Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2022	**Praxisaufgabe 2** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	**24.07.2022 23:59 Uhr**

DOS OS

Sie arbeiten in dem kleinen Softwarestartup DOS (*Distributed Operating Systems*), dass sich auf die Entwicklung von Betriebssystemen spezialisiert hat. Vor kurzem hat Ihr Chef ein umfangreicheres Projekt gestartet, das zukünftig das Flaggschiff von DOS' Sortiment werden soll: Das *DOS OS*. Das *DOS OS* soll ein Nischenprodukt werden, das einen speziellen Anwendungsfall abdeckt: Es soll besondere Sicherheit gewähren, indem es ausschließlich auf dem RAM der ausführenden Maschine agiert. Das hat den Vorteil[1], dass keine Daten persistent gespeichert werden, was es deutlich schwieriger macht, sich unerlaubten Zugriff auf sensible Daten zu verschaffen.

Im Rahmen der bevorstehenden Aufgabe hat sich ergeben, dass Sie sich um einen Allokator , die Memory-Management-Unit (MMU) und um die Bedienung des Kaffeeautomaten kümmern.

Aufgabe 2.1: Allokator (20P)

Ein Allokator ist eine Struktur, die Speicherbereiche auf Anfrage reservieren und wieder freigeben kann (wie z.B. *malloc*). In dieser Teilaufgabe sollen Sie Ihren eigenen Allokator basierend auf dem Prinzip einer verketteten Liste implementieren. Der Allokator soll eine Funktionalität vergleichbar mit der von *calloc*[2] haben.

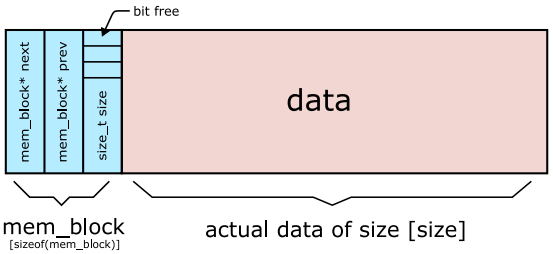

Abbildung 1: Ein einzelnes Element der verketteten Liste

Das Konzept funktioniert folgendermaßen: Jeder reservierte Bereich wird durch ein Element in der verketteten Liste repräsentiert. Dabei besteht ein solches Element aus zwei Teilen: Dem tatsächlich reservierten Speicherbereich und einer voranstehenden Verwaltungsstruktur (siehe *mem_block* in *calloc.h* und Abb. 1). Die verkettete Liste soll eine *doppelt verkettete Liste* sein. Eine solche kennzeichnet sich dadurch, dass ihre Elemente sich nicht nur einen

[1] ja, das ist tatsächlich ein Vorteil
[2] https://linux.die.net/man/3/calloc

Pointer auf das jeweils nächste, sondern auch auf das jeweils vorherige Element speichern. Dies erleichtert das Navigieren innerhalb der Liste und ermöglicht, die Liste in beide Richtungen abzulaufen.

Der *mem_block* enthält drei Parameter:

- **mem_block* next** ein Pointer auf das nächste Listenelement
- **mem_block* prev** ein Pointer auf das vorherige Listenelement
- **size_t size** gibt die Größe des zugehörigen Speicherbereiches ausschließlich der Größe des *mem_blocks* an. Dieser Parameter ist immer ein Vielfaches von 8 und muss entsprechend aufgerundet werden. Da **size** immer ein vielfaches von 8 ist, sind die drei least significant bits (LSBs) standardmäßig immer 0 und können daher anderweitig verwendet werden. In unserem Fall soll das LSB anzeigen, ob der entsprechende Speicherbereich frei (LSB = 0) oder belegt (LSB = 1) ist.

Um Speicherplatz zu reservieren, wird zunächst ein passendes Element in der Liste gesucht. Ein Element gilt als passend, wenn es nicht bereits belegt ist und seine Speicherkapazität mindestens genau so groß wie die (auf ein Vielfaches von 8 aufgerundete) angefragte Kapazität ist. Wurde ein passendes Element gefunden, so kann es für den Nutzer reserviert werden. Die folgenden Fälle können dabei auftreten:

a) **Die Kapazität des ausgewählten Elements entspricht exakt der Anforderung.** Dies ist der einfachste Fall. Hier muss lediglich der Status des *mem_blocks* auf "belegt"geändert und der pointer auf das Datensegment szurückgegeben werden.

b) **Die Kapazität des ausgewählten Elements (nennen wir es *old*) ist deutlich größer als die geforderte Speichermenge.** In diesem Fall muss das Element in zwei verschiedene Elemente aufgeteilt werden. Das erste Element soll anschließend für den Nutzer reserviert werden, das zweite, neue Element bleibt unbelegt. Für diesen Fall sind die folgenden Schritte nötig:
   - berechnen Sie, wie viel Speicherplatz von Element *old* übrig bleibt, nachdem die Anforderung des Nutzers erfüllt wurde. Der übrig gebliebene Speicher muss anschließend in einen neuen *mem_block* und seinen zugehörigen Speicherbereich unterteilt werden.
   - erstellen Sie einen neuen *mem_block* (nennen wir ihn *new*) hinter dem zu reservierenden Speicherbereich. *new* repräsetiert den übrig gebliebenen Speicherplatz von *old* nachdem der angeforderte Speicherplatz reserviert wurde).
   - übernehmen Sie den *next*-pointer von *old* in den neu erstellten *mem_block new*, tragen Sie die Größe des verbliebenen Speicherbereiches (abzüglich der Größe des neuen *mem_blocks*) ein und passen Sie den Belegungsstatus an
   - passen Sie die Parameter von *old* an: Berechnen Sie die Adresse des neuen *mem_block new* und tragen Sie sie zusammen mit dem aktualisierten Belegungsstatus und der neuen *size* ein. Legen Sie außerdem seinen *next*-pointer auf *new* um
   - passen Sie die *prev*-pointer von *new* und dem darauf folgenden Element an

c) **Die Kapazität des ausgewählten Elements ist nur geringfügig (weniger als *sizeof(mem_block)* zuzüglich der minimal reserviertbaren Kapazität von 8 Byte) größer als die angeforderte Speichermenge.** In diesem Fall wäre die übrig bleibende Speichermenge zu klein, als dass sie in ein neues Element umgewandelt werden könnte. Deshalb verhält sich dieser Fall genau wie Fall 1 (abgesehen davon, dass der reservierte Bereich geringfügig größer als die angeforderte Speichermenge ist).

Außerdem verfügt unser *calloc* über die Eigenschaft, frisch reservierten Speicher mit einem vom Benutzer gewählten Wert zu initialisieren. Dabei soll jedes Byte im reservierten Speicherbereich auf den gewählten Wert gesetzt werden. Achtung: *my_calloc()* bekommt für diesen Parameter eine Integer übergeben, von der aber nur das unterste Byte übernommen werden soll[3]. Für diese Anforderung ist die Benutzung der Funktion *memset()*[4] ratsam!

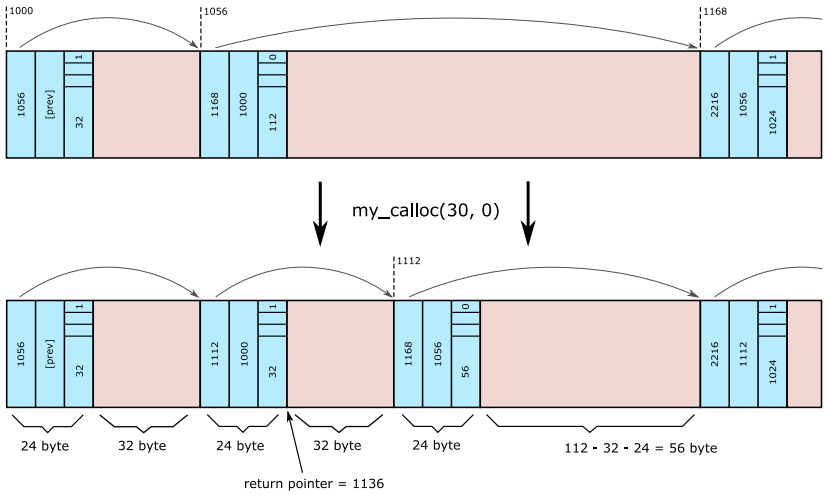

Abbildung 2: Reservieren eines neuen Elements

Neben der Allokation eines neuen Speicherbereiches muss der Allokator natürlich auch in der Lage sein, bereits reservierte Bereiche wieder freizugeben (siehe Abb. 3).

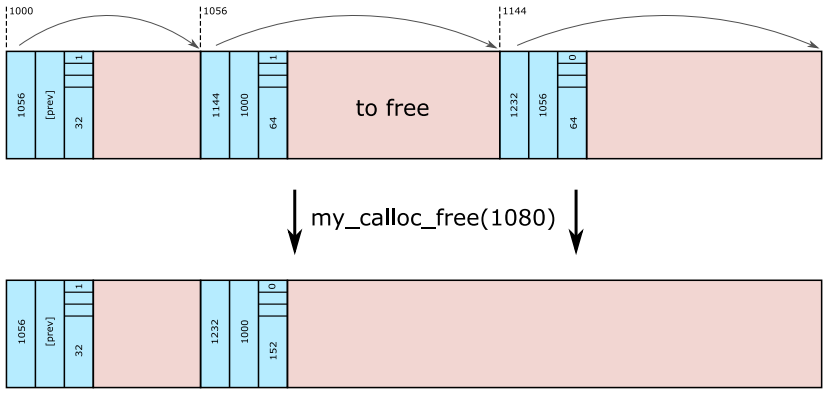

Abbildung 3: Freigabe eines Elements

Um einen Speicherbereich freizugeben, reicht es theoretisch aus, den Belegungsstatus des zugehörigen *mem_block* auf freißu setzen. Allerdings muss beachtet werden, dass zwei (oder mehr) nebeneinanderliegende, freie Blocke zu einem größeren Block zusammengefasst werden müssen. Folgende Fälle sind möglich:

- **Sowohl der Block davor als auch der Block dahinter sind belegt.** In diesem Fall reicht die Aktualisierung des Belegungstatus aus.
- **Entweder der Block davor oder der Block dahinter ist belegt.** Hier müssen die beiden betroffenen Blöcke miteinander verschmolzen werden:

[3]https://stackoverflow.com/questions/5919735/why-does-memset-take-an-int-instead-of-a-char
[4]https://linux.die.net/man/3/memset

  - die Kapazität des hinteren Blocks muss so angepasst werden, dass sie zusätzlich noch die Kapazität des vorderen Blocks und die Größe des vorderen *mem_blocks* umfasst (der hintere Block wird quasi um den vorderen Block erweitert).
  - der *next*-pointer des hinteren Blocks muss so angepasst werden, dass er auf das übernächste Element zeigt
  - der *prev*-pointer des übernächsten Elements muss so angepasst werden, dass er auf den hinteren Block zeigt
  - ggf. muss der Belegungsstatus des hinteren Blocks aktualisiert werden
- **Sowohl der Block davor (*current→prev*) als auch der Block dahinter (*current→next*) sind frei.** Gleiche Vorgehensweise wie bei Fall 2, allerdings müssen diesmal drei Blöcke miteinander verschmolzen werden:
  - die Kapazität von *current→prev* muss so angepasst werden, dass sie zusätzlich noch die Kapazität der beiden vorderen Blöcke und die Größen der vorderen *mem_blocks* umfasst
  - der *next*-pointer von *current→prev* muss so angepasst werden, dass er auf das über-übernächste Element zeigt
  - der *prev*-pointer des über-übernächsten Elements muss so angepasst werden, dass er auf *current→prev* zeigt

Um Ihnen den Einstieg in die Aufgabe zu erleichtern, wird Ihnen die Funktion *my_calloc_init()* vorgegeben. Die Funktion *my_calloc_init()* wird bei der Initialisierung des Systems aufgerufen. Dabei bekommt der Allokator den Speicherbereich gegeben, auf dem er arbeiten darf. Bevor er aber Reservierungen und Freigaben bearbeiten kann, muss er den gegebenen Speicherbereich in ein großes Listenelement umwandeln. Dies geschieht in *my_calloc_init()* (siehe Abb. 4). Die vorgegebene Implementierung dieser Funktion zeigt Ihnen auch, wie Sie an einer willkürlichen Adresse einen *mem_block* ablegen können. Stellen Sie sich dies so vor, als würden Sie eine *mem_block*-Schablone über den betreffenden Speicherbereich legen.

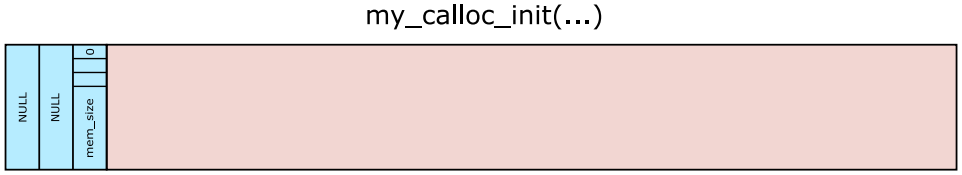

Abbildung 4: So sollte der gesamte Speicherbereich des Allokators nach *my_calloc_init()* aussehen.

Aufgabenstellung:

Als Belegungsstrategie für Ihren Allokator sollen Sie die *Next-Fit*-Strategie implementieren. Um diese erfolgreich zu Implementieren, muss sich Ihr Allokator merken, wo die letzte Reservierung stattgefunden hat (nennen wir dieses Element *last_allocation*) und von dort aus seine Suche nach passenden Elementen fortsetzen. Dabei soll *last_allocation* bei der nächsten Suche **erneut** überprüft werden. Dies birgt zwei Sonderfälle: Der erste Sonderfall tritt auf, wenn *last_allocation* freigegeben wird, während der vorherige Block bereits frei ist. In diesem Fall wird *last_allocation* mit seinem Vorgänger verschmolzen und *last_allocation* existiert nicht mehr. Umgehen Sie dieses Problem, indem Sie, sollte dieser Fall eintreten, *last_allocation* auf das vorherige Element umlegen.

Der zweite Sonderfall tritt auf, wenn *last_allocation* $\rightarrow$ *prev* freigegeben wird, während *last_allocation* bereits frei ist. Auch in diesem Fall wird *last_allocation* mit seinem Vorgänger verschmolzen und *last_allocation* existiert nicht mehr. Umgehen Sie auch dieses Problem, indem Sie, sollte dieser Fall eintreten, *last_allocation* auf das vorherige Element umlegen. Innerhalb eines gewählten Elements soll immer zuerst der ganz am Anfang liegende Speicher reserviert werden.

a) **my_calloc() (10P)**
Implementieren Sie in der Funktion *my_calloc()* einen Allokator nach dem oben beschriebenen Prinzip! Für weitere Informationen konsultieren Sie bitte *calloc.h*.

b) **my_calloc_free() (10P)**
Implementieren Sie die Funktion *my_calloc_free()* so, wie oben beschrieben! Für weitere Informationen konsultieren Sie bitte *calloc.h*.

Beachten Sie:

- da der für den Allokator nutzbare Speicherplatz nicht unbegrenzt ist, hat das letzte Element der List keinen Nachfolger, sein *next*-pointer hat den Wert NULL. Ebenso hat das erste Element der Liste keinen Vorgänger, sein *prev*-pointer hat auch den Wert NULL. Beachten Sie dies beim Traversieren Ihrer Liste.
- beachten Sie außerdem, dass sich der *size*-Parameter eines jeden *mem_block* ausschließlich auf die Kapazität des zugehörigen Speicherplatzes bezieht, nicht aber auf die Größe des *mem_block*-structs.
- in den oben gegebenen Beispielen wird davon ausgegangen, dass *sizeof(mem_block) = 24* gilt. Gehen Sie **nicht** davon aus, dass dies auch auf Ihrer Maschine der Fall ist. Benutzen Sie stattdessen immer den *sizeof()*-Operator.
- *my_calloc()*-Aufrufe müssen selbstverständlich den Pointer auf den reservierten Speicherplatz, nicht auf sich selbst, zurückgeben.
- analog zum oberen Punkt bekommen *my_calloc_free()*-Aufrufe nur den Pointer auf den freizugebenden Speicherplatz anstelle des Pointers auf seinen *mem_block*.
- der Allokator muss ein 8-Byte-Alignment einhalten. Das heißt, dass er nur Speicherbereiche reservieren darf, deren Größe ein Vielfaches von 8 sind. Beispiel: *my_calloc(19)* würde einen Pointer auf ein Speicherbereich mit einer Größe von 24 Byte zurückgeben.
- es dürfen keine nicht-nutzbaren Speicherbereiche entstehen. Damit ist folgendes gemeint: Dadurch, dass jedes Element einen Verwaltungsblock braucht, ergibt sich eine Mindestgröße für jedes Element (24 Byte Verwaltungsblock + 8 Byte Daten = 32 Byte). Sollte eine Allokation dafür sorgen, dass ein Speicherbereich entsteht, der kleiner als besagte Mindestgröße ist, so muss dieser kleine Speicherbereich dem reservierten Bereich hinzugefügt werden. Siehe dazu Fall 2 der möglichen Allokations-Fälle.

- sollte für eine *my_calloc()*-Anfrage nicht genügend Speicherplatz vorhanden sein, muss NULL zurückgegben werden.
- der Aufruf *my_calloc_free(NULL)* soll - wie *free(NULL)* auch - erlaubt sein, allerdings keine Wirkung haben.
- double-frees (zweimal *my_calloc_free()* auf den gleichen Pointer) müssen nicht extra abgefangen oder behandelt werden.
- achten Sie darauf, dass sowohl die Kapazität, als auch der Belegungsstatus in der selben Variable gespeichert werden. Ein unbedachtes Schreiben eben dieser könnte dazu führen, dass Informationen ungewollt überschrieben werden. Nutzen sie **unbedingt** Bitmasken, um auf die jeweilgen Teile/Bits der Variable zuzugreifen.
- sollte ein freizugebendes Element gleichzeitig das *last*-Element sein, vergessen Sie nicht, den *last*-Pointer umzulegen (falls nötig)

Aufgabe 2.2: Memory-Management-Unit (10P)

Warum extra Hardware nutzen? Mit diesem Gedanken wachte Ihr Chef eines Morgens auf und gab Ihnen den Auftrag, eine für die Übersetzung der virtuellen in physische Adressen zuständige Memory Management Unit (MMU) in Software zu schreiben.
Sie machen sich also ans Werk, einen minimalistischen Protoypen zu entwickeln. Dafür fehlen noch drei Teilfunktionalitäten.

a) **switch_process() (2P)**
Die Testumgebung soll 4kB Speicher für beispielsweise 32 Prozesse verwalten. Entsprechend wird pro Prozess eine Seitentabelle angelegt, die im gegebenen Beispiel mit einer Seiten- und Kachelgröße von 256 Byte also 16 Einträge für das Mapping von virtuellem auf physischen Speicher benötigt. Die Tabellen werden im physischen Speicher direkt an Adresse Null abgelegt (mem_start_addr). Damit die MMU, abhängig vom Kontext des gerade auszuführenden Prozesses, immer auf die korrekte Seitentabelle zugreift, gibt es das Seitentabellenbasisregister (Page Table Basis Register oder ptbr), welches die Basisadresse der aktuellen (kontext-spezifischen) Seitentabelle enthalten soll. Wenn ein Prozesswechsel auftritt, muss dieses Register entsprechend gesetzt werden. Implementieren Sie die Funktion *switch_process(int proc_id)*, welche das ptbr abhängig von der übergebenen process id setzt. Beachten Sie, dass die Adressen im Prototypen nur 12 Bit groß sind, aber in einem 16 Bit Format abgespeichert werden (siehe *mmu.h*). Prüfen Sie außerdem, ob eine gültige process id übergeben wurde.

b) **mmu_check_request() (2P)**
Wenn ein Speicherzugriff durch einen entsprechenden Prozess erfolgt, so wird dies der MMU mittels einer Anfrage (Request) mitgeteilt. Diese enthält die entsprechenden Informationen, anhand derer Sie die MMU korrekt initialisieren, um anschließend mit der Funktion mmu_translate die virtuelle Adresse des anfragenden Prozesses in eine

physische zu übersetzen. Geben Sie diese physische Adresse zurück. Bei Fehlern muss entsprechend *NULL* zurückgegeben werden.

c) **mmu_translate() (6P)**
Die Hauptaufgabe der MMU ist die Übersetzung einer virtuellen in die korrekte physische Adresse. Um diese Aufgabe korrekt auszuführen, muss kontrolliert werden, ob ein Prozess die nötigen Zugriffsrechte auf die entsprechende Seite hat. Hierfür enthält jeder Seitentabelleneintrag entsprechende Statusbits (READ, WRITE, EXECUTE) in den niederwertigsten Bits (siehe dazu auch die Dokumentation in der *mmu.h*). Um die Speicherverwaltung zu unterstützen, werden die 4 höchstwertigen Bits der Seitentabelleneinträge für die Informationsbits (Presence, Accessed, Modified) wie in der Vorlesung vorgestellt genutzt. Die Adressübersetzung an sich erfolgt genau so, wie im Tutorium besprochen (nur der Seitenindex wird übersetzt und anschließend mit dem Offset zusammengefügt).
Sorgen Sie dafür, dass die korrekte physische Adresse nur dann zurückgegeben wird, wenn der Prozess die nötigen Rechte besitzt und die Seite im Speicher vorhanden ist. Aktualisieren Sie dabei die entsprechenden Informationsbits.
Ist eine der Bedingungen für den Zugriff nicht gegeben, wird NULL zurückgegeben.

Beachten Sie:

- Schauen Sie sich unbedingt die Beschreibungen in *mmu.h* an! Diese Aufgabe ist auf Verständnis, nicht auf Fleiß ausgelegt.

Aufgabe 2.3: Kaffeeautomat (0P)

Bereiten Sie sich in der Küche eine Tasse Kaffee oder ein alternatives Getränk Ihrer Wahl zu.

Hinweise:

- Vorgaben: Bitte ändern Sie bestehende Datenstrukturen, Funktionsnamen, ... nicht. Eine Missachtung kann zu Punktabzug führen. Gerne können Sie weitere Hilfsfunktionen oder Datenstrukturen definieren, die Ihnen die Implementierung leichter gestalten. **Verändern Sie dafür nicht die header-Dateien!**
- Makefile: Bitte verwenden Sie für diese Aufgabe das Makefile aus der Vorgabe. Führen Sie hierzu im Projektordner *make* aus. Durch den Befehl *make clean* werden kompilierte Dateien gelöscht. Gerne können Sie das Makefile um weitere Flags erweitern oder anderweitig anpassen. Ihr Programm sollte aber mit dem vorgegebenen Makefile weiterhin kompilierbar bleiben. An dieser Stelle sei auf die README hingewiesen.
- **Abgabe:** Nutzen Sie zur Erstellung des Abgabeverzeichnisses **unbedingt** den Befehl *make pack*.