Obligatorisk uppgift 3 - `mdu`

Din uppgift är att skriva ett C-program som beräknar hur stor plats en uppsättning specificerade filer tar på disken. Om en specificerad fil är en katalog så ska storleken för alla filer i katalogen summeras inklusive eventuella underkatalogers summor. Om `-j` flaggan används ska katalogerna traverseras parallellt.

Uppgiften ska utföras enskilt.

Gränsyta

Programmet ska heta *mdu* och indata ska tas som argument när programmet körs.

Programmet har följande synopsis:

```
mdu [-j antal_trådar] fil ...
```

Om användaren specificerar ett antal trådar, större än 1, ska detta antal trådar användas för att göra sökningen parallellt. Observera att flaggan -j med antal trådar ska kunna skrivas på godtyckligt ställe i argumentlistan. Vid inläsning av argument till programmet så ska `getopt` användas.

Beteende

Om storleken för mer än en fil ska beräknas så ska beräkningen av respektive storlek göras helt klar innan nästa fils storlek beräknas.

En viktig del av uppgiften är att lista ut när trådarna ska avsluta. Fundera på hur en tråd ska veta att allt arbete är utfört, det räcker inte med att listan av kataloger att traversera är tom. En *semafor* eller en *conditional variable* kommer att behövas för att lösa detta.

Kompilering

Programmet ska gå att kompilera utan varningar med hjälp av `make` och den *make*-fil du levererar tillsammans med programmet. Separatkompilering, som även tar hänsyn till alla eventuella *header*-filer, ska användas. Observera att din *make*-fil ska skrivas manuellt och inte genereras av något verktyg.

Vid kompilerig ska åtminstone följande flaggor användas i lämpliga steg:

```
-g -std=gnu11 -Werror -Wall -Wextra -Wpedantic
-Wmissing-declarations -Wmissing-prototypes -Wold-style-definition
```

I denna obligatorisk uppgift ska du skilja på kompileringsflaggor och länkningsflaggor. Respektive typ av flaggor ska bara användas där det är lämpligt.

Kontroll av anrop

Alla funktionsanrop som returnerar eller på annat sätt ger tillbaka en indikation för hur anropet gick ska kontrolleras. Detta innefattar till exempel `pthread_create` som returnerar en felkod om något gick fel.

Dessa funktioner behöver du *inte* kontrollera:

- `write`, `printf` och `fprintf` (eller annan utskriftfunktion)
- `close` och `fclose`
- `free`

Om ett anrop misslyckas ska programmet skriva ut ett lämpligt felmeddelande på *standard error* för att sedan avsluta med returkoden `EXIT_FAILURE`. Om anropet ändrar värdet på `errno` vid fel ska `perror` användas för att skriva ut felmeddelandet. Om anropet inte ändrar värdet på `errno` utan indikerar fel via en felkod, så ska `strerror` användas för att omvandla felkoden till en sträng som ska skrivas ut på *standard error*.

Det normala beteendet för UNIX program som av någon orsak inte kan läsa en eller flera filer är att skriva ut ett felmeddelande och fortsätta med resten av filerna/katalogerna. Du ska därför inte avsluta programmet vid ett sådant problem utan hantera det snyggt, hoppa över problemområdet och fortsätta med resten av katalogerna/filerna.

Övriga krav på lösningen

Förutom de krav som nämns i ovanstående beskrivning ska även programmet uppfylla följande krav:

- Din lösning ska var så enkel som möjligt. Det gäller både algoritmiskt och kodmässigt. Din lösning ska enkelt förstås av annan student som läser denna kurs och som tagit till sig av kursmaterialet som examineras i denna obligatoriska uppgift.
- Programmet ska ha samma output som `du -s -l -B512 {fil} [filer ...]` förutom antalet blanka tecken mellan kolumner.
- Lösningen får *inte* använda `ftw`.
- Lösningen får *inte* använda globala variabler.
- Programmet ska använda sig av:
  - `lstat`
  - `opendir`
  - `readdir`
  - `closedir`
  - `pthread_create`
  - `pthread_join`
  - `getopt` (inklusive `optind`) för att läsa in argument.
- Programmet får ej ha några minnesläckor eller läcka fildeskriptorer, använd `valgrind`. Detta gäller oavsett väg genom programmet.
- Programmet ska vara trådsäkert, använd `valgrind --tool=helgrind`.

- Progammet ska klara trådsäkerhetstester som genomförs efter inlämnning.
- Vid minst ett fall där fler trådar än en används ska exekveringstiden halveras jämfört med när endast en tråd används.
- Programmet ska ej hamna i tillståndet *busy wait*.
- Programmet ska kunna ta ett godtyckligt antal argument och ska därför inte använda hårdkodade storlekar på datatyper. Använd `realloc` eller en dynamisk datatyp för att lagra data.
- Programmet ska klara tillräcklig längd på filsökväg för Linux.
- Koden ska ha god abstraktion, ha bra struktur, ha bra modularisering, samt ha en lämplig uppdelning i funktioner.
- Koden ska skrivas med god kodkvalité (indentering, variabelnamn, kommentarer, funktionskommentarer, etc.).
- Programmets huvudfil ska heta *mdu.c*.
- Inlämningen ska klara alla tester på Labres. Kom ihåg att även studera utskrifterna från `valgrind`.

Rapport

Förutom kod och en *Makefile* ska en rapport lämnas in. Den rapport som ska skrivas ska ha formen av en vanlig rapport och innehållsmässigt vara en "utökad" *man*-sida. Målgruppen är andra studenter på denna kurs som har tagit till sig av kursmaterialet som examineras i denna obligatoriska uppgift. Rapporten ska innehålla följande delar:

- Försättsblad
- Namn (*Name*)
- Synopsis (*Synopsis*)
- Beskrivning (*Description*)
- Lösning
- Exit-status (*Exit status*)
- Diskussion och reflektion

Försättsbladet ska vara en separat sida och innehålla titel, ditt namn, din cs-användare samt kursens namn.

För att se vad delarna namn, synopsis, beskrivning och exit-status bör innehålla får du studera ett antal olika *man*-sidor.

Lösningsdelen är det som är den huvudsakliga utökningen jämfört med en *man*-sida. I denna del ska du dels ge en övergripande beskrivning av din lösning och dels ska en analys av programmets prestanda presenteras. Den övergripande beskrivningen av lösningen ska inkludera en beskrivning av huvudalgoritmen för programmet (tänk på vad uppgiften syftar till att behandla så du inte abstraherar bort viktig funktionalitet från din huvudalgoritm). Analysen ska innehålla en figur som visar vad som händer med programmets körtid när antalet trådar ökar, samt innehålla ett resonemang om varför du tror att programmets körtid ser ut som det gör. Analysen ska även inkludera hur stor förbättring i tid du lyckades uppnå som bäst jämfört med att enbart köra programmet med en tråd.

I diskussions- och reflektionsdelen kan du skriva om problem som du stötte på när du gjorde uppgiften, vad du tyckte om den och övriga åsikter som du vill framföra.

Övriga krav på rapporten:

- Väl strukturerad
- Väl formaterad
- Väl formulerad
- Utan språkliga fel
- Eventuell(a) tabeller, figurer, kod, pseudokod, etc ska hanteras korrekt
- Eventuella referenser ska hanteras korrekt enligt Harvard-stilen (https://www.umu.se/bibliotek/soka-skriva-studera/skriva-referenser/)

Analys och bash-skript

När du analyserar prestandan av din lösning ska detta göras på datorn *Itchy*. Du ska undersöka tiden det tar för ditt program att returnera storleken på katalogen */pkg/* med t antal trådar, där t är varje heltal i intervallet 1-100. För att göra detta ska du skriva ett bash-skript som gör körningarna åt dig.

Ditt bash-skript ska använda UNIX-kommandot `time` för att mäta tiden på ditt program. Det är möjligt att formatera den utdata som `time` ger så att den kan läsas in av exempelvis MATLAB. För att göra detta kan (men inte måste) kommandon som exempelvis `grep`, `cut`, `awk` eller `sed` användas.

Detta skript ska lämnas in i labres tillsammans med övriga dela av din inlämning.

Bedömningsmall

Tillsammans med denna specifikation finns den bedömningsmall som kommer att användas vid bedömningen av din inlämning (ligger i Canvas på samma sida som specifikationen).

Inlämning

Kod, *make*-fil, bash-skript och rapport lämnas in via labres innan utsatt deadline.

Tävling!

Kopplat till denna uppgift finns en tävling för den som vill! Det är helt frivilligt att vara med och det påverkar inte ditt betyg negativt på något sätt ifall du inte är med. Så känner du att det räcker med uppgiften som den ser ut nu, så sluta läs.

Tävlingen går ut på att vi mäter ditt programs prestanda. Det som kommer mätas är wall-clock time precis som du gör när du analyserar din lösning för rapporten. Hur vi gör denna mätning specifieras inte men vi kommer mäta ditt program väldigt många gånger med ett specifikt antal trådar som är högre än

vad maskinen har att ge. Resultaten kommer publiceras i ett anslag på kursens Canvas-sajt som en rankad tabell/graf med ditt namn och ditt resultat.

Denna tävling har pågått under flera år vilket betyder att det inte enbart är en tävling mellan er som går kursen i år, utan det finns även chans att tävla mot andra studenter som gått kursen tidigare år. Gör man det riktigt bra så kanske man tar sig in på den åtråvärda all-time-topp-10 listan, eller kanske rent av högst upp!?

För att vara med i tävlingen ska du lämna in en EXTRA makefile med namnet *makefile_competition*. Den ska kompilera ditt tävlingsprogram som ska heta *mdu_competition*. Du kan antingen bara återanvända din vanliga lösning i tävlingen eller skapa nya .c/.h-filer. Skapar du nya filer som inte används av ditt vanliga program utan enbart till tävlingen bryr vi oss inte om kodkvaliten på dessa filer utan du är fri att optimera hur du vill. Notera dock att detta gällar bara de filer som INTE används till den 'vanliga' inlämningen. De filer som hör till den 'vanliga' inlämningen ska uppfylla alla krav som tidigare nämnts vad gäller läsbarhet, felhantering etc. Tänk på att i ordinarie inlämning vill vi se en enkel lösning vilket det kanske inte blir i en tävlingslösning.

Då de flesta kanske inte optimerat ett program tidigare följer här en lista av saker som du kan tänka på. Kom ihåg att mycket av detta är överkurs/onödigt samt potentiellt poänglöst då denna uppgift har väldigt specifika flaskhalsar så känn absolut inte att du måste testa allt. Det är dock väldigt lärorikt i sig att identifiera dessa flaskhalsar.

- Skriva en optimal algoritm gällande hur du låser/öppnar trådar.
- Mät hur lång tid specifika funktioner/delar av din kod tar så du vet vad som faktiskt behöver prioriteras. Funktionerna i <time.h> kan ventuellt användas till detta.
- Se till att en tråd alltid har något att göra.
- https://en.wikipedia.org/wiki/Locality_of_reference - Ha programmets minne/cache i åtanke.
- https://en.wikipedia.org/wiki/Branch_predictor - Undvik att förvirra din processors branch predictor.
- Använda dig av mer avancerade keywords som inline och register.
- Ta bort all error checking (Ahhhh...). Går att göra på ett bra sätt via preprocessordirektiv dock.
- https://caiorss.github.io/C-Cpp-Notes/compiler-flags-options.html - Undersök avancerade GCC flaggor.
- Testa att googla "*how to optimize in c for newbies*".

Deadline för tävlingen återfinns i Canvas på sidan Examination som finns i modulen Kursinformation.

Lycka till!