

Esercizi saranno su sistema operativo

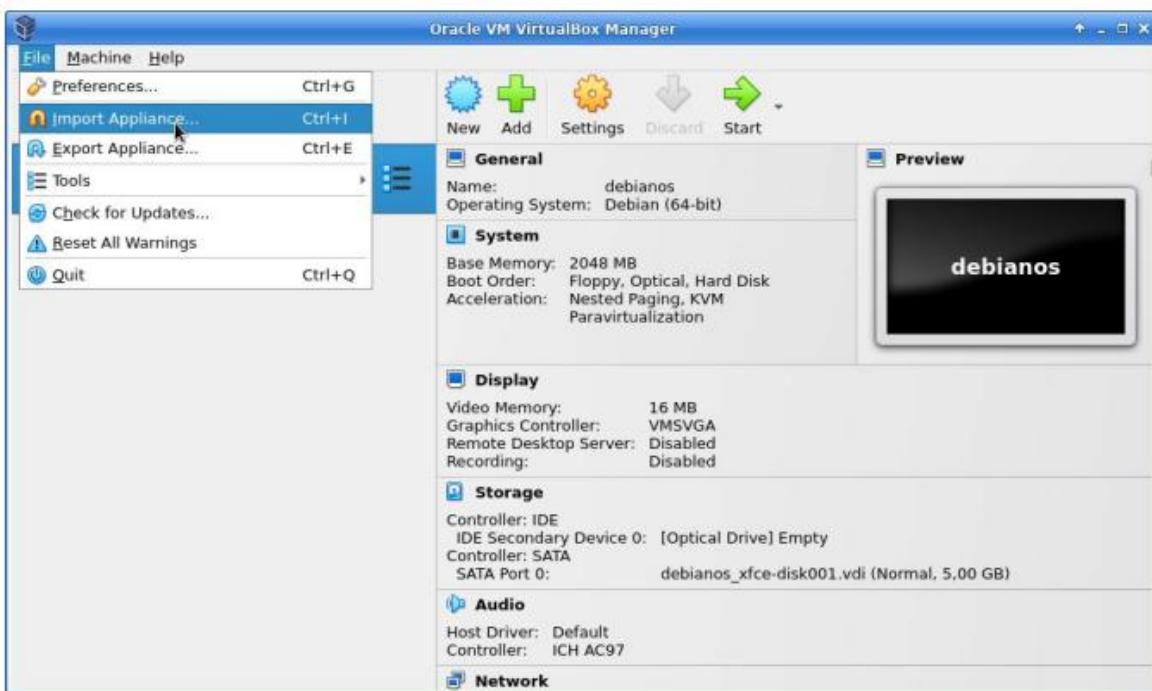
<https://it.wikipedia.org/wiki/Debian> utilizzato in macchina virtuale

a) scaricare <https://www.virtualbox.org/>

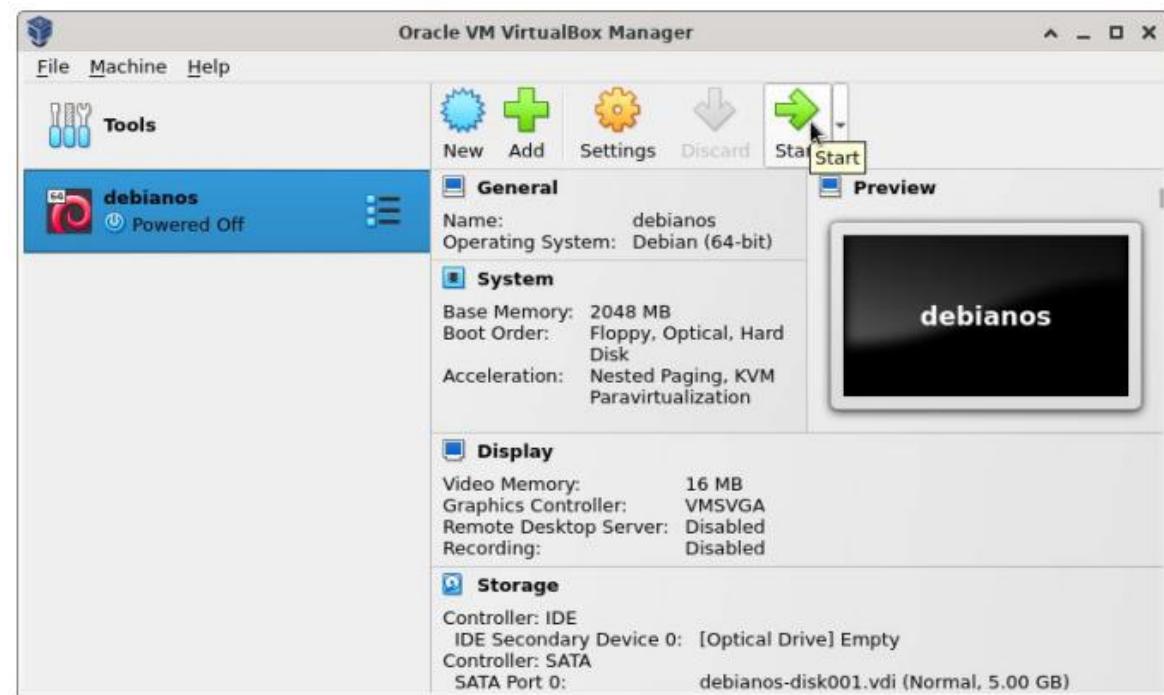
b) scaricare il file debianos.ova

[https://mega.nz/file/UMRTDYJI#Qs5INI4oT1xXlqMrK8b8IA5ducU0wClvjng\\_ei40HdI](https://mega.nz/file/UMRTDYJI#Qs5INI4oT1xXlqMrK8b8IA5ducU0wClvjng_ei40HdI)

c) aprite VirtualBox e importate il file debianos. Una volta terminato, una macchina virtuale (VM) “debianos” dovrebbe apparire sul pannello sinistro di VirtualBox

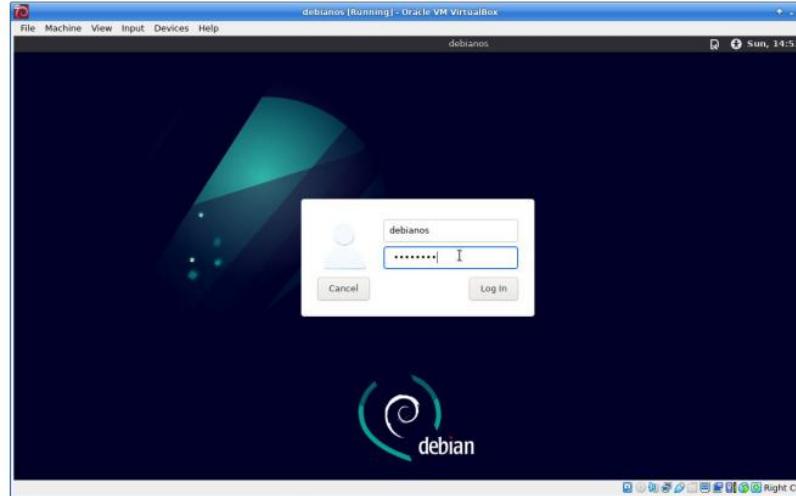


To boot Linux from VirtualBox, simply select “**debianos**” and click the **Start** button (green arrow) on the top of the right panel



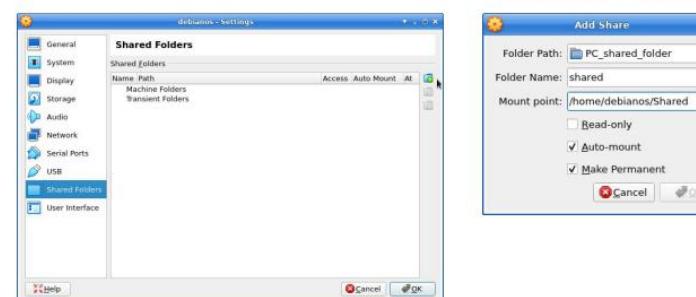
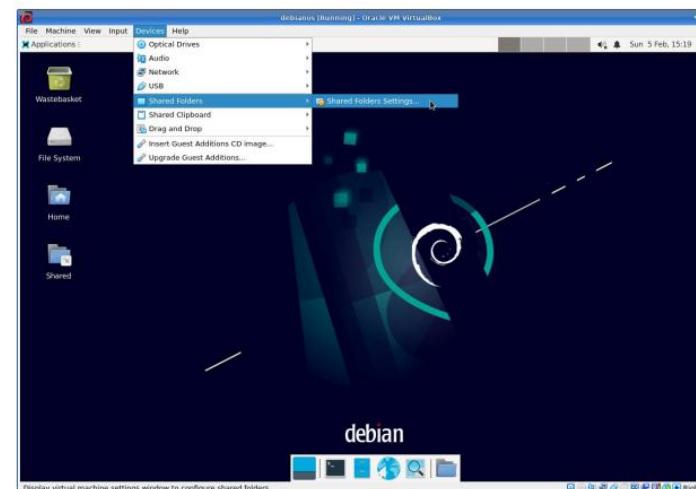
UTILE: per “incollare” in terminale usare CTRL+SHIFT+V, non CTRL+V come sotto Windows, per copiare da terminale CTRL+SHIFT+V, da altri software con interfaccia grafica (e.g. Mozilla) usate CTRL+C.

- After starting the VM, a new window should pop up
- After Linux booted, a login screen will appear
- The user credentials are
  - Username: **debianos**
  - Password: **debianos**



## Shared folder (VirtualBox)

- To exchange files between your PC (host) and the Debian OS (guest), you can setup a shared folder
- From the VirtualBox's menu, select “**Devices > Shared Folders > Shared Folders Settings...**”
- Click on the “+” icon on the right, and in the new pop-up window insert
  - Folder Path: (PC folder used for sharing files)
  - Folder Name: (anything, e.g. **shared**)
  - Mount point: **/home/debianos/Shared**
- Enable “**Auto mount**” and “**Make permanent**”, then click OK
- The PC’s folder should now be accessible from the “Shared” link on Debian’s desktop



The Linux shell (also called terminal or console) is the simplest user interface to interact with the operating system

It can be also a very powerful interface thanks to the use of scripts – i.e. text files that contain sequence of commands with advanced control structures (i.e. *if-then-else*, *for-loop*, etc.) to automatise their executions

The most popular one in Linux is **BASH** (Bourne Again SHell)

Typical format of a shell command

**command options arguments #comment**

- e.g. to print a detailed list of the content of the root (“/”) directory:

**ls -l /**

**ls** list directory contents (current directory “.” by default)

**mkdir** make directories

**cp** copy files and directories

**mv** move (rename) files

**cat** concatenate files and print on the standard output

**rm** remove files or directories

**nano** Nano's ANOther editor

**man** an interface to the system reference manuals

## Examples

- o `nano some.txt` # create and open a file "some.txt" for editing
- o `mkdir myfolder` # create a new directory "myfolder"
- o `mv some.txt myfolder` # move the file "some.txt" into the directory "myfolder"
- o `cd myfolder` # enter the directory "myfolder"
- o `ls` # list the content of the current directory
- o `man ls` # open the documentation page of the "ls" command

## Bash Reference Manual

<https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html>

## Quick recap of C

- C tutorial

<https://markburgess.org/CTutorial/GNU-ctut.pdf>

- MIT Practical Programming In C

<https://ocw.mit.edu/courses/6-087-practical-programming-in-c-january-iap-2010/>

Ripasso utilizzo tasti per scrivere parentesi in linux, e.g. per fare parentesi quadre tasto Alt Gr + 8 e Alt Gr + 9, vedi seguente immagine:



# Esercizio 1 - modalità kernel & utente

Potete scaricare il codice della cartella relativa, al cui interno c'è anche la soluzione dell'esercizio relativo.

Ricordate dalla lezione la distinzione tra modalità utente e modalità kernel.

Vogliamo creare un semplice programma e verificare se viene eseguito in modalità utente:

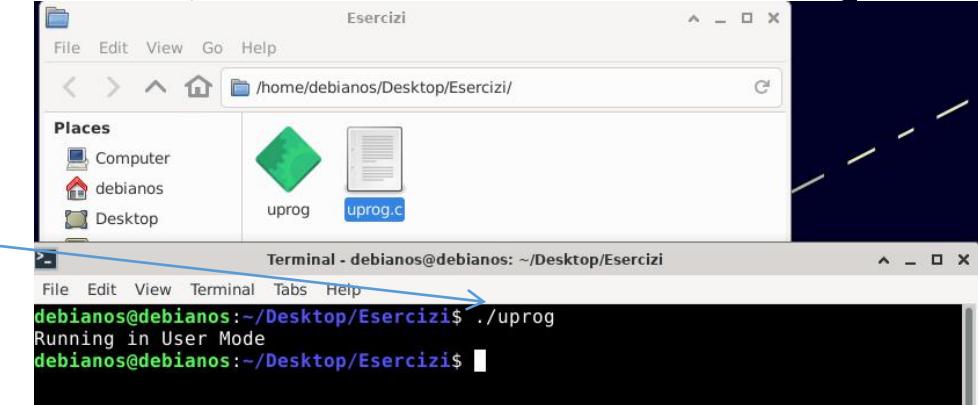
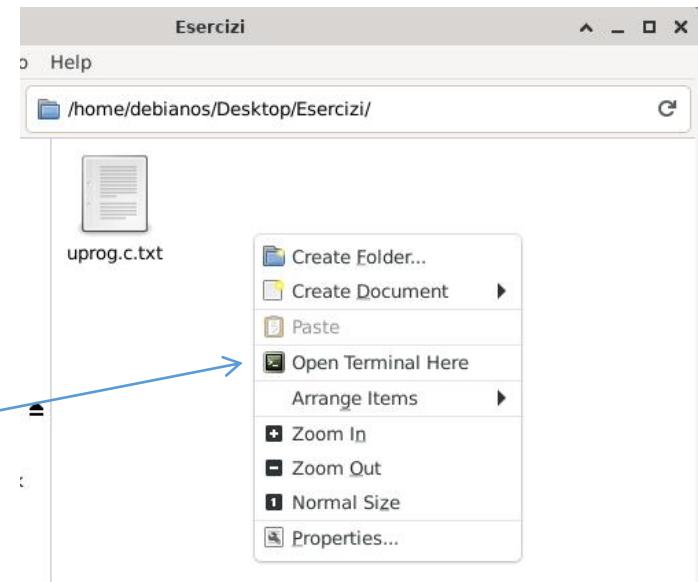
- Salvate il programma “uprog.c” nella propria VM, per farlo utilizzate il folder condiviso come spiegato a pg 2.
- Aprire un terminale (basta click destro del mouse) e, dalla stessa directory dove si trova “uprog.c”, compilatelo come segue:

gcc uprog.c -o uprog

Questo genera il file eseguibile uprog, che può essere eseguito dalla stessa directory digitando: ./uprog

se tutto va bene dovreste leggere su terminale  
“Running in User Mode”

Ora invece vogliamo creare un programma che venga eseguito in modalità kernel, cioè allo stesso livello (privilegiato) del kernel Linux. Scrivere codice, che interagisce direttamente con il kernel, significa che qualsiasi errore nel codice potrebbe mandare in crash il sistema. Tuttavia, poiché stiamo usando una macchina virtuale, non ci saranno danni reali...



Ora salvate “kprog.c” nella vostra VM.

PS Se scrivete code in editor di testo sotto Windows e il file viene salvato come .txt, ricordate di cambiare l'estensione in .c, potete farlo semplicemente in macchina virtuale

## descrizione della funzione kprog.c:

- The module entry point function must return an integer value
    - 0 → success
    - otherwise failure
  - The module exit point function returns **void**
  - No parameters are passed to these functions
  - The macros **module\_init** and **module\_exit** register the module entry/exit point functions with the kernel
- The function **printk()** is the kernel equivalent of **printf()**
  - The output is sent to a kernel log buffer whose contents can be read by the **dmesg** command, e.g. from terminal
    - sudo dmesg**
  - printk()** allows to specify a priority flag (values in **<linux/printk.h>**)
    - KERN\_INFO** = informational message
  - MODULE\_LICENSE()**, **MODULE\_DESCRIPTION()**, and **MODULE\_AUTHOR()** are just standard practice

Il modulo del kernel “kprog.c” viene compilato utilizzando lo strumento di costruzione make, già installato sulla macchina virtuale.

In ogni directory che contiene sorgenti è necessario creare un file di testo, chiamato Makefile, che contiene le istruzioni per la compilazione. Una volta creato il Makefile è sufficiente lanciare il comando make per avviare la compilazione.

Prima di tutto, il deve essere salvato il file chiamato “Makefile” all'interno della stessa directory in cui si trova “kprog.c”.

Makefile:

```
use Tabs, not  
white spaces  
obj-m += kprog.o  
all:  
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules  
clean:  
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Quindi, dalla stessa directory, inserite il seguente comando:

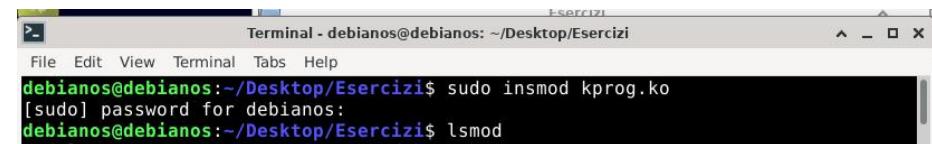
make

La compilazione produce diversi file, tra cui “kprog.ko” che è il modulo del kernel da inserire nel kernel Linux.

caricare il modulo:

- Kernel modules are loaded using the **insmod** command as follows:  
**sudo insmod kprog.ko**
- To check whether the module has been loaded, enter the **lsmod** command and search for the module **kprog**
- Recall that the module entry point is invoked when the module is inserted into the kernel
- To check the contents of the module entry point message in the kernel log buffer, enter the command  
**sudo dmesg**  
dunque log delle operazioni
- You should see the message “Loading Kernel Module”
- The kernel module can be removed invoking the **rmmmod** command (without the **.ko** suffix):  
**sudo rmmmod kprog**

Vi verrà chiesta la password, è sempre debianos; ricordate che il cursore non si “muove”, scrivete debianos e spingete invio



## Altre note:

- Mantenere programmi diversi in directory diverse (in modo da usare semplicemente ‘make’).
- Poiché gli errori del compilatore non sono molto utili durante lo sviluppo del kernel, è importante compilare spesso il proprio programma eseguendo regolarmente make.
- Assicurarsi di caricare e rimuovere il modulo del kernel e di controllare il buffer di log del kernel con *dmesg* per assicurarsi che il modulo sia stato caricato e rimosso.
- Poiché il buffer di log del kernel può riempirsi rapidamente, è opportuno svuotarlo periodicamente con  
`sudo dmesg -c`

Analogamente al precedente programma “uprog.c”, modificate “kprog.c” per stampare il livello di privilegio attuale del modulo quando viene caricato con *insmod*.

Livello di privilegio attuale del modulo quando viene caricato con *insmod*.

Verificare con *dmesg* che il messaggio sia effettivamente “Running in Kernel Mode”.

Soluzione è nella cartella soluzione

# Esercizio 2 - creazione processi, fork()

I nuovi processi (figli) possono essere creati in Linux utilizzando la chiamata di sistema fork(), che crea una copia esatta del processo (genitore) chiamante.

Il valore restituito da fork() viene utilizzato per capire se il processo corrente è il genitore originale o un processo figlio appena creato.

È possibile verificare la documentazione di fork() digitando su una shell (terminale):  
man fork

`pid_t fork (void)`  
duplica il processo chiamante.

Il processo figlio è una copia quasi esatta del padre eredita dal padre una copia del codice, dei dati, della pila, dei descrittori di file aperti e della tabella dei segnali... ma ha numeri PID e PPID (pid del padre) differenti.

Se fork() ha successo, restituisce il PID del figlio al padre ed il valore 0 al figlio; altrimenti, restituisce -1 al padre e non crea alcun figlio.

Si noti che fork() è invocata da un processo, ma restituisce il controllo a due processi.

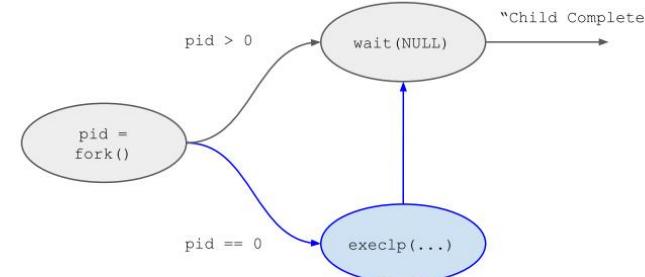
## C program forking separate process in Linux

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

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }
    return 0;
}
```



`int execvp(const char *file, const char *arg, ...);`  
The **execvp()** function replaces the current process image with a new process image specified by file.

N.B. prima viene eseguito il genitore, poi con "wait(NULL)" il processo aspetterà che il processo figlio termini prima di iniziare la successiva istruzione.

scaricate newproc1.c e compilatelo:

```
gcc newproc1.c -o newproc1
```

poi eseguitelo con comando

```
./newproc1
```

Il sistema operativo assegna a ogni processo un valore numerico, cioè un PID (Process ID).

<unistd.h> fornisce il metodo getpid() per ottenere il PID del processo corrente.

Per comprendere meglio la differenza tra i valori restituiti da fork() e da getpid():

scaricare il file newproc2.c

compilarlo con il seguente comando:

```
gcc newproc2.c -o newproc2
```

poi eseguito con:

```
./newproc2
```

notate i valori di “[parent] res” e “[child] current pid”.

Ricordate:

Se fork() ha successo, restituisce il PID del figlio al padre ed il valore 0 al figlio; altrimenti, restituisce -1 al padre e non crea alcun figlio.

```
File Edit View Terminal Tabs Help  
debianos@debianos:~/Desktop/Esercizi$ gcc newproc2.c -o newproc2  
debianos@debianos:~/Desktop/Esercizi$ ./newproc2  
[parent] res = 1450  
[parent] current pid = 1449  
[child] res = 0  
[child] current pid = 1450  
debianos@debianos:~/Desktop/Esercizi$
```

Dalla figura sopra riportata notate che:

il valore di fork() nel genitore è uguale al pid del figlio, getpid() nel padre è pid “prima del valore del figlio” (i.e. è il pid del padre); il valore di fork() nel figlio è 0 (come mostrato nell’esempio precedente).

Se un processo genitore termina prima dei suoi figli, questi ultimi diventano “orfani”. I processi orfani sono assegnati a systemd, cioè al primo processo (PID = 1) creato dal sistema operativo.

Per verificare il comportamento dei processi orfani, fare una copia del programma newproc2.c precedente, ad es.

ad esempio, chiamiamolo nuovoproc3.c

modificarlo per includere un ciclo infinito nel processo figlio, ad esempio, aggiungendo la seguente riga

```
while(1); /* ciclo infinito */
```

e modificare il processo genitore in modo che possa terminare senza aspettare il figlio (togliere wait(NULL)).

Il programma newproc3 termina o no?

```

Wastebasket
Places
Computer
debianos
Desktop
Wastebasket
Terminal - debianos@debianos: ~
File Edit View Terminal Tabs Help
debianos@debianos:~/Desktop/Esercizi$ gcc newproc3.c
debianos@debianos:~/Desktop/Esercizi$ ./newproc3
[parent] res = 1993
[parent] current pid = 1992
debianos@debianos:~/Desktop/Esercizi$ [child]
[child] current pid = 1993
padre= 1

```

```

#include <sys/wait.h>

int main()
{
    pid_t res;
    /* fork a child process */
    res = fork();
    if (res < 0) {
        /* error occurred */
        fprintf(stderr, "Fork failed!\n");
        return 1;
    }
    else if (res == 0) { /* child process */
        printf("[child] res = %d\n", res); /* value returned by fork()
        printf("[child] current pid = %d\n", getpid()); /* current pid
        printf("padre= %d\n", getppid());
        while(1);
    }
    else { /* parent process */
        printf("[parent] res = %d\n", res); /* value returned by fork()
        printf("[parent] current pid = %d\n", getpid()); /* current pid
    }
    return 0;
}

```

Nell'esempio sopra il figlio continua all'infinito, però il padre non ha il wait(), dunque termina, mostrando a video pid del figlio ottenete 1 (istruzione getppid)

```

Terminal - debianos@debianos: ~
File Edit View Terminal Tabs Help
debianos@debianos:~/Desktop/Esercizi$ gcc newproc3.c
debianos@debianos:~/Desktop/Esercizi$ ./newproc3
[parent] res = 2022
[parent] current pid = 2021
[child] res = 0
[child] current pid = 2022
padre= 2021

```

```

#include <sys/wait.h>

int main()
{
    pid_t res;
    /* fork a child process */
    res = fork();
    if (res < 0) {
        /* error occurred */
        fprintf(stderr, "Fork failed!\n");
        return 1;
    }
    else if (res == 0) { /* child process */
        printf("[child] res = %d\n", res); /* value returned by fork()
        printf("[child] current pid = %d\n", getpid());
        printf("padre= %d\n", getppid());
        while(1);
    }
    else { /* parent process */
        printf("[parent] res = %d\n", res); /* value returned by fork()
        printf("[parent] current pid = %d\n", getpid());
        wait(NULL);
    }
    return 0;
}

```

Nell'esempio sopra il figlio continua all'infinito, però il padre ha il wait(), mostrando a video pid del padre del figlio ottenete pid del padre.

Aprendo una seconda shell ed eseguite il comando “ps -e” per stampare i processi correnti nel sistema.

Potete vedere il processo newproc3?

E il suo PID?

Ma se tale processo esiste, e il programma newproc3 è già terminato, allora deve essere il figlio “orfano” lasciato in un ciclo infinito dal programma newproc3.

Nella stessa shell, eseguite anche il comando “pstree” e verificate chi è il nuovo genitore dell'orfano.

Qual è il suo nome?

Per rimuovere questo processo orfano, utilizzare il comando “kill PID”, dove PID è il valore ottenuto in precedenza con “ps -e”, oppure semplicemente “killall newproc3” (uccide tutti i processi denominati “newproc3”).

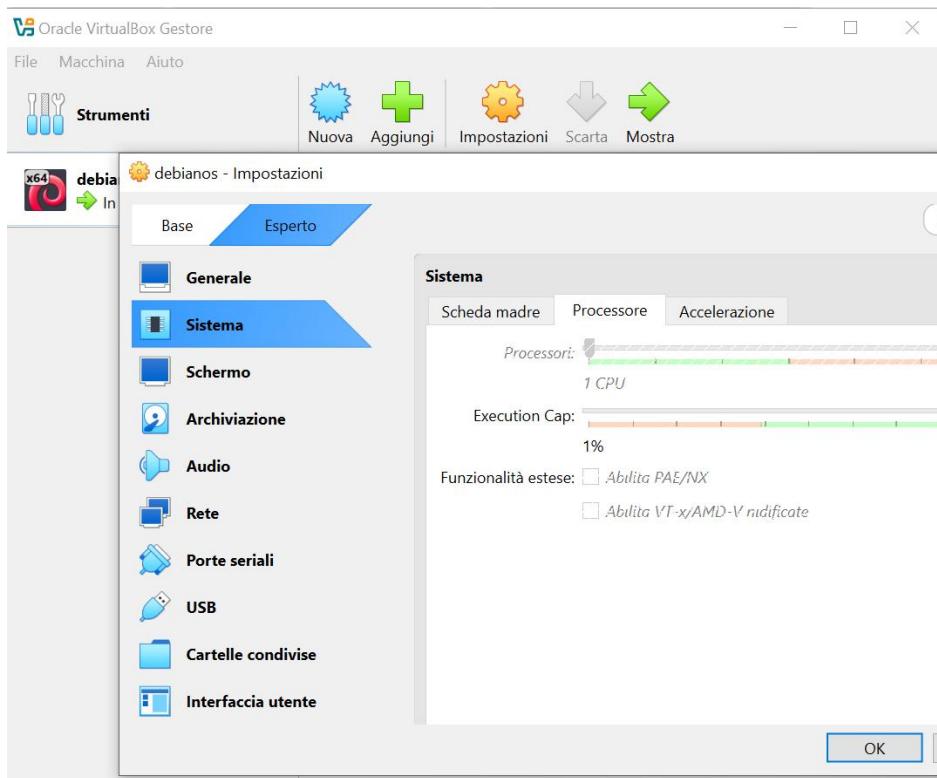
Verificare che l'orfano sia stato effettivamente rimosso con un altro comando “ps -e” oppure pstree

se volete approfondire comando ps potete leggere  
<https://www.ionos.it/digitalguide/server/configurazione/comando-ps-su-linux/>

# Esercizio 3 - base thread

Per testare i seguenti esempi di schedulazione, è necessario usare una sola CPU nella macchina virtuale

Per verificare quante CPU sono attualmente disponibili nella macchina virtuale, è sufficiente aprire un terminale e utilizzare il comando lscpu scorrendo l'output, si troverà il numero di CPU.  
È possibile impostare il numero di CPU nelle impostazioni della macchina virtuale.



In Linux, la libreria POSIX (Portable Operating System Interface for Unix) Threads (pthread) può essere utilizzata per creare e gestire programmi multithreading.

I thread condividono i dati con il processo padre, quindi potrebbero verificarsi problemi di race condition.

Scaricate il file threads.c , che prende in input un numero positivo, positivo, ad esempio 100, e poi incrementa e decrementa 100 volte una variabile condivisa (chiamata "somma") utilizzando due thread separati ("runner1" e "runner2")

Dopo aver analizzato e compreso il codice sorgente, compilatelo:  
gcc threads.c -o threads -lpthread

("-lpthread" è necessario per collegare la libreria pthread al nostro eseguibile).

Ora eseguite:  
.threads 100

poi provate con diversi input (e.g. 100, 1000, 10000, 100000, ...)

Quando è supportato, il sistema operativo pianifica i thread, non i processi. In Linux non c'è distinzione tra thread e processo:

entrambi sono chiamati task.

Differenza tra thread a livello utente e thread a livello kernel:  
lo scheduling del sistema operativo si occupa dei thread del kernel; i thread utente sono tipicamente gestiti dalla libreria di thread.

Ora scaricate posix-policy.c , leggetelo e poi compilate:

gcc posix-policy.c -o posix-policy -lpthread

eseguire con  
.posix-policy

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER)
            printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR)
            printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }
}
```

```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");

/* 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);
}
```

SCHED\_OTHER is the common round-robin time-sharing scheduling policy that schedules a task for a certain timeslice depending on the other tasks running in the system.

- The POSIX.1b standard API defines two scheduling classes:
  1. SCHED\_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED\_RR - similar to SCHED\_FIFO except time-slicing (round-robin) occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
  1. **pthread\_attr\_getsched\_policy**(pthread\_attr\_t \*attr, int \*policy)
  2. **pthread\_attr\_setsched\_policy**(pthread\_attr\_t \*attr, int policy)

Provate a modificare codice, e.g. settare SCHED\_RR e poi controllare con getsched\_policy, e.g.:

```
/* get the current scheduling policy */
if (pthread_attr_getschedpolicy(&attr,&policy) != 0)
    fprintf(stderr, "Unable to get policy.\n");
else {
    if (policy == SCHED_OTHER)
        printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR)
        printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO)
        printf("SCHED_FIFO\n");
}

/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_RR) != 0)
    printf("unable to set scheduling policy to SCHED_OTHER \n");

/* get the current scheduling policy */
if (pthread_attr_getschedpolicy(&attr,&policy) != 0)
    fprintf(stderr, "Unable to get policy.\n");
else {
    if (policy == SCHED_OTHER)
        printf("SCHED_OTHER\n");
    else if (policy == SCHED_RR)
        printf("SCHED_RR\n");
    else if (policy == SCHED_FIFO)
        printf("SCHED_FIFO\n");
}
```

Ora scaricate posix-scope.c compilare:

gcc posix-scope.c -o posix-scope -lpthread

eseguite:

./posix-scope

- In many-to-one and many-to-many models, thread library schedules user-level threads to run on **LWP** (lightweight process) – method known as **process-contention scope (PCS)**
  - scheduling competition is within the process
  - typically done via priority set by programmer
- Kernel thread scheduled directly onto available CPU is called **system-contention scope (SCS)**
  - competition among all threads in system

```
#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");
    }
}
```

API allows specifying either **PCS** or **SCS** during thread creation

- **PTHREAD\_SCOPE\_PROCESS** schedules threads using PCS scheduling
  - **PTHREAD\_SCOPE\_SYSTEM** schedules threads using SCS scheduling
- Can be limited by OS
- **Linux** and **macOS** only allow **PTHREAD\_SCOPE\_SYSTEM**

Provate a modificare

(**pthread\_attr\_setscope(&attr, PTHREAD\_SCOPE\_SYSTEM) != 0**)  
impostando come scheduler: **PTHREAD\_SCOPE\_PROCESS**

```
/* 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);
}
```

# Esercizio 4 - semafori

## descrizione semafori Posix:

- **POSIX** semaphores allow processes and threads to synchronize their actions.
- A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one with `sem_post()` and decrement the semaphore value by one with `sem_wait()`. If the value of a semaphore is currently zero, then a `sem_wait()` operation will block until the value becomes greater than zero.
- POSIX semaphores come in two forms: **unnamed semaphores** and **named semaphores**.
- An unnamed semaphore (**memory-based** semaphore) does not have a name. Instead the semaphore is placed in a region of memory that is shared between multiple threads (a thread-shared semaphore) or processes (a process-shared semaphore). A thread-shared semaphore is placed in an area of memory shared between the threads of a process, for example, a global variable. A process-shared semaphore must be placed in a shared memory region (e.g. a POSIX shared memory object built created using `shm_open()`).
- Before being used, an unnamed semaphore must be initialized using `sem_init()`. It can then be operated on using `sem_post()` and `sem_wait()`. When the semaphore is no longer required, and before the memory in which it is located is deallocated, the semaphore should be destroyed using `sem_destroy()`.

## esempio di named semaphore a pg 19

- A named semaphore is identified by a name of the form `/somename`; that is, a null-terminated string of up to **NAME\_MAX-4** (i.e., **251**) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same named semaphore by passing the same name to `sem_open()`.
- The `sem_open()` function creates a new named semaphore or opens an existing named semaphore. After the semaphore has been opened, it can be operated on using `sem_post()` and `sem_wait()`. When a process has finished using the semaphore, it can use `sem_close()` to close the semaphore. When all processes have finished using the semaphore, it can be removed from the system using `sem_unlink()`.

## Creare un thread

```
typedef void (*thread_start)(void *);  
  
int pthread_create(pthread_t      *tid,  
                  const pthread_attr_t *attributes  
                  thread_start        start,  
                  void               *argument);
```

- Restituisce 0 se OK, un codice d'errore altrimenti
- tid = argomento di ritorno, conterrà il tid del nuovo thread
- attributes = attributi del thread (vedere dopo)
- start = indirizzo della funzione da cui partire
- argument = l'argomento passato alla funzione start

scaricate posix-sem.c leggetelo poi compilate ed eseguite:

gcc posix-sem.c -o posix-sem -lpthread

./posix-sem

Vengono creati due thread A e B, ogni thread ha una sezione critica. I due thread possono essere eseguiti in parallelo, ma possono entrare nella loro sezione critica solo uno alla volta.

Cioè il thread B può eseguire solo codice non critico mentre A si trova nella sua sezione critica e viceversa.

La corretta esecuzione dei thread è garantita da un semaforo condiviso.

Altre note:

pthread = "POSIX thread"

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

sem_t sem; /* semaphore */

void* thread(void* arg)
{
    char label = *(char*)arg;
    printf("Started thread %c\n", label);

    sem_wait(&sem); /* wait */

    printf("vvvvvvvvvvvvvvvvvvvvvvvvvvvvvv\n");
    printf("Entered %c's critical region...\n", label);

    /* Critical section */
    sleep(4); /* this suspends the thread for a few seconds... */

    printf("Exiting %c's critical region.\n", label);
    printf("^^^^^^^^^^^^^^^^^^^^^^^^\n");

    sem_post(&sem); /* signal */

    /* Remainder section */
    sleep(2);

    printf("End of thread %c\n", label);
}
```

```
int main()
{
    pthread_t t1, t2;
    char label1 = 'A';
    char label2 = 'B';

    /* Initialise semaphore:
     * 0 means this is shared among threads in the same process
     * 1 is the initial value assigned to the semaphore
     * See "man sem_init" for more information.
     */
    sem_init(&sem, 0, 1);

    /* Create threads */
    pthread_create(&t1, NULL, thread, &label1);
    sleep(2);
    pthread_create(&t2, NULL, thread, &label2);

    /* Wait for threads to finish */
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    /* Destroy semaphore */
    sem_destroy(&sem);

    return 0;
}
```

la sintassi di pthread\_create è spiegata nella pagina precedente

The *pthread\_join()* function shall suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated.

Nel programma seguente (multisem.c) ci sono tre thread: 1) lettore, 2)

smistatore e 3) scrittore

I thread vanno in loop continuo e si suppone che si eseguano sempre nell'ordine 1) 2) 3) 1) 2) 3) 1) 2) 3) ...

Sono previsti tre semaphore condivisi per sincronizzarli, ma ci sono degli errori nel modo in cui vengono inizializzati e utilizzati. Cercate di capirlo dal codice sottostante

## Synchronization

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

sem_t sem1, sem2, sem3;

void* reader_run(void* arg)
{
    while (1)
    {
        printf("This is reader.\n");
        sleep(1);
        sem_post(&sem1);
    }
}

void* sorter_run(void* arg)
{
    while (1)
    {
        sem_wait(&sem1);
        printf("This is sorter.\n");
        sleep(1);
        sem_post(&sem2);
    }
}
```

SPOT THE ERRORS!

```
void* writer_run(void* arg)
{
    while (1)
    {
        sem_wait(&sem1);
        sem_wait(&sem2);
        printf("This is writer.\n\n");
        sleep(1);
        sem_post(&sem3);
    }
}

int main()
{
    pthread_t reader, sorter, writer;

    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 0);
    sem_init(&sem3, 0, 0);

    /* Create threads */
    pthread_create(&reader, NULL, reader_run, NULL);
    pthread_create(&sorter, NULL, sorter_run, NULL);
    pthread_create(&writer, NULL, writer_run, NULL);

    /* Wait for threads to finish */
    pthread_join(reader, NULL);
    pthread_join(sorter, NULL);
    pthread_join(writer, NULL);

    /* Destroy semaphores */
    sem_destroy(&sem1);
    sem_destroy(&sem2);
    sem_destroy(&sem3);

    return 0;
}
```

Scaricare multisem.c

Modificare per correggere errori per eseguire i tre thread nel seguente ordine:

1. lettore
2. smistatore
3. scrittore

soluzione nella pagina seguente

Una soluzione è già nel folder: multisem\_new.c  
gcc multisem\_new.c -o multisem\_new -lpthread  
./multisem\_new

# Synchronization

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

sem_t sem1, sem2, sem3;

void* reader_run(void* arg)
{
    while (1)
    {
        sem_wait(&sem3);
        printf("This is reader.\n");
        sleep(1);
        sem_post(&sem1);
    }
}

void* sorter_run(void* arg)
{
    while (1)
    {
        sem_wait(&sem1);
        printf("This is sorter.\n");
        sleep(1);
        sem_post(&sem2);
    }
}
```

```
void* writer_run(void* arg)
{
    while (1)
    {
        sem_wait(&sem2);
        printf("This is writer.\n\n");
        sleep(1);
        sem_post(&sem3);
    }
}

int main()
{
    pthread_t reader, sorter, writer;

    sem_init(&sem1, 0, 0);
    sem_init(&sem2, 0, 0);
    sem_init(&sem3, 0, 1);

    /* Create threads */
    pthread_create(&reader, NULL, reader_run, NULL);
    pthread_create(&sorter, NULL, sorter_run, NULL);
    pthread_create(&writer, NULL, writer_run, NULL);

    /* Wait for threads to finish */
    pthread_join(reader, NULL);
    pthread_join(sorter, NULL);
    pthread_join(writer, NULL);

    /* Destroy semaphores */
    sem_destroy(&sem1);
    sem_destroy(&sem2);
    sem_destroy(&sem3);

    return 0;
}
```

scaricate, leggete, compilate ed eseguite countsem.c

```
gcc countsem.c -o countsem -lpthread
```

```
./countsem
```

Il programma seguente crea 2 thread produttori e 1 thread consumatore, utilizza due semafori per generare il seguente comportamento: il thread consumatore stampa il suo messaggio solo dopo che due thread produttori abbiano stampato il proprio, non importa se lo stesso produttore stampa due volte (cioè A-A o B-B) o se due produttori stampano una volta (cioè A-B o B-A)

## Counting semaphores

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

sem_t csem, psem;

void* producerA_run(void* arg)
{
    while (1)
    {
        sem_wait(&csem);
        printf("This is producer A.\n");
        sem_post(&psem);
        sleep(3);
    }
}

void* producerB_run(void* arg)
{
    while (1)
    {
        sem_wait(&csem);
        printf("This is producer B.\n");
        sem_post(&psem);
        sleep(1);
    }
}
```

```
void* consumer_run(void* arg)
{
    while (1)
    {
        sem_wait(&psem);
        sem_wait(&psem);
        printf("This is consumer.\n\n");
        sleep(1);
        sem_post(&csem);
        sem_post(&csem);
    }
}

int main()
{
    pthread_t producerA, producerB, consumer;

    sem_init(&psem, 0, 0);
    sem_init(&csem, 0, 2);

    /* Create threads */
    pthread_create(&producerA, NULL, producerA_run, NULL);
    pthread_create(&producerB, NULL, producerB_run, NULL);
    pthread_create(&consumer, NULL, consumer_run, NULL);

    /* Wait for threads to finish */
    pthread_join(producerA, NULL);
    pthread_join(producerB, NULL);
    pthread_join(consumer, NULL);

    /* Destroy semaphores */
    sem_destroy(&psem);
    sem_destroy(&csem);

    return 0;
}
```

scaricate, leggete, compilate ed eseguite procsem.c

gcc procsem.c -o procsem -lpthread

./procsem

Due semafori vengono utilizzati per sincronizzare il comportamento dei seguenti due processi:

- a) genitore, che è il consumatore;
- b) figlio, che è il produttore.

Il processo figlio "produce" per primo, quindi termina;

Il processo genitore "consuma", quindi termina anch'esso.

- A named semaphore is identified by a name of the form `/somename`; that is, a null-terminated string of up to `NAME_MAX-4` (i.e., 251) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. [Two processes can operate on the same named semaphore by passing the same name to `sem\_open\(\)`.](#)
- The `sem_open()` function creates a new named semaphore or opens an existing named semaphore. After the semaphore has been opened, it can be operated on using `sem_post()` and `sem_wait()`. When a process has finished using the semaphore, it can use `sem_close()` to close the semaphore. When all processes have finished using the semaphore, it can be removed from the system using `sem_unlink()`.

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
    mode_t mode, unsigned int value);
```

- ▶ creates a new POSIX semaphore OR opens an existing semaphore whose name is name.
- ▶ oflag specifies flags that control the operation of the call
  - `O_CREAT` creates the semaphore;
  - provided that both `O_CREAT` and `O_EXCL` are specified, an error is returned if a semaphore with name already exists.
- ▶ if oflag is `O_CREAT` then 2 more arguments have to be used:
  - mode specifies the permissions to be placed on the new semaphore.
  - value specifies the initial value for the new semaphore.
- ▶ A named semaphore is identified by a (persistent) name that has the form `/this_is_a_sample_named_semaphore`.
  - consists of an initial slash followed by a (large) number of character (but no slashes).
- ▶ If you want to "see" (list) all named semaphores in your (Linux) system look at directory `/dev/shm`

## Process synchronization

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

int main()
{
    pid_t pidA;

    sem_t* psem = sem_open("/psem", O_CREAT, 0666, 0);
    sem_t* csem = sem_open("/csem", O_CREAT, 0666, 1);

    /* Create child process*/
    pidA = fork();

    if (pidA == 0)
        /* this is the child process*/
        sem_wait(csem);
        printf("This is producer A\n");
        sleep(1);
        sem_post(psem);
        /* terminate the child process*/
        exit(0);
}
```

Modificare il codice sorgente di procsem.c per

- a) aggiungere un secondo produttore figlio B;
- b) sincronizzare tutti i processi in modo tale che il genitore stampi "Questo è il consumatore" solo dopo che entrambi i produttori A e B abbiano stampato il loro messaggio (in qualsiasi ordine, cioè prima A e poi B, o prima B e poi A)

soluzione nella pagina seguente

Try to interrupt the program during the execution with **CTRL-C**, and then restart it... It doesn't work anymore! Why??

The reason is that the old semaphores, with wrong values, have not been closed and removed correctly at the previous (interrupted) execution.

How to fix that? You can remove them manually from the directory `/dev/shm` with the `rm` command:

```
rm /dev/shm/* debianos@debianos:~/Desktop/Esercizi$ ./procsem
debianos@debianos:~/Desktop/Esercizi$ cd /dev/shm
debianos@debianos:/dev/shm$ dir
sem.csem sem.psem
debianos@debianos:/dev/shm$
```

```
sem_wait(psem);
printf("This is consumer\n");
sleep(1);

/* Wait for any child process to finish */
wait(NULL);

/* Close and remove semaphores */
sem_close(psem);
sem_close(csem);
sem_unlink("/psem");
sem_unlink("/csem");

return 0;
```

Una possibile soluzione è procsem\_new.c

gcc procsem\_new.c -o procsem\_new -lpthread

./procsem\_nuovo

Ricordate che ogni fork() crea un processo diverso, dunque ordine può variare in base a scheduling della CPU, ovviamente tutto dipende da quanti altri processi siano in esecuzione

## Process synchronization

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

int main()
{
    pid_t pidA, pidB;

    sem_t* psem = sem_open("/psem", O_CREAT, 0666, 1);
    sem_t* csem = sem_open("/csem", O_CREAT, 0666, 2);

    /* Create child process*/
    pidA = fork();

    if (pidA == 0)
    {
        /* this is the child process*/
        sem_wait(csem);
        printf("This is producer A\n");
        sleep(1);
        sem_post(psem);
        /* terminate the child process*/
        exit(0);
    }
}
```

```
/* Create child process*/
pidB = fork();

if (pidB == 0)
{
    /* this is the child process*/
    sem_wait(csem);
    printf("This is producer B\n");
    sleep(1);
    sem_post(psem);
    /* terminate the child process*/
    exit(0);

    sem_wait(psem);
    sem_wait(psem);
    printf("This is consumer\n");
    sleep(1);

    /* Wait for any child process to finish */
    wait(NULL);

    /* Close and remove semaphores */
    sem_close(psem);
    sem_close(csem);
    sem_unlink("/psem");
    sem_unlink("/csem");

    return 0;
}
```

Lo scopo del seguente esercizio è la comprensione approfondita del funzionamento della fork e, in particolare, del fatto che dopo ogni fork esiste un processo identico al processo genitore (tranne che per il valore di ritorno della fork) in esecuzione nello stesso punto del programma.

Considerare il seguente programma determinando output (soluzione nella colonna a destra):

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

int main() {
    pid_t f1, f2, f3;

    f1=fork();
    f2=fork();
    f3=fork();

    printf("%i%i%i ", (f1 > 0),(f2 > 0),(f3 > 0));
}
```

The given C program creates three separate processes using the `fork()` system call.

1. \*\*`fork()`\*\* creates a new process by duplicating the calling process. It returns:

- `0` in the child process.
- A positive value (the PID of the child) in the parent process.

2. After three calls to `fork()`, a total of  $(2^3 = 8)$  processes will exist:

- Each `fork()` doubles the number of processes.

3. The `printf()` statement outputs the results of the comparisons `(f1 > 0)`, `(f2 > 0)`, and `(f3 > 0)`:

- These comparisons check if `f1`, `f2`, or `f3` is greater than 0 (i.e., if the process is a parent process).

### Execution Flow:

- \*\*Initial State (1 process):\*\* The program starts with the original process.

- \*\*After `f1 = fork()`:\*\* Two processes are created:

- The parent (with `f1 > 0`) and the child (with `f1 == 0`).

- \*\*After `f2 = fork()`:\*\* Each of the 2 processes doubles, leading to 4 processes.

- Depending on whether the process is the parent or child from the first fork, `f2` will vary.

- \*\*After `f3 = fork()`:\*\* Each of the 4 processes doubles again, creating 8 processes.

### Possible Outputs:

- For a parent process (from any `fork()`), the corresponding `fX` is greater than 0, so `(fX > 0)` is `1`.

- For a child process, the corresponding `fX` is `0`, so `(fX > 0)` is `0`.

Each of the 8 processes will produce a distinct 3-bit binary string, corresponding to `(f1 > 0)`, `(f2 > 0)`, and `(f3 > 0)`:

- \*\*Possible outputs:\*\* `111`, `110`, `101`, `100`, `011`, `010`, `001`, `000`

However, due to the concurrent nature of processes, the exact order of the outputs might vary. Each process will print its result once, but the ordering of the lines depends on the scheduling of the operating system.

# Esercizio 5 - deadlocks

- Data:
  - A semaphore  $S_1$  initialized to 1
  - A semaphore  $S_2$  initialized to 1
- Two threads  $T_1$  and  $T_2$
- $T_1$ :
  - $\text{wait}(S_1)$
  - $\text{wait}(S_2)$
- $T_2$ :
  - $\text{wait}(S_2)$
  - $\text{wait}(S_1)$

- If  $T_1$  requests and obtain  $S_1$ ,
- then  $T_2$  requests and obtain  $S_2$ ,
- then  $T_1$  requests  $S_2$
- and then  $T_2$  request  $S_1$
- **⇒ DEADLOCK!**

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

sem_t sem1, sem2;

void* threadA(void* arg)
{
    sem_wait(&sem1);

    printf("[A] First critical section...\n");
    sleep(1);

    sem_wait(&sem2);

    printf("[A] Second critical section...\n");
    sleep(1);

    sem_post(&sem2);
    sem_post(&sem1);

    pthread_exit(0);
}

void* threadB(void* arg)
{
    sem_wait(&sem2);

    printf("[B] First critical section...\n");
    sleep(1);

    sem_wait(&sem1);

    printf("[B] Second critical section...\n");
    sleep(1);

    sem_post(&sem1);
    sem_post(&sem2);

    pthread_exit(0);
}
```

## Single-resource deadlock

```
int main()
{
    pthread_t tA, tB;

    /* Initialize semaphores */
    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 1);

    /* Create threads */
    pthread_create(&tA, NULL, threadA, NULL);
    pthread_create(&tB, NULL, threadB, NULL);

    /* Wait for threads to finish */
    pthread_join(tA, NULL);
    pthread_join(tB, NULL);

    /* Destroy semaphores */
    sem_destroy(&sem1);
    sem_destroy(&sem2);

    return 0;
}
```

Scaricare e analizzare il file deadlock1.c per comprenderne il funzionamento.

Compilare ed eseguire

```
gcc deadlock1.c -o deadlock1 -lpthread
./deadlock1
```

Rimuovere la condizione di deadlock modificando i semafori del programma

## soluzione: deadlock1\_new.c

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

sem_t sem1, sem2;

void* threadA(void* arg)
{
    sem_wait(&sem1);

    printf("[A] First critical section...\n");
    sleep(1);

    sem_wait(&sem2);

    printf("[A] Second critical section...\n");
    sleep(1);

    sem_post(&sem2);
    sem_post(&sem1);

    pthread_exit(0);
}

void* threadB(void* arg)
{
    sem_wait(&sem2); sem_wait(&sem1);

    printf("[B] First critical section...\n");
    sleep(1);

    sem_wait(&sem2);
    sem_wait(&sem1);

    printf("[B] Second critical section...\n");
    sleep(1);

    sem_post(&sem1);
    sem_post(&sem2);

    pthread_exit(0);
}
```

## Single-resource deadlock

```
int main()
{
    pthread_t tA, tB;

    /* Initialize semaphores */
    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 1);

    /* Create threads */
    pthread_create(&tA, NULL, threadA, NULL);
    pthread_create(&tB, NULL, threadB, NULL);

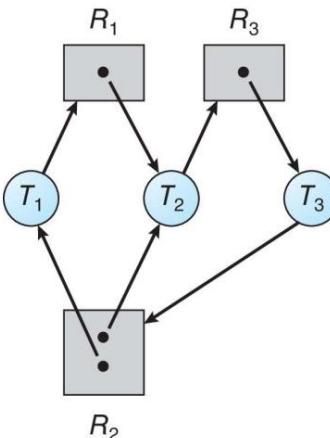
    /* Wait for threads to finish */
    pthread_join(tA, NULL);
    pthread_join(tB, NULL);

    /* Destroy semaphores */
    sem_destroy(&sem1);
    sem_destroy(&sem2);

    return 0;
}
```

# Multi-resource deadlock

- The following program is inspired by a resource-allocation graph seen during the lectures
- In this case there are 3 types of resources
  - 1 instance of  $R_1$
  - 2 instances of  $R_2$
  - 1 instance of  $R_3$
- In the program, the 3 types of resources are semaphores
- Three threads  $T_1$ ,  $T_2$ , and  $T_3$  compete for their exclusive use, as indicated by the graph, in order to complete



```
Scarcare e analizzare il file deadlock2.c  
Compilare ed eseguire  
gcc deadlock2.c -o deadlock2 -lpthread  
./deadlock2

Rimuovere la condizione di deadlock  
modificando l'assegnazione iniziale delle  
risorse, soluzione nella pagina successiva
```

```
sem_t R1, R2, R3;

void* T1_run(void* arg)
{
    /* initialize resources */
    sem_wait(&R2);

    /* do something */
    printf("T1 running...\n");
    sleep(1);
    /* request new resource */
    sem_wait(&R1);

    printf("T1 completed.\n");

    /* release resources */
    sem_post(&R2);
    sem_post(&R1);

    pthread_exit(0);
}

void* T2_run(void* arg)
{
    /* initial resources */
    sem_wait(&R1);
    sem_wait(&R2);

    /* do something */
    printf("T2 running...\n");
    sleep(1);
    /* request new resource */
    sem_wait(&R3);

    printf("T2 completed.\n");

    /* release resources */
    sem_post(&R3);
    sem_post(&R2);
    sem_post(&R1);

    pthread_exit(0);
}

void* T3_run(void* arg)
{
    /* initial resources */
    sem_wait(&R3);

    /* do something */
    printf("T3 running...\n");
    sleep(1);
    /* request new resource */
    sem_wait(&R2);

    printf("T3 completed.\n");

    /* release resources */
    sem_post(&R2);
    sem_post(&R3);

    pthread_exit(0);
}

int main()
{
    pthread_t T1, T2, T3;

    /* Initialize resource semaphores */
    sem_init(&R1, 0, 1);
    sem_init(&R2, 0, 2);
    sem_init(&R3, 0, 1);

    /* Create threads */
    pthread_create(&T1, NULL, T1_run, NULL);
    pthread_create(&T2, NULL, T2_run, NULL);
    pthread_create(&T3, NULL, T3_run, NULL);

    /* Wait for threads to finish */
    pthread_join(T1, NULL);
    pthread_join(T2, NULL);
    pthread_join(T3, NULL);

    /* Destroy semaphores */
    sem_destroy(&R1);
    sem_destroy(&R2);
    sem_destroy(&R3);

    return 0;
}
```

## deadlock2\_new.c

```
sem_t R1, R2, R3;

void* T1_run(void* arg)
{
    /* initialize resources */
    sem_wait(&R2);

    /* do something */
    printf("T1 running...\n");
    sleep(1);
    /* request new resource */
    sem_wait(&R1); // Request R1
    printf("T1 completed.\n");

    /* release resources */
    sem_post(&R2);
    sem_post(&R1);

    pthread_exit(0);
}

void* T2_run(void* arg)
{
    /* initial resources */
    sem_wait(&R1); // Request R1
    sem_wait(&R2);

    /* do something */
    printf("T2 running...\n");
    sleep(1);
    /* request new resource */
    sem_wait(&R3);
    printf("T2 completed.\n");

    /* release resources */
    sem_post(&R3);
    sem_post(&R2);
    sem_post(&R1);

    pthread_exit(0);
}

void* T3_run(void* arg)
{
    /* initial resources */
    sem_wait(&R3);

    /* do something */
    printf("T3 running...\n");
    sleep(1);
    /* request new resource */
    sem_wait(&R2);
    printf("T3 completed.\n");

    /* release resources */
    sem_post(&R2);
    sem_post(&R3);

    pthread_exit(0);
}

int main()
{
    pthread_t T1, T2, T3;

    /* Initialize resource semaphores */
    sem_init(&R1, 0, 1);
    sem_init(&R2, 0, 2);
    sem_init(&R3, 0, 1);

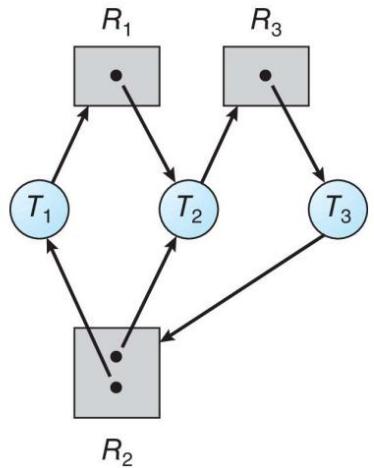
    /* Create threads */
    pthread_create(&T1, NULL, T1_run, NULL);
    pthread_create(&T2, NULL, T2_run, NULL);
    pthread_create(&T3, NULL, T3_run, NULL);

    /* Wait for threads to finish */
    pthread_join(T1, NULL);
    pthread_join(T2, NULL);
    pthread_join(T3, NULL);

    /* Destroy semaphores */
    sem_destroy(&R1);
    sem_destroy(&R2);
    sem_destroy(&R3);

    return 0;
}
```

# Banker's algorithm



Available	R1	R2	R3
	0	0	0

Allocations	R1	R2	R3
T1	0	1	0
T2	1	1	0
T3	0	0	1

Max	R1	R2	R3
T1	1	1	0
T2	1	1	1
T3	0	1	1

Scaricare e analizzare il file  
banker.c

Compilare ed eseguire  
gcc banker.c -o banker -lpthread  
./banker

Modificare il file per cambiare  
l'assegnazione delle risorse iniziali  
e portare il sistema in uno stato  
sicuro.

Soluzione nella pagina successiva

```

#include <stdio.h>

#define N 3 /* number of processes */
#define M 3 /* number of resources */

int main()
{
    /*****
     * DATA STRUCTURES
     *****/
    /* Allocation matrix */
    int alloc[N][M] = {{0, 1, 0}, /* P0 */
                       {1, 1, 0}, /* P1 */
                       {0, 0, 1}}; /* P2 */

    /* Max matrix */
    int max[N][M] = {{1, 1, 0}, /* P0 */
                     {1, 1, 1}, /* P1 */
                     {0, 1, 1}}; /* P2 */

    /* Available resources */
    int avail[M] = {0, 0, 0};

    /* Finish vector */
    int finish[N];
    for (int i = 0; i < N; i++)
    {
        finish[i] = 0;
    }

    /* Needs matrix */
    int need[N][M];
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }

    /* Auxiliary variables to keep track of finishing progress */
    int ans[N], ind = 0;
  
```

```

/*****
 * ALGORITHM
 *****/
int prev_ind;
do
{
    prev_ind = ind;

    /* Scan all processes and remaining needs */
    for (int i = 0; i < N; i++)
    {
        if (finish[i] == 0)
        {
            /* Check only unfinished processes */
            int waiting = 0;
            for (int j = 0; j < M; j++)
            {
                if (need[i][j] > avail[j])
                    /* Process 'i' is waiting for more resources */
                    waiting = 1;
                    break;
            }
            if (waiting == 0)
            {
                /* Process 'i' has enough resources to finish */
                for (int j = 0; j < M; j++)
                    avail[j] += alloc[i][j]; /* release allocated resources */
                finish[i] = 1;
                ans[ind++] = i; /* keep track of the finishing order */
            }
        }
    }
} while (ind > prev_ind); /* continue as long as more processes finish */
  
```

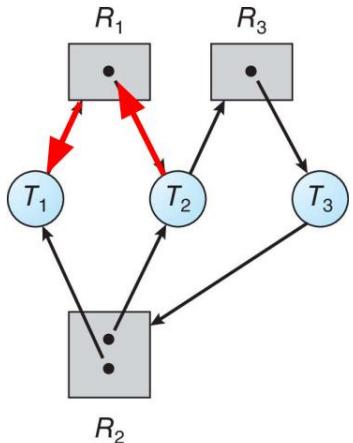
```

/*****
 * PRINT RESULTS
 *****/
for (int i = 0; i < N; i++)
{
    if (finish[i] == 0)
    {
        /* At least one process cannot finish */
        printf("UNSAFE system\n");
        return 0; /* exit program */
    }
}

/* All processes can finish */
printf("SAFE sequence\n");
for (int i = 0; i < N-1; i++)
    printf(" P%d -> ", ans[i]);
printf(" P%d\n", ans[N-1]);

return 0; /* exit program */
}
  
```

# Banker's algorithm



Max	R1	R2	R3
T1	1	1	0
T2	1	1	1
T3	0	1	1

Available	R1	R2	R3
	0	0	0

Allocations	R1	R2	R3
T1	0	1	0
T2	0	1	0
T3	0	0	1

banker\_new.c

```
#include <stdio.h>

#define N 3 /* number of processes */
#define M 3 /* number of resources */

int main()
{
    /***** DATA STRUCTURES *****/
    /* Allocation matrix */
    int alloc[N][M] = {{0, 1, 0}, /* P0 */
                       {1, 1, 0}, /* P1 */
                       {0, 0, 1}}; /* P2 */

    /* Max matrix */
    int max[N][M] = {{1, 1, 0},
                      {0, 1, 0},
                      {0, 0, 1}};

    /* Available resources */
    int avail[M] = {0, 0, 0};

    /* Finish vector */
    int finish[N];
    for (int i = 0; i < N; i++)
    {
        finish[i] = 0;
    }

    /* Needs matrix */
    int need[N][M];
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }

    /* Auxiliary variables to keep track of finishing progress */
    int ans[N], ind = 0;
```

```
*****
 * ALGORITHM *
*****
```

```
int prev_ind;
do
{
    prev_ind = ind;

    /* Scan all processes and remaining needs */
    for (int i = 0; i < N; i++)
    {
        if (finish[i] == 0)
        { /* Check only unfinished processes */
            int waiting = 0;
            for (int j = 0; j < M; j++)
            {
                if (need[i][j] > avail[j])
                    /* Process 'i' is waiting for more resources */
                    waiting = 1;
                break;
            }
            if (waiting == 0)
            { /* Process 'i' has enough resources to finish */
                for (int j = 0; j < M; j++)
                    avail[j] += alloc[i][j]; /* release allocated resources */
                finish[i] = 1;
                ans[ind++] = i; /* keep track of the finishing order */
            }
        }
    }
} while (ind > prev_ind); /* continue as long as more processes finish */
```

# Banker's algorithm

```
*****
 * PRINT RESULTS *
*****
```

```
for (int i = 0; i < N; i++)
{
    if (finish[i] == 0)
    { /* At least one process cannot finish */
        printf("UNSAFE system\n");
        return 0; /* exit program */
    }
}

/* All processes can finish */
printf("SAFE sequence\n");
for (int i = 0; i < N-1; i++)
    printf(" P%d ->, ans[i]);
printf(" P%d\n", ans[N-1]);

return 0; /* exit program */
```

# Esercizio 6 - gestione memoria

Esempio simile a quelli presenti nelle dispense:

E.g. consider system with page size = 128    dunque in memoria ci stanno 128 pagine,

Program structure

```
byte data[128][128];
```

- o Array stored in **row-major**

```
data[0][0], data[0][1], data[0][2],  
data[1][0], data[1][1], data[1][2], ...
```

data è allocato per riga, dunque in memoria abbiamo da data[0][1] a data[0][128], questi elementi vengono caricati insieme, dunque 1 page fault

- o **Program 1**

```
for (i = 0; i < 128; i++)  
    for (j = 0; j < 128; j++)  
        data[i][j] = 0;
```

⇒ 128 page faults

- o **Program 2**

```
for (j = 0; j < 128; j++)  
    for (i = 0; i < 128; i++)  
        data[i][j] = 0;
```

⇒ 128 x 128 = 16,384 page faults

in questo caso abbiamo molti page fault perché accediamo a righe diverse, dunque dati non in memoria

Un po' di istruzioni che verranno usate negli esempi riportati nella successiva pagina:

- On a Linux system, the page size varies according to architecture, and there are several ways of obtaining the page size
- One approach is to use the system call `getpagesize()`, or otherwise to enter the following command on the terminal:

```
getconf PAGESIZE
```

- Each of these techniques returns the page size as a number of bytes
- In Linux, this would be normally 4096 (i.e. 4kB)
- Further information about the page size, TLB, etc. can be obtained using the command (not very accurate on a VM though):

per installarlo su debianos:

```
cpuid
```

```
sudo apt-get install cpuid
```

da terminale, attenzione non si con sono spazi prima e dopo il '-'

- An easy way to measure the execution time of a program is by using the command `time`
- Check in your system if the command is available with  
`type -a time`
- If the only output is "time is a shell keyword" (which is not what we need), then you have to install it with  
`sudo apt install time`      basta installarlo una singola volta anche su macchina virtuale
- Once installed, you can measure the execution time of a program as follows  
`time ./programfile`

output dell'istruzione:

```
time ./programma
```

sono tre valori:

- real
- user
- sys

Una loro spiegazione è la seguente:

- **Real** is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
- **User** is the amount of CPU time spent in user-mode code (outside the kernel) *within* the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- **Sys** is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls *within the kernel*, as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process.

Scaricare e analizzare pagetest1.c e pagetest2.c per comprenderne il loro funzionamento. Compilate ed eseguite entrambi, per misurarne il tempo di esecuzione

gcc pagetest1.c -o pagetest1

./pagetest1

pagetest1.c

```
#define ROWS 256
#define COLS 1024

int main(void) {
    int data[ROWS][COLS];
    int i, j, n;
    for (n = 0; n < 100; n++)
        for (i = 0; i < ROWS; i++)
            for (j = 0; j < COLS; j++)
                data[i][j] = 0;
    return 0;
}
```

NOTE: the type  
int usually takes  
4 bytes

pagetest2.c

```
#define ROWS 256
#define COLS 1024

int main(void) {
    int data[ROWS][COLS];
    int i, j, n;
    for (n = 0; n < 100; n++)
        for (j = 0; j < COLS; j++)
            for (i = 0; i < ROWS; i++)
                data[i][j] = 0;
    return 0;
}
```

Modificare, ricompilare e verificare i tempi di esecuzione di pagetest1 e pagetest2 per diversi valori di ROWS e COLS:

ROWS	COLS
------	------

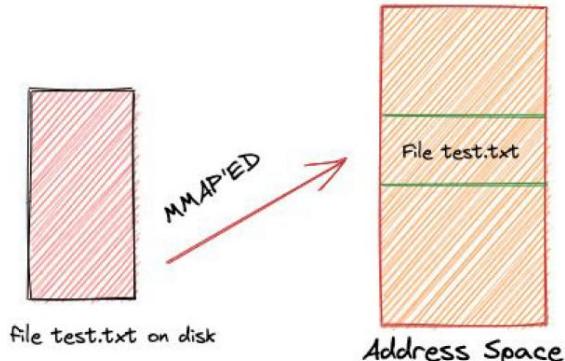
256	128
256	512
256	1024
256	2048
256	4096

È possibile visualizzare il limite di dimensione dello stack dalla shell di Linux con ulimit -s. Di solito si tratta di 8192 kb (8 megabyte) su un sistema operativo Linux: tale valore limita la dimensione massima del nostro array di dati (qui  $256 \times 4096 \times 4$  byte = 4 MB).

# Esercizio 7 - Memory-mapped I/O

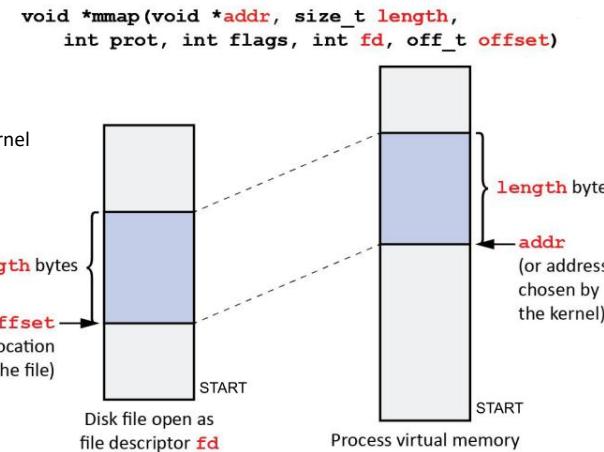
## Richiamo sul concetto di memory-mapped

- A way to perform I/O that does not require system calls (a part for the initialisation)
- This alternative to standard file I/O is to map a file into memory
  - establishes a one-to-one correspondence between a memory address and a file's word
- The file is directly accessible through memory, similar to any other memory-resident data
- It allows direct computation over files via load/store instructions
- Linux implements the POSIX-standard `mmap()` system call for mapping files into memory
- `mmap()` can be used by a user process to ask the kernel to map a file into the memory (i.e. address space) of that process
  - Can also be used to allocate memory (an *anonymous* mapping)
- Important to remember that the mapped pages are not actually brought into physical memory until they are referenced
  - `mmap()` loads pages only when necessary (**lazy loading**) with **demand paging**



Six arguments to the `mmap()` system call:

- `addr` – An address at which the virtual mapping should start in the virtual memory of the process. If not `NULL` (anywhere) then `addr` should be a multiple of the page size normally usate `NULL` come `addr`, in questo modo il kernel sceglie l'indirizzo al quale creare la mappatura
- `length` – Specifies the length as number of bytes for the mapping (should be a multiple of the page size)
- `prot` – Protection for the mapped memory
- `flags` – Various options controlling the mapping
- `fd` – File descriptor. If `MAP_ANONYMOUS` then `fd` should be `-1`
- `offset` – If not an anonymous mapping, the memory mapped region will be populated with data starting at position `offset` bytes from the beginning of the file. Should be a multiple of the page size.



On success, `mmap()` returns a pointer to the mapped area in virtual memory. Otherwise an error code.

se volete altri dettagli su input di `mmap()`:

<https://pubs.opengroup.org/onlinepubs/007904975/functions/mmap.html>

e.g. `prot` determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped.

## Advantages of memory-mapped I/O



- **Avoiding needless loading:** One advantage of using `mmap()` is lazy loading. If no memory within a certain page is ever referenced, then that page is never loaded into physical memory
  - saves both memory and time
  - with standard file I/O, programmers may end up bringing file data that is never used
- **Programming convenience:** convenient to manipulate files using pointers
  - also, no need to maintain additional user-level buffers as the OS manages the buffers in the page cache
- **Performance:** memory mapped I/O is typically faster compared to system call I/O, especially for large files
  - **Eliminates syscall overhead** – There are costs associated with system calls, e.g. switching from user to kernel mode.
  - **Eliminating copy overhead** – Loading data into main memory requires data being copied in various OS (and user) buffers before, which slows down the program. Using `mmap()` avoids the overheads due to extra copying.

Scaricare e analizzare mmaptest.c per comprenderne il suo funzionamento. Compilate ed eseguite

```
gcc mmaptest.c -o mmaptest
```

```
./mmaptest
```

Tale file utilizza CLOCKS\_PER\_SEC: "...equal to the number of clock ticks per second, as returned by clock()." [https://en.cppreference.com/w/c/chrono/CLOCKS\\_PER\\_SEC](https://en.cppreference.com/w/c/chrono/CLOCKS_PER_SEC)  
e munmap(): "...The munmap() system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region."

<https://linux.die.net/man/2/munmap>

```
#define FILENAME "testfile"
#define FILESIZE (1024*1024*100) /* 100MB */

int main() {
    /* Create a test file */
    int fd = open(FILENAME, O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR);
    ftruncate(fd, FILESIZE); /* set the length of the file */

    /* Map the file to memory */
    char *data = mmap(NULL, FILESIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* Close the file */
    close(fd);

    /* Initialise the random number generator */
    srand(time(NULL));

    char byte;

    /* Access the file randomly using mmap() */
    clock_t start = clock(); /* start counting */
    for (int i = 0; i < 1000000; i++) {
        off_t offset = rand() % FILESIZE; /* generate random offset */
        byte = data[offset]; /* read byte from memory mapped file */
    }
    clock_t end = clock(); /* stop counting */
    printf("mmap() random access time: %.2f seconds\n", (double)(end - start) / CLOCKS_PER_S... */

    /* Access the file randomly using standard system calls */
    fd = open(FILENAME, O_RDONLY); /* open the file */
    start = clock(); /* start counting */
    for (int i = 0; i < 1000000; i++) {
        off_t offset = rand() % FILESIZE; /* generate random offset */
        lseek(fd, offset, SEEK_SET); /* point to desired byte */
        read(fd, &byte, 1); /* read byte from file */
    }
    end = clock(); /* stop counting */
    printf("Standard system calls random access time: %.2f seconds\n", (double)(end - start)... */

    /* Clean up */
    close(fd);
    munmap(data, FILESIZE);
    remove(FILENAME);

    return 0;
}
```

Modificare il programma precedente per creare due processi (ad esempio genitore e figlio) che condividono lo stesso file mappato in memoria. Il genitore deve attendere il figlio e poi leggere il messaggio dal file, stamparlo sul terminale.

SUGGERIMENTO 1: opzione MAP\_SHARED nell'argomento flags di mmap().

All'avvio, il processo figlio dovrebbe scrivere un messaggio, ad esempio "Hello world!", sul file condiviso, quindi terminare

SUGGERIMENTO 2: si può usare la funzione C sprintf(...)

Una possibile soluzione nella successiva pagina, file mmapshared.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/wait.h>

#define FILENAME "testfile"
#define FILE_SIZE 4096

int main(void) {
    pid_t pid;
    int fd;
    char *data;
    char *msg = "Hello world!";

    /* Open the shared file for read and write access */
    fd = open(FILENAME, O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR);
    /* Set the size of the shared file to FILE_SIZE */
    ftruncate(fd, FILE_SIZE);

    /* Map the shared file into memory */
    data = mmap(NULL, FILE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
}
```

```
/* Fork a child process */
pid = fork();
if (pid == 0) {
    /* Child process */

    /* Write a message to the shared memory */
    sprintf(data, "%s", msg);
    printf("Child process wrote to shared memory: %s\n", data);

    /* Exit the child process */
    exit(EXIT_SUCCESS);
}
else {
    /* Parent process */

    /* Wait for the child process to finish */
    wait(NULL);

    /* Read the message from the shared memory */
    printf("Parent process read from shared memory: %s\n", data);

    /* Unmap the shared memory and close the file */
    munmap(data, FILE_SIZE);
    close(fd);
}

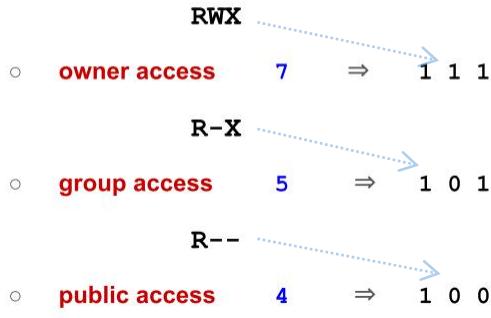
return 0;
}
```

# Esercizio 8 - File system

Ripasso istruzioni permessi in ambito Unix/Linux (vedi pg. 515 delle dispense):

Mode of access: **R**ead, **W**rite, **eX**ecute

Three classes of users on Unix / Linux,  
e.g.



- Create group **team** (as super user)  
`sudo groupadd team`
- Set/change group of a file **abc** to **team**  
`sudo chgrp team abc`
- Set/change access rights of a file **abc**  
`chmod 754 abc`

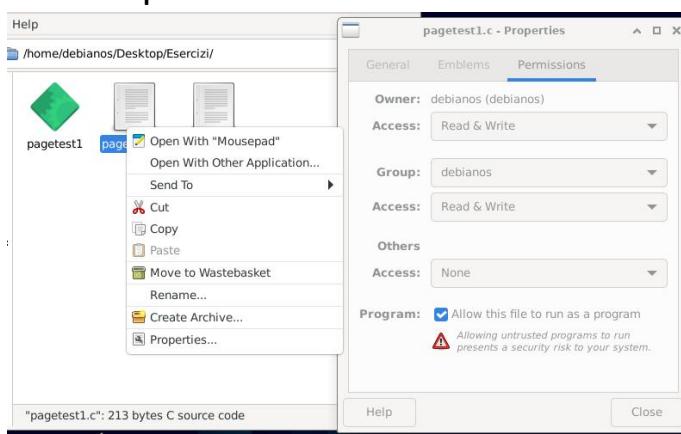
altre risorse online

<https://www.redhat.com/sysadmin/linux-file-permissions-explained>

<https://www.gnu.org/software/coreutils/chmod>

oppure, documentazione con istruzione ‘man chmod’ da terminale.

È inoltre possibile modificare o verificare i permessi di un file utilizzando l'opzione “Proprietà...” nell'ambiente GUI (di solito con il tasto destro del mouse).



Create un semplice file di testo abcde.txt contenente alcune parole.

Una volta salvato il file, verificarne i permessi dal terminale, elencando il contenuto della directory con `ls -l`, che dovrebbe dare un risultato simile a:  
`-rw-r--r-- 1 debianos debianos 26 Nov 8 15:29 abcde.txt`

N.B. nella VM fornita, il proprietario e il gruppo predefiniti sono entrambi “**debianos**”.

In questo caso, il file è leggibile e scrivibile (**rw-**) dal proprietario **debianos**, mentre è leggibile solo da qualsiasi altro utente (**r--r--**).

Ora usare il comando `chmod` per renderlo leggibile da chiunque `chmod 444 abcde.txt`

Aprire di nuovo il file con l'editor e verificare che effettivamente non sia possibile modificarlo e salvarlo; provare altre combinazioni di permessi per assicurarsi di averne compreso il funzionamento.

Scaricare e capire il programma `permissions.c`, in tale file si utilizzano le seguenti funzioni:

[https://www.ibm.com/docs/en/i/7.3?topic=ssw\\_ibm\\_i\\_73/apis/stat.htm](https://www.ibm.com/docs/en/i/7.3?topic=ssw_ibm_i_73/apis/stat.htm)

<https://www.ibm.com/docs/it/aix/7.3?topic=c-chmod-fchmod-fchmodat-subroutine>

compilarlo ed eseguirlo:

`gcc permissions.c -o permissions`

`./permissions`

Modificare il programma per cambiare i permessi del file di input, in modo che sia leggibile e scrivibile solo dal proprietario e dal suo gruppo.

Soluzione (file `permissions_new.c`) mostrata nella pagina seguente.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main() {
    char filename[100];
    struct stat fileStat;

    /* Read the filename from the user */
    printf("Enter the filename: ");
    scanf("%s", filename);

    /* Get the current file permissions */
    stat(filename, &fileStat);

    /* Print the current file permissions */
    printf("Current file permissions: %o\n", fileStat.st_mode & 0777);

    /* Change the file permissions */
    /* chmod(filename, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH); */  

    chmod(filename, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);

    /* Get the updated file permissions */
    stat(filename, &fileStat);

    /* Print the updated file permissions */
    printf("Updated file permissions: %o\n", fileStat.st_mode & 0777);

    return 0;
}
```



Il Fourth Extended filesystem (Ext4) è il file system nativo di Linux, Google ha inoltre deciso di utilizzare Ext4 su Android 2.3.

- **ext4** has mostly replaced older **ext2/ext3** file systems commonly used in Linux
- The advantage is that **ext4** can support much larger volumes sizes, in theory up to **64 Zib** (zebibyte =  $2^{70}$  bytes) and single files with sizes up to **16 TiB** (tebibytes =  $2^{40}$  bytes), still using standard **4 KiB** block size
- An important difference from previous **ext2/ext3** is that **ext4** doesn't use the combined allocation mode with *single*, *double*, and *triple* indirect pointers (too complex for large files)
- Instead, **ext4** adopted an extent-based system
  - a range of contiguous physical blocks that can map up to **128 MiB** of contiguous space
  - there can be **4** extents referenced directly by the inode
  - when there are more than 4 extents to a file, the rest of the extents are indexed in a tree

Per ulteriori dettagli:

<https://developer.ibm.com/tutorials/l-anatomy-ext4/>

[https://ext4.wiki.kernel.org/index.php/Ext4\\_Disk\\_Layout](https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout)

Controllare le proprietà di un file (e.g. precedente permission.c) con il comando da terminale:  
stat permission.c

## Output of stat

- In this examples, the physical size of the block on the disk is **4096 bytes**, but the size of the counted blocks is **512 bytes**
- Therefore in the first example, for a file size of **1948 bytes**, we need **1** physical block of **4096 bytes**, equivalent to **8** blocks of **512 bytes**
- In the second example, for **17112 bytes** we need **5** physical blocks (i.e. counted blocks =  $5 \times 8 = 40$ )

```
Terminal - debianos@debianos: ~/SO/lab9
File Edit View Terminal Tabs Help
debianos@debianos:~/SO/lab9$ stat fileinfo.c
  File: fileinfo.c
  Size: 1948          Blocks: 8          IO Block: 4096   regular file
Device: 801h/2049d  Inode: 132011      Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/debianos)  Gid: ( 1000/debianos)
Access: 2023-05-12 09:51:14.539109959 +0200
Modify: 2023-05-12 01:12:11.000000000 +0200
Change: 2023-05-12 09:47:01.888671602 +0200
Birth: 2023-05-12 09:47:01.884671566 +0200
debianos@debianos:~/SO/lab9$ stat fileinfo
  File: fileinfo
  Size: 17112         Blocks: 40         IO Block: 4096   regular file
Device: 801h/2049d  Inode: 135675      Links: 1
Access: (0755/-rwxr-xr-x)  Uid: ( 1000/debianos)  Gid: ( 1000/debianos)
Access: 2023-05-12 09:51:21.183176408 +0200
Modify: 2023-05-12 09:51:14.575110319 +0200
Change: 2023-05-12 09:51:14.575110319 +0200
Birth: 2023-05-12 09:51:14.563110199 +0200
debianos@debianos:~/SO/lab9$
```

Dall'output dei comandi precedenti, verificare il numero di blocchi allocati per un file.

- Verificare che anche un file di dimensione 1 byte occupi almeno 4096 byte (cioè  $8 \times 512$  byte).
- Scrivere un singolo carattere nel precedente abcde.txt e verificarlo con stat
- Scaricare il seguente file nella propria home directory  
[https://upload.wikimedia.org/wikipedia/commons/6/6c/ENIAC\\_Penn1.jpg](https://upload.wikimedia.org/wikipedia/commons/6/6c/ENIAC_Penn1.jpg)  
per evitare di usare il browser, si può scaricare direttamente dal terminale con wget [https://upload.wikimedia.org/wikipedia/commons/6/6c/ENIAC\\_Penn1.jpg](https://upload.wikimedia.org/wikipedia/commons/6/6c/ENIAC_Penn1.jpg)
- Controllare il numero di blocchi allocati per il file.
- Calcolare il numero di byte "sprecati" a causa della frammentazione interna.

Soluzione nella pagina seguente.

1000 Kilobyte (KB)	1024 Kibibyte (KiB)
$1000^2$ Megabyte (MB)	$1024^2$ Mebibyte (MiB)
$1000^3$ Gigabyte (GB)	$1024^3$ Gibibyte (GiB)
$1000^4$ Terabyte (TB)	$1024^4$ Tebibyte (TiB)

**SOLUTION:**

file size: **2148811**

I/O block size: **4096**

⇒ **525** blocks needed

internal frag:  $4096 * 525 - 2148811$

$$= \color{blue}{\mathbf{1589}}$$

# Esercizio 9 - Sistemi operativi Real Time

RIOT is an open-source microcontroller operating system, designed to match the requirements of Internet of Things (IoT) devices and other embedded devices. <https://www.riot-os.org/>

- Small OS for networked and memory-constrained systems
- Focus on low-power wireless IoT devices
- Open-source software (GPL)
- Initially developed by FU Berlin, INRIA, HAW Hamburg
- Derived from FireKernel, made for sensor networks
- Microkernel architecture
- Programmable in C, C++, and Rust
- Multithreading and real-time
- Runs on 8-, 16- and 32-bit processors
- Provides multiple network stacks and protocols (IPv6, 6LoWPAN, UDP, TCP, CoAP)

## Arduino Mega 2560

- Based on ATmega2560 8-bit microcontroller
- 54 digital I/O pins (of which 15 can be used as PWM outputs)
- 16 analog inputs
- 4 UARTs (hardware serial ports)
- 16 MHz crystal oscillator



Microcontroller	ATmega2560
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	54 (of which 15 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	256 KB of which 8 KB used by bootloader
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz
LED_BUILTIN	13
Length	101.52 mm
Width	53.3 mm
Weight	37 g

## INSTALLARE RIOT nella VM:

- Installare due pacchetti, il compilatore e le librerie C per il microcontrollore Arduino, da terminale:  
`sudo apt install gcc-avr avr-libc`
- Scaricare RIOT, da terminale:  
`wget https://github.com/RIOT-OS/RIOT/archive/refs/tags/2023.04.zip`
- Infine, estrarre il contenuto del file zip come segue, da terminale:  
`unzip 2023.04.zip`
- Se tutto è andato bene, ora si dovrebbe avere una cartella RIOT-2023.04 nella propria home. Controllare con istruzione `ls`

se avete problemi di spazio per installare, potete usare i seguenti tips:

- **IMPORTANT:** before starting, check that in your VM there are at least **500MB** of disk available
  - you can check on the terminal with the command  
`df -m /`
  - the information is available under the "Available" column
- In case, you can free some space by removing any large files used in previous workshops, or from other directories
  - e.g. check size of current subdirectories  
`du -sm *`
- You can also clean some unnecessary system files with
  - `sudo apt autoremove`
  - `sudo apt autoclean`
  - `sudo apt clean`

Primo test di compilazione su RIOT.

- accedete al folder “hello-world”, cartella di RIOT:

```
cd RIOT-2023.04/examples/hello-world
```

- compilare con:

```
make BOARD=arduino-mega2560
```

in output dovreste ottenere una nuova cartella:

```
bin/arduino-mega2560/
```

al cui interno si trova il file `hello-world.elf`

- aprite, da browser, il simulatore arduino  
<https://wokwi.com/projects/new/arduino-mega>

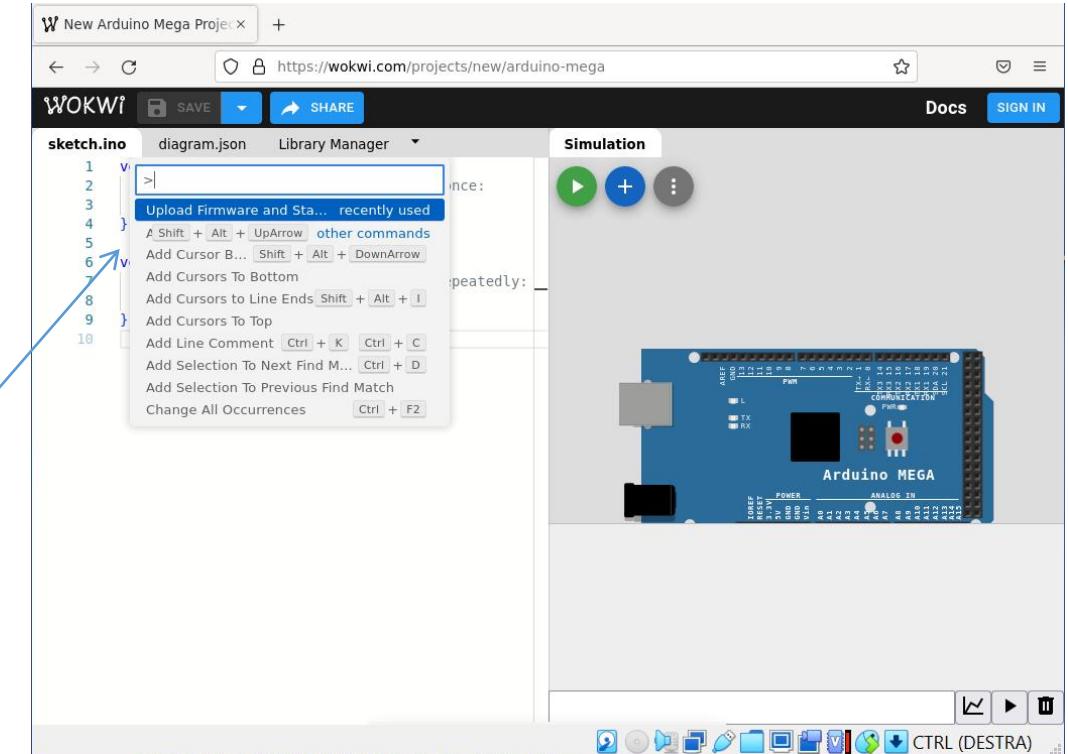
- Dall'editor (lato sinistro della pagina web) premere il tasto  
il tasto [F1] e selezionare “Upload Firmware and Start Simulation...”  
(elenco è lungo, scrivete pure nel box “up” per velocizzare ricerca)

e caricate il file:

```
hello-world.elf
```

si dovrebbe trovare in `/home/debianos/RIOT-2023.04/examples/hello-world/bin/arduino-mega2560`

- In caso di successo, il messaggio “Hello World” dovrebbe essere stampato sul terminale del simulatore, insieme ad altre informazioni su RIOT e sulla scheda Arduino.



Ora cerchiamo di creare nuovi comandi:

- comando 'board' restituirà il nome della scheda
- comando 'cpu' restituirà il nome della cpu.

Scaricate i file Makefile e main.c (vedi folder Esercizio9\ParteA).

IMPORTANTE: salvare sempre i file in directory che NON contengano spazi, altrimenti la compilazione successiva fallirà! e.g. non usare (come nome) 'my folder' ma 'my\_folder'.

Attenzione, quando usate Makefile controllate che nella stessa cartella ci sia solo main.c, non copiate anche altri .c come main\_new.c

Aggiungere una shell di base all'applicazione, la prima cosa da fare è includere l'intestazione

shell.h all'inizio del file main.c, appena sotto la riga

#include <stdio.h>, dunque avremo:

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

```
#include "shell.h"
```

Quindi, nella funzione principale, la shell può essere avviata aggiungendo il seguente codice prima di "return 0" di main(void):

```
char line_buf[SHELL_DEFAULT_BUFSIZE];
shell_run(NULL, line_buf, SHELL_DEFAULT_BUFSIZE);
```

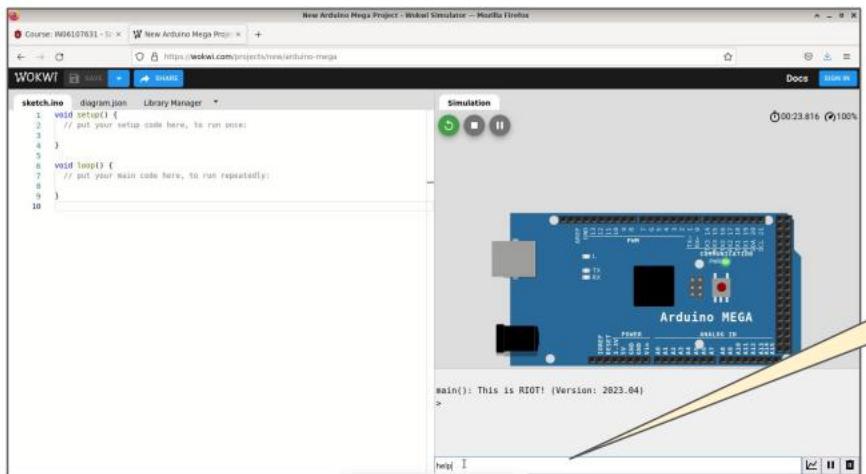
compilate con i seguenti comandi:

```
make BOARD=arduino-mega2560 RIOTBASE=path-to-RIOT-directory
```

dove path-to-RIOT-directory può essere, ad esempio,  
/home/debianos/RIOT-2023.04 (se questo è il punto in cui è stato decompresso)

Nella cartella dove si trova Makefile dovreste vedere il folder bin. Caricare ed eseguire il file binario bin/arduino-mega2560/basic-shell.elf sul simulatore WOKWI, come abbiamo fatto per l'esempio "hello world".

Per impostazione predefinita, il modulo shell fornisce un comando ‘help’ per elencare i comandi disponibili. Come si può vedere, non ci sono altri comandi disponibili, quindi implementiamone un paio per stampare alcune informazioni sulla scheda e sulla cpu.



```
static int _board_handler(int argc, char **argv)
{
    /* Not used, avoid a warning during build */
    (void)argc;
    (void)argv;

    puts(RIOT_BOARD);
    return 0;
}
```

RIOT fornisce due macro predefinite che contengono la scheda e la cpu corrente in cui l'applicazione è in esecuzione: RIOT\_BOARD e RIOT\_CPU, che verranno utilizzate per stampare board e cpu nei corrispondenti comandi di shell.  
In main.c, aggiungete le funzioni di callback per i comandi board e cpu, appena sotto #include “shell.h”

```
static int _cpu_handler(int argc, char **argv)
{
    /* Not used, avoid a warning during build */
    (void)argc;
    (void)argv;

    puts(RIOT_CPU);
    return 0;
}
```

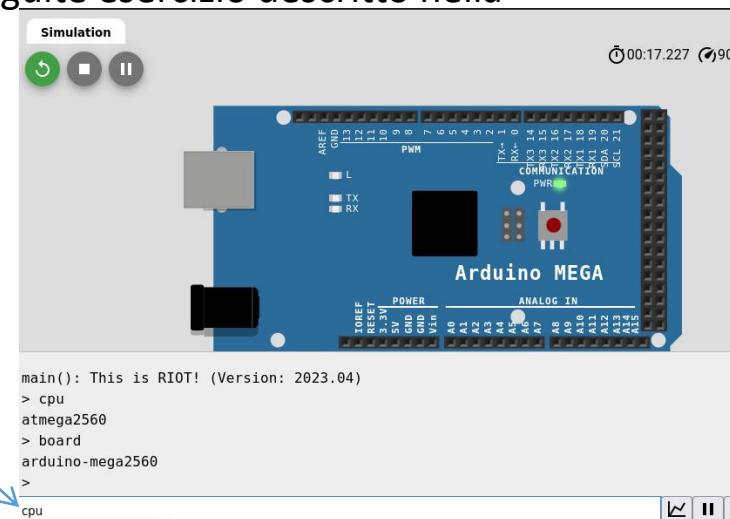
Definiamo, nell'elenco dei comandi di shell disponibili, il nome del comando, il messaggio di aiuto e la funzione di richiamo associata. Questo si ottiene semplicemente aggiungendo il seguente codice tra la funzione callback e la funzione principale:

```
static const shell_command_t shell_commands[] = {  
    { "board", "Print the board name", _board_handler },  
    { "cpu", "Print the cpu name", _cpu_handler },  
    { NULL, NULL, NULL }  
};
```

Nella funzione principale, modificare la precedente chiamata shell\_run per integrare il nuovo elenco di shell\_commands:  
shell\_run(shell\_commands, line\_buf, HELL\_DEFAULT\_BUFSIZE);

Soluzione è nella pagina successiva, prima di controllare soluzione file Esercizio9\ParteA\main\_new.c eseguite esercizio descritto nella colonna destra di questa pagina.

Ricompilare ed eseguire la nuova applicazione nel simulatore WOKWI.  
Testare i nuovi comandi.



È possibile passare uno o più argomenti al gestore di un comando di shell.

Come in una tipica funzione main del C, i parametri argc e argv contengono il numero di token della riga di comando (compreso il comando stesso) e il numero di token nella riga di comando (incluso il comando stesso) e i puntatori ad essi, rispettivamente.

Aggiungere un nuovo comando echo alla shell RIOT, che si limita a visualizzare le parole parole passate come argomenti  
Ad esempio, il comando  
echo Hello world!  
dovrebbe stampare le parole Hello world!  
Attenzione, affinchè funzioni la simulazione deve essere attiva, non sospesa.

Compilare e testare questo programma, soluzione nella pagina successiva, è il file  
Esercizio9\ParteA\main\_new.c controllatelo se qualcosa non vi torna.

Aggiungete un nuovo comando echo alla shell RIOT, che visualizza semplicemente le parole passate come argomenti

```
#include <stdio.h>

/* Include the shell header */
#include "shell.h"

/* Implement the shell function callback here */
static int _board_handler(int argc, char **argv)
{
    /* Not used, avoid a warning during build */
    (void)argc;
    (void)argv;

    puts(RIOT_BOARD);
    return 0;
}

static int _cpu_handler(int argc, char **argv)
{
    /* Not used, avoid a warning during build */
    (void)argc;
    (void)argv;

    puts(RIOT_CPU);
    return 0;
}
```

```
static int _echo_handler(int argc, char **argv)
{
    int token = 1;
    while (argc-- > 1)
    {
        printf("%s ", argv[token++]);
    }
    puts("");
    return 0;
}

/* Add the shell command to the list of commands here */
static const shell_command_t shell_commands[] = {
    { "board", "Print the board name", _board_handler },
    { "cpu", "Print the cpu name", _cpu_handler },
    { "echo", "Display a line of text", _echo_handler },
    { NULL, NULL, NULL }
};

int main(void)
{
    /* Start the shell here */
    char line_buf[SHELL_DEFAULT_BUFSIZE];
    shell_run(shell_commands, line_buf, SHELL_DEFAULT_BUFSIZE);

    return 0;
}
```

Prossimo esercizio è monitorare i thread in esecuzione,  
l'applicazione deve includere una shell con alcuni comandi extra:  
Scaricare il file Makefile (dal folder Esercizio9\ParteB) e aggiungere  
quanto segue:

USEMODULE += shell

USEMODULE += shell\_cmds\_default

USEMODULE += ps

Il modulo shell\_cmds\_default costruirà automaticamente i  
comandi di shell forniti da altri moduli.

Il modulo ps fornisce un comando di shell per monitorare l'attività  
dei thread.

### Eseguire la shell

Scaricare il file main.c da Esercizio9\ParteB e, nella funzione main,  
aggiungere quanto segue per eseguire la shell

```
char line_buf[SHELL_DEFAULT_BUFSIZE];
shell_run(NULL, line_buf, SHELL_DEFAULT_BUFSIZE);
```

PS in linux per fare parentesi quadre tasto Alt Gr + 8 e  
Alt Gr + 9



### Testare l'applicazione:

a) Da un terminale:

make BOARD=arduino-mega2560 RIOTBASE='percorso della  
directory RIOT'

b) Caricare quindi il file bin/arduino-mega2560/monitor.elf  
su WOKWI e avviare la simulazione.

c) Nella shell RIOT, elencare i comandi disponibili con il  
comando help:

> help

Un nuovo thread può essere avviato utilizzando la funzione `thread_create`.

Anche la memoria di stack utilizzata dal thread deve essere dichiarata globalmente. Infine, il codice in esecuzione nel thread deve essere implementato in una funzione di gestione del thread.

### Implementare nuovo thread

In `main.c`, includere `thread.h` (dopo `shell.h`) e dichiarare la memoria di stack utilizzata dal nuovo thread:

```
#include "thread.h"  
static char stack[THREAD_STACKSIZE_MAIN];
```

Aggiungere l'implementazione della funzione di gestione dei thread (prima della funzione principale)

```
static void *thread_handler(void *arg)  
{  
    (void)arg;  
    puts("thread!");  
    return NULL;  
}
```

Infine, possiamo creare e avviare il ‘nuovo thread’ nella funzione principale, prima dell'esecuzione della shell  
`thread_create(stack, sizeof(stack), THREAD_PRIORITY_MAIN - 1, 0, thread_handler, NULL, “nuovo thread”);`  
N.B. Il thread deve essere creato prima di chiamare la funzione `shell_run`, perché quest'ultima non ritorna mai, quindi il codice successivo non verrà eseguito.

Compilare l'applicazione e simularla su WOKWI

Modificare il gestore di thread per eseguire un ciclo busy:

```
static void *thread_handler(void *arg)
{
    (void)arg;
    while (1) /* ciclo occupato */
        return NULL;
}
```

Compilare l'applicazione e simularla su WOKWI

Cambiare la priorità del thread a THREAD\_PRIORITY\_MAIN

+ 1:

```
thread_create(stack, sizeof(stack),
```

```
THREAD_PRIORITY_MAIN + 1,
```

```
0, thread_handler, NULL, "nuovo thread");
```

La nuova priorità è THREAD\_PRIORITY\_MAIN + 1, quindi il

thread creato ha una priorità più bassa del thread

principale (numero più alto ⇒ priorità più bassa).

Nel seguente esercizio due thread scriveranno e leggeranno simultaneamente in un buffer condiviso a livello globale.

Il mutex utilizzato garantirà che il thread di lettura possa leggere solo dopo che il thread di scrittura abbia completato il suo lavoro di scrittura.

I due thread si comporteranno come segue:

- a) il thread principale legge il contenuto di un buffer condiviso e ne stampa il contenuto ogni 250 ms;
- b) un secondo thread scrive il contenuto di questo buffer condiviso ogni 100 ms, ma ogni scrittura richiede 50 ms per essere completata. Quindi, se la concorrenza non fosse gestita correttamente, il thread di lettura potrebbe stampare contenuti corrotti.

Utilizzeremo alcuni ritardi temporali nel nostro codice, scaricate Makefile da Esercizio9\ParteC, aggiungere i seguenti moduli aggiuntivi:

USEMODULE += ztimer

USEMODULE += ztimer\_msec

Il modulo ztimer fornisce un'astrazione di alto livello dei timer hardware per le esigenze di temporizzazione delle applicazioni.

Caricando il modulo ztimer\_msec, il sistema viene automaticamente configurato per l'utilizzo di un clock che fornisca millesimi.

Implementazione del writer thread:

a) scaricate main.c da Esercizio9\ParteC e aggiungete:

```
#include "mutex.h"
```

b) Ora dichiariamo e istanziamo un buffer condiviso globale con un mutex associato:

```
typedef struct {
```

```
    char buffer[128];
```

```
    mutex_t lock;
```

```
} data_t;
```

```
static data_t data;
```

c) Poiché viene creato un nuovo thread, deve essere allocato anche uno stack di memoria dedicato:

```
static char writer_stack[THREAD_STACKSIZE_MAIN];
```

Il ciclo infinito del filo di scrittura è suddiviso in 3 fasi:

a) step 1: ottenere lock mediante `mutex_lock(&data.lock);`

b) step 2: scrivere dati nel buffer

```
size_t p = sprintf(data.buffer, "%s", "one");
```

```
ztimer_sleep(ZTIMER_MSEC, 50);
```

```
sprintf(&data.buffer[p], "%s", " two");
```

c) step 3: rilasciare il lock

```
mutex_unlock(&data.lock);
```

Nella funzione principale, creiamo finalmente il writer thread:

```
thread_create(writer_stack, sizeof(writer_stack), THREAD_PRIORITY_MAIN - 1,  
0, writer_thread, NULL, "writer thread");
```

Implementare il ciclo di lettura.

Nella funzione principale, all'interno del ciclo while,  
aggiungiamo il seguente codice:

```
mutex_lock(&data.lock);
printf("%s\n", data.buffer);
mutex_unlock(&data.lock);
```

Ogni accesso al buffer deve essere protetto da chiamate  
mutex\_lock / mutex\_unlock per assicurarsi che il buffer possa  
essere letto in modo sicuro.

Compilare l'applicazione (file threading.elf ) e simularla su  
WOKWI

Provate a rimuovere l'uso del mutex dappertutto e verificate  
che l'output sia talvolta incompleto.

Modificare il programma reader-writer per aggiungere un  
comando di shell per avviare/arrestare entrambi i thread writer  
e reader. Ad esempio, print off per fermarli e poi print on per  
avviare entrambi i thread,

Per possibile soluzione della gestione dei due thread, vedi file  
in: Esercizio9\ParteC\soluzioni