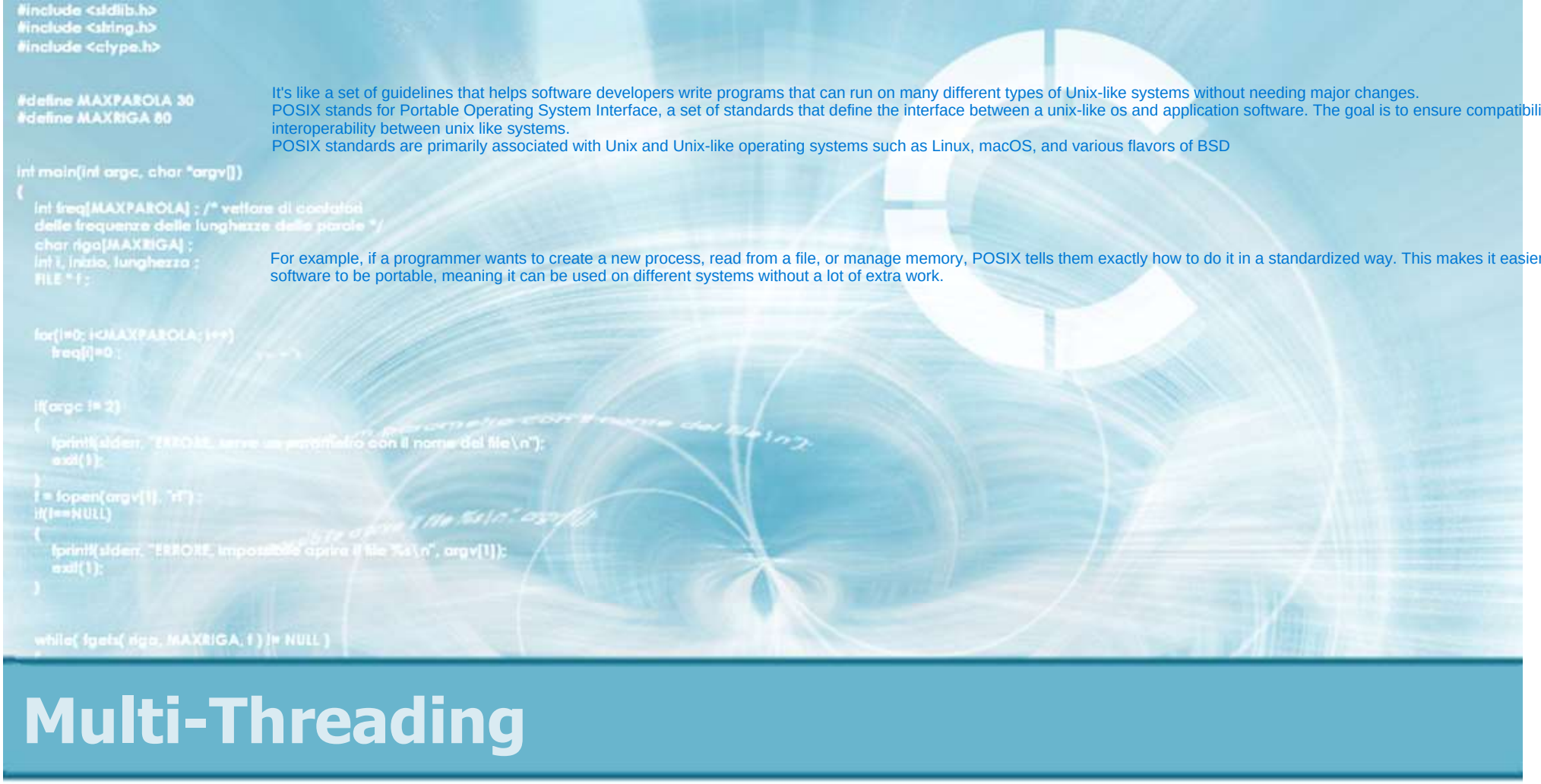


It's like a set of guidelines that helps software developers write programs that can run on many different types of Unix-like systems without needing major changes.
POSIX stands for Portable Operating System Interface, a set of standards that define the interface between a unix-like os and application 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

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**

A whole process, which includes all its threads, can terminate in the following ways:
1. Its thread calls exit (or _exit or _Exit): These functions cause the entire process to terminate.
2. The main thread executes return: When the main function of the main thread returns, it causes the entire process to terminate.
3. The main thread receives a signal whose action is to terminate: If the main thread receives a signal (like SIGTERM or SIGKILL) that is set to terminate the process, the entire process will terminate.

Executing return from its start function:
When a thread is created, it starts executing a specific function (often called the start function).
If this function completes and returns, the thread will terminate.

```
void* threadFunction(void* arg) {
    // Thread work here
    return NULL; // This will terminate the thread
}
```

Executing pthread_exit:
A thread can explicitly call pthread_exit to terminate itself.
This function allows the thread to exit and optionally return a value to any thread that might be joining it.

```
void* threadFunction(void* arg) {
    // Thread work here
    pthread_exit(NULL); // This will terminate the thread
}
```

Receiving a cancellation request performed by another thread using pthread_cancel:
One thread can request the termination of another thread by calling pthread_cancel.
The target thread will terminate if it reaches a cancellation point or if it is set to be cancellable.

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, NULL);
// Some other thread or the main thread can cancel this thread
pthread_cancel(thread); // Request to terminate the thread
```

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)
}
```

This function is the start routine for the new threads.
It takes a single argument (par), which is a pointer to an integer.
tidP = (int *) par; casts the argument to an integer pointer.
tid = *tidP; dereferences the pointer to get the value passed to the thread.
pthread_exit(NULL); terminates the thread.

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);
```

Declares an array to hold thread identifiers.

Declares integers for the return code and loop counter.

for (t = 0; t < NUM_THREADS; t++) { ... }: Loop to create multiple threads.
rc = pthread_create(&th[t], NULL, tF, (void *) &t);: Creates a new thread.
&th[t]: Pointer to the thread identifier.
NULL: Default thread attributes.
tF: The function to be executed by the thread.
(void *) &t: Passes the address of t as the argument to the thread function.
if (rc) { ... }: Checks if the thread creation was successful. If not, it handles the error.
pthread_exit(NULL);: Terminates the main thread, allowing other threads to continue running.

Argument Passing:
The argument passed to the thread function is the address of the loop counter t.
This can lead to issues because t is being modified by the main thread while the created threads are reading its value.
Race Condition:
The slide highlights a potential problem: the main thread changes the value of t in concurrency with the created threads that read its value.
This can lead to unpredictable results because the threads might read the value of t while it is being modified.

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

this slide presents an alternative approach to passing arguments to threads, which avoids the race condition issue discussed in the previous slide.

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);
}
```

This function is the start routine for the new threads
It takes a single argument (par), which is a pointer.
tid = (long int) par; casts the argument to a long int.
pthread_exit(NULL); terminates the thread.

```
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);
```

pthread_t th[NUM_THREADS];: Declares an array to hold thread identifiers.
int rc; long int t;: Declares an integer for the return code and a long int for the loop counter.
for (t = 0; t < NUM_THREADS; t++) { ... }: Loop to create multiple threads.
rc = pthread_create(&th[t], NULL, tF, (void *) t);: Creates a new thread.
&th[t]: Pointer to the thread identifier.
NULL: Default thread attributes.
tF: The function to be executed by the thread.
(void *) t: Passes the loop counter t cast to a void * as the argument to the thread function.
if (rc) { ... }: Checks if the thread creation was successful. If not, it handles the error.
pthread_exit(NULL);: Terminates the main thread, allowing other threads to continue running.

Argument Passing:
Instead of passing the address of t, this approach passes the value of t cast to a void *.
This avoids the race condition because each thread gets a unique value of t.
Casting:
The slide highlights the casting of a long int to a void * and vice versa.
This is a bit tricky because pthread_create expects a void * as its last argument, so the value of t is cast to void * when passed and cast back to long int inside the thread function.

Pointer to the thread identifier.

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];
};
```

An integer to hold the thread ID.

A character array to hold a string.

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

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

The function tF is the start routine for the new threads.
It takes a single argument par, which is a void * (generic pointer).
struct tS *tD;: Declares a pointer to a struct tS.
tD = (struct tS *) par;: Casts the void * argument to a pointer to struct tS.
This means that par is now treated as if it points to a struct tS.
tid = tD->tid;: Accesses the tid member of the structure pointed to by tD.
strcpy(str, tD->str);: Copies the string from the str member of the structure to a local variable str.
pthread_exit(NULL);: Terminates the thread.

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]);
  ...
  }
...
```

pthread_t t[NUM_THREADS];: Declares an array to hold thread identifiers.

struct tS v[NUM_THREADS];: Declares an array of structures to hold the arguments for each thread.

pthread_t t[NUM_THREADS];: Declares an array to hold thread identifiers.
struct tS v[NUM_THREADS];: Declares an array of structures to hold the arguments for each thread.
for (t = 0; t < NUM_THREADS; t++) { ... }: Loop to create multiple threads.
v[t].tid = t;: Sets the tid member of the structure for each thread.
strcpy(v[t].str, str);: Copies a string to the str member of the structure for each thread.
rc = pthread_create(&t[t], NULL, tF, (void *) &v[t]);: Creates a new thread.
&t[t]: Pointer to the thread identifier.
NULL: Default thread attributes.
tF: The function to be executed by the thread.
(void *) &v[t]: Passes the address of the structure as the argument to the thread function.
&v[t] is the address of the t-th element in the array of structures.
(void *) &v[t] casts this address to a void * to match the expected argument type for pthread_create.

Structure as Argument:
Each thread receives a pointer to a unique structure containing its own data.
This avoids the race condition and allows passing multiple pieces of data (e.g., an integer and a string) to each thread.
Casting:
The argument passed to the thread function is cast to a pointer to struct tS.
Inside the thread function, the argument is cast back to struct tS * to access the structure members.

Casting in C:
Casting is used to convert a variable from one type to another.
In this case, we are casting a pointer to a structure (struct tS *) to a generic pointer (void *) and vice versa.
Why Casting is Needed:
pthread_create expects the argument to the thread function to be of type void *.
We need to pass a pointer to our structure (struct tS *), so we cast it to void * when passing it to pthread_create.
Inside the thread function, we cast it back to struct tS * to access the structure members.

Address of a struct

System call pthread_join

A joinable thread is one that another thread can wait for using 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

Another thread can "wait" for its termination using pthread_join.
The termination status of the thread is retained until another thread performs a pthread_join for that thread.
This allows the waiting thread to collect the exit status of the terminated thread.

  - Detached

Definition: A detached thread is one that cannot be waited for using pthread_join.
Behavior:
No thread can explicitly wait for its termination.
The termination status of the thread is immediately released upon termination.
This means that the resources associated with the thread are automatically freed when the thread terminates.

    - No thread can explicitly wait for its termination (not joinable)
    - The termination status of the thread is immediately released

When you create a thread, it is joinable by default.
You can wait for a joinable thread to terminate using pthread_join.
This is useful when you need to ensure that a thread has completed its work before proceeding.

```
pthread_t thread;
 void *status;

 // Create a thread
 pthread_create(&thread, NULL, thread_function, NULL);

 // Wait for the thread to terminate
 pthread_join(thread, &status);

 // Use the status returned by the thread
 printf("Thread returned: %ld\n", (long) status);
```

```
pthread_t thread;
  pthread_attr_t attr;

  // Initialize thread attributes
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

  // Create a detached thread
  pthread_create(&thread, &attr, thread_function, NULL);

  // Destroy the thread attributes
```

System call pthread_join

The pthread_join function is used to wait for a specific thread to terminate.
It allows the calling thread to retrieve the exit status of the terminated thread.

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

pthread_t tid: The identifier of the thread to wait for.

void **valuePtr: A pointer to a pointer where the exit status of the thread will be stored.

This value is returned by pthread_exit or by the thread's return statement.
If the thread was canceled, PTHREAD_CANCELED is returned.
If you are not interested in the return value, you can pass NULL.

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

void **valuePtr: A pointer to a pointer where the exit status of the thread will be stored.
If the thread terminates by calling pthread_exit, the value passed to pthread_exit is stored in valuePtr.
If the thread terminates by returning from its start routine, the return value is stored in valuePtr.
If the thread was canceled, PTHREAD_CANCELED is stored in valuePtr.
If you do not need the exit status, you can pass NULL for valuePtr.

Example

1. Thread Creation:
For t = 0 to 4, pthread_create is called, creating 5 threads.
Each thread runs thread_function, printing its ID and then calling pthread_exit with its ID as the exit status.
2. Thread Joining:
For t = 0 to 4, pthread_join is called, waiting for each thread to terminate.
The exit status of each thread is stored in status.
3. Printing Exit Status:
After each pthread_join, the main thread prints the exit status stored in status.
Since status is a void *, it is cast to long for printing.

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

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

th[t] collects the **tids**

```
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) { ... }
}
...
```

Wait for each thread, and collects its exit status

System call pthread_cancel

pthread_cancel
What it does: Sends a request to a thread to terminate.
Key point: The thread may or may not terminate immediately, depending on its settings.
When to use: When you need to stop a thread before it finishes its work.

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

pthread_t tid: The identifier of the target thread that you want to cancel.

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

Key Points
Cancellation Request:
pthread_cancel only sends a cancellation request; it does not force immediate termination.
The target thread must be in a state where it can respond to the cancellation request.
Resource Cleanup:
When a thread is canceled, it should perform any necessary cleanup (e.g., releasing resources).
This can be done using cleanup handlers or by checking for cancellation points.
Cancellation Points:
Certain functions are cancellation points where a thread checks for pending cancellation requests (e.g., sleep, pthread_join).

System call pthread_detach

pthread_t tid: The identifier of the thread to be detached.

```
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

pthread_detach
What it does: Marks a thread so that its resources are automatically cleaned up when it finishes.
Key point: You can't use pthread_join on a detached thread.
When to use: When you don't need to wait for the thread to finish or get its result.

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);
}
```

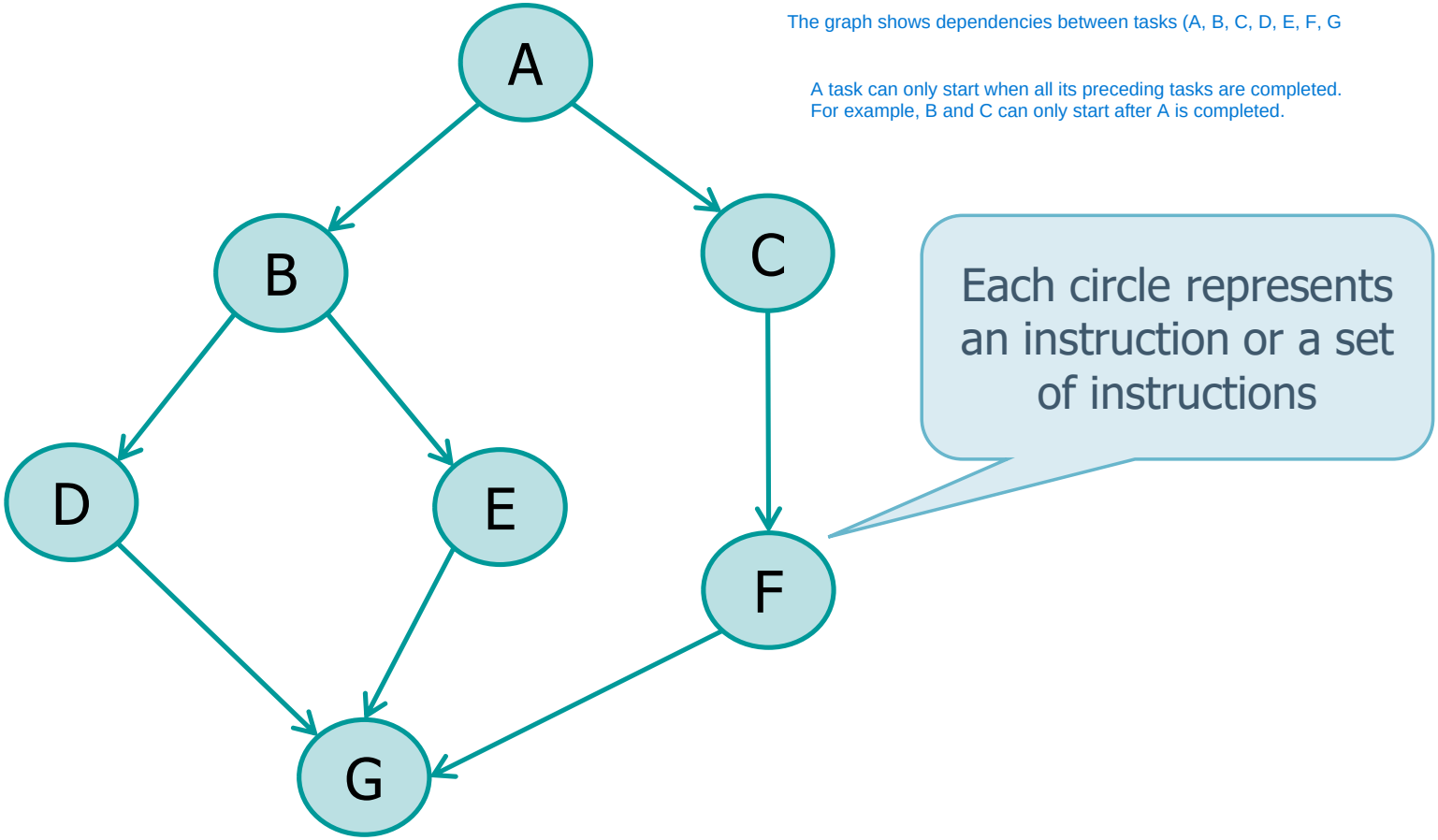
Creates a detached thread

Destroys the attribute object

Error if try to join

Exercise

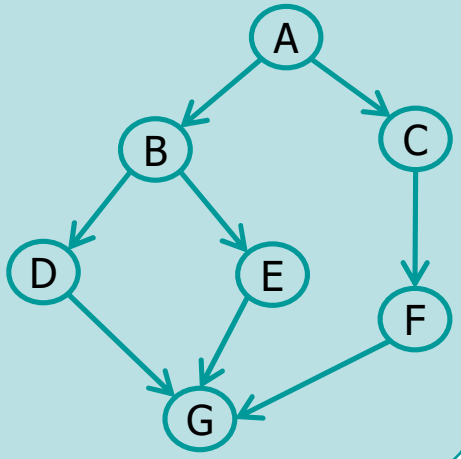
- Implement, using threads, the following precedence graph using threads



Solution

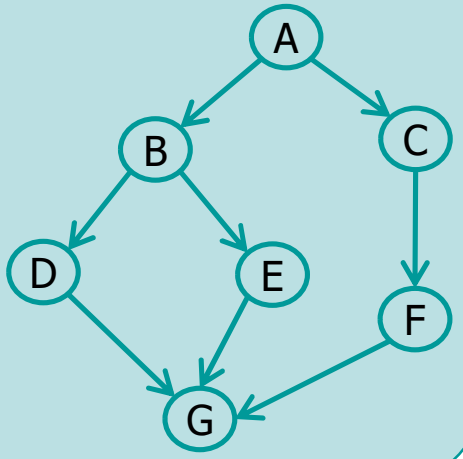
```
void waitRandomTime (int max){
  sleep ((int)(rand() % max) + 1);
}

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

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

Purpose: Simulates a random delay to mimic the execution time of a task.
Parameters: max is the maximum number of seconds to sleep.
Functionality: Sleeps for a random time between 1 and max seconds.



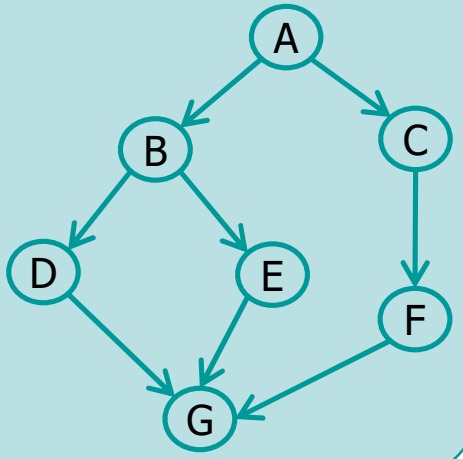
Solution

```
  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
}
```



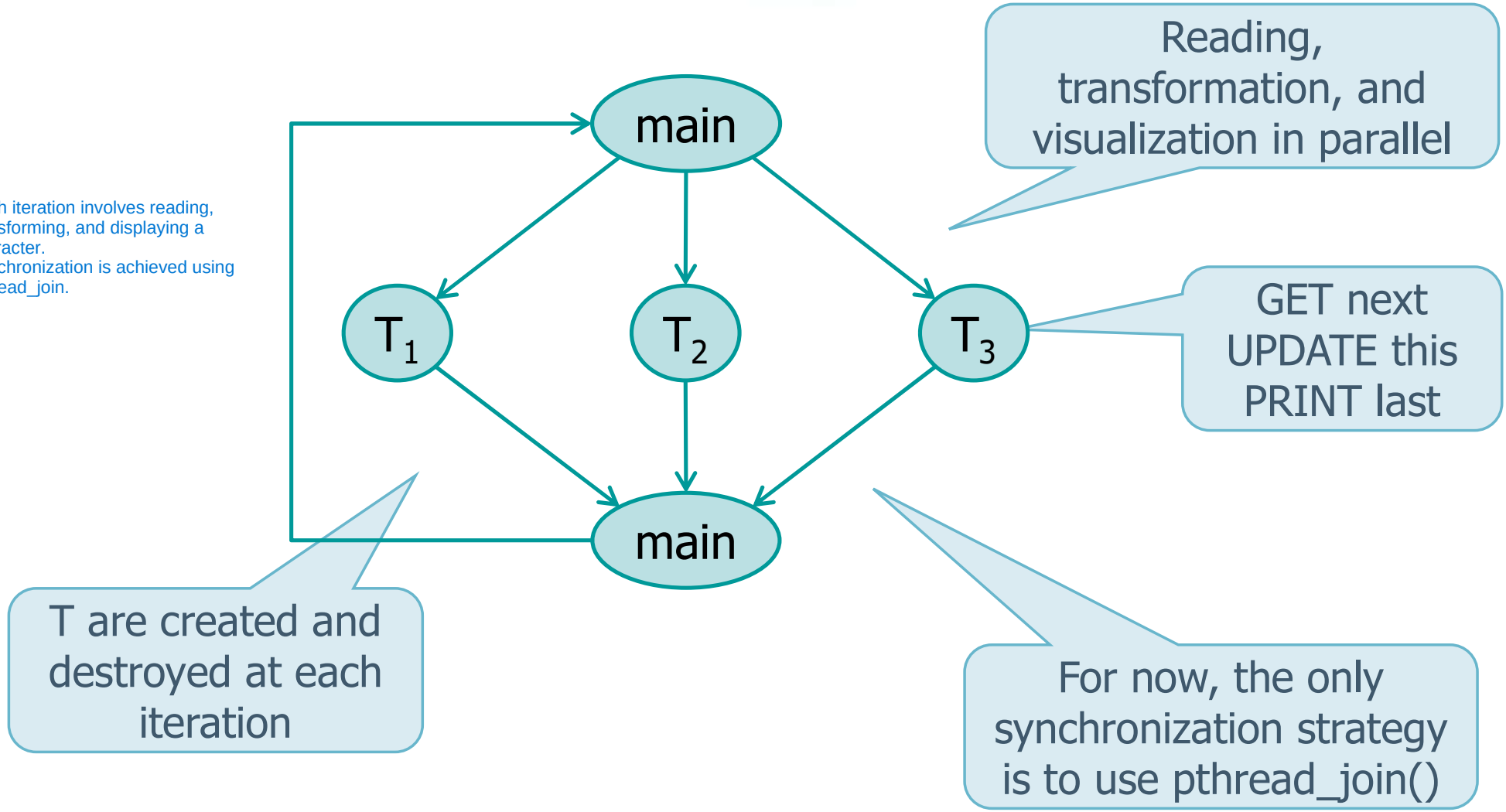
Exercise

- 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

Each iteration involves reading, transforming, and displaying a character.
Synchronization is achieved using pthread_join.



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;
}
```

Purpose: Reads the next character from the file.
Parameters: arg is a pointer to a character where the read character will be stored.
Functionality: Reads a character using fgetc and stores it in *c.

Purpose: Transforms the character to uppercase.
Parameters: arg is a pointer to a character to be transformed.
Functionality: Converts the character to uppercase using toupper.

Purpose: Displays the character.
Parameters: arg is a pointer to a character to be displayed.
Functionality: Prints the character using putchar.

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 = ' ';
```

FILE *fg: File pointer for the input file.
char next, this, last: Characters for processing.
pthread_t tGet, tUpd, tPrint: Thread identifiers for the three tasks.
void *retv: Variable to store the return value of the joined threads.
Opens the file specified in the command line argument.
Initializes the characters this, last, and 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;
}
```

Thread Creation:
pthread_create(&tGet, NULL, GET, &next);: Creates a thread to read the next character.
pthread_create(&tUpd, NULL, UPD, &this);: Creates a thread to transform the character.
pthread_create(&tPrint, NULL, PRINT, &last);: Creates a thread to print the character.
Thread Joining:
pthread_join(tGet, &retv);: Waits for the GET thread to finish.
pthread_join(tUpd, &retv);: Waits for the UPD thread to finish.
pthread_join(tPrint, &retv);: Waits for the PRINT thread to finish.
Character Management:
Updates last to this and this to next for the next iteration.

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;
}
```

Final Processing:
Ensures the last two characters are processed and printed.
Creates and joins threads for the final transformations and printing.

After all three threads have finished, the last variable is set to the value of this, and this is set to the value of next, preparing for the next iteration of the loop. This way, each character goes through a pipeline of three stages: it's read from the file, converted to uppercase, and then printed. The loop continues until next is EOF, which indicates the end of the file. After the loop, there are two more characters (this and last) that still need to be processed, so the UPD and PRINT functions are called two more times to process these characters.In the provided code, next, this, and last are variables used to hold characters read from the file. The next variable holds the next character to be read, this holds the current character being processed, and last holds the last character that was processed. The file is being read character by character, not line by line. The GET function reads a single character from the file and stores it in next. After a character is read, the UPD function converts this (the current character) to uppercase, and the PRINT function prints last (the last character). The main loop in the main function creates three threads for each character in the file: one to read the next character (tGet), one to convert the current character to uppercase (tUpd), and one to print the last character (tPrint). After creating the threads, it waits for them to finish with pthread_join.