Multithreaded Processes

Systems Capstone CS4284

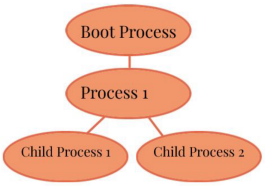
Spencer Bone, John Jang, Steve Park, and Rebecca Yeung

Introduction

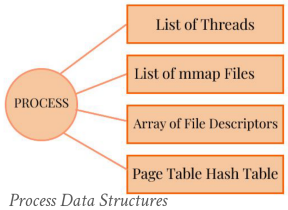
Multithreaded Processing is the design of executed processes to wield multiple threads. By wielding multiple threads, processes are able to distribute workload over more workers, attempting to hasten execution. This makes the overall goal of the project to observe speedup within executed programs.

Process Design

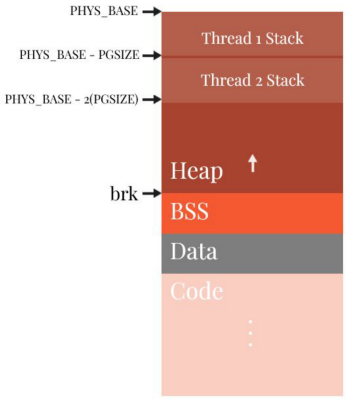
Previously, each process was only to be executed by a single thread. Now, processes have the capability to utilize multiple threads. This requires the creation of a new process structure which will hold the data to be shared between the worker threads. A list of child processes is included as well.



Process Parents have a Children List



Process Data Structures



Process stack, including pthread stacks and the heap for sbrk

Thread Creation and PThread API

There needs to be a way for users to create said additional threads. This called for a user library of functions in which the kernel is called to make and add threads to a given process, pthread_create. This project works under the assumption that the user will call pthread_join.

Malloc and Sbrk

The first extra challenge was implementing the memory allocation call malloc on the user side. While this isn't a requirement to get multithreaded processes working, adding malloc allows for practical use and tests for this project. Most of the work was already done in CS3214 malloc project, the main addition required was a sbrk system call.

Synchronization

The second challenge is allowing for users to protect data within user code. The simple implementation is using a system call to use the given kernel locks and semaphores, making a slight inefficiency in the amount of context switches into the kernel for each synchronization interaction. The other alternative is to implement something close to the futex API, making threads able to check ownership without a context swap.

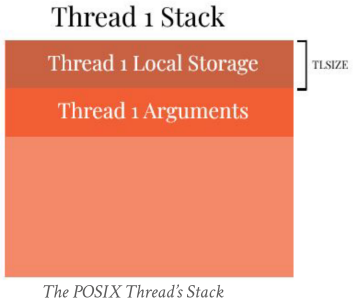
An example of how a process (P) can run on multiple CPU's with multiple worker threads (T)

Testing

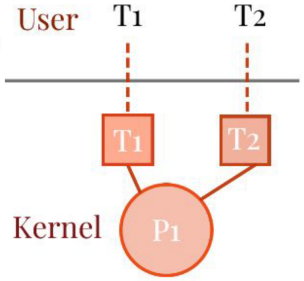
Tests were also created in order to check correctness within instantiation and later joining of threads. These tests allowed for any issues within the implementation to be observed and analyzed when debugging.

Thread Local Variables

Another addition required for getting the CS3214 Threadpool project to run was implementing thread local variables, where data can be stored exclusively on the threads stack. This is done by reserving a small chunk of memory at the top of the thread's stack.



The POSIX Thread's Stack

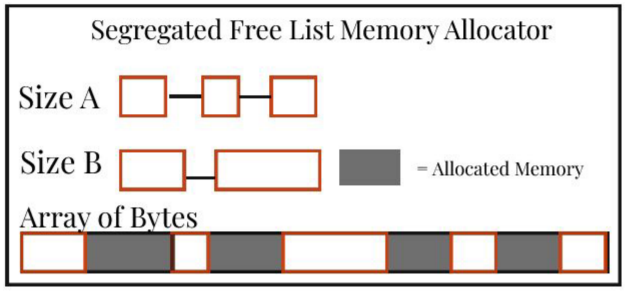


Users access threads via handles to kernel threads

Fixing Prior PintOS Issues

The last issue requiring attention was the flaws in the code from the previous projects of this Capstone course:

1) The load balancing required a check to make sure all the CPUs are up.
2) The system calls required frame pinning for I/O, as well as some argument accesses were off.
3) The frame table as well obtained the clock algorithm for frame eviction (instead of LRU).



The design used for the user side memory allocator via malloc

Conclusion

Overall, multithreaded processes are complex, from the one-to-one process-thread execution becoming one-to-many, creating an API for not only making, joining threads and using thread local variables but also for the user synchronization primitives, adding a user side memory allocator, and lastly making new tests to specifically examine these new capabilities. While multithreading isn't brand new, it is interesting to build the foundation of parallel processing used in other classes like CS3214 Systems.