Fall 2004 Handout 5

Time a program: pp. 69, 92

```
/* This program is looper.c.  It takes a long time to do nothing. */
#include <stdio.h>
#include <stdlib.h>

void f(void);
void g(void);

int main()
{
    f();
    g();
    printf("All done.\n");
    return EXIT_SUCCESS;
}

void f(void)
{
    long i;

    for (i = 0; i < 1000000; ++i) {
    }
}

void g(void)
{
    long i;

    for (i = 0; i < 10000000; ++i) {
    }
}
```

The word **`time`** is a keyword in the C shell language, just like **`set`** and **`setenv`**; see p. 20 of **`csh`**(1). To run the **`time`** program, you must therefore specify its full pathname. **`time`** sends its output to **`stderr`** to avoid mixing it with the **`stdout`** of the program being timed.

```
1$ gcc -o looper looper.c
2$ /usr/bin/time looper > /dev/null
real    2.1
user    0.8
sys     0.0
```

The **`sys`** time is the amount of CPU time spent executing the program's system calls. For example, the **`printf`** function ultimately calls the Unix system call **`write`**, and the **`scanf`** function ultimately calls **`read`**; see pp. 202−204. The **`user`** time is the amount of CPU time spent executing all code other than Unix system calls.

Profile: p. 285

```
1$ gcc -p -o looper looper.c
2$ looper                                                    Create mon.out.
All done.
3$ ls -l mon.out
4$ prof looper mon.out

Profile listing generated Sun Oct 22 22:16:00 2004 with:
   prof looper mon.out
----------------------------------------------------------------------------
*  -p[rocedures] using pc-sampling;                                        *
*  sorted in descending order by total time spent in each procedure;       *
*  unexecuted procedures excluded                                          *
----------------------------------------------------------------------------
Each sample covers 4.00 byte(s) for 0.21% of 0.4551 seconds

%time     seconds  cum %   cum sec  procedure (file)

 90.8      0.4131   90.8      0.41 g (<stripped>)
  9.2      0.0420  100.0      0.46 f (<stripped>)

5$ prof -v looper mon.out | lpr -g        Print a graph (not in our version of Unix).
```

▼ **Homework 5.1: which is the faster way to loop ten million times?**

```
for (i = 0; i < 10000000; ++i) {      /* ascending */

for (i = 10000000; i > 0; --i) {      /* descending */
```

Write a C program with two functions named **ascending** and **descending**, each containing the empty loop shown above and nothing else. Which function takes less time? Why? Do you get the same profile each time you run the program? Hand in a profile.

What is added to the symbol table of a **.o** file when you create it with the **-p** option? Compile your program with and without the **-p** option, and use **nm** to save the two symbol tables in two temporary files. Then use **comm** to find all the words in the last column of one file that are not in the last column of the second file.

▲

How much memory will a process occupy?

ls -l does not show you how much memory a program will occupy as it runs; use the **size** command instead. **strip** decreases the size output by **ls -l**, but has no effect on the size output by **size**.

The memory occupied by a process is divided into *segments*. The **text** segment holds the executable instructions. The **data** segment holds explicitly initialized data. The **bss** segment holds data that is implicitly initialized to zero. It's named after an old IBM 7090 pseudo-op: ‘‘block started by symbol’’. Our **gcc** compiler uses it to hold **static** data with no explicit initialization. Its size is **0** in the above example because **moon** had no variables with declarations such as

```
static int i;                           /* implicitly initialized to 0 */

1$ cd $m46/moon
2$ gcc -c moonmain.c moonphase.c moondraw.c
3$ gcc -o moon moonmain.o moonphase.o moondraw.o -lm
```



```
4$ ls -l
-rwx------   1 mm64     users       9520 Jan  9 00:41 moon
-rw-------   1 mm64     users       1776 Jan  9 00:41 moondraw.o
-rw-------   1 mm64     users       1416 Jan  9 00:41 moonmain.o
-rw-------   1 mm64     users       2768 Jan  9 00:41 moonphase.o

5$ size moonmain.o moonphase.o moondraw.o moon
moonmain.o: 906 + 0 + 0 = 906
moonphase.o: 2259 + 0 + 0 = 2259
moondraw.o: 1272 + 0 + 0 = 1272
moon: 4592 + 472 + 352 = 5416
```

Each segment is divided into *pages*. On a given machine, every page has the same size:

```
6$ pagesize                              How many bytes in a page of memory?
8192
```

▼ Homework 5.2: find the size of the segments of a C program

Use **pagesize** to find the page size on your computer. Is there a function that a C program could call to do this? Write a little C program and see if the sizes of the three segments are multiples of the page size. If not, why not?

▲

▼ Homework 5.3: how big are the segments of a program?

How big are the segments of a C program that does nothing except

```
printf ("hello\n");
```

Will the **-O** option (“optimize”) of **gcc** make it smaller?

Change the **printf** to

```
write (1, "hello\n", 6);                       /* KP pp. 201-204 */
```

and remove the **#include <stdio.h>**. Verify that the program still produces the same output and measure its size again.

▲

Dynamic memory allocation

size shows how much memory a process occupies when it starts running. But a process can get bigger as it runs by calling **malloc**; see **end**(3) for other ways in which a process can grow.

```
1$ ps -o vsz,comm | more
  VSZ COMMAND
1.89M -csh (csh)
1.79M sh -c ps -o vsz,comm
2.33M vi size.ms
```

```
/* This C program is named little.c. */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    const char *const command =
        "ps -o vsz,comm | awk ’NR == 1 || $2 == \"little\"’";
```



```
    char *p;

    system(command);
    p = malloc(1000 * 1024);
    system(command);

    free(p);
    return EXIT_SUCCESS;
}
```

```
2$ gcc -o little little.c
3$ little
  VSZ COMMAND
1.21M little
  VSZ COMMAND
2.20M little
```

Sample C program to run under the control of dbx

```
/* This file is primemain.c.
Print the prime numbers between 1 and 100, eight per line. */

#include <stdio.h>
#include <stdlib.h>

int isprime(int n);

main()
{
    int n;
    int i = 0;      /* count how many numbers printed so far */

    printf ("The prime numbers from 1 to 100 are:\n");

    for (n = 2; n < 100; ++n) {
        if (isprime(n)) {
            printf ("%5d", n);
            if (++i % 8 == 0) {
                printf ("\n");
            }
        }
    }

    /* If the last number was not followed by a newline, add one now. */
    if (i % 8 != 0) {
        printf ("\n");
    }
    exit (0);
}
```

```
/* This file is primeis.c. */
int isprime(int n);

```

```
/* Return 1 if n is prime, 0 otherwise. */
int isprime(int n)
{
    int i;

    for (i = 2; i < n; ++i) {
        if (n % i == 0) {
            return 0;
        }
    }

    return 1;
}
```

```
The prime numbers from 1 to 100 are:
    2    3    5    7   11   13   17   19
   23   29   31   37   41   43   47   53
   59   61   67   71   73   79   83   89
   97
```

Run a C program under the control of dbx

Print **dbx**(1). Compile all of the **.c** files of the above program with the **-g** option of **gcc**:

1$ gcc -g -o prime primemain.c primeis.c	
2$ ls -l	*Make sure you created an executable* **prime**.
3$ dbx prime	

Display lines in the .c files

dbx will display the lines whose numbers you specify in the file that contains the **main** function. The **dbx** prompt is **(dbx)**.

(dbx) list 1	*List lines 1−12 of the file* **primemain.c**.
(dbx) list 10	*List lines 10−21 of the file* **primemain.c**.
(dbx) list 5,18	*List lines 5−18 of the file* **primemain.c**.
(dbx) list	*List the next 12 lines.*
(dbx) list	*List the next 12 lines.*
(dbx) alias l list	*Let lowercase* **l** *be an abbreviation for the word* **list** *(already done).*
(dbx) alias	*See a list of all the* **alias***'s; press* **RETURN**.
(dbx) l	*List the next 10 lines.*
(dbx) l 10	
(dbx) alias ll "list; list"	*List the next 20 lines.*
(dbx) l main	*List the first lines in the function* **main**.
(dbx) /printf	*List the next line that contains the string* **printf***, as in* **vi**.
(dbx) /	*Repeat the search.*
(dbx) /	*Repeat the search.*
(dbx) ?printf	*List the previous line that contains the string* **printf***, as in* **vi**.

`(dbx) help`	
`(dbx) help most_used`	
`(dbx) quit`	*Exit from* **`dbx`** *when you've had enough*

Change to a different file

You can see any of the source code files that constitute your program:

`(dbx) file primeis.c`	*Go to the source file* **`primeis.c`**.
`(dbx) l`	*List the first 10 lines of* **`primeis.c`**.
`(dbx) file /usr/include/stdio.h`	*Go to the source file* **`/usr/include/stdio.h`**.
`(dbx) l`	*List the first 10 lines of* **`/usr/include/stdio.h`**.
`(dbx) file primemain.c`	*Go back to the source file* **`primemain.c`**.
`(dbx) l`	*List the first 10 lines in* **`primemain.c`**.
`(dbx) file`	*if you forget what source file you're looking at*

Set and remove breakpoints

Now that we're back in the file that contains the function **`main`**, set a breakpoint at the first executable line of that function.

`(dbx) stop at 14`	*Put a breakpoint at line 14.*
`(dbx) status`	*See a numbered list of all your breakpoints.*
`(dbx) delete 2`	*Remove breakpoint number 2 (your only breakpoint).*
`(dbx) status`	*Make sure that the breakpoint is gone.*
`(dbx) stop at 14`	*Put the breakpoint back.*
`(dbx) status`	*Make sure that it's back.*
`(dbx) stop in main`	*another way to stop at the beginning of a function*

Warning: a breakpoint on a **`for`** line is not located within the loop. Put the breakpoint on the next line.

Execute and single step through the program

Now run the program. It will stop immediately because we've put a breakpoint at the first line of the function **`main`**.

`(dbx) run`	*Command line arguments and i/o redirection go here:* **<**, **>**
`(dbx) step`	*This gets us to the* **`for`** *in line 16.*
`(dbx) step`	*This gets us to the* **`if`** *in line 17.*
`(dbx) where`	*What line am I at? Also show the runtime stack.*
`(dbx) cont`	*Continue until next breakpoint.*

The **`next`** command is just like **`step`**, except that it jumps over function calls.

`(dbx) alias`	*See a list of your* **`alias`***'es.*
`(dbx) alias n next`	*Create these* **`alias`***'s if they don't already exist.*
`(dbx) alias s step`	

Print and change the values of variables once the program is under way

`(dbx) print i`	*Print the value of* **`i`** *in decimal.*
`(dbx) printf "%X", i`	*Print the value of* **`i`** *in hex. Make an* **`alias`** *for this.*
`(dbx) printf "%o", i`	*Print the value of* **`i`** *in octal.*
`(dbx) &main/10i`	*Print the first ten assembly language instructions in* **`main`**.

To automatically print the value of **`i`** whenever you **`step`**,

`(dbx) alias s "step; print i"`	*may need to parenthesize* **`i`**
`(dbx) assign i = 10`	
`(dbx) assign i = 0xFFFF`	*hexadecimal*
`(dbx) assign i = 'A'`	*Put 65 into* **`i`**.
`(dbx) print i`	
`(dbx) dump`	*Print the value of every variable in the current function.*
`(dbx) func`	*if you forget which function is the current one*
`(dbx) dump isprime`	*Print the value of every variable in another active function.*
`(dbx) print i + 100`	
`(dbx) print i & 15`	*Print the four lowest bits of* **`i`**.
`(dbx) print i>>4 & 15`	*Print the next four lowest bits of* **`i`**.
`(dbx) alias nib "print i>>4 & 15"`	*need double quotes around string with blanks*
`(dbx) nib`	
`(dbx) alias nib(expr) "print (expr) >> 4 & 15"`	*an* **`alias`** *with an argument*
`(dbx) nib(i)`	
`(dbx) alias nib(expr, n) "print (expr) >> 4*(n) & 15"`	
`(dbx) nib(i, 2)`	
`(dbx) print &i`	*Print the address of* **`i`** *in hex.*
`(dbx) print sizeof(i)`	*parentheses required*
`(dbx) whatis i`	*if you forgot that* **`i`** *is an* **`int`**.

Conditional breakpoints

Use a **`stop`** command to stop; use a **`when`** command to do other things. If you have a **`stop`** and a **`when`** at the same line number, it will do the **`when`** first and then **`stop`**.

`(dbx) delete all`	
`(dbx) status`	
`(dbx) func main`	*Make it clear which* **`n`** *you're about to mention.*
`(dbx) stop at 18 if n == 4`	
`(dbx) status`	
`(dbx) run`	*It will stop at line 18.*
`(dbx) where`	
`(dbx) print n`	**`n`** *will be 4.*

```
(dbx) stop at 18 if 4 <= n && n <= 8
(dbx) stop in isprime
(dbx) stop if n == 4
```



```
(dbx) func main
(dbx) when at 18 {print n}
(dbx) when at 18 {where; print n}
(dbx) when in isprime {where; dump}
```

Two variables with the same name

The functions **`main`** and **`isprime`** each contain a variable named **`i`**. Use a dot to specify the name of the function that contains the variable. If necessary, you can also specify the name of the file (minus the trailing **`.c`**) that contains the function.

`(dbx) when at 11 {print i}`	*the* **`i`** *in the current function*
`(dbx) when at 11 {print isprime.i}`	*function name, variable name*
`(dbx) when at 11 {print primemain.isprime.i}`	*file, function, variable*
`(dbx) when at 11 {print main.i}`	
`(dbx) when at 11 {print primemain.i}`	*if there was an* **`i`** *above* **`main`**
`(dbx) whereis i`	*see a list of all the* **`i`***'s in the program.*

If you make **`isprime`** the *current function*, then you won't need to type **`isprime`**. in front of the names of **`isprime`**'s variables. Warning: you need the **`func`** command to change the current function. The **`list`** and **`file`** commands do not change the current function.

`(dbx) func isprime`	*Make* **`isprime`** *the current function.*
`(dbx) when at 11 {print i}`	*This* **`i`** *is guaranteed to be the one in* **`isprime`**.
`(dbx) func`	*if you forget which function is the current one*

Two invocations of the same function at runtime

The **`where`** command displays the runtime stack, with **`main`** at the bottom and the most recently called function at the top. The **`print`** command will print the variables in the current function. With a dot, you can print the variables in another function on the stack. Or you can avoid the dot by traveling along the stack to the other function first.

`(dbx) stop in isprime`	
`(dbx) run`	
`(dbx) where`	*We're in* **`isprime`**, *called from* **`main`**.
`(dbx) print i`	*the* **`i`** *in* **`isprime`**
`(dbx) print main.i`	*the* **`i`** *in* **`main`**
`(dbx) func`	*It prints* **`isprime`**.
`(dbx) up`	*Travel towards* **`main`**.
`(dbx) func`	*It prints* **`main`**.
`(dbx) print i`	*the* **`i`** *in* **`main`**
`(dbx) down`	*Travel away from* **`main`**.
`(dbx) func`	*It prints* **`isprime`**.
`(dbx) print i`	*the* **`i`** *in* **`isprime`**

Although our prime program does not illustrate this, a function in C can call itself. When this happens, the function will be on the runtime stack twice. Use **`up`** and **`down`** to travel to each invocation of the function to examine its variables.

Trace

`(dbx) func main`	
`(dbx) trace i`	
`(dbx) run`	*Print line number, old value, new value whenever* **`i`** *changes.*

`(dbx) trace i if 3 <= i && i <= 6`	*See if this works correctly.*
`(dbx) trace in isprime`	*Print message when entering the function* **`isprime`**.
`(dbx) file primemain.c; trace 18`	*Print message whenever line 18 is executed.*

Print char's, arrays, and strings

```
char c = 'A';
char *p = "hello";
char s[] = "goodbye";
```

`(dbx) print c`	*Prints* **`'A'`**.
`(dbx) printf "%d", c`	*Prints* **`65`**.
`(dbx) print (int)c`	*Prints* **`65`**.

`(dbx) print p`	*Prints value of* **`p`** *(i.e., address of the* **`h`***) in hex; then print* **`"hello"`**
`(dbx) print *p`	*Prints* **`'h'`**.
`(dbx) print p[1]`	*Prints* **`'e'`**.

In C, **`s`** means the address of the array. In **`dbx`**, however, **`s`** means the *values* of the array's elements.

`(dbx) print s`	*Print the value of every array element: the whole string.*
`(dbx) print &s`	*Print the address of* **`s[0]`**.
`(dbx) print s[3]`	*Print the* **`'d'`** *in* **`goodbye`**.

Examine a core dump

`1$ prime`	*It dumps core.*
`2$ ls -l`	*Make sure there's a* **`core`** *file.*
`3$ dbx prime core`	**`core`** *is the default*
`(dbx) where`	*the first* **`dbx`** *command to use when examining a core dump.*

Here's a program that will always dump core:

```
#include <stdio.h>

main()
{
    char *p = NULL;

    for (;;) {
        *p++ = '\0';
    }
}
```

And if that doesn't work, see pp. 225−229, **`kill`**(2), **`signal`**(3):

```
/* Excerpts from the file /usr/include/signal.h. */
#define SIGQUIT    3    /* quit */
#define SIGKILL    9    /* kill (cannot be caught or ignored) */
```

```
#include <sys/types.h>
```



```
#include <signal.h>

main()
{
    kill(getpid(), SIGQUIT);     /* dump core */
}
```

Create a .dbxinit file

```
#This file is $HOME/.dbxinit

alias l list   #if these aliases are not already created
alias n next
alias s step
alias d delete

#Output one nibble (i.e., 4 consecutive bits) of an expression.
#The first argument is the expression, the second is the number of
#the nibble.  Nibble number 0 is the least significant.
alias nib(expr, n) "print (expr) >> 4*(n) & 15"

list main      #List first lines of main function.
```

□