BS Praktikumsaufgabe 03

Virtuelle Speicherverwaltung

Version 0.1 - Vorstellung der Aufgabe am 29 November 2016

Alexander Mendel
Karl-Fabian Witte

erstellt am 21. Februar 2023

Der Mechanismus des virtuellen Pagings wird nachgebildet und mit den Algorithmen **FIFO**, **CLOCK** und **LRU** getestet. Dabei ist die Datei *pagefile.bin* der Festplattenersatz. Der physikalische Speicher wird als "Shared Memory" abgebildet. Anstelle von Interrupts werden Signale verwendet. Ein Quellcodegerüst wurde uns vorab zur Verfügung gestellt, welches einige Objekte im Vorhinein definiert.

Inhaltsverzeichnis

1 Entwurf

Im Großen und Ganzen werden die Funktionsköpfe und Definitionen, welche uns gegeben wurden, beibehalten und damit das Vorhaben realisiert. Nur kleine Änderungen haben wir uns erlaubt. Die Funktion `find_remove_frame` in `mmanage` wird zu einem Funktionspointer umgewandelt. Zudem wurde `next_alloc_idx` in der `vmem_adm_struct` in `last_alloc_idx` umbenannt.

1.1 Grundgerüst

Die Simulation besteht aus 2 Prozessen, die sich einen gemeinsamen Speicherbereich teilen. Es wird zunächst `mmanage` aufgerufen. Dieser Prozess legt den gemeinsamen Speicher fest und verwaltet. Zudem legt er auch den SWAP bzw. Auslagerung auf der Festplatte an und verwaltet diesen als einziger. Nachdem `mmanage` alles initialisiert hat, wartet er auf ein Signal, damit er unter anderem die page fault routine `allocate_page()` ausführen kann, worin er den nächsten Pageslot aussucht, ggf. die Daten der alten page ins `pagefile.bin` schreibt, den frame der benötigten page holt und die Daten als page in den gemeinsamen Speicherbereich schreibt. Danach wird der `vmappl` wieder frei gegeben.

`vmappl` sortiert Daten und interagiert mit dem gemeinsamen Speicher, der viel zu klein ist, um alle Daten zu halten. Wenn die gewünschte Seite zum Lesen bzw. Schreiben nicht im gemeinsamen Speicher liegt, haben wir einen *page fault* und ein Signal wird an `mmanage` geschickt und `vmappl` blockiert sich selbst.

1.2 von der page zum frame und zurück

Um von einem Frameindex (Framestartadresse / VMEM_PAGESIZE) zur entsprechenden Page zu kommen, sind die Anweisungen wie folgt:

```
1            int page = vmem->framepage[frame_idx];
```

Um von einer Pageindex die ensprechende Framestartadress zu erhalten, muss man den entsprechende Befehle ausfüren:

```
1            int frame_adr = vmem->pt.entries[page_idx].frame;
```

1.3 Schlüsselobjekte

1.3.1 Datei statt Festplatte

Anstelle des gesonderten Festplattenbereiches, auf der die auszulagernden Daten gespeichert werden, erstellen wir mit `fopen()` eine Dateien, aus der wir bei einem page fault die gewünschte Daten in den RAM-Ersatz lesen und ggf. (dirty) Daten vom RAM Ersatz in die Datei schreiben. Dabei wird die Datei mit sehr großen Zahlen initalisiert ($0 \leq x \leq RAND_MAX = 2^{32} - 1 \widehat{=} \mathcal{O}(10^{10})$). Da im Anwendungsprogramm `vmappl` nur Zahlen erzeugt werden, die kleiner sind als 1000, kann man so erkennen, bis wohin unser `mmanage` die neuen Daten auslagert. Wir verwenden ein festes Format, mit der wir die Daten in die Datei schreiben ("%10d" + Delimiter). Zwei Funktionen in `mmanage` greifen auf diese Datei direkt zu:

`fetch_page()`: greift lesend auf die Datei zu, indem der Positionszeiger der Datei mit `fseek()` gesetzt wird und mit `fscanf()` von dieser Position an gelesen wird.

`store_page()`: greift schreibend auf die Datei zu, indem der Positionszeiger der Datei mit `fseek()` gesetzt wird und mit `fprintf()` von dieser Position an beschrieben wird.

1.3.2 Shared Memory statt RAM

Der Arbeitsspeicher wird mit einem Prozess übergreifenden Speicherbereich (Shared Memory) simuliert. Wir verwenden die ältere System V Form. Dabei gehen wir wie folgt vor. `vmem_init` erstellt den gemeinsamen Speicher mit `shmget()`, wobei IPC_CREATE mit als Argument übergeben wird, mit dem der Speicherbereich alloziert wird. Mit der erhaltenden ID erhält man durch `shmat()` einen Zeiger auf den Bereich, welcher zu der Struktur aus `vmem.h` gecastet wird. In `vmmacces.c : vm_init()` funktioniert das ähnlich , jedoch wird hier der IPC_CREATE nicht übergeben. Damit beide Prozesse den selben Speicherbereich erhalten, wird `shmget()` der selbe key SHMNAME übergeben. Die Zerstörung des Shared Memory erfolgt in `clean_up()` in `mmanage`. Hier wird `shmdt` zum Lösen und danach `shmctl()` mit IPC_RMID zum Zerstören aufgerufen.

Der Inhalt der `vmem` Structur wird in `mmanage: vemm_init()` gefüllt. Es wird zuerst alles auf default Werte bzw. VOID_IDX gesetzt. Auch die Prozess ID von `mmanage` wird in dieser gespeichert, damit `vmappl` bei einem Seitenfehler ein Signal an `mmanage` schicken kann. Der Semaphor zum blocken von `vmappl` wird ebenfalls dort erstellt.

1.3.3 Signal statt Interrupts

Ein Betriebssystem operiert bei *page faults* mit Interrupts. Wir operieren hingegen mit Signalen. Bei einem *page fault* schickt der `vmappl` Prozess über `vmaccess` Funktionen bei einem *page fault* (PRESENT_FLAG ist nicht gesetzt -> Seite ist nicht im Speicher (vmem)) ein Signal (SIGUSR1) an den `mmanage` Prozess und blockiert sich selber. Dieser reagiert mit `sighandler()` und ruft die Funktion `allocate_page()` auf, welches die geforderte Seite in den Speicher lädt.

Mit dem Signal SIGINT wird `mmanage` nach dem Zerstören des *Shared Memory* durch `clean_up()` das Programm beendet. Ein weiteres Signal (SIGUSR2) ruft die Funktion `dump_pt()` auf, welches die Seitentabelle einmal auflistet. Das nutzt uns für das Debugging.

1.3.4 `vmem_read` und `vmem_write`

Zuerst wird überprüft, ob der Speicher schon bekannt ist und ggf. bekannt gemacht. `vmappl` ruft nämlich nicht direkt die Funktion `vm_init` auf. `vmappl` nimmt aus der quasie Bibliothek `vmaccess` die Funktionen `vmem_read(address)` und `vmem_write(address)`. Da `address` eine volle virtuelle Adresse ist, muss ein Seitenindex aus dieser errechnet werden. Dies erfolgt mit der Ganzzahldivision mit der Seitengröße des Speichers VMEM_PAGESIZE. Der Offset wird ähnlich wie der des Seitenindexes berechnet, jedoch mit Modulo statt Division.

Es wird geprüft, ob die Seite schon im Speicher liegt, wenn nicht wird ein Signal gesendet und blockiert (page fault). Nun ist die Seite zum Lesen bzw. Schreiben verfügbar. Beim Schreiben wird das DIRTY_FLAG gesetzt. Beim Lesen und Schreiben wird das USED_FLAG gesetzt (und für LRU wichtig, der jeweilige counter auf 0 gesetzt.)

Nachdem die Seite nun geladen ist, wird noch der Offset an den Seitenindex angehängt und zurückgegeben.

1.4 Algorithmen

Welche Seite ersetzt werden soll, entscheidet der Seitenersetzungsalgorithmus. Die Adresse der jeweiligen Seitenersetzungsfunktion wird bei der Algorithmusabfrage mit getopt einem Funktionszeiger übergeben. Für die `CLOCK` und `LRU` werden in der Schreib- und Lesefunktion von `vmaccess` eine Funktion `update_pt_algo` aufgerufen,

die entsprechend die Statusvariablen für diese Algorithmen berechnet. Außerdem wird immer von der Variable `last_alloc_idx` aus angefangen zu "zählen".

1.4.1 FIFO

Der FiFo ist der einfachste und kann mit einer Zeile Code realisiert werden. Dabei wird einfach der `last_alloc_idx` um einen erhöht und das Modulo mit der Rahmenanzahl VMEM_NFRAMES gebildet. Wie ein Ringbuffer.

1.4.2 CLOCK

Dieser ist dem FiFo relativ ähnlich. Dabei wird ein Zeiger `clock` wie der FiFo berechnet, jedoch wird mit der page, welche daraus resultiert, das `USED_FLAG` geprüft. Wenn es gesetzt ist, wird es gelöscht, solange bis der Zeiger auf eine Seite zeigt, die ein gelöschtes `USED_FLAG` besitzt und somit raus geworfen wird. Das `USED_FLAG` wird zudem bei jedem Lese- oder Schreibzugriff gesetzt.

1.4.3 LRU

Als pseudo Zeit wird ein page counter verwendet, der in jeder page erhöht wird, wenn ein Schreibe- oder Lesezugriff erfolgt. Der Zähler der benutzten Seite wird auf Null gesetzt. Der Algorithmus selber sucht alle Seiten einmal ab und speichert die Seite mit dem größten Zählerstand und gibt diesen zurück.

1.5 Sonstiges

Die Auswahl der Algorithmen ist mittels der `getopt_long` Funktion von GNU realisiert. Dabei sind kurze als auch lange Optionen erlaubt. In der Schleife für die Unterscheidung der Optionen wird zunächst auf die `vmem_algo` Variable ein entsprechender Wert für den Algorithmus gesetzt. Es wird höchstens nur eine Option erlaubt, sonst bricht das Programm ab. Bei keiner Option wird der FiFo ausgewählt. Nach der Getoptschleife wird anhand `vmem_algo` die entsprechende Funktion dem Funktionspointer `find_remove_frame` zugewiesen und eine entsprechende Debugnachricht ausgegeben.

2 How to compile - Makefile

Da wir zwei Programme übersetzten werden, werden die Linkeranweisungen aufgeteilt. Mit dem Befehl `make all` bzw. `make run_all` werden alle Programme erstellt bzw. ausgeführt. Beim Ausführen wird der `mmanage` Prozess mit dem Programm `pkill -INT mmanage`, nach dem `vmappl` terminiert,beendet. Dann werden das `logfile` und das `pagefile` entsprechende nach den Seitenersetzungsalgorithmen umbenannt. Somit haben wir ein Automatisierung und haben den Grundstein für automatische Tests entdeckt. (Mit `diff` der `logfiles` mit Referenzdatein, könnte man dies realisieren).

Die Headerabhängigkeiten werden diesmal mit eine `foreach` Schleife ausgeführt, da sich die Objektdatein und die Quellcodedatein in unterschiedlichen Verzeichnissen befnden. Der Gnucompiler erstellt mit `-M` leider nur die Targets mit Abhängigkeiten, ohne den Dateipfad zu berücksichtigen. Mit dem `-MQ` kann man das Target umbenennen und mit `-MM` wird die Abhängigkeit umbenannt. Das `Dependfile` wird mittels `-include` in das Makefile eingefügt. Das Minus hat den Effekt, dass Nichtvorhandensein der Datei führt zu keinem Fehler.