Solution

Problem 1: (17 points)

1 A\n
counter =3

2 A\n or B\n or A\n B\n

counter = 3 or 1 or 4

OR

B\n or A\n B\n , counter = 1 or 4

3 A\n or B\n or A\n B\n or A\n B\n B\n or B\n B\n

OR

B\n or A\n B\n or A\n B\n B\n or B\n B\n

Because the sequential 2 sigusr1 will cause signal pending and which will make some signal thrown away. So there are many possible outputs for B\n

Problem 2: (14 points)

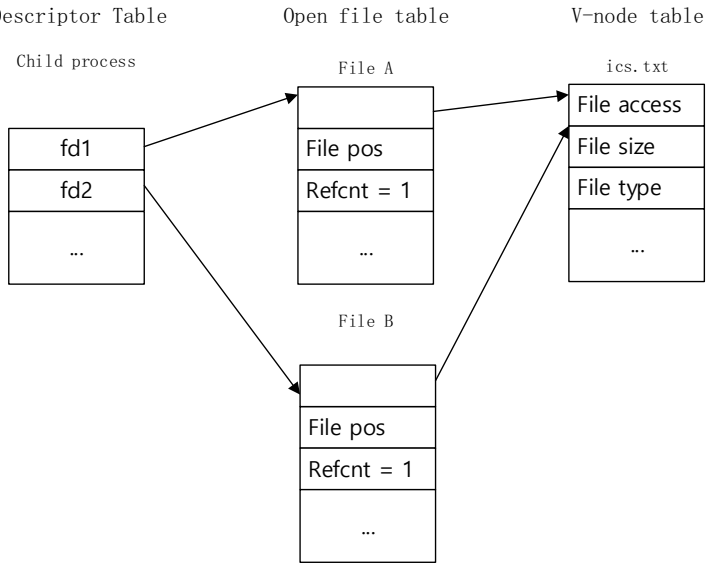
1 **Child fd1 S**

Child fd2 S

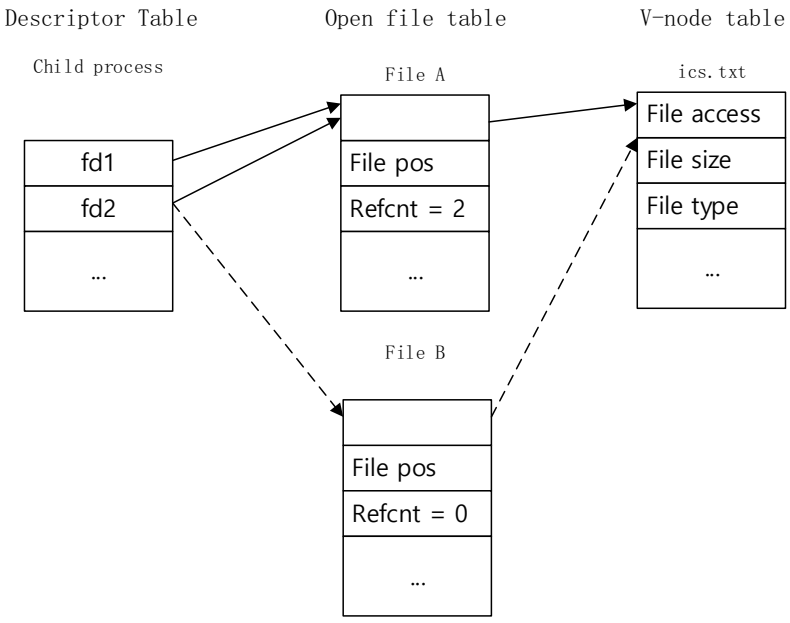
2 **Child fd1 S**

Child fd2 J

Before: (If Refcnt of File A is 2, it's also right)



After: (If Refcnt of File A is 3, it's also right)



3 **Parent fd1 T**

Parent fd2 S

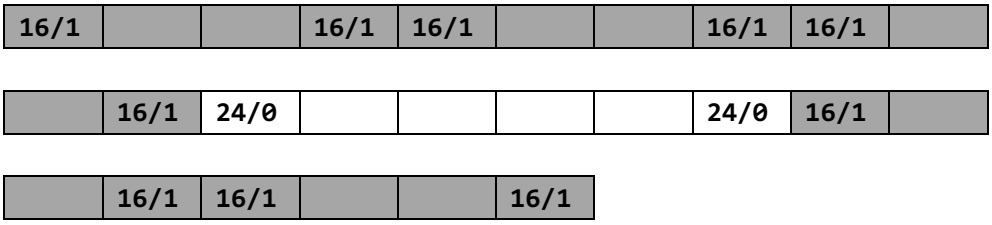
4 Parent fd1 H

Parent fd2 S

The content of ics.txt is SHTU120

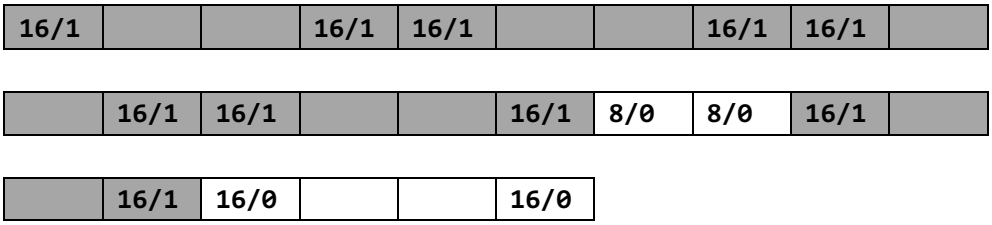
Problem 3: (16 points)

1



Internal fragments: 45 bytes

2



Internal fragments: 45 bytes

3 For first-fit: Yes. Internal fragments: 69 bytes

For best-fit: No. Failure in step 6, **P7 = malloc(2)**

Problem 4: (20 points)

1 [1] 12 [2] 2 [3] 2

2. 1) 32 bytes

2) [1] 1 [2] Hit [3] 0xca

[4] 1 [5] Hit [6] 0xff

[7] 0 [8] Miss [9] --

[10] 2 [11] Miss [12] --

[13] 0 [14] Miss [15] 0x34 OR --

Problem 5: (13 points)

1 25%

2 100%

3 25%

4 Only miss rate on cache C3 is reduced.

The cache misses in matrix_sum1() are cold miss. A larger cache block will fetch more bytes on cache line replacement and make them ready for subsequent accesses.

Problem 6: (20 points)

1

```
#define K 2 // (number of ways)
void dosth_and_find1(pixels_t *p, pixel_t *mintrans)
{
    int i, j;
    /* Elimminating Loop Inefficiencies  */
    int length = get_length(p);
    int limit = length - K;
    /* Used record index of mintrans pixel(Reducing Unneeded
Reference) */
    int minIndex1, minIndex2 = 0;
    int minAlpha1, minAlpha2 = 255;

    /* Used to reduce procedure call */
    pixel_t *cur1, *cur2;

    for ( i = 1; i < 4; i++ ) {
        for ( j = 0; j < limit; j+=2 ) {

            cur1 = get_pixel_at(j);
            cur2 = get_pixel_at(j+1);

           /* Unrolling & Parallelism */
            cur1->argb[i] = cur1->argb[i] * 2 % 255;
            cur2->argb[i] = cur2->argb[i] * 2 % 255;
```

```
            if ( cur1->argb[0] < minAlpha1 ) {
                minIndex1 = j;
                minAlpha1 = cur1->argb[0];
            }
            if ( cur2->argb[0] < minAlpha2 ) {
                minIndex2 = j+1;
                minAlpha2 = cur2->argb[0];
            }
        }

        for ( ; j < length; j++) {
            cur1 = get_pixel_at(j);
            cur1->argb[i] = cur1->argb[i] * 2 % 255;
            if ( cur1->argb[0] < minAlpha1 ) {
                minIndex1 = j;
                minAlpha1 = cur1->argb[0];
            }
        }

        mintrans = get_pixel_at( minIndex1 < minIndex2 ?
                    minIndex1 : minIndex2 );
 }
```

2

```
void dosth_and_find3(pixels_t *p, pixel_t *mintrans)
{
    int i;
    /* Elimminating Loop Inefficiencies  */
    int length = get_length(p) * 4;
    /* Used record index of mintrans pixel(Reducing Unneeded
Reference) */
    int minIndex = 0;
    int minAlpha = 255;
    /* treat data as one-dimension array */
    int *data = (int *)get_pixel_at(0);

    /* length is multiple of 4, don't need limit */
    for ( i = 0; i < length; i+=4 ) {
        data[i+1] = data[i+1] * 2 % 255;
        data[i+2] = data[i+2] * 2 % 255;
        data[i+3] = data[i+3] * 2 % 255;

        if ( data[i] < minAlpha ) {
```

```
            minIndex = i/4;
            minAlpha = data[i];
        }
    }
    mintrans = get_pixel_at(minIndex);
}
```