- # Timing a Program:
  - Run a tool to time program execution
    - E.g., Unix **`time`** command

      - Real: Wall-clock time between program invocation and termination
      - • User: CPU time spent executing the program
      - • System: CPU time spent within the OS on the program's behalf

```
$ time sort < bigfile.txt > output.txt
real    0m12.977s
user    0m12.860s
sys     0m0.010s
```

Timing Parts of a Program:

Call a function to **compute wall-clock time** consumed

- E.g., Unix `gettimeofday()` function (time since Jan 1, 1970)

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

struct timeval startTime;
struct timeval endTime;
double wallClockSecondsConsumed;

gettimeofday(&startTime, NULL);
<execute some code here>
gettimeofday(&endTime, NULL);
wallClockSecondsConsumed =
   endTime.tv_sec - startTime.tv_sec +
   1.0E-6 * (endTime.tv_usec - startTime.tv_usec);
```

Call a function to compute CPU time consumed

- E.g. `clock()` function

```
#include <time.h>

clock_t startClock;
clock_t endClock;
double cpuSecondsConsumed;

startClock = clock();
<execute some code here>
endClock = clock();
cpuSecondsConsumed =
  ((double)(endClock - startClock)) / CLOCKS_PER_SEC;
```

Identifying Hot Spots

- statistics about your program's execution
  - How much time did execution of a particular function take?
  - How many times was a particular function called?
  - How many times was a particular line of code executed?
  - Which lines of code used the most time.
- How?
  - Use an execution profiler
  - • Example: `gprof` (GNU Performance Profiler)

Example program for GPROF analysis

- Sort an array of 10 million random integers
- Artificial: consumes much CPU time, generates no output

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

enum {MAX_SIZE = 10000000};
int a[MAX_SIZE]; /* Too big to fit in stack! */

void fillArray(int a[], int size)
{  int i;
   for (i = 0; i < size; i++)
      a[i] = rand();
}

void swap(int a[], int i, int j)
{   int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

```
int partition(int a[], int left, int right)
{  int first = left-1;
   int last = right;
   for (;;)
   {  while (a[++first] < a[right])
         ;
      while (a[right] < a[--last])
         if (last == left)
            break;
      if (first >= last)
         break;
      swap(a, first, last);
   }
   swap(a, first, right);
   return first;
}
```

```
…
void quicksort(int a[], int left, int right)
{  if (right > left)
   {  int mid = partition(a, left, right);
      quicksort(a, left, mid - 1);
      quicksort(a, mid + 1, right);
   }
}

int main(void)
{  fillArray(a, MAX_SIZE);
   quicksort(a, 0, MAX_SIZE - 1);
   return 0;
}
```

Step 1: Instrument the program

`gcc217 –pg mysort.c –o mysort`

- Adds profiling code to mysort, that is...
- "Instruments" `mysort`

Step 2: Run the program

`mysort`

- Creates file `gmon.out` containing statistics

Step 3: Create a report

`gprof mysort > myreport`

Uses `mysort` and `gmon.out` to create textual report

Step 4: Examine the report

`cat myreport`

The GPROF Report

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
84.54	2.27	2.27	6665307	0.00	0.00	partition
9.33	2.53	0.25	54328749	0.00	0.00	swap
2.99	2.61	0.08	1	0.08	2.61	quicksort
2.61	2.68	0.07	1	0.07	0.07	fillArray

Each line describes one function

- **name**: name of the function
- **%time**: percentage of time spent executing this function
- **cumulative seconds**: [skipping, as this isn't all that useful]
- **self seconds**: time spent executing this function
- **calls**: number of times function was called (excluding recursive)
- **self s/call**: average time per execution (excluding descendents)
- **total s/call**: average time per execution (including descendents)

```
/* Program to demonstrate time taken by function fun() */
#include <stdio.h>
#include <time.h>

// A function that terminates when enter key is pressed
void fun()
{
    printf("fun() starts \n");
    printf("Press enter to stop fun \n");
    while(1)
    {
        if (getchar())
            break;
    }
    printf("fun() ends \n");
}

// The main program calls fun() and measures time taken by fun()
int main()
{
    // Calculate the time taken by fun()
    clock_t t;
    t = clock();
    fun();
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds

    printf("fun() took %f seconds to execute \n", time_taken);
    return 0;
}
```