CS 537: Intro to Operating Systems (Fall 2017)
Worksheet 7 - Thread and Concurrent Data Structures
This worksheet is only for practice and will NOT be graded.

1. Threads vs Processes!

Part I: Assume that the code snippet below compiles successfully, all the APIs like `pthread_create()` do not fail, and the values in the `malloc`ed memory are all initialized to 0.

```c
void worker(int *balance) {
    int *counter = malloc(sizeof(int));
    for (int i = 0; i < 1000000; i++) {
        ( *balance )++;
        ( *counter )++;
    }
    printf("balance : val %d, addr %p\n", *balance, balance);
    printf("counter : val %d, addr %p\n", *counter, counter);
}

int main() {
    int *balance;
    balance = malloc(sizeof(int));
    pthread_t t[2];
    for (int i = 0; i < 2; i++) // Creating new threads
        pthread_create(&t[i], NULL, worker, balance);
    for (int i = 0; i < 2; i++)
        pthread_join(t[i], NULL);
}
```

a. What are the values of the 2 variables (balance and counter) after the 2 threads (t1 and t2) finish execution? Value here means the contents printed using the following print statements. If the value of a variable may be different in different runs of the program, you should write N/A.

```
printf("%d\n", *balance);  printf("%d\n", *counter);
```

<table>
<thead>
<tr>
<th>Value</th>
<th>t1</th>
<th>t2</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>counter</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
b. Consider the Virtual Addresses (VA) printed using the following print statements in the 2 threads (t1 and t2). PA stands for Physical Address.

\[
\begin{align*}
\text{printf("%p
", balance); & \quad \text{printf("%p
", counter);} \\
i. VA \text{ of balance in } t1 \text{ == VA of balance in } t2? \ (\text{TRUE / FALSE}) \\
ii. VA \text{ of counter in } t1 \text{ == VA of counter in } t2? \ (\text{TRUE / FALSE}) \\
iii. PA \text{ of balance in } t1 \text{ == PA of balance in } t2? \ (\text{TRUE / FALSE}) \\
iv. PA \text{ of counter in } t1 \text{ == PA of counter in } t2? \ (\text{TRUE / FALSE})
\end{align*}
\]

**Part II:** Now assume that the code given below compiles successfully, all the APIs like fork() do not fail, and the values in the malloc’ed memory are all initialized to 0.

```c
void worker(int *balance) {
    int *counter = malloc(sizeof(int));
    for (int i = 0; i < 1000000; i++) {
        (*balance)++;
        (*counter)++;
    }
    printf("balance : val %d, addr %p\n", *balance, balance);
    printf("counter : val %d, addr %p\n", *counter, counter);
}

int main() {
    int *balance;
    balance = malloc(sizeof(int));
    for (int i = 0; i < 2; i++) { // Creating new processes
        if (fork() == 0) {
            worker(balance);
            exit(0);
        }
    }
    for (int i = 0; i < 2; i++)
        wait(NULL);
}
```

c. What are the values of the 2 variables (balance and counter) after the 2 processes (p1 and p2) created using fork() finish execution? Value here means the contents printed using the following print statements. If the value of a variable may be different in different runs of the program, you should write N/A.

\[
\begin{align*}
\text{printf("%d\n", *balance); & \quad \text{printf("%d\n", *counter);}
\end{align*}
\]
d. Consider the **Virtual Addresses (VA)** printed using the following print statements in the 2 processes (p1 and p2). **PA** stands for **Physical Address**.

\[
\text{printf("%p\n", balance); printf("%p\n", counter);}\]

i. VA of balance in p1 == VA of balance in p2? (TRUE / FALSE)
ii. VA of counter in p1 == VA of counter in p2? (TRUE / FALSE)
iii. PA of balance in p1 == PA of balance in p2? (TRUE / FALSE)
iv. PA of counter in p1 == PA of counter in p2? (TRUE / FALSE)
2. Locked Data Structures

Assume you have the following code for removing the head of a shared linked list. Assume each line is performed atomically. Assume a list $L$ originally contains nodes with keys 1, 2, 3 and 4. Now there are two threads $T$ and $S$ that are popping the list concurrently.

```c
typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
} list_t;

int pop(list_t *L) {
    if (!L->head) return -1; // line 1
    int rkey = L->head->key; // line 2
    L->head = L->head->next; // line 3
    return rkey; // line 4
}
```

a. Given the following sequences, fill in the results. The sequence contains $T_i$ and $S_j$, designating that the $i$th line of code was scheduled for thread $T$ and the $j$th line of code was scheduled for thread $S$ respectively. For example, a sequence of $T_1T_2T_3S_1S_2$ indicates that 3 lines (lines 1, 2, and 3) were run from thread $T$ followed by 2 lines (lines 1 and 2) from thread $S$. You should assume that each sequence is executed independently. In other words, the state of the linked list is the same (with 4 nodes) at the start of each sequence. The right most column in the table below represents the value of $L->head->key$ at the end of the sequence.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>rkey from $T$</th>
<th>rkey from $S$</th>
<th>$L-&gt;head-&gt;key$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1T_2S_1S_2S_3S_4T_3T_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_1T_2T_3S_1S_2S_3S_4T_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_1T_2S_1S_2T_3T_4S_3S_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_1S_1S_2S_3T_2T_3T_4S_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b. In the pop() method given above, which line(s) of code form the critical section? Our goal here is to maximize the concurrency among threads that are trying to pop from this shared linked list.