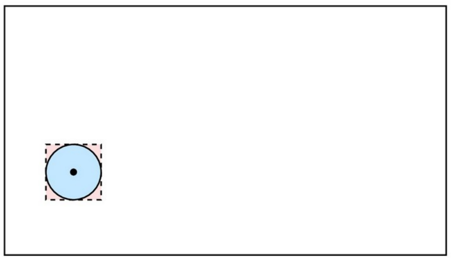
CS39002 Operating Systems Laboratory
Spring 2026

Lab Assignment: 9
Date posted: 27-Mar-2026

Demand Paging with LRU Page Replacement

A multimedia application plans to blur an image. We assume that the image size is 1920×1080 (this may, for example, be a frame of a Full-HD video). The blurring algorithm generates each image (i, j) of the output image as follows. The amount of blurring is specified by a blurring radius k (in pixels). Consider a circle of radius k centered at the point (i, j). All pixels of the input image in that region are averaged to generate the (i, j)-th output pixel. Averaging may follow the uniform or the normal (Gaussian) or any other distribution. To identify the pixels that are at a distance $\leqslant k$ from (i, j), it suffices to focus on the $(2k+1) \times (2k+1)$ square region centered at (i, j). Those pixels (x, y) in that square, that satisfy $(x-i)^2 + (y-j)^2 \leqslant k^2$ belong to the circle (this calculation can be done by integer arithmetic only; there is no need to use math library calls). This algorithm is illustrated in the figure to the right. An example of the working of this algorithm with blurring radius $k = 10$ and with the uniform distribution is given below.



Original image

Blurred image

A color image is commonly stored in the RGB format. Each pixel is specified by three basic color components (R, G, B) (red, green, and blue), where each of these components is (usually) an 8-bit unsigned integer (in the range $0-255$). The blurring algorithm computes the (uniform or weighted) average for each individual component.

Since the size of the image is fixed (1920×1080) in this assignment, we may store an image in three or one static two-dimensional arrays. The same storage mechanism is used for both the input image and the output image (these are stored separately). We use the row-major storage format, so we need 1080×1920 arrays.

Scheme 1:
```
unsigned char R[1080][1920], G[1080][1920], B[1080][1920];
```

Scheme 2:
```
unsigned char RGB[1080][5760];
```

In the first scheme, the (i, j)-th pixel is stored at the array locations R[i][j], G[i][j], and B[i][j]. In the second scheme, the (i, j)-th pixel is stored at the array locations RGB[i][3j], RGB[i][3j+1], and RGB[i][3j+2].

These schemes need three array accesses for every single pixel. In order to reduce the number of array accesses, we can pack the three components of each pixel in a 32-bit `int` data type. Another option is to use a structure of three `char` variables. Since the width of a structure is usually a multiple of 4 (or 8), the structure representation too calls for 32 bits to store a single pixel. The third scheme employs this idea. We will assume that `sizeof(pixel)` = 4.

Scheme 3:
```
typedef struct {
    unsigned char R, G, B;
} pixel;

pixel RGB[1080][1920];
```

While the above schemes are suitable for images of a fixed size (1920 × 1080 in our case), working with images of arbitrary dimensions calls for dynamic allocation of two-dimensional arrays. This leads to the following schemes.

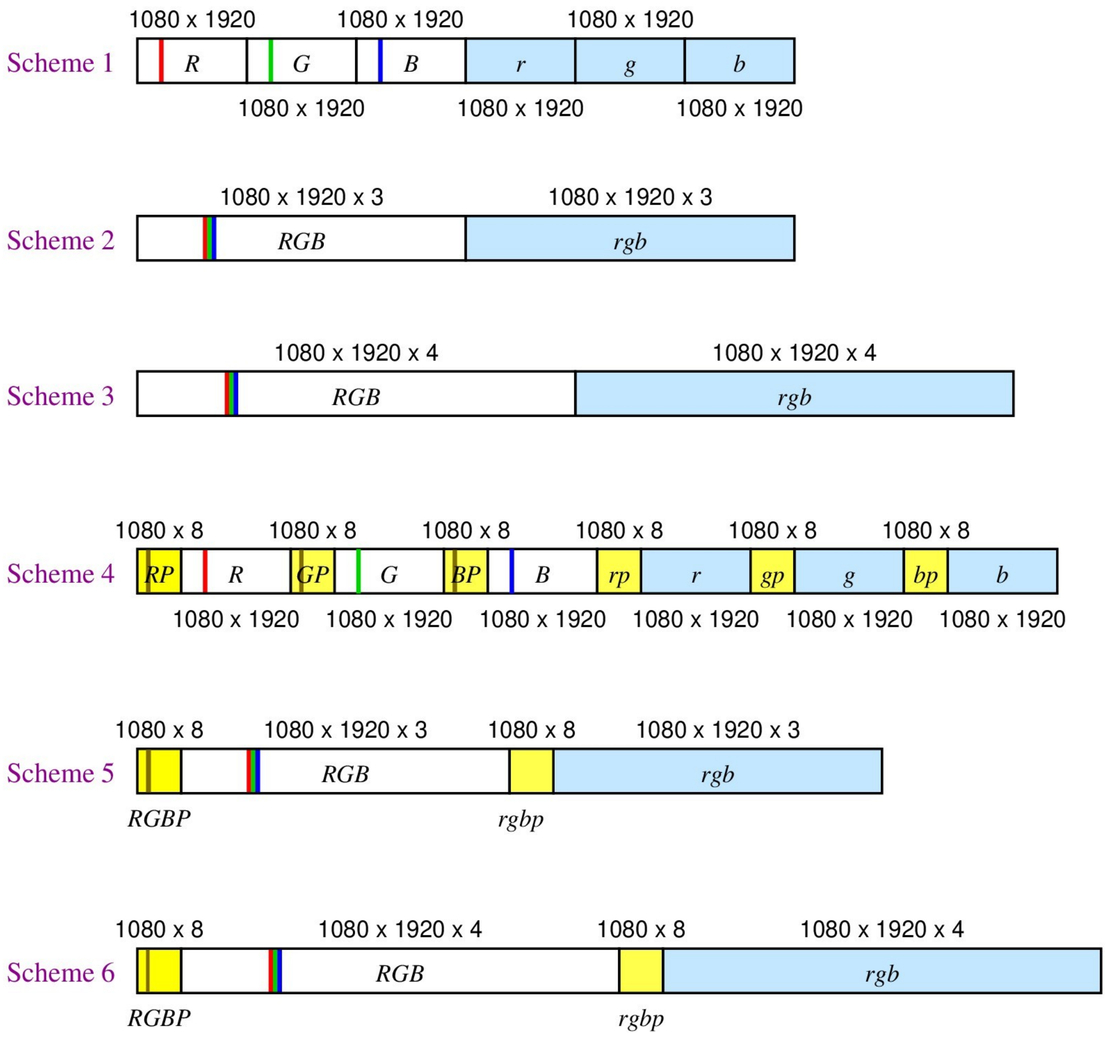
Scheme 4: `unsigned char **R, **G, **B;`

Scheme 5: `unsigned char **RGB;`

Scheme 6: `pixel **RGB;`

We assume that these pointers are allocated memory of suitable sizes, just sufficient to store 1920 × 1080 images. Although this scheme is flexible to accommodate images of various sizes, there is a problem. Accessing each single array element A[*i*][*j*] now requires two memory accesses. First, you need to access the row pointer A[*i*], and then you locate the *j*-th item A[*i*][*j*] from that row.

In the figure below, the virtual memory layout for these six schemes is given. In the figure, the input image is specified by three arrays *R*, *G*, and *B* or by a single array *RGB*. For the dynamic representations, we name the row-pointer arrays as *RP*, *GP*, *BP*, and *RGBP*. For the output image, we use the the lower-case names *r*, *g*, *b*, *rgb*, *rp*, *gp*, *bp*, *rgbp*. The byte sizes of the different chunks are as shown. Assume that the width of a pointer is 8 bytes. For dynamic arrays, assume that the pointer block comes first and is followed by the rows 0, 1, 2, …, 1079 in that order.



In this assignment, we investigate the performance of demand paging with LRU page replacement. The application uses the blurring algorithm on 1920 × 1080 images. You do not actually have to blur any image. Your program should only keep track of the pages accessed during a run of the algorithm. Assume that the page size is 4 KB. The major memory requirement for the application is for the storage of the input and the output images. Assume that the pages of these images stay in a secondary storage (hard drive or swap area), and are loaded only on demand. A total of *f* frames are given to the application in order to hold a subset of the pages of the two images. The application also needs a few more frames to store the code and other variables. Since these frames are frequently used, we assume that they are never replaced. All the *f* frames are for the two images only. At the beginning, all these *f* frames are empty. For every page access, first check whether that page currently resides in the memory. If so, no special thing needs to be done. Otherwise a page fault occurs. If not all of the f pages are loaded, any free frame can be allocated to the new page (assume that f frames are always available to the application). If all the f pages are loaded to frames, (local) LRU page replacement is used to swap out the least recently accessed page and to swap in the new page. In all these cases, mark the new page as the most recently used at that moment.

Use the following blurring loop that goes through the image in the row-major fashion. Only the pixel accesses are shown. Since you do not have to do the blurring itself, do not worry about the computations involved in the blurring algorithm (in particular, whether it uses uniform blurring or Gaussian blurring). You only need the blurring radius k to determine how many neighboring pixels the algorithm will look at.

```
for (i=0; i<1080; ++i) {
    for (j=0; j<1920; ++j) {

        /* First find the bounding box around (i,j), that lies inside the image */
        rowstart = i - k; if (rowstart < 0) rowstart = 0;
        rowend = i + k; if (rowend >= 1080) rowend = 1079;
        colstart = j - k; if (colstart < 0) colstart = 0;
        colend = j + k; if (colend >= 1920) colend = 1919;

        /* Traverse the bounding box in the row-major order */
        for (row = rowstart; row <= rowend; ++row) {
            for (col = colstart; col <= colend; ++col) {
                if ((row - i) * (row - i) + (col - j) * (col - j) <= k * k)  /* Use integer arithmetic only */
                    retrieve the (row, col)-th pixel of the input image.
            }
        }

        /* The output pixel is now ready */
        Write the (i,j)-th pixel to the output image.
    }
}
```

Maintain the f most recently used pages of the input and the output images taken together, in a doubly-linked list LP (list of active, that is, memory-resident pages). The list should be ordered with respect to the access times. The most recently used page is at the front, and the least recently used page is at the back. Also maintain a page table PT (assume that the maximum page-table size is 5000). $PT[p]$ stores a pointer to the node of the page p in the list LP provided that the page p is currently loaded in memory; it is NULL otherwise. After every page access, the node for that page will be brought to the front of LP (if it is not already there). During a page replacement, the last entry in the list is to be replaced, so a pointer LAST pointing to the last node in LP needs to be maintained.

The running time of the above algorithm is dominated by the inner loop on the bounding box, and takes a total time proportional to $1080 \times 1920 \times (2k+1)^2$. For $k = 5$, this is about one quarter of a billion. Moreover, for schemes 1, 2, 4, and 5, the program needs to access arrays three (or six) times for each pixel. So we need to make accesses to the list LP as efficient as possible (that is, independent of f). If a page p is accessed, the pointer $PT[p]$ is first looked at. If that pointer is not NULL, then the page is already loaded to memory, so the only task is to bring that node to the front of LP using pointer adjustments only. If $PT[p]$ is NULL, then this is a case of page fault. If not all of the f frames are used, then simply add a new node at the front of LP. Otherwise, delete the last node of LP, set the pointer in PT for the replaced page to NULL, readjust the LAST pointer, and insert a new node for p at the front of LP.

Make your own implementation of the data structures for LP and PT (because you cannot control the running times of ready-made library calls). We want each page-access overhead to be O(1).

For each of the six schemes of the representation of images, report the total number of memory accesses (a memory access can be either an array access or a pointer access). Also report the number of page faults the program encountered, and the page-fault rate (in percents) relative to the total number of memory accesses. Assume that servicing each page fault takes 10 ms, and each memory access (with the page loaded in memory) takes 100 ns. In each case, compute and report the total memory-access time.

Submit a single C/C++ file.

Sample Output

The following output shows the dependence of the performance of various array-storage schemes on the number f of frames. The page-fault counts (more or less) saturate at around $f = 30$. This happens because both the input and the output images are traversed only once in the row-major order. We however need a few previous and a few next rows of the input image. Nevertheless, if f is beyond a certain threshold, a page replaced is never needed again (except in an extremely small number of cases). For $f \geqslant 4055$, no page replacement is necessary (verify this).

```
$ gcc -Wall -O2 LRU.c
```

```
$ ./a.out 5 8
+++ k = 5, f = 8

+++ Scheme 1: Three static arrays
    Total number of memory accesses = 508540608
    Total number of page faults     = 41522452
    Page fault rate (percentage)    = 8.17
    Total memory access time        = 415275.37 sec

+++ Scheme 2: One static array of char
    Total number of memory accesses = 508540608
    Total number of page faults     = 24884931
    Page fault rate (percentage)    = 4.89
    Total memory access time        = 248900.16 sec

+++ Scheme 3: One static array of struct
    Total number of memory accesses = 169513536
    Total number of page faults     = 24901361
    Page fault rate (percentage)    = 14.69
    Total memory access time        = 249030.56 sec

+++ Scheme 4: Three dynamic arrays
    Total number of memory accesses = 1017081216
    Total number of page faults     = 54077052
    Page fault rate (percentage)    = 5.32
    Total memory access time        = 540872.23 sec

+++ Scheme 5: One dynamic array of char
    Total number of memory accesses = 1017081216
    Total number of page faults     = 27071257
    Page fault rate (percentage)    = 2.66
    Total memory access time        = 270814.28 sec

+++ Scheme 6: One dynamic array of struct
    Total number of memory accesses = 339027072
    Total number of page faults     = 27094547
    Page fault rate (percentage)    = 7.99
    Total memory access time        = 270979.37 sec
```

```
$ ./a.out 5 16
+++ k = 5, f = 16

+++ Scheme 1: Three static arrays
    Total number of memory accesses = 508540608
    Total number of page faults     = 41344850
    Page fault rate (percentage)    = 8.13
    Total memory access time        = 413499.35 sec

+++ Scheme 2: One static array of char
    Total number of memory accesses = 508540608
    Total number of page faults     = 15077
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 201.62 sec

+++ Scheme 3: One static array of struct
    Total number of memory accesses = 169513536
    Total number of page faults     = 24198
    Page fault rate (percentage)    = 0.01
    Total memory access time        = 258.93 sec

+++ Scheme 4: Three dynamic arrays
    Total number of memory accesses = 1017081216
    Total number of page faults     = 47938427
    Page fault rate (percentage)    = 4.71
    Total memory access time        = 479485.98 sec

+++ Scheme 5: One dynamic array of char
    Total number of memory accesses = 1017081216
    Total number of page faults     = 18126
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 282.97 sec

+++ Scheme 6: One dynamic array of struct
    Total number of memory accesses = 339027072
    Total number of page faults     = 27124
    Page fault rate (percentage)    = 0.01
    Total memory access time        = 305.14 sec
```

```
$ ./a.out 5 24
+++ k = 5, f = 24

+++ Scheme 1: Three static arrays
    Total number of memory accesses = 508540608
    Total number of page faults     = 3042
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 81.27 sec

+++ Scheme 2: One static array of char
    Total number of memory accesses = 508540608
    Total number of page faults     = 3039
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 81.24 sec

+++ Scheme 3: One static array of struct
    Total number of memory accesses = 169513536
    Total number of page faults     = 4050
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 57.45 sec

+++ Scheme 4: Three dynamic arrays
    Total number of memory accesses = 1017081216
    Total number of page faults     = 45490870
    Page fault rate (percentage)    = 4.47
    Total memory access time        = 455010.41 sec

+++ Scheme 5: One dynamic array of char
    Total number of memory accesses = 1017081216
    Total number of page faults     = 3045
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 132.16 sec

+++ Scheme 6: One dynamic array of struct
    Total number of memory accesses = 339027072
    Total number of page faults     = 10281
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 136.71 sec
```

```
$ ./a.out 5 32
+++ k = 5, f = 32

+++ Scheme 1: Three static arrays
    Total number of memory accesses = 508540608
    Total number of page faults     = 3042
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 81.27 sec

+++ Scheme 2: One static array of char
    Total number of memory accesses = 508540608
    Total number of page faults     = 3039
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 81.24 sec

+++ Scheme 3: One static array of struct
    Total number of memory accesses = 169513536
    Total number of page faults     = 4050
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 57.45 sec

+++ Scheme 4: Three dynamic arrays
    Total number of memory accesses = 1017081216
    Total number of page faults     = 3062
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 132.33 sec

+++ Scheme 5: One dynamic array of char
    Total number of memory accesses = 1017081216
    Total number of page faults     = 3045
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 132.16 sec

+++ Scheme 6: One dynamic array of struct
    Total number of memory accesses = 339027072
    Total number of page faults     = 4058
    Page fault rate (percentage)    = 0.00
    Total memory access time        = 74.48 sec
```