CSE2031 Software Tools - Pointers, Allocations, Structures once again

Summer 2010

Przemyslaw Pawluk

Department of Computer Science and Engineering
York University
Toronto

June 15, 2010

Notes

Table of contents

1. Midterm summary and review
   - Pointers
   - Typedef
   - Structures and Unions
   - Complex structures
2. File access in C
   - FILE and file pointers
3. System calls
   - Low level access to files in UNIX

Notes

Exam summary

You did well

- Standard input processing
- Testing
- "Debugging"

Weak points

- Memory allocation
- Pointers (especially pointers to functions)
- Structures
- Typedef

Notes

How do we define pointers?

Pointers to variables

```c
int* pi;
float *pf;
```

Pointers to structures

```c
struct str* pi;
```

Pointers to functions

```c
returned_type (*pfoo)(types_of_params);
float (*pf)(int*, void*);
```

Notes

Memory allocation

Functions

- void * malloc(int size);
- void * calloc(int n, int size);
- void * realloc(void * ptr, int size);
- void free(void *ptr);

```c
#define SIZE 10
int main(){
  int i;
  char * buffer = (char *) malloc (SIZE);
  if (buffer==NULL) exit (1);
  for (i=0; i<SIZE; i++)
    buffer[n]=rand()%26+'a';
  buffer[SIZE]='\0';
  printf ("Random string: %s\n",buffer);
  free (buffer);
  return 0;
}
```

Notes

Syntax

The syntax is the same as when defining variable (except typedef).

Variable - x is variable of type int*

```c
int *x;
```

Type - x is equivalent type to int *

```c
typedef int *x;
x i; /*equivalent to int *i; */
```

Notes

Defining new or renaming existing type

Typedef

```
/*aaa is new name for int*/
typedef int aaa;
/*cAr100 is new name for array of 100 chars*/
typedef char cAr100[100];
/*func is a function taking
two ints and returning int*/
typedef int func(int, int);
/*pfunc is a pointer to function
taking two ints and returning int*/
typedef int (*pfunc)(int, int);
/*tStr is a equivalent to sname,
tpStr is equivalent to *sname*/
typedef struct sname{
    member_type1 member_name1;
    ...
} tStr, *tpStr;
```

Notes

Structures

How structures are defined?

```
struct sname{
    member_type1 member_name1;
    member_type2 member_name2;
    ...
    member_typeN member_nameN;
} s_var1, *ps_var1;
```

Notes

Linked List

Definition

- List can be empty (NULL) or
- List has a head (list element) and tail (list)
- Each element has a pointer to the next element (last points to NULL)

```
struct listNode{
    int x;
    struct listNode *next;
} *head;
typedef struct listNode list;
```

Notes

Operations

Add to the end

```
list* addEnd(list *head, int newVal){
    list *new = (list *) malloc(sizeof(list));
    if(head==NULL)
        return new;
    while((head->next)!=NULL)
        head=head->next;
    head->next=new;
    return head;
}
```

Notes

Operations

Remove head

```
list* freeFirst(list *head){
    list *tmp;
    if(head==NULL)
        return NULL;
    tmp=head->next;
    free(head);
    return tmp;
}
```

Notes

Binary tree

Definition

- Tree can be empty or
- Tree has a root (tree node) and two children (trees)
- Each node has two pointers to left and right child

```
struct treeNode{
    int x;
    struct treeNode *lchild;
    struct treeNode *rchild;
} *root;
typedef treeNode tree;
```

Notes

Operations

Add leaf (l¡p¡=r)

```
tree *add(tree *root, tree *new){
    if(root==NULL)
        return new;
    if(root->x>new->x && root->lchild!=NULL)
        add(root->lchild, new);
    else if(root->x>new->x && root->lchild==NULL){
        root->lchild=new;
    else if(root->x<=new->x && root->rchild!=NULL)
        add(root->rchild, new);
    else
        root->rchild=new;
    return root;
}
```

Notes

Operations

This kind of traverse can be used to print entire tree.

In order traverse

```
void inOrder(tree *root){

    if(root==NULL)
        return;
    inOrder(root->lchild);
    printf("%d,␣", root->x);
    inOrder(root->rchild);
    return;
}
```

Notes

Operations

This kind of traverse can be used to free entire tree.

Post order traverse

```
void postOrder(tree *root){

    if(root==NULL)
        return;
    inOrder(root->lchild);
    inOrder(root->rchild);
    printf("%d,␣", root->x);/*put free(root) to free memory*/
    return;
}
```

Notes

Files in C

- stdio.h provides necessary declarations
- FILE is a structure holding all information about file

File access

```
FILE *fp; /*pointer to file*/
char name[] = "test.txt"; /*name of file*/
char mode[] = "r"; /*mode - read*/
fp = fopen(name, mode);
```

Notes

Possible modes

- r – read
- w – write (overwrites)
- a – adds content to the end of the file
- b – required for binary files in some cases

If file does not exist and is opened in "w" or "a" mode it is created. Opening file that does not exist in "r" mode causes error (fopen returns NULL).

Notes

How can we read or write file?

Similarly to the standard input there are several possible ways of reading input from files:

- simplest one
  - int getc(FILE *fp) – reads next char from file, returns EOF for end of file or error
  - int getc(int c, FILE *fp) – writes a character c to the file and returns written char or EOF if error occurs
- formatted I/O, works like scanf and printf
  - int fscanf(FILE *fp, char *format, ...)
  - int fprintf(FILE *fp, char *format, ...)

Notes

Closing file!

```
fclose(FILE *fp);
```

- closes a file pointed by fp
- brakes a connection between program and file
- **Flushes a buffer where output of putc is collected** (you can use `int fflush(FILE *fp)` to do it without closing file

Notes

Example

Let's write a program that will write an input to the file provided as a parameter.

Notes

System calls library

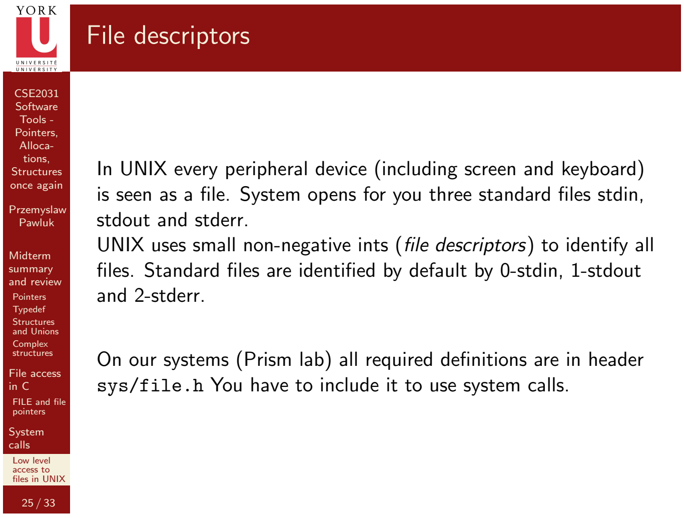
System interface

UNIX allows us to use several services through a set of *system calls*, which are functions of operating system that may be called by our programs.

Why system calls?

It is to show you how previously described functions are implemented with functionality provided by UNIX OS.

Notes

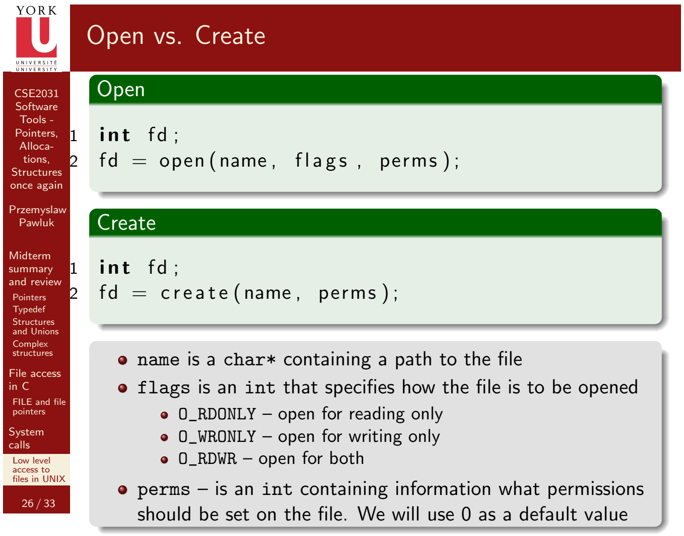


File descriptors

In UNIX every peripheral device (including screen and keyboard) is seen as a file. System opens for you three standard files stdin, stdout and stderr.
UNIX uses small non-negative ints (*file descriptors*) to identify all files. Standard files are identified by default by 0-stdin, 1-stdout and 2-stderr.

On our systems (Prism lab) all required definitions are in header `sys/file.h` You have to include it to use system calls.

Notes



Open vs. Create

Open

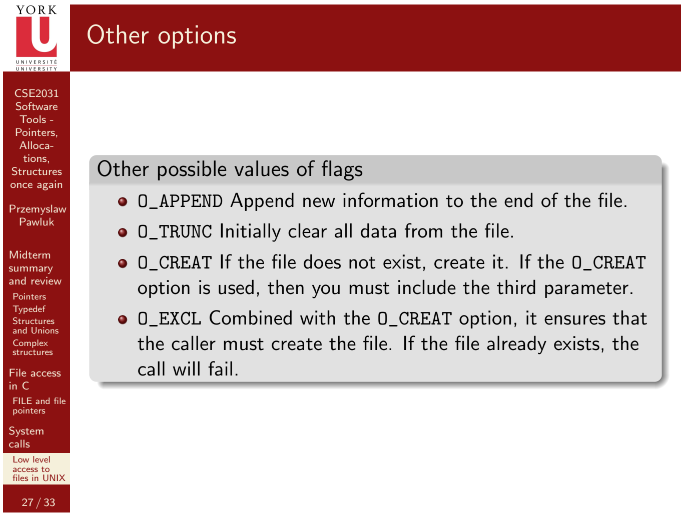
```
int fd;
fd = open(name, flags, perms);
```

Create

```
int fd;
fd = create(name, perms);
```

- `name` is a `char*` containing a path to the file
- `flags` is an `int` that specifies how the file is to be opened
  - `O_RDONLY` – open for reading only
  - `O_WRONLY` – open for writing only
  - `O_RDWR` – open for both
- `perms` – is an `int` containing information what permissions should be set on the file. We will use 0 as a default value

Notes



Other options

Other possible values of flags

- `O_APPEND` Append new information to the end of the file.
- `O_TRUNC` Initially clear all data from the file.
- `O_CREAT` If the file does not exist, create it. If the `O_CREAT` option is used, then you must include the third parameter.
- `O_EXCL` Combined with the `O_CREAT` option, it ensures that the caller must create the file. If the file already exists, the call will fail.

Notes

Permissions

Values of perms

- `S_IRUSR` Set read rights for the owner to true.
- `S_IWUSR` Set write rights for the owner to true.
- `S_IXUSR` Set execution rights for the owner to true.
- `S_IRGRP` Set read rights for the group to true.
- `S_IWGRP` Set write rights for the group to true.
- `S_IXGRP` Set execution rights for the group to true.
- `S_IROTH` Set read rights for other users to true.
- `S_IWOTH` Set write rights for other users to true.
- `S_IXOTH` Set execution rights for other users to true.

Notes

Close

- brakes connection between descriptor and file
- frees the file descriptor so it can be used for another file
- it is done by system on `exit` or `return` from main.

Notes

Unlink

- **Removes** the file pointed by name from the **file system**!
- It corresponds to `remove` from standard library
- Look out there is no warning before removing!!!

Notes

File access - read and write

```
int read(int fd, char *buf, int n);
int write(int fd, char *buf, int n);
```

- `fd` – file descriptor
- `buf` – an array of characters where the data is to go to or came from
- `n` – number of bytes to be transfered
- Both return a number of bytes transfered (read or wrote)

Notes

Example

Let's write a program that will implement a copy functionality. It takes two paths and copy first into second.

Notes

Random access

Read and write are normally sequential. We can use, however, `lseek` function to move our cursor in the file into any place.

```
long lseek(int fd, long offset, int origin);
```

sets the position in the file whose descriptor is `fd` to `offset` calculated relatively to the location specified by `origin` Origin can be:

- 0 – means offset is calculated from the beginning of the file
- 1 – means offset is calculated from current position
- 2 – means offset is calculated from the end of the file

```
/*go to the beginning of the file*/
lseek(fd, 0L, 0);

/*go to the end of the file*/
lseek(fd, 0L, 2);
```

Notes