SCC.231 (Week 3) - IPC and Concurrency

Week 3 — IPC & Concurrency

Learning Objectives

In this lab we will expand our previous activities on running code on multiple cores and explore (i) how to allow processes to communicate with each other using **Inter-Process Communication (IPC)** mechanisms, and (ii) how to write concurrent code using **threads**. In your lab tasks, you will need to download the provided template code. The templates include both code that needs fixing, as well as, completed code which you can use to experiment with the benefits of parallelism. By the end of this lab you will be able to:

- ☐ Understand and implement message passing between processes using **pipes**.
- ☐ Use the `select()` system call to monitor multiple file descriptors for I/O readiness.
- ☐ Understand the concept of threads and how they differ from processes.
- ☐ Implement a producer–consumer pattern using threads and a shared queue.

> [!NOTE] At the end of each task, we include a set of **reflection questions**. These are designed to help you consolidate your understanding of the concepts covered in the task. You do not need to produce written answers for these questions, but we encourage you to think about them and discuss them with your peers or instructors.

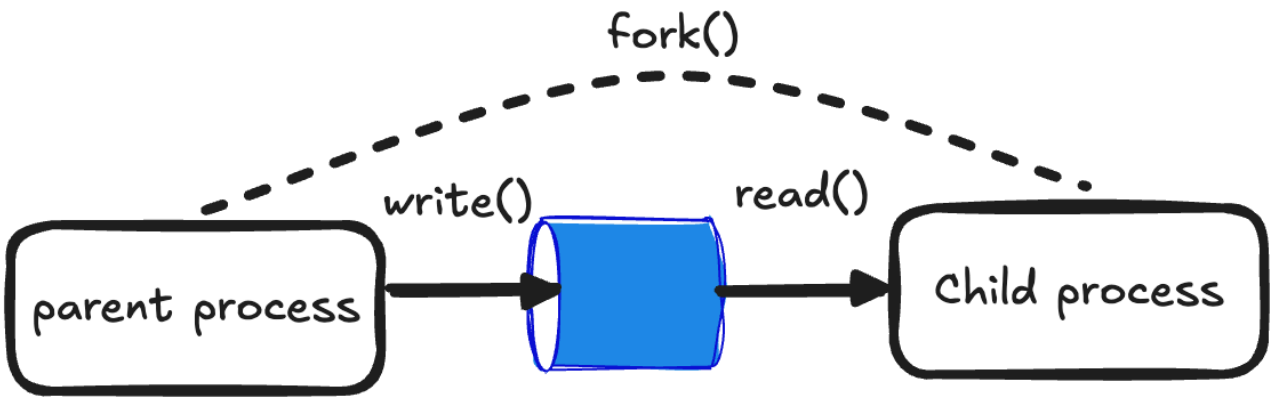
Part 1 — Message Passing Using Pipes

Task 1A — A Simple Pipe Example

What is a pipe?

A *pipe* is a unidirectional communication channel provided by the operating system (OS). It consists of a **write end** and a **read end**, connected by a bounded FIFO buffer in kernel memory. The function of a pipe is simple and follows the following principles:

- When a process calls `write()`, bytes are appended to the pipe buffer.
- When another process calls `read()`, bytes are removed in the same order.
- If the buffer becomes **full**, a writer **blocks** (i.e., the function will not return) until the reader consumes data.
- If all writers close their end, readers see **End-Of-File (EOF)** (i.e., `read()` returns an empty byte string `b''`).
- If a writer attempts to write after the reader has closed its end, the kernel raises an `EPIPE` signal — in Python this appears as a `BrokenPipeError`.



Example use of a pipe command.

File descriptors (FDs).

In UNIX-like systems, each process maintains a table of integer file descriptors (FDs) for any open I/O channel, including open files, sockets, and pipes.
The system call `os.pipe()` creates the pipe structure in the kernel and returns a tuple of two values `(rfd, wfd)` — the read and write descriptors. A pipe is typically used between a parent and child process created via `os.fork()`, where the parent writes data and the child reads it (or vice versa). Selecting the direction of communication is a design choice and it is

enforced once the pipe is created programmatically. It is good practice once your select which process will read and which will write to immediately close the unused ends of the pipe in each process. If bidirectional communication is needed, two pipes must be created.

Scenario

As part of this lab activity you should download the template code provided with your assignment. For this activity, you will complete the implementation of a simple producer–consumer system using a pipe between a parent and child process found in the file `task1A.py`.

Pipes are often used for streaming data between processes, in order to realise a software pattern known as **producer–consumer** (https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem). The benefit of this pattern is that the producer and consumer can operate at different speeds, with the pipe buffer acting as a cushion to absorb bursts of data from the producer or temporary slowness from the consumer. It is esxcellent for realising architectures that use pipelines of processing stages.

The parent process (the **producer**) writes data to a pipe in small chunks.
The child process (the **consumer**) initially reads only once, then closes its read end prematurely.
This causes the parent's next write to fail with a **BrokenPipeError**, since there are no readers left.

The implementation of the parent process is complete and the code can handle both blocking and `BrokenPipeError` situations gracefully. You do not need to modify it.
It measures how long each write takes and prints a clear red message if the reader closes early.

Your goal is to **fix the child process** so that it reads repeatedly until a read operation returns an EOF — i.e., until `os.read()` returns an empty byte string.
This ensures the parent can write all its data without encountering a broken pipe.

Task1A Reflection Questions

1. Are read() and write() on a pipe blocking calls?
2. When does os.write() block, and when does it fail with BrokenPipeError?

3. How does the reader know that the writer has finished sending data?
4. Closing unused ends: Why must the parent close rfd and the child close wfd immediately after fork()?
5. Reader closed early: What happens if the writer tries to write after the read end is closed?

Task 1B — Event-Based Concurrency with `select()`

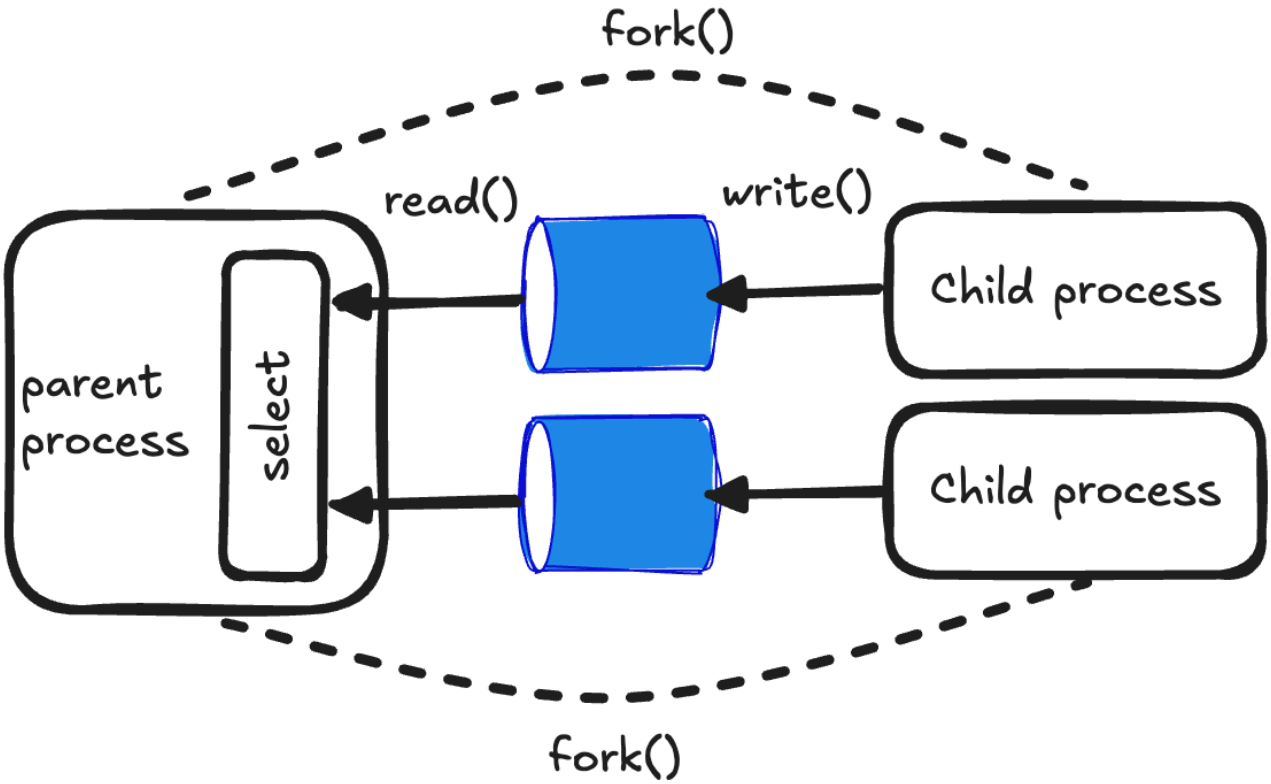
In Task 1A, a single pipe connects one parent and one child. Assume that the producer stage is much faster than the consumer stage. A good solution to avoid a consumer bottleneck is to use more CPUs and child processes to parallelise the consumer work. In the sample code the parent write to a single pipe only and the write operations will be stopped if the consumer is slow and the pipe is full. But what if the parent must listen to **multiple pipes** at once — for example, several processes sending data to a single process? Using the existing code and simple for loop to write an element at a time to each pipe would be inefficient, as the parent would block on one pipe and potentially starve other child processes which are ready to process new data.

What is `select()`?

`select()` is a system call that allows a process to inspect the read and write status of multiple file descriptors (FDs) at once, i.e. whether the process can read or write without blocking.
A call to select accepts a list of FDs and blocks until one or more of them become “ready” for I/O — for example, a read descriptor has data available, or a write descriptor has buffer space.
In Python, you can use it as:

```
rlist, wlist, xlist = select.select(read_fds, write_fds, except_fds)
```

The call returns three lists of FDs that are ready. If you only care, for example, about readability (data to read or EOF), you can pass empty lists for write_fds and except_fds. When a pipe reaches EOF (because the writer has closed it), a read on that FD returns an empty byte string b”, and it should be removed from the monitored set.



I/O multiplexing with select() system call.

Scenario

For this task you will use the template code in file `task1B.py`. The code implements a producer-consumer scenario, where multiple child processes write data to their own pipes. The parent process forks multiple child processes and reads for all pipes simultaneously.

Each child process repeatedly writes short messages into its pipe and then exits. The parent uses `select()` to monitor all the read ends of the pipes simultaneously and print whichever messages arrive first.

This example demonstrates event-driven I/O — the parent doesn't waste CPU cycles polling or waiting on one pipe at a time. Instead, the kernel notifies it only when at least one child's pipe is ready.

In order to complete this task, you should modify the select() call in the function `parent_event_loop()`. Make sure that the select function is called with the correct arguments so that the parent can monitor all the read file descriptors of the pipes created for each child process. Please see the Python documentation for `select`: Python `select` module documentation (https://docs.python.org/3/library/select.html)

Task1B Reflection Questions

1. Why select()? What advantage does event-based concurrency provide compared to looping with blocking reads?
2. Why must you remove an FD from the monitored set after EOF? What goes wrong if you leave it in?
3. If one child is “chatty” (writes very often), how can the parent avoid starving other children in the event loop?

Part 2 — Threads & Shared Memory

Task 2A — Concurrency with a Producer–Consumer Queue

Threads are *lightweight* execution contexts that share the same memory address space. Because they share memory, threads can communicate through shared variables instead of using pipes as in message passing. This makes communication fast — but also dangerous if multiple threads modify the same data simultaneously, which can lead to *race conditions* (https://en.wikipedia.org/wiki/Race_condition).

Because Python is an interpreted language, it cannot run multiple threads in parallel; it has a Global Interpreter Lock (GIL) that prevents multiple native threads from executing Python bytecodes at once. Nonetheless, for **I/O-bound** programs, Python threads can execute concurrently, even with the Global Interpreter Lock (GIL), because when a thread performs blocking I/O (for example, reading from disk or waiting to receive network data), it **releases the GIL** and lets another thread run.

A classic example is the **producer–consumer pattern**, where:

- The **producer** generates tasks (for instance, URLs to download or jobs to process).
- The **consumers** process these tasks concurrently.

The shared data structure between producers and consumers is typically a **queue** data type. Accessing the queue represents a **critical section** — the part of code where the memory shared by multiple threads is modified and must be protected against concurrent access.

Python's `queue.Queue` (https://docs.python.org/3/library/queue.html#queue.Queue) class which implements this protection internally using a `Lock` and two `Condition` variables (`not_empty`, `not_full`). You don't need to add your own locks around `put()` and `get()`. However, it's still useful to think of those calls as the conceptual critical section.

For this task we provide you with a fully implemented program in file `task2A.py`. The code simulates a network workload, where several web pages are "downloaded" (simulated with a sleep to represent I/O wait) and then a small CPU-bound computation is performed on the downloaded data (the program will not download real data). The simulation allows you to control the number of parallel threads and the amount of network and CPU work for each task, thus allowing you to experiment with multiple scenarios. For each scenario, the simulation replicates two scenarios; a threaded scenario, where multiple threads execute concurrently, and a sequential scenario, where all tasks are performed one after the other. The time taken for each scenario is printed at the end of the run for comparison.

The implementation realizes a producer–consumer system using threads and a shared queue. At the top of the source code you will find several tunable parameters:

- `N_TASKS`: number of simulated download+compute tasks to perform.
- `IO_DELAY`: seconds of simulated I/O wait per task.
- `NUM_THREADS`: number of consumer threads.
- `COMPUTE_SIZE`: small CPU work to follow the I/O wait.
- `VERBOSE`: set False for quieter output.

For this task, you do not need to modify the code, but use the provided code to explore the benefits of multithreading. You should study the code and understand how it works. If some parts of the code are clear, **do not hesitate to discuss them with a lab assistant**. Following that, you should experiment with changing the tunable parameters to see how they affect performance. In particular, run the following experiments:

- Increase N_TASKS; keep IO_DELAY fixed.
- Change NUM_THREADS to 1, 2, 4, 8 and observe diminishing returns.
- Set IO_DELAY near zero to see that threads no longer help (now it's CPU-bound).

Record the timings for each experiments and measure the performance improvements (or lack thereof) when using threads for different workloads. For each experiment, you should:

- Compare sequential vs threaded execution of simulated I/O tasks.
- Observe how threads overlap waiting and computation.

- Understand where the critical section lies in the producer–consumer model.

Task 2B Reflection Questions

1. Why do threads help here? Explain how blocking I/O allows multiple Python threads to make progress.
2. How does this shared-memory approach differ from Task 1A and 1B (pipes and select)? What are the trade-offs?
3. Scaling limits. Why do speedups diminish as you add more threads?

Task 2B — Coordinating Threads with Condition Variables

In many situations, threads must **wait for each other** — not because of a race condition, but because one thread depends on another thread's progress.
For example:

- A consumer may need to wait until the producer adds something to the buffer.
- A thread might wait until a computation or I/O completes before continuing.

We could repeatedly check ("busy-wait") until the condition becomes true, but that wastes CPU time. Instead, we use a **Condition Variable**, which lets a thread sleep until another thread **signals** that something has changed.

In Python, `threading.Condition` (https://docs.python.org/3/library/threading.html#threading.Condition) wraps a lock. A thread calls:

- `condition.wait()` (https://docs.python.org/3/library/threading.html#threading.Condition.wait) → releases the lock and sleeps until another thread calls `condition.notify()` (https://docs.python.org/3/library/threading.html#threading.Condition.notify).
- `condition.notify()` (https://docs.python.org/3/library/threading.html#threading.Condition.notify) or `condition.notify_all()` (https://docs.python.org/3/library/threading.html#threading.Condition.notify_all) → wakes one or all waiting threads.

A sample Python code snippet is shown below for the waiter:

```
with condition:
    while not <condition_is_true>:
        condition.wait()
    # safe to proceed; condition holds
```

and the signaler:

```
with condition:
    <change shared state>
    condition.notify_all()
```

Scenario

For this task, we provide the template code in file `task2B.py`. The code:

- implements a small producer–consumer example using a shared buffer protected by a lock.
- uses a condition variable to coordinate the producer and consumer threads without busy waiting.
- observes the output as the threads take turns producing and consuming.

How it works:

1. The producer acquires the condition lock before modifying the buffer (line 16). If the buffer is full, it calls condition.wait(), which releases the lock and suspends the producer (*lines 17-20*).
2. When a consumer removes an item, it calls condition.notify() to wake the producer, which can now add another item (*line 53*).
3. Similarly, consumers wait when the buffer is empty and are woken by the producer's condition.notify() (*lines 39-42*).
4. A sentinel value (None) tells all consumers to stop gracefully (*line 45-48*).

This example illustrates coordination, not mutual exclusion: the lock protects shared state, while the condition variable provides a way to sleep until a condition changes.

For this task, you do not need to modify the code, but you should study it carefully to understand how the condition variable is used to coordinate the producer and consumer threads. You can run the code as is to observe its behavior. Using your code understanding,

answer the reflection questions below. If some parts of the code are unclear, **do not hesitate to discuss them with a lab assistant**.

Task 2B Reflection Questions

1. What would happen if consumers repeatedly checked BUFFER in a loop instead of condition.wait()?
2. Explain how the condition uses a lock for mutual exclusion and a queue for thread coordination.
3. **Notify vs notify_all**. Why might you use notify_all() instead of notify() in some cases? What happens if multiple consumers are waiting?
4. **Race prevention**. Why do we check the condition (while not BUFFER) again after waking from wait()?
5. **Generalization**. How does this pattern apply to real-world systems such as I/O scheduling, thread pools, or event-driven servers?

Hacker's Edition — ThreadPool + Prime Finder (and a multiprocessing challenge)

This exercise ties together IPC, threads, and processes. Below a simple **ThreadPool** implementation and a CPU-bound task: finding prime numbers in integer ranges. The exercise demonstrates two important lessons:

- **Threads** are useful for I/O-bound tasks but *do not* give true CPU parallelism in python because of the **Global Interpreter Lock (GIL)**. Running this prime finder with multiple **threads** will usually *not* produce speedup for CPU-bound work.
- **Processes** (via `multiprocessing`) provide real parallelism across multiple CPU cores. Your challenge is to implement a process-based version of the same workload and compare performance.

In this task, you will experiment with the provided ThreadPool implementation and a prime-search demo. The demo splits the range of numbers to search for primes into tasks, submits them to the ThreadPool, and measures wall-clock time. You will:

1. Use the provided `ThreadPool` + prime-search demo to run experiments and observe that threads do not speed up this CPU-bound task.
2. Implement a **multiprocess** version that uses multiple processes to achieve true parallelism.
3. Perform measurements with varying parallelism.

For this task, we provide the template file `hacker_primes_threadpool.py`. This file is ready-to-run. It implements a small ThreadPool, splits the number range into tasks, submits them to the pool, and measures wall-clock time.

Scenario:

1. Run the threaded baseline on a lab machine with a few different settings:
    - Keep tasks reasonably large (e.g., 8 or 16), and vary workers among 1, 2, 4, 8.
    - Example:

```
python3 hacker_primes_threadpool.py --total 200000 --tasks 8 --
         workers 1
python3 hacker_primes_threadpool.py --total 200000 --tasks 8 --
         workers 4
```

    - Record wall-clock times (use the printed elapsed).
2. Observe: you will normally see little to no speedup as workers increases. Explain why (GIL → only one Python thread executes bytecode at a time; the task is CPU-bound).
3. Implement a multiprocessing version that parallelises the same work across processes.
4. Measure wall-clock times with the same total/tasks decomposition and process counts (1, 2, 4, …).