Make and Gprof

Prof. David August

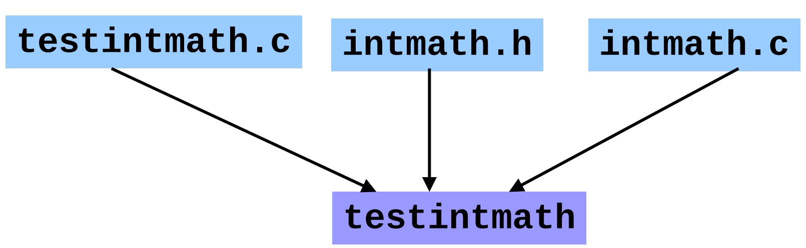
COS 217

Goals of Today's Lecture

- Overview of two important programming tools
  - Make for compiling and linking multi-file programs
  - Gprof for profiling to identify slow parts of the code
- Make
  - Overview of compilation process
  - Motivation for using Makefiles
  - Example Makefile, refined in five steps
- Gprof
  - Timing, instrumenting, and profiling
  - GNU Performance Profiler (Gprof)
  - Running gprof and understanding the output

Example of a Three-File Program

- Program divided into three files
  - **`intmath.h`**: interface, included in **`intmath.c`** and **`testintmath.c`**
  - **`intmath.c`**: implementation of math functions
  - **`testintmath.c`**: implementation of tests of the math functions
- Creating the **`testintmath`** binary executable




```
gcc -Wall -ansi -pedantic -o testintmath testintmath.c intmath.c
```


Many Steps, Under the Hood

- Preprocessing (**`gcc -E intmath.c > intmath.i`**)
  - Removes preprocessor directives
  - Produces **`intmath.i`** and **`testintmath.i`**
- Compiling (**`gcc -S intmath.i`**)
  - Converts to assembly language
  - Produces **`intmath.s`** and **`testintmath.s`**
- Assembling (**`gcc -c intmath.s`**)
  - Converts to machine language with unresolved directives
  - Produces the **`intmath.o`** and **`testintmath.o`** binaries
- Linking (**`gcc -o testintmath testintmath.o intmath.o -lc`**)
  - Creates machine language exectutable
  - Produces the **`testintmath`** binary

Motivation for Makefiles

- Typing at command-line gets tedious
  - Long command with compiler, flags, and file names
  - Easy to make a mistake
- Compiling everything from scratch is time-consuming
  - Repeating preprocessing, compiling, assembling, and linking
  - Repeating these steps for every file, even if just one has changed
- UNIX Makefile tool
  - Makefile: file containing information necessary to build a program
    - Lists the files as well as the dependencies
  - Recompile or relink only as necessary
    - When a dependent file has changed since command was run
    - E.g. if intmath.c changes, recompile intmath.c but not testintmath.c
  - Simply type “make”, or “make –f <makefile_name>”

Main Ingredients of a Makefile

- Group of lines
  - Target: the file you want to create
  - Dependencies: the files on which this file depends
  - Command: what to execute to create the file (after a TAB)
- Examples

```
testintmath: testintmath.o intmath.o
   gcc –o testintmath testintmath.o intmath.o
```

```
intmath.o: intmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o intmath.o intmath.c
```


Complete Makefile #1

- Three groups
  - testintmath: link testintmath.o and intmath.o
  - testintmath.o: compile testintmath.c, which depends on intmath.h
  - intmath.o: compile intmath.c, which depends on intmath.h

```
testintmath: testintmath.o intmath.o
   gcc –o testintmath testintmath.o intmath.o

testintmath.o: testintmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o testintmath.o testintmath.c

intmath.o: intmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o intmath.o intmath.c
```

Adding Non-File Targets

- Adding useful shortcuts for the programmer
  - “**make all**”: create the final binary
  - “**make clobber**”: delete all temp files, core files, binaries, etc.
  - “**make clean**”: delete all binaries
- Commands in the example
  - “**rm –f**”: remove files without querying the user
  - Files ending in ‘**~**’ and starting/ending in ‘#’’ are temporary files
  - “**core**” is a file produced when a program “dumps core”

```
all: testintmath
clobber: clean
    rm -f *~ \#*\# core
clean:
    rm -f testintmath *.o
```

Complete Makefile #2

```
# Build rules for non-file targets
all: testintmath
clobber: clean
   rm -f *~ \#*\# core
clean:
   rm -f testintmath *.o

# Build rules for file targets
testintmath: testintmath.o intmath.o
   gcc –o testintmath testintmath.o intmath.o
testintmath.o: testintmath.c intmath.h
   gcc -Wall -ansi -pedantic -c -o testintmath.o testintmath.c
intmath.o: intmath.c intmath.h
   gcc -Wall -ansi -pedantic -c -o intmath.o intmath.c
```

Useful Abbreviations

- Abbreviations
  - Target file: $@
  - First item in the dependency list: $<
- Example

```
testintmath: testintmath.o intmath.o
   gcc –o testintmath testintmath.o intmath.o
```

```
testintmath: testintmath.o intmath.o
   gcc –o $@ $< intmath.o
```


Complete Makefile #3

```
# Build rules for non-file targets
all: testintmath
clobber: clean
    rm -f *~ \#*\# core
clean:
    rm -f testintmath *.o

# Build rules for file targets
testintmath: testintmath.o intmath.o
   gcc –o $@ $< intmath.o
testintmath.o: testintmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o $@ $<
intmath.o: intmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o $@ $<
```

Useful Pattern Rules: Wildcard %

- Can define a default behavior
  - Build rule: `gcc -Wall -ansi -pedantic -c -o $@ $<`
  - Applied when target ends in “.o” and dependency in “.c”

```
%.o: %.c
    gcc -Wall -ansi -pedantic -c -o $@ $<
```

- Can omit command clause in build rules (even some rules!)

```
testintmath: testintmath.o intmath.o
   gcc –o $@ $< intmath.o
testintmath.o: testintmath.c intmath.h

intmath.o: intmath.c intmath.h
```

Macros for Compiling and Linking

- Make it easy to change which compiler is used
  - Macro: **`CC = gcc`**
  - Usage: **`$(CC) -o $@ $< intmath.o`**
- Make it easy to change the compiler flags
  - Macro: **`CFLAGS = -Wall -ansi –pedantic`**
  - Usage: **`$(CC) $(CFLAGS) -c -o $@ $<`**

```
CC = gcc
# CC = gccmemstat

CFLAGS = -Wall -ansi -pedantic
# CFLAGS = -Wall -ansi -pedantic -g
# CFLAGS = -Wall -ansi -pedantic -DNDEBUG
# CFLAGS = -Wall -ansi -pedantic -DNDEBUG -O3
```


Sequence of Makefiles (see Web)

1. Initial Makefile with file targets
   testintmath, testintmath.o, intmath.o
2. Adding non-file targets
   all, clobber, and clean
3. Adding abbreviations
   $@ and $<
4. Adding pattern rules
   %.o: %.c
5. Adding macros
   CC and CFLAGS

References on Makefiles

- Brief discussion in the King book
  - Section 15.4 (pp. 320-322)
- GNU make
  - http://www.gnu.org/software/make/manual/html_mono/make.html
- Cautionary notes
  - Don't forget to use a TAB character, rather than blanks
  - Be careful with how you use the "**`rm –f`**" command

Timing, Instrumenting, Profiling

- How slow is the code?
  - How long does it take for certain types of inputs?
- Where is the code slow?
  - Which code is being executed most?
- Why is the code running out of memory?
  - Where is the memory going?
  - Are there leaks?
- Why is the code slow?
  - How imbalanced is my hash table or binary tree?

Timing

- Most shells provide tool to time program execution
  - E.g., bash "**time**" command

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

- Breakdown of time
  - Real: elapsed time between invocation and termination
  - User: time spent executing the program
  - System: time spent within the OS on the program's behalf
- But, which *parts* of the code are the most time consuming?

Instrumenting

- Most operating systems provide a way to get the time
  - e.g., UNIX "**gettimeofday**" command

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

struct timeval start_time, end_time;

gettimeofday(&start_time, NULL);
  <execute some code here>
gettimeofday(&end_time, NULL);

float seconds = end_time.tv_sec - start_time.tv_sec +
     1.0E-6F * (end_time.tv_usec - start_time.tv_usec);
```


Profiling

- Gather statistics about your program's execution
  - e.g., how much time did execution of a function take?
  - e.g., how many times was a particular function called?
  - e.g., how many times was a particular line of code executed?
  - e.g., which lines of code used the most time?
- Most compilers come with profilers
  - e.g., **pixie** and **gprof**
- Gprof (GNU Performance Profiler)
  - `gcc –Wall –ansi –pedantic –pg –o intmath.o intmath.c`

Profiler Basics

- Profiler is just a tool
  - Only as good as its user
  - Can help find hotspots, but ***you*** must analyze them
- Analysis includes
  - Deciding to do nothing
  - Changing algorithm
  - Changing low-level details
  - Knowing when to stop – Amdahl's law
- Process
  - Write code
  - Make sure it's correct, verify correctness, test correctness
  - Run profiler
  - Possibly "optimize" code
  - Make sure it's correct, verify correctness, test correctness

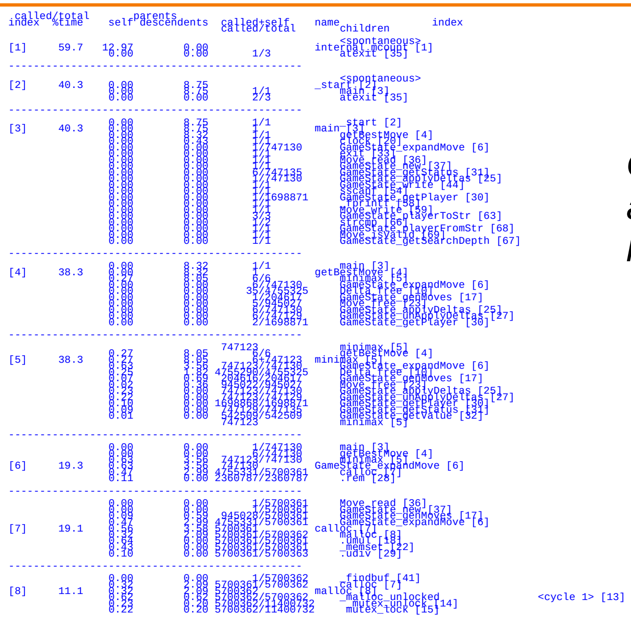
Gprof (GNU Performance Profiler)

- Instrumenting the code
  - `gcc –Wall –ansi –pedantic -pg –o intmath.o intmath.c`
- Running the code (e.g., **`testintmath`**)
  - Produces output file **`gmon.out`** containing statistics
- Printing a human-readable report from **`gmon.out`**
  - **`gprof testintmath > gprofreport`**

Two Main Outputs of Gprof

- Call graph profile: detailed information per function
  - Which functions called it, and how much time was consumed?
  - Which functions it calls, how many times, and for how long?
  - We won't look at this output in any detail…
- Flat profile: one line per 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 ms/call: average time per execution (excluding descendents)
  - total ms/call: average time per execution (including descendents)

Call Graph Output



Complex format at the beginning… let's skip for now.

Flat Profile

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked [12]
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expandMove [6]
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock [14]
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock [15]
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlocked [13]
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc [24]
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas [25]
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]
1.0	22.21	0.22	747129	0.00	0.00	GameState_unApplyDeltas [27]
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPlayer [30]
0.4	22.61	0.09	747135	0.00	0.00	GameState_getStatus [31]
0.3	22.68	0.07	204617	0.00	0.00	GameState_genMoves [17]
0.1	22.70	0.02	945027	0.00	0.00	Move_free [23]
0.0	22.71	0.01	542509	0.00	0.00	GameState_getValue [32]
0.0	22.71	0.00	104	0.00	0.00	_ferror_unlocked [357]
0.0	22.71	0.00	64	0.00	0.00	_realbufend [358]
0.0	22.71	0.00	54	0.00	0.00	nvmatch [60]
0.0	22.71	0.00	52	0.00	0.00	_doprnt [42]
0.0	22.71	0.00	51	0.00	0.00	memchr [61]
0.0	22.71	0.00	51	0.00	0.00	printf [43]
0.0	22.71	0.00	13	0.00	0.00	_write [359]
0.0	22.71	0.00	10	0.00	0.00	_xflsbuf [360]
0.0	22.71	0.00	7	0.00	0.00	_memcpy [361]
0.0	22.71	0.00	4	0.00	0.00	.mul [62]
0.0	22.71	0.00	4	0.00	0.00	___errno [362]
0.0	22.71	0.00	4	0.00	0.00	_fflush_u [363]
0.0	22.71	0.00	3	0.00	0.00	GameState_playerToStr [63]
0.0	22.71	0.00	3	0.00	0.00	_findbuf [41]

Second part of profile looks like this; it's the simple (i.e.,useful) part; corresponds to the "prof" tool

Overhead of Profiling

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	**12.97**	**12.97**				**internal_mcount**
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked
4.4	**15.04**	**0.99**				**_mcount (693)**
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expa
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlo
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_appl
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]
1.0	22.21	0.22	747129	0.00	0.00	GameState_unAp
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPl
0.4	22.61	0.09	747135	0.00	0.00	GameState_getSt

Malloc/calloc/free/...

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	**14.05**	**1.08**	**5700352**	**0.00**	**0.00**	**_free_unlocked [12]**
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expandMove
2.5	**17.67**	**0.56**	**5700361**	**0.00**	**0.00**	**calloc [7]**
2.1	**18.14**	**0.47**	**11400732**	**0.00**	**0.00**	**_mutex_unlock [14]**
1.9	**18.58**	**0.44**	**11400732**	**0.00**	**0.00**	**mutex_lock [15]**
1.9	**19.01**	**0.43**	**5700361**	**0.00**	**0.00**	**_memset [22]**
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	**19.85**	**0.41**	**5157853**	**0.00**	**0.00**	**cleanfree [19]**
1.4	**20.17**	**0.32**	**5700366**	**0.00**	**0.00**	**_malloc_unlocked [13]**
1.4	**20.49**	**0.32**	**5700362**	**0.00**	**0.00**	**malloc [8]**
1.3	**20.79**	**0.30**	**5157847**	**0.00**	**0.00**	**_smalloc [24]**
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	**21.54**	**0.23**	**5700352**	**0.00**	**0.00**	**free [9]**
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas
1.0	**21.99**	**0.22**	**5157845**	**0.00**	**0.00**	**realfree [26]**
1.0	22.21	0.22	747129	0.00	0.00	GameState_unApplyDeltas
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPlayer
0.4	22.61	0.09	747135	0.00	0.00	GameState_getStatus
0.3	22.68	0.07	204617	0.00	0.00	GameState_genMoves [17]

expandMove

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked [12]
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	**17.11**	**0.63**	**747130**	**0.00**	**0.01**	**GameState_expandMove**
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock [14]
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock [15]
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlocked [13]
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc [24]
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]

May be worthwhile to optimize this routine

Don't Even *Think* of Optimizing These

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
57.1	12.97	12.97				internal_mcount [1]
4.8	14.05	1.08	5700352	0.00	0.00	_free_unlocked [12]
4.4	15.04	0.99				_mcount (693)
3.5	15.84	0.80	22801464	0.00	0.00	_return_zero [16]
2.8	16.48	0.64	5700361	0.00	0.00	.umul [18]
2.8	17.11	0.63	747130	0.00	0.01	GameState_expandMove [6]
2.5	17.67	0.56	5700361	0.00	0.00	calloc [7]
2.1	18.14	0.47	11400732	0.00	0.00	_mutex_unlock [14]
1.9	18.58	0.44	11400732	0.00	0.00	mutex_lock [15]
1.9	19.01	0.43	5700361	0.00	0.00	_memset [22]
1.9	19.44	0.43	1	430.00	430.00	.div [21]
1.8	19.85	0.41	5157853	0.00	0.00	cleanfree [19]
1.4	20.17	0.32	5700366	0.00	0.00	_malloc_unlocked <cycle 1> [13]
1.4	20.49	0.32	5700362	0.00	0.00	malloc [8]
1.3	20.79	0.30	5157847	0.00	0.00	_smalloc <cycle 1> [24]
1.2	21.06	0.27	6	45.00	1386.66	minimax [5]
1.1	21.31	0.25	4755325	0.00	0.00	Delta_free [10]
1.0	21.54	0.23	5700352	0.00	0.00	free [9]
1.0	21.77	0.23	747130	0.00	0.00	GameState_applyDeltas [25]
1.0	21.99	0.22	5157845	0.00	0.00	realfree [26]
1.0	22.21	0.22	747129	0.00	0.00	GameState_unApplyDeltas [27]
0.5	22.32	0.11	2360787	0.00	0.00	.rem [28]
0.4	22.42	0.10	5700363	0.00	0.00	.udiv [29]
0.4	22.52	0.10	1698871	0.00	0.00	GameState_getPlayer [30]
0.4	22.61	0.09	747135	0.00	0.00	GameState_getStatus [31]
0.3	22.68	0.07	204617	0.00	0.00	GameState_genMoves [17]
0.1	22.70	0.02	945027	0.00	0.00	Move_free [23]
0.0	22.71	0.01	542509	0.00	0.00	GameState_getValue [32]
0.0	22.71	0.00	104	0.00	0.00	_ferror_unlocked [357]
0.0	22.71	0.00	4	0.00	0.00	_thr_main [367]
0.0	22.71	0.00	3	0.00	0.00	GameState_playerToStr [63]
0.0	22.71	0.00	2	0.00	0.00	strcmp [66]
0.0	22.71	0.00	1	0.00	0.00	GameState_getSearchDepth [67]
0.0	22.71	0.00	1	0.00	0.00	GameState_new [37]
0.0	22.71	0.00	1	0.00	0.00	GameState_playerFromStr [68]
0.0	22.71	0.00	1	0.00	0.00	GameState_write [44]
0.0	22.71	0.00	1	0.00	0.00	Move_isValid [69]
0.0	22.71	0.00	1	0.00	0.00	Move_read [36]
0.0	22.71	0.00	1	0.00	0.00	Move_write [59]
0.0	22.71	0.00	1	0.00	0.00	check_nlspath_env [46]
0.0	22.71	0.00	1	0.00	430.00	clock [20]
0.0	22.71	0.00	1	0.00	0.00	exit [33]
0.0	22.71	0.00	1	0.00	8319.99	getBestMove [4]
0.0	22.71	0.00	1	0.00	0.00	getenv [47]
0.0	22.71	0.00	1	0.00	8750.00	main [3]
0.0	22.71	0.00	1	0.00	0.00	mem_init [70]
0.0	22.71	0.00	1	0.00	0.00	number [71]
0.0	22.71	0.00	1	0.00	0.00	scanf [53]

Using a Profiler

- Test your code as you write it
  - It is very hard to debug a lot of code all at once
  - Isolate modules and test them independently
  - Design your tests to cover boundary conditions
- Instrument your code as you write it
  - Include asserts and verify data structure sanity often
  - Include debugging statements (e.g., #ifdef DEBUG and #endif)
  - You'll be surprised what your program is really doing!!!
- Time and profile your code **only** when you are done
  - Don't optimize code unless you have to (you almost never will)
  - Fixing your algorithm is almost always the solution
  - Otherwise, running optimizing compiler is usually enough

Summary

- Two valuable UNIX tools
  - Make: building large program in pieces
  - Gprof: profiling a program to see where the time goes
- "Always" use make, selectively use gprof
  - A little thinking saves a lot of effort
  - Extra performance not always achievable
  - Understand concept of diminishing returns
    - When is being lazy the right choice

Travel Time and Time Travel

- You plan to visit a friend in Turkey
- Concorde to Paris + 737 to Istanbul = $3500
- 747 to Paris + 737 to Istanbul = $1200

Equipment	New York to Paris	Paris to Istanbul	Total
747 + 737	8 Hours	4 Hours	12 Hours
SST + 737	3 Hours	4 Hours	7 Hours

- Taking the SST (which is 2.7 times faster) speeds up the overall trip by only a factor of 1.7!
- Teleporter to Paris? (Teleporter is 10^6 times faster)
- Time Machine to Paris?

Amdahl's Law

- Fraction optimized limits overall speedup

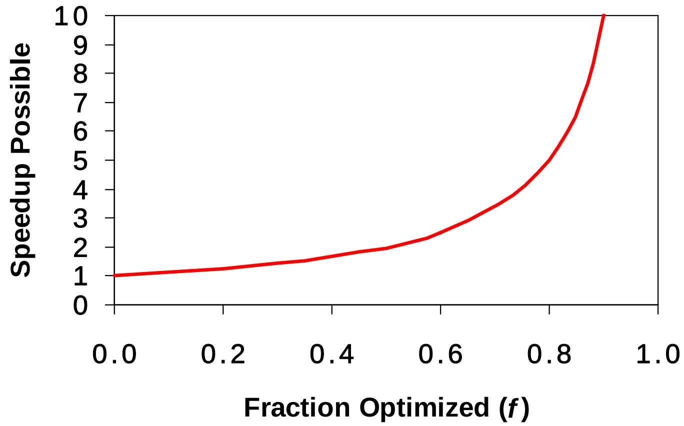
- Amdahl's Law:

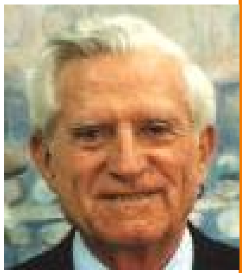
$$Speedup = \frac{1}{1 - f + \frac{f}{s}}$$

where f is fraction optimized,
s is speedup of that fraction

Amdahl's Law

Speed Enhancement is limited by fraction optimized:





$$\lim_{s \to \infty} \frac{1}{1 - f + \frac{f}{s}} = \frac{1}{1 - f}$$

where f is fraction optimized,
s is speedup of that fraction

Example Parallelism

Parallel Processing - throw more processors at problem

- 1024 parallel processors - LOTS OF MONEY!
- 90% of code is parallel (f = 0.9)
- Parallel portion speeds up by 1024 (s = 1024)
- Serial portion of code (1-f) limits speedup

$$\lim_{s \to \infty} \frac{1}{1 - f + \frac{f}{s}} = \frac{1}{1 - f}$$

- Serial portion limits to 10x speedup!