DES_text.py:

Script:

```
#!/usr/bin/env python
## Homework Number: HW2_P1
## Name: ZhiFei Chen
## ECN Login: chen2281
## Due Date:  1/27/2020


import sys
import os
from BitVector import *

key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                      9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                     62,54,46,38,30,22,14,6,61,53,45,37,29,21,
                     13,5,60,52,44,36,28,20,12,4,27,19,11,3]

key_permutation_2 = [13,16,10,23,0,4,2,27,14,5,20,9,22,18,11,
                      3,25,7,15,6,26,19,12,1,40,51,30,36,46,
                     54,29,39,50,44,32,47,43,48,38,55,33,52,
                     45,41,49,35,28,31]

shifts_for_round_key_gen = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]


expansion_permutation = [31,  0,  1,  2,  3,  4,
                          3,  4,  5,  6,  7,  8,
                          7,  8,  9, 10, 11, 12,
                         11, 12, 13, 14, 15, 16,
                         15, 16, 17, 18, 19, 20,
                         19, 20, 21, 22, 23, 24,
                         23, 24, 25, 26, 27, 28,
                         27, 28, 29, 30, 31, 0]
permutation_box = [15, 6, 19, 20, 28, 11, 27, 16,
                   0, 14, 22, 25, 4, 17, 30, 9,
                   1, 7, 23, 13, 31, 26, 2, 8,
                   18, 12, 29, 5, 21, 10, 3, 24]

s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7],
               [0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8],
               [4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0],
               [15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13] ]

s_boxes[1] = [ [15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10],
               [3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5],
               [0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15],
               [13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9] ]

s_boxes[2] = [ [10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8],
               [13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1],
               [13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7],
               [1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12] ]

s_boxes[3] = [ [7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15],
```

```
            [13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9],
            [10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4],
            [3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14] ]

s_boxes[4] = [ [2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9],
            [14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6],
            [4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14],
            [11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3] ]

s_boxes[5] = [ [12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11],
            [10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8],
            [9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6],
            [4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13] ]

s_boxes[6] = [ [4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1],
            [13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6],
            [1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2],
            [6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12] ]

s_boxes[7] = [ [13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7],
            [1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2],
            [7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8],
            [2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11] ]

def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes" step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector(intVal = s_boxes[sindex][row][column], size = 4)
    return output


def extract_round_key(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
    return round_keys

def get_encryption_key(key):
    #key = ""
    key = BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key

# encryption_key = get_encryption_key()
# round_keys = generate_round_keys(encryption_key)
# print("\nHere are the 16 round keys:\n")
# for round_key in round_keys:
#     print(round_key)
```

```
def encrypt():
    key_file = sys.argv[3]
    key_for_encrypt = get_encryption_key(open(key_file, 'r').read())  # From the key.txt
    round_key = extract_round_key(key_for_encrypt)  # obtain the round key
    file_to_read = sys.argv[2]
    #print(file_to_read)
    bv = BitVector( filename= file_to_read )

    #print(sys.argv[0])
    if sys.argv[1] == "-e":
        #print(1)
        encrypted = open(sys.argv[4], 'w+')
        while (bv.more_to_read):
            bitvec = bv.read_bits_from_file( 64 )
            if len(bitvec) > 0:
                [LE, RE] = bitvec.divide_into_two()
                #print('A')
                #print((LE + RE).get_bitvector_in_hex())
                #
                #print(len(round_key))
                for r_key in round_key:

                    newRE = RE.permute( expansion_permutation )

                    out_xor = newRE ^ r_key
                    #out_xor = newRE ^ round_key[0]
                    RE_sub = substitute(out_xor)

                    RE_permute = RE_sub.permute(permutation_box)

                    RE_forLE = RE   # save the RE to assign to LE later

                    RE = LE ^ RE_permute

                    LE = RE_forLE
                    #print('B')
                #print((LE + RE).get_bitvector_in_hex())

                    #print((LE + RE).get_bitvector_in_hex())
                    #exit(0)

                #block = RE + LE
                block = (RE + LE).get_bitvector_in_hex()
                    #print(block)
                encrypted.write(block)
                #block.write_to_file(encrypted)

    if sys.argv[1] == '-d':
        decrypted_temp = open('temp.txt', 'w+b')
        #bv1 = BitVector(hexstring=open(sys.argv[2], 'rb'))
        readfile = open(sys.argv[2], 'r')
        bv1 = BitVector(hexstring=readfile.read())
        bv1.write_to_file(decrypted_temp)
        readfile.close()
        decrypted_temp.close()
        bv = BitVector( filename= 'temp.txt' )
        os.remove('temp.txt')
        decrypted = open(sys.argv[4], 'w+b')
```

```
        revered_round_key = round_key[::-1]
        while (bv.more_to_read):
            bitvec = bv.read_bits_from_file(64)
            #print(bitvec)
            if len(bitvec) > 0:
                [LE, RE] = bitvec.divide_into_two()
                for r_r_key in revered_round_key:
                    newRE = RE.permute(expansion_permutation)
                    out_xor = newRE ^ r_r_key
                    RE_sub = substitute(out_xor)
                    RE_permute = RE_sub.permute(permutation_box)

                    RE_forLE = RE  # save the RE to assign to LE later

                    RE = LE ^ RE_permute

                    LE = RE_forLE
                block = (RE + LE)
                block.write_to_file(decrypted)


            '''
            now comes the hard part --- the substition boxes

            Let's say after the substitution boxes and another
            permutation (P in Section 3.3.4), the output for RE is
            RE_modified.

            When you join the two halves of the bit string
            again, the rule to follow (from Fig. 4 in page 21) is
            either

            final_string = RE followed by (RE_modified xored with LE)

            or

            final_string = LE followed by (LE_modified xored with RE)

            depending upon whether you prefer to do the substitutions
            in the right half (as shown in Fig. 4) or in the left
            half.

            The important thing to note is that the swap between the
            two halves shown in Fig. 4 is essential to the working
            of the algorithm even in a single-round implementation
            of the cipher, especially if you want to use the same
            algorithm for both encryption and decryption (see Fig.
            3 page 15). The two rules shown above include this swap.
            '''
if __name__ == '__main__':
    encrypt()
```

Short explanation to the code:

For this problem, I first pass the first key to the get_encrytion_key function to generate the encryption for later use for getting the round_key. Then I read the message.txt file and converted it into bitvector class then use the Fiestel algorithm to encrypt it using the round_key for 16 rounds, and then convert it into hex string. For decryption, I need to convert the hex string back into Bitvector class and write it to a temp file, this is because the "more_to *read" function only supports instantiate a Bitvector class using a file name. And everything else is the same except for the reverse use of the round*_key for decryption.

Encrypted input:

605c6f3e13083a378764e40a8f2254f45b6ca29b034a7780ca45d40d67cc02bd44db1d8e453ceda
55d9e5465152afef9caeb8ec0f02d82bc7ffabfe89b887e4d60e21e9c9eecc280b91b4f7005743f09
ca25bad6b3d5208d5f20dea2715d10dec7d59e19e835d9edc78cb6086a7de91ca60d5fa49e79e8
550b519e0b275243c311346f917df2aff2c18680db97de8f64781405c1c6d57594ced10cec5c5f25
533f96066cf395136779ad02c46a68de8866dc905616d46729cf82d3e402b7daf98adaa11e2bfa2
7785b774487e0d51205b74361d8330187dd32a0d498c4743d023cbb6c8d18eae20dd1dc3ceb1c
7477855d439e250bc8dc6fbb0c5af42ce813e47b8e0daf5cbafa003e6609bf6ae29030381a819ee
5dec49be9bb0ca9dafde688038f76f9ce344abbc269281db6417db0c423e86aded601870c60fb93
e1624b5b8d94df99ab41f61ca6e846f836a3e1261fcaa2febe41fc17c459b83582f182ff9a65126a7
a0dc7e2776aa23c20792057edbb681ed4e0e56c9e91cf9fc1b9b1266d66bc30f968978041822c9b
9c8ab919429881422f3c1556b2a16facdabb84677c9aadae181d83aeb66688ffb35dd0dd14dbcab
ff8b9990375fea81b347d7318808a2f7231bcf90363a94c2e4a0a9133477336634c7a44e213b2c6
e86258509d6770ea58895bcf57f9e5ad412374897bd67d467d34fd077db85489bd996efcf0352af
07645b4614c7a41126731e4e8357f6c1a9f19d164683c84c9154bb6271438cf1ac748653d1c5f77
bc336e2831084b453ff68ac7c5870ea1f2994058b81e10f5699586c9718df402a1c6cc710a9a0591
043525df23249aaa3e9d599cb9a055ef7bfc360bc19a4baa9ec5f6c2117916669fab00c240e64ce1
00345f92618cf1b6f16f7b76614dc26a70de7dea0f37426211591095b172cd327446424512353fb
1960c67bfa5fe5cb543d7440cdfdf1c92ebd7a6e4a14c7c9aadae181d83ae367c2d09fa57949ad3b
80db0426c72a576239f51083965ca6770ea58895bcf571ab1c366b6e2904911554eb2b508534f4
1b423a02c41c30f100fe12f65fcf3401fdf2946d24e651446bfab2c48e5bf20b7e8431f1740031213
b3aebc2cc8059f4dc37efaa9b16d065f653c52ef06ee72fb61a8375b77209ac2236d345892a4aac2
12665ea415844122f22d777e02aa52e18d4416d7c33df7eaf44d6d3cc43388b7bb16d8dd25bcfc7
8b5a5d82ec5657c5d6ff4025aef08b0285aa47b24eeb850f5dd6e02f1d3fe73a960cd5afdcee7ac8
82ec8590551c3c016c3c51e9c9a6e7e045fddd184aa60ba16343fab24601f9b882ec8590551c3c0
16c3c51e9c9a6e7ef8afba5eb270d8493b506939fb6f39170b22bf3dbe7f7e55297dc61b9ec15b07
abba294bbd23834802469307f609c9232529622488901efd608835825777cf05527faafff91f6550
ea299e9c005501361600c17b99e8d5134523fee0dd15b65cc157b0b48c6e166023ff42df2446af7
4f8d28d235d94ba8fd25d09d33972eee1714a2e4a8e54310f9f14c918f60c717536a64cca35c181
e82dff5a431d60ad981b5f587b7b321527a5014fd5e8de2f04d713549f570efa46

Decrypted output:

Earlier this week, security researchers took note of a series of changes Linux and Windows developers began rolling out in beta updates to address a critical security flaw: A bug in Intel chips allows low-privilege processes to access memory in the computer's kernel, the machine's most privileged inner sanctum. Theoretical attacks that exploit that bug, based on quirks in features Intel has implemented for faster processing, could allow malicious software to spy deeply into other processes and data on the target computer or smartphone. And on multi-user machines, like the servers run by Google Cloud Services or Amazon Web Services, they could even allow hackers to break out of one user's process, and instead snoop on other processes running on the same shared server. On Wednesday evening, a large team of researchers at Google's Project Zero, universities including the Graz University of Technology, the University of Pennsylvania, the University of Adelaide in Australia, and security companies including Cyberus and Rambus together released the full details of two attacks based on that flaw, which they call Meltdown and Spectre.

DES_image.py

Script:

```
#!/usr/bin/env python
## Homework Number: HW2_P2
## Name: ZhiFei Chen
## ECN Login: chen2281
## Due Date:  1/27/2020


import sys
import os
from BitVector import *

key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                     9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                    62,54,46,38,30,22,14,6,61,53,45,37,29,21,
                    13,5,60,52,44,36,28,20,12,4,27,19,11,3]

key_permutation_2 = [13,16,10,23,0,4,2,27,14,5,20,9,22,18,11,
                     3,25,7,15,6,26,19,12,1,40,51,30,36,46,
                    54,29,39,50,44,32,47,43,48,38,55,33,52,
                    45,41,49,35,28,31]

shifts_for_round_key_gen = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]


expansion_permutation = [31,  0,  1,  2,  3,  4,
                          3,  4,  5,  6,  7,  8,
                          7,  8,  9, 10, 11, 12,
                         11, 12, 13, 14, 15, 16,
                         15, 16, 17, 18, 19, 20,
```

```
                      19, 20, 21, 22, 23, 24,
                      23, 24, 25, 26, 27, 28,
                      27, 28, 29, 30, 31, 0]
permutation_box = [15, 6, 19, 20, 28, 11, 27, 16,
                   0, 14, 22, 25, 4, 17, 30, 9,
                   1, 7, 23, 13, 31, 26, 2, 8,
                   18, 12, 29, 5, 21, 10, 3, 24]

s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7],
               [0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8],
               [4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0],
               [15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13] ]

s_boxes[1] = [ [15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10],
               [3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5],
               [0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15],
               [13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9] ]

s_boxes[2] = [ [10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8],
               [13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1],
               [13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7],
               [1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12] ]

s_boxes[3] = [ [7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15],
               [13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9],
               [10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4],
               [3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14] ]

s_boxes[4] = [ [2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9],
               [14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6],
               [4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14],
               [11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3] ]

s_boxes[5] = [ [12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11],
               [10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8],
               [9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6],
               [4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13] ]

s_boxes[6] = [ [4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1],
               [13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6],
               [1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2],
               [6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12] ]

s_boxes[7] = [ [13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7],
               [1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2],
               [7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8],
               [2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11] ]

def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes" step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector(intVal = s_boxes[sindex][row][column], size = 4)
    return output
```

```
def extract_round_key(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
    return round_keys

def get_encryption_key(key):
    #key = ""
    key = BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key

# encryption_key = get_encryption_key()
# round_keys = generate_round_keys(encryption_key)
# print("\nHere are the 16 round keys:\n")
# for round_key in round_keys:
#     print(round_key)




def encrypt():
    key_file = sys.argv[2]
    key_for_encrypt = get_encryption_key(open(key_file, 'r').read())  # From the key.txt
    round_key = extract_round_key(key_for_encrypt)  # obtain the round key

    img = open(sys.argv[1],'rb')
    data = img.readlines()
    #print(data)
    img.close()

    img1 = open(sys.argv[1], 'r')
    header = data[0:3]

    #img1.close()

    exclude_header = open(sys.argv[1], 'wb').writelines(data[3:])
    #exclude_header.close()
    bv = BitVector( filename= sys.argv[1] ) # this is the img data without the header
    encrypted_data = open(sys.argv[3], 'wb')
    while (bv.more_to_read):
        bitvec = bv.read_bits_from_file( 64 )
        #print(bitvec)
        if len(bitvec) > 0:
            if bitvec.length() < 64:
                bitvec = bitvec + BitVector(intVal=0, size=32)
            [LE, RE] = bitvec.divide_into_two()
            for r_key in round_key:
                newRE = RE.permute(expansion_permutation)
                #print(newRE)

                out_xor = newRE ^ r_key
```

```
            RE_sub = substitute(out_xor)

            RE_permute = RE_sub.permute(permutation_box)

            RE_forLE = RE  # save the RE to assign to LE later

            RE = LE ^ RE_permute

            LE = RE_forLE
        block = (RE + LE)

        block.write_to_file(encrypted_data)
    encrypted_data.close()

    encrypted_img_data = open(sys.argv[3], 'rb').readlines()
    #print(encrypted_img_data)
    f = open(sys.argv[3], 'wb')
    f.seek(0)  # get to the first positin
    #f.write('AA')
    f.writelines(header)
    f.writelines(encrypted_img_data)
    # f.writelines("P6")
    # f.writelines("155 51")
    # f.writelines("255")
    #with open(sys.argv[3], 'wb') as img:
     #   img.writelines(header)
      #  img.writelines(encrypted_img_data)


if __name__ == '__main__':
    encrypt()
```

Short explanation to the code:

For this problem, the algorithm used is the same as in problem 1, except we need to exclude the first 3 lines for header, then we encrypt what's left and write the header as it used to be back to the image_enc.ppm and the encrypted data.

Encrypted PPM image: