Discussion 2

Threads, I/O

02/02/24

Staff

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
			Project 1 Release			
			Homework 1 Due	Homework 2 Release		Project 1 Design Doc Due
				Midterm 1 (7-9 PM)		

Project Timeline

1. Work on design document.
2. Submit design document.
3. TA reads through design document.
4. Meet with TA for a 30-minute design review.
5. Start coding.
6. Submit code, report, evaluations.

Project Advice

Start early.

- Low-level programming has high variance of completion time.
- OH will get crowded and you shouldn’t expect to get a working solution through OH.

Read through the spec in its entirety multiple times.

- Don’t have to remember or understand everything.

Take design document and review seriously.

- Good design can save hours of debugging and refactoring.
- Come to design review prepared to answer and ask questions.

Meet regularly, ideally in-person.

- Take good notes to keep track of different ideas and approaches.
- **Notify TA right away if you’re having group dynamic issues.**

Use software efficiently.

- VS Code Live Share
- Dropbox Paper
- Zoom Audio Transcription

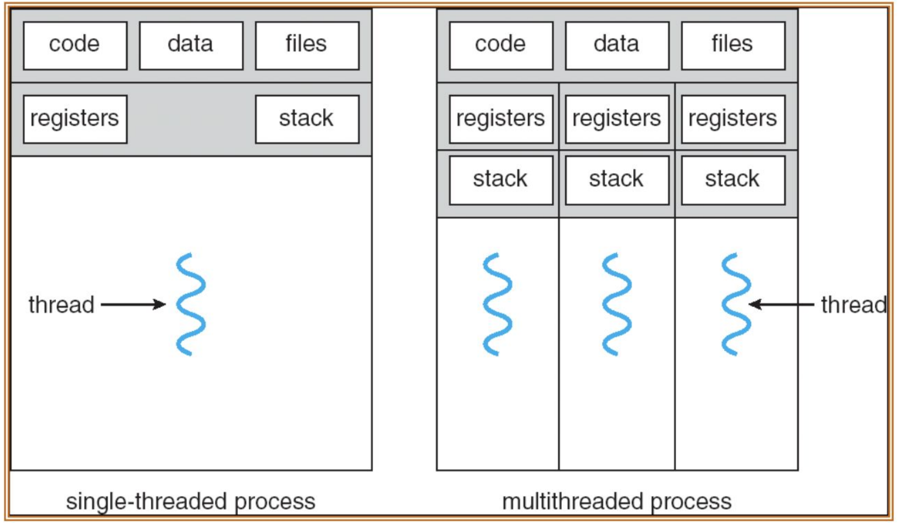
Project Design Document

- Please do not submit 30+ page long design documents—it should not include any large blocks of code!
- Hard ceiling of 15 pages

Threads

Threads

- Threads are single unique execution contexts with their own set of registers and stack.
- They are sometimes referred to as “lightweight processes”.
- Components that are shared between threads in the same process (e.g. heap, global variables) do not need to be persisted by each individual thread.
- A thread still needs to persist registers and its stack in the thread control block (TCB).



Syscalls

POSIX defines a pthread library for syscalls, 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.
- Thread starts execution by invoking `start_routine` that takes in `arg` as its sole argument.
    - Allows for a general interface.
    - Often `arg` will be a struct which you will cast in `start_routine`.

`void pthread_exit(void *retval)`

- Terminates the calling thread.
- `start_routine` will implicitly call `pthread_exit` similar to how `main` implicitly calls `exit`.

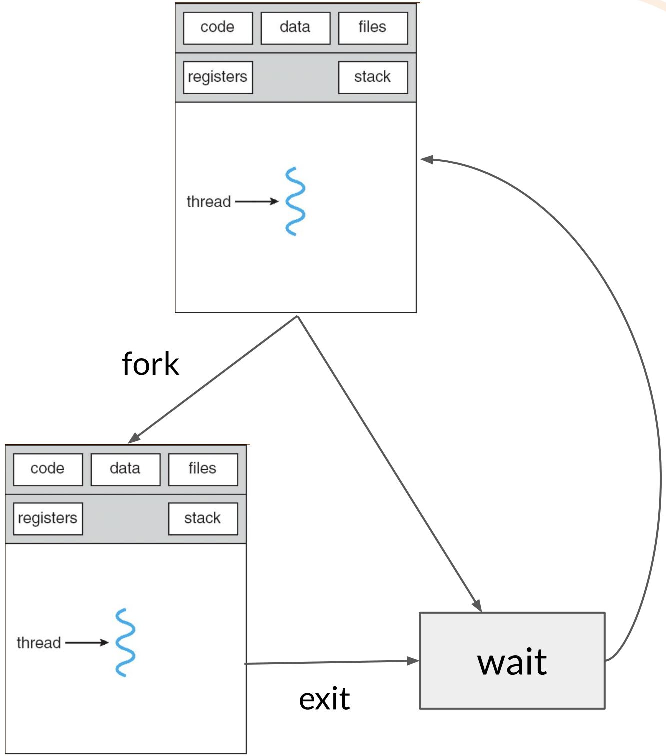
`int sched_yield(void)`

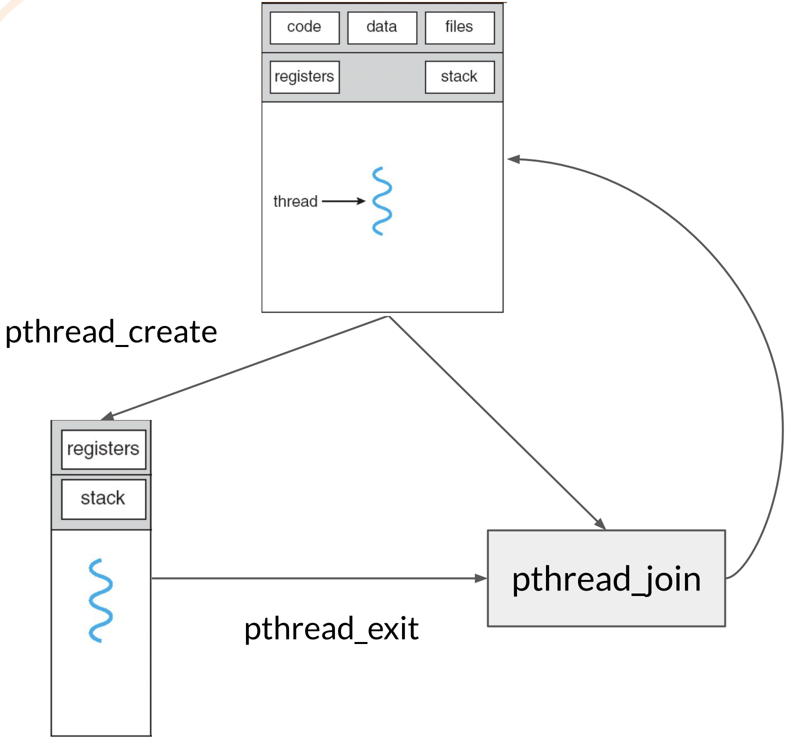
- Calling thread relinquishes the CPU and is put at the end of the run queue.

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

- Calling thread waits for `thread` to terminate.

Processes vs Threads





Warm-up Question!

```
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;
}
```

What does the following program print?

Warm-up Question!

```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;
}
```

What does the following program print?

“`i is 2`” will be printed since both threads share the same address space as part of the same process. Even though each thread has its own stack, it can still access each others’ stacks.

pthreads Party

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

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

pthreads Party

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

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

pthreads Party

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

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

Output can be

- “MAINHELPER”
- “HELPERMAIN”
- “MAIN”

since no guarantee that helper will be run first (even with **`sched_yield`**).

pthreads Party

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

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

Output can be

- "MAINHELPER"
- "HELPERMAIN"
- "MAIN"

since no guarantee that helper will be run first (even with `sched_yield`).

Use `pthread_join` instead of `sched_yield` to ensure main thread will wait until the helper thread has returned.

pthreads Party

2. What does the following program print?

```
void* helper(void *arg) {
  int* num = (int*) arg;
  printf("%d", *num);
  return NULL;
}
void spawn_thread(void) {
  int i = 162;
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, &i);
  return;
}
int main() {
  spawn_thread();
  return 0;
}
```

pthreads Party

```
void* helper(void *arg) {
  int* num = (int*) arg;
  printf("%d", *num);
  return NULL;
}
void spawn_thread(void) {
  int i = 162;
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, &i);
  return;
}
int main() {
  spawn_thread();
  return 0;
}
```

2. What does the following program print?

This results in undeterministic behavior. It's problematic in that the stack space gets reclaimed after spawn_thread returns in the main thread, so it's possible that the new thread can be accessing garbage memory.

pthreads Party

```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;
}
```

3. What does the following program print?

pthreads Party

```
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;
}
```

3. What does the following program print?

Either “`I am the main`” or “`I am the helper`” since `pthread_join` is called after `printf`.

I/O

Design Philosophy

Set of design philosophies from POSIX for all UNIX-based OS

Uniformity

- All I/O is regularized behind a single common interface.
- “Everything is a file”.

Open Before Use

- Device/file must be opened before any I/O operation.
- Allows OS to perform various checks, metadata bookkeeping.

Byte Oriented

- All data from devices are addressed in bytes (even for devices with different block sizes)

Kernel Buffered Reads/Writes

- Data from devices/files stored in kernel buffer and returned to application on request.
- Outgoing data is stored in a kernel buffer until the device/file is ready to be written to.

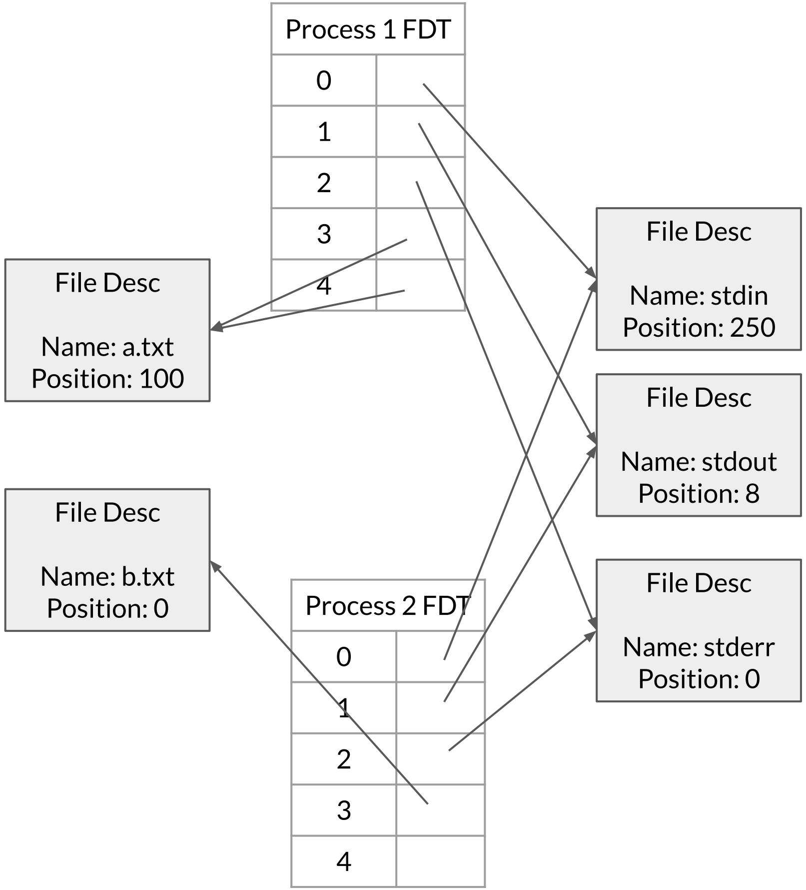
Explicit Close

- Need to explicitly close devices/files for bookkeeping, garbage collection.

Low-Level API

Operate directly with **file descriptors** which are indices into the **file descriptor table (FDT)**.

- File descriptor tables are per process.
- Each entry in the FDT points to a **file description** which represents an open file, keeping track of metadata (e.g. offset position).
- Same file *descriptor* can correspond to different file *descriptions* across different processes.
- File descriptors 0, 1, and 2 initialized to stdin, stdout, stderr but can be reset (see dup/dup2).



Low-Level API

`int open(const char*pathname, int flags)`

- Opens a file specified by pathname with access control permission flags.
- Returns a file descriptor or -1 on error.

`int close(int fd)`

- Closes a file descriptor, so it can be reused.
- File description is freed if `fd` is the last one pointing to it.

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

- *Attempts* to read up to count bytes from `fd` into buffer starting at `buf`.
- Returns 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 `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`.

`int dup(int oldfd)`

- Allocates the next available file descriptor to point to the same file description as `oldfd`.

`int dup2(int oldfd, int newfd)`

- Same as `dup` but with a specific file descriptor `newfd`.

High-Level API

Operate on **streams** of data which are unformatted sequences of bytes.

- Streams can be any data type (e.g. raw binary, text).
- **FILE*** represents an open stream and corresponds to a file descriptor.
    - Opaque type meaning you can't access the struct members.

Buffers data in *user memory* (different from kernel buffering).

- Usually in larger chunks (e.g. 1024 bytes).
- Not every invocation of high-level API will require a syscall (i.e. low-level API).

Generally more expressive and user friendly than low-level API.

High-Level API

`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 fwrite(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.

Concept Check

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

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).

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

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` returns 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).

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

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` returns 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).**

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

1. Writes 200 bytes of 'a'.

2. Reset offset to 0 and read 100 bytes, putting the offset at 100.

3. Seek offset to 600 and then write 100 more bytes of 'a'.

Bytes 0-199 are 'a', 200-599 are null bytes (different from the actual letter '0'), 600-699 are 'a'.

File Descriptor Fun

```
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);
}
```

1. Consider a method that 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.

   What is the purpose of the `while` loops in `copy_low`?

File Descriptor Fun

```
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);
}
```

1. Consider a method that 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.

   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.

Need the while loop to make sure desired size is read/written.

File Descriptor Fun

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

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

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

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

File Descriptor Fun

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

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

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

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

Standard out will only print

`The last digit of pi is …`

since we replaced file descriptor 1 (i.e. standard out) to point to the same file description as the one corresponding to file descriptor `newfd`. “`five`” will end up in `out.txt`.

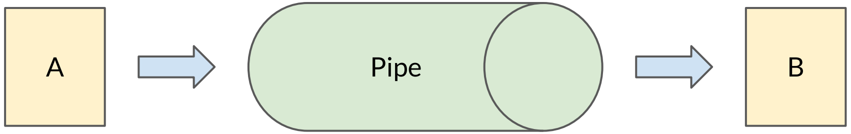
Interprocess Communication

Interprocess communication (IPC) provide mechanisms to communicate between processes to manage shared data.

- Could theoretically use files on disk, but painfully slow.
- IPC uses memory instead of disk.

Pipes are one-way communication channels between processes on the same physical machine.

- Single queue where you can read from one end and write to another.
- Use `int pipe(int pipedfd[2])` syscall to create a pipe.



Interprocess Communication

Sockets are two-way communication channels between processes.

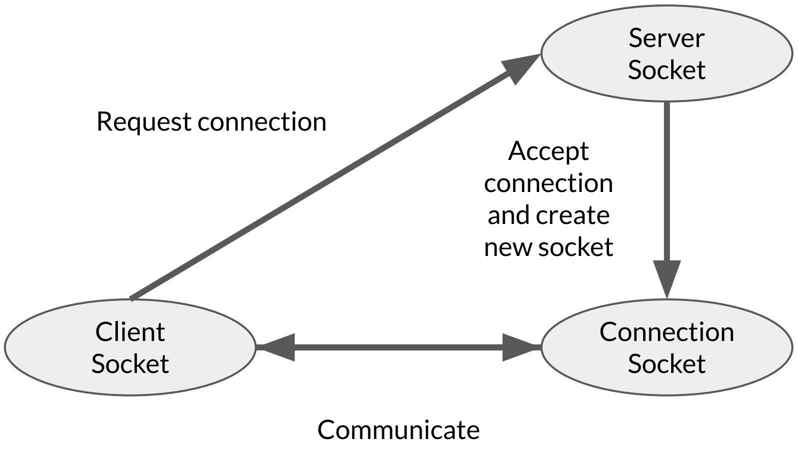
- Not necessarily limited to processes within the same machine.
- Uses two queues, one in each direction.
- Server socket is special in that it's only meant to accept connection requests (i.e. no read/writes allowed)

Server socket setup is

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

Can accept new connections using `accept` syscall.

- Client sends a 5-tuple uniquely identifying a connection.
    - Source IP address
    - Destination IP address
    - Source port number
    - Destination port number
    - Protocol (e.g. TCP)



Echo Server

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

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

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

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

Echo Server

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`)?

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

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

Echo Server

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?

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

Echo Server

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?

Echo Server

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.

Echo Server

```
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);
  }
}
```

5. Fill in the blanks to complete the implementation.

Echo Server

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

5. Fill in the blanks to complete the implementation.

Configure the server socket by creating, binding, and listening per question 1.

Echo Server

```
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);
  }
}
```

5. Fill in the blanks to complete the implementation.

Create a new thread for each request that runs `serve_client`.

On error, exit main thread only (i.e. use `pthread_exit` instead of `exit`) to allow other connections to finish execution.

Echo 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) {
      ____________________;
      ____________________;
    }
  }
  ____________________;
  ____________________;
}
```

5. Fill in the blanks to complete the implementation.

Echo 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) {
      close(client_socket);
      pthread_exit(NULL);
    }
  }
  close(client_socket);
  pthread_exit(NULL);
}
```

5. Fill in the blanks to complete the implementation.

Must close client socket on exit by POSIX design philosophy (i.e. explicit close).