CMSC 216
Introduction to Computer Systems
Lecture 19
Process Control and
System-Level I/O
Section 16.3, Reek

TIME MEASUREMENT (CONT.)
Date and time functions

clock_t clock(void);
- returns the process time since the start of program execution
- to convert to time, divide by CLOCK_PER_SEC (also in time.h)

_time_t time(_time_t *val);
- fills val with the current time (in an implementation-dependent format)

char *ctime(_time_t *val);
- returns a character representation of the passed time
- Example: Sun Oct 28 09:02:48 2007

double difftime(_time_t time1, _time_t time2);
- returns the number of seconds between time1 and time2

struct _tm *gmtime(_time_t val);

struct _tm *localtime(_time_t val);
- converts a time to UTC or local time, in the form of a struct tm
Adding timing calls to your program

- **Wall time**
  
  ```c
  int gettimeofday(struct timeval *tv,
                  struct timezone *tz);
  ```
  
  - `tv` is a structure of time `tv_sec` and `tv_usec` (10^{-6} seconds)
  - `tz` is no longer used (just pass `NULL`)

- **Process time**
  
  ```c
  int getrusage(int who, struct rusage *usage);
  ```
  
  - `who` is `RUSAGE_SELF` or `RUSAGE_CHILDREN`
    - `RUSAGE_CHILDREN` is all terminated children
  - `rusage` contains fields for
    ```c
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    ```
    - and fields for various other OS statistics
Adding timing calls, cont.

- Include `<sys/time.h>` to use `gettimeofday()`.
- Include `<sys/time.h>`, `<sys/resource.h>`, and `<unistd.h>` to use `getrusage()`.
Example measuring time

```c
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int main() {
    struct rusage start_ru, end_ru;
    struct timeval start_wall, end_wall;

    gettimeofday(&start_wall, NULL);
    getrusage(RUSAGE_SELF, &start_ru);

    /* code to time */

    gettimeofday(&end_wall, NULL);
    getrusage(RUSAGE_SELF, &end_ru);

    /* compute difference */

    return 0;
}
```
Calculating the difference of 2 times

- Not trivial, as two fields are involved in each struct timeval, but not too complicated
- Example (calculating end - start):

```c
struct timeval tv_delta(struct timeval start, struct timeval end)
{
    struct timeval delta = end;
    delta.tv_sec -= start.tv_sec;
    delta.tv_usec -= start.tv_usec;
    if (delta.tv_usec < 0) {
        delta.tv_usec += 1000000;
        delta.tv_sec--;
    }
    return delta;
}
```
Section 13.3, Reek

FUNCTION POINTERS
Function Pointers

• Each function is located somewhere in memory; this means we can create a pointer to it

• Declared like this:
  
  \[ \text{\texttt{void}} \, (*\text{fp})(\text{int}); \]
  
  • \text{fp} is a pointer to a function that returns \texttt{void} and has a single parameter (which is an \texttt{int})

  \[ \text{\texttt{int}} \, *(*\text{fp2})(\text{char} \, * \, , \, \text{int}); \]
  
  • \text{fp2} is a pointer to a function that returns a pointer to an \texttt{int}, and has 2 parameters (a pointer to \texttt{char}, and an \texttt{int})
void print_decimal(unsigned int i) {
    printf("%u\n", i);
}
void print_hex(unsigned int i) {
    printf("%x\n", i);
}
void print_octal(unsigned int i) {
    printf("%o\n", i);
}

...  

void (*fp)(unsigned int);
fp = print_hex;
fp(16); /* prints "10" */
fp = &print_octal;
fp(16); /* prints "20" */
fp = print_decimal;
(*fp)(16); /* prints "16" */
Using **typedef** with function pointers

• To make things a bit more clear, we can use **typedef** to create a specific function pointer type

• Example:

```c
typedef char *(*Str_func)(char *);

char *strdup(char *str) { ... }
...

Str_func sf = strdup;
char *copy = sf(str);
```
Understanding complex declarations

• Even people who've programmed in C for a long while may have trouble deciphering this declaration:
  ```c
  int *(*f[8])(char *);
  ```

• The program `cdecl` can be of use here:
  ```bash
  $ cdecl
  Type `help' or `?' for help
  cdecl> explain int *(*f[8])(char *);
  declare f as array 8 of pointer to function (pointer to char) returning pointer to int
  ```

• In other words, `f` is an array containing 8 function pointers, each of which can point to a function that takes a `char *` as an argument and returns an `int *`
Parts of Sections 2.1-2.4, Bryant and O'Hallaron

**DATA REPRESENTATION**
Representing characters

• We need:
  – to be able to represent common characters
  – to have standards so computers can interoperate

• Common formats
  – ASCII
    • is the most commonly used character code
    • uses 7 bits for characters (stored in 8 bits normally)
  – EBCDIC
    • an 8-bit code, used now only by some IBM mainframes
  – UNICODE
    • a family of encodings - 8, 16, and 32 bits per character
    • allows a greater variety of characters
    • is able to represent virtually any character in use today in any language, and some no longer in use
ASCII

• Represents normal characters on US keyboards
  – A-Z (the characters numbered 65-90)
  – a-z (97-122)
  – 0-9 (48-57)
  – space (32)
  – control characters (0-31, 127)
    • the first 26 (after 0) of the 33 ASCII control characters have names Ctrl-A - Ctrl-Z
      • for example, ASCII character 13, Ctrl-M, is CR (carriage return) (\r in C); ASCII char. 9, Ctrl-I, is HT (horizontal tab) (\t in C)
    – punctuation: !@#$%^&*()_+-=[]{}|\;:"'<>?,./ (the remaining characters)
• The UNIX command "man ascii" shows the ASCII character set
UNICODE

• Different representations

• UTF-32: a 32-bit representation of all characters
  – all characters are the same size
  – uses lots of space (twice as much as UTF-16 for most things, four times as much as ASCII for many things)

• UTF-16: a 16-bit representation of characters
  – some characters are stored in two-character forms
  – is popular since most things can be represented in 16 bits

• UTF-8: an 8-bit representation of characters
  – provides backwards compatibility with ASCII
    • the low 7 bits are exactly ASCII
    • if the high bit is on it indicates part of UNICODE extensions
  – popular for web and other applications
Representing unsigned integers

- **All data** is stored in binary
  - all digits are 0 or 1
- In an unsigned number every bit position $i$ represents the value $2^i$, where $i$ is 0 for the rightmost bit. The value of a number is the sum of the values of the bit positions containing a 1.
- Example bit position values for an 8-bit number:

<table>
<thead>
<tr>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
</table>

- The range of values that can be stored is 0 to $2^n - 1$, where $n$ is the number of bits used
  - for example, using 16 bits, the numbers 0 to 65,535 can be represented
Representing signed integers

- Signed integers are usually stored using two's complement
  - the leftmost bit indicates if a number is positive (0) or negative (1)
- To get the two's complement representation of a negative value, flip all the bits of the positive value and add 1
- Two's complement allows easy addition of positive and negative numbers (just ignore overflow)
- The range of values that can be stored is $-2^{n-1}$ to $2^{n-1}-1$ for $n$ bits
  - for example, using 16 bits, the numbers -32,768 to 32,767 can be represented
- Two's complement isn't the same thing as an unsigned number with a sign bit
  - -5 is $\sim(5) + 1 = 11111010 + 1 = 11111011$, not 10000101