Advanced Process Manager with Process Synchronization

This project implements a process manager, with an emphasis on process synchronization. The process manager allows users to create, manage, synchronize, and interact with processes in a multi-threaded environment. It provides a command-line interface and uses system calls for process and thread control.

Functionalities of the Process Manager

The process manager has the following functionalities:

1. Process creation – the user can choose to create a new child process, in which case the program will call fork() system call.
2. Process management – the user can tell the program to:
    a. List all running processes, displaying their process ID, process name, parent process ID, and state.
    b. Terminate a running process by entering its process ID.
3. Threading – the user can tell the program to:
    a. Create a new thread.
    b. Terminate a newly created thread.
    c. Synchronize multiple threads - display a demonstration of thread synchronization.
4. Inter-Process Communication (IPC) – the program implements two types of IPC:
    a. IPC via pipe – allowing users to see how two processes communicate with each other via pipe. The user can enter a message that will be sent by a sender process and received by a receiver process.
    b. IPC via shared memory – the program employs a shared array of size 5, and the user can make a process write to the shared memory, and a process read from the shared memory.
5. Process Synchronization – the program provides demonstrations of:
    a. Mutex – the users can increment or decrement a shared value using a mutex.
    b. Semaphore – the users can increment or decrement a shared value using a semaphore.
    c. The use of synchronization mechanisms to solve the Producer-Consumer Problem.
    d. The use of synchronization mechanisms to solve the Reader-Writer Problem.
6. Command-line interface (CLI) allows users to interact with the process manager and choose what to do using a clear and informative syntax and options.

7. Logging and reporting – the process manager implements a logging mechanism and logs all significant events to the log file. To help the user track the execution of processes and threads, the program also displays every important detail to the user using CLI.

Running the Program

All of the libraries used in the program are included in the Python standard library, except one – psutil. To install psutil, run the following command in the terminal:

```
pip install psutil
```

To run the program, download the source file, which includes all of the source code of the project, including the log file, and run main.py by using the Python interpreter.

Functionality Discussion and Explanation

1. Command-Line Interface

   The process manager uses a command-line interface to interact with the user. The user can choose from the following options:

   1. Create a new process
   2. List running processes
   3. Terminate a process
   4. Create a thread
   5. Terminate a thread
   6. Synchronize threads
   7. Inter-Process Communication (IPC) using pipe
   8. Inter-Process Communication (IPC) using shared memory
   9. Process synchronization – mutex and semaphore
   10. Producer-consumer problem
   11. Reader-writer problem
   12. Exit

   The user can choose any of the options by entering the corresponding number of the choice. This is what the CLI looks like:

```
Options:
1. Create a new process
2. List running processes
3. Terminate a process
4. Create a thread
5. Terminate a thread
6. Synchronize threads
7. Inter-Process Communication (IPC) using pipe
8. Inter-Process Communication (IPC) using shared memory
9. Process Synchronization - mutex and semaphore
10. Producer-consumer problem
11. Reader-writer problem
12. Exit

Enter your choice:
```

2. Process Creation

   If the user chooses the first option – to create a new process, the process creation method is called, which is in process_and_thread_management.py file, called create_process(). It uses os.fork() call to fork a child process, and if fork is successful the program displays the message saying that process creation was successful, along with the process' PID, as well as its parent's PID.

   This is what the output looks like when the user chooses option #1:

```
Enter your choice: 1
Child process created successfully. PID: 0, Parent PID: 14628
```

3. List Running Processes

   If the user chooses to list running processes, the list_processes() method is called from process_and_thread_management.py file. The method uses psutil library to list information about all currently running processes, which includes their PID, name, parent PID, and state (e.g. running).

This is a snippet of what the output might look like – its details are going to vary across different devices at different times:

```
Enter your choice: 2
PID: 0, Name: kernel_task, Parent PID: 0, State: running
PID: 1, Name: launchd, Parent PID: 0, State: running
PID: 327, Name: logd, Parent PID: 1, State: running
PID: 328, Name: smd, Parent PID: 1, State: running
PID: 329, Name: UserEventAgent, Parent PID: 1, State: running
PID: 331, Name: uninstalld, Parent PID: 1, State: running
PID: 332, Name: fseventsd, Parent PID: 1, State: running
PID: 333, Name: mediaremoted, Parent PID: 1, State: running
PID: 336, Name: systemstats, Parent PID: 1, State: running
PID: 339, Name: configd, Parent PID: 1, State: running
PID: 340, Name: endpointsecurity, Parent PID: 1, State: running
PID: 341, Name: powerd, Parent PID: 1, State: running
PID: 342, Name: IOMFB_bics_daemo, Parent PID: 1, State: running
PID: 343, Name: biomed, Parent PID: 1, State: running
PID: 345, Name: amfid, Parent PID: 1, State: running
```

4. Terminate a Process

   If the user wishes to terminate a process, they will first be asked to enter a PID of the target process (process about to be terminated). Depending on the input, either the program is going to terminate the process by calling terminate_process() function from process_and_thread_management.py, or in case of an error, it's going to display that message as well.

   If the process can be terminated, the program will do so by calling terminate() method from the psutil library. If there's no such process with the PID entered, the program is going to display a message saying so, and if the access to terminate the process has been denied, the program will display that, as well.

   Program's output in the case of a successful termination:

```
Enter your choice: 3
Enter PID: 14640
Process with PID 14640 has been terminated.
```

Program's output if the process with entered PID does not exist:

```
Enter your choice: 3
Enter PID: 1231231
Process with PID 1231231 does not exist.
```

Program's output if the access has been denied:

```
Enter your choice: 3
Enter PID: 1
You don't have permission to terminate the process with PID 1.
```

5. Thread Creation

   If the user chooses to create a thread, create_thread() method is called from process_and_thread_management.py. The method creates a thread using the threading library (threading.Thread()), gives it some function that simulates work, and starts the thread. The name of the thread is printed out to the console:

```
Enter your choice: 4
Thread Thread-1 (worker) is working...
Thread Thread-1 (worker) is done.
Thread Thread-1 (worker) created.
```

6. Thread Termination

   If the user chooses to terminate a thread, first they have to choose which thread to terminate. All the threads created by the user beforehand will be listed, and the user will have to choose the corresponding number. After choosing the thread, the program calls terminate_thread() function from process_and_thread_management.py, which

uses join() method to terminate the thread. This is what the output looks like if we have 4 threads:

```
Enter your choice: 5
Which Thread would you like to terminate?
1. Thread-1 (worker)
2. Thread-2 (worker)
3. Thread-3 (worker)
4. Thread-4 (worker)

Enter your choice: 2
Thread Thread-2 (worker) terminated successfuly.
```

7. Thread Synchronization

   If the user chooses the sixth option – synchronize threads, the process manager is going to display how multiple (in this case 5) threads can be synchronized using a semaphore. The program calls synchronize_threads() function from process_and_thread_management.py, which also employs methods from the threading library. First, we define a semaphore with the initial value of 1; then 5 threads are initialized, each given a task, and they complete the task by acquiring semaphore, doing the work, and releasing it one by one. After all threads are done, the semaphore is released. This is what the output looks like:

```
Synchronization of 5 threads using a Semaphore:


Thread 0 is waiting for the semaphore.

Thread 1 is waiting for the semaphore.

Thread 2 is waiting for the semaphore.

Thread 3 is waiting for the semaphore.

Thread 4 is waiting for the semaphore.

Thread 0 has acquired the semaphore.

Thread 0 has released the semaphore.
Thread 1 has acquired the semaphore.


Thread 1 has released the semaphore.
Thread 2 has acquired the semaphore.


Thread 2 has released the semaphore.
Thread 3 has acquired the semaphore.


Thread 3 has released the semaphore.
Thread 4 has acquired the semaphore.


Thread 4 has released the semaphore.
All threads have finished.
```

8. Inter-Process Communication (IPC) Using Pipe

If the user chooses to demonstrate inter-process communication using a pipe, the program calls IPC_pipe() function from IPC.py. The function employs two types of process – sender and receiver of a message – for demonstration purposes and creates those processes using the multiprocessing library. The pipe between them is also created using the same library by calling multiprocessing.Pipe().

The user is given 3 choices – to send a message, to receive a message, and to quit the IPC system. If the user wants to send a message, the program is going to take input of the user and assign it to the message variable, and the sender process is going to put the message in the pipe. If the user chooses to receive a message, the receiver process is going to get the oldest message from the pipe and display it to the console. The quit option returns to the main CLI.

This is what the interaction with the IPC pipe system looks like:

```
Enter your choice: 7

Choose an option:
1. Send a message
2. Receive a message
3. Quit
1
Enter a message to send: Hello
Sender process sent a message: Hello


Choose an option:
1. Send a message
2. Receive a message
3. Quit
1
Enter a message to send: second message
Sender process sent a message: second message


Choose an option:
1. Send a message
2. Receive a message
3. Quit
2
Receiver process received a message: Hello
```

9. Inter-Process Communication (IPC) Using Shared Memory

   If the user chooses to demonstrate inter-process communication using shared memory, IPC_shared_memory() method from IPC.py will be called. Similar to the previous method, this one also creates 2 processes, but in this case, one is a writer process,

which writes to the shared memory, and the second one reads from the shared memory.

For simple demonstration purposes, shared memory in this case is an array of size 5 which can store integers. When the user chooses to write to the shared memory, a writer process is initialized, and the user is able to choose where to write the information, and what information to write (can choose the integer value). If the user chooses to read from the shared memory, they will be asked to enter the value of index they want to read from, and the program is going to initialize a reader process, which is going to read from the shared memory. This is what the interaction with this type of IPC system looks like:

```
Enter your choice: 8
Shared memory is an array of size 5 storing integers.

Choose an option:
1. Write to the shared memory
2. Read from the shared memory
3. Quit
1
Enter the index to write to: 0
Enter the value to write: 345

Choose an option:
1. Write to the shared memory
2. Read from the shared memory
3. Quit
2
Enter the index to read from: 0
Read from shared memory: 345

Choose an option:
1. Write to the shared memory
2. Read from the shared memory
3. Quit
3
```

10. Process Synchronization – Demonstration of Mutex and Semaphore

    If the user chooses the ninth option, the program is going to demonstrate how mute and semaphore work by creating a shared value using process_synchronization_demo() function in mutex_and_semaphore_demo.py. The program lets the user choose to increment or decrement the shared value using a mutex, or increment or decrement the shared value using a semaphore.

Once the user chooses what to do, the program creates a process using the multiprocessing library, and gives the task according to the user's input.

This is what the output might look like when interacting with this demo:

```
Enter your choice: 9
Initial value of shared value: 0


Choose an option:
1. Increment with mutex
2. Decrement with mutex
3. Increment with semaphore
4. Decrement with semaphore
5. Quit
1
Mutex lock acquired.
Shared value incremented with mutex: 1
Mutex lock released.

Choose an option:
1. Increment with mutex
2. Decrement with mutex
3. Increment with semaphore
4. Decrement with semaphore
5. Quit
4
Semaphore acquired.
Shared value decremented with semaphore: 0
Semaphore released.
```

11. Producer-Consumer Problem

If the user chooses the 10th option, the program is going to call producer_consumer_problem() from producer_consumer_problem.py. This function demonstrates the producer-consumer problem with 2 producers and 2 consumers. We have a buffer of fixed size; a producer thread can produce an item and place it in the buffer, and a consumer can pick items from the buffer and consume them. The main goal is to ensure that when a producer is placing an item in the buffer, consumers should not consume any item at the same time.

To achieve that goal, we create a mutex for buffer access, a semaphore for empty slots, and a semaphore for filled slots.

When a producer thread is tasked with producing an item, first it waits for an empty slot, and gets exclusive access to the buffer using the mutex. After that, it adds the produced item to the buffer, releases the mutex, and notifies that a slot is filled using the second semaphore.

When a consumer thread is tasked with consuming an item, first it waits for a filled slot, then gets exclusive access to the buffer, consumes the first item in the buffer, releases the mutex, and notifies that a slot is empty using the first semaphore called empty. The program displays how buffer changes on each iteration.

This is what the output looks like:

```
Enter your choice: 10
Producer-consumer problem with 2 producers and 2 consumers.

We have a buffer of fixed size. A producer can produce an item and can place in the buffer.
A consumer can pick items and can consume them. We need to ensure that when a producer is
placing an item in the buffer, then at the same time consumer should not consume any item.

Produced 29. Buffer: [29]
Produced 72. Buffer: [29, 72]
Consumed 29. Buffer: [72]
Consumed 72. Buffer: []
Produced 1. Buffer: [1]
Consumed 1. Buffer: []
Produced 29. Buffer: [29]
Consumed 29. Buffer: []
Produced 60. Buffer: [60]
Consumed 60. Buffer: []
Produced 68. Buffer: [68]
Consumed 68. Buffer: []
Produced 32. Buffer: [32]
Consumed 32. Buffer: []
Produced 35. Buffer: [35]
Produced 61. Buffer: [35, 61]
Consumed 35. Buffer: [61]
Produced 40. Buffer: [61, 40]
Consumed 61. Buffer: [40]
Consumed 40. Buffer: []
Produced 55. Buffer: [55]
Consumed 55. Buffer: []
Produced 68. Buffer: [68]
Produced 31. Buffer: [68, 31]
```

```
Consumed 68. Buffer: [31]
Produced 88. Buffer: [31, 88]
Consumed 31. Buffer: [88]
Produced 9. Buffer: [88, 9]
Consumed 88. Buffer: [9]
Produced 87. Buffer: [9, 87]
Consumed 9. Buffer: [87]
Consumed 87. Buffer: []
Produced 82. Buffer: [82]
Consumed 82. Buffer: []
Produced 61. Buffer: [61]
Produced 48. Buffer: [61, 48]
Consumed 61. Buffer: [48]
Produced 51. Buffer: [48, 51]
Consumed 48. Buffer: [51]
Consumed 51. Buffer: []
```

12. Reader-Writer Problem

    If the user chooses the 11th option, the program calls reader_writer_problem() method from reader_writer_problem.py. This particular implementation of the problem involves 2 readers and 2 writers, and an object, which is shared between all four of the processes. Reader processes only want to read the data from the object and writer processes only want to write into the object. By using two semaphores, read_semaphore and write_semaphore, we ensure that the processes have exclusive access to the shared object.

    This is what the output looks like:

```
Enter your choice: 11
Reader-writer problem with 2 readers and 2 writers.

The reader-writer problem relates to an object such as a file that is
shared between multiple processes. Some of these processes are readers
i.e. they only want to read the data from the object and some of the
processes are writers i.e. they want to write into the object.

Reader 1 is reading.
Writing data...
Reader 1 is reading.
Writing data...
Reader 1 is reading.Reader 2 is reading.

Writing data...
Writing data...
Reader 1 is reading.Reader 2 is reading.

Writing data...
Writing data...
Reader 1 is reading.
Reader 1 is reading.
Writing data...
Writing data...
Reader 1 is reading.
Writing data...
Writing data...
Reader 1 is reading.
Reader 1 is reading.
```

```
Writing data...
Writing data...
Reader 1 is reading.
Reader 1 is reading.
Writing data...
Writing data...
Reader 1 is reading.
Reader 1 is reading.
Writing data...
Writing data...
Reader 1 is reading.
Reader 1 is reading.
Writing data...
Writing data...
Reader 1 is reading.
Reader 1 is reading.
Writing data...
Writing data...
Reader 1 is reading.
```

13. Logging

    In the project files, there is one called logger_config.py, which uses Python's logging library to create a logger and log all significant events, warnings, and errors to the log file called log_file.log. The logger records the time of the event, the name of the logger, the level of the logger, and the message.

    This is a snippet of what log_file.log could look like – it all depends on what the user's choices are in the command-line interface:

```
2023-10-28 17:17:48,661 - logger - INFO - Processes have been listed.
2023-10-28 17:22:53,243 - logger - INFO - Process with PID 14640 has been terminated.
2023-10-28 17:23:36,908 - logger - WARNING - Process termination - process with PID 1231231 does not exist.
2023-10-28 17:23:52,518 - logger - WARNING - Process termination - Access Denied.
2023-10-28 17:56:03,859 - logger - INFO - Thread Thread-1 (worker) is working...
2023-10-28 17:56:03,859 - logger - INFO - Thread Thread-1 (worker) created.
2023-10-28 17:56:03,860 - logger - INFO - Thread Thread-1 (worker) is done.
2023-10-28 17:56:15,573 - logger - INFO - Thread Thread-2 (worker) is working...
2023-10-28 17:56:15,573 - logger - INFO - Thread Thread-2 (worker) created.
2023-10-28 17:56:15,573 - logger - INFO - Thread Thread-2 (worker) is done.
2023-10-28 17:56:19,612 - logger - INFO - Thread Thread-3 (worker) is working...
2023-10-28 17:56:19,613 - logger - INFO - Thread Thread-3 (worker) created.
2023-10-28 17:56:19,613 - logger - INFO - Thread Thread-3 (worker) is done.
2023-10-28 17:56:20,309 - logger - INFO - Thread Thread-4 (worker) is working...
2023-10-28 17:56:20,309 - logger - INFO - Thread Thread-4 (worker) is done.
2023-10-28 17:56:20,309 - logger - INFO - Thread Thread-4 (worker) created.
2023-10-28 17:58:42,119 - logger - INFO - Thread Thread-2 (worker) terminated successfuly.
2023-10-28 18:02:15,658 - logger - INFO - Thread 0 is waiting for the semaphore.
2023-10-28 18:02:15,658 - logger - INFO - Thread 1 is waiting for the semaphore.
2023-10-28 18:02:15,659 - logger - INFO - Thread 2 is waiting for the semaphore.
2023-10-28 18:02:15,659 - logger - INFO - Thread 3 is waiting for the semaphore.
2023-10-28 18:02:15,659 - logger - INFO - Thread 4 is waiting for the semaphore.
2023-10-28 18:02:15,659 - logger - INFO - Thread 0 has acquired the semaphore.
2023-10-28 18:02:17,665 - logger - INFO - Thread 0 has released the semaphore.
2023-10-28 18:02:17,665 - logger - INFO - Thread 1 has acquired the semaphore.
2023-10-28 18:02:19,670 - logger - INFO - Thread 2 has acquired the semaphore.
2023-10-28 18:02:19,671 - logger - INFO - Thread 1 has released the semaphore.
2023-10-28 18:02:21,676 - logger - INFO - Thread 3 has acquired the semaphore.
2023-10-28 18:02:21,677 - logger - INFO - Thread 2 has released the semaphore.
2023-10-28 18:02:23,682 - logger - INFO - Thread 4 has acquired the semaphore.
2023-10-28 18:02:23,683 - logger - INFO - Thread 3 has released the semaphore.
2023-10-28 18:02:25,686 - logger - INFO - Thread 4 has released the semaphore.
2023-10-28 18:02:25,687 - logger - INFO - Thread synchronization - all threads have finished.
```

Discussion of the Results

The project aimed to design and implement an advanced Process Manager with a strong emphasis on process synchronization. The main purpose was to create something that would allow users to interact with processes and threads and explore different functionalities of process synchronization in a multithreaded environment.

By interacting with this program, the user can learn a lot about how things work – how to harness system calls to create, manage, and synchronize processes and threads effectively. Users can better understand the importance of managing system resources efficiently to minimize conflicts after their time with the project. The addition of a user-friendly command-line interface simplifies interactions with the process manager and greatly enhances its usability.

Also, the inclusion of logging and reporting features directly to the user enables us to effectively track the execution of processes and threads. This not only provides a comprehensive view of the system's behavior but also can be very useful if debugging or error handling is needed.

Personally, the project expanded my technical skills in process and thread management. The user-friendly interface, comprehensive logging, and synchronization mechanisms make the process manager a valuable tool for learning more about multithreaded environments.

References

1. https://gyaanibuddy.com/assignments/assignment-detail/producer-consumerreader-writer-using-semaphore/
2. https://docs.python.org/3/howto/logging.html
3. https://docs.python.org/3/library/threading.html
4. https://docs.python.org/3/library/multiprocessing.html
5. https://www.datadoghq.com/blog/python-logging-best-practices/
6. https://www.geeksforgeeks.org/python-os-fork-method/