Week 10 Laboratory Sample Solutions

Objectives

- learn how to build a pathname from a environment variable
- practice file operations
- understanding how virtual memory works

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this lab called `lab10`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab10
$ cd lab10
$ 1521 fetch lab10
```

Or, if you're not working on CSE, you can download the provided code as a zip file or a tar file.

EXERCISE — INDIVIDUAL:
Append to a Diary File

We wish to maintain a simple diary in the file `$HOME/.diary`

Write a C program, `diary.c`, which appends 1 line to `$HOME/.diary`.

The line should be its command-line arguments separated by a space followed by a '\n'.

`diary.c` should print nothing on stdout. It should only append to `$HOME/.diary`.

```
$ dcc diary.c -o diary
$ ./diary Lisa
$ cat $HOME/.diary
Lisa
$ ./diary in this house
$ ./diary we obey the laws of thermodynamics
$ cat $HOME/.diary
Lisa
in this house
we obey the laws of thermodynamics
```

HINT:

The lecture example getstatus.c shows how to get the value of an environment variable.

snprintf is a convenient fucntion for constructing the pathname of the diary file.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest diary
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab10_diary diary.c
```

You must run `give` before **Monday 23 November 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `diary.c`

```c
// append arguments with newline to $HOME/.diary

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char *home_pathname = getenv("HOME");
    if (home_pathname == NULL) {
        home_pathname = ".";
    }

    char *basename = ".diary";
    int diary_pathname_len = strlen(home_pathname) + strlen(basename) + 2;
    char diary_pathname[diary_pathname_len];
    snprintf(diary_pathname, sizeof diary_pathname, "%s/%s", home_pathname, basename);

    FILE *stream = fopen(diary_pathname, "a");
    if (stream == NULL) {
        perror(diary_pathname);
        return 1;
    }

    for (int arg = 1; arg < argc; arg++) {
        fprintf(stream, "%s", argv[arg]);
        if (arg < argc - 1) {
            fprintf(stream, " ");
        }
    }
    fprintf(stream, "\n");
    fclose(stream);

    return 0;
}
```

EXERCISE — INDIVIDUAL:

Simulate LRU Replacement of Virtual Memory Pages

Your task is to simulate least-recently-used (LRU) replacement of virtual memory pages.

Write a C program, `lru.c`, which takes two arguments, both integers.

This will be respectively the size in pages of simulated physical memory and and the size in pages of simulated virtual memory.

`lru.c` should then read integers from stdin until EOF.

Each integer will be the number of a virtual page being accessed.

`lru.c` should print one line of output for indicate what action occurs with a phyical memory of the given size and LRU replacement.

```
$ dcc lru.c -o lru
$ ./lru 4 6
Simulating 4 pages of physical memory, 6 pages of virtual memory
5
Time 0: virtual page 5 loaded to physical page 0
3
Time 1: virtual page 3 loaded to physical page 1
5
Time 2: virtual page 5 -> physical page 0
3
Time 3: virtual page 3 -> physical page 1
0
Time 4: virtual page 0 loaded to physical page 2
1
Time 5: virtual page 1 loaded to physical page 3
2
Time 6: virtual page 2  - virtual page 5 evicted - loaded to physical page 0
2
Time 7: virtual page 2 -> physical page 0
3
Time 8: virtual page 3 -> physical page 1
5
Time 9: virtual page 5  - virtual page 0 evicted - loaded to physical page 2
```

In the files for this week's lab you'be been give code which implements a suitable data structure to store information about accesses,

For each access this function is called:

```c
void access_page(int virtual_page, int access_time, int n_physical_pages, struct ipt_entry *ipt) {

    // PUT YOUR CODE HERE TO HANDLE THE 3 cases
    //
    // 1) The virtual page is already in a physical page
    //
    // 2) The virtual page is not in a physical page,
    //    and there is free physical page
    //
    // 3) The virtual page is not in a physical page,
    //    and there is no free physical page
    //
    // don't forgot to update the last_access_time of the virtual_page

    printf("Time %d: virtual page %d accessed\n", access_time, virtual_page);
}
```

You can complete the lab by adding code to this function (you are also free to write your own program).

HINT:

If you implement this in the most obvious way you can ignore the number of virtual pages.

NOTE:

No error checking is required.
Your program can assume it is always given the names of 2 valid arguments.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest lru
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab10_lru lru.c
```

You must run `give` before **Monday 23 November 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `lru.c`

```c
// Simulate LRU replacement of page frames

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

typedef struct ipt_entry {
    int virtual_page;
    int last_access_time;
} ipt_entry_t;


void lru(int n_physical_pages, int n_virtual_pages);
void access_page(int virtual_page, int access_time, int n_physical_pages, struct ipt_entry *ipt);

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <n-physical-pages> <n-virtual-pages>\n", argv[0]);
        return 1;
    }
    lru(atoi(argv[1]), atoi(argv[2]));
    return 0;
}

void lru(int n_physical_pages, int n_virtual_pages) {
    printf("Simulating %d pages of physical memory, %d pages of virtual memory\n",
          n_physical_pages, n_virtual_pages);
    struct ipt_entry *ipt = malloc(n_physical_pages * sizeof *ipt);
    assert(ipt);

    for (int i = 0; i < n_physical_pages; i++) {
        ipt[i].virtual_page = -1;
        ipt[i].last_access_time = -1;
    }

    int virtual_page;
    for (int access_time = 0; scanf("%d", &virtual_page) == 1; access_time++) {
        assert(virtual_page >= 0 && virtual_page < n_virtual_pages);
        access_page(virtual_page, access_time, n_physical_pages, ipt);
    }
}

void access_page(int virtual_page, int access_time, int n_physical_pages, struct ipt_entry *ipt) {
    int physical_page_to_use = 0;

    for (int pp = 0; pp < n_physical_pages; pp++) {
        if (ipt[pp].virtual_page == virtual_page) {
            physical_page_to_use = pp;
            break;
        }

        if (ipt[pp].last_access_time < ipt[physical_page_to_use].last_access_time) {
            physical_page_to_use = pp;
        }
    }

    int vp = ipt[physical_page_to_use].virtual_page;

    printf("Time %d: virtual page %d ", access_time, virtual_page);
    if (vp == virtual_page) {
        printf("->");
    } else if (vp == -1) {
        printf("loaded to");
    } else {
        printf(" - virtual page %d evicted - loaded to", vp);
    }
    printf(" physical page %d\n", physical_page_to_use);

    ipt[physical_page_to_use].virtual_page = virtual_page;
    ipt[physical_page_to_use].last_access_time = access_time;
}
```

CHALLENGE EXERCISE — INDIVIDUAL:

Simulate Virtual Memory

Your task is a simple simulation of virtual memory pages.

Write a C program, `page_table.c`, which takes two arguments, both integers.

This will be respectively the size in pages of simulated physical memory and and the size in pages of simulated virtual memory.

`page_table.c` should then read from stdin until EOF.

Each line will contains a letter describing an action and the number of a virtual page.

`page_table.c` should print one line of output for indicate what action occurs with a phyical memory of the given size and LRU replacement.

An action will be described by one of these 5 letters.

- **R** make page available for read access
- **W** make page available for write access
- **U** make page no longer available for access
- **r** make a read access to a page
- **w** make a write access to a page

Page accesses are checked:

```
$ dcc page_table.c -o page_table
$ ./page_table 4 6
Simulating 4 pages of physical memory, 6 pages of virtual memory
r 0
Time 0: virtual page 0  - read access - illegal
R 0
Time 1: virtual page 0 mapped read-only
r 0
Time 2: virtual page 0  - read access - loaded to physical page 0
r 0
Time 3: virtual page 0  - read access - at physical page 0
w 0
Time 4: virtual page 0  - write access - illegal
W 0
Time 5: virtual page 0 mapped read-write
w 0
Time 6: virtual page 0  - write access - at physical page 0
```

And a LRU replacement strategy should be implemented like the previous exercise, for example:

```
$ ./page_table 4 6
Simulating 4 pages of physical memory, 6 pages of virtual memory
R 0
Time 0: virtual page 0 mapped read-only
R 1
Time 1: virtual page 1 mapped read-only
R 2
Time 2: virtual page 2 mapped read-only
W 3
Time 3: virtual page 3 mapped read-write
W 4
Time 4: virtual page 4 mapped read-write
r 0
Time 5: virtual page 0  - read access - loaded to physical page 0
r 1
Time 6: virtual page 1  - read access - loaded to physical page 1
w 4
Time 7: virtual page 4  - write access - loaded to physical page 2
w 3
Time 8: virtual page 3  - write access - loaded to physical page 3
r 0
Time 9: virtual page 0  - read access - at physical page 0
r 2
Time 10: virtual page 2  - read access -  virtual page 1 evicted - loaded to physical page 1
```

A reference implementation is available by running

```
$ 1521 page_table 4 6
Simulating 4 pages of physical memory, 6 pages of virtual memory
...
```

NOTE:

No error checking is required.
Your program can assume it is always given the names of 2 valid arguments.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest page_table
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab10_page_table page_table.c
```

You must run `give` before **Monday 23 November 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `page_table.c`

```c
// Simulate LRU replacement of page frames

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#define READ_ACCESS  0x1
#define WRITE_ACCESS 0x2

// represent an entry in a simple page table
typedef struct pt_entry {
    int physical_page;
    int permissions;      // READ_ACCESS and/or WRITE_ACCESS may be set
} pt_entry_t;

// represent an entry in a simple inverted page table

typedef struct ipt_entry {
    int virtual_page;        // == -1 if physical page free
    int last_access_time;
} ipt_entry_t;


void page_table(int n_physical_pages, int n_virtual_pages);
void page_access(int action, int virtual_page, int access_time,
                 int n_virtual_pages, int n_physical_pages,
                 struct pt_entry *page_table, struct ipt_entry *inverted_page_table);

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <n-physical-pages> <n-virtual-pages>\n", argv[0]);
        return 1;
    }
    page_table(atoi(argv[1]), atoi(argv[2]));
    return 0;
}


void page_table(int n_physical_pages, int n_virtual_pages) {
    printf("Simulating %d pages of physical memory, %d pages of virtual memory\n",
          n_physical_pages, n_virtual_pages);

    struct pt_entry *page_table = malloc(n_virtual_pages * sizeof *page_table);
    assert(page_table);

    struct ipt_entry *inverted_page_table = malloc(n_physical_pages * sizeof *inverted_page_table);
    assert(inverted_page_table);

    for (int i = 0; i < n_virtual_pages; i++) {
        page_table[i].physical_page = -1;
        page_table[i].permissions = 0;
    }

    for (int i = 0; i < n_physical_pages; i++) {
        inverted_page_table[i].virtual_page = -1;
        inverted_page_table[i].last_access_time = 0;
    }

    int virtual_page;
    char action;
    for (int access_time = 0; scanf(" %c%d", &action, &virtual_page) == 2; access_time++) {
        assert(virtual_page >= 0 && virtual_page < n_virtual_pages);
        assert(strchr("RWrwU", action) != NULL);
        page_access(action, virtual_page, access_time, n_virtual_pages, n_physical_pages,
                    page_table, inverted_page_table);
    }
}


void page_access(int action, int virtual_page, int access_time,
                 int n_virtual_pages, int n_physical_pages,
                 struct pt_entry *page_table, struct ipt_entry *inverted_page_table) {
    printf("Time %d: virtual page %d ", access_time, virtual_page);
    if (action == 'R') {
```

```
    if (action == 'R') {
        page_table[virtual_page].permissions |= READ_ACCESS;
        printf("mapped read-only\n");
        return;
    }

    if (action == 'W') {
        page_table[virtual_page].permissions |= (READ_ACCESS|WRITE_ACCESS);
        printf("mapped read-write\n");
        return;
    }

    int physical_page = page_table[virtual_page].physical_page;

    if (action == 'U') {
        if (page_table[virtual_page].permissions == 0) {
            printf("not mapped\n");
            return;
        }
        page_table[virtual_page].physical_page = -1;
        page_table[virtual_page].permissions = 0;
        printf("unmapped");
        if (physical_page != -1) {
            inverted_page_table[physical_page].virtual_page = -1;
            printf(" - physical page %d now free" , physical_page);
        }
        printf("\n");
        return;
    }

    //action must be 'r' (read) or 'w'(write)
    char *access_type = action == 'w' ? "write" : "read";
    printf(" - %s access - ", access_type);

    if (
        (action == 'r' && !(page_table[virtual_page].permissions & READ_ACCESS)) ||
        (action == 'w' && !(page_table[virtual_page].permissions & WRITE_ACCESS))
       ) {
        printf("illegal\n");
        return;
    }

    if (physical_page != -1) {
        printf("at physical page %d\n" , physical_page);
        inverted_page_table[physical_page].last_access_time = access_time;
        return;
    }

    // Page fault must find a free page or evict the least-recently-accessed page

    int physical_page_to_use = 0;
    for (int pp = 0; pp < n_physical_pages; pp++) {
        if (inverted_page_table[pp].virtual_page == -1) {
            physical_page_to_use = pp;
            break;
        }
        if (inverted_page_table[pp].last_access_time <
            inverted_page_table[physical_page_to_use].last_access_time) {
            physical_page_to_use = pp;
        }
    }

    int evicted_virtual_page = inverted_page_table[physical_page_to_use].virtual_page;

    if (evicted_virtual_page == -1) {
        printf("loaded to");
    } else {
        printf("virtual page %d evicted - loaded to", evicted_virtual_page);
        page_table[evicted_virtual_page].physical_page = -1;
    }
    printf(" physical page %d\n", physical_page_to_use);

    page_table[virtual_page].physical_page = physical_page_to_use;
    inverted_page_table[physical_page_to_use].virtual_page = virtual_page;
    inverted_page_table[physical_page_to_use].last_access_time = access_time;
}
```

Submission

When you are finished each exercises make sure you submit your work by running `give`.

You can run `give` multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Mon Nov 23 21:00:00 2020** to submit your work.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer several days after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run by the lecturer you can view your results here. The resulting mark will also be available via give's web interface.

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:

```
$ 1521 classrun -sturec
```