October 7th

Lab #4: **Memory Management**

Introduction

This lab looks at how processes work with memory at a low level.

Learning Objectives

1. Understand how arrays and linked lists work in memory
2. Understand how arrays work with a Memory Management Unit
3. Understand how C program loops work at a low level.

Part I Arrays and Linked Lists

1. Use `make` to compile the **`lab4In1.c`** program and run the executable file. This simple C program shows how variables and arrays are allocated on the stack. Show the output of this program in your report and use this output to draw a memory map of the stack for your report (no hand-drawn pictures; see Lab #3 for examples of memory maps). How many variables named "**`i`**" are there in this program?

2. The **`lab4In2.c`** program has errors that we may not be able to detect just by reading the code. Compile and run this program and show the output of the program in your report: what does this indicate about errors in the program?
3. Run the command
   ```
   valgrind ./lab4In2
   ```
   If your VM does not have valgrind, install it (see Lab #1 for instructions on installing programs).

   The output of valgrind will include one or more numbers with a "==" before and after like this:
   ```
   ==21283== Memcheck, a memory error detector
   ```
   The number (in the above case, `21283`) is the process id of this run of the program and so it will change if you run it again. We have seen how we can use `gdb` to debug a program when it does not work as we would expect it; `valgrind` is another tool we can use to figure out what the errors might be.

   It can help to know what a rough memory map of our process looks like when we run a program because we will get output like this:
   ```
   Process terminating with default action of signal 6 (SIGABRT)
       at 0x4E7AF47: raise (raise.c:51)
       by 0x4E7C8B0: abort (abort.c:79)
       by 0x4EC5906: __libc_message (libc_fatal.c:181)
       by 0x4F70E80: __fortify_fail_abort (fortify_fail.c:33)
       by 0x4F70E41: __stack_chk_fail (stack_chk_fail.c:29)
       by 0x108734: main
   ```
   We can guess that the `0x108734` virtual address is some place in the Code segment and the addresses starting with `0x4`*`nnnnnn`* like `0x4E7AF47` is in the shared library part of virtual memory. We could also `gdb` a program and have `valgrind` run separately and attach to the running process to debug a program with both tools. Read the man page for `valgrind` and this page for more information. Analyzing the outputs of `valgrind` does require some critical thinking, troubleshooting skills, and some educated guessing.

   In your report show the output of running `valgrind` on our executable and explain what this output reveals about our program that simply running the program does not.

4. The **`lab4In3.c`** program sets up a linked list. Show the output of this program in your report. Use this output to draw a memory map showing all the data members of each node in the linked list of your program just before it completes execution. You are free to modify this program to get more information about how the linked list is stored in memory.
   Run `valgrind` on this program and explain what additional information `valgrind` tells us.

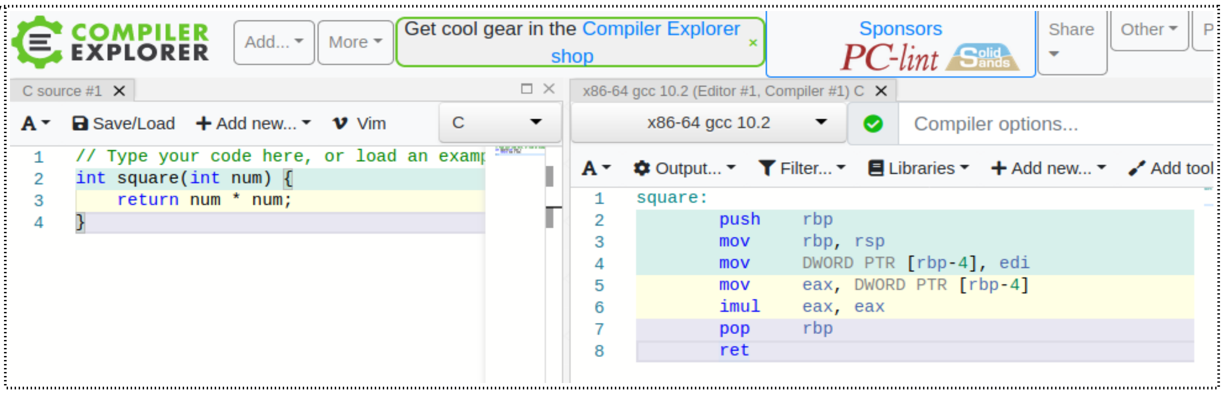
Part II C and Assembly

1. The **`lab4IIn1.c`** program has a simple C function that we can use to understand how C code is translated to assembly and machine code. In your report, list the files produced by the commands:

   ```
   make clean; ls -l; make lab4IIn1.s lab4IIn1.objdump; ls -l
   ```

   You can also run these 4 commands separately if you wish. In your report, list the contents of the files produced by this last `make` command. The `Makefile` has the actual commands to get the assembly code.

   Besides commands like `objdump` and `gcc -S` , we can also use this website: https://gcc.godbolt.org/ to disassemble C/ C++ code:

   ```
   // Type your code here, or load an examp
   int square(int num) {
       return num * num;
   }
   ```

   ```
   square:
           push    rbp
           mov     rbp, rsp
           mov     DWORD PTR [rbp-4], edi
           mov     eax, DWORD PTR [rbp-4]
           imul    eax, eax
           pop     rbp
           ret
   ```

   On the left is some source C code. If you see something else here, use what's in **`lab4IIn1.c`**.

   On the right is the assembly language code that the chosen compiler will produce from the C code. Make sure to select:
   i. the `x86-64 gcc 10.2` compiler
   This should be the version of gcc that Ubuntu 20.04 currently uses – check your VM.
   ii. "Intel asm syntax" under Output so we can see the assembly code in "Intel notation"
   Compare this assembly code with the outputs of `objdump` and `gcc -S` in the Ubuntu VM.
   In your report, highlight the differences between the three versions of assembly code for **`lab4IIn1.c`**.

You may notice a number of patterns as you work with the assembly equivalents of various functions below. One such pattern is the presence of a "function prologue" - code that is executed in most functions before the main task of that function. It can often be as simple as:

```
push rbp
mov rbp, rsp
```

These two instructions set up the stack frame for the function: push the current value in the "frame pointer" or "base pointer register" (`rbp`) onto the top of the stack (pointed to by the stack pointer register (`rsp`)); copy the value in the stack pointer register (`rsp`) into the `rbp`. Remember that on Intel CPUs a push decrements the `rsp` while the pop increments it.
See if you can spot the "function epilogue" which undoes the stack frame before returning from the function. Both of these, the prologue and epilogue, coordinate where a function "lives" in the stack and can be more complicated if we have local variables in the function – try it!

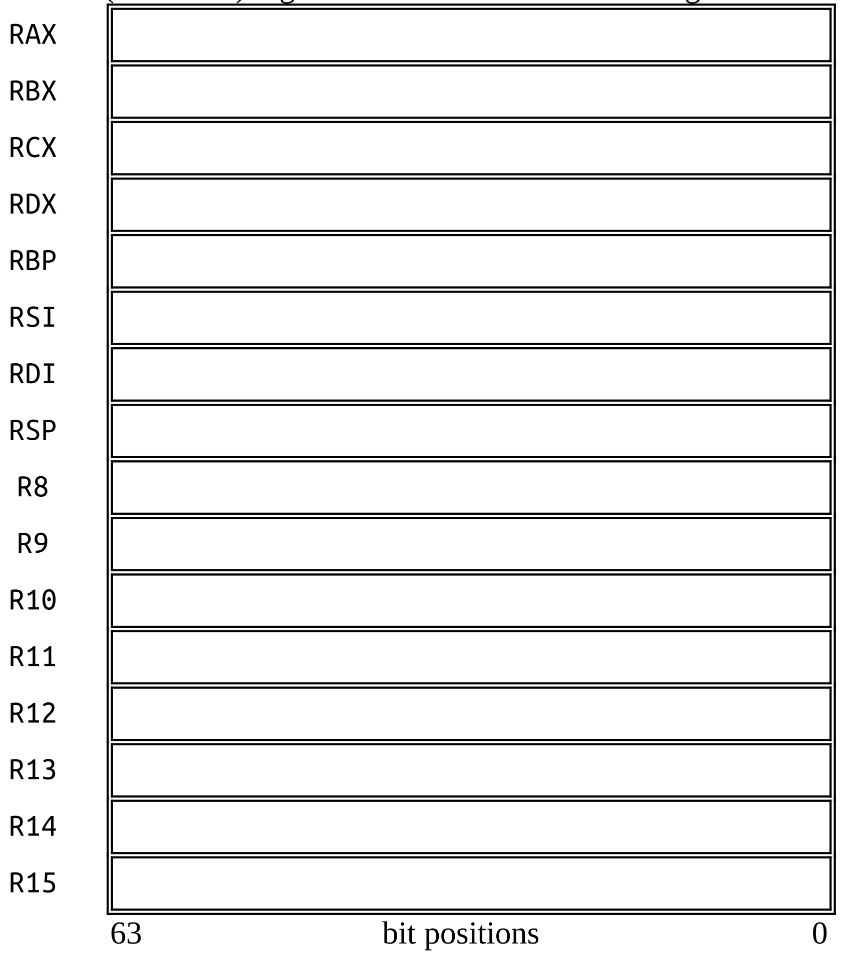
2. Modify **`lab4IIn1.c`**: add a new function that does the same thing as `square` but takes a `char` input parameter and returns a `char` for a result (note: C does not allow function overloading).
   In your lab report:
   i. show the assembly code for the `char` version of the function
   ii. highlight the differences in the assembly code for the `int` and `char` versions
   iii. explain what these differences tell us about how the stack is used for `ints` *vs.* `chars`.

3. Repeat step 2 for `long`, `float`, and `double` versions of the `square` function.

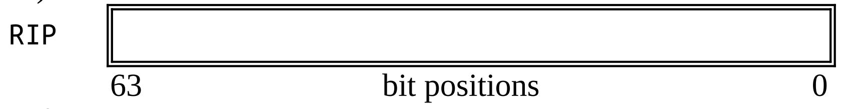
x86-64 Instructions, registers, and architecture

As you may have noticed in the questions above and from previous labs, there are many registers such as `rsp`, `rbp`, `edi`, `eax`, and `ebx` that show up in assembly versions of our C programs.

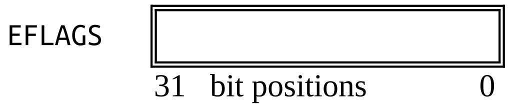
The most straightforward way to look at the x86-64 (or amd64) registers is as a set of 16 64-bit registers:



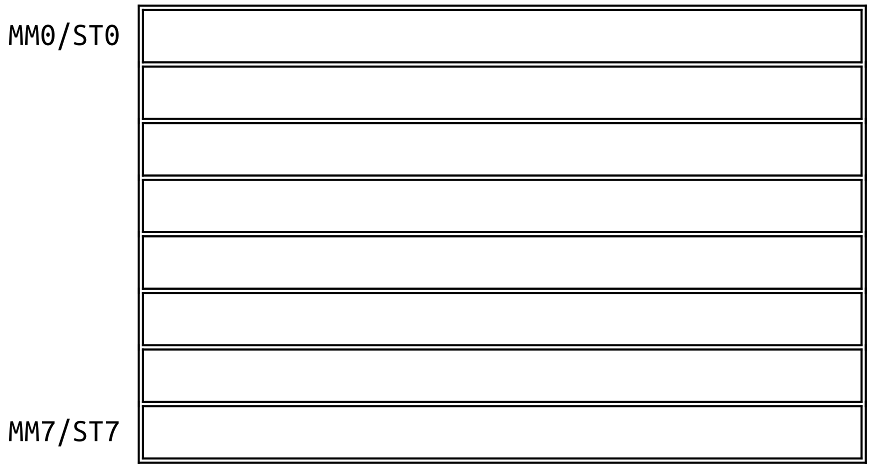
Plus an instruction pointer (or program counter):



And a register for "flags" that track the status of the CPU:

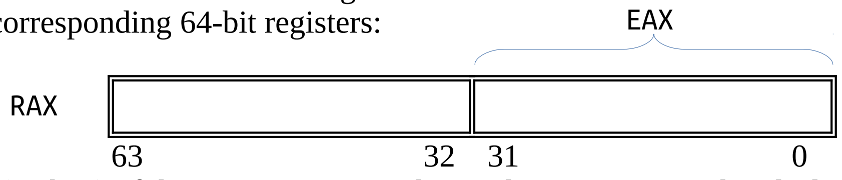


For floating-point (float, double) operations and for multimedia operations, we can use another set of 8 64-bit registers:



And finally, the Streaming SIMD Extension (SSE) instructions brought us a set of 8 extra wide **128**-bit registers: `xmm0-xmm8`. And the second version (SSE2) brought us 8 more: `xmm8-xmm15`.

You may have noticed that the above list does not have some of the registers you saw in the assembly programs earlier like `edi`, `eax`, and `ebx`. The reason is that these "e" registers are the least significant 4 bytes (32 bits) of the corresponding 64-bit registers:



Furthermore, the least significant two bytes (16 bits) of the "e" registers can be used in instructions that deal with 16-bit quantities. In the case of the 32-bit EAX register, bits 0-15 of EAX is the 16-bit AX register. And lastly the 16-bit "X" registers can be viewed as made up of two single-byte (8-bit) registers. For example, the 16-bit AX register is made up of registers AH (bits 7-15) and AL (bits 0-7).

You may also have noticed that there are different "addressing modes" for instructions (e.g. `push`**`q`**, `mov`**`l`** and `mov`**`b`**) depending on the types of data we are dealing with. The last letter usually indicates the addressing mode – the width of the data. Four of these modes are byte (**`b`** – 1 byte), word (**`w`** – 2 bytes), long (**`l`** – 4 bytes), and quadword (**`q`** – 8 bytes):

`mov`**`b`** moves a single byte
`mov`**`w`** moves a word – 2 bytes
`mov`**`l`** moves a long – 4 bytes
`mov`**`q`** moves a quadword – 8 bytes

Other modes like `movzbl` will "zero-extend" to the full register – i.e. it will pad the register with leading 0s.

An operand like `%rbp` indicates that we are using "**register mode**": the instruction uses the value in the register itself. On the other hand, parentheses - e.g. `(%rbp)` - implies "**indirect addressing mode**" is used to refer to the value at the memory location pointed to by the register.
We can also use "**base-relative indirect**" addressing – e.g. `-4(%rbp)` means the contents of memory pointed to by "<register value> minus 4". This is handy when we want to use the register to point to something (like the stack) and we also want to refer to things above or below it without changing the register value itself.
Then there is the "**offset-scaled-base-relative**" mode – e.g. `-8(%rbx,%rcx,4)` means the contents of memory pointed to by "<value in `%rbx`> + 4*<value in `%rcx`> - 8". This mode is useful for arrays. If the starting address of an int array is in register `%rbx` and the array index is in `%rcx`; since each array location is 4 bytes apart, we have to multiply by 4 to get the correct location.

Arithmetic Operations
Integer arithmetic can be done with the four instructions: `add`, `sub`, `imul`, and `idiv`. The `add` and `sub` instructions have two operands and the result of the operation goes into the second operand. For example, `addl %rax, %rbx` adds the contents of the two registers and puts the result into `%rbx`. i.e. `b = a + b`
The `imul` instruction has a single argument – the second operand comes from `%rax` and parts of the 128-bit result go into `%rax` and `%rdx`.
The `idiv` instruction has a single argument, the divisor, and the 128-bit dividend comes `%rax` and `%rdx` and the result goes into the “source” argument.
Other common operations include `inc` and `dec` to increment and decrement and bitwise boolean operations `and`, `or`, `not`, and `xor`.

Jump and Jump-if instructions
The simplest jump instruction, `jmp <address>`, places a specified address in the instruction pointer, `%rip`. This causes the CPU to “go to” this memory location, fetch the contents, decode it, and execute the instruction there. Jump instructions are used to implement if and switch statements and loops and to do this, a jump instruction is combined with checking bits in the `eflags` register:

`je`	jump if equal
`jne`	jump if not equal
`jl`	jump if less than
`jle`	jump if less than or equal
`jge`	jump if greater than or equal
`jg`	jump if greater than

The `eflags` register bits can be set by the `cmp` compare instruction or the arithmetic instructions.

Function calls
Calling a function can be viewed as “jumping” to the location of the function in memory and returning from a function is like “jumping” to the location after the function call. However, they require special instructions that in addition to jumping, they also have to deal with input parameters and return values by using the stack.

In the Intel architecture, the stack grows towards lower memory addresses – pushing a value onto the stack decrements the stack pointer, `%rsp`, and popping the stack increments it. In order to push `%rbx` onto the stack, we use the push instruction in quadword (8 byte) mode:

```
pushq %rbx
```

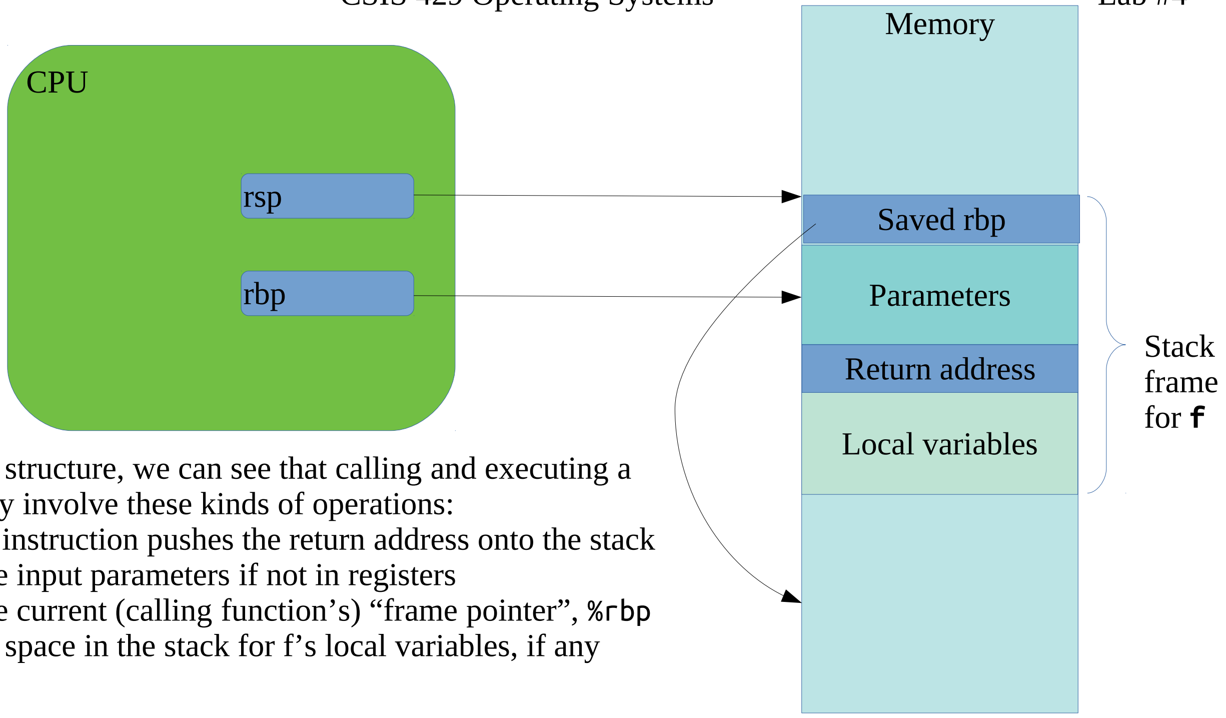
This will decrement `%rsp` by 8 and copy the contents of `%rbx` to the 8 bytes pointed to by the new value of `%rsp`. That is, the above push operation is equivalent to:

```
subq $8, %rsp
movq %rbx, (%rsp)
```

See if you can figure out the assembly language equivalent of `popq %rbx`

A `call` instruction effectively does two things: push the `%rip` onto the stack and jump to the starting address of the function code. A `ret` (return) instruction does the opposite: pop the old value of `%rip` stored in the stack back into the `%rip` - effectively jumping back to the return address.

Stack frames
A stack frame represents the invocation of a function. In order to support function calls in the various languages we use today, most compilers and operating systems use specific structures for their stack frames.
Here is an example of a stack frame for a function **`f`** (the actual positions depend on the details of the function, the compiler used, and the OS):



With this stack structure, we can see that calling and executing a function, `f`, may involve these kinds of operations:

- the call instruction pushes the return address onto the stack
- push the input parameters if not in registers
- push the current (calling function's) "frame pointer", `%rbp`
- making space in the stack for f's local variables, if any

If the function, f, calls some function, g, the stack frame for g will go "above" f's stack frame in the above diagram while the code for g is executed.

4. Compile and run the C program, **`lab4IIn2.c`**. In your report: show the assembly language equivalent for the debug version (-g) of the C code. Add code to print out the addresses (not the values) of all of the input parameters and local variables of the `square` and `sum` functions and recompile in debug mode.
   In your report, draw **5** memory maps showing the locations and contents of the various stack frames of your program at each of the following execution points:
   i. In `main`, before `sum` is called
   ii. In `sum`, before the first call to `square`
   iii. In `square`, for the second call to `square`
   iv. In `sum`, after the second call to `square`
   v. Back in `main` after the call to `sum`
   Note:
   - Because of Address Space Layout Randomization (ASLR), each time we run a program, we will get slightly different values for the addresses of things on the stack. So in your memory maps, use addresses **relative to the stack frame for `main`**.
   - The `gdb` command "**`info register`**" prints register values and "**`info frame`**" prints out information about the stack frame as you step through a program
   - Using `gdb`, figure out which of these are in your stack frames: **local variables**, **saved `%rbp`**, **return address**, and **parameters** and what their relative addresses are.

   Submit your modified **`lab4IIn2.c`** program as a separate file.

5. Compile the C program, **`lab4IIn3.c`**, and the **`square.s`** assembly code. The Makefile shows you how to do this. Run the executable and make sure it works properly.
   In your report: show the assembly language equivalent for the debug version (-g) of the C code.
   Step through the code with `gdb` and use your critical thinking and troubleshooting skills to explain why the `square.s` assembly code works even though it ***may seem like*** it is "missing" some instructions. Explain what the advantages and disadvantages are of using the **`square.s`** code to compute the square of an int.

Part III Pages and TLBs

1. Compile and run the **`lab4IIIn1.c`** program. In your report, show the output of the program and draw a map of the memory locations showing the addresses (in increasing order) of the sub arrays. Explain how the two-dimensional array is laid out in memory.
2. Compile the **`lab4IIIn2.c`** program – this file contains compile instructions. Run it 3 times. Include the outputs in your report and note the average matrix computation "throughput"s for the 3 runs. The program has a two dimensional array, the elements of which are accessed in a particular order.
   In case you are wondering, the `diff` and `nanodiff` together give us an accurate timer – the difference between the "end" time and the "start" time is how much time elapsed to compute `rows()`. To do this, we use the two data members in the `timespec struct`:
   ```
   struct timespec {
       time_t   tv_sec;        /* seconds */
       long     tv_nsec;       /* nanoseconds */
   };
   ```
3. Add a new function to the **`lab4IIIn2.c`** program named "`columns()`" that is exactly like the "`rows()`" function except that the order in the code for the **`i`** and **`j`** for-loops are switched around:
   In the `columns()` function, the **`i`** for-loop should be nested inside the **`j`** for-loop; the assignment statement for A[j][i] should remain the same as in `rows()`.
   Note that these two functions should accomplish the same task. Modify main() so that besides timing the execution of `rows()`, we can also time the execution of the **`columns()`** function.
4. Compile and run 3 times. There should be a big difference between the row and column average throughputs. Each person on a team should run the program independently and make sure the general trends are the same for each person's VM even if the exact numbers are different.
5. In the group report:
   - describe the order in which the elements of the A array are accessed in the `rows()` and `columns()` functions. What is the main difference?
   - include the output of each run and show what the average throughput is
   - is there a difference in the throughput or time required to execute "rows()" versus "columns()"? If there is a difference, explain the cause of the difference.

   Make sure all members of your group agree on the changes made to the code in modified **`lab4IIIn2.c`**. Each person should use these 3 values for MAX_ITERATIONS: 200, 500, and 1000. Run the program 3 times on your own VM for each value of MAX_ITERATIONS and report separately in the group report:
      - the output of the command: `cat /proc/cpuinfo`
      - the version of gcc (or whichever compiler) and all compiler flags you used to compile the program
      - the "rows" and "columns" execution times and throughput numbers in each run
      - the average row and column throughputs
    - Are the results what we should expect? Why?
   - In your report show the output of these two commands in each person's VM:
     ```
     cat /proc/cpuinfo
     valgrind --tool=cachegrind ./lab4IIIn2
     ```
   - Based on the above information how could your CPU's TLB be playing a part in this observed program behavior?

   Submit the modified **`lab4IIIn2.c`** program in your .tar.gz file.

Submit a single tar.gz file containing:
- all C programs as separate .c files
- a Makefile that can compile all C programs with one make command
- your report document