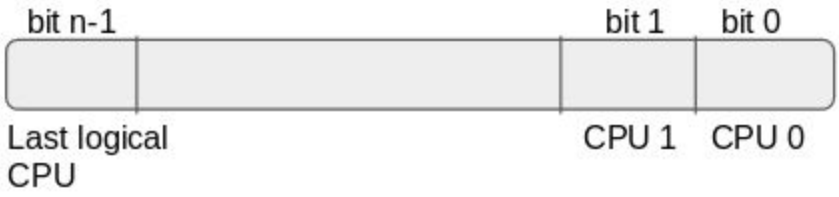
Tasks to be done on Saturday

taskset :

- It is used to set or retrieve the CPU affinity of a process.
- **CPU affinity** is a property that bonds a process to a given set of CPUs on the system.
- CPU affinity is represented by a **bitmask**.



  - Here **n** is the total number of CPUs

- How to set CPU affinity mask ?
  - If you want to specify that you want to restrict your process to run on either of cpu 0,1, 2 or 3 then the bitmask will be **0xF**.
  - If you want to run the process on a particular cpu (let's say) 4, then the bitmask will be **0x10**.

- Some useful commands :
  1. You can retrieve the CPU affinity of an existing task:
     taskset -p *pid*
  2. You can set the CPU affinity of an existing task:
     taskset -p *mask pid*
  3. You can run a task with a given affinity mask:
     taskset *mask* command
  4. You can also use:
     taskset -cp *cpuid pid*
     where cpuid ϵ [*0, 1,..., total cpu-1*]
- **DEMO**

Assert :

- It can be used to add diagnostics in your program.
- You can use it to verify assumptions made by your program and to perform sanity checks as well.
- **Prototype :**

```
void assert(int expression);
```

- The expression must always evaluate to *true*.
- If the expression evaluates to *false* then respective error message is sent to the standard error and abort() function is called.
- Example :
    - Suppose you wrote a C program to compute pow(2, x) where 'x' is user input from the terminal and you want 'x' to be less than 20 always. Then after reading the input you can simply write **assert(x < 20);** in your program.
    - If x is not less than 20 then the program will abort once it encounters **assert(x < 20);**
- Benefits of assert over printf :
    - No unnecessary printf
    - Less code
    - Clear and precise

Vim editor :

- Some basic useful commands :
    - Open a file - vi filename
    - Close a file - :q
    - Save the changes - :w
    - Save changes and exit - :wq
    - Forcefully quit - :q!
    - Insert mode - i
    - Command mode - Esc
    - Visual mode - v
- Fore more commands follow the link : https://vim.rtorr.com/

GDB :

- GDB stands for GNU Project Debugger and is a powerful debugging tool for C(along with other languages like C++).

- It helps you to poke around inside your C programs while they are executing and also allows you to see what exactly happens when your program crashes.
- GDB operates on executable files which are binary files produced by the compilation process.

Here are few useful commands to get started with gdb

run or r –> executes the program from start to end.

break or b –> sets breakpoint on a particular line.

disable -> disable a breakpoint.

enable –> enable a disabled breakpoint.

next or n -> executes the next line of code, but don't dive into functions.

step –> go to next instruction, diving into the function.

list or l –> displays the code.

print or p –> used to display the stored value.

quit or q –> exits out of gdb.

clear –> to clear all breakpoints.

continue –> continue normal execution till next breakpoint or watch-point

watch VariableName –> Monitor variable whenever value of that variable gets changed

You can play with any code using GDB by following some simple steps:

1. gcc -g test.c -o test (-g adds debugging information (variable names, line numbers etc) to the executable file)
2. gdb test
3. b LineNumber/FunctionName (you can add multiple breakpoints)
4. run
5. n
6. p VariableName
7. info r (to get information about registers)
8. disassemble /m FunctionName (to get assembly code)
9. backtrace (to see function call sequence)

mmap / munmap:

void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);

int munmap(void *start, size_t length);

- The **mmap**() function asks to map *length* bytes starting at offset *offset* from the file (or other object) specified by the file descriptor *fd* into memory, preferably at address *start*. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by **mmap**().

- The *prot* argument describes the desired memory protection (and must not conflict with the open mode of the file). It is either PROT_NONE or is the bitwise OR of one or more of the other PROT_* flags.

Tag	Description
PROT_EXEC	Pages may be executed.
PROT_READ	Pages may be read.
PROT_WRITE	Pages may be written.
PROT_NONE	Pages may not be accessed.

- The *flags* parameter specifies the type of the mapped object, mapping options and whether modifications made to the mapped copy of the page are private to the process or are to be shared with other references.

Tag	Description
MAP_PRIVATE	Create a private copy-on-write mapping. Stores to the region do not affect the original file. It is unspecified whether changes made to the file after the **mmap**() call are visible in the mapped region.
MAP_SHARED	Share this mapping with all other processes that map this object. Storing to the region is equivalent to writing to the file. The file may not actually be updated until **munmap**() are called.

GCC Inline Assembly

1. **GCC Assembler Syntax:** GCC, the GNU C Compiler for Linux, uses **AT&T/UNIX** assembly syntax.
    a. **Source-Destination Ordering:**
        Intel: Op-code dst src
        AT&T/Unix: Op-code src dst
    b. **Register Naming:** Register names are prefixed by % or %% i.e., if eax is to be used, write %eax or %%eax.
    c. **Immediate Operand:**
        i. Every immediate operands are preceded by '$'
        ii. In hexadecimal constant, we prefix the '0x' to the constant. So for a hexadecimal constant **deadbeef**, we first write '$', then '0x' and finally **deadbeef** i.e., the final hexadecimal immediate operand representation for **deadbeef** is **$0xdeadbeef**.

d. **Memory Operands:** In Intel, the base register is enclosed in '[' and ']' where as in AT&T/Unix they change to '(' and ')'. Additionally, for indirect memory reference, syntax are as follows:

   i. **In Intel:** [base + index*scale + disp]

   ii. **AT&T/Unix:** disp(base, index, scale)

   One exception is in immediate operand when used with memory operand for disp/scale, '$' should not be prefixed.

For example:

Intel Representation	**AT&T/Unix Representation**
mov eax,1	movl $1,%eax
mov ebx,0ffh	movl $0xff,%ebx
mov ebx, eax	movl %eax, %ebx
mov eax, [ecx]	movl (%ecx),%eax
mov eax, [ebx+3]	movl 3(%ebx),%eax
add eax,[ebx+ecx*2h]	addl (%ebx,%ecx,0x2),%eax
sub eax,[ebx+ecx*4h-20h]	subl -0x20(%ebx,%ecx,0x4),%eax

2. **Basic Inline Assembly (Basic Asm):** Basic inline assembly deals with assembler instruction that does not use operands (Input/output). A basic **asm** or **__asm__** (Use when compiled with **'-ansi'** or **'-std'** option) statement has the following syntax:

   asm (AssemblerInstructions)

   Or

   asm asm-qualifiers (AssemblerInstructions)

a. **Qualifiers**: The following qualifiers can be used with the basic inline assembly

   i. volatile

   ii. Inline

b. **AssemblerInstructions**: These are the literal string that specifies the assembler code as mentioned in point 1 (GCC Assembler Syntax).

   For example:

   asm("movl %ecx %eax"); /* moves the contents of ecx to eax */

   If we have more than one instructions, we write one per line in double quotes, and also suffix anyone of the following separators i.e. **';'**, **'\n'** or **'\n\t'**.

   For example:

   asm ("movl %eax, %ebx\n\t"

```
"movl $56, %esi\n\t"
"movl %ecx, $0x4(%edx,%ebx,$4)\n\t"
"movb %ah, (%ebx)");
```

c.

3. **Extended Asm:** In extended assembly, we can also specify the operands. It allows us to specify the input registers, output registers and a list of clobbered registers. An extended **asm** or **__asm__** statement has the following syntax:

```
asm asm-qualifiers ( AssemblerTemplate
    : OutputOperands
    [ : InputOperands
    [ : ClobbersList ] ])

        Or

asm asm-qualifiers ( AssemblerTemplate
        :
        : InputOperands
        : ClobbersList
        : GotoLabels)
```

   a. **Qualifiers:** A new qualifier **goto** has also been used with the extended Asm, which informs the compiler that the asm statement may perform a jump to one of the labels listed in GotoLabels
   b. **AssemblerTemplates:** It is the literal string similar to the **AssemblerInstructions** used in the **Basic Asm**.
   c. **Operands (Input/Output):** C expression serves as operands for the assembly instruction in the **asm**. Each operand is written as first operand constraint in double quotes.

      For example:

      "constraints" (C expression)

      It is the general form. For more than one input/output operands, a comma separated list of C expressions with constraints should be enlisted. An empty list is also permitted.

      i. InputOperands: Input operands are always in general form.
      ii. OutputOperands: For output operands an additional modifier will be there.

      Each operand is referenced by a number. If there are n operands (both input and output), then first output operands is numbered a 0, continuing in increasing

order, and the last input operand is numbered n-1. Operands corresponding to the C expressions are represented by %0, %1 ... etc.

For example: We want to multiply a number by 5. For that we use the instruction lea.

```
asm ("leal (%1,%1,4), %0"
        : "=r" (five_times_x)
        : "r" (x)
        );
```

d. **ClobbersList:** Some instructions clobber some hardware registers. We have to list those registers in the ClobbersList. We should not list the input and output registers in this list.

   For example:

```
asm ("movl %0,%%eax\n\t"
        "movl %1,%%ecx\n\t"
        "call _foo"
        : /* no outputs */
        : "g" (from), "g" (to)
        : "eax", "ecx"
         );
```

e. **GotoLabels: asm goto** allows assembly code to jump to one or more C labels. The **GotoLabels** section in an **asm goto** statement contains a comma-separated list of all C labels to which the assembler code may jump. To reference a label in the assembler template, prefix it with **'%l'** (lowercase **'L'**) followed by its (zero-based) position in GotoLabels plus the number of input operands.

   For example:

```
asm goto ("btl %1, %0\n\t"
        "jc %l2"
        : /* No outputs. */
        : "r" (p1), "r" (p2)
        : "cc"
        : carry);

return 0;

carry:
return 1;
```

4. **Constraints:** Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address;

whether the operand may be an immediate constant, and which possible values (ie range of values) it may have.... etc.

a. **Commonly used constraints**

i. **Register operand constraint(r):** When operands are specified using this constraint, they get stored in General Purpose Registers(GPR).
For example:

asm ("movl %%eax, %0" :"=r"(myval));

When the "r" constraint is specified, gcc may keep the variable in any of the available GPRs. To specify the register, you must directly specify the register names by using specific register constraints. They are:

r	Register(s)
a	%eax, %ax, %al
b	%ebx, %bx, %bl
c	%ecx, %cx, %cl
d	%edx, %dx, %dl
S	%esi, %si
D	%edi, %di

ii. **Memory operand constraint(m):** When the operands are in memory, any operations performed on them will occur directly in the memory location.
For example, the value of idtr is stored in the memory location loc:

asm("sidt %0\n" : :"m"(loc));

iii. **Matching(Digit) constraints:** In some cases, a single variable may serve as both the input and the output operand. Such cases may be specified in **asm** by using matching constraints.
For example:

asm ("incl %0" :"=a"(var):"0"(var));

iv. **Other Constraints:** Some other constraints used are:

1. "m" : A memory operand is allowed, with any kind of address that the machine supports in general.
2. “o" : A memory operand is allowed, but only if the address is offsettable. ie, adding a small offset to the address gives a valid address.

3. "V" : A memory operand that is not offsettable. In other words, anything that would fit the `m' constraint but not the `o'constraint.
4. "i" : An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.
5. "n" : An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use 'n' rather than 'i'.
6. "g" : Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

b. Constraints Modifiers: For more precise control over the effects of constraints, GCC provides us with constraint modifiers. Mostly used constraint modifiers are

i. "=" : Means that this operand is write-only for this instruction; the previous value is discarded and replaced by output data.

ii. "&" : Means that this operand is an early clobber operand, which is modified before the instruction is finished using the input operands.

5. Memory Barrier (fence instruction): Fence instruction is a type of barrier instruction that causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction. These instructions are:

a. LFENCE: Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes.

b. SFENCE: Orders processor execution relative to all memory stores prior to the SFENCE instruction. The processor ensures that every store prior to SFENCE is globally visible before any store after SFENCE becomes globally visible.

c. SFENCE: Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. The processor ensures that every load and store prior to MFENCE is globally visible before any load and store after MFENCE becomes globally visible.

Hands-on