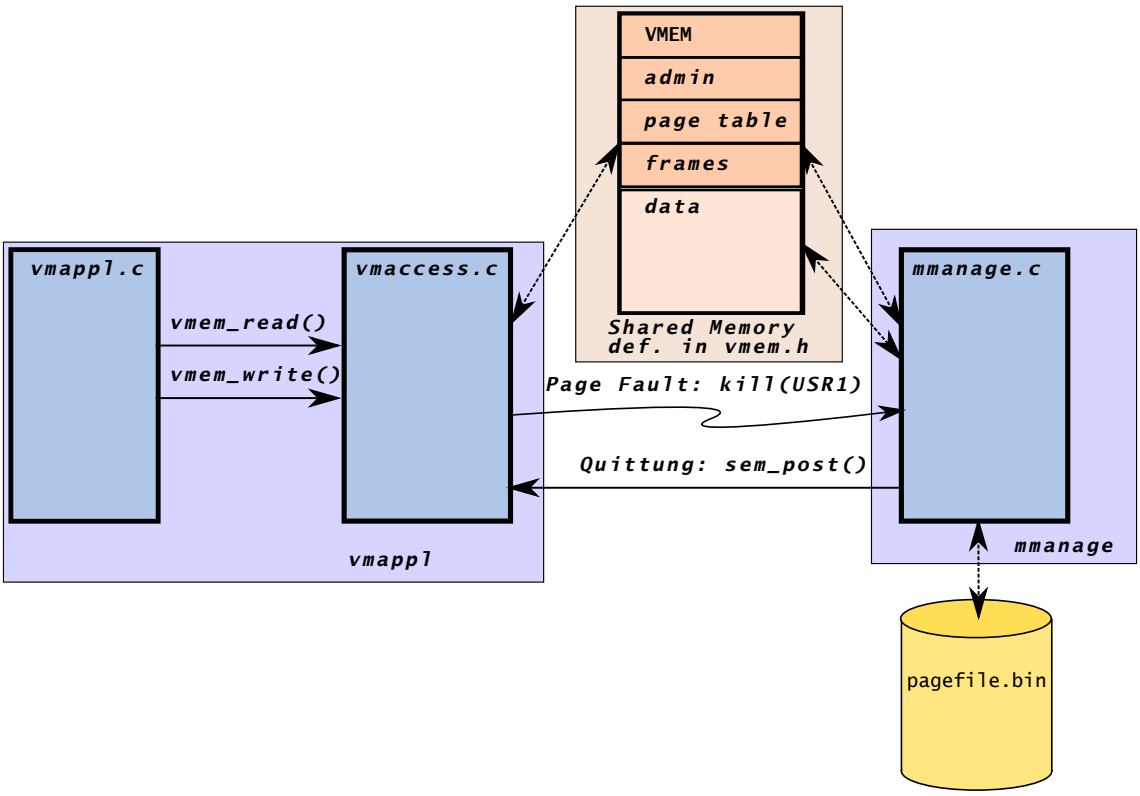
# 3 Virtuelle Speicherverwaltung

In dieser Laboraufgabe sollen Sie eine Anwendung schreiben, die Mechanismen der virtuellen Speicherverwaltung nachbildet, und mit der Sie die Algorithmen FIFO, LRU und CLOCK testen können. An die Stelle des physikalischen Speichers in einem realen System tritt hier ein Speicherbereich im *Shared Memory*, und anstatt Interrupts und ISRs verwenden Sie asynchrone Signale und Signalhandler.




Dies sind die Komponenten der Anwendung:

**vmappl.c** Das Anwendungsprogramm. Es werden zufällige Daten erzeugt, angezeigt, sortiert und nochmal angezeigt. Den Quellcode finden Sie in meinem Pub, daran dürfen Sie nichts ändern, damit die Anzahl der Seitenfehler Ihrer Speicherverwaltung identisch mit dem Resultat meiner Musterlösung ist.

**vmaccess.c** Die Schnittstelle zum virtuellen Speicher. Im realen System wäre das die Adress-Dekodierungseinheit (Umsetzung von virtuellen auf reale Adressen). Die Methoden `vm_read` und `vm_write` berechnen aus der virtuellen Speicheradresse die Frame-Nummer und den Offset. Wenn die benötigte Seite nicht geladen ist wird die Speicherverwaltung (`mmanage.c`) über ein asynchrones Signal mit `kill(pid, USR1)` aktiviert, die einen freien Frame sucht, ggf. einen Frame auslagert entsprechend den Algorithmen FIFO, LRU oder CLOCK, und die gewünschte Seite einliest. Die Routine aus `vmaccess.c` blockiert so lange mit `sem_wait()` auf einem Semaphor, bis die neue Seite geladen ist, und `mmanage.c` den Zugriff mit `sem_post()` wieder frei gibt.

**mmanage.c** Dies ist die Verwaltung der im Hauptspeicher vorhandenen Frames. Wie oben beschrieben, wartet das Programm mit `pause()` auf ein asynchrones Signal und sorgt für das Laden und Speichern von Pages aus der Datei pagefile.bin in den Hauptspeicher. Beim Start initialisiert `mmanage.c` den Speicher als shared memory, erzeugt den Semaphoren für die Koordination mit `vmaccess.c`, installiert mit `sigaction()` die Signalhandler und erzeugt falls nötig die Datei pagefile.bin und initialisiert die Datenstruktur.

Außerdem ruft `mmanage.c` bei jedem Seitenfehler die Methode `logger()` auf, die die Aktion im Logfile logfile.txt protokolliert. Sie finden einen Rumpf von `mmanage.c` in meinem Pub-Verzeichnis. Die Methode `logger()` dürfen Sie nicht ändern, da ich Ihr Logfile mit dem Resultat meiner Musterlösung mit dem Programm diff vergleichen werde. Im Idealfall sollten Ihr und mein Logfile identisch sein.

Beendet wird das Programm mit <Strg>-<C> (also über das Signal SIGINT). Beim Beenden muss das Shared Memory wieder freigegeben und der Semaphor gelöscht werden, sowie die zuvor geöffneten Dateien geschlossen werden. Also müssen Sie auch für `SIGINT` einen Signalhandler schreiben und installieren.

**vmem.h** Hier wird die Datenstruktur **`struct`** `vmem_struct` mit allen weiteren benötigten Strukturen und Konstanten definiert. Auch diese Datei finden Sie in meinem Pub. Unterstrukturen in **`struct`** `vmem_struct` sind

- Strukturen für die Verwaltungsdaten des Speichers **`struct`** `vmem_adm_struct`,
- die Seitentabelle **`struct`** `pt_struct`,
- sowie die Frame-Daten **`int`** `data[]`.

**Aufgabe:**

- Schreiben Sie die oben spezifizierte Anwendung, bestehend aus dem „Anwendungsprogramm“ `vmappl` und der Speicherverwaltung `mmanage`. Das Programm `vmappl` wird aus den beiden Quellfiles `vmappl.c` und `vmaccess.c` erzeugt.
- Implementieren Sie die Algorithmen FIFO, LRU und CLOCK entsprechend den Erläuterungen in der Vorlesung.
- Sie *müssen* auf alle Fehlerzustände beim Aufruf von Bibliotheksfunktionen reagieren. Das darf rustikal so erfolgen:

  ```
  perror("...hier Ihre Fehlermeldung...");
  exit(EXIT_FAILURE);
  ```

- Schreiben Sie ein Makefile, das mit dem Befehl `make` die beiden ausführbaren Dateien erzeugt. Hier können Sie auch über entsprechende Compiler-Flags die entsprechenden symbolischen Konstanten für die Auswahl des Algorithmus und zum Einschalten von Debug-Meldungen setzen. Für die Auswahl der Algorithmen wäre es allerdings vornehmer, sie `mmanage`

als Parameter mitgeben zu können. Zur Unterstützung gibt es die Bibliotheksfunktion `getopt`, der Befehl
`man 3 getopt`
hilft Ihnen weiter.

## Hinweise

Bis hier her *müssen* Sie den Aufgabentext lesen. Nun kommt der optionale Teil des Textes, dessen Lektüre Ihnen vermutlich ca. einen Tag Entwicklung und Fehlersuche spart.

### Quellcode

Folgende Dateien stelle ich Ihnen im Archiv `Aufgabe3_pub_w16.zip` zur Verfügung:

**vmem.h** Dies ist die vollständige Beschreibung der Datenstruktur für den „virtuellen Speicher“ sowie aller im Gesamtprojekt benötigten Konstanten. Einige Bemerkungen zu Komponenten von `vmem_adm_struct`:

**mmanage_pid** wird von `vmaccess` für den `kill`-Aufruf benötigt.

**sema** Damit der Semaphor zwischen Prozessen verwendet werden kann, muss er im Shared Memory liegen.

Den Array `framepage[]` benötigen Sie zur Vereinfachung des Seitenersetzungsalgorithmus, weil Sie ja entscheiden, welcher *Frame* ausgelagert wird, sie dann aber die *Page* in die Datei auslagern müssen. Um dabei nicht dauernd durch die Page Table iterieren zu müssen hier der Array, der die Zuordnung Frame → Page enthält.

**mmanage.h** Dieses Headerfile enthält neben der Definition der Datenstruktur für den Logger die Dateinamen des Logfiles und des Pagefiles sowie symbolische Konstanten für die Algorithmen FIFO, LRU und CLOCK. Die Konstante `VOID_IDX` dient zur Initialisierung von Indexwerten, um Fehler abfangen zu können.

Als Anregung für Sie habe ich die Funktionsprototypen meiner Musterlösung hier stehen lassen. Sie können natürlich eigene Namen und eigene Konzepte verwirklichen.

**mmanage.c** Diese Datei enthält den kompletten Code der `main()`-Routine mit der Installation der Signalhandler und die `logger()`-Funktion. Die aufgerufenen Funktionen müssen Sie natürlich selbst schreiben.

**vmaccess.h** Hier finden Sie unter anderem die Funktionsprototypen, die Sie in `vmaccess.c` implementieren müssen. Da das Anwendungsprogramm `vmappl.c` keinen Aufruf von `vm_init` enthält, müssen Sie zu Beginn von `vmem_read` und `vmem_write` prüfen, ob Sie sich schon mit dem Shared Memory verbunden haben.

**vmappl.c, vmappl.h** Das komplette Anwendungsprogramm. Hier müssen und dürfen Sie nichts ändern. Die Funktion `vmem_cleanup` sorgt dafür, dass Sie das Shared Memory wieder freigeben können.

## Tipps

**Debugging** Debuggen von nebenläufigen Anwendungen ist die Pest! (Darum sollen Sie es hier mal üben.) Man kann sein Programm auf folgende Weise mit Debug-Ausgaben spicken:

```
if(!vmem) {
    perror("Error initialising vmem");
    exit(EXIT_FAILURE);
}
else {
    PDEBUG(stderr, "vmem successfully created\n");
}
```

Das Makro `PDEBUG` finden Sie in `vmem.h`

Im Makefile kann man dann mit einer Zeile

```
CFLAGS  += -DDEBUG_MESSAGES
```

die Ausgabe der Debugmeldungen einschalten

**Verwendung des Debuggers** Sie können sowohl `mmanage` als auch `vmappl` unter dem Debugger `gdb` ausführen. Dazu folgende Hinweise:

- Damit der Debugger die Signale *INT*, *USR1* und *USR2* durchlässt, müssen Sie im Arbeitsverzeichnis eine Datei `.gdbinit` anlegen, die diesen Inhalt hat:

  ```
  handle SIGUSR1 nostop
  handle SIGUSR2 nostop
  handle SIGINT nostop
  ```

- Den Debugger `gdb` können Sie entweder auf der Kommandozeile aufrufen (Brrrrrr!!), vom Editor Emacs aus mit M-x gdb (Naja...), oder von Eclipse aus, wie gewohnt. Eine etwas schlankere IDE ist CodeLite, eine Alternative zu Eclipse wäre Netbeans. Beide IDEs bieten ein GDB-Frontend.

**Dump mit SIGUSR2** Wo Sie doch sowieso schon Signalhandler installieren, könnte es nützlich sein, als Reaktion auf `SIGUSR2` einen Dump der Struktur `vmem` auszugeben.

**Race Conditions** Sie müssen die Seitenallokation und das abschließende `sem_post()` komplett im Signalhandler abhandeln, sonst bekommen Sie merkwürdigste Race Conditions und sehen irgendwann weiße Mäuse.

**Initialisierung des Pagefile** Es könnte nützlich sein, das Pagefile mit Zufallszahlen zu initialisieren, damit man beim Dump sieht, ob sich überhaupt was getan hat. (Um eine reproduzierbare Initialisierung zu erreichen, habe ich die Konstante `SEED` in `mmanage.h` definiert, mit der ich zu Beginn des Hauptprogramms *einmal* die Funktion `srand()` aufrufe.)

**Prozessübergreifende Phreads-Semaphoren** Anders als der Name vermuten lässt, lassen sich die Semaphoren der pthreads-Bibliothek auch zur Koordination von *Prozessen* verwenden. Mit `info sem_init` erfahren Sie, wie es geht.

**Installation des Signalhandlers** Eines der in der Dokumentation sorgfältiger gehüteten Geheimnisse zur Initialisierung der für `sigaction()` benötigten Datenstruktur soll hier gelüftet werden:

```
sigact.sa_handler = sighandler;
sigemptyset(&sigact.sa_mask);
sigact.sa_flags = 0;
/* Now install the handlers for the desired signals */
if(sigaction(SIGUSR1, &sigact, NULL) == -1) { /* ... */
```

**Shared Memory** Es gibt zwei Möglichkeiten, Shared Memory einzurichten, die alte System-V-Methode mit den Funktionen `shmget` und `shmat` zum Erzeugen, und `shmctl` und `shmdt` zum Freigeben, oder Sie verwenden die modernere Version des Memory-Mappings mit einem etwas aufgeräumteren API. Hier erzeugen Sie das Shared Memory mit den Funktionen `shm_open`, `ftruncate` und `mmap`. Das Freigeben erfolgt bei dieser Variante über die Funktionen `munmap` und `close`.

**Hinweis:** Der mit `shm_open` usw. erzeugte Speicher ist *eigentlich* nicht persistent, wird also beim Neustart von `mmanage` neu angelegt und durch den Aufruf von `ftruncate(shm_id, VMEMSIZE)` mit Nullen initialisiert, **aber:** wenn bei der Entwicklung eine nicht sauber beendete Instanz von `mmanage` noch eine Referenz auf den Speicher hält, entfällt das Initialisieren mit Nullen. Daher am besten `ftruncate` zwei Mal aufrufen: das erste Mal mit der Größe 0, danach mit der tatsächlich gewünschten Größe.