Explanation:

After carefully reading and identifying the constraints the problem puts forth one probable and efficient solution comes to my mind viz. **Shared Memory Architecture.** At Operating System level there are a lot of processes trying to exchange information with each other in order to provide some functionality. This can be either done through a mechanism called IPC(Interprocess Communication) but in scenarios where there is requirement of no message passing it is a good idea to have something called shared memory between different processes.

Use case:For this question we consider an Application for ordering a list of items for an organization at a local supplier chain

- Our Application-1 is the GUI built using python framework called Tkinter which asks the user to select products, quantity, brand and category

- Once the GUI gets all the required fields of input . It creates a requirements module for that order. Based on the requirements it creates a set of tasks to be completed by Process 2

- Based on this we define our custom made tasks in Application 2 (different stages of processing)

- Application-1 then creates a shared memory backed by a file on local disk using mmap - a builtin python library for multiprocessing.

- Then finally it initializes Application-2 using a python module named subprocess with list of tasks(defaults completed =”False”) as parameters to it

- It then constantly polls the shared memory to see the progress of Application 2

Logic Behind Application 2:

- Once Application 2 receives the tasks list to be executed . It executes the task in order and updates the corresponding task value as completed in shared memory . Use of locks ensures synchronization and thread safety

- Once all the tasks are completed it returns the control back to Application 1 and Application 1 verifies it.

Other Alternatives:

With shared memory there always comes the idea of message queues(**one way pipe**). The basic idea is to enqueue(add) task from one end and dequeue(remove) from the other end. But disadvantages of such being the additional overhead of defining the message size and queue length by the process may lead to significant performance loss but also ensures complete or no delivery of message in contrast to partial delivery. Also the time to read/write to from queue may result in longer response times for the Application

Shared Memory Architecture Justification:

With the above mentioned drawbacks of message queues . For a scenario where two processes cannot communicate directly , shared memory comes in as a winner. Given that synchronization constructs can be created at developer side and using the fact that we have full freedom of how we develop both the applications(App-1 and App2) will ensure thread safety. Shared memory can be deemed as faster (low overhead, high volume of data passing) than message queues. But queues on the other hand, requires high overhead (the set up for making a queue to be permanent etc) with low volume of data.