System and Device Programming

UNIX Thread

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

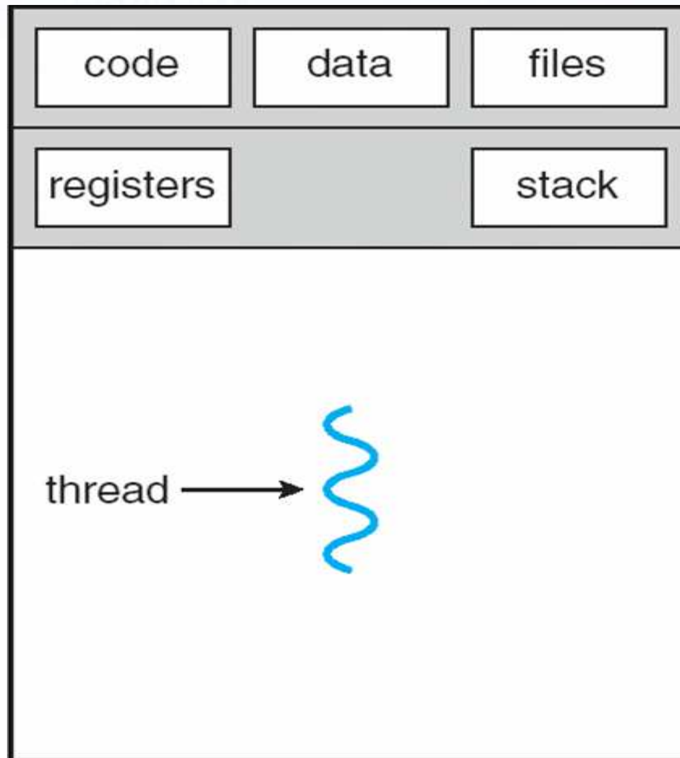
Threads

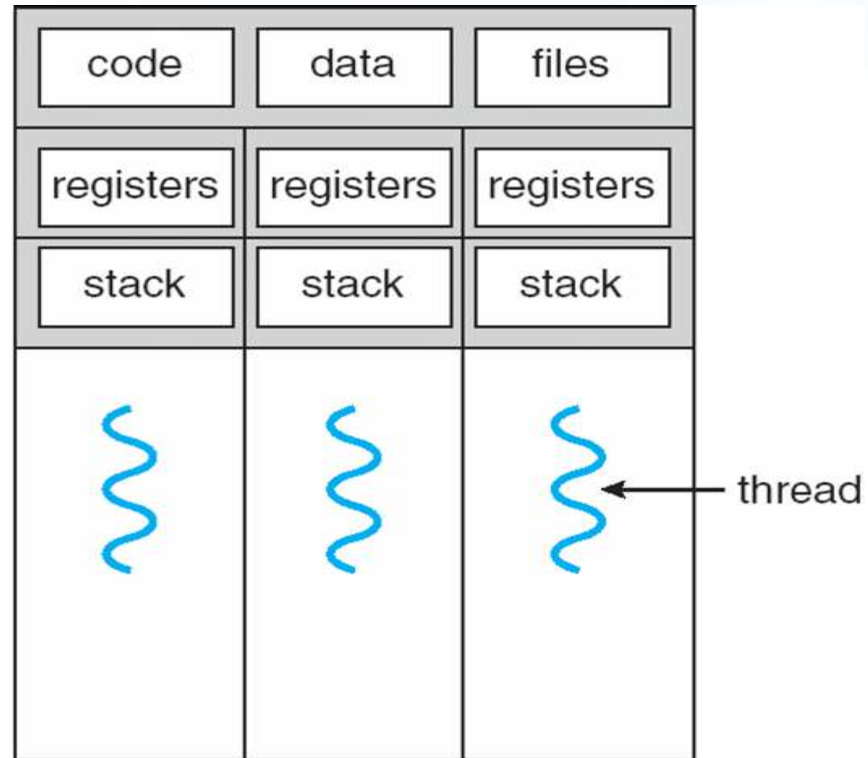
- Processes involve
  - High data transfer cost for cooperating processes
  - A significant increase in the memory used
  - Creation time overhead
  - Expensive context switching operations (with kernel intervention)
- There are several cases where it would be useful to have
  - Lower creation and management costs
  - A single address space
  - Multiple execution threads (concurrency) within that address space

Threads

- In 1996 the 1003.1c POSIX standard introduces the concept of **threads**
  - The thread model allows a program to control multiple different flows of operations (**scheduled and executed independently**) that overlap in time
  - Threads
    - Share the code section, the data section (variables, file descriptors, etc.), and operating system resources (e.g., signals)
    - Have private program counter, hardware registers, stack (i.e., local variables and execution history)

Threads





A process with a single thread

A process with three threads
Sharing requires protection !

Pros and Cons

- The use of threads allows
  - Advantages
    - Shorter response time
    - Shared resources
    - Lower costs for resource management
    - Increased scalability
  - Disadvantages
    - There is no protection for threads
      - If the threads are not synchronized, access to shared data is not thread safe
    - There is not a parent-child hierarchical relationship between thread

Multithread programming models

- There are three thread models
  - Kernel-level thread
    - Thread implemented at kernel-level
    - The kernel directly supports the thread concept
  - User-level thread
    - Thread implemented at user-level
    - The kernel is not aware that threads exist
  - Mixed or hybrid solution
    - The operating system provides both user-level and kernel threads

The kernel effort to manage threads reduced

The choice is moderately controversial

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 (Pthreads)
  - C
  - Windows 32/64
  - Java

Implemented at user and kernel level

Implemented at kernel-level

Implemented through the thread library of the system hosting Java

Pthreads

- POSIX threads or Pthreads
  - Is the standard UNIX library for threads (1003.1c born in 1996, revised in 2004)
  - Defined for C language, but available in other languages (e.g., FORTRAN)
  - A thread is a **function** that is executed in concurrency with the main 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
  - 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

System call pthread_equal

```
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

System call pthread_create

```
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

System call pthread_create

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

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

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
);
```

- 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

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

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

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

```
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

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);
}
```

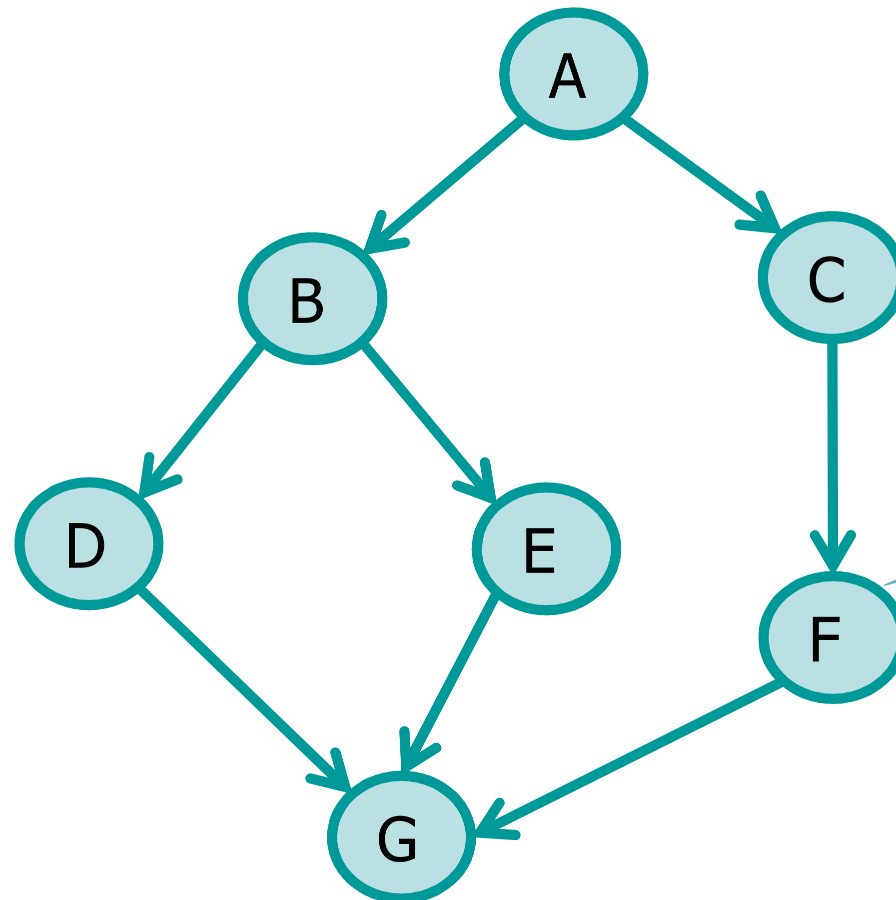
Creates a detached thread

Destroys the attribute object

Error if try to join

Exercise

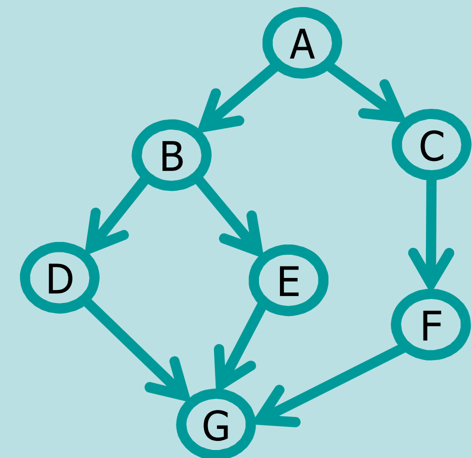
- Implement, using threads, the following precedence graph using threads



Each circle represents an instruction or a set of instructions

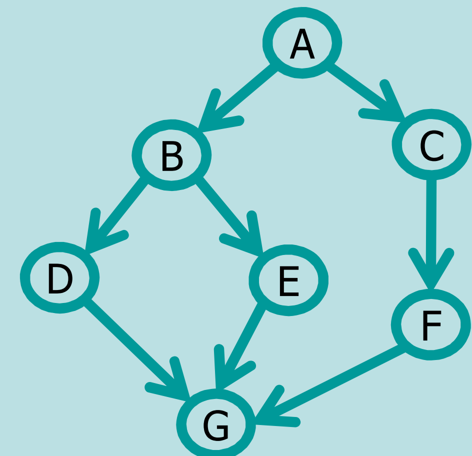
Solution

```
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
}
```