Discussion 2: Threads, I/O

February 4, 2022

Contents

1 Threads

Threads are single unique execution contexts with its own set of registers and stack. Threads are sometimes referred to as “lightweight processes”.

Thread Control Block

Components that are shared between threads in the same process (e.g. heap, global variables) do not be persisted by each individual thread. However, the thread still needs to persist registers and its stack in the **thread control block (TCB)**.

Syscalls

POSIX defines a pthread library which are syscalls for threads similar to the process syscalls.

`int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)`

Starts a new thread in the calling process. The thread starts execution by invoking `start_routine` that takes in `arg` as its sole argument.

`void pthread_exit(void *retval)`

Terminates the calling thread and stores a value in `retval` that is available to another thread in the same process that joins. Similar to how a return from `main` will implicitly call `exit` for processes, a return from `start_routine` will implicitly call `pthread_exit`.

`int pthread_yield(void)`

Calling thread relinquishes the CPU and is put at the end of the run queue.

`int pthread_join(pthread_t thread, void **retval)`

Calling thread waits for `thread` to terminate. For a non-`NULL` `retval`, the exit status of `thread` (i.e. the one from `pthread_exit`) is made available in `retval`.

1.1 pthreads Party

1. What are the possible outputs of the following program? How can you change the program to make it print "HELPER" before "MAIN"?

```
void *helper(void *arg) {
  printf("HELPER");
  return NULL;
}
int main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, NULL);
  pthread_yield();
  printf("MAIN");
  return 0;
}
```

pthread_order.c

The output of the program can be "MAINHELPER", "HELPERMAIN", or "MAIN". The `pthread_yield` does not guarantee that `helper` will be run first. While the main thread will relinquish the CPU, there is no guarantee that the entire helper thread executes before the CPU is given back to the main thread. It might even happen that the main thread runs all the way through before the helper thread ever runs the print statement. When the main thread returns, the process will return as well since it exits from the `main` function. This will kill the process and thereby its associated threads.

A simple fix is to replace the pthread_yield with a pthread_join. With this change, the main thread will wait until the helper thread has returned, meaning "HELPER" will always be printed before "MAIN".

2. What does the following program print?

```c
void *helper(void *arg) {
  int *num = (int*) arg;
  *num = 2;
  return NULL;
  }
int main() {
  int i = 0;
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, &i);
  pthread_join(thread, NULL);
  printf("i is %d", i);
  return 0;
}
```

pthread_stack.c

Since both threads are in the same process, they share the same address space. Even though each thread will have a separate stack, a shared address space means threads in the same process can access each others' stacks. As a result, helper thread will modify i in the main thread, giving an output of "i is 2".

3. What does the following program print?

```c
void *helper(void *arg) {
  char *message = (char *) arg;
  strcpy(message, "I am the helper");
  return NULL;
}
int main() {
  char *message = malloc(100);
  strcpy(message, "I am the main");
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, message);
  printf("%s", message);
  pthread_join(thread, NULL);
  return 0;
}
```

pthread_heap.c

The program can print either "I am the main" or "I am the helper". pthread_join is called after the printf, meaning the order of execution is not guaranteed by the printf statement.

2 I/O

The operating system must be able to interface with a wide variety of **input/output (I/O)** devices including but not limited to keyboard, mouse, disk, USB port, WiFi. While there are many different types of devices, POSIX unifies all of them behind a single common interface.

Design Philosophy

POSIX has a set of design philosophies that define the standard for UNIX-based operating systems.

Uniformity
All I/O is regularized behind a single common interface under the idea that "everything is a file". This means all I/O (e.g. file operations, interprocess communications) use the same set of syscalls: `open`, `close`, `read`, `write`.

Open Before Use
Before any I/O operation, the "device" must be opened before using. This allows the operating system to perform various checks (e.g. access control permissions) and bookkeeping of metadata. For instance, a device might only be allowed to be opened by one application.

Byte Oriented
All data from devices are addressed in bytes even for devices with different block sizes.

Kernel Buffered Read/Writes
Data from devices are stored in a kernel buffer and returned to the application on request. Similarly, outgoing data is stored in a kernel buffer until the device is ready to be written to. Kernel buffering allows for same interface between devices with different read block sizes and different write speeds.

Explicit Close
An application must explicitly call `close` when it's done with the device. Similar to `open`, this allows the operating system to perform bookkeping and garbage collection.

Low-Level API

Low-level API operates directly with **file descriptors** which are indices into a **file descriptor table (FDT)**. Each process has its own FDT stored in kernel memory, where each entry poitns to a **file description** which represents an open file or device and keeps track of metadata (e.g. offset position). This means that the same file descriptor can correspond to different file descriptions across different processes.

Specifically, file descriptors 0, 1, and 2 are initialized to standard input (stdin), standard output (stdout), and standard error (stderr), respectively. However, they can be redefined (see `dup` and `dup2`).

Select methods from the low-level API are listed below. All low-level methods are syscalls, meaning they occur entirely in kernel space.

`int open(const char*pathname, int flags)`
Opens a file specified by `pathname` with access control permisions `flags`. Returns a file descriptor or -1 on an error.

`int close(int fd)`
Closes a file descriptor, so that it no longer refers to any file and may be reused. If `fd` is the last file descriptor referring to the file description, the resources associated with the open file description are freed. Returns 0 on success or -1 on an error.

`ssize_t read(int fd, void *buf, size_t count)`
Attempts to read up to `count` bytes from file descriptor `fd` into the buffer starting at `buf`. Returns the number of bytes read on success or -1 on an error.

`ssize_t write(int fd, const void *buf, size_t count)`
: *Attempts* to write `count` bytes to file descriptor `fd` from the buffer starting at `buf`. Returns number of bytes written on success or -1 on an error.

`off_t lseek(int fd, off_t offset, int whence)`
: Repositions the file offset of the open file description associated with the file descriptor `fd` to the argument `offset` according to the directive `whence`.

`int dup(int oldfd)`
: Allocates the next available file descriptor to point to the same file description as `oldfd`.

`int dup2(int oldfd, newfd)`
: Same as `dup` but with a specific file descriptor `newfd`.

A particular note about `read` and `write` is that these methods are not guaranteed to fully succeed, meaning `read` and `write` might read and write less bytes than specified. However, this isn't considered an error.

High-Level API

High-level API operates on **streams** of data which are unformatted sequence of bytes with a position. Streams can encompass any data type from raw binaries to text. An open stream is represented by a pointer to `FILE`, which contain information pertaining to an open stream such as the file descriptor. For example, `FILE *stdin`, `FILE *stdout`, and `FILE* stderr` respectively correspond to file descriptors 0, 1, and 2.

High-level API buffers data in *user memory* in larger chunks (e.g. 1024 bytes). This is different from the kernel buffered philosophy since not every invocation of the high-level API requires a syscall. For instance, a `fread` with `size = 100` will still fetch the data in a larger chunk (e.g. 1024 bytes), meaning there will not be a syscall when `fread` is called a second time with `size = 100`.

Select methods from the high-level API are listed below.

`FILE *fopen(const char *pathname, const char *mode)`
: Opens the file associated with `pathname` and associates a stream with it. Returns a `FILE *` or `NULL` on an error.

`int fclose(FILE *stream)`
: Flushes the stream pointed to by `stream` and closes the underlying file descriptor. Returns 0 on success or `EOF` on an error.

`int fread(void *ptr, size_t size, size_t nmemb, FILE *stream)`
: Reads `nmemb` items of data, each `size` bytes long, from the stream pointed to by `stream`, storing them at the location given by `ptr`. Return the number of *items* read.

`int write(void *ptr, size_t size, size_t nmemb, FILE *stream)`
: Writes `nmemb` items of data, each `size` bytes long, to the stream pointed to by `stream`, obtaining them from the location given by `ptr`. Returns the number of *items* written.

`int fflush(FILE *stream)`
: Forces a write of all data in the user space buffer to `stream`.

`int fprintf(FILE *stream, const char *format, ...)`
: Prints to `stream` according to `format`.

`int fscanf(FILE *stream, const char *format, ...)`
: Scans from `stream` according to `format`.

Contrary to `read` and `write`, `fread` and `fwrite` guarantee that they will read and write the specified amount. If `fread` and `fwrite` return an amount less than the specified amount, it is considered an error.

In general, high-level APIs provide a more expressive way of interacting with devices. For instance, `fprintf` allows printing with a specific format where as `write` can only write raw sequence of bytes. There are also different API methods defined for character oriented devices (e.g. `fputc`) and block oriented devices (e.g. `fread`).

Interprocess Communication

In theory, processes can communicate using files, where one process writes to a file and another reads from a file. However, this is a painfully slow process due to the overhead of disk speeds. **Interprocess communciation (IPC)** provide mechanisms to communicate between processes to manage shared data. Often times, this communication will be facilitated using memory instead of disk.

Pipes are one-way communication channels between processes on the same physical machine. A pipe can be thought of as using a single queue where you can read from one end and write to another. When opened, each end of the queue corresponds to its own file descriptor. You can use `int pipe(int pipedfd[2])` syscall to create a pipe.

Sockets are two-way communication channels between processes. These processes can be on the same or different machines. Sockets can be thought of as using two queues, one in each direction.

For a connection to to be established, the server needs to be able to accept a request connection from the client. The steps to do this are

1. Create server socket using `socket` syscall.
2. Bind the server socket to a specific address using `bind` syscall.
3. Start listening for connections using `listen` syscall.

Once these steps succeed, the server can accept new connections using the `accept` syscall. When the client wants to make a connection request, it sends a 5-Tuple that uniquely identifies a connection. In the 5-Tuple are

1. Source IP Address
2. Destination IP Address
3. Source Port Number
4. Destination Port Number
5. Protocol (e.g. TCP)

2.1 Concept Check

1. What's the difference between `fopen` and `open`?

   `fopen` is a high-level API, while `open` is a low-level API. `fopen` will return a `FILE*` type, while `open` will return an integer.

2. What will the `test.txt` file look like after this program is run? You may assume `read` and `write` fully succeed (i.e. read/write the specified number of bytes).

   For reference, `SEEK_SET` will set the offset to `offset` bytes, while `SEEK_CUR` will set the offset to its current location plus `offset` bytes. Seeking past the end of a file will set the bytes to 0.

```
int main() {
  char buffer[200];
  memset(buffer, 'a', 200);
  int fd = open("test.txt", O_CREAT|O_RDWR);
  write(fd, buffer, 200);
  lseek(fd, 0, SEEK_SET);
  read(fd, buffer, 100);
```

```c
  lseek(fd, 500, SEEK_CUR);
  write(fd, buffer, 100);
}
```

lseek_test.c

The first write gives us 200 bytes of 'a'. Then we seek to the offset 0 (i.e. beginning of the file) and then read 100 bytes. This will set the current offset to 100.

We seek again to offset 600 (500 bytes on top of existing 100) then write 100 more bytes of a. Since we seek past the end of the file, the 200 to 599th bytes will be set to 0. An important distinction is that this is not the character '0' but rather the literal byte 0.

2.2 File Descriptor Fun

1. Consider a method that copies copies n bytes from src file to dest file. You may assume both files are already created, and src is at most 100 bytes long. This method has been written in two different ways, one using the high-level API and low-level API.

```c
void copy_high(const char* src, const char* dest) {
  char buffer[100];
  FILE* rf = fopen(src, "r");
  int buf_size = fread(buffer, 1, sizeof(buffer), rf);
  fclose(rf);
  FILE* wf = fopen(dest, "w");
  fwrite(buffer, 1, buf_size, wf);
  fclose(wf);
}

void copy_low(const char* src, const char* dest) {
  char buffer[100];
  int rfd = open(src, O_RDONLY);
  int buf_size = 0;
  while ((bytes_read = read(rfd, &buffer[buf_size], sizeof(buffer) - buf_size)) > 0)
    buf_size += bytes_read;

  close(rfd);
  int bytes_written = 0;
  int wfd = open(dest, O_WRONLY);
  while (bytes_written < buf_size)
    bytes_written += write(wfd, &buffer[bytes_written], buf_size - bytes_written);

  close(wfd);
}
```

copy.c

What is the purpose of the while loops in copy_low?

Contrary to fread and fwrite, read and write are not guaranteed to read and write the specified size. As a result, we need the while loop to make sure the desired number of bytes are read and written.

2. What does the following program print to standard out?

```c
int main(int argc, char **argv) {
  int fd;
  if ((fd = open("out.txt", O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0)
    exit(1);

  printf("The last digit of pi is ...");
  fflush(stdout);

  dup2(fd, 1);
  printf("five");
  exit(0);
}
```

dup_pi.c

> Standard out will only print
>
> `The last digit of pi is ...`
>
> since we replaced the file descriptor 1 was remapped to correspond to the file description corresponding to the open `out.txt`. `"five"` will end up in `out.txt`.

2.3 Echo Server

Consider a server implementation that echoes back the bytes that the client sent. Each connection needs to be handled its own thread. Threads should be allowed to handle connections concurrently. For simplicity assume `read` and `write` do not return short.

```
#define BUF_SIZE 1024

struct addrinfo* setup_address(char* port) {
  struct addrinfo* server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;

  int rv = getaddrinfo(NULL, port, &hints, &server);
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}

void* serve_client(void* client_socket_arg) {
  int client_socket = (int) client_socket_arg;
  char buf[BUF_SIZE];
  ssize_t n;

  while ((n = read(client_socket, buf, BUF_SIZE)) > 0) {
    buf[n] = '\0';
    printf("Client Sent: %s\n", buf);

    if (write(client_socket, buf, n) == -1) {
      ____________________;
      ____________________;
    }
  }
  ____________________;
  ____________________;
}

int main(int argc, char** argv) {
  if (argc < 2) {
    printf("Usage: %s <port>\n", argv[0]);
    return 1;
  }

  struct addrinfo* server = setup_address(argv[1]);
  if (server == NULL)
    return 1;
  int server_socket = ______(server->ai_family, server->ai_socktype, server->ai_protocol);
  if (server_socket == -1)
    return 1;
  if (_____(server_socket, server->ai_addr, server->ai_addrlen) == -1)
    return 1;
  if (_____(server_socket, 1) == -1)
    return 1;

  while (1) {
    int connection_socket = accept(server_socket, NULL, NULL);
    if (connection_socket == -1)
      pthread_exit(NULL);

    pthread_t handler_thread;
    int err = pthread_create(____________________);
    if (err != 0)
      ____________________;
    pthread_detach(handler_thread);
  }
}
```

echo_server.c

1. What are the first three steps that a server needs perform to be able to accept new connections? Specify the specific syscalls that need to be used.

   The server needs to create a socket using `socket` syscall, bind it to an address using `bind` syscall, and start listening for new connections using `listen` syscall.

2. What function should each thread start at (i.e. `start_routine` argument in `pthread_create`)?

   `serve_client`. It has the basic outline of reading what the client sent and writing it back.

3. What should the server do when it's finished with a client according to POSIX design philosophies?

   An explicit close of the client socket is required.

4. What are the dangers of this approach compared to making a new process for each connection?

   Each connection is handled in the same process, meaning a malicious connection could attack the server and make the server and other clients vulnerable. If every connection was handled in a new process, the server and other clients would still be safe.

5. Fill in the blanks to complete the implementation.

```c
void* serve_client(void* client_socket_arg) {
  int client_socket = (int) client_socket_arg;
  char buf[BUF_SIZE];
  ssize_t n;

  while ((n = read(client_socket, buf, BUF_SIZE)) > 0) {
    buf[n] = '\0';
    printf("Client Sent: %s\n", buf);

    if (write(client_socket, buf, n) == -1) {
      close(client_socket);
      pthread_exit(NULL);
    }
  }
  close(client_socket);
  pthread_exit(NULL);
}

int main(int argc, char** argv) {
  if (argc < 2) {
    printf("Usage: %s <port>\n", argv[0]);
    return 1;
  }

  struct addrinfo* server = setup_address(argv[1]);
  if (server == NULL)
    return 1;
  int server_socket = socket(server->ai_family, server->ai_socktype, server->ai_protocol);
  if (server_socket == -1)
    return 1;
  if (bind(server_socket, server->ai_addr, server->ai_addrlen) == -1)
    return 1;
  if (listen(server_socket, 1) == -1)
    return 1;

  while (1) {
    int connection_socket = accept(server_socket, NULL, NULL);
    if (connection_socket == -1)
      pthread_exit(NULL);

    pthread_t handler_thread;
    int err = pthread_create(&handler_thread, NULL, serve_client, connection_socket);
    if (err != 0)
      pthread_exit(NULL);
    pthread_detach(handler_thread);
  }
}
```

echo_server.c