Advanced Programming

Process-based Parallelism

IMPROVE APPLICATION PERFORMANCE

- Start reviewing all the key algorithms you have learned and their find their worst -case performance
- Look for different ways to improve the performance of an algorithm – For example Bubble sort->Merge Sort -> Quick Sort
- Choose the data structure wisely – cost of accessing an element close to $O(1)$

CACHING

```python
import time

# Function to calculate numbers recursively
def fibonacci_recursive(n):
    if n <= 1:
        return n
    else:
        return fibonacci_recursive(n-1) + \
               fibonacci_recursive(n-2)

# Function to calculate with caching
fib_cache = {}
def fibonacci_with_cache(n):
    if n in fib_cache:
        return fib_cache[n]
    if n <= 1:
        return n
    else:
        fib_cache[n] = fibonacci_with_cache(n-1) + \
                       fibonacci_with_cache(n-2)
        return fib_cache[n]
```

```python
if __name__ == '__main__':
    # Calculate without caching
    start_time = time.time()
    fibonacci_recursive(35)
    end_time = time.time()
    print(f"Without caching = \
          {end_time - start_time:.4e} s")

    # Calculate with caching
    start_time = time.time()
    fibonacci_with_cache(35)
    end_time = time.time()
    print(f"With caching =
         {end_time - start_time:.4e}”)
```

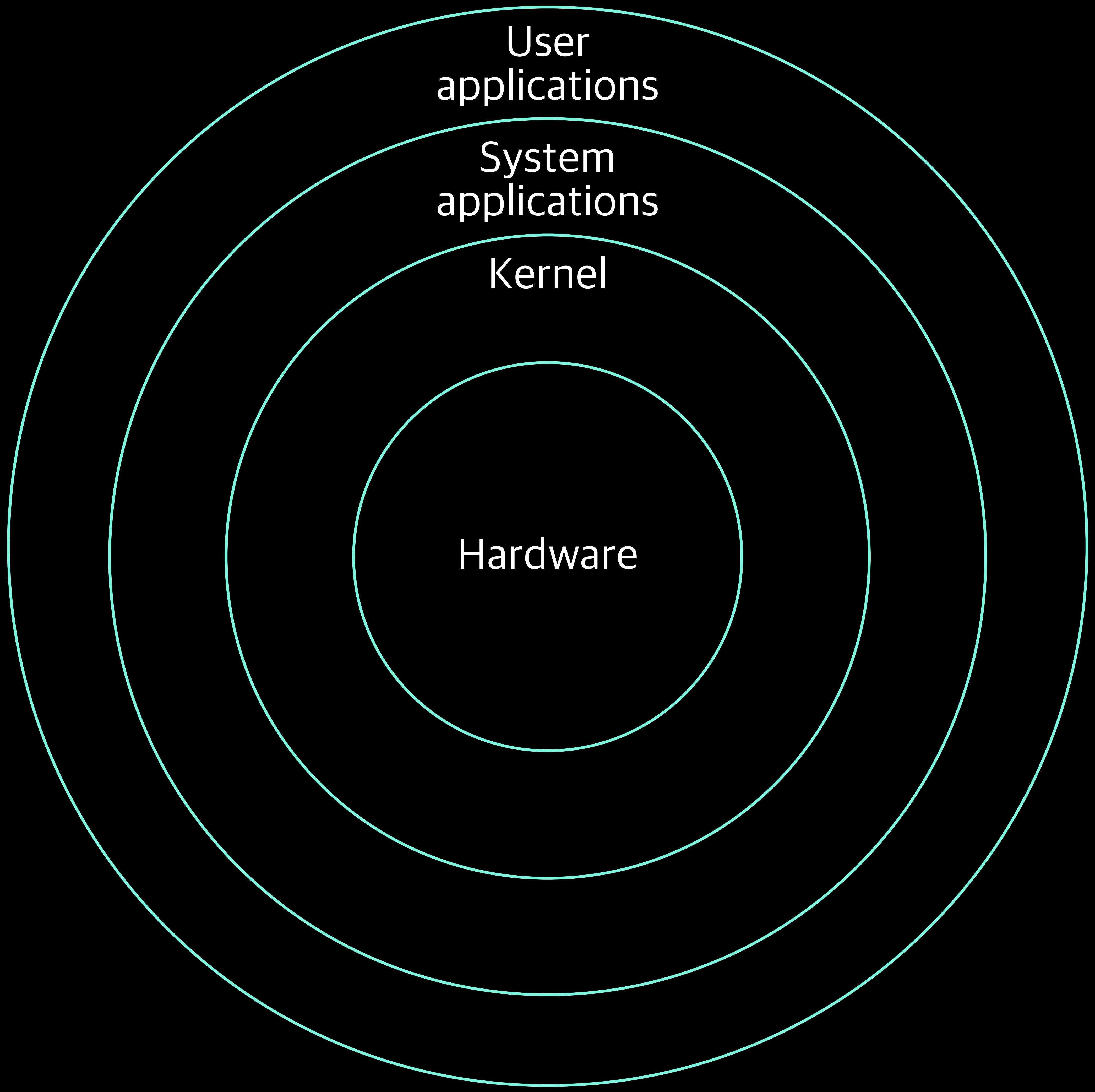
```
Without caching = 9.2852e-01 s

   With caching = 2.0981e-05 s
```

CACHING USING A DECORATOR

```python
from functools import lru_cache
import time

# Function to calculate Fibonacci numbers
@lru_cache(maxsize=None)  # No maximum cache size
def fibonacci(n):
  if n <= 1:
    return n
  else:
    return fibonacci(n-1) + fibonacci(n-2)

if __name__ == '__main__':
    # Without caching
    start_time = time.time()
    fibonacci(35)
    end_time = time.time()
    print(f"{'Without caching':<16}: {end_time - start_time:0.10e}")

    # With caching
    start_time = time.time()
    fibonacci(35)
    end_time = time.time()
    print(f"{'With caching':<16}: {end_time - start_time:0.10e}")
```

```
Without caching : 7.8678131104e-06
With caching    : 0.0000000000e+00
```

ARCHITECTURE OF A LINUX SYSTEM



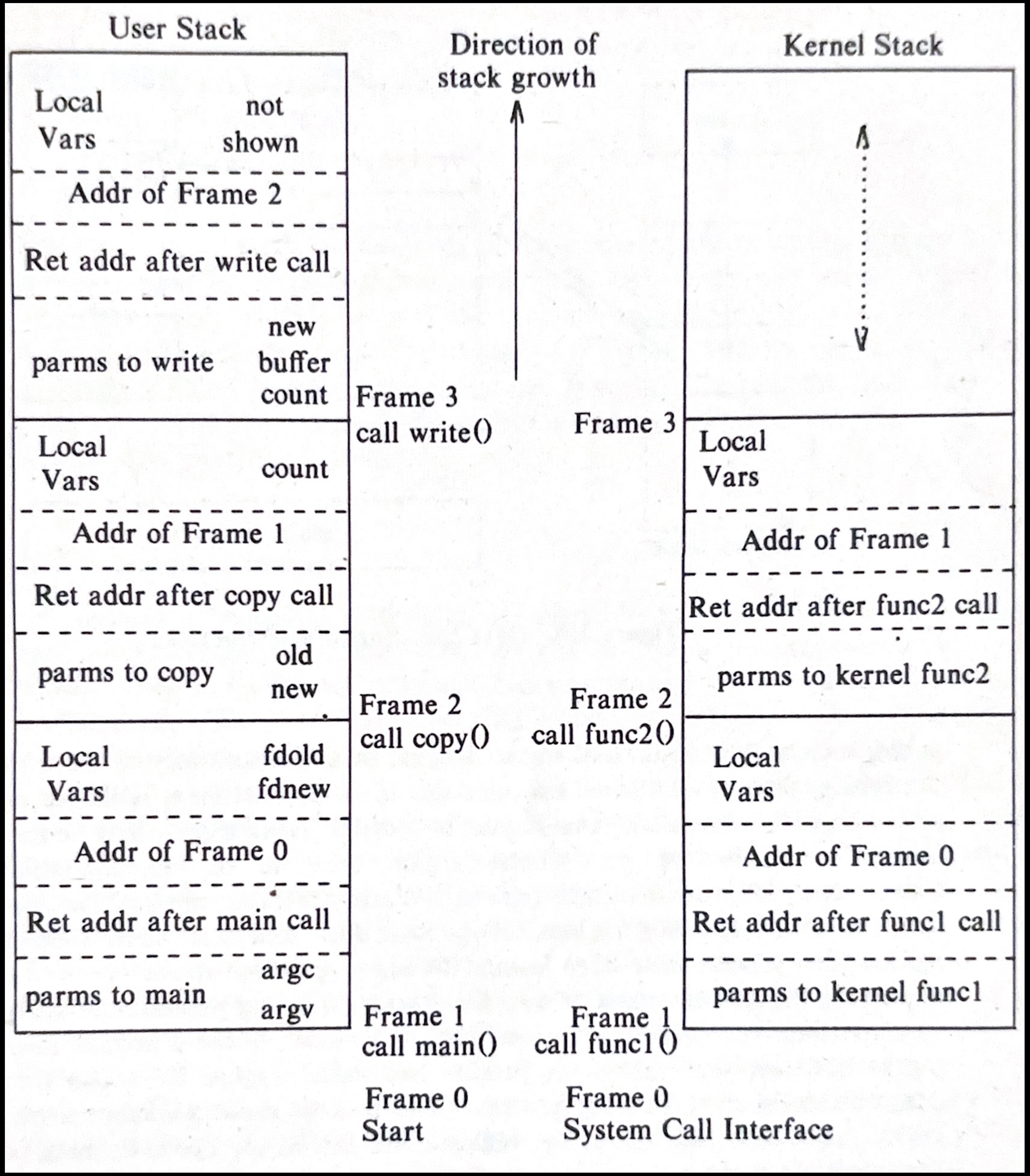
PROCESS

- A process is an active entity in an OS
- Fundamental unit of a program in an execution state
  - Consists of regions
    - patterns of bytes interpreted as instructions by CPU, called as text
    - Data
    - Stack
- Self-contained
- Reads and writes its data and stack
- Cannot read and write another process directly
- Communicates with other processes through messages using system calls

PROCESS

```
import sys

def copy(source_file, destination_file):
    try:
        with open(source_file, 'r') as src, \
        open(destination_file, 'w') as dst: \
             dst.write(src.read())
        print(f"File copied successfully from
              {source_file} to {destination_file}")
    except FileNotFoundError:
        print(f"Error: Source file '{source_file}'
              not found.")
    except IOError as e:
        print(f"Error accessing files: {e}")

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print(f"Usage: python {sys.argv[0]}
              <source_file> <destination_file>")
        sys.exit(1)

    else:
        copy(sys.argv[1], sys.argv[2])
```

USER AND KERNEL STACK FOR COPY




```python
import sys

def copy(source_file, destination_file):
    try:
        with open(source_file, 'r') as src, \
        open(destination_file, 'w') as dst: \
             dst.write(src.read())
        print(f"File copied successfully from
              {source_file} to {destination_file}")
    except FileNotFoundError:
        print(f"Error: Source file '{source_file}'
              not found.")
    except IOError as e:
        print(f"Error accessing files: {e}")

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print(f"Usage: python {sys.argv[0]}
              <source_file> <destination_file>")
        sys.exit(1)

    else:
        copy(sys.argv[1], sys.argv[2])
```

Source: The design of UNIX operating system, M.J. Bach

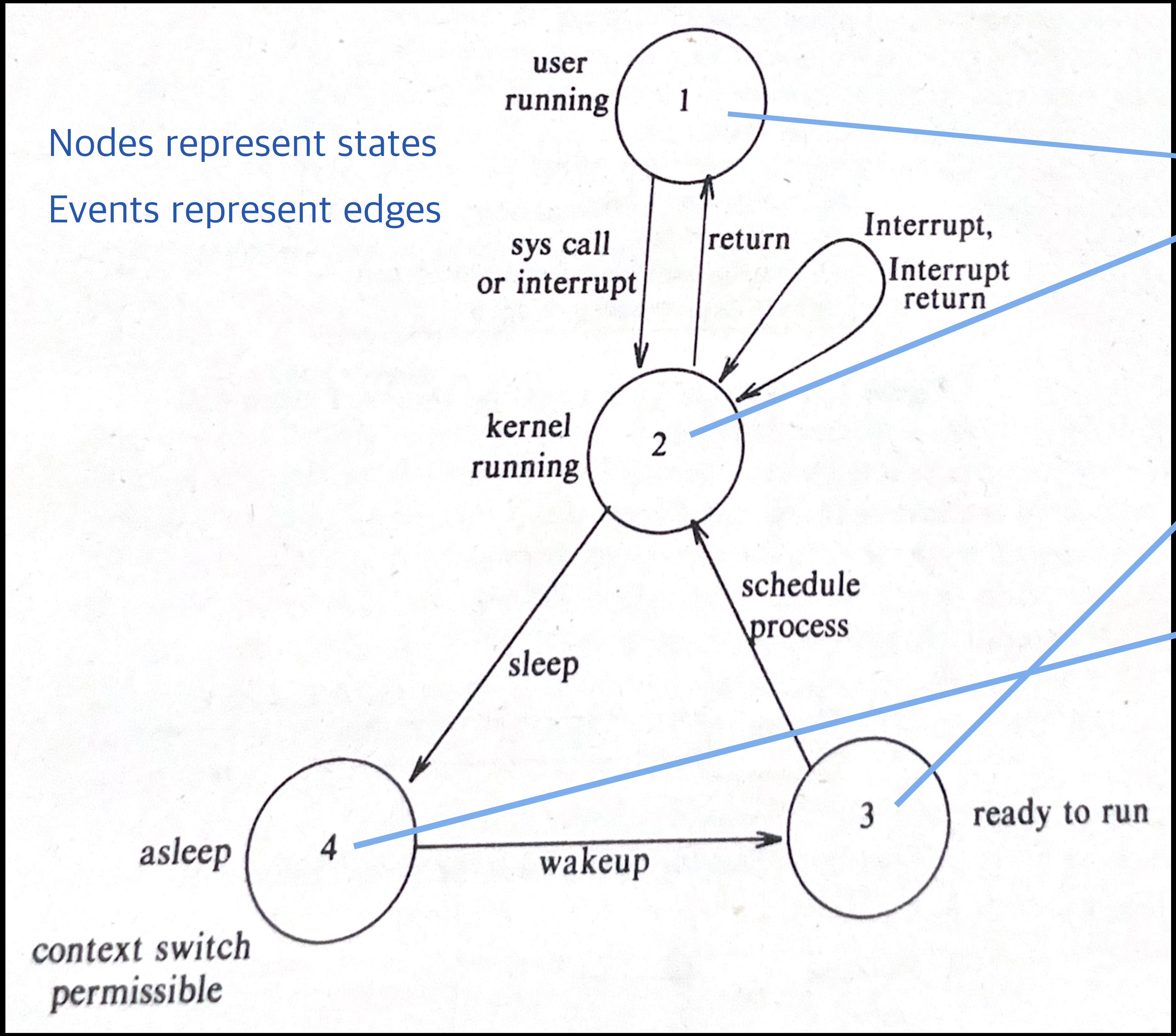
PROCESS STATES

- Currently executing in user mode
- Currently executing in kernel mode
- Waiting – Not in execution mode – waiting for the scheduler to allocate CPU
- Sleeping – Waiting for an I/O to complete

PROCESS STATES AS DIGRAPH

Nodes represent states

Events represent edges



A process may be in state 1 or 2

Many process may be in this state

Waiting for I/O to complete

CONTEXT SWITCH

- In any multitasking operating system, multiple processes can run simultaneously
- A single CPU can only execute one process at a time
- The context of a process its state – its text, values, data structure and the machine register values, content of user and kernel stack frames
- Context switching saves the context of a currently running process at time t_1 in order to continue at a future time t_n
- Allows the OS to switch between processes efficiently

- Allows all processes to share a single CP
- Facilitates efficient CPU utilisation
- Increased CPU Usage
- frequency of context switching significantly affects performance

CONCURRENCY AND PARALLELISM

- Creates an impression that multiple are running at the same time
  - Manages multiple tasks
  - Tasks seem to run at o
  - Achieved through conte
    switching on a single CPU
  - Concurrency exhibits non-deterministic control flow
  - Browser - download and browsing

Due to

Asynchronous Execution
Context-switching
Resource sharing
Synchronisation

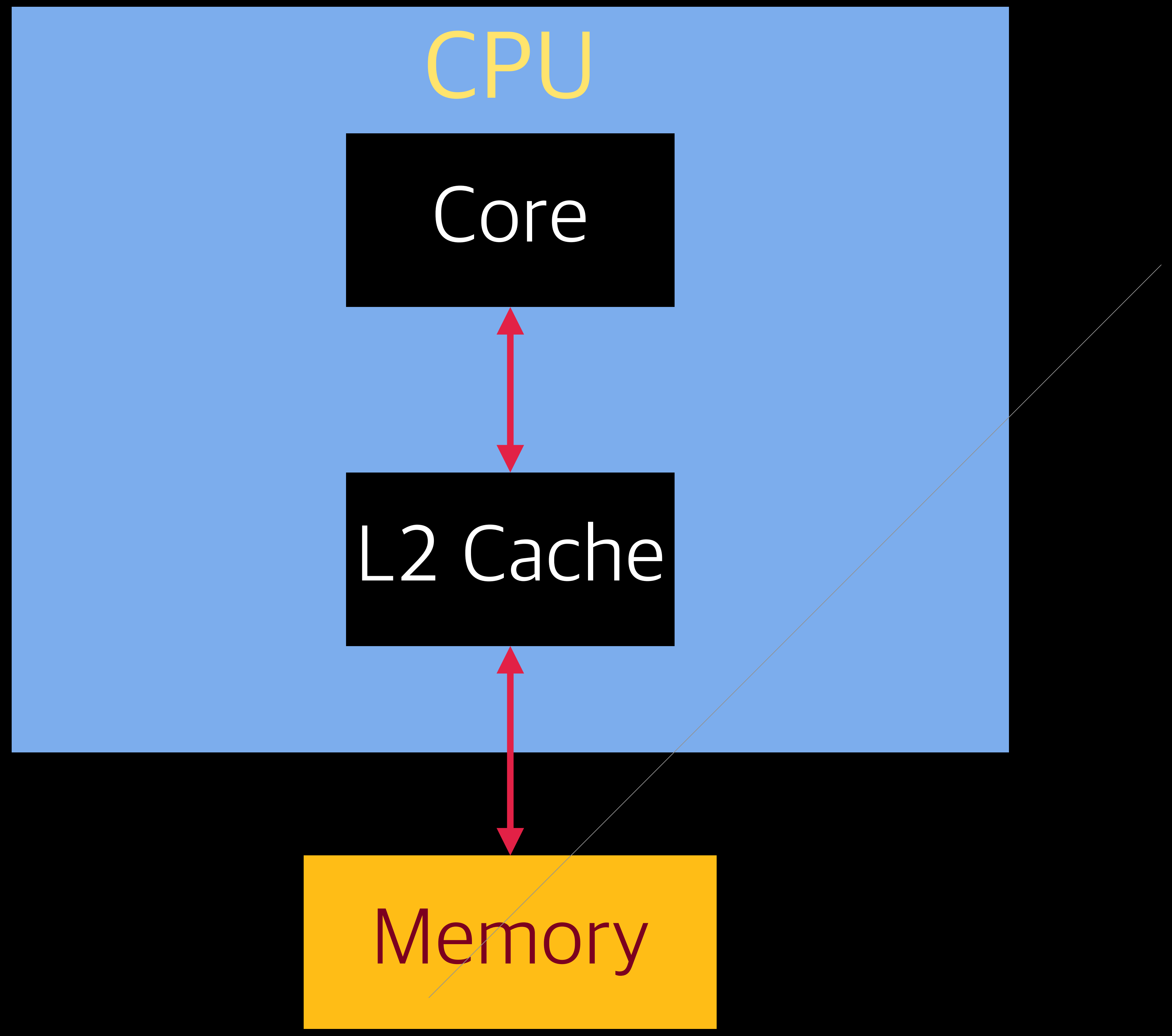
To learn more about Concurrent programming, listen to this lecture

https://www.youtube.com/watch?v=XbdDSUI8NXE

- Runs multiple computations simultaneously
- Tasks truly run at once
- Multiple CPU cores/distributed systems

LECTURE 2 - MULTIPROCESSING

SINGLE CORE CPU



L1 Cache – small memory – faster than main memory

L2 Cache > L1 Cache – faster than main memory

Store frequently accessed data and instructions to speed up the CPU's access to them

BERNSTEIN'S CONDITIONS

- Conflict Serializability
  - One transaction reads and another writes the same data item
- Write-Write Conflict
  - Two different processes/transactions writing on the same critical section at the same time
- Write-Read Conflict

THREADS

- A thread is a lightweight process with in a program
- Threads within a program share the same memory space. - communication is efficient
- Threads can execute concurrently,
- Synchronization - Due to sharing of memory, it is important to manage critical section
- OS manages the execution of threads using context switching
- Threads have a lifecycle consisting of creation, running, blocking, and termination phases

PYTHON MODULES

- Threading
- Multiprocessing

THREADS IN PYTHON

- Concurrency: Allows concurrent execution of tasks,
  - Improves (?) performance on multi-core systems
- Lightweight: efficient for smaller tasks - suitable for tasks such as I/O operations
- Shared Memory: Threads within a program share the same memory space
- Synchronisation: Required to avoid race conditions
- Context Switching: OS manages thread execution using context switching
- Global Interpreter Lock (GIL) restricts true parallelism
  - Must acquire a lock to run a thread in the interpreter space
  - -Ensures only one thread at any point in time - prhibits protrue parallelism

```
import threading
```

THREAD EXAMPLE 1

```python
import threading
import time

def print_message(message):
  print(message)
  time.sleep(1)

# Create a new thread
thread = threading.Thread(target=print_message,
                          args=("Hello from Thread!",))

# Start the thread
thread.start()
```

THREAD EXAMPLE 2

```
import threading

def square(numbers):
    for num in numbers:
        print(f"Square of {num}: {num*num}")

if __name__ == '__main__':
    numbers = list(range(1, 11))

    half_len = len(numbers) // 2
    first_half = numbers[:half_len]
    second_half = numbers[half_len:]

    thread1 = threading.Thread(target=square, args=(first_half,))
    thread2 = threading.Thread(target=square, args=(second_half,))

    thread1.start();thread2.start()
    # Wait for both threads to finish
    thread1.join();thread2.join()

    print(“Finished and exiting main thread")
```

SERIAL COMPUTING

```python
import numpy as np
import time

def square(seed):
    np.random.seed(seed)
    random_num = np.random.randint(0, 1000000)
    return random_num**2

def serial_main():
    start_time = time.time()
    for i in range(100000):
        square(i)
    end_time = time.time()
    print(f'Serial:Total time taken =  {end_time - start_time:.2f} seconds')

if __name__ == '__main__':
    serial_main()
```

CRITICAL SECTION

- A critical section is a specific part of a code that needs to be executed by multiple threads or processes atomically
  - To ensure data consistency and integrity
- Threats:
  - Race Conditions: If multiple threads access the same critical section concurrently

MUTEX

Synchronisation primitives used for coordinating access to shared resources in multi-threaded environments

SEMAPHORE

- A semaphore is a variable that acts as a control mechanism for managing access to shared resources by multiple threads, preventing race conditions
- semaphore = 4 - 4 processes can run at a time concurrently
- If semaphore == 0, the process requesting will wait until semaphore > 0

MUTEX VS SEMAPHORE

Feature	Mutex	Semaphore
Function	Ensures only one thread can access a critical section at a time	Signaling mechanism - Controls access to a shared resource by multiple threads
Ownership	Yes - Thread that acquires the lock becomes the owner.	No - Any thread can wait on and signal the semaphore.
Signaling	Limited - Thread waits until lock is released.	Wait and signal operations - Thread waits until value > 0 (resource available) and increments when done.
Use Cases	Protecting critical sections to avoid race conditions.	Managing access to a limited pool of resources (e.g., database connections, network requests).

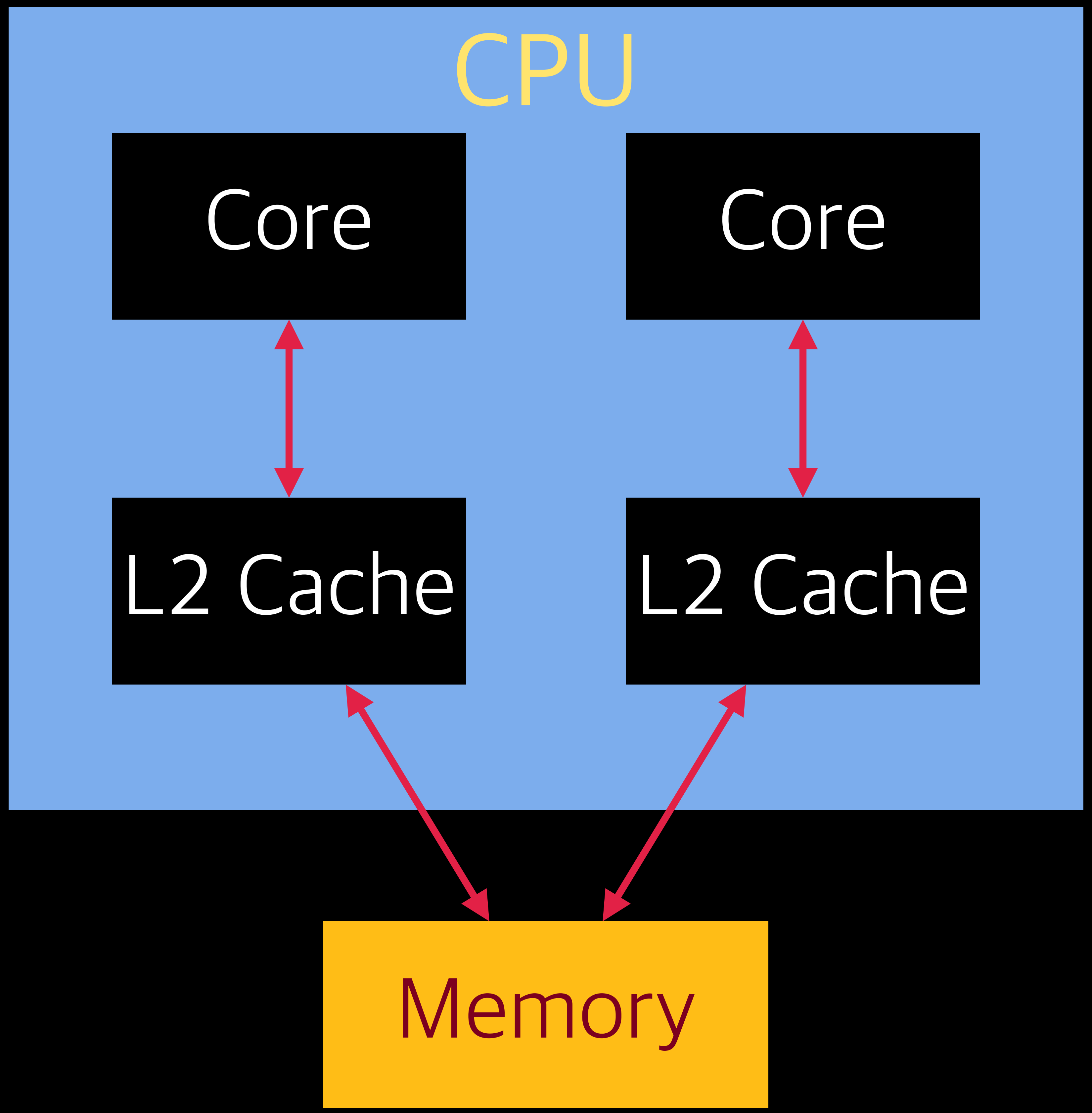
THREAD SAFETY

- Thread safety refers to the ability of a program to function correctly when executed by multiple threads concurrently
- Unpredictable behavior happens when critical sections are accessed without proper synchronisation
- Multithreaded environment,
  - The program state must remain consistent
  - Data integrity is maintained

THREADS COMPUTING :)

```python
import numpy as np
import time
import threading

def threads_main():
    start_time = time.time()
    threads = []
    for i in range(100000):
        thread = threading.Thread(target=square,args=(i,))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    end_time = time.time()
    print(f"Total time taken: {end_time - start_time} seconds")
```

DUAL CORE CPU



PROCESS

- A process is an independent unit of execution in a computer program.
- Processes are independent - have separate memory spaces.
- Creation - Creation typically involves forking a new process from an existing one
- Communication - Inter-Process Communication (IPC) using pipes, shared memory, message queues, and sockets
- Resource Management: Processes consume system resources such as memory, CPU time, and I/O resources.
- Process Scheduling - The OS scheduler determines the order in which processes are executed on the CPU.
- Process Identification - Each process is unique process

MULTIPROCESSING IN PYTHON

- Supports spawning process.
- Offers local and remote concurrency
- Multiprocessing utilises the physical cores the CPU
  - Creates subprocesses instead of threads
  - Leverages multiple cores

```
import multiprocessing as mp

def square(num, result):
  result.value = num * num

if __name__ == "__main__":
    number = 5
    # Shared variable to store the result
    result = mp.Value('i', 0)  # 'i' indicates integer type
    # Create a process to perform the calculation
    process = mp.Process(target=square, args=(number, result))
    process.start()
    process.join()
    print(f"The square of {number} is: {result.value}")
```

```
thread = threading.Thread(target=square,args=(i,))
```

MULTIPROCESSING WITH MANAGER

```python
def square(numbers,
           start_index,
           end_index,
           squared_list):
  for i in range(start_index, end_index + 1):
    squared_list.append(numbers[i] * numbers[i])
```

```python
if __name__ == "__main__":
  numbers = list(range(1, 11))
  num_processes = 4
  chunk_size = len(numbers) // num_processes
  sub_ranges = []
  for i in range(num_processes):
    start = i * chunk_size
    end = min((i + 1) * chunk_size - 1, len(numbers) - 1)
    sub_ranges.append((start, end))
  manager = Manager()
  squared_list = manager.list()
  processes = []
  for start, end in sub_ranges:
    process = Process(target=square,
                      args=(numbers, start, end, squared_list))
    process.start()
    processes.append(process)
  for process in processes:
    process.join()

  print(f"Square for {numbers}")
  for num in squared_list:
    print(num)
```

MULTIPROCESSING USING POOL

- The Pool class of multiprocessing simplifies the coding using a map-reduce process
  - Distributes the work among the subprocess using map
  - Collect the return value as a list
  - Relieves the programer from the burner of
    - managing processes
    - Mainting a shared data/state

```
from multiprocessing import Pool
import multiprocessing as mp


pool = Pool(processes=4)
#OR
cpu_count = mp.cpu_count()
pool = Pool(processes=cpu_count)
```

MULTIPROCESSING EXAMPLE

```python
import numpy as np
import time
import multiprocessing as mp

def parallel_main():
    start_time = time.time()

    cpu_count = mp.cpu_count()

    pool = mp.Pool(processes=cpu_count)
    _ = [pool.map(square, range(100000))]
    end_time = time.time()
    print(f"Parallel:Total time taken: {end_time - start_time} seconds")

if __name__ == '__main__':
    parallel_main()
```

PRACTICAL EXAMPLES

- Reading files in parallel and combine the results
- Estimating π using Monte Carlo Simulation

POOL

- One can create a pool of processes which will carry out tasks submitted to it with the Pool class
- If processes is None then the number returned by os.cpu_count() is used

```
if chunksize is None:
    chunksize, extra = divmod(len(iterable), len(self._pool) * 4)
    if extra:
        chunksize += 1
if len(iterable) == 0:
    chunksize = 0
```

```
p = Pool(processes=os.cpu_count())

print(len(p._pool)) #8
```

ITERABLE - (LIST, DICT, TUPLE)

- An object capable of returning its members one at a time.
- List, str, tupe, dict, file objects, and objects of any classes you define with an __iter__() method or with a __getitem__() method that implements sequence semantics

```
text = "This is iterable"
for alpha in text:
    print(alpha)

tokens = ["This", "is", "iterable"]
for token in tokens:
    print(token)

dict_tokens = {"This":1, "is":2, "iterable":3}
for key in dict_tokens:
    print(f'key={key}, value ={dict_tokens[key]}')


trigram = ("This", "is", "iterable")
    for token in trigram:
        print(token)
```

ITERABLE

- An object capable of returning its members one at a time.
- List, str, tupe, dict, file objects, and objects of any classes you define with an **__iter__()** method or with a __getitem__() method that implements sequence semantics

```python
class FibSequence:
  def __init__(self, max_count):
    self.max_count = max_count
    self.a, self.b = 0, 1
    self.count = 0

  def __iter__(self):
    return self

  def __next__(self):
    if self.count > self.max_count:
      raise StopIteration
    current_value = self.a
    self.a, self.b = self.b, self.a + self.b
    self.count += 1
    return (self.count,current_value)

if __name__ == '__main__':
    fib_seq = FibSequence(20)
    for i,num in enumerate(fib_seq):
        print(f'{i}:{num}')
```

ITERABLE

- An object capable of returning its members one at a time.
- List, str, tupe, dict, file objects, and objects of any classes you define with an __iter__() method or with a **__getitem__()** method that implements sequence semantics

```python
class ColorList:

  def __init__(self, color_names):
    self.color_names = color_names

  def __getitem__(self, index):
    if 0 <= index < len(self.color_names):
      return self.color_names[index]
    else:
      raise IndexError("Index out of range")

if __name__ == "__main__":
    color_names = ColorList(["Red", "Green", "Orange"])
    for color in color_names:
        print(color)
    # Access elements by index
    print(color_names[0])
    print(color_names[2])
```

PYTHON OBJECT SERIALISATION - PICKLE

- Useful in transmitting Python objects
- **Pickling** - The process of converting an object into a byte stream
- **Unpickling**: The process of reconstructing an object from a byte stream
- **Picklable**: An object that can be successfully pickled and unpickled

Importance of Piclability

- **Data Persistence**: Save objects to disk for later use
- **Remote Procedure Calls (RPC)**: Python objects across processes or machines in distributed systems
- **Machine Learning Pipelines**: Saving trained models and deploying them in different environments.

What can be pickled and unpickled?

Warning The pickle module is not secure. Only unpickle data you trust.

PICKLE - EXAMPLE

```python
if __name__ == '__main__':
    fib_seq = FibSequence(21)
    with open('fib_seq.pickle', 'wb') as f:
        pickle.dump(fib_seq, f)

    with open('fib_seq.pickle', 'rb') as f:
        load_fib_seq:FibSequence = pickle.load(f)

    for fib in load_fib_seq:
        print(fib)
```

PROCES TIME

Chunk size	Elapsed Time	# processes
None	11.7	8
2	12.08	8
4	11.01	8
8	11.62	8
16	11.87	8
32	11.64	8
64	10.75	8
128	10.47	8
256	10.76	8

Chunk size	Elapsed Time	# processes
None	10.66	16
2	11.46	16
4	11.97	16
8	9.21	16
16	9.03	16
32	8.83	16
64	8.54	16
128	9.30	16
256	9.97	16

```
Elapsed time for converting a json text to plain text using multiprocessing
```

```
Seq_Process: elapsed time = 40.385626792907715
```

JSON2TEXT

```python
def par_write(files):
    cpu_count = os.cpu_count()
    file_count = len(files)
    p = Pool(processes=16)
    p.map(write_file,files, chunksize=16)
    p.close()
```

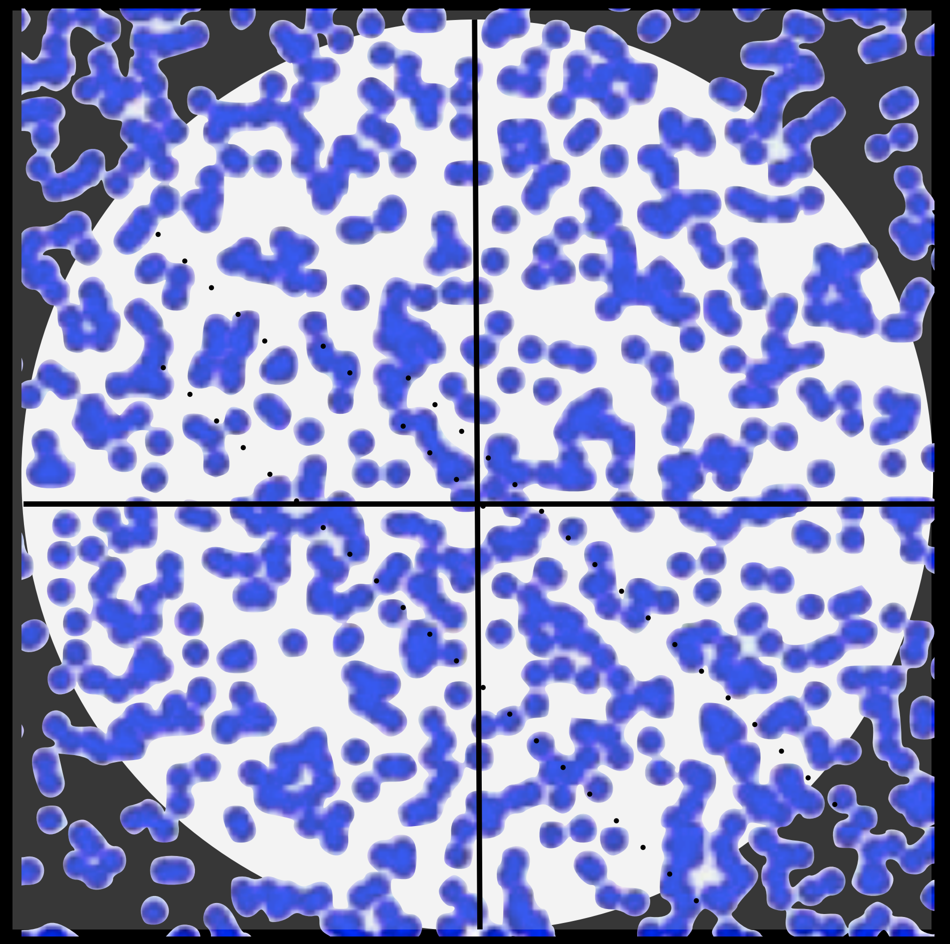
```python
def par_main():
    json_path = '/Users/ram/Teaching/Demos/NLP/Python/COVID19/pdf_json'
    json_files = glob.glob(f'{json_path}/**/*.json',recursive=True)
    start_time = time.time()
    par_write(json_files)
    end_time = time.time()
    print(f'elapsed time = {end_time - start_time :0.2f}')
```

MONTE CARLO SIMULATION

- Developed in the 1940s during the creation of the atomic bomb
- Monte Carlo methods are a class of computational algorithms
- They can be applied to a wide range of problems, not limited to statistics
- Instead of relying on pure statistical analysis, repeated random sampling to arrive at solutions is used
- Monte Carlo methods typically provide approximate solutions, not exact answers
- Valuable when analytical or numerical solutions are either unavailable or too complex to implement

ESTIMATION OF Π

- Define the Simulation Space:
  - Imagine a square with a side length 2 units, centred at the origin (0, 0)
  - This square will enclose a circle with radius 1 unit
- Set the total number of random points (num_points)
  - The points ether lie in the square or in the circle
- Set the $points\_in\_circle = 0$
- Iterate for all num_points times
  - In each iteration a random point (x, y) is thrown
  - Generate a random x-coordinate (x) between 0 and 1
  - Generate a random y-coordinate (y) between 0 and 1
  - If the point (x, y) is with in the circle,
  - $points\_in\_circle \leftarrow points\_in\_circle + 1$
    - Calculate the Euclidean distance, $E(x, y) = \sqrt{(x^2 + y^2)}$
    - If $E(x, y) \leq 1$, then the point falls inside the circle.
- $E_\Pi = \dfrac{points\_in\_circle}{num\_points} \times 4$
- The more points simulated (higher num_points), the closer the pi_estimate will be to the actual value of Pi (approximately 3.14159).



Area of the circle = πr^2

Area of the square = $4r^2$

$$\frac{\text{Area of the circle}}{\text{Area of the square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

ESTIMATION OF PI

```
def calculate_pi(iterations):
    points_in_circle = 0
    for _ in range(iterations):
        x = random.random()
        y = random.random()
        if (x**2 + y**2) <= 1:
            points_in_circle += 1
    pi_estimate = (4 * points_in_circle) / iterations
    return pi_estimate
```

```
def main():
    num_processes =  os.cpu_count()
    num_iterations = 100000

    with Pool(processes=num_processes) as pool:
        pi_estimates = pool.map(calculate_pi, [num_iterations] * num_processes)

    average_pi = sum(pi_estimates) / len(pi_estimates)
    print(f"Estimated pi using Monte Carlo simulation: {average_pi}")


if __name__ == "__main__":
    main()
```