Datalogi 1F Foråret 2002: G1
Tre mindre opgaver

6. februar 2002

1 Indledning

Opgave G1 stilles onsdag den 6. februar 2002 og skal afleveres senest onsdag den 13. februar kl. 14:00 i DIKU's 1.dels-administration, Universitetsparken 1, 2100 København Ø. Besvarelser, der sendes med posten, skal være DIKU i hænde senest ved afleveringsfristens udløb. Besvarelser kan udarbejdes individuelt eller i grupper op til tre deltagere. Der gives ikke reduktion i opgavens omfang ved individuel aflevering, så dette kan ikke anbefales. Gruppesammensætningen behøver ikke at være den samme i G1 og i de senere rapportopgaver.

Vigtige meddelelser vedrørende denne opgave vil blive meddelt i kursets nyhedsgruppe og/eller på kursets hjemmeside `http://www.diku.dk/teaching/2002f/dat1f/`. Ændringer der meddeles på nyhedsgruppen og på opslagstavlen, underskrevet af Klaus Hansen, er officielle og obligatoriske.

Der vil blive brug for kernemaskinerne under denne opgave. T-kort til reservationstavlen udleveres af instruktorerne. Hver person må max. have ét T-kort.

Studerende, der tidligere har været til eksamen i dat1F rapporter bør kontakte 1.dels-administrationen.

2 Kort om opgaven

Denne opgave har til formål at give den studerende praktisk erfaring i brug af DIKU's Alpha-maskiner til brug ved løsning af rapportopgaverne der stilles i marts og april. Opgaven består af tre mindre opgaver, der alle skal løses og afleveres. En godkendt løsning er en forudsætning for at få bedømt de karaktergivende rapportopgaver. Hvis G1 ikke godkendes, vil det være muligt at genaflevere med frist onsdag 27. februar. Opgaverne beskrives i detaljer i de følgende afsnit.

De tre opgaver vægtes således:

1. D-kernen: 15%
2. Ackermann's funktion: 30%
3. Dynamisk lagerallokerring: 55%

3 Krav til besvarelsen

Til denne opgave skal der ikke skrives en rapport i normal forstand. Der skal til hver af de tre opgaver blot afleveres:

- En udskrift af de udviklede (skrevne) programmer. Programmerne skal være velkommenterede.
- Et skærmdump af det, I har skrevet for at oversætte og køre programmet.
- Et skærmdump (max. 1 side pr. program!) af programmets uddata. Brug eventuelt UNIX `script`.
- Evt. en ekstra side eller to med kommentarer til det, I har lavet, hvis der er noget, som ikke er umiddelbart indlysende.

Det afleverede skal være fortløbende sidenummereret. Hver udskrift eller skærmdump skal være tydeligt markeret med en angivelse af hvad udskriften eller skærmdumpet er. F.eks.: "Udskrift af `main.cc` til opgave 2".

Selvom I ikke skulle blive færdige med alle opgaverne, bør I aflevere alligevel. Hvis I ikke kan få færdiggjort besvarelsen af en af opgaverne, bør I aflevere jeres forsøg alligevel. Alt hvad afleveres, skal dog demonstrere, at I har prøvet tingene *i praksis*. Opgaver, der kun er lavet på tegnebordet, vil ikke give nogen point.

4 Opgave 1: d-kernen (15%)

I Kursusbogens bind 4 (KB4) er angivet fire færdige kerner. Hvordan disse fungerer er ikke nødvendigt at vide for at besvare denne opgave. I kataloget `~dat1f/KB4/kerner` findes disse fire kerner:

KB4 afsnit nr.	Kernebeskrivelse	Afbrydelser?	Kernenavn
6.2	Kerne med decentralt processkift.	Nej	a-kerne.
6.3	Kerne med centralt processkift.	Nej	b-kerne.
7.1	Kerne med løst koblede drivprogrammer.	Ja	c-kerne.
7.2	Kerne med tæt koblede drivprogrammer.	Ja	d-kerne.

I denne opgave skal I hente d-kernen ned i jeres hjemmekatalog, oversætte og downloade den og køre den på kernemaskinerne.

Til denne opgave skal selve programmet ikke afleveres (det står i KB4), men kun, hvad I skriver for at få det til at køre samt en udskrift af uddata ved kørslen.

Følg evt. proceduren som beskrevet i KB5, afsnit 3.5.

5 Opgave 2: Ackermanns funktion (30%)

Ackermanns funktion er en funktion, som kendetegnes ved at returnere meget store værdier for selv små værdier i inddata og bruge meget plads på stakken.

En implementering af funktionen i C/C++ ser ud som følger:

```
long ack(long x,long y)
{
  if (x==0)
    return y+1;
  if (y==0)
    return ack(x-1,1);
  return ack(x-1,ack(x,y-1));
}
```

Denne funktion skal I *håndoversætte* til Alpha-assemblerkode og oversætte til maskinkode med assembleren. Dernæst skal der skrives et lille C++ program (i en anden fil) som kalder assemblerfunktionen et vist antal gange. Der skal udskrives en tabel af alle værdier af *ack(x,y)* for $0 <= x <= 3$ og $0 <= y <= 11$. Programmet skal ikke køre på kernemaskinerne men på Alpha-udviklingsmaskinerne under UNIX. I kan bruge `cout` til udskriften.

Det er muligt at bruge visse programmer til at generere assemblerkoden automatisk. Dette betragtes som eksamenssnyd. I vil i øvrigt gøre jer selv en bjørnetjeneste ved at prøve at snyde, da den viden I tilegner jer i denne opgave skal bruges senere.

5.1 Tips og råd

- I kan læse om kaldskonventionerne og om assemblerprogrammering i Kursusbog 5, afsnit 4.3 og 4.7. Læg især mærke til den sikkert noget uvante brug af *$gp* og *$pv*.
- Husk at på en Alpha er en `int` i C en 32-bit værdi og en `long` er en 64-bit værdi.
- Brug gdb til at afluse (eng: debug) koden, som med sikkerhed vil være fejlbehæftet i det første forsøg.
- `Ack(3,11) = 16381.`

6 Opgave 3: Dynamisk lagerallokering (55%)

Hver gang I i C++ eller Java bruger `new`, allokeres der lager fra hoben (eng: the heap). Normalt sørger både køretidssystemet for det givne programmeringssprog og operativsystemet for at håndtere lagerallokeringen. På kernemaskinerne findes der hverken køretidssystem, standardbiblioteker eller operativsystem. Vi må derfor lave disse funktioner selv.

I denne opgave skal I udvide d-kernen med et multiprogram skrevet i C/C++, der implementerer dynamisk lagerallokering og lagerfrigivelse.

Der skal laves to funktioner `void *malloc(size_t size)` og `void free(void *p)` som har samme funktionalitet som i C standardbiblioteket. Se evt. *K&R: The C Programming language* (side 145 og 252) for mere information. I må gerne lade programmets tilstand være udefineret, hvis brugeren kalder `free` med en variabel, som ikke tidligere har været tildelt med `malloc()`. `size_t` er defineret i standardheaderen `stddef.h`, som dog blot indeholder nogle definitioner. Den kan man godt bruge på kernemaskinerne, da den ikke kræver, at der linkes med noget kode fra standardbiblioteket.

Pseudokode for `malloc()` og `free()` findes nedenfor. Denne skal bruges som udgangspunkt besvarelsen. Den fulde besvarelse kan udtrykkes på mindre end 50 linier C++ kode.

6.1 Praktiske oplysninger

Til denne opgave er der på forhånd lavet et multiprogram der tester allokering og frigivelse. Programmet er færdigt, med undtagelse af at `malloc()`, `free()` og `allocinit()` i filen `alphaalloc.cc` ikke er defineret – det er jeres opgave. I må kun ændre i `alphaalloc.cc`; andre filer må kun modificeres, hvis der kan dokumenteres gode grunde til det.

`allocinit()` kaldes inden første proces startes og kan bruges til at initialisere variable til brug for `malloc()` og `free()` (altså en slags constructor). Hvis man ønsker det kan man også blot lade `allocinit()` forblive tom.

Til denne opgave bruges en lettere modificeret udgave af d-kernen fra KB4. Teksterne findes i `~dat1f/G1`. Der er kun sket ændringer i filerne `dkerne.h` og `dkerne.cc`. Det gamle multiprogram i filen `multiprogram.cc` er blevet udskiftet med det nye multiprogram bestående af filerne `alphaalloc.cc`, `alphaalloc.h`, `testprogram.cc` og `testprogram.h`. Ændringerne i d-kernen er minimale og består hovedsageligt kun af at al information der hører til multiprogrammet er blevet flyttet fra `dkerne.cc` til `testprogram.cc`. Resten af filerne fra d-kernen er de samme som tidligere.

Mens testprogrammet kører, skal hobens fulde kapacitet være sat til netop 4000 byte (500 quad words).

I skal aflevere en udskrift af `alphaalloc.cc` samt det I har skrevet for at få det til at køre på kernemaskinerne. I skal ydermere aflevere en udskrift af uddata. Hvis alt går vel, vil der blive udskrevet et digt af en kendt eventyrforfatter. Digtet er 8 linier langt, og der for hver linie blive udregnet en checksum der undersøger om data er blevet ødelagt.

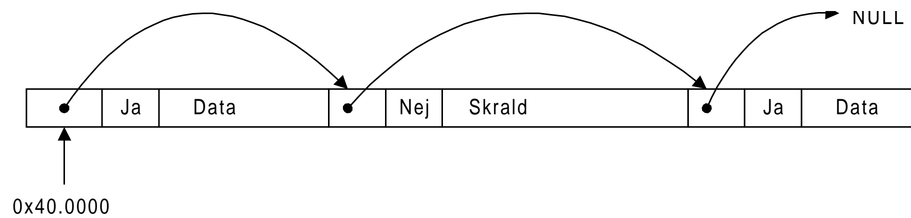
6.2 Algoritmen og pseudokode

En nem måde at implementere dynamisk allokering ved er at bruge en variant af "buddy"-algoritmen. I denne haves en liste af klodser, som enten kan være tomme eller fulde. I starten er der kun én klods og hver gang en del af en klods skal bruges, klippes den over og bliver til to klodser. En klods har en peger (eng: pointer) til næste klods samt en bit, der angiver, om klodsens indhold er allokeret. For at allokere en klods, skal man så blot lede efter den første klods, der er stor nok til at tilfredsstille allokeringen og som samtidig ikke er i brug. Kan man ikke dette, må man opgive og returnere NULL.

Frigørelse af en klods med `free()` kan blot gøres ved at bruge parameteren til `free()` til at finde klodsen og markere den som uden indhold.

På et givet tidspunkt vil strukturen i hoben således kunne være som vist på figur 1:

Algoritmen udtrykt som pseudokode er: (se følgende side)



Figur 1: Datastruktur for hob

```
Alloker en hob.

malloc(klodsstørrelse)
{
  Prøv at allokere en klods(klodsstørrelse)

  returner en peger til den nye klods eller NULL hvis ingen
}

free(peger til klods)
{
  Marker klodsen der peges på som ikke brugt.
}

Prøv at allokere en klods(klodsstørrelse)
{
  For hver klods i hoben
    Hvis klodsen er større end klodsstørrelse og
      klodsen ikke allerede er i brug
    Så
      Marker klodsen som i brug.
      Hvis klodsstørrelse er mindre en størrelsen på
          den tomme klods vi bruger, så skal denne klippes
          over i to dele og en ny klods oprettes.
        Hop ud af for-løkken og returner den allokerede klods.
  Hvis ingen klods blev fundet returner NULL.
}
```

Algoritmen er absolut ikke optimal. Der kan laves ændringer, som gør den betydeligt hurtigere og giver en mere effektiv anvendelse af lageret. Ambitiøse studerende kan måske være interesseret i at implementere visse forbedringer:

- Den angivne algoritme lider af fragmentering, da lageret efter lang tid vil være delt op i mange små klodser, der muligvis er tomme. Inden man opgiver at finde en klods, kan man undersøge om der skulle være klodser ved siden af hinanden der skulle være tomme og dernæst slå dem sammen. Overvej hvor en sådan test for om klodser kan slås sammen indsættes bedst i programmet.
- Man kan lade rutinen, der leder efter en tom klods, starte et andet sted end fra starten hver gang, da dette ellers ville lade den lede forbi alle de allokerede klodser hver gang.
- Man kan skifte strategien for allokering, så man i stedet for blot at bruge den første man finder ("first-fit") i stedet finder den der eksempelvis passer bedst i størrelsen ("best-fit").
- Vi gemmer her klodserne i en hægtet liste. Som I ved er lister ikke altid den mest effektive datastruktur. Find evt. på andre.
- Man skulle umiddelbart tro, at hvis der opstår mange små huller mellem klodser, der er i brug, at vi bare kunne rykke dem sammen. Det er dog ikke tilladt, da enhver ændring af placeringen af data vil gøre alle pegere til disse områder ugyldige, hvilket er galt.

Det skal bemærkes at disse ændringer absolut ikke er nødvendige for at få fulde point for opgaven.

6.3 Tips og råd

- Skriv og fejlret `malloc()` og `free()` på Alpha-udviklingsmaskinerne, inden I slipper dem løs på kernemaskinerne. Derved får du operativsystemets hjælp til at checke for stakoverløb, udskrivning, mv. Når det virker, kan du portere det til kernemaskinerne – dvs. linke det med d-kernen og fjerne brugen af alle faciliteter der ikke findes på kernemaskinerne. Dette blev eksempelvis gjort da `testprogram.cc` blev skrevet. Kik i den for at se et eksempel på hvordan et program kan være skrevet til både kernemaskiner og Alpha-udviklingsmaskiner.
- På Alpha-maskinerne kan hoben nemmest allokeres ved at lave et stort statisk array. På kernemaskinerne er dette også en løsning, men dette vil gøre downloadtiden til kernemaskinerne stor hvis arrayet er stort. En bedre løsning er blot at oprette en peger til et område vi ved er tomt (f.eks. 0x40.0000) og bruge dette sted i lageret. Denne sidste løsning kan ikke umiddelbart bruges på Alpha-udviklingsmaskinerne, men kun på kernemaskinerne.
- Vær meget påpasselig med typer i programmet. Det bliver nødvendigt at lave en del typecasts, så man skal hele tiden holde tungen lige i munden.
- Alle adresser der returneres fra `malloc()` bør være delelige med 8, da brugeren har lov til at typecaste en returneret peger til at være en peger til en `long`. `Longs` skal placeres på adresser der er delelige med 8. Ellers vil der komme en fejl; typisk en "Bus Error"når man kører under UNIX.
- Du kan ikke bruge klasser med konstruktører eller destruktører medmindre du overloader `new` og `delete` for alle klasser i dit program - som det er gjort i det udleverede test program. Dette kan passende gøres ved at alle arver `new` og `delete` fra en klasse `allocate`, som den

der allerede er lavet i `alphaalloc.cc`. Hvis du glemmer dette vil du få en linker fejl, når du linker til kernemaskinerne (det er ikke noget problem under UNIX). Den vil hævde at `__builtin_delete` ikke er defineret.

- Vær opmærksom på at statisk allokerede klasser ikke automatisk kalder deres konstruktører og destruktører på kernemaskinerne.
- Hvis `malloc()` og `free()` skulle bruges af flere multiprogrammer ad gangen skulle man sikre udelelig adgang til dem ved hjælp af binære semaforer. Dette behøver I ikke at gøre i denne opgave.
- Alphaerne har ikke division "/" i hardware, så division bevirker kald til procedurer i standardbiblioteket. Men kernemaskinerne har ikke disse, så division med andet end to-potenser vil give en lænkningsfejl (f. eks. "Error: Undefined: __remqu __divqu").
- I testprogrammet er der lavet en stak- og en strengklasse. Disse kan evt. bruges i senere opgaver på dat1F.

7 Generelle gode råd

Dette afsnit er en del gode råd og tips som I kan bruge hvis I skulle have lyst.

- Skim KB5 igennem, før I starter. Gør grundigt brug af KB5 under opgaveperioden.
- Løs alle opgaverne i fællesskab i gruppen. På den måde sikres det, at alle har en basal viden inden K1. Hvis alle er med nu, vil også de andre i gruppen kunne drage fordel af det i K1.
- Opgaverne kan løses i vilkårlig rækkefølge, de er ovenfor listet i stigende sværhedsgrad.
- Arbejd ikke alene.
- Det er tilladt at diskutere opgaven med andre grupper på en uformel måde "over kaffen", dog uden at man direkte hjælper hinanden med kode eller med assistance ved terminalen. Udnyt dette ved at arbejde i terminalrummene. Mange gode tips flyver rundt i dette miljø. For din egen skyld bør du dog forstå de tips, du får, inden du bare bruger dem.
- Hvis I arbejder sammen i en gruppe, kan I reservere en kernemaskine (til opgave 3) længere tid ad gangen ved at reservere samme maskine i flere fortløbende perioder.
- I har som rapportskrivere førsteret til terminalerne på 1.sal syd (Hel) og 1.sal midterfløjen (Midgård). Hvis der er mangel på maskiner, har I ret til at fjerne andre ikke-rapportskrivere. Dette gælder især folk der bruger MUDs, spiller spil, chatter eller surfer på nettet - også selvom de påstår, at det de laver er studierelevant. Hvis der er nogen problemer med at fjerne folk eller I ikke kan lide konfrontationen, kontakt venligst operatøren eller en dat1F instruktor.