ECE 459: Programming for Performance
Assignment 1

Pavel Shering

January 26, 2018

Part 0: Resource Leak

The resource leak is caused by not freeing the *png_write_struct*() in the *write_png_file*() function. It is fixed by destroying the png struct at the end of the write function using png_destroy_write_struct (&png_ptr, &info_ptr) which takes in a png info struct pointer and a write struct pointer.

The next task is to make sure that there are no reachable bytes at the end of each program. The reason for the left over reachable byte is due to the *curl_easy_init*() function call that automatically calls *curl_global_init*() [curl_easy_init reference manual], however, *curl_easy_cleanup*() does not release all resources acquired by *curl_global_init*(). Thus is fixed by cleaning up the curl initialization using *curl_global_cleanup*().

The Valgrind output 1 shows all programs passing the leak check with no reachable or lost resources.

Part 1: Pthreads

0.1 Thread Safety

The use of libcurl in paster_parallel.c is thread safe. The *curl_global_init*() [curl_easy_init reference manual] and *curl_global_cleanup*() functions are not thread-safe, thus are only called in *main*(). Additionally, *curl_easy_init*() is thread-safe and can be called in each thread along with *curl_easy_cleanup*()

"Libpng is thread safe, provided the threads are using different instances of the structures. Each thread should have its own *png_struct* and *png_info* instances, and thus its own image. Libpng does not protect itself against two threads using the same instance of a structure." [libpng reference manual] Thus, in paster_parallel.c each *ThreadRoutine*() utilizes its own *png_struct* and *png_info* instances making it thread-safe.

Lastly, glibc use is also thread-safe because the *rand_r*() function is marked as part of the Thread-Safe Functions option [glibc reference manual] that is used to randomly choose a server to create a load distribution for threads.

0.2 Race Conditions

By definition a race condition occurs when there are two or more concurrent accesses to the same memory location, and at least one of them is a **write**. There are two types of race conditions that occur in this program.

First, WAW (write after write), this occurs when multiple threads access the same output buffer to write the retrieved fragment. Although, two threads require access to the same output buffer and at times even the same location in the buffer if the same fragment is retrieved, this posses no danger to the final output, nor any loss in data, since one will just overwrite the same fragment. Furthermore, all the threads are joined right before the output buffer is passed into the $write\_png\_file()$ function, thus overwriting of data does finish to the end. One way to stop the exact same location access is to not write the same fragment twice (which will also be quicker, however this is not implemented).

As an aside, the threaded parallelization is implemented in two different ways, using the detached threads and joined threads techniques. The detached thread technique (paster_parallel_detach.c can be found in git commit history, the assignment only asks for 3 files) also does not have the issue of write after write because after all the fragments are received the $sem\_wait()$ function is called num_threads amount of times to make sure all detached threads finish. The detach technique turns out to be the slowest (average run time 140.85 sec) due to the overhead of cleaning up detached threads, initializing new ones and detaching those, as well as the use of semaphores that add extra overhead to make sure all threads are completed. For the joined thread technique the threads are joined after they are created and all fragments are received thus completing all write calls to the output buffer.

Second, WAR (write after read) occurs when checking the $received\_fragments$ boolean array. Although, one thread could be writing to the array as the read check occurs, this posses no danger as all the values in the array are initially set to false, and the write only sets the array value to only true. Thus, even if the read reads false as the write occurs (this is if the write is not atomic) the only consequence is that the do-while loop runs once more time through without corruption to data. Thus, there are no race conditions that harm the output data.

The experiments are run on an Intel(R) Core(TM) i3-4170 CPU @ 3.70GHz CPU. It has 2 physical cores and 4 virtual CPUs. Tables 1 and 2 present the results. All results are retrieved on Friday January 26th around 6:00pm, thus the amount of runs is increased to five per test to have a better understanding of the average run time, and outliers are rejected.

	Time (s)
Run 1	19.346241
Run 2	25.317257
Run 3	20.679055
Run 4	16.710824
Run 5	26.077884
Average	21.6262522

Table 1: Sequential executions terminate in a mean of 21.6262522 seconds.

	N=4, Time (s)	N=64, Time (s)
Run 1	15.432344	27.854541
Run 2	11.565613	28.768013
Run 3	18.103361	38.788469
Run 4	15.629653	23.659233
Run 5	17.761410	32.851811
Average	15.6985	30.3844

Table 2: Parallel executions terminate in a mean of 15.6985 seconds for 4 threads, and 30.3844 seconds for 64 threads.

The parallelization does not scale well with higher number of threads, as N increases the execution time increases due to the overhead of joining all the created threads once all the fragments have been retrieved. Hence, four threads are cleaned up faster than 64 even though the 64 threads may get all the fragments faster. However, the parallelization for retrieving multiple fragments at once (4 at once) is close to 30% faster.

Part 2: Nonblocking I/O

Table 3 presents results from my non-blocking I/O implementation. I started N requests simultaneously.

	N=4, Time (s)	N=64, Time (s)
Run 1	12.218069	7.963569
Run 2	15.640460	6.557022
Run 3	13.075839	6.712527
Run 4	16.853745	6.513255
Run 5	13.847099	6.780882
Run 6	12.252948	6.728201
Average	13.98136	6.8759

Table 3: Non-blocking I/O executions terminate in a mean of 13.98136 seconds for 4 open connections and 6.8759 seconds for 64 open connections.

Discussion. Not surprisingly, the sequential execution ran the slowest. The slowest part is retrieving the fragments from the servers, thus only one thread or process of such will be slower up until the overhead of joining threads takes over as seen in Table 2 where with 64 threads the execution becomes slower than sequential. The non blocking I/O is the fastest, again not surprising as it does not require the joining threads or their clean up, it simply retrieves data and processes it as soon as any becomes available. Thus, non blocking I/O with 64 open connections has the fastest execution time taken up by the first wait for data by the fastest server and from then on the time to write to output buffer and final write.

Part 3: Amdahl's Law and Gustafson's Law

To measure the sequential portion of `paster_parallel` the writing png to file portion is measured. Over 3 runs, it took an average of 3.129849 seconds. By Amdahl's Law it would be theoretically possible to then get the execution time to as close to 3.129849 seconds as possible assuming that includes all the overhead of initializations and cleanup by increasing the number of threads or open connections, thus, essentially bringing the parallelizable part to zero.

1 Appendix

Listing 1: Valgrind output for all files

```
pshering@ece-tesla1:~/ece459-1181-a1-pshering$ valgrind --leak-check=full --show
    ↪ -leak-kinds=all bin/paster
==8213== Memcheck, a memory error detector
==8213== Copyright (C) 2002-2015, and GNU GPL'd,␣by␣Julian␣Seward␣et␣al.
==8213==␣Using␣Valgrind-3.11.0␣and␣LibVEX;␣rerun␣with␣-h␣for␣copyright␣info
==8213==␣Command:␣bin/paster
==8213==
cpu␣␣:␣47.932079␣secs
user␣:␣81.797257␣secs
==8213==
==8213==␣HEAP␣SUMMARY:
==8213==␣␣␣␣␣in␣use␣at␣exit:␣0␣bytes␣in␣0␣blocks
==8213==␣␣␣total␣heap␣usage:␣231,280␣allocs,␣231,280␣frees,␣238,770,906␣bytes␣
    ↪ allocated
==8213==
==8213==␣All␣heap␣blocks␣were␣freed␣--␣no␣leaks␣are␣possible
==8213==
==8213==␣For␣counts␣of␣detected␣and␣suppressed␣errors,␣rerun␣with:␣-v
==8213==␣ERROR␣SUMMARY:␣0␣errors␣from␣0␣contexts␣(suppressed:␣0␣from␣0)
pshering@ece-tesla1:~/ece459-1181-a1-pshering$␣valgrind␣--leak-check=full␣--show
    ↪ -leak-kinds=all␣bin/paster_parallel␣-t␣4
==9060==␣Memcheck,␣a␣memory␣error␣detector
==9060==␣Copyright␣(C)␣2002-2015,␣and␣GNU␣GPL'd, by Julian Seward et al.
==9060== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==9060== Command: bin/paster_parallel -t 4
==9060==
cpu  : 66.480388 secs
user : 82.259978 secs
==9060==
==9060== HEAP SUMMARY:
==9060==     in use at exit: 0 bytes in 0 blocks
==9060==   total heap usage: 170,591 allocs, 170,591 frees, 226,032,632 bytes
    ↪ allocated
==9060==
==9060== All heap blocks were freed -- no leaks are possible
==9060==
==9060== For counts of detected and suppressed errors, rerun with: -v
==9060== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
pshering@ece-tesla1:~/ece459-1181-a1-pshering$ valgrind --leak-check=full --show
    ↪ -leak-kinds=all bin/paster_parallel -t 64
==12856== Memcheck, a memory error detector
==12856== Copyright (C) 2002-2015, and GNU GPL'd,␣by␣Julian␣Seward␣et␣al.
==12856==␣Using␣Valgrind-3.11.0␣and␣LibVEX;␣rerun␣with␣-h␣for␣copyright␣info
==12856==␣Command:␣bin/paster_parallel␣-t␣64
==12856==
cpu␣␣:␣72.916693␣secs
user␣:␣122.470446␣secs
```

```
==12856==
==12856==␣HEAP␣SUMMARY:
==12856==␣␣␣␣␣in␣use␣at␣exit:␣0␣bytes␣in␣0␣blocks
==12856==␣␣␣total␣heap␣usage:␣267,856␣allocs,␣267,856␣frees,␣938,760,570␣bytes␣
   ↪ allocated
==12856==
==12856==␣All␣heap␣blocks␣were␣freed␣--␣no␣leaks␣are␣possible
==12856==
==12856==␣For␣counts␣of␣detected␣and␣suppressed␣errors,␣rerun␣with:␣-v
==12856==␣ERROR␣SUMMARY:␣0␣errors␣from␣0␣contexts␣(suppressed:␣0␣from␣0)
pshering@ece-tesla1:~/ece459-1181-a1-pshering$␣valgrind␣--leak-check=full␣--show
   ↪ -leak-kinds=all␣bin/paster_parallel_detached␣-t␣4
==14327==␣Memcheck,␣a␣memory␣error␣detector
==14327==␣Copyright␣(C)␣2002-2015,␣and␣GNU␣GPL'd, by Julian Seward et al.
==14327== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==14327== Command: bin/paster_parallel_detached -t 4
==14327==
cpu  : 54.753208 secs
user : 188.459511 secs
==14327==
==14327== HEAP SUMMARY:
==14327==     in use at exit: 0 bytes in 0 blocks
==14327==   total heap usage: 204,982 allocs, 204,982 frees, 875,908,743 bytes
   ↪ allocated
==14327==
==14327== All heap blocks were freed -- no leaks are possible
==14327==
==14327== For counts of detected and suppressed errors, rerun with: -v
==14327== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
pshering@ece-tesla1:~/ece459-1181-a1-pshering$ valgrind --leak-check=full --show
   ↪ -leak-kinds=all bin/paster_parallel_detached -t 64
==16407== Memcheck, a memory error detector
==16407== Copyright (C) 2002-2015, and GNU GPL'd,␣by␣Julian␣Seward␣et␣al.
==16407==␣Using␣Valgrind-3.11.0␣and␣LibVEX;␣rerun␣with␣-h␣for␣copyright␣info
==16407==␣Command:␣bin/paster_parallel_detached␣-t␣64
==16407==
cpu␣␣:␣70.734554␣secs
user␣:␣112.268500␣secs
==16407==
==16407==␣HEAP␣SUMMARY:
==16407==␣␣␣␣␣in␣use␣at␣exit:␣0␣bytes␣in␣0␣blocks
==16407==␣␣␣total␣heap␣usage:␣339,350␣allocs,␣339,350␣frees,␣1,439,907,711␣bytes
   ↪ ␣allocated
==16407==
==16407==␣All␣heap␣blocks␣were␣freed␣--␣no␣leaks␣are␣possible
==16407==
==16407==␣For␣counts␣of␣detected␣and␣suppressed␣errors,␣rerun␣with:␣-v
==16407==␣ERROR␣SUMMARY:␣0␣errors␣from␣0␣contexts␣(suppressed:␣0␣from␣0)
pshering@ece-tesla1:~/ece459-1181-a1-pshering$␣valgrind␣--leak-check=full␣--show
   ↪ -leak-kinds=all␣bin/paster_nbio␣-t␣4
==17981==␣Memcheck,␣a␣memory␣error␣detector
==17981==␣Copyright␣(C)␣2002-2015,␣and␣GNU␣GPL'd, by Julian Seward et al.
==17981== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==17981== Command: bin/paster_nbio -t 4
==17981==
cpu  : 91.516369 secs
user : 120.215240 secs
```

```
==17981==
==17981== HEAP SUMMARY:
==17981==     in use at exit: 0 bytes in 0 blocks
==17981==   total heap usage: 240,900 allocs, 240,900 frees, 244,544,544 bytes
   ↪ allocated
==17981==
==17981== All heap blocks were freed -- no leaks are possible
==17981==
==17981== For counts of detected and suppressed errors, rerun with: -v
==17981== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
pshering@ece-tesla1:~/ece459-1181-a1-pshering$ valgrind --leak-check=full --show
   ↪ -leak-kinds=all bin/paster_nbio -t 64
==21656== Memcheck, a memory error detector
==21656== Copyright (C) 2002-2015, and GNU GPL'd,␣by␣Julian␣Seward␣et␣al.
==21656==␣Using␣Valgrind-3.11.0␣and␣LibVEX;␣rerun␣with␣-h␣for␣copyright␣info
==21656==␣Command:␣bin/paster_nbio␣-t␣64
==21656==
cpu␣␣:␣60.419141␣secs
user␣:␣64.169157␣secs
==21656==
==21656==␣HEAP␣SUMMARY:
==21656==␣␣␣␣␣in␣use␣at␣exit:␣0␣bytes␣in␣0␣blocks
==21656==␣␣␣total␣heap␣usage:␣149,744␣allocs,␣149,744␣frees,␣844,661,642␣bytes␣
   ↪ allocated
==21656==
==21656==␣All␣heap␣blocks␣were␣freed␣--␣no␣leaks␣are␣possible
==21656==
==21656==␣For␣counts␣of␣detected␣and␣suppressed␣errors,␣rerun␣with:␣-v
==21656==␣ERROR␣SUMMARY:␣0␣errors␣from␣0␣contexts␣(suppressed:␣0␣from␣0)
pshering@ece-tesla1:~/ece459-1181-a1-pshering$
```