Lösningsförslag till tentamen i TDIU11 2025-03-28

Filip Strömbäck

1. Se sista sidan.
2. Vi har 62 bitar logiska adresser och 4 bitars page-table-entry. Vi vill också ha maximal 32 KiB pagestorlek.

   (a) För att undvika fragmentering vill vi att alla pagetabeller utnyttjas till fullo. För att hitta den största möjliga pagestorleken kan vi helt enkelt testa 32 KiB och successivt gå nedåt.

   Vi börjar med att testa 32 KiB, dvs. 2^{15} bytes. I och med att varje page-table-entry är 4 bytes stort så kan varje page-table innehålla $\frac{2^{15}}{2^2} = 2^{13}$ element. Varje nivå kan därmed ”täcka” 13 bitar. Av de 62 bitarna logisk adress så används 15 bitar för offset, så finns 47 bitar kvar. Eftersom 47 inte är delbart med 13 så kommer vi inte kunna använda alla pagetabeller fullt ut. Ett exempel är 8 + 13 + 13 + 13. Det illustrerar problemet med en pagestorlek på 32 KiB.

   Vi testar i stället med 16 KiB, dvs. 2^{14} bytes. Då kan varje page innehålla $\frac{2^{14}}{2^2} = 2^{12}$ element. Av de 62 bitar logisk adress som finns så används då 14 bitar till offset. Kvarvarande 48 bitar passar då perfekt till 4 nivåer paging med 12 bitar i varje nivå. En logisk adress är alltså uppdaterad som 12 + 12 + 12 + 12 + 14.

   (b) Varje element i pagetabell innehåller 32 bitar. Enligt uppgiften så behövs 2 bitar i varje element till annat. Det finns alltså 30 bitar kvar för frame-nummer. Med en pagestorlek på 2^{14} bitar får vi maximalt $30 + 14 = 44$ bitar fysisk adressrymd.

   (c) Givet att vi har 4 nivåer paging så får vi:

$$
\begin{aligned}
\alpha &= 0.9 \\
t_{ram} &= 10 \text{ ns} \\
t_{good} &= t_{ram} = 10 \text{ ns} \\
t_{bad} &= 5 \cdot t_{ram} = 50 \text{ ns} \\
EAT &= \alpha \cdot t_{good} + (1 - \alpha) \cdot t_{bad} \\
&= 0.9 \cdot 10 + 0.1 \cdot 50 = 9 + 5 = 14 \text{ ns}
\end{aligned}
$$

3. (a) Den virtuella adressen `0xA31` översätts till en fysisk adress på följande sätt: Först extraheras pagenumret, i detta fall de första 4 bitarna, dvs. `0xA`. Vi tittar sedan på rad `0xA` i respektive tabell. För process 1 är valid-biten inte satt, så process 1 får pagefault. För process 2 är valid-biten satt, och frame är `0xB7`. Frame-numret kombineras sedan med offset från originaladressen (`0x31`), vilket blir `0xB731`.

   (b) Processerna delar frames `0xD3` (page `0x3` resp. `0x0`). Man kan tänka att de även delar frame `0x93` (page `0xF` resp. `0x8`), men valid är 0 i process 1, så detta är inte delat i praktiken.

   Virtuella adresser för frame `0xD3` är `0x300`–`0x3FF` i process 1 och `0x000`–`0x0FF`.

   (c) Frame `0xB7` finns tillgänglig både som page `0x3` och `0xA`. Detta gör att adresser `0x300`–`0x3FF` är samma som `0xA00`–`0xAFF`.

4. Vi har en disk som är 4 TiB ($= 2^{42}$ bytes) stor, och fysiska block som är 512 bytes stora.

   (a) Vi vill ha maximalt 2^{32} block. Detta ger att minimal blockstorlek blir: $\frac{2^{42}}{2^{32}} = 2^{10}$, alltså 1 KiB.

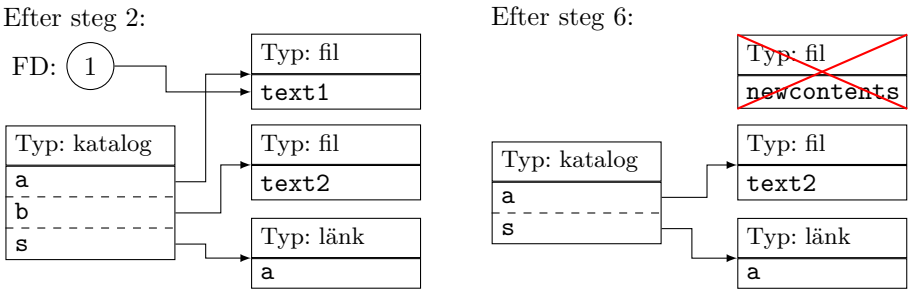
   (b) Givet 1 KiB block och 32-bitars pekare så kan vi lagra $\frac{2^{10}}{2^2} = 2^8$ pekare i varje indexblock. Med två nivåer indexering får vi alltså: $2^8 \cdot 2^8 \cdot 2^{10} = 2^{26}$ bytes, vilket är 64 MiB.

   (c) i. För länkad allokering behöver vi totalt 100 accesser, i och med att vi behöver läsa block 1–99 för att hitta och läsa det 100:e blocket. Totalt $100 \cdot 10$ ms $= 1000$ ms $= 1$ s.

   ii. För 2 nivåers indexerad allokering räcker 3 läsningar, 2 läsningar av indexblock och 1 läsning av datablock. Alltså: $3 \cdot 10$ ms $= 30$ ms.

   (d) SSTF väljer den sektor som är närmast läsarmens nuvarande position. Alltså kommer den behöva flytta läshuvudet i följande sekvens: 620 → 600 → 650 → 670 → 880 → 360 → 160. Totalt: 1020

5. (a)



   (b) Det går att öppna `s`. Enligt bilden till höger ovan så är `s` en symbolisk länk som refererar till filen `a`. Så om ett program öppnar `s` så kommer systemet att öppna filen `a` i stället. Filen `a` innehåller i sin tur `text2`.

   (c) Inoden som till en början hette `a` tas bort i steg 6. Man kan tro att det sker i steg 3, men eftersom filen är öppen så kan inoden inte tas bort då. Det sker först när filen sedan stängs i steg 6.

6. Vi vill skapa filer under `/handins` som motsvarar studenters inlämningar.

   (a) Detta går exempelvis att lösa med vanliga filrättigheter:

   Vi behöver en grupp `teachers`, som alla lärare är medlemmar i.

   Vi låter `/handins` ägas av användaren `root` och gruppen `teachers`. Rättigheterna får vara `rwxrwxrw-`. Dvs. ”others” får läsa och skriva, men inte se innehållet i katalogen, medan lärare får göra vad de vill.

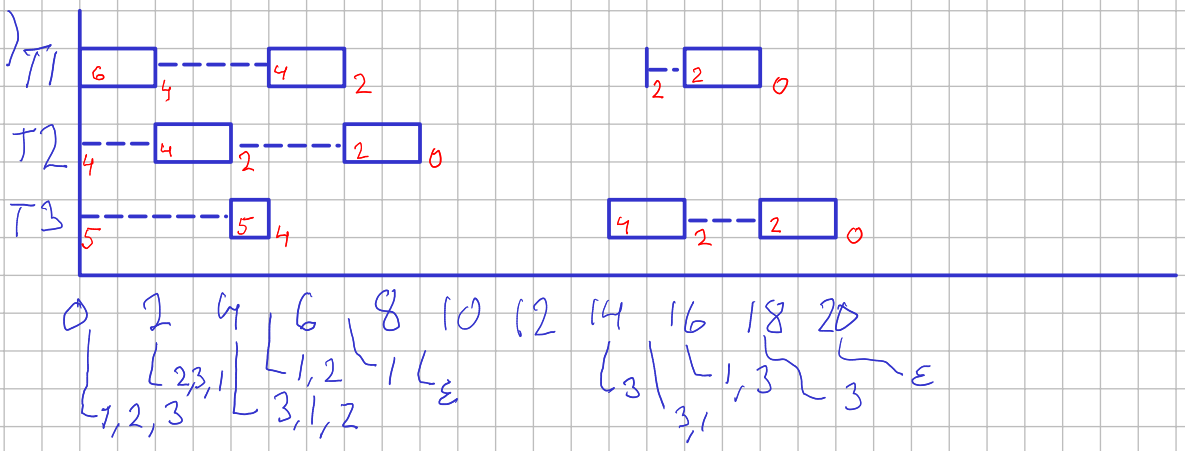
   Programmet `submit` behöver då bara kopiera filen som ska lämnas in till `/handins/<student-id>`, sätta filens grupp till `teachers`, och se till att rättigheterna är `rw-rw----`. Programmet `submit` kan exempelvis ägas av `root` och ha rättigheterna `rwx-r-xr-x`.

   Programmet `grade` ska ägas av `root` och gruppen `teachers`. För att se till att bara lärare kan köra det sätts filrättigheterna till `rwxr-x---`. Programmet kan då helt enkelt titta i `/handins` och läsa de inlämningar som finns där.

   (b) Om en student kopierar sin inlämning till `/handins/` själv så kommer den inte att få korrekt grupp och korrekta filrättigheter. Typiskt sett ägs filen av gruppen med samma namn som användaren och har antingen `rw-rw-r--` eller `rw-r--r--`. Detta gör att läraren och alla andra studenter kan läsa inlämningen (åtminstone om de kan gissa filnamnet).

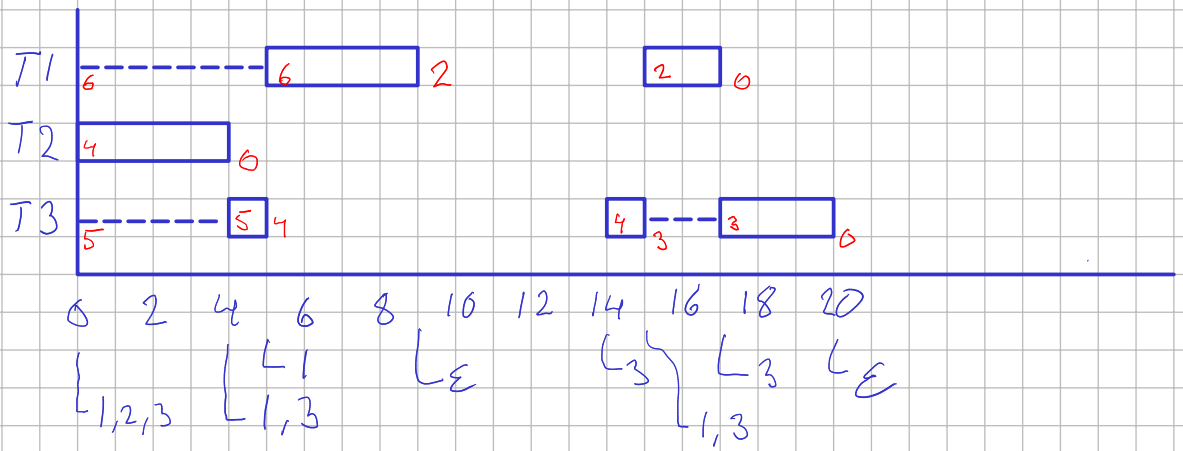
   (c) I och med att studenter inte har `x` på `/handins` så kommer de inte kunna se vilka filer som finns där (dvs. `rm /handins/*` kommer inte fungera rakt av). De kommer däremot kunna ta bort filer som de kan namnet på (student-id:n är inte så svåra att gissa).

1. a)



b) T1: 4, T2: 5, T3: 6

c)



d) T1: 5, T2: 0, T3: 6

e) i a) $1 - \frac{5}{20} = 1 - 0,25 = 75\%$

i b) $1 - \frac{5}{20} = 75\%$