OPERATING SYSTEMS: PROCESSES

Threads and Processes

Content

- **Thread Concept.**
- Thread models.
- Design aspects.
- Threads in PThreads.
- Thread scheduling

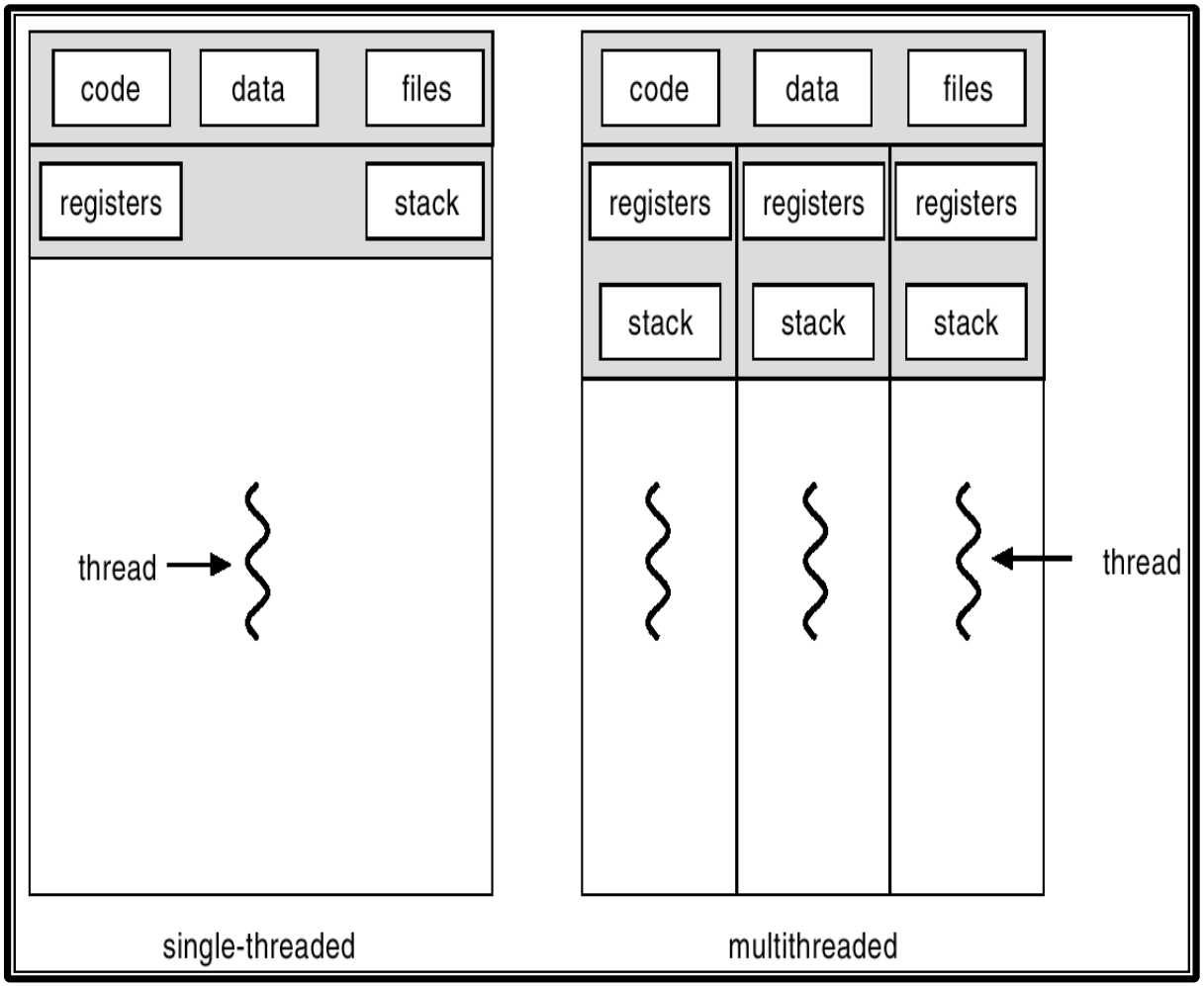
Applications with concurrent processes

- A process includes a single thread of execution.
- Design of applications with multiple concurrent tasks:
  - A receiver process gets requests and launches a process per request.
  - A receiver process and a fixed set of request handling processes.

Performance of concurrent processes

- Time overhead in process creation and destruction.
- Time overhead due to context switching.
- Problems with sharing resources.

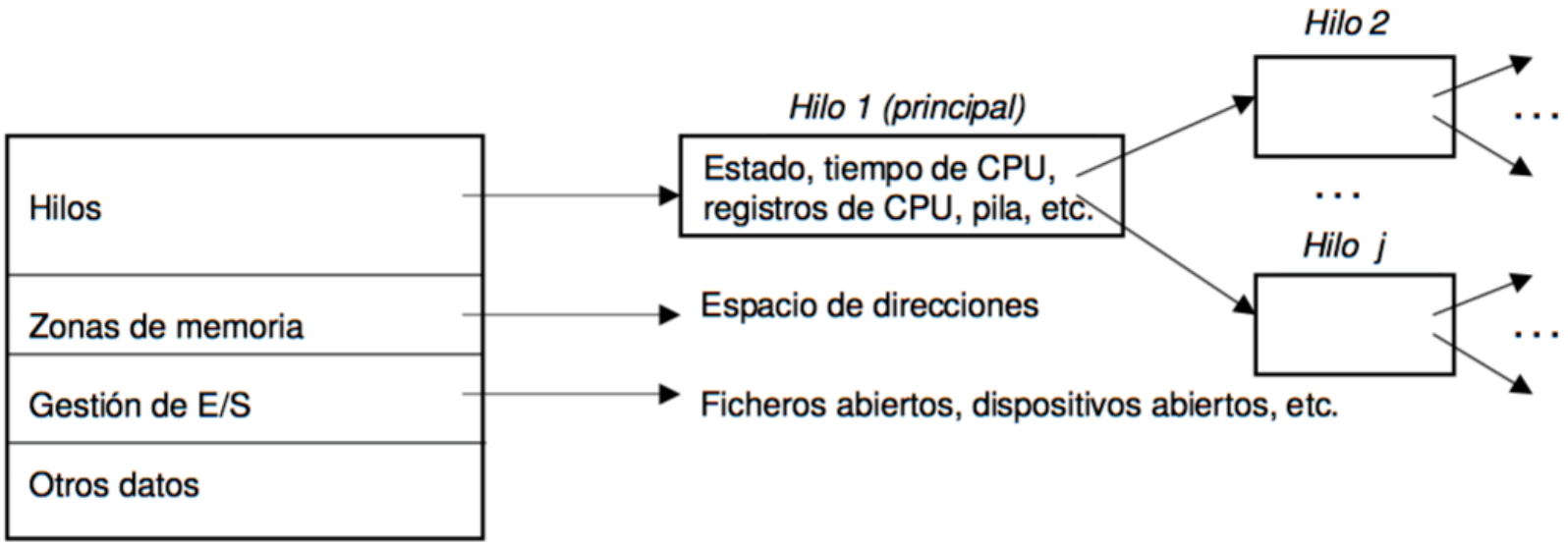
Threads


code
data
files
registers
stack
thread
code
data
files
registers
registers
registers
stack
stack
stack
thread
single-threaded
multithreaded

Threads

- Most modern OS provide processes with multiple instruction sequences or threads of control inside them.
- Used as the basic unit of CPU usage.
- Each one consists of:
  - Thread Id.
  - Program counter.
  - Registers set.
  - Stack.
- The share with the rest of threads in the process:
  - Memory map (code region, data region, shmem).
  - Open files.
  - Signals, semaphores, and timers.

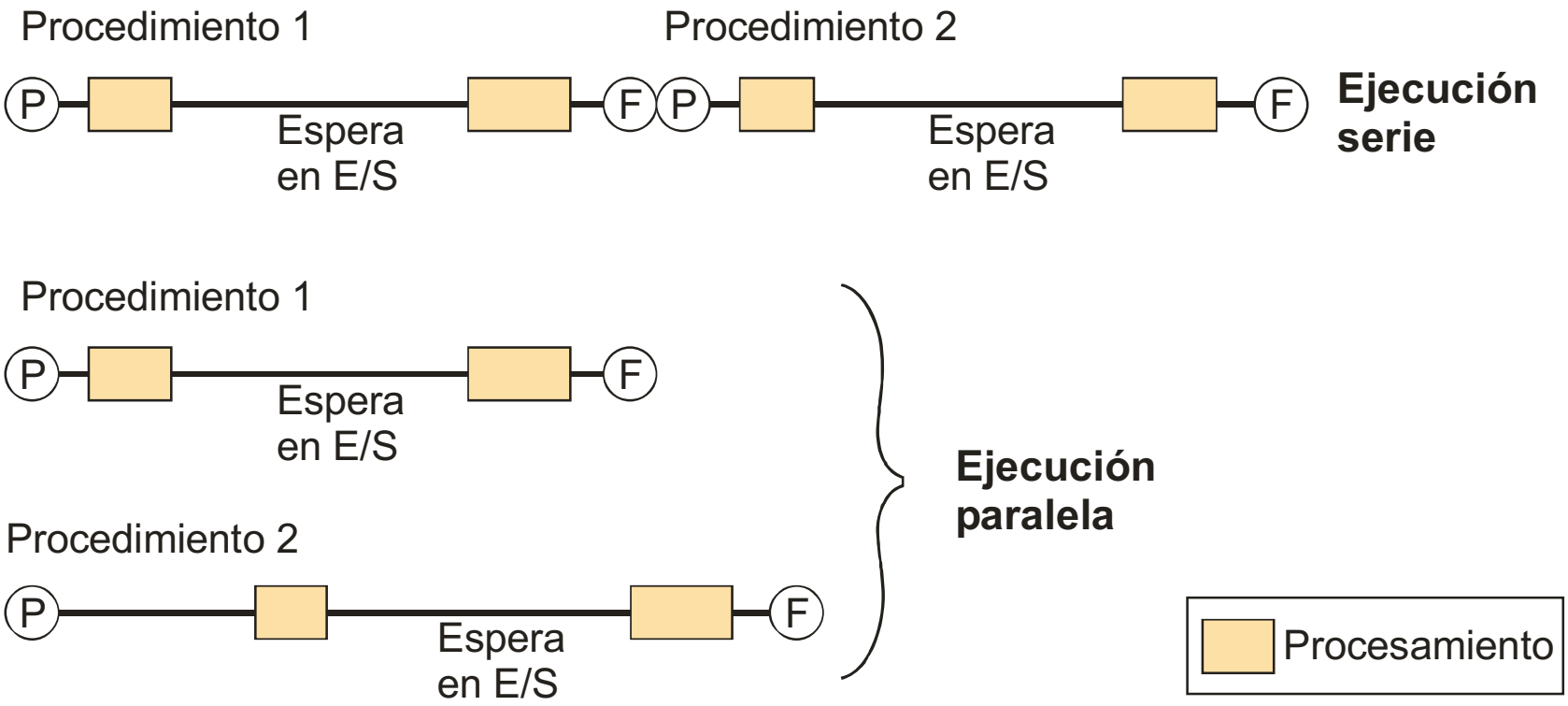
BCP with threads



Benefits

- Response capacity.
  - Higher interactivity as user interactions are separated from processing tasks in different threads.
- Resource sharing.
  - Threads share most of the resources automatically.
- Resource economy.
  - Creating a process has a much higher overhead than creating a thread (e.g.: in Solaris the ratio is 30 to 1).
- Use in multiprocessor architectures.
  - Higher concurrency degree allocating different threads to different processors.
  - Most modern operating systems use thread as unit of scheduling.

Threads allow parellization of applications



Thread support

User space

ULT – User Level Threads

- Implemented in form of functions library in user space.
- Kernel has no knowledge about them.
  - No support in any form.
- Much faster, but some problems arise.
  - Blocking system calls.

Kernel space

KLT – Kernel Level Threads

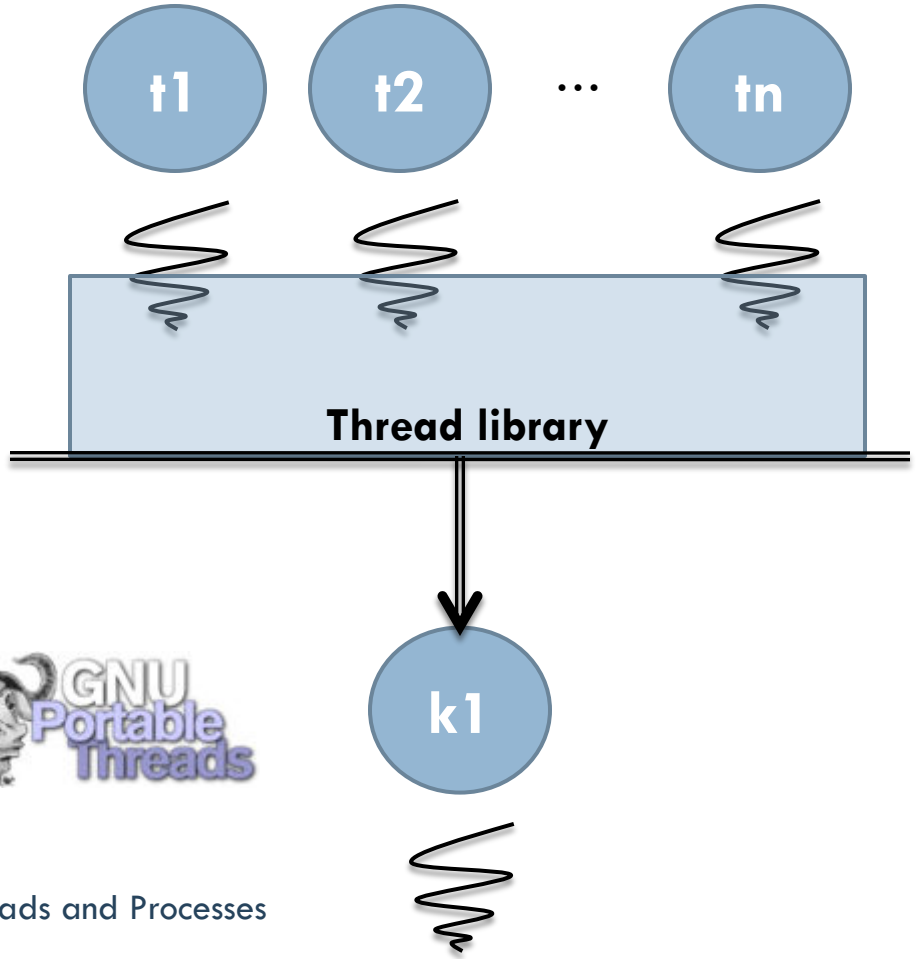
- Kernel in charge of creating, scheduling and destroying them.
- Slightly slower as kernel needs to participate implying execution mode switching.
- In blocking system calls, only the involved thread is blocked.
- In SMP, multiple threads can run at a time.
- No support thread code in applications.
- Kernel may also use threads to perform its own tasks.

Content

- Thread Concept.
- **Thread models.**
- Design aspects.
- Threads in PThreads.
- Thread scheduling

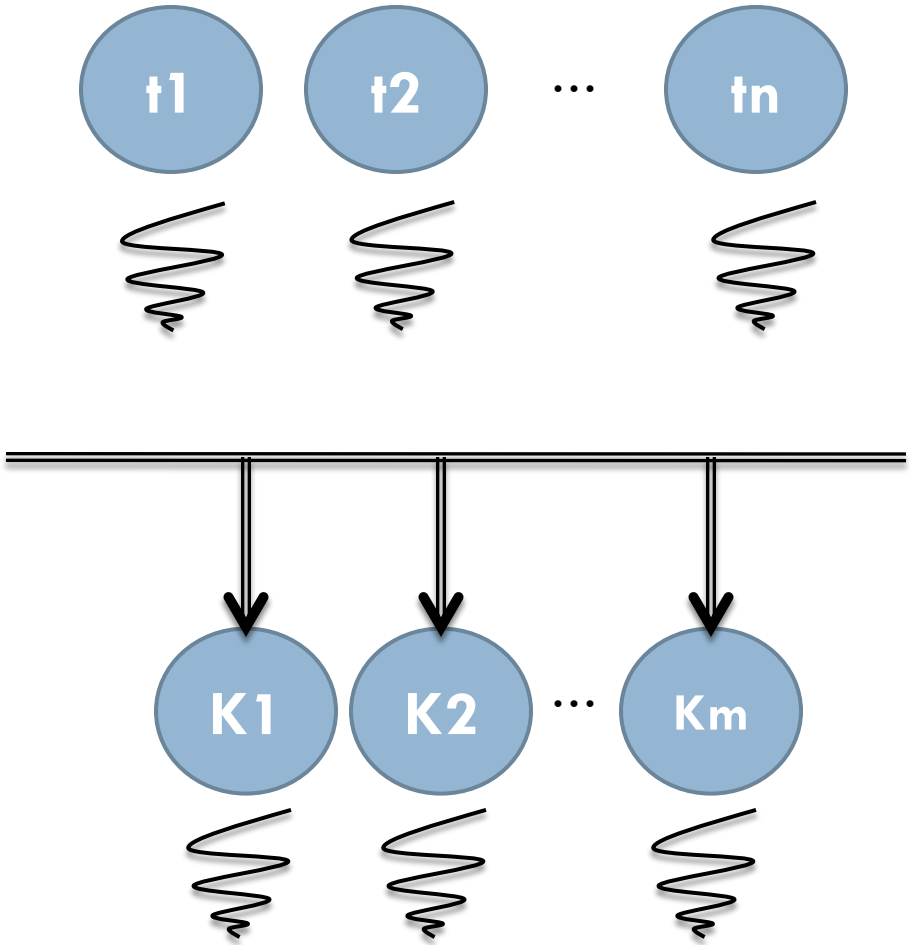
Multiple threads model: Many to one

- Maps multiple user threads to a single kernel thread.
- Thread library in user space.
- Blocking call:
  - All threads are blocked.
- In multiprocessors, multiple threads cannot run at same time.



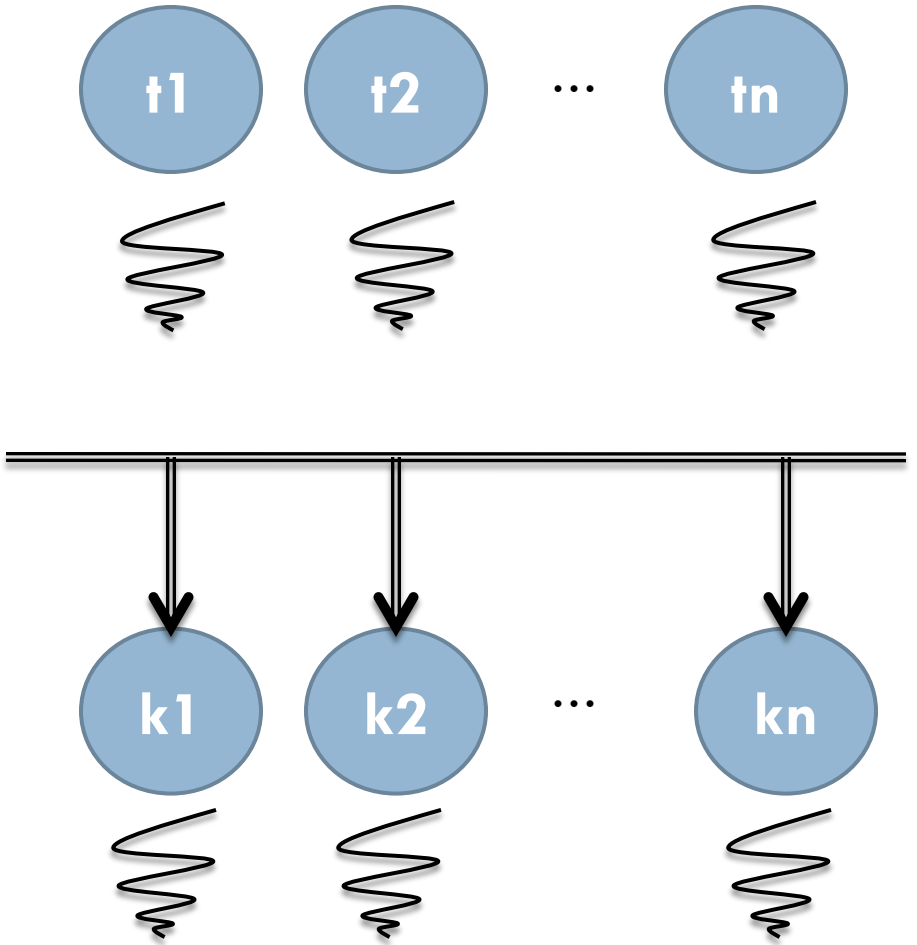
Multiple threads model: Many to many

- This model multiplexes user threads into a fixed number of kernel threads.
- Operating System threads gets much more complex.
- Examples:
  - Solaris (before version 9).
  - HP-UX.
  - IRIX.



Multiple threads model: One to one

- Maps a kernel thread to every user thread.
- Most implementations restrict the number of threads that can be created.
- Examples:
  - Linux 2.6.
  - Windows.
  - Solaris 9.



Content

- Thread Concept.
- Thread models.
- **Design aspects.**
- Threads in PThreads.
- Thread scheduling

Fork and exec calls

- In UNIX type systems. What to do if fork is called from a thread?
  - Duplicate process with all the threads.
    - Appropriate if exec is not going to be called immediately after to change process image.
  - Duplicate process with only the calling thread.
    - More efficient if exec is going to be called and all threads would be cancelled in any case.
- Linux solution: Two versions of fork.

Threads cancellation

- Situation when a thread notifies to others that the must terminate.
- Options:
  - Asynchronous cancellation: Forces immediate termination of thread.
    - Problems with resources allocated to thread.
  - Deferred cancellation: Thread checks periodically whether it should terminate.
    - Better approach.

Threads and request processing

- Applications receiving requests an processing them may make use of threads for handling.

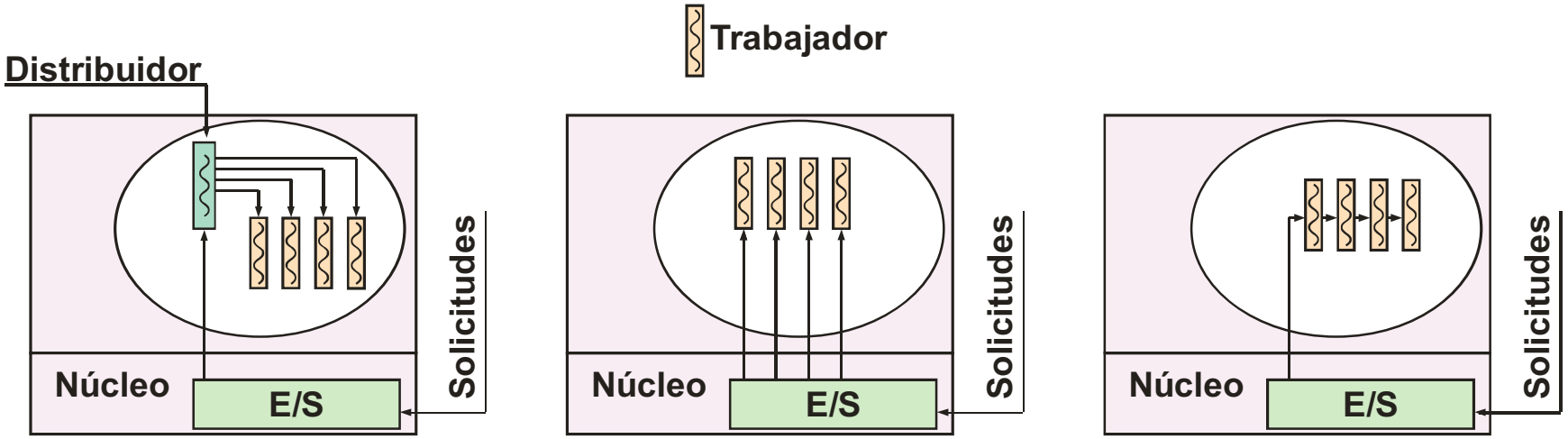
- But:
  - Thread creation/destruction time is a delay (although lower than process creation/destruction).
  - A limit in the number of concurrent threads is not established.
  - If a request avalanche comes resources may be exhausted.

Thread Pools

- A *Thread Pool* is created and threads wait until requests arrive.

- Advantages:
  - Delay minimization: Thread already exists.
  - There is a established limit over the number of concurrent threads.

Software architectures based on threads



Content

- Thread Concept.
- Thread models.
- Design aspects.
- **Threads in PThreads.**
- Thread scheduling

Thread creation

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*func)(void *), void *arg)`
  - Creates a thread and starts its execution.
  - **thread**: Must pass the addres of a variable of type **pthread_t** used as handle.
  - **attr**: Must pass the address of a structures with attributes. NULL may be passed to specify default attributes.
  - **func**: Function with thread execution code.
  - **arg**: Pointer to thread parameter. Only one parameter can be passed.
- `pthread_t pthread_self(void)`
  - Returns thread identifier for the calling thread.

Waiting and termination

- `int pthread_join(pthread_t thread, void **value)`
  - Invoking thread waits until the thread identified by the handle has terminated.
  - **thread**: Handle to thread that must be waited to terminate.
  - **value**: Thread termination value

- `int pthread_exit(void *value)`
  - Allows a thread to terminate its execution, stating its termination status.
  - Termination status cannot be a pointer to a local variable.

Example: sum with threads


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

struct addparam{
  int n, m, r;
};
typedef struct addparam
   addparam_t;

void add(addparam_t * par) {
  int i;
  int sum=0;
  for (i=par->n;i<=par->m;i++)
   {
    sum+=i;
  }
  par->r=sum;
}
```

```
int main() {
  pthread_t th1, th2;
  sumapar_t s1 = {1,50,0};
  sumapar_t s2 = {51,100,0};

  pthread_create(&th1, NULL,
   (void*)add, (void*)&s1);
  pthread_create(&th2, NULL,
   (void*)add, (void*)&s2);

  pthread_join(th1, NULL);
  pthread_join(th2, NULL);

  printf("Total sum=%d\n",
    s1.r+s2.r);
}
```

Thread attribute

- Each thread has a set of attributes associated to it.
- Attributes represented by a variable of **pthread_attr_t** type.

- Attributes control:
  - Whether a thread is detached or joinable.
  - Size of the thread private stack.
  - Location of the thread stack.
  - Thread scheduling policy.

Attributes

- `int pthread_attr_init(pthread_attr_t * attr);`
  - Initializes a thread attribute structure.
- `int pthread_attr_destroy(pthread_attr_t * attr);`
  - Destroys a thread attribute structure.
- `int pthread_attr_setstacksize(pthread_attr_t * attr, int stacksize);`
  - Defines the stack size for a thread.
- `int pthread_attr_getstacksize(pthread_attr_t * attr, int *stacksize);`
  - Allows to obtain the size of the thread stack.

Hilos dependientes e hilos independientes

- `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)`
  - Sets the termination state for a thread.
  - If **detachstate == PTHREAD_CREATE_DETACHED**
    - Thread releases its resources upon thread termination.
  - if **detachstate = PTHREAD_CREATE_JOINABLE**
    - Resources are not released automatically.
    - A call to **pthread_join()** is needed.

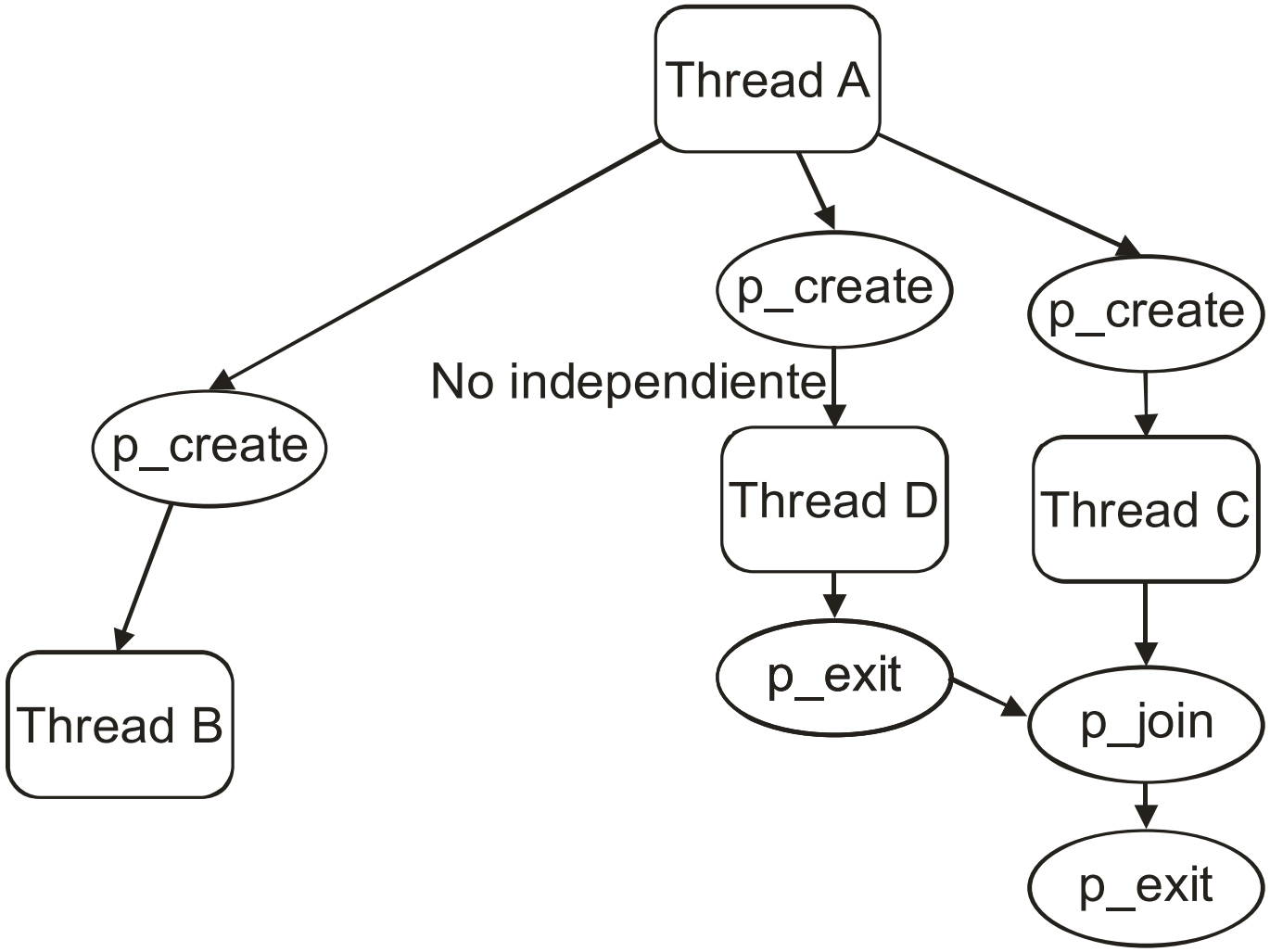
- `int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate)`
  - Allows to get the termination status.

Example: Detached threads


```
#include <stdio.h>
#include <pthread.h>
#define MAX_THREADS 10
void func(void)  {
  printf("Thread %d \n", pthread_self());
  pthread_exit(0);
}

int main() {
  int j;
  pthread_attr_t attr;
  pthread_t thid[MAX_THREADS];
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
  for(j = 0; j < MAX_THREADS; j ++)
    pthread_create(&thid[j], &attr,  func, NULL);
  sleep(5);
}
```

Example of threads hierarchy

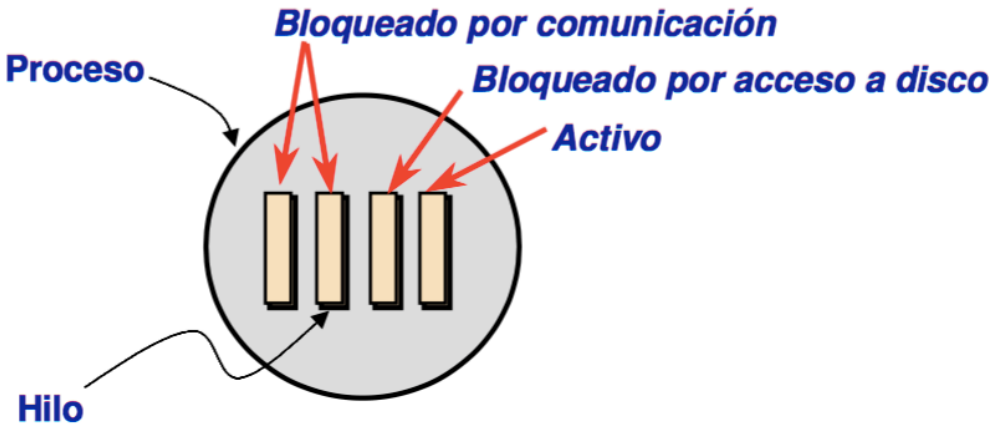


Content

- Thread Concept.
- Thread models.
- Design aspects.
- Threads in PThreads.
- **Thread scheduling**

State of a process with threads

- Combination of the states of its threads:
  - If there is a thread executing ->Process executing
  - If no thread executing, but ready -> Process ready
  - If no threads blocked ->Process blocked



Thread scheduling into a process

- Based on priority model.
  - Segments of time not used.

- A thread will keep on running on the CPU until it gos to stopped or blocked state.
  - If you want to alternate threads in a process, it must be ensured and forced
  - For example using *sleep()*

Thread scheduling

- API to especify scheduling policy (PCS o SCS) on thread creation
  - PTHREAD_SCOPE_PROCESS: PCS scheduling
  - PTHREAD_SCOPE_SYSTEM: SCS scheduling
- SO can limit policy and numbers:
  - LINUX and MacOS only allows PCS for users (no kernel threads)

Thread scheduling API


```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Thread scheduling API


```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Summary

- A process may have several threads of execution.
- A multi-threaded application consumes less resources than a multi-process application.
- Each system has a thread support mode:
  - ULT versus KLT.
- PThreads is a user library for threading.
- Win32 offers kernel threads with support for Thread Pools.