- General Multiprocessing: Storage pointer swapping, Refcounting, Cleanup Daemon
- Python Multiprocessing: Custom Picker and subclassing python.multiprocessing

General Multiprocessing: Storage pointer swapping, Refcounting, Cleanup Daemon

Multiprocessing involves sharing memory among processes.

On Unix systems, one can do this with - mmap - shm_open - shm_unlink - ftruncate

Sharing Storages among Process `A`, `B`

Creating Shared Storages

Usually this is the way one shares memory among processes.

Let's say process `A` wants to share memory `[M, M+D]`, i.e. starting at `M` with size `D`.
`A` wants to share `M` with process `B`.

1. Process `A` calls `shm_open`, which will create a shared memory object and map it to a file under the folder `/dev/shm/`. Let's say the file is `/dev/shm/torch_shmfile_1` . `shm_open` will return a file descriptor to this file.
2. Then, process `A` truncates the shared memory object to the size `D`, using `ftruncate`
3. Then, process `A` calls `mmap` on this file descriptor, to get a pointer to this memory. Let's call this pointer `SHM`
4. Then, process `A` copies over memory `[M, M+D]` to `[SHM, SHM+D]`.
5. Then, process `A` swaps all pointer references of `M` to be `SHM`. There is no straight forward way to do this in general.

For (5), Torch has a Tensor / Storage abstraction. We can simply swap the data pointer inside the Storage and all Tensors referring to this Storage do not notice anything changing.

Now, `A` communicates the file path of the shared memory along with the size to `B`, i.e. `("/dev/shm/torch_shmfile_1", D)`.

`B` will also `shm_open` the same file, calls `mmap` on the file descriptor and can map the same pointer `SHM` to a new Storage.

Now, `A` and `B` have succesfully shared memory. a storage in `A` points to the same memory as a storage in `B`.

Note: Shared Storages are not resizeable.

Freeing Shared Storages, need for reference counting

The next question then comes - how do we free this memory.

For this, one has to call the function `shm_unlink` on the file descriptor of `SHM`. Calling `shm_unlink` on the memory will remove the file `/dev/shm/torch_shmfile_1` , but keeps existing file descriptors and the memory pointer alive.
Let's say that process `A` exits, and as part of exiting, the process deallocates all of it's Storage objects and calls `shm_unlink` on `SHM`. Process `B` still can access and use `SHM`, but the file that pointed to `SHM`, i.e. `/dev/shm/torch_shmfile_1` will be removed from the filesystem.

So, the problem in this scenario then becomes this: If process `A` shares a Tensor with `B`, then exits. Then process `B` cannot further share the Tensor to a process `C`. This is a problem for us. One can often think of a scenario where `A` creates a process pool (of which `B` is part of). `B` creates a Tensor, and shares it with `A`. Then `A` terminates the process pool, and creates another pool, with which it tries to share the same Tensor, and this does not work as `B` has called `shm_unlink` on the Tensor.

To resolve this, we now have to add **reference counting** to this `SHM` object, and only call `shm_unlink` on this object once the references to this object reach 0.

So, what we hence do is that if you want to allocate `SHM` of size `D`, we will allocate `SHM` of size `D + 4`, and use the last 4 bytes as an integer for reference counting. To make sure that two processes changing this refcount integer at the same time does not have conflicts, we use hardware atomic instructions to increment or decrement this counter properly (they are available in all processors that we care about).

So far, we have covered: - Creating the shared memory - Swapping the memory pointers in Storage - Refcounting the shared memory so that we only call shm_unlink once

Handling cleanup when process dies abruptly with `SIGKILL`

There is another problem that we have not touched upon, which is: what happens to this shared memory if the process gets a `SIGKILL`.

`A` lot of users often call the command `killall [processname]` or `kill -9 [processname]`. It is the only command they know to kill a process, and they use it all the time :)
This sends a `SIGKILL` signal to the process. When a process gets a `SIGKILL` as opposed to a `SIGINT`, it is not given a chance to cleanup after itself.
This is a problem because, if we do not call `shm_unlink` on the shared memory `SHM`, it will remain occupied until you restart your computer of manually run the command: `rm -f /dev/shm/torch_shmfile_1` So, if the Tensor is of 8GB

memory, then we essentially have leaked this 8GB of memory to never be used by any process again until the system restarts. This is horrible.

Hence, we have a new problem to solve, which is: how do we ensure that we cleanup safely, even if processes `A`, `B`, `C` are given a `SIGKILL` and die abruptly.

There are two solutions to this:

1. Remove the file `/dev/shm/torch_shmfile_1` as soon as we create it using `unlink()` (not `shm_unlink`), and simply exchange file descriptors among processes. This way, if we have `SIGKILL`, on the processes, there is nothing left to cleanup.
2. Launch a daemon process and unlink it from it's parent process. This daemon process stays alive even when it's parent process dies, and when parent `A` gets a `SIGKILL`, it detects that `A` has died, and will cleanup afterwards. So, whenever we share a Tensor, we have to give the path of the `SHM` file to the daemon process (for example `/dev/shm/torch_shmfile_1`).

###Solution 1 This is an elegant solution, but it suffers from the problem that, each shared Storage has to have one open file descriptor.

So, if we share 4000 Storages between `A` and `B`, then there are 4000 open file descriptors. This is usually not a problem in modern Operating Systems, but quite a few academic clusters limit the number of file descriptors per process, sometimes to as little as 1024.

So, keeping in mind users and support, we decided that this will not be a fully practical solution.

###Solution 2 This solution also seems to be robust to processes `A`,`B`,`C` getting a `SIGKILL`.
Since the daemon removed `A` as it's parent process, it does not die immediately when `A` dies.
As soon as the daemon process is launched, it opens a socket connection with process `A`.
When the socket connection dies, the daemon knows that `A` has died abruptly (possibly by a `SIGKILL`) and cleans up afterwards.

After considering both the solutions thoroughly, we have implemented both Solution 1 and Solution 2 to solve this `SIGKILL` / cleanup problem, and they can be switched at runtime. By default, you are set to Solution 1