Protokolldokument

Betriebssysteme Praktikum Nr. 3

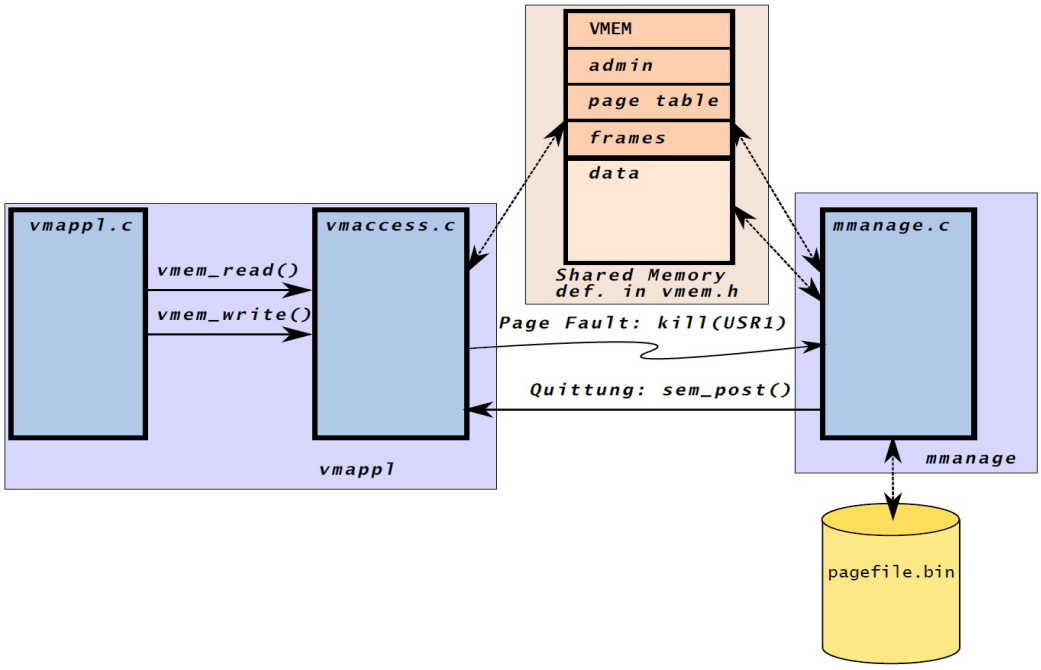
Gruppe 01

Ivan Morozov und Chandrakant Swaneet Kumar Sahoo

In der Aufgabe 3, implementieren wir eine virtuelle Speicherverwaltung.

Folgende Komponenten sind gegeben:

- „vmappl.c“ (Das Anwendungsprogramm)
- „vmaccess.c“ (stellt dem Anwendungsprogramm die Schreib- und Lesefunktionen zur Verfügung)
- „mmanage.c“ (verwaltet die virtuelle Speicherverwaltung)
- „*vmem.h*“ „*mmanage.h*“ „*vmaccess.h*“ „vmappl.h“ (Die jeweiligen Header files)



Wir haben folgenden Entwurf der Aufgabe. Mmanage soll zuerst gestartet werden und führt dann in seiner main-Prozedur initialisierende Funktionen aus. Danach kann vmappl gestartet werden welcher dann mit vm_init/vm_read Zugriffe auf den Speichern machen kann. Das mmanage ist mittlerweile bereit und wartet auf SIGUSR1, SIGUSR2 und SIGINT. Mmanage kümmrt sich um die Pagefaults und den Schreiben in/Lesen vom pagefile.bin und vmaccess.c übernimmt für vmappl.c den Zugrif auf den vmem.

Vmaccess.c:

Die vmaccess.c Datei enthält:

```
/* Connect to shared memory (key from vmem.h) */
void vm_init(void);

/* Read from "virtual" address */
int vmem_read(int address);

/* Write data to "virtual" address */
void vmem_write(int address, int data);

// refactores functions for writing and eading into vmem->data
void write_page(int frame, int offset, int data);
int read_page(int frame, int offset);

// check if connected and maybe call vm_init();
void vm_init_if_not_ready();

void countUsed(int page);
int calcIndexFromPageOffset(int page, int offset);

// Misc. - for testing purposes
void dump();
```

In der vm_init-Prozedur vebinden wir uns mit den Virtuellen Speicher unter Verwendung der Bibliotheksfunktionen shm_open, ftruntace und mmap. Wir brechen beim Fehlschlagen einer der Initialisierungsfunktionen das Programm sofort ab.

In der vmem_read-Funktion wird zunächst zum vmem verbunden, falls dies noch nicht geschehen ist. Es wird dann der Page auf dem zuzugreifen ist in das req_page_no in der vmem-Struktur reingeschrieben. Aus der gegebenen Adresse wird der Page und das Offset berechnet und dann nachgeguckt, ob die Page bereist geladen ist. Falls die Page nicht geladen ist, wird der Memory Manager mittels SIGUSR1 benachrichigt und der Prozess wartet auf dem Semaphoren in der vmem. Danach ist die Page auf jedenfall im Speicher und wir lesen mit einer Unterprozedur read_page vom Speicher und geben es zurück.

Die read_page-Prozedur vermerkt den Zugriff auf die Page mit countUsed, berechnet mit calcIndexFromPageOffset den Index im vmem->data Array und greift darauf zu und liest.

Die countUsed-Funktion erhöht die Anzahl der gesetzten USEDBITS.

Die calcIndexFromPageOffset-Funktion ist die Umkehrung von Adress->(Page, Offset) und berechnet zu einem Page und den Offset den Index in vmem->data.

Die beiden write Funktionen sind analog zu den read-Funktionen definiert. Bei write_page ist außerdem das Flags das eine Änderung markiert gestetzt.

Das vm_init_if_not_ready prüft ob bereits eine Verbindung zum vmem besteht und ruft ggf. vm_init auf. Die dump-Prozedur wurde ab und zu von uns in vmappl.c verwendet um uns den Dump zeigen zu lassen. Diese Prozedur sendet das SIGUSR2 Signal an mmanage.

Mmanage.c:

Die main Prozedur der mmanage.c Datei initialisiert das pagefile, das logfile, die Signalhandler sowie das Virtual Memory und geht dann in den signal processing loop.

Die mmanage.c Datei enthält:

```
/* Prototypes */
void sighandler(int signo);

void vmem_init(void);

void fetch_page(int pt_idx);

void store_page(int pt_idx);

void update_pt(int frame);

void update_load(int frame);

void update_unload(int oldpage);

int find_remove_frame(void);

int use_algorithm(void);

int find_remove_fifo(void);

int find_remove_clock(void);

int find_remove_clock2(void);

void signal_proccessing_loop(void);

// checks if a SIGUSR1 was caught and calls page_fault()
void case_page_fault(void);

// opens pagefile and maybe fills
// it with random data for easier debugging
void init_pagefile(const char *pfname);

void open_logfile();

// destroy all data and structurs because
// the process is ending
void cleanup();
```

```
// returns whether all frames are already occupied
int frames_are_occupied();


// print debug statement that we noticed a
// signal and reset signal number
void noticed(char *msg);


void page_fault();


void rotate_alloc_idx();


void dump_vmem_structure();
```

void sighandler(int signo)

Diese Funktion wird dem sigact als signal-handler übergeben. Sie schreibt das empfangene Signal in die globale variable signal_number. Der Fall der Pagefaults wird bereits hier (mit case_page_fault) behandelt, da es sonst nichtdeterministisches Verhalten aufgrd. Von Race-Conditions gäbe.

void vmem_init(void)

Diese Funktion erzeugt mit shm_open, ftruncate und mmap den Virtuelllen Speicher, initialisiert den Semaphoren in der vmem Struktur. Außerdem werden hier alle einträge in der vmem-Struktur mit Default-Werten initialisiert.

void fetch_page(int pt_idx)

Gegeben einer pageNummer sucht sich diese Funktion die Framenummer in der diese Page zu speicheren ist. Mit der PageNummer wird auf die entsprechende Stelle im pagefile.bin gesprungen und gelesen. Der Inhalt wird in das dazugehörige Frame kopiert.

void store_page(int pt_idx)

Analog zu fetch_page wird der Inhalt des Pages (welches gerade im Speicher geladen ist) in die entsprechende Stelle im pagefile.bin kopiert. Dies passiert nur, wenn der Inhalt des Frames sich verändert hatte.

void update_pt(int frame)

Diese Funktion berechnet zu dem zu ersetztenden Frame die dazugehörige alte Page und setzt den Eintrag im Pagetable zurück(update_load). Der Eintrag des zu ladenden Pages wird mit update_load aktualisiert.

void update_load(int frame)

Diese Funktion nimmt sich den „req_page_no“ und schreibt in das fragepage und in das Pagetable, dass diese Seite nun den gegebenen Frame zugeordnet ist. Außerdem wird das PTF_PRESENT bit gesetzt.

void update_unload(int oldpage)

Löscht alle flags dieser Page und setzt die Frame wieder auf den Default-Wert.

int find_remove_frame(void)

Diese Funktion liefert, falls Frames frei sind(mit frames_are_occupied()) den Index eines freien Frames. Sonst bestimmt sie mit use_algorithm einen zu ersetztenden Frame.

int use_algorithm(void)

Delegiert auf einen konkreten Algorithmus.

int find_remove_fifo(void)

Bei jedem Aufruf wird das Frame auf dem nexT_alloc_idx zeigt zurückgegeben und der Zeiger wird weiterrotiert. Damit wird das Frame das zuerst reinkam auch zuerst rauskommen. Die Rotation des Index wird mit einer Unterprozedur rotate_allox_idx erledigt.

int find_remove_clock(void)

Diese Funktion rotiert mit rotate_alloc_idx stets weite rund setzt bei jedem gefundenen Frame das erste Usedbit zurück bis ein Frame gefunden wird, dessen erstes USEDBIt nicht gesetzt ist. Dieses Frame wird dann zurückgegeben.

int find_remove_clock2(void)

Diese Funktion ist analog wie Clock1 definiert. Nur wird bei jeder Iteration entweder das zweite oder das erste USEDBIT gelöscht. (Wobei das zweite USEDBIT jeweils zuerst gelöst wird.) Beim Fund eines Frames ohne usedbits wird dieses dann verwendet.

void signal_proccessing_loop(void)

Diese Funktion wartet in jeder Iteration auf ein Signal, liest das aufgetretene Signal(von signal_number), macht einen noticed darauf und delegiert auf die entsprechende Funktion. Im Fall eines SIGINT wird schließlich break aufgeruffen und der Prozess terminiert in der Main-Prozedur.

void noticed(char *msg)

Gibt einen Debug-Statement aus und setzt den signal_number wieder auf 0.

void case_page_fault(void)

Prüft ob das empfangene Signal das SIGUSR1 ist und ruft ggf. page_fault auf.

void init_pagefile(const char *pfname)

Öffnet die Pagefile und füllt sie mit random-Werten.

void open_logfile()

Augelagerte Unterprozedur die das logfile öffnet.

void cleanup()

Diese Unterprozedur schließt mit unmap, close und shm_unlink den Shared Memory und die beiden Dateien.

int frames_are_occupied()

Lies die size-Variable von der adm-Struktur und gibt zurpck, ob bereits alle Frames besetzt sind oder nicht.

void page_fault()

Die Pagefault Prozedur delegeiert im wesentlichen nur an die verschiedenenn unterprozeduren. Die Ablaufreihenfolge ist folgendermaßen:

- Pagefault-Counter in der adm-Stuktur hochzählen
- mittles find_remove_frame() den zu ersetztenden Frame finden
- den Index des zu entladennen Pages daraus berechnen und speichern
- die zu-unentladende Page speichern, falls find_remove_frame tatsächlich gebrauchten Frame zurückgegeben hatte.
- Das Pagetable aktualiseren.
- Die neue Page in das Frame laden.
- Loggerevent erstellen und logger aufrufen
- sem_post auf dem aufrufenden Prozess machen.

void rotate_alloc_idx()

Erhöht den Index um eins und verwendet die Modulo-Funktion um ggf. wieder bei 0 zu beginnen.

void dump_vmem_structure()

Gibt die administrativen Strukturen des vmem und die derzeit gepseicherten Inhaltee dort aus.

Makefile:

Wir haben beim Kompilieren öfters Hinweise von gcc erhalten. Wir ahben daher die lrt- und pthread flags hinzugefügt. Außerdem wurden wir beim includen bestimmter Bilbiontheken (z.B. ipc.h) auf Verwendung einer open source flagge hingewiesen. Wir haben etwas recherchiert und anschließend den Flag „-D_XOPEN_SOURCE=600“ hinzugefügt. Das std=99 erlaubt uns (wie auch letzte Aufgabe angeprochen) etwas Al-ler angelehnte Variablendeklarationen zu schreiben.

Vorgehensweise:

alias c="clear"

alias r="./run.sh"

alias c1="./cmpclock1.sh"

alias c2="./cmpclock2.sh"

alias fifo="./cmpfifo.sh"

alias ki="killall -9 mmanage"

Zum Starten - zwei tabs im bash:

1. Hier wird nur ./vmappl ausgeführt und evtl. mit c gecleart

2. Hier wird mit „r“ erneut gestartet und nach dem Beenden vom ./mmanage mit Ctrl+C mit „fifo“/„c1“/„c2“ der Unterschied ermittelt

Vorgehensweise:

Tab 2: mit r starten

Tab 1: mit ./vmappl ausführen

Tab 2: mit Ctrl + C beenden

Tab 2: vergleichen (fifo/c1/c2)

Im Quellcode stehen zu den Header-Files weitere Kommentare.