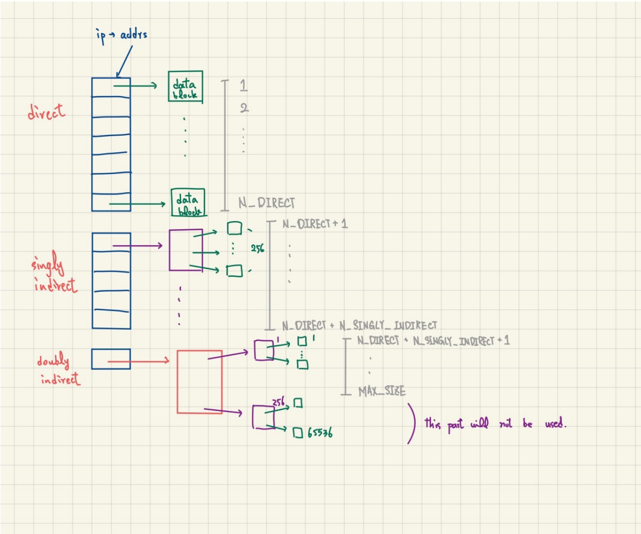
Problem 1



每個 inode 會有 13 個 addresses，也就是 `ip->addrs[ ]` 大小為 13；每個元素都是 `uint`，是一個 block 的 address。每個 block（可以想成 storage）大小為 1024 bytes。

這 13 個 addresses 又可分為：

- 7 個指向 direct blocks:
  這 7 個 blocks 直接拿來存取 data。因此共有 7 個 blocks 可使用。
- 5 個指向 singly-indrect blocks:
  這 5 個 blocks 存放第二層 blocks 的 address，而第二層 blocks 才會真正拿來存取 data。每個第一層 blocks 可存 1024/`sizeof(uint)` = 256 個 addresses。因此最多可以指向 $5 \times 256 = 1280$ 個可用的 blocks。
- 1 個指向 doubly-indirect blocks:
  和 singly-indirect 類似，第一層 blocks 指向最多 256 個第二層的 blocks。不過第二層的 blocks 也並非用來儲存資料，而是再指向 256 個 blocks；第三層的 blocks 才真正用來存資料。因此共有 $1 \times 256 \times 256 = 65536$ 個最多可用 blocks。

令 `N_DIRECT=7`，`N_SINGLY_INDIRECT=1280`，就可將 66666 個 blocks 分配道不同的類別：

- `[1, N_DIRECT]`：分配給 direct blocks。
- `[N_DIRECT + 1, N_DIRECT + N_SINGLY_INDIRECT]`：分配給 singly-indirect blocks。
- `[N_DIRECT + N_SINGLY_INDIRECT, 66666]`：分配給 doubly-indirect。

在一個 inode `ip` 上查詢第 `bn` 個 block 的 pseudocode 如下：（為求方便 $0 \leq$ `bn` < 66666）

Algorithm 1:

```
// This function returns the address of bn-th blocks on ip.
Function map (ip, bn)
    if bn ≥ 66666 then
        return 0;
    end
    if bn < N_DIRECT then
        return ip->addrs[bn]
    end
    ;
    bn -= N_DIRECT;
    if bn < N_SINGLY_INDIRECT then
        idx := bn // 256;
        addr := ip->addrs[7 + idx];
        return addr[bn % 256];
    end
    ;
    bn -= N_SINGLY_INDIRECT;
    idx1 := bn // 256;
    idx2 := bn % 256;
    addr1 := ip->addrs[(7 + 5)] addr2 := addr1[idx1];
    return addr2[idx2];
end
```

這個 pseudocode 中忽略了需要 allocate block 的情形。此過程基本上類似於將 `bn` 轉換為「256 進位制」，再存取每層對應的 blocks。

所以考慮找尋第 66666 個 block 的例子（`bn = 66665`），就會將 `bn = bn - 7 + 1280 = 65378`，再計算 `idx1 = 65378 // 256 = 255` 及 `idx2 = 65378 % 256 = 98`。最後回傳的 block address 就會是 `ip->addrs[12][255][98]`。

Problem 2

For a symlink `ip`, if the `O_NOFOLLOW` flag is not set, then we recursively get the target until the inode is not a symlink. The target of a symlink is stored in the data block when calling `symlink()`. Below is the pseudocode:

Algorithm 2:

```
Function open (path, flag)
    depth := 0;
    while (ip = namei(path)) != 0 do
        if ip.type != T_SYMLINK || flag & O_NOFOLLOW then
            break;
        end
        if depth ≥ 20 then
            return -1
        end
        path = read(ip);
        depth++;
    end
    // handling the inode......
end
```

For the given example, in the first loop we look up the `ip` of `/a`, which is a symlink. Then, we read the path `/b` from data block. In the next loop, we look up the inode of `/b`. Since the inode is not a symlink, the loop terminates. Now the rest part will work the same as `open("/b")`.

Problem 3

We can implement symlink directories with modifications to `namex`, which look up and return the inode for a path name. Here we take advantage of a built-in function in xv6, `skipelem(path, name)`, which copy the next "element" (i.e., the uppermost directory/file) in `path` to `name`, and return the rest part of `path`.

In `namex`, we iterate over each element of `path`. For `name` under current parent directory `ip`, using `dirlookup` to locate that inode, which becomes the next parent directory. If we find that this inode is a symlink and not the last element (which implies it is linked to a directory), then recursively call `namex` on the target path, and replace inode with the result. We also pass the recursion depth to `namex`, to prevent infinite loops.The pseudocode is as following:

Algorithm 3:

```
Function namex (path, depth)
    ip := ROOT_INODE;
    ;
    while (path = skipelem(path, name)) != 0 do
        next = dirlookup(ip, name);
        ;
        if next.type == T_SYMLINK and not path.empty() then
            target := read(next);
            next = namex(target, depth+1);
        end
        ;
        ip = next;
    end
    return ip;
end
```

Now we consider the given example. When kernel openning `/x/a/b`, `namex("/x/a/b", depth=0)` will be called. Then in each steps the state should be:

#	ip	path	name	next
1	`/`	`a/b`	`x`	`/x`
2	`/x`	`b`	`a`	`/x/a` → `/y`
3	`/y`	`'\0'`	`b`	`/y/b`

Finally the function will return inode of `/y/b` (if it is not symlink). Note in this table we represented inode by corresponding path for simplicity, although slightly not accurate. Thus the kernel will actually write to `/y/b`, or get an error if that behavior is not allowed.