CS 140 Lab Report 6

Dale Sealtiel T. Flores
2023-11373
THX/WXY

1. Regarding Section 3.1, explain **(1)** what happens when `global.c` is executed and **(2)** why its output is not consistent across multiple reruns of the same executable file.

   Answer: When `global.c` is executed, the program creates two threads `t1` and `t2` using `pthread_create`. The threads run `increment()` and `decrement()` respectively. These two functions modify a global variable named `global` by incrementing and decrementing it. This is why the output of `global.c` is inconsistent across multiple runs. This is because both threads are accessing and changing the same variable `global` at the same time.

2. Regarding Section 3.2, provide a screenshot of the output of `stack` and explain how threads maintain locality of local variables despite having shared address space by referring to the addresses in the screenshot. Relate your answer to the contrasting behavior of `global`.

   Answer:

   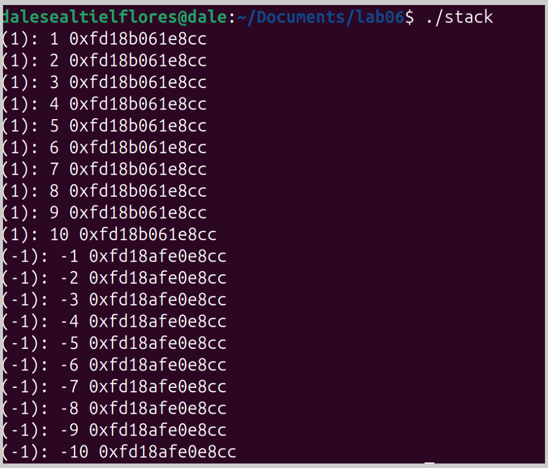

   Figure 1: Output of `stack`

   We can see from the screenshot that the two different threads with delta `(1)` and `(-1)` have different stack addresses. This is because each thread has its own stack. In their respective stack, they have their own local variables, namely `x` and `delta`. Compared to `global.c`, where both threads are accessing the same `global` variable. This is why the output of `stack` is consistent across multiple runs.

3. Regarding Section 3.3.3, provide screenshots of output of `strace` for both the threaded and the forking C programs an identify which `clone` flag is responsible for ensuring shared address spaces. State all references used.

```
dalesealtielflores@dale:~/Documents/lab06$ strace ./thread
execve("./thread", ["./thread"], 0xffffeeceb9f0 /* 49 vars */) = 0
brk(NULL)                               = 0xb4709f70d000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xe762aeb8f000
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=64751, ...}) = 0
mmap(NULL, 64751, PROT_READ, MAP_PRIVATE, 3, 0) = 0xe762aeb7f000
close(3)                                = 0
openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\360\206\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1722920, ...}) = 0
mmap(NULL, 1892240, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_DENYWRITE, -1, 0) = 0xe762ae988000
mmap(0xe762ae990000, 1826704, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0xe762ae990000
munmap(0xe762ae988000, 32768)           = 0
munmap(0xe762aeb4e000, 32656)           = 0
mprotect(0xe762aeb2a000, 77824, PROT_NONE) = 0
mmap(0xe762aeb3d000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19d000) = 0xe762aeb3d000
mmap(0xe762aeb42000, 49040, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xe762aeb42000
close(3)                                = 0
set_tid_address(0xe762aeb8ffb0)         = 2960
set_robust_list(0xe762aeb8ffc0, 24)     = 0
rseq(0xe762aeb90600, 0x20, 0, 0xd428bc00) = 0
mprotect(0xe762aeb3d000, 12288, PROT_READ) = 0
mprotect(0xb4709debf000, 4096, PROT_READ) = 0
mprotect(0xe762aeb94000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0xe762aeb7f000, 64751)           = 0
rt_sigaction(SIGRT_1, {sa_handler=0xe762aea12840, sa_mask=[], sa_flags=SA_ONSTACK|SA_RESTART|SA_SIGINFO}, NULL, 8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
mmap(NULL, 8454144, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0xe762ae180000
mprotect(0xe762ae190000, 8388608, PROT_READ|PROT_WRITE) = 0
getrandom("\xd6\x11\x2f\xb0\x1b\xd8\xcf\xf7", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0xb4709f70d000
brk(0xb4709f72e000)                     = 0xb4709f72e000
rt_sigprocmask(SIG_BLOCK, ~[], [], 8)   = 0
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHIL
D_CLEARTID, child_tid=0xe762ae98f270, parent_tid=0xe762ae98f270, exit_signal=0, stack=0xe762ae180000, stack_size=0x80ea60, tls=0
xe762ae98f8e0} => {parent_tid=[2961]}, 88) = 2961
Hello world!
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Figure 2: Output of `strace` for `thread`

```
dalesealtielflores@dale:~/Documents/lab06$ strace ./fork
execve("./fork", ["./fork"], 0xffffe82ef1f0 /* 49 vars */) = 0
brk(NULL)                               = 0xab78157d9000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf62da7f9e000
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=64751, ...}) = 0
mmap(NULL, 64751, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf62da7f8e000
close(3)                                = 0
openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0\360\206\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1722920, ...}) = 0
mmap(NULL, 1892240, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_DENYWRITE, -1, 0) = 0xf62da7d97000
mmap(0xf62da7da0000, 1826704, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0xf62da7da0000
munmap(0xf62da7d97000, 36864)           = 0
munmap(0xf62da7f5e000, 28560)           = 0
mprotect(0xf62da7f3a000, 77824, PROT_NONE) = 0
mmap(0xf62da7f4d000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19d000) = 0xf62da7f4d000
mmap(0xf62da7f52000, 49040, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xf62da7f52000
close(3)                                = 0
set_tid_address(0xf62da7f9ef90)         = 3154
set_robust_list(0xf62da7f9efa0, 24)     = 0
rseq(0xf62da7f9f5e0, 0x20, 0, 0xd428bc00) = 0
mprotect(0xf62da7f4d000, 12288, PROT_READ) = 0
mprotect(0xab77e5a1f000, 4096, PROT_READ) = 0
mprotect(0xf62da7fa3000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0xf62da7f8e000, 64751)           = 0
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0xf62da7f9ef90) = 3155
parent
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=3155, si_uid=1000, si_status=0, si_utime=0, si_stime=0} ---
getrandom("\xef\xb1\x22\x01\x43\xba\xaf\x58", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0xab78157d9000
brk(0xab78157fa000)                     = 0xab78157fa000
write(1, "child\n", 6child
)                    = 6
exit_group(0)                           = ?
+++ exited with 0 +++
```

Figure 3: Output of `strace` for `fork`

> **Answer:** The `CLONE_VM` flag is responsible for ensuring that for the `thread` program, the threads share the same address space. For the `fork` program, there is no `CLONE_VM` flag, which means that the child has its own address space.

4. Regarding Section 3.4, answer the following:
   (a) Provide the realtime execution durations recorded earlier for `speedup` (at least ten for each value of `N`) and compute the average of each set. Explain why the execution time of `speedup` is shorter when `NTHREADS` is changed from 1 to 2.

Table 1: Execution times for `NTHREADS = 1`

Run	Execution Time
1	0m0.966s
2	0m0.893s
3	0m0.790s
4	0m0.773s
5	0m0.807s
6	0m0.783s
7	0m0.856s
8	0m0.622s
9	0m0.601s
10	0m0.591s

Table 2: Execution times for `NTHREADS = 2`

Run	Execution Time
1	0m0.591s
2	0m0.670s
3	0m0.586s
4	0m0.780s
5	0m0.748s
6	0m0.696s
7	0m0.496s
8	0m0.701s
9	0m0.676s
10	0m0.595s

   (b) Explain why there is no race condition for the threads in `speedup.c` despite operating on the same array.

   > **Answer:** There are no threads in `speedup.c` that operate on the same element. This is because each thread has a specific `start` that it operates on.

5. Regarding Section 3.5, answer the following:
   (a) Explain why the output of the original `sync.c` is inconsistent across several runs.

   > **Answer:** Just like in `(1)`, the threads are accesssing and modifying the same global variable `sum` at the same time.

(b) Explain how adding a mutex to `sync.c` makes it output consistent across several runs.

> **Answer:** By adding mutex, we ensure that only one thread can access and modify `sum` at a time. It locks the critical section of the code, specifically `sum += 1`, so that if one thread is executing this section, the others have to wait until it is unlocked.

6. Regarding Section 3.6, answer the following:
   (a) Given `N` threads synchronized by a properly initialized barrier, determine how many threads will end up reaching the line labeled:
      i. `(A)`

      > **Answer:** Only `1` thread will end up reaching line `(A)`

      ii. `(B)`

      > **Answer:** There will be `N-1` threads that will reach line `(B)`

   (b) Illustrate a case in which `NTHREADS` is `3` that shows some interleaved order of thread execution in which the threads end up being synchronized by the barrier. Make reference to relevant variable values and conditions

   > **Answer:** We have the following order of execution:
   > 1. Thread 1 will reach the barrier first then it will run `wait_barrier(&b, 3)`. It will then go to line `pthread_mutex_lock(&b->lock)` which will lock the barrier. It will then go to `if(b->num_waiting == 0)` which is true then it will go to `if(b->num_exited == n)` which is true then it will now run `b->is_open = 0` after that, it will exit the if statement then will now go to `b->num_waiting += 1` then it will unlock the barrier. It will now go to `if (b->num_waiting == n)` which is false then it will go to the `else` statement which is `while (b->is_open == 0) {}` which is where it will be stuck.
   > 2. Thread 2 will then reach the barrier while Thread 1 is stuck in the `while` loop. It will then run `wait_barrier(&b, 3)`. It will then go to `pthread_mutex_lock(&b->lock)` like Thread 1, it will then go `if (b->num_waiting == 0)` which will be false so it goes to `b->num_waiting += 1` then it will run `pthread_mutex_unlock(&b->lock)` now it will go to `if(b->num_waiting == n)` which is false then it will go to the `else` statement which is `while (b->is_open == 0) {}` which is where it will be stuck.
   > 3. Thread 3 will have a similar execution to Thread 2: `wait_barrier(&b, 3)`, `pthread_mutex_lock(&b->lock)`, `if(b->num_waiting == 0)`, `b->num_waiting += 1`, `pthread_mutex_unlock(&b->lock)`, `if (b->num_waiting == n)` this is where it diverges from Thread 2 because this will be true. Now it will now go to `b->num_waiting = 0`, `b->num_exited = 1`, `b->is_open = 1` then Thread 3 will be done executing. Now that `b->is_open = 1`, Thread 1 and Thread 2 will now be able to exit the `while` loop where they were stuck.
   > 4. Thread 1 will now run `pthread_mutex_lock(&b->lock)` then `b->num_exited += 1` then it will go to `pthread_mutex_unlock(&b->lock)` then it will finish executing.
   > 5. Thread 2 will now run the same as Thread 1: `pthread_mutex_lock(&b->lock)`, `b->num_exited += 1`, `pthread_mutex_unlock(&b->lock)` then it will finish executing.

   (c) Assume a barrier `b` supporting `N > 1` threads that is currently in the process of letting the `N` threads through *(i.e., the barrier is currently open for them).* Illustrate how a new thread *(i.e., not part of the `N` barrier-exiting threads* that attempts to use the same barrier via a call to `wait_barrier(&b, N)` will properly be denied passage through the barrier. Make reference to relavant variable values and conditions.

Answer: The last thread will run `b->num_waiting = 0`. This means that the new process will be able to enter `if(b->num_waiting == 0)`. It will then run `if(b->num_exited == n)` which will be false since it was stated that it is still in the process of letting the `N` threads through. It will then go to the `else` statement which will run `pthread_mutex_unlock(&b->lock)` then it will get stuck in `while(b->num_exited != n) {}` until all `N` threads have exited the barrier. Then it will run `pthread_mutex_lock(&b->lock)` then run `b->is_open = 0` then it will run `b->num_waiting += 1` then it will run `pthread_mutex_unlock(&b->lock)` then it will go to `if(b->num_waiting == n)` which will be false since it is a new thread. It will then go to the `else` statement where it will get stuck in `while(b->is_open == 0) {}` until the barrier is open again.

(d) For each `pthread_mutex_lock` call in `barrier.c` *(see comment labels*, enumerate all possible `pthread_mutex_calls` that can undo the lock that it sends on the mutex *(i.e., a possible unlock pair for the lock call)*. Listing of labels for each will suffice *(e.g., `(2)`, `(6)`)*:

- `(1)`

  Answer: `(2)`, `(4)`

- `(3)`

  Answer: `(4)`

- `(5)`

  Answer: `(6)`

7. Regarding Section 3.7, answer the following:

   (a) Explain how the semaphore-based barrier is able to ensure that no more than `N` threads are able to pass through the barrier. Make reference to semaphore values and calls to `wait` and `post`.

   (b) Illustrate with a concrete example how commenting out the line labeled `(A)` in Code Block 7 may potentially cause a *deadlock*

Bibliography

Kerrisk, M. (2025,). *clone(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/clone.2.html