CS-202 Exercises on Parallelism & Concurrency (L18)

Exercise 1: Parallelism vs. Concurrency [Basic]

1. For each scenario below, explain whether it is an example of parallelism, concurrency, or neither. In some scenarios, there may be more than one correct explanation at different levels of detail (e.g., components A and B run *concurrently*, and component C runs *in parallel* to both).
   - Two cooks work at the same time in a kitchen preparing different dishes. *[Parallelism]*
   - One cook prepares a dish while switching between chopping vegetables and stirring a pot. *[Concurrency] (The cook performs the two tasks concurrently. The pot cooks the disk in parallel to the cook.)*
   - A multi-core processor runs two threads simultaneously on separate cores. *[Parallelism]*
   - A single-core CPU switches rapidly between two threads to handle user input and a background task. *[Concurrency]*
   - A program performs multiple mathematical computations in a loop. *[Neither]*
   - A server handles multiple client connections using asynchronous I/O. *[Concurrency] (The server handles connections concurrently. The I/O device runs in parallel to the server.)*
   - A teacher grading one exam, then another, then another, one after the other. *[Neither]*
   - A team of movers each carries boxes at the same time to speed up the move. *[Parallelism]*

2. Consider the scenarios described below, composed of tasks lists with time estimates. For each scenario, explain how parallelism and concurrency can help. Calculate the end-to-end time for the scenario to be accomplished sequentially, with concurrency, and with parallelism (using as many people in parallel as you see fit).

Scenario 1: Preparing dinner

Task	Description	Estimated Time
A	Chop vegetables	10 minutes
B	Boil pasta	12 minutes
C	Set the table	5 minutes
D	Make salad dressing	7 minutes
E	Clean up prep area	6 minutes

Solution:
Sequential: 10 + 12 + 5 + 7 + 6 = 40 minutes
Concurrent:

- *Boiling can be done in the background. Everything else is sequential.*
- *10 + 5 + 7 + 6 = 28 minutes*

Parallel:

- *2 people working in parallel. Boiling still in the background. Cleaning can only be done at the end.*
- *Person1: chop vegetables (10 minutes).*
- *Person2: prepare dressing and set table (12 minutes).*
- *12 + 6 = 18 minutes,*

Scenario 2: Moving out of an apartment

Task	Description	Estimated Time
A	Pack clothes	15 minutes
B	Disassemble furniture	20 minutes
C	Load truck	25 minutes

D	Sweep floors	10 minutes
E	Return keys	5 minutes

Solution:
Sequential: 15 + 20 + 25 + 10 + 5 = 75 minutes
Concurrent:

- *No tasks can be alternated to improve time, still 75 minutes.*

Parallel:

- *2 people working in parallel.*
- *A and B can happen in parallel. + 20 minutes,*
- *Loading the truck needs to be done after A and B are done. + 25 minutes.*
- *D and E can happen at the same time while loading the truck.*
- *Total: 20 + 25 = 45 minutes.*

Scenario 3: Office Morning Routine

Task	Description	Estimated Time
A	Unlock doors and disable alarm	3 minutes
B	Brew coffee using the machine	20 minutes
C	Boot all computers	7 minutes
D	Check voicemails and emails	10 minutes
E	Refill printer paper	5 minutes

Solution:
Sequential: 3 + 20 + 7 + 10 + 5 = 45 minutes
Concurrent:

- *A needs to be done first. + 3 minutes.*
- *Assuming that the coffee machine takes 1 minute to turn on, and brewing is passive. +1 minute.*
- *Booting computer needs to happen before D, paper can be refilled while computers turn on. +7 minutes.*
- *D has to be at the end. + 10 minutes.*
- *3 minutes left for coffee brewing.*

- *Total: 3 + 1 + 7 + 10 + 3 = 24 minutes.*

Parallel:

- *A needs to be done first. + 3 minutes.*
- *All other tasks can be done in parallel (C and D are done sequentially, for a total minutes), so time is 20 minutes (defined by B, the longest task).*

Exercise 2: Variables [Advanced]

Consider the following code that reverses sentences in place.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define MAX_SENTENCE_LEN 256
#define NUM_SENTENCES 3

typedef struct {
   char* sentence;
} ThreadArg;

// Runs in each worker thread
void* reverse_sentences(void* arg) {
   ThreadArg* data = (ThreadArg*)arg;
   char* sentence = data->sentence;
   int sentence_len = strlen(sentence);

   for (int i = 0; i < sentence_len / 2; i++) {
       char temp = sentence[i];
       sentence[i] = sentence[sentence_len - i - 1];
       sentence[sentence_len - i - 1] = temp;
   }

   return NULL;
}

// Runs in the main thread
int main() {
   const char* original_sentences[NUM_SENTENCES] = {
       "Liechtenstein is a long word to reverse",
       "A bad beginning makes a bad ending",
       "A dog! A panic in a pagoda!"
   };

   pthread_t threads[NUM_SENTENCES];
   ThreadArg args[NUM_SENTENCES];

   // Allocate and prepare each sentence
   for (int i = 0; i < NUM_SENTENCES; i++) {
       args[i].sentence = malloc(MAX_SENTENCE_LEN);
       if (args[i].sentence == NULL) {
           (...)
       }
       strncpy(args[i].sentence, original_sentences[i], MAX_SENTENCE_LEN - 1);
       args[i].sentence[MAX_SENTENCE_LEN - 1] = '\0';

       if (pthread_create(&threads[i], NULL, reverse_sentences, &args[i]) != 0) {
           (...)
       }
   }
```

```
    // Wait for all threads to finish
    for (int i = 0; i < NUM_SENTENCES; i++) {
        pthread_join(threads[i], NULL);
    }

    for (int i = 0; i < NUM_SENTENCES; i++) {
        printf("%s\n", args[i].sentence);
        free(args[i].sentence);
    }

    return 0;
}
```

For each of the variables listed below, say where they stay in memory, and which thread (main or worker thread) is their owner.

`Original_sentences, "A dog! A panic in a pagoda",threads, args, args[0].sentence, sentence, temp.`

Solution:

Main

Stack	Heap	Text/data segment
original_sentences [array of pointers] *args* *threads*	*args[0].sentence [allocated with malloc]*	*A dog! (...) [string literal]*

Thread

Stack	Heap	Text/data segment
sentence [points to heap] *temp*		

Exercise 3: Data races - shared counter [Advanced]

Consider the following program, which uses 5 threads to increment a shared counter. Identify all values of the counter the program may print. For each such value, write down a concrete execution schedule that produces the value.

```c
#include <stdio.h>
#include <pthread.h>
int counter = 0;

void *incr(void *arg) {
  counter = counter + 1;
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t threads[5];
  // Create two threads T1 and T2
  for (int i=0; i < 5; i++) {
     pthread_create(&threads[i], NULL, incr, NULL);
  }
  for (int i=0; i < 5; i++) {
     pthread_join(threads[i], NULL);
  }
  printf("Counter: %d\n", counter);
  return 0;
}
```

Solution:

Two main cases:

1. *No interference: counter will be 5.*
    a. *Thread 1: counter 0->1*
    b. *Thread 2: counter 1->2*
    c. *Thread 3: counter 2->3*
    d. *Thread 4: counter 3->4*
    e. *Thread 5: counter 4->5*
2. *With interference: counter can be any of 1,2,3,4.*
    - *Example: counter prints 3 (other results follow the same logic)*
    1. *Thread 1 & Thread 2 read at the same time: counter 0->1 (one increment is lost)*
    2. *Thread 4: counter 1->2*
    3. *Thread 3 & Thread 5 read at the same time: counter 2->3 (one increment is lost)*

Exercise 3: Data races - dining philosophers

Remember the dining philosophers problem from the lecture.

The dining philosophers problem

- Philosophers think, eat, think, eat…
- They need both forks to eat
- Should not try to grab the same fork simultaneously

And consider the following bogus “solution”, presented as the first candidate during the lecture.

```
/* problem setup */

void philosopher(size_t i) {
   while (true) {             // repeat forever
       think(i);
       take_forks(i);         // acquire two forks
       eat(i);                // yum-yum, spaghetti
       put_forks(i);          // put both forks back
   }
}

int main() {
   pthread_t t1, t2, t3, t4, t5;

   pthread_create(&t1, NULL, philosopher, 1);
   pthread_create(&t2, NULL, philosopher, 2);
   ...
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   ...
}
```

```
/* solution */

bool fork_taken[5] = {false};

void take_forks(i) {
   while (fork_taken[i] || fork_taken[(i + 1) % 5])
{
       // wait until both forks are free
   }
   // take fork to the left
   fork_taken[i] = true;
   // take fork to the right
   fork_taken[(i + 1) % 5] = true;
}

void put_forks(i) {
   // put down the fork to the left
   fork_taken[i] = false;
   // put down fork to the right
   fork_taken[(i + 1) % 5] = false;
}
```

1. Explain, in a few sentences, what causes a data race in this solution, and what problem the data race may lead to.

Solution:
A data race happens in this example because a shared array is accessed and modified by multiple threads without synchronization, and since the operations are not atomic, two philosophers may simultaneously see both forks as available and proceed to set the same fork as taken.

2. Data races are problematic when a thread gets interrupted at the “wrong time”, and the program executes with a schedule that leads to an unintended result. Write down a

concrete problematic execution schedule for the threads executing philosopher 1 and philosopher 2**.** (You can express the execution schedule using a table with a column per thread, where each row corresponds to one atomic operation, like in slides 29-34).

Solution:

Step	*Philosopher 1*	*Philosopher 2*
1	*fork_taken[1] == false*	
2	*fork_taken[2] == false*	
3		*fork_taken[2] == false*
4		*fork_taken[3] == false*
5	*fork_taken[1] = true*	
6	*fork_taken[2] = true*	
7		*fork_taken[2] == true*
8		*fork_taken[3] == true*

The problem is that Philosoper 2 may get interrupted between checking that fork 2 is available, and proceeding to claim fork 2. In the meantime, Philosopher 1 may take fork 2.