MP1 and MP2 - Threads

CS241 Discussion Section

Week 3

February 11, 2010

Outline

- MP1 Issues
- MP2 Overview
- File I/O
- POSIX threads - pthreads

MP1 Issues

- Q: My **`printf`** prints my string correctly but appends a lot of garbage after it. Why?

MP1 Issues

- Q: My **`printf`** prints my string correctly but appends a lot of garbage after it. Why?

- A: The line probably does not have a termination char ('**`\0`**')

MP1 Issues

- Q: Why there's no `'\0'` at the end of my string?

MP1 Issues

- Q: Why there's no `'\0'` at the end of my string?

- A: Some functions (e.g., **`strncpy()`**, **`strncat()`**), do not add the **`\0`** at the end of the string

- S: add the termination char manually:

```
    strncpy(str,src,i);
    str[i]='\0';
```

MP1 Issues

What’s wrong with this code?

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    char *str =  (char *) malloc(5);
   strcpy(str,"Hello");
   printf("%s",str);

}
```

MP1 Issues

What’s wrong with this code?

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    char *str =  (char *) malloc(5);
   strcpy(str,"Hello");
   printf("%s",str);

}
```

No space for **\0**

Valgrind helps finding these errors:

```
==15648== Invalid write of size 1
==15648==    at 0x4027167: memcpy (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==15648==    by 0x804847A: main (a.c:8)
==15648==  Address 0x419802d is 0 bytes after a block of size 5 alloc'd
```

MP1 Issues

What’s wrong with this code?

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
   int i;
   char str[6];

   scanf("%s",str);
   if (!strcmp(str,"A")) i=1;
   if (!strcmp(str,"B")) i=2;
   printf("%d",i);

}
```

MP1 Issues

What’s wrong with this code?

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
   int i;
   char str[6];

   scanf("%s",str);
   if (!strcmp(str,"A")) i=1;
   if (!strcmp(str,"B")) i=2;
   printf("%d",i);

}
```

`i` might not be initialized

Valgrind helps finding these errors:

```
==15721== Conditional jump or move depends on uninitialised value(s)
```

MP1 Issues

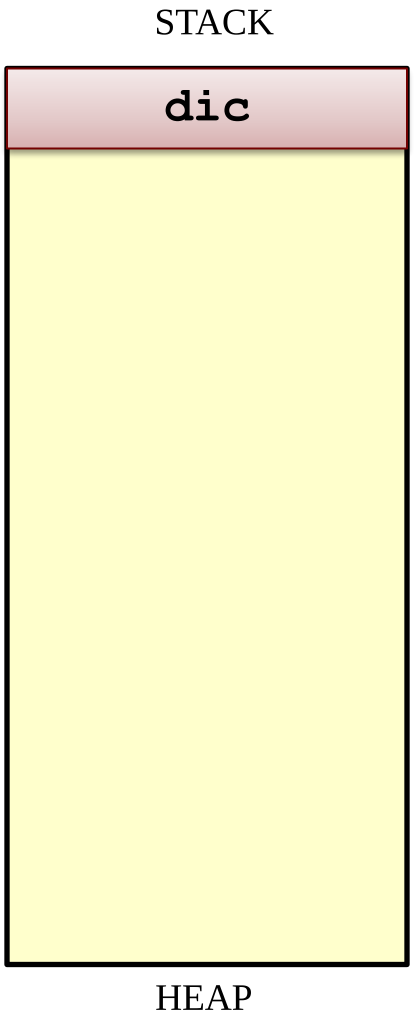
What’s wrong with this code?

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void init(dictionary_t* d){
    d = malloc(sizeof(dictionary_t));
    d->next = NULL;
}

int main(){
    dictionary_t dic;
    init(&dic)
}
```

MP1 Issues

What’s wrong with this code?

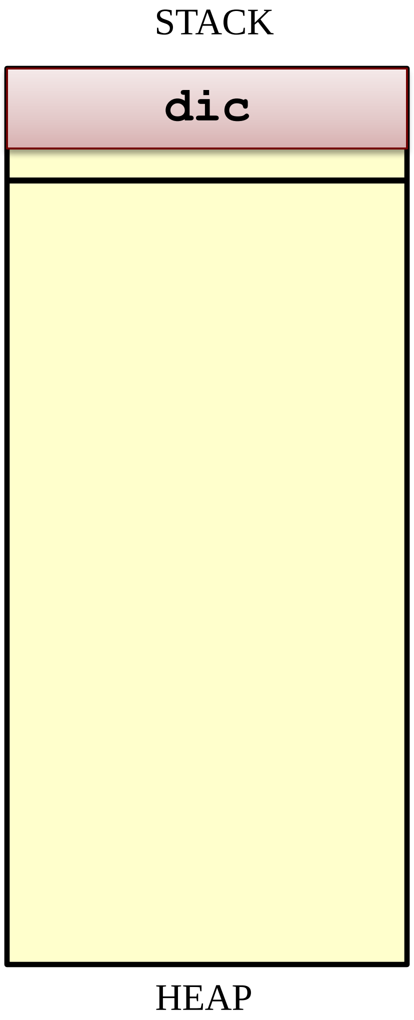
```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void init(dictionary_t* d){
    d = malloc(sizeof(dictionary_t));
    d->next = NULL;
}

int main(){
    dictionary_t dic;
    init(&dic)
}
```



MP1 Issues

What’s wrong with this code?

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void init(dictionary_t* d){
    d = malloc(sizeof(dictionary_t));
    d->next = NULL;
}

int main(){
   dictionary_t dic;
    init(&dic)
}
```



STACK

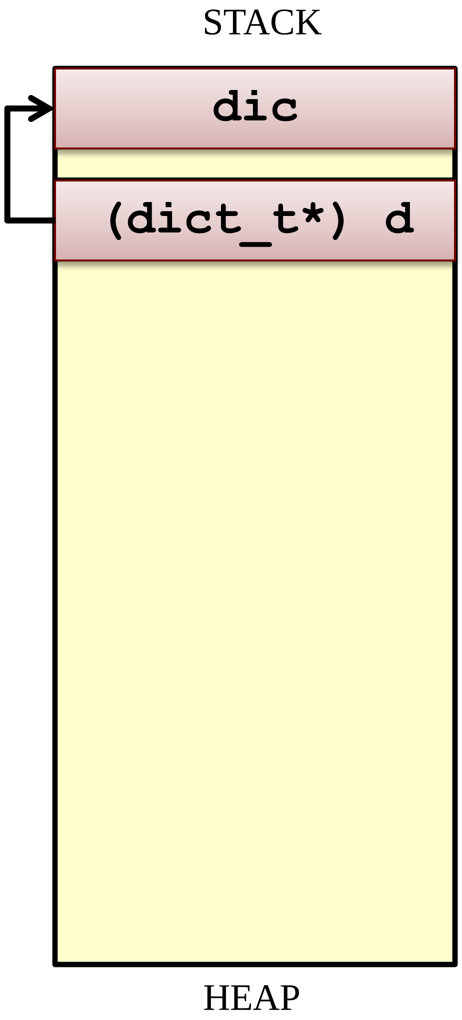
dic

HEAP

MP1 Issues

What’s wrong with this code?

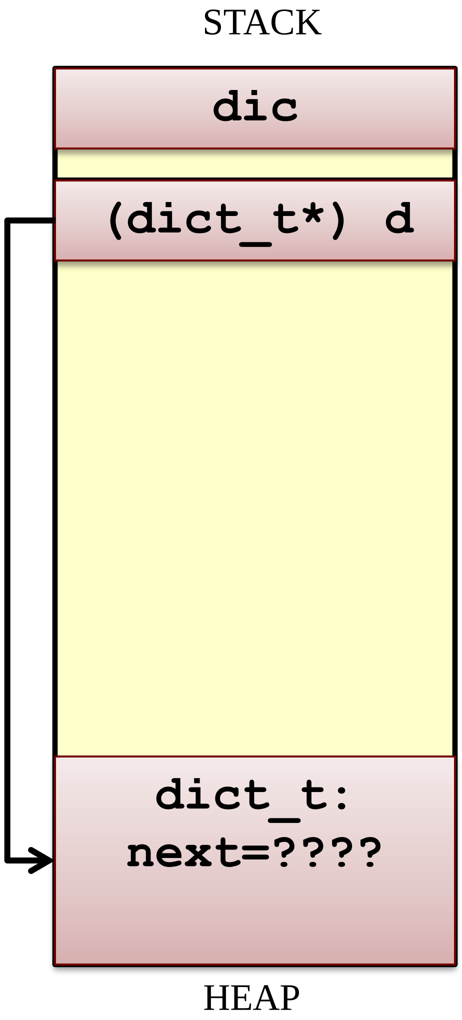
```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void init(dict_t* d){
    d = malloc(sizeof(dictionary_t));
    d->next = NULL;
}

int main(){
   dict_t dic;
    init(&dic)
}
```



MP1 Issues

What’s wrong with this code?

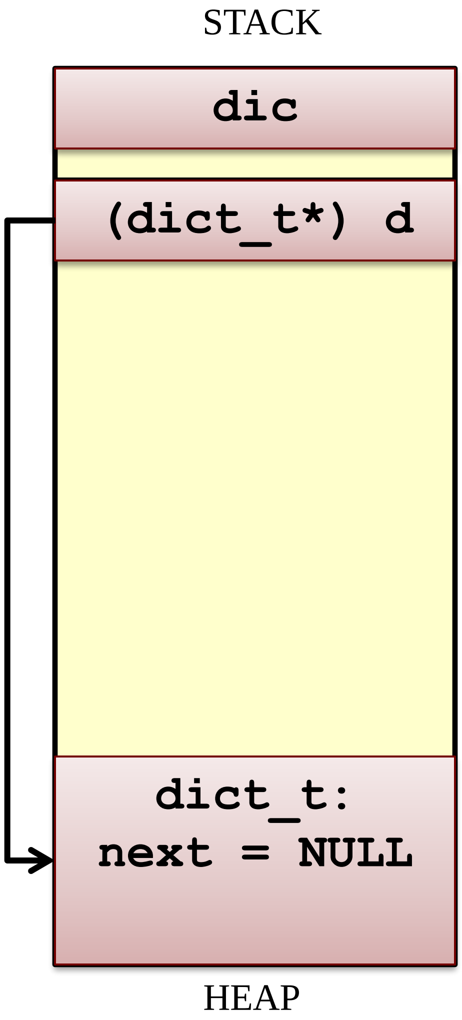
```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void init(dictionary_t* d){
    d = malloc(sizeof(dictionary_t));
    d->next = NULL;
}

int main(){
   dictionary_t dic;
    init(&dic)
}
```



MP1 Issues

What’s wrong with this code?

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void init(dictionary_t* d){
    d = malloc(sizeof(dictionary_t));
    d->next = NULL;
}

int main(){
   dictionary_t dic;
    init(&dic)
}
```



STACK

dic

(dict_t*) d

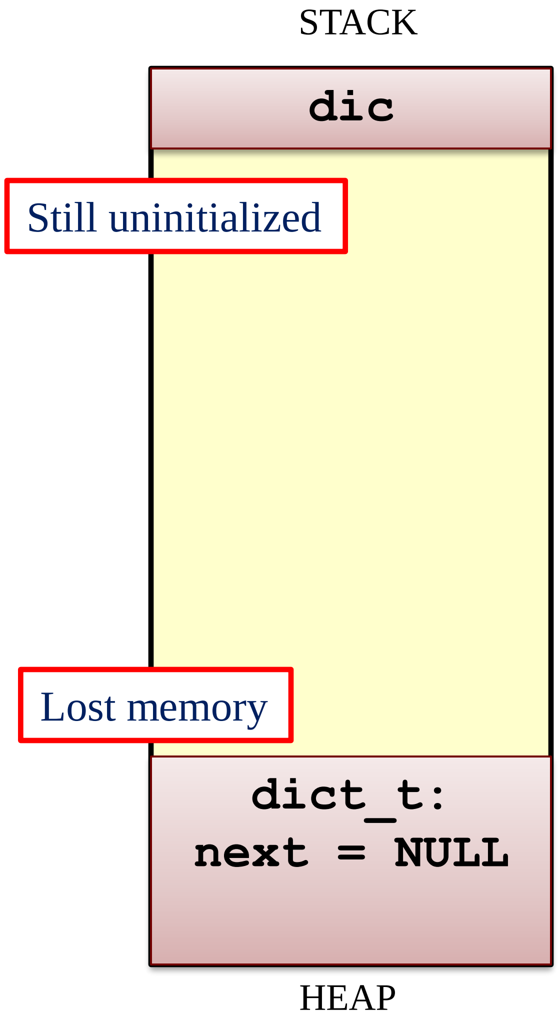
dict_t:
next = NULL

HEAP

MP1 Issues

What’s wrong with this code?

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void init(dictionary_t* d){
    d = malloc(sizeof(dictionary_t));
    d->next = NULL;
}

int main(){
   dictionary_t dic;
    init(&dic)
}
```



STACK

dic

Still uninitialized

Lost memory

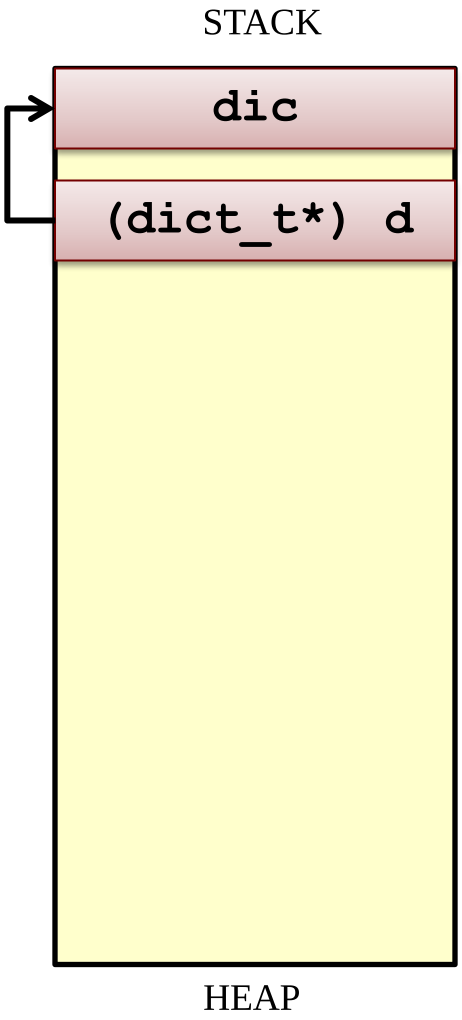
dict_t:
next = NULL

HEAP

MP1 Issues

Solution:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void init(dictionary_t* d){
    ~~d = malloc(sizeof(dictionary_t));~~
    d->next = NULL;
}

int main(){
   dictionary_t dic;
    init(&dic)
}
```



MP2 Overview

MP2 is an introduction to threads

Goal: sort an enormous data set in parallel using threads

MP2 Overview

Part 1: [Multi-threaded sorting]

Each input file is sorted by a different thread, and the output is saved to a file with the same name plus "`.sorted`".

Ignore empty lines.

<u>Reverse </u>lexicographical (alphabetical) order.

Use **`qsort`**:

```
void qsort(void *base, size_t nmemb, size_t size,
                 int(*compar)(const void *, const void *));
```

Pointer to a function

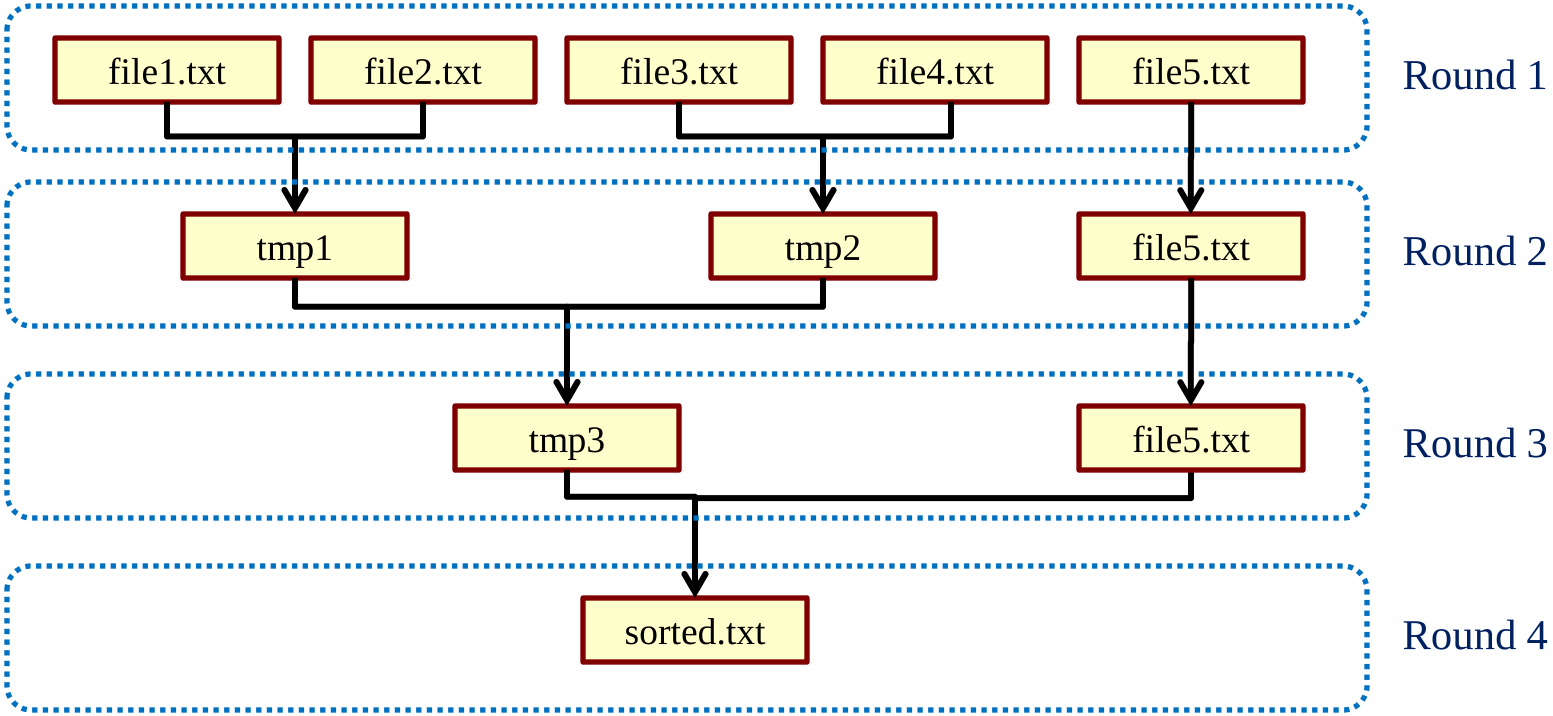
MP2 Overview

Part 2: [Multi-threaded merging]

Each pair of files is merged until only one is left.

A new round is started when all files in the previous one are merged.

Ignore duplicates and emptylines.



File I/O

I/O in C

MP2 requires you to read and write text files in C.

Two primary means of doing I/O in C:

Through lightly-wrapped system calls

`open()`, `close()`, `read()`, `write()`, etc

Through C-language standards

`fopen()`, `fclose()`, `fread()`, `fwrite()`, etc

I/O in C

Opening a file (Method #1):

```
fopen(const char *filename, const char *mode);
```

filename: path to file to open

mode: what do you wish to do with the file?

- **`r`**: read only
- **`r+`**: read and write (file must already exist)

- **`w`**: write (or overwrite) a file
- **`w+`**: write (or overwrite) a file and allow for reading

- **`a`**: append to the end of the file (works for new files, too)
- **`a+`**: appends to end of file and allows for reading anywhere in the file; however, writing will always occur as an append

I/O in C

Opening a file (Method #2):

```
open(const char *filename, int flags, int mode);
```

filename: path to file to open

flags: what do you wish to do with the file?

One of the following is required:

O_RDONLY, **O_WRONLY**, **O_RDWR**

And any number of these flags (yo "add" these flags, simply binary-OR them together):

O_APPEND: Similar to "a+" in fopen()

O_CREAT: Allows creation of a file if it doesn't exist

O_SYNC: Allows for synchronous I/O (thread-safeness)

mode: what permissions should the new file have?

(**S_IRUSR** | **S_IWUSR**) creates a user read-write file.

Opening Files in C

Return value of opening a file:

Having called **`open()`** or **`fopen()`**, they will both create an entry in the OS’s file descriptor table.

Specifics of a file descriptor table will be covered in-depth in the second-half of CS 241.

Both **`open()`** and **`fopen()`** returns information about its file descriptor:

`open()`: Returns an int.

`fopen()`: Returns a **`(FILE *)`**.

Reading Files in C

Two ways to read files in C:

```
fread(void *ptr, size_t size, size_t count, FILE *s);
```

`*ptr`: Where should the data be read into?

`size`: What is the size of each piece of data?

`count`: How many pieces?

`*s`: What (FILE *) do we read from?

```
read(int fd, void *buf, size_t count);
```

`fd:` What file do we read from?

`*buf:` Where should the data be read into?

`count:` How many bytes should be read?

Reading Files in C

Reading more advancely…

`fscanf(FILE *stream, const char *format, …);`

Allows for reading at a semantic-level (eg: ints, doubles, etc) rather than a byte-level.The format string (**`*format`**) is of the same format as **`printf()`**.

`fgets(char *s, int size, FILE *stream);`

reads in at most **`size -1`** characters from stream and stores them into the buffer pointed to by s. Reading stops after an **`EOF`** or a newline. If a newline is read, it is stored into the buffer. A **`'\0’`** is stored after the last character in the buffer.

Writing Files in C

Writing is a lot like reading…

```
fwrite(void *ptr, size_t size, size_t count, FILE *s);
```

Writing of bytes with (**`FILE *`**).

```
write(int fd, void *buf, size_t count);
```

Writing of bytes with a file descriptor (**`int`**)

```
fprintf(FILE *stream, const char *format, …);
```

Formatted writing to files (works like **`printf()`**)

Closing Files in C

Always close your files!

```
fclose(FILE *stream);
close(int fd);
```

`write()`, and especially **`fwrite()/fprintf()`**, may be buffered before being written out to disk.

If a file is never closed after writing:

- the new data may never be written on the actual file;
- the files may be corrupted.

Function Pointers

Passing Functions in C

In this MP, you must use **`qsort()`**:

```
void qsort (void *base, size_t num, size_t size,
         int (*comparator)(const void *, const void *));
```

Passing Functions in C

In this MP, you must use **`qsort()`**:

```
void qsort (void *base, size_t num, size_t size,
         int (*comparator)(const void *, const void *));
```

Requires a function of the following format:

```
    int ___(const void *a, const void *b);
```

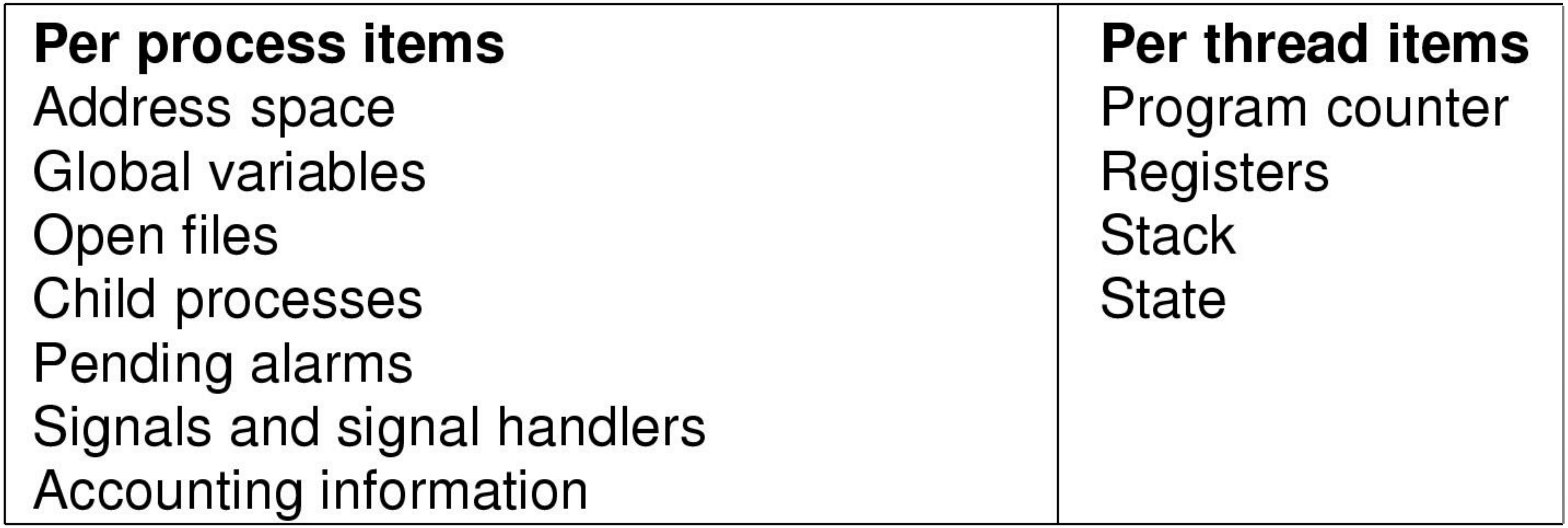
That function should return:

(negative)	if	(first param) < (second param)
0	if	(first param) == (second param)
(positive)	if	(first param) > (second param)

`a` and **`b`** are **pointers** to the elements being sorted.

Threads

Threads vs. Processes



Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Each thread execute separately

Threads in the same process share resources

No protection among threads!!

POSIX Threads (Pthreads)

Standardized, portable thread API

To use POSIX thread functions

```
#include <pthread.h>
gcc –o main main.c -lpthread
```

Creating a thread with pthread

A thread is created with

```
int pthread_create(
     pthread_t *thread,
     const pthread_attr_t *attr,
     void *(*start_routine)(void *),
     void *arg);
```

The creating process (or thread) must provide a location for storage of the thread id.

The third parameter is just the name of the function for the thread to run.

The last parameter is a pointer to the arguments.

Problem 1

Hello World! (thread edition)

We’ll create two threads and one will print out “Hello”, and the other “World”.

Problem 1

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *hello_thread(void *arg)
{
   fprintf(stderr, "Hello ");
   return NULL;
}
```

```
void *world_thread(void *arg)
{
fprintf(stderr, "World!\n");
return NULL;
}
```

Problem 1

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *hello_thread(void *arg)
{
   fprintf(stderr, "Hello ");
   return NULL;
}
```

```
void *world_thread(void *arg)
{
fprintf(stderr, "World!\n");
return NULL;
}
```

```
int main(int argc, char **argv)
{
pthread_t hello, world;
pthread_create(&hello, NULL, hello_thread, NULL);
pthread_create(&world, NULL, world_thread, NULL);
return 0;
}
```

Problem 1

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *hello_thread(void *arg)
{
   fprintf(stderr, "Hello ");
   return NULL;
}
```

```
void *world_thread(void *arg)
{
fprintf(stderr, "World!\n");
return NULL;
}
```

```
int main(int argc, char **argv)
{
pthread_create(&hello, NULL, hello_thread, NULL);
pthread_t hello, world;
pthread_create(&world, NULL, world_thr
return 0;
}
```

What happens here?

Waiting for completion

All running threads are killed when:

- **`main()`** returns;
- any thread calls **`exit()`**.

`pthread_exit(void* retval):`

- If called from any thread exits that thread but does not affect the other running threads
- If thread is joinable returns the pointer to retvalue to the thread that joined the exiting one
- If called in **`main()`** waits for the completion of all threads before terminating the process.

Joining Threads

Pointer to memory where the return value is stored

```
int pthread_join(pthread_t thread, void **value_ptr);
```

The joined thread joined must be joinable. Default setting, but don’t count on it. Set the attributes instead:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
```

Detaching Threads

We have another option:

```
int pthread_detach (pthread_t thread);
```

Lets the system reclaim the thread's resources after it terminates

Good practice:

- call `pthread_detach` or `pthread_join` for each thread
- Explicitly set the attributes for each thread

Problem 1

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *hello_thread(void *arg)
{
    fprintf(stderr, "Hello ");
    return NULL;
}
```

```
void *world_thread(void *arg)
{
fprintf(stderr, "World!\n");
return NULL;
}
```

```
int main(int argc, char **argv)
{
pthread_t hello, world;
pthread_create(&hello, NULL, hello_thread, NULL);
pthread_create(&world, NULL, world_thread, NULL);
pthread_join(hello, NULL);
pthread_join(world, NULL);
return 0;
}
```

Problem 1

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *hello_thread(void *arg)
{
    fprintf(stderr, "Hello ");
    return NULL;
}
```

```
void *world_thread(void *arg)
{
fprintf(stderr, "World!\n");
return NULL;
}
```

```
int main(int argc, char **argv)
{
pthread_t hello, world;
pthread_create(&hello, NULL, hello_thread, NULL);
pthread_create(&world, NULL, world_thread, N
pthread_join(hello, NULL);
pthread_join(world, NULL);
return 0;
}
```

Hello world!

Or

world!Hello

?

Passing Arguments to Threads

```
int pthread_create(
     pthread_t *thread,
     const pthread_attr_t *attr,
     void *(*start_routine)(void *),
     void *arg);
```

Pointer to any data type

Have to cast it to a specific pointer type before dereferencing

Aside: Review of structs in C

Keyword **`struct`** used to define complex data types:

```
typedef struct _stats_t {
    char *longest, *shortest;
    unsigned int numlines;
} stats_t;
```

Structs can contain **variables, arrays, pointers, other structs**…

Can structs contain **pointers to functions?**

Does that remind you of anything?

Can threads have more than one argument?

Yes! Sort of. We can pass a pointer to a struct, e.g.:

```
typedef struct {
     int arg1;
     char *arg2;
} myargs;
void main(){
   myargs a;
   pthread_create(…, myfunc, &a);
}
```

Can threads have more than one argument?

Yes! Sort of. We can pass a pointer to a struct, e.g.:

```
typedef struct {
    int arg1;
    char *arg2;
} myargs;
void main(){
  myargs a;
  pthread_create(…, myfunc, &a);
}
void *myfunc(void *arg){
  myargs *args= (myargs *)arg;
  …
}
```

Thread Return Values

Threads return a **`void*`**, too. Return value can be retrieved by **`pthread_join()`**

Be careful about not returning pointers to local variables!

Concurrency

Threads execute concurrently

- True concurrency on multiple processors
- Interleaving on a uniprocessor machine

All memory, except the stack, is shared between the threads in a process

What happens if multiple threads access a shared variable concurrently?

Modifying a shared variable

- Write a program with global variable **`x = 0`**
- One thread increments it N times (**`x++`**)
- One thread decrements it N times (**`x--`**)
- **`main()`** joins the threads and prints out **`x`**

Modifying a shared variable

```
#include <pthread.h>
#include <stdio.h>
int x=0, N=10000000;
void* inc(void *args){
   int i;
   for (i=0;i<N;i++) x++;
}
void* dec(void *args){
   int i;
   for (i=0;i<N;i++) x--;
}
int main(){
   pthread_t t1,t2;
   int j;
   pthread_create(&t1,NULL,inc,NULL);
   pthread_create(&t2,NULL,dec,NULL);
   pthread_join(&t1,NULL);
   pthread_join(&t2,NULL);
   printf("x = %d\n",x);
}
```

Increase x N times

Decrease x N times

X == 0?

What is going on?

Thread 1

```
x++;
```

```
read x
Increment
write x
```

Thread2

```
x--;
```

```
read x
Decrement
write x
```

What is really going on

Thread 1		Thread2
`read x` (100)		
`Increment` (101)		
Context switch!	→	`read x` (100)
		`Decrement` (99)
		`write x` (99)
`write x` (101)	←	Context switch!

```
x + 1 - 1 = x + 1 !!!
```

A few useful Pthreads functions

POSIX function	**Description**
pthread_create	create a thread
pthread_detach	set thread to release resources
pthread_equal	test two thread IDs for equality
pthread_exit	exit a thread without exiting process
pthread_join	wait for a thread
pthread_self	find out own thread ID