Lecture Notes # 2
17 January 2012

Lecturer: Professor Suresh Jagannathan

Scribe: Matthew Barrett

Contents:

1 Section 1

First we resolve some issues raised during lecture on 12 January 2012 regarding the organization of processes on a machine.

A program runs in the context of a process, which is depicted in the following Figure.

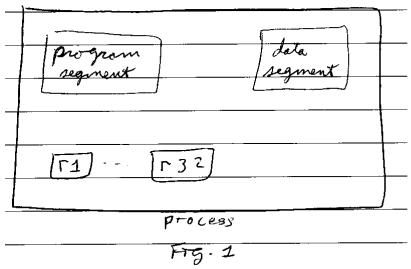

Fig. 1

In Figure 1 the "program segment" represents the instructions to run, the "data segment" contains the heap and stack (as is depicted in Figure 2), and r1, ..., r32 are the register values.

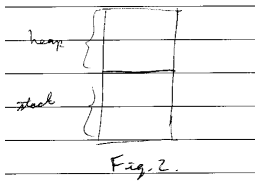

Fig. 2.

The heap contains memory segments which were obtain by calls to *malloc*() while the stack contains all of the procedure calls and their local (non-*malloc*()ed) data. Each procedure call is placed in a frame which contains certain data for the procedure.

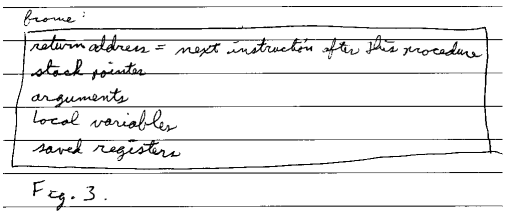

Fig. 3.

For example if our program begins execution then a frame for $main()$ is pushed onto the stack.

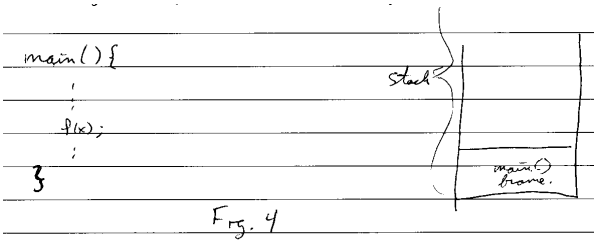

Fig. 4

Then if our $main()$ function calls a function $f()$ with parameter x, then

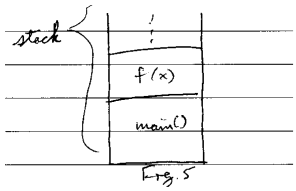

Fig. 5

the stack looks like this.

When the (any) new function is called all state information for the current function is saved in the frame before passing to the new frame.

Each process believes that it has access to the entire system memory address space. Processes typically use much less memory than the system offers and so the operating system maps the process' addresses to some subsection of the system memory. This is made possible by a partitioning scheme of the process' address space into "pages," where the used pages are kept in physical (system) memory.

2 Section 2

Homework #1 was assigned and is **due 31 January 2012** [4]. To introduce this homework assignment we recall "*call/cc*" from the previous lecture and the notion of a "co-routine," being a mechanism which allows the transfer of program control without returning. (I.e., when control is transferred, if the new routine throws control back to the original routine then the original routine resumes execution where it left off.) To implement said functionality use a "continuation" as mentioned in the previous lecture. See also lecture slides 2 and 3, [2,3]. For this assignment we will need to download and install Racket [5]. There are additional readings listed on the course web page. The solution will be limited to replacing the lines "Suspend _____" with approximately 3 lines of code to implement provider/consumer transfer of control. (**Use call/cc**.)

3 Section 3

New material began on slide 14 of Lecture Slides 2, [2].

Slide 14

These two trees have different structures but the letters labeling the leaves appear in the same order on each, which is called a "same fringe" pair. Given two trees, how can we determine if they are a same fringe pair?

Slide 15

A simple approach would be to collect all of the leaves of the trees into two arrays and then compare the leaves. This approach however has several drawbacks, most notably that too many steps will be executed if even the first leaves differ!!!

Slide 16

This slide introduces a slick approach to solving the problem by using co-routines and passing control. The philosophy is simple and described here. Let routine A find the first leaf of trees T_A and then throw control to routine B which finds the first leaf of tree T_B. Routine B then compares the leaves and if they fail to match then routine B acknowledges so and terminates the search. Otherwise routine B throws control back to routine A which finds the next leaf of tree T_A, and so on ad infinitum. This slide provides code and asks how to write such routines A, B.

Slide 17

This slide tells us that a computation (state information) may be captured and stored by "call/cc".

Slide 18

More of slide 17.

Slide 19

This slide produces all of the code for a similar example, similar to the "same fringe" problem. A procedure called '$generate - numbers()$' will generate numbers in the order $0, 1, 2, \ldots$. However, after each generation the "call/cc" function saves the computation (state information) and throws control, with parameter the generated number, to the procedure '$make - generator()$'. This procedure executes and then throws control back to '$generate - numbers()$' to generate the next number in the list. This repeats ad infinitum. It seems that the syntax of the code was the tricky part of this slide, ye be warned.

Slide 20

Puts into words the content, briefly, of slide 19.

Slide 21

If we wish to have more explicit control over the continuation-capture process then we may implement this in a language like **Scheme**, ML, Haskell or others.

$$!The homework is to be completed in \textbf{Scheme}!$$

This slide contains one such example in ML.

Slide 22

This slide suggests that a determined computer scientist may be able to implement threading (scheduling, cooperation) and concurrency with continuations. Caveat coder: this ain't so easy.

The remainder of the lecture focused on the first 5 slides of Lecture Slides 3, [3].

Slides 2 and 3 re-iterate what was stated at the beginning of class in Section 1.

Slides 3 and 4 indicate that different processes have memory spaces which are distinct and protected from other processes. However, a process may contain multiple threads which may share resources (e.g., memory!).

Slide 5 re-iterates that care must be taken by the process/programmer when managing resources for multiple threads within a single process.

References

[1] Suresh Jagannathan, Course Lecture. Purdue University, CS 39000-PCP. 17 January 2012.

[2] Suresh Jagannathan, Lecture Slides 2. Purdue University, CS 39000-PCP. http://www.cs.purdue.edu/homes/suresh/390C-Spring2012/lectures/Lecture-2.pdf. 17 January 2012.

[3] Suresh Jagannathan, Lecture Slides 3. Purdue University, CS 39000-PCP. http://www.cs.purdue.edu/homes/suresh/390C-Spring2012/lectures/Lecture-3.pdf. 17 January 2012.

[4] Suresh jagannathan, Assignment # 1. Purdue University, CS 39000-PCP. http://www.cs.purdue.edu/homes/suresh/390C-Spring2012/hw/hw1.html. 17 January 2012.

[5] Racket. http://racket-lang.org. 17 January 2012.

[6] Essentials of Programming Languages, Friedman, Wand, Haynes (2001).

[7] Continuation-based Multiprocessing, Wand (1980).

[8] Continuations and Threads: Expressing Machine Concurrency Directly in Advanced Languages (1997).

[9] Continuations and Concurrency, Hieb and Dybvig (1990).

Purdue University, West Lafayette IN 47907 USA
suresh@cs.purdue.edu
mrbarret@purdue.edu