Operating Systems Lab - Week 9: exercise - with answers

Implement a program that creates two concurrent threads to merge the content of two input files. Each thread reads from one of the files and writes on the *same* output file. The required Pthread functions are introduced in this week's slides. The main program waits for both threads to return, reads the content of the output file, and prints it on the terminal.

Similarly to the examples in the slides, start from a program that performs the two tasks sequentially, without creating any sub-procedure, and rewrite it as a multi-threads program. `0threadMerging.c`, the C code of the starting program is in Section 1. Sections 1 and 2 contain the instructions to rewrite it using one or two Pthreads. In Section 2, you implement

- the *sequential* setup, where the two tasks are performed *in series* by two independent Pthreads, i.e. the second Pthread starts when the first one has finished (in Section 2.1), and
- the *simultaneous* setup, where the two tasks are performed *in parallel* by two concurrent Pthread, i.e. the second Pthread starts immediately, without waiting for the first one to terminate (in Section 2.2).

The procedures open a given file, `input1.txt` or `input2.txt`, extract an integer from each *word* in the file, and print the integer on a given output file, `output.txt`. A word is a *group of characters between two single spaces*. The procedures transform it into an integer, as seen in previous labs. The extra spaces and the words containing only non-numerical characters would correspond to a 0 and should be ignored. To copy the integers into the output file, the procedures call the *formatted-I/O* function `fprintf`. The procedures call the same auxiliary function, `readIntegers`, with slightly different parameters, e.g. the name of the input file. The main program waits for both procedures to terminate and calls another function, `writeIntegers`, that i) reads the integers from the output file, ii) prints them on `stdout`, and iii) returns their sum. Finally, the program prints the sum on `stdout` and exits.

1 A program without Pthread

Copy `0threadMerging.c` and run it. Create the two input files by running

```
echo one1 two 2 three34four f5i5v5e and six=6 >input1.txt
echo 6six6 five54four t3h3r3ee and 2 and one1 > input2.txt
```

Then run the program to see what happens. Modify the two files to see how the output changes.

1.1 Making `readIntegers` Pthread-compatible

This is probably the hardest part of this exercise. Start by reading the code of the input-parsing function in `0threadMerging.c`, i.e.

```
int readIntegers(char *input, char *output,  char *threadName, char time) {
  char c = '\0';
  int nInt = 0;
  FILE *pf = fopen(input, "r");
  while (c != EOF) {
    int n = 0;
    while((c = fgetc(pf))!= ' ' && c != '\n' && c != EOF) {
      int t = 0;
      while (t < time) t++;
      if (c >= '0' && c <= '9')
        n = n * 10 + c — '0';
    }
    if (n) {
      nInt = nInt + 1;
      FILE *pfOut = fopen(output, "a");
      printf("%s writes %d \n", threadName, n);
```

```
      fprintf(pfOut, "%d ", n);
      fclose(pfOut);
    }
  }
  printf("%s wrote %d integers \n", threadName, nInt);
  fclose(pf);
  return nInt;
}
```

Try to understand the role of each argument and the meaning of the return value. Make `readIntegers` Pthread compatible by rewriting its argument and return value as *pointers to* `void`, define a new function declared as

```
void *PTreadIntegers(void *arguments)
```

Even if you pass it as a pointer to `void`, the argument should be a pointer to the following structure

```
struct pars {
  char in[MAXCHARS];
  char out[MAXCHARS];
  char threadName[MAXCHARS];
  int time;
  int nInt;
};
```

where the first four members represent the four parameters of `readIntegers` and `n` its return value. To write `PTreadIntegers`, look at how `brian` and `dennis` are transformed into their Pthread compatible versions in this week's slides. You need to *cast* the function argument, `void * argument`, to a pointer to `struct args`. This is to access the structure members of the structure as usual. Otherwise, `argument` would remain a pointer to `void` and the compiler would produce an error if you write something like `argument->in`. More practically, start the definition of `PTreadIntegers` with

```
void *PTreadIntegers(void *par) {
  struct pars *p = par;
  ...
```

Below this first line, copy the code of `readIntegers` all occurrences of the parameters replaced by the corresponding structure members, e.g. `in` will become `p->in`.

Answer:

```
void *readIntegers(void *par) {
  struct pars *p = par;
  char c = '\0';
  FILE *pf = fopen(p->in, "r");
  while (c != EOF) {
    int n = 0;
    while((c = fgetc(pf))!= ' ' && c != '\n' && c != EOF) {
      int t = 0;
      while (t < p->time) t++;
      if (c >= '0' && c <= '9')
        n = n * 10 + c - '0';
    }
    if (n) {
      p->nInt = p->nInt + 1;
      FILE *pfOut = fopen(p->out, "a");
      printf("%s writes %d \n", p->threadName, n);
      fprintf(pfOut, "%d ", n);
      fclose(pfOut);
    }
  }
  printf("%s wrote %d integers \n", p->threadName, p->nInt);
```

```
  fclose(pf);
  return NULL;
}
```

1.2 Rewriting `main`

Changing the function definition requires

- including the definition of `struct pars` on the top of the file,

```
#include <stdio.h>
#include <pthread.h>
#define MAXCHARS 100
struct pars{
  char in[MAXCHARS];
  char out[MAXCHARS];
  char threadName[MAXCHARS];
  int time;
  int nInt;
};
```

- declaring two objects of type `struct pars` , in `main` and initialize them by calling

```
void initialisePar(struct pars *par, char *in, char *out, char *name, int time) {
  copyString(in, par->in);
  copyString(out, par->out);
  copyString(name, par->threadName);
  par->time = time;
  par->nInt = 0;
}
```

  where you can use

```
int copyString(char *in, char *out) {
  int n = 0;
  while (*(in + n) != '\0') {
    *(out + n) = *(in + n);
    n++;
  }
  *(out + n) = '\0';
  return n;
}
```

- changing the input-parsing function calls,
- replacing `writeIntegers` with

```
int writeIntegers(struct pars *t1, struct pars *t2) {
  int i = 0, sum = 0, n = 0;
  FILE *pf = fopen(t1->out, "r");
  while(n < t1->nInt + t2->nInt) {
    fscanf(pf, "%d", &i);
    sum = sum + i;
    printf("main reads %d\n", i);
    n++;
  }
  fclose(pf);
  return sum;
}
```

- including all redefined functions in the function declaration list on the top of the file.

Answer:

```
#include <stdio.h>
#include <pthread.h>
#define MAXCHARS 100
struct pars{
        char in[MAXCHARS];
        char out[MAXCHARS];
        char threadName[MAXCHARS];
        int time;
        int nInt;
};
int copyString(char *in, char *out);
void initialisePar(struct pars *par, char *in, char *out, char *name, int time);
void initialiseFile(char *out);
void *PTreadIntegers(void *parameters);
int writeIntegers(struct pars *t1, struct pars *t2);
int main() {
        struct pars par1, par2;
        initialiseFile("output.txt");
        initialisePar(&par1, "input1.txt", "output.txt", "t1", 1);
        initialisePar(&par2, "input2.txt", "output.txt", "t2", 1);
        PTreadIntegers(&par1);
        PTreadIntegers(&par2);
        int sum = writeIntegers(&par1, &par2);
        printf("sum=%d\n", sum);
}
```

2 A program with 2 Pthreads

The program can now call `pthread_create` and other functions defined in `pthread.h` and have two independent Pthreads execute the subroutine `PTreadIntegers`. In Sections 2.1 and 2.2, you implement the *sequential* and *concurrent* setups.

2.1 Pthreads in series

Two threads execute *sequentially* if the second is called just after the first one has terminated, i.e. if you wait for the first to terminate by calling `pthread_join` before starting the second. In this case, the Pthread commands should be

```
pthread_t t1, t2;
pthread_create(&t1, NULL, PTreadIntegers, &par1);
pthread_join(t1, NULL);
pthread_create(&t2, NULL, PTreadIntegers, &par2);
pthread_join(t2, NULL);
```

Answer:

```
#include <stdio.h>
#include <pthread.h>
#define MAXCHARS 100
struct pars{
  char in[MAXCHARS];
  char out[MAXCHARS];
  char threadName[MAXCHARS];
```

```
  int time;
  int nInt;
};
int copyString(char *in, char *out);
void initialisePar(struct pars *par, char *in, char *out, char *name, int time);
void initialiseFile(char *out);
void *readIntegers(void *parameters);
int writeIntegers(struct pars *t1, struct pars *t2);

int main() {
  struct pars par1, par2;
  initialiseFile("output.txt");
  initialisePar(&par1, "input1.txt", "output.txt", "t1", 1);
  initialisePar(&par2, "input2.txt", "output.txt", "t2", 1);
  pthread_t t1;
  pthread_create(&t1, NULL, readIntegers, &par1);
  pthread_join(t1, NULL);
  pthread_create(&t1, NULL, readIntegers, &par2);
  pthread_join(t1, NULL);

  int sum = writeIntegers(&par1, &par2);
  printf("sum=%d\n", sum);
}
int copyString(char *in, char *out) {
  int n = 0;
  while (*(in + n) != '\0') {
    *(out + n) = *(in + n);
    n++;
  }
  *(out + n) = '\0';
  return n;
}

void initialisePar(struct pars *par, char *in, char *out, char *name, int time) {
  copyString(in, par->in);
  copyString(out, par->out);
  copyString(name, par->threadName);
  par->time = time;
  par->nInt = 0;
}
void initialiseFile(char *out) {
  FILE *pfOut = fopen(out, "w");
  fprintf(pfOut, "%s", "");
  fclose(pfOut);
}

void *readIntegers(void *par) {
  struct pars *p = par;
  int c = '\0';
  FILE *pf = fopen(p->in, "r");
  c = fgetc(pf);
  while (c != EOF) {
    int n = 0;
    while((c = fgetc(pf))!= ' ' && c != '\n' && c != EOF) {
      int t = 0;
      while (t < p->time) t++;
      if (c >= '0' && c <= '9')
        n = n * 10 + c - '0';
    }
```

```
    if (n) {
      (p->nInt) = (p->nInt) + 1;
      FILE *pfOut = fopen(p->out, "a");
      printf("%s writes %d \n", p->threadName, n);
      fprintf(pfOut, "%d ", n);
      fclose(pfOut);
    }
  }
  printf("%s wrote %d integers \n", p->threadName, p->nInt);
  fclose(pf);
  return NULL;
}
int writeIntegers(struct pars *t1, struct pars *t2) {
  int i = 0, sum = 0, n = 0;
  FILE *pf = fopen(t1->out, "r");
  while(n < t1->nInt + t2->nInt) {
    fscanf(pf, "%d", &i);
    sum = sum + i;
    printf("main reads %d\n", i);
    n++;
  }
  fclose(pf);
  return sum;
}
```

2.2 Pthreads in parallel

Two threads will execute *concurrently* if the second is called just after the first has started. In this case, the order of the Pthread commands should be

```
pthread_t t1, t2;
pthread_create(&t1, NULL, readIntegers, &par1);
pthread_create(&t2, NULL, readIntegers, &par2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
```

Answer:

```
int main() {
  struct pars par1, par2;
  initialiseFile("output.txt");
  initialisePar(&par1, "input1.txt", "output.txt", "t1", 1);
  initialisePar(&par2, "input2.txt", "output.txt", "t2", 1);
  pthread_t t1, t2;
  pthread_create(&t1, NULL, readIntegers, &par1);
  pthread_create(&t2, NULL, readIntegers, &par2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  int sum = writeIntegers(&par1, &par2);
  printf("sum=%d\n", sum);
}
```

2.3 Make the program *Pthread-safe*

As the threads are writing in the same file, `output.txt`, you must regulate their access to it. Introduce a *mutex* variable, `m`, declared outside all functions. In the function definition, use `pthread_mutex_lock(&m)` and `pthread_mutex_lock(&m)` to protect the writing statements, i.e. let

```
FILE *pfOut = fopen(p->out, "a");
printf("%s writes %d \n", p->threadName, n);
fprintf(pfOut, "%d ", n);
fclose(pfOut);
```

where `p` is the local pointer declared and initialized in the first line of `PTreadIntegers`.