

MPD Notes

These notes are general. They contain items that are specific to assignments this semester 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.

Parameter Passing:

- MPD supports three different ways to pass parameters: **val**, **res**, and **var**. There are advantages and drawbacks to each both in terms of the memory space required on the stack, and the execution efficiency. 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.

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.

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.

Local vs. Global Variables:

- 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**

Process:

- 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:

- 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;**

XWindows:

- 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.

Dynamic Memory Allocations and Records (structs):

MPD supports a record data structure, which is analogous to a **struct** in C. This can be used to create an array of records, a linked-list of records, etc. Declarations of records start by declaring a new type. For example:

```
type person = rec( string[40] name; int height; int weight);
```

You can then declare individual instances of **person**:

```
person me, you.
```

You use 'dot' notation to reference individual fields of a record:

```
me.height = 185;  
write("My weight is:", me.weight);
```

MPD also supports pointers as a type; for example, you want to build a linked list:

```
ptr int qPtr;      # defines qPtr as a pointer to an integer  
int xray;  
qPtr = @xray;     # @ provides the address of its operand  
qPtr^ = 17;       # will make 17 the value stored in xray
```

For a pointer to a record, you have to declare a new type:

```
type personPtr = ptr person;  
personPtr people = null; # use null for an empty pointer.
```

Use the functions **new()** and **free()** to dynamically allocate and deallocate memory.

```
people = new(person);  
people^.name = "Patrick";  
people^.height = 185;
```

To use **free()**, you supply a pointer:

```
free(people);
```