COMP 3511 Operating System (Spring 2023)

Midterm Exam

Date: 03-April-2023 (Monday)

Time: 19:00 – 21:00 (2 hours)

Name (Write the name printed on your Student ID card)	
Student ID	
ITSC email	@connect.ust.hk

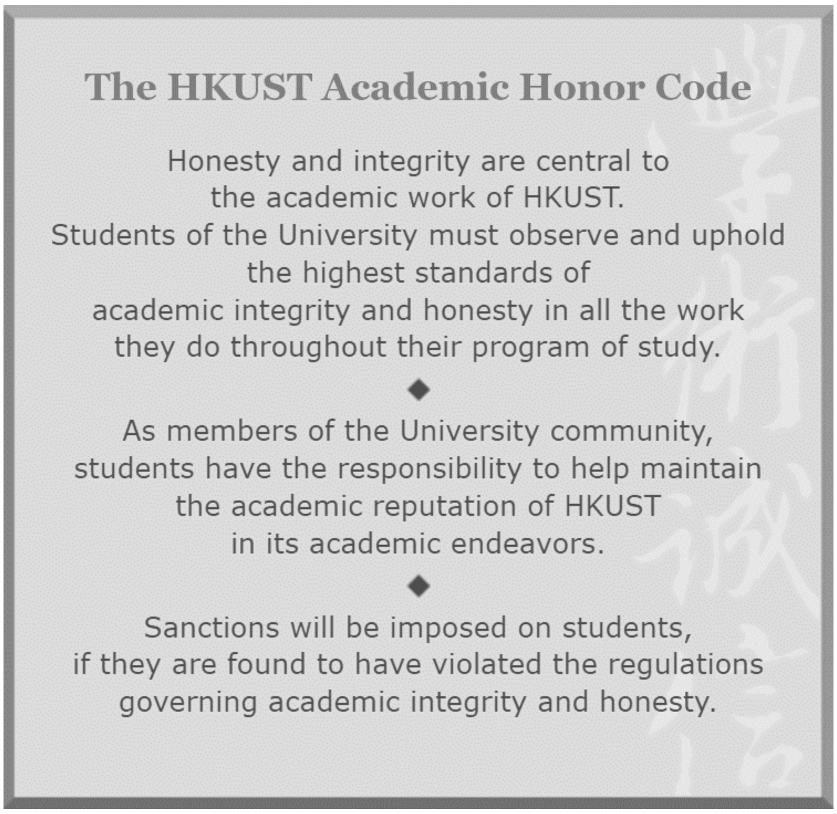
Exam format and rules:

- It is an open-book, open-notes exam (Reference: Chapter 1)
- Answer all questions within the space provided on the examination paper.
- The exam booklet is single-sided. You may use the back of the pages for your rough work. If you want to continue your answer, draw an arrow to point to the back page.
- Please read each question very carefully and answer the question clearly to the point.
- Make sure that your answers are neatly written, legible, and readable.
- An electronic calculator is allowed, but not other electronic devices (e.g., mobile phones, tablets, … are not allowed)

The score table used by the TAs:

Multiple Choices (25 points)	MC		
Q1: Process and Thread (25 points)	Q1.1 pthread	Q1.2 fork	Q1.3 syscalls
Q2: CPU scheduling (35 points)	Q2.1 Scheduling	Q2.2 MLFQ	Q2.3 RM/EDF
Q3: Synchronization (15 points)	Q3.1 Syn. Part 1	Q3.2 Syn. Part 2	
Total	**Total Score:** **/ 100**		

The HKUST Academic Honor Code


The HKUST Academic Honor Code

Honesty and integrity are central to
the academic work of HKUST.
Students of the University must observe and uphold
the highest standards of
academic integrity and honesty in all the work
they do throughout their program of study.

◆

As members of the University community,
students have the responsibility to help maintain
the academic reputation of HKUST
in its academic endeavors.

◆

Sanctions will be imposed on students,
if they are found to have violated the regulations
governing academic integrity and honesty.

Declaration of Academic Integrity

Your signature: ___________________________

I fully understand the HKUST Academic Honor Code and confirm to follow it during the exam.

Part I. Multiple Choices [25 points] – 1 point for each question

Write down your answers in the boxes below:

MC1	MC2	MC3	MC4	MC5	MC6	MC7	MC8	MC9	MC10

MC11	MC12	MC13	MC14	MC15	MC16	MC17	MC18	MC19	MC20

MC21	MC22	MC23	MC24	MC25

[Introduction]

MC1. Which of the following components is not a part of the Central Processing Unit in the Von Neumann architecture?

A. Control unit
B. Registers
C. Arithmetic Logic Unit
D. Main memory

MC2. Which of the following is true about the storage system?

A. Electrical storage is generally larger and less expensive per byte than mechanical storage.
B. Temporal locality in caching exploited the tendency of programs to access the same data item over and again within a small time duration.
C. Non-Uniform Memory Access is a major approach in system design that bypasses the CPU in the data transfer between the I/O device and memory.
D. In the caching process, the recently accessed data items tend to be copied from smaller to larger storage for sufficient storage capacity.

MC3. Which of the following statement is <u>false</u> about multiprocessor systems?

A. A multiprocessor system significantly saves power consumption compared to a single-processor system during the same time period.
B. In a typical symmetric multiprocessor system, the processors usually have their own local cache while sharing the system bus.
C. Contention for shared resources lowers the expected performance gain from additional processors in multiprocessor systems.
D. Multiprocessor system can be implemented with a single chip containing multiple computing cores.

[Operating System Structures]

MC4. Which of the following is <u>false</u> about operating system services?

A. OS services are a set of functions that are helpful to the user or ensure the system's efficient operation via resource management.
B. GUI interface is not a mandatory part of OS services.
C. To execute a C program, it's the OS's responsibility to compile the program, load it into memory and run it.
D. OS needs to be aware of possible errors and take appropriate actions to ensure correct and consistent computing.

MC5. During the execution of a system call, the following four events occur:

1. Returning to user mode.
2. Execution of the trap instruction.
3. Passing parameters to the system call.
4. Execution of the system call.

The correct order of execution is ____.

A. 2→3→1→4
B. 2→4→3→1
C. 3→2→4→1
D. 3→4→2→1

MC6. Which of the following statement is true about system calls?

A. The APIs are alternate versions of system calls as they can completely replace the system calls in the OS implementation.
B. Programs implemented with APIs can often achieve better performance compared to those implemented directly with system calls.
C. The number of parameters passed to system calls is limited if the stack method is used in parameter passing.
D. In the shared-memory interprocess communication model, processes use system calls to create and gain access to regions of memory owned by other processes.

MC7. Which of the following statement is true about the system structure?

A. Process scheduling function is often provided by the microkernel in a microkernel operating system.
B. The bi-directional dependency relationship between different layers allows more flexibility in a pure layered operating system.
C. The operating system Darwin is designed in the layered approach.
D. The performance of microkernels can greatly suffer due to the overhead of traversing through multiple layers to obtain an OS service.

[Processes]

MC8. Which of the following statements is true regarding the relationship between a process and a thread?

A. A thread is a lightweight version of a process, and each thread has its own thread control block.
B. A process is a collection of threads, each with its own address space.
C. A process can only have one thread, while a thread can be associated with multiple processes.
D. A process and a thread are identical concepts with different names used by different operating systems.

MC9. Which of the following statements about the pipe is true?

A. An ordinary pipe supports two-way interprocess communication.
B. The ordinary pipeline can only be accessed by one process at a time.
C. In most Unix systems, a named pipe will be automatically deleted by default when the last process using this pipe closes it.
D. A named pipe is a type of file in the file system in UNIX systems that multiple processes can access.

MC10. Which of the following is <u>true</u> about a C program's address space?

A. The program uses the stack to allocate and deallocate memory dynamically.
B. The text section contains the program's executable code.
C. The program counter stores the return address of a function call.
D. The data section is read-only and cannot be modified at runtime.

MC11. Which of the following lines is not included in the output of the following C code fragment?

(suppose all *fork()* are successful and necessary header files are included)

```
int a = -1, b = -1;
a = fork();
if (a == 0) b = fork();
a = a > 0 ? 1 : a;
b = b > 0 ? 1 : b;
printf("%d, %d\n", a, b);
fflush(stdout);
```

A. 1, 1
B. 1, -1
C. 0, 1
D. 0, 0

MC12. How many processes will be running concurrently after the *while()* loop? (suppose only one process runs before the loop, all *fork()* are successful, and necessary header files are included.). <u>Assume *n* is a non-negative integer</u>.

```
int i = n;
while (i--){
     if (fork()) fork();
}
```

A. 2^n (i.e., 2 to the power n)
B. 3^n
C. 4^n
D. $2n$ (i.e., 2 multiplies n)

[Threads]

MC13. Which of the following statements about *clone()* and *fork()* in Linux is false?

A. fork() creates an exact copy of the calling process, while *clone()* allows for more control over the creation of the new process or thread.
B. With the flag CLONE_FILES, *clone()* creates a child task that shares the file descriptors with the parent.
C. *fork()* only works in single-threaded programs, while *clone()* can be used in multi-threaded programs.
D. All of the above statements are correct.

MC14. Which of the following statements about user threads and kernel threads is true?

A. User threads are managed and scheduled with kernel support.
B. The operating system creates a thread control block for each user thread.
C. Each user thread maps to a kernel thread in the two-level multithreading model.
D. Blocking of one user thread does not influence the execution of other user threads.

MC15. Suppose an application is 60% parallel and 40% serial. According to Amdahl's Law, what is the theoretical minimum number of processors required to achieve a speedup of 1.95 times compared to single-core implementation?

A. 7
B. 6
C. 5
D. 4

[CPU Scheduling]

MC16. The convoy effect is associated with _____.

A. FCFS
B. SJF
C. Priority
D. Multi-level queue

MC17. Which of the following scheduling algorithms gives minimum average waiting time?

A. FCFS
B. Round Robin
C. Multi-level feedback queue
D. SJF

MC18. In multilevel feedback queue algorithm, which of the following statement is true?

A. Prior knowledge on the next CPU burst time is required for scheduling.
B. It handles interactive jobs well by delivering similar performance as SJF.
C. Classification of ready queue is permanent.
D. It won't suffer from starvation problem.

MC19. Which of the following is true of the rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling algorithms?

A. RM uses a dynamic priority policy.
B. Both RM and EDF are non-preemptive.
C. The rationale for RM is to assign a higher priority to tasks requiring CPU more often.
D. EDF uses a static priority policy.

MC20. We have 6 processes arriving at the ready queue at the same time. Their estimated runtimes are: 1, 2, 3, 4, 5 and 6 seconds. Their priorities are 3, 1, 4, 2, 5, and 4. A smaller number means higher priority (e.g., priority 1 has a higher priority than priority 2). Ignoring process switching overhead, calculate the average waiting time for priority scheduling.

A. 5.5s
B. 8.67s
C. 8s
D. 7.67s

[Synchronization]

MC21. Which of the following statements about race condition is true?

A. A race condition results when several threads try to access and modify the same data separately
B. A race condition results when several threads try to access the same data concurrently
C. A race condition will result only if the outcome of execution does not depend on the order in which instructions are executed
D. None of the above

MC22. Which of the following statements is false?

A. Spinlocks can be used to prevent busy waits in semaphore execution.
B. Two standard operations to modify Semaphore: wait() and signal().
C. Counting semaphores can be used to control access to resources with a limited number of instances.
D. If a semaphore value is negative, its magnitude is the number of processes currently waiting on the semaphore.

MC23: Which of the following is true about semaphore implementation with no busy waiting?____.

A. When a process executes wait() and finds the semaphore value is positive, it will suspend itself with block().
B. A suspended process waiting on the semaphore, may be restarted with wakeup() when some other process executes a signal() operation.
C. With wakeup(), it will remove one of processes in the waiting queue and place it in the waiting queue.
D. With block(), it will place the process invoking the operation on the ready queue.

MC24. Assume a semaphore S, initialized to 1, shared by several processes. Each process must execute wait(S) before entering the critical section and signal(S) afterward. Suppose a process executes in the following manner.

```
      signal(S);
        .....
/* critical section */
        .....
       wait(S);
```

Which of the following is true?

A. the critical section is well-protected.
B. it will get stuck, i.e., a deadlock will occur.
C. more than one processes may be executing in their critical section.
D. no process can enter critical section.

MC25: Three processes use 4 counting semaphores, initialized as S0 = 1, S1 = 0, S2 = 0, S3 = 0.

`/*Process P0*/`
`while(true) {`
`    wait(S0);`
`    print '1';`
`    release(S1);`
`    release(S2);`
`    release(S3);`
`}`
	`/*Process P1*/`
	`wait(S1);`
	`release(S0);`
		`/*Process P2*/`
		`wait(S2);`
		`release(S0);`
		`wait(S3);`
		`release(S0);`

How many times will '1' be printed out?

A. At least twice
B. Exactly twice
C. Exactly three times
D. Exactly four times

Part II. Calculations [75 points]

1. [25 Points] Process and Thread

Suppose all *fork()* are successful and necessary header files are included.

1) (6 points) Consider the following program.

```
void *child(void *arg) {
     int *counter = (int *) arg;
     (*counter)++;
     return NULL;
}
int main(int argc, char *argv[]) {
     int counter = 0;
     pthread_t thread;
     pthread_create(&thread, NULL, child, (void *)&counter);
     pthread_join(thread, NULL);
     printf("%d\n", counter);
     fflush(stdout);
     counter = 0;
     pid_t pid = fork();
     if(pid) wait(NULL);
     else{
          child((void *)&counter);
          exit(0);
     }
     printf("%d\n", counter);
     fflush(stdout);
     return 0;
}
```

What is the output of the program (4 points)?
Briefly explain your answer (2 points).

2) (9 points) Consider the following program.

```
void recursion(pid_t pid, int counter){
     if (counter == 0) return;
     if (pid) recursion(fork(), counter - 1);
     printf("%d\n", counter);
     fflush(stdout);
}
int main() {
    int n;
    scanf("%d", &n);
    recursion(fork(), n);
    return 0;
}
```

(a) Given an integer `n` (`n` > 0), what is the total number of processes (including the parent process and all the children processes created during execution) (2 points)? Please explain (1 point).

The total number of processes:
You should write your answer in terms of `n`.

Explanation of your answer:

(b) (6 points) How many times are each of the numbers "`1`", "`2`", and "`3`" printed when the input number `n`=1, 2, and 3, respectively? Write your answer in the following table. Some zeros are filled in the table.

Input value of the variable `n`	Number of "`1`"s in the output	Number of "`2`"s in the output	Number of "`3`"s in the output
1		0	0
2			0
3			

3) (10 points) First, we introduce the function `waitpid()`:

```
pid_t waitpid(pid_t pid, int *status_ptr, int options);
```

This function is similar to the `wait()` function. When the parameter `options=0`, it waits for the termination of the specific child process whose process ID is equal to the parameter `pid`.

In this question, the input comes from three files ("1.txt", "2.txt" and "3.txt") which exist in the same directory as the executable file. Each file includes two lines of input. The first line is a number indicating the length of the vector in the file. The second line is the vector, including multiple integers (See examples for details).

We intend to employ fork(), pipe(), and input redirection to compute the sum of all numbers in the vectors across the three files with multiprocessing.

Requirements:
1. You cannot use system calls except for `close` and `dup` (e.g., `dup2` can't be used).
2. You can write at most 20 characters in each blank.

Here are examples of reading/writing an integer from a file:

```
     read(file_id, address_of_int_variable, sizeof(int));
     write(file_id, address_of_int_variable, sizeof(int));
```

```
#define SIZE 3
char file_name[SIZE][10] = {"1.txt", "2.txt", "3.txt"};
int main() {
    int fd[SIZE - 1][2], i, j;
    pid_t pid[SIZE - 1];
    for(i = 0; i < SIZE - 1; i++) {
        pipe(fd[i]);
        pid[i] = fork(); // Record the pid of the child
        if(!pid[i]) break;
    }
    int fin = open(file_name[i], O_RDONLY, S_IRUSR | S_IWUSR);
    int length, number, sum = 0;
    ____Blank 1____; // Close the standard input
    ____Blank 2____; // Input redirecting
    scanf("%d", &length); // Input length of the vector
    for (j = 0; j < length; j++){
        scanf("%d", &number); // Input the vector
        ______________Blank 3______________;
    }
    if(!pid[i]) {
        close(fd[i][0]); // Close read end
        write(___Blank 4.1___, ___Blank4.2___, sizeof(int)); // Send data
        exit(0);
    }
    // the program continues on the next page
    for(i = 0; i < SIZE - 1; i++) {
        int status, child_sum;
        waitpid(pid[i], &status, 0); // Wait for child process
        close(fd[i][1]); // Close write end
        read(____Blank 5.1___,___Blank5.2___, sizeof(int)); // Receive data
        ________________Blank 6________________; // summing up
    }
    printf("Summation is %d.\n", sum);
    return 0;
}
```

Sample Input and Output:

1.txt	**2.txt**	**3.txt**	**Output on the console**

`4`
`20 10 30 20`
	`2`
	`40 30`
		`3`
		`20 10 50`
			`Summation is 230.`

Explanation of the sample: 230 = 20+10+30+20+40+30+20+10+50.
Please write your answers in the following table:

Blank1	
Blank2	
Blank3	
Blank4.1	
Blank4.2	
Blank5.1	
Blank5.2	
Blank6	

2. [35 Points] CPU Scheduling

1) (13 points) The following table contains 5 processes (P1-P5). Each process has its arrival time, burst time, and priority.

Process	**Arrival Time**	**Burst Time**	**Priority (Only used in Priority Scheduling)**
P_1	0	5	2
P_2	1	3	1
P_3	4	8	2
P_4	8	2	1
P_5	10	6	2

- Whenever there is a tie among processes (the same arrival time, remaining time, priority, etc), they are inserted into the ready queue in the ascending order of process id.
- A smaller number means a higher priority (e.g., priority 1 has a higher priority than priority 2)

For each of the following scheduling algorithms, draw the Gantt charts depicting the sequence of the process execution and calculate the average turnaround time.

1. FCFS
2. SJF (non-preemptive)
3. Preemptive priority scheduling with RR (with quantum of 2 milliseconds)
4. SRTF

- **FCFS** (3 points)

The Gantt chart:

The average turnaround time: ______________________________

- **SJF** (3 points)

The Gantt chart:

The average turnaround time: ________________________________

- **Preemptive priority scheduling with RR (TQ=2)** (4 points)

The Gantt chart:

The average turnaround time: ________________________________

- **SRJF** (3 points)

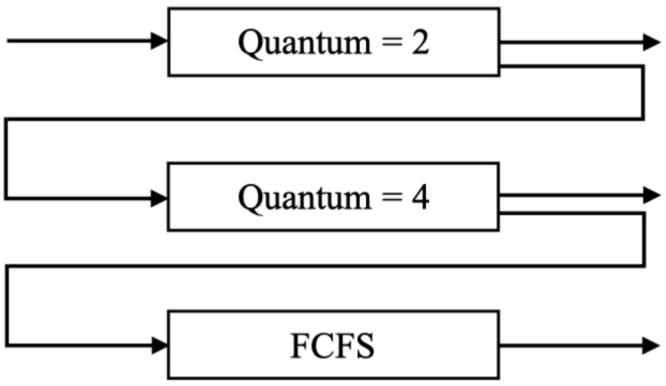
The Gantt chart:

The average turnaround time: ________________________________

2) (10 points) Consider the following single-thread processes, arrival times, burst times and the following three queues:

Q0 – RR with time quantum 2 milliseconds
Q1 – RR with time quantum 4 milliseconds
Q2 – FCFS

Process	Arrival Time	Burst Time
P_1	0	5
P_2	4	13
P_3	6	7
P_4	11	4
P_5	14	14
P_6	20	6



a) (6 points) Draw the Gantt chart depicting the scheduling procedures for these processes.

b) (4 points) Calculate the average waiting time.

3) (12 points) Consider the following single-thread processes, executing times, deadlines and periods. Assume all processes arrive at timeslot 0. Fill in the table with the ID of the process that is running on the CPU with Rate-Monotonic (RM) scheduling and Earliest Deadline First (EDF) scheduling in the first 20 timeslots, and show how many deadlines are missed in each scheduler.

Process	**Processing Time**	**Deadline**	**Period**
P_1	1	4	5
P_2	2	5	10
P_3	4	13	16
P_4	2	7	8

RM																				
EDF																				

Time 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

How many deadlines are missed for RM: ______________________________

How many deadlines are missed for EDF: ______________________________

3. [15 points] Synchronization

1) (5 points) Suppose two threads execute the following C code concurrently and access shared variables a, b, and c.

Initialization:
```
int a = 1;
int b = 3;
int c = 0;
```

// Thread 1	// Thread 2

`a = b + 1`
`c = a - b`
	`a = 13`
	`b = 10`
	`c = 2`

What are <u>all the possible values</u> for `c` after both threads complete?

You can assume that readings and writings of the variables are atomic, and that the order of execution of statements within each thread is preserved by the C compiler and the hardware, so it matches the code above.

2) (10 points) Barrier: A common parallel programming pattern is to perform processing in a sequence of parallel stages: all threads work independently during each stage, but they must synchronize at the end of each stage at a synchronization point called a ***barrier***. If a thread reaches the barrier before all other threads have arrived, it waits. When all threads reach the barrier, they are notified and can begin the execution on the next phase of the computation. For example:

```
while (true) {
    Compute stuff;
    BARRIER();
    Read other threads results;
}
```

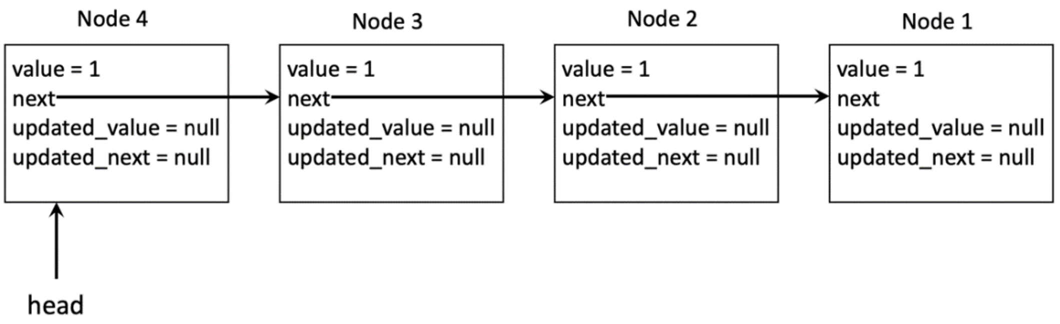
a) (4 points) The following implementation of Barrier is incomplete and has four lines missing. Fill in the missing lines so that the Barrier works according to the prior specifications. Hint: Here are the possible answers for the blanks:

```
CV.signal()  CV.signalAll()  CV.wait()   L.acquire()  L.release()
```

```
class Barrier {
private:
    int numWaiting {0}; // Initially, no one at barrier
    int numExpected {0}; // Initially, no one expected
    Lock L;
    ConditionVar CV;
public:
    void threadCreated() {
        L.acquire();
        numExpected++;
        L.release();
    }
    void enterBarrier() {
        ______Blank1______;
        numWaiting++;
        if (numExpected == numWaiting) { // If we are the last
            numWaiting = 0; // Reset barrier and wake threads
            ____Blank2___;
        } else { // Else, put me to sleep
           ____Blank3___;
        }
        ____Blank4___;
   }
}
```

Blank1	
Blank2	
Blank3	
Blank4	

b) (6 points) Now, let us use the above Barrier implementation in a parallel algorithm. Consider the linked list below:



In our parallel algorithm, there are four threads (Thread 1, Thread 2, Thread 3, Thread 4). Each thread has its own instance variable ***node***, and all threads share the class variable ***barrier***. Initially, Thread 1's ***node*** references Node 1, Thread 2's ***node*** references Node 2, Thread 3's ***node*** references Node 3, and Thread 4's ***node*** references Node 4.
In the initialization steps, **barrier.threadCreated()** is called once for each thread created, so we have **barrier.numExpected == 4** as a starting condition.
Once all four threads are initialized, each thread calls its **run()** method. The **run()** method is identical for all threads:

```
void run() {
    bool should_print = true;
    while (true) {
        if (node.next != NULL) {
            node.updated_value = node.value + node.next->value;
            node.updated_next = node.next->next;
        } else if (should_print) {
            std::cout << node.value;
            should_print = false;
        }
        barrier.enterBarrier();
        node.value = node.updated_value;
        node.next = node.updated_next;
        barrier.enterBarrier();
    }
}
```

List all the values that are printed to stdout along with the thread that prints each value. For example, "thread 1 prints xxx".

Thread 1 prints ____________________

Thread 2 prints _____________________

Thread 3 prints _____________________

Thread 4 prints _____________________