Lab Solution 6: Threads vs Processes

Students are encouraged to share their ideas in the #lab6 Slack channel. SCPD students are welcome to reach out to me directly if they have questions that can't be properly addressed without being physically present for a discussion section.

Solution 1: Threads vs Processes

According to some CAs, a good number of students are pondering the pros and cons of threads versus processes. To provide some answers, we'll lead you through a collection of short answer questions about multiprocessing and multithreading that focuses more on the big picture.

- What does it mean when we say that a process has a private address space?
    - *It means that its range of addresses are, by default, private to the owning process, and that it's impossible for an another arbitrary, unprivileged process to access any of it.*
- What are the advantages of a private address space?
    - *The fact that it's private, of course. Most other programs can't accidentally or intentionally examine another process's data.*
- What are the disadvantages?
    - *That fact that it's private, of course. Address space privacy makes it exceptionally difficult to exchange data with other processes, and works against any efforts to breezily distribute work over multiple CPUs using multiprocessing.*
- What programming directives have we used in prior assignments and discussion section handouts to circumvent address space privacy?
    - `ptrace` *was used in Assignment 3 to allow one process to examine pretty much everything in a second one. Memory maps (via* `mmap` *and* `munmap`*) were used in the parallel* `mergesort` *section example to share large data structures between many processes. Signals and pipes have been used to foster communication between multiple processes in many assignments and section examples.*
- In what cases do the processes whose private address spaces are being publicized have any say in the matter?
    - *An individual process has very little protection against* `ptrace`*, which is why it's never a good idea to store sensitive data in the clear within a process, since everything is discoverable. System administrators can disable* `ptrace` *for a specific set of executables, or for everything, but they rarely do that for development machines like the* `myth`*s,* `rye`*s, or* `corn`*s. For more information about how system administrators can limit the use of* `ptrace` *system-wide, check out this link.*
    - *Processes also have little say about pipes, since they're generally set up and used to rewire standard input, standard output, and standard error before* `execvp` *is used to bring a new executable into the space of running processes.*

  - *Don't like signals? You can install `SIG_IGN` to ignore most of them and/or block them for the lifetime of the executable if you want to. The only signals that can't be ignored or fully blocked are `SIGKILL` and `SIGCONT`.*
  - *Memory mapped segments can be forced onto child processes, but those memory mapped segments are pretty much ignored the instant a virtual address space is cannibalized by an `execvp` call.*
- When architecting a larger program like `farm` or `stsh` that relies on multiprocessing, what did we need to do to exchange information across process boundaries?
  - *`farm` relied on pipes to forward data to each of the workers, and it relied on `SIGTSTP`, `SIGCHLD`, `SIGCONT`, `SIGKILL`, and a `SIGCHLD` handler to manage synchronization issues. And in some sense, `farm` relied on the `execvp`'s string arguments to influence what each of the workers should be running.*
  - *`stsh` relied on pretty much the same thing, albeit for different reasons. `stsh` relied on signals and signal handlers to manage job control, terminal control transfer to be clear who's responding to keyboard events and any signals associated with them, and pipes to foster communication between neighboring processes in a pipeline.*
- Can a process be used to execute multiple executables? Restated, can it `execvp` twice to run multiple programs?
  - *Nope, not in general. A program like `stsh` can fork off additional processes, each of which `execvp`s, but that's not the same thing. Bottom line is that process spaces can't be reused to run multiple executables. They can only be transformed once.*
- Threads are often called lightweight processes. In what sense are they processes? And why the lightweight distinction?
  - *Each thread of execution runs as if it's a miniature program. Threads have their own stack, and they have access to globals, dynamic memory allocation, global data, system calls, and so forth. They're relatively lightweight compared to true processes, because you don't need to create a* ***new*** *process when creating a thread. The thread manager cordons off a portion of the full stack segment for the new thread's stack, but otherwise the thread piggybacks off existing text, heap, and data segments previously establish by `fork` and `execvp`. (For more info, go ahead and read the man page for `clone`).*
- Threads are often called virtual processes as well. In what sense are threads an example of virtualization?
  - *Virtualization is an abstraction mechanism used to make a single resource appear to be many or to make many resources appear to be one. In this case, the process is subdivided to house many lightweight processes, so that one process is made to look like many.*
- Threads running within the same process all share the same address space. What are the advantages and disadvantages of allowing threads to access pretty much all of virtual memory?
  - *It's a double-edged sword, as you're all learning with Assignments 5 and 6. Because all threads have access to the same virtual address space, it's relatively trivial to share information. However, because two or more threads might want to perform surgery*

on a shared piece of data, directives must be implanted into thread routines to ensure data and resources are shared without inspiring data corruption via race conditions, or deadlock via resource contention. That's a very difficult thing to consistently get right.

Each thread within a larger process is given a thread id, also called a **tid**. In fact, the thread id concept is just an extension of the process id. For singly threaded processes, the pid and the main thread's tid are precisely the same. If a process with pid 12345 creates three additional threads beyond the main thread (and no other processes or threads within other processes are created), then the tid of the main thread would be 12345, and the thread ids of the three other threads would be 12346, 12347, and 12348.

- What are the advantages of leveraging the pid abstraction for thread ids?
    - *To the extent that the OS can view threads as processes, the OS can provide services on a per-thread basis instead of just a per-process one.*
- What happens if you pass a thread id that isn't a process id to `waitpid`?
    - *Poorly documented, but the takeaway here is that `waitpid` does not consider it to be an error if a thread id is passed to `waitpid`. `waitpid`'s man page includes some information about how `waitpid` and threads interact with one another.*
- What happens if you pass a thread id to `sched_setaffinity`?
    - *It actually works, and informs the OS that the identified thread should only be run on those CPUs included in the provided `cpuset_t`.*
- What are the advantages of requiring that a thread always be assigned to the same CPU?
    - *Each CPU typically has a dedicated L1 cache that stores data fetched by instructions run on that CPU. CPU-specific L1 caches are more likely to retain content relevant to the thread if the same thread keeps coming back instead of being arbitrarily assigned to other CPUs.*

In some situations, the decision to rely on multithreading instead of multiprocessing is dictated solely by whether the code to be run apart from the main thread is available in executable or in library form. But in other situations, we have a choice.

- Why might you prefer multithreading over multiprocessing if both are reasonably good options?
    - *Ease of data exchange and code sharing.*
- Why might you prefer multiprocessing over multithreading if both are reasonably good options?
    - *Protection, privacy, insulation from other processes errors.*
- What happens if a thread within a larger process calls `fork`?
    - *It creates a clone of the entire process surrounding it, so that the process clone also has the same number of threads running as the original at the time of the **fork**.*

- What happens if a thread within a larger process calls `execvp`?
    - *It transforms the entire process to run a new executable. The fact that the pre-`execvp` process involved threading is irrelevant once `execvp` is invoked. It's **that** drastic.*

And some final questions about how `farm`, `aggregate`, and `stsh` could have been implemented:

- Assume you know nothing about multiprocessing but needed to emulate the functionality of `farm`. How could you use multithreading to design an equally parallel solution without ever relying on multiprocessing?
    - *You'll see over the coming week that a ThreadPool is a generalization of what we need here.*
        - *Spawn one thread for each CPU, and set the thread routine to the code that computes and prints a factorization (using `oslock` and `osunlock`, of course). Note that all workers are initially available by storing something on behalf of each thread in the parent (array indices, thread ids, whatever you want).*
        - *Implement the `main` thread to continue and read all numbers to be factored, forwarding numbers on to available workers (I'm being intentionally vague here as as to not steal the thunder of the `ThreadPool`).*
        - *As workers are assigned numbers to be factored, some note they're occupied, and as each worker finishes printing its factorization, have it mark itself as available.*
        - *Use `semaphore`s to coordinate rendezvous communication between orchestrator and each workers. Rely on `mutex`es to guard shared data structures (e.g. the set of available workers) from race conditions.*
        - *When all numbers have been factored and all threads are available, prompt each thread to gracefully exit, `join` on all of those threads, and exit the program.*
- Inversely, how could you have implemented `aggregate` to rely on multiprocessing (without threading) to arrive at an equally parallel solution?
    - *Architecture is similar, except that:*
        - *forked processes (without `execvp`'ing, so that shared memory segments aren't lost) stand in for threads,*
        - *shared memory (via `mmap`) stores process count limits and simple data structures,*
        - *one can guard against race conditions using `flock` (man `flock` for more information),*
        - *one can rely on `waitpid` to note when child and grandchild processes finish,*
        - *each grandchild can create a file storing the URL and the raw stream of tokens for each URL. In fact, you can use a hash of the URL to name the files so that you can easily detect whether a URL has been downloaded already, and*
        - *once all subprocesses finish, have the parent process read in data from all local files to build the index, and then advance on to the query phase as your Assignment 5 aggregate does*

- Could multithreading have contributed to the implementation of `stsh` in any meaningful way? Why or why not?
  - *Not really. A shell is all about process control–`forking` and `execvp`-ing, and there are no in-shell algorithms that are intensely CPU, I/O, or network-bound, so introducing parallelism within `stsh` itself is arguably unnecessary.*

Solution 2: Threading Short Answer Questions

Here are some more short answer questions about the specifics of threads and the directives that help threads communicate.

- Is `i--` thread safe? Why or why not?
  - *`i--` is not thread safe if it expands to two or more assembly code instructions, as it does in x86_64. For a local variable not already in a register, the generated code might look like this:*

    ```
    mov    (%rbp), %rax
    sub    $1, %rax
    mov    %rax, (%rbp)
    ```

  - *If the thread gets swapped off the processor after the first or second of those two lines and another thread is scheduled and changes `i`, the original thread will eventually resume and continue on with stale data.*
  - *Some architectures **are** capable of doing an in-memory decrement (or, more broadly, a memory-memory instruction) in one instruction, so i– **could** compile to one assembly code instruction on some architectures **if** i is a global variable. But in general, you write code to be portable and architecture-agnostic, so you would need to operate as if i–, even where i is a global, is thread-unsafe.*
- What's the difference between a `mutex` and a `semaphore` with an initial value of 1? Can one be substituted for the other?
  - *The `semaphore` could be used in place of the `mutex`, but not vice versa. First off, the thread that acquires the `lock` on a `mutex` must be the one to unlock it (by C++ specification), whereas one thread can `signal` a `semaphore` while a second thread waits on it. There's also nothing to prevent a `semaphore` initialized to 1 from being signaled to surround an even higher number (like 2, or 2 million), whereas a `mutex` can really only be in one of two states.*
- What is the `lock_guard` class used for, and why is it useful?
  - *It's used to layer over some kind of mutex (typically a `mutex`, but possibly some other mutex classes like `timed_mutex` and `recursive_mutex`) so that it's locked when it needs to be locked (via the `lock_guard` constructor) and unlocked when the `lock_guard` goes out of scope (via its destructor). It's useful because it guarantees that a mutex that's locked through a `lock_guard` will be unlocked no matter how the*

function, method, or enclosing scope ends. In particular, a function might have two or more exits paths (some early returns, some exceptions being thrown, and so forth).

- What is busy waiting? Is it ever a good idea? Does your answer to the good-idea question depend on the number of CPUs your multithreaded application has access to?
  - *Busy waiting is consuming processor time waiting for some condition to change, as with `while (numAvailableWorkers == 0) {;}`. In a single CPU machine, it makes sense for a process or thread that would otherwise busy wait to yield the processor, since the condition can't possibly change until some other thread gets the processor and makes changes to the variables that are part of the condition. That's what `sigsuspend` (for processes) and `conditional_variable_any::wait` (for threads) are for. In a few scenarios where multithreaded code is running on a machine with multiple CPUs, it's okay to busy wait **if and only** if you know with high probability that another thread is running at the same time (on another CPU) and will invert the condition that's causing the first thread to busy wait.*
- As it turns out, the semaphore's constructor allows a negative number to be passed in, as with `semaphore s(-11)`. Identify a scenario where -11 might be a sensible initial value.
  - *If one thread adds twelve thunks to a `ThreadPool` and then needs to wait on just those twelve threads (and none of the others that happen to be in the `ThreadPool` for other reasons), then you could implement this one of two ways:*
    - *Initialize a `semaphore` to surround a 0, share that `semaphore` with each of the twelve thunks, have each thunk `signal` the `semaphore` once just as it's exiting, and require the parent thread to `wait` on that `semaphore` 12 times,* ***or***
    - *Initialize a `semaphore` to surround a -11, share that `semaphore` with each of the twelve thunks, have each thunk `signal` the `semaphore` once just as it's exiting, and require the parent thread to `wait` on that `semaphore` just once! `wait` still requires that the encapsulated `count` be positive before it decrements and returns, so this works just as well without prompting the parent to keep waking make it through one of twelve `wait` calls.*
- What would the implementation of `semaphore::signal(size_t increase = 1)` need to look like if we wanted to allow a `semaphore`'s encapsulated value to be promoted by the `increase` amount? Note that `increase` defaults to 1, so that this version could just replace the standard `semaphore::signal` that's officially exported by the `semaphore` abstraction.
  - *The key line in signal would need to replace the ++ with a += increase. It would also need to decide whether the condition variable needs to be notified by checking to see if the encapsulated value pre-increment was less than or equal to 0 and the value post-increment is positive. If so, then it should call `notify_one` if the new value become 1, and `notify_all` if the new value is 2 or more.*
- What's the multiprocessing equivalent of the `mutex`?
  - *It's not a perfect parallel, but if I had to choose one, it'd be signal masks. We used signal masks to remove the possibility of interruption by signal, which is the only thing that can otherwise introduce a race condition into a sequential program. (Another answer is something I referenced in an earlier solution, even though I didn't teach it in*

CS110. Investigate the `flock` function and think about how it could be used to prevent interprocess race condition on shared memory mapped segments.)

- What's the multiprocessing equivalent of the `condition_variable_any`?
  - *`sigsuspend` is the equivalent of `conditional_variable_any::wait`, and a signal outside the `sigsuspend` signal mask is the equivalent of `conditional_variable_any::notify_[one|all]`.*