งานชิ้นที่ 1 วิชา 01076262 Compiler Construction

“Binary Executable File Formats”

เสนอ

อาจารย์ อัครเดช วัชระภูพงษ์

จัดทำโดย

นายศรณรงค์ ศรีมาคาม รหัสนักศึกษา 55011187 Sec 2

นางสาวอัญชัญ ฉันทวรลักษณ์ รหัสนักศึกษา 55011454 Sec 2

ภาคการเรียนที่ 2 ปีการศึกษา 2557

ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

สารบัญ

เรื่อง **หน้า**

1. a.out
2. COFF
3. ELF
4.Mach-O
5..COM
6..EXE(MZ)
7..EXE(PE)
8.การนำไฟล์ซอร์สโค้ดภาษาระดับสูงมาแปลงเป็นไฟล์ออปเจ็กต์โค้ด
9.การเพิ่มข้อมูลสำหรับดีบัก
10.การออปติไมซ์
11.การเชื่อมไปไลบรารี่ทั้งแบบสแตติกและแบบไดนามิก

Structure and feature of binary executable file formats

a.out

a.out เป็นรูปแบบไฟล์ที่ใช้ในระบบ Unix เวอร์ชั่นเก่า เป็นไฟล์ที่สามารถปฏิบัติงาน เป็น object code ต่อมาในระบบอื่นๆก็เป็น library ที่ใช้ร่วมกัน (a.out = assembler output) นอกจากนี้ a.out ยังเป็นชื่อสำหรับค่าเริ่มต้นของ output ไฟล์ปฏิบัติงานได้ที่ได้จาก compiler และ linker เมื่อไม่ได้ระบุชื่อไฟล์ output ไว้ แต่รูปแบบไฟล์ดังกล่าวจะไม่ใช่ a.out

Format

- ✓ OMAGIC หลายๆ segment ติดกับหลัง header โดยไม่มีการแยกกันระหว่างข้อความหรือข้อมูล เรียกอีกชื่อหนึ่งว่า object file
- ✓ NMAGIC ข้อมูลจะถูกโหลดบน page แรกสุดหลังจากจบ segment ข้อความ และ segment ข้อความสามารถอ่านได้อย่างเดียว
- ✓ ZMAGIC เพื่อรองรับ paging ตามความต้องการ ความยาวของ code และ segment ข้อมูล ในไฟล์จะมีขนาดเป็นเท่าของขนาด page
- ✓ QMAGIC รูปแบบไฟล์นี้จะทำการโหลด 1 page จากพื้นที่ virtual address ล่างสุด ในการที่จะอนุญาตให้วางตัวดักการนำ reference ของ null pointer ออกผ่าน error ของ segmentation
- ✓ CMAGIC เวอร์ชั่นเก่าของ Linux ใช้รูปแบบนี้สำหรับ core dumps

โครงสร้างของ a.out

exec header
text segment
data segment
text relocations
data relocations
symbol table
string table

exec header ประกอบไปด้วย parameter ที่ใช้โดย kernel สำหรับโหลดไบนารี่ไฟล์มาเก็บในหน่วยความจำ และถูกใช้โดย link editor เพื่อรวมไบนารี่ไฟล์หลายๆไฟล์

โครงสร้างของ exec header

ชื่อ	คำอธิบาย
a_midmag	เก็บค่าอย่างใดอย่างหนึ่ง ระหว่าง Flag,machine id และ magic number
a_text	เก็บขนาดของ text segment ในหน่วย byte
a_data	เก็บขนาดของ data segment ในหน่วย byte
a_bss	เก็บจำนวนของ byte ใน bss segment และถูกใช้โดย kernel เพื่อเซตค่า initial break หลัง data segment
a_syms	เก็บขนาดของ symbol table ในหน่วย byte
a_entry	เก็บ address ในหน่วยความจำของ entry point ของโปรแกรมหลังจากที่ kernel ได้โหลดแล้ว
a_trsize	เก็บขนาดของ text relocation table ในหน่วย byte
a_drsize	เก็บขนาดของ data relocation table ในหน่วย byte

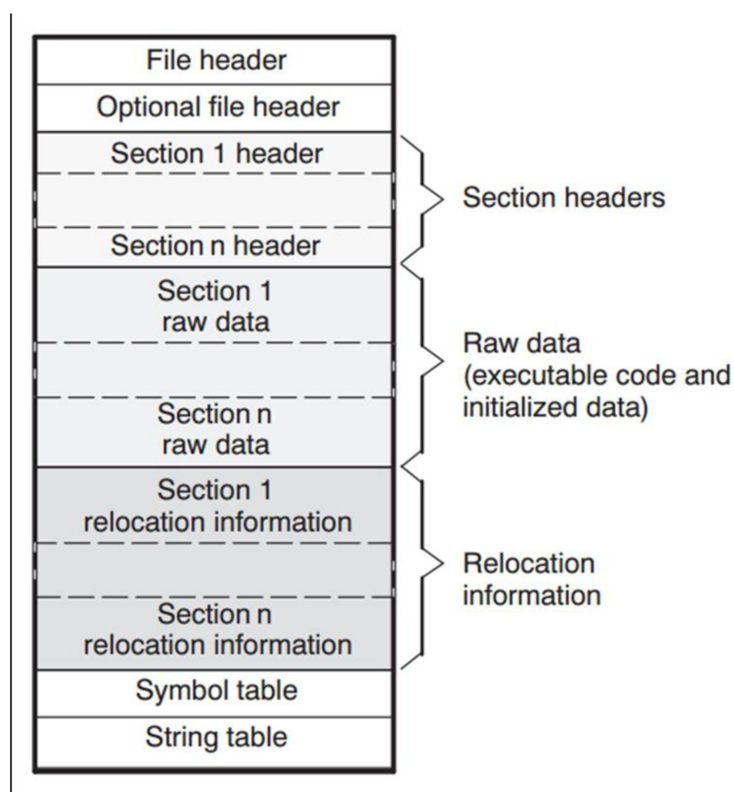
text segment เก็บ machine code และข้อมูลที่เกี่ยวข้องที่จะถูกโหลดเข้ามาในหน่วยความจำเมื่อโปรแกรมรัน โดยการโหลดอาจจะเป็นแบบอ่านอย่างเดียว
data segment เก็บข้อมูลที่เซตไว้ตอนเริ่มต้น โดยการโหลดเข้ามาในหน่วยความจำนั้นจะโหลดเข้ามาเป็นแบบอ่านเขียนได้เสมอ
text relocation เก็บ record ที่ถูกใช้โดย link editor เพื่อที่จะปรับปรุงข้อมูล pointer ใน text segment เมื่อมีการรวมไบนารี่ไฟล์
data relocation คล้ายกับ text relocation แต่ปรับปรุง pointer ใน data segment
symbol table เก็บ record ที่จะถูกใช้โดย link editor เพื่อทำการอ้างอิงข้าม address ของชื่อตัวแปรและฟังก์ชัน (symbol) ระหว่างไบนารี่ไฟล์ โดย symbol table จะเก็บใน Nlist ดังตารางด้านล่าง

ชื่อ	คำอธิบาย
n_strx	เก็บ offset ที่เข้าถึงชื่อ symbol ใน string table
n_type	ใช้โดย link editor เพื่อที่จะกำหนดว่าจะปรับปรุงค่า symbol อย่างไร
n_other	ให้ข้อมูลบนธรรมชาติของ symbol ซึ่งเป็นอิสระจากตำแหน่งของ symbol ในบริบทของ segment โดยข้อมูลดังกล่าวถูกกำหนดโดยฟิลด์ n_type
n_desc	สงวนไว้สำหรับใช้โดย debugger
n_value	เก็บค่าของ symbol (ถ้าเป็นข้อความ/ข้อมูล/bss ฟิลด์นี้เก็บ address,สำหรับ symbol อื่นๆ ค่าของ symbol จะเป็นอะไรก็ได้

string table เก็บ string ที่สอดคล้องกับชื่อ symbol

Common Object File Format (COFF)

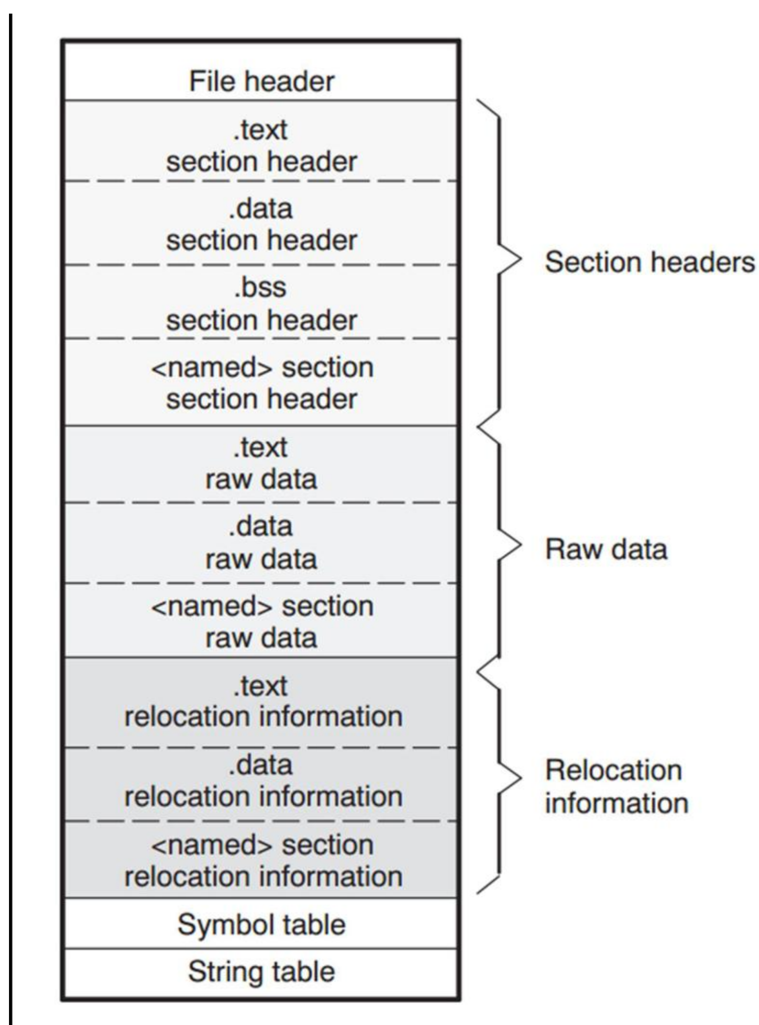
เป็นไฟล์ฟอแมตเหมือนกับ a.out ใช้ใน UNIX แต่ถูกเริ่มนำมาใช้ใน UNIX System V ซึ่งนำมาแทนไฟล์ฟอแมตแบบเก่าซึ่งก็คือ a.out นั่นเองและยังมีรูปแบบที่แยกไปตามคุณสมบัติอีกก็คือ ECOFF , XCOFF ก่อนที่จะถูกแทนที่ใน SVR 4 ด้วย ELF (Executable and linkable format) ก็มีการนำไปใช้งานบน Unix-like ,Microsoft windows และ EFI



COFF file structure

ส่วนประกอบของ COFF ออบเจคไฟล์ได้แบ่งเป็นเซคชัน ซึ่งประกอบไปด้วย

- A File header
- Optional header information
- A table of section headers
- Raw data for each initialized section
- Relocation information for each initialized section
- A symbol table
- A string table



Sample COFF Object File

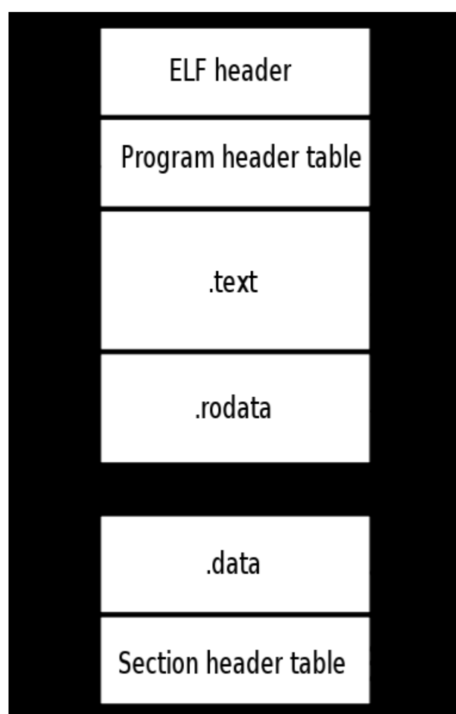
ELF

ELF (Executable and Linkable Format) เป็นรูปแบบมาตรฐานของ executable, object code, shared library และ core dump เผยแพร่ครั้งแรกใน System V Release 4 ได้รับการยอมรับจากหลายๆรุ่นของระบบปฏิบัติการ Unix และในปี 1999 ก็ได้รับเลือกให้เป็นมาตรฐานของ Unix ถูกออกแบบให้มีความยืดหยุ่นสูงและไม่ผูกกับ processor หรือ สถาปัตยกรรมใดซึ่งทาให้ถูกนาไปใช้บนระบบปฏิบัติการหลายระบบ

ELF เกิดจาก header ELF 1 header ตามด้วยข้อมูลไฟล์ดังต่อไปนี้

- ✓ Program header table สามารถมีกี่ segment ก็ได้
- ✓ Section header table สามารภมีกี่ section ก็ได้
- ✓ ข้อมูลที่อ้างถึง entry หลายๆ entry ใน header ของโปรแกรม หรือ header ของ section

ELF แต่ละ Segment จะประกอบไปด้วยไฟล์ที่จำเป็นต่อการ execution ส่วน section ประกอบไปด้วยข้อมูลสำคัญสำหรับทำการเชื่อมต่อและการเปลี่ยนตำแหน่ง Byte ใดๆใน ELF จะมี section เป็นเจ้าของได้ไม่เกิน 1 section



ELF header จะประกอบไปด้วยโครงสร้างดังนี้

```
typedef struct {
unsigned char e_ident[EI_NIDENT];
Elf32_Half e_type;
Elf32_Half e_machine;
Elf32_Word e_version;
Elf32_Addr e_entry;
Elf32_Off e_phoff;
Elf32_Off e_shoff;
Elf32_Word e_flags;
Elf32_Half e_ehsize;
Elf32_Half e_phentsize;
Elf32_Half e_phnum;
Elf32_Half e_shentsize;
Elf32_Half e_shnum;
Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

e_ident Byte เริ่มต้นของ ELF file

e_type บอกชนิดของ object file

e_machine ระบุ architecture สาหรับ file

e_version ระบุ version ของ object file

e_entry เก็บ virtual address ของ entry point

e_phoff เก็บ program header table

e_shoff เก็บ section header table

e_flags flag

e_ehsize ขนาดของ ELF header

e_phentsize ระบุขนาดของแต่ละ entry ใน program header table

e_phnum จานวน entry ภายใน program header table

e_shentsize ระบุขนาดของแต่ละ section ใน section header table

e_shnum จานวน section ภายใน section header table

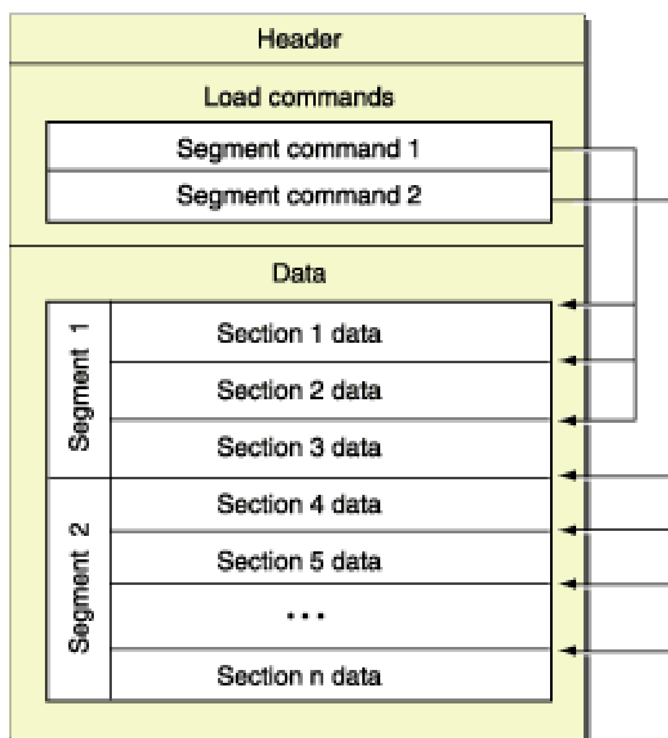
e_shstrndx ดัชนีของ section header table

Mach-O

หรือ Mach object ไฟล์ชนิดนี้ได้มีการนำมาแทนรูปแบบ a.out โดย Mach-O สามารถขยายขีดจำกัดของสิ่งที่ทำได้มากกว่าเดิมและเข้าถึง symbol table เร็วกว่า

Mach-O ถูกใช้เป็นครั้งแรกในระบบปฏิบัติการที่ใช้ Mach kernel ซึ้ง NeXTSTEP,OS X และ iOS เป็นตัวอย่างของระบบปฏิบัติการที่ใช้รูปแบบไฟล์นี้

Format



Mach-O ประกอบไปด้วยส่วนหลักๆได้แก่

- ✓ **Header** : เอาไว้บอกว่าไฟล์นี้เป็นไฟล์ Mach-O และประกอบไปด้วยข้อมูลชนิดไฟล์,สถาปัตยกรรม และ flag
- ✓ **Load commands :** เป็นส่วนที่ถัดออกมาจาก header เป็นลาดับของคาสั่ง load ต่างๆ ที่จะระบุเค้าโครงและคุณลักษณะการเชื่อมต่อของไฟล์ ตลอดจนข้อมูลอื่นๆ

- ✓ เค้าโครงเริ่มต้นของไฟล์ใน virtual memory
- ✓ ตำแหน่งของ symbol table (ใช้สาหรับการเชื่อมต่อแบบไดนามิก)
- ✓ ค่าเริ่มต้นของการทางานของ main thread
- ✓ ชื่อของ shared libraries

Specifications

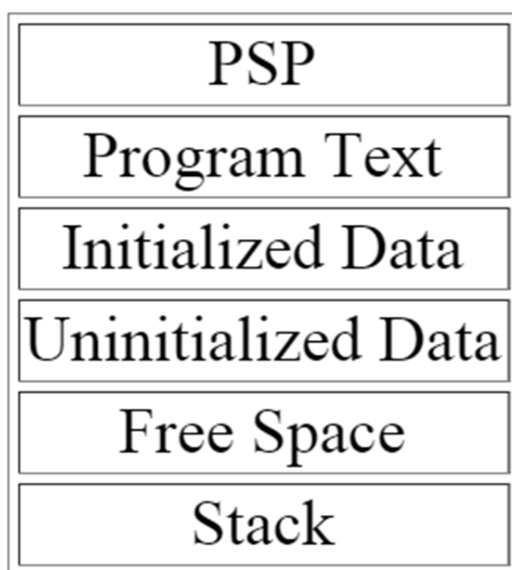
- ✓ การเรียกใช้ Symbol table โดยมี load commands ในการระบุขนาดละตาแหน่งของ Symbol table
- ✓ Relocation หรือก็คือกระบวนการในการย้าย symbols บางอย่างไปยัง address ใหม่ซึ่งจะทาให้คาสั่งที่อ้างอิงไปยัง address เก่าผิดพลาดได้ จึงต้องมีการจัดเก็บรายการการ relocation ไว้ใน Mach-O
- ✓ Multi-architecture binaries ภายใต้ระบบปฏิบัติการ NeXTSTEP, OPENSTEP, OS X, และ iOS ทาให้ code รองรับ instruction set ที่ต่างกันได้

Mach-O มีโครงสร้างครบทุกอย่างตามที่ไฟล์ปัจจุบันมี เช่น header, section, segment แต่ออกแบบมาเพื่อรองรับกับ NeXTSTEP, OPENSTEP, OS X, และ iO เท่านั้น จึงส่งผลให้มีความเหมาะสมต่อการใช้งานกับ NeXTSTEP, OPENSTEP, OS X, และ iO ในปัจจุบัน 18

.COM

ไฟล์ .COM ประกอบด้วย executable code และข้อมูล โดยมีขนาดไม่เกิน 64 KB ตัวอย่างการดำเนินการ เมื่อไฟล์ X.COM จะถูกดำเนินการ (โดยการพิมพ์ทั้ง X.COM บน DOS prompt) เนื้อหาทั้งหมดในไฟล์จะถูกโหลดไปยังmemory และเมื่อเนื้อหาในไฟล์ถูกโหลดไปจนครบจะเริ่มทำการดำเนินการตั้งแต่ไบต์แรกของไฟล์ ใน 256 ไบต์แรกของsegment จะเป็น Program Segment Prefix (PSP)) ซึ่งจะประกอบไปด้วยข้อมูลมากมายหลายประเภท ที่เกี่ยวกับการดำเนินการโปรแกรมส่วนที่มีประโยชน์ที่สุดในส่วนของ PSP คือท้ายของคำสั่ง เช่น หากเราทำการดำเนินการไฟล์ X.COM ด้วยการพิมพ์คำสั่ง X/full C:\Temp แล้ว สตริง /full C:\Temp จะถูกเก็บอยู่ใน PSP โปรแกรมสามารถเข้าถึงสตริงค์ส่วนนี้ตั้งแต่ offsetที่ 80h ไบต์แรกจะเป็นความยาวของส่วนหาง

โครงสร้างของไฟล์ประเภท .COM ในหน่วยความจำมีดังนี้



.EXE (MZ)

ไฟล์ประเภท DOS MZ จะเป็น ไฟล์ที่สามารถรันได้ โดยจะใช้สำหรับ .EXE ใน DOS ไฟล์จะสามารถระบุได้ว่าเป็นไฟล์ประเภทนี้โดย ดูที่ ASCII สตริง “MZ” (เลขฐานสิบหก คือ 4D 5A) ที่จุดเริ่มต้นของไฟล์ ไฟล์ประเภทนี้จะมาทีหลังไฟล์นามสกุล .COM และมีความแตกต่างกันที่ไฟล์ DOS นี้จะมี header ที่ทำการเก็บการย้ายที่ของข้อมูล ซึ่งจะอนุญาตให้segment หลายๆ segment สามารถโหลดได้โดย arbitrary memory addresses และจะสนับสนุนไฟล์ที่มีขนาดมากกว่า 64KB ด้วย มีส่วนของ stub เพื่อแสดงข้อความในกรณีที่ไม่สามารถดำเนินการได้ไว้บอกผู้ใช้

โครงสร้างของ EXE (MZ)

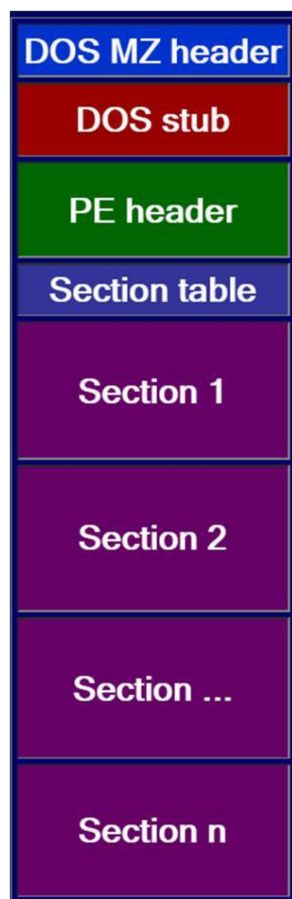
ไฟล์ประเภทนี้จะประกอบไปด้วย 2 โครงสร้างคือ header และ relocation ส่วนของ header จะมีส่วนประกอบดังต่อไปนี้

Offset	Field	ประเภท	คำอธิบาย
0	Signature	word	ตัวระบุวาเปนไฟล MZ โดยเก็บคา 0x5A4D ไว
2	Extra bytes	word	จำนวนของไบตใน page สุดทาย
4	Page	word	จำนวนของ page ทั้งหมด และ page บางสวน
6	Relocation item	word	จำนวนของ entry ในตารางการยายขอมูล
8	Header size	word	จำนวนของยอหนาทีขึ้นโดย header
10	Minimum allocation	word	จำนวนยอหนาที่ตองการโดยโปรแกรม โดยไมรวม PSP และโปรแกรมอิมเมจ ถาไมมี block วางพอ การโหลดจะหยุด
12	Maximum allocation	word	จำนวนของยอหนาที่รองขอโดยโปรแกรม ถาไมมี block วางที่ใหญพอ ยอหนาที่ใหญที่สุดจะไดจองไป
14	Initial SS	word	ตำแหนง segment ที่มีการเปลี่ยนตำแหนงไดสำหรับ SS
16	Initial SP	word	คาเริ่มตนของ SP
18	Checksum	word	เมื่อเพิ่มผลรวมไปยัง word ทั้งหมดในไฟล ผลลัพธที่ไดควรเปน 0
20	Initial IP	word	คาเริ่มตนของ IP
22	Initial CS	word	ตำแหนง segment ที่มีการเปลี่ยนตำแหนงไดสำหรับ CS
24	ตารางการเปลี่ยนตำแหนง	word	เก็บคา offset สัมบูรณไปยังตารางการเปลี่ยนตำแหนง
26	Overlay	word	คาที่ใชในการจัดการ overlay ถาเปน 0 แสดงวามันเปนการปฏิบัติงานหลัก
28	Overlay info.	UNKNOWN	ไฟล MZ บางครั้งอาจเก็บขอมูลเพิ่มเติมเกี่ยวกับการจัดการ overlay ของโปรแกรมหลัก

.EXE (PE)

ไฟล์ประเภท Portable Executable (PE) คือ ไฟล์ประเภทที่ใช้สำหรับรัน FON Font files , DLLs, , object code และ การใช้งานอื่นๆ ใน ระบบปฏิบัติการ Windows โดยไฟล์ประเภท PE เป็นโครงสร้างของข้อมูลที่ทำการห่อหุ้มข้อมูลทั้งหมดของโค้ดในส่วนที่สามารถรันได้ ซึ่งจะรวมไปถึง dynamic library references for linking, I/O ของ API,ข้อมูลแบบ thread-local storage (TLS) และ การจัดการทรัพยากรข้อมูลในระบบปฏิบัติการ Window ไฟล์ประเภท PE จะถูกใช้สำหรับไฟล์ประเภท DLL,EXE, SYS(device driver) และไฟล์ประเภทอื่นๆ โดย Extensible Firmware Interface (EFI) ได้ระบุไว้ว่า ไฟล์ประเภท PE คือ ไฟล์ที่สามารถรันได้ที่เป็นมาตรฐานในระบบปฏิบัติการ Window ไฟล์ประเภท PE ณ ปัจจุบัน สนับสนุนการทำงานของ IA-32, IA-64, x86-64 (AMD64/Intel64) และ ชุดคำสั่งแบบ ARM (ISAs) คำว่า portable executable หมายถึง ไฟล์ประเภทนี้จะสามารถทำงานข้าม win32 แพลตฟอร์มได้

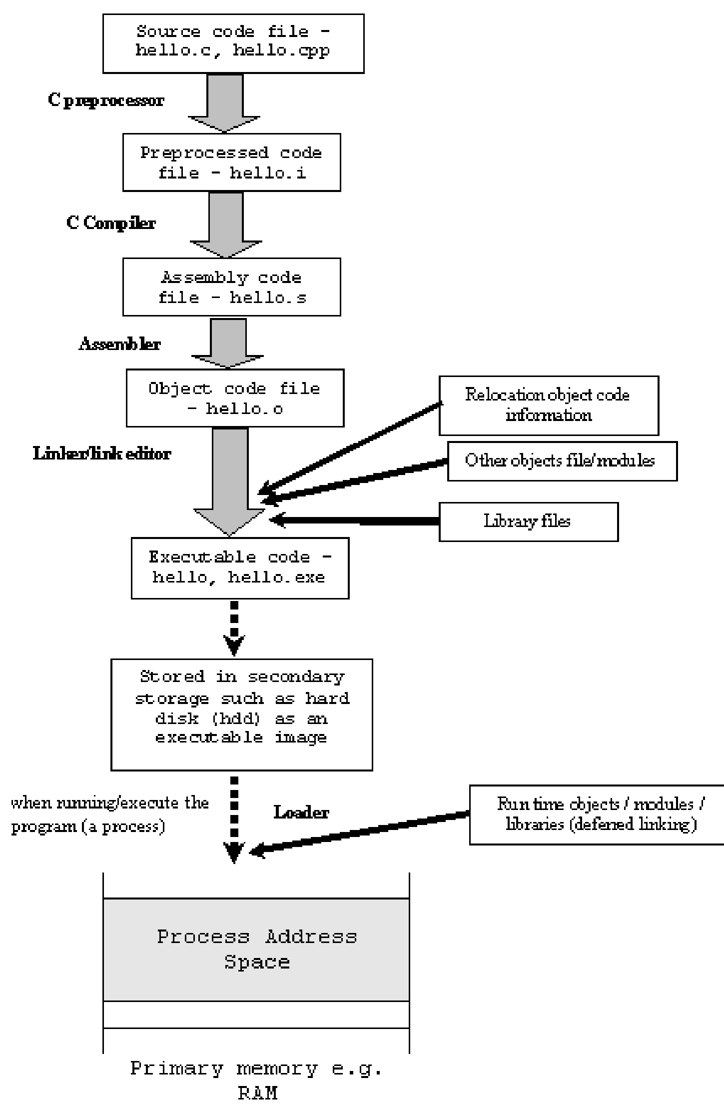
องค์ประกอบ EXE (PE)



องค์ประกอบของ ไฟล์ประเภท PE ทุกๆไฟล์ จะต้องเริ่มต้นด้วย DOS MZ เฮดเดอร์ ส่วนของ DOS stub จะเป็นส่วนที่เป็น EXE ที่แท้จริงที่จะถูกดำเนินการเมื่อระบบปฏิบัติการไม่รู้จักไฟล์ประเภท PE ซึ่งมันก็จะสามารถแสดงข้อความง่ายๆไว้บอกผู้ใช้ได้ว่า โปรแกรมนี้ต้องการระบบปฏิบัติการ Windows ในการใช้งาน ต่อมาคือส่วนของ PE header ในส่วนนี้จะเก็บข้อมูลที่สำคัญที่จะถูกใช้ใน PE loader ส่วนที่เป็นเนื้อหาจริงๆจะถูกแบ่งและเก็บอยู่ในส่วนที่เรียกว่า section ซึ่งในส่วนนี้ จะสามารถเป็นได้ทั้ง code หรือ เป็นข้อมูลก็ได้

กลไกต่างๆของ compiler

การนำไฟล์ซอร์สโค้ดภาษาระดับสูงมาแปลงเป็นไฟล์ออปเจ็กต์โค้ด



การทำงานของ compiler ในภาษา C

การนำไฟล์ซอร์สโค้ดภาษาระดับสูงมาแปลงเป็นไฟล์ออปเจ็กต์โค้ด มี 2 ขั้นตอนใหญ่ๆคือ

Compilation

กระบวนการคอมไพล์นั้นแบ่งเป็น 2 ส่วนคือ front end และ back end

Front end

ส่วนนี้คอมไพล์เลอร์จะทำการวิเคราะห์ source code เพื่อที่จะสร้างการนำเสนอภายในของโปรแกรมเราเรียกว่า intermediate representation เรียกสั้นๆว่า IR โดยมันจะจัดการตาราง symbol ซึ่งเป็นตารางที่แมพแต่ละ symbol ใน source code ไปยังข้อมูลที่เกี่ยวข้องเช่น ตำแหน่ง ชนิด และ ขอบเขต โดย phase ของการทำงานใน front end สามารถมี phase ต่างๆได้ดังนี้

- Line reconstruction ภาษาโปรแกรมที่อนุญาตให้ตั้งชื่อตัวแปรซ้ำกับ keyword ได้ต้องการ phase นี้ก่อนที่จะเริ่มทำ 3 phase หลักอื่นๆ โดยการทำงานคร่าวๆก็คือจะแปลงลำดับของตัวอักษรที่ต่อเนื่องกันให้เป็นรูปแบบที่ parser ยอมรับได้ Top-down,recursive-descent,table-driven parser เป็นสิ่งที่ใช้ในช่วงทศวรรษ 1960 โดยทำการอ่านตัวอักษรที่ละตัวและไม่ต้องการทำงานที่ต้องมาแยก phase ย่อยเพื่อระบุเป็นtoken อีก

- Lexical analysis จะทำการแบ่ง source code ให้อยู่ในข้อมูลชิ้นเล็กๆ เรียกว่า token แต่ละ token จะเป็นหน่วยเล็กที่สุดของภาษาโปรแกรมนั้นๆแล้ว เช่น keyword,ชื่อตัวแปร วิธีการระบุว่า input ที่รับเข้ามามันจะเป็น token อะไรนั้น ใช้ finite automata ซึ่งสร้างจาก regular expression โปรแกรมที่ทำหน้าที่นี้คือ lexical analyzer/scanner

- Preprocessing บางภาษาเช่น C ต้องการ phase นี้ซึ่งจะรองรับการแทนที่ของ macro และการคอมไพล์แบบมีเงื่อนไข โดยทั่วไปแล้ว phase นี้จะเกิดขึ้นก่อนการทำ syntax analysis และ semantic analysis

- Syntax analysis เกี่ยวข้องกับกระบวนการ parsing ลำดับของ token ที่ได้เพื่อระบุโครงสร้างเชิง syntax ของตัวโปรแกรม โดยทั่วไปจะทำการสร้าง parse tree มาซึ่งใช้แนวคิดของการแทนที่ลำดับของ input/token ที่เป็นเส้นตรง ด้วยโครงสร้าง tree ที่เข้าใจง่ายกว่า โดยหลักการคือ ถ้าสามารถหา tree ให้มันได้ แสดงว่า source code เขียนถูกต้องตาม syntax

- Semantic analysis เป็น phase ที่คอมไพล์เลอร์จะเพิ่มข้อมูลการตีความหมายเข้าไปใน parse tree ที่ได้จาก phase syntax analysis และทำการ build ตาราง symbol ภายใน phase นี้จะดำเนินการตรวจสอบต่างๆ เช่น การเช็ค type การเชื่อมโยงของ object ต่างๆ การทำ definite assignment(เช็คว่าตัวแปรทุกตัวต้องมีการเซ็ตค่าก่อนนำไปใช้งาน หากการตรวจสอบใดผิดพลาดก็ให้แจ้งกลับไปว่า error หรือ warning semantic analysis ต้องการ parse tree ที่สมบูรณ์จาก phase ก่อนหน้า

Back end

Phase การทำงานหลักๆใน back end มีดังนี้

- Analysis phase นี้เป็นการนำ IR ที่ได้จาก back end data-flow analysis ใช้ในการสร้าง use-define chains ไปพร้อมๆกับการทำ dependence analysis, alias analysis, pointer analysis, escape analysis และอื่นๆ การวิเคราะห์ที่แม่นยำเป็นพื้นฐานของการออปติไมซ์ในคอมไพล์เลอร์ใดๆ call graph และ control flow graph ก็ถูกสร้างในระหว่าง phase นี้เช่นกัน

- Optimization ทำการนำ IR มาแปลงให้อยู่ในรูปที่สมมูลกันแต่ทำงานเร็วกว่าเดิมหรือ IR อยู่ในรูปที่เล็กกว่าเดิม กระบวนการออปติไมซ์จะยกมาอธิบายอีกครั้งในหัวข้อต่อๆไป

- Code generation ใน phase นี้ IR ที่ได้จาก front end จะถูกแปลงให้อยู่ในภาษา assembly เพื่อส่งต่อให้กับ assembly ไปแปลงเป็นภาษาเครื่องต่อไป

Assembly

ในขั้นตอนนี้ จะนำไฟล์นามสกุล .s ที่ได้จากขั้นตอนการคอมไพล์มาแปลงให้เป็น object file หรือ object code ถ้าใช้ compiler ภาษา C เราจะได้ไฟล์นามสกุล .o ซึ่งไฟล์นามสกุล .o นี้จะเก็บคำสั่งระดับเครื่องไว้ เนื่องจากผลลัพธ์ของขั้นตอนนี้จะได้เป็นภาษาเครื่องดังนั้น หากเราลองเปิดไฟล์นามสกุล .o นี้ดูจะพบว่าเป็นภาษาที่มนุษย์ไม่สามารถอ่านได้ดังภาพด้านล่าง

การเพิ่มข้อมูลสำหรับดีบัก

Bug คือข้อผิดพลาดที่ทำให้โปรแกรมทำงานไม่เป็นไปตามที่โปรแกรมเมอร์สิ่งกำหนด การเกิด bug อาจจะส่งผลกระทบต่อโปรแกรมเล็กน้อย และอาจทำให้เกิด error ในบางฟังก์ชัน หรือถ้าร้ายแรงมากก็ทำให้โปรแกรม crash และปิดตัวลง เคยมีคนได้กล่าวไว้ว่า “โปรแกรมที่ไม่มี bug คือโปรแกรมที่ยังไม่ได้เขียน” ไม่ว่าจะเขียนโปรแกรมง่ายหรือยาก ต่อให้เป็นโปรแกรมเมอร์เก่งแค่ไหนก็ต้องเขียนโปรแกรมผิดพลาดบ้าง ซึ่งถือเป็นเรื่องปกติธรรมดา เมื่อเกิด bug สิ่งที่โปรแกรมเมอร์ต้องทำก็คือการหาข้อผิดพลาดและแก้ไขหรือที่เรียกว่า Debug ทักษณะนี้จำเป็นอย่างยิ่งที่โปรแกรมเมอร์จะต้องเรียนรู้ และฝึกฝนให้ชำนาญ ยิ่งมีประสบการณ์ debug มากเท่าไหร่ ก็ยิ่งได้เปรียบมากเท่านั้น เพราะในบางครั้งเวลาที่ใช้สำหรับการ debug นั้นมากกว่าเวลาที่ใช้เขียนโปรแกรมเสียอีก เนื้อหาในบทนี้จะอธิบายถึงการใช้ debugger รวมถึงเทคนิควิธีการขั้นสูงที่ใช้ในการ debug เช่น exception breakpoint

Breakpoint

การดีบั๊กนั้นสามารถทำได้หลายวิธีหนึ่งในวิธีง่ายที่สุดคือการใช้ NSLog เพื่อแสดงค่าของตัวแปรที่ต้องการ หากโปรแกรมง่ายไม่ซับซ้อน ก็อาจจะใช้ NSLog ได้ แต่เมื่อโปรแกรมใหญ่และซับซ้อนขึ้น การใช้ NSLog อย่างเดียวนั้นไม่เพียงพอต่อการแก้ไข bug ที่เกิดขึ้นได้ การใช้ debugger เป็นทางออกที่ดีกว่าการใช้ NSLog เพราะดีบั๊กเกอร์สามารถวาง breakpoint เพื่อให้โปรแกรมหยุดทำงานชั่วขณะ จากนั้นก็จะสามารถดูค่าของตัวแปรต่างในขณะนั้นได้ทันที การกำหนด breakpoint นั้นสามารถทำได้ง่ายๆ ด้วยการคลิ๊กที่ด้านหน้าของบรรทัดที่ต้องการ จากนั้นก็จะเห็นสัญลักษณ์ ลูกศรสีฟ้าเข้ม

```
 9        int x = 10;
10        int y = 20;
11        int z = 0;
12        int square = 0;
13        int sum = 0;
14
15        z = x + y;
16
17        square = [Math square:x];
18        sum = [Math sum:x and:y];
19
```

Exception Breakpoints

นอกจากการวาง breakpoint แบบปกติทั่วไป ที่ได้อธิบายไป ยังมีเบรกพ้อนต์แบบพิเศษที่เรียกว่า Exception Breakpoint ให้ลองพิจารณาโปรแกรมต่อไปนี้

```
NSMutableArray* list = [NSMutableArray array];
NSString* a = @"Hello";
NSString* b = nil;

[list addObject:a];
[list addObject:b];
```

เมื่อสั่งให้ทำงาน โปรแกรมจะเกิดข้อผิดพลาดขึ้นและปิดตัวลง เพราะอ็อบเจ็ก b มีค่าเป็น nil ทำให้เมธอด addObject: นั้นทำงานผิดพลาด และ debug console จะแจ้งข้อผิดพลาดดังนี้

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason:
'*** -[__NSArrayM insertObject:atIndex:]: object cannot be nil'
*** First throw call stack:
(
0   CoreFoundation    0x00007fff84e9141c __exceptionPreprocess + 172
1   libobjc.A.dylib   0x00007fff84fe3e75 objc_exception_throw + 43
2   CoreFoundation    0x00007fff84d528b7 -[__NSArrayM insertObject:atIndex:] + 951
```

3 Program 20.1 0x0000000100000ec1 main + 177

4 libdyld.dylib 0x00007fff898985fd start + 1

)

libc++abi.dylib: terminating with uncaught exception of type NSException

การออปติไมซ์

การออปติไมซ์ คือกระบวนการที่คอมไพล์เลอร์พยายามที่จะลดหรือเพิ่มคุณสมบัติบางอย่างของโปรแกรมให้ถึงที่สุด ส่วนมากจะนิยมทำเพื่อลดเวลาที่ใช้ในการรันโปรแกรม บางส่วนก็ทำเพื่อลดปริมาณของหน่วยความจำที่ใช้ การออปติไมซ์โดยทั่วไปแล้วสามารถสร้างกลไกได้โดยใช้ลำดับของการแปลงออปติไมซ์ (Optimizing transformations) ซึ่งเป็นอัลกอริทึมที่จะนำโปรแกรมเข้ามาและแปลงมันให้อยู่ในรูปของ output โปรแกรมที่สมมูลกันซึ่งมีการใช้ทรัพยากรที่น้อยกว่า

หลักการทำงานโดยทั่วไปของการออปติไมซ์

1. Optimize the common case คือการออปติไมซ์กรณีที่เกิดขึ้นบ่อยที่สุดก่อน ซึ่งจะเป็นผลลัพธ์ที่ทำให้เกิดประสิทธิภาพในตอนท้ายที่ดีกว่า

2. Avoid redundancy ใช้ผลลัพธ์ที่มีการคำนวณไว้ก่อนแล้วและเก็บมันเอาไว้ใช้ต่อ โดยที่ไม่ต้องคำนวณใหม่ซ้ำ

3. Less code เอาการคำนวณที่ไม่จำเป็นและค่าคงที่ออก ทำให้ CPU และหน่วยความจำทำงานน้อยลงและเร็วขึ้น

4. Fewer jumps ใช้การ Branch ให้น้อยที่สุด เพราะเวลาโปรแกรมทำงานจะมีการ prefetch คำสั่งที่จะถูกทำไว้ล่วงหน้า เป็นผลทำให้เวลาในการทำงานช้าลง การใช้คำสั่ง inline หรือการ loop โดยไม่มีการม้วนจะช่วยลดการกระโดดได้

5. Locality โค้ดและข้อมูลที่สามารถเข้าถึงด้วยกันได้ในเวลาเดียวกันควรที่เก็บอยู่ในหน่วยความจำเดียวกันเพื่อเพิ่มการอ้างอิงของ spatial locality

6.Exploit the memory hierarchy พยายามนำข้อมูล/คำสั่งที่ใช้บ่อยๆไปเก็บไว้ใน register ให้ได้ ถ้าเต็มแล้วก็นำไปเก็บไว้ในแคชตามลำดับ

7. Parallelize ให้สามารถทำงานแบบขนานได้ ทั้งในระดับคำสั่ง/หน่วยความจำและระดับ thread

8. More precise information is better คอมไพล์เลอร์มีประสิทธิภาพมาก ก็จะส่งผลให้การออปติไมซ์ดีขึ้น

9. Runtime metrics can help ข้อมูลที่ถูกเก็บในระหว่างการทดสอบการรันสามารถที่จะใช้ในการออปติไมซ์แบบมีโปรไฟล์ได้ ข้อมูลที่ถูกเก็บในระหว่างการรันสามารถที่จะใช้โดยคอมไพล์เลอร์ JIT เพื่อที่จะเพิ่มประสิทธิภาพในการออปติไมซ์แบบไดนามิก

10.Strength reduction ลดความซ้ำซ้อน เช่น แทนที่การหารด้วยการคูณเลขกับส่วนกลับ (1/n) หรือใช้การวิเคราะห์ตัวแปรอุปนัยเพื่อที่จะแทนที่การคูณโดยใช้ loop ของการบวกแทน

การเชื่อมไปไลบรารี่ทั้งแบบสแตติกและแบบไดนามิก

การเชื่อมแบบสแตติก

การเชื่อมแบบสแตติกเป็นกระบวนการของการคัดลอกโค้ดของไลบรารี่ทั้งหมดที่ใช้ในโปรแกรมเข้าไปในอิมเมจที่ปฏิบัติงานได้ กระบวนการนี้ทำโดย linker และเป็นกระบวนการสุดท้ายของการ compile ตัว linker จะรวมโค้ดใน library กับโค้ดโปรแกรมที่มีอยู่ในการที่จะแก้การอ้างอิงภายนอก และเพื่อที่จะสร้างอิมเมจที่สามารถปฏิบัติงานได้เหมาะกับการโหลดเข้าไปทำงานในหน่วยความจำ

ไลบรารี่แบบสแตติก

ไลบรารี่แบบสแตติกเป็นคอลเลคชันของไฟล์ไบนารี่ object ที่ช่วยในระหว่างการเชื่อม โดยทั่วไปแล้วไลบรารี่นั้นจะประกอบไปด้วย object ไฟล์จำนวนมากๆ ประโยชน์ก็คือในกรณีที่เราต้องการสร้างโปรแกรมที่ปฏิบัติงานได้แล้วเรามี object ไฟล์จำนวนมากที่จะต้องเชื่อมเข้าด้วยกัน สำหรับไลบรารี่แบบสแตติกนั้นจะเป็นก้อนของ object ไฟล์ที่มีการเปลี่ยนตำแหน่งได้

การเชื่อมแบบไดนามิก

ไลบรารี่การเชื่อมโยงแบบไดนามิกเป็นรูปแบบที่หายากของไลบรารีการเชื่อมโยงแบบไดนามิก, DLL เป็นโปรแกรมที่มีใช้งานร่วมกันโดยรหัสหลายและห้องสมุดข้อมูล DLL ที่ไม่ได้เป็นแฟ้มที่ปฏิบัติการเชื่อมโยงแบบไดนามิกให้วิธีที่จะทำให้กระบวนการสามารถเรียกใช้ไม่ได้อยู่ในรหัสปฏิบัติการของฟังก์ชั่น รหัสปฏิบัติการฟังก์ชันใน DLL และ DLL ที่มีมากกว่าหนึ่งได้รับการรวบรวมและเชื่อมโยงกับการใช้งานของกระบวนการของการทำงานของพวกเขาจัดเก็บแยกกัน DLL ยังช่วยในการแบ่งปันข้อมูลและทรัพยากร การใช้งานหลายคนพร้อมกันสามารถเข้าถึงสำเนาเดียวของ DLL ในเนื้อหาของหน่วยความจำ DLL เป็นโปรแกรมที่มีใช้งานร่วมกันโดยหลายรหัสข้อมูลและห้องสมุด มีประโยชน์ช่วยในการลดขนาดของไฟล์ปฏิบัติงานได้ dll ก็ยังมีข้อดีของการแบ่งปันการใช้ไลบรารี่ให้กับโปรแกรมหลายๆโปรแกรมได้ การเชื่อมมันกับโปรแกรมหลายๆโปรแกรมเลยเรียกว่าการเชื่อมแบบไดนามิก ไลบรารี่ที่ใช้ร่วมกันถูกโหลดเข้าไปในหน่วยความจำโดยโปรแกรมเมื่อเริ่มต้นทำงาน เมื่อไลบรารี่ดังกล่าวโหลดครบแล้ว โปรแกรมที่เหลือที่เริ่มต้นทำงานทีหลังก็จะสามารถใช้ไลบรารี่นี้ในการทำงานได้ทันที

บรรณานุกรม

http://th.swewe.net/word_show.htm/?52314_1&ไลบรารี่การเชื่อมโยงแบบไดนามิก

https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/index.html http://en.wikipedia.org/wiki/Compiler

http://en.wikipedia.org/wiki/Optimizing_compiler

http://www.ti.com/lit/an/spraao8/spraao8.pdf

http://www.debuginfo.com/articles/gendebuginfo.html

http://cs-fundamentals.com/c-programming/static-and-dynamic-linking-in-c.php

http://win32assembly.programminghorizon.com/pe-tut1.html

http://jak-stik.ac.id/materi/Pemrograman_Sistem2/LectureNotes/sp12.ppt

http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

http://en.wikipedia.org/wiki/COFF

http://osr507doc.sco.com/en/topics/COFF.html

http://wiki.osdev.org/MZ

http://www.csc.depauw.edu/~bhoward/asmtut/asmtut11.html

http://en.wikipedia.org/wiki/Comparison_of_executable_file_formats

http://en.wikipedia.org/wiki/Mach-O

http://en.wikipedia.org/wiki/A.out

http://www.skyfree.org/linux/references/ELF_Format.pdf