Betriebssysteme Projekt 2024/25

Autor: Vinosan Kanagaratnam(367222),
Rakhim Khasukhanov(361464)
Projekt Betreuer: Prof. Dr. Ole Blaurock und Sophie Jent

January 20, 2025

Contents

1 Getting Started

Die Inbetriebnahme unserer Implementierung erfolgt einfach und schnell über Visual Studio oder die mitgelieferte `.exe`-Datei.

Voraussetzungen

Damit die Speicherverwaltung korrekt funktioniert, ist es erforderlich, eine Datei namens `processes.txt` bereitzustellen. Diese Datei muss entweder im Projektordner (bei Verwendung von Visual Studio) oder im gleichen Ordner wie die `.exe`-Datei liegen. Sie enthält eine Liste der Prozesse.

Aufbau der `processes.txt`

- Die **erste Zeile** wird als Kommentar betrachtet und enthält eine Beschreibung des Inhalts.
- Jede **weitere Zeile** repräsentiert einen Prozess mit den folgenden Informationen in genau dieser Reihenfolge:
  1. **OwnerID**: Identifikationsnummer des Prozesses.
  2. **start**: Startzeitpunkt des Prozesses.
  3. **duration**: Ausführungsdauer des Prozesses.
  4. **size**: Benötigter Speicherplatz des Prozesses.
  5. **type**: Art des Prozesses (z. B. `batch`, `interactive`).

Beispiel einer `processes.txt`

```
# OwnerID    start    duration    size    type
 00          0        5000        256     batch
 01          1000     3000        128     interactive
 02          2000     7000        512     batch
```

Schritte zur Inbetriebnahme

1. **Ausführung über Visual Studio**:
   - Öffnen Sie die Projektmappe in Visual Studio und gehen Sie auf "Starten ohne Debuggen".
2. **Ausführung über die `.exe`-Datei**:
   - Platzieren Sie die `.exe`-Datei und die `processes.txt` im gleichen Ordner.
   - Starten Sie die `.exe`-Datei in der Konsole mit dem Pfad zur `.exe`-Datei und der process.txt.

2 Designreport

2.1 Annahmen

Die Annahmen die man sich festgelegt hat ist, dass man ab der PID 1 (Prozess-ID 1) anfängt und aufwärts zählt, dass bedeutet dass man keine PID 0 hat, jeddoch wird ein Speicherblock mit der PID 0 als freier Speicherblock angesehen.

2.2 Entwurfsentscheidungen

2.2.1 Datenstrukturen

MemoryBlock-Liste : Für die Verwaltung des physischen Speichers wurde eine verkettete Liste von MemoryBlocks gewählt. Dabei speichert jeder Block seine Startaddresse, Grösse, Belegungsstatus und ggf. die dazugehörige Prozess-ID (PID). Diese Datenstruktur erlaubt es eine einfache Kompaktierung der Blöcke durchzuführen.

WaitQueue : Für blockierte Prozesse wurde eine verkettete Warteschlange implementiert. Dabei werden blockierte Prozesse mit ihrer PID, in die WaitingQueue vermerkt.

Mit den genannten Datenstrukturen kann man eine effiziente und simple Verwaltung des Speichers gerwährleisten.

2.2.2 Auswahlstrategie: passender Speicherblock

Für die Auswahlstrategie hat man sich für die First-Fit Strategie entschieden. Das heisst der erste freie Speicher Block soll belegt werden. Dabei Durchsucht man die Speicher Blöcke sequentiell von Anfang des Speichers bis zum Ende ab. Diese Strategie ist in `memory_manager.c` unter `findFreeBlock()` zu finden.

2.2.3 Kompaktierung

Die Kompaktierung wird in dieser Implementierung erst nur dann ausgeführt, wenn es genügend freien Gesamtspeicher für einen Prozess gibt, aber kein einzelner freier Block gross genug für den Prozess ist. Dieses Verhalten kann man in `memory_manager.c` unter `needsCompaction(unsigned requiredSize))` beobachten.

2.3 Funktionsumfang

Wir erweiterten die gegebene Implementierung um mehrere zentrale Funktionen, die sowohl die Verwaltung des Speichers als auch die effiziente Organisation von Prozessen betreffen.

Ein wesentlicher Schwerpunkt liegt auf der Verwaltung des Speichers, die in `memory_manager.c, wait_queue.c und core.c` realisiert wurde. Dabei wurde eine Speicherzuweisung entwickelt, die sicherstellt, dass neue Prozesse nur dann gestartet werden, wenn ausreichend Speicher verfügbar ist. Zusätzlich wurde eine Funktionalität zur Kompaktierung des Speichers implementiert, um fragmentierte Speicherbereiche zu kompaktieren und dadurch die Nutzung zusammenhängender freier Blöcke zu ermöglichen. Prozesse, die zu groß für den aktuellen freien Speicher sind, werden blockiert und in einer Warteschlange organisiert, bis genügend Speicher zur Verfügung steht.

Neben der grundlegenden Speicherverwaltung wurden Mechanismen zur Speicherfreigabe entwickelt, die nach dem Abschluss eines Prozesses automatisch den belegten Speicher freigeben. Dabei sorgt eine integrierte Funktion zur Zusammenführung angrenzender freier Speicherblöcke für eine verbesserte Ressourcenausnutzung.

Die Prozessorganisation wurde durch die Einführung einer Warteschlange optimiert, in der blockierte Prozesse gespeichert werden, bis sie gestartet werden können. Es wird regelmäßig geprüft, ob Prozesse aus der Warteschlange durch Speicherfreigabe oder Kompaktierung gestartet werden können, und sie werden automatisch in den Zustand `running` überführt.

Die genannten Erweiterungen und Optimierungen machen das System effizienter. Näheres zu den spezifischen Implementierungen und Funktionen kann in den Kommentaren der Quellcodedateien

`memory_manager.c, wait_queue.c und core.c`, sowie in den zugehörigen Header-Dateien nachgelesen werden.

2.4 Implementierung

2.4.1 Flussdiagramm

Das folgende Flussdiagramm zeigt den Ablauf der Speicherverwaltung:

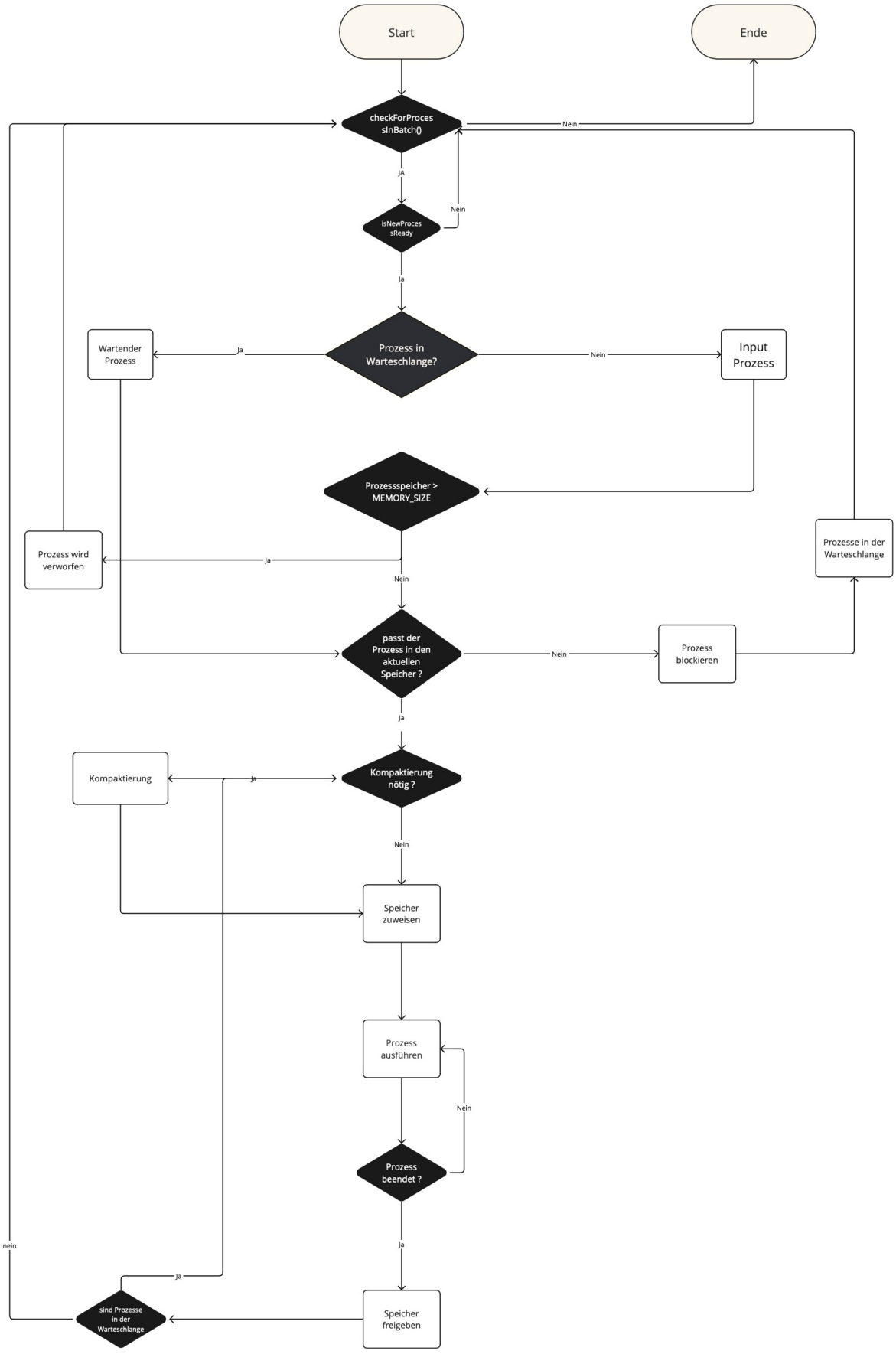

Figure 1: Flussdiagramm der Speicherverwaltung

Ablaufbeschreibung

Das Programm beginnt damit, dass geprüft wird, ob ein neuer Prozess aus einer Batch-Liste eingelesen werden kann. Diese Überprüfung erfolgt über die Funktion `checkForProcessInBatch()`. Ist ein neuer Prozess verfügbar, wird er eingelesen und zur weiteren Verarbeitung vorbereitet. Falls jedoch kein neuer und wartender Prozess verfügbar ist, endet das Programm.

Nach dem erfolgreichen Einlesen eines Prozesses wird überprüft, ob dieser bereit ist, ausgeführt zu werden. Diese Prüfung wird mit der Funktion `isNewProcessReady()` durchgeführt. Ist der Prozess noch nicht bereit, wartet das System, bis alle Voraussetzungen für seine Ausführung erfüllt sind. Sobald der Prozess die Anforderungen erfüllt, wird die Verarbeitung fortgesetzt.

Im nächsten Schritt wird geprüft, ob sich bereits blockierte Prozesse in der Warteschlange befinden. Wenn dies der Fall ist, wird der erste Prozess aus der Warteschlange bevorzugt behandelt und erneut auf seine Speicheranforderungen geprüft. Andernfalls wird der neu eingelesene Prozess weiter analysiert.

Ein zentraler Punkt im Ablauf ist die Überprüfung, ob die Speicheranforderungen des aktuellen Prozesses die verfügbare Gesamtspeichergröße überschreiten. Sollte ein Prozess mehr Speicher benötigen, als das System bereitstellen kann, wird dieser Prozess als nicht ausführbar markiert, blockiert und aus dem System entfernt.

Falls der Prozess grundsätzlich in den verfügbaren Speicher passen könnte, prüft das System, ob der aktuelle freie Speicher ausreicht. Ist genügend Speicher verfügbar, wird untersucht, ob eine Kompaktierung notwendig ist. Diese wird durchgeführt, wenn fragmentierter Speicher die Zuweisung des Prozesses verhindert. Nach der Kompaktierung oder wenn keine Fragmentierung vorliegt, wird dem Prozess Speicher zugewiesen, und der Prozess wechselt in den Zustand `running`. Reicht der Speicher jedoch nicht aus, wird der Prozess blockiert und in die Warteschlange verschoben.

Sobald einem Prozess Speicher zugewiesen wurde, beginnt dessen Ausführung. Während dieser Zeit prüft das System regelmäßig, ob der Prozess abgeschlossen ist. Ist dies der Fall, wird der belegte Speicher des Prozesses freigegeben, und blockierte Prozesse in der Warteschlange werden erneut geprüft, um festzustellen, ob sie nun ausgeführt werden können. Sollte der Prozess nicht abgeschlossen sein, bleibt er im Zustand `running`, bis er vollständig abgearbeitet ist.

Nach Abschluss eines Prozesses überprüft das System, ob blockierte Prozesse aus der Warteschlange gestartet werden können. Durch die Freigabe von Speicher könnte ausreichend Platz geschaffen worden sein, um diese Prozesse zu starten. Blockierte Prozesse werden priorisiert behandelt, um eine effiziente Nutzung der Speicherressourcen zu gewährleisten. Dabei sorgt das System durch geeignete Maßnahmen wie Kompaktierung dafür, dass die Speicherfragmentierung minimiert wird.

Dieser Ablauf wiederholt sich, bis alle Prozesse aus der Batch-Liste sowie der Warteschlange vollständig verarbeitet wurden. Sobald keine weiteren Prozesse mehr vorhanden sind, wird das Programm beendet.

2.4.2 MemoryBlock

initializeMemory() : Initialisiert den Speichermanager mit einem freien Block der Größe MEMORY_SIZE.

findFreeBlock(unsigned size) : Sucht nach First-Fit Strategie den ersten freien Speicherblock, der groß genug ist.

splitBlock(MemoryBlock_t* block, unsigned size) : Teilt einen zu großen freien Block in zwei Teile: einen der gewünschten Größe und einen Rest.

allocateMemory(PCB_t* process) : Weist einem Prozess Speicher zu, nutzt findFreeBlock und splitBlock.

deallocateMemory(PCB_t* process) : Gibt den Speicher eines Prozesses frei und führt ein Merging benachbarter freier Blöcke durch.

mergeAdjacentFreeBlocks() : Führt benachbarte freie Speicherblöcke zu größeren Blöcken zusammen.

needsCompaction(unsigned requiredSize) : Prüft ob eine Kompaktierung nötig ist, basierend auf der benötigten Prozessgröße.

compactMemory() : Führt eine Speicherkompaktierung durch, verschiebt belegte Blöcke an den Anfang.

printMemoryState() : Gibt den aktuellen Zustand der Speicherbelegung tabellarisch aus.

2.4.3 WaitQueue

initWaitQueue() : Initialisiert eine leere Warteschlange für blockierte Prozesse.

addToWaitQueue(pid_t pid) : Fügt einen Prozess am Ende der Warteschlange ein.

removeFromWaitQueue() : Entfernt den ersten Prozess aus der Warteschlange und gibt dessen PID zurück.

isInWaitQueue(pid_t pid) : Prüft ob ein Prozess mit der angegebenen PID in der Warteschlange ist.

tryStartWaitingProcesses() : Versucht wartende Prozesse zu starten wenn genügend Speicher verfügbar ist.

3 Quellen

Prof. Dr. Ole Blaurock, Vorlesung "Betriebssysteme" (Speicherverwaltung), Fachbereich Elektrotechnik und Informatik, Fachhochschule Lübeck, Stand: 02.12.2024, S. 18-21