Section 3:
File I/O, JSON, Generics

Meghan Cowan

POSIX

- Family of standards specified by the IEEE
- Maintains compatibility across variants of Unix-like OS
- Defines API and standards for basic I/O: file, terminal and network
- Also defines a standard threading library API

Basic File Operations

- Open the file
- Read from the file
- Write to the file
- Close the file / free up resources

System I/O Calls

```
int open(char* filename, int flags, mode_t mode);
```

Returns an integer which is the file descriptor.

Returns -1 if there is a failure.

`filename`: A string representing the name of the file.

`flags`: An integer code describing the access.

- O_RDONLY -- opens file for read only
- O_WRONLY – opens file for write only
- O_RDWR – opens file for reading and writing
- O_APPEND --- opens the file for appending
- O_CREAT -- creates the file if it does not exist
- O_TRUNC -- overwrite the file if it exists

`mode`: File protection mode. Ignored if O_CREAT is not specified.

`[man 2 open]`

System I/O Calls

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

`fd`: file descriptor.

`buf`: address of a memory area into which the data is read.

`count`: the maximum amount of data to read from the stream.

The return value is the actual amount of data read from the file.

```
int close(int fd);
```

Returns 0 on success, -1 on failure.

[man 2 read]

[man 2 write]

[man 2 close]

Errors

- When an error occurs, the error number is stored in **`errno`**, which is defined under `<errno.h>`
- View/Print details of the error using **`perror()`** and **`errno`**.
- POSIX functions have a variety of error codes to represent different errors. Some common error conditions:
  - **`EBADF`** **-** *fd* is not a valid file descriptor or is not open for reading.
  - **`EFAULT`** **-** *buf* is outside your accessible address space.
  - **`EINTR`** **-** The call was interrupted by a signal before any data was read.
  - **`EISDIR`** **-** *fd* refers to a directory.
- `errno` is shared by all library functions and overwritten frequently, so you must read it right after an error to be sure of getting the right code

[man 3 errno]

[man 3 perror]

Reading a file

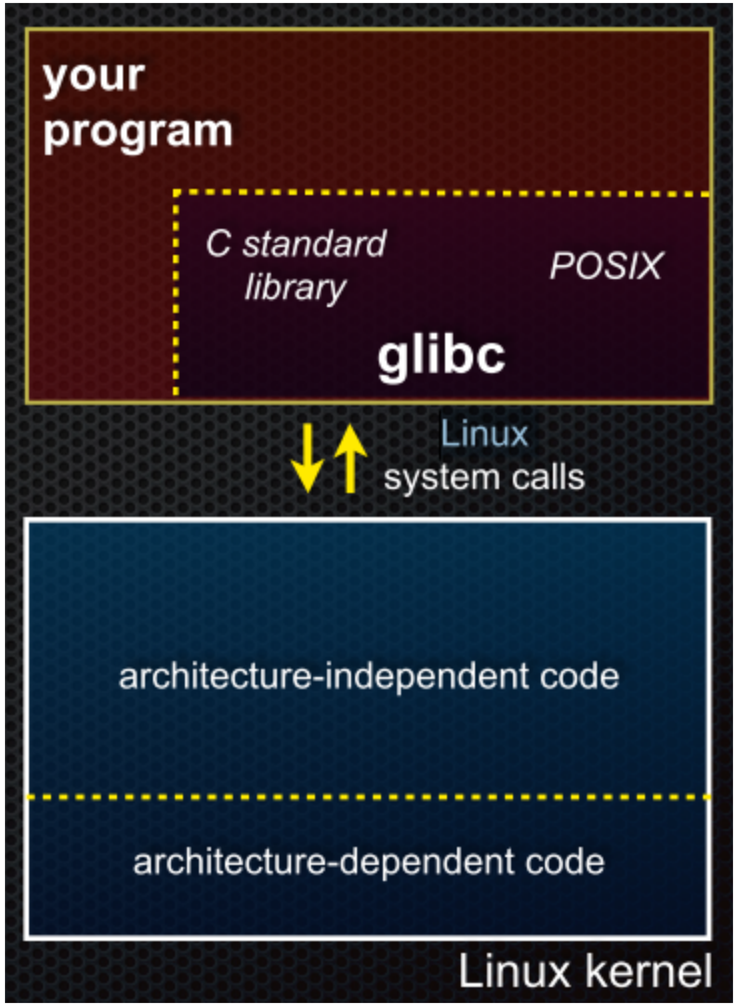
```
#include <errno.h>
#include <unistd.h>

...

  char *buf = ...;      // buffer has size n
  int bytes_left = n;   // where n is the length of file in bytes
  int result = 0;

  while (bytes_left > 0) {
      result = read(fd, buf + (n-bytes_left), bytes_left);
      if (result == -1) {
          if (errno != EINTR) {
            // a real error happened, return an error result
          }
          // EINTR happened, do nothing and loop back around
          continue;
      }
      bytes_left -= result;
  }
```

STDIO vs. POSIX Functions

- User mode vs. Kernel mode.

- STDIO library functions
  - fopen, fread, fwrite, fclose, etc.
    use FILE* pointers.

- POSIX functions
  - open, read, write, close, etc.
    use integer file descriptors.



JSON & Jannsson

JSON

- Data format to transmit objects in human readable text
  - Not specific to JavaScript – derived from javascript but any language can write and parse it
- In HW2 use it to *serialize* a 2D array or in general any complicated object
  - Serialize -> create a one dimensional representation of this
- Will use the JSON output to test your input
  - Not defining the interface for you so we can’t run unit tests. Instead will compare against runtime data stored

JSON cont.

- Represents simple types like integer and string plus two complex types: arrays and maps
- Arrays using square brackets [1, 2, “hello”]
- Maps using curly braces {“key”: 1, “cat” 2}

Jannsson

- Library we provide to help read and write JSON files.
- Use it serialize your 2D array by creating a Jansson object and populating with values from your 2D array, then use Jansson to write JSON to file

```
json_t *array = json_array();
json_array_append_new(array,
json_integer(42));

json_t *obj = json_object()
Json_object_set_new(obj, “foo”, array);
```

Jansson cont.

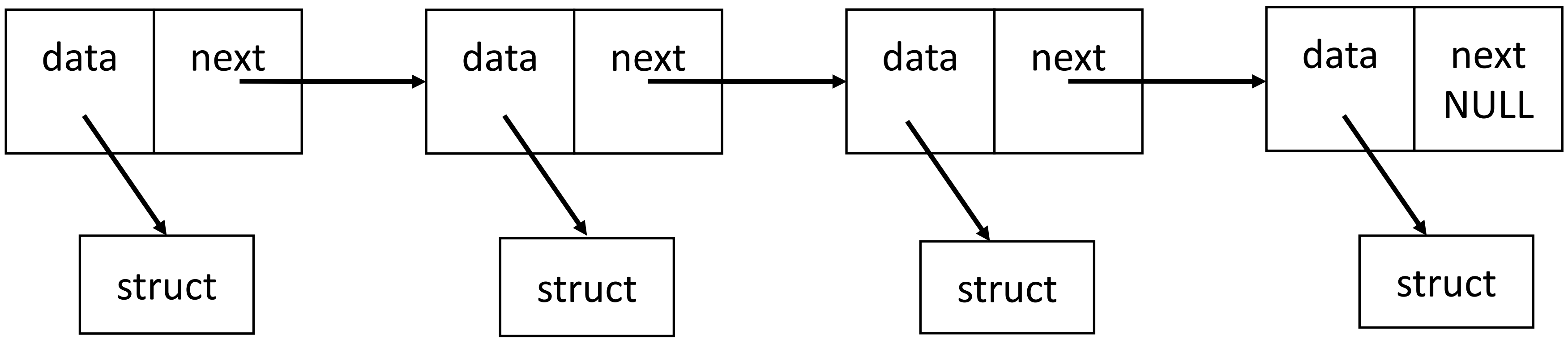
- Deserialize JSON data into a Jansson object and fetch values from it to re-populate your 2D array

```
// Loading
json_t *root;
json_error_t error;
root = json_loads( data, 0, &error );
… error checking

// Extract functions
json_object_get(root, "field)
json_array_get(root)
```

- Documentation provided in the library!

Generics

Using void pointers



- Data is a void* - can be a pointer to anything
  - Can also directly store primitive sizes like ints, floats (as long as < size of pointer) to avoid allocating extra memory
  - Up to the programmer to keep track of types of elements in the list
  - User must cast to the appropriate type to operate on the data

void*generics - callbacks

- Data structure can provide functions that apply user specified callback to elements
- User can explicitly cast void* pointers to desired type and preform an operation
  - Custom free function – frees pointers to malloc'd data, does nothing for primitives
  - Map function, etc.
- Implemented generic LinkedList in HW1
  - Free and sort functions that were type specific

Using the preprocessor

- Use the preprocessor to expand macros and generate type specific versions of the data structure.

```
#define CREATE_LLIST_TYPE(t,s)                \
typedef struct llist_node_t_ ## s {           \
    struct llist_node_t_ ## s  *next;         \
    t data;                                   \
} LList_node_ ## s;                           \
```

- Each call to CREATE_LIST_TYPE(t,s) generates the appropriate code during preprocessing. You explicitly tell the preprocessor what code to create.
- Notice each version must have a different name to link -> name mangling

concatenates with no spaces between them

```
#define CREATE_LLIST_TYPE(t,s)            \
typedef struct llist_node_t_ ## s {       \
    struct llist_node_t_ ## s  *next;     \
    t data;                               \
} LList_node_ ## s;                       \
```

```
#define CREATE_LLIST_TYPE(int,int)
```

↓

```
typedef struct llist_node_t_int {
   struct llist_node_t_int *next;
   int data;
} LList_node_int;
```

```
#define CREATE_LLIST_TYPE(char*,string)
```

↓

```
typedef struct llist_node_t_string {
    struct  llist_node_t_string *next;
    char* data;
} LList_node_string;
```

Preprocessor caveats

- Can’t hide any implementation from the user (no private headers)
- Source code written in the headers
- Hard to debug…

- Will see something similar with how C++ implements generics using templates.