In the last class, we looked at a few x86 instructions. We talked about physical address space. A processor executes the instructions which are stored in RAM. Initially, PC is set to the address of first instruction in the RAM. Afterward, a processor executes and automatically increment the program counter in an infinite loop.

Similarly, for to every variable in your program, some space is needed in RAM to store the content of these variables. For certain kind of variables, the memory (RAM) is managed by the compiler, whereas for some variables the memory is managed by the programmer.

An allocator does the memory management. There are several ways to implement an allocator. Let us look at buddy allocator. An allocator takes a memory region say (1024 to 102400) and supports allocation and recycling of an object.

LIST of different sizes

8
16
32
64
128
256
512
1024

The buddy allocator keeps lists of free objects of different sizes. If a list for a particular size is empty, the allocator tries to allocate a large size object, split into the smaller sizes and then add to the free list. Similarly, when an object is freed, it is added to free-list. Whenever possible, the freed object is also merged with the adjacent objects and moved to the list of greater size. Buddy allocator can be used for your malloc and free implementation.

Another example is the bump allocator. The bump allocator has an allocation pointer. An object can be allocated by simply adding the object size to the allocation pointer. However, freeing of an object is tricky, because the freed objects can be scattered across the memory region and it's not obvious, how to recycle them. However, if the objects are always deallocated sequentially, then we can easily recycle memory.

For example:
malloc (a);
malloc (b);
malloc (c);
free (c);
free (b);
free (a);

Local variables are also deallocated in the above manner, so a bump allocator is used for managing them. In the case of local variables, the bump allocator also behaves like a stack. Whoever allocated last get freed first.

There are other allocators as well; we are not going to discuss them in details. For, this class you can assume that the C language uses bump allocator for allocating and freeing local variables. For malloc and free it uses the buddy allocator.

When an application is loaded into the RAM, the loader also allocates space for bump and buddy allocator. Let us say (10000 to 20000) is allocated for buddy allocator and (1000 - 2000) is allocated for the buddy allocator. Bump allocator is usually of small size. Linux kernel used 8192 bytes, and PINTOS uses 4096 bytes for the bump allocator. In programming languages, the bump allocator is implemented using a stack. The direction of the allocation pointer is reversed to make use of some faster x86 instructions.

On x86, "esp" register is reserved for the allocation pointer. When the program is loaded, the loader sets the "esp" to the end of the space reserved for buddy allocator (2000) in this case. An allocation decrements the stack pointer.

```
main()
{
   int a = 0;
   int b = 10;
   return a + b;
}
```

For example, in this case initially the esp is set to 2000.
After, allocation of a, esp is moved to 1996. It means that &a == 1996.
After, allocation of b, esp is moved to 1992. &b == 1992
When a function returns the program automatically deallocates the memory for the local variable. For example, in this case, it will first deallocate "b" by simply adding 4 to the allocation pointer. Then it deletes "a" by again adding 4 to the stack pointer.

Argument passing:
Argument passing is also done via stack. When a function calls another function, it allocates space for all the arguments and destroys them after returning from the function.

```
main ()
{
   int a = 0;
   int b = 0;
   foo (a, b);
}
```

pass-by value, pass-by reference

Instruction relevant to stack,
        PUSH and POP

GCC calling conventions:

- X86 dictates that stack grows down

Example:

```
pushl %eax          subl   $4, %esp
                    movl  %eax, (%esp)
popl %eax           movl (%esp), %eax
                    addl $4, %esp
call 0x12345        push %eip
                    movl $0x12345, %eip
ret                 popl %eip
```

- Use example of a function foo() calling bar() with two arguments. Assume the bar returns the sum of two arguments; write bar's code in C and assembly and explain the conventions.
- Formally, introduce the conventions. GCC dictates how the stack is used. Contract between caller and callee in x86:
    - An entry to a function (i.e., just after call):
        - %eip points to the first instruction of function
        - %esp+4 points at first argument
        - %esp points at return address
    - After ret instruction
        - %eip contains return address
        - %esp points to the argument pushed by caller
        - Called function may have trashed arguments
        - %eax (and %edx, if return type is 64-bit) contains return value (or trash if the function is void)
        - %eax, %edx, and %ecx may be trashed
        - %ebp, %ebx, %esi, and %edi must contains content from time of call
    - Terminology:
        - %eax, %ecx, %edx are "callee save" registers
        - %ebp, %ebx, %esi, and %edi are "callee save" registers

- Discuss the frame pointer, and why it is needed – so that name of an argument, or a local variable does not change throughout the function body. Discuss the implication of using a frame pointer on the prologue and epilogue of the function body.
    - Function prologue
        - pushl %ebp
        - move %esp, %ebp
    - Function epilogue
        - movl %ebp, %esp
        - popl %ebp
        -
- The frame pointer (in ebp) is not strictly needed because a compiler can compute the address of its return address and function arguments based on its knowledge of the current depth of the frame pointer.
- Discuss the code for backtrace in gdb.

    -
      ```
      do {
        printf ("%p\n", (char*)ebp[1]);  // assuming ebp is unsigned*
        ebp = ebp[0];
      ```

```
        } while (ebp);
```

- Compiling, linking, and loading:
    - Preprocessor takes C source code, expands #include etc, produce C source code
    - Compiler takes C source code, produces assembly language
    - Assembler takes assembly language, produces .o file
    - Linker takes multiple '.o' , produces a single program image
    - Loader loads the program image into the memory at run-time and starts its execution

An application can execute multiple streams of sequential instructions. A stream of sequential instructions is called a thread. For example, consider the following program:

```
int print_mul (int arr[], int len) {
   int res = 1, i;
   for (i = 0; i < len; i++)
        res *= arr[i];
   printf ("mul : %d\n", res);
}

int print_average (int arr[], int len) {
   int sum = 0, i;
   for (i = 0; i < len; i++)
      sum += arr[i];
   printf ("avg: %d\n", sum);
}

int compute () {
   int arr1[100], arr2[100];
   read_from_file (arr1, 100);
   read_from_file (arr2, 100);
   create_thread (print_mul, arr1, 100);
   print_average (arr2);
}
```

The application finishes when all the threads terminate. e.g., even if "compute" completes the application remains active until "print_mul" is done and vice-versa.

Interrupts:

Interrupts are events from external devices that force a processor (CPU) to execute a different stream of Instructions. Interrupts can be triggered by multiple devices. To distinguish between interrupts from different devices, a vector number is associated with every interrupt. The x86 hardware supports 256 different interrupt vectors. An IDT contains the handler for every interrupt. On interrupt, the hardware automatically saves the EIP and EFLAGS on the stack before jumping to the target handler. An IRET instruction automatically pops the EFLAGS and EIP from the stack and restore them.

How can we implement a 100 ms sleep using the timer interrupt? Scheduler.