ARCOS Group
Universidad Carlos III de Madrid

Lesson 3
Process and threads

Operating Systems
Computer Science and Engineering

To remember…

Before classes	Class	After class

Prepare the prerequisites.

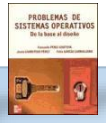
Study the material associated with the **bibliography**: slides alone are not enough.
Please ask questions (especially after study).

Exercising skills:

- Perform all **exercises**.
- Carrying out the **practice notebooks** and **the practical exercises** progressively.

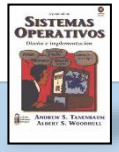
Recommended reading

Base



1. Carretero 2020:
   1. Cap. 5
2. Carretero 2007:
   1. Cap. 3.6 and 3.7
      Cap. 3.9 and 3.13

Suggested



1. Tanenbaum 2006:
   1. (es) Cap. 2.2
   2. (en) Cap.2.1.7
2. Stallings 2005:
   1. 4.1, 4.4, 4.5 and 4.6
3. Silberschatz 2006:
   1. 4

¡ATENCIÓN!

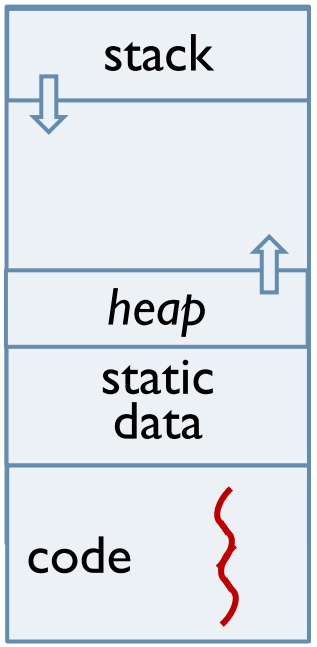
- This material is a script for the class but it is not the notes of the full course.
- The books given in the bibliography together with what is explained in class represent the study material for the course syllabus.

Contents

Contents

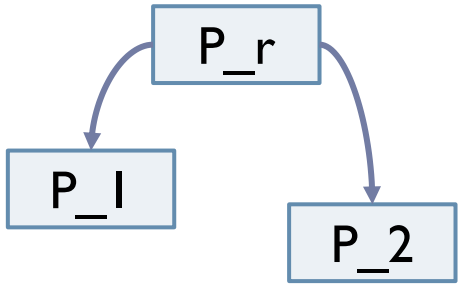
Where to start from: process…

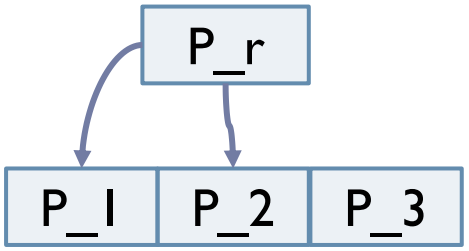
- Definition:
  - Program in execution.
  - Processing unit managed by the operating system.
- Model:
  - Groups resources used:
    - Files, signals, memory, …
  - Multiprocessing, multitasking and multiprocessing:
    - Each process has a single execution thread.
    - Process hierarchy with inter-process protection/sharing.



Where to go: Concurrent applications

- Client/server-based design:
  - On demand:
    - A process awaits the arrival of requests (request receiver)
    - When a request arrives, it creates a process to handle the request.
  - Thread pool:
    - One receiver process and N blocked request processing processes.
    - When a request arrives, a process is unblocked, attends the request and blocks again.

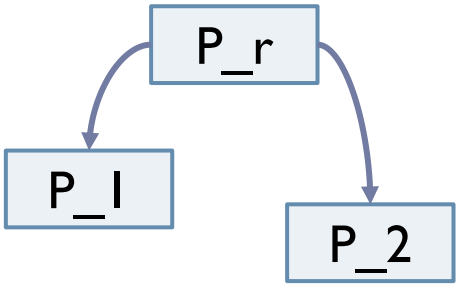




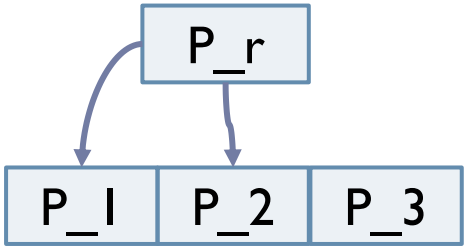
Where to go: Concurrent applications

- Time spent in creating and terminating processes
- Time spent on context switches
- Resource sharing is not easy (and can be problematic)

- On demand:
  - A process awaits the arrival of requests (request receiver)
  - When a request arrives, it **creates** a process to handle **the request**.



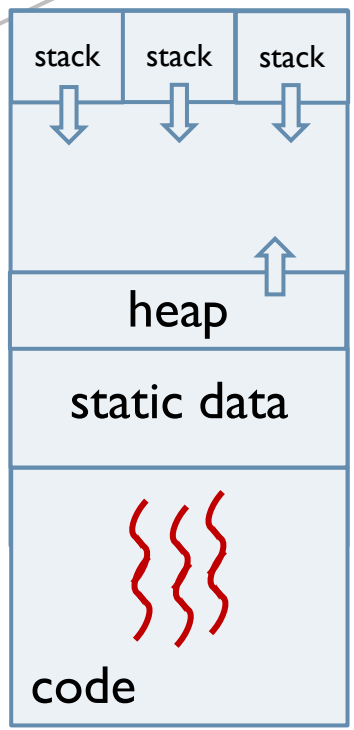
- Thread pool:
  - One receiver process and N blocked request processing processes.
  - When a request arrives, a process is **unblocked**, attends the **request** and **blocks again**.



Thread or light process

- Time spent in creating and terminating processes
- Time spent on context switches
- Resource sharing is not easy (and can be problematic)

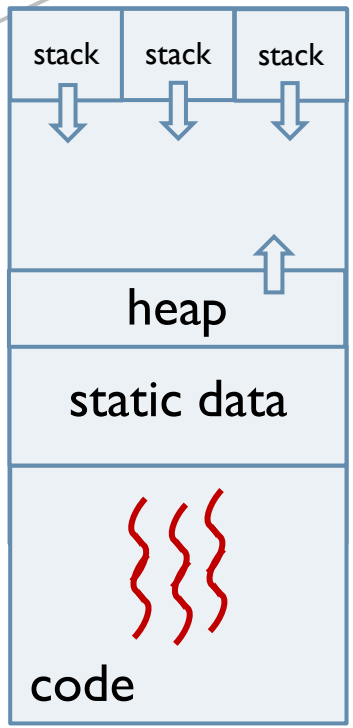
- Groups resources used:
  - Files, signals, memory, …
- Multiprocessing, multitasking and multiprocessing:
  - **Each process can have several "threads" of execution**. The thread is a basic unit of CPU "utilization" ("schedulable" unit).
  - **Threads of the same process share all** process resources **except stack and context** (PC, SR, SP, ...).
  - Process hierarchy with inter-process protection/sharing.



Thread or light process

- Time consumption in the creation and termination of processes
- Time consuming context switches
- Resource sharing is not easy (and can be problematic).

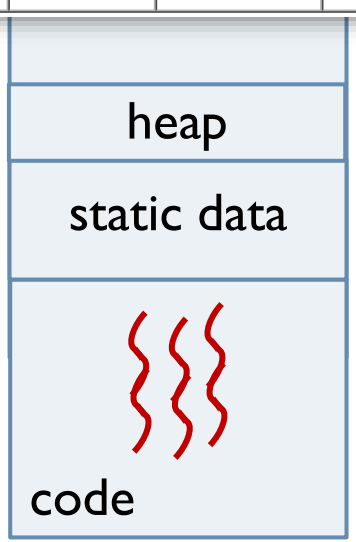
- Benefits:
  - Resource sharing.
    - It facilitates communication with shared memory between threads of the same process.
  - Economy of resources.
    - Creating a thread takes less time. (E.g. Solaris -> 30x)
      Simpler context switching and leverages virtual memory translation tables, cache, etc.
  - Response capacity.
    - Separating the processing from the threaded user interactions offers greater interactivity.
  - Better utilization of multiprocessor and multicore architectures.



Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

threads of the same process.

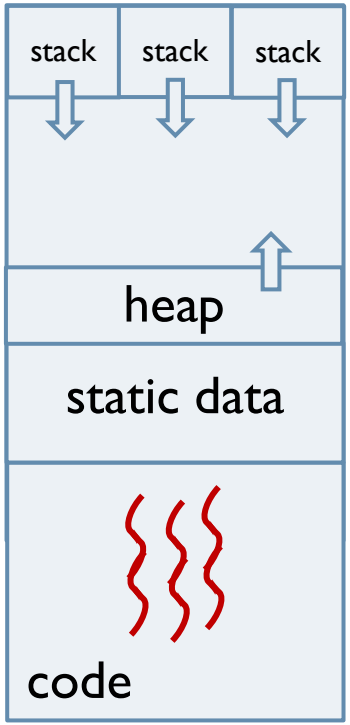
- Economy of resources.
  - **Creating a thread takes less time.**
    Simpler context switching and leverages virtual memory translation tables, cache, etc.
- Response capacity.
  - Separating the processing from the threaded user interactions offers greater interactivity.
- Better utilization of multiprocessor and multicore architectures.



https://computing.llnl.gov/tutorials/pthreads/

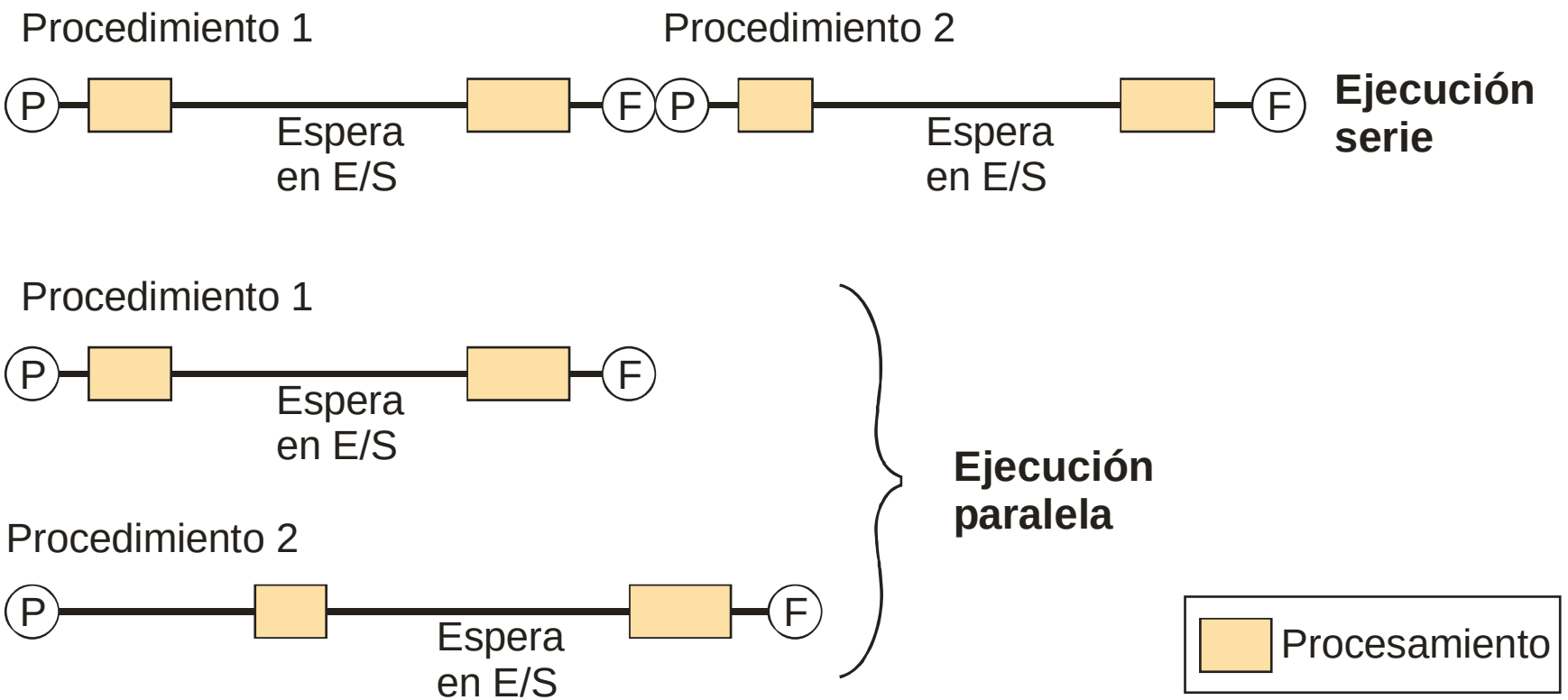
Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

- Resource sharing.
  - **It facilitates communication with shared memory between threads of the same process.**
- Economy of resources.
  - Creating a thread takes less time.
    Simpler context switching and leverages virtual memory translation tables, cache, etc.
- Response capacity.
  - Separating the processing from the threaded user interactions offers greater interactivity.
- Better utilization of multiprocessor and multicore architectures.

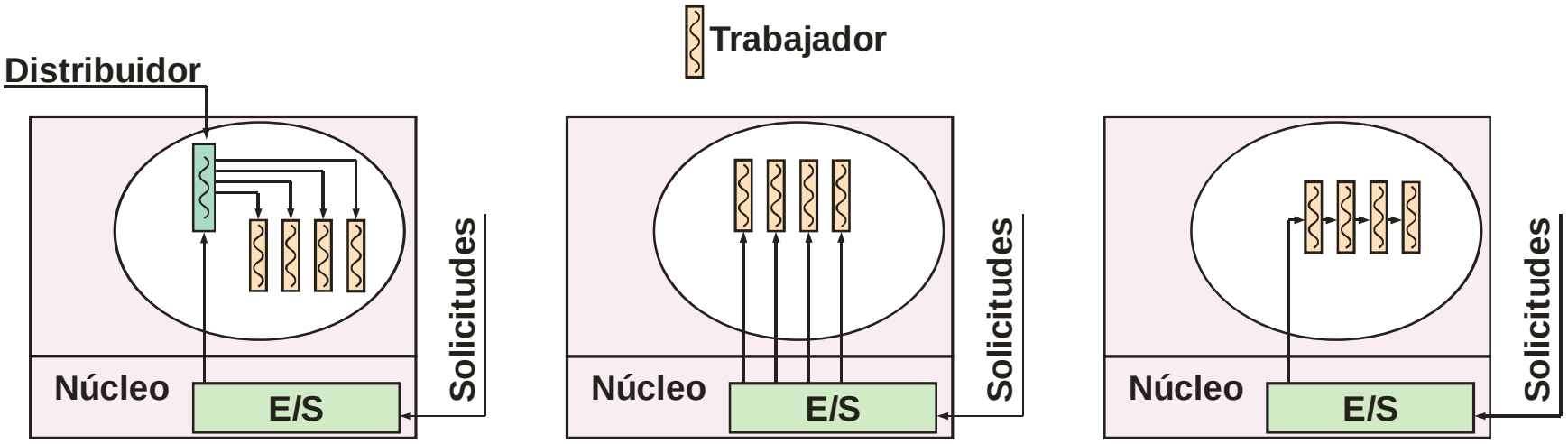


https://computing.llnl.gov/tutorials/pthreads/

Threads make it possible to parallelize the execution of an application



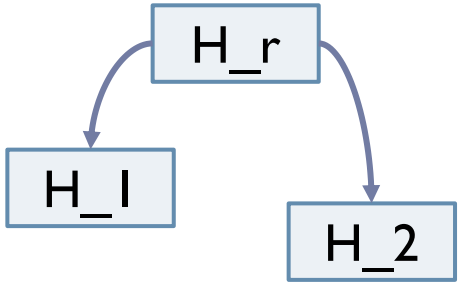
Thread-based software architectures
(1/3) using threads

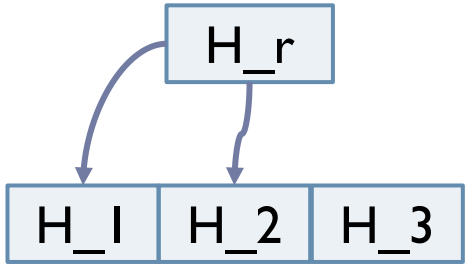


Sistemas operativos: una visión aplicada (© J. Carrete et al.)

Threads and application processing
(2/3) creating threads

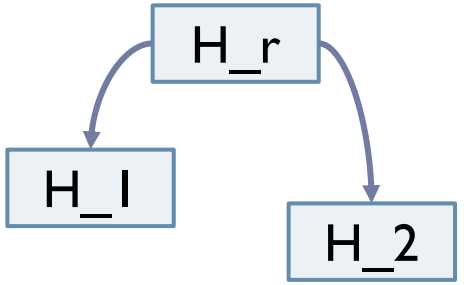
- Client/server-based design:
  - On demand:
    - A thread waits for the arrival of requests (request receiver) and when a request arrives, a thread is created to attend the request.
    - Disadvantages:
      - Creation time involves a delay.
      - If an avalanche of requests arrive -> DoS
  - Thread pool:
    - A set of threads is created waiting for requests to arrive.
    - Advantages:
      - Minimize the attention delay (thread exists).
      - A limit of # of concurrent threads is kept.

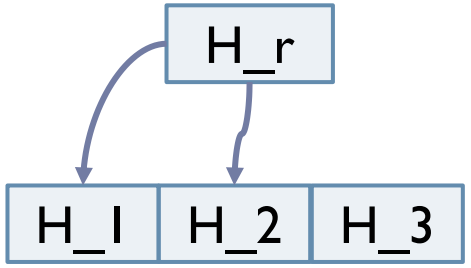




Thread cancellation/termination
(3/3) ending threads

- Client/server-based design:
  - On demand:
    - Coordinating thread does not create more threads and waits for the completion of the existing ones.
  - Thread pool:
    - One thread notifies the rest that they should terminate.
    - Options:
      - Asynchronous cancellation: the thread is forced to termination.
        - Problem: release of resources in use.
      - Deferred cancellation: the thread is notified that a "special" request for termination of execution arrives.
        - Preferable although it may take some time.



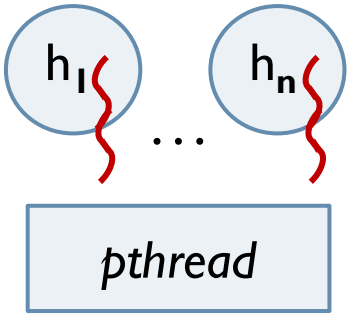


Contents

1. Introduction
   - Process definition.
   - Model offered: resources, multiprogramming, multitasking and multiprocessing
2. Process life cycle: process status.
3. Services to manage processes provided by the operating system.
4. Definition of thread
5. **Kernel and library threads**
   - **Thread model.**
6. Services for threads in the operating system.

Library vs. kernel threads

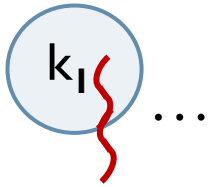
- ULT (*User Level Threads*)
  - **Implementation of thread services: as a user space library**.
    - The kernel is NOT aware of the existence of threads.
  - Advantage: faster by not having to make changes from user to kernel mode (and vice versa)
  - Disadvantage: if a thread makes a blocking call then it blocks the whole process.



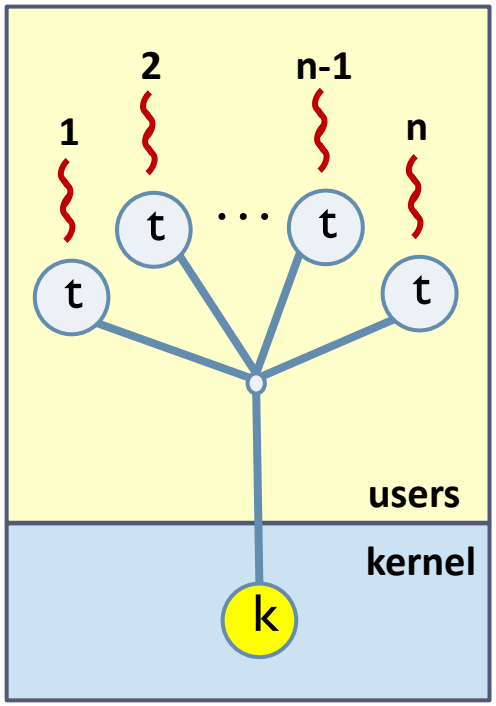
users

kernel

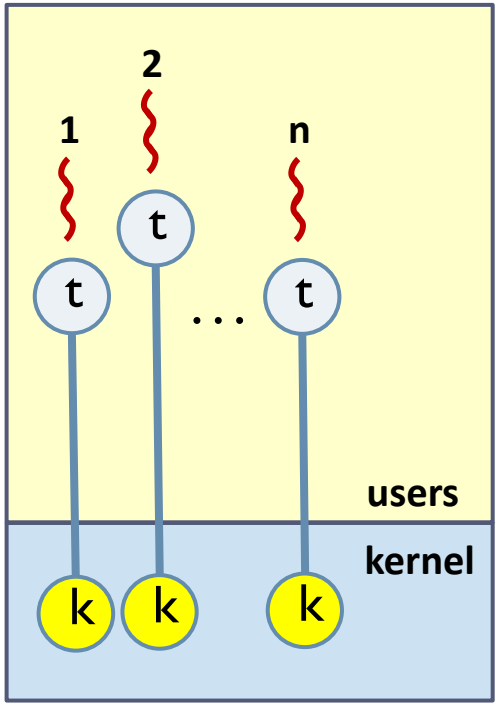
- KLT (*Kernel Level Threads*)
  - **Implementing thread services: as a kernel service**.
    - The kernel has thread support (for itself and users)
  - Disadvantage: slower by having to make changes from user mode to kernel (and vice versa)
  - Advantage: if a thread makes a blocking call then it only blocks that thread.



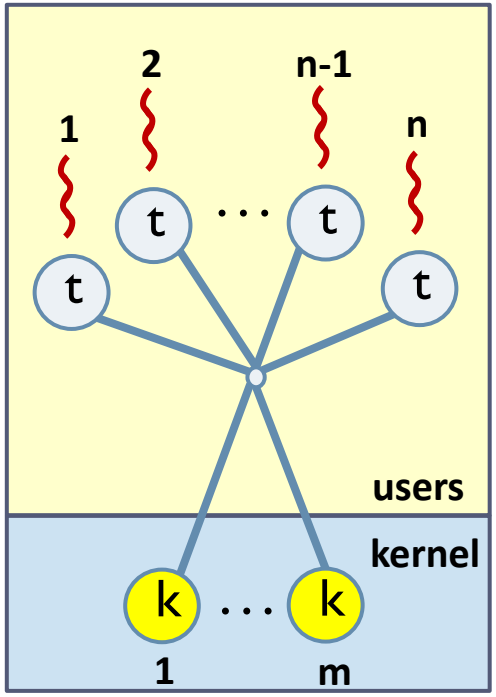
Multi-threaded model



Many-to-one



One-to-one

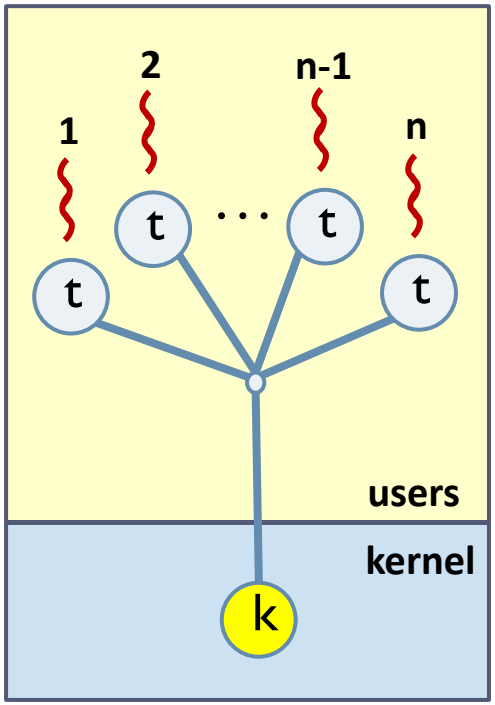


Many-to-many

http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Multi-thread model

- **Many-to-one**
  - Multiple user threads correspond to a single kernel thread.
  - E.g.: library of user threads.
  - A/D:
    - Blocking call blocks all threads.
    - Multiple threads cannot run at the same time in SMP.



- One-to-one
  - Each user thread corresponds to a kernel thread.
  - E.g.: Linux 2.6, Windows, Solaris 9
  - A/D:
    - Blocking call does NOT block all threads..
    - In SMP, several threads can be executed at the same time.
- Many-to-many
  - User threads are multiplexed into a number of threads in the kernel.
  - E.g.: IRIX, HP-UX, Solaris (prior to 9)
  - A/D:
    - Blocking call does NOT block all threads..
    - In SMP, several threads can be executed at the same time.

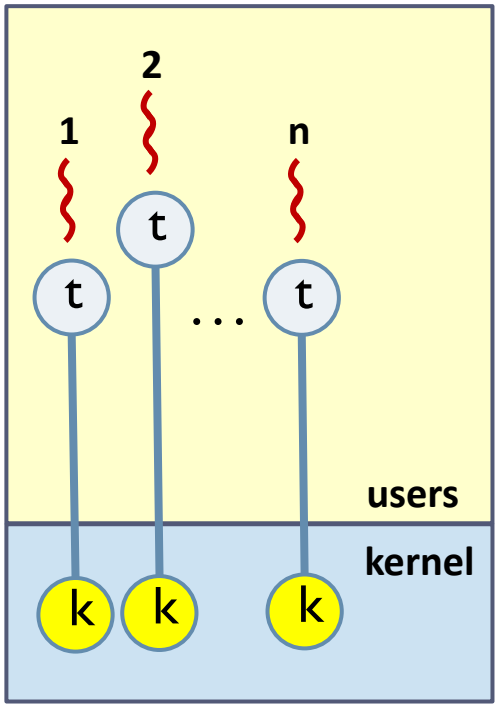
http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Multi-thread model

- Many-to-one
  - Multiple user threads correspond to a single kernel thread.
  - E.g.: library of user threads.
  - A/D:
    - Blocking call blocks all threads.
    - Multiple threads cannot run at the same time in SMP.

One-to-one

- Each user thread corresponds to a kernel thread.
- E.g.: Linux 2.6, Windows, Solaris 9
- A/D:
  - Blocking call does NOT block all threads..
  - In SMP, several threads can be executed at the same time.



- Many-to-many
  - User threads are multiplexed into a number of threads in the kernel.
  - E.g.: IRIX, HP-UX, Solaris (prior to 9)
  - A/D:
    - Blocking call does NOT block all threads..
    - In SMP, several threads can be executed at the same time.

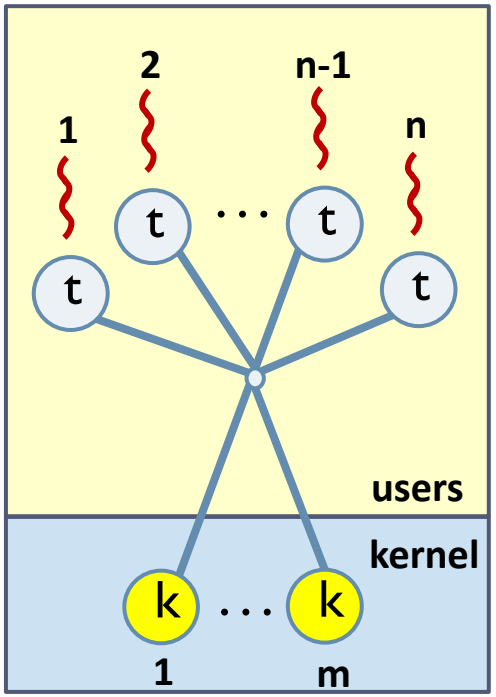
http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Multi-thread model

- Many-to-one
  - Multiple user threads correspond to a single kernel thread.
  - E.g.: library of user threads.
  - A/D:
    - Blocking call blocks all threads.
    - Multiple threads cannot run at the same time in SMP.
- One-to-one
  - Each user thread corresponds to a kernel thread.
  - E.g.: Linux 2.6, Windows, Solaris 9
  - A/D:
    - Blocking call does NOT block all threads..
    - In SMP, several threads can be executed at the same time.

Many-to-many

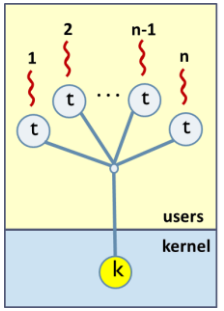
- User threads are multiplexed into a number of threads in the kernel.
- E.g.: IRIX, HP-UX, Solaris (prior to 9)
- A/D:
  - Blocking call does NOT block all threads..
  - In SMP, several threads can be executed at the same time.



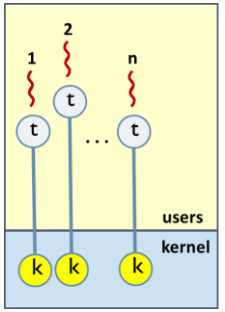
http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Multi-thread model
summary

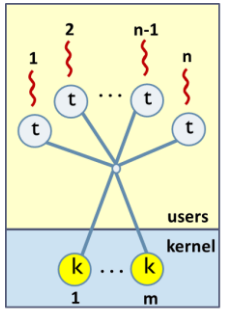
- Many-to-one
  - Multiple user threads correspond to a single kernel thread.
  - E.g.: library of user threads.
  - A/D:
    - Blocking call blocks all threads.
    - Multiple threads cannot run at the same time in SMP.



- One-to-one
  - Each user thread corresponds to a kernel thread.
  - E.g.: Linux 2.6, Windows, Solaris 9
  - A/D:
    - Blocking call does NOT block all threads..
    - In SMP, several threads can be executed at the same time.



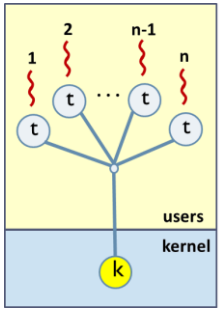
- Many-to-many
  - Each user thread corresponds to a kernel thread.
  - E.g.: Linux 2.6, Windows, Solaris 9
  - A/D:
    - Blocking call does NOT block all threads..
    - In SMP, several threads can be executed at the same time.



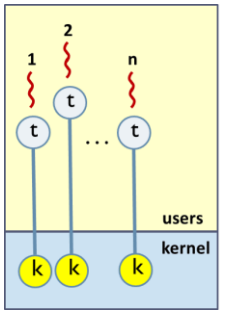
http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

Multi-thread model
summary

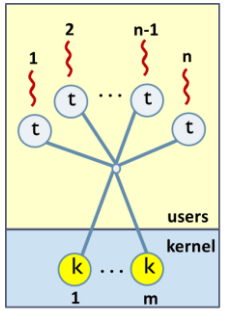
- Many-to-one
  - Multiple user threads correspond to a single kernel thread.
  - E.g.: library of user threads.
  - A/D:
    - Blocking call blocks all threads.
    - Multiple threads cannot run at the same time in SMP.



- One-to-one
  - Each user thread corresponds to a kernel thread.
  - E.g.: Linux 2.6, Windows, Solaris 9
  - A/D:
    - Blocking call does NOT block all threads..
    - In SMP, several threads can be executed at the same time.

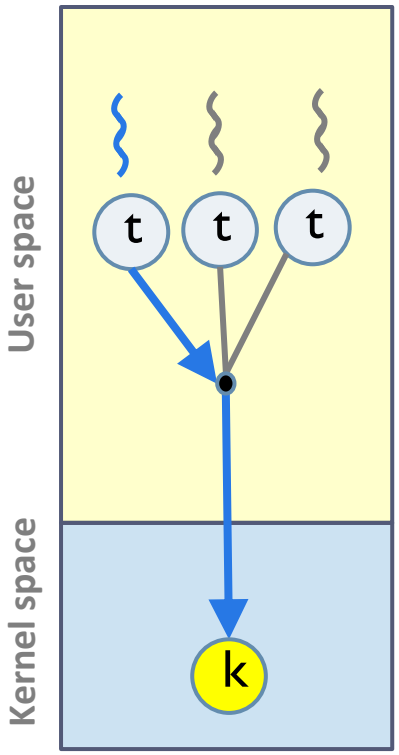


- Many-to-many
  - Each user thread corresponds to a kernel thread.
  - E.g.: Linux 2.6, Windows, Solaris 9
  - A/D:
    - Blocking call does NOT block all threads..
    - In SMP, several threads can be executed at the same time.

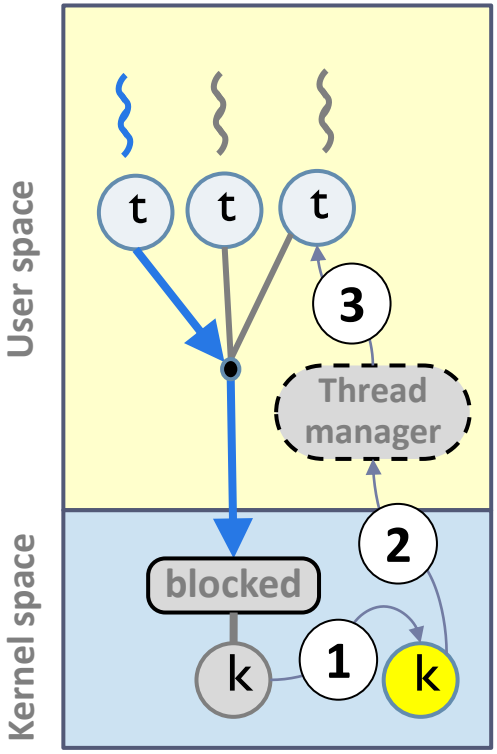


http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

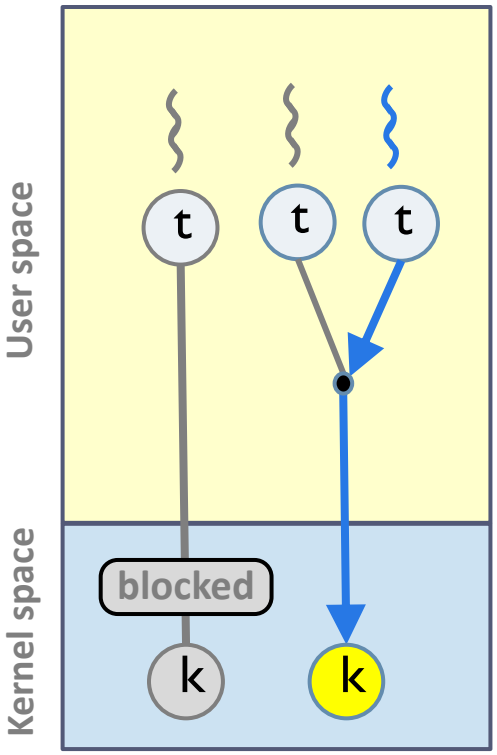
Activation by scheduler



Initial situation



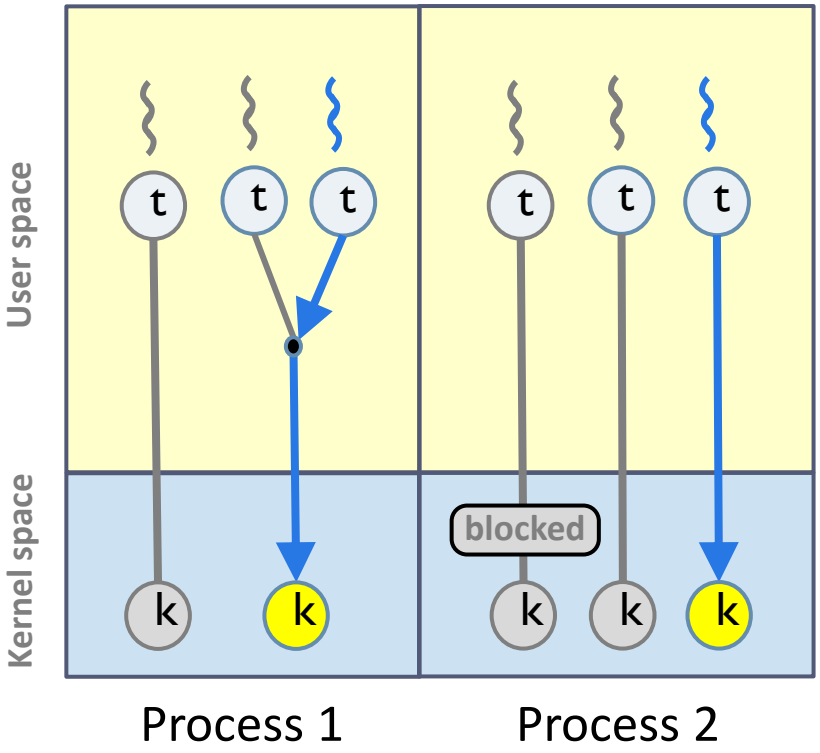
Activation



Final situation

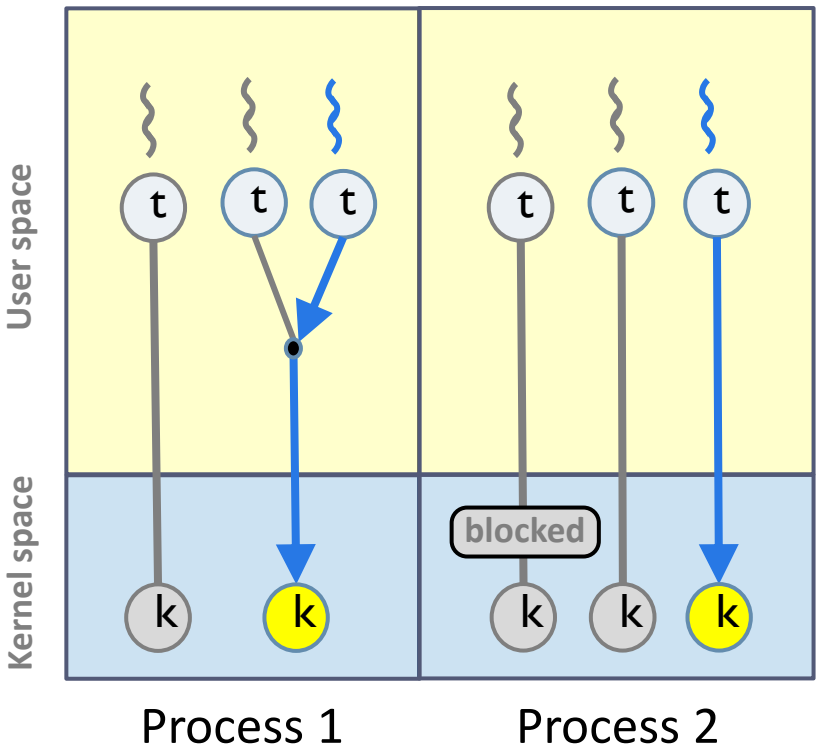
http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

States in a process with threads



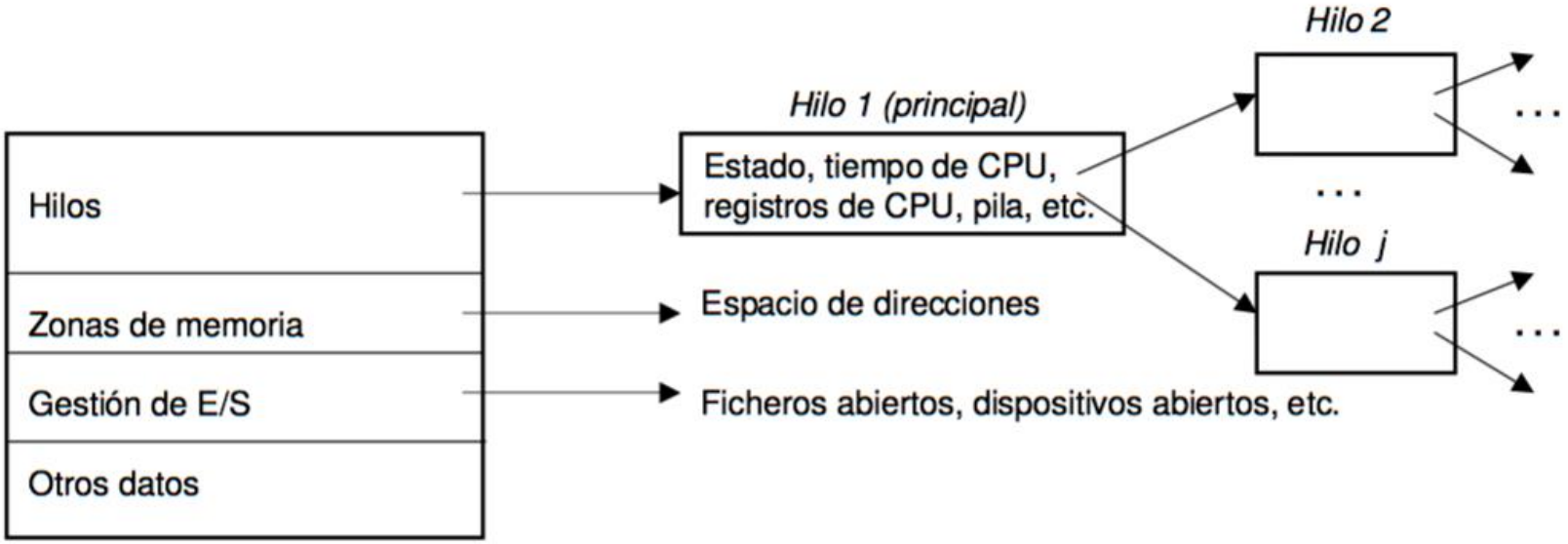
- State of a threaded process is the combination of the states of its threads:
  - If a thread is running
    -> Process running
  - If there are no threads running but they are ready
    -> Process ready
  - If all your threads are locked
    -> Process blocked

Thread scheduling (by default)



- Thread execution scheduling is based on the priority model and does not use the time segmentation model.
  - For example, FIFO with priorities (no *Round-Robin*)
- A thread will continue executing on the CPU until it reaches a state that does not allow it to continue executing.
  - Explicit alternation via sleep() or yield()

BCP of a process with threads



Problems/Peculiarities using threads

- Signal management.
  - *Mono-thread*: Signals in UNIX are used to notify a process that an event has occurred.
  - *Multithread*: Which thread does the signal reach?
    - A) Send the signal to the thread involved. B) Send the signal to all threads. C) Send the signal to certain threads of the process. D) Assign to a specific thread the reception of all process signals.
  - Can signals be sent between threads of the same process?
- Fork() system call.
  - *Mono-thread*: Call to create a copy of the calling process.
  - *Multithread*: If a thread calls *fork()*, does it make a copy of the process with all threads or a copy with only the thread that called fork()?
    - A) System dependent. B) In the copy there is only the thread involved. C) In the copy are all the threads involved. D) There are different system calls that allow the selection of the behavior. E.g. in Linux: clone()

Example (1/3): thread_create() + fork()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
        int tid = pthread_self() ;
        for (int i=0; i<NSECONDS; i++)
        {
             printf("(%d) <pid=%d, tid=%d>\n", i, getpid(), tid) ;
             sleep(1) ;
        }

        pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
        int ret, status ;
        pthread_t tid[NTHREADS+1] ;
        pthread_attr_t attr ;

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

        // Two process making threads...
        // int pid = fork() ;

        // creat threads
        for (int j=0; j<NTHREADS; j++)
        {
             ret = pthread_create(&(tid[j]), &attr, do_something1, NULL) ;
             if (ret) {
                 printf("ERROR on pthread_create(): %d\n", ret) ;
                 exit(-1) ;
             }
        }

        // ONE process making threads...
        int pid = fork() ;

        if (pid != 0)
        {
                // father wait for threads
                for (int j=0; j<NTHREADS; j++)
                {
                     ret = pthread_join(tid[j], NULL) ;
                     if (ret) {
                         printf("ERROR on pthread_join(): %d\n", ret) ;
                         exit(-1) ;
                     }
                }

                // resources back...
                pthread_attr_destroy(&attr) ;
        }
        if (pid != 0)
        {
                // father wait for children
                while (pid != wait(&status)) ;

        }

        return 0;
}
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
    int tid = pthread_self() ;
    for (int i=0; i<NSECONDS; i++)
    {
        printf("(%d) <pid=%d, tid=%d>\n",
                i, getpid(), tid) ;
        sleep(1) ;
    }

    pthread_exit(NULL);
}
```

Example (2/3): thread_create() + fork()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
        int tid = pthread_self() ;
        for (int i=0; i<NSECONDS; i++)
        {
             printf("(%d) <pid=%d, tid=%d>\n", i, getpid(), tid) ;
             sleep(1) ;
        }

        pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
        int ret, status ;
        pthread_t tid[NTHREADS+1] ;
        pthread_attr_t attr ;

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

        // Two process making threads...
        // int pid = fork() ;

        // creat threads
        for (int j=0; j<NTHREADS; j++)
        {
             ret = pthread_create(&(tid[j]), &attr, do_something1, NULL) ;
             if (ret) {
                 printf("ERROR on pthread_create(): %d\n", ret) ;
                 exit(-1) ;
             }
        }

        // ONE process making threads...
        int pid = fork() ;

        if (pid != 0)
        {
                // father wait for threads
                for (int j=0; j<NTHREADS; j++)
                {
                     ret = pthread_join(tid[j], NULL) ;
                     if (ret) {
                         printf("ERROR on pthread_join(): %d\n", ret) ;
                         exit(-1) ;
                     }
                }

                // resources back...
                pthread_attr_destroy(&attr) ;
        }
        if (pid != 0)
        {
                // father wait for children
                while (pid != wait(&status)) ;

        }

        return 0;
}
```

```
int main ( int argc, char *argv[] )
{
    int ret, status ;
    pthread_t tid[NTHREADS+1] ;
    pthread_attr_t attr ;

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

    // creat threads
    for (int j=0; j<NTHREADS; j++)
    {
        ret = pthread_create(&(tid[j]),
                             &attr,
                             do_something1,
                             NULL);
        if (ret) {
            printf("ERROR p_create(): %d\n",
                    ret) ;
                   exit(-1) ;
        }
    }
```

Example (3/3): thread_create() + fork()

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
        int tid = pthread_self() ;
        for (int i=0; i<NSECONDS; i++)
        {
             printf("(%d) <pid=%d, tid=%d>\n", i, getpid(), tid) ;
             sleep(1) ;
        }

        pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
        int ret, status ;
        pthread_t tid[NTHREADS+1] ;
        pthread_attr_t attr ;

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

        // Two process making threads...
        // int pid = fork() ;

        // creat threads
        for (int j=0; j<NTHREADS; j++)
        {
             ret = pthread_create(&(tid[j]), &attr, do_something1, NULL) ;
             if (ret) {
                 printf("ERROR on pthread_create(): %d\n", ret) ;
                 exit(-1) ;
             }
        }

        // ONE process making threads...
        int pid = fork() ;

        if (pid != 0)
        {
                // father wait for threads
                for (int j=0; j<NTHREADS; j++)
                {
                     ret = pthread_join(tid[j], NULL) ;
                     if (ret) {
                         printf("ERROR on pthread_join(): %d\n", ret) ;
                         exit(-1) ;
                     }
                }

                // resources back...
                pthread_attr_destroy(&attr) ;
        }
        if (pid != 0)
        {
                // father wait for children
                while (pid != wait(&status)) ;

        }

        return 0;
}
```

```
        // new process with fork()
        int pid = fork() ;

        if (pid != 0)
        {    // father wait for threads
             for (int j=0; j<NTHREADS; j++)
             {
                 ret = pthread_join(tid[j], NULL);
                 if (ret) {
                    printf("ERROR p_join(): %d\n",
                           ret);
                    exit(-1) ;
                 }
             }

             // resources back...
             pthread_attr_destroy(&attr) ;
        }
        if (pid != 0) {
             // father wait for children
             while (pid != wait(&status)) ;
        }

        return 0;
}
```

Example: thread_create() + fork()

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NTHREADS 2
#define NSECONDS 3

void * do_something1 ( void *arg )
{
        int tid = pthread_self() ;
        for (int i=0; i<NSECONDS; i++)
        {
             printf("(%d) <pid=%d, tid=%d>\n", i, getpid(), tid) ;
             sleep(1) ;
        }

        pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
        int ret, status ;
        pthread_t tid[NTHREADS+1] ;
        pthread_attr_t attr ;

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

        // Two process making threads...
        // int pid = fork() ;

        // creat threads
        for (int j=0; j<NTHREADS; j++)
        {
             ret = pthread_create(&(tid[j]), &attr, do_something1, NULL) ;
             if (ret) {
                 printf("ERROR on pthread_create(): %d\n", ret) ;
                 exit(-1) ;
             }
        }

        // ONE process making threads...
        int pid = fork() ;

        if (pid != 0)
        {
                // father wait for threads
                for (int j=0; j<NTHREADS; j++)
                {
                     ret = pthread_join(tid[j], NULL) ;
                     if (ret) {
                         printf("ERROR on pthread_join(): %d\n", ret) ;
                         exit(-1) ;
                     }
                }

                // resources back...
                pthread_attr_destroy(&attr) ;
        }
        if (pid != 0)
        {
                // father wait for children
                while (pid != wait(&status)) ;

        }

        return 0;
}
```

```
alex@patata:$ gcc -Wall -g \
                  -o ths_and_fork \
                  ths_and_fork.c -lpthread


alex@patata:$ ./ths_and_fork
(0) <pid=426, tid=-1144539392>
(0) <pid=426, tid=-1136146688>
(1) <pid=426, tid=-1144539392>
(1) <pid=426, tid=-1136146688>
(2) <pid=426, tid=-1144539392>
(2) <pid=426, tid=-1136146688>
```

Contents

1. Introduction
   - Process definition.
   - Model offered: resources, multiprogramming, multitasking and multiprocessing
2. Process life cycle: process status.
3. Services to manage processes provided by the operating system.
4. Definition of thread
5. Kernel and library threads.
6. **Services for threads in the operating system.**

“heavy” process versus “light” process

similar but not the same calls

	Heavy process	Light process (thread)
Create	fork ()	pthread_create (…)
Wait	wait (…)	pthread_join (…)
Exit	exit (…)	pthread_exit (…)
Identify	getpid()	pthread_self (…)

Creation of a thread

Service	`int pthread_create(pthread_t *thread,`
	`const pthread_attr_t *attr,`
	`void *(*func)(void *),`
	`void *arg);`

Arguments	▫ **thread**: address of a pthread_t variable to store the thread identifier. ▫ **attr**: address of a structure with thread attributes. NULL can be passed to use default attributes. ▫ **func**: pointer to function to be executed by the thread. ▫ **arg**: pointer to the thread parameter. Only one parameter can be passed (it can be a pointer to a structure).
Returns	▫ 0 if everything goes well. ▫ Error code in case of error.
Description	▫ Request the creation of a thread.

Waiting for thread

Service	`int pthread_join (pthread_t thread, void **value) ;`
Arguments	▫ **thread**: identifier of the thread to wait for. ▫ **value**: if the value is other than NULL, it will be the memory address where to store the thread termination value.
Returns	▫ 0 if everything goes well. ▫ Error code in case of error.
Description	▫ The thread invoking the function waits until the termination of the thread whose identifier is specified by parameter. If the thread had already terminated, then *pthread_join* terminates immediately. Two threads executing *pthread_join* by the same thread implies undefined behavior.

Thread termination

Service	`void pthread_exit(void *retval) ;`
Arguments	▫ **retval**: thread termination status. CANNOT be a pointer to a local variable.
Returns	▫ Returns nothing.
Description	▫ Allows a lightweight process to terminate its execution, indicating its termination status to the parent thread (if it is "joinable"). ▫ When a thread terminates, the shared resources at the process level (mutex, semaphores, files, etc.) are not released.

Thread identification

Service	`pthread_t pthread_self(void) ;`
Arguments	▫ No arguments.
Returns	▫ Returns the identifier of the thread executing the call.
Description	▫ Allows a thread to know its identifier.

Example: creating and waiting

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

#define NUM_THREADS     5
pthread_t threads[NUM_THREADS];

void *th_function ( void *arg )
{
    printf("Hello world from thread #%ld!\n", (long)arg);
    pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
    for (int t=0; t<NUM_THREADS; t++) {
        int rc = pthread_create(&(threads[t]), NULL, th_function, (void *)(long)t);
        if (rc) { printf("ERROR from pthread_create(): %d\n", rc); exit(-1); }
    }
    for (int t=0; t<NUM_THREADS; t++) {
        int rc = pthread_join(threads[t], NULL) ;
        if (rc) { printf("ERROR from pthread_join(): %d\n", rc); exit(-1); }
    }
    pthread_exit(NULL);
}
```

Example: crea

```
# gcc -Wall -g -o h h.c –lpthread
# ./h
Hello world from thread #0!
Hello world from thread #2!
Hello world from thread #1!
Hello world from thread #3!
Hello world from thread #4!
```

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

#define NUM_THREADS     5
pthread_t threads[NUM_THREA

void *th_function ( void *arg )
{
    printf("Hello world from thread #%ld!\n", (long)arg);
    pthread_exit(NULL);
}

int main ( int argc, char *argv[] )
{
    for (int t=0; t<NUM_THREADS; t++) {
        int rc = pthread_create(&(threads[t]), NULL, th_function, (void *)(long)t);
        if (rc) { printf("ERROR from pthread_create(): %d\n", rc); exit(-1); }
    }
    for (int t=0; t<NUM_THREADS; t++) {
        int rc = pthread_join(threads[t], NULL) ;
        if (rc) { printf("ERROR from pthread_join(): %d\n", rc); exit(-1); }
    }
    pthread_exit(NULL);
}
```

Thread attributes

Service	`int pthread_create(pthread_t *thread,`
	`const pthread_attr_t *attr,`
	`void *(*func)(void *),`
	`void *arg);`

Arguments	▫ **thread**: address of a pthread_t variable to store the thread identifier. ▫ **attr**: address of a structure with thread attributes. N[illegible] can be passed to use default attributes. ▫ [illegible]nter to function to be executed by the thread. [illegible]o the thread parameter. Only one parameter can be

- A thread is either independent or dependent.
- The size of the thread's private stack.
- The location of the thread's stack.
- The scheduling policy of the thread.

Attributes of a thread

	Get	Set
Split or not	pthread_attr_getdetachstate (…)	pthread_attr_setdetachstate (…)
Stack size	pthread_attr_getstacksize (…)	pthread_attr_setstacksize (…)
Stack location	pthread_attr_getstackaddr (…)	pthread_attr_setstackaddr (…)
Scheduling policy	pthread_attr_getscope (…)	pthread_attr_setscope (…)

Example: attributes for threads

```
....

int main ( int argc, char *argv[] )
{
    pthread_attr_t attr;
    int ret;  size_t stacksize;

    ret = pthread_attr_init(&attr) ;
    ret = pthread_attr_getstacksize(&attr, &stacksize);
    ret = pthread_attr_setstacksize(&attr,   stacksize);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

- `int pthread_attr_init(pthread_attr_t * attr);`
  - Initiates a thread attribute structure.
- `int pthread_attr_destroy(pthread_attr_t * attr);`
  - Destroys a thread attribute structure.

```
    int ret;  size_t st

    ret = pthread_attr_init(&attr) ;
    ret = pthread_attr_getstacksize(&attr, &stacksize);
    ret = pthread_attr_setstacksize(&attr,   stacksize);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

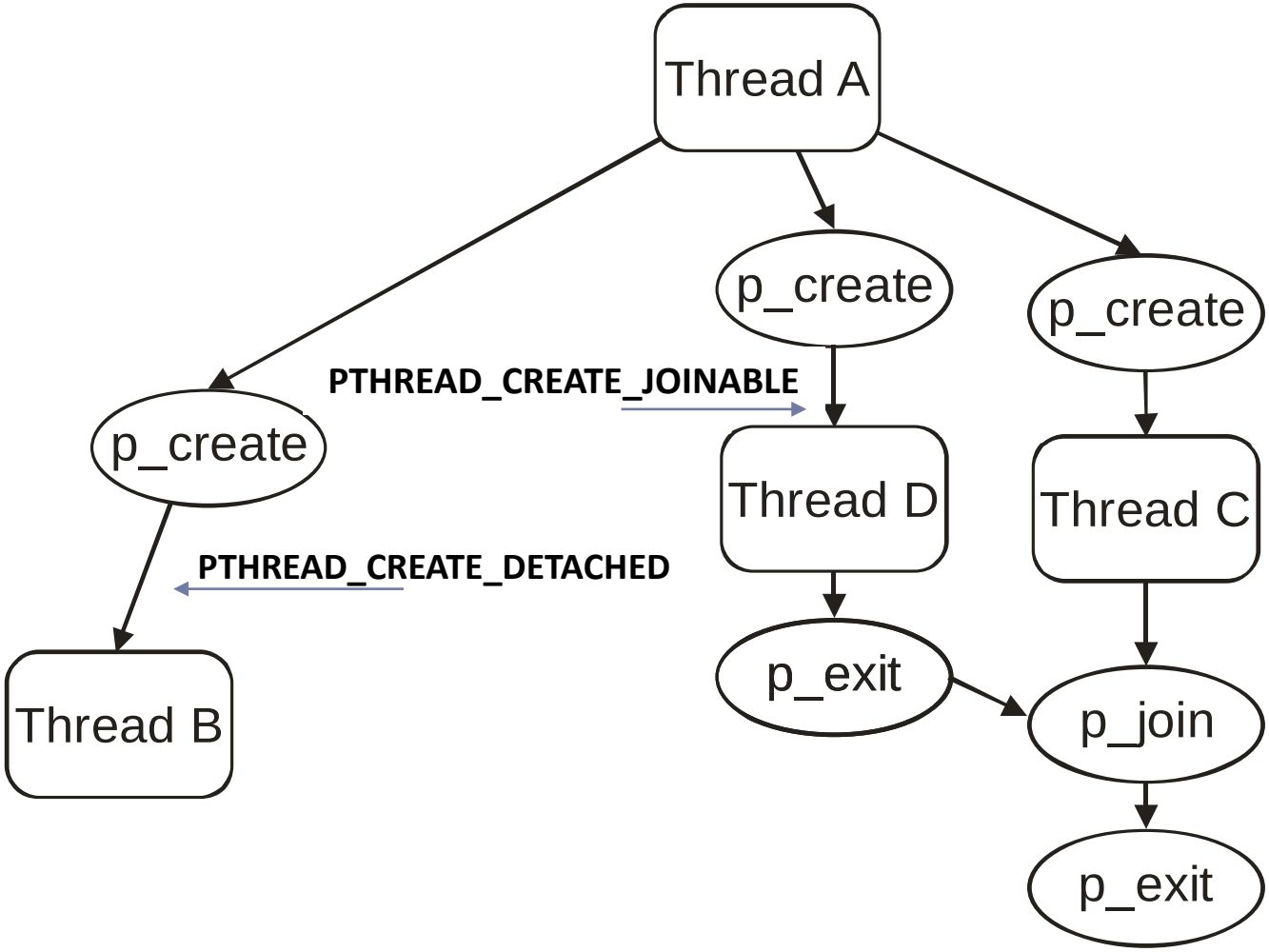
- `int pthread_attr_setdetachstate ( pthread_attr_t *attr, int detachstate);`
  - Set the type of behavior at exit with detachstate:
    - PTHREAD_CREATE_JOINABLE: join with pthread_join
    - PTHREAD_CREATE_DETACHED: free resources and detach.
- `int pthread_attr_getdetachstate ( pthread_attr_t *attr, int *detachstate);`
  - Allows to obtain the type of behavior at exit.

```
    ret = pthread_a
    ret = pthread_attr_ge                (&attr, &stacksize);
    ret = pthread_attr_setstacks.e(&attr,   stacksize);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

Example of thread hierarchy



- `int pthread_attr_setscope ( pthread_attr_t *attr, int scope);`
  - Allows to indicate the desired scheduler:
    - PTHREAD_SCOPE_PROCESS: threads of the same process -> PCS.
    - PTHREAD_SCOPE_SYSTEM: threads from any process -> SCS.
  - It is possible that the OS can limit, E.g.: Linux and MacOS to users -> SCS.
- `int pthread_attr_getscope ( pthread_attr_t *attr, int *scope);`
  - Allows to obtain the type of scheduler.

```
    ret = pthread_attr_...
    ret = pthread_attr_getscope(&attr, &scope);
    ret = pthread_attr_setscope(&attr,   scope);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

- `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 stack size of a thread.

```
    int ret;  size_t stacksize;

    ret = pthread_attr_init(&attr) ;
    ret = pthread_attr_getstacksize(&attr, &stacksize);
    ret = pthread_attr_setstacksize(&attr,   stacksize);
    ret = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED) ;

    for (int t=0; t<NUM_THREADS; t++) {
          ret = pthread_create(&(threads[t]), &attr, th_function, (void *)(long)t);
          if (ret) { printf("ERROR from pthread_create(): %d\n", ret); exit(-1);  }
    }
    sleep(10) ;

    pthread_exit(NULL);
    ret = pthread_attr_destroy(&attr) ;
}
```

ARCOS Group
Universidad Carlos III de Madrid

Lesson 3
Process and threads

Operating Systems
Computer Science and Engineering