Protokoll

Vorgelegt von: Daniel Bilic und Pedro Martin Moralejo

Aufgabe: Entwickeln und implementieren Sie einen Simulator zur Visualisierung verschiedener Realisierungskonzepte für das Speicherverwaltungskonzeptdynamische Partitionierung. Der Simulator soll auf Kommandozeilenebene die Realisierungskonzepte First Fit, Best Fit, Next Fit und Random für dynamische Partitionierung visualisieren.

Konzept: Um die Aufgabenstellung zu bewältigen, wurde zum einen der Main-Part verwendet und zum anderen Funktionen, die unterschiedliche Vorgänge in Gang setzen, um den reibungslosen Ablauf des Simulators zu gewährleisten. Sie unterstützen die Hauptanwendung durch verschiedene Berechnungen.

Der Main-Part übernimmt die Interaktion mit dem User und führt dann je nach Eingabe, eine der unten aufgeführten Funktionen aus.

Führt der User das Skript zu Beginn aus, findet er sich in einem Start-Screen wieder, in dem er aufgefordert wird, die gewünschte Speichergröße für die aktuelle Visualisierung zu wählen.

Nach der Eingabeaufforderung der gewünschten Speichergröße wird in der Methode check2expn() geprüft, ob es sich bei der Zahl um eine Zweierpotenz handelt. Wenn dies der Fall ist, setzt die Funktion die Variable $check auf 0. Die while-Schleife im Main-Teil wird nun unter entsprechender Fehlermeldung und erneuter Eingabeaufforderung, solange weiterlaufen bis die Variable check gleich 0 ist.

Wenn wir uns nun weiter im Main-Part bewegen, folgt die Deklaration von blockCounters=99 und lastBlock=0, sowie arr=memArr, mit dem ersten Wert im Index=0 von “1|00|$memory“ und arr=processArr. $memory entspricht dem Wert den der User als Speicher deklariert hat.

Dabei dient das Array memArr als Simulation der Partionen/Blöcke, die hintereinander im Speicher liegen und gibt an erster Stelle als Wert an, ob der entsprechende Block belegt (0) oder frei (1) ist, die Id des Blockes und die betreffende Größe in KB. Der initial erstellte, erste Wert entspricht dem gesamten Speicher im Ausgangszustand.

Das Array processArr hingegen repräsentiert die Prozesse, welche gerade im Speicher liegen. Ein Wert sieht dementsprechend so aus: “BlockId|Prozessname“. Ein Prozess ist mit seiner Prozessbezeichnung und dem entsprechend Verweis auf den von ihm belegten Speicherblock im Array hinterlegt.

Nun haben wir mithilfe einer select-Operation die Auswahl des Realisierungskonzeptes implementiert. In einer if-Anweisung wird nun der Variablen $concept der Wert der Variable $option zugewiesen.

Als nächstes wird dem Nutzer eine Liste mit möglichen Kommandos und deren Parametern ausgeben. Darauffolgend wird in eine while-Schleife gesprungen. Dort wird nun eine

Eingabeaufforderung gestartet und entsprechend der Variable $command in die entsprechende Funktion: createProcess $name $size, deleteProcess $name $size oder showInfo gesprungen. Bei der Eingabe von end wird das Skript mit dem Befehl exit beendet. Dabei wird auch eine Fehlerbehandlung implementiert, die auf die Richtigkeit von Kommandos und das Fehlen einer Eingabe achtet und entsprechende Fehlermeldungen ausgibt.

Die nachfolgend aufgeführten Funktionen unterstützen die vier Realisierungskonzepte bei Ihrer Umsetzung der Simulation.

Funktionen:

1. check2expn

   Nachdem der User eine Speichergröße eingegeben hat, wird geprüft, ob die Zahl eine 2-er Potenz ist.
   Dazu wird die Zahl in Binärer Schreibweise dargestellt und von links nach rechts durchlaufen und aufsummiert.
   Bei einer 2er-Potenz ist die erste Zahl immer eine 1 und die restlichen Stellen sind 0. Nach jeder Addition wird deswegen geprüft, ob das Ergebnis größer als 1 ist. Wenn das der Fall ist, muss der User erneut eine Eingabe tätigen.
   Diese Prozedur wird so oft wiederholt, bis der User eine Zahl eingibt, die den Anforderungen entspricht.

2. createProcess

   Diese Funktion springt in eine if-Anweisung. Je nachdem welches Realisierungskonzept ausgewählt wurde, wird eine der folgenden Funktionen gestartet (`randomFit()`, `nextFit()`, `firstFit()` oder `bestFit()`). Die Fehlerbehandlung greift, wenn ein Prozess erstellt wird, der schon existiert, wenn das zweite Argument fehlt oder wenn die Angabe der Speichergröße keine Zahl ist.

4. randomFit

   In dieser Funktion wird das Realisierungskonzept RandomFit realisiert. Dabei wird so oft willkürlich ein Block im Array memArr ausgewählt, bis dieser Frei ist und größer gleich der Prozessgröße ist. Wenn der Prozess einem Speicherblock zugeordnet wurde, wird die Variable $allocated auf 1 gesetzt und die Schleife wird verlassen.

5. nextFit

   In dieser Funktion wird das Realisierungskonzept NextFit durchgeführt. Zunächst, wird geprüft, ob in processArr, schon Prozesse vorhanden sind. Falls das der Fall ist, wird der Block an der Stelle $index angesprochen und die Speichergröße wird mit der

Prozessgröße verglichen. Sind sie gleichgroß wird der Prozess diesem Block zugewiesen. Falls der Block größer ist, entsteht ein neuer Block mit der Differenz der Speichergröße und der Prozess wird dem Block zugewiesen.

Tritt der Fall ein, dass processArr noch keine Prozesse beinhaltet, wird genauso vorgegangen, wie bei firstFit().

Nach erfolgreicher Allokation eines Prozesses, wird die Variable $found auf 1 gesetzt. Diese dient dazu zu prüfen, ob der Prozess abgelegt werden konnte. Falls das nicht der Fall ist, wird die Schleife von Null an erneut durchlaufen, um die übrigen nicht geprüften Speicherblöcke zu vergleichen.

6. firstFit

In firstFit() wird in einer for-Schleife das Array memArr durchsucht und der erste freie Block mit einem Speicher größergleich des zu erstellenden Prozesses ausgewählt. Nun wird die Funktion splitBlock() aufgerufen. Ihr wird die BlockId des ausgewählten Blocks, die Größe des Prozesses und die Id des neuen Blocks übergeben. Letztere wird durch das Dekrementieren von der Variable $BlockCounter ermittelt. In dem gleichen if-Block wird der Wert für den erstellten Prozess im ProcessArr hinzugefügt. Je nachdem, ob der Speicher des ausgewählten Blocks größer oder gleich der Prozessgröße ist, wird entweder der $BlockCounter als Zuweisung genommen oder die Id des ausgewählten Blocks, da in dem Fall der bestehende Block nicht aufgeteilt werden muss.

7. bestFit

In der Funktion bestFit() wird zu Beginn eine Variable diff=-1 deklariert, die die kleinstmöglich Differenz in der Speichergröße zwischen Speicherblock und zu erstellendem Prozess repräsentiert, wenn die Funktion durchgelaufen ist. Zunächst wird in einer for-Schleife der erste freie Block des Arrays memArr[] ermittelt und dessen Speicherdifferenz und BlockId in Variablen übernommen. Nun wird in einer zweiten for-Schleife nach freien Speicherblöcken mit einer kleineren Differenz gesucht und wenn ein solcher Speicherblock gefunden ist, werden dessen Werte als Variablen gesetzt. Zum Schluss wird die Variable $diff geprüft. Wenn diese größer als 0 ist, wurde ein entsprechend großer, freier Block gefunden, ansonsten würde eine Fehlermeldung ausgegeben werden. Wenn $diff gleich 0 ist, wird ein Eintrag im processArr mit der gleichen $BlockId des betreffenden Blockes als Referenz erstellt. Wenn $diff größer als 0 ist, wird eine neue $BlockId durch Dekrementieren des $BlockCounter an den Eintrag im Prozess-Array übergeben. So und so wird bei $diff= 0 die Funktion splitBlock() und showMemoryUsage() aufgerufen.

8. putTogetherFreeBlocks

Die putTogetherFreeBlocks() sorgt dafür, dass bei Löschen eines Prozess geprüft wird, ob vor oder hinter frei gewordenen Speicherblock ebenfalls ein freier Block liegt und verbindet diese in dem Fall zu einem Block. Der Funktion wird dabei nur die BlockId des gelöschten Prozesses übergeben. In der Funktion selbst werden zu Beginn die Variablen zaehler1, zaehler2 und sum alle mit 0 deklariert. Nun wird das Array memArr durchlaufen und in mehreren if-Statements zuerst geprüft, ob die entsprechende BlockId gleich der übergebenen BlockId ist und ob der Speicherblock frei ist. Wenn ja wird geguckt, ob es sich um den ersten Eintrag im Array handelt und dieser auch frei ist, weil dann der freie Block nur „vor“ dem betreffenden Block sein kann. In dem Fall wird $zaehler2 auf $index+1 gesetzt und die Variable $sum mit der Summe der Speichergrößen der beiden betreffenden Blöcke überschrieben. Zudem wird ein Eintrag als freier Block mit übergebener BlockId und $sum an der Stelle $index erstellt (“1|$1|$sum“). Wenn der Block allerdings nicht an erster Stelle im Array ist, wird geprüft, ob sich „hinter“ betreffendem Block ein freier Block befindet. Wenn dies nicht der Fall ist, wird geguckt, ob der Block „vor“ dem betreffenden Block frei ist, wenn ja, wird das genauso wie im obigen, ersten Fall vorgegangen.

Wenn allerdings der Block „hinter“ dem betreffenden Block frei sein sollte, wird geprüft, ob der Block „vor“ betreffendem Block auch frei ist. Wenn ja, wird im Unterschied zu den beiden obigen Fällen, die Variable $zaehler1 angesprochen, die Summe aller drei Speicher-größen der Blöcke genommen und der Eintrag im Block-Array an der Stelle $index-1 gesetzt. Der vierte Fall wäre, dass sich nur „hinter“ dem betreffenden Block ein weiterer freier Block befinden würde. In dem Fall wird $zaehler2 wieder mit entsprechendem $Index überschrieben, der Eintrag jedoch an die Stelle $index-1 im Array geschrieben.

Immer noch in der for-Schleife, aber im unabhängig if-Statement wird in einem Fall geprüft, ob $zaehler1 nicht gleich 0 ist und kleiner als die Größe des Arrays – 2. Wenn dem so ist, wird das Array an der Stelle $zaehler1 mit dem Wert aus der Stelle $zaehler1+2 überschrieben und die Variable $zaehler1 um 1 erhöht. Der Unterschied zum zweiten Fall besteht darin, dass dort die Variable $zaehler2 abgefragt wird und nicht immer die zwei, sondern die 1 von der Variablen abgezogen bzw. addiert wird. Das kommt daher, dass sich bei den Fällen, wo der $zaehler2 angesprochen wurde, immer nur ein freier Block vor dem initialen freien Block befindet und der $zaehler2 schon um 1 erhöht würde, dementsprechend muss man ihn nur noch um 1 erhöhen, um so die den betreffenden Block mit den Werten aus dem Block zwei Stellen über ihm zu überschreiben.

Zu guter Letzt werden die letzten beiden Einträge aus dem Array entfernt, wenn $zaehler1 nicht 0 ist und nur der letzte Eintrag, wenn $zaehler2 nicht 0 ist.

9. deleteProcess

Die Funktion deleteProcess() löscht einen Eintrag aus dem Array processArr[] und setzt entsprechenden Speicherblock auf frei. Dazu wurde wieder eine Variable counter5 = -1 eingeführt. Zunächst wird das Prozess-Array in einer for-Schleife durchlaufen. Wenn die Prozessbezeichnung dem Übergabeparameter entspricht, wird in den if-Block gesprungen und das Block-Array durchlaufen bis zu dem Eintrag, an dem der Verweis des entsprechenden Prozesses mit der BlockId im Block-Array übereinstimmt. Die Variable counter5 wird auf den Index des Prozesses im Prozess-Array gesetzt. Nun wird der entsprechende Block auf freigesetzt und ein echo-Kommando als Bestätigung ausgeführt.

Es wird die Funktion putTogetherBlocks() mit betreffender BlockId des gelöschten Prozesses als Übergabeparameter angesprochen. Es wird mit break sicherheitshalber aus der inneren for-Schleife gesprungen.

Nachfolgend wird in ein if-Block gesprungen, wenn $counter5 nicht gleich -1 ist und kleiner als die Prozess-Array-Länge -1 ist. Dort wird der Eintrag ab $counter5 des gelöschten Prozesses mit dem Eintrag der Stelle $counter5 + 1 des Prozess-Arrays überschrieben. Danach wird die Variable $counter5 um 1 inkrementiert. Dies geschieht so lange $counter5 kleiner als die Prozess-Array-Länge -1 ist, um zu verhindern, dass der letzte Eintrag mit null überschrieben wird.

Abschließend wird geprüft, ob $counter5 gleich -1 ist, wenn ja, wird eine Fehlermeldung ausgegeben, dass der angesprochene Prozess nicht existiert. Wenn $counter5 nicht gleich

-1 ist, wird geguckt, ob er kleiner als die Prozess-Array-Länge – 1 ist. Wenn dies der Fall ist, wird genau an dieser Stelle der Eintrag im Prozess-Array gelöscht. Wenn die Variable gleich groß ist, wird der Eintrag an der Stelle $counter5 gelöscht.

10. splitBlock

Die Funktion splitBlock(), wird angesprochen wenn ein Prozess erstellt wird und der entsprechende Speicherblock aufgeteilt werden soll. Als Parameter werden die Id des Blockes, der aufgeteilt werden soll, die Prozessgröße und die neue Prozess-Id übergeben. In der Funktion selbst wird zuerst die Variable counter=0 deklariert. Nun wird in der äußeren for-Schleife das Array memArr[] durchsucht. Wenn die die Id des Blockes, der des Übergabeparameters entspricht, wird in ein if-Block gesprungen. Dort wird geprüft, ob die Blockgröße größer, gleich oder kleiner der Prozessgröße ist. Wenn die Blockgröße kleiner ist, wird eine entsprechende Fehlermeldung ausgegeben. Bei gleicher Größe muss nur das Flag auf 0 gesetzt werden, um zu signalisieren, dass der entsprechende Speicherblock nun mit einem Prozess belegt ist. Sollte der Block allerdings größer der Prozessgröße sein, wird es etwas kniffliger, da

nun die Speicherdifferenz irgendwo untergebracht werden muss. Im if-Block wird der $counter gleich der Länge des memArr-Arrays gesetzt. Zunächst läuft erneut eine for-Schleife ab betreffenden Speicherblock das Array weiter ab und ersetzt solange die Einträge des Arrays beginnend ab Index $counter+1 mit denen des Index $counter, die Variable $counter wird jedes Mal um 1 erniedrigt. Dies geschieht, solange der $counter nicht gleich $index + 1 ist. $index ist hierbei die Stelle im äußeren Array, in die in die if-clauses gesprungen wurde, sprich der Speicherblock für den zu erstellenden Prozess. Das Array wurde somit um eine Stelle nach „hinten“ gerückt und der Eintrag nach dem betreffenden Prozessblock kann nun überschrieben werden. Im gleichen if-Block wird nun Speicherdifferenz aus Blockgröße – Prozessgröße ermittelt und zusammen mit der Block-Id des alten Blocks als ein neuer Eintrag in an die Stelle $index+1 als freien Block geschrieben. Der Speicherblock an der Stelle $index, wird nun mit einer neuen $BlockId als belegt überschrieben.

11. showMemoryUsage

Die Funktion showMemoryUsage() gibt alle Blöcke aus, die in dem Speicher-Array memArr vorhanden sind. Dabei werden die Objekte in memArr, mit den Objekten im Array processArr verglichen. Je nachdem, ob die Blöcke frei sind, oder belegt, erfolgt eine unterschiedliche Ausgabe. In der Konsole werden die Speicherblöcke untereinander angezeigt, jeweils mit dem Vermerk frei oder belegt, dem Namen des Prozesses und der Speichergröße, die zur Verfügung steht, oder vom Prozess beansprucht wird.

12. showInfo

Die Funktion showInfo() gibt Auskunft über den Grad der externen Fragmentierung, den größten freien Block, den kleinsten freien Block und die Anzahl der freien und belegten Blöcke.
Dafür werden in einer Schleife alle Blöcke durchlaufen und je nach dem, ob diese belegt sind, oder nicht werden entweder die Variablen $free oder $belegt um 1 erhöht.
Ist der Speicherblock frei, wird außerdem gesucht, ob er der größte oder kleinste freie Block ist. Dazu werden die Werte in die Variablen $gIndex gespeichert und in einer Iteration, solange überschrieben, bis kein größerer Speicherblock gefunden wurde.
Anschließend werden die Ergebnisse untereinander ausgegeben.

Fazit:

Nach zwei-maligen Treffen, um die Vorgehensweise zu planen und um die Umsetzung auf einen Nenner zu bringen, haben wir uns an die Arbeit gemacht.

Die Implementierung der Funktionen haben wir uns aufgeteilt. Um parallel arbeiten zu können, haben wir uns deshalb einen GitHub-Repository eingerichtet. An dieser Stelle haben wir leider etwas Zeit gespart bei dem Erlernen der Funktionen, die GitHub bietet.

So haben wir relativ schnell den selbst erstellten Code, durch den des anderen Teammitglieds überschrieben. Dieser Fehler hätte, durch eine entsprechende Einarbeitungszeit in GitHub, vermieden werden können.

Die Zusammenarbeit war ansonsten reibungslos. Wenn einer nicht weiterkam, hat der andere geholfen und falls keine Lösung parat war, zumindest die richtigen Denkanstöße mitgegeben, die den Lösungsprozess effizienter gestaltet haben. Auch die wöchentlichen Termine, um den Fortschritt zu beobachten und etwaige Fragen zu klären, wurden immer wahrgenommen und sehr effektiv durchgeführt.

Das Thema an sich hat uns keine Probleme bereitet. Lediglich der Umgang mit Bash war ungewohnt und hat an manchen Stellen für „Stirnrunzeln“ gesorgt. Aber auch hier konnten wir uns gegenseitig immer korrigieren und den einen oder anderen Fehler des anderen schnell beseitigen.

Die Dynamische Partitionierung fanden wir in der Umsetzung einfach. Die externe Fragmentierung, lässt uns allerdings daran zweifeln, ob die Umsetzung heutzutage sinnvoll wäre. Zumindest ohne die Speicherblöcke zu verschieben, um die nichtgenutzten kleinen Blöcke zu größeren zusammenzufügen, fanden wir sehr ineffektiv.