Theory of Operating Systems

Week 4: Inter-Process Communication

Faye (Chi Zhang)

CUNY Hunter

February 18, 2026

Today's Outline

Processes Are Isolated

By design, processes cannot access each other's memory

- Each process has its own address space
- Memory protection enforced by hardware
- One process crashing doesn't affect others
- Security: Processes can't read each other's data

But sometimes processes need to communicate...

- Share data
- Coordinate actions
- Send notifications

Why Do Processes Need to Cooperate?

Information Sharing:

- Multiple apps access same file
- Database and web server
- Clipboard between apps

Computation Speedup:

- Parallel processing
- Divide large task
- Multiple cores

Modularity:

- Microservices architecture
- Separate concerns
- Independent development

Convenience:

- Shell pipelines
- Client-server model
- Plugin architectures

IPC Mechanisms Overview

Mechanism	Type	Scope
Pipes	Message Passing	Parent-Child
Named Pipes (FIFOs)	Message Passing	Same machine
Message Queues	Message Passing	Same machine
Shared Memory	Shared Memory	Same machine
Signals	Notification	Same machine
Sockets	Message Passing	Network

Today's Focus: Pipes, Message Queues, Shared Memory, Signals

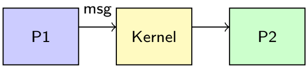
Two Fundamental IPC Models

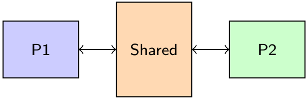
Message Passing:

- Processes exchange messages
- OS handles data transfer
- send() and receive() operations
- Easier to program
- Higher overhead (syscalls)

Shared Memory:

- Processes share memory region
- Direct read/write access
- Fastest IPC method
- Needs synchronization
- More complex





Message Passing vs Shared Memory

Aspect	Message Passing	Shared Memory
Speed	Slower (kernel involved)	Faster (direct access)
Synchronization	Built-in	Programmer's job
Data size	Better for small messages	Better for large data
Complexity	Simpler API	More complex
Network support	Yes (sockets)	No (same machine)

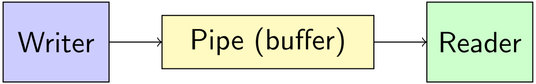
When to use which?

- Message Passing: Coordination, small data, different machines
- Shared Memory: Large data transfer, high performance needs

Pipes: The Simplest IPC

Definition: A unidirectional byte stream between two processes

- Data flows in one direction only
- FIFO order: First In, First Out
- Limited capacity (buffer in kernel)
- `write()` blocks if full, `read()` blocks if empty



Unix Pipeline Concept

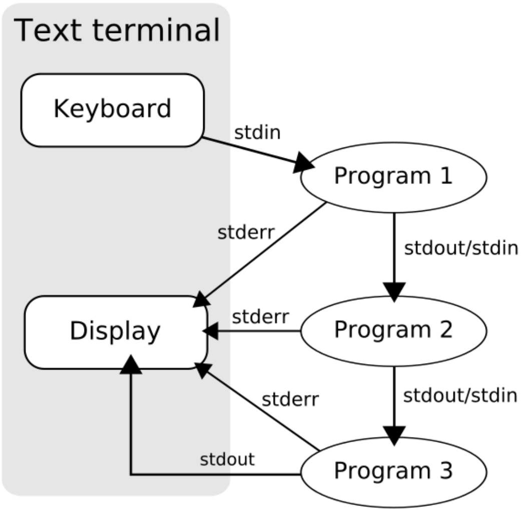
Pipeline Philosophy:

- Connect simple programs
- Each does one thing well
- Data flows through pipes
- Powerful combinations

Example:

- `cat file | grep pattern | wc -l`
- Three processes connected by pipes

Source: Wikimedia Commons



Creating Pipes: pipe()

```c
#include <unistd.h>

int main() {
    int fd[2];  // fd[0] = read end, fd[1] = write end

    if (pipe(fd) == -1) {
        perror("pipe failed");
        exit(1);
    }

    // Now fd[0] and fd[1] are connected
    // Write to fd[1], read from fd[0]

    return 0;
}
```

Note: Pipe exists only in kernel memory, no file on disk

Pipes Between Parent and Child

```
int fd[2];
pipe(fd);

pid_t pid = fork();
if (pid == 0) {
    // Child: read from pipe
    close(fd[1]);  // Close unused write end
    char buf[100];
    read(fd[0], buf, sizeof(buf));
    printf("Child received: %s\n", buf);
    close(fd[0]);
} else {
    // Parent: write to pipe
    close(fd[0]);  // Close unused read end
    write(fd[1], "Hello from parent!", 18);
    close(fd[1]);
    wait(NULL);
}
```

Shell Pipes: The — Operator

```
$ cat file.txt | grep "error" | wc -l
```

What happens:

1. Shell creates two pipes
2. Forks three child processes
3. Redirects stdout $\rightarrow$ stdin via pipes
4. Each process runs independently

Named Pipes (FIFOs)

Problem with anonymous pipes: Only work between related processes

Solution: Named pipes (FIFOs)

- Have a name in the filesystem
- Any process can open and use
- Persist until explicitly deleted

```
# Create a named pipe
$ mkfifo /tmp/myfifo

# In terminal 1 (blocks until reader connects)
$ echo "Hello" > /tmp/myfifo

# In terminal 2
$ cat /tmp/myfifo
Hello
```

Named Pipes in C

```c
#include <sys/stat.h>
#include <fcntl.h>

// Create FIFO
mkfifo("/tmp/myfifo", 0666);

// Writer process
int fd = open("/tmp/myfifo", O_WRONLY);
write(fd, "Hello", 6);
close(fd);

// Reader process (different program)
int fd = open("/tmp/myfifo", O_RDONLY);
char buf[100];
read(fd, buf, sizeof(buf));
close(fd);
```

10-Minute Break We'll continue with Message Queues

Message Queues

Definition: A queue of messages stored in the kernel

- Messages have type and data
- Multiple readers/writers supported
- Messages can be retrieved by type
- Persist until explicitly removed

Advantages over pipes:

- Message boundaries preserved
- Can select messages by type
- Multiple senders/receivers
- No need for reader to be running when sending

POSIX Message Queues

```c
#include <mqueue.h>

// Sender
mqd_t mq = mq_open("/myqueue", O_CREAT | O_WRONLY, 0644, NULL);
mq_send(mq, "Hello", 6, 0);  // priority 0
mq_close(mq);

// Receiver
mqd_t mq = mq_open("/myqueue", O_RDONLY);
char buf[256];
unsigned int priority;
mq_receive(mq, buf, 256, &priority);
printf("Received: %s\n", buf);
mq_close(mq);
mq_unlink("/myqueue");  // Remove queue
```

Compile with: `gcc -lrt program.c`

Message Queue Properties

Key Attributes:

- **mq_maxmsg**: Maximum number of messages
- **mq_msgsize**: Maximum size of each message
- **mq_curmsgs**: Current number of messages in queue

Blocking Behavior:

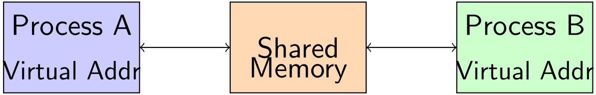
- `mq_send()` blocks if queue is full
- `mq_receive()` blocks if queue is empty
- Non-blocking mode available with `O_NONBLOCK`

Priority: Messages with higher priority delivered first

Shared Memory: The Fastest IPC

Definition: Memory region accessible by multiple processes

- Same physical memory, different virtual addresses
- No kernel involvement for read/write (after setup)
- Fastest IPC mechanism
- Requires synchronization!



POSIX Shared Memory: Creating

```c
#include <sys/mman.h>
#include <fcntl.h>

// Create shared memory object
int fd = shm_open("/myshm", O_CREAT | O_RDWR, 0666);

// Set size
ftruncate(fd, 4096);

// Map to address space
char *ptr = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
                 MAP_SHARED, fd, 0);

// Now ptr points to shared memory
sprintf(ptr, "Hello from process A!");

// When done
munmap(ptr, 4096);
close(fd);
```

POSIX Shared Memory: Reading

```
// Another process can access the same memory

int fd = shm_open("/myshm", O_RDONLY, 0666);

char *ptr = mmap(NULL, 4096, PROT_READ,
                 MAP_SHARED, fd, 0);

printf("Read from shared memory: %s\n", ptr);

munmap(ptr, 4096);
close(fd);
```

Warning

No synchronization here! If both processes write simultaneously, data corruption occurs. Need semaphores or mutexes.

The Synchronization Problem

Without synchronization:

- Race conditions: Result depends on timing
- Data corruption: Partial writes
- Lost updates: One overwrites another

Solutions (covered in Week 6):

- Semaphores
- Mutexes
- Condition variables

Preview

We'll cover synchronization in detail next week. For now, understand that shared memory needs coordination.

Signals: Asynchronous Notifications

Definition: Software interrupts sent to a process

- Notify process of an event
- Can interrupt normal execution
- Process can handle, ignore, or use default action
- Limited data: just the signal number

Common signals:

Signal	Number	Default Action
SIGINT	2	Terminate (Ctrl+C)
SIGKILL	9	Terminate (cannot be caught)
SIGSEGV	11	Core dump (segfault)
SIGTERM	15	Terminate (polite request)
SIGCHLD	17	Ignore (child terminated)

Sending and Handling Signals

```c
#include <signal.h>

void handler(int sig) {
    printf("Caught signal %d\n", sig);
}

int main() {
    // Install signal handler
    signal(SIGINT, handler);

    // Or use sigaction (recommended)
    struct sigaction sa;
    sa.sa_handler = handler;
    sigaction(SIGINT, &sa, NULL);

    // Send signal to another process
    kill(pid, SIGTERM);

    while(1) pause();  // Wait for signals
```

Signal Use Cases

Common uses:

- **Process control**: Kill, stop, continue
- **Notification**: Child exited, alarm timer
- **Error handling**: Segfault, bus error
- **User-defined**: SIGUSR1, SIGUSR2

Limitations:

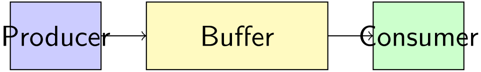
- No data payload (just signal number)
- Signals can be lost (not queued by default)
- Handler must be careful (async-signal-safe)
- Not suitable for complex IPC

Classic IPC problem:

- Producer: Creates data items
- Consumer: Uses data items
- Buffer: Shared storage between them

Challenges:

- Producer must wait if buffer is full
- Consumer must wait if buffer is empty
- Need to coordinate access to buffer



Producer-Consumer with Pipes

```c
int fd[2];
pipe(fd);

if (fork() == 0) {
    // Producer
    close(fd[0]);
    for (int i = 0; i < 10; i++) {
        int item = produce();
        write(fd[1], &item, sizeof(item));
    }
    close(fd[1]);
} else {
    // Consumer
    close(fd[1]);
    int item;
    while (read(fd[0], &item, sizeof(item)) > 0) {
        consume(item);
    }
    close(fd[0]);
```

Bounded Buffer Problem

More complex version:

- Fixed-size buffer (e.g., 10 slots)
- Multiple producers and consumers
- Need explicit synchronization

Requires (covered in Week 6):

- Mutex: Protect buffer access
- Semaphore (empty): Count empty slots
- Semaphore (full): Count full slots

Assignment 2

You'll implement this in your Producer-Consumer assignment!

IPC Mechanism Comparison

Mechanism	Speed	Complexity	Data Size	Scope
Pipes	Medium	Low	Stream	Parent-child
Named Pipes	Medium	Low	Stream	Same machine
Message Queues	Medium	Medium	Messages	Same machine
Shared Memory	Fast	High	Large	Same machine
Signals	Fast	Medium	None	Same machine
Sockets	Slow	Medium	Stream/Datagram	Network

Choosing the Right IPC Mechanism

Use Pipes when:

- Simple parent-child communication
- Unidirectional data flow
- Shell-style pipelines

Use Message Queues when:

- Multiple unrelated processes
- Need message boundaries/priorities
- Asynchronous communication

Use Shared Memory when:

- Large data, high performance
- Willing to handle synchronization
- Frequent data exchange

Key Takeaways

1. **IPC needed** because processes are isolated by design
2. **Two models**: Message passing (kernel involved) vs Shared memory (direct)
3. **Pipes**: Simple, unidirectional, parent-child
4. **Named pipes**: Like pipes, but unrelated processes
5. **Message queues**: Messages with types/priorities
6. **Shared memory**: Fastest, needs synchronization
7. **Signals**: Notifications, no data payload

This Week's Tasks

- **Quiz 4**: IPC mechanisms
- **Reading**: Textbook Chapters 3.4-3.7
- **Assignment 1 Due**: Process Scheduling Simulator
- **Assignment 2 Released**: Producer-Consumer Problem
  - Use pipes or shared memory
  - Due: March 4
- **Next Week**: Threads and Concurrency
  - Thread models, pthreads API
  - Race conditions preview

Questions?