Python proporciona capacidades nativas para concurrencia a través del módulo `concurrent.futures`, el módulo `asyncio` para programación asíncrona, y el módulo `threading` para subprocesos. Además, hay bibliotecas de terceros como `multiprocessing` para trabajar con procesos paralelos.

1. Creación y manejo de tareas concurrentes:
   - concurrent.futures: Este módulo proporciona la clase `ThreadPoolExecutor` para la ejecución de funciones en hilos y `ProcessPoolExecutor` para la ejecución en procesos separados. Permite la creación de tareas concurrentes mediante la interfaz de `submit`, que acepta una función y devuelve un objeto `Future` que representa el resultado futuro de la función.

   - asyncio: Para la programación asíncrona, `asyncio` utiliza las palabras clave `async` y `await` para definir funciones asíncronas y esperar su ejecución. La creación y manejo de tareas concurrentes se realiza utilizando la interfaz de `asyncio.create_task()`.

   - threading y multiprocessing: Estos módulos permiten la creación de subprocesos (`Thread`) y procesos (`Process`) respectivamente, para realizar tareas concurrentes. Sin embargo, se debe tener en cuenta que debido al GIL (Global Interpreter Lock) en CPython, el subprocesamiento puede no ser eficiente para ciertos tipos de trabajo intensivo en CPU.

2. Control de memoria compartida y/o pasaje de mensajes:
   - multiprocessing: Proporciona mecanismos para compartir memoria entre procesos a través de objetos como `Value` y `Array`. También utiliza el concepto de tuberías y colas para el intercambio de datos entre procesos.

   - threading: Los subprocesos comparten la memoria global del intérprete de Python, lo que facilita el intercambio de datos. Sin embargo, se deben utilizar mecanismos de sincronización, como `Lock` o `Queue`, para evitar condiciones de carrera.

   - asyncio: La programación asíncrona utiliza corutinas y no comparte directamente la memoria. La comunicación entre tareas se realiza a menudo a través de canales de eventos o colas.

3. Mecanismo de sincronización en Python:
   - threading: Utiliza objetos como `Lock`, `Semaphore`, y `Event` para sincronizar el acceso a recursos compartidos y prevenir condiciones de carrera en entornos de subprocesos.

   - multiprocessing: Ofrece mecanismos similares a `threading` para sincronización, como `Lock` y `Event`, pero adaptados para entornos de procesos.

   - asyncio: Utiliza corutinas y `awaitables` para lograr la sincronización sin bloqueo en el contexto de la programación asíncrona. Además, proporciona objetos como `asyncio.Lock` para gestionar el acceso concurrente a recursos compartidos en entornos asíncronos.

En resumen, Python ofrece varias opciones para la concurrencia, cada una con sus propias características y casos de uso recomendados. La elección de la mejor opción depende de los requisitos específicos de la aplicación.