

Betriebssysteme Praktikum WS2015/16

Jan Lukas Deichmann, Veith Güntel
Betreuer: Prof. Dr. Ole Blaurock

January 25, 2016

Contents

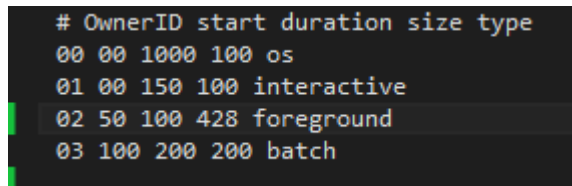
1	Präambel	1
2	Getting Started	2
3	Designreport	4
3.1	Getroffene Annahmen	4
3.2	Funktionsumfang	4
3.3	Betriebsbedingungen	4
3.4	Implementierung	4
3.4.1	processQueue	4
3.4.2	mList	5
4	Entwurfsentscheidungen	6
4.1	Auswahlstrategie für Speicherallozierung	6
4.2	Speichermanagement als verkettete Liste	6
4.3	ProcesseManagement als Queue	6
4.4	Kompaktierungszeitpunkt	6
5	Quellen	7

1 Präambel

In diesem Dokument wird die Lösung zu der gegebenen Projektaufgabe im Kurs Betriebssysteme WS2015/16 dargestellt. Bei dieser Aufgabe ging es darum, für eine gegebene Simulierung eines Betriebssystems, SimOS, eine geeignete Speicherverwaltung zu entwickeln.

2 Getting Started

Die Inbetriebnahme unserer Implementation erfolgt sehr simpel über die entsprechend Funktion von Visual Studio, sollte man sich zum Import in die IDE entscheiden, oder über die beigefügte .exe Datei. Es ist zu beachten, dass SimOS eine *processes.txt* Datei entweder im Projektordner für Visual Studio oder im selben Ordner wie die .exe erwartet. Diese Datei dient als Liste von Prozessen, die von SimOS einzulesen und auszuführen sind. Die *processes.txt* baut sich folgendermaßen auf: Man legt pro Zeile die Informationen zu einem Prozess fest, wobei die erste Zeile als Kommentar gewertet wird. Die Informationen müssen sich in der Reihenfolge OwnerID, start, duration, size, type in der Datei befinden. Eine beispielhafte Datei würde etwa so aussehen:

A screenshot of a text editor showing the content of a file named processes.txt. The file contains five lines of text. The first line is a comment starting with a hash symbol. The following four lines define processes with their respective OwnerID, start time, duration, size, and type.

```
# OwnerID start duration size type
00 00 1000 100 os
01 00 150 100 interactive
02 50 100 428 foreground
03 100 200 200 batch
```

Figure 1: processes.txt

Die entsprechende Datei würde folgende Ausgaben generieren:

```

Starting system. Available memory: 1024
Read from File: # OwnerID start duration size type
0 : Process info file opened
0 : System Initialised, starting batch
0 : No candidate read, reading next process from file
For Process 1 read from File: 0 0 1000 100 os
0 : PID 1 : Process loaded from file
5 : PID 1 : Used memory: 100 ! Process started and memory allocated
5 : No candidate read, reading next process from file
For Process 2 read from File: 1 0 150 100 interactive
5 : PID 2 : Process loaded from file
10 : PID 2 : Used memory: 200 ! Process started and memory allocated
10 : No candidate read, reading next process from file
For Process 3 read from File: 2 50 100 428 foreground
10 : PID 3 : Process loaded from file
10 : PID 3 : Used memory: 200 ! Process read but not yet ready
55 : PID 3 : Used memory: 628 ! Process started and memory allocated
55 : No candidate read, reading next process from file
For Process 4 read from File: 3 100 200 200 batch
55 : PID 4 : Process loaded from file
55 : PID 4 : Used memory: 628 ! Process read but not yet ready
105 : PID 4 : Used memory: 828 ! Process started and memory allocated
105 : No candidate read, reading next process from file
105 : No further process listed for execution.
445 : PID 3 : Used memory: 400 ! Process terminated, memory freed
445 : No candidate read, reading next process from file
445 : No further process listed for execution.
535 : PID 2 : Used memory: 300 ! Process terminated, memory freed
535 : No candidate read, reading next process from file
535 : No further process listed for execution.
705 : PID 4 : Used memory: 100 ! Process terminated, memory freed
705 : No candidate read, reading next process from file
705 : No further process listed for execution.
1470 : PID 1 : Used memory: 0 ! Process terminated, memory freed
1470 : Batch complete, shutting down

```

Figure 2: Output für processes.txt

3 Designreport

3.1 Getroffene Annahmen

In unserer Implementierung wird die Annahme getroffen, dass keinem Prozess jemals die PID 0 zugewiesen wird. Zudem wird angenommen, dass keine Prozesse gestartet werden, deren Speicherbedarf die Größe des Systemspeichers überschreitet.

3.2 Funktionsumfang

Wir erweiterten die gegebene Implementierung von SimOS im wesentlichen um Funktionen zur Verwaltung des Arbeitsspeichers (realisiert in `m_list.c`) und zur temporären Speicherung von Prozessen, die aktuell zu groß für den Arbeitsspeicher sind (realisiert in `process_queue.c`). Nähere Informationen zu `m_list` und `process_queue` befinden sich in der Erklärung zur Implementierung. Weitere Änderungen waren die Umlagerung von Variablen, um bessere Lokalität zu gewährleisten, und Änderungen, welche die Laufzeit optimieren.

3.3 Betriebsbedingungen

Die einzige Betriebsbedingung ist die in "Getting Started" beschriebene, korrekt formatierte `processes.txt` Datei.

3.4 Implementierung

3.4.1 processQueue

Die Process Queue die wir in `process_queue.c` realisiert haben ermöglicht es dem Betriebssystem Prozesse die aktuell, mangels ausreichenden Speichers, nicht ausgeführt werden können in einer Warteschlange zu speichern und später aufzuführen ohne dafür die Ausführung anderer Prozesse blockieren zu müssen. Die Process Queue basiert in ihrer Implementierung auf einer Listenstruktur und stellt die structs `processQueue` und `processQueueNode` zur Verfügung (siehe `m_list.h`). Desweiteren stellt die Schnittstelle folgende Funktionen zur Verwendung in SimOS zur Verfügung:

`processQueue* makeProcessQueue()` erzeugt eine neue `processQueue`, alloziert passenden Speicher und gibt einen Pointer auf die Queue zurück.

`processQueueNode* makeProcessQueueNode(PCB_t* pcb)` erzeugt eine neue `processQueueNode`, die als Inhalt den übergebenen Pointer auf eine `PCB_t` enthält, alloziert passenden Speicher und gibt einen Pointer auf die QueueNode zurück.

`void enqueue(processQueue* queue, PCB_t* process)` ermöglicht das einreihen eines Processes in die Queue. Es wird intern eine `processQueueNode` erzeugt und in die gegebene queue eingereiht.

PCB_t* dequeue(processQueue* queue) ermöglicht es den ersten Prozess aus der gegebenen Queue zu entfernen und gibt einen Pointer auf den Prozess zurück.

Boolean isEmpty(processQueue* queue) eine Hilfsfunktion die zurückgibt ob gegebene queue leer ist.

3.4.2 mList

Die Memory list ermöglicht die tatsächliche Speicherverwaltung. Sie basiert auf einer Listenstruktur und beinhaltet Blöcke die für die Segmente im Arbeitsspeicher stehen, die jeweils frei oder genutzt sein können. Zur Realisierung dieser Aufgabe stellt die Implementierung die structs mList und mListNode, sowie einen enum mType für die zustände der Blöcke bereit. Ausserdem wird durch den Aufbau des struct mListNode die Realisierung von Base- und Limitregistern gestellt. Desweiteren stellt die Schnittstelle folgende Funktionen zur Verwendung in SimOS zur Verfügung:

mList* makeMList() erzeugt eine neue MemoryList, alloziert passenden Speicher und gibt einen Pointer auf die Liste zurück.

mListNode* makeMListNode(mType type, unsigned pid, int start, int length) erzeugt eine neue mListNode aus gegebenen Parametern, alloziert passenden Speicher und gibt einen Pointer auf die Node zurück.

Boolean addProcess(mList* list, PCB* process) fügt den process der gefragten list hinzu.

void removeProcess(mList* list, PCB_t* process) entfernt den process aus der gefragten Liste.

void compact(mList* list) kompaktiert die gegebene Liste um der ihrer Fragmentierung entgegenzuwirken. Die Funktion wird automatisch aufgerufen sobald ein Prozess nicht in den Speicher passt.

mListNode* swapper(mListNode* a, mListNode* b) eine Hilfsmethode zum Austausch zweier Node innerhalb einer Liste, gibt einen Pointer zur neuen mListNode b zurück. Wird innerhalb von compact genutzt.

mListNode* merge(mListNode* a, mListNode* b) eine Hilfsmethode zum zusammenfügen zweier Nodes, gibt einen Pointer zur zusammengeführten mListNode zurück. Wird innerhalb von compact verwendet.

mListNode* findNextFit(mList* list, int length) eine Hilfsmethode zum finden der nächsten passenden Lücke innerhalb der Liste. Gibt einen Pointer auf die mListNode* VOR der passenden Lücke zurück.

4 Entwurfsentscheidungen

4.1 Auswahlstrategie für Speicherallozierung

Wir entschieden uns für **First-Fit**, da ohne genaues Wissen über Menge, Größe, und Dauer aller kommenden Prozesse nur sehr schwer zu sagen ist welche Auswahlstrategie tatsächlich zum besten Ergebnis führen wird. First-Fit gibt und allerdings den Vorteil sehr Effizient in der Suche, und leicht zu Implementieren zu sein.

4.2 Speichermanagement als verkettete Liste

Wir entschieden uns zur Implementierung der Speicherverwaltung in Form einer **Linkedlist**, da wir deutlich weniger des Speichers zur eigentlichen Verwaltung gegenüber einer BitMap aufwenden müssen und da die Suche ebenfalls deutlich effizienter zu gestalten ist.

4.3 ProzesseManagement als Queue

Wir entschieden uns dazu das Management von Prozessen die aktuell nicht in den Speicher passen als eine **queue** zu implementieren, da sie uns auf sehr simple art und weise ermöglicht Prozesse in der Reihenfolge in der sie ursprünglich aufgerufen wurden auch auszuführen. Dies verhindert das unter bestimmten bedingungen Prozesse niemals ausgeführt werden.

4.4 Kompaktierungszeitpunkt

Unsere Kompaktierung wird jedes mal eingeleitet, wenn ein Prozess **nicht in den aktuellen Speicher passt**. Wir halten dies für die effizienteste Lösung, da die aufwendige Kompaktierung nur dann ausgeführt wird wenn sie auch tatsächlich gebraucht wird.

5 Quellen

Es wurden keine Externen Quellen für unsere Implementierung verwendet.