Lösningsförslag till tentamen i TDIU11 2026-03-27

Filip Strömbäck

1. Se sista sidan.
2. (a) Som anges i uppgiften så delas den logiska adressen upp i 2 + 2 + 2 bitar.

   Uppslagningen sker i två steg. Vi börjar med den pagetabell som finns i registret för första nivåns pagetabell. I den tittar vi på raden som anges av de två första bitarna i den logiska adressen. Om *valid* är 1 så går vi vidare till den frame som är angiven där. Annars genererar vi ett pagefault.

   Vi gör samma sak i nästa nivå, men vi använder de mellersta två bitarna ur den logiska adressen (detta inkluderar ett eventuellt pagefault). Denna gång anger den frame vi hittar den frame som vi ska läsa från. För att få den fysiska adressen så tar vi det framenummer vi hittar och lägger på de sista två bitarna från den logiska adressen.

   Uppslagningen ger följande resultat för P1:

   - `000000`: Rad `00` i pagetabell `00001` är inte *valid*. Alltså får vi pagefault.
   - `100110`: Rad `10` i pagetabell `00001` är *valid* och pekar oss till frame `00110`. Rad `01` i pagetabell `00110` är valid och ger oss frame `00000`. Den fysiska adressen blir alltså: `0000010`.

   Uppslagningen ger följande resultat för P2:

   - `000000`: Rad `00` i pagetabell `00010` är *valid* och pekar oss till frame `00101`. Rad `00` i pagetabell `00101` är *valid* och pekar oss till frame `10100`. Den fysiska adressen blir alltså: `1010000`.
   - `100110`: Rad `10` i pagetabell `00010` är *valid* och pekar oss till frame `00011`. Rad `01` i pagetabell `00011` är inte *valid*. Alltså får vi pagefault.

   (b) Alla adresser i pagetabellerna `00011`–`00111` är unika. Den enda dublett som finns är rad `01` i `00001` och rad `00` i `00101`. Det innebär att P1 och P2 delar en nivå-2 tabell. Alltså är alla adresser `01xxxx` i P1 samma som alla adresser `00xxxx` i P2. Exempelvis är adress `010000` i P1 samma som `000000` i P2 (fysisk adress `1010000`).
3. - Givet att varje element i en pagetabell är 32-bitar stort (4 bytes á 8 bitar), och att 6 av dessa bitar används till metadata, har vi 26 bitar kvar att lagra ett framenummer i.

     Utöver det vet vi att pagestorleken är 8 KiB = 2^{13} bytes. Vi kan alltså producera $26 + 13 = 39$ bitar fysiska adresser från vår MMU. Vi kan alltså använda 2^{39} bytes = 512 GiB fysiskt minne.
   - Givet pagestorleken och storleken av elementen i pagetabellerna så vet vi att varje pagetabell kan innehålla $\frac{2^{13}}{2^2} = 2^{11}$ element.

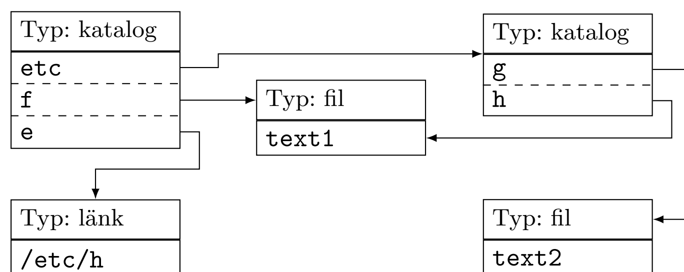
Sedan kan vi börja bygga bakifrån. Vi vet att de lägsta 13 bitarna av den logiska adressen är en offset. Lägger vi på en nivå paging så får vi logiska adresser som är $11+13=24$ bitar. Detta är för lite, så vi testar en nivå till. Det ger: $11+11+13=35$ bitar. Det är också för litet, så vi testar en nivå till. Det ger oss: $11+11+11+13=46$ bitar, vilket är acceptabelt.

Vi kan alltså välja logiska adresser som är 46 bitar stora, och 3 nivåer paging. Detta gör att alla pagetabeller är fulla. Vi vet också att 3 nivåer paging är minimalt eftersom vi har sett att 1 och 2 nivåer är för lite. (4 nivåer kommer också att fungera. Det ger oss 56 bitar virtuella adresser. Det är dock inte minimalt.)

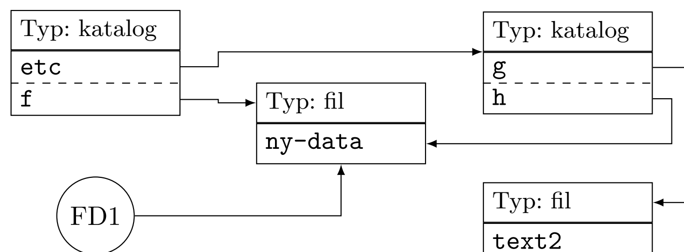
- Vi har 3 nivåer paging (n) från (b), och en accesstid på 10 ns (t). Detta ger oss en EAT enligt nedan. Hit-ratio, α är 99%, dvs. 0.99.

$$
\begin{aligned}
EAT &= \alpha \cdot t_{good} + (1-\alpha) \cdot t_{bad} \\
&= \alpha \cdot t + (1-\alpha) \cdot (nt + t) \\
&= 0.99 \cdot 10 \text{ ns} + 0.01 \cdot 4 \cdot 10 \text{ ns} \\
&= 9.9 \text{ ns} + 0.4 \text{ ns} \\
&= 10.3 \text{ ns}
\end{aligned}
$$

4. (a) Efter steg 3:



Efter steg 6:



(b) Enligt bilden ovan kan vi se att båda filerna går att öppna. `/etc/g` innehåller `text2` och `/etc/h` innehåller `ny-data`.

(c) Detta sker i steg 5 när vi tar bort den symboliska länken. Filsystemet håller reda på detta genom att lagra en referensräknare i varje inod. När denna blir 0 (och när inoden inte längre är öppen) så markeras inoden som ledig. Detta är varför ingen inod markeras som ledig i steg 2.

5. Disken är 8 TiB = 2^{43} bytes stor. Blockstorleken är 2^9 bytes.

   (a) Det finns $\frac{2^{43}}{2^9} = 2^{34}$ fysiska block på disken. Vi behöver alltså 34 bitar för att indexera alla fysiska block.

   (b) Med en logisk blockstorlek av 4 KiB = 2^{12} bytes får vi $\frac{2^{43}}{2^{12}} = 2^{31}$ logiska block.

   Vi kan notera att detta gör att vi kan indexera alla logiska block med ett 32-bitars tal. Det är bra, annars hade vi inte kunnat använda FAT32 för att komma åt hela disken!

   Varje element i FAT är 4 bytes (= 32 bitar). Detta ger en total storlek på $2^{31} \cdot 2^2$ bytes = 2^{33} bytes = 8 GiB.

   (c) Om vi använder 512 bytes logiska block så har vi enligt (a) 2^{34} block. Vi klarar oss alltså inte på ett 32-bitars tal för att indexera alla block. Vi väljer alltså 64-bitars tal, vilket gör att varje element i indexblocken blir 8 bytes stort. Vi kan alltså ha $\frac{2^9}{2^3} = 2^6$ element i varje indexblock.

   2 nivåers indexering ger oss alltså en maximal filstorlek av $2^6 \cdot 2^6 \cdot 2^9$ bytes = 2^{21} bytes = 2 MiB.

   (d) Så länge vi har en logisk blockstorlek som är större än 2^11 bytes så klarar vi oss med 32-bitars element i indexblocken (uträkningen i (b) visade att vi var en bit ifrån med 2^12 bytes).

   Antag att blockstorleken är 2^n bytes ($n \geq 11$). Då kan varje indexblock innehålla $\frac{2^n}{2^2} = 2^{n-2}$ element.

   Utifrån detta så blir den maximala filstorleken: $2^{n-2} \cdot 2^{n-2} \cdot 2^n$ bytes. Vi vill nu hitta ett n som löser olikheten:

$$
\begin{aligned}
& 2^{n-2} \cdot 2^{n-2} \cdot 2^n \geq 2^{35} \\
\iff & (n-2) + (n-2) + n \geq 35 \\
\iff & 3n - 4 \geq 35 \\
\iff & 3n \geq 39 \\
\iff & n \geq 13
\end{aligned}
$$

   Vi kan alltså välja $n = 13$, vilket ger en blockstorlek på 2^{13} bytes = 8 KiB.

   (Det går självklart också bra att testa sig fram. $n = 13$ är nära gränsen till 32-bitarspekare som är $n = 11$.)

6. (a) Vi kan lösa detta genom att låta användaren `boss` äga både katalogen `/time` och alla filer där under. Notera att vi *inte* vill låta anställda äga sina egna filer i `/time`, eftersom de då kan använda `chmod` för att modifiera filrättigheterna.

   Programmet `/bin/work` måste vara ett `setuid`-program för att kunna skriva till filerna under `/time`. Intressant nog så räcker det om `/bin/work` körs som `boss` (dvs. det behöver *inte* köras som `root`).

   För scenariot i uppgiften så skulle systemet se ut som följer:

- Vi behöver inga extra grupper. Jag antar dock att varje användare har en egen grupp (som Linux/UNIX-system brukar vara uppsatta). Det finns alltså en grupp som heter `filst04` som innehåller användaren `filst04`, och en grupp `chrho44` som innehåller användaren `chrho44`.
- `/time`: ägare `boss`, grupp `boss`, rättigheter `rwxr-xr-x`
- `/time/filst04`: ägare `boss`, grupp `filst04`, rättigheter `rw-r-----`
- `/time/chrho44`: ägare `boss`, grupp `chrho44`, rättigheter `rw-r-----`
- `/bin/work`: ägare `boss`, grupp `boss`, rättigheter `rwsr-xr-x` (första `s` anger setuid)

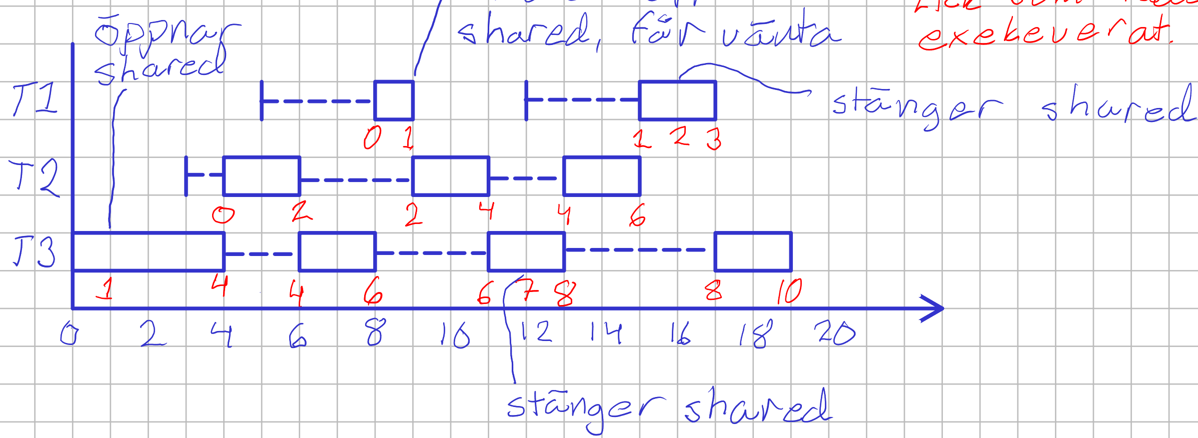
(Det går också att använda ACL:er i stället för grupper för filerna under `/time`)

(b) Största risken i svaret ovan är setuid-programmet. Om programmet innehåller buggar så finns en risk att en anställd kan köra kod som `boss`, vilket gör det möjligt att se och ändra i alla filer under `/time`. En positiv sak med att `/bin/work` inte körs som `root` är att programmet inte kan utnyttjas för att göra något som `boss` inte får göra.

Streckad linje = i ready-kö

Röd text anger antal tick som tråden har exekverat.

a) Round Robin



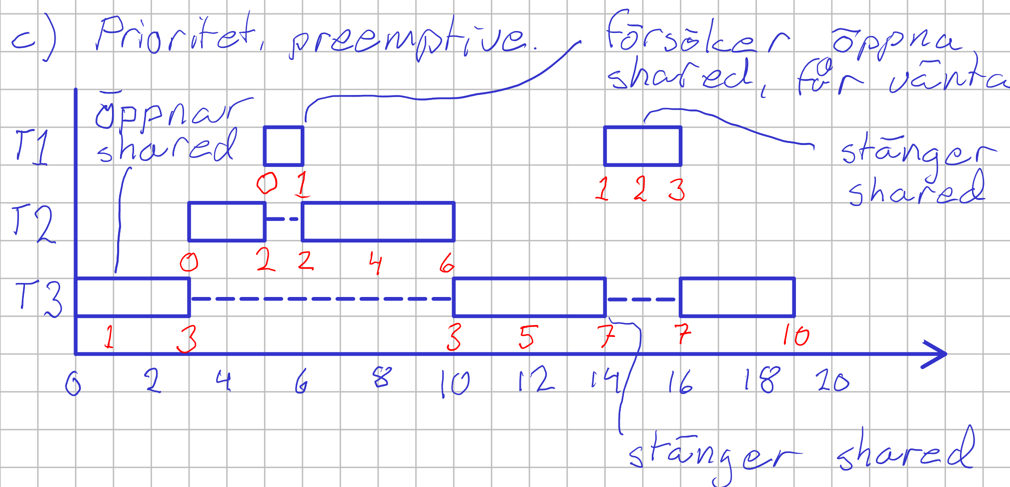
b) Waiting time – Vi kan räkna tid i streckad linje.

T1: 6

T2: 6

T3: 9

c) Prioritet, preemptive.



d) Det som händer i (c) är "priority inversion".

Det kan lösas med "priority inheritance", dvs att T3 ärver prioriteten av de trådar som väntar på shared när T3 har den öppen.