XYZ Software Engineer Coding Exercise

Muhammad Ahsan

June 12, 2021

CONTENTS:

CHAPTER

ONE

XYZ SOFTWARE ENGINEER CODING EXERCISE

1.1 Task Description

Create two C applications that can communicate with each other (high-level IPC) where application A monitors changes to a file and application B displays those changes.

1.2 Proposed Solution

After going through the coding exercise's description and understanding the problem I decided to decompose the problem into smaller problems. As per my understanding, these are the main tasks in the problem:

1. Write a C application that monitors the changes to a file
2. Write another application that is notified about the changes monitored in the previous step
3. Use this application to display the changes

1.3 C Application to Monitor Changes in a File

In Linux files can be monitored with *inotify* API. The key steps in using the *inotify* API are as follows:

1. Use *inotify_init()* or *inotify_init1()* to create an *inotify instance*. This system call returns a file descriptor that is used to refer to the *inotify* instance in later operations.
2. The application informs the kernel about which files are of interest by using *inotify_add_watch()* to add items to the watch list of *inotify instance* in the previous step. Each watch item consists of a pathname and an associated bit mask. The bit mask specifies the set of events to be monitored for the pathname. *inotify_add_watch()* returns a watch descriptor that is used to refer to the watch in later operations.

3. Event notifications are gathered with *read()* operations on *inotify* file descriptor. Each successful *read()* returns one or more *inotify_event* structures containing information about the event that occurred on the pathname being watched via *inotify* instance.
4. When the application has finished monitoring, it closes *inotify* file descriptor. This auto removes all watch items associated with the *inotify* instance.

1.3.1 The *inotify* API

The *inotify_init()* system call creates a new *inotify* instance.

```
#include <sys/inotify.h>
// below function returns a file descriptor on success or -1 on error
int inotify_init(void);
```

The file descriptor is a handle that is used to refer to *inotify* instance in subsequent operations.

The *inotify_add_watch()* system call either adds a new watch item to or modifies an existing watch item in the watch list for the *inotify* instance referred to by the file descriptor *fd*.

```
#include <sys/inotify.h>
// below function returns watch descriptor on success or -1 on error
int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
```

The *pathname* argument identifies the file for which a watch item is created or modified. The caller must have read permissions for the file. The *mask* argument is a bit mask that specifies the events to be monitored for the file.

The *inotify_rm_watch()* system call removes the watch item specified by *wd* from the *inotify* instance referred to by the file descriptor *fd*.

```
#include <sys/inotify.h>
// below function returns 0 on success or -1 on error
int inotify_rm_watch(int fd, uint32_t wd);
```

wd is a watch descriptor returned by a previous call to *inotify_add_watch()*. The *uint32_t* is an unsigned 32-bit integer data type.

1.3.2 *inotify* Events

When we create or modify a watch using *inotify_add_watch()*, the *mask* bit-mask argument identifies the event(s) to be monitored for the given *pathname*. There are more than 20 events, however, the one we need for this assignment is called **IN_MODIFY**. This event is generated when file is modified.

1.4 Reading *inotify* Events

Having registered items in the watch list, an application can determine which events have occurred by using *read()* to read events from the *inotify* file descriptor. If no event has occurred so far, then *read()* blocks until an event occurs.

After events have occurred, each *read()* returns a buffer containing one or more structures of the following type:

```
struct inotify_event {
    int wd; // Watch descriptor on which event occurred
    uint32_t mask; // Bits describing event that occurred
    uint32_t cookie; // Cookie for related events;
    uint32_t len; // Size of 'name' field
    char name[]; // Optional null-terminated filename
}
```

wd identifies the watch descriptor, as obtained by *inotify_add_watch()*, and *mask* represents the events. If *wd* identifies a directory and one of the watched-fo events occurred on a file within that directory, *name* provides the filename relative to the directory. In this case, *len* is *0*. If we are watching a file directly, *len* would be *0* and name would be zero-length. In this case, we should not touch it. As, we are watching the *access_points.json* directly, the latter case applies and we won't touch the *name* or *len* field. For this coding task, we will ignore the *cookie* field as well as it is used to link together two related but disjoint events. For now, we will ignore this field.

1.5 First Program - *appA.c*

Following *C* program monitors the file *acess_points.json*'s contents for modifications and generates an **IN_MODIFY** event on any changes in the file.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/inotify.h>
#include <limits.h>
#include <unistd.h>

 // function prototype to display information from inotify_event structure
```

(continues on next page)

(continued from previous page)

```c
static void displayInotifyEvent(struct inotify_event *ptr);

#define BUF_LEN (10 * (sizeof(struct inotify_event) + NAME_MAX + 1))

int main(int argc, char *argv[]) {
    int fd = 0, wd = 0;
    char buf[BUF_LEN];
    ssize_t nRead;
    char *p = NULL;
    struct inotify_event *event = NULL;

    if (argc < 2 || strcmp(argv[1], "--help") == 0) {
        fprintf(stderr, "Usage: %s filepath \n", argv[0]);
        return EXIT_FAILURE;
    }

    // create inotify instance
    fd = inotify_init1(0);

    if (fd < 0) {
        perror("inotify_init1");
        exit(EXIT_FAILURE);
    }

    // add a watch for file modification event
    wd = inotify_add_watch(fd, argv[1], IN_MODIFY);

    if (wd < 0) {
        perror("inotify_add_watch");
        exit(EXIT_FAILURE);
    }
    else {
        fprintf(stdout, "Watching %s using wd %d\n", argv[1], wd);
    }

    while(1) { // read events indefinitely
        nRead = read(fd, buf, BUF_LEN);
        if (nRead < 0) {
            perror ("read");
            exit(EXIT_FAILURE);
        }
        if (nRead == 0){
            perror("read() from inotify fd returned 0!");
        }
        fprintf(stdout, "Read %zd bytes from inotify fd\n", nRead);

        p = buf;
        while ( p < buf + nRead ) {
            event = (struct inotify_event *) p;
```

(continues on next page)

(continued from previous page)

```
            displayInotifyEvent(event);
            p += sizeof(struct inotify_event);
        }
    }
    exit(EXIT_SUCCESS);
}

static void displayInotifyEvent(struct inotify_event *ptr) {
    printf("wd = %d; ", ptr->wd);
    printf("mask = ");
    if (ptr->mask & IN_MODIFY) {
        printf("IN_MODIFY \n");
        printf("File was written to!\n");
    }
}
```

The process to monitor the file is as follows:

```
$ gcc -Wall -o appA appA.c # compile appA
$ ./appA access_points.json # start monitoring access_points.json
```

Open another terminal window and introduce any changes in the file *access_points.json*. File modification event will be generated as follows:

```
$ ./appA access_points.json
Watching access_points.json using wd 1
Read 16 bytes from inotify fd
wd = 1; mask = IN_MODIFY
File was written to!
Read 16 bytes from inotify fd
wd = 1; mask = IN_MODIFY
File was written to!
```

File modification event is picked up by the *appA*. All the code and testing is carried out on *Fedora* Linux. Due to lack of time, I couldn't verify the results on Ubuntu.

1.6 Second Program - *appB.c*