```
#include <stdio.h>
#include <list.h>
#include <assert.h>
#include <slink.h>

#define VERSION "0.1"

static Elf_Section *    Pad( Elf_File *filePtr, Elf_Section *nextPtr,
                        Elf32_Addr addr, int size);
static void             SortSections(Elf_File *filePtr);
static void             SortSegments(Elf_File *filePtr);
static void             Usage(char *name);
static void             Debug(char *fmt,...);
static void             NukeSection(Elf_Section *sectionPtr);
static void             Relocate(Elf_File *, Elf_Section *);
static void             RelocateA(Elf_File *, Elf_Section *);
static void             Patch(Elf_File *, Elf32_Addr addr, Elf32_Sword value);

int
main(int argc, char **argv) {
    Status      status;
    List_Links  fileList;
    Elf_File    *filePtr;
    FILE        *f;
    int         i;
    Elf_Section *sectionPtr;
    Status      rs;
    Elf_File    *execPtr = NULL;
    int         fileOffset;
    Elf_Segment *segmentPtr;
    Elf_Segment *newSegmentPtr;
    Elf_Section *newSectionPtr;
    int         size;
    Elf_Section *stringPtr;
    int         index;
    char        *strings;
    int         padding;
    Elf_File    *outPtr;
    char        *outputFile = NULL;
    int         option;
    List_Links  *nextPtr;
    Boolean     strip = TRUE;
    char        template[] = "/tmp/slink.XXXXXX";
    Elf_Section *mappingPtr;
    int         fd;
    FILE        *stream;
    int         rc;
    Boolean     sectionMapping = FALSE;


    do {
        option = getopt(argc, argv, "o:VsSm");
        switch(option) {
            case 'o':
                outputFile = optarg;
                break;
            case 'V':
                printf("%s Version %s\n", argv[0], VERSION);
                exit(0);
                break;
            case 's':
                strip = TRUE;
```



```
                break;
            case 'S':
                strip = FALSE;
                break;
            case 'm':
                sectionMapping = TRUE;
                break;
            case -1:
                break;
            case '?':
            default:
                Usage(argv[0]);
        }
    } while(option != -1);

    if (outputFile == NULL) {
        fprintf(stderr, "No output file specified.\n");
        Usage(argv[0]);
    }
    List_Init(&fileList);
    for (i = optind; i < argc; i++) {
        Debug("Input file: %s\n", argv[i]);
        f = fopen(argv[i], "r");
        assume(f != NULL);
        filePtr = Elf_FileNew();
        filePtr->name = argv[i];
        status = Elf_FileRead(f, filePtr);
        if (status != SUCCESS) {
            fprintf(stderr, "Elf_FileRead failed: %s\n", Status_Msg(status));
            exit(1);
        }
        fclose(f);
        List_InitElement(&filePtr->links);
        List_Insert(&filePtr->links, LIST_ATREAR(&fileList));
    }

    /*
     * Create a new file that has the PT_LOAD segments and all sections fro
     * the input files.
     */
    outPtr = Elf_FileNew();
    filePtr->name = "output";
    LIST_FORALL(&fileList, (List_Links *) filePtr) {
        if (filePtr->ehdr.e_type == ET_EXEC) {
            if (execPtr != NULL) {
                fprintf(stderr, "Only one file of type ET_EXEC allowed\n");
                exit(1);
            }
            execPtr = filePtr;
        }
        LIST_FORALL(&filePtr->segmentList, (List_Links *) segmentPtr) {
            if (segmentPtr->phdr.p_type != PT_LOAD) {
                continue;
            }
            newSegmentPtr = Elf_SegmentNew();
            *newSegmentPtr = *segmentPtr;
            List_Insert(&newSegmentPtr->links,
                LIST_ATREAR(&outPtr->segmentList));
            outPtr->numSegments++;
        }
        LIST_FORALL(&filePtr->sectionList, (List_Links *) sectionPtr) {
            newSectionPtr = Elf_SectionNew();
```

```
            *newSectionPtr = *sectionPtr;
            List_Insert(&newSectionPtr->links,
                LIST_ATREAR(&outPtr->sectionList));
            outPtr->numSections++;
        }
    }
    if (execPtr == NULL) {
        fprintf(stderr, "You must specify one executable file.\n");
        Usage(argv[0]);
    }
    SortSegments(outPtr);
    SortSections(outPtr);
    /*
     * Perform all remaining relocations. Prelink has done all it could,
     * but some remain because they resolve to a larger scope than the
     * library that uses them (e.g. the library uses an initialized
     * global variable. The binary will have its own copy in its .dynbss
     * section that is initialized via an R_368_COPY relocation. The
     * library must refer to the binary's copy of the variable, but that
     * will be different for each binary that uses the library).
     */
    LIST_FORALL(&outPtr->sectionList, (List_Links *) sectionPtr) {
        if (((sectionPtr->shdr.sh_type == SHT_REL) ||
            (sectionPtr->shdr.sh_type == SHT_RELA)) &&
            (strcmp(sectionPtr->name, ".gnu.conflict") == 0)) {

            if (sectionPtr->shdr.sh_type == SHT_REL) {
                Relocate(outPtr, sectionPtr);
            } else {
                RelocateA(outPtr, sectionPtr);
            }
        }
    }

    if (strip) {
        /*
         *  Remove all unneeded sections.
         */
        LIST_FORALL_DEL(&outPtr->sectionList, (List_Links *) sectionPtr,
                nextPtr) {
            int delete;

            delete = 0;
            if ((sectionPtr->shdr.sh_type == SHT_DYNSYM) ||
                (sectionPtr->shdr.sh_type == SHT_REL) ||
                (sectionPtr->shdr.sh_type == SHT_RELA) ||
                (sectionPtr->shdr.sh_type == SHT_HASH) ||
                (sectionPtr->shdr.sh_type == SHT_NOTE) ||
                (sectionPtr->shdr.sh_type == SHT_NULL) ||
                (sectionPtr->shdr.sh_type == SHT_DYNAMIC)) {

                delete = 1;

            } else if (sectionPtr->shdr.sh_type == SHT_STRTAB)  {
                if ((strcmp(sectionPtr->name, ".dynstr") == 0) ||
                     (strcmp(sectionPtr->name, ".shstrtab") == 0)) {
                    delete = 1;
                }
            } else if ((strcmp(sectionPtr->name, ".gnu.prelink_undo") == 0) ||
                       (strcmp(sectionPtr->name, ".gnu.liblist") == 0) ||
                       (strcmp(sectionPtr->name, ".gnu.conflict") == 0) ||
                       (strcmp(sectionPtr->name, ".gnu.version") == 0) ||
```



```
                       (strcmp(sectionPtr->name, ".gnu.version_d") == 0) ||
                       (strcmp(sectionPtr->name, ".gnu.version_r") == 0) ||
                       (strcmp(sectionPtr->name, ".gnu.libstr") == 0) ||
                       (strcmp(sectionPtr->name, ".interp") == 0)) {
                    delete = 1;
            }
            if (delete) {
                List_Remove((List_Links *) sectionPtr);
                outPtr->numSections--;
            }
        }
    }
    /*
     * Adjust segment addresses to match the address of their first
     * section. This accounts for deleted sections and the program headers
     * at the beginning of the file. This code assumes that sections
     * are not deleted from the end of a segment.
     */
    LIST_FORALL(&outPtr->segmentList, (List_Links *) segmentPtr) {
        Debug("Aligning segment starting at 0x%x\n", segmentPtr->phdr.p_vaddr);
        LIST_FORALL(&outPtr->sectionList, (List_Links *) sectionPtr) {
            int diff;
            if (sectionPtr->shdr.sh_addr >= segmentPtr->phdr.p_vaddr) {
                Debug("Section \"%s\" is first in segment.\n",
                    sectionPtr->name);
                segmentPtr->sectionPtr = sectionPtr;
                diff = sectionPtr->shdr.sh_addr - segmentPtr->phdr.p_vaddr;
                if (diff != 0) {
                    segmentPtr->phdr.p_vaddr += diff;
                    segmentPtr->phdr.p_paddr += diff;
                    segmentPtr->phdr.p_memsz -= diff;
                    segmentPtr->phdr.p_filesz -= diff;
                }
                break;
            }
        }
    }
    fileOffset = outPtr->ehdr.e_ehsize +
        outPtr->numSegments * outPtr->ehdr.e_phentsize;
    LIST_FORALL(&outPtr->sectionList, (List_Links *) sectionPtr) {
        Debug("Section \"%s\" fileOffset %d (0x%x)\n", sectionPtr->name,
            fileOffset, fileOffset);
        if (sectionPtr->shdr.sh_addr != 0) {
            padding = PAGE_OFFSET(sectionPtr->shdr.sh_addr) -
                PAGE_OFFSET(fileOffset);
            if (padding < 0) {
                padding += PAGE_SIZE;
            }
            if (padding != 0) {
                Elf_Section *padPtr;

                padPtr = Pad(outPtr, sectionPtr, segmentPtr->phdr.p_vaddr,
                             padding);
                padPtr->shdr.sh_offset = fileOffset;
                fileOffset += padding;
            }
            assert(PAGE_OFFSET(sectionPtr->shdr.sh_addr) ==
                PAGE_OFFSET(fileOffset));
        }
        if (sectionPtr->shdr.sh_type != SHT_NULL) {
            sectionPtr->shdr.sh_offset = fileOffset;
        }
```

```
        if ((sectionPtr->shdr.sh_type != SHT_NULL) &&
            (sectionPtr->shdr.sh_type != SHT_NOBITS)) {
            fileOffset += sectionPtr->size;
        } else {
            Debug("Skipping section %s when computing file offsets.\n",
                sectionPtr->name);
        }
    }
    LIST_FORALL(&outPtr->segmentList, (List_Links *) segmentPtr) {
        segmentPtr->phdr.p_offset = segmentPtr->sectionPtr->shdr.sh_offset;
    }

    /*
     *  The first section must be a NULL section.
     */
    sectionPtr = Elf_SectionNew();
    sectionPtr->name = "";
    List_Insert((List_Links *) sectionPtr, LIST_ATFRONT(&outPtr->sectionList));
    outPtr->numSections++;
    /*
     * Add a section that maps section indices to file names, if necessary.
     */
    if (sectionMapping) {
        mappingPtr = Elf_SectionNew();
        mappingPtr->name = ".slink.mapping";
        mappingPtr->shdr.sh_type = SHT_PROGBITS;
        List_Insert(&mappingPtr->links, LIST_ATREAR(&outPtr->sectionList));
        outPtr->numSections++;
    }
    /*
     * Create section header string table.
     */
    stringPtr = Elf_SectionNew();
    stringPtr->name = ".shstrtab";
    stringPtr->shdr.sh_type = SHT_STRTAB;
    List_Insert(&stringPtr->links, LIST_ATREAR(&outPtr->sectionList));
    outPtr->numSections++;
    /*
     * Compute section indices.
     */
    i = 0;
    LIST_FORALL(&outPtr->sectionList, (List_Links *) sectionPtr) {
        sectionPtr->index = i++;
    }
    assert(i == outPtr->numSections);
    /*
     * Fill in the section to file mapping information.
     */
    if (sectionMapping) {
        fd = mkstemp(template);
        assume(fd != -1);
        unlink(template);
        stream = fdopen(fd, "w+");
        assume(stream != NULL);
        LIST_FORALL(&fileList, (List_Links *) filePtr) {
            fprintf(stream, "%s ", filePtr->name);
            LIST_FORALL(&outPtr->sectionList, (List_Links *) sectionPtr) {
                if (sectionPtr->filename != NULL) {
                    if (strcmp(filePtr->name, sectionPtr->filename) == 0) {
                        fprintf(stream, "%d ", sectionPtr->index);
                    }
                }
```



```
            }
            fprintf(stream, "\n");
        }
        fflush(stream);
        rc = fseek(stream, 0, SEEK_END);
        assume(rc == 0);
        sectionPtr->size = ftell(stream);
        sectionPtr->section = mmap(NULL, sectionPtr->size, PROT_WRITE,
                MAP_SHARED, fd, 0);
        assume(sectionPtr->section != NULL);
        mappingPtr->shdr.sh_size = sectionPtr->size;
        mappingPtr->shdr.sh_offset = fileOffset;
        fileOffset += sectionPtr->size;
    }
    /*
     * Fill in section header string table.
     */
    size = 1;
    LIST_FORALL(&outPtr->sectionList, (List_Links *) sectionPtr) {
        if (sectionPtr->shdr.sh_type != SHT_NULL) {
            size += strlen(sectionPtr->name) + 1;
        }
    }
    stringPtr->shdr.sh_size = size;
    stringPtr->shdr.sh_offset = fileOffset;
    Debug("string table at offset %d\n", fileOffset);
    stringPtr->section = xmalloc(size);
    fileOffset += size;
    Debug("string table size %d\n", size);
    strings = (char *) stringPtr->section;
    strings[0] = '\0';
    index = 1;
    LIST_FORALL(&outPtr->sectionList, (List_Links *) sectionPtr) {
        int len;
        if (sectionPtr->shdr.sh_type != SHT_NULL) {
            len = strlen(sectionPtr->name) + 1;
            assert(index + len <= size);
            strcpy(&strings[index], sectionPtr->name);
            sectionPtr->shdr.sh_name = index;
            index += len;
        } else {
            sectionPtr->shdr.sh_name = 0;
        }
    }
    outPtr->ehdr.e_shstrndx = outPtr->numSections - 1;
    outPtr->ehdr.e_shoff = fileOffset;
    outPtr->ehdr.e_entry = execPtr->ehdr.e_entry;
    outPtr->ehdr.e_phnum = outPtr->numSegments;
    outPtr->ehdr.e_shnum = outPtr->numSections;
    /*
     * Fix the sh_link and sh_info fields.
     */
    LIST_FORALL(&outPtr->sectionList, (List_Links *) sectionPtr) {
        Elf_Section *linkPtr;
        switch (sectionPtr->shdr.sh_type) {
            case SHT_REL:
            case SHT_RELA:
                rs = Elf_GetSectionByID(outPtr, sectionPtr->id.file,
                        sectionPtr->shdr.sh_info, &linkPtr);
                assume(rs == SUCCESS);
                sectionPtr->shdr.sh_info = linkPtr->index;
                Debug("Setting section %d sh_info to %d\n",
```

```
                    sectionPtr->index, sectionPtr->shdr.sh_info);
        }
        if (sectionPtr->shdr.sh_link != SHN_UNDEF) {
            rs = Elf_GetSectionByID(outPtr, sectionPtr->id.file,
                    sectionPtr->shdr.sh_link, &linkPtr);
            assume(rs == SUCCESS);
            sectionPtr->shdr.sh_link = linkPtr->index;
            Debug("Setting section %d sh_link to %d\n",
                sectionPtr->index, sectionPtr->shdr.sh_link);
        }
    }
    /*
     * Double-check that the section address and offset fields are aligned.
     */
    LIST_FORALL(&outPtr->sectionList, (List_Links *) sectionPtr) {
        if ((sectionPtr->shdr.sh_type != SHT_NULL) &&
            (sectionPtr->shdr.sh_type != SHT_NOBITS) &&
            (sectionPtr->shdr.sh_type != SHT_LOUSER) &&
            (sectionPtr->shdr.sh_addr != 0)) {
            if (PAGE_OFFSET(sectionPtr->shdr.sh_addr) !=
                PAGE_OFFSET(sectionPtr->shdr.sh_offset)) {
                fprintf(stderr,
                    "Section %d not aligned properly.\n", sectionPtr->index);
                Elf_FileDump(stderr, outPtr);
                abort();
            }
        }
    }
    if (outputFile != NULL) {
        f = fopen(outputFile, "w+");
        assume(f != NULL);
    } else {
        f = stdout;
    }
    rs = Elf_FileWrite(f, outPtr);
    assume(rs == SUCCESS);
    exit(0);
}

static Elf_Section *
Pad(
    Elf_File    *filePtr,
    Elf_Section *nextPtr,
    Elf32_Addr  addr,
    int         size)
{
    Elf_Section *padPtr;
    static char buffer[100];
    static int count = 0;

    padPtr = Elf_SectionNew();
    snprintf(buffer, 100, ".pad%d", count++);
    padPtr->name = strdup(buffer);
    padPtr->shdr.sh_type = SHT_LOUSER;
    padPtr->shdr.sh_addr = addr;
    padPtr->shdr.sh_size = size;
    padPtr->size = size;
    padPtr->section = NULL;
    List_Insert((List_Links *) padPtr,
        LIST_BEFORE((List_Links *) nextPtr));
    filePtr->numSections++;
    Debug("Added %d bytes of padding in section %s before section %s (%d:%d).\n",
```





```
        size, padPtr->name, nextPtr->name, nextPtr->id.file, nextPtr->id.in
    return padPtr;
}

static void
SortSections(
    Elf_File    *filePtr)
{
    Elf_Section *sectionPtr;
    Elf_Section *insertPtr;
    List_Links  *nextPtr;
    Elf_Section *prevPtr;
    List_Links  sortList;
    List_Links  saveList;
    int         before;
    int         after;

    List_Init(&sortList);
    List_Init(&saveList);

    before = 0;
    LIST_FORALL_DEL(&filePtr->sectionList, (List_Links *) sectionPtr, nextP
        List_Remove((List_Links *) sectionPtr);
        before++;
        if (sectionPtr->shdr.sh_addr == 0) {
            List_Insert((List_Links *) sectionPtr, LIST_ATREAR(&saveList));
        } else {
            List_Insert((List_Links *) sectionPtr, LIST_ATREAR(&sortList));
        }
    }
    assert(List_IsEmpty(&filePtr->sectionList));
    LIST_FORALL_DEL(&sortList, (List_Links *) sectionPtr, nextPtr) {
        List_Remove((List_Links *) sectionPtr);
        LIST_FORALL(&filePtr->sectionList, (List_Links *) insertPtr) {
            if (insertPtr->shdr.sh_addr > sectionPtr->shdr.sh_addr) {
                break;
            }
        }
        List_Insert((List_Links *) sectionPtr, LIST_BEFORE(insertPtr));
    }
    List_ListInsert(&saveList, LIST_ATREAR(&filePtr->sectionList));
    assert(List_IsEmpty(&sortList));
    prevPtr = NULL;
    after = 0;
    LIST_FORALL(&filePtr->sectionList, (List_Links *) sectionPtr) {
        after++;
        if (prevPtr != NULL) {
            if (sectionPtr->shdr.sh_addr != 0) {
                assert(sectionPtr->shdr.sh_addr >= prevPtr->shdr.sh_addr);
            }
        }
        prevPtr = sectionPtr;
    }
    assert(before == after);
}


static void
SortSegments(
    Elf_File    *filePtr)
{
    Elf_Segment *segmentPtr;
```

```
    Elf_Segment *insertPtr;
    List_Links  *nextPtr;
    Elf_Segment *prevPtr;
    List_Links  sortedList;
    int         before;
    int         after;

    List_Init(&sortedList);

    before = 0;
    LIST_FORALL_DEL(&filePtr->segmentList, (List_Links *) segmentPtr, nextPtr) {
        List_Remove((List_Links *) segmentPtr);
        before++;
        LIST_FORALL(&sortedList, (List_Links *) insertPtr) {
            if (insertPtr->phdr.p_vaddr > segmentPtr->phdr.p_vaddr) {
                break;
            }
        }
        List_Insert((List_Links *) segmentPtr, LIST_BEFORE(insertPtr));
    }
    assert(List_IsEmpty(&filePtr->segmentList));
    List_ListInsert(&sortedList, LIST_ATREAR(&filePtr->segmentList));
    prevPtr = NULL;
    after = 0;
    LIST_FORALL(&filePtr->segmentList, (List_Links *) segmentPtr) {
        after++;
        if (prevPtr != NULL) {
            assert(segmentPtr->phdr.p_vaddr >= prevPtr->phdr.p_vaddr);
        }
        prevPtr = segmentPtr;
    }
    assert(before == after);
}

static void
Usage(
    char        *name)
{
    fprintf(stderr, "Usage: %s [-o output] [-sS] file1 file2...\n", name);
    exit(1);
}

static void
Debug(
    char *fmt,
    ...)
{
#ifdef DEBUG
    va_list     ap;

    va_start(ap, fmt);
    vfprintf(stderr, fmt, ap);
    va_end(ap);
#endif /* DEBUG */
}

static void
NukeSection(
        Elf_Section     *sectionPtr)
{
    Debug("Nuking section %s\n", sectionPtr->name);
    sectionPtr->name = "unused";
```



```
    sectionPtr->shdr.sh_type = SHT_PROGBITS;
    sectionPtr->shdr.sh_link = 0;
    sectionPtr->shdr.sh_info = 0;
    /*
     * Memory leak. The section may contain a string table and
     * therefore still be in use.
     */
    sectionPtr->section = xmalloc(sectionPtr->shdr.sh_size);
    memset(sectionPtr->section, 0, sectionPtr->shdr.sh_size);
}

static void
Relocate(
    Elf_File    *filePtr,
    Elf_Section *sectionPtr)
{

    fprintf(stderr, "Relocation of section type SHT_REL not implemented.\n");
    abort();
}
static void
RelocateA(
    Elf_File    *filePtr,
    Elf_Section *sectionPtr)
{
    Elf32_Rela  *relArray;
    Elf32_Rela  *relPtr;
    int         count;
    int         i;

    assume(sectionPtr->shdr.sh_type == SHT_RELA);
    relArray = (Elf32_Rela *) sectionPtr->section;
    assume(sectionPtr->shdr.sh_size % sizeof(Elf32_Rela) == 0);
    count = sectionPtr->shdr.sh_size / sizeof(Elf32_Rela);
    for (i = 0; i < count; i++) {
        relPtr = &relArray[i];
        Debug("RelA %d: offset = 0x%x info = 0x%x addend = 0x%x\n", i,
            relPtr->r_offset, relPtr->r_info, relPtr->r_addend);

        /*
         * Prelink puts the address to be patched in r_offset and
         * the value with which to patch it in r_addend.
         */
        Patch(filePtr, relPtr->r_offset, relPtr->r_addend);
    }
}

static void
Patch(
    Elf_File            *filePtr,
    Elf32_Addr          addr,
    Elf32_Sword         value)
{
    Elf32_Sword         *ptr;
    int                 offset;
    Elf_Section         *sectionPtr;

    /*
     * Find the memory address of the data to be patched in the
     * executable.
     */
    LIST_FORALL(&filePtr->sectionList, (List_Links *) sectionPtr) {
```

```
        if ((sectionPtr->shdr.sh_addr <= addr) &&
            (sectionPtr->shdr.sh_addr + sectionPtr->shdr.sh_size > addr)) {
            assume(sectionPtr->shdr.sh_type != SHT_NOBITS);
            offset = addr - sectionPtr->shdr.sh_addr;
            ptr = (Elf32_Sword *) (sectionPtr->section + offset);
            Debug("Patching addr 0x%x (ptr 0x%x) with 0x%x\n",
                addr, ptr, value);
            *ptr = value;
            break;
        }
    }
}
```