Python Multiprocessing

Debugging

Please see the [Troubleshooting](#troubleshooting-python-multiprocessing) page for information on known issues and how to solve them.

Introduction

:::{important}
The source code references are to the state of the code at the time of writing in December, 2024.
:::

The use of Python multiprocessing in vLLM is complicated by:

- The use of vLLM as a library and the inability to control the code using vLLM
- Varying levels of incompatibilities between multiprocessing methods and vLLM dependencies

This document describes how vLLM deals with these challenges.

Multiprocessing Methods

[Python multiprocessing methods](https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods) include:

- `spawn` - spawn a new Python process. This will be the default as of Python 3.14.

- `fork` - Use `os.fork()` to fork the Python interpreter. This is the default in Python versions prior to 3.14.

- `forkserver` - Spawn a server process that will fork a new process on request.

Tradeoffs

`fork` is the fastest method, but is incompatible with dependencies that use threads.

`spawn` is more compatible with dependencies, but can be problematic when vLLM is used as a library. If the consuming code does not use a `__main__` guard (`if __name__ == "__main__":`), the code will be inadvertently re-executed when vLLM spawns a new process. This can lead to infinite recursion, among other problems.

`forkserver` will spawn a new server process that will fork new processes on demand. This unfortunately has the same problem as `spawn` when vLLM is used as a library. The server process is created as a spawned new process, which will re-execute code not protected by a `__main__` guard.

For both `spawn` and `forkserver`, the process must not depend on inheriting any global state as would be the case with `fork`.

Compatibility with Dependencies

Multiple vLLM dependencies indicate either a preference or requirement for using `spawn`:

- <https://pytorch.org/docs/stable/notes/multiprocessing.html#cuda-in-multiprocessing>
- <https://pytorch.org/docs/stable/multiprocessing.html#sharing-cuda-tensors>
- <https://docs.habana.ai/en/latest/PyTorch/Getting_Started_with_PyTorch_and_Gaudi/Getting_Started_with_PyTorch.html?highlight=multiprocessing#torch-multiprocessing-for-dataloaders>

It is perhaps more accurate to say that there are known problems with using `fork` after initializing these dependencies.

Current State (v0)

The environment variable `VLLM_WORKER_MULTIPROC_METHOD` can be used to control which method is used by vLLM. The current default is `fork`.

- <https://github.com/vllm-project/vllm/blob/d05f88679bedd73939251a17c3d785a354b2946c/vllm/envs.py#L339-L342>

When we know we own the process because the `vllm` command was used, we use `spawn` because it's the most widely compatible.

- <https://github.com/vllm-project/vllm/blob/d05f88679bedd73939251a17c3d785a354b2946c/vllm/scri

pts.py#L123-L140>

The `multiproc_xpu_executor` forces the use of `spawn`.

- <https://github.com/vllm-project/vllm/blob/d05f88679bedd73939251a17c3d785a354b2946c/vllm/executor/multiproc_xpu_executor.py#L14-L18>

There are other miscellaneous places hard-coding the use of `spawn`:

- <https://github.com/vllm-project/vllm/blob/d05f88679bedd73939251a17c3d785a354b2946c/vllm/distributed/device_communicators/custom_all_reduce_utils.py#L135>
- <https://github.com/vllm-project/vllm/blob/d05f88679bedd73939251a17c3d785a354b2946c/vllm/entrypoints/openai/api_server.py#L184>

Related PRs:

- <gh-pr:8823>

Prior State in v1

There was an environment variable to control whether multiprocessing is used in the v1 engine core, `VLLM_ENABLE_V1_MULTIPROCESSING`. This defaulted to off.

-

<https://github.com/vllm-project/vllm/blob/d05f88679bedd73939251a17c3d785a354b2946c/vllm/envs.py#L452-L454>

When it was enabled, the v1 `LLMEngine` would create a new process to run the engine core.

-
<https://github.com/vllm-project/vllm/blob/d05f88679bedd73939251a17c3d785a354b2946c/vllm/v1/engine/llm_engine.py#L93-L95>
-
<https://github.com/vllm-project/vllm/blob/d05f88679bedd73939251a17c3d785a354b2946c/vllm/v1/engine/llm_engine.py#L70-L77>
-
<https://github.com/vllm-project/vllm/blob/d05f88679bedd73939251a17c3d785a354b2946c/vllm/v1/engine/core_client.py#L44-L45>

It was off by default for all the reasons mentioned above - compatibility with dependencies and code using vLLM as a library.

Changes Made in v1

There is not an easy solution with Python's `multiprocessing` that will work everywhere. As a first step, we can get v1 into a state where it does "best effort" choice of multiprocessing method to maximize compatibility.

- Default to `fork`.
- Use `spawn` when we know we control the main process (`vllm` was executed).

- If we detect `cuda` was previously initialized, force `spawn` and emit a
  warning. We know `fork` will break, so this is the best we can do.

The case that is known to still break in this scenario is code using vLLM as a library that initializes `cuda` before calling vLLM. The warning we emit should instruct users to either add a `__main__` guard or to disable multiprocessing.

If that known-failure case occurs, the user will see two messages that explain what is happening. First, a log message from vLLM:

```console
WARNING 12-11 14:50:37 multiproc_worker_utils.py:281] CUDA was previously
    initialized. We must use the `spawn` multiprocessing start method. Setting
    VLLM_WORKER_MULTIPROC_METHOD to 'spawn'. See
    https://docs.vllm.ai/en/latest/getting_started/debugging.html#python-multiprocessing
    for more information.
```

Second, Python itself will raise an exception with a nice explanation:

```console
RuntimeError:
        An attempt has been made to start a new process before the
        current process has finished its bootstrapping phase.

        This probably means that you are not using fork to start your
        child processes and you have forgotten to use the proper idiom
        in the main module:

            if __name__ == '__main__':
                freeze_support()
                ...

        The "freeze_support()" line can be omitted if the program
        is not going to be frozen to produce an executable.

        To fix this issue, refer to the "Safe importing of main module"
        section in https://docs.python.org/3/library/multiprocessing.html
```

Alternatives Considered

Detect if a `__main__` guard is present

It has been suggested that we could behave better if we could detect whether
code using vLLM as a library has a `__main__` guard in place. This [post on
stackoverflow](https://stackoverflow.com/questions/77220442/multiprocessing-pool-in-a-python-class-without-name-main-guard)
was from a library author facing the same question.

It is possible to detect whether we are in the original, `__main__` process, or
a subsequent spawned process. However, it does not appear to be straight forward
to detect whether a `__main__` guard is present in the code.

This option has been discarded as impractical.

Use `forkserver`

At first it appears that `forkserver` is a nice solution to the problem. However, the way it works presents the same challenges that `spawn` does when vLLM is used as a library.

Force `spawn` all the time

One way to clean this up is to just force the use of `spawn` all the time and document that the use of a `__main__` guard is required when using vLLM as a library. This would unfortunately break existing code and make vLLM harder to use, violating the desire to make the `LLM` class as easy as possible to use.

Instead of pushing this on our users, we will retain the complexity to do our best to make things work.

Future Work

We may want to consider a different worker management approach in the future that works around these challenges.

1. We could implement something `forkserver`-like, but have the process manager be something we initially launch by running our own subprocess and a custom entrypoint for worker management (launch a `vllm-manager` process).

2. We can explore other libraries that may better suit our needs. Examples to consider:

- <https://github.com/joblib/loky>