Computer Science 315

Laboratory 3 - Concurrent Processes

Due: 7:00am Monday February 13, 2023.

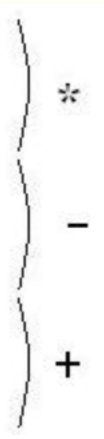
Many programming problems are solved using *serial processing*. That is, an individual part of the problem is solved, a result stored, another part solved, additional results stored, etc.. Consider the arithmetic expression:

```
      answer = a - b + c * d
```

Obeying the usual operator precedence rules, one serial solution goes like this:

Step	Operation	Result
1	`c * d`	Store product in `t1`
2	`a - b`	Store difference in `t2`
3	`t2 + t1`	Store sum in `answer`

Graphically, we could represent the sequence of operations as shown below, where the multiplication operation is done followed by the subtraction operation followed by the addition operation:



If we were to program this so that we could isolate each operation, the program might look like this:

```
#include <stdio.h>
#include <stdlib.h>     // For atoi, atof
#include <unistd.h>     // For sleep

int main(int argc, char * args[]) {

  int a, b, c, d, answer, t1, t2;
  a = atoi(args[1]);            /* extract numbers from command line args */
  b = atoi(args[2]);
  c = atoi(args[3]);
  d = atoi(args[4]);

  t1 = c * d;       sleep(1);   // The sleep(1) statements are included just
to
  t2 = a - b;       sleep(1);   // make the operation feel time consuming.
Here
  answer = t2 + t1; sleep(1);   // each operation is takes 1 second to
complete.

  printf("%d - %d + %d * %d = %d\n", a, b, c, d, answer);
  return 0;
}
```

The `sleep(1);` statements are there to make it feel as though the arithmetic operations take some time. If I were to run this, I would see that the total time needed for the calculation of `answer = a - b + c * d` would be a little more than 3 seconds. Try compiling it and running it like this:

```
      $ gcc fee.c -o fee
      $ fee 2 3 4 5
      2 - 3 + 4 * 5 = 19
      $
```

To see how long it takes, run it like this:

```
$ date; fee 2 3 4 5 ; date
Sun Jan 29 08:07:34 PST 2023
2 - 3 + 4 * 5 = 19
Sun Jan 29 08:07:37 PST 2023
$
```

Running the date command before and after roughly marks the time the program began and ended. Notice the time difference is about 3 seconds as expected.

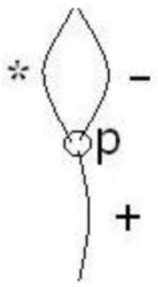
Serial processing is the norm on most systems, since we are typically limited to a single processor for our programs. On systems that have more than one processor, we may be able to accomplish the same task in a fewer number of steps. In some problems (this one, for example), there are parts of the problem that are not dependent on other parts of the problem and could be solved *concurrently*, if we had several processors or processes. For example:

Step	**Process(or)**	**Operation**	**Result**

1	1
	2
		`c * d`
		`a - b`
			Store product in `t1`
			Store difference in `t2`

2	1 or 2	`t2 + t1`	Store sum in `answer`

The two operations indicated in Step 1 could be done at the same time. Graphically, we could represent these operations as:



We can model concurrency by having forked processes calculate parts of the expression, and then use pipes to send results to other related processes. The processes will "rendezvous" at some point, meaning one process needs to wait for another to finish because it needs the work of that process in order to proceed. A pipe (eg. `p`, in the diagram above) is a convenient way of enforcing a rendezvous, because the process which is reading from the pipe waits (*blocks*) until the process that is writing to the pipe sends some characters (*blocking IO*). Here's a program which solves the problem above using concurrency. All four numbers are entered into the program as command line arguments:

```
#include <stdio.h>
#include <stdlib.h>    // For atoi, atof
#include <unistd.h>    // For fork(), pipe(), read(), write()

int main(int argc, char * args[]) {

  int a, b, c, d, answer, t1, t2;
  int p[2];
  int bufsize;

  a = atoi(args[1]);
  b = atoi(args[2]);
  c = atoi(args[3]);
  d = atoi(args[4]);
  pipe(p);                        /* set up pipe */
  bufsize = sizeof(int);          /* get size of data to be piped */

  if (!fork()) {                  /* child's code */
    t2 = a - b;  sleep(1);
    write(p[1], &t2, bufsize);
    return 0;
  }
                                  /* parent's code */
  t1 = c * d;    sleep(1);
  read(p[0], &t2, bufsize);
  answer = t2 + t1;  sleep(1);

  printf("%d - %d + %d * %d = %d\n", a, b, c, d, answer);
  return 0;
}
```

The program is run as follows:

```
$ gcc foo.c -o foo
$ date; foo 2 3 4 5 ; date
Sun Jan 29 08:09:47 PST 2023
2 - 3 + 4 * 5 = 19
Sun Jan 29 08:09:49 PST 2023
$
```

Keep in mind that on a uni-processor computer, if each operation actually did take 1 second of CPU time to complete, splitting the computation into separate pieces is **not** efficient, since all computations will be done serially anyway (since we only have one processor)! Here, we are *modelling* concurrency. However, if the operation takes a long time due to something that isn't CPU intensive (e.g. waiting for I/O or data to download) a uni-processor system could then take advantage of the benefits of concurrent processing. Web servers are a good example of this, where establishing a connection is time consuming, but not CPU intensive.

Notes:

- The function: `int atoi(char *)` -- returns an `int` value when given a C-style string variable as an argument. The string contained in the C-string variable must be an integer and not contain any other characters.

- For your future reference, the function: `double atof(char *)` -- returns a `double` value when given a C-style string variable as an argument. The string contained in the C-style string variable must be a proper floating point number.

- To capture command line arguments, the header of `main` looks like this:

  ```
  int main(int argc, char * args[])
  ```

  where `argc` is a count of the command line arguments (which includes the program), and `args` is an array of character strings that contain the command line arguments. For example, `args[0]` is a C-style string containing the name of the program as it was invoked at the command line (i.e. `"lab3sample"`). In my example above, `args[1]` would contain the string `"2"`, `args[2]` would contain the string `"3"`, and so on.

Exercise for grading:

In a file called `lab3.c`, write a solution to the following problem where all inputs are `double`s:

```
    answer = (((a + b) / 4) ^ c) + d / 2) - (3 * a + (4 * e / b))
```

Here are two sample runs of a program which solves the expression, using concurrency:

```
      $lab3 1 2 3 4 5
      (((1.0 + 2.0) / 4) ^ 3.0 + 4.0 / 2) - (3 * 1.0 + (4 * 5.0 / 2.0)) =
-10.578
      $ lab3 -2.5 19.8 5.4 1.6 3.1
      (((-2.5 + 19.8) / 4) ^ 5.4 + 1.6 / 2) - (3 * -2.5 + (4 * 3.1 / 19.8)) =
2726.147
      $
```

The user of the program enters 5 floating point numbers, which represent the variables a, b, c, d, and e in the expression above. The final answer turns out to be `-10.578`, in the first run of the program and be `2726.147`, in the second run of the program. The program was written so that it maximized parallelism (i.e. concurrency). Write your own version which also maximizes parallelism. Note that every arithmetic operation should be performed **on a separate line of code** like the sample code at the start of this lab. This will make it clear to see the flow of execution of operations. For example, an expression like this `(a - 2) * 2` should be written like this:

```
t1 = a - 2;   sleep(1);
t2 = t1 * 2;  sleep(1);
```

Doing this also affords you the opportunity to test the concurrency effectiveness. Remember that providing a `sleep(1);` statement after each operation can let you know how long your program takes to execute. **If you do not break every arithmetic operation onto a separate line of code, there will be a 50% deduction in addition to deductions for a lack of concurrency.**

Implementation Notes:

- My version solved the problem with 4 processes. I used 3 pipes to transfer calculated values between processes. Be sure to use a separate pipe each time you need to transfer data between processes. **Use different letters for your pipes**. Alan used **p**, **q** and **r** for his pipes, I used **pipeA**, **pipeB**, **pipeC** for my pipe variables as they are consecutive letters. Such a convention will help you debug your code and help me grade your code.
- You should draw a diagram for the sequence of operations on a piece of paper first (as in the graphical representations above), and ask a friend to check it for maximum parallelism before implementing your solution. **The problem becomes much, much more difficult to manage without a diagram**. **If you ask me for help, I will ask you to show me your diagram first. If you need help making a diagram, come see me.**
- You should not include more processes than what is necessary. I will deduct marks for too many processes.

- To raise one number to the power of another number, use the `pow` function (see `man pow`). You will also need the following compiler directive:

  ```
  #include <math.h>
  ```

  When you compile your program, you'll need the math library during the link stage, so the correct compile command is:

  ```
  $ gcc lab3.c -o lab3 -lm
  ```

  Notice the library switch (that's an "ell", not a "one") which specifies the math library (ie. `m`).

Submit:

Submit your program via Moodle prior to the deadline. Marks will be awarded according to how well you've taken advantage of the inherent parallelism of the problem.