Threads

Pthread library

Stefano Quer, Pietro Laface, and Stefano Scanzio
Dipartimento di Automatica e Informatica
Politecnico di Torino
skenz.it/os stefano.scanzio@polito.it

Thread libraries

- It provides the programmer the interface to use the threads
- The management can be done
  - At user-level (by functions)
  - At kernel-level (by system calls)
- The most used thread libraries are
  - POSIX threads
  - Windows 32/64
  - Java

Implemented at user and kernel level

Implemented at kernel-level

Implemented by means of a thread library of the system hosting Java (Pthread POSIX or Windows 32/64)

Pthreads

- POSIX threads or Pthreads
  - Is the standard UNIX library for threads
    - POSIX 1003.1c del 1995
    - Revised in IEEE POSIX 1003.1 2004 Edition
  - Defined for C language, but available in 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 = a set of independently executing functions 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 a **`pthread_*`** prefix
    - `pthread_equal, pthread_self, pthread_create, pthread_exit, pthread_join, pthread_cancel, pthread_detach`

Library linkage

- The Pthread system calls are defined in
  - pthreads.h
- It is necessary to remember
  - To insert in the `.c` files
    - `#include <pthread.h>`
  - Compile your program linking 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

pthread_equal() system call

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

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

pthread_self() system call

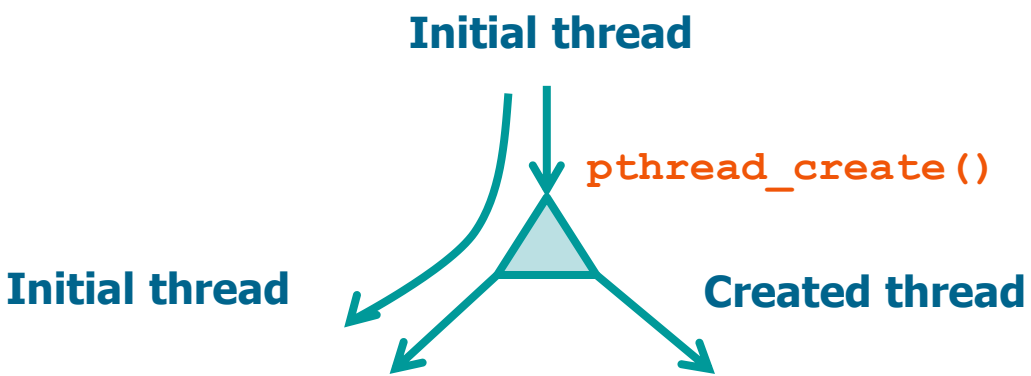
```
pthread_t pthread_self (
  void
);
```

- 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

pthread_create() system call

- At the beginning, a program consists of one process and one thread
- `pthread_create` allows creating a new thread
  - The maximum number of thread that can be created is undefined and implementation dependent



pthread_create() system call

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

- 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

A **single** argument

pthread_create() system call

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

- Returned values
  - 0 on success
  - Error code on failure

pthread_exit() system call

- 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`**

pthread_exit() system call

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

- 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

Terminates the process (all its threads)

Example

Creation of N threads with 1 argument

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

Collects the tids

```
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);
```

Address of `t` (pointer to integer)

Example

Creation of N threads with 1 argument

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

A thread can be executed when t is changed

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

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

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

```
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 argument

Cast of a vector of pointers
void * ↔ **int**

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

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

The pointer to a vector element

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`

pthread_join() system call

- At its creation a thread can be declared
  - Joinable
    - Another thread may "wait" (`pthread_join`) for its termination, and collect its exit status
  - Detached
    - No thread can explicitly wait for its termination (not joinable)

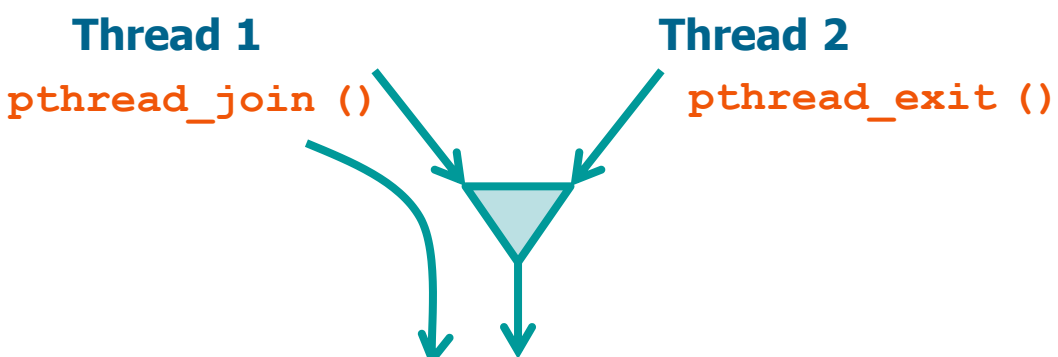
pthread_join() system call

- If a thread
  - is **joinable**, its termination status is retained until another thread performs a **`pthread_join`** for that thread
  - is **detached** its termination status is immediately released
- In any case
  - A thread calling **`pthread_join`** waits until the required thread calls **`phread_exit`**

pthread_join() system call

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

- Used by a thread to wait the termination of another thread



pthread_join() system call

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

valuePtr can be set to **NULL** if you are not interested in the return value

- Arguments
  - Identifier (tid) of the waited-for thread
  - The void pointer **`ValuePtr`** will obtain the value returned by thread **`tid`**
    - Returned by **`pthread_exit`**
    - Returned by **`return`**
    - **`PTHREAD_CANCELED`** if the thread was deleted

pthread_join() system call

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

- Returned values
  - 0 on success
  - Error code on failure
    - If the thread was detached **`pthread_join`** should fail
      - Depends on the OS and timing, it may also terminate correctly
    - If it fails, it returns the constant **`EINVAL`** or **`ESRCH`**

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`

Waits each thread, and collects its exit status

Example

- Use of a global variable common to many threads

```
int myglobal;

void *threadF (void *arg) {
  int *argc = (int *) arg;
  int i, j;
  for (i=0; i<20; i++) {
    j = myglobal;
    j = j + 1;
    printf ("t");
    if (*argc > 1) sleep (1);
    myglobal = j;
  }
  printf ("(T:myglobal=%d)", myglobal);
  return NULL;
}
```

The global variable is incremented by means of a copy on `j`

The thread can sleep or not

Example 2

```
int main (int argc, char *argv[]) {
  pthread_t mythread;
  int i;
  pthread_create (&mythread, NULL, threadF, &argc);
  for (i=0; i<20; i++) {
    myglobal = myglobal + 1;
    printf ("m");
    sleep (1);
  }
  pthread_join (mythread, NULL);
  printf ("(M:myglobal=%d)", myglobal);
  exit (0);
}
```

Example 2

If thread executes immediately
No loss of increments

```
> ./pgrm
mttttttttttttttttttt(T:myglobal=21)mmmmmmmmmmmmmmmmmmm
m(M:myglobal=40)
```

Thread and main thread alternates their execution every second. The increments of the thread are lost

```
> ./pgrm 1
mtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmtmt(T:myglobal=21)
M:myglobal=21)
```

2sec waiting for the main thread
Only some increments are lost

```
> ./pgrm 1
mttmttmttmttmttmttmttmttmttmttm(T:myglobal=21)mmmmmmmm
m(M:myglobal=30)
```

pthread_cancel() system call

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

- Terminates the target thread
  - The effect is similar to a call to **`pthread_exit(PTHREAD_CANCELED)`** performed by the target thread
- The thread calling **`pthread_cancel`** does not wait for termination of the target thread (it continues immediately after the calling)

pthread_cancel() system call

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

- Arguments
  - Target thread (tid) identifier
- Returned values
  - 0 on success
  - Error code on failure

pthread_detach() system call

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

The **attribute** of the **pthread_create** allows and alternative way to create a detached thread

- 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 with error code **EINVAL** or **ESRCH**

pthread_detach() system call

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

- Arguments
  - Thread (**tid**) identifier
- Returned values
  - 0 on success
  - Error code on failure

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 thread using the attribute of the pthread_create()

```
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);
}
```

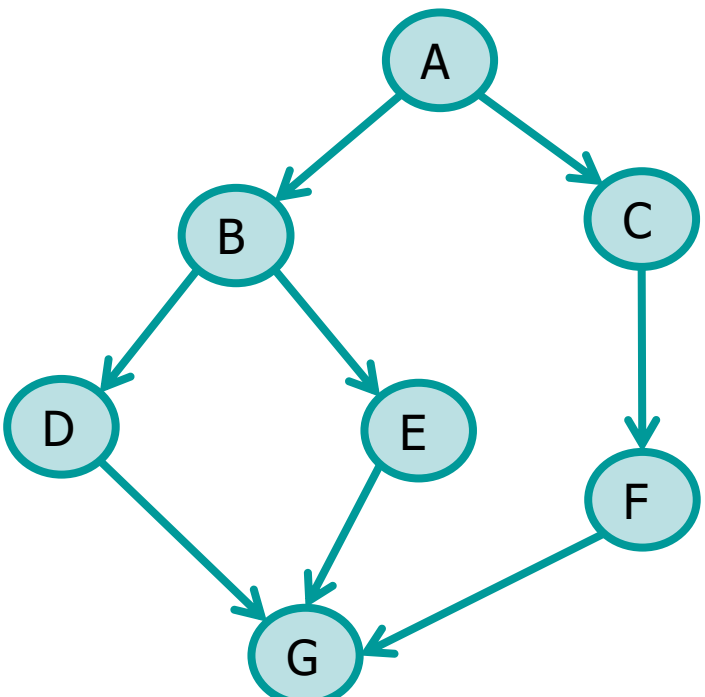
Creates a detached thread

Destroys the attribute object

Error if try to join

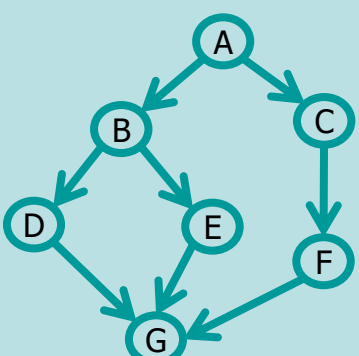
Exercise

- Implement, using threads, this precedence graph



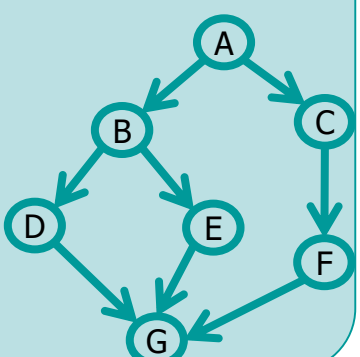
Solution

```
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");
```



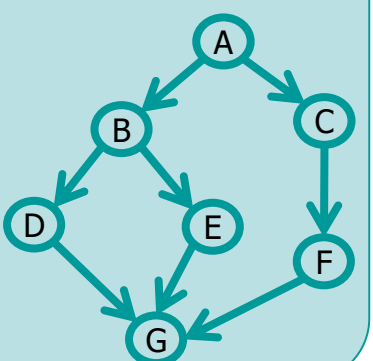
Solution

```
  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;
}
```



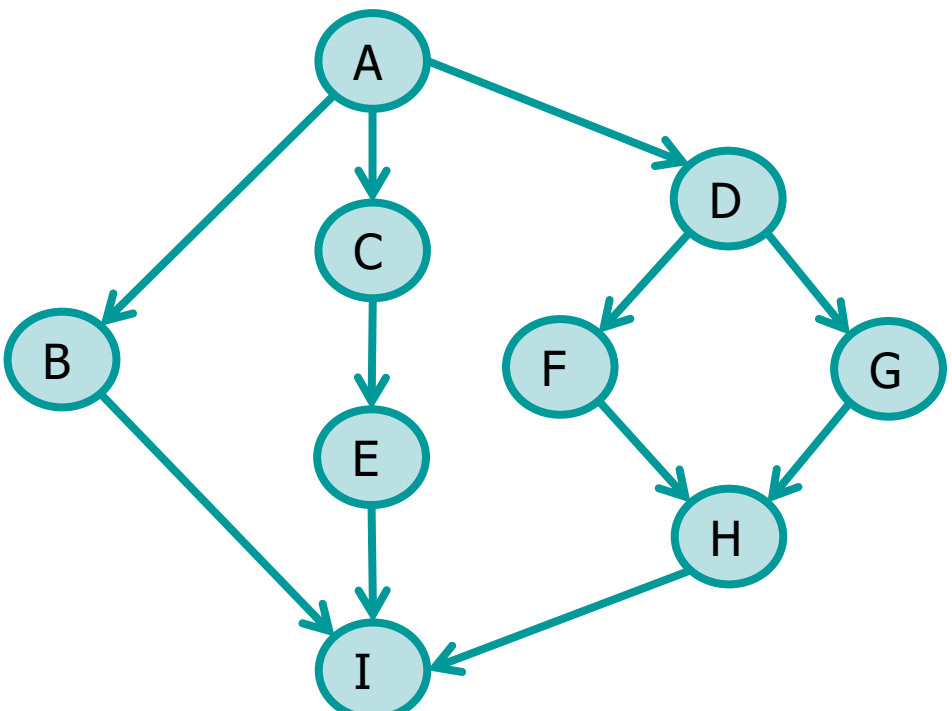
Solution

```
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

- Implement, using threads, this precedence graph

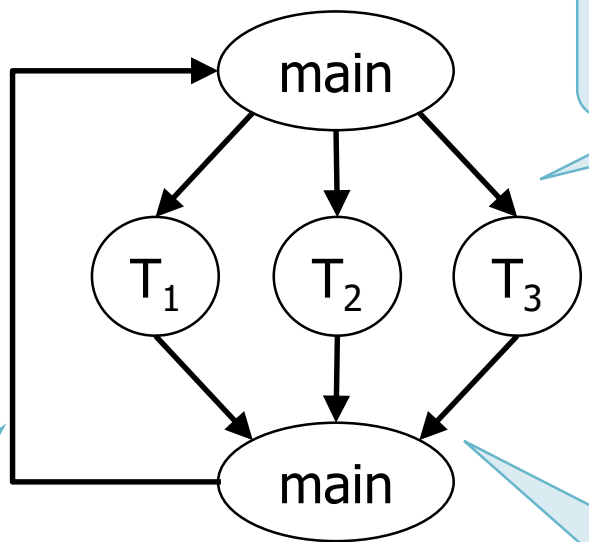


Exercise

- 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



Reading, transformation, and visualization in parallel

GET next
UPDATE this
PRINT last

T are created and destroyed at each iteration

For now, the only synchronization strategy is to use 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;
}
```

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;
}
```

Exercises

- Some other exercises about threads
  - https://www.skenz.it/cs/posix/threads