How to read a code repository?

Reading egos-2000 as an example

Read a repository: 3 passes

- ➡ 1st pass
  - read documents and filenames
- 2nd pass
  - track the execution: earth → grass → applications
- 3rd pass
  - read details of specific functionality, such as system call

Documents of egos-2000

- Explain why the project is important
  - README.md
- Explain how to use this project
  - references/USAGES.md
- Explain the internal design of the project
  - references/README.md

Software companies use tools like

ATLASSIAN Confluence

Read filenames: earth

- sd
- ~~bus_gpio.c~~
- ~~bus_uart.c~~
- cpu_intr.c
- cpu_mmu.c
- dev_disk.c
- dev_page.c
- dev_tty.c
- earth.S
- earth.c
- earth.lds

- `gpio` and `uart` are buses connecting the CPU with I/O devices, just like `usb`
- in the first pass of reading code, knowing what they are on the high-level is enough

Read filenames: earth

- sd
- ~~bus_gpio.c~~
- ~~bus_uart.c~~
- cpu_intr.c
- cpu_mmu.c
- ~~dev_disk.c~~
- ~~dev_page.c~~
- ~~dev_tty.c~~
- earth.S
- earth.c
- earth.lds

- `dev_disk` controls ROM and SD card
- `dev_tty` reads keyboard input and print output to the screen
- `dev_page` does paging from memory to the first 1MB of the SD card

Described in references/README.md, the document for internal design.

Read filenames: earth

- sd
- ~~bus_gpio.c~~
- ~~bus_uart.c~~
- ~~cpu_intr.c~~
- ~~cpu_mmu.c~~
- ~~dev_disk.c~~
- ~~dev_page.c~~
- ~~dev_tty.c~~
- earth.S
- earth.c
- earth.lds

- `cpu_intr`: interrupt/exception handling
- `cpu_mmu`: memory management unit (MMU)

Described in references/README.md, the document for internal design.

- sd
- ~~bus_gpio.c~~
- ~~bus_uart.c~~
- ~~cpu_intr.c~~
- ~~cpu_mmu.c~~
- ~~dev_disk.c~~
- ~~dev_page.c~~
- ~~dev_tty.c~~
- ~~earth.S~~
- ~~earth.c~~
- ~~earth.lds~~

Read filenames: earth

- `earth.S` and `earth.c` are for earth layer initialization (e.g., the `main()` of earth)
- `earth.lds` specifies the memory layout

Read filenames: earth/sd

- sd.h
- sd_init.c
- sd_rw.c
- sd_utils.c

- `sd.h` provides basic definitions
- `sd_init.c` initializes the SD card
- `sd_rw.c` provides SD card read and write
- `sd_utils.c` provides helper functions
- There is an optional project P4 on SD driver

Read filenames: grass

- grass.S
- grass.c
- grass.lds
- process.c
- process.h
- scheduler.c
- syscall.c
- syscall.h
- timer.c

- `grass.S` and `grass.c`: initialization
- `grass.lds`: memory layout

Grass layer (hardware independent)

- `grass/timer` : timer control registers
- `grass/syscall` : system call interfaces to user applications
- `grass/process` : data structures for managing processes (touched by P1)
- `grass/scheduler` : preemptive scheduling and inter-process communication

Described in references/README.md, the document for internal design.

Read a repository: 3 passes

- 1st pass
  - read documents and filenames
- ➡ 2nd pass
  - track the execution: earth → grass → applications
- 3rd pass
  - read details of specific functionality, such as system call

The Key:

Find `main()` functions

and track executions from there

grep is a useful command

```
> cd egos-2000
> grep "main(" -r *
```

Main functions in the repository

```
> cd egos-2000
> grep "main(" -r *
earth/earth.S:    /* Call main() of earth.c */
earth/earth.c:int main() {
grass/grass.S:    /* Call main() of grass.c */
grass/grass.c:int main() {
tools/mkrom.c:int main() {
tools/mkfs.c:int main() {

apps/*.c: /* Every application has a main() function */
```

Main function in earth

- Read earth.s and earth.c
  - Boot loader disable interrupt and call earth `main()`
  - Earth `main()` essentially
    - initialize dev_tty, dev_disk, cpu_intr, cpu_mmu
    - load and enter the grass layer

Main function in grass

- Read grass.s and grass.c
  - Initialize data structures for processes (like P1)
  - Initialize and enable timer interrupt
  - Load and enter the first application: GPID_PROCESS
- Where is GPID_PROCESS defined?

Find GPID_PROCESS

```
> cd egos-2000
# Find which header file contains GPID_PROCESS
> grep "GPID_PROCESS" -r * | grep “\.h"

library/servers/servers.h:    GPID_PROCESS,
library/servers/servers.h: /* GPID_PROCESS */
```

Kernel servers (aka. Daemon)

```
enum grass_servers {
    GPID_UNUSED,
    GPID_PROCESS,
    GPID_FILE,
    GPID_DIR,
    GPID_SHELL,
    GPID_USER_START
};
```

- `GPID_PROCESS`
  - spawn and kill processes
- `GPID_FILE & GPID_DIR`
  - something about file system
- `GPID_SHELL`
  - shell for entering user commands

Control flow sketch

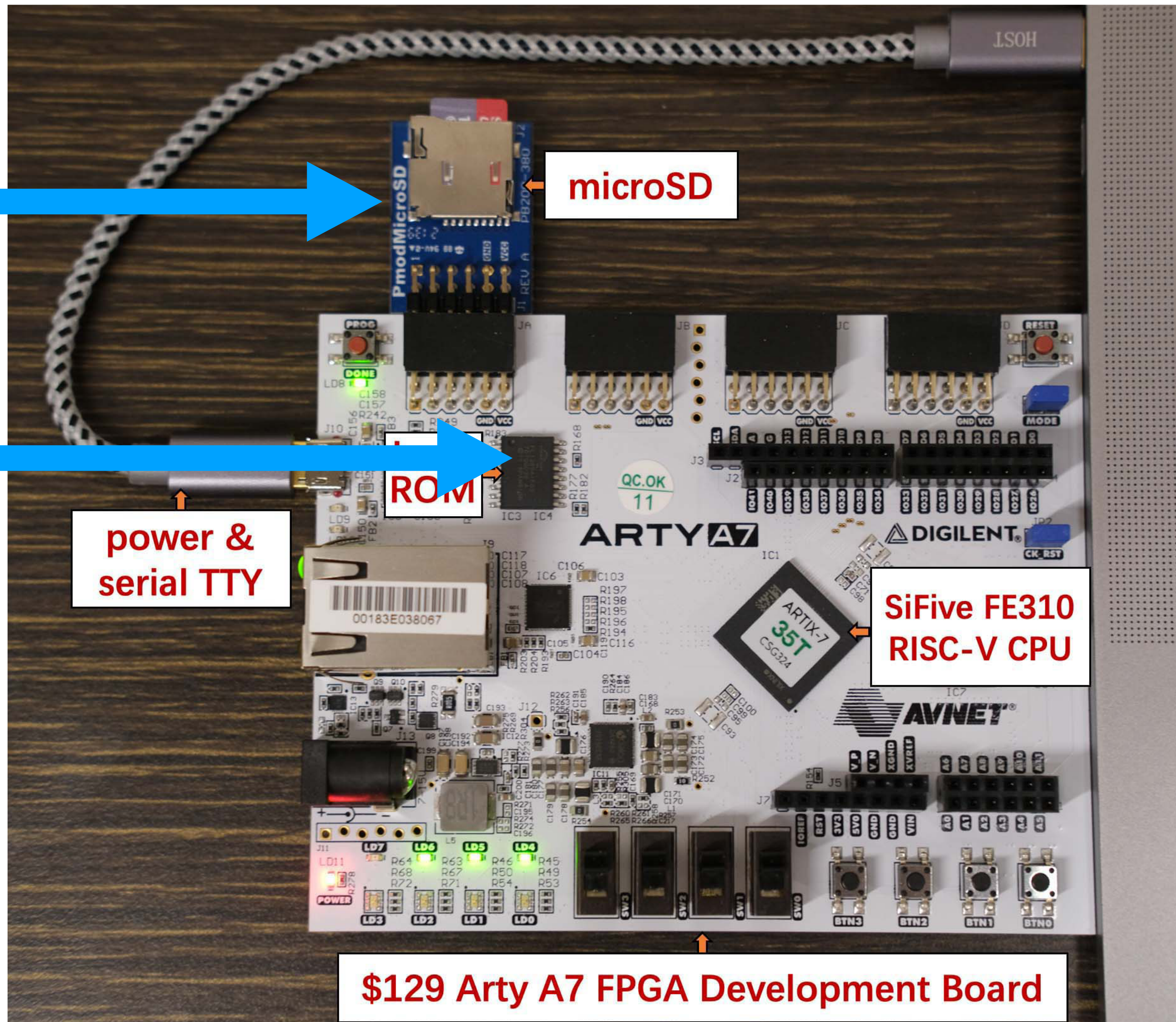
- During boot up
  - earth main() → grass main() → GPID_PROCESS
  - GPID_PROCESS → GPID_FILE
  - GPID_PROCESS → GPID_DIR
  - GPID_PROCESS → GPID_SHELL
- After boot up
  - GPID_SHELL → GPID_PROCESS → user applications

Two more main functions to read

```
> cd egos-2000
> grep "main(" -r *
earth/earth.S:      /* Call main() of earth.c */
earth/earth.c:int main() {
grass/grass.S:      /* Call main() of grass.c */
grass/grass.c:int main() {
tools/mkrom.c:int main() {
tools/mkfs.c:int main() {

apps/*.c: /* Every application has a main() function */
```

mkfs and mkrom

- During make, the RISC-V compiler compiles egos-2000
  - i.e., create everything under build/
- During make install,
  - mkfs creates disk.img
  - mkrom creates bootROM.bin



Control flow provides a rough picture

```
> cd egos-2000
> grep "main(" -r *
earth/earth.S:      /* Call main() of earth.c */
earth/earth.c:int main() {
grass/grass.S:      /* Call main() of grass.c */
grass/grass.c:int main() {
tools/mkrom.c:int main() {
tools/mkfs.c:int main() {

apps/*.c: /* Every application has a main() function */
```

Control flow provides a rough picture

We now know the **structure** of the work and **some details**.

Read a repository: 3 passes

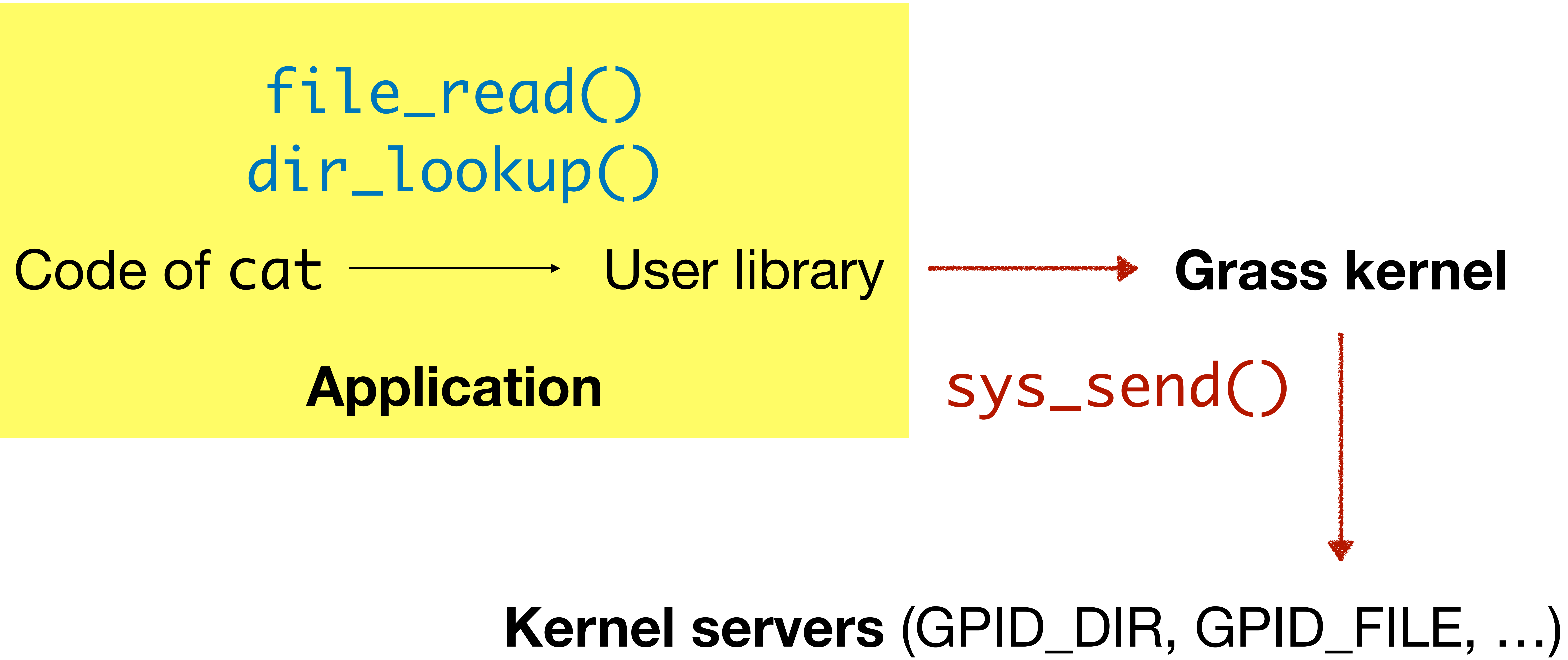
- 1st pass
  - read documents and filenames
- 2nd pass
  - track the execution: earth → grass → applications
- 3rd pass
  - read details of specific functionality, such as system call

Consider apps/user/cat.c

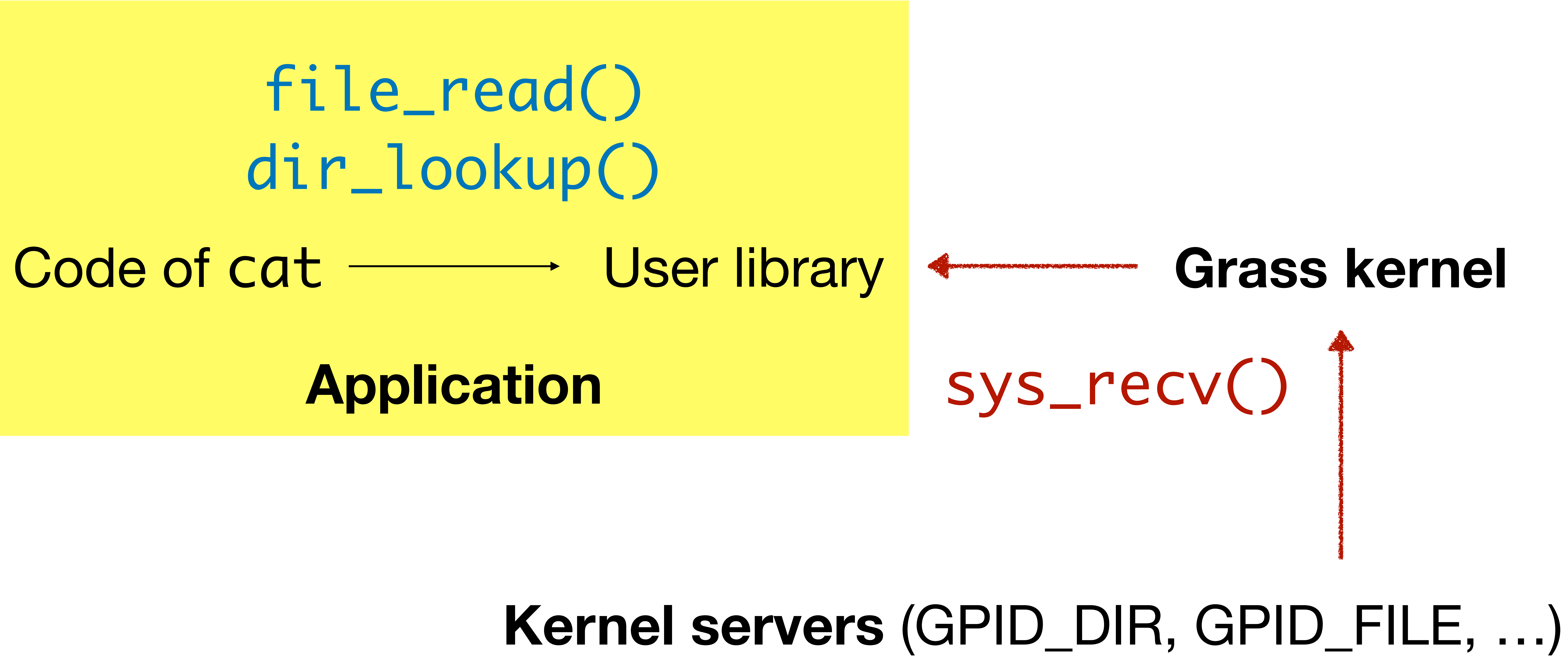
```c
#include "app.h"
#include <string.h>

int main(int argc, char** argv) {
    if (argc == 1) {
        INFO("usage: cat [FILE]");
        return -1;
    }

    /* Get the inode number of the file */
    int file_ino = dir_lookup(grass->workdir_ino, argv[1]);
    if (file_ino < 0) {
        INFO("cat: file %s not found", argv[1]);
        return -1;
    }

    /* Read and print the first block of the file */
    char buf[BLOCK_SIZE];
    file_read(file_ino, 0, buf);
    printf("%s", buf);
    if (buf[strlen(buf) - 1] != '\n') printf("\r\n");

    return 0;
}
```


```c
#include "app.h"
#include <string.h>

int main(int argc, char** argv) {
    if (argc == 1) {
        INFO("usage: cat [FILE]");
        return -1;
    }

    /* Get the inode number of the file */
    int file_ino = dir_lookup(grass->workdir_ino, argv[1]);
    if (file_ino < 0) {
        INFO("cat: file %s not found", argv[1]);
        return -1;
    }

    /* Read and print the first block of the file */
    char buf[BLOCK_SIZE];
    file_read(file_ino, 0, buf);
    printf("%s", buf);
    if (buf[strlen(buf) - 1] != '\n') printf("\r\n");

    return 0;
}
```

Send requests to file system



Receive data from file system



Data structures for system call

```
struct syscall {
    enum syscall_type type;
    struct sys_msg msg;
    int retval;
};
```

```
enum syscall_type {
        SYS_UNUSED,
        SYS_RECV,
        SYS_SEND,
        SYS_NCALLS
};
```

See header file grass/syscall.h

App invoking syscall step#1

```
static struct syscall *sc = (struct syscall*)SYSCALL_ARG;
```

a well-known memory address

```
static void sys_invoke() {
    *((int*)0x2000000) = 1;
}

int sys_send(int receiver, char* msg, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;

    sc->type = SYS_SEND;
    sc->msg.receiver = receiver;
    memcpy(sc->msg.content, msg, size);
    sys_invoke();
    return sc->retval;
}
```

App invoking syscall step#2

```
static struct syscall *sc = (struct syscall*)SYSCALL_ARG;

static void sys_invoke() {
    *((int*)0x2000000) = 1;
}
// The sys_send function takes 3 parameters
int sys_send(int receiver, char* msg, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;

    sc->type = SYS_SEND;
    sc->msg.receiver = receiver;
    memcpy(sc->msg.content, msg, size);
    sys_invoke();
    return sc->retval;
}
```

App invoking syscall step#3

```
static struct syscall *sc = (struct syscall*)SYSCALL_ARG;

static void sys_invoke() {
    *((int*)0x2000000) = 1;
}

int sys_send(int receiver, char* msg, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;
    // Prepare the system call data structure
    sc->type = SYS_SEND;
    sc->msg.receiver = receiver;
    memcpy(sc->msg.content, msg, size);
    sys_invoke();
    return sc->retval;
}
```

App invoking syscall step#4

```
static struct syscall *sc = (struct syscall*)SYSCALL_ARG;

static void sys_invoke() {
    *((int*)0x2000000) = 1;  // Trigger a software interrupt
}                            // which is interrupt #3

int sys_send(int receiver, char* msg, int size) {
    if (size > SYSCALL_MSG_LEN) return -1;

    sc->type = SYS_SEND;
    sc->msg.receiver = receiver;
    memcpy(sc->msg.content, msg, size);
    sys_invoke();
    return sc->retval;
}
```

App invoking syscall step#5

```
void kernel() {
    int mcause;
    __asm__ volatile("csrr %0, mcause" : "=r"(mcause));

    int id = mcause & 0x3ff;
    if (mcause & (1 << 31)) {
        if (id == 3) { syscall_handler(); }
        if (id == 7) { timer_handler(); } // last lecture
    } else {
        fault_handler();
    }
}
```

Homework

- Handle system call with the `ecall` instruction
  - Replace `*((int*)0x2000000) = 1` by `asm(“ecall”)` which triggers exception#8/#11 instead of interrupt#3
- P2 will be due on Mar 24
- Next lecture: memory exception and protection