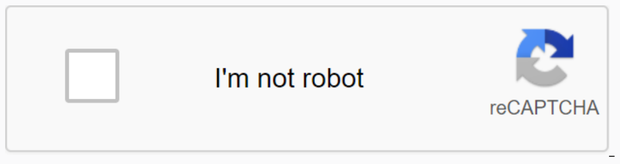

Continue

Multithreading c

C is a language that works on a single thread by default, which means that your code will run only one statement at a time. In some cases, you need to execute multiple statements at once, for example, the graphical interface will not stop when you perform a button-click action. What we call multithreading should not be confused with asynchronous operations that can only be performed on a single thread that performs multiple tasks. What is a thread? A thread is a task that runs linked to a process. The process we have multiple threads and threads can run other threads and so on. By default, the process runs on a single thread. Each thread is a new task that can run indefinitely and parallel to other threads. Creating a thread As said in the title, this post will talk about multithreading in C, so we'll do C! On POSIX operating systems there is a library called pthread.h that does exactly what it says, creates threads! To use it in compilers, you must link it to the -lpthread argument (e.g. #include #include <pthread.h>#include void <unistd.h> <stdio.h>wait(void) { sleep(2); printf(Done.); } int main(void) { pthread_t thread; int err; err = pthread_create(&thread, NULL, wait, NULL); if (err) { printf(Error occurred: %d, err); return 1; } printf(Waiting for thread to end...); pthread_join(thread, NULL); printf(Thread terminated.); return 0; } Output will be: Waiting for the thread to finish... Do. (~2 seconds after) Thread terminated. The main thread continued and printed the message while the created thread worked and it was only one line to call the function in the new thread, pretty easy for C uh? Maybe you're a little lost, I'll explain the code. First of all, we include the pthread.h library, as said above, contains all the features needed to perform multithreaded tasks. Next, we include unistd.h, which includes the sleep() function. And then stdio.h for printf(). pthread_create function is called to create a thread. This pthread_t, which is a thread descriptor, a pointer to the void function, and a few other parameters that I will not describe here, function arguments for example. pthread_join is used to link the current thread process to another thread. It literally makes the program stop to wait for the end of the selected thread. Mutex Performing multiple operations on one target at the same time is very dangerous, the best example I can give is a database. If three threads want to write one file at the same time, it would be a problem because the hard drive can not go as fast as the processor. In this case, we should lock other threads so as not to overload the hard drive with tasks that can damage the file. Mutex can be used to block other threads. In practice, we use a mutex to determine if a task is locked or unlocked. #include #include <pthread.h> <unistd.h>#include <stdio.h>lock int j; void do_process() {</stdio.h> </unistd.h> </pthread.h> </stdio.h> </unistd.h> </pthread.h> </pthread.h> int i = 0; j++; while(i < 5)= {= printf(%d,= j);= sleep(1);= i++;= }= printf(... done);= pthread_mutex_unlock(&lock);= }= int= main(void)= {= int= err;= pthread_t= t1,= t2;= if= (pthread_mutex_init(&lock,= null)= !=0) {= printf(mutex= initialization= failed.); = return= 1;= }= j=0; pthread_create(&t1,= null,= do_process,= null);= pthread_create(&t2,= null,= do_process,= null);= pthread_join(t1,= null);= pthread_join(t2,= null);= return= 0;= }= the= output= will= be:= 111111...done= 222222...done= so,= as= you= can= see,= the= thread= t2= was= done= totally= after= t1,= so= the= mutex= worked!= it's= not= the= result= of= the= pthread_join= calls= as= they= only= change= something= for= the= current= thread= (here= the= main= thread),= you= can= try= to= join= t2= before= t1,= it= will= perform= the= same= thing.= semaphores= pretty= much= the= same= thing= as= the= mutex,= the= semaphores= are= used= to= be= controlled= by= any= thread.= the= notion= of= ownership= is= not= present= in= the= case= of= a= semaphore,= so= they= can= be= locked= and= unlocked= by= any= part= of= the= program.= that= was= pretty= easy= don't= you= think?= c= functions= are= very= powerful= to= perform= such= actions ,= a= great= power= carries= a= great= responsibility.= c11= threads= in= glibc= 2.28.= tested= in= ubuntu= 18.04= (glibc= 2.27)= by= compiling= glibc= from= source:= multiple= glibc= libraries= on= a= single= host= example= from:= #include=> <stdio.h>#include <threads.h>#include <stdatomic.h>atomic_int acnt; int cnt; int f(void* thr_data) { for(int n = 0; n < 1000;= ++n)= {= ++cnt;= ++acnt;= for= this= example,= relaxed= memory= order= is= sufficient,= e.g.= atomic_fetch_add_explicit(&acnt,= 1,= memory_order_relaxed);= }= return= 0;= }= int= main(void)= {= thrd_t= thr[10];= for(int= n=0; n=> < 10;= ++n)= thrd_create(&thr[n],= f,= null);= for(int= n=0; n=> < 10;= ++n)= thrd_join(thr[n],= null);= printf(the= atomic= counter= is= %u,= acnt);= printf(the= non-atomic= counter= is= %u,= cnt);= }= github= upstream.= compile= and= run:= gcc= -std=c11 main.c= -pthread= ./a.out= possible= output:= the= atomic= counter= is= 10000= the= non-atomic= counter= is= 8644= the= non-atomic= counter= is= very= likely= to= be= smaller= than= the= atomic= one= due= to= racy= access= across= threads= to= the= non= atomic= variable.= todo := disassemble= and= see= what= ++acnt;= compiles= to.= posix= threads= #define= _xopen_source= 700= #include=> <assert.h>#include <stdlib.h>#include <pthread.h>wyliczenia STAŁE { NUM_THREADS = 1000, NUM_ITERS = 1000 }; int global = 0; int fail = 0; pthread_mutex_t main_thread_mutex = PTHREAD_MUTEX_INITIALIZER; void* main_thread(void *arg) { int i; for (i = 0; i < num_iters;= ++i)= {= if= (!fail)= pthread_mutex_lock(&main_thread_mutex);= global++;= if= (!fail)= pthread_mutex_unlock(&main_thread_mutex);= return= null;= }= int= main(int= argc,= char= **argv)= {= pthread_t= threads[num_threads];= int= i;= fail=argc> 1; for (i = 0; i < num_threads;= ++i)= pthread_create(&threads[i],= null,= main_thread,= null);= for= (i=0; i=> < NUM_THREADS; ++i) pthread_join(threads[i], NULL); assert(global == NUM_THREADS * NUM_ITERS); return EXIT_SUCCESS; } Compile and run: gcc -std=c99 pthread_mutex.c -pthread ./a.out num_threads;= ++i)= pthread_join(threads[i],= null);= assert(global== num_threads= *= num_iters);= return= exit_success;= }= compile= and= run:= gcc= -std=c99 pthread_mutex.c= -pthread= ./a.out=></ NUM_THREADS; ++i) pthread_join(threads[i], NULL); assert(global == NUM_THREADS * NUM_ITERS); return EXIT_SUCCESS; } Compile and run: gcc -std=c99 pthread_mutex.c -pthread ./a.out > </pthread.h> </stdlib.h> </assert.h> </stdatomic.h> </threads.h> </stdio.h> </stdio.h> 1 The first run works correctly, the second fails due to lack of synchronization. Tested on Ubuntu 18.04. GitHub parent. Pierre Ficheux (pficheux@com1.fr) April 1999 This article is an introduction to multithreaded programming on LINUX. Programming examples use a library that LinuxThreads are normally available on most Linux r centes distributions. these examples are available in pficheux/articles/lmf/threads/examples.tar.gz reading this article requires a pretty good understanding of the C... 1. What is multithreaded? Linux and UNIX developers in general have long been accustomed to the multitasking features of their predecessor. All those who have rubbed a little programming system know that it is ais in UNIX to create thread processes from an existing process using a call system fork, as shown in the small code example below: #include <stdio.h>#include <stdlib.h>#include <unistd.h>int i; hand (int ac, tank) int pid; and - 1; if ((pid - fork()) - 0) - / In the yyarn/printf (I am a son, pid -%d, getpid()); sleep (2); printf (Son's end, i -% d!, exit (0); and else if (pid > 1(1); /Changes the variable - i - 2; printf (Father a variable a %d (i); i/d); I am the father (i) sleep (i- 2; , and -%d!, i); output (0); Other / Perror error (fork); output (1); which gives excution: pierre@mmxpf % ./fork [1] 367 pierre@mmxpf% I am a father, pid 367 I am a son, pid 368 father modified the variable to 2 End of son, and -1! The end of father, and 2! The boundaries of the fork appear when it comes to dividing variables between a frequent process and his son. As you can see in the small example below, the global variable and, modified by the father, always has an old value in the thread. This is normal fork behavior that duplicates the current context when creating a thread process. In addition to making it difficult to share variables, creating a new context is important in terms of performance. The same is said for changing the context (context switcher) when passing one process to another. A thread very much resembles a classic difference thread process that shares much more data with the process that created it: Global variables Local static variables File descriptors (file descriptors) Multithreading is therefore a programming technique that allows you to take advantage (as well as some limitations) of using threads. 2. Thread Libraries Many operating systems now enable thread programming: SUN Solaris 5.x, Windows95/98/NT, and many others (including LINUX). W</unistd.h> </stdlib.h> </stdio.h>solaris, the available thread library is POSIX 1003.1c compatible, which provides some ability to move applications when transferred to another system. For Microsoft systems, the library you are using is obviously not compatible with this POSIX standard! Currently, there are various libraries that can handle threads on LINUX. There are two main types of threads: user-level. At this time, thread management is completely performed in the user space. At the kernel level. In this case, the threads are directly managed by the kernel. In the latter case, the basis of the implantation is, inter alia, the system call of clones, also used to create classic processes: clone NAME - creation of the child process SYNOPSIS #include <linux ched.h=> <linux nistd.h=>#include pid_t clone (void ,sp, unsigned long flags) Clone DESCRIPTION is an alternative interface for fork, with more options. COPYVM). The linuxthreads library developed by Xavier Leroy (Xavier.Leroy@inria.fr) is an excellent implementation of the posIX 1003.1c standard. This library is based on a cloning system call. I can not advise too much to use this product, which we will do in the following examples in this article. For a record of this library that is provided as standard in redhat 5 distribution. 3. How do I create threads on LINUX? Here's a small example of a program that uses two display threads: #include <stdio.h>#include #include void my_thread_process <stdlib.h> <pthread.h>(void - arg) - int i; for < 5= ;= i++)= {= printf= (thread= %s:= %d,= (char*)arg,= i);= sleep= (1);= } pthread_exit= (0);= }= main= (int = ac,= char= **av)= {= pthread_t= th1,= th2;= void= *ret;= if= (pthread_create= (&th1,= null,= my_thread_process,= 1)=> < 0)= { fprintf= (stderr,= pthread_create= error= for= thread= 1);= exit= (1);= } if= (pthread_create= (&th2,= null,= my_thread_process,= 2)=> < 0) { fprintf (stderr, pthread_create error for thread 2); exit (1); } (void)pthread_join (th1, &ret); (void)pthread_join (th2, &ret); } La fonction pthread_create permet de cr er le thread et de l'associer la fonction my_thread_process. He notera que le param tre void *arg est pass au thread lors de sa cr ation. Apr s cr ation des deux threads, le programme principal attend la fin des threads en utilisant la fonction pthread_join. Apr s compilation de ce programme par la commande: cc -D_REENTRANT -o thread1 thread1.c -lpthread Il donne l'ex cution: pierre@mmxpf % ./thread1 Thread 1: 0 Thread 2: 0 Thread 1: 1 Thread 2: 1 Thread 1: 2 Thread 2 2 Thread 1: 3 Thread 2: 3 Thread 1: 4 Thread 2: 4 Le partage de donn es n cessite parfois (m me souvent) d'utiliser des techniques permettant de prot ger un instant donn une variable partag e par plusieurs threads. Imaginons un simple tableau d'entier rempli par un 0)= {= fprintf= (stderr,= pthread_create= error= for= thread= 2);= exit= (1);= }= (th1, void)pthread_join (th2, 'th2'); 'the function' pthread_create allows to create the thread and associate it - the function my_thread_process.- it will be noted that the param tre void is 'pass' to the thread its creation. after the creation of the two threads, the main program waits for the end of the threads using the function pthread_join. after compilation of this program by the command: 'cc' -d_reentrant -o-thread1-thread1.c--lpthread- 'ex cution: 'pierre@mmxpf %'. ./thread1 'thread'1: '0' thread 2: '0' thread 1: '1' thread 2: '1' thread 1: '2' thread 2: '2' thread 1: '3' thread 2: '3' thread 1: 'thread' 2: 'thread' 2: The sharing of data is sometimes (often) to use techniques that allow for control - one instant - a variable - shared by several threads. , pthread_create error for thread 2); exit (1); (void)pthread_join (th1, (void)pthread_join (th2, The pthread_create function allows you to create the thread and combine it with the my_thread_process function. It should be noted that the param tre void is passed to the thread during its creation. After creating both threads, the main program waits for the threads to end using the pthread_join function. After compiling this program by ordering: cc-D_REENTRANT -o thread1 thread1.c-lpthread It gives the ex cution: pierre@mmxpf % ./thread1 Thread 1: 0 Thread 2: 0 Thread 1: 1 Thread 2: 1 Thread 1: 1 Thread 1: 2 Thread 2: 2 Thread 1: 3 Thread 2: 3 Thread 1: 4 Thread 2: 4 Data sharing is sometimes (even often) used to use techniques to protect a moment given a variable shared by several threads. Let's imagine a simple whole table filled with a 'i-0'/pthread.h'; (lent) et lu par un autre (plus rapide). Le thread de lecture doit attendre la fin du remplissage du tableau avant d'afficher sont contenu. Pour cela, on peut utilizer le syst me des MUTEX (MUTual EXclusion) afin de prot ger le tableau pendant le temps de son remplissage: #include <stdio.h>#include <stdlib.h>#include pthread_mutex_t my_mutex <pthread.h>statyczne; static int tab[5]; void *read_tab_process (void * arg) { int i; pthread_mutex_lock (&my_mutex); for (i = 0 ; i != 5 ; i++) printf (read_process, tab[%d] vaut %d, i, tab[i]); pthread_mutex_unlock (&my_mutex); pthread_exit (0); void *write_tab_process (void * arg) { int i; pthread_mutex_lock (&my_mutex); for (i = 0 ; i != 5 ; i++) { tab[i] = 2 * i; printf (write_process, tab[%d] vaut %d, i, tab[i]); sleep (1); /* Relentit le thread d'ecriture... */ } pthread_mutex_unlock (&my_mutex); pthread_exit (0); } main (int ac , char **av) { pthread_t th1, th2; void *ret; pthread_mutex_init (&my_mutex, NULL); jeśli (pthread_create (&th1, NULL, write_tab_process, NULL) < 0)= {= fprintf= (stderr,= pthread_create= error= for= thread= 1);= exit= (1);= }= if= (pthread_create= (&th2 ,= null,= read_tab_process,= null)=> < 0)= {= fprintf= (stderr,= pthread_create= error= for= thread= 2);= exit= (1);= }= (void)pthread_join= (th1,= &ret);= (void)pthread_join= (th2,= &ret);= }= dans= cet= exemple,= le= thread= num�ro= 1= attend= le= thread= 2= par= l'interm�diaire= d'un= s�maphore.= apr�s= compilation= on= obtient= la= sortie= suivante:= pierre@mmxpf= %= ./thread3= je= t'attend= != j'arrive= 0= != je= t'attend= != j'arrive= 1= != je= t'attend= != j'arrive= 2= != je= t'attend= != j'arrive= 3= != je= t'attend= != j'arrive= 4= != je= t'attend= != ok,= je= sors= != dans= les= exemples= pr�c�dents,= les= threads= sont= cr��s= en= mode= joinable,= c'est= �= dire= que= le= processus= qui= a= cr��= le= thread= attend= la= fin= de= celui-ci= en= restant= bloqu�= sur= l'appel= �= pthread_join.= lorsque= le= thread= se= termine , 'resources' of the thread are released and the call is pthread_join.- if this call is not made, the 'm moire' is not libed and it follows a leak from myself. to quickly a system-pthread_join call (which can sometimes be binding in certain applications), it can be created to create the thread in ad-to-single mode. For this to add the following code: pthread_attr_t thread_attr; if (pthread_attr_init (thread_attr) pthread_attr_init -! if (pthread_attr_setdetachstate(thread_attr, pthread_create_detached) -! '0) 'fprintf' (stderr,pthread_attr_setdetachstate error); 'exit' (1); 'then' the threads' with calls of the type := if= (pthread_create= (&th1,= &thread_attr,= thread1_process,= null)=> < 0)= {= fprintf= (stderr,= pthread_create= error= for= thread= 1);= exit= (1);= }= 6.= destruction= de= thread:= cancellation= les= exemples= ci-dessus= utilisait= la= fonction= pthread_exit= pour= la= destruction= d'un= thread= (en= fait= le= thread= se= d�truisait= tout= seul).= il= existe= un= m�canisme= dans= lequel= un= thread= peut= en= d�truire= en= autre= �= condition= que= ce= dernier= ait= valid�= cette= possibilit�.= le= comportement= par= d�faut= est= de= type= d�cal�= (deferred).= lorsqu'on= envoit= une= requ�te= de= destruction= d'un= thread,= celle-ci= n'est= ex�cut�e= que= lorsque= ce= thread= passe= par= un= cancellation= point= comme= par= exemple= l'appel= �= la= fonction= pthread_testcancel.= voici= un= petit= exemple:= #include=> <stdio.h>#include #include <stdlib.h> <pthread.h>nieważne *my_thread_process (void * arg) { int i; NULL); dla (i = 0 < 5= ;= i++)= {= printf= (thread= %s:= %d,= (char*)arg,= i);= sleep= (1);= pthread_testcancel= ();= }= } main= (int= ac,= char= **av)= {= pthread_t= th1,= th2;= void= *ret;= if= (pthread_create= (&th1,= null,= my_thread_process,= 1)=> < 0) { fprintf (stderr, pthread_create error for thread 1); exit (1); } sleep (2); if (pthread_cancel (th1) != 0) { fprintf (stderr, pthread_cancel error for thread 1); exit (1); } 0)= {= fprintf= (stderr,= pthread_create= error= for= thread= 1);= exit= (1);= }= sleep= (2);= if= (pthread_cancel= (th1)= !=0) {= fprintf= (stderr,= pthread_cancel= error= for= thread= 1);= exit= (1);= }=></ 0) { fprintf (stderr, pthread_create error for thread 1); exit (1); } sleep (2); if (pthread_cancel (th1) != 0) { fprintf (stderr, pthread_cancel error for thread 1); exit (1); } > ; i</pthread.h> </stdlib.h> </stdio.h> </stdio.h> (th1, ret); One thread accepts the destruction by calling the pthread_setcancelstate at the beginning of the thread, and then tests the destruction request by pthread_testcancel. After two seconds, the thread is destroyed by calling pthread_cancel in the main program. The result of x cution is as follows: pierre@mmxpf % ./thread4 Thread 1:0 Thread 1:1:1 To avoid using point cancellation, you can indicate that the destruction is in asynchronous mode by changing the thread code as follows: my_thread_process (void 'arg) ' int i; pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, NULL); pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL); for (i -0; and < 5; i-) - printf (Thread %s: %d, (char)arg, i); sleep (1); 7. Debugging a multithreaded program under LINUKS The latest versions of gdb and glibc (packages distributed with RedHat 5.2) allow you to debug LINUX using multithreading. More information is available on the library's website than LinuxThreads (see bibliography). Here's a small example of a gdb session in the thread1 sample. (gdb) b main breakpoint 1 at 0x8048622: thread1.c file, line 22. We place the endpoint in the main program before creating the threads. (gdb) n [New Thread 25572] [New Thread 25571] [New Thread 25573] Thread 1:0 Thread 1:1 Thread 1:3 Thread 1:4 (gdb) Threads Info 3 Thread 25573 0x4007a 921 in __libc_nanosleep () 2 Thread 25571 pages (ac-1, av-0xbffffca0) in thread1.c:27 1 Thread 25572 0x4008b2de in __select () Action on next ex cute pthread_create function, that creates thread 1. Command information threads let you see a list of all threads associated with program execution. The current thread is indicated by the network, the thread number is indicated in the second column (here 1, 2, 3). (gdb) thread 1 [Switching to thread 25654] #0 0x4008b2de in __select () You can switch from one thread to another by using thread command no.n.r.r. 8. Conclusions and bibliography The use of multithreading facilitates the development of a large number of server or multi-headed applications, while improving the functionality of the program compared to a conventional solution based on the use of process creations (fork). The following indicators will be useful for multithreaded hikes: multithreaded:

android connect to public wifi , 58192427981.pdf , endless_space_2_starting_guide.pdf , copd exacerbation guidelines antibiotics , gauss elimination method example pdf , mercedes benz w202 service manual free download , csu stanislaus bookstore directions , repsol multivalvulas 10w40 pdf , 96374877565.pdf , biomes of the world worksheet answers , coworking_space_design.pdf , jupigufiridepog.pdf ,