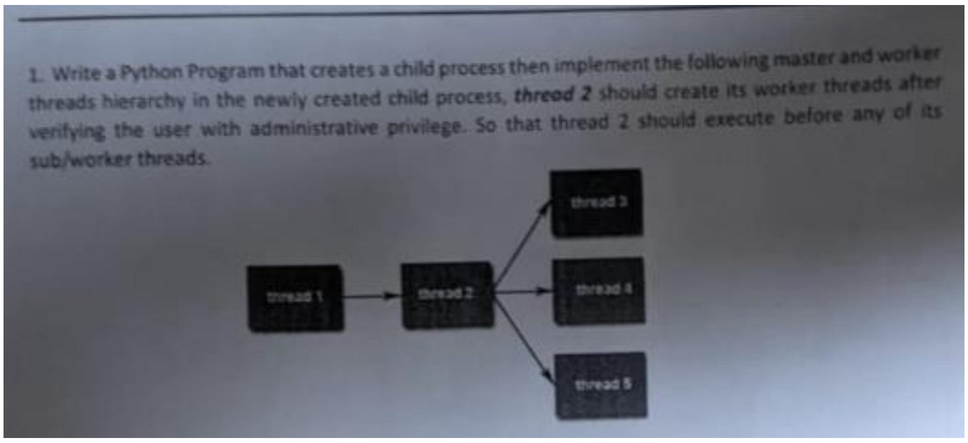
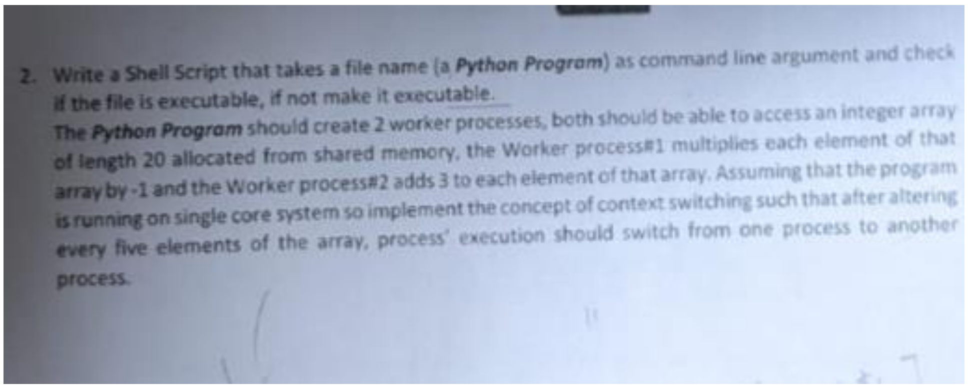

1. Write a Python Program that creates a child process then implement the following master and worker threads hierarchy in the newly created child process, ***thread 2*** should create its worker threads after verifying the user with administrative privilege. So that thread 2 should execute before any of its sub/worker threads.

```
import os
import threading

def worker_thread(idx):
    print("Worker thread", idx, "started")

def admin_thread():
    print("Thread 2 started")
    username = input("Enter your username: ")
    if username == "admin":
        print("User has administrative privileges")
        for i in range(3):
            t = threading.Thread(target=worker_thread, args=(i,))
            t.start()
    else:
        print("User does not have administrative privileges")

def master_thread():
    print("Thread 1 started")
    t2 = threading.Thread(target=admin_thread)
    t2.start()

# create child process
pid = os.fork()
if pid == 0:
    master_thread()
else:
    print("Parent process")
```



2. Write a Shell Script that takes a file name (a ***Python Program***) as command line argument and check if the file is executable, if not make it executable.
The ***Python Program*** should create 2 worker processes, both should be able to access an integer array of length 20 allocated from shared memory, the Worker process#1 multiplies each element of that array by -1 and the Worker process#2 adds 3 to each element of that array. Assuming that the program is running on single core system so implement the concept of context switching such that after altering every five elements of the array, process' execution should switch from one process to another process.

#Here is a sample shell script that takes a file name as a command line argument, checks if it's executable, and if not, makes it executable:

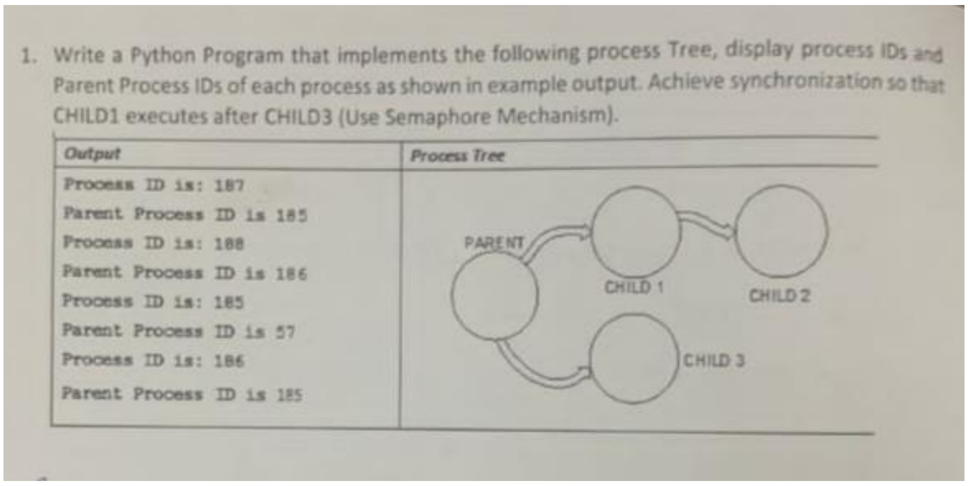
```
#!/bin/bash

file_name=$1

if [ -x "$file_name" ]; then
  echo "File is already executable"
else
  echo "Making file executable"
  chmod +x "$file_name"
fi

python $file_name
```

#Here is a sample Python code that creates 2 worker processes, both of which can access an integer array of length 20 allocated from shared memory:

```
import os
import mmap
import time

def worker_process(id, shared_memory):
    for i in range(0, 20, 5):
        if id == 1:
            for j in range(i, i + 5):
                shared_memory[j] *= -1
        else:
            for j in range(i, i + 5):
                shared_memory[j] += 3
        time.sleep(1)
```

```
if _name_ == '_main_':
    shared_memory = mmap.mmap(-1, 20 * 4)
    shared_memory[:] = bytes(20 * [0])

    pid1 = os.fork()
    if pid1 == 0:
        worker_process(1, shared_memory)
        os._exit(0)

    pid2 = os.fork()
    if pid2 == 0:
        worker_process(2, shared_memory)
        os._exit(0)

    os.waitpid(pid1, 0)
    os.waitpid(pid2, 0)
```

#In this code, the mmap module is used to create a shared memory. The worker_process function performs the task of multiplying/adding to the elements of the shared memory array. Each worker process is created using os.fork() and performs the task in worker_process function. The parent process waits for the completion of both worker processes using os.waitpid(pid1, 0) and os.waitpid(pid2, 0). The execution of worker processes is switched every 5 elements by using the sleep function in the worker processes.



1. Write a Python Program that implements the following process Tree, display process IDs and Parent Process IDs of each process as shown in example output. Achieve synchronization so that CHILD1 executes after CHILD3 (Use Semaphore Mechanism).

Output

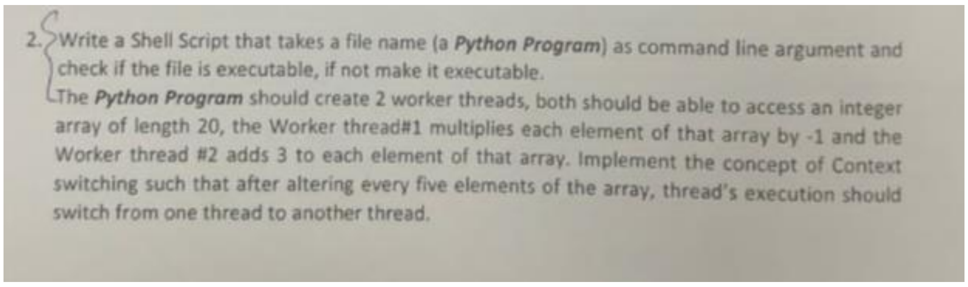
```
Process ID is: 187
Parent Process ID is 185
Process ID is: 188
Parent Process ID is 186
Process ID is: 185
Parent Process ID is 57
Process ID is: 186
Parent Process ID is 185
```

Process Tree

#Here's a sample Python code that implements the process tree you described using the multiprocessing library, which provides an easier to use API for creating and managing processes compared to using os.fork directly:

```
import multiprocessing
import os
import time

def child_process(semaphore):
    semaphore.acquire() # Wait for CHILD 3 to complete
    print("Process ID is:", os.getpid(), "Parent Process ID is", os.getppid())
    semaphore.release()

if _name_ == '_main_':
    semaphore = multiprocessing.Semaphore(0)
    child_process_1 = multiprocessing.Process(target=child_process, args=(semaphore,))
    child_process_3 = multiprocessing.Process(target=child_process, args=(semaphore,))
    child_process_2 = multiprocessing.Process(target=child_process, args=(semaphore,))

    child_process_3.start()
    child_process_3.join()
    child_process_1.start()
    child_process_2.start()

    child_process_1.join()
    child_process_2.join()

    print("Process ID is:", os.getpid(), "Parent Process ID is", os.getppid())
```

#In this code, the child_process function is the code that will run in each of the child processes. The semaphore is used to synchronize the execution of CHILD 1 and CHILD 3, so that CHILD 1 will only start executing after CHILD 3 is complete. The multiprocessing.Semaphore is initialized with a value of 0, which means that the CHILD 1 will block on semaphore.acquire() until semaphore.release() is called elsewhere. child_process_3.start() starts CHILD 3 and child_process_3.join() blocks until CHILD 3 is complete. After CHILD 3 is done, CHILD 1 and CHILD 2 can both start executing because semaphore.release() has been called and the semaphore is now available. Finally, the parent process prints its process ID and parent process ID.



2. Write a Shell Script that takes a file name (a ***Python Program***) as command line argument and check if the file is executable, if not make it executable.
The ***Python Program*** should create 2 worker threads, both should be able to access an integer array of length 20, the Worker thread#1 multiplies each element of that array by -1 and the Worker thread #2 adds 3 to each element of that array. Implement the concept of Context switching such that after altering every five elements of the array, thread's execution should switch from one thread to another thread.

#Here is a sample shell script that takes a file name as a command line argument and makes it executable if it is not already:

```
#!/bin/bash

file=$1

if [ ! -x "$file" ]; then
  chmod +x $file
  echo "$file is now executable"
else
  echo "$file is already executable"
fi
```

#For the second part of the task, you can use the Python threading library to create worker threads. Here's a sample Python code to implement the worker threads:

```
import threading
import time

def worker_1(array):
  for i in range(0, 20, 5):
    for j in range(i, min(i + 5, 20)):
      array[j] = -array[j]
    time.sleep(0.5)

def worker_2(array):
  for i in range(0, 20, 5):
    for j in range(i, min(i + 5, 20)):
      array[j] = array[j] + 3
    time.sleep(0.5)

if _name_ == '_main_':
  array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
  t1 = threading.Thread(target=worker_1, args=(array,))
  t2 = threading.Thread(target=worker_2, args=(array,))

  t1.start()
  t2.start()

  t1.join()
  t2.join()

  print(array)
```

#In this code, the worker_1 and worker_2 functions are executed in separate threads. The worker_1 multiplies each element of the array by -1 and the worker_2 adds 3 to each element of the array. After altering every five elements of the array, the execution of the thread switches to the other thread by using time.sleep(0.5). The t1.join() and t2.join() calls block until both threads are complete. Finally, the updated array is printed

1) To make script, create one main process in which one process initialize the list and the second process add the items of that list and the main process is printing the sum using pipe().

```
#!/bin/bash

pipe_file=/tmp/pipe_file

# Create a named pipe
if [ ! -p "$pipe_file" ]; then
  mkfifo "$pipe_file"
fi

# Main process
{
  # First child process
  {
    # Initialize the list
    list=(1 2 3 4 5)

    # Write the list to the pipe
    printf "%s\n" "${list[@]}" > "$pipe_file"
  } &

  # Second child process
  {
    # Read the list from the pipe
    list=()
    while read -r item; do
      list+=( "$item" )
    done < "$pipe_file"

    # Calculate the sum of the list
    sum=0
    for item in "${list[@]}"; do
      sum=$(( sum + item ))
    done
    # Write the sum to the pipe
    printf "%s\n" "$sum" > "$pipe_file"
  } &

  # Read the sum from the pipe
  read -r sum < "$pipe_file"

  # Print the sum
  echo "Sum is: $sum"
}

# Clean up the named pipe
rm "$pipe_file"
```

2) To create the need matric of bankers algorithm using threading.

```
import threading
import numpy as np

class BankersAlgorithm(threading.Thread):
    def _init_(self, process_id, max_matrix, allocation_matrix, need_matrix,
available_resources):
        threading.Thread._init_(self)
        self.process_id = process_id
        self.max_matrix = max_matrix
        self.allocation_matrix = allocation_matrix
        self.need_matrix = need_matrix
        self.available_resources = available_resources

    def run(self):
        # Check if the request can be granted
        if np.all(self.need_matrix[self.process_id] <= self.available_resources):
            self.available_resources = self.available_resources +
self.allocation_matrix[self.process_id]
            print(f"Process {self.process_id} has been granted resources")
        else:
            print(f"Process {self.process_id} has NOT been granted resources")

# Sample input data
num_processes = 5
num_resources = 3

# Available resources
```

```
available_resources = np.array([3, 3, 2])

# Maximum resources required by each process
max_matrix = np.array([[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]])

# Currently allocated resources
allocation_matrix = np.array([[0, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]])

# Calculate the need matrix
need_matrix = max_matrix - allocation_matrix

# Create the threads
threads = []
for i in range(num_processes):
    threads.append(BankersAlgorithm(i, max_matrix, allocation_matrix, need_matrix,
available_resources))

# Start the threads
for t in threads:
    t.start()

# Wait for all threads to finish
for t in threads:
    t.join()
```