Styresystemer og Multiprogrammering
K-opgave 2008

Anders Bjerg Pedersen, 070183-XXXX

6. april 2008

A1

Generelt skal der knyttes et par kommentarer, før vi begynder:

- Al kildekode kan oversættes og køres fejlfrit på både DIKUs Linux-systemer og på en lokal OSX-maskine ved hjælp af den udleverede Make-fil.
- Al kildekode er udskrevet om bilag sidst i rapporten og ligger desuden i elektronisk form på DIKUs systemer under:
  /home/disk22/di070106/4-3-OSM/Eksamen/src-exam/.

a) TLB udskiftningsalgoritme

I denne opgave ønskes implementationen af TLB-udskiftningsalgoritmen ændret til second-chance algoritmen. Eneste nævneværdige ændringer er i funktionen `tlb_insert()`, hvor vi i stedet for at udvælge en tilfældig side, der skal udskiftes, vælger en vha. second-chance algoritmen. Vi anvender hertil en global variabel, `chance` (ligger mellem 0 og `TLB_SIZE`), der angiver, hvilket sted i TLB'en vi næste gang skal anvende algoritmen. Altså hvilket entry, der næste gang skal testes som offer. Funktionen leder først efter et tomt element. Hvis et sådant ikke findes, starter et while-loop på plads `chance` og leder fremad, indtil den finder et entry, hvor `TLB_REFERENCED` ikke er sat. Dette skiftes ud med det nye element. Undervejs gives der second chances til eventuelle elementer med `TLB_REFERENCED` sat.

Til testen fylder vi først TLB'en med 8 elementer og tjekker undervejs, at `tlb_insert()` rent faktisk indsætter de korrekte ting på de korrekte pladser. Ved næste insert skulle second-chance gerne løbe alle elementer igennem, unsette deres `TLB_REFERENCED` og indsætte næste element på plads 0. Derefter refererer vi elementer på plads 1, 2, 3 og 5, hvilket gerne skulle betyde, at second-chance ved næste indsættelse skulle ramme plads 4. Til sidst prøver vi at referere en logisk adresse, der ikke findes i TLB'en (en TLB miss). Output af testen ses herunder:

```
brok > ./tlb-test
Indsat TLB entry, log: 0x16000, phys: 0x1000 på plads 0
Indsat TLB entry, log: 0x26000, phys: 0x1010 på plads 1
Indsat TLB entry, log: 0x36000, phys: 0x1020 på plads 2
0x16000: physical=0x1000 writeable=1 readable=1.
Indsat TLB entry, log: 0x46000, phys: 0x1030 på plads 3
Indsat TLB entry, log: 0x56000, phys: 0x1040 på plads 4
Indsat TLB entry, log: 0x66000, phys: 0x1050 på plads 5
0x26000: physical=0x1010 writeable=0 readable=1.
Indsat TLB entry, log: 0x76000, phys: 0x1060 på plads 6
Indsat TLB entry, log: 0x86000, phys: 0x1070 på plads 7
0x36000: physical=0x1020 writeable=1 readable=1.
Offer fundet på plads 0
Indsat TLB entry, log: 0x96000, phys: 0x1080 på plads 0
0x26000: physical=0x1010 writeable=0 readable=1.
0x36000: physical=0x1020 writeable=1 readable=1.
0x46000: physical=0x1030 writeable=0 readable=1.
0x66000: physical=0x1050 writeable=0 readable=1.
Offer fundet på plads 4
Indsat TLB entry, log: 0x106000, phys: 0x1090 på plads 4
0x106000: physical=0x1090 writeable=0 readable=1.
0xfff000: no mapping.
brok >
```

Det ses af testen, at `tlb_insert()` fungerer efter hensigten; specielt at second-chance indsætter det 9. element på plads 0 og det 10. element på plads 4.

b) Hukommelsestilgang

I denne opgave skal de tre funktioner `mem_read()`, `mem_write()` og `tlb_fault()` implementeres, så de simulerer en proces' muligheder for at læse og skrive til hukommelse. De to førstnævnte funktioner er konstrueret, så de tjekker adgangstilladelser på allerede eksisterende frames i TLB og sidetabel. Hvis de har tilladelse til at læse/skrive, returnerer funktionerne de pågældende fysiske adresser. Herunder følger en kort gennemgang af implementationen af de tre funktioner:

tlb_fault():

`tlb_fault()` kaldes, hvis vi får et TLB miss, altså at den pågældende side ikke findes i TLB'en. Funktionen forsøger herefter at slå siden op i sidetabellen. Finder den ikke siden i sidetabellen (`pagetable_get()` returnerer `NULL`), returnerer funktionen et 0, der får TLB'en til at stoppe opslaget og returnere 0 til `mem_read()` eller `mem_write()`. Hvis siden derimod findes, indsættes den i TLB'en, som fortsætter søgningen, finder siden og returnerer den fysiske adresse til den kaldende funktion.

mem_read():

`mem_read()` starter med at lave et TLB-lookup. Hvis dette returnerer 0, betyder det (jævnfør ovenstående), at den ønskede logiske adresse ikke findes i hverken TLB'en eller sidetabellen. Der meldes så "Illegal address fault!"til skærmen, og der returneres 0. Ellers testes det, om der må læses fra filen (hvis `protection & 1 != 0`,

dvs. hvis den mindst betydende bit er sat). Hvis ja returneres den fysiske adresse; ellers skrives "Protection fault!"til skærmen, og der returneres 0.

mem_write():
`mem_write()` er næsten fuldstændig identisk med `mem_read()`, blot testes der her efter skriverettigheder i stedet for læserettigheder. Der må skrives til framen, hvis `(protection >> 1) & 1` (dvs. hvis den anden-mindst betydende bit er sat).

Til testen starter vi med at køre de udleverede tests af `pagetable.c` for at vise, at disse stadig virker uændret. Outputtet er delt op i 3 for at lette beskrivelsen. Output af første testpulje ses herunder:

```
brok > ./mem-test

************ BEGYND UDLEVERET TEST ************

0x56000: physical=0x1000 writeable=0 readable=1.
0x560f0: physical=0x1000 writeable=0 readable=1.
0x80fff: physical=0x23000 writeable=1 readable=1.
0x81fff: no mapping.
```

Det ses, at de på forhånd implementerede funktioner sjovt nok stadig virker...

Dernæst fylder vi en ekstra side i sidetabellen (én der kun må skrives til). Vi bruger så `mem_read()` til at forsøge at læse forskellige sider i hukommelsen. Først forsøger vi at læse en side, der har både læse- og skriveadgang, dernæst en kun med læseadgang, en kun med skriveadgang og til sidst en, som slet ikke findes i hukommelsen. Her skulle testen så gerne resultere i to succesfulde læsninger, en protection fault og en illegal address fault. Outputtet ses herunder:

```
************ BEGYND EGEN TEST, mem_read() ************

Indsat TLB entry, log: 0x80000, phys: 0x23000 på plads 0
mem_read() læst 0x80000, fysisk adresse retur: 0x23000
Indsat TLB entry, log: 0x56000, phys: 0x1000 på plads 1
mem_read() læst 0x56000, fysisk adresse retur: 0x1000
Indsat TLB entry, log: 0x17000, phys: 0x10000 på plads 2
FEJL: Protection fault!
FEJL: Illegal address fault!
```

Det ses heraf, at `mem_read()` opfører sig korrekt i forhold til forventningerne. Det kan også bemærkes, at TLB'en indsætter de to adresser undervejs, når `mem_read()` beder om dem.

Til sidst testes `mem_write()` på de eksisterende sider i sidetabellen. Igen forsøger vi først at skrive til en side, der har både læse- og skriveadgang, en kun med skriveadgang, dernæst en kun med læseadgang og til sidst en, som slet ikke findes i hukommelsen. Igen skulle testen så gerne resultere i to succesfulde skrivninger, en protection fault og en illegal address fault. Outputtet ses herunder:

```
************ BEGYND EGEN TEST, mem_write() ************

mem_write() skrevet 0x80000, fysisk adresse retur: 0x23000
mem_write() skrevet 0x17000, fysisk adresse retur: 0x10000
FEJL: Protection fault!
FEJL: Illegal address fault!

brok >
```

Igen ses det, at `mem_write()` opfører sig som forventet. Da alle gyldige sider allerede ligger i sidetabellen, sker der ikke andet end opslag i TLB'en.

Bilagsoversigt

Bilag 1: Makefile (uændret)

```
CC = gcc
CCFLAGS = -Wall -O2 -g -I.

all: tlb-test mem-test

tlb.o: tlb.c tlb.h mem.h
    $(CC) $(CCFLAGS) -c tlb.c

pagetable.o: pagetable.c pagetable.h
    $(CC) $(CCFLAGS) -c pagetable.c

mem.o: mem.c mem.h tlb.h pagetable.h
    $(CC) $(CCFLAGS) -c mem.c

tlb-test: tlb-test.c tlb.o mem.h
    $(CC) $(CCFLAGS) -o tlb-test tlb-test.c tlb.o

mem-test: mem-test.c tlb.o pagetable.o mem.o pagetable.h mem.h
    $(CC) $(CCFLAGS) -o mem-test mem-test.c tlb.o pagetable.o mem.o


clean :
    rm -f *.o *~ tlb-test mem-test
```

Bilag 2: mem-test.c (ændret)

```
#include <stdio.h>

#include "mem.h"
#include "pagetable.h"

void page_info(uint32_t logic) {
    page_entry_t *pe;

    pe = pagetable_get(logic);

    if (pe == NULL) {
        printf("0x%x: no mapping.\n", logic);
        return;
    }

    printf("0x%x: physical=0x%x writeable=%d readable=%d.\n",
            logic,
            pe->physical,
            (pe->protection & MEM_WRITE) != 0,
            (pe->protection & MEM_READ) != 0
    );
```

```
}

int main() {
    page_entry_t pe;
    uint32_t physical;

    pagetable_init();   // Intialisér sidetabel

    /*
    Vi indsætter elementer i sidetabellen og tester, om de er blevet korrekt indsat
    (dette er den oprindelige udleverede test, umodificeret).
    */

    printf("\n************ BEGYND UDLEVERET TEST ************\n\n");

    pe.physical = 0x1000;
    pe.protection = MEM_READ;
    pagetable_set(0x56000, &pe);

    pe.physical = 0x23000;
    pe.protection = MEM_READ | MEM_WRITE;
    pagetable_set(0x80000, &pe);

    page_info(0x56000);
    page_info(0x560f0);
    page_info(0x80fff);

    page_info(0x81fff);


    /*
    Vi indsætter nu et element mere i sidetabellen og tester med
    mem_read(), om siderne kan tilgås korrekt mht. tilladelser.
    De to første skulle gerne gå godt, den tredje skulle gerne give en protection
    fault, og den sidste skulle gerne give en illegal address fault.
    */

    printf("\n************ BEGYND EGEN TEST, mem_read() ************\n\n");

    pe.physical = 0x10000;
    pe.protection = MEM_WRITE;
    pagetable_set(0x17000, &pe);

    // Forsøg at læse side med både MEM_READ og MEM_WRITE tilladelser:
    physical = mem_read(0x80000);
    if(physical != 0)
        printf("mem_read() læst 0x80000, fysisk adresse retur: 0x%x\n", physical);

    // Forsøg at læse side med kun MEM_READ tilladelser:
    physical = mem_read(0x56000);
    if(physical != 0)
        printf("mem_read() læst 0x56000, fysisk adresse retur: 0x%x\n", physical);
```

```
    // Forsøg at læse side med kun MEM_WRITE tilladelser:
    physical = mem_read(0x17000);
    if(physical != 0)
        printf("mem_read() læst 0x17000, fysisk adresse retur: 0x%x\n", physical);

    // Forsøg at læse side, der ikke findes i TLB eller sidetabel:
    physical = mem_read(0x100000);
    if(physical != 0)
        printf("mem_read() læst 0x100000, fysisk adresse retur: 0x%x\n", physical);


    /*
    Vi tester nu mem_write() på de eksisterende indgange i sidetabellen.
    De to første skulle gerne gå godt, den tredje skulle gerne give en protection
    fault, og den sidste skulle gerne give en illegal address fault.
    */

    printf("\n************ BEGYND EGEN TEST, mem_write() ************\n\n");

    // Forsøg at skrive side med både MEM_READ og MEM_WRITE tilladelser:
    physical = mem_write(0x80000);
    if(physical != 0)
        printf("mem_write() skrevet 0x80000, fysisk adresse retur: 0x%x\n", physical);

    // Forsøg at læse side med kun MEM_WRITE tilladelser:
    physical = mem_write(0x17000);
    if(physical != 0)
        printf("mem_write() skrevet 0x17000, fysisk adresse retur: 0x%x\n", physical);

    // Forsøg at læse side med kun MEM_WRITE tilladelser:
    physical = mem_write(0x56000);
    if(physical != 0)
        printf("mem_write() skrevet 0x56000, fysisk adresse retur: 0x%x\n", physical);

    // Forsøg at læse side, der ikke findes i TLB eller sidetabel:
    physical = mem_write(0x100000);
    if(physical != 0)
        printf("mem_write() skrevet 0x100000, fysisk adresse retur: 0x%x\n", physical);

    printf("\n");

    return 0;
}
```

Bilag 3: mem.c (ændret)

```
#include <inttypes.h>
#include <stdio.h>

#include "mem.h"
#include "tlb.h"
#include "pagetable.h"

/*
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) {
    uint32_t physical;
    uint16_t protection;

    protection = tlb_lookup(logical, &physical);

    if(protection == 0) {   // Adressen findes ikke i hverken TLB eller sidetabel
        printf("FEJL: Illegal address fault!\n");
        return 0;
    }

    if(protection & 1) {    // Der må læses fra filen
        return physical;
    }
    else {                  // Der må ikke læses fra filen
        printf("FEJL: Protection fault!\n");
        return 0;
    }
}

/*
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) {
    uint32_t physical;
    uint16_t protection;

    protection = tlb_lookup(logical, &physical);

    if(protection == 0) {       // Adressen findes ikke i hverken TLB eller sidetabel
        printf("FEJL: Illegal address fault!\n");
        return 0;
    }

    if((protection >> 1) & 1) { // Der må skrives til filen
        return physical;
    }
    else {                      // Der må ikke skrives til filen
```

```
        printf("FEJL: Protection fault!\n");
        return 0;
    }
}

/*
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 prøver den igen.
*/
int tlb_fault(uint32_t logical) {
    page_entry_t *pe;

    pe = pagetable_get(logical);    // Forsøg at hente siden fra sidetabellen

    if(pe == NULL) {                // Adressen eksisterer ikke; stop TLB søgning
        return 0;
    }
    else {                          // Indsæt siden i TLB'en og gennemsøg denne igen
        tlb_insert(logical, pe->physical, pe->protection);
        return 1;
    }
}
```

Bilag 4: mem.h (uændret)

```
#ifndef _MEMORY_H
#define _MEMORY_H

#include "inttypes.h"

#define PAGE_SIZE 4096
#define PAGE_MASK 0xFFFFF000

/* Tilgangsrettigheder */
#define MEM_READ      0x1
#define MEM_WRITE     0x2

uint32_t mem_read(uint32_t logical);
uint32_t mem_write(uint32_t logical);

#endif
```

Bilag 5: pagetable.c (uændret)

```
#include <stdlib.h>
#include <inttypes.h>

#include "pagetable.h"

#define LEVEL_SIZE 1024
#define LEVEL_MASK 0x3FF
#define LEVEL1_BIT 22
#define LEVEL2_BIT 12

typedef struct level {
    unsigned int count;
    void *childs[LEVEL_SIZE];
} level_t;

static level_t level1;

/* Initialiser et niveau i sidetabellen */
static void init_level(level_t *l) {
    int i=0;

    l->count = 0;

    for(i=0; i<LEVEL_SIZE; i++) {
        l->childs[i] = NULL;
    }
}

/* Initialiser sidetabellen */
void pagetable_init() {
    init_level(&level1);
}

/*
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) {
    page_entry_t *pe;
    level_t *l2;
    uint32_t p1, p2;

    p1 = (logical >> LEVEL1_BIT) & LEVEL_MASK;

    /* Opslag på første niveau. */
    l2 = level1.childs[p1];

    /* Skal et nyt niveau allokeres.*/
    if (l2 == NULL) {
        l2 = level1.childs[p1] = malloc(sizeof(level_t));
```

```
        init_level(l2);
        level1.count++;
    }

    p2 = (logical >> LEVEL2_BIT) & LEVEL_MASK;

    /* Opslag på andet niveau. */
    pe = l2->childs[p2];

    /* Skal der allokeres et nyt sideelement. */
    if (pe == NULL) {
        pe = l2->childs[p2] = malloc(sizeof(page_entry_t));
        l2->count++;
    }

    *pe = *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) {
    level_t *l2;
    uint32_t p1, p2;

    p1 = (logical >> LEVEL1_BIT) & LEVEL_MASK;

    l2 = level1.childs[p1];

    if (l2 != NULL) {
        p2 = (logical >> LEVEL2_BIT) & LEVEL_MASK;
        return l2->childs[p2];
    }

    return NULL;
}
```

Bilag 6: pagetable.h (uændret)

```
#ifndef _PAGETABEL_H
#define _PAGETABEL_H

#include <inttypes.h>

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

void pagetable_init();
void pagetable_set(uint32_t logical, page_entry_t *pe);
```

```
page_entry_t *pagetable_get(uint32_t logical);

#endif
```

Bilag 7: tlb-test.c (ændret)

```
#include <stdio.h>

#include "tlb.h"
#include "mem.h"

int tlb_fault(uint32_t logical) {
    return 0;
}

void page_info(uint32_t logic) {
    uint32_t phys;
    uint16_t prot;

    prot = tlb_lookup(logic, &phys);
    if (prot) {
        printf("0x%x: physical=0x%x writeable=%d readable=%d.\n",
            logic,
            phys,
            (prot & MEM_WRITE) != 0,
            (prot & MEM_READ) != 0
        );
    } else {
        printf("0x%x: no mapping.\n", logic);
    }
}


int main() {

    tlb_init(); // Initialisér TLB'en

    tlb_insert(0x16000, 0x1000, MEM_READ | MEM_WRITE);
    tlb_insert(0x26000, 0x1010, MEM_READ);
    tlb_insert(0x36000, 0x1020, MEM_READ | MEM_WRITE);
    page_info(0x16000); // Tjek, om tlb_insert() fungerer korrekt
    tlb_insert(0x46000, 0x1030, MEM_READ);
    tlb_insert(0x56000, 0x1040, MEM_READ | MEM_WRITE);
    tlb_insert(0x66000, 0x1050, MEM_READ);
    page_info(0x26000); // Tjek, om tlb_insert() fungerer korrekt
    tlb_insert(0x76000, 0x1060, MEM_READ | MEM_WRITE);
    tlb_insert(0x86000, 0x1070, MEM_READ);
    page_info(0x36000); // Tjek, om tlb_insert() fungerer korrekt
```

```
    /*
    TLB'en er nu fuld, og alle entries har sat deres TLB_REFERENCED.
    Ved næste insert bør second chance løbe alle igennem,
    rydde TLB_REFERENCED og erstatte plads 0 med det nye element.
    */

    tlb_insert(0x96000, 0x1080, MEM_READ | MEM_WRITE);
    page_info(0x26000);
    page_info(0x36000);
    page_info(0x46000);
    page_info(0x66000);

    /*
    De 4 page_info()-kald ovenfor sætter TLB_REFERENCED til true
    på plads 1, 2, 3 og 5. Ved næste indsættelse skulle second-chance
    derfor gerne ramme plads 4.
    */

    tlb_insert(0x106000, 0x1090, MEM_READ);
    page_info(0x106000); // Tjek, om tlb_insert() fungerer korrekt

    page_info(0xFFF000); // Forsøger at tilgå logisk adresse, der ikke findes i TLB'en

    return 0;
}
```

Bilag 8: tlb.c (ændret)

```
#include <stdlib.h>
#include <inttypes.h>
#include <stdio.h>          // Inkluderes så vi kan bruge printf()

#include "tlb.h"
#include "mem.h"

#define TLB_SIZE 8

#define TLB_USED           0x1
#define TLB_REFERENCED     0x2

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;

static tlb_entry_t tlb[TLB_SIZE];
```

```
// Variabel, der bruges til at holde styr på second-chance algoritmen
int chance;

/* Initialiser TLB'en. */
void tlb_init() {
    int i;

    for(i=0; i<TLB_SIZE; i++) {
        tlb[i].flags      = 0;
    }

    chance = 0;
}

/*
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) {
    int i;

    /* Fjern offset. */
    logical &= PAGE_MASK;

    /* Led efter et tomt element. */
    for(i=0; i<TLB_SIZE; i++) {
        if (!(tlb[i].flags & TLB_USED)) {
            break;
        }
    }

    if (i == TLB_SIZE) {    // Der blev ikke fundet et tomt element.

        /*
        Find en kandidat til udskiftning vha. second-chance algoritmen.
        Algoritmen leder fra og med plads chance efter urefereret TLB entry
        og giver second chances til de allerede refererede entries, der
        passeres.
        */

        while(tlb[chance].flags & TLB_REFERENCED) {
            tlb[chance].flags = TLB_USED;
            chance = (chance + 1) % TLB_SIZE;   // Flyt pointeren til næste TLB entry
        }

        i = chance;
        printf("Offer fundet på plads %d\n", i);
        chance = (chance + 1) % TLB_SIZE;       // Flyt pointeren til næste TLB entry
    }

    tlb[i].logical    = logical;
```

```
    tlb[i].physical   = physical;
    tlb[i].protection = protection;
    tlb[i].flags      = TLB_USED | TLB_REFERENCED;

    printf("Indsat TLB entry, log: 0x%x, phys: 0x%x på plads %d\n", logical, physical, i);
}

/*
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) {
    int i;

    /* Fjern offset. */
    logical &= PAGE_MASK;

    do {
        /* Find siden */
        for(i=0; i<TLB_SIZE; i++) {
            if (tlb[i].logical == logical) {
                tlb[i].flags |= TLB_REFERENCED;
                *physical = tlb[i].physical;
                return tlb[i].protection;
            }
        }
    } while (tlb_fault(logical));
    return 0;
}
```

Bilag 9: tlb.h (uændret)

```
#ifndef _TLB_H
#define _TLB_H

#include <inttypes.h>

void tlb_init();
void tlb_insert(uint32_t logical, uint32_t physical, uint16_t protection);
uint16_t tlb_lookup(uint32_t logical, uint32_t *physical);

int tlb_fault();

#endif
```