ID 2069997

Exam for 30203 and 30421 - Systems Programming in C/C++

Question 1

(a)
There will not be memory leakage in the program. In line 3 memory is allocated, with the address of the allocated memory stored in integer pointer A.
Although the value that A points to is modified in line 4, the memory address is not changed. A still contains the address of the allocated memory.
Line 5 initializes the integer pointer B. A is unchanged.
In line 6 this same memory address is stored in integer pointer B by assigning B = A.
Therefore B contains the memory address of the memory allocated in line 3.
The free(int *) function frees the allocated memory with the address that is passed to it. By free(B) in line 7, the allocated memory allocated in line 3 is freed.
There are no other allocated memories. Therefore there will not be a memory leakage.

(b)
This function returns an integer pointer pointing to unassigned values.
When this function is called, the array values are stored in stack memory and only has scope within this function. After returning the address of the start of the array (line 6), the function terminates, and the array values are deleted.
A fix could be achieved by storing the array in heap using the malloc() function.

```
int* randomArray(void)
{
        int * array = malloc(sizeof(int) * 10);
        for (int i = 0; i < 10; i++)
                array[i] = rand();
        return array;
}
```

Question 1 (cont.)
(c)
Sum2Darray2 is able to achieve faster computation time.
In the function sum2Darray1, data is fetched with column-major order, while the function sum2Darray2 fetches data in row-major order.

When loading values from the main memory to the cache, values stored in nearby memory addresses will also be loaded into the cache to decrease the chance of the need to load values from the main memory, which is much slower.

The C compiler stores 2D arrays in row-major order.
By fetching data in row-major order, the same order as how values are stored, a higher rate of cache hit can be achieved when running sum2Darray2. This results in a higher CPU utilization rate and therefore faster computation.

Question 2
(a)

```
//Process P1
void P1(void){
        wait(B);
        Statement A
        signal(A);
}
```

```
//Process P2
void P2(void){
        Statement B
        signal (B);

}
```

```
//Process P3
void P3(void){
        wait(A);
        Statement C
}
```

Trace:

```
wait(A);
wait(B);
Statement B
signal (B);
Statement A
signal(A);
Statement C
```

(b)
Race condition appears. A pointer pointing to i is passed to the threads, instead of a copy of the current value of i. Therefore, multiple threads are accessing a value in the same memory location. When a thread is created, the main thread continues to run, changes the value of i and creates a new thread. If this is done before the printf() function in the earlier thread, the pointer passed into the printf() function points to the updated value, and the program has an incorrect output.

Question 2 (cont.)
(c)

```
int main(){
        pthread_t tid[4];
        int i;
        int t[4] = {0,1,2,3};

        for(i=0; i<4; i++)
                pthread create(&tid[i], NULL, foo, &t[i]);

        for(i=0; i<4; i++)
                pthread join(tid[i], NULL);

        return 0;
}
```

By assigning the values into an array, pointers pointing to different memory locations are passed into different threads. The values in the array t are also unmodified when creating new threads. This avoids race condition.

Question 3
(a)
There is only one possible output:

```
2
3
```

Explanation: line 6 initializes integer x = 1.
Line 7 uses the function fork() and creates a child process.
As the child process and parent process are 2 separate processes, they have their own values of x and don't interfere.
The child process, with pid == 0, executes line 8 and x = 2. The result "2" is printed on screen with a new line after it and the child process terminates.
The parent process, with pid > 0, executes line 11 and 12.
With wait(NULL) in line 11, the parent process is blocked until the child process is terminated, that is, "2" is printed on screen with a new line after it. x is assigned to be 3 and is printed out with a new line. Parent process terminates.

Question 3 (cont.)
(b)
Total turnaround time = 106

0	P2(32)	P2 arrives and gets processed
3	P2(29), P0(20)	P0 arrives
5	P2(27), P0(20), P3(3)	P3 arrives
10	P0(20), P3(3), P2(22)	Quantum time expires, P2 is forced out of CPU. P0 gets processed.
15	P0(15), P3(3), P2(22), P1(22)	P1 arrives
20	P3(3), P2(22), P1(22), P0(10)	Quantum time expires, P0 is forced out of CPU. P3 gets processed.
23	P2(22), P1(22), P0(10)	P3 gets completed, P2 gets processed.
33	P1(22), P0(10), P2(12)	Quantum time expires, P2 is forced out of CPU. P1 gets processed.
43	P0(10), P2(12), P1(12)	Quantum time expires, P1 is forced out of CPU. P0 gets processed.
49	P0(4), P2(12), P1(12), P4(29)	P4 arrives
53	P2(12), P1(12), P4(29)	P0 gets completed, P2 gets processed.
63	P1(12), P4(29), P2(2)	Quantum time expires, P2 is forced out of CPU. P1 gets processed.
73	P4(29), P2(2), P1(2)	Quantum time expires, P1 is forced out of CPU. P4 gets processed.
83	P2(2), P1(2), P4(19)	Quantum time expires, P4 is forced out of CPU. P2 gets processed.
85	P1(2), P4(19)	P2 gets completed, P1 gets processed.
87	P4(19)	P1 gets completed, P4 gets processed.
97	P4(9)	Quantum time expires, but no other processes are in queue. P4 gets processed.
106		P4 gets processed.

(c)
(i) 20495
(ii) invalid
(iii) invalid
(iv) 16385

Question 3 (cont.)

(d)

FIFO: 8 page faults

0	1	2	3	4	2	3	0	1	4	2
0	0	0	0	4	4	4	4	4	4	4
	1	1	1	1	1	1	0	0	0	0
		2	2	2	2	2	2	1	1	1
			3	3	3	3	3	3	3	2

LRU: 9 page faults

0	1	2	3	4	2	3	0	1	4	2
0	0	0	0	4	4	4	4	1	1	1
	1	1	1	1	1	1	0	0	0	0
		2	2	2	2	2	2	2	4	4
			3	3	3	3	3	3	3	2

It may seem weird to see that LRU has more page faults than FIFO as it is generally the better algorithm. This is mainly due to the fact that the page accesses has shown little repetition.

(e)

In scenario 1, both CPU utilization and disk utilization is low, which indicates that the CPU is possibly bounded by IO bound processes and there is a lot of waiting time. CPU bound processes should be scheduled, if there is any, when the CPU is waiting for IO response.

In scenario 2, CPU utilization is low but the paging disk utilization is high. Thrashing occurs, and the operating system should consider allowing the current process to use more memory space.

In scenario 3, CPU utilization is high but the paging disk utilization is low. This is good, and no additional measures has to be taken.

Do not write below this line