Assignment A2: Concurrent Programming

Computer Systems 2025
Department of Computer Science
University of Copenhagen

Martin Oldhøj Lavrentiew <zhb423>
Kristian Gert Sørensen <jxk652>
Tobias Dübeck Kristensen <rlj430>

Søndag, 26. oktober, 16:00

GitHub Link: `https://github.com/mullk/CompSys-MTK-`

1 Introduktion

Formålet med denne opgave er at udvide to C-programmer, `fauxgrep-mt` og `fhistogram`, som kan bruges til at lede efter forekomster af et bestemt ord ("needle") i et antal filer i *directories* samtidigt (concurrently). Programmet `fhistogram` arbejder på en lignende måde, men den analyserer i stedet den statistiske fordeling af bits i bytes på tværs af filerne i en mappe. I opgaven viser vi ved hjælp af vores programmer, hvor stor en indflydelse ekstra tråde i et program har på effektiviteten og køretiden. Vores programmer testes på vores datasæt i bigtest eller testdir. De kører 10 gange og måler den reelle køretid (real), og til sidst beregner og printer den gennemsnittet.

2 Job-køen

Job køen udgør fundamentet for hele vores multitrådede system. Den er ansvarlig for at fordele filer som arbejdsopgaver mellem de forskellige worker-tråde, der behandler data parallelt. Implementationen af `job_queue.c` anvender tre mutexer – én til `push`, én til `pop` og én fælles `count_lock` til at beskytte antallet af elementer i køen samt koordinere ventende tråde via `pthread_cond_wait()` og `pthread_cond_signal()`. Vi har valgt at anvende tre separate mutexer (push, pop og `count_lock`) for at muliggøre fine grained locking og dermed reducere contention mellem tråde.

Denne struktur muliggør *fine-grained locking*, hvor ind og udlæsning kan ske samtidigt, uden at trådene blokerer hinanden unødigt. Det øger mægnden af data og sikrer, at flere tråde kan arbejde effektivt samtidigt. For at undgå deadlocks sikres en konsekvent låseorden, og vi benytter `pthread_cond_broadcast()` i `destroy()` for at vække alle ventende tråde, når køen lukkes. Dette prioriterer en robust nedlukning frem for marginal performancegevinst.

Ved at anvende jobkøen som central mellemstation mellem producent- og konsumenttråde kan arbejdet udføres asynkront: hovedtråden tilføjer filer, mens worker-tråde uafhængigt popper og behandler dem. Dette giver et fleksibelt og skalerbart design, der effektivt udnytter CPU-kernerne.

3 Job-køen: Mutexes og håndtering af delte ressourcer

Vores implementation af `job_queue` illustrerer, hvordan *concurrent* programmering kan fungere i praksis. Programmet gør det muligt for flere tråde at

arbejde samtidigt. En årsag til, at dette fungerer, er brugen af *mutexes*, som sikrer gensidig adgang til fælles ressourcer. [1]

Vi anvender mutexes i `job_queue` for at sikre trådsikker adgang til de delte datafelter, når flere tråde arbejder samtidigt. Dette kommer f.eks. til udtryk i følgende hjælpefunktion: `job_queue_push()`. Vi starter med at låse køen, inden vi foretager ændringer af dens interne variabler, såsom `count` og `buffer`. Dette garanterer, at kun én tråd ad gangen kan opdatere køens tilstand. Når et nyt element tilføjes, øges tælleren (*count*), hvorefter en ventende tråd vækkes med `pthread_cond_signal()`, så den kan begynde at behandle et nyt job. Til sidst frigives låsen igen med `pthread_mutex_unlock()`, så andre tråde kan få adgang til køen.

Formålet med denne struktur er at forhindre *race conditions*, hvor flere tråde forsøger at ændre de samme data samtidigt. [2] Ved at placere `unlock` først, når en tråd er færdig med at ændre en delt ressource, sikres, at alle operationer er fuldført, inden andre tråde får adgang. Vi vil altså beskytte de indre variabler mod ændringer, inden en anden tråd får adgang.

3.1 fauxgrep-mt

De ovennævnte principper kommer også til udtryk i vores implementation af `fauxgrep-mt`, hvor multithreading anvendes til at behandle flere filer samtidigt. Flere worker-tråde henter opgaver fra den fælles job-kø og udfører dem uafhængigt. Derfor har vi defineret hjælpefunktionerne *initializeJobQueue* og `worker_function` for at opdele oprettelsen af job-køen fra behandlingen af jobs.

I `worker_function` har vi valgt at frigøre hukommelsen, der er allokeret til filstien, straks efter, at worker-tråden har behandlet filen med *fauxgrep_file*. Dette gør vi for at sikre, at hukommelsen frigøres med det samme. Dette forhindrer, at programmet får problemer med memory leaks.

4 Benchmark af multi-threaded versioner og forklaringer:

For at teste vores implementationer og anvendelsen af mutexes har vi udført tests af både single-threaded versionen og multi-threaded versionen af både fauxgrep og histogrammet. Med hensyn til *fauxgrep*, så har vi testet det ved at søge efter et ord i en mappe (90 mb), som indeholder over 500 filer. Hver fil indeholder hele sektionen på Wikipedia om 2. verdenskrig, hvilket gør datasættet ret stort. I det følgende eksempel sammenlignes kørslen med 8 tråde og faux-grep (én tråd). Der søges efter ordet ”weapon”:

[1] Concurrency I, slides 27 og 30
[2] `https://chatgpt.com/share/68fd1d04-3fa8-800f-82c6-7fb2573549f2`

Tabel 1: Single-threaded kørsel med `fauxgrep`

Run	Tid [s]
1	0,15
2	0,06
3	0,06
4	0,06
5	0,06
6	0,06
7	0,06
8	0,06
9	0,06
10	0,06
Gennemsnit	**0,069**

Tabel 2: Kørsel med otte tråde med `fauxgrep-mt(-n 8)`

Run	Tid [s]
1	0,03
2	0,01
3	0,01
4	0,01
5	0,01
6	0,01
7	0,01
8	0,01
9	0,01
10	0,01
Gennemsnit	**0,012**

Man kan se, at køretiden bliver forøget, når antallet af tråde øges fra 1 til 8. Tabellerne viser, at `fauxgrep-mt` i dette eksempel er cirka 6 gange hurtigere end single-threaded versionen:

$$S = \frac{T_{1 \text{ tråd}}}{T_{8 \text{ tråde}}} = \frac{0.069}{0.012} \approx 5.75$$

4.1 Køretid og karakterisering af throughput i *files per second* og *bytes per second*:

Med hensyn til `fauxgrep-mt`-programmet har vi også beregnet filer per sekund og bytes per sekund ved at dividere henholdsvis antallet af filer og det samlede antal bytes med den målte kørselstid. Dette viser følgende udregninger i denne tabel:

Tabel 3: Throughput for `fauxgrep-mt` med 1 og 8 tråde

Konfiguration	Filer per sekund	Bytes per sekund
1 tråd	7.246	1,27 GB/s
8 tråde	41.667	7,30 GB/s

5 Histogrammet

Programmet `fhistogram-mt` udvider det sekventielle `fhistogram` til at udnytte multithreading. Her anvendes jobkøen fra opgave 1 til at tildele filer til flere worker-tråde, som hver især beregner et *lokalt histogram*. Dette histogram registrerer, hvor ofte hver af de 8 bits forekommer i filen. Når en tråd har færdiggjort sin behandling, låses en global mutex kortvarigt, så tråden kan flette sit lokale resultat ind i det fælles `global_histogram`.

Denne struktur reducerer behovet for hyppige låsninger og minimerer *lock contention*, idet tråde arbejder uafhængigt i lange perioder. Strategien er direkte inspireret af *"Gotta Go Fast"*-princippet: tråde udfører så meget som muligt lokalt uden lås og benytter mutex kun i korte sektioner, hvor data deles. Printfunktionen blev ligeledes beskyttet med en mutex for at undgå overlappende udskrifter i terminalen.

Tabel 4: Benchmarkresultater for `fhistogram` og `fhistogram-mt`

Antal tråde	Program	Gennemsnitlig tid [s]	Speedup
1	`fhistogram`	8.844	1.00x
2	`fhistogram-mt`	4.562	1.94x
3	`fhistogram-mt`	3.419	2.58x
4	`fhistogram-mt`	2.998	2.95x

Som det ses, falder køretiden markant, når antallet af tråde øges fra 1 til 4. Speedup'en på næsten 3x viser, at løsningen skalerer effektivt op til antallet af fysiske kerner. Gevinsten flader dog ud i overensstemmelse med *Amdahl's lov*, da visse dele, såsom I/O og den globale merge, stadig udføres sekventielt.

Trådene arbejder efter følgende princip:

```
pthread_mutex_lock(&global_lock);
merge_histogram(local_hist, global_histogram);
pthread_mutex_unlock(&global_lock);
```

For at sikre en jævn opdatering af histogrammet uden at reducere gennemstrømningen væsentligt, valgte vi at udskrive histogrammet efter cirka hver 100.000 læste bytes. Dette interval blev fastlagt empirisk via målinger med `gettimeofday()` og giver en glidende visuel opdatering uden at påvirke performance markant.

Samlet set demonstrerer implementeringen et velafbalanceret kompromis mellem korrekthed, skalerbarhed og performance. Ved at kombinere lokal beregning med kortvarig global låsning og en bevidst brug af mutexer opnås et trådsikkert system, der udnytter parallelisering effektivt. Resultaterne viser, hvordan principper som *fine-grained locking*, *Gotta Go Fast* og *Amdahl's lov* kan anvendes i praksis til at skabe robuste, effektive løsninger i multitrådet programmering.

5.1 Kompilering og kørsel af benchmark

Programmerne kompileres med `make` i roden af projektmappen. Herefter kan både `fauxgrep(-mt)` og `fhistogram(-mt)` testes ved hjælp af de medfølgende bash-scripts. En detaljeret vejledning findes i filen `For at køre vores Bash filer` i `src`-mappen.

Benchmark scriptet for histogrammet gøres eksekverbart og køres således:

```
chmod +x benchmark_avg_histogram.sh
./benchmark_avg_histogram.sh
```

Dette script kører både `fhistogram` og `fhistogram-mt` med 1–4 tråde, 10 gange hver, og beregner derefter gennemsnitlig køretid og speedup. Resultaterne udskrives direkte i terminalen som et samlet overblik over performance-scalingen.

Tilsvarende kan benchmark-scriptet for `fauxgrep-mt` køres (testet på Windows systemer) med:

```
chmod +x benchmark_avg_fauxgrep.sh
./benchmark_avg_fauxgrep.sh malloc bigtest 4
```

6 Design og strategi

Formålet med designet har været at opbygge et system, der både er trådsikkert og effektivt. Vores overordnede strategi var at anvende et *producer–consumer*-mønster, hvor jobkøen fungerer som kommunikationsled mellem hovedtråden (producenten) og worker-tråde (konsumenterne). Dette muliggør asynkron arbejdsfordeling og sikrer, at flere filer kan behandles parallelt, mens CPU'ens ressourcer udnyttes fuldt ud.

For at opnå høj ydeevne har vi bevidst anvendt principper fra *fine-grained locking* og *Gotta Go Fast*-strategien, der blev introduceret i forelæsningerne. Disse principper indebærer, at hver tråd udfører så meget arbejde som muligt lokalt uden at holde en lås, og kun låser kortvarigt, når fælles data skal opdateres. På den måde reduceres *lock contention*, og flere tråde kan arbejde effektivt side om side.

De mest ikke-trivielle beslutninger i designet omfattede balancen mellem robusthed og performance. Vi valgte eksempelvis at bruge tre separate mutexer i jobkøen (push, pop og count_lock) for at tillade samtidige operationer uden at skabe race conditions. Derudover blev jobkøen designet til at kunne lukkes sikkert med `pthread_cond_broadcast()` for at vække alle ventende tråde ved nedlukning — en løsning, der prioriterer stabilitet frem for marginal performancegevinst.

Endelig har vi lagt vægt på læsbar og vedligeholdelig kode, hvor hver tråd har en klart defineret rolle, og de delte ressourcer beskyttes konsekvent. Denne struktur gør det muligt at udvide systemet senere med flere tråde eller yderligere funktionalitet uden at ændre på de grundlæggende synkroniseringsprincipper.

7 Ambiguiteter i opgavebeskrivelsen

En mindre uklarhed opstod omkring fortolkningen af, hvornår `job_queue_destroy()` skulle kaldes. Vi fortolkede det som, at funktionen først måtte anvendes, når alle filer var pushet i køen, og alle tråde havde hentet deres jobs. Dette sikrer, at ingen filer går tabt under afslutning.

Derudover var det ikke entydigt beskrevet, hvor ofte histogrammet skulle opdateres. Vi valgte derfor empirisk at udskrive histogrammet for hver ca. 100.000 bytes, efter at have målt, at dette gav en god balance mellem jævn visuel opdatering og høj gennemstrømning.

8 Theory:

8.1 0.1:

Hvis OS-funktionalitet blev implementeret i hardware, ville systemet miste fleksibilitet. Software kan eksempelvis relativ nemt tilpasses og opdateres i modsætning til hardware. Når en fejl opstår, sørger hardware for at stoppe instruktionen, gemme adressen og registrere årsagen, mens operativsystemet derefter vurderer, hvordan fejlen skal håndteres. Derfor kan man konkludere, at hvis OS-funktionalitet blev implementeret direkte i hardware, ville vi miste denne fleksibilitet og egenskab.[3]

8.2 0.2:

Selvom en computer kan køre adskillige programmer samtidigt, er princippet om locality stadig centralt mht. at forbedre ydeevnen. Ifølge *Computer Organization and Design* (s. 389) opstår locality i programmer på grund af deres struktur. De fleste programmer indeholder loops, hvor de samme instruktioner og data anvendes gentagne gange, hvilket skaber temporal locality. Samtidig tilgås instruktioner og data ofte sekventielt, hvilket giver spatial locality.

Selv når flere programmer kører samtidigt og benytter forskellige dele af hukommelsen, udviser hvert program internt locality.

8.3 0.3:

En pointer i et program peger ikke direkte på en fysisk adresse i hukommelsen eller på disken, men på en virtuel adresse. Den virtuelle adresse oversættes af systemets hukommelsesstyring til en fysisk adresse i RAM ved hjælp af address translation (adresseoversættelse).

Som beskrevet på side 442 i *Computer Organization and Design* muliggør virtuel hukommelse *relocation*, hvor de virtuelle adresser, som et program bruger, kortlægges til forskellige fysiske adresser, før de anvendes til at tilgå hukommelsen.

[3] COD, s. 339

9 Kildeliste

- **Kilder læst for at forstå opgaven og den teoretiske baggrund**
  Disse sider blev ikke brugt direkte i koden, men har været nyttige for at forstå principperne bag køer, datastrukturer og histogrammer i C.
  - AccuWeb Cloud. *How to create a queue in C language*. `https://accuweb.cloud/resource/articles/how-to-create-a-queue-in-c-language`
  - Wikipedia. *Histogram*. `https://da.wikipedia.org/wiki/Histogram`
  - Computer Organization and Design [COD], RISC-V Edition: The Hardware Software Interface, David A. Hennessy and John L. Patterson, 2nd edition, Morgan Kaufmann, ISBN: 978-0128203316
  - CompSys forelæsningsslides: Concurrency I.

- **Kilder om jobkøer, tråde og synkronisering i C**
  Disse kilder blev brugt til at forstå implementeringen af `job_queue.c`, brugen af `pthread`, mutex'er, condition variables og trådhåndtering. De har hjulpet med at opbygge den delte jobkø og kommunikationen mellem worker-tråde.
  - DigitalOcean. *Queue in C*. `https://www.digitalocean.com/community/tutorials/queue-in-c`
  - GeeksForGeeks. *Queue in C*. `https://www.geeksforgeeks.org/c/queue-in-c/`
  - Medium (Leo88). *How to build a good enough message queue in C – a non-production ready guide*. `https://leo88.medium.com/how-to-build-a-good-enough-message-queue-in-c-a-non-production-ready-guide-6f1f222b02aa`
  - Programiz. *Queue – Data Structure in C*. `https://www.programiz.com/dsa/queue`
  - StackOverflow. *What is a worker in C*. `https://stackoverflow.com/questions/15444385/what-is-a-worker-in-c`
  - Lawrence University. *Worker Threads Tutorial*. `https://www2.lawrence.edu/fast/GREGGJ/CMSC480/net/workerThreads.html`
  - GeeksForGeeks. *Thread functions in C/C++*. `https://www.geeksforgeeks.org/c/thread-functions-in-c-c/`
  - GeeksForGeeks. *Function pointers in C*. `https://www.geeksforgeeks.org/c/function-pointer-in-c/`
  - StackOverflow. *Difference between global / local / static mutex*. `https://stackoverflow.com/questions/62829474/difference-between-global-local-static-mutex`

  - CodeQuoi. *Threads, Mutexes and Concurrent Programming in C*. `https://www.codequoi.com/en/threads-mutexes-and-concurrent-programming-in-c/`
  - CBoard C-Programming Forum. *Histogram using Pthreads*. `https://cboard.cprogramming.com/c-programming/162739-c-programming-histogram-using-pthreads.html`

- **Kilder brugt direkte til at skrive koden for histogrammet (`fhistogram-mt`)**
  Disse artikler gav praktiske eksempler på multithreading i C med Pthreads, brug af mutex'er og jobkøer, som blev brugt direkte under implementeringen af histogrammet.
  - GeeksForGeeks. *Multithreading in C (with Pthreads)*. `https://www.geeksforgeeks.org/c/multithreading-in-c/`
  - GeeksForGeeks. *Thread management functions in C*. `https://www.geeksforgeeks.org/c/thread-functions-in-c-c/`
  - W3Resource. *Multithreading in C using POSIX Threads*. `https://www.w3resource.com/c-programming/c-multithreading.php`
  - TechForTalk. *Building a Multi-Threaded Task Queue with Pthreads*. `https://techfortalk.co.uk/2025/03/08/building-a-multi-threaded-task-queue-with-pthreads/`

10 Declaration of using generative AI tools (for students)

☒ I/we have used generative AI as an aid/tool (please tick)

☐ I/we have **NOT** used generative AI as an aid/tool (please tick)

If generative AI is permitted in the exam, but you haven't used it in your exam paper, you just need to tick the box stating that you have not used GAI. You don't have to fill in the rest.

List which GAI tools you have used and include the link to the platform (if possible):

Example: [Copilot with enterprise data protection (UCPH license), `https://copilot.microsoft.com`] [1em] **List which GAI tools you have used and include the link to the platform (if possible):**

Example: [Copilot with enterprise data protection (UCPH license), `https://copilot.microsoft.com`] [1em]

- **ChatGPT:**
- Forståelse af opgaven og brug af formlerne Amdahl og Gustafson: `https://chatgpt.com/g/g-p-68bfcaedbd9c8191bd630ce921a0446f-compsys/c/68e3aa3e-e2d8-8329-81e0-820bd58cdde7`
- Gennemgang og forståelse af concurrency og mutexes: `https://chatgpt.com/g/g-p-68bfcaedbd9c8191bd630ce921a0446f-compsys/c/68e3c148-caa8-8329-a728-3264a0cc8a2e`
- Tjek af løsning imod reference fundet online: `https://chatgpt.com/g/g-p-68bfcaedbd9c8191bd630ce921a0446f-compsys/c/68e675c1-2690-8333-97e8-ecb7c5fdc0da`
- Forståelse af job-queue og opgavespørgsmålet om "shared resources"samt mutexes ift. task 2: `https://chatgpt.com/share/68fd1d04-3fa8-800f-82c6-7fb2573549f2`
- Forståelse af datastruktur (job-queue.h) og forskellen på throughput og latency: `https://chatgpt.com/share/68fdffd5-80b0-800f-8143-1a54e4364282`

Describe how generative AI has been used in the exam paper:

1. **Purpose (what did you use the tool for?)**
   Vi har anvendt ChatGPT som en faglig sparringspartner under udviklingen af vores programmer, test og rapport. AI-værktøjet blev brugt til at opnå en bedre forståelse af de udleverede filer og hjælpe med at diskutere centrale begreber inden for trådprogrammering, herunder brugen af `pthread`, mutex-mekanismer, condition variables og jobkø-arkitektur. Derudover blev det anvendt som støtte til at forklare og reflektere over designvalg i forbindelse med performance, *fine-grained locking*, og "Gotta Go Fast"-princippet. I rapportskrivningen blev værktøjet brugt som hjælp til at formulere tekniske afsnit tydeligere, strukturere refleksionerne om parallelisering, samt sikre sproglig og terminologisk konsistens.

2. **Work phase (when in the process did you use GAI?)**
   ChatGPT blev anvendt i tre hovedfaser af arbejdet: (1) Under udviklingen af implementeringen af `job_queue`, `fauxgrep-mt` og `fhistogram-mt`, hvor vi brugte det til at forstå og diskutere korrekt anvendelse af tråde, mutex'er og condition variables. (2) Under test- og benchmarkfasen, hvor værktøjet hjalp med at designe og optimere vores `benchmark_avg_histogram.sh`-script samt analysere resultater ud fra Amdahl's lov og throughput-beregninger. (3) I skrivefasen, hvor ChatGPT blev brugt som sproglig og teknisk sparringspartner til at strukturere afsnit om design, strategi og performanceanalyse. AI blev dermed brugt som en lærings- og refleksionsressource, men ikke som en erstatning for selvstændig kodning eller analyse.

3. **What did you do with the output? (including any editing of or continued work on the output)**
   Outputtet fra AI blev brugt som udgangspunkt for videre refleksion og egen formulering. Vi redigerede og omskrev al tekst, så den matchede vores egen implementering, testdata og forståelse af opgaven. Eksempler og forklaringer fra AI blev løbende verificeret gennem egne tests og sammenlignet med pensum (bl.a. *Computer Organization and Design* og forelæsningsmaterialet om concurrency). Den endelige kode, benchmark-målinger og rapportafsnit er derfor udarbejdet selvstændigt med udgangspunkt i vores egen forståelse, hvor AI har fungeret som et pædagogisk værktøj.

Please note: Content generated by GAI that is used as a source in the paper requires correct use of quotation marks and source referencing. Read the guidelines from Copenhagen University Library at KUnet.

`https://copilot.microsoft.com/`
`https://kunet.ku.dk/faculty-and-department/copenhagen-university-library/library-access/Pages/AI.aspx`