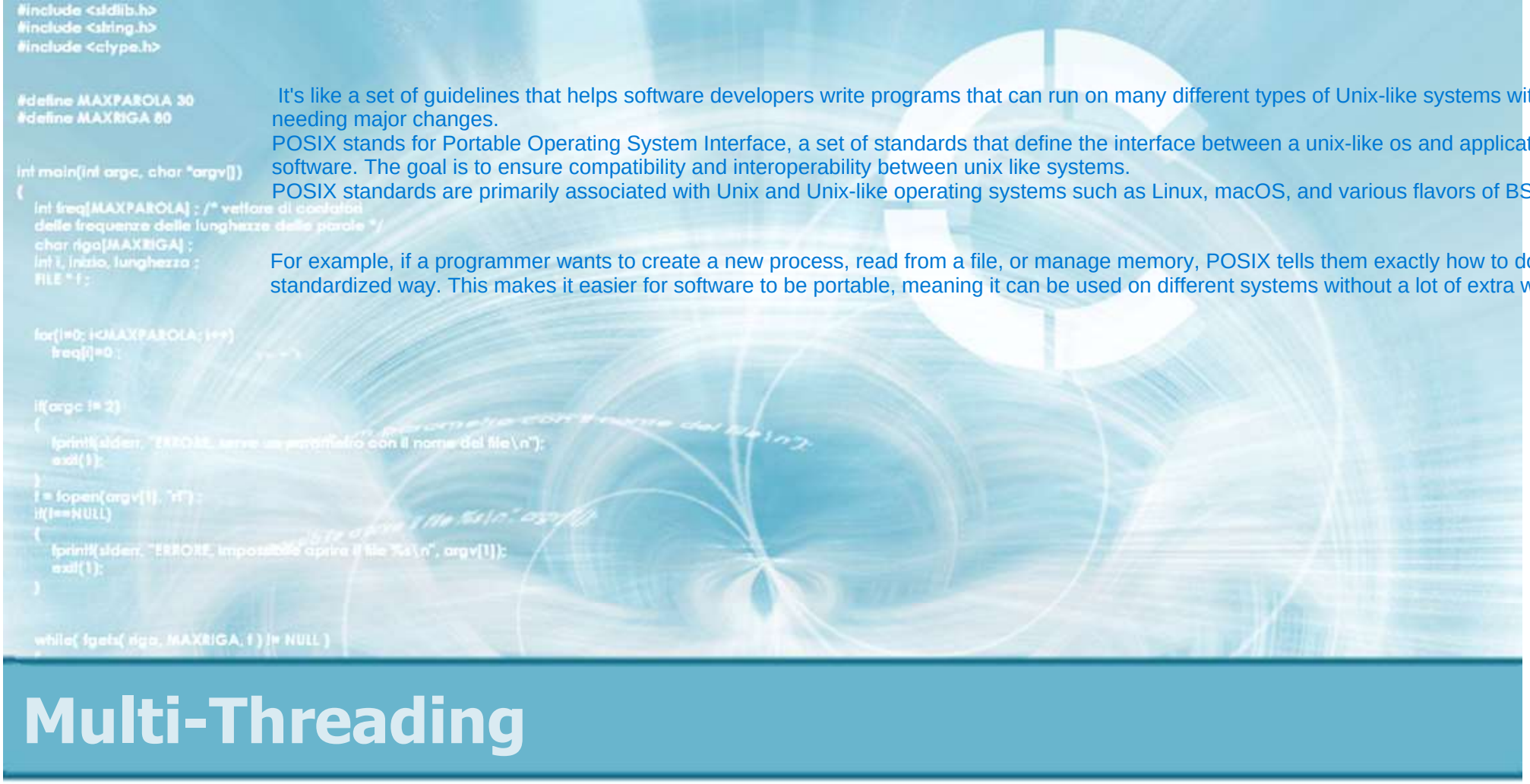


Multi-Threading

It's like a set of guidelines that helps software developers write programs that can run on many different types of Unix-like systems without needing major changes.
POSIX stands for Portable Operating System Interface, a set of standards that define the interface between a unix-like os and application software. The goal is to ensure compatibility and interoperability between unix like systems.
POSIX standards are primarily associated with Unix and Unix-like operating systems such as Linux, macOS, and various flavors of BSD

For example, if a programmer wants to create a new process, read from a file, or manage memory, POSIX tells them exactly how to do it in a standardized way. This makes it easier for software to be portable, meaning it can be used on different systems without a lot of extra work.

Multi-Threading POSIX

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Pthreads

Pthreads (POSIX) is the standard library for handling threads in unix-like os. It defines a set of functions and data structures for creating, controlling, and synchronizing threads.

- POSIX threads or Pthreads
  - Is the standard UNIX library for threads
    - The version POSIX 1003.1c was born in 1995
    - Revised in version IEEE POSIX 1003.1-2017
  - Defined for the C language, but available for other languages (e.g., FORTRAN)
- Using Pthreads
  - A thread is a **function** that is executed in concurrency with the main thread

A process with multiple threads is a set of independent that share the process resources

Pthreads

- The Pthreads library allows
  - Creating and manipulating threads
  - Synchronizing threads
  - Protection of resources shared by threads
  - Thread scheduling
  - Destroying thread
- It defines more than 60 functions
  - All functions have the prefix **pthread_**
    - pthread_equal, pthread_self, pthread_create, pthread_exit, pthread_join, pthread_cancel, pthread_detach

Library linkage

- The Pthread system calls are defined in pthread.h
  - Insert in all *.c files
    - #include <pthread.h>
  - Link programs with the pthread library
    - gcc -Wall -g -o <exeName> <file.c> -lpthread

Thread Identifier

- A thread is uniquely identified
  - By a type identifier **pthread_t**
    - Similar to the PID of a process (`pid_t`)
  - The type **pthread_t** is opaque
    - Its definition is implementation dependent
    - Can be used only by functions specifically defined in Pthreads
    - It is not possible compare directly two identifiers or print their values
  - It has meaning only within the process where the thread is executed
    - Remember that the PID is global within the system

The pthread_t identifier is meaningful only within the process where the thread is executed. Unlike process IDs, which are global within the system, pthread_t identifiers are local to the process. This means that two threads in different processes can have the same pthread_t value without any conflict.

System call pthread_equal

```
int pthread_equal (
  pthread_t tid1,
  pthread_t tid2
);
```

The pthread_equal function is a useful tool for comparing two thread identifiers (pthread_t values) to determine if they represent the same thread.

- Compares two thread identifiers
- Arguments
  - Two thread identifiers
- Returned values
  - Nonzero if the two threads are equal
  - Zero otherwise

System call pthread_create

```
pthread_t pthread_self (
   void
);
```

pthread_self returns the thread identifier of the calling thread. It's often used by a thread to identify itself within the program.

- Returns the thread identifier of the calling thread
  - It can be used by a thread (with **`pthread_equal`**) to self-identify

Self-identification can be important to properly access the data of a specific thread

System call pthread_create

```
int pthread_create (
   pthread_t *tid,
   const pthread_attr_t *attr,
   void *(*startRoutine)(void *),
   void *arg
);
```

pthread_create is used to create a new thread. It takes several arguments including a pointer to a pthread_t variable where the thread identifier of the newly created thread will be stored, attributes for the thread (or NULL for default attributes), a function pointer to the starting routine of the new thread, and an argument to pass to the starting routine.

Function Pointer: The starting routine is specified as a function pointer of type void *(*start_routine)(void *). This means it's a pointer to a function that takes a single void * argument and returns a void * result.

Return value:
0, on success
error code, on failure

- Arguments
  - Identifier of the generated thread (**tid**)
  - Thread attributes (**attr**)
    - NULL is the default attribute
  - C function executed by the thread (**startRoutine**)
  - Argument passed to the start routine (**arg**)
    - NULL if no argument

```
void *thread_function(void *arg) {
    int *thread_id_ptr = (int *)arg;
    int thread_id = *thread_id_ptr;
    printf("Hello from thread %d!\n",
thread_id);
    // Thread's logic...
    return NULL;
}

int main() {
    pthread_t thread_id;
    int thread_arg = 1; // Example
argument
    pthread_create(&thread_id,
NULL, thread_function,
&thread_arg);
    // Additional logic...
    pthread_join(thread_id, NULL); //
Wait for the thread to finish
    return 0;
}
```

Starting routine refers to the function that will be executed by the newly created thread when it starts running.
This function, referred to as the starting routine, contains the code that the new thread will execute. It's the entry point for the new thread's execution. Whatever functionality you want the new thread to perform should be implemented within this function.
Any data that needs to be passed can be done through void *arg argument of the pthread

A **single** argument

System call pthread_exit

- A whole process (with all its threads) terminates if
  - Its thread calls **exit** (or **_exit** or **_Exit**)
  - The main thread execute **return**
  - The main thread receives a signal whose action is to terminate
- A single thread can terminate (without affecting the other process threads)
  - Executing **return** from its start function
  - Executing **pthread_exit**
  - Receiving a cancellation request performed by another thread using **pthread_cancel**

System call pthread_exit

```
void pthread_exit (
  void *valuePtr
);
```

When a thread calls pthread_exit, it immediately terminates, and the control returns to the thread that created it or to the main thread if it's the main thread itself.

Thread Termination: pthread_exit is used by a thread to terminate itself. When called, the thread stops executing and returns a termination status.

valuePtr: This argument allows the thread to return a termination status or value. It's a pointer to the value that the thread wants to pass to the thread that calls pthread_join when it waits for this thread to terminate.

- It allows a thread to terminate returning a termination status
- Arguments
  - The **ValuePtr** value is kept by the kernel until a thread calls **pthread_join**
  - This value is available to the thread that calls **pthread_join**

Example

Thread creation of 1 thread without parameters

```
void *tF () {
   ...
   pthread_exit (NULL);
}
```

Attributes

Arguments

```
pthread_t tid;
int rc;
rc = pthread_create (&tid, NULL, tF, NULL);
if (rc) {
   // Error ...
   exit (-1);
}
...
pthread_exit (NULL);
// exit (0);
// return (0); (in main)
```

Terminates only the main thread

This code creates a simple multithreaded program with one thread. The main thread creates a new thread using pthread_create, and both threads print a message indicating their execution. Finally, both threads terminate using pthread_exit(NULL).

This program prints,
Main thread is running
Thread is running.
In practice, the order in which threads are scheduled to run can depend on various factors, including the operating system's scheduling algorithm, system load, and other concurrent activities happening on the system.

So, while the main thread is typically scheduled to run first, it's not guaranteed, and the actual order of execution may vary.

Terminates the process (all its threads)

Example

Creation of N threads with 1 argument

A thread can be executed when t is changed

```
void *tF (void *par) {
  int *tidP, tid;
  ...
  tidP = (int *) par;
  tid = *tidP;
  ...
  pthread_exit (NULL)
}
```

The content is being modified by the main thread

```
pthread_t th[NUM_THREADS];
int rc, t;

for (t=0; t<NUM_THREADS; t++) {
  rc = pthread_create (&th[t], NULL, tF,
    (void *) &t);
  if (rc) {...}
}
pthread_exit(NULL);
```

ERROR

`&t` is the address of a variable, the main thread changes its content in concurrency with the created threads that read its value

Example

Creation of N threads with 1 argument

Cast of a value
`void *` ↔ **`long int`**

```
void *tF (void *par) {
  long int tid;
  ...
  tid = (long int) par;
  ...
  pthread_exit(NULL);
}
```

```
pthread_t th[NUM_THREADS];
int rc; long int t;

for (t=0; t<NUM_THREADS; t++) {
  rc = pthread_create (&th[t], NULL, fF,
    (void *) t);
  if (rc) { ... }
}
pthread_exit (NULL);
```

Tricky:
We pass a **`long int`** as it were an address, because **`pthread_create`** requires an address as its last argument

Example

Creation of N threads with 1 struct

```
struct tS {
  int tid;
  char str[N];
};
```

```
void *tF (void *par) {
  struct tS *tD;
  int tid; char str[L];

  tD = (struct tS *) par;
  tid = tD->tid; strcpy (str, tD->str);
  ...
```

Cast to a vector of structs

```
pthread_t t[NUM_THREADS];
struct tS v[NUM_THREADS];
...
for (t=0; t<NUM_THREADS; t++) {
  v[t].tid = t;
  strcpy (v[t].str, str);
  rc = pthread_create (&t[t], NULL, tF, (void *) &v[t]);
  ...
  }
...
```

Address of a struct

System call pthread_join

- At its creation a thread can be declared
  - Joinable
    - Another thread may "wait" (**pthread_join**) for its termination, and collect its exit status
    - The termination status of the trhead is retained until another thread performs a **pthread_join** for that thread
  - Detached
    - No thread can explicitly wait for its termination (not joinable)
    - The termination status of the thread is immediately released

System call pthread_join

```
int pthread_join (
  pthread_t tid,
  void **valuePtr
);
```

Return value:
0, on success
error code, on failure

- Arguments
  - Identifier (tid) of the waited-for thread
  - The void pointer **ValuePtr** will be the value returned by thread **tid**
    - Returned by **pthread_exit** or by **return**
    - **PTHREAD_CANCELED** if the thread was deleted
    - Can be set to NULL if you are not interested in the return value

Example

Returns the exit status (**tid** in this example)

```
void *tF (void *par) {
  long int tid;
  ...
  tid = (long int) par;
  ...
  pthread_exit ((void *) tid);
}
```

```
void *status;
long int s;
...
/* Wait for threads */
for (t=0; t<NUM_THREADS; t++) {
  rc = pthread_join (th[t], &status);
  s = (long int) status;
  if (rc) { ... }
}
...
```

th[t] collects the **tids**

Wait for each thread, and collects its exit status

System call pthread_cancel

```
int pthread_cancel (
  pthread_t tid
);
```

Return value:
0, on success
error code, on failure

- Terminates the target thread
- The thread calling **pthread_cancel** does not wait for termination of the target thread (it continues immediately after the calling)
- Arguments
  - Target thread (tid) identifier

System call pthread_detach

```
int pthread_detach (
  pthread_t tid
);
```

Return value:
0, on success
error code, on failure

- Declares thread **tid** as detached
  - The status information will not be kept by the kernel at the termination of the thread
  - No thread can join with that thread
    - Calls to **pthread_join** should fail
- Arguments
  - Thread (tid) identifier

Example

Create a thread and then make it detached

```
pthread_t tid;
int rc;
void *status;

rc = pthread_create (&tid, NULL, PrintHello, NULL);
if (rc) { ... }

pthread_detach (tid);

rc = pthread_join (tid, &status);
if (rc) {
  // Error
  exit (-1);
}

pthread_exit (NULL);
```

Detach a thread

Error if try to join

Example

Create a detached thread using the attribute field

```
pthread_attr_t attr;
void *status;

pthread_attr_init (&attr);
pthread_attr_setdetachstate (&attr,
  PTHREAD_CREATE_DETACHED);
  //PTHREAD_CREATE_JOINABLE);

rc = pthread_create (&t[t], &attr, tF, NULL);
if (rc) {...}

pthread_attr_destroy (&attr);

rc = pthread_join (thread[t], &status);
if (rc) {
  // Error
  exit (-1);
}
```

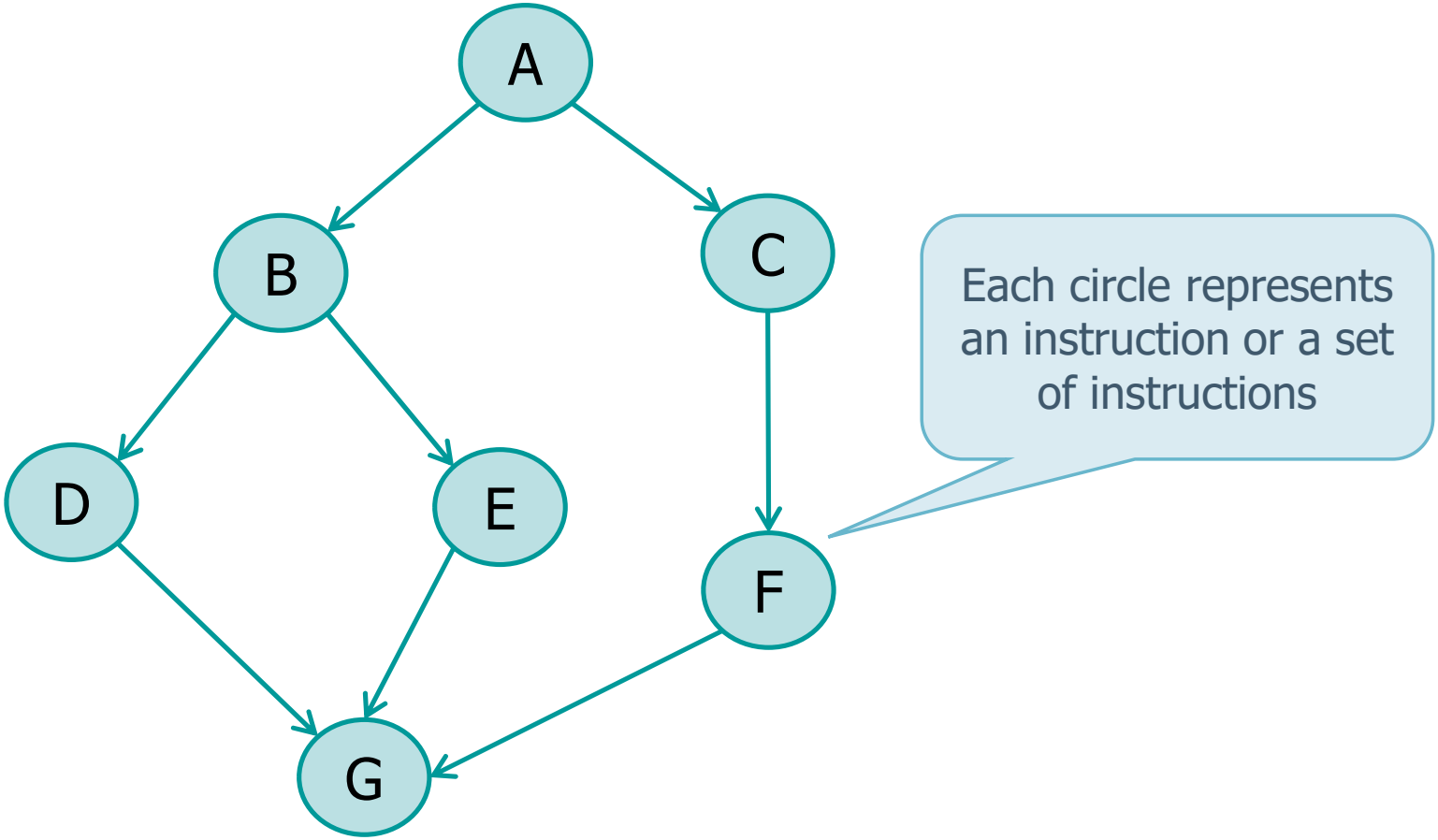
Creates a detached thread

Destroys the attribute object

Error if try to join

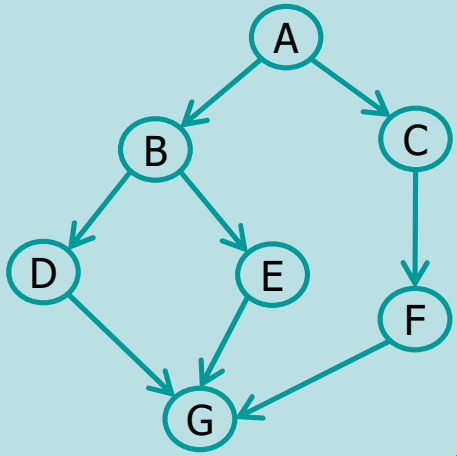
Exercise

- Implement, using threads, the following precedence graph using threads



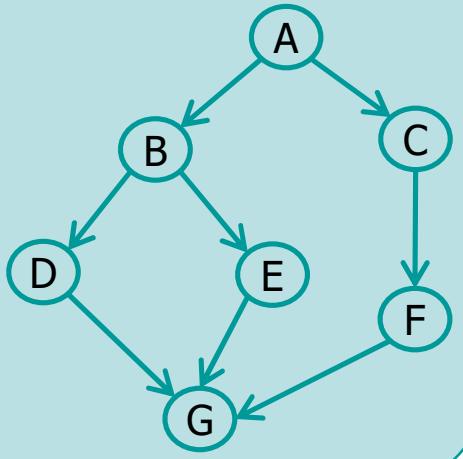
Solution

```
void waitRandomTime (int max){
  sleep ((int)(rand() % max) + 1);
}

int main (void) {
  pthread_t th_cf, th_e;
  void *retval;

  srand (getpid());
  waitRandomTime (10);
  printf ("A\n");
```



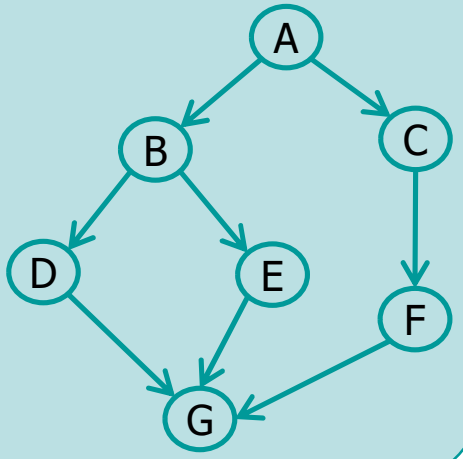
Solution

```
  waitRandomTime (10);
  pthread_create (&th_cf, NULL, CF, NULL);
  waitRandomTime (10);
  printf ("B\n");
  waitRandomTime (10);
  pthread_create (&th_e, NULL, E, NULL);
  waitRandomTime (10);
  printf ("D\n");
  pthread_join (th_e, &retval);
  pthread_join (th_cf, &retval);
  waitRandomTime (10);
  printf ("G\n");

  return 0;
}
```



Solution

```
static void *CF () {
  waitRandomTime (10);
  printf ("C\n");
  waitRandomTime (10);
  printf ("F\n");
  return ((void *) 1); // Return code
}

static void *E () {
  waitRandomTime (10);
  printf ("E\n");
  return ((void *) 2); // Return code
}
```

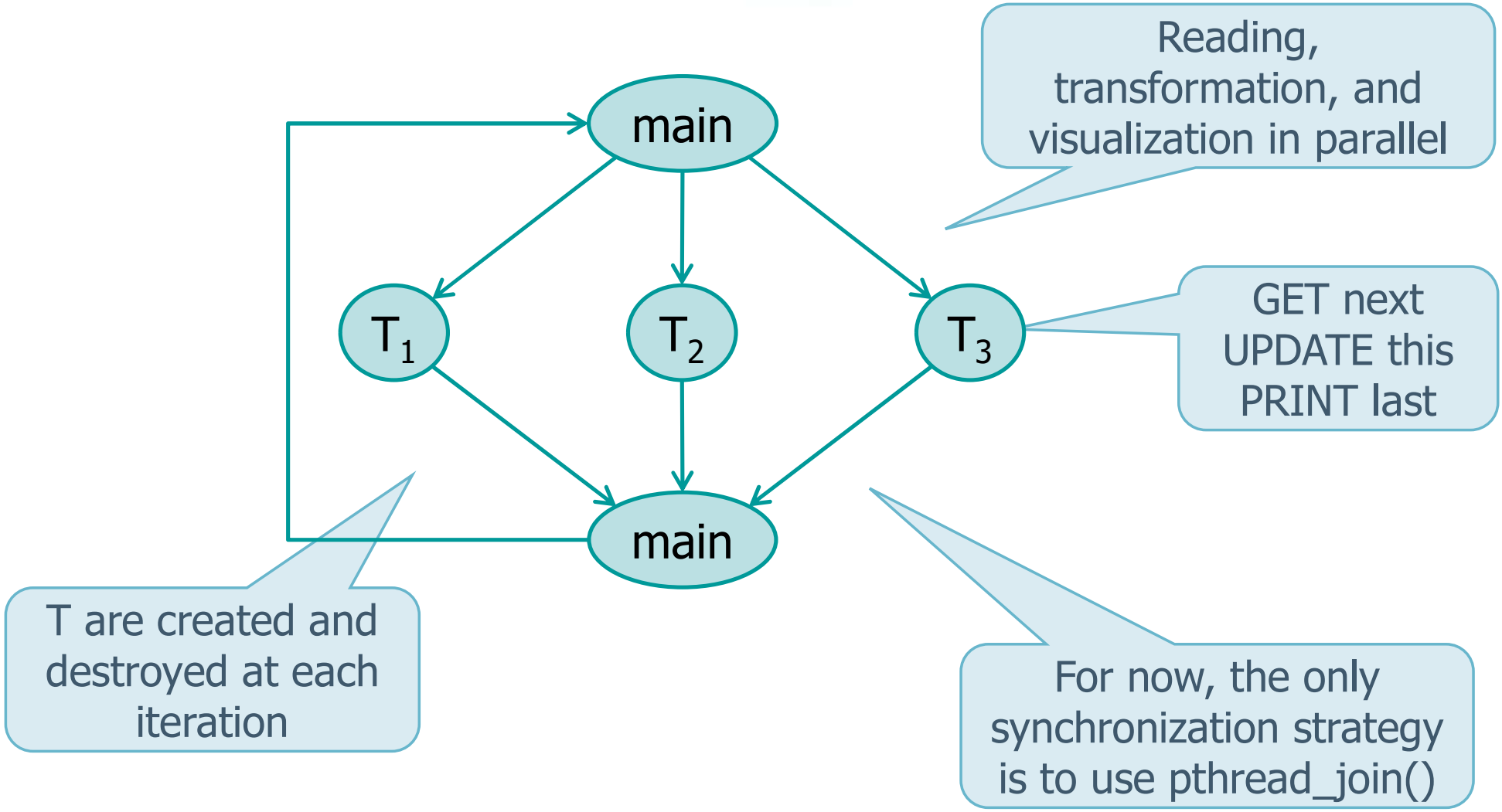


Exercise

- Given a text file, with an undefined number of characters, passed as an argument of the command line
- Implement a concurrent program using three threads (T_1, T_2, T_3) that process the file content in pipeline
  - T_1: Read from file the next character
  - T_2: Transforms the character read by T_1 in uppercase
  - T_3: Displays the character produced by T_2 on standard output

Solution

- Implement, using threads, this precedence graph



Solution

```
static void *GET (void *arg) {
  char *c = (char *) arg;
  *c = fgetc (fg);
  return  NULL;
}

static void *UPD (void *arg) {
  char *c = (char *) arg;
  *c = toupper (*c);
  return  NULL;
}

static void *PRINT (void *arg) {
  char *c = (char *) arg;
  putchar (*c);
  return  NULL;
}
```

Solution

```
FILE *fg;

int main (int argc, char ** argv) {
  char next, this, last;
  int retC;
  pthread_t tGet, tUpd, tPrint;
  void *retV;

  if ((fg = fopen(argv[1], "r")) == NULL){
    perror ("Error fopen\n");
    exit (0);
  }
  this = ' ';
  last = ' ';
  next = ' ';
```

Solution

The first two characters can be managed separately

```
while (next != EOF) {
  retC = pthread_create (&tGet, NULL, GET, &next);
  if (retC != 0) fprintf (stderr, ...);
  retC = pthread_create (&tUpd, NULL, UPD, &this);
  if (retC != 0) fprintf (stderr, ...);
  retC = pthread_create (&tPrint, NULL, PRINT, &last);
  if (retcode != 0) fprintf (stderr, ...);
  retC = pthread_join (tGet, &retV);
  if (retC != 0) fprintf (stderr, ...);
  retC = pthread_join (tUpd, &retV);
  if (retC != 0) fprintf (stderr, ...);
  retC = pthread_join (tPrint, &retV);
  if (retC != 0) fprintf (stderr, ...);
  last = this;
  this = next;
}
```

Solution

Management of the last two characters (queue)

```
  // Last two chars processing
  retC = pthread_create(&tUpd, NULL, UPD, &this);
  if (retC!=0) fprintf (stderr, ...);
  retC = pthread_create(&tPrint, NULL, PRINT, &last);
  if (retC != 0) fprintf (stderr, ...);
  retC = pthread_join (tUpd, &retV);
  if (retC != 0) fprintf (stderr, ...);
  retC = pthread_join (tPrint, &retV);
  if (retC != 0) fprintf (stderr, ...);
  retC = pthread_create(&tPrint, NULL, PRINT, &this);
  if (retC != 0) fprintf (stderr, ...);
  return 0;
}
```