46          //printf("Read returned: %i", R);
47      }
48  }
```

Listing 12: pipe reader metode

Mypipe metoden minder meget om pipe metoden i tempStub programmet, men forskellen her er at metoden læser fra pipen i stedet for. Der er desuden noget error handling idet der bliver kaldt til metoden ExitProg. Denne metode stopper programmet hvis der ikke er modtaget nyt data de sidste 6 gange der er blevet hentet fra pipe.

```
1  int main(){
2      pthread_t threads[2];
3      printf("creating threads\n");
4      pthread_create(&threads[0], NULL, readPipe, NULL);
5      pthread_create(&threads[1], NULL, writeThread, NULL);
6      pthread_join(threads, NULL);
7
8      return 0;
```

```
9 }
```

Listing 13: main metode i hovedprogrammet

I main metoden bliver 2 threads lavet, disse to threads er allerede blevet beskrevet i listing 11 og listing 10.

Synkroniseringsproblemer

Læser funktionen fra en named pipe er en blocking funktion, hvis der er en funktion på skriver enden af pipen. Derfor bliver der kun læst værdier fra pipen hvis der bliver skrevet til den. Ellers hvis skriver processen stopper, så bliver der læst 0 bytes af data så hurtig som muligt. Derfor er der i main en error handler, der stopper programmet, hvis ingen data bliver modtaget i 6 sekunder.

Imellem de to threads i main er der ikke lavet noget synkronisering. Dette skete på grund af tidsforhold, som var ikke optimale. På grund af mangel af synkronisering, så kan det være at den gamle data bliver skrevet i filen fordi læser threaden blev ikke opdateret før der bliver skrevet i filen. Dette kan man undgå ved bruge af mutexes eller semaphorer.

Mutexes er en mekanisme for to eller flere tråde at håndtere en delt ressource. Hvis to tråde kører samtidigt, kan den ene tråd anmode om en mutex lock. Hvis ressourcen er fri, bliver tråden bevilliget en lås - hvorefter den anden tråd ikke kan få fat i låsen, og dermed ikke få adgang til ressourcen.

I vores program kunne det implementeres ved, at læser threaden tager låsen mens den læser fra pipen, og først opgiver låsen igen når den har sat hvad den har læst ind i den delt array. Derefter kan threaden der skriver i en fil tager data fra den delt array, og sætte det i filen.

Semaforer er typisk implementeret som en variabel eller en abstrakt datatype, som kan holde styr på mængden af ressourcer der er tilbage. Opbruges ressourcen, kan semaforens værdi sænkes, mens den hæves når ressourcerne ikke længere er opbrugt. Semaforer kan i visse tilfælde sammenlignes med mutexes, hvis de er implementeret som binære semaforer - hvilket ville være tilstrækkeligt i denne opgave, da der kun er to tråde som skal tilgå en fællesressource.

Hukommelsesforbrug

Den måde vi har håndteret hukommelse er at der bliver allokeret en mængde hukommelse til et array når et array bliver defineret. Eks. på dette kan ses i hovedprogrammet i Listing 10. linje 1 hvor et char array bliver lavet med plads til 200.000.000 chars. Den mængde hukommelse der bliver allokeret til arrayet vil så bliver bestemt ud fra datatypen, altså i dette tilfælde char. Char har så en hukommelses brug på 2 bytes. Så når vi har et array på størrelsen 200.000.000 vil der her blive allokeret 400.000.000 bytes. Så måden vi i processerne har håndteret hukommelse er ved at

allokere hukommelsen ved oprettelse af arrays. Hele processen bruger så i sidste ende omkring 400 MB hukommelse, der skrives her "omkring"400 MB da der jo også er de 2 andre arrays, som har en meget mindre størrelse sammenlignet med dette array.

Den anden proces har så 2 arrays hvor den ene er et double array og det andet er et string array. Disse har størrelserne 60 og 50, i den rækkefølge. Det vil sige at vi har et array med 60 doubles og et med 50 strings. Dette vil sige vi har 60 gange double hukommelses brug som vil giver 480 bytes, da en double har størrelsen 8 bytes. Vi har derefter string array som ville få størrelsen 50 gange string hukommelses forbrug som er 1 byte per og dermed 50 bytes i alt. Processen der genererer data vil derfor allokere 530 bytes memory til de 2 arrays.