Paging in Multics

Jonas Rabenstein
FAU Erlangen-Nürnberg
Martensstraße 1
90158 Erlangen
jonas.rabenstein@studium.uni-erlangen.de

ABSTRACT

Mit dieser Seminararbeit soll ein Einblick in die Verwendung von Paging in Multics gegeben werden. Wenn auch die Idee der Speicherverwaltung durch Blöcke gleicher Größ nicht von Multics selbst stammt, war die Umsetzung durch ein enges Zusammenspiel mit Segmentierung ein interessanter Ansatz. Im Grundkonzept ist für Multics alles ein Segment und somit die Segmentierung ein zentrales Konzept. Da Segmente im Allgemeinen auch in Seiten eingeteilt wurden, war Paging ebenfalls entsprechend tief im System verwurzelt und hatte einen großen Einfluss auf die Effizienz des kompletten Systems. Nach einem Überblick der geschichtlichen Eckdaten, die zu dieser Art der Speicherverwaltung geführt haben, wird die Umsetzung von Multics genauer betrachtet. Dabei soll, neben Vorstellung der nötigen Datenstrukturen, vor allem auf die verwendten Algorithmen eingegangen werden.

1. EINLEITUNG

Mit einer Reihe von Ausarbeitungen auf der 1965 Fall Joint Computer Conference in Las Vegas, wurde Multics[1] das erste Mal der Öffentlichkeit präsentiert. Mit dem gemeinsamen Forschungsprojekt wollten das MIT, die Bell Labs sowie General Electronics ein System kreieren, dass zuverlässig eine großen Zahl an Nutzern bedienen kann und dessen Leistungsfähigkeit auch Ansprüchen der Zunkunft gewachsen ist [7]. Um dieses Ziel zu erreichen, wurden Ideen bestehender Betriebssysteme übernommen und mit eigenen erweitert. Dadurch ist Multics oftmals nicht die Quelle der umgesetzten Konzepte, doch durch deren Verbindung ein Gesamtpaket, mit Einfluss auf nachfolgende Systeme.

Eine dieser weitreichenden Innovationen, die bis heute in vielen Systemen vorzufinden ist, ist das Entkoppeln von logischem und physischen Adressraum. Durch das sogenannte Paging, bei dem der Speicher in Seiten gleicher Größe eingeteilt wird, gelingt die, für den Nutzer transparente, Zusammenführung von schnellen aber teuren mit billigeren, dafür aber langsameren Speicher zu koppeln. Damit lassen sich System umsetzen, die zuvor entweder wirtschaftlich nicht tragbar waren oder deren Effizienz aufgrund des Mangels an ausreichender Kapazität oder Geschwindigkeit des Speichers begrenzt war [9].

Neben der Erleichterung für den Nutzer, sich nicht mehr mit dem Transfer zwischen den beiden Speichern kümmern zu müssen, erlaubte diese Technik eine erhöhte Auslastung der Systeme. Zwar wurde auch zuvor versucht, durch Time-Sharing Betrieb, die Wartezeit eines Programmes durch die Ausführung eines anderen Prozesses sinnvoll zu nutzen, der begrenzte Speicher beschränkt jedoch auch hier die maximal mögliche Zahl parallel eingelagerter Anwendungen. Die vorgenommene Unterteilung durch Paging erlaubt es, Programme auszuführen, die nicht komplett im Speicher liegen. Es reicht, wenn die Seiten, mit den eventuell benötigten Anweisungen und Daten eingelagert sind [6].

[1]Multiplexed Information and Computing Service

2. HISTORISCHE ENTWICKLUNG

Bevor auf die Details der Implementierung von Paging unter Multics eingegangen wird, soll in diesem Abschnitt der historische Kontext aufgezeigt werden. Einerseits dient dies dem Verständnis, welche Probleme durch die Art der Speicherverwaltung angegangen werden sollten. Andererseits können Unterschiede in der Umsetzung des Konzeptes durch Multics im Vergleich zu anderen Systemen erkannt werden.

Durch die steigende Zahl an gleichzeitigen Nutzern, sowie die zunehmende Komplexität und Größe der Programme erforderte immer leistungsstärkere Rechensysteme. Einen limitierenden Faktor stellt dabei der Speicher dar. Ist die Kapazität nicht ausreichend, so können große Programme nicht ausgeführt oder die Zahl an simultanen Benutzern muss begrenzt werden. Aber auch bei ausreichendem Platz, wird die Performance beeinträchtigt, wenn der Speicher zu langsam ist und somit ständig auf diesen gewartet werden muss. Diese beiden Eigenschaften stoßen zum einen an technische Probleme und zum anderen an wirtschaftliche Grenzen. Häufig muss sich deshalb zwischen Fassungsvermögen und Geschwindigkeit entschieden werden [4].

Um diesem Dilemma zu entgehen, können mehrere, verschiedene Speicherelemente verbaut werden. Ein Beispiel für diese Vorgehensweise, ist der, erstmals 1958 an der Universität von Manchester in Betrieb genommene, Mercury der Firma Ferranti. Bei diesem System war ein Hauptspeicher von 40.000 Bits für den schnellen Zugriff verfügbar und ein Trommelspeicher mit 640.000 Bits. Ein Nachteil dieser Methode ist jedoch, dass die Komplexität, sich um den Transfer zwischen beiden Speichergeräten zu kümmern, bei dem Anwender liegt. Der Einsatz von seitenbasierter Speicherverwaltung nimmt dies dem Anwender ab. Darüber hinaus erlaubt es, von der tatsächlich verbauten Speicherkapazität der verschiedenen Speicher zu abstrahieren [9, Abschnitt 3].

Dies wird vor allem im Mehrbenutzer-Betrieb deutlich. Kann durch viel Aufwand der Speicherbedarf des eigenen Programmes und damit eine optimale Anordnung der Datentransfers vorgenommen werden, wird dies durch Unterbrechungen und Einlastung anderer Programme unvorhersehbar. In einem kleinen Experiment bei Einführung des At-

las Computers (siehe Abschnitt 2.2), wurden die Anwender des Systems gebeten, ein Programm zur Multiplikation von Matrizen zu schreiben. In allen Fällen wurden durch das automatische Austauschen der Seiten weniger Übertragungen benötigt, als bei den manuell vorgenommenen Programmen. Lediglich ein sehr stark optimiertes Programm konnte den verwendeten Algorithmus unterbieten [9].

2.1 Segmentierung

Durch Einteilung der Programme in logisch voneinander unabhängige Segmente, können deren Elemente relativ zur Basis adressiert werden. Somit ist die Position der Daten nicht mehr an eine feste Speicheradresse gebunden und das Segment kann beliebig umplatziert werden. Da diese Einteilung dem Betriebssystem bekannt ist, kann die Komplexität des Transfers in diesen verlagert werden.

Bei Verwendung dieser Methode, bestehen Zeiger aus zwei Teilen (vgl. Figure 3). Dabei ist der erste Teil die Segmentnummer, über die ein Speicherbereich ausgewählt wird. Jedes Segment hat dazu einen Deskriptor, in dem neben der Startadresse des Bereichs dessen Länge gespeichert ist. Nur wenn der zweite Teil der Adresse, der sogenannten Offset, kleiner der Segmentlänge ist, wird dieser auf den Segmentbeginn addiert und so die tatsächliche Speicheradresse gebildet. Andernfalls signalisiert die Hardware dem Betriebssystem einen Fehler, sodass eine angemessen Behandlung gestartet werden kann.

Der effiziente Einsatz dieser Methode hängt jedoch stark davon ab, dass die einzelnen Segmente im Vergleich zum Primärspeicher möglichst klein sind. Logischerweise ist die Wahrscheinlichkeit, ein referenziertes Segment im Hauptspeicher zu finden höher, je mehr Segmente sich in diesem befinden. [5, Abschnitt 3]

Neben dem Problem, die Segmente möglichst klein zu halten um eine hohe Trefferquote zu erzielen, erwies sich auch die Verwaltung des Speichers als relativ komplex. Grund dafür ist, dass Segmente weder eine einheitlich noch konstante Länge haben. Soll ein ausgelagertes Segment in den primären Speicher geladen werden, muss zuerst ein Segment[2], dass mindestens die selbe Länge hat, in den Sekundärspeicher verschoben werden. Der Platz des alten Segmentes kann dabei meist nicht verwendet werden, da die Längen nicht übereinstimmen (vgl. Figure 1). Es ist also anzunehmen, dass, ohne zusätzliche Gegenmaßnahmen, der Speicher fragmentiert und somit weniger Kapazität nutzbar ist [12].

2.2 Paging

Eine mögliche Lösung dieser Probleme wurde 1962 durch den Atlas Computer, an der Universität von Manchester mit dem „drum transfer learning program“ umgesetzt. Wie

[2]oder mehrere, im Speicher hintereinander liegende

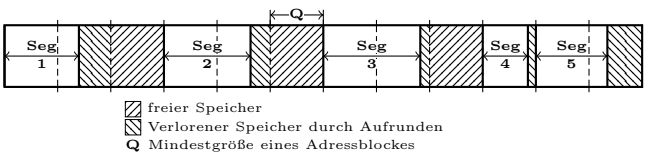


Figure 1: Verschwendung und Fragmentierung: Segmente $> Q$ können nicht mehr untergebracht werden. [12]

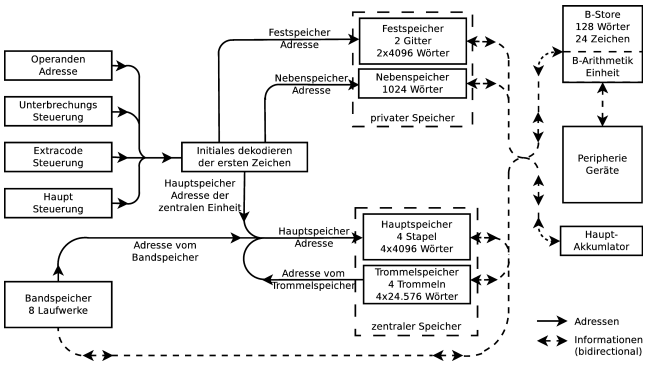


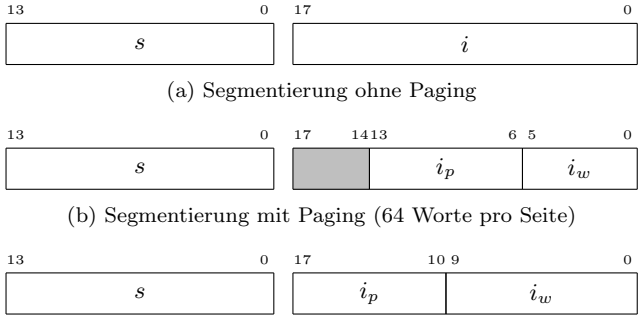
Figure 2: Aufbau des Atlas Computers: der zentrale Speicher erscheint dem Anwender als eine Einheit[9]

das Modell in Figure 2 zeigt, hatte die Maschine vier Speichereinheiten. Der „central store“ ist dabei Zusammengesetzt aus vier schnellen Hauptspeichern *(„main core store“)* mit einem Fassungsvermögen von jeweils 4096 Wörtern und ebenfalls vier Trommelspeichern mit jeweils 24.576 Wörtern. Aufgrund der Tatsache, dass der Trommelspeicher Daten in Blöcken von 512 Worten verarbeitet, wurde diese Blockgröße für den kompletten „central store“ verwendet und entsprechend der „main core store“ in Seiten mit jeweils 512 Worten eingeteilt [9, Abschnitt 2].

Ziel des entworfenen „drum transfer learning program“ war es, im Hintergrund dafür zu sorgen, dass zu jeder Zeit mindestens ein freier Seitenrahmen im Hauptspeicher zur Verfügung stand. Wird eine Seite angefragt, kann direkt mit dem Einlesen begonnen werden. Wäre kein Seitenrahmen frei, müsste das System erst eine Seite auswählen, die verdrängt werden soll, den Transfer der Seite in den Trommelspeicher abwarten, um dann mit dem Einlesen beginnen zu können. Während die angefragte Seite vom Trommelspeicher gelesen wurde[3], hat der Algorithmus eine zu verdrängende Seite bestimmt. Dieser war so konzipiert, dass die Zeit von $2\mu s$ ausreichend war und die Seite, die am längsten nicht benutzt wurde ausgewählt werden soll. Dazu wurde alle 1024 Instruktion überprüft, auf welche Seiten in diesem Zeitraum zugegriffen wurde[4]. Je öfter eine Seite dabei an einem Stück

[3]je nach Position der Leseköpfe zwischen 2 und $14\mu s$

[4]Wann immer eine Seite referenziert wurde, setzte die Hardware ein entsprechendes Bit in deren Deskriptor *(used bit)*

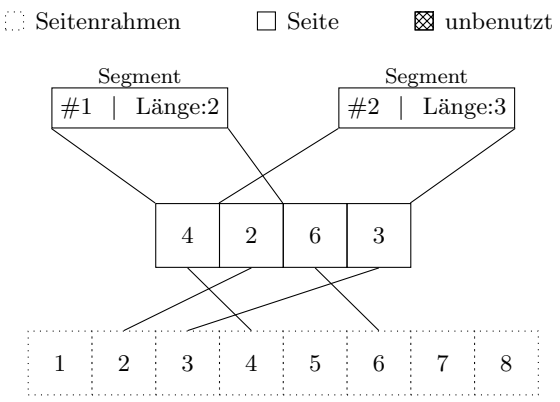


(a) Segmentierung ohne Paging

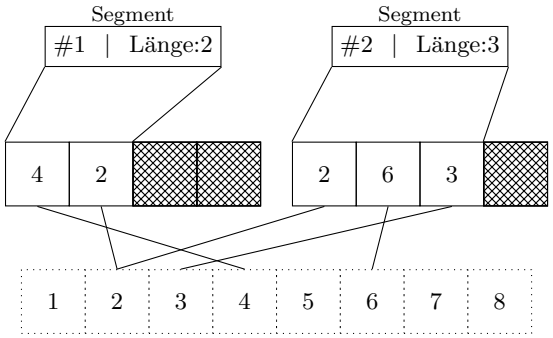
(b) Segmentierung mit Paging (64 Worte pro Seite)

(c) Segmentierung mit Paging (1024 Worte pro Seite)

Figure 3: Adressformate im Überblick



(a) globales (einstufiges) Paging (x86)



(b) Segmentbasiertes Paging (GE-645)

Figure 4: Möglichkeiten der Kombination von Paging und Segmentierung

nicht als benutzt markiert war, desto höher war die Wahrscheinlichkeit, dass diese bei der nächsten Anfrage ausgelagert wurde. Sobald die angeforderte Seite komplett übertragen war, konnte der Transfer der zu verdrängende Seite an den Trommelspeicher angestoßen und parallel dazu, das Programm fortgesetzt werden. Solange während der Übertragung nur auf vorhandene Seiten zugegriffen wurde, ist der Transfer komplett durch das Programm überlagert.

3. UMSETZUNG IN MULTICS

Virtueller Speicher in Multics wurde durch ein Zusammenspiel von Segmentierung und Paging umgesetzt. Die verwendete Zielarchitektur[5] koppelte diese beiden Prinzipien sehr eng. So wird Paging nicht global, wie beispielsweise bei x86, sondern für jedes Segment unabhängig konfiguriert. Zum besseren Verständniss ist diese „Verdrehung“ in Figure 4a und 4b schematisch abgebildet. Dabei muss jedoch beachtet werden, dass bei globalem Paging im Normalfall mehrstufige Tabellen eingesetzt werden, um den benötigten Speicherplatz zu reduzieren. Aufgrund der begrenzten Segmentgröße stellt sich dieses Problem bei separaten Seitentabellen für jedes Segment nicht.

Neben dem selektiven (De)Aktivieren von Paging erlaubte die Hardware zwischen Seiten mit jeweils 64 oder 1024 Worten zu wählen. Segmente eines Nutzers verwenden in Multics automatisch Paging mi einer Seitengröße von 1024 Worten. Änderungen durch den Anwender wurden von Multics dabei nicht erlaubt. Ledigliche für einige Segmente des Betriebssystems wurde ursprünglich die 64 Wort Variante eingesetzt. Da die Verwaltung verschieden großer Seiten die Komplexität stark erhöhte, wurde dies 1968 abgeschafft und konsequent auf Seiten mit 1024 Worten gesetzt [13, page]. Dies wäre auch eine Erklärung, weshalb sich in verschiedenen Quellen diesbezüglich widersprüchliche Aussagen finden lassen (vgl. [5, Abschnitt 5.2] und [11, S. 39]). Ehemals mit Seiten zu je 64 Worten verwaltete Segmente, wie das, in dem alle Seitentabellen untergebracht waren *(„system segment table“/SST)*, sollen durch Segmente ersetzt worden sein, die ohne Paging und über das *wired bit* (siehe Abschnitt 3.1.5) vor dem Auslagern aus dem Hauptspeicher geschützt waren [13, page table][11, S. 39].

3.1 Datenstrukturen

Bevor die Algorithmen, die unter Multics für die Implementierung von Paging zuständig waren, erläutert werden, sollen in diesem Abschnitt die relevanten Datenstrukturen und deren Zusammenhang beschrieben werden. Zum besseren Verständniss, ist in Figure 5 ein graphischer Überblick abgebildet.

3.1.1 Adressformat

Wie bei Segmentierung üblich, besteht eine Adresse aus dem Tupel (s, i), wobei s die Segmentnummer und i der Index in dem Segment ist. Unter Multics werden bis zu 2^{14} Segmente mit je einer maximalen Länge von 2^{18} Wörtern unterstützt (siehe Figure 3a) [8, S. 2]. Durch den Einsatz von Seitenverwaltung für die meisten Segmente, wird der Index i ein weiteres mal in die Seitennummer i_p und die Wortnummer i_w geteilt. Abhängig von der Seitengranularität ist der Wertebereich von i_w zwischen 0 und 63 bzw. 1023. Hingegen ist i_p immer 8 Bit lang (vgl. Figure 3b und 3c) [11, S. 12].

3.1.2 Segment Deskriptor Wort (SDW)

Wie der Name bereits vermuten lässt, werden in diesem Wort Informationen über ein Segment kodiert. Jeder Prozess hat ein „descriptor segment“ *(DS)*, in dem all seine gültigen Segment Deskriptoren gesammelt werden. Der Index s des Wortes in diesem Segment ist die zugehörige Segmentnummer. Wird ein neues Segment benötigt, wird überprüft ob ein bereits vorhandener Deskriptor unbenutzt ist und gegebenenfalls verwendet. Ungültige Deskriptoren können über das gesetzte *invalid* Bit identifiziert werden. Kann kein freies Wort gefunden werden, so wird ein weiterer Deskriptor angefügt[6].

Neben einer absoluten Hauptspeicher-Adresse und der Länge in Seiten werden die Zugriffsberechtigungen[7] des zugehörigen Prozesses auf dieses Segment gespeichert und jeweils ein Bit zeigt an, ob dieses Segment im Hauptspeicher (aktiv) ist, ob Seitenverwaltung in dem Segment gewünscht ist und ob Seiten mit 64 Worten verwendet werden sollen. Nur falls das Segment als aktiv markiert ist, wird die Hardware, abhängig von der aktivierten Seitenverwaltung, die Adresse als direkter Einstieg in das Segment oder als Beginn der zugehörigen Seitentabelle verwendet [11, Kapitel 1.2.6].

[5] General Electric 645 / GE-645

[6] bis zu einem Maximum von 2^{14} Segmenten [8]

[7] read, write, execute und append

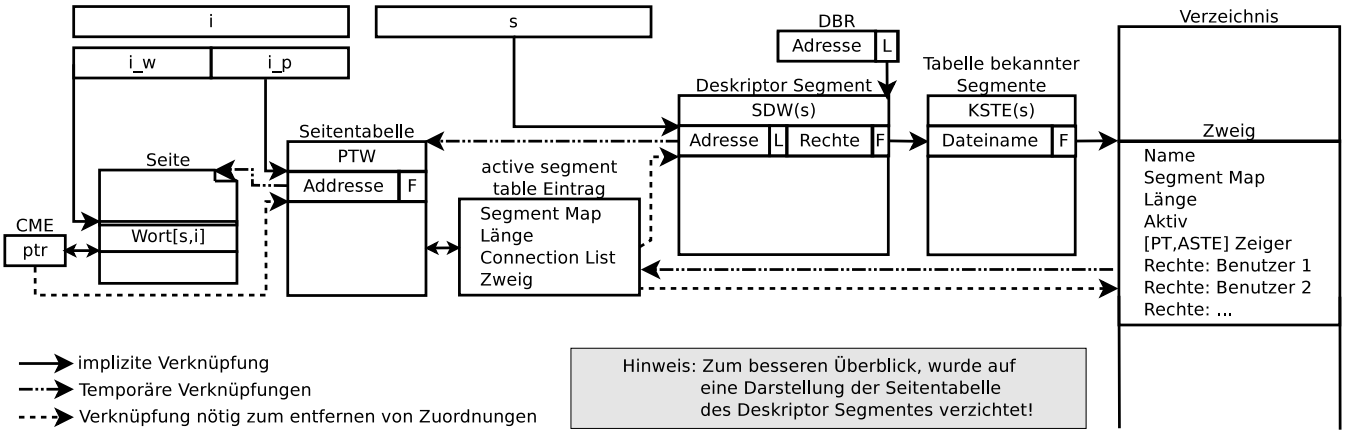


Figure 5: Zusammenhang der Strukturen zur Implementierung von virtuellem Speicher in Multics[5]

3.1.3 Tabelle bekannter Segmente

Für jedes SDW ist in der Tabelle bekannter Segmente *(„known segment table“/KST)* der Eintrag unter dem selben Index s reserviert. Jeder dieser Einträge *(„known segment table entry“/KSTE)* enthält den Pfad des zugehörigen Segmentes sowie einen Verweis auf den jeweiligen Zweig im Dateisystem. Die Trennung dieser beiden Strukturen ist nötig, da das SDW von der Hardware verarbeitet wird, dessen Struktur somit vorgegeben ist und kein Platz für diese erweiterten Informationen ist.

Wird ein Segment durch eine Pfadangabe referenziert, muss nach dem passendem KSTE gesucht werden. Sollte kein passender Eintrag gefunden werden, das Segment wurde von diesem Prozess also noch nicht verwendet, muss das Laden der zugehörigen Daten veranlasst werden [5, Kapitel 8.2].

3.1.4 Tabelle aktiver Segmente

Einer der Vorteile, der den erhöhten Aufwand durch Segmentierung und Paging rechtfertigt, ist die Tatsache, dass die Verwaltung des limitierten, häufig nicht ausreichenden, Hauptspeichers vereinfacht wird [6]. Entsprechend ist davon auszugehen, dass nicht alle bekannten Segmente im Primärspeicher Platz finden. Um zu Wissen, welche Segmente direkt zugreifbar sind werden diese in der Tabelle aktiver Segmente *(„active segment table“/AST)* eingetragen. Im Gegensatz zu den bisherigen Strukturen, ist diese Tabelle logischerweise global gültig und wird nicht für jeden Prozess separat geführt.

Jeder Eintrag dieser Tabelle *(„active segment table entry“/ASTE)* speichert die Adressen ausgelagerter Seiten, die Länge des Segmentes, Verweise auf die zugehörige Segment Deskriptoren[8] und den Zweig im Dateisystem. Die Duplizierung der Länge sowie der Adressen ausgelagerter Seiten[9] dient der Performance, da bei einem Seitenfehler diese Informationen direkt vorliegen.[5, Kapitel 8.3]

3.1.5 Seitentabelle

Zu jedem ASTE besteht eine feste Kopplung zu einer Seitentabelle *(„page table“/PT)* weshalb häufig von einem (PT/ASTE)-Paar gesprochen wird. Je nach Konfiguration des Systems, wird während des Systemstarts Speicher für eine bestimmte Zahl dieser Paare reserviert. Um bei kleinen Segmenten dennoch keinen Speicherplatz zu verschwenden, wurden ursprünglich sowohl Seitentabellen mit 64 wie auch 256 Einträgen angelegt. Jede dieser Tabellen wurde in die, der größe entsprechenden Liste freier *((PT/ASTE) free list)* oder benutzter *((PT/ASTE) used list)* Tabellen eingetragen. [5;, 13, page table]

Hardware Anforderungen.

Da die Seitenverwaltung durch die Hardware umgesetzt wird, muss auch deren Format für die Einträge *(„page table word“/PTW)* befolgt werden. Eines der wohl wichtigsten Bit in diesem Zusammenhang ist das *„missing page“* Bit, über das ein Eintrag als ungültig markiert werden kann. Wird ein derartiger Eintrag referenziert, bricht die Maschine die Operation ab und signalisiert einen Seitenfehler. Bei gültigen Einträgen hingegen, verwendet die Hardware die angegebene Adresse als Basis für den entsprechenden Seitenrahmen. Handelt es sich dabei um einen lesenden Zugriff, so wird lediglich das *„used bit“* gesetzt. Wurde der Inhalt der Seite jedoch verändert, setzt die Hardware das *„modified bit“* im zugehörigen Wort [6]. Einige weitere Bits wurden ebenfalls durch die Hardware verändert, die von Multics nicht weiter genutzt wurden und somit ignoriert werden können.

Software Bits.

Die verbleibenden Bits waren für die Hardware nicht relevant und konnten von der Systemsoftware frei belegt werden. Multics verwendete dies unter anderem, um jedem PTW einen der Typen `core`, `disk`, `pd`[10] oder `null` zuzuweisen. [3, 1]. Die ersten 3 Typen geben an, auf welchem Gerätetypen die Seite im Moment zu finden ist und somit, wie die gespeicherte Adresse zu interpretieren ist. Logischerweise, sollte bei deaktiviertem *„missing page“* Bit, der Typ auch `core` sein. Durch `null` können leere Seiten vorgemerkt werden. Sobald auf eine derartige Seite zugegriffen wird, sorgt das Modul *page control* dafür, dass ein Seitenrahmen allokiert wird, mit dieser Seite verknüpft und mit Nullen gefüllt wird.

Ein weiteres Bit *(„wired bit“)* diente dazu, eine Seite zu

[8] Segmente, die von Prozessen geteilt werden, haben in jedem einen Deskriptor, liegen aber nur einmal im Speicher

[9] liegen im Segment Deskriptor beziehungsweise Dateisystem Objekt vor

[10] „Paging Device“: Eine Art Cache zwischen `core` und `disk`

„verdrahten“. Derart markierte Seiten werden von dem Algorithmus zur Ersetzung von Seiten (siehe Abschnitt 3.2.2) nicht berücksichtigt und können somit niemals ausgelagert werden [13, wired].

Letztlich wird für die verwendete Ersetzungsstrategie für jede Seite ein k Bit Schieberegister benötigt um für die letzten k Zyklen zu Speichern ob die Seite in dem entsprechendem Zeitraum benutzt wurde. Um benötigte Seiten schneller auszusortieren, wird zusätzlich noch ein *first time bit* verwendet. Dies wird gesetzt sobald die Seite durch die Behandlung eines entsprechenden Fehlers geladen wird und bei der ersten Kontrolle deaktiviert. Eine genauere Beschreibung des Algorithmus und der entsprechenden Relevanz dieser Bit folgt in Abschnitt 3.2.1. Nach einigen Experimenten hat sich eine Länge von 1 oder 2 Bit für das Schieberegister als am Effizientesten erwiesen. Aus Gründen der Einfachheit sowie besserer Anpassungsfähigkeit wurde sich letztlich für die Verwendung eines einzelnen Bits entschieden [6].

3.1.6 Segment map

Dies Struktur bietet, ähnlich einer Seitentabelle, Platz für die benötigten Seitenzahl eines Segments. Aufgabe der Struktur ist es, Informationen zu speichern, um die Adressen ausgelagerte Seiten wiederfinden zu können. Abhängig davon, ob das Segment aktiv ist, liegt diese Datenstruktur in dem zugehörigen ASTE oder ist im zugehörigen Zweig des Dateisystems untergebracht [5]. Während eine derartige Struktur im Zweig des Dateisystems für ausgelagerte Segmente nötig ist, bin ich mir bei der Unterbringung im ASTE nicht sicher. Weder findet sich ein Hinweis auf diese Struktur im Glossar[13] noch in der entsprechenden Quelldatei[2]. Bei aktiven Segmenten wird, für ausgelagerte Dateien, die Adresse über den zugewiesen Typen in der zugehörigen Seitentabelle bereitgestellt [3].

3.1.7 Core map

In der „core map“ *(CM)* werden die Zuordnungen einzelner Seitenrahmen zu ihren Seiten gespeichert. Dies entsprecht der gegensätzlichen Richtung der Verbindung über eine Seitentabelle. Jeder Eintrag der CM wurde, abhängig ob dieser gerade benutzt wurde oder nicht, in die Liste der freien *(core map free list)* oder belegten *(core map used list)* Elemente eingetragen. Damit war es bei der Bearbeitung von Seitenfehlern möglich, direkt einen momentan ungenutzten Bereich im Hauptspeicher auszuwählen [5, Abschnitt 8.4].

3.2 Algorithmen

Die Einführung von seitenverwaltetem Speicher durch den Atlas Computer wurde gefolgt von einer Reihe verschiedener Studien über mögliche Strategien und Algorithmen zur Auswahl der auszulagernden Seite(n). Dabei stellte sich heraus, dass die Effizienz stark von dem Systemaufbau so wie der Nutzung abhängt. Erlaubt die Verwaltung durch Seiten, dass der benötigte Hauptspeicher gering gehalten werden kann[11], besteht vor allem bei zu kleinen Hauptspeichern die Gefahr, durch den automatisierten Datentransfer zu blockieren [6].

3.2.1 Behandlung von Seitenfehlern

Mit einem Seitenfehler signalisiert die Hardware dem Betriebssystem, dass bei der Auflösung einer Adresse eine benögtige Seite als inaktiv markiert ist. In Multics ist für alles,

[11] anstatt komplette Segmente reichen einzelne Seiten

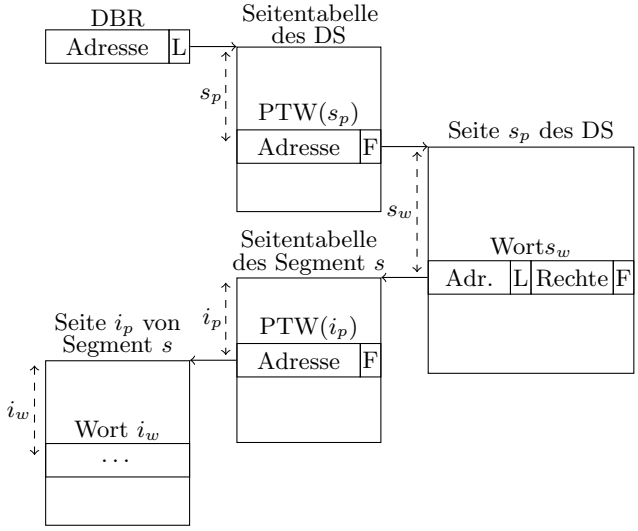


Figure 6: Segmentierung mit Paging der GE-645 am Beispiel der Adresse $\langle s, i \rangle$

was mit Seiten zusammenhängt das sogenannte *page control* Modul zuständig. Bevor jedoch auf die Funktionsweise dieses Moduls eingegangen wird, sollte zu Aller erst betrachtet werden, an welchen Stellen es zu solchen Fehlern kommen kann.

Adressauflösung durch die Hardware.

Erinnern wir uns an den Aufbau einer Adresse wie in Abschnitt 3.1.1 beschrieben. Entsprechend lässt sich die Adresse in die 3 Teile s, i_p und i_w unterteilen. Wenn man es aber ganz genau nimmt, so ist das Deskriptor Segment selbst in einzelne Seiten unterteilt. Somit lässt sich die Segmentnummer s ebenfalls in eine Seitennumer s_p und Wortnummer s_w unterteilen. Wie aus Figure 6 ersichtlich, werden also vier verschieden Adressen besucht. Zu erst wird in der Seitentabelle des DS das PTW mit Index s_p gesucht. An dieser Stelle kann der erste Seitenfehler auftreten, da das Deskriptor Segment kein *wired bit* gesetzt hat. Sobald diese Seite verfügbar ist, kann die Adressauflösung weitergehen. In der referenzierten Seite im DS wird über den Index s_w das passende SDW gesucht. Ist das Segment nicht aktiv, so wird die Hardware von hier die Kontrolle an das Betriebssystem übergeben um das Segment zu laden. Das laden eines Segmentes stellt aber lediglich sicher, dass eine Seitentabelle reserviert ist [5, Abschnitt 8.3]. Somit wird die Hardware für den Eintrag mit Index i_p erneut einen Seitenfehler signalisieren. Wurde auch dieser behoben kann schlussendlich das passende Wort i_w ausgelesen werden.

Theoretisch könnte bei einem länger inaktiven Prozess auch das DS ausgelagert sein, wodurch ein weiterer Segmentfehler aufgetreten wäre und die Kontrolle zum vierten mal an das Betriebssystem weitergereicht worden wäre. Bevor 1968 die Unterstützung der kleinen Seiten mit 64 Worten ersetzt wurde und das SST, in dem die einzelnen Seitentabellen untergebracht sind, auch mit aktivierter Seitenverwaltung betrieben wurde, konnte das Zugreifen auf die beiden Seitentabellen ebenfalls in jeweils einem Seitenfehler enden. Somit konnte eine einzige Adresse dazu führen, dass 4 Seitenfehler und 2 Segmentfehler auftraten. Somit die Kontrolle

6 mal an das Betriebssystem übergeben wurde. Dies konnten durch weitere Features der Hardware (ITS und ITB Zeiger) noch erheblich verstärkt werden [10].

Das page control *Modul.*

Ist ein Seitenfehler aufgetreten, so wird über die Fehlerbehandlung die Kontrolle an dieses Modul übergeben. Der Ablauf, der nun folgt, kann schematisch Figure 7 entnommen werden.

Zu Beginn der Fehlerbehandlung muss überprüft werden, ob die Seite bereits bearbeitet wird. Die Bearbeitung kann durch zwei Ereignisse hervorgerufen werden:

- Ein anderer Prozess benötigt die Seite ebenfalls und hat den Transfer in den Hauptspeicher bereits angestoßen (pull)
- Die Ersetzungsstrategie kam zu dem (falschen) Entschluss, diese Seite werde nicht gebraucht und ist dabei ausgelagert zu werden (push)

In beiden Varianten wird die Übertragung abgewartet und anschließend die Anweisung wiederholt, die zu dem Seitenfehler führte. Während nach einem pull der erneute Seitenzugriff gelingt, wird nach dem push der Zugriff direkt den selben Seitenfehler erzeugen und entsprechend behandelt.

Häufiger sollte jedoch der Fall eintreten, dass die Seite gerade nicht bearbeitet wird und die Ausführung in Block 2 landen. Um die angefragte Seite aus dem Sekundärspeicher zu holen, muss ein freier Seitenrahmen gesucht werden. Durch das führen der *core map free list* kann einfach das erste Element der Liste verwendet werden. Wie beim Atlas Computer, wird auch hier versucht immer einen freien Seitenrahmen vorzuhalten um die notwendigen Transfers mit Zugriffen auf den Hauptspeicher zu überlagern. Um dies Sicherzustellen, wird die, im nachfolgenden Abschnitt näher beschriebenen, Ersetzungsstrategie n mal aufgerufen um Platz für neue Seiten zu machen. Entsprechend befinden sich immer zwischen einer und n freie Seiten in der *core map free list*. Bei Multics wurde dieses n auf 3 festgesetzt [6].

Würden die Transfers direkt angestoßen, hätte man nichts gewonnen, da man erst auf deren Fertigstellung warten müsste um weiter machen zu können. Deshalb gibt es eine *push* und eine *pull* Liste. In diese Listen werden die Quell- und Zieladressen eingetragen und die Listen werden im Hintergrund automatisch abgearbeitet.

Nachdem jetzt sichergestellt ist, dass mindestens ein Element in der *core map free list* ist, kann dieser Seitenrahmen für die angefragte Seite verwendet werden. Dazu muss die Adresse der ausgelagerten Seite im Sekundärspeicher aus dem AST geholt werden und anschließend die Übertragung eingeleitet werden.

In Block 6 werden anschließend Aufräumarbeiten durchgeführt. Dabei ist es wichtig, zu beachten, dass die dort bearbeiteten Elemente nicht die eigene Seite ist, sondern bereits abgeschlossene Übertragungen anderer Prozesse, die in Block 8 darauf warten, über die Fertigstellung benachrichtigt zu werden. Für jede beendete Transaktion muss bei einem *pull* das zugehörige PTW angepasst werden. Dabei ist es wichtig, dass der neue Seitenrahmen, bei Verwendung des *first time bit*, trotz FIFO-Struktur der Liste an den Anfang gesetzt wird, um als erstes die Liste zu verlassen. Der Hintergrund dafür wird im nächsten Abschnitt genauer erläutert.

Während der Prozess mit dem Seitenfehler auf die Fertigstellung des Transfers wartet, werden andere Prozesse be-

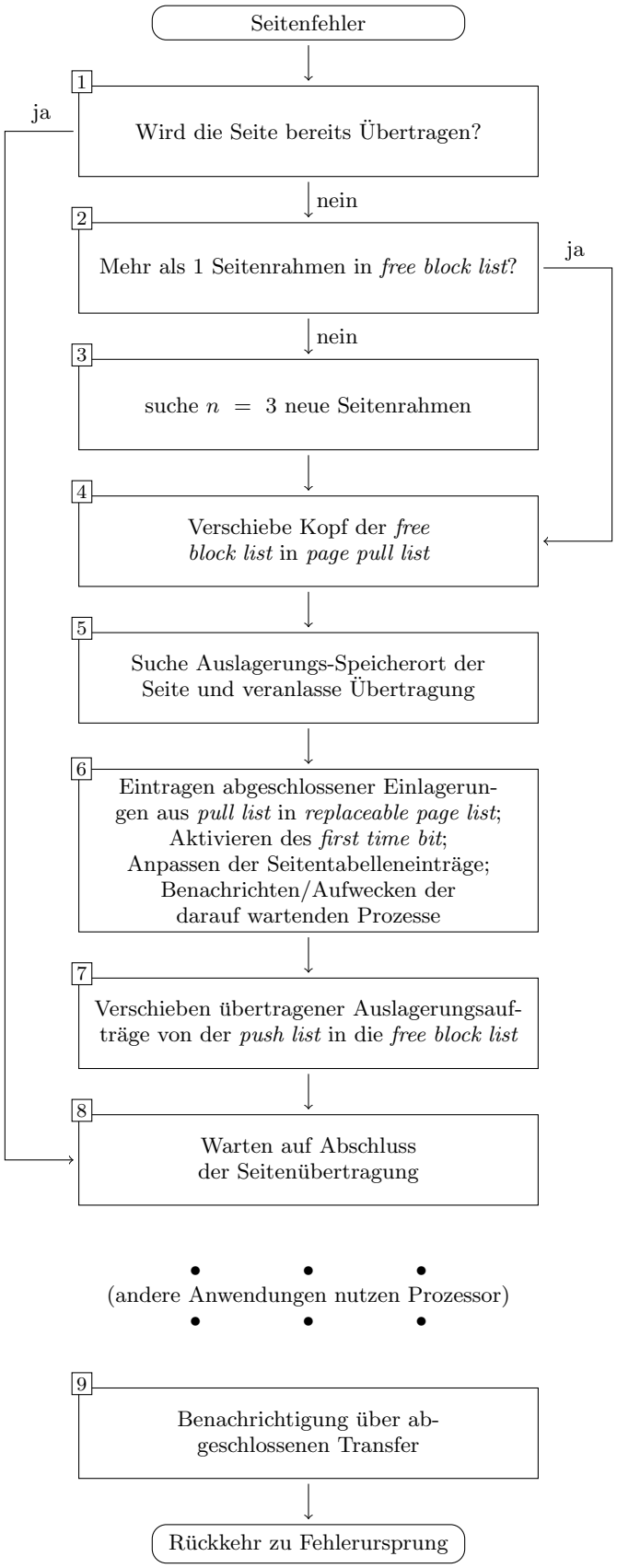


Figure 7: Flußgraph des page control Moduls [6]

dient. Irgendwann wird durch die Aufräumarbeiten eines anderen Prozesses in Block 6, dieser Prozess wieder aufgeweckt, sobald die Datenübertragung abgeschlossen ist und die Ausführung kann an der Stelle fortgesetzt werden, an der ursprünglich der Seitenfehler auftrat.

3.2.2 Ersetzungsstrategie

Wie Effizient eine seitenbasierte Speicherverwaltung ist, hängt in großen Teilen von der verwendeten Strategie, zur Auswahl der auszulagernden Seite ab. Werden schlechte Entscheidungen getroffen, so steigt die Zahl an Speicherübertragungen. Trifft sie hingegen gute Entscheidungen, muss dafür aber zu viel Aufwand betieben werden, wird auch dies die Leistung nicht steigern. Es muss also ein Algorithmus gefunden werden, der mit verkraftbaren Aufwand bestmögliche Entscheidungen trifft [9, Abschnitt V].

Basis des Multics verwendeten Algorithmus, bildete eine Liste nach dem first-in-first-out (FIFO) Prinzip, wobei zwei Modifikationen vorgenommen wurden:

- Jeder Eintrag erhält ein k Bit Schieberegister, mit dessen Hilfe häufig genutzte Seiten nicht ausgelagert, sondern unter bestimmten Voraussetzungen wieder an das Ende angehängt werden.
- Das Einfügen eines neuen Eintrags erfolgt nach dem last-in-first-out (LIFO) Prinzip, wobei dies in einem speziellen Flag *(first time bit)* vermerkt wird.

Nach dem die Randbedingungen geklärt sind, kommen wir nun zum eigentlichen Algorithmus. Um eine Entscheidung zu fällen, werden die folgenden Schritte durchgeführt:

1. nächstes Element e aus der FIFO holen
2. Ist das *first time bit* gesetzt, werden *first time bit* und *used bit* deaktiviert, e an die FIFO angehängt und wieder von vorne begonnen
3. Schieberegisters von e um eine Stelle verschieben und *used bit* an den freigewordenen Platz kopieren
4. *used bit* zurücksetzen[12]
5. Ist mindestens eines der k Bits gleich 1 wird e an die FIFO angehängt und wieder von vorne begonnen
6. e wurde während der letzten k Prüfungen nicht benutzt und wird ausgelagert

[12]da das Bit automatisch bei einem Seitenzugriff gesetzt wird, kann das nächste mal geprüft werden, ob die Seite erneut genutzt wurde

k	Seitenfehler	Behandlungsdauer
0	8309	$184.5sec$
1	4250	$107.9sec$
2	4098	$106.5sec$
4	4205	$112.5sec$
7	4317	$120.2sec$

(a) Last: Systemstart

k	Seitenfehler	Behandlungsdauer
0	3628	$72.9sec$
1	1659	$36.4sec$
2	1635	$37.5sec$
4	1598	$38.7sec$
7	1725	$44.3sec$

(b) Last: 10 Standard Befehle

Table 1: Einfluss von k auf die Performance [6]

Hintergrund des Einfügens nach LIFO-Prinzip und des Schritt 2 ist, dass eine Seite nur eingelagert wird, wenn auf sie auch zugegriffen wird. Wird eine neue Seite in die Liste aufgenommen (Figure 7, Block 6) wird der zugehörige Prozess aufgewecket und verlässt die Fehlerbehandlung um den Seitenzugriff zu wiederholen. Dadurch wird das *used bit* gesetzt. Würde die Seite an das Ende gesetzt, so müsste die komplette Liste einmal durchlaufen werden, bis dieses Bit ausgelesen wird. Da dies dann immer gesetzt ist, bleibt die Seite eine weitere Runde in der FIFO. Wenn die Seite also nur einmal genutzt wird, bleibt sie dennoch lange Zeit erhalten. Wird dieses Bit jedoch direkt verworfen (Schritt 2), so wird eine einmalig genutzte Seite schneller wieder ausgelagert.

Durch anpassen der Werte von k kann die Komplexität des Algorithmuses beliebig verändert werden. Offensichtlich ist für $k = 0$ der Algorithmus ein normaler FIFO. Je weiter sich k an ∞ annähert, desto mehr geht der Ansatz in Richtung *least-resently-used* Strategie an. Es ist jedoch zu beachten, dass jede Erhöhung von k zu einer zusätzlichen Iteration über die Liste aller Seiten führen kann [6].

3.2.3 Betrachtung der Effizienz

Der im vorherigen Abschnitt beschrieben und unter Multics verwendete Algorithmus lässt die Komplexität frei wählen. Aus diesem Grund wurde zur Bestimmung des besten Wertes für k ein Experiment durchgeführt. Wichtig ist dabei jedoch, dass der Algorithmus unter anderen Umständen als der verwendeten Testlast zu anderen Ergebnis kommen kann. Es also keine allgemeingültige Bestimmung von k sein kann.

Als Testfälle wurden zum einen der Systemstart verwendet und zum anderen ein kleines Programm mit 10 Standard Befehlen [6]. Durch die Einführung des Schieberegisters, kann die Zahl an Seitenfehlern sowie die benötigte Behandlungsdauer annähernd halbiert werden (vgl. Table 1a und 1b). Teilweise kann eine Verbesserung für $k = 2$ festgestellt werden sowie eine verminderte Fehlerzahl für $k = 4$. Die verminderte Fehlerzahl von $k = 4$ gleicht jedoch die erhöhte Laufzeit nicht aus, sodass die Behandlungsdauer insgesamt zunimmt. Für mich unerwartet, steigt die Zahl der Fehler bei dem Testfall mit $k = 7$ bereits merklich an, die Gründe werden in der entsprechenden Veröffentlichung jedoch nicht weiter untersucht.

Als Resultat der Tests wurde deutlich, dass sich unter den gegebenen Bedingungen für k entweder 1 oder 2 gewählt werden sollte. Letztlich wurde sich in Multics für $k = 1$ entschieden, da dies zu einem beständigeren Verdrängungsmuster neigt und somit mehr Platz für weiter Optimierungen aufgrund erkannter und wiederholter Muster bietet [6].

4. FAZIT

Auch wenn Multics weder Segmentierung noch die auf Seiten basiernde Speicherverwaltung eingeführt hat, wurde durch die geschickte Verbindung der beiden Konzepte ein interessantes System geschaffen. Durch die nathlose Integration und dem Stellenwert von Segmenten in dem System, wurde diese Art der Speicherverwaltung überall genutzt. Somit musste die Kommunikation zwischen Hauptspeicher und

Sekundärspeicher nur einmal durchdacht und effizient implementiert werden. Neben der besseren Wartbarkeit durch diese Kapselung, wird damit aber auch den Entwicklern anderer Systemmodulen sowie dem Endanwender diese Arbeit abgenommen. Somit profitieren Alle von der Expertise der jeweiligen Spezialisten.

Es kann also von einer gelungenen Integration der einzelnen Prinzipien in das gesamt Paket gesprochen werden. Ein Grund dafür, dürfte wohl auch darin liegen, dass die Entwickler nicht davor zurückscheuten, Ideen zu verwerfen, falls diese die Effizienz zu sehr beeinträchtigten oder mehr Probleme bereiteten als erwartet.

Betrachtet man den Einfluss auf heutige Architekturen, so kann festgestellt werden, dass die Grundidee, der kleineren Seitengröße, Seiten an die benötigten Tabellen anzupassen, auch heute nocht vorzufinden ist. Auch wenn die Option der verschiedenen Seitengrößen aufgrund der Verwaltungskompläxität nicht weiter umgesetzt wurde, wird dies mit Large und Huge Pages auf heutigen Architekturen noch angeboten und von vielen Betriebssystemen auch genutzt.

5. REFERENCES

[1] Multics Quelldatei: include/add_type.incl.pl1. http://web.mit.edu/multics-history/source/Multics/ldd/include/add_type.incl.pl1.

[2] Multics Quelldatei: include/aste.incl.pl1. http://web.mit.edu/multics-history/source/Multics/ldd/include/aste.incl.pl1.

[3] Multics Quelldatei: include/ptw.incl.pl1. http://web.mit.edu/multics-history/source/Multics/ldd/include/ptw.incl.pl1.

[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[5] A. Bensoussan, C. T. Clingen, and R. C. Daley. The multics virtual memory: Concepts and design. *Commun. ACM*, 15(5):308–318, May 1972.

[6] F. J. Corbato. A paging experiment with the multics system. In *In Honor of Philip M. Morse*, chapter 19, pages 217–228. M.I.T. Press, 1968.

[7] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 185–196, New York, NY, USA, 1965. ACM.

[8] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in multics. *Communications of the ACM*, 11(5):306–312, 1968.

[9] T. Kilburn, D. Edwards, M. Lanigan, and F. Sumner. One-level storage system. *Electronic Computers, IRE Transactions on*, EC-11(2):223–235, April 1962.

[10] M. Krüger. Der General Electric 645 Computer. https://www4.cs.fau.de/Lehre/WS15/MS_AKSS/papers/05-GE645-Maximilian-Krueger.pdf, 2015.

[11] E. I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972.

[12] B. Randell. A note on storage fragmentation and program segmentation. *Communications of the ACM*, 12(7):365–ff, 1969.

[13] T. H. Van Vlerck. Multics glossary. www.multicians.org/mgloss.html, 2015. [Online; abgerufen 22. November 2015].