Multix

CS380L: Mike Dahlin

October 27, 2009

Dealing with failure is easy: Work hard to improve. Success is also easy to handle: You've solved the wrong problem. Work hard to improve. From ACM's SIGPLAN publication
Article "Epigrams in Programming", by Alan J. Perlis of Yale University, Sept 1982

Simplicity does not precede complexity, but follows it.
ibid.

1 Preliminaries

1.1 Review

More admin stuff in middle of class. Let's get started with technical.

1.2 Outline

1. Multix overview – sounds great
2. Details – gets complex
   - Sharing: Virtual memory and sharing
   - Protection: Fine-grained protection policies
3. Reflections

1.3 Preview

Next class: THE, Unix

1.4 Quiz!

2 Corrections/clarifications

TBD: Fold the following corrections/clarifications into main notes

```
From:  thvv@multicians.org
Subject:  Multics
Date:  October 26, 2009 10:10:28 AM CDT
To:  dahlin@cs.utexas.edu
```

Hello Prof. Dahlin,

I am the edtitor of multicians.org, a site that
contains information you use in class CS380L.

I backtracked a link form your course website and
looked at your notes on "Multix" in the file
 http://www.cs.utexas.edu/users/dahlin/Classes/GradOS/lectures/multix.html
and noticed a few errors.

Here are some corrections, respectfully submitted.
Feel free to ignore my comments. I submit them in the hope that
students won't spend time learning things they later have to
unlearn.

regards, tom

(I worked on both CTSS and Multics from 1965 to 1981 as a
programming staff member at MIT and Honeywell, and I have
collected a lot of information, so I think these are accurate.)

1. The name of the system is Multics. not MULTICS, not Multix.
 http://www.multicians.org/multics-humor.html

2. To say it "never worked" is peculiar. Would you say
that of Windows? It never took over the OS field the way
we hoped. But there were a lot of systems sold and used.
(would you like me to add a section to
  http://www.multicians.org/myths.html
pointing at your class notes, and explaining how I disagree?)

3. Regarding CTSS. "4 consoles, 2 tape drives per console"
describes the CTSS tape incarnation, demonstrated in 1961
on the 7090. But the system that went into service for
MIT users by about 1963 had harware memory protection, swapped
to drum and disk, and supported about 30 dialup users.
 http://www.multicians.org/thvv/7094.html

$. Just as you conflated the initial demo CTSS with the
system that became a service at MIT for 10 years, you have
combined the initial second-system Multics of 1965-70,
implemented on the GE 645, with the "third system" Multics,
implemented on the Honeywell 6180, which became a commercial
product and service at about 80 sites. You probably know
that the last few Multics systems were shut down in 2000
at the Pentagon and Canadian National Defence.

6. Dynamic linking. You do a nice job of developing the need for the linkage section, and how it is used. There are more details in
http://www.multicians.org/exec-env.html
There are a few fine points though. The segmentation system does indeed keep str_seg, which keeps a list of descriptor segments that have a segment mapped, so that an access control change can reset these SDWs and cause processes to recalculate access, if e.g. the access control list of a segment changes. This does not cause all processes to regenerate their linkage section.

7. Typo: s/Salzer/Saltzer/

8. Your discussion of "protected subsystem" does not correspond to the actual Multics implementation.
Multics descriptors contain ring brackets.
Mike Schroeder describes the actual Multics mechanism:
http://www.multicians.org/protection.html
the details in exec-env.html are based on the actual code (on-line at MIT).

You ask, "how do you make this fast?" and the answer is a) caching ("associative memory") in the CPU plus hard circuitry for doing the comparison of ring to ring brackets.

We understood that the linear ring structure was not completely expressive. Proposals for more complex support for mutually suspicious subsystems were never worked out: they imposed higher overhead, did not match the semantics of existing languages, and had bothersome corner cases.

3 Multics overview

Hugely influential operating system
    Very ambitiou
    Not widely deployed
    But core ideas live on in Unix, NT, MacOS
Originally a small subset (early 70's unix)
This subset is growing
    Purpose of using paper in this class

- Historical context: many basic concepts we take for granted today come from Multics

- Technical depth: something very like the dynamic linking mechanism still used today

3.1 Background

3.1.1 "Second system effect"

- 1st system

  - terrified of failure
  - simplified to bare bones
  - successful beyond its intended life-span

- 2nd system:

  - hugely ambitious
  - usually conceived by academics
  - many good ideas
  - a little ahead of its time
  - doomed to fail

- 3rd system:

  - pick and choose essence
  - usually made by good hackers
  - emphasize elegance and utility over performance and generality
  - become widely adopted

- 4th systems:

  - maturation

3.1.2 Multics lineage

- 1st system: CTSS
- 2nd system: Multics
- 3rd system: Unix
- 4th system: BSD

3.1.3 CTSS

- Compatible Time Sharing System

  - transistion from batch processing to time sharing
  - work on line and store info on line

- Features

  - Hardware: 4 consoles, 2 tape drives per console
  - interactive debugging (!)
  - editors
  - shells
  - no protection among users

- By early 60's, CTSS is showing its limitations

  - more memory, disks
  - Want sharing and protection!

3.1.4 Multics

- MIT, Bell labs, GE
- Redesign everything (second system!)

  - custom hardware
  - New programming languages
  - New OS

- Success or failure?

  - Never really worked
  - Probably the single most influential OS
  - Explored virtually every single aspect of OS design (except networking)

3.1.5 "Extreme Research"

- Hallmark of much influential work:

  - Take a simple good idea and push it to its logical extreme
  - Learn a lot
  - Then, take a step back and make it practical

- What is extreme idea in this paper?

  - *Extreme sharing*
  - Share anything (code, data)
  - Fine grained sharing
  - Dynamic sharing
  - Flexible protection

3.2 Principles

4 key ideas

1. Naming and addressing

   Combine virtual memory and file system

   File name + offset in file == segment # + offset in segment

2. Fine grained sharing

   Share procedures instead of entire programs

3. Dynamic linking

   Recompile anything without relinking anything

4. Autonomy

   independent address spaces

   (Making virtual addresses independent in presence of fine-grained sharing is hard!)

Consequence: "The values of identifiers that denote addresses within a segment which may change wiht recompilation must not appear in the content of any other segment."

3.3 Security principles

Enunciated by Salzer in this paper (and still very influential)
(refined and extended in a later paper – I'll use that terminology)

1. Failsafe defaults

   Base protection mechanisms on permission rather than exclusion

   → Conservative design – if you make a mistake, system is still safe

2. Complete mediation

   Check every access to every object for *current* authority

   e.g., make sure cached mappings are consistent with truth

   In this paper, concern about how changing a user's name could affect system → don't allow re-use of name

3. Open design

   Mechanism should not depend on ignorance of potential attackers, but on possession of specific keys

   Allow design review

   Obvious (but still violated today. Russian security expert got arrested for violating DMCA in 2001 for revealing that a particular commercial copy-protection scheme was just using ROT13!)

4. Least privlege

   Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job

   Obvious (?) – how well do we do today? (Sendmail? Web helper apps?)

5. Psychological acceptability

   Design human interface for naturalness, ease of use, and simplicity so that users will routinely and automatically apply the protection mechanisms

   Not obvious (until stated). Really hard.

   Fundamental problem (in 1973 and 2002): fine grained sharing + least privlege v. psychological acceptability

6. Economy of mechanism

   Not stated explicitly in this paper (but explicit later)

   Have one access control mechanism and use it everywhere

This paper: “the storage system...handles almost all of the protection responsibility in Multics.”

Moderately controversial (“elegance” v. “defense in depth”)

Evaluation of principles **QUESTION**: are these principles obviously good? Are they routinely followed?

- Basically non-controversial/obviously good
- Routinely violated in commercially acceptable system
- QUESTION: examples of violation of each?
- Even in this paper, after these principles are listed (and you say “sounds right”), they start going into details (and you say “argh”)
- Where does the complexity creep come from?
- The paper identifies two Multics goals: “decentralization of setting of protection” and support for “protection schemes not anticipated in the original design” (protected subsystem). Do these interact with the principles to increase/decrease complexity?
- Is there tension among these competing goals? Where/why?
- WHAT IS THE LESSON?

Preview Rest of lecture, look at details in more depth
o Virtual memory system: Fine grained protection and sharing
o Details of protection in overall system
Both areas: more complexity than expected
Think about how these systems meet the principles. Where could compromises have been introduced or features given up to get 80% of functionality with 20% of effort?

4 Admin

Pre-req quiz – Don’t worry about grade (people did OK, I won’t weight it heavily); as you see from Multics paper – need to have a good grasp of undergrad concepts (paging, segmentation) to make it possible to get through the papers we will be reading.
Updates schedule – next week THE and Unix
Comments on SHARP and Slice?
Project: topic selection, schedule

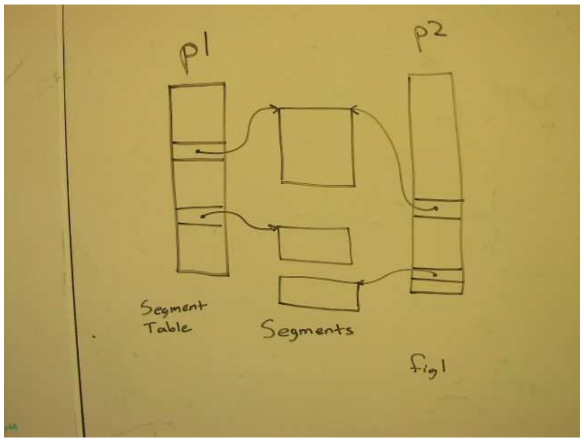
- By end of this week
- project teams and topic
- For ”type 1 projects” (building a mini-os):
  - Modify the list of milestones and assignments from the undergraduate OS class to include new milestones for porting functionality to the TRIPS emulation board environment (discuss these milestones with the instructor and with Bill Yoder).

- For ”type 2 and 3 projects” (repeating a prior result or original research):
  - Summary of the issue you want to address and why (no more than 5 lines)
  - List at least 3 papers you plan to begin reading for your literature survey (note that these should serve as a starting point; you will probably want to read more for the project proposal.)
- *You have less than 2 weeks until the project proposal deadline.*

5 Details: Sharing

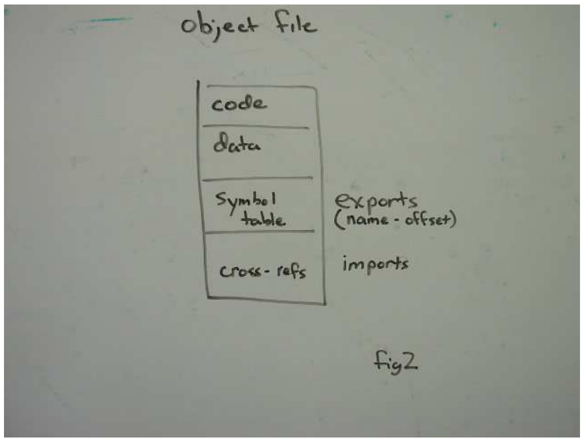
5.1 Virtual memory and sharing

5.1.1 Basics



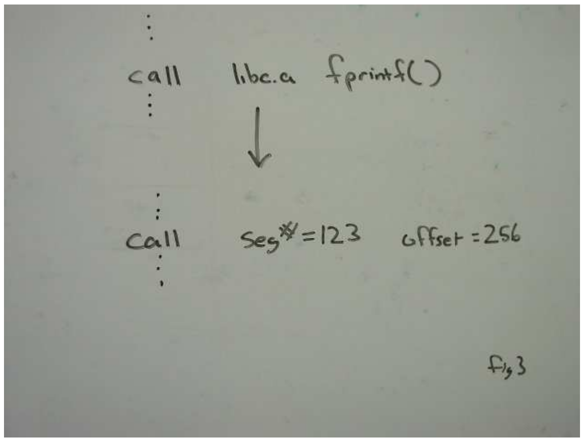
- Process = address space $\doteqdot$ segment table (“Descriptor table” in Daley and Dennis”)
- Generalized (virtual) address = segment index + offset
- (In reality, there is an additional level of paging, but not important for understanding “basics” of VM and sharing)

5.1.2 Problem 1: Dynamic linking



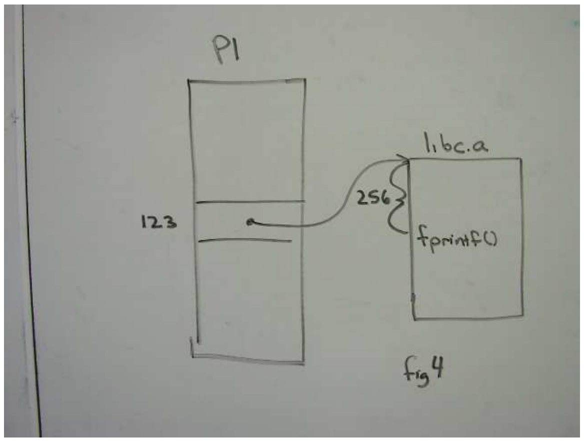
- Review: what's in an object file? (e.g., libc.a)
- Static linker: resolve addresses statically – compile into your executable
- Dynamic linking: resolve these addresses at run time
- Problem 1: How does dynamic linking work?

5.1.3 Dynamic Linking



- Cross-refs are initially symbolic names
- "Link trap" on first reference
- Dynamic linker overwrites symbolic address with general address
- Return and retry instruction

5.1.4 "Making Known" segments and the dynamic linker

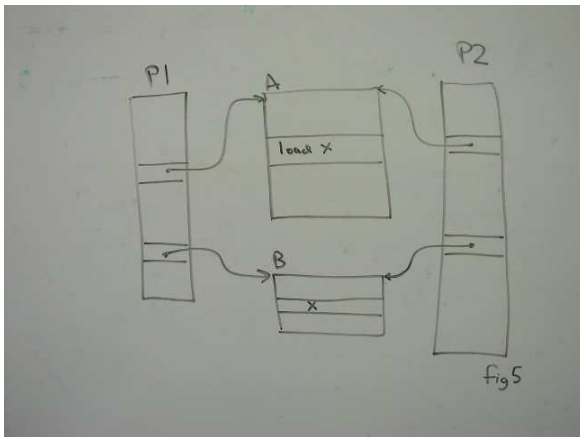


1. Is the referenced file in the segment table?
   segment table: seg nuame → seg num
2. No: "Make known" the segment
   – the OS maps the file (like mmap())

3. Find the symbol in the symbol table of the referenced segment

So dynamic linking is pretty easy: just keep a **segment table** (per process) and **symbol table** (per segment)

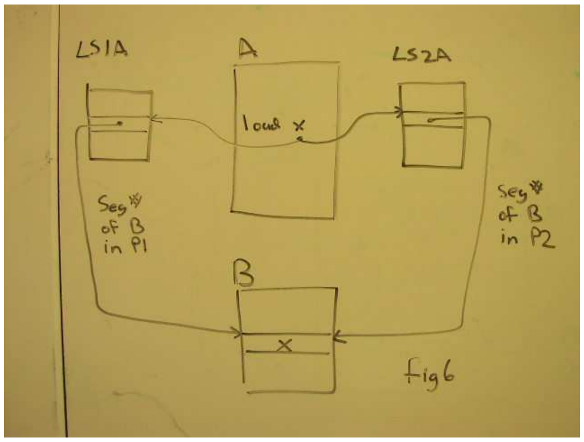
5.1.5 Problem 2: Shared data in shared code



QUESTION: What's the problem?

- Virtual addresses (x, for example) are different for different processes
  – B/c independence of address spaces
- But the code segment (load x, for example) are shared by different processes
- So, can't overwrite the addresses in the code segment
- Can only overwrite process-specific state!

5.1.6 Linkage Section



- A linkage section per segment, per process
- Within linkage section: one entry for every item imported from other segments
  Contains segment number in this address space of referenced segment
- *References to external items now: offset within "my" linkage section*

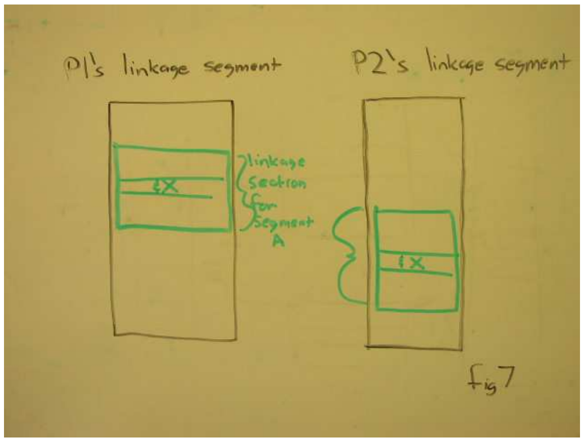
- "My" linkage section not ambiguous – linkage section for segment of current instruction within current process

So dynamic linking is pretty easy: just keep a **segment table** (per process) and **symbol table** (per segment), **linkage table** (per segment per process)

Sequence of events

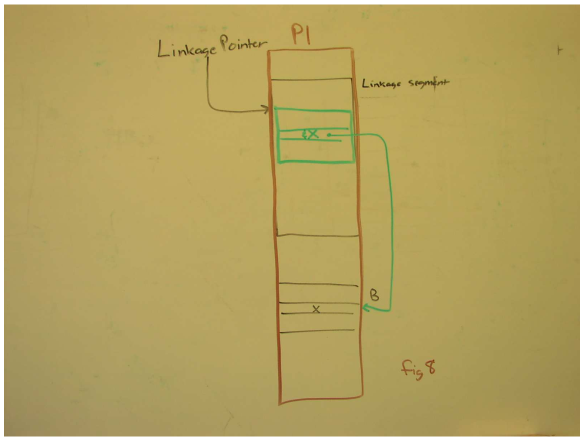
1. Link trap for data reference
2. Map referenced segment if necessary (make known)
3. Modify address in linkage section

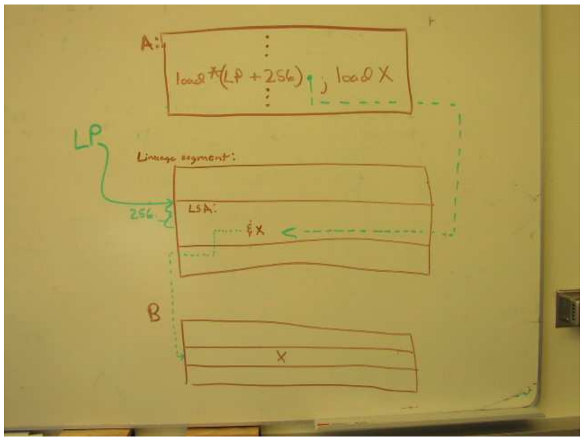
5.1.7 Problem 3: how to find (per segment, per process) linkage table? how to make this efficient?



- Linkage segment: all linkage sections of a process grouped into one segment
- Layout of each linkage section of the same segment is the same
- Relative ordering of linkage sections within a linkage segment differs
- *QUESTION:* How do I reference an entry in a linkage section?

5.1.8 Linkage pointer

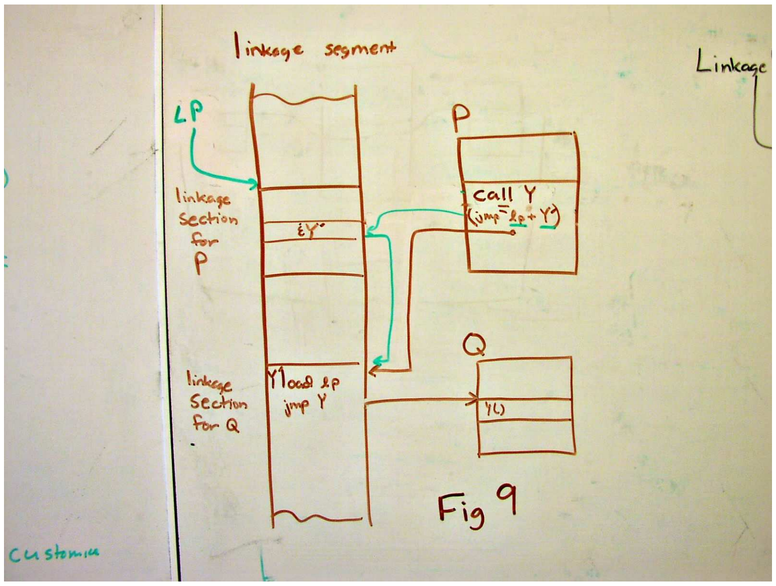




- HW Register (per-process)
- When jump to code segment Z, put beginning address of corresponding linkage section in register
- All references to external data via linkage section are relative to linkage pointer
- e.g., `load X` → `load *(LP + constX)`
- QUESTION: how to set this up on subroutine call? What do you do for "jsr 'seg=A offset=o"'?
  - e.g., `jsr ‘‘seg=A offset=o’’` →
    ```
    push LP
    mv ‘‘A’s linkage section pointer’’ → LP
    jsr ‘‘seg=A offset=o’’
    pop LP
    ```
  - And the "linkage section pointer" and "seg=A offset=o" pointers can be overwritten with the right per-process virtual address at segment load time, right?
  - Problem: what if call is from shared segment? Can't really update the instructions in place!

  → Put code in linkage section!

5.1.9 Procedure call: Detailed sequence of events



Questions:
When procedure X in segment P calls procedure Y in segment Q

- Where does new linkage section for Q come from?

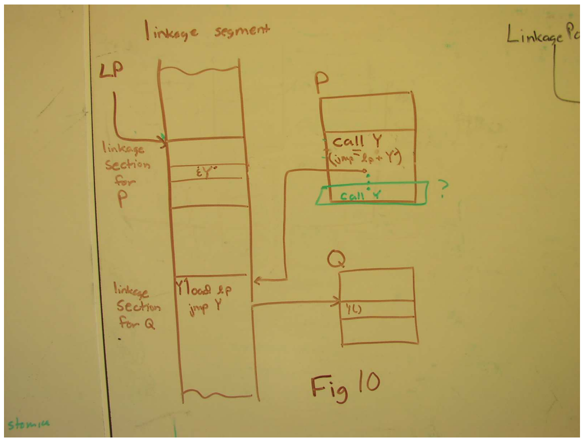
  When Q is referenced for first time, instantiate a new linkage section for procedure Q.

- How do I change linkage pointer?

  Put the code that changes the linkage pointer in the callee's (Q's) linkage section.

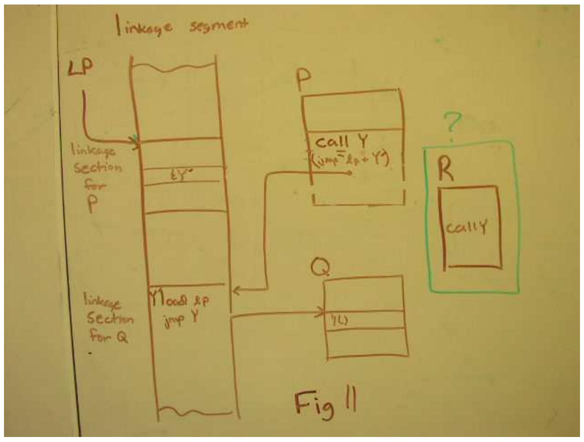
1. Link trap when referencing a symbolic form of Y'
2. Map code segment of Q (making it known)
3. Instantiate a linkage section for Q
    - Copy external references from Q's object file
    - Manufacture 2 instructions for each exported procedure
      – one to load linkage pointer – one to jump to procedure Y
4. Change linkage for P to point to right place (optimization: avoid future traps)
5. Resume execution

5.1.10 Question 1



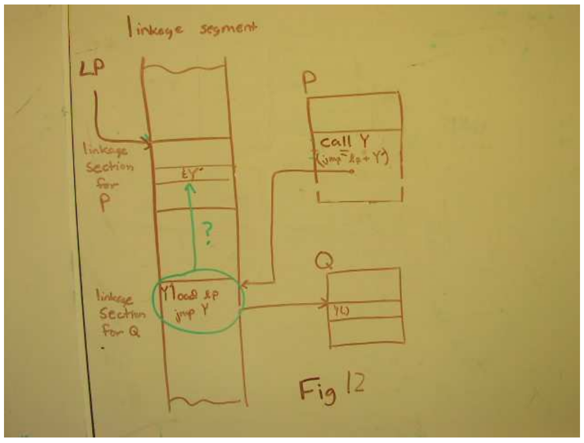
What happens if there is a second call to Y in P?

5.1.11 Question 2



What happens if there is another segment R that also calls Y?

5.1.12 Question 3



Can I move the “prolog code” into the caller’s linkage section?

5.1.13 Question 4

What if Q is recompiled and reloaded to a new physical address? What if in the new version, the offset to Y within Q changes?

Paper points out that for security, multics keeps "back pointers" from segment table to all address spaces that have mapped the segment. So, we can find all of the references to the segment, and update the segment-to-physical address mapping. Furthermore, we can regenerate all of the linkage segments to update the offsets, too.

5.2 Reflections: Yikes. How did we get into this mess?

This started simply – segmented addressing right out of undergraduate OS class "This fact [that segment numbers and word numbers are nonlocation dependent data and that location dependent information is contained only in processor registers] greatly simplifies the bookkeeping required by the system in carrying out relocation activity." [Daley and Dennis p 309]

What were the four principles?

1. Naming and addressing
2. Fine grained sharing
3. Dynamic linking
4. Autonomy (*independent* address spaces)

5.3 Reflections: Evolution of dynamic linking

Post-traumatic stress syndrom for next 20 years in terms of dynamic linking and fine-grained sharing
- Unix
- Pilot

Until the empire strikes back in the 80s
- MIT X windows
- Megabytes of X toolkits cause Unix workstations to thrash - Need shared libraries

Similar mechanisms now standard in all major OS's - *disclaimer: I don't know the details...*

6 Details: Protection

6.1 On-disk ACL

- Each segment on disk is stored with an ACL (Access Control List)
  - *name.project.role rwx*
  - e.g., dahlin.380l.* rwx
  - e.g., tiwari.380l.* rwx
  - e.g., *.380l.* r

6.2 In-memory descriptor

- ACL = "truth"
- But ACL is really flexible, list of strings, etc. → slow to check it on each access

→ In-memory descriptor – second level of protection – fast path that still expresses semantics of ACL

- Requirement: Different protection for different users
- How? Include protection information in the *descriptor* stored in segment table entry (note: Salzer paper calls this the "descriptor segment.")
- So, when I update a linkage section entry, I can set the r/w/x bits as needed *on a per-process (that is, per user) basis* according to what is in the ACL for the segment
  - Note: back-pointers from ACL to all segment tables in which they appear so that updates can be propagated.

Segment table entry: (fig 1 salzer)

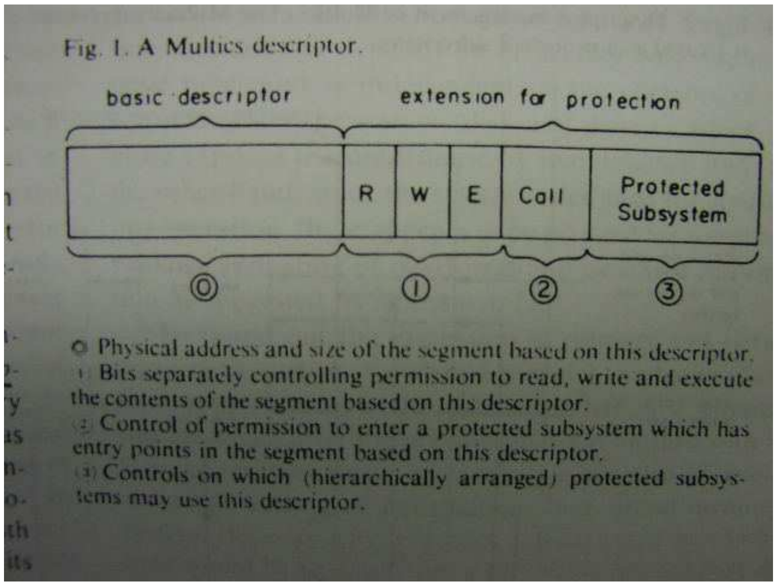


Fig. 1. A Multics descriptor.

⓪ Physical address and size of the segment based on this descriptor.
① Bits separately controlling permission to read, write and execute the contents of the segment based on this descriptor.
② Control of permission to enter a protected subsystem which has entry points in the segment based on this descriptor.
③ Controls on which (hierarchically arranged) protected subsystems may use this descriptor.

0. physical address and size of segment
1. read/write/execute bits
2. Control of permission to enter protected subsystem with entry points in this descriptor
3. Controls on which hierarically-arranged protected subsystems can use this descriptor

- How do you make this fast? (In multics: I dunno; CISC, I guess. Today: TLB caches per-process permissions...)
- That just uses entries 0 and 1; what are 2 and 3 for?

6.2.1 Protected subsystem

What:

- HW support to generalize user mode/supervisor mode (8 rings of protection)
- Kernel (and other protected subsystems) mapped into user's address spaces
- Want user to be able to call *some* kernel procedures (i.e., public methods) but not be able to call arbitrary kernel code (i.e., private methods or arbitrary instructions)
  - Note: multex (and x86) – jump to entry point in more privleged code
  - RISC: trap (through trap vector) to more privleged code
  - Basically same idea...
- Once in kernel procedure, want kernel code to be able to call other methods (i.e., private methods) and/or read/write other protected data
- *QUESTION:* Can we do this with just the r/w/x bits and ACL above?
  - No: segments are mapped in with per-user privleges; need per-code privlege.

6.2.2 Strawman: simple rings

- Each segment is associated with a ring number ("3" above)
- Each process has a *current ring number*
- A process executing in ring i can read and write segments in ring j if $j >= i$ but can not read or write segments in ring k if $k < i$.
- A process executing in ring i can execute a segment in any ring
  - The current ring number is set to the ring of the segment being executed
- How would we do this with mechanisms in place?
- What does it give us?
  - We can (almost) arrange for "protected" data to only be accessed by "trusted" code.
  - E.g., on-line grade-book that allows anyone to query their grade after entering their (class-supplied) password. You can't read the raw grade-book file, but you can query your entry by going through code I supply
  - em QUESTION: What's wrong?

6.2.3 Multics rings of protection

- 2 problems with straw man
  1. Must not allow arbitrary instructions to be executed. Instead, restrict execution to special *entry points.*

2. Not all inner ring programs should be accessible from all outer ring programs. E.g., allow a privleged mail-move program to only be accessed by (semi-privleged) mail-reader program

- Solution: Each segment descriptor stores
  - *Access bracket*: a pair of integers: $b1$ and $b2$ such that $b1 <= b2$
  - *Limit*: integer $b3$ such that $b3 >= b2$
  - *List of grates*: entry points at wich the segment may be called
- Case 1: Process executing in ring i calls segment with $b1 <= i <= b2$
  - Allow call
  - current ring number of process remains i
- Case 2: Process executing in ring i calls segment with $i < b1$
  - Allow call
  - Change current ring number to $b1$ (access rights of process are reduced)
    * Housekeeping: if parameters passed in that refer to segments in a ring lower than $b1$, these segments copied to an area accessible to ring $b1$
- Case 3: Process executing in ring i calls segment with $b2 <= i <= b3$
  - Allow call *if* call is to a gate
  - Change process current ring number to $b2$
  - *Allows carefully controlled call from process with limited access rights to gain more rights*
- Case 4: Process executing in ring i calls segment with $b3 < i$
  - Forbid
- Implementation details
  - Above was for jump/call. Similar for R/W: process with ring i can r/w a segment with access bracket $b1$, $b2$ if $i <= b2$
  - *I think* that case 1 was the "fast path" in hardware, and if that check fails, a trap occurred
  - So way 8 levels instead of 2? To support least privlege. Is linear range of numbers good enough? (Java is more general; what, if anything, do they gain?)
- example: classroom grading program
- ring 0: devoted to kernel
- ring 1: mail program (which needs to be able to create files in arbitrary directories)
- ring 2: grading program, used to evaluate student programs
- ring 3: student programs

6.2.4 Complexity creep example

- 3 partitions to principal ID: user.project.compartment
- Is this the right 80/20 point?
  - Yes: I wish I could run, say, my browser in a restricted version of "me"
  - No: Too complex to really understand ("Dad, you're the right guy, and you set your group right, but you forgot to reset your partition...") Read through the section and examples, watch Gilligan's island, and explain it to me.
  - No: Too restricted. It is complex but still cannot describe important cases. (Subsequent work describes more general algebras for group description, though the "principal", "role", "group" notion lives on...)

  What other examples of possible complexity creep?

6.2.5 2 main weaknesses

Paper raises two main weaknesses:

1. "Large number of trusted modules"
   - It cites 3 causes: economics, rush, lack of understanding
   - Are these implementation limitations (as the paper seems to imply) or fundamental? (e.g., won't every software system face these constraints?)
   - Solution (conventional wisdom): "security kernel": explicitly identify/minimize code on which security depends
2. User interface complexity

   Fundamentally hard...

6.2.6 Reflections: commercially successful systems violate "obvious" security policies even today

6.2.7 Research questions

- Your good ideas here
- How to resolve tension between fine-grained access control, least privlege v. psychological acceptability (probably no general answer, but perhaps a direction from which to attack practical questions like Outlook e-mail viruses, browser helper viruses, Java VM design, ...)
- Many ideas "taken out" of Multics in Unix and successors were gradually added back in (e.g., dynamic linking). Revisit the "protected subsystem" idea, which allows user-specified access-control extensions. Given the wealth of work on extensible operating systems in the 90's, perhaps protected subsystems can now be re-introduced. How to do it simply? Is there a killer app? (Possible app: web hosting/application service provider hosting?) (Additional related work: *doors* were reintroduced in Solaris a few years back.)