Karaktergivende opgave i "Styresystemer og multiprogrammering"

Praktisk opgave

I spørgsmålene refereres til kursets lærebog, Silberschatz, Galvin, Gagne: Operating System Concepts. 7th Edition, 2005 [SGG'05].

Besvarelsen skal indeholde en rapport på maksimalt 5 maskinskrevne sider, plus et bilag med kildetekst. Hvad rapporten skal indeholde er nærmere beskrevet i hver delopgave.

A1 (60%)

De praktiske opgaver omhandler hukommelsessystemet på en 32-bit arkitektur. Mere præcist TLB'en (translation look-aside buffer) og sidetabellen, samt samspillet mellem disse.

a) TLB udskiftningsalgoritme (30%)

I denne opgave skal du ændre sideudskiftningsalgoritmen i en software-baseret TLB implementation. Lærebogen [SGG'05] beskriver i afsnit 8.4.2 hvordan en TLB virker.

I filen 'tlb.c' fra den udleverede kildetekst finder du implementationen af TLB'en. 'tlb-test.c' er et eksempel på hvordan TLB'en anvendes. TLB'en er normalt en del af CPU'en, men er her bare en datastruktur med to operationer:

```
/*
Indsætter den logiske side 'logical' som refererer til den fysiske side
(frame) 'physical'. Parameteren 'protection' specificerer hvilke
tilgangsrettigheder der skal være til siden (read og/eller write).
*/
void tlb_insert(uint32_t logical, uint32_t physical, uint16_t protection);

/*
Laver et opslag på den logiske adresse 'logical'. Den tilsvarende
fysiske side gemmes i den variabel som 'physical' peger på. Funktionen
returnerer tilgangsrettighederne.

Hvis siden for den logiske adresse ikke findes i TLB'en kaldes
funktionen 'tlb_fault'.
*/
uint16_t tlb_lookup(uint32_t logical, uint32_t *physical);
```

En rigtig TLB ville have flere operationer, som f.eks. en operation til at slette elementer i bufferen, men dette er ikke nødvendigt i denne opgave.

Selve datastrukturen er et array af følgende type:

```
typedef struct {
    uint32_t logical;       // Logisk side
    uint32_t physical;      // Fysisk side
    uint16_t protection;    // Tilgangsrettigheder
    uint16_t flags;         // Flag, TLB_USED, TLB_REFERENCED
} tlb_entry_t;
```

Array'et har en fast længde som er defineret i konstanten 'TLB_SIZE'. Dette betyder at bufferen på et tidspunkt vil blive fyldt, og at der derfor skal tages en beslutning om, hvilket element der skal udskiftes til fordel for det nye. Metoden for denne udskiftning kaldes udskiftningsalgoritmen (replacement algorithm).

Den udleverede TLB implementation bruger en algoritme som tilfældigt vælger det element der skal udskiftes (se 'tlb_insert'). Der tages altså ikke højde for tilgangsmønsteret, sådan at ofte brugte sider ikke udskiftes.

Problematikken er den samme som for sideudskiftning ved *demand paging* (lærebog [SGG'05] afsnit 9.2). Derfor kan de samme algoritmer bruges, med den forskel at TLB'er har nogle langt højere hastighedskrav og skal kunne implementeres effektivt i hardware.

Din opgave er at forstå TLB implementationen og ændre den til at bruge *second-chance* algoritmen (lærebog [SGG'05] afsnit 9.4.5.2). Dvs. at du skal ændre 'tlb_insert' funktionen i 'tlb.c' så den bruger *second-chance*. Derudover skal du lave et program i 'tlb-test.c' som tester at din implementation fungerer korrekt. Til hjælp for testen, kan du eventuelt udskrive hvilket indeks fra bufferen som bliver udskiftet i 'tlb_insert'.

Besvarelsen af dette delspørgsmål skal indeholde:

- Rapport: Beskrivelse af implementationen og testen, samt gennemgang af testresultater.
- Bilag: Kildetekst.

b) Hukommelsestilgang (30%)

I denne opgave skal du lave en simulation af hukommelsestilgangen for en brugerproces. Dvs. de skridt der foretages af CPU og operativsystem, når en proces tilgår en logisk adresse. Det er kun TLB og sidetabel delen af disse skridt som skal behandles.

Den udleverede kildetekst indeholder en implementation af en to-niveau sidetabel ('page-table.c') som beskrevet i lærebogen [SGG'05] afsnit 8.5 og figur 8.14. Sidetabellen har følgende operationer:

```
/*
Indsætter side informationen fra 'page' på pladsen for den logiske side
'logical'. Det antages at funktionen ikke kan fejle.
*/
void pagetable_set(uint32_t logical, page_entry_t *page);

/*
Returnerer side informationen for den logiske side 'logical'.
Hvis den logiske side ikke eksisterer returneres 'NULL'.
*/
page_entry_t *pagetable_get(uint32_t logical);
```

Information om en logisk side indsættes og udtages vha. typen 'page_entry_t', hvilket giver mere fleksibilitet, da man let kan ændre typen. Operativsystemer vedligeholder oftest meget

information om sider, til f.eks. *demand paging* og *memory mapped I/O*. Dette skal ikke bruges i denne opgave, så typen har bare følgende definition ('pagetable.h'):

```
typedef struct {
    uint32_t physical;      // Logisk side
    uint16_t protection;    // Tilgangsrettigheder
} page_entry_t;
```

I 'mem-test.c' kan du se et eksempel på brug af sidetabellen.

Simulationen af de skridt der skal udføres når en proces tilgår hukommelsen skal du implementere i 'mem.c', ved at udfylde tre funktioner:

```
/*
Svarer til at en proces læser den logiske adresse 'logical'.
Den tilsvarende fysiske adresse returneres, eller '0' ved fejl.
*/
uint32_t mem_read(uint32_t logical);

/*
Svarer til at en proces skriver den logiske adresse 'logical'.
Den tilsvarende fysiske adresse returneres, eller '0' ved fejl.
*/
uint32_t mem_write(uint32_t logical);

/*
Hvis TLB'en ikke kan finde den logiske adresse 'logical'
i 'tlb_lookup' kaldes denne funktion. Hvis den returnerer
'0' stopper TLB'en med opslaget, ellers gentages opslaget.
*/
int tlb_fault(uint32_t logical);
```

Funktionerne 'mem_read' og 'mem_write' simulerer hhv. at processen læser eller skriver. I disse funktioner skal der laves opslag i TLB'en for at teste adressen. Det kan resulterer i følgende fejl, som du skal håndtere ved at udskrive en fejlbesked og returnere '0':

- **Protection fault**: Processen har ikke rettigheder til at læse/skrive fra/til adressen.
- **Illegal address fault**: Den logiske adresse er ikke en del af processens gyldige adresser og er derfor ikke tildelt en fysisk adresse.

Hvis tilgangen er i orden, returneres den fysiske adresse.

Hvis TLB'en ikke kan finde adressen kaldes 'tlb_fault'. I denne skal du lave opslag i sidetabellen og opdatere TLB'en, hvis muligt. Figur 8.11 i lærebogen [SGG'05] giver et samlet overblik over det der ønskes implementeret.

Til sidst skal du lave et program som tester din implementation i filen 'mem-test.c'. Dvs. sidetabellen skal fyldes med sider, hvorefter hukommelsen skal tilgås på en måde der viser at de forskellige dele af din implementation fungerer.

Besvarelsen af dette delspørgsmål skal indeholde:

- Rapport: Beskrivelse af implementationen og testen, samt gennemgang af testresultater.
- Bilag: Kildetekst.

K-opgave: Exam Set A.

						–				

Write your CPR number in the field above.
Important: This exam set is for students with an odd birthday (1, 3, ..., 31).

Write *all* answers into the boxes below. No other forms or papers will be accepted as hand-in for the following three questions (A2, A3, A4).

Read the questions *carefully* before you start. Double check your answers before you write them into the boxes. Write your answers in a clear and readable style. English and Danish are both acceptable. All questions refer to the terminology and algorithms in the course book Silberschatz, Galvin, Gagne: "*Operating System Concepts. 7th Edition*", Wiley 2005 (referenced as [SGG'05]).

A2. Demand Paging (10%).

a) Basic concepts: Under what circumstances do *page faults* occur? Describe shortly the actions taken by the operating system when a page fault occurs.

Sidefejl kan generelt forekomme i to tilfælde: når der forsøges at hente en ugyldig adresse i hukommelsen, eller når der forsøges at hente en side, der ikke er indlæst i hukommelsen endnu. Demand paging sørger netop for kun at hente de sider ind i hukommelsen, der bliver bedt om.

Sidefejl kan fanges ved at tilføje f.eks. en valid-invalid bit i sidetabellen; hvis denne står til invalid, falder siden ind under én af ovenstående kategorier.

Beder en process om en side, der er invalid, trappes styresystemet. Hvis der er tale om en ugyldig adresse, afbrydes processen med besked om ulovlig tilgang til hukommelse. Hvis der er tale om en side, der blot ikke er i hukommelsen, sker følgende:

- Styresystemet tilgår sin liste af frie frames og finder en ledig frame, som den ønskede side hentes ind i fra disken. Dette kan godt give ventetid, hvis disken er optaget, og i dette tilfælde kan CPU'en skifte til en anden process og arbejde på denne imens.

- Adressen på denne frame skrives ind i sidetabellen, som opdateres (adresse og valid-invalid bitten).

- Styresystemet genstarter nu den instruktion i processen, der forårsagede sidefejlen. Denne instruktion kan nu fortsætte, som om intet var hændt og få den relevante side fra hukommelsen.

b) Programs and paging: Which of the following data structures and programming techniques are “good” for a demand-paged system? Which are “bad”? Explain your answers shortly.

Stack:

En stak kan fint anvendes sammen med demand paging. Stakken bliver udvidet og formindsket sekventielt (altså ting lægges på og fjernes i samme rækkefølge). Stakken er en lineær datastruktur, hvilket oftest vil kunne gøre brug af locality-princippet ("locality of reference"), der siger, at det er forholdsvis usandsynligt, at elementerne på f.eks. stakken for en process ligger spredt rundt omkring i hukommelsen.

Hashed symbol table:

Er ikke så godt med demand paging. En hash-tabel vil hurtigt kunne finde det relevante symbol, men derefter skal der hoppes et andet sted hen i hukommelsen for rent faktisk at tilgå information om symbolet. Denne information kan og vil ofte ligge i en anden side.

Sequential search:

Sekventiel søgning er udmærket med demand paging, da søgningen foregår sekventielt, hvilket giver os den fordel, at vi ikke skal hoppe så meget rundt i hukommelsen. Dette er ofte en konsekvens af locality-princippet (eller "locality of reference), der siger, at et program oftest tilgår hukommelse i den samme region (locality), hvilket fint harmonerer med en sekventiel søgning.

Binary search:

Binær søgning er ikke så godt med demand paging. Tænk på en situation, hvor man skal finde et element i en sorteret liste vha. binær søgning. Vi starter med at se, om elementet er større eller mindre end det midterste og leder så videre efter samme princip i én af halvdelene af listen. Vi så at sige springer rundt i listen for at finde den rette plads, hvilket potentielt ville kunne betyde, at vi skal bede om en ny side, hver gang vi hopper.

c) Effective access time: Assume you have a demand-paged memory system. The page table is held in registers. *Memory-access time* is 100 nanoseconds. It takes 8 milliseconds to service a *page fault* if an empty frame is available or the replaced page is not modified, but 22 milliseconds if the replaced page is modified and has to be saved to the paging disk. Assume that the page to be replaced is modified in 60% of all page faults. What is the maximum acceptable *page-fault rate p* ($0 \leq p \leq 1$) for an *effective access time of 200 nanoseconds*? Show your calculation and the result.

Vi beregner den gennemsnitlige tid for at behandle en sidefejl:

avg. pagefault time = 0.40 * 8 milisec + 0.60 * 22 milisec = 16.4 milisec.

Og følger derefter fremgangsmåden på side 325 i bogen:

effective access time = (1-p) * ma + p * page fault time

200 nanosec > (1-p) * 100 nanosec + p * 16.400.000 nanosec

=> 200 > 100 + p * 16.399.900 => 100 > p * 16.399.900 => p < 0.0000061

Altså kan en værdi af p under ca. 0.0000061 accepteres.

Thus, one cannot allow more than 1 page fault out of

10.000.000 / 61 ≈ 163935 memory accesses.

A3. Page-replacement algorithms (15%).

a) Assume that you have a page-reference string for a process with 3 frames. For each of the following three page-replacement algorithms, show which pages are replaced. All frames are initially empty, so all of your first unique pages will cost one page fault each.

	1	**2**	**3**	**2**	**4**	**3**	**1**	**2**	**5**	**1**	**5**	**2**	**3**	**4**	**5**
FIFO	1	1	1		4		4	4	5				5	5	
		2	2		2		1	1	1				3	3	
			3		3		3	2	2				2	4	

	1	**2**	**3**	**2**	**4**	**3**	**1**	**2**	**5**	**1**	**5**	**2**	**3**	**4**	**5**
LRU	1	1	1		4		4	2	2				2	2	5
		2	2		2		1	1	1				3	3	3
			3		3		3	3	5				5	4	4

	1	2	3	2	4	3	1	2	5	1	5	2	3	4	5
OPT	1	1	1		1			1	1				3	3	
		2	2		4			2	2				2	4	
			3		3			3	5				5	5	

How many page faults occur for each of the page-replacement algorithms?

3 frames	FIFO	LRU	OPT
Page faults	9	10	8

b) Consider the same page reference string and allocate 4 frames to the process. Show which pages are replaced and how many page faults occur with the FIFO strategy.

	1	2	3	2	4	3	1	2	5	1	5	2	3	4	5
FIFO	1	1	1		1				5	5		5	5	4	4
		2	2		2				2	1		1	1	1	5
			3		3				3	3		2	2	2	2
					4				4	4		4	3	3	3

4 frames	FIFO
Page faults	10

c) Which effect do you observe when comparing the number of page faults versus the number of frames using the FIFO algorithm in parts (a) and (b)? Describe the effect.

Ved at tildele processen 1 frame mere får vi faktisk FIFO til at lave én sidefejl mere også. Dette kaldes Beladys anomali og husker os på, at ved at tildele mere hukommelse til en proces får vi ikke nødvendigvis færre sidefejl. I de fleste tilfælde vil det dog hjælpe på det, ligesom det er ganske afhængigt af den rækkefølge, hvori hukommelsestilgangene forekommer.

d) Which of the following page-replacement algorithms suffer from the same problem? Circle the corresponding box:

OPT ☐[yes] ☑[no]
LRU ☐[yes] ☑[no]

A4. Memory Management (15%). Consider the *Intel Pentium* address-translation scheme as described in Chapter 8.7 [SGG'05][*)] which supports *segmentation with two-level paging*. The following questions deal with this address-translation scheme.

a) Describe shortly the main steps taken by the Intel Pentium in translating a *logical address* (consisting of a selector and an offset) into a 32-bit *linear address*.

Den logiske adresse består af en selector (16 bit) og et offset (32 bit). Først afkodes selectoren (16 bit) i dens 3 bestanddele: s (13 bit), g (1 bit) og p (2 bit). Værdien af g afgør, om der skal slås op i LDT eller GDT. Der slås så op ud fra værdien af s.

Dette opslag giver os en limit og en base for det pågældende segment. Først tjekkes limit op mod vores 32 bit offset for at se, om processen tilgår en lovlig del af hukommelsen; ellers trappes med en fejl.

Derefter adderes vores fundne base med vores oprindelige 32 bit offset fra den logiske adresse, hvilket giver os vores 32 bit lange lineære adresse.

b) Describe shortly the main steps taken by the Intel Pentium in translating a *linear address* into a 32-bit *physical address*.

Pentium-processoren understøtter sidestørrelser på både 4KB og 4MB.

Den ovenfor fundne lineære adresse opdeles i 3 dele: p1 (ydre sidetabel, 10 bit), p2 (indre sidetabel, 10 bit) og d (offset, 12 bit).

p1 bruges til at lave et opslag i den ydre sidetabel. I denne findes et flag (Page size), der afgør, om der er tale om 4KB eller 4MB sider. Er der tale om 4KB, indeholder den ydre sidetabel adressen på en indre sidetabel. Er der tale om 4MB, indeholder den direkte adressen på en 4MB blok i hukommelsen, som adderes med offsettet for at finde den fysiske adresse.

I tilfældet med 4KB får vi adressen på den indre sidetabel, hvori der slås op efter p2. Her findes så adressen på den pågældende frame i hukommelsen, som adderes med offsettet for at ramme den rigtige del af framen.

I begge tilfælde ganges den fysiske adresse med 4K.

Consider the following information about a process: the global descriptor table (GDT), the local descriptor table (LDT), the page directory, and two page tables. The page frame size is the standard 4KB; there are no 4MB pages. In the selector of a logical address, a table indicator *g*=0 refers to the GDT and *g*=1 refers to the LDT. The protection *p* is present in the selector, but unused during the address translation.

GDT (*g*=0)

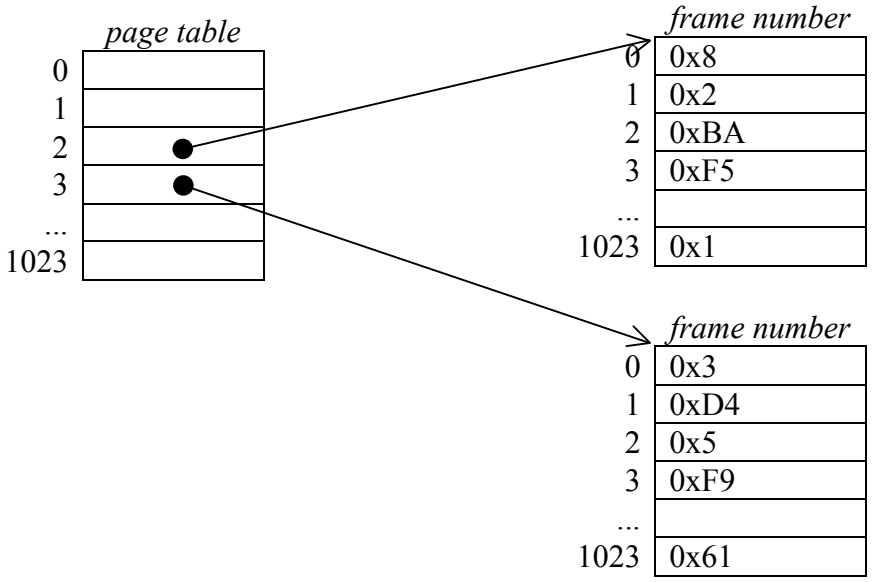
	base	*limit*
0	0xC01000	0xC3
...		
184	0xC03000	0x16B1
185	0xFFF000	0x5
...		
8191	0x802000	0x27

LDT (*g*=1)

	base	*limit*
0	0xC02000	0x78
...		
53	0x800000	0xB81
54	0x801000	0x7A1
...		
8191	0xBFF000	0x605

Page directory

Page tables



What are the linear addresses and the physical addresses for the following logical addresses? Give your answers as hexadecimal numbers on the next page. As usual, hexadecimal numbers are prefixed with "0x"; decimal numbers have no prefix.

1. selector = 0x1B5, offset = 0x3.

2. selector = 0x5C3, offset = 0x5A1.

3. selector = 0xFFF8, offset = 0x534.

c) Segmentation unit. Decode the selector:

	1.	2.	3.
Selector	**0x1B5**	**0x5C3**	**0xFFF8**
s	0x 36	0x B8	0x 1FFF
g	0x 1	0x 0	0x 0
p	0x 1	0x 3	0x 0

d) Segmentation unit. Compute the linear address from the logical address, and decode the linear address:

	1.	2.	3.
Linear address	0x 801003	0x C035A1	0x TRAP
p_1	0x 2	0x 3	0x TRAP
p_2	0x 1	0x 3	0x TRAP
d	0x 3	0x 5A1	0x TRAP

e) Paging unit. Compute the physical address from the linear address:

	1.	2.	3.
Physical address	0x 5000	0x 69A000	0x TRAP

*) Errata for [SGG'05], Chapter 8.7:
Page 305, line -1: 16 KB → 16 K
Page 306, line 2: 8 KB → 8 K
Page 306, line 3: 8 KB → 8 K
Page 308, Fig. 8.23, top: logical → linear