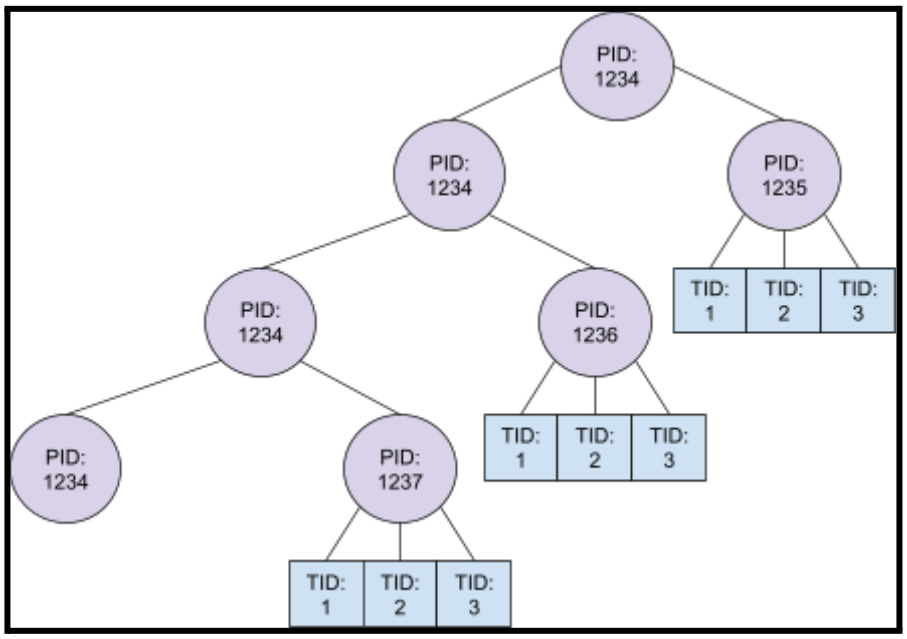
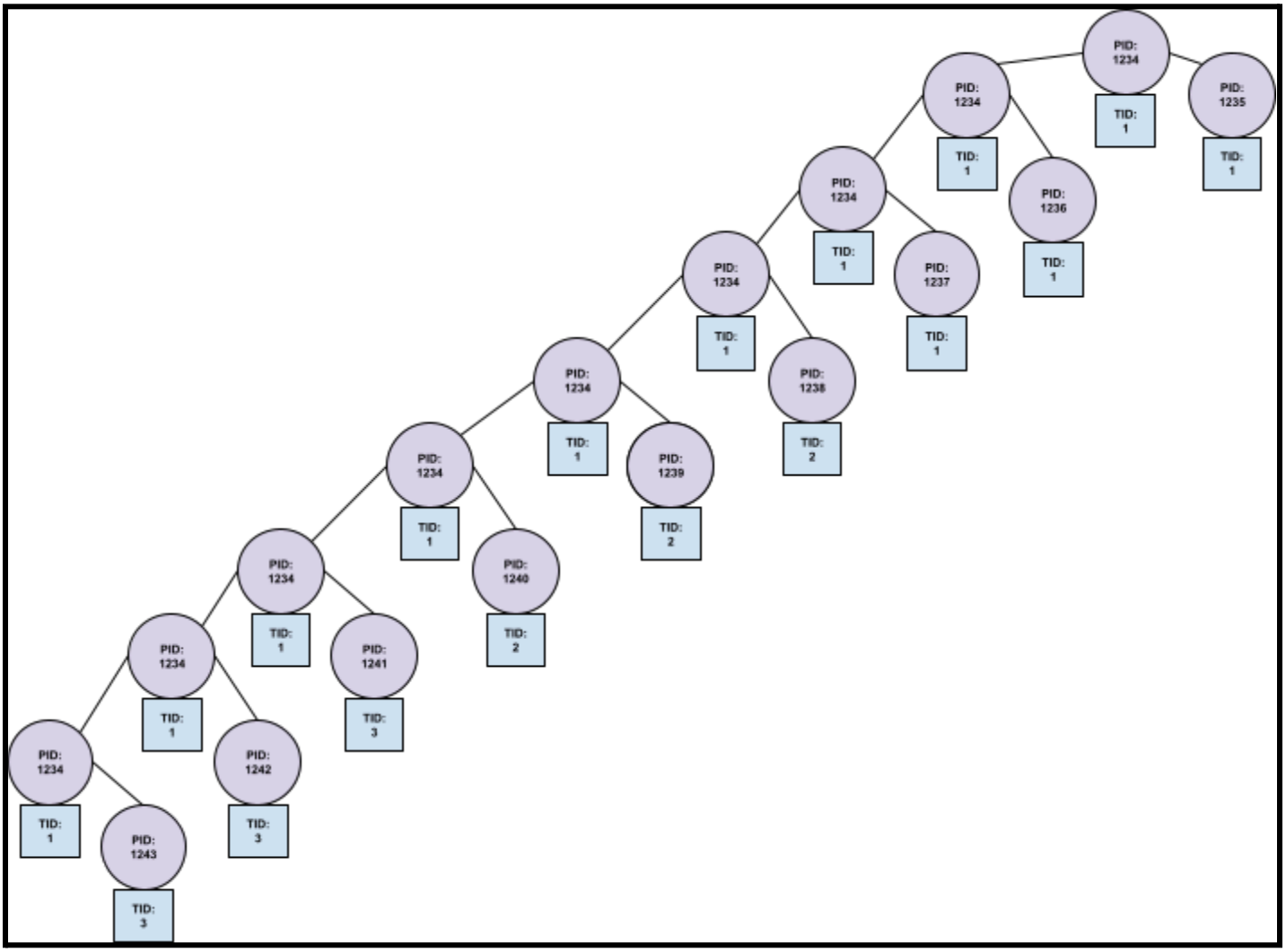
1a.

	global_var	static_var	local_var	dynamic_var
TID: 1	1	1	1	1
TID: 2	1	1	2	2
TID: 3	1	1	3	3



1b. assignment02_Duff_April_Question1.c

2a. 9 processes and 3 threads:



	global_var	static_var	local_var	dynamic_var
TID: 1	2	2	2	2
TID: 2	3	3	2	2
TID: 3	4	4	2	2

2b. assignment02_Duff_April_Question2b.c

2c. To give each thread its own instance of static_var, I moved its declaration to inside of *thread_function*. This way, its changes can no longer be seen or made by any unrelated processes or threads.

3a.

Output (in order)	Reason
Main process started	Main starts with a print statement indicating the start of the main process.
Thread 1 created	A thread is created with start routine *thread_function1* where a print statement executes indicating the creation of Thread 1.
Thread 2 created	A thread is created with start routine *thread_function2* where a print statement executes indicating the creation of Thread 2.
Thread 2 is done	Thread 2 finishes *thread_function2* by printing it is done and exiting the function.
Parent process continuing	*fork()* is called on the current process. Parent process does not have a pid of 0, so it prints the statement “Parent process continuing”.
Child process executing	The child process created by the *fork()* has a pid of 0, so it will print “Child process executing”
Main process is done	Main process executes a print statement that indicates it has finished.
total 40 *list of files in current directory*	Thread 1 executed *thread_function1*, which has the system call execvp(). The argument “ls -l” is passed into execvp() which then returns “ls -l” as if it were a command. This lists the current directory files in long format. This consists of the file name, owner, permissions, and modified date / time.

3b. The print statement indicating when Thread 1 is done needs to be before the *execvp()* call because anything after this system call will not execute. To fix this, *fork()* was used to have the child process run *execvp()* while the parent process prints whenever Thread 1 is done.

4. There will be five threads created. *local_var* is initialized to 42 and a for loop will execute to create the threads. If the thread creation fails, it will print the error. Each threads' start routine will be *thread_func* where the variables will undergo a for loop. This loop will increment each variable by one 10,000 times. After the for loop is finished, a print statement executes indicating that that thread has finished. It will also print the variables' results after each thread ends. The dynamic variable is then freed and we return back to main(). Because these threads are executing parallely, the global variables (*global_var* and *static_var*) will be building off of each thread taking turns. The variable *dynamic_var* stays the same since it is exclusive to each thread and is released after the for loop. Once all threads have completed and terminated, the main thread prints a statement containing the ending variables' results.

5. The output will be: alue = 1
alue = 2

When the two threads are created, both of their start routines are the function *func1*. This function increments the variable *value* by one, prints the result, then exits. The first thread is created and called into *func1* with *value* equaling zero. It will increment and print "alue = 1" and then a new line. The second thread is now created and called into the same function; however, *value* will now initially be equal to one since the first process called *funct1* already. It will increment this value of one and print "alue = 2" and a new line. This variable is global, which is why the second thread could see the changes made from the first thread.

6. The output will be: Thread1 (before fun1_1 call): Count value is = 11
Thread1 (inside fun1_1 call): Count value is = 13
Thread2: Count value is = 12
Thread1 (after fun1_1 call): Count value is = 13

Global variable *count* is initialized to 10. The first thread is created with its start routine being *fun1*, where *count* is incremented by one making it 11. This is printed with an indication that it is Thread1 before the *fun1_1* call. As Thread1 begins to execute *fun1_1*, Thread2 is created with its start routine being *fun2*. Like *fun1*, this function increments *count* by one and prints the result which would be 12. After printing this and indicating it is Thread2, it returns to main(). *fun1_1* is called where *count* is incremented by one again, now making the value 13. This is printed with an indication that it is Thread1 inside the *fun1_1* call. Thread1 returns to *fun1* and prints *count* value 13 with an indication that it is Thread1 after the *fun1_1* call.

7. Using Amdahl's Law: $speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$, where S is the serial portion and N is number of processors. So, $speedup \leq \frac{1}{0.5 + \frac{(1-0.5)}{4}} \leq 1.6$, where S = 0.5 and N = 4.

8. Using Amdahl's Law: $speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$, where S is the serial portion and N is number of processors. So, $speedup \leq \frac{1}{0.25 + \frac{(1-0.25)}{4}} \leq 2.28571$, where S = 0.25 and N = 4.