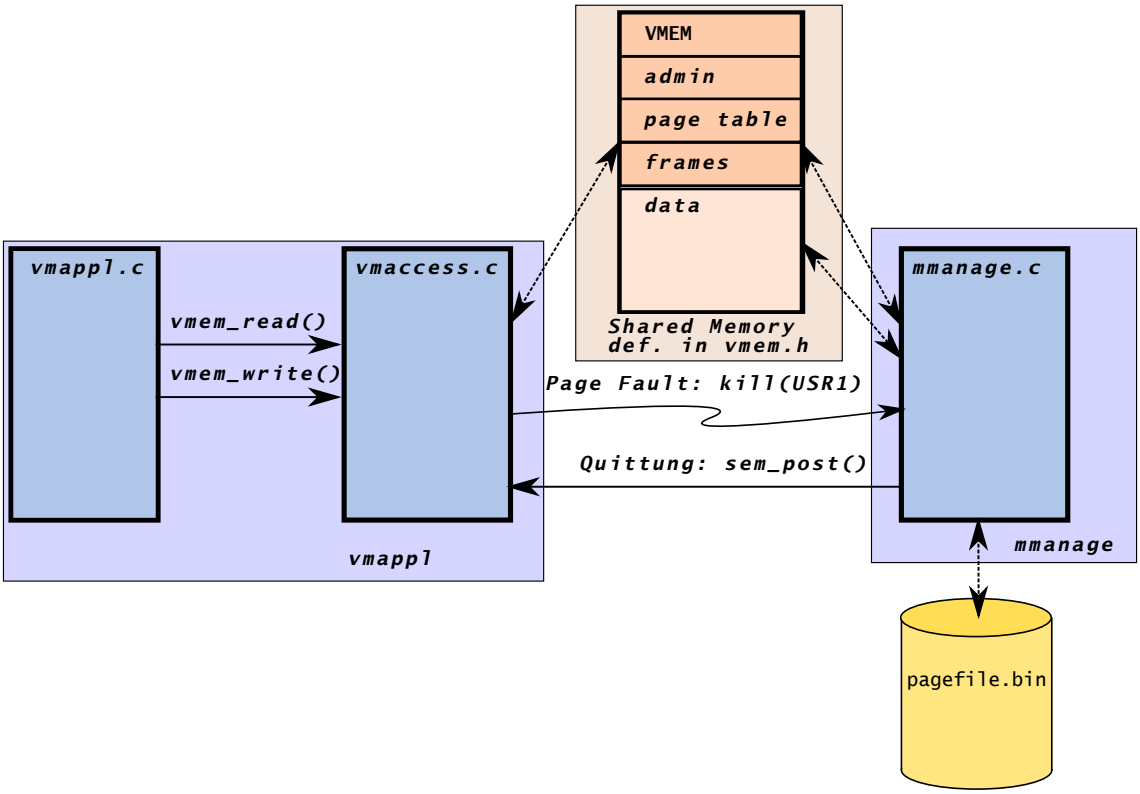
3 Virtuelle Speicherverwaltung

Diese Aufgabe bildet Mechanismen der virtuellen Speicherverwaltung nach. Auf dieser Basis werden die Algorithmen FIFO, LRU und CLOCK untersucht. An die Stelle des physikalischen Speichers eines realen Systems tritt hier ein Speicherbereich im *Shared Memory*. Interrupts und ISRs werden durch asynchrone Signale nachgebildet.



Dies sind die Komponenten der Anwendung:

vmappl.c Das Anwendungsprogramm. Es werden zufällige Daten erzeugt, angezeigt, sortiert und erneut angezeigt. Den Quellcode finden Sie in `Aufgabe3_s13_pub.tgz` unter EMIL. Ändern Sie vmappl.c nicht, damit die Anzahl der Seitenfehler Ihrer Speicherverwaltung identisch mit dem Resultat der Musterlösung ist. (Falls Sie schwere Probleme oder Fehler finden, können Sie natürlich den Code ändern.)

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.
Ist die benötigte Seite nicht geladen, wird die Speicherverwaltung (`mmanage.c`) über ein asynchrones Signal mit `kill(pid, USR1)` aktiviert. Sie sucht einen freien Seitenrahmen, lagert ggf. eine Seite entsprechend den Algorithmen FIFO, LRU oder CLOCK aus und liest die gewünschte Seite ein.
Die Routine aus `vmaccess.c` blockiert so lange mit `sem_wait()` auf einen Semaphor, bis die neue Seite geladen ist, und `mmanage.c` den Zugriff mit `sem_post()` frei gibt.

mmanage.c Dies ist die Verwaltung der im Hauptspeicher vorhandenen Seitenrahmen. Wie oben beschrieben, wartet das Programm mit `pause()` auf ein asynchrones Signal und sorgt für das Laden und Speichern von Seiten aus der Datei pagefile.bin in den Hauptspeicher.
Beim Start initialisiert `mmanage.c` den Speicher als shared memory, erzeugt den Semaphor für die Koordination mit `vmaccess.c` , installiert mit `sigaction()` die Signalhandler, erzeugt bei Bedarf die Datei pagefile.bin und initialisiert die Datenstrukturen.
Außerdem ruft `mmanage.c` bei jedem Seitenfehler die Methode `logger()` auf, die die Aktion im Logfile logfile.txt protokolliert (s. `Aufgabe3_s13_pub.tgz`). Die Methode `logger()` sollten Sie nicht ändern. So kann man Logfiles mit diff vergleichen.
Beendet wird das Programm mit <Strg>-<C> (also über das Signal SIGINT). Beim Beenden muss das Shared Memory freigegeben, der Semaphor gelöscht und die zuvor geöffneten Dateien geschlossen werden. Also müssen Sie für `SIGINT` einen Signalhandler schreiben und installieren.

vmem.h Hier wird die Datenstruktur **`struct`** `vmem_struct` mit allen weiteren benötigten Strukturen und Konstanten definiert (s. `Aufgabe3_s13_pub.tgz`). Unterstrukturen in **`struct`** `vmem_struct` sind

- Strukturen zur Verwaltung 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` . `vmappl` wird aus den Dateien `vmappl.c` und `vmaccess.c` erzeugt.
- Erzeugen Sie zu FIFO, LRU und CLOCK jeweils drei Varianten mit Seitengrößen von 8, 16, 32 und 64 Datenworten (**`int`**). Geben Sie die Anzahl der Seitenfehler in Abhängigkeit der Seitengröße tabellarisch aus.
- Sie *müssen* auf alle Fehlerzustände beim Aufruf von Bibliotheksfunktionen reagieren. Das darf rustikal erfolgen:

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

- Schreiben Sie ein Makefile zur Erzeugung von `vmappl` und `mmanage` . Wählen Sie über Compiler-Flags den FIFO, LRU oder CLOCK Algorithmus aus. Steuern Sie Debug-Meldungen ebenfalls über Compiler-Flags.

Hinweise

Quellcode Folgende Dateien stehen im Archiv `Aufgabe3_s13_pub.tgz` 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 Threads unterschiedlicher Prozess den Semaphor verwenden können, muss er im Shared Memory liegen.

g_count ist ein globaler Zähler, der bei jedem Speicherzugriff in den Routinen von `vmaccess` inkrementiert werden muss. Er dient als Zeitmaß für den LRU-Algorithmus. Ignorieren Sie die Tatsache, dass dieser Zähler überlaufen kann.

Das Array `framepage[]` benötigen Sie zur Vereinfachung des Seitenersetzungsalgorithmus. Es liefert die Zuordnung Frame → Page.

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 finden Sie die Funktionsprototypen der Musterlösung. 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.

Dump mit SIGUSR2 Da Sie 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 können Race Conditions auftreten.

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

Prozessübergreifende Semaphoren Die Semaphoren der pthread-Bibliothek unterstützen die Koordination von *Prozessen*. Mit `info sem_init` erfahren Sie, wie es geht.

Installation des Signalhandlers Initialisierung der für `sigaction()` benötigten Datenstruktur:

```
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) { /* ... */
```

Debugging Man kann sein Programm wie folgt um Debug-Ausgaben anreichern:

```
    if(!vmem) {
        perror("Error␣initialising␣vmem");
        exit(EXIT_FAILURE);
    }
#ifdef DEBUG_MESSAGES
    else {
        fprintf(stderr, "vmem␣successfully␣created\n");
    }
#endif /* DEBUG_MESSAGES */
```

Im Makefile kann man dann mit einer Zeile

```
CFLAGS  += -DDEBUG_MESSAGES
```

die Ausgabe der Debugmeldungen einschalten.