Operating Systems
CS 415

Lecture 6:

Interprocess Communication

Suyash Gupta

Assistant Professor

Distopia Labs and ONRG

Dept. of Computer Science

(E) suyash@uoregon.edu

(W) gupta-suyash.github.io

UNIVERSITY OF OREGON

Announcements

- **Suyash Gupta**
  - Office Hours: Thursday, 10-11am, Deschutes 334

- **Nihal Balivada (TA)**
  - Office Hours: Wednesday/ Friday, 11-12pm, Deschutes 335

- **Ranjitha Rani (TA)**
  - Office Hours: Monday /Tuesday, 1-2pm, Deschutes 229

Assignment 1 is Out!

- **Deadline → April 21, 2026 at 11:59pm PST**
- Please start working and talk to us if you are stuck.

- **Midterm → April 28, 2026 (in-class)**
  - Closed book, no cheat sheets, no discussions.

Last Class

- Process Scheduling
- Process Creation and Termination

Independent and Cooperating Processes

Independent and Cooperating Processes

- Processes can be either independent or cooperating with respect to each other.
- They are **independent** if neither can affect or be affected by the execution of the other process.
- They are **cooperating** if either can affect or be affected by the execution of the other process.
- Why processes to cooperate?
  - To share information.
  - To speed up a computation.
  - To increase modularity of an application.

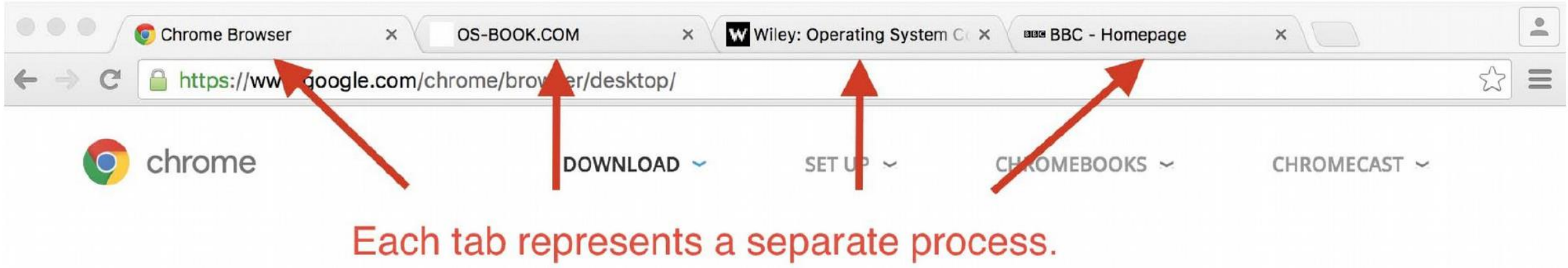
Google Chrome

Google Chrome

- Many web browsers ran as single process (some still do).
- If one web site causes trouble, entire browser can hang or crash

- Google Chrome Browser is multi-process with 3 different types of processes:
  - **Browser process** → manages user interface, disk and network I/O
  - **Renderer process** → renders web pages, deals with HTML, Javascript.
    - A new renderer created for each website opened.
    - Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits.
  - **Plug-in process** → for each type of plug-in.

Google Chrome



Use case of Interprocess Communication

Use case of Interprocess Communication

- Client–Server Model
- Pipelines and Data Processing
- Background Services
- Parallelism / Multiprocessing
- Synchronization Across Processes
- System Event Notifications
- Processes manage physical devices

Requirements for IPC?

Requirements for IPC?

- We want some sort of messaging system.
- IPC would provide 2 operations to processes:
  - **Send(message)** → sender process to receiver process.
  - **Receive(message)** → receiver process from sender process.
- There also needs to be a **communication channel**.
  - If processes P and Q wish to communicate, they open a channel.
  - Then the exchange of messages can proceed.

Communication Channel

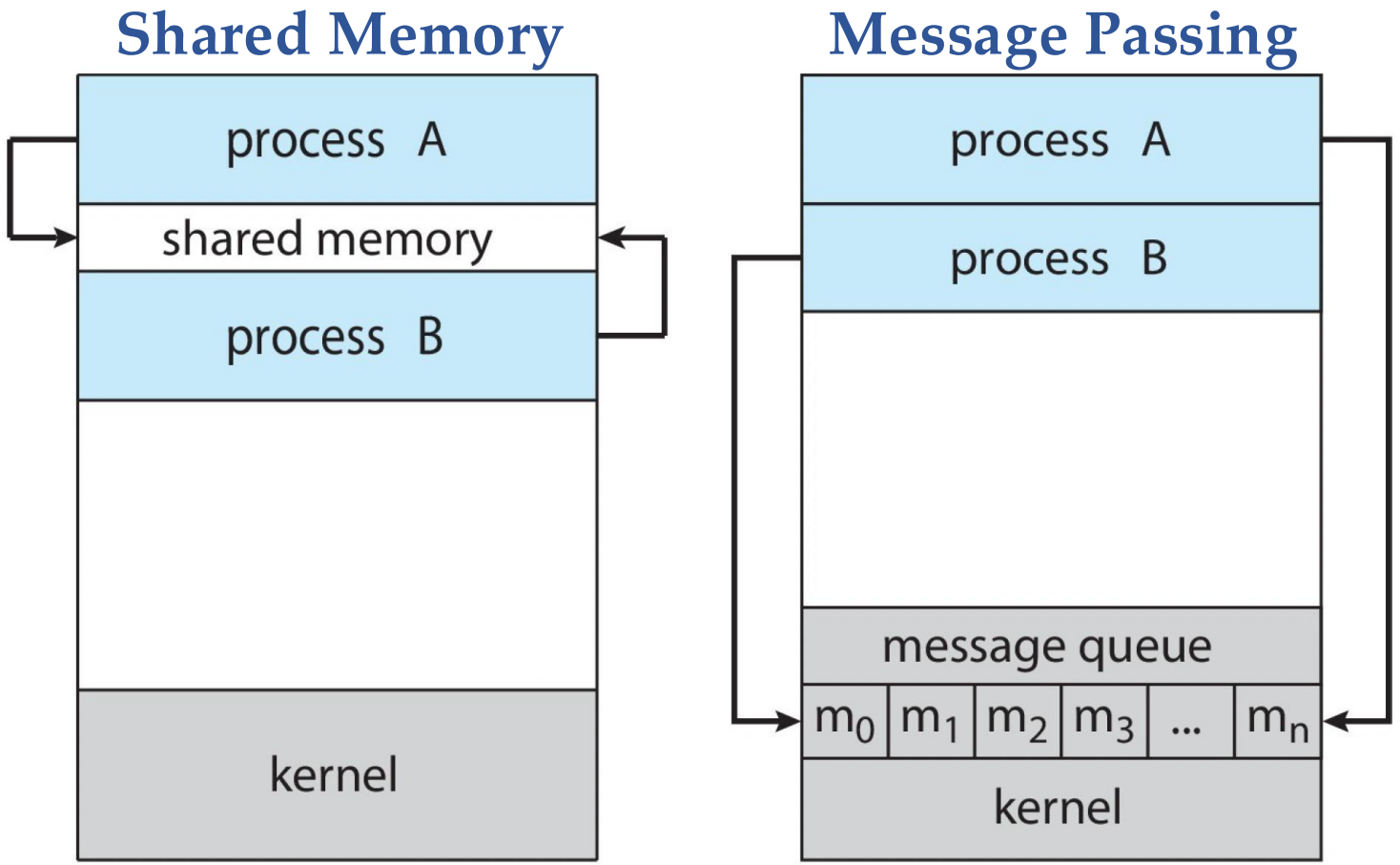
Communication Channel

- How is the communication channel (link) realized?
- Physically, many forms → memory, storage, networks.
- Logically, need to define:
  - abstract interfaces and properties
  - protocols for correct communication

Shared Memory vs. Message Passing

Shared Memory vs. Message Passing

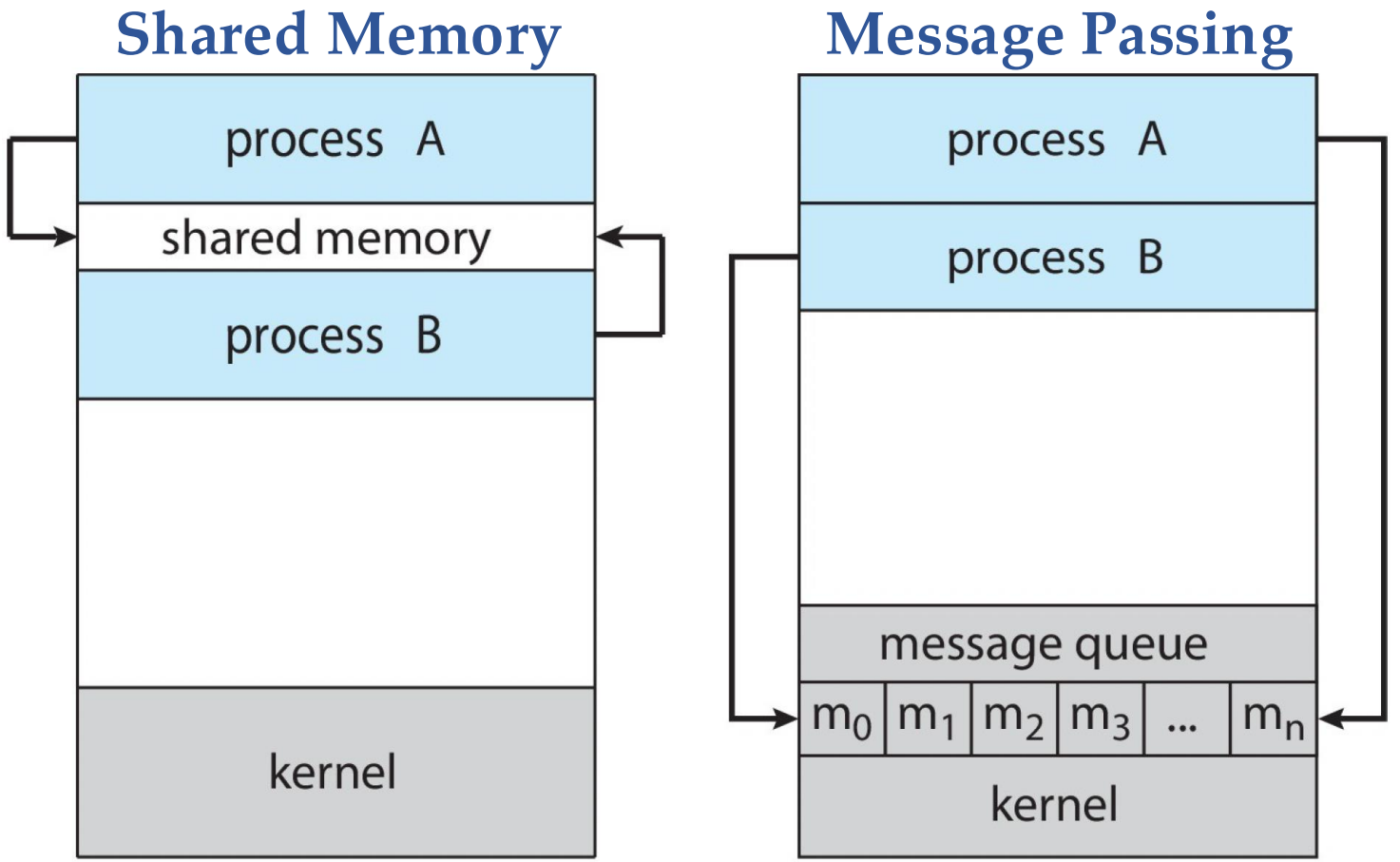
- Two popular communication models supported by OS.



Shared Memory vs. Message Passing

- Two popular communication models supported by OS.

A memory area is created by the OS such that each process can reference the data using the same subset of addresses



The kernel maintains the message buffers in its memory. These buffers hold messages sent between processes.

Properties of IPC

- Every IPC method is a combination of the following properties:

- **Direct or indirect communication**
  - Do the processes directly talk to each other or is there a manager in between?

- **Blocking or non-blocking communication**
  - What happen when the buffer is full (when we want to send) or empty (when we want to read)?

- **Automatic or explicit buffering**
  - Who manages the buffer: programmer (explicitly) or the OS (automatically)?

IPC: Shared Memory

IPC: Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the user processes, not the operating system.

Challenges for Shared Memory IPC

- Processes do not have access to same memory, so OS must provide mechanism to allow them to create a shared memory region.

- When processes share memory to communicate, they must synchronize, otherwise race conditions!

Producer-Consumer Problem

Consider two processes:

- Producer writes
- Consumer reads

Producer-Consumer Problem

```
item nextProduced;

while (1) {
  while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing … buffer is full */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Circular buffer

Producer

Producer-Consumer Problem

```
item nextProduced;

while (1) {
  while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing … buffer is full */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Circular buffer

Producer

```
item nextConsumed;

while (1) {
  while (in == out)
    ; /* do nothing … buffer is empty */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
}
```

Circular buffer

Consumer

Producer-Consumer Problem

```
item nextProduced;

while (1) {
  while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing … buffer is full */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Circular buffer

Any Problems!

Producer

```
item nextConsumed;

while (1) {
  while (in == out)
    ; /* do nothing … buffer is empty */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
}
```

Circular buffer

Consumer

Syscall to create shared memory?

Syscall to create shared memory?

- Use **mmap** → To map memory.
- Mmap can provide shared-memory abstraction in two ways:
  - File-backed mapping.
  - POSIX-Shared Memory mapping.
  - Its like two different backends.

File-Backed Shared Memory

- A regular shared memory experience, but underneath it's a file.
  - Just pointers and loads/stores.
  - Access files as arrays in memory

- Kernel does the I/O instead of explicit read/write calls.
  - File semantics underneath → a real file; dirty pages are written back to it.

- Can also be used to map memory between processes.

File-Backed IPC

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("open"); exit(1); }
    if (ftruncate(fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
    void *addr = mmap(NULL, SIZE,
          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    const char *msg = "Hello via file-backed mmap";
    memcpy(addr, msg, strlen(msg) + 1);
    printf("Producer wrote message into mapped file.");
    sleep(5);  // Keeping session alive for sometime
    munmap(addr, SIZE);
}
```

Producer

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_RDONLY);
    if (fd == -1) { perror("open"); exit(1); }
    void *addr = mmap(NULL, SIZE,
                PROT_READ, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    printf("Consumer read: %s", (char *)addr);
    munmap(addr, SIZE);
    return 0;
}
```

Consumer

File-Backed IPC

Producer

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("open"); exit(1); }
    if (ftruncate(fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
    void *addr = mmap(NULL, SIZE,
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    const char *msg = "Hello via file-backed mmap";
    memcpy(addr, msg, strlen(msg) + 1);
    printf("Producer wrote message into mapped file.");
    sleep(5);  // Keeping session alive for sometime
    munmap(addr, SIZE);
}
```

File-Backed IPC

Producer

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("open"); exit(1); }
    if (ftruncate(fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
    void *addr = mmap(NULL, SIZE,
          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    const char *msg = "Hello via file-backed mmap";
    memcpy(addr, msg, strlen(msg) + 1);
    printf("Producer wrote message into mapped file.");
    sleep(5);  // Keeping session alive for sometime
    munmap(addr, SIZE);
}
```

Open the file

File-Backed IPC

Producer

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("open"); exit(1); }
    if (ftruncate(fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
    void *addr = mmap(NULL, SIZE,
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    const char *msg = "Hello via file-backed mmap";
    memcpy(addr, msg, strlen(msg) + 1);
    printf("Producer wrote message into mapped file.");
    sleep(5);  // Keeping session alive for sometime
    munmap(addr, SIZE);
}
```

Truncate the file size to the length that you want to read/write.

File-Backed IPC

Producer

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("open"); exit(1); }
    if (ftruncate(fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
    void *addr = mmap(NULL, SIZE,
          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    const char *msg = "Hello via file-backed mmap";
    memcpy(addr, msg, strlen(msg) + 1);
    printf("Producer wrote message into mapped file.");
    sleep(5);  // Keeping session alive for sometime
    munmap(addr, SIZE);
}
```

→ **Creates a shared memory representation of the file.**

File-Backed IPC

Producer

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("open"); exit(1); }
    if (ftruncate(fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
    void *addr = mmap(NULL, SIZE,
          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    const char *msg = "Hello via file-backed mmap";
    memcpy(addr, msg, strlen(msg) + 1);
    printf("Producer wrote message into mapped file.");
    sleep(5);  // Keeping session alive for sometime
    munmap(addr, SIZE);
}
```

Copy your desired message to shared memory.

File-Backed IPC

Producer

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("open"); exit(1); }
    if (ftruncate(fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
    void *addr = mmap(NULL, SIZE,
          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    const char *msg = "Hello via file-backed mmap";
    memcpy(addr, msg, strlen(msg) + 1);
    printf("Producer wrote message into mapped file.");
    sleep(5);  // Keeping session alive for sometime
    munmap(addr, SIZE);
}
```

→ **Making producer wait for data to be consumed.**

File-Backed IPC

Producer

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("open"); exit(1); }
    if (ftruncate(fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
    void *addr = mmap(NULL, SIZE,
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    const char *msg = "Hello via file-backed mmap";
    memcpy(addr, msg, strlen(msg) + 1);
    printf("Producer wrote message into mapped file.");
    sleep(5);  // Keeping session alive for sometime
    munmap(addr, SIZE);
}
```

Unmap the memory region, OS can now reclaim the memory.

File-Backed IPC

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_CREAT | O_RDWR, 0666);
    if (fd == -1) { perror("open"); exit(1); }
    if (ftruncate(fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
    void *addr = mmap(NULL, SIZE,
          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    const char *msg = "Hello via file-backed mmap";
    memcpy(addr, msg, strlen(msg) + 1);
    printf("Producer wrote message into mapped file.");
    sleep(5);  // Keeping session alive for sometime
    munmap(addr, SIZE);
}
```

Producer

```
#define FILE_PATH   "/tmp/ipc_file.bin"
#define SIZE        4096
int main(void) {
    int fd = open(FILE_PATH, O_RDONLY);
    if (fd == -1) { perror("open"); exit(1); }
    void *addr = mmap(NULL, SIZE,
                PROT_READ, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
    close(fd);
    printf("Consumer read: %s", (char *)addr);
    munmap(addr, SIZE);
    return 0;
}
```

Reading data

Consumer

What if the file is too large?

What if the file is too large?

- Maps the whole file into your virtual address space.
  - Every byte has an address you could access.

- It does not load the whole file into physical RAM immediately.
  - Pages are brought into RAM on demand when you actually touch them (page faults).
  - May be evicted later under memory pressure.

- You can map a multi-GB file on a machine with much less RAM.

POSIX Shared Memory

POSIX Shared Memory

- Uses a special shared-memory object.
- Object lives in a RAM-backed area like **/dev/shm** → no real disk file to manage.
- Good for fast, temporary sharing between processes in the same system.
- More natural semantics for “shared memory segment”.

POSIX Shared Memory

```
#define SHM_NAME  "/my_shm_example"
#define SIZE      4096
int main(void) {
   int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
   if (shm_fd == -1) { perror("shm_open"); exit(1); }
   if (ftruncate(shm_fd, SIZE) == -1) { perror("ftruncate"); exit(1); }
   void *addr = mmap(NULL, SIZE,
            PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
   if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
   close(shm_fd);
   const char *msg = "Hello via POSIX shared memory \n";
   memcpy(addr, msg, strlen(msg) + 1);
   printf("Producer wrote message into shared memory. \n");
   sleep(5);  // give consumer time
   munmap(addr, SIZE);
}
```

Producer

```
#define SHM_NAME  "/my_shm_example"
#define SIZE      4096
int main(void) {
   int shm_fd = shm_open(SHM_NAME,
                         O_RDONLY, 0666);
   if (shm_fd == -1) { perror("shm_open"); exit(1); }
   void *addr = mmap(NULL, SIZE,
                  PROT_READ, MAP_SHARED, shm_fd, 0);
   if (addr == MAP_FAILED) { perror("mmap"); exit(1); }
   close(shm_fd);
   printf("Consumer read: %s", (char *)addr);
   munmap(addr, SIZE);
   shm_unlink(SHM_NAME);
   return 0;
}
```

Consumer

Properties of IPC: Shared Memory

- Every IPC method is a combination of the following properties:

- **Direct or indirect communication**
  - Do the processes directly talk to each other or is there a manager in between?

- **Blocking or non-blocking communication**
  - What happen when the buffer is full (when we want to send) or empty (when we want to read)?

- **Automatic or explicit buffering**
  - Who manages the buffer: programmer (explicitly) or the OS (automatically)?

Message Passing Mechanisms

Message Passing Mechanisms

- In the message-passing model, processes send data via the kernel, typically with explicit send/receive operations.

- We will look at the following:
  - Pipes
  - Sockets
  - Other ways (we will not cover) Message Queues, Remote Procedure Calls.

Pipes

Pipes

- The OS gives you a **read end** and a **write end**.
- Data (bytes) written at one end come out in FIFO order at the other end.
  - A kernel-managed byte channel between processes.
- Can be either unidirectional or bi-directional.

Type of Pipes

Type of Pipes

- **Un-named Pipes**:
  - Created with **pipe(fd)** in C.
  - Two file descriptors: **fd[0] (read end) and fd[1] (write end)**.
  - Only processes that inherit those FDs (typically via fork) can use the pipe.
  - Exist only as long as there is at least one open FD referencing them; once all ends are closed, the pipe disappears automatically.

Type of Pipes

- **Un-named Pipes**:
  - Created with **pipe(fd)** in C.
  - Two file descriptors: **fd[0] (read end) and fd[1] (write end)**.
  - Only processes that inherit those FDs (typically via fork) can use the pipe.
  - Exist only as long as there is at least one open FD referencing them; once all ends are closed, the pipe disappears automatically.

- **Named pipes (FIFOs):**
  - Created as a special file (FIFO) with **mkfifo("mypipe", 0666) or mknod**.
  - Show up in the filesystem (type **p in ls -l**).
  - Any unrelated process that knows the path and has permissions can open, read, and write it, just like a file.
  - Persist until you rm/unlink them; they outlive the processes that use them.

IPC using Pipes

```
int main(void) {
   int fd[2];  // fd[0] = read end, fd[1] = write end
   if (pipe(fd) == -1) {  perror("pipe");  }
   pid_t pid = fork();
   if (pid < 0) {  perror("fork");  }

   if (pid == 0) {
      close(fd[1]);  // close unused write end
      char buffer[128]; ssize_t n;
      printf("Child: waiting for data...\n");
      while ((n = read(fd[0], buffer, sizeof(buffer) - 1)) > 0) {
         buffer[n] = '\0';  // null-terminate
         printf("Child (consumer) received: %s", buffer);
      }
      close(fd[0]);
   }
```

Consumer

```
else {
      close(fd[0]);  // close unused read end
      const char *messages[] = {
         "item 1", "item 2", "item 3"
      };
      for (int i = 0; i < 3; i++) {
         printf("Parent (producer) sending: %s", messages[i]);
         if (write(fd[1], messages[i], strlen(messages[i])) == -1) {
            perror("write"); break;
         }
         sleep(1);  // just to make the behavior visible
      }
      close(fd[1]);  // signal EOF to consumer
      wait(NULL);    // wait for child
} }
```

Producer

Properties of IPC: Pipes

- Every IPC method is a combination of the following properties:
- **Direct or indirect communication**
  - Do the processes directly talk to each other or is there a manager in between?
- **Blocking or non-blocking communication**
  - What happen when the buffer is full (when we want to send) or empty (when we want to read)?
- **Automatic or explicit buffering**
  - Who manages the buffer: programmer (explicitly) or the OS (automatically)?

Properties of IPC: Pipes

- Every IPC method is a combination of the following properties:

- **Direct or indirect communication**
  - Do the processes directly talk to each other or is there a manager in between?

- **Blocking or non-blocking communication**
  - What happen when the buffer is full (when we want to send) or empty (when we want to read)?

- **Automatic or explicit buffering**
  - Who manages the buffer: programmer (explicitly) or the OS (automatically)?

Sockets

Sockets

- Sockets as **network-capable message passing.**

- Unlike pipes, sockets can connect unrelated processes.
  - Can work either on the same machine or across machines, depending on the socket type.

- A socket is an endpoint for communication.

- Two processes exchange data by sending and receiving through those endpoints.

- A pipe is like a hallway inside one building; a socket is like a phone line with an address.

Sockets

- **socket()**
  - Creates a new communication endpoint and returns a file descriptor for all later operations on that socket (like bind, listen, connect, read, write).

- **bind()**
  - Attaches the socket to a specific local address (a pathname for Unix domain sockets, or an IP+port for TCP), so the OS knows "which address this server lives at."

- **listen()**
  - Marks the bound socket as a passive listening socket and tells the kernel to start queuing incoming connection requests for you, up to a specified backlog.

Sockets

- **accept()**
  - Takes the next pending connection from the listen queue and returns a new socket file descriptor dedicated to talking to that one client.
  - Leaves the original listening socket still in listen mode for future clients.

- **read() (or recv())**
  - Receives bytes from a connected socket into a buffer.
  - It blocks by default until data arrives or the peer closes the connection, and returns how many bytes were actually read.

- **write() (or send())**
  - Sends bytes from your buffer out through the connected socket to the peer.
  - It may block if kernel buffers are full and returns how many bytes were actually written.

Sockets

```
#define SOCKET_PATH "/tmp/demo.sock"

int main(void) {
    int server_fd, client_fd;
    struct sockaddr_un addr;      char buffer[100];
    server_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (server_fd < 0) { perror("socket"); }

    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SOCKET_PATH, sizeof(addr.sun_path) - 1);
    unlink(SOCKET_PATH);  // remove old socket file if it exists
    if (bind(server_fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind");
        close(server_fd);
    }
```

Server

```
    if (listen(server_fd, 5) < 0) {
        perror("listen");  close(server_fd);
    }
    printf("Server: waiting at %s\n", SOCKET_PATH);
    client_fd = accept(server_fd, NULL, NULL);
    if (client_fd < 0) {
        perror("accept");  close(server_fd);
    }
    int n = read(client_fd, buffer, sizeof(buffer) - 1);
    if (n > 0) {
        buffer[n] = '\0';
        printf("Server received: %s\n", buffer);
        write(client_fd, "Hello from server", 17);
    }
    close(client_fd);  close(server_fd);
    unlink(SOCKET_PATH);    }
```

Sockets

```
#define SOCKET_PATH "/tmp/demo.sock"

int main(void) {
    int sock_fd;
    struct sockaddr_un addr;  char buffer[100];

    sock_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (sock_fd < 0) { perror("socket");  }
    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SOCKET_PATH, sizeof(addr.sun_path) - 1);

    if (connect(sock_fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("connect");
        close(sock_fd);
    }
    write(sock_fd, "Hello from client", 17);

    int n = read(sock_fd, buffer, sizeof(buffer) - 1);
    if (n > 0) {
        buffer[n] = '\0';
        printf("Client received: %s\n", buffer);
    }

    close(sock_fd);
    return 0;
}
```

Client

How can sockets be used to communicate between two machines?

Sockets over Internet

Server

```
int server_fd = socket(AF_INET, SOCK_STREAM, 0);  // AF_INET instead of AF_UNIX
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(12345);              // choose a port, e.g. 12345
addr.sin_addr.s_addr = htonl(INADDR_ANY);   // any local IP
```

Client

```
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(12345);              // same port
inet_pton(AF_INET, "192.168.1.10", &addr.sin_addr);  // server machine's IP
```