Reza Shisheie

2708062

CIS620

Project 1

login ID: reshishe

Objective:

This project can be divided into 2 sections:

1. Do copy on write for process and thread
2. Calculate matrix multiplication using threads

The objectives are

1. To compare the results results from COPY-ON-W using process and thread and do analysis.
2. To do the lapsed time of matrix multiplication using various number of threads and do analysis.

Function and code explanation:

1. crt.c

crt.c is main function to run pcrt ans tcrt. All the function prototypes which are used in main are added before main starts. These function prototypes are needed as there is no header file in the code and the compiler needs to know what the return value/type of each function is. Here is a list of all function prototypes in “crt.c”:

- void ep(int arg): the function in ep.c which executes the processes and show the lapsed time.
- void et(int arg): the function in et.c which executes the threads and show the lapsed time.
- void *update(void *data): the function in et.c which updates the heap at copy on write using pthread_create. ptherad_create needs a function with a void pointer as input and output to tun the thread using it. A void pointer is a special type of pointer that can be pointed at objects of any data type.

The main gets the number of arguments using “argc” and the actual value of arguments using “argv”. Right after entering to main a malloc function is invoked which saves the all input arguments into cells of array “new_argv”. The second input argument is taken as an integer using the following line: e.g. ./pcrt 1024

```
int arg1 = atoi(argv[1]);
```

This value holds the size of heap in KB. e.g. 1024 means to assign 1024 KB to heap. The first argument is either ./pcrt or ./tcrt in the new_argv[0]. By checking the third value which is either “p” or “t” I can decide to run the thread or process. I could check the whole argument and see if process or thread is requested. There is obviously no need to do that since if there is any other input the main code does not executed as the executable file does not exist after compilation.

Once the execution is over, malloc is freed to avoid memory leak.

2. ep.c:

ep.c executes the processes including the parent and child and does copy on write in the child process. Here is a list of all the header files and reason:

- #include <stdio.h> // to printf
- #include <sys/types.h> // to be used for pid_t, pthread_t, etc
- #include <sys/wait.h> // to be used for waitpid
- #include <stdlib.h> // to be used for calloc

In c language integers has a size of 32 bit or 4 bytes, so if I want to make a heap worth 1024 KB, the number of elements in calloc has to be 1024/4. This parameter is needed for the “calloc” function. Calloc function takes the number of elements that I want ot be created and the size of each element and generates space and initializes all of them to zero. If I have 1024/4 = 256 number of elements and each

element is 4 bytes then I am making a heap with 1024 Byte size. This heap can only hold 1024/4=256 elements and each element is 4 bytes.

Once calloc is assigned to a pointer “*a”, etime() is executed which initializes the timer. In the if-statement cpid is initialized by the return value of fork(). If cpid is 0, it means that the current process is “child process” and thus copy-on-write is implemented in the if-statement. It copies the value of each element of calloc and sum it to its element number and places that value back to malloc. The child process run it in a for loop for the size of elements in calloc. Once done pointer *a which is calloc is freed to avoid memory leak followed by an exit(0) which closes the child. If exit(0) does not exist child process will remain as a process and is not freed.

The same with child, in parent, I wait for the child to finish using waitpid(cpid, &status, 0) and status. waitpid system call is used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change can be child terminated, child was stopped, or child was resumed by a signal. If a wait is not performed, then the terminated child remains in a "zombie" state. If a child has already changed state, then these calls return immediately. Otherwise they block until either a child changes state or a signal handler interrupts the call. A combination of exit(0) in “child” process and waitpid in the “parent” process is to let the “parent” know that “child” task is completed and process is terminated so “parent” process can proceed. If not implemented correctly many child processes can be generated which can eat up the memory and cause slow down and eventual crash of OS.

Once child process returns “a” is freed to avoid memory leak. Then etime() is invoked again and the return value is placed in result and printed.

3. et.c:

like ep.c, et.c has include libraries. The only new one is the following:

- #include <pthread.h> // to be used for pthread_create

The only new function prototype is the following:

- void *update(int *data): is used for updating the created heap. Used in pthread_create

A global integer “int n” is defined before all functions which is used for the size of heap which will be used in the update function. This global variable is updated using the input argument of “et” function.

Before invoking pthread_create, etime() is invoked to save the statrt time. Then pthread_create is invoked using the thread ID (cpid), update function, and input argument pointer which points to the first element of calloc pointer. If ptherad_create is successful it returns 0 and if it is not successful it return an error number which can be looked up to see what the problem is. If the return value is not 0, then the printf inside the if statement gets implemented which says there is an error. The last argument in the pthread_create is used to track calloc in the update function and update element by element. In the update function all elements are updated to number 3.

Right after the if-statement there us pthread_join which waits for all threads to finish. What pthread_joint does, is that it cleans up the resources associate with the thread. Each child make its own stack once it is created and once it is joined that stack is deallocated.

Once all threads are done, calloc is freed to avoid memory leak. etime() is invoked again for the second time and result is saved and printed.

4. etime.c:

etime.c has a function float etime(void) inside which has a static struct to hold the start and stop time:

- static struct timeval start_time,stop_time;

One important header is the following which will be used by "gettimeofdat" function which returns sec and usec values:

- #include <sys/time.h>

Using static stuct, it saves the value in global section of code and saves it for next time running it. In this case, start time and stop time are saved every time code is ran.

Thus, every time code is run, it saves the stop time first. Then it get the elapsed time by subtracting the start time from stop time. And then initializes start time for next time invoke. If the start time is not initialized, the return value would be some garbage value. However, the next times etime() is invoked, both start and stop times have some values and thus the return value would be meaningful.

5. mm_thread.c:

This is the main function for the matrix multiplication. The only new function prototype is the following:

- void *Calculate(void *data): used to do calculation for each thread. This function is used after main is over and since it is after the main function, a prototype is needed. I could put it behind main and avoid function prototype.

In addition, global variables/array and constants are defined as follows:

- #define dim 160 // size of arrays
- int a [dim][dim]; // size of array a
- int b [dim][dim]; // size of array b
- int c [dim][dim]; // size of array c which is the result array : c = a * b
- int dx = 0; // number of rows assigned to each thread to multiply

pthread-create only allows one argument to be passed into the function. This can be a struct holding multiple values or just one value. For the sake of simplicity I set all these variables as global so I can access them easier in the “Calculate” function and there would be no need to pass them using struct and pointer. This is not the most elegant way but it works. It is not a good programming habit to define too many global variables.

Once I get into the “main” function in “mm_thr.c”, I initialize “a” and “b” arrays to “1” and “c” - which is the result of a and b multiplication - to “0”. I get the second argument of argv and using atoi to turn it to integer to be used as the number of threads:

. e.g. ./mm_bench 4 means 4 threads to do multiplication.

Using arg1, an array of thread IDs are defined:

```
pthread_t thread_id[arg1];
```

Integer “dx”, which is the number of rows for each thread, is calculated by dividing dimension (160) by number of threads (e.g. 4). Each thread multiply a portion of matrix “a” to “b”. The start row of each thread is defined by each element of “array”. I invoke pthread_create in a for loop and assign elements of “array” one by one to function “Calculate”.

Before the creation of threads in the for loop “etime()” is invoked to initialize the start and stop time. Once all threads were generated, pthread_join for all therads IDs is generated in a for loo to wait for all threads to be done. A thread would exit and terminated once its task is done.

Once all threads are executed and finished etime() is invoked again and elapsed time is saved is result and printed.

I assume the matrix multiplication is easy and there is not need to explain the calculate function. Just as an example, “./mm_bench 4” makes 4 threads. For the first thread it multiplies the first 40 row and 160 columns of “a” by 160 rows and columns of “b” and generates the first 40 row and 160 columns of “c”. After the whole process all elements of “c” should be 160.

6. Makefile:

In the makefile, in target “all” I defined 3 commands

1. make mm_bench: make executable mm_bench
2. make pcrt: make executable pcrt
3. ln -f pcrt tcrt: make hard link between pcrt and tcrt which means amje a copy of pcrt and save it under tcrt. -f means remove an existing link and replaces it.

The first one makes the target “mm_bench” from all the dependency object files in the proceeding lines. The first line which is :

```
mm_bench: mm_thr.o etime.o
```

is a linker which links all the object files and makes and executable in the target section. In the next two lines two object files are made out of the “mm_thr.c” and “etime.c”. Obviously the compilation starts from bottom and goes up. Thus the order of compilation is:

etime.o --> mm_thr.o -->mm_bench (executable)

target mm_bench has to compile first since etime.o is compiled and generated there. If etime.o is not generated, pcrt does not work.

Likewise, in the second line of target “all”, executable “pcrt” is made using “make pcrt”. Executable “pcrt” depends on the following object files:

1. crt.o
2. ep.o
3. et.o
4. etime.o

Thus all object files are made in the proceeding lines and executable “pcrt” is generated using linked using the following command:

pcrt: crt.o ep.o et.o etime.o

it links all the object files next to pcrt and and generates an executable using them.

The third line in target all is the following:

ln -f pcrt tcrt

which makes a hard link between pcrt and tcrt which is essentially making a copy of pcrt and save as tcrt. -f means remove an existing link and replaces it.

line is a clean target which cleans all the executable and object files if the following is entered on terminal:

rm pcrt tcrt mm_bench *.o

A few notes:

1. Note “gcc -c” makes as object file and does not link it to anything else. That is essential to use if separate compilation is used.
2. Also you need to use -pthread after gcc:

   gcc -pthread -o pcrt crt.o ep.o et.o etime.o

   The reason for it is that you using -ptherad after gcc tells the compiler to compiler the program using <ptherad.h> which enables the executable to run using threads.

7. run.sh:

I did not submit that file on “turnin” as it was no requested. This file is nothing but a shell script which executes bash commands which is executing all the commands that I actually need to run to get data. I put a sleep 0.1 which is 0.1 second sleep to make sure each command is executed and done before executing the next one. I got better results using this method. Here is just a piece of it for pcrt parts

```
#!/bin/bash
printf "pcrt 0 \n"
./pcrt 0
sleep 0.1
./pcrt 0
sleep 0.1
./pcrt 0
sleep 0.1

printf "\npcrt 1024 \n"
./pcrt 1024
sleep 0.1
./pcrt 1024
sleep 0.1
./pcrt 1024
sleep 0.1

printf "\npcrt 2048 \n"
./pcrt 2048
sleep 0.1
./pcrt 2048
sleep 0.1
./pcrt 2048
sleep 0.1

printf "\npcrt 4096 \n"
./pcrt 4096
sleep 0.1
./pcrt 4096
sleep 0.1
./pcrt 4096
sleep 0.1

printf "\npcrt 8092 \n"
./pcrt 8092
sleep 0.1
./pcrt 8092
sleep 0.1
./pcrt 8092
sleep 0.1
```

8. Debugging

One of the debugging issues was in the mm_thr.c function. Initially I made the following pthread_create call:

```
pthread_create( &thread_id[i], NULL, (void *)&Calculate, (void *)&prt
```

in which I am passing a pointer into the function “Calculate”. Pointer “prt” points to the begging row of matrix “a” for each thread. For instance if I am making 4 threads I would loop through it and assign 0, 40,80,120 as inputs to the “Calculate” function. Knowing the number of rows for each thread, which is 40 row in this example, the first thread starts from row 0 and does calculation to row 39. Then thread #2 starts from row 40 and does the calculation for 40 steps and tun to row 79 and etc.

Note there is no need for MUTEX to be used since these threads are running interdependently and are not accessing any parts of result matrix “c” simultaneously. Considering that I got segmentation error and core dump several times. The reason behind it was that I was passing the address of ptr to each thread while its value was being incremented by 40 in the loop to start next thread. Thus, all these thread were accessing a shared value without any protection. A protection here means MUTEX or a lock. Since there is not need for MUTEX – all thread are doing independent segment calculation – and MUTEX compromises speed, I made an array and put the initial row for each thread in it.

```
int array [arg1];
```

and then pass the correct element of pointer array to ptherad_create in the following:

```
pthread_create( &thread_id[i], NULL, (void *)&Calculate, (void *)&array[i]
```

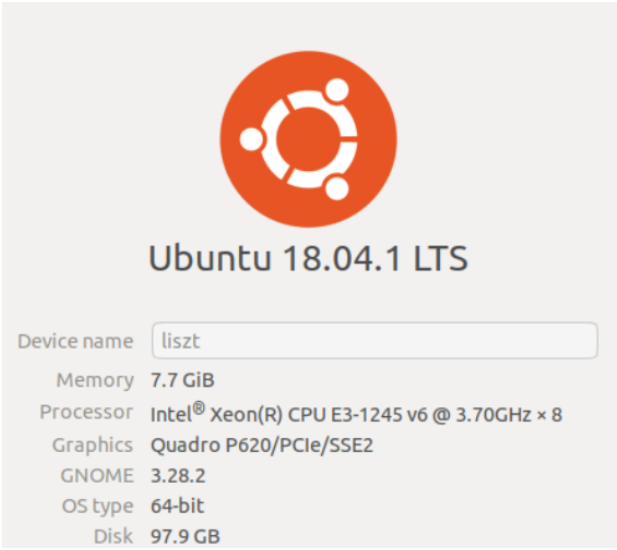
Using this technique, each thread is accessing its own element of the array and there would be no conflict between threads.

I also had difficulties using “pthread_create” as the input argument is the address of a void pointer. In “Update” and “Calculate” functions imported the void pointer as integer using casting. I tried other methods to import as void and then I cast it and I got into compilation troubles.

9. Results and Analysis:

9.1. Computer station specs:

Here is screenshot of the computer specs and Intel website shown below. As shown it has 4 core and 8 threads. The difference between cores and threads are explained in details in the following. But for now knowing the computer has 4 cores, my assumption is that we should get better performance moving from 1 process to 4 but programs with more that 4 processes are just slightly better.



Technical Specifications

Essentials

Vertical Segment	Server	Processor Number	E3-1245V6
Status	Launched	Launch Date	Q1'17
Lithography	14 nm		

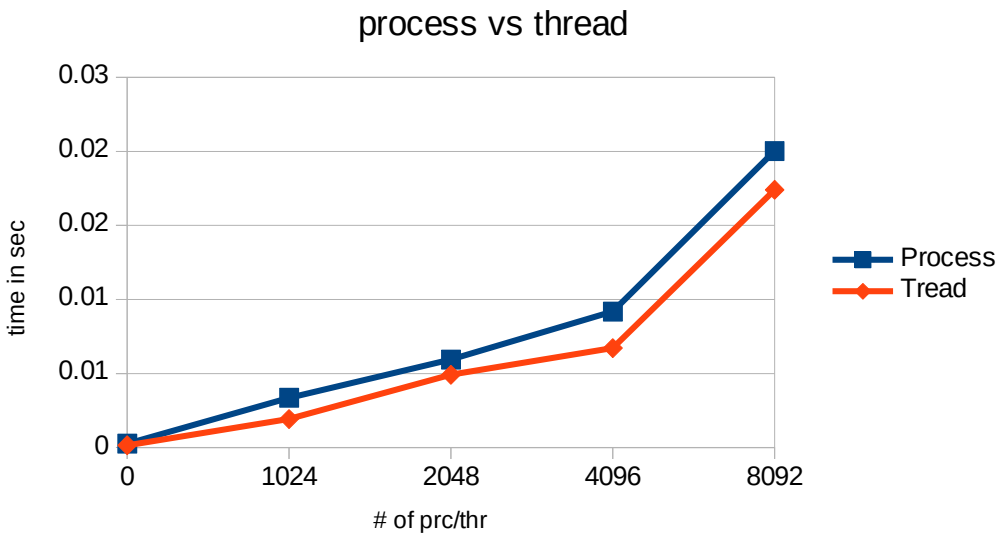
Performance

# of Cores	4	# of Threads	8
Processor Base Frequency	3.70 GHz	Max Turbo Frequency	4.10 GHz
Cache	8 MB SmartCache	Bus Speed	8 GT/s DMI3
TDP	73.0 W	VID Voltage Range	0.55V-1.52V

9.2. pcrt vs tcrt:

Running the shell “run.sh” will run each pcrt and tcrt 3 times for heap size 0,1024,2048,4096, and 8096 KB. Here is the table and graph to show how it is changing:

Heap size KB	0	1024	2048	4096	8092
Process time	0.000264	0.003357	0.005939	0.009182	0.020027
Tread tim	0.000148	0.001919	0.004911	0.006713	0.017409



As you can see the lapsed time is increasing as the heap size is increasing. However, the performance of threads are better all the time. In old UNIX, once you fork, a new process is made which has its own memory address including code, global variable, heap, and stack. In the new UNIX, all of them are made but not everything is copied. For reading a child process can read from parent’s memory address unless it wants to write. This is where COPY-ON-WRITE concept is important and it is implemented in the ep.c.

For thread, however, they don’t have their own memory address and each thread have only its own stack. Unlike process they don’t need to allocated a whole memory space and they can use shared memory. The only thing a new thread needs is a separate stack. And thus they run faster than process.

9.3. mm_thr:

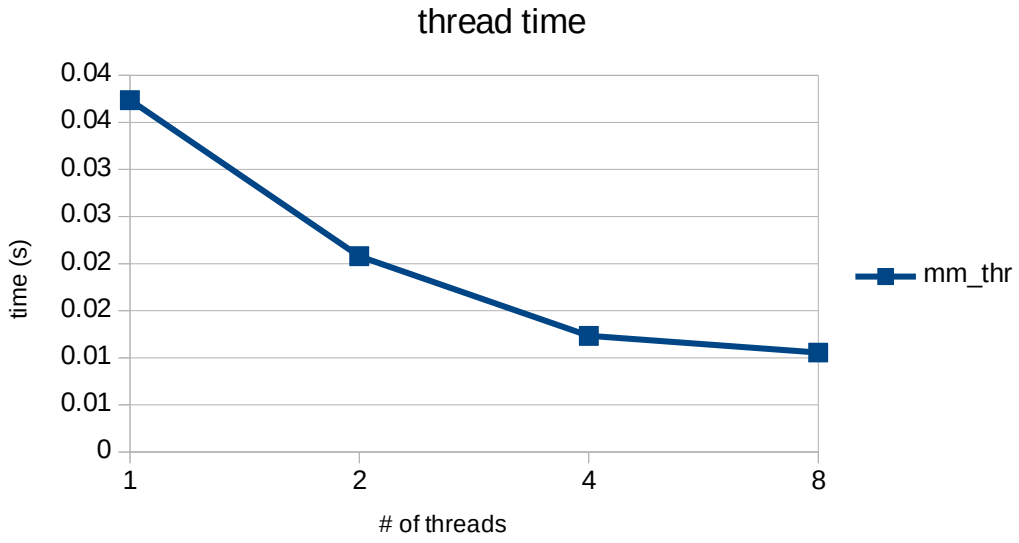
Before I start lets get familiar with two concepts in computer architecture as it will impact our performance. These two concepts are “Hyper threading” and “Multiple cores”.

Hyper-threading was introduced by Intel and its purpose is to bring parallel computation. A single physical CPU core with hyper-threading appears as two logical CPUs to an operating system. The CPU is still a single CPU, while the OS sees two CPUs for each core, the actual CPU hardware only has a single set of execution resources for each core. In other words, the OS is seeing two CPUs for each actual CPU core but actually there is only one actual/physical CPU core existing and that only core is the one that has ALU for real process execution. ALU is the unit that does all the computation and can be used by only one of the two virtual processes on the core. Hyper-threading allows the two logical CPU cores to share physical execution resources. This can speed things up somewhat, if one virtual CPU is stalled and waiting, the other virtual CPU can borrow its execution resources. Hyper-threading can help speed performance but it is not as good as having actual additional cores.

Unlike hyper-threading, there are no tricks on multiple cores. A dual-core CPU literally has two central processing units on the CPU chip with two ALUs. This helps dramatically improve performance. This means that the performance of multiple cores versus multiple thread should be much better.

Here are results for matrix multiplication on multiple threads:

# of therads	1	2	4	8
Time	0.037354	0.020781	0.012323	0.01055



As shown there is almost linear decrease in lapsed time as the number of threads are growing. However, the lapsed time of 8 threads is almost the same as 4 threads. In some runs, 8 threads showed worse results. The reason behind it is that: up to 4 threads, each thread is running on a separate core and thus the expectation is that time should be halved. Which is almost what t is: 37 – 20 – 12 msec.

Once you run 8 threads on a machine, every two thread are running on each core and thus they are competing with each other for resources.

This competition may result is slightly better results, if all threads are doing calculation symmetrically and finish all together and there is no lagging. It might also yield worse results if some threads are lagging or switching threads on each core takes too much time and thus the main has to wait for all threads to finish.

For 8 thread running I did some extra testing. Instead of running 8 threads together – in which each core has 2 thread – I divided them into two groups and I ran the first 4 threads first and joined them. Then I ran the second 4 threads and then joined. In this case I made sure there are only 4 threads running on 4 core all the time. The results of 8 threads running together was almost twice better than results from dividing which is another justification that running all threads together yields better results since it uses the resources of each core more efficiently.

9.4. Compilation

A few notes on compilation:

1. I use the best results from all the ones printed.
2. There were a few occasions where results were fluctuating and were not following the pattern that was expected. A few time it was caused by other users running remotely which can be found typing “w” on terminal. Even without other users on the machine, there might be some fluctuating as the machine itself is running threads and processes on the CPU too.

Here is a list of commands I used to run:

To compile shell:
    > chmod +x ./run.sh pcrt tcrt mm_bench
To remove all previously compiled files:
    > make clean
To make files:
    > make
To run the shell and get results:
    > ./run.sh

Here is a screen shot of my compilation and run: