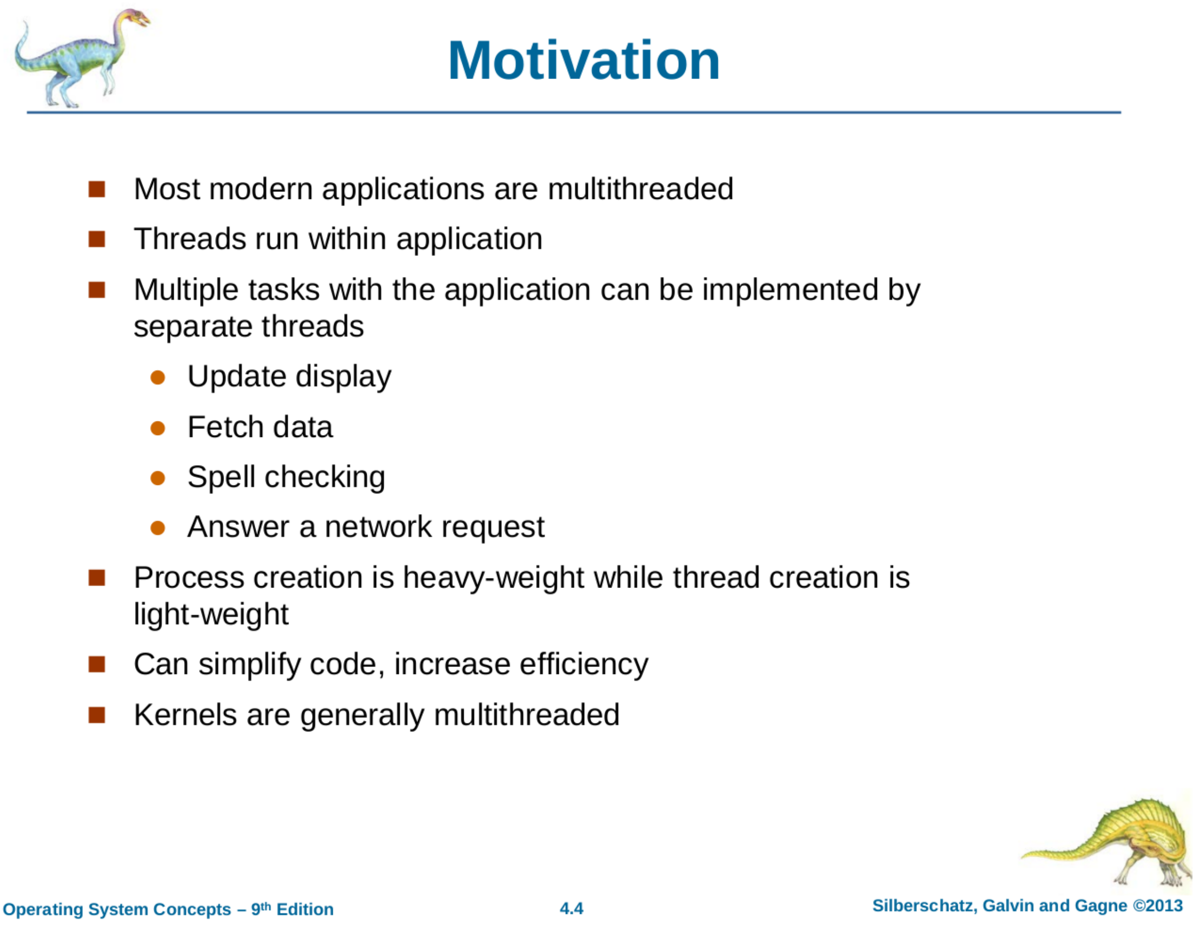
Tutoriat 5 SO



Motivation

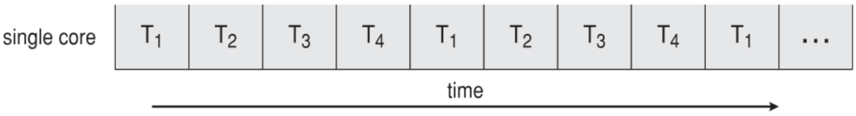
- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Concurrency vs. Parallelism

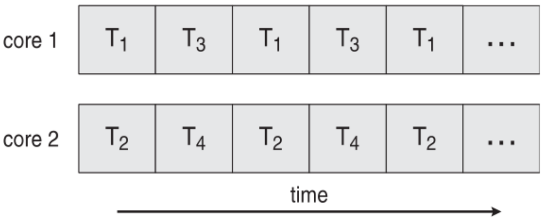
- **Concurrent execution on single-core system:**



single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	. . .

time

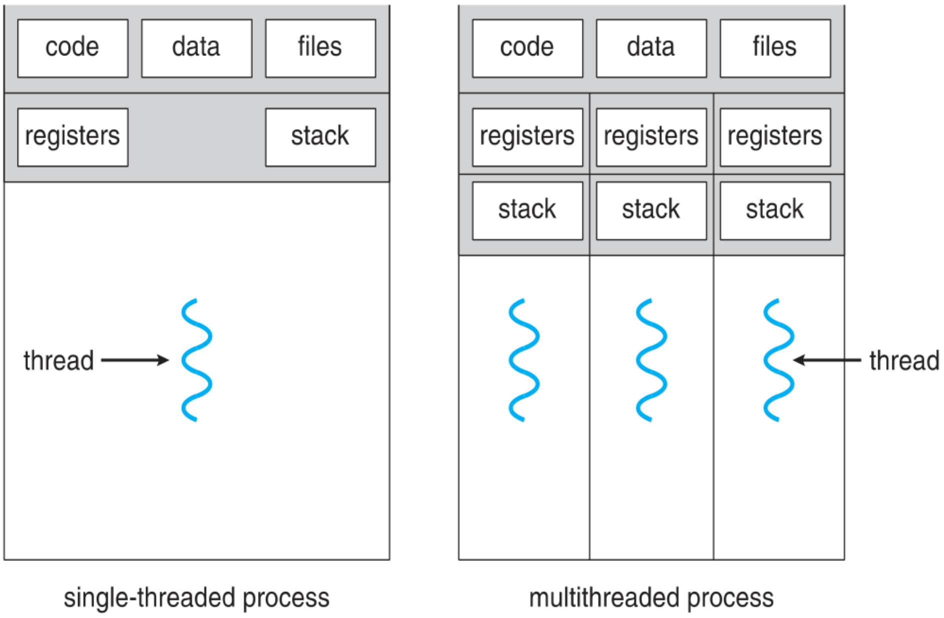
- **Parallelism on a multi-core system:**



core 1	T_1	T_3	T_1	T_3	T_1	. . .
core 2	T_2	T_4	T_2	T_4	T_2	. . .

time

Single and Multithreaded Processes

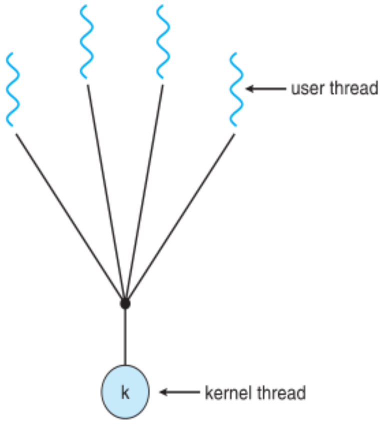


Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

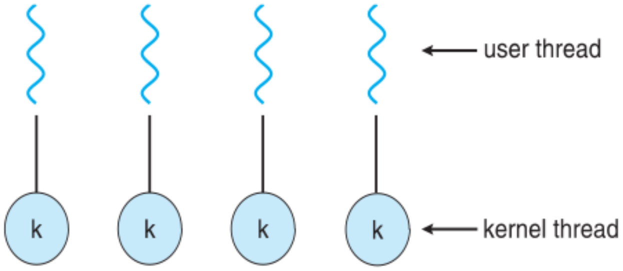
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



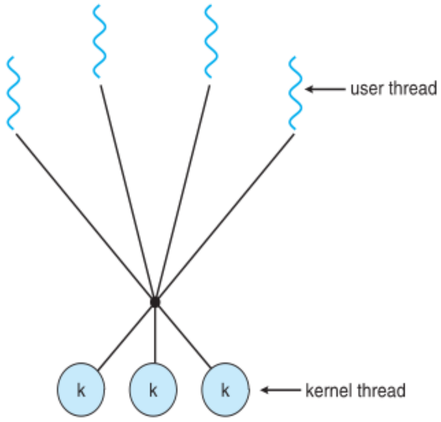
One-to-One

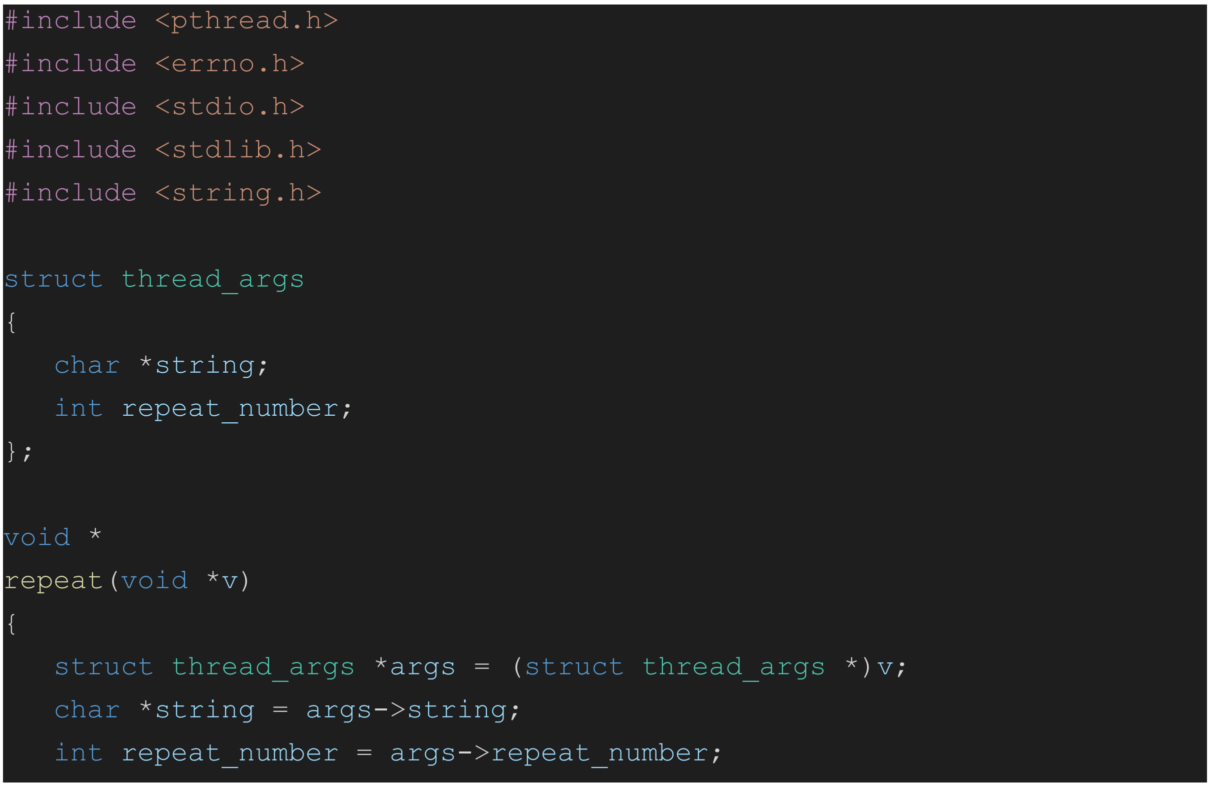
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package





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

struct thread_args
{
   char *string;
   int repeat_number;
};

void *
repeat(void *v)
{
   struct thread_args *args = (struct thread_args *)v;
   char *string = args->string;
   int repeat_number = args->repeat_number;
   int len = strlen(string) * repeat_number;
   char *new_string = (char *)malloc(len + 1);

   while (repeat_number--)
   {
       strcat(new_string, string);
   }

   printf("Thread finished\n");

   return new_string;

   // Functia din thread aloca dinamic (pe heap) sirul nou,
   // apoi returneaza adresa acelui sir
}

int main()
{

   /*

   NAME
       pthread_create - create a new thread

   SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t
*attr,
                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.

   DESCRIPTION
       The  pthread_create()  function  starts  a  new  thread  in the
calling
       process.  The new thread starts execution by invoking
start_routine();
       arg is passed as the sole argument of start_routine().

       The attr argument points to a pthread_attr_t structure  whose
contents
```

```
       are  used  at  thread creation time to determine attributes for the new
       thread; this structure is initialized  using  pthread_attr_init(3)  and
       related  functions.   If  attr is NULL, then the thread is created with
       default attributes.

   RETURN VALUE
       On  success,  pthread_create() returns 0; on error, it returns an error
       number, and the contents of *thread are undefined.

   */


   // Pentru a trimite mai multe argumente catre functie prin void *arg
   // putem construi o structura care sa contina toate argumentele si sa
   // ii dam lui arg adresa obiectului.
   struct thread_args args;
   args.string = "Ceva";
   args.repeat_number = 5;


   pthread_t thr;

   if (pthread_create(&thr, NULL, repeat, &args))
   {
       perror(NULL);
       return errno;
   }

   /*

   NAME
       pthread_join - join with a terminated thread

   SYNOPSIS
       #include <pthread.h>

       int pthread_join(pthread_t thread, void **retval);
       Compile and link with -pthread.

   DESCRIPTION
       The pthread_join() function waits for the thread specified by
thread to
       terminate.  If that thread has already terminated, then
pthread_join()
       returns immediately.  The thread specified by thread must be
joinable.

       If  retval  is  not NULL, then pthread_join() copies the exit
status of
       the target thread (i.e., the value that the target thread
supplied  to
       pthread_exit(3)) into the location pointed to by retval.  If the
target
       thread was canceled, then PTHREAD_CANCELED is placed  in  the
location
       pointed to by retval.

   RETURN VALUE
       On success, pthread_join() returns 0; on error,  it  returns  an
error
       number.

   */

   void *result;

   if (pthread_join(thr, &result))
   {
       perror(NULL);
       return errno;
   }

   printf("Main thread received: %s\n", (char *)result);

   free(result);

   return 0;
}
```

Problema 2

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

#define SIZE 5

/// 2. Scrieti un program ce aduna 2 vectori intre ei in thread-uri separate

int mat1[SIZE][SIZE] = {
       {1, 3, 5, 7, 1},
       {1, 3, 5, 7, 1},
       {1, 3, 4, 7, 1},
       {1, 3, 5, 0, 1},
       {1, 3, 5, 7, 1},
};
int mat2[SIZE][SIZE] = {
       {10, 4, -10, 11, 1},
       {11, 4, -10, 11, 1},
       {10, 4, -10, 11, 1},
       {15, 4, -19, 11, 1},
       {10, 4, -10, 11, 1},
};
int result[SIZE][SIZE];

struct pos {
   int i, j;
};

void *sum(void *pos_void) {
   struct pos* index = (struct pos*)pos_void;
   result[index->i][index->j] = mat1[index->i][index->j] + mat2[index->i][index->j];
   return (void*) (index->i * SIZE + index->j);
}

int main() {
   pthread_t *threads = malloc(sizeof(pthread_t) * SIZE * SIZE);
   /// threads [0 ... 24]
   for (int i = 0; i < SIZE; ++i) {
       for (int j = 0; j < SIZE; ++j) {
           struct pos *index = malloc(sizeof(struct pos));
```

```
            index->i = i;
            index->j = j;
            pthread_create(threads + (i*SIZE + j), NULL, sum, 
index);
        }
    }
    for (int i = 0; i < SIZE * SIZE ; ++i){
        void *void_index = malloc(sizeof (int));
        pthread_join(threads[i], void_index);
        int* index = (int*)void_index;
        printf("%d ", *index);
    }
    printf("\n");
    for (int i = 0; i < SIZE; ++i) {
        for (int j = 0; j < SIZE; ++j)
            printf("%d ", result[i][j]);
        printf("\n");
    }
    return 0;
}
```