Week 07

Compiling C to MIPS

What does the compiler need to do to convert C to MIPS?

- convert `#include` and `#define`
- *parse* code to check syntactically valid
- manage a list of *symbols* used in program
- decide how to represent data structures
- allocate local variables to registers or stack
- map control structures to MIPS instructions

C Pre-processor

Maps C→C, performing various *substitutions*

- **`#include`** *File*
  - replace `#include` by contents of file
  - `"name.h"` ... uses named *File*`.h`
  - `<name.h>` ... uses *File*`.h` in `/usr/include`
- **`#define`** *Name Constant*
  - replace all occurences of symbol *Name* by *Constant*
  - e.g `#define MAX 5`
    `char array[MAX]` → `char array[5]`
- **`#define`** *Name*`(`*Params*`)` *Expression*
  - replace *Name*`(`*Params*`)` by *SubstitutedExpression*
  - e.g. `#define max(x,y) ((x > y) ? x : y)`
    `a = max(b,c)` → `a = ((b > c) ? b : c)`

... C Pre-processor

More C pre-processor substitions

Before cpp

```
x = 5;
#if 0
x = x + 1;
#else
x = x + 2;
#endif

#ifdef DEBUG
printf("x=%d\n",x);
#endif
x = x * 2;
```

After cpp

```
x = 5;
x = x + 2;
printf("x=%d\n",x);
x = x * 2;
```

Assuming ...

```
#define DEBUG 1
```

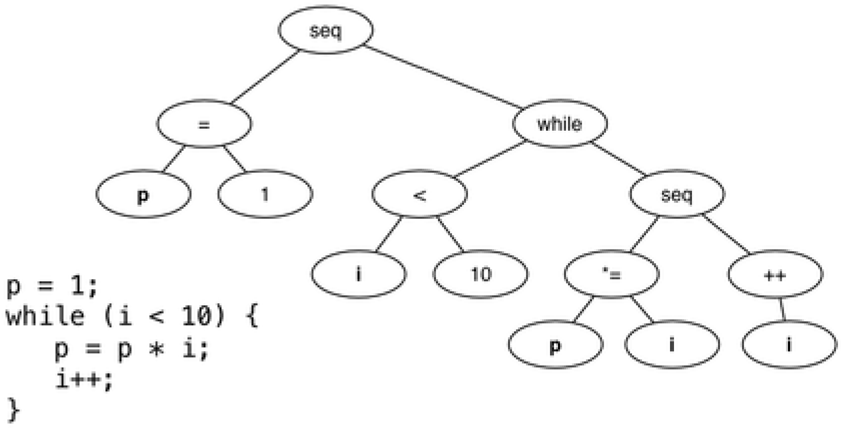
or

```
gcc -DDEBUG=1 ...
```

C Parser

Understands syntax of C language

Attempts to convert C program into *parse tree*



Symbol Table Management

5/34

Compiler keeps track of names

- scope, lifetime, locally/externally defined
- disambiguates e.g. `x` in `main()` vs `x` in `fun()`
- resolves symbols to specific locations (data/stack/registers)
- external symbols may remain unresolved until linking
- however, need to have a type for each external symbol

Example:

```
double fun(double x, int n);

int main(void) {
   int i;  double res;
   scanf("%d", &i);
   res = fun((float)i, 5);
   return 0;
}
```

Local Variables

6/34

Two choices for local variables

- on the stack ... +persist for whole function, -`lw`/`sw` needed in MIPS
- in a register ... +efficient, -not many, useful if var used in small scope
  - if need to persist across function calls, use `$s?` register
  - if used in very localised scope, can use `$t?` register

Example:

```
int sum(List L)
{
   if (L == NULL) return 0;
   int first = L->value;      // must be in $s?
   int rest = sum(L->next);   // can be in $t?
   return first + rest;
}
```

Expression Evaluation

7/34

Uses temporary registers

- even complex expressions don't generally need > 3-4 registers

Example:

```
   x = ((y+3) * (z-2) * x) / 4;
```

```
    lw   $t0, y
    addi $t0, $t0, 3     # t0 = y + 3
    lw   $t1, z
    addi $t1, $t1, -2    # t1 = z - 2
    mul  $t0, $t0, $t1   # t0 = t0 * t1
    lw   $t1, x
    mul  $t0, $t0, $t1   # t0 = t0 * x
    li   $t1, 4
    div  $t0, $t0, $t1   # t0 = t0 / 4
```

Complex boolean expressions handled by short-circuit evaluation.

Mapping Control Structures

Use templates, e.g.

```
while (Cond) { Stat1; Stat2; ... }

loop:
   MIPS code to check Cond; result in $t0
   beqz $t0, end_loop
   MIPS code for Stat1
   MIPS code for Stat2
   MIPS code for ...
   j    loop
end_loop:
```

... Mapping Control Structures

Concrete example:

```
while (i < N) { p = p*i; i++; }

    lw   $s0, 8($sp)     # N is on stack
loop5:
    lw   $t1, 4($sp)     # i is on stack
    slt  $t0, $t1, $s0   # (i < N)
    beqz $t0, end_loop
    lw   $t0, 0($sp)     # p is on stack
    mul  $t0, $t0, $t1
    sw   $t0, 0($sp)     # p = p * i
    add  $t1, $t1, 1
    sw   $t1, 4($sp)     # i++
    j    loop
end_loop5:
```

Could easily optimise this to maintain all variables in registers

---

... Mapping Control Structures

Template for `if...else if... else`

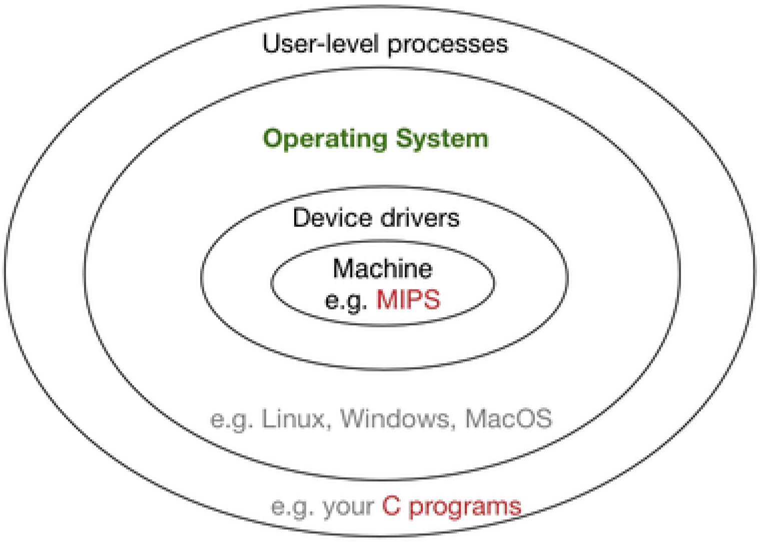
```
if (Cond1) Stat1 else if (Cond2) Stat2 else Stat3

if:
   MIPS code to check Cond1; result in $t0
   beqz $t0, else1
   MIPS code for Stat1
   j    end_if
else1:
   MIPS code to check Cond2; result in $t0
   beqz $t0, else2
   MIPS code for Stat2
   j    end_if
else2:
```

```
    MIPS code for Stat3
end_if:
```

Coputer Systems Architecture

11/34



Evolution of Operating Systems (OSs)

12/34

1940's (e.g. ENIAC)

- no OS ... one program at a time, manually loaded
- programs had to take account of details of machine/devices

1950's (e.g. Whirlwind)

- batch processing ... load several programs at once, run in sequence
- programs had to take account of details of machine/devices

1960's (e.g. IBM360)

- computers proliferate ... programmers want to transport code
- having to cope with different config on each machine was tedious
- solution: layer of software between raw machine and user programs

Nice example of using abstraction to enhance code portability

Operating Systems

13/34

Operating systems

- have privileged access to the raw machine
- manage use of machine resources (CPU, disk, memory, etc.)
- provide uniform interface to access machine-level operations
- arrange for controlled execution of user programs
- provide multi-tasking and (pseudo) parallelism

Abstractions provided by modern OSs

- users, privileges ... e.g. `whoami`, `groups`, `seteuid()`
- file system, i/o ... e.g. `ls`, `open()`, `read()`
- processes ... e.g. `ps`, `top`, `fork()`
- communication ... e.g. `connect()`, `send()`, `recv()`

Core OS functions form the *kernel*, which runs in *privileged mode*

System Calls

14/34

SPIM has no OS, but provides a simple set of "system calls"

- primarily for i/o (read/write) on various types
- also memory allocation and process exit

An OS like Unix/Linux provides 100's of system calls

- process management (e.g. `fork()`, `exec()`, `_exit()`, ...)
- file management (e.g. `open()`, `read()`, `fstat()`, ...)
- device management (e.g. `ioctl()`, ...)
- information maintenance (e.g. `settimeofday()`, `getuid()`, ...)
- communication (e.g. `pipe()`, `connect()`, `send()`, ...)

User programs invoke sys calls through an API (POSIX + Linux)

... System Calls 15/34

... System Calls 16/34

System calls are invoked ...

- directly, through a library of system calls
  - documented in Unix Programmers Manual section 2 (e.g. `man 2 open`)
- indirectly, through functions in the C libraries
  - documented in Unix Programmers Manual section 3 (e.g. `man 3 fopen`)

Example of system call library vs C library

- file descriptors, `open()`, `close()`, `read()`, `write()`

  (via `#include <unistd.h>`)

- file pointers (`FILE*`), `fopen()`, `fclose`, `scanf()`, `printf()`

  (via `#include <stdio.h>`)

... System Calls 17/34

System calls attempt to perform actions, but may fail

User programs can detect this in several ways

- check return value of sys call function (-1 typically flags an error)
- check global variable **`errno`** (contains specific error)

C programs need to check and handle errors themselves (no exceptions)

Library function to make it easy to report errors and exit

- **`error(`***Status*, *ErrNum*, *Format*, *Expressions*, ...**`)`**
- print error message using prog name, *Format* (`printf()`), and *Expressions*
- if *Status* is non-zero, invoke `exit(`*Status*`)` after printing message
- if *ErrNum* is non-zero, also print standard system error message

Note: successful system calls generally return 0

Exercise 1: Failed System Call 18/34

What is displayed after an attempt to open a non-existent file

```
#include <unistd.h>
#include <fcntl.h>
#include <error.h>
#include <errno.h>
```

```
int main(int argc, char *argv[])
{
   int in;
   if (argc < 2)
      error(1, 0, "Usage: %s File", argv[0]);
   in = open(argv[1],O_RDONLY);
   if (in < 0)
      error(errno, errno, "Can't open %s", argv[1]);
   close(in);
   return 0;
}
```

File Systems

19/34

File systems provide a mechanism for managing *stored data*:

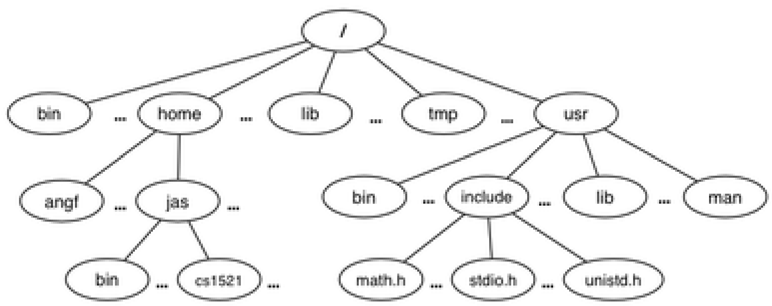
- typically on a disk device (or, nowadays, on SSD)
- allocating chunks of space on the device to *files*
    - where a file is viewed as a sequence of bytes
- allowing access to files by name and with access rights
- arranging access to files via *directories* (folders)
- maintaining information about files/directories (*meta-data)*
- dealing with damage on the storage device ("bad blocks")

A *file system* is an important mechanism provided by an OS.

Unix/Linux File System

20/34

Unix/Linux file system is tree-structured



(We say it's "tree structured", but symlinks actually make it into a graph)

Processes have a notion of their location within the file system

- *current working directory* (CWD)

... Unix/Linux File System

21/34

The file system is used to accessing various types of objects:

- files, directories (folders), devices, processes, sockets, ...

Objects are referenced via a *path* (.../`x/y/z/`...)

Paths can be

- *absolute* (full path from root)

  e.g. `/usr/include/stdio.h`, `/home/jas/cs1521/`

- *relative* (path starts from CWD)

  e.g. `../../another/path/prog.c`, `./a.out`, `a.out`

Q: Why do we have to run `a.out` as `./a.out`?

... Unix/Linux File System

22/34

Unix defines a range of file-system-related types:

- **`off_t`** ... offsets within files
  - typically, `long` and signed to allow backward refs
- **`size_t`** ... number of bytes in some object
  - unsigned, since objects can't have negative size
- **`ssize_t`** ... sizes of read/written blocks
  - like `size_t`, but signed to allow for error values
- **`struct stat`** ... file system object metadata
  - stores information about file, but stores no content
  - requires `ino_t`, `dev_t`, `time_t`, `uid_t`, ...

... Unix/Linux File System

23/34

Metadata for file system objects is stored in *inodes*

- physical location on storage device of file data
- file type (regular file, directory, ...), file size (bytes/blocks)
- ownership, access permissions, timestamps (create/access/update)

Each file system volume has a table of inodes in a known location

Note: an inode does not contain the name of the file

Access to a file by name requires a *directory*

- where a directory is effectively a list of (name,inode) pairs

---

... Unix/Linux File System

24/34

Access to files by name proceeds as ...

- open directory and scan for *name*
- if not found, "No such file or directory"
- if found as (*name*,`ino`), access inode table `inodes[ino]`
- collect file metadata and ...
  - check file access permissions given current user/group
    - if don't have required access, "Permission denied"
  - collect information about file's location and size
  - update access timestamp
- use physical location to access device and read/write file's data

---

File System Operations

25/34

Unix presents a uniform interface to file system objects

- functions/syscalls manipulate objects as a *stream of bytes*
- accessed via a *file descriptor* (index into a system table)

Some common operations:

- `open()` ... open a file system object, returning a file descriptor
- `close()` ... stop using a file descriptor
- `read()` ... read some bytes into a buffer from a file descriptor
- `write()` ... write some bytes from a buffer to a file descriptor
- `lseek()` ... move to a specified offset within a file
- `stat()` ... get meta-data about a file system object

... File System Operations

26/34

`int open(char *`*Path*`, int `*Flags*`)`

- attempt to open an object at *Path*, according to *Flags*
- flags (defined in `<fcntl.h>`)
    - `O_RDONLY` ... open object for reading
    - `O_WRONLY` ... open object for writing
    - `O_APPEND` ... open object for writing at end
    - `O_RDWR` ... open object for reading and writing
    - `O_CREAT` ... create object if doesn't exist
- flags can be combined e.g. `(O_WRONLY|O_CREAT)`
- if successful, return file descriptor (small +ve `int`)
- if unsuccessful, return -1 and set `errno`

... File System Operations

27/34

`int close(int `*FileDesc*`)`

- attempt to release an open file descriptor
- if this is the last reference to object, release its resources
- if successful, return 0
- if unsuccessful, return -1 and set `errno`

Could be unsuccessful if *FileDesc* is not an open file descriptor

An aside: removing an object e.g. via `rm`

- removes the object's entry from a directory
- but the inode and data persist until
    - all processes accessing the object `close()` their handle
    - all references to the inode from other directories are removed
- after this, the inode and the blocks on storage device are recycled

---

... File System Operations

28/34

`ssize_t read(int `*FileDesc*`, void *`*Buffer*`, size_t `*Count*`)`

- attempt to read *Count* bytes from *FileDesc* into *Buffer*
- if "successful", return number of bytes actually read (*NRead*)
- if currently positioned at end of file, return 0
- if unsuccessful, return -1 and set `errno`
- does not check whether *Buffer* contains enough space
- advances the file offset by *NRead*
- does not treat `'\n'` as special

Once a file is `open()`'d ...

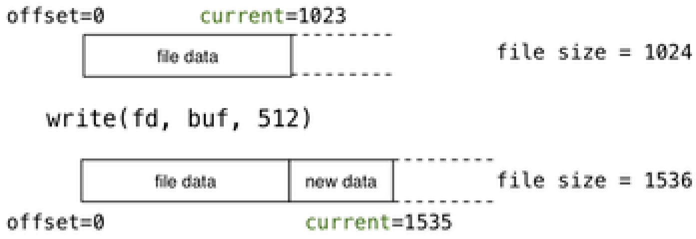
- the "current position" in the file is maintained as part of the fd entry
- the "current position" is modified by `read()`, `write()` and `lseek()`

... File System Operations

29/34

`ssize_t write(int `*FileDesc*`, void *`*Buffer*`, size_t `*Count*`)`

- attempt to write *Count* bytes from *Buffer* onto *FileDesc*
- if "successful", return number of bytes actually written (*NWritten*)
- if unsuccessful, return -1 and set `errno`
- does not check whether *Buffer* has *Count* bytes of data
- advances the file offset by *NWritten* bytes



Exercise 2: (FILE *) vs FileDesc

30/34

Write three programs to scan a file and write it to stdout

- for one use `stdio.h` and read char-by-char
- for one use `stdio.h` and read line-by-line
- for one use `unistd.h` and read block-by-block

Notes:

- stdout is accessible via file descriptor 1
- check whether the size of `read()`'s buffer matters
- system calls are relatively expensive operations

... File System Operations

31/34

Functions from `stdio.h` tend to be `char`-oriented

File-descriptor-based system calls deal with byte sequences

- bytes can be interpreted as `char`, `int`, `struct`, etc
- so, many kinds of objects can be `read()` or `write()` **

Allows programmers to manipulate files of data items, e.g.

- list of `double` values read from sensor device
- collection of `Student` records

** you cannot save/restore pointer values using `write()`/`read()`

- because they refer to memory addresses within a process instance
- and a different process instance might already have used those addresses

---

... File System Operations

32/34

Files of *records* can be produced by

- either, `write()`ing chunks of bytes from `struct` objects
- or, printing formatted text representation of `struct` data

The latter approach is a form of *serialisation*

For the `write()` approach:

- no need to worry about formatting issues
- writes entire structure, even if string buffers half empty
- can `lseek()` to i^{th} struct via *i*sizeof(StructType)*

For the printing approach:

- produces files that are human-readable
- only uses as many bytes as required from string buffers
- can access structures only sequentially (unless using padding)

... File System Operations

33/34

Example of `write()`ing records vs `printf()`ing records

```
typedef struct _student {
    int id; char name[99]; float wam;
} Student;
int infd, outfd; // file descriptors
FILE *inf, *outf; // file pointers

Student stu;  ... set values in stu.id, etc ...

write(outfd, &stu, sizeof(struct _student));
  vs
fprintf(outf, "%d:%s:%f\n",
             stu.id, stu.name, stu.wam);

read(infd, &stu, sizeof(Student));
  vs
fscanf(inf, "%d:[^:]:%f\n", // maybe?
            &(stu.id), &(stu.name), &(stu.wam));
```

Exercise 3: File of Structs

34/34

Write a program to ...

- read in data about one student
- append the data to a file of students

Write a program to ...

- scan the file of students and print data for each one

Write two versions of each program ...

- one using the `write()`/`read()` approach
- one using the `printf()`/`scanf()` approach

Produced: 7 Sep 2017