

Homework 1

Assignment due:

8:00 pm, Wednesday, February 20th.

Problems 1, 2, 3: Turnin typed (not hand-written) answers to the non-programming problems. You can hand-in a printed copy, or use electronic turnin. Turnin of printed copies can be done by giving them to me during class on (or before) Tuesday, February 19th. They can be delivered to the CS department office (GS-917) by 5 pm on Wednesday, February 20th. If using electronic turnin, provide a pdf file named **answers.pdf**. See the instructions at the end regarding the use of turnin. Problems 1-3 are worth 36 points (12 points each).

Program: Submit the program solutions electronically. The programming exercise is worth 64 points.

A reminder: Homework assignments are individual efforts. The two projects can be done in teams of two, not the homework. You may discuss the meanings of questions with classmates, but the answers and programs you turn in are to be yours alone. For the exercises from the book, explain your answers clearly and succinctly.

Problem 1:

This problem is similar to 2.17 and 2.18.

Consider the following program:

```

co <await (x != 0) x = x - 2;> /* A */
    // <await (x != 0) x = x - 3;> /* B */
    // <await (x == 0) x = x + 5;> /* C */
oc

```

- For what initial value(s) of x is the program guaranteed to terminate, assuming scheduling is weakly fair? What are the corresponding final values? Explain. “Guaranteed to terminate” means that all three arms of the **co** will complete.
- Are there additional initial value(s) of x for which the program might terminate? If so, show how those values might result in termination, and the final values that might result. Explain. “Might terminate” means that there is at least one scenario in which all three arms of the **co** will complete, but they are not guaranteed to do so.

Problem 2: Exercise 2.20, pages 86-87

(Only three of the parts)

Let $\mathbf{a}[1:m]$ and $\mathbf{b}[1:n]$ be integer arrays, $m > 0$ and $n > 0$. Write predicates to express the following properties.

- All elements of \mathbf{a} are less than all elements of \mathbf{b} .
- It is not the case that both \mathbf{a} and \mathbf{b} contain zeros.
- Every element of \mathbf{a} is an element of \mathbf{b} .

Problem 3: Exercise 2.20, pages 86-87

(Only three of the parts)

Let $\mathbf{a}[1:m]$ and $\mathbf{b}[1:n]$ be integer arrays, $m > 0$ and $n > 0$. Write predicates to express the following properties.

b.) Either \mathbf{a} or \mathbf{b} contains a single zero, but not both.

d.) The values in \mathbf{b} are the same as the values in \mathbf{a} , except they are in reverse order. (Assume for this part the $m == n$.)

f.) Some element of \mathbf{a} is larger than some element of \mathbf{b} , and vice versa.

Problem 3: Exercise 2.33

Consider the following program:

```
int x = 10, c = true;
co <await x == 0>; c = false;
// while (c) <x = x - 1>;
oc
```

a.) Will the program terminate if scheduling is weakly fair? Explain.

b.) Will the program terminate if scheduling is strongly fair? Explain.

c.) Add the following as a third arm of the **co** statement:

```
while (c) {if (x < 0) <x = 10>;}
```

Repeat parts (a) and (b) for this three-process program.

Programming Exercise:

(Variations on Exercise 1.8, page 36)

The parallel adaptive quadrature program in Section 1.5 will create a large number of processes, usually way more than the number of processors. For example, the MPD web site has a parallel solution that is based on the recursive solution. If you try running this on *lectura* or *voltron*, you will find that it will fail after creating about 50 processes. This is a limitation imposed by the department on the number of threads that can be created by one user.

Write two versions:

a.) Modify/extend the recursive solution to limit the number of processes. The number of processes will become a command-line argument. Name the program **parquad.mpd**. The command-line will become:

```
parquad epsilon left right numProcs
```

where:

epsilon is the error to stop the recursion

left, right define the lower- and upper-limits of the integration

numProcs is the number of processes to use in the solution

You can assume a maximum upper limit of **16** processes. We are using machines with at most four processors, but it is useful to explore whether there is any advantage in this algorithm to having more processes than processors.

b.) Extend your solution to a.) to show a plot of the results using the **MPDWin** package. Name the program **parquadWin.mpd**. The MPD web site has some information about **MPDWin** (and the SR web site has information about **SRWin**, the predecessor package). Both mention a technical report that describes the **SRWin** package. The technical report link is to a PDF file that has the pages in reverse order(!). I have put a copy of that file on the class web site, with the pages in the correct order. There is a manpage named **srwin** which lists all the library calls and data

structures used with short explanations. I have added a PDF version of this manpage to the class web site.

You can assume a fixed size for the window. The x-axis should scale to the range of values specified by left and right. You can set upper- and lower-bounds on the y-axis. For example, I have been testing my solution with periodic functions that range from -1 to $+1$.

For both programs, print the formula used, the epsilon value, the x boundaries, the number of processors used, the area, and the execution time. For example:

```
lectura-> parquad 0.000001 -20 50 4
Formula: sin(20*x) * cos(x/3) + sin(20*x) * cos(x/6)
epsilon = 1.00000e-06  a = -20.0000  b = 50.0000
processors = 4
area = 0.0306066
time = 26 milliseconds
```

Prepare a short report (not more than two pages) that describes the test cases you ran and the results you obtained. Try at least two formulas. Try various numbers of processes. You can also test `lectura` vs. `voltron`. Turnin your report along with your two programs. The report can be a text file or a pdf file. Name the report: `report.txt` or `report.pdf`.

Turnin: Use the `turnin` program to turn in three files: `sort-seq.mpd`, `sort-co.mpd`, `sort-process.mpd`, `timing.txt`. The command is:

```
turnin 422assign1 parquad.mpd parquadWin.mpd report.txt
```

If you are using `turnin` for the answers to problems 1 to 3, the `turnin` command is:

```
turnin 422assign1 answers.pdf
```

See the man page for the `turnin` program for details on what `turnin` can do and how you can confirm that your file was turned in.

MPD Notes:

These notes are general. They contain items that are specific to this assignment and items that may not directly apply. Since I have notes that I have included with assignments from previous semesters, I am including those here.

- The MPD web page has a sequential quicksort program. It is a nice example of how to use array slices (portions of arrays) within procedures. However, it has a serious problem when working with very large arrays of strings. The problem is caused by the slices of the array that are passed to each recursive call. While array slices are a nice feature of the language, they can be expensive in terms of memory space on the stack when passed as arguments to a procedure. Here is an excerpt from the book: *The SR Programming Language* by Gregory R. Andrews and Ronald A. Olsson. The description of the different parameter passing techniques also applies to MPD:

Parameters are passed to procedures by copying or by address; the return value is passed by copying. Value parameters (**val**) are copied in, result parameters (**res**) and the return value are copied out, and variable parameters (**var**) are copied both in and out. On

the other hand, reference parameters (**ref**) are passed by address. The default parameter kind is **val**...

Passing a parameter by copy-in/copy-out (i.e., **var**) often has the same effect as passing it by reference. However, passing by reference is more efficient for large data structures such as arrays because passing just the address of an array takes less time and space than does copying all the elements of an array. However, reference parameters should not be passed between virtual machines (address spaces) since the value of a reference parameter, which is a memory address, cannot be used safely.

This is a somewhat detailed explanation of why the **var** parameter-passing of the provided sample does not work well with large arrays. The solution of using **ref** parameters avoids the stack space problems of **var** parameters; however, **ref** parameters do not work with array slices.

You can pass the address of the array as a **ref** parameter, along with two integers to specify the starting and ending indices of a slice. Or, you can treat the array as a global and not pass the array itself in any form. In this version, you would pass the starting and ending indices of a slice. For quicksort, this works because each process that gets created works on its own slice of the array independent of the slices being used by the other processes.

- Regarding local vs. global. A variable that is declared within a proc or procedure is local to the proc or procedure; that is, it exists on the stack of the proc or procedure. A variable that is declared within a resource, but not within a proc or procedure, is visible to any code within the resource, including code that is within a proc or procedure.
- There is a second way to create a process in MPD (in addition to using the process statement). Within a body of code, such as a loop, you can create a process by using a **send** to a **proc** rather than calling the **proc**. For example, if I have a simple **proc** whose **op** declaration is:

```
op zap(int a);
```

I normally call the function with **zap(17)**. If I want a process to be created, I instead use the syntax: **send zap(17)**. A process in MPD is actually a thread that begins executing when the **send** occurs and the calling thread can then continue with the code that follows the **send**. There is no waiting for the process to terminate.

When using this technique, you may need a **final** clause. The **final** clause will not execute until all the threads have terminated.

- Strings in MPD:

```
string[20] x;
```

declares a string named **x** that can contain up to 20 characters; the individual letters within the string will be at locations 1 to 20 within the array.

```
string[40] z[5];
```

declares an array named **z** that contains 5 strings, each of which can contain up to 40 characters.

MPD supports a swap operator that will swap the values of its operands. The operator work with strings (as well as all other types): **x ::= y;**

- The **MPDWin** package uses variables of type **winWindow**. These are handles to windows that you can create. The **WinOpen()** function call returns this type. For example,
winWindow mywin;
mywin = WinOpen("", "ParQuad", null, UseDefault, WINWIDTH, HEIGHT);
The upper-left corner of this window has the coordinates **(0, 0)**. The lower-right corner of the window has the coordinates **(WINWIDTH - 1, HEIGHT - 1)**.
- The **MPDWin** package supports a “window context”. This allows multiple processes to each have a different graphics context for the same window. To create a new context for an existing window, you need a variable of type **winWindow**. You then use the **WinNewContext()** function, passing as the parameter a **winWindow** that was returned by a previous call to **WinOpen()**. This allows each process to set its own foreground color, for example.
- You can set the color for drawing using the **WinSetForeground()** function call. There are predefined colors, including red, yellow, blue, green, white, cyan, black.
- The type **winPoint** is predefined in **MPDWin**. It is a record containing two fields, the **x** and **y** coordinates of a point. For example:
winPoint vertex = winPoint(13, 27);
defines a point that is **13** points to the right and **27** points down from the upper-left corner of the window.
- There are multiple functions for drawing individual pixels, polygons, lines, etc., including filled versions of these. For example, **WinFillPolygon()** will draw a polygon whose vertices are defined by an array of **winPoint**'s.