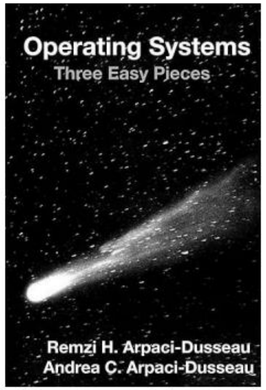
Lecture 5 – Inter-Process Communication (IPC) and Threads

20/10/2025

Chapters:
26: Concurrency and Threads
30: Condition Variables
33: Event-based Concurrency



Outline

- Why IPC? What is IPC?
- Two models of IPC:
  - Shared-memory
  - Message passing
- Message Passing IPC
- Shared memory IPC
- Synchronisation concepts
- IPC to Networking

Why IPC? What is IPC?

- Recall that an OS isolates processes for protection and stability
- But real applications often need to **cooperate**:
  - E.g., GUI + background worker
  - Advantages: to speed up computation, improve responsiveness
- **Problem**: processes can’t directly access each other’s address space or registers.
- IPC refers to OS-provided mechanisms that allow processes to exchange data and synchronize their actions.

IPC Models Overview

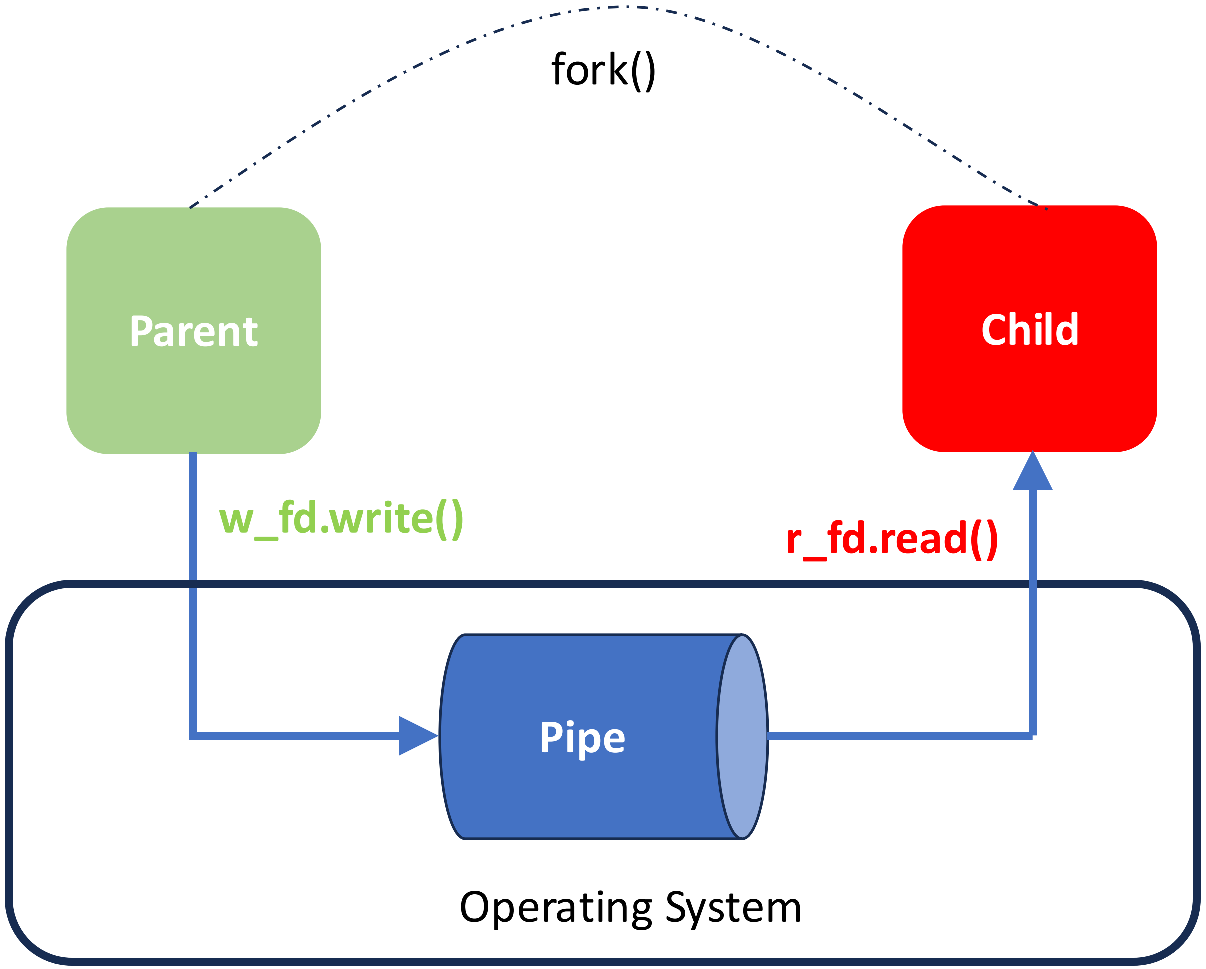
- **Message passing:** exchange of data via OS-managed channels such as pipes, queues, sockets. [We will cover Sockets in Week 4, Lecture 1]
  - This is simple but possibly a little bit slower
- **Shared memory:** processes map a common memory region into their address spaces
  - Once setup, shared memory has **no per-message syscall/copy** overhead.

Model	Example	Advantages	Disadvantages
Message Passing	Pipe, Socket	Simpler, safe	Kernel overhead
Shared Memory	mmap	Fast, flexible	Race conditions

IPC with Message Passing using Pipes

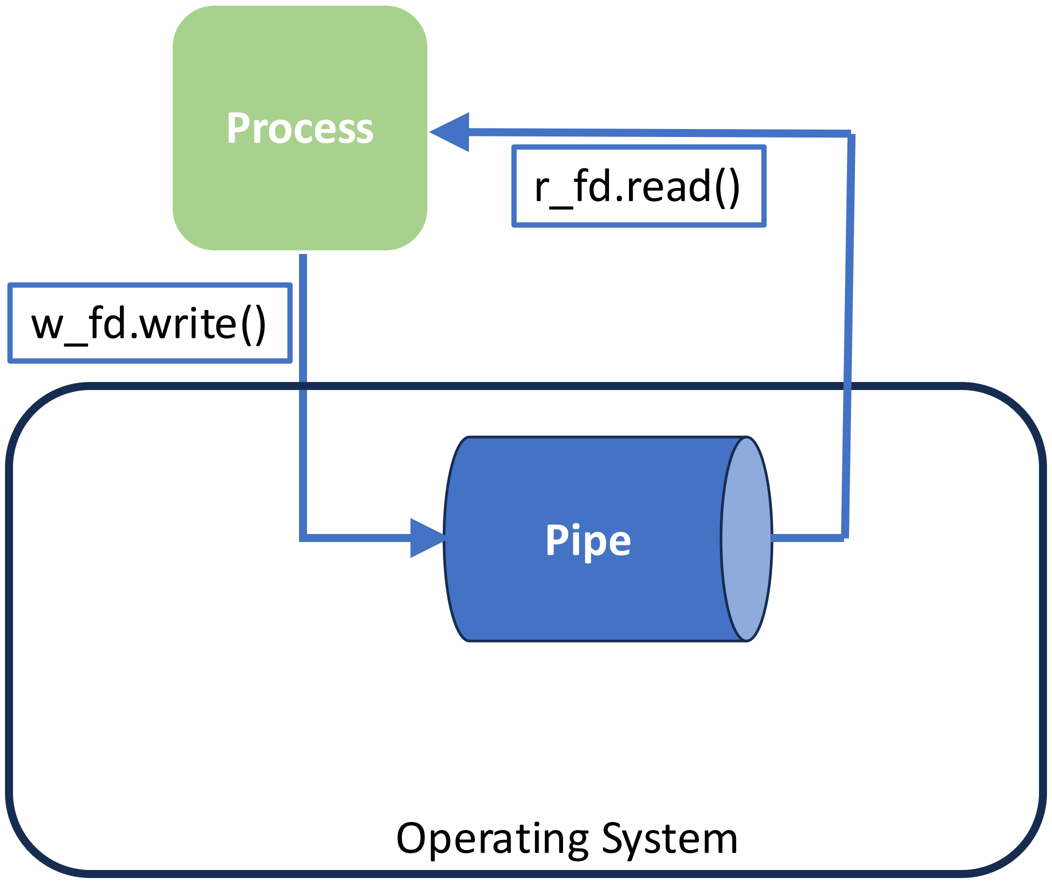
- Pipe is a unidirectional communication channel:
    - Implemented as a **FIFO buffer in the kernel**
    - Two ***file descriptors*** *(FDs): one* to write data to the pipe and one to read data from the pipe.

- Typically, a parent process creates a pipe and calls fork() to create a child process
- After **fork()**, **both processes inherit both FDs**; they refer to **the same open pipe** (same kernel object).



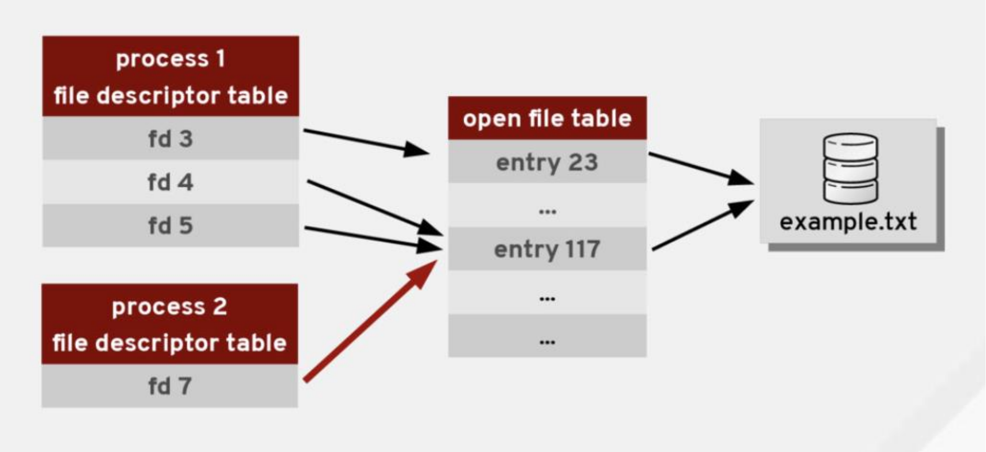
IPC with Message Passing using Pipes

- pipe() system call:
  - r_fd,w_fd = pipe()
- The state immediately after pipe() (before fork()) is shown in the picture
- How do we go from this (on the right) to using a pipe between two processes?
  - Hint: fork()



What is a file descriptor?

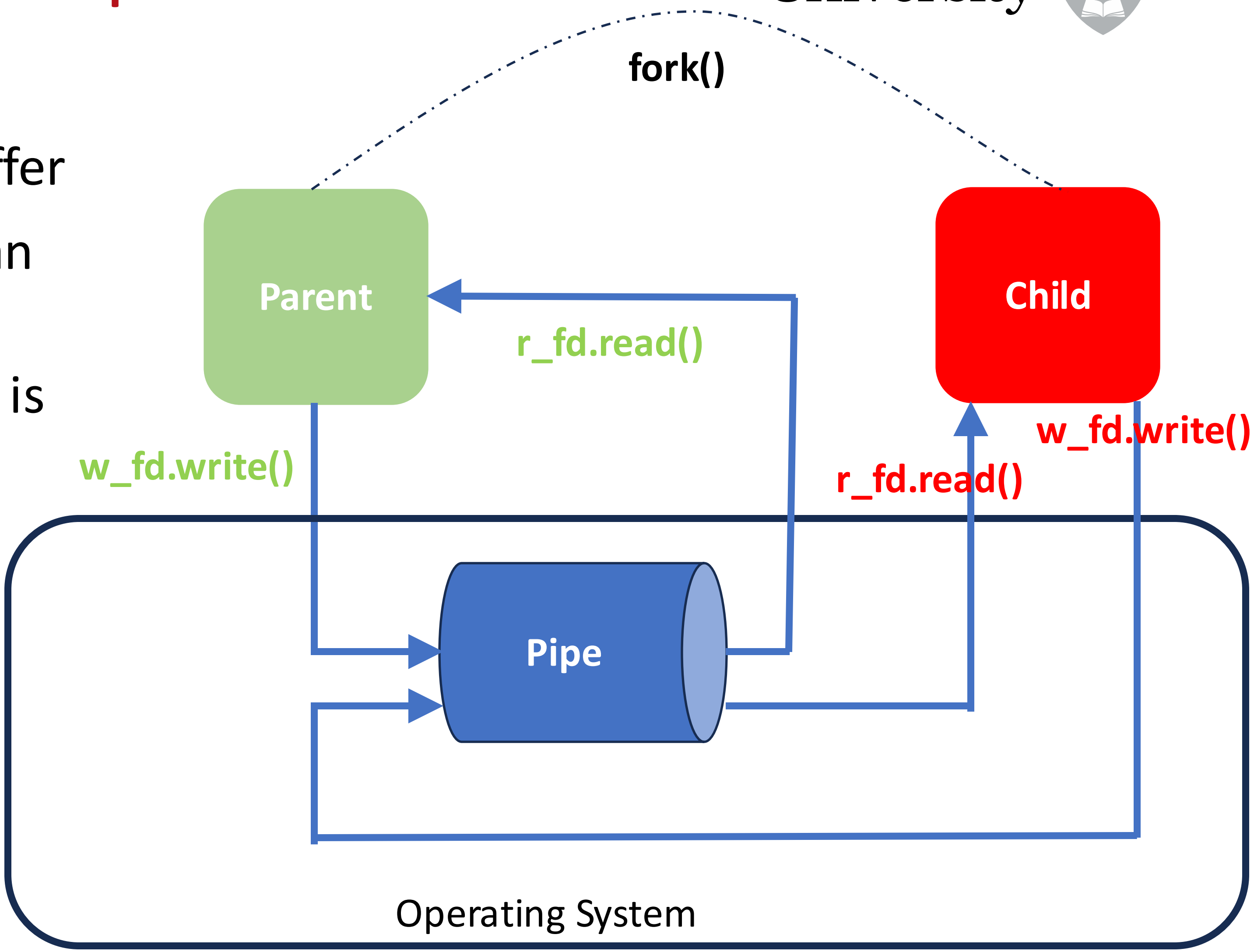
- A **file descriptor (FD)** is a **small integer** that indexes your **process's FD table**.
- Each FD table entry points to a **kernel "open file table"** (shared state):
- Current **file offset** (read/write position)
- Pointer to the underlying **object type**: regular file, **pipe**, **socket**, device, etc.
- **Standard FDs:** 0=stdin, 1=stdout, 2=stderr.



Pipes: Example

- Two processes → same kernel buffer
- Data copied via kernel (slower than shared memory but safe)
- The state immediately after fork() is shown in the picture

```
r_fd, w_fd = os.pipe()
pid = os.fork()
```

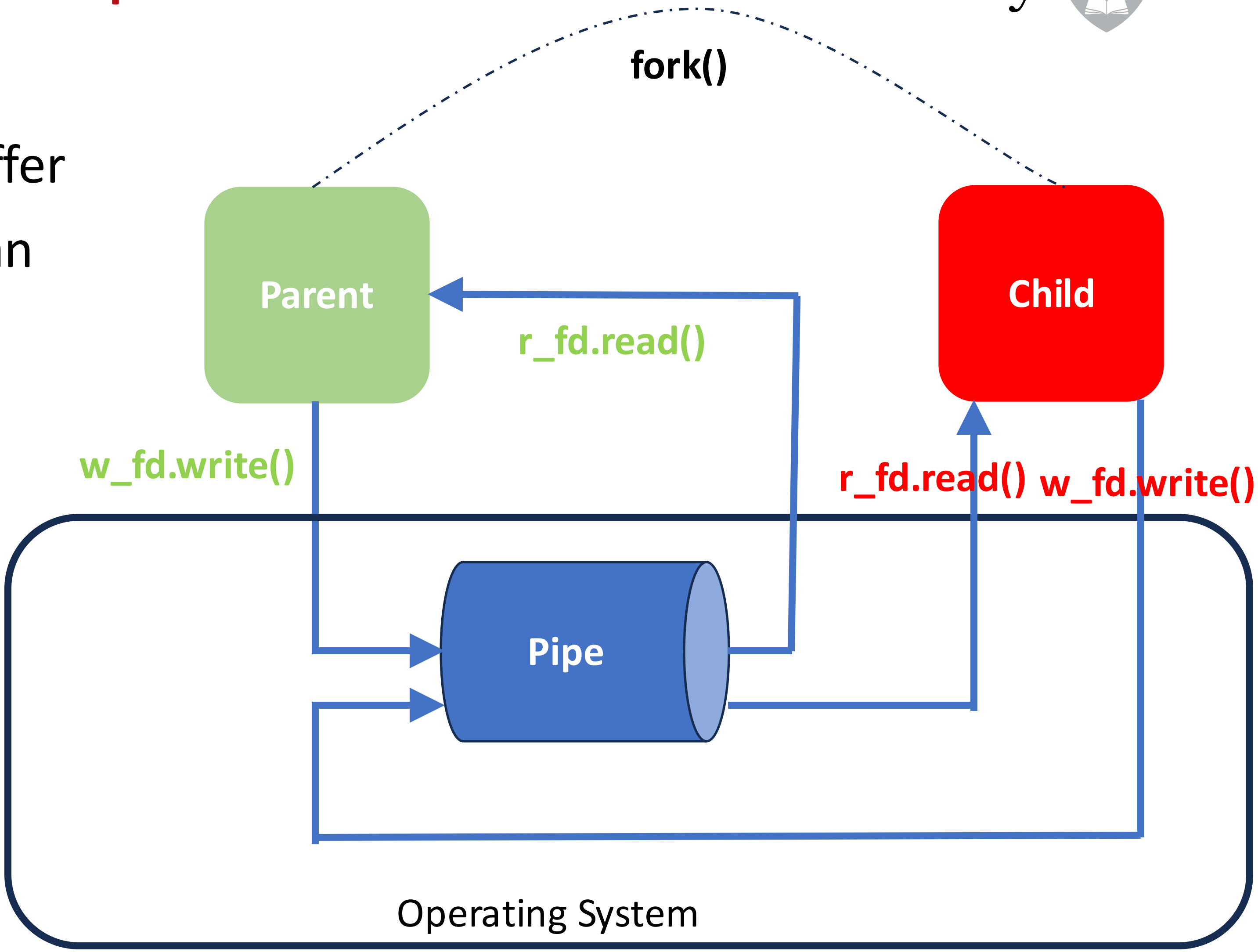


Pipes: Example

- Two processes → same kernel buffer
- Data copied via kernel (slower than shared memory but safe)

```
r_fd, w_fd = os.pipe()
pid = os.fork()

if pid == 0:  # Child
    os.close(w_fd)
    msg = os.read(r_fd, 100)
    print("Child received:", msg.decode())
    os.exit(0)
else:         # Parent
    os.close(r_fd)
    os.write(w_fd, b"Hello from parent!")
```

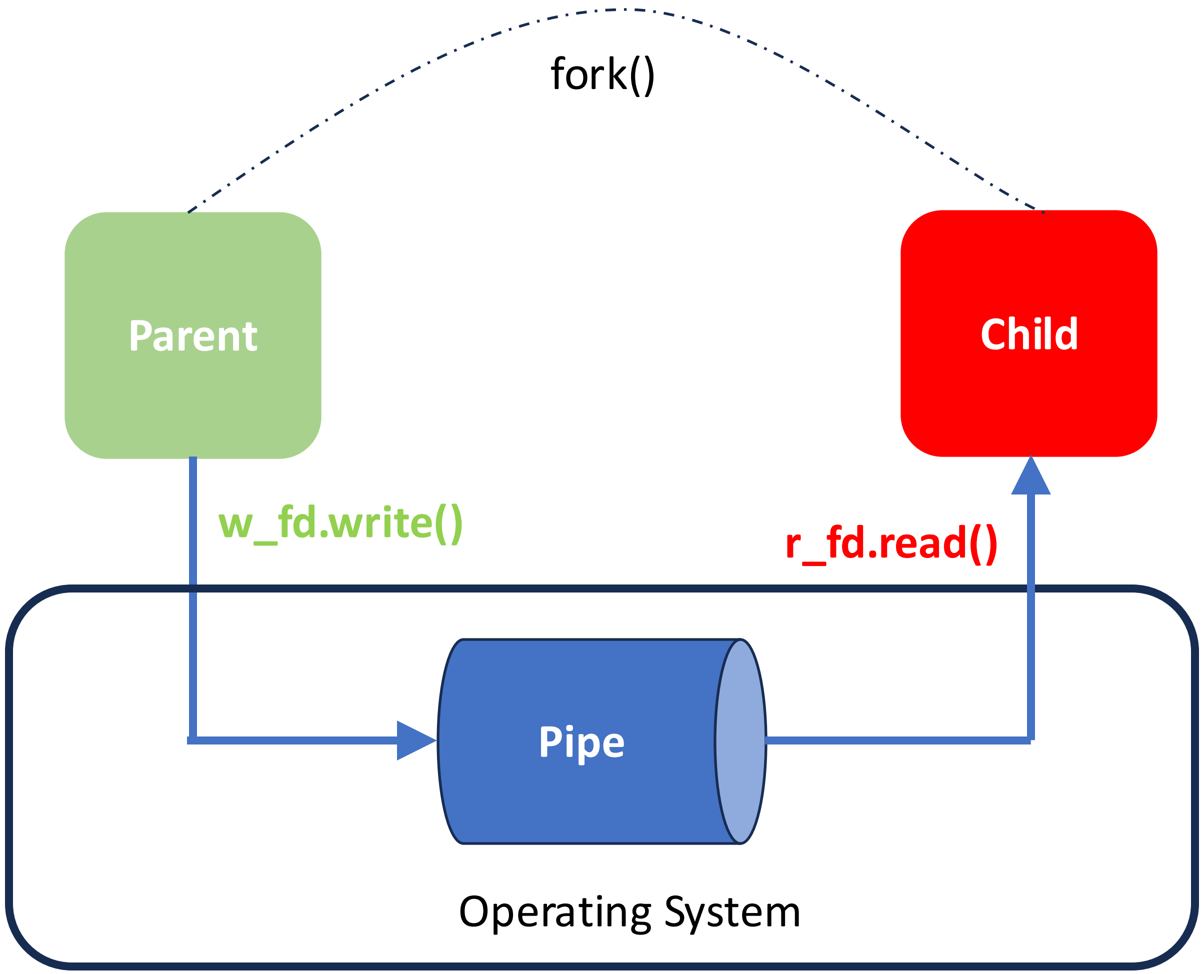


Pipes: Example

- Two processes → same kernel buffer
- Data copied via kernel (slower than shared memory but safe)

```
r_fd, w_fd = os.pipe()
pid = os.fork()

if pid == 0:  # Child
    os.close(w_fd)
    msg = os.read(r_fd, 100)
    print("Child received:", msg.decode())
    os.exit(0)
else:         # Parent
    os.close(r_fd)
    os.write(w_fd, b"Hello from parent!")
```



Pipes: Blocking, Backpressure, End-of-Stream

- **Reading** from an **empty pipe blocks** the reader *until data becomes available* (unless the read end is set to non-blocking).
- **Writing** to a **full pipe blocks** the writer *until space becomes available*.
- The **pipe buffer** has a **finite size** (e.g. typically 64 KiB on Linux).
  - Applications cannot set it directly in pipe() — use fcntl(fd, F_SETPIPE_SZ, size) if you need to adjust it.
- Why two file descriptors for each pipe?
  - This allows the kernel to detect end-of-stream conditions:
    - If **all readers** are closed → further **writes** raise SIGPIPE / return EPIPE.
    - If **all writers** are closed → **reads** return **0 to signal end of stream**.

Pipes: Handling exceptions

```
r_fd, w_fd = os.pipe()
pid = os.fork()
```

```
if pid == 0:  # child (reader)
    os.close(w_fd)
    data = os.read(r_fd, 100)

    if data:
        print("Child received:", data.decode())
    else:
        print("Child: EOF")
    os.close(r_fd)
    os._exit(0)
```

```
else:        # parent (writer)
    os.close(r_fd)
    try:
        os.write(w_fd, b"Hello from parent!")
    except BrokenPipeError:
        print("Parent write error: no reader (EPIPE)",
file=sys.stderr)
    finally:
        os.close(w_fd)              # signal EOF
        os.waitpid(pid, 0)           # reap child
```

Pipes: In action (1)

- When you do in the shell:
  - ps aux | grep python
- The shell does fork twice and eventually exec() for each created process
- Before calling exec, the shell uses dup2() to connect the **stdout** of ps to the **stdin** of grep.
  - **dup2(oldfd, newfd)**: duplicates an existing FD **onto a specific FD number** (overwriting newfd if open).

```
~$ ps aux | grep python3
uceeoas          40413   0.0  0.0 34129548    648 s011  R+    2:48pm   0:00.00 grep python3
~$ 
```

Pipes: In action (2)

```
r, w = os.pipe()
pid = os.fork()
if pid == 0:                # child: writer
    os.dup2(w, 1)           # stdout -> pipe write end
    os.close(r); os.close(w)
    os.execvp("ps", ["ps", "aux"])
else:                       # parent: reader child
    pid2 = os.fork()
    if pid2 == 0:
        os.dup2(r, 0)       # stdin -> pipe read end
        os.close(r); os.close(w)
        os.execvp("grep", ["grep", "python3"])
    os.close(r); os.close(w)

 # parent: close both ends and call wait below
…
```

IPC with Message Passing: Sockets

- Also works for local processes running on the same host
- Similar concept to pipes, but mostly used across machines
- socket.send() ↔ socket.recv()
- See: [**Week 4 Lecture 1**]

IPC with shared memory

- Each process has its own virtual address space, but the OS can **map the same physical frame** into both.
- Mechanisms:
  - **We will look at POSIX shared memory:** mmap()
  - The region can be **anonymous** (OS-allocated) or **file-backed** (maps a file into memory).After mapping, both processes can **read/write directly** to that region.
- **No kernel mediation per access** — only the initial setup goes through the kernel.

IPC with shared memory

- The parent creates the shared region.
- close() removes it when done.
- Both processes have access to the same memory to communicate
- No kernel involvement (unlike pipe) once the memory is created

```
import os, mmap

m = mmap.mmap(-1, 32)      # shared memory (32 bytes)

if os.fork() == 0:         # child
    m[:11] = b'hello world'
    m.close()
    os._exit(0)
else:                      # parent
    os.wait()              # ensures child finished writing
    print(m[:11])          # b'hello world'
    m.close()
```

IPC with shared memory

- Shared memory requires synchronisation (mutual exclusion) to avoid races.
- Without the lock, the parent's read could overlap the child's write → stale/partial ("torn") read.
- The lock provides mutual exclusion
- The kernel **doesn't synchronise** shared-memory accesses—only the mapping setup—so you must do that.

```
import os, mmap

m = mmap.mmap(-1, 32)      # shared memory (32 bytes)

if os.fork() == 0:         # child
    m[:11] = b'hello world'
    m.close()
    os._exit(0)
else:                      # parent
    os.wait()              # ensures child finished writing
    print(m[:11])          # b'hello world'
    m.close()
```

IPC with shared memory

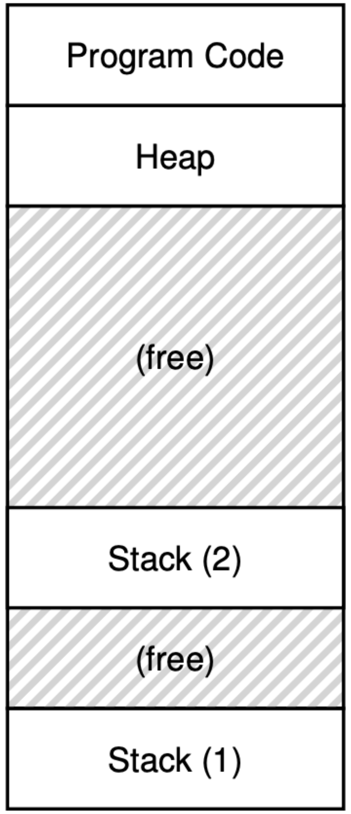
- What is a shared memory?
  - Map a region of a memory into a process's **virtual address space** so you can read/write via **regular memory read and write operations (without system calls!)**.
- Where does the memory come from?
  - **POSIX (Python):** mmap.mmap(-1, size, …)
  - Use for **shared memory across fork()**

Thread abstraction: Shared Memory within a Process

- A **thread** is a *separate point of execution* within a process.
- Each thread has:
  - its own **program counter**
  - its own **registers and stack**
- All threads share the **same address space**.
- Threads = *lightweight processes* sharing:
  - address space
  - open files
  - global variables
  - **Lightweight**: faster to create them (less copying of resources)
  - Used for parallelism within a single process

Threads: Shared Memory within a Process

- Thread is very much like a separate process, except for one difference
  - they *share* the same address space and thus can access the same data
    - Heap (dynamically allocated data)
- Threads have separate stacks (why?)



Threads: in Python

- Possible output
  - A
  - B
  - Main done
- Possible output
  - B
  - A
  - Main done
- Thread scheduler decides which thread runs next
- **Uncontrolled interleaving →** nondeterministic order

```
import threading

def worker(name):
    print(f"Worker {name} starting")

t1 = threading.Thread(target=worker, args=("A",))
t2 = threading.Thread(target=worker, args=("B",))
t1.start()
t2.start()
t1.join()
t2.join()
print("Main done")
```

Threads: Shared data (race condition)

```
from threading import Thread

counter = 0

def increment():
    global counter
    for _ in range(100000):
        counter += 1            # Critical section (variable counter shared by two threads)

t1 = Thread(target=increment)
t2 = Thread(target=increment)
t1.start(); t2.start()
t1.join(); t2.join()
print(counter)
```

Threads: Synchronisation

- *Critical section (CS)*: part of code accessing shared state
- *Atomic operation*: appears indivisible
- Need mechanisms to ensure *mutual exclusion*
- **Locks**: only one thread in CS
- **Condition variables**: signal state change

Threads: Synchronisation

```
import threading
counter = 0
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

t1 = Thread(target=increment)
t2 = Thread(target=increment)
t1.start(); t2.start()
t1.join(); t2.join()
print(counter)
```

Threads: Python's GIL

- The **Global Interpreter Lock (GIL)** prevents true parallel bytecode execution
  - Only one thread executes Python code at a time
- Parallelisation with threads **does not** lead to speedups for **CPU-bound** code in Python. (Lab exercise – hacker's edition)
- But the **concept** of race conditions still matters
  - When you have large blocks of critical regions
  - Some libraries can turn off GIL

Threads: Beyond Mutual Exclusion

- Sometimes threads must **wait for a condition**, not just a lock
  - e.g. Consumer waits until Producer adds an item
- Locks protect data
- **Condition variables** coordinate *when* threads proceed
  - cond.wait() releases the lock and suspends
  - cond.notify() wakes one waiting thread
- **Sentinel value:** a special marker (e.g., None) placed in the shared queue to signal consumers to stop when production is finished.

```
import threading, time
queue, lock = [], threading.Lock()
cond = threading.Condition(lock)

def producer():
    for i in range(5):
        with cond:
            queue.append(i)
            cond.notify()     # wake one waiting consumer

def consumer():
    while True:
        with cond:
            while not queue: # while queue is empty
                cond.wait()    # releases lock, waits for signal
            item = queue.pop(0)
        print("Consumed", item)
# Create the producer and consumer threads (omitted)
```

Threads: How conditional variables work

- Combine **lock** + **queue of waiting threads**
- wait():
  - releases lock, sleeps
  - reacquires lock before returning
- notify() or notify_all() wakes waiting threads
  - Useful for signalling *state changes*

From IPC to Networking

- When Processes Live on Different Machines…

- Message-passing generalises to sockets (Week 4)
- Socket = endpoint for inter-machine IPC
- Same idea, different address space and transport medium

- Local IPC → pipe / thread queue
- Networked IPC → socket / TCP

Conclusions

- **Processes vs Threads**
  - Processes: isolated address spaces → communication via **IPC** (pipes, sockets, shared memory).
  - Threads: share the same address space → communicate via **shared variables**.
- **Pipes**
  - Unidirectional, kernel-mediated channel.
  - Simpler but slower due to **kernel involvement** on each read/write.
- **Shared Memory**
  - Fastest IPC — no kernel mediation after setup.
  - Requires **explicit synchronisation** (locks).
- **Threads**
  - Lightweight execution units within a process.
  - Enable parallelism and shared-state concurrency — but introduce **race conditions**.
- **Synchronisation Tools**
  - **Locks** for mutual exclusion.
  - **Condition variables** for coordination between threads.