Operating Systems: Practical One

Mike Spivey

Michaelmas Term 1993

1 The Task

Your task is to implement the software for a computer-controlled digital thermometer. The thermometer is able to display temperatures in two windows, one labelled INSIDE and the other OUTSIDE. The temperature in the INSIDE window is measured by a sensor built into the display, and that in the OUTSIDE window is measured by a remote sensor, which can be placed outside. The sensors used for outside readings are often wrong by a small, constant offset, so there is provision to adjust the temperature displayed in the OUTSIDE window up or down relative to the temperature measured by the sensor.

Some design decisions have already been taken: the software will be structured as a collection of processes, with communication by synchronous message passing as in MINIX. The two windows will be updated by separate processes.

To help you implement the software, a package that supports MINIX-style processes and communication is provided, together with a ready-made process for simulating the display, and a simulation of the temperature measurement hardware.

Your task is as follows:

1. Study the description of the problem (Section 2).
2. Study the information about concurrent processes (Section 3), the specification of the ready-made parts (Section 4), and the example program that implements a simpler thermometer (Appendix A).
3. Prepare a program that satisfies the requirements and make it work.
4. Submit a report that contains a well-formatted and annotated listing of your program, and a discussion of the major design decisions that you made.

A well laid-out solution to the programming problem need be no longer than 150 lines of C code, not counting comments.

2 The requirements

The 'Gardener's Thermometer' has a display with two windows. One window is labeled INSIDE and the other OUTSIDE, and each shows a temperature in Celsius. The thermometer has three buttons, marked `+`, `-`, and `s`. There are two temperature sensors, one in the same case as the displays, and another remote from it. Initially, each window shows the temperature measured by the corresponding sensor, but the temperature shown in the OUTSIDE window can be adjusted upwards or downwards from the sensor value in increments of 0.1 C by pressing the `+` or `-` button respectively. In the production model of the thermometer, one window will have larger digits than the other. Pressing the `s` button swaps the two temperature windows, so that either the INSIDE or the OUTSIDE display can be put in the window with larger digits.

The thermometer hardware generates a signal approximately once a second that is sent to the software. Immediately after receiving the signal, the software must update the temperature displays. There is a single, multiplexed analogue to digital converter that is used to obtain temperature readings from both sensors.

3 Processes and communication

A package is provided that lets you write programs that run as a single UNIX process, but have several concurrent tasks inside. These tasks are often called 'threads', and from this the package gets its name. The package takes the share of machine time allocated to the UNIX process and further divides it among the threads inside. The threads are scheduled non-preemptively, meaning that once a thread has begun execution, no other thread gets a share of the processor until the first thread either gives it up voluntarily or is blocked, waiting to send or receive a message.

To use the threads package, you first create the processes you want to run, associating each of them with a C function. When the processes have all been created, control passes from the main program to the threads package, which starts all the functions as independent processes, transferring messages between them, and running each of them when it is ready. A program that uses the threads package should include the header file `threads.h` with the directive

```
#include "threads.h"
```

The rest of this section contains detailed information about how to use the threads package. Most of the examples are taken from the program in Appendix A. The header file `threads.h` is listed as Appendix C.

3.1 Creating and running processes

Processes are created by calling the function `co_call`; its heading is

```
PUBLIC int co_call(name, code)
char *name;
void (*code)();
```

The `name` argument is a string that is used only for debugging (see Section 3.3). The `code` argument is (a pointer to) the function that is the code for the process. `Co_call` does not call this function immediately, but saves the name and the function in a private table for later use. It returns an integer, the internal process identifier (PID) of the process. You will need this later to send messages to the process, so it is best to save it in a global variable.

The `main` function of a program that uses threads typically contains a section like this:

```
keyboard = co_call("keyboard", do_keyboard);
update   = co_call("update",   do_update);
display  = co_call("display",  do_display);
```

This code creates three processes. The code for each process is a function like `do_keyboard` defined elsewhere, with a heading like this:

```
PRIVATE void do_keyboard()
```

The result of `co_call` is saved in a global variable, for later use as an argument to `send` or `receive`. It is possible to pass the same function as argument to `co_call` several times, thereby creating several processes that share the same code but execute it independently.

Once all the processes are created, they can all be started concurrently by calling the function `co_start`; here is its heading:

```
PUBLIC void co_start()
```

There is no way to create new processes dynamically once concurrent execution has started.

Once `co_start` has been called, the threads package takes control, and `co_start` does not return until one of the following three things happens:

1. All the processes die because the functions associated with them have returned.

2. One of the processes calls the special function `co_exit`.

3. Not all of the processes have died, but none of the ones remaining is ready to run (deadlock).

The nicest concurrent programs terminate by having all their processes die at the same instant; but that is often difficult to arrange without an elaborate system of messages saying "That's all, folks". That is why the function `co_exit` is useful:

```
PUBLIC void co_exit()
```

Stopping the program by having it deadlock is considered extremely poor style. The threads package reports it as an error, after printing a table that shows the status of all the processes in the program. This can be helpful during debugging.

3.2 Messages

Processes communicate with each other using messages. These are objects of type `message`, with the following definition taken from `threads.h`:

```
/* Message format */
typedef struct {
     int m_source;                /* PID of sender     */
     int m_type;                  /* Type of message   */
     union {                      /* Contents:         */
          int m__int;             /*   a number,       */
          char m__char;           /*   a character,    */
          char m__string[10];     /*   or a string.    */
     } m_u;
} message;

/* Short form selectors */
#define m_int m_u.m__int
#define m_char m_u.m__char
#define m_string m_u.m__string
```

Each message identifies its sender (filled in automatically when it is sent), the kind of message it is, and one piece of data, which may be an integer, a character, or a short string. The integer field of a message `m` is referred

to by the (slightly awkward) expression `m.m_u.m__int`, or by the abbreviation `m.m_int` . The format of messages is not very general, but for the thermometer, one data item per message will be enough.

The `m_type` field of a message can be any number you choose, and it is conventional to define a few symbolic constants for the message types used in your program; for example:

```
/* Message types */
#define VAL 1
#define PUT 2
#define GET 3
```

Two functions are used by processes to send and receive messages. Here are their headings:

```
PUBLIC void send(dst, msg)
int dst;
message *msg;

PUBLIC void receive(src, msg)
int src;
message *msg;
```

The `send` function takes the PID of a process to send to, and a pointer to the message to send. If the destination process is waiting to receive a message from the calling process, then the transfer takes place immediately, and both sender and recipient may continue their execution. If the destination is not ready to receive, then execution of the caller is suspended until the destination is ready.

The `receive` function is complementary. Its arguments are the PID of a process from which a message may be accepted, and a pointer to a buffer to receive the message. The first argument may be the special value `ANY` to indicate that a message may be accepted from any source. If an acceptable process is waiting to send, the transfer takes place immediately; otherwise the current process is suspended until an acceptable sender comes along.

A message transfer can only happen when the sender is executing (or waiting in) a call to `send`, and the receiver is executing (or waiting in) a call to `receive`. When a transfer takes place, the message is copied from the sender's buffer to the receiver's, with the true identity of the sender filled in. This makes it possible to reply to a message, by sending to the PID that was stamped on the message when it was received.

It is not permitted for the main program to call either `send` or `receive`, either before or after its call to `co_start`.

3.3 Debugging

The threads package has a tracing factility that can be useful while you are debugging your program. The easiest way to turn on tracing is to set the global variable `trace_flag` to 1 as the first action of your main program.

If tracing is turned on, then a report is printed on standard output every time a process resumes execution, and every time a message is delivered. The report about a message gives the source and destination, and also the integer value of the `m_type` field in the message. Process names in these messages are the ones you supplied as the first argument of `co_call`.

Note that the report about delivery of a message is printed only when the transfer actually takes place. If the destination is not ready to receive the message when the source calls `send`, then the report will not appear straight away.

Another debugging feature of the threads package is the function

```
PUBLIC void dump_state()
```

which prints on the standard output the state of each process: whether the process is ready to run, or is waiting to send or recieve a message. This function is called automatically in case of total deadlock, but you might use this function by attaching it to a key on the keyboard. That way you can investigate partial deadlock situations where some processes in your program have stopped communicating, but others (including the keyboard!) are still working.

In debugging your program, it will help if you bear in mind the three great mistakes of C programming. They are these: writing `=` to test for equality when `==` is needed, forgetting to put `break` between the arms of a `switch` statement, and calling a function with the wrong arguments. None of these errors are caught by the compiler, of course.

4 Ready-made parts

A number of ready-made parts are provided for you to use in building the thermometer program. These parts are simulations of the parts used in the production model of the thermometer, so it is cheating not to use them! To make the parts accessible, you should put the line

```
#include "parts.h"
```

at the top of your program.

4.1 Display

The first ready-made part is a process `do_display` that simulates the display. You can use it in your program by including the file `parts.h` and creating a process like this:

```
display = co_call("display", do_display);
```

The display process shows on the screen an image of the thermometer display, rather like this:

```
Gardener's thermometer

[INT]  [ 23.6]    [EXT]  [ 22.8]
```

The process accepts the four message types `UPDATE0`, ..., `UPDATE3` to update the four display fields, counting from left to right. When it receives one of these messages, the corresponding field in the display is immediately updated from the appropriate component of the message: `m_string` for the labels, and `m_int` for the numbers. which are expressed as a whole number of tenths of a degree, so that 73.6 would be represented by 736.

4.2 Tick signal

The hardware generates a 'tick' signal once a second, and this is used to trigger the update of the temperature displays. When a tick signal occurs, a fictitious process `HARDWARE` sends a message of type `TICK` to a chosen process in your program. Apart from its type, the `TICK` message contains no other information.

A process can connect itself to the tick signal by making the function call

```
connect(&tick_dev);
```

The function `connect` is provided as part of the interface in `threads.h`, and `tick_dev` is an object of type `device` provided in `parts.h` – but you don't need to know the details. The tick signal may be connected to only one process at a time, and if another process calls `connect(&tick_dev)`, the old client is disconnected before the new one is connected.

If the connected process is not ready to receive a `TICK` message when it arrives, the message is saved until the process is ready for it; but if more ticks should arrive in the mean time, the second and any subsequent ones are discarded.

The constants `HARDWARE` and `TICK` (and also constants like `KEYPRESS` and `TEMPVAL` mentioned below) are defined in the header file `parts.h`. The message types are negative numbers, so they will not clash with message types you choose, provided you make yours positive.

4.3 Keyboard

When a key is pressed, the fictitious `HARDWARE` process sends a message of type `KEYPRESS` to another chosen process in your program. The content of this message (in the `m_char` component) is the character that was received.

A process can connect itself `KEYPRESS` messages by the function call

```
connect(&kbd_dev);
```

Like the tick signal, the keyboard can be connected to only one process at a time. Keyboard characters that are not immediately accepted by the process that is connected to the keyboard are queued and sent later.

Calling `connect(&kbd_dev)` automatically sets up the mechanism needed to monitor the keyboard: this puts it into a state where input characters are accepted immediately without echoing or waiting for a carriage return. Normal keyboard service is resumed when concurrent execution finishes.

The keyboard interface is implemented by creating a UNIX process that runs a program called `kbd`. This program monitors the keyboard for key depressions, and sends the characters to your program through a UNIX pipe, along with a signal that input has arrived. This technique makes it possible for other threads to continue running whilst one of them is waiting for keyboard input. Normally, you will not even notice that the `kbd` program is there, but if your program crashes, it can happen that `kbd` still continues to run. Since `kbd` turns off echoing of the characters you type, this can be a nuisance. An easy solution is to delete the window in which the rogue `kbd` is running.

5 A-to-D converter

The two temperature sensors are connected to a single analogue to digital converter (ADC) through an analogue multiplexer. To obtain a temperature reading, the ADC samples the required sensor and converts the voltage to digital form. Because the ADC is a cheap one, this takes some time, and the hardware is arranged so that the processor receives an interrupt when conversion is complete.

To begin a temperature reading, your program should call the function

```
PUBLIC void start_adc(chan)
int chan;
```

The `chan` argument tells the ADC which sensor channel to use: 0 for the outside sensor, and 1 for the inside one. Whilst the ADC is at work, your

program must not call the `start_adc` function again. Doing so results in nonsense readings, and may cause the loss of the completion signal.

When conversion is complete, the ADC interrupt causes a message of type `TEMPVAL` to be sent from `HARDWARE` to a chosen process in your program. The `m_int` component of this message contains the required temperature in tenths of a degree centigrade. A process can connect itself to the `TEMPVAL` messages, like the `TICK` and `KEYPRESS` messages, by calling the `connect` function:

```
connect(&adc_dev);
```

As before, only one process can be connected at a time.

The temperature readings generated by the simulation are sine waves with a period of 60 seconds; the exterior temperature varies between −5 C and 15 C, and the interior temperature varies between 0 C and 20 C. In real life, the temperature readings might go through the same variations in the course of a day, rather than a minute.

6 Practical instructions

This section contains suggestions for the order of work during the practical session. You are not required to do the work in the order specified or in the stages suggested, but you are required to describe and justify the major design decisions that you take, and these decisions are enumerated as part of the work plan.

6.1 Getting started

The files containing the threads package and the ready-made components are all in the directory `/usr/local/opsys/prac1`. These are the files you will need:

`Makefile`	Project description file
`threads.h`	Header file for threads package
`parts.h`	Header file for ready-made parts
`threads.o`	Object code for threads package
`parts.o`	Object code for ready-made parts
`thermo.c`	Source code for a simple thermometer

In addition to the files already listed, you will find the source code for the threads package and the ready-made parts in two files `threads.c` and `parts.c`; the file `iosim.h` documents the interface to the threads package that is used by the simulated I/O devices. You will not need to look at these files, although you may do so to satisfy your curiosity; but be warned that

the threads package is implemented in a highly machine-dependent way that would be bad style if it were not unavoidable.

Begin your work at the machine by compiling and running the `thermo` program for yourself – this will make sure you can use the C compiler and tools before you start work on programs of your own. Make a fresh directory, and into it copy the files `Makefile` and `thermo.c`. The first of these describes how to compile and link the `thermo` program, in a way that is understood by the UNIX tool `make`. Both files are reproduced at the end of these notes. Now type just

```
make
```

and the `make` program will take over, first compiling `thermo.c`, then linking the result with the other components. The result is an executable program `thermo` that you can run by typing its name in the usual way. Do it.

The `thermo` program maintains one temperature display (showing the outside temperature) in the left-hand window, leaving the other one blank. Apart from a process to monitor the keyboard, and the ready-made display process, it contains a single main process that receives ticks from the hardware, gets the temperature, and shows it on the display. When you are bored with watching the program, type a full stop. The sole function of the keyboard process in this program is to wait for this full stop, and then terminate the program.

Devotees of modern C style will notice (with disgust) that we are using the 'old-fashioned' Sun C compiler in this practical. They will be disappointed, however, if they try to substitute the Gnu C compiler, because the dirty trick by which the threads package is implemented will not work with the Gnu compiler.

6.2 Twin displays

Next, design a program that maintains two temperature displays. It will have two main processes in place of the single process of the original `thermo` program, each sending update messages to its own window in the display. You need not worry yet about supporting the keyboard functions, but it is a good idea to keep the full-stop mechanism from the `watch` program, so that you have some way of stopping the program neatly.

Since the hardware tick signal can be received by only a single process – a second call of `connect(&tick_dev)` disconnects the first recipient – you will have to devise some way of distributing the ticks to both processes. Also, both processes will need to use the facilities for getting temperature readings, so you will have to come up with a way to prevent them from interfering with each other. If it detects interference, the A-to-D simulation

returns the impossible (in England) temperature −99.9 C in place of a proper temperature.

With care, you can arrange for both the main processes to run the same code. When it starts, neither process will know which window it is supposed to be using; but you could arrange that one of the other processes in the program sends each of them a message to tell it which window to use. With even more care, you can arrange for all the processes in the program to be written in a way that does not assumes that there are exactly two displays: see if you can work out how to do it.

Put the source code of your new program in a file `thermo2.c`. It may be easiest to begin with a copy of `thermo.c` and edit it to add the new functions. When you are ready, compile and link it by saying

```
make thermo2
```

(this only works if you have put your program in `thermo2.c`). Try it out.

6.3 Adjusting the EXT display

Now add code to the keyboard process to recognize the characters `+` and `-`, and send an appropriate message to the process that is maintaining the EXT display. How many different message types do you need to add?

Modify the updating process to recognize the messages to adjust the temperature and act on them. Make sure that the operator sees the effect of the change immediately, rather than having to wait for the next ordinary display update.

6.4 Switching windows

Add code to the keyboard process to recognize the letter `s` and notify the other processes appropriately, by sending messages. Modify the other processes to act on these messages. You can either keep each of the updating processes attached to a fixed window, and change the temperature display that each maintains in its window; or you can give each process a fixed display to maintain, and change the connection between the processes and the windows. Which is simpler?

6.5 To think about

Think about these questions and make sure you answer them in your report:

1. The software design you have implemented uses two largely independent processes to maintain the two displays. Another design might

use a single process that maintains both displays, but it would have to be able to respond to a wider variety of events. What would be the advantages and disadvantages of this design? What changes to the requirements would make each of these designs preferable to the other one?

2. When the operator is not pressing buttons, how many messages are sent for each one-second update of the display? How could you change the design to reduce this number?

3. How would you modify the process structure you have designed if the thermometer was also required to provide an audible frost warning?

7 Appendices

Appendix A `thermo.c`

Appendix B `Makefile`

Appendix C `threads.h`

Appendix D `parts.h`