

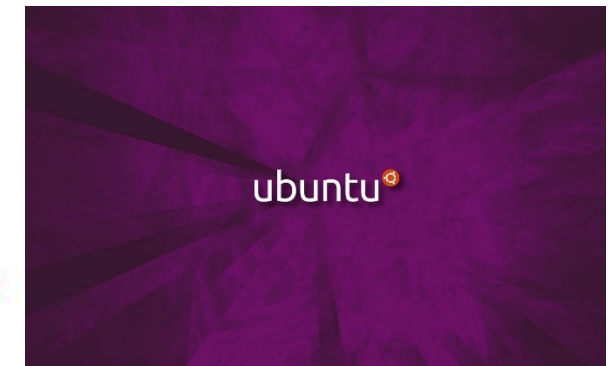
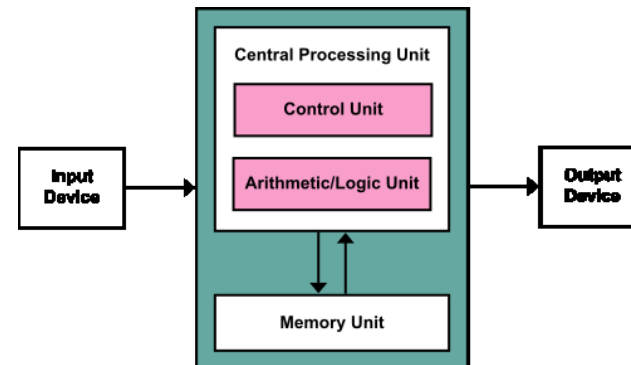
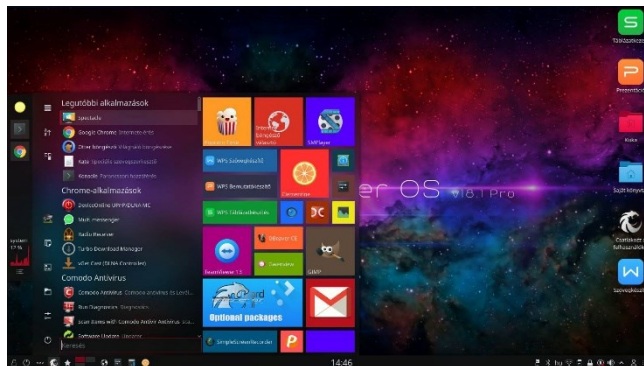
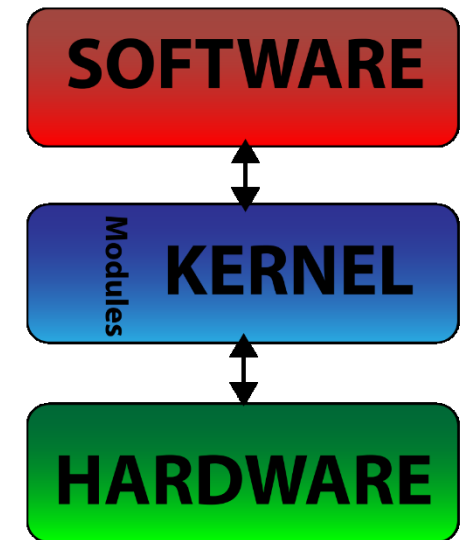
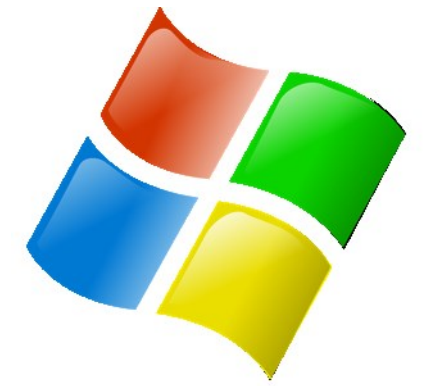


**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

## *Corso di Sistemi Operativi*

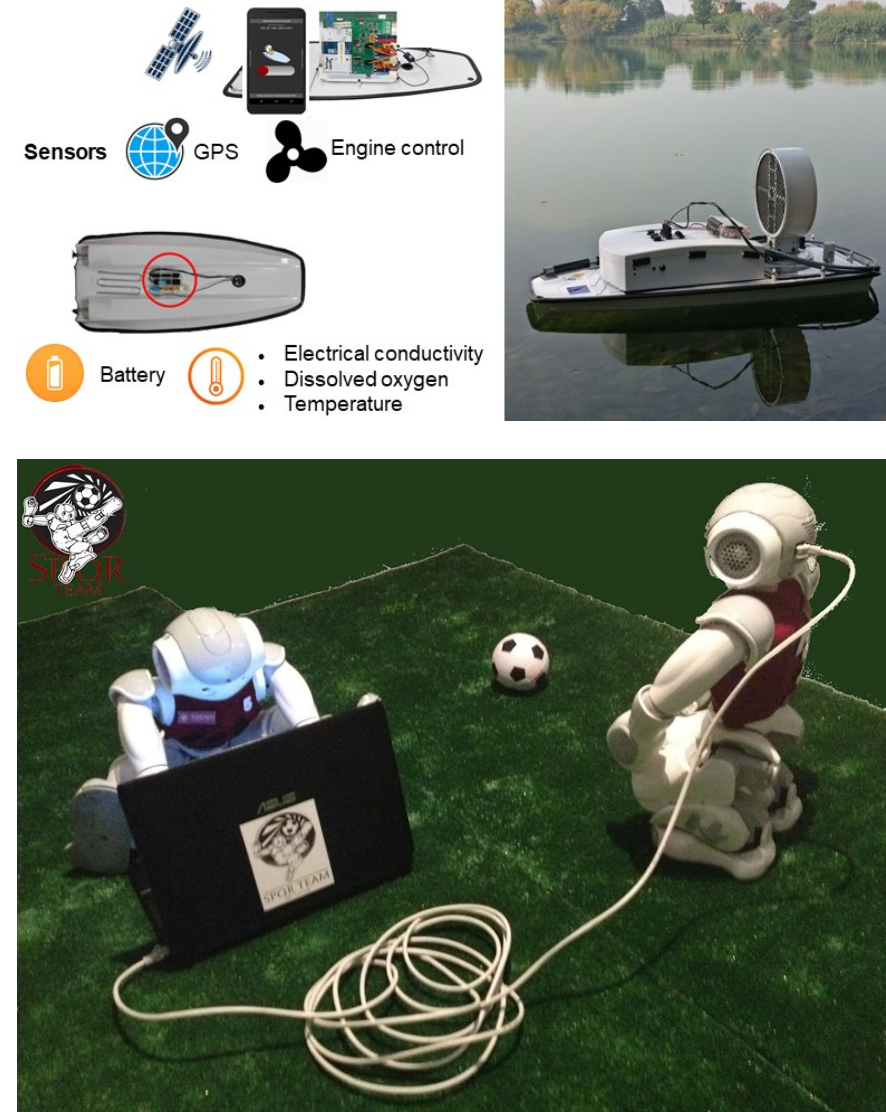
# Esempi di sincronizzazione

Docente:  
**Domenico Daniele  
Bloisi**



# Domenico Daniele Bloisi

- Ricercatore RTD B  
Dipartimento di Matematica, Informatica  
ed Economia  
Università degli studi della Basilicata  
<http://web.unibas.it/bloisi>
- SPQR Robot Soccer Team  
Dipartimento di Informatica, Automatica  
e Gestionale Università degli studi di  
Roma “La Sapienza”  
<http://spqr.diag.uniroma1.it>



# Informazioni sul corso

---

- Home page del corso:  
<http://web.unibas.it/bloisi/corsi/sistemi-operativi.html>
- Docente: Domenico Daniele Bloisi
- Periodo: I semestre ottobre 2020 – febbraio 2021
  - Lunedì 15:00-17:00
  - Martedì 9:30-11:30



**Le lezioni saranno erogate in modalità esclusivamente on-line**

Codice corso Google Classroom:

<https://classroom.google.com/c/MTQ2ODE2NTk3ODIz?cjc=67646ik>

# Ricevimento

---

- Su appuntamento tramite Google Meet

Per prenotare un appuntamento inviare  
una email a

[domenico.bloisi@unibas.it](mailto:domenico.bloisi@unibas.it)



# Programma – Sistemi Operativi

---

- Introduzione ai sistemi operativi
- Gestione dei processi
- Sincronizzazione dei processi
- Gestione della memoria centrale
- Gestione della memoria di massa
- File system
- Sicurezza e protezione

# Semafori

---

Un **semaforo S** è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: **wait()** e **signal()**.

```
wait(S) {  
    while(S <= 0)  
        ; /* busy wait */  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

# Implementazione semafori

---

- This implementation is based on **busy waiting** in critical section implementation (that is, the code for `wait()` and `signal()`)
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Can we implement semaphores with no busy waiting?



# Semafori senza busy waiting

---

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:

- value (of type integer)
- pointer to next record in the list

```
typedef struct{  
    int value;  
    struct process *list;  
} semaphore;
```

- Two operations:
  - **sleep()** - place the process invoking the operation on the appropriate waiting queue
  - **wakeup (P)** – remove one of processes in the waiting queue and place it in the ready queue



# Semafori senza busy waiting

---

```
■ wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep();
    }
}

■ signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

# Classici problemi di sincronizzazione

---

Problema del  
produttore/consumatore  
con memoria limitata

Problema  
dei lettori-scrittori

Problema dei filosofi a cena  
(dining philosophers)

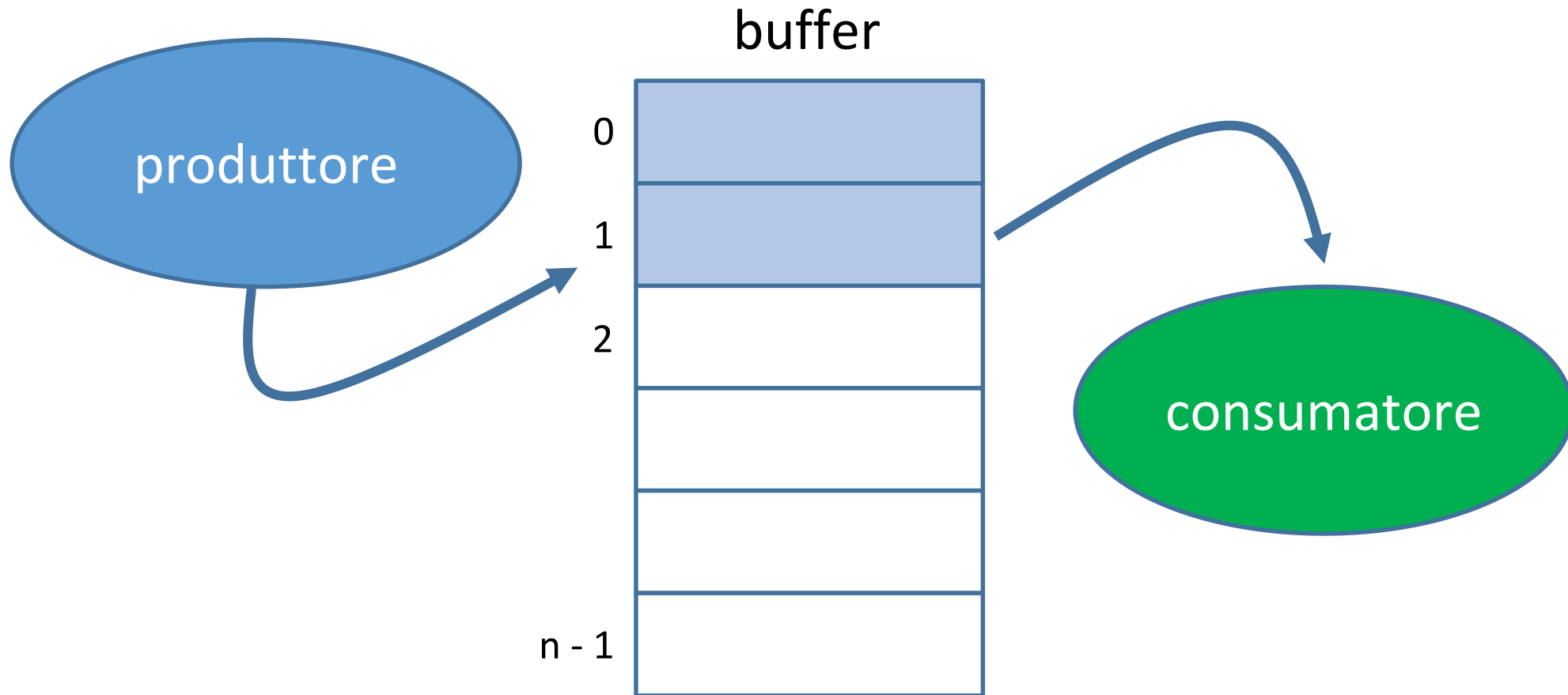
# Produttore/consumatore con memoria limitata

---

- Il **problema del produttore/consumatore con memoria limitata** si usa generalmente per illustrare la potenza delle primitive di sincronizzazione.
- Il produttore produce informazioni che verranno inserite in un buffer di memoria avente  $n$  locazioni. Il consumatore legge le informazioni liberando locazioni di memoria
- *Simmetria* esistente tra il produttore e il consumatore → il consumatore produce spazi di memoria liberi per il produttore, il quale a sua volta li consumerà

# Produttore/consumatore con memoria limitata

---



# Produttore/consumatore con memoria limitata

---

Strutture dati condivise:

```
int n;  
semaphore mutex = 1;  
semaphore empty = n;  
semaphore full = 0;
```

# Produttore/consumatore con memoria limitata

---

```
while (true) {  
    . . .  
    /* produci un elemento in next_produced */  
    . . .  
    wait(empty);  
    wait(mutex);  
    . . .  
    /* inserisci next_produced in buffer */  
    . . .  
    signal(mutex);  
    signal(full);  
}
```

**Figura 7.1** Struttura generale del processo produttore.

# Produttore/consumatore con memoria limitata

---

```
while (true) {  
    wait(full);  
    wait(mutex);  
    . . .  
    /* rimuovi un elemento da buffer e mettilo in next_consumed */  
    . . .  
    signal(mutex);  
    signal(empty);  
    . . .  
    /* consuma l'elemento contenuto in next_consumed */  
    . . .  
}
```

**Figura 7.2** Struttura generale del processo consumatore.



# Problema dei lettori/scrittori

---

- Readers read data
- Writers write data
- Rules
  - Multiple readers may read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot access data simultaneously
- Locking table
  - Whether any two can be in the critical section simultaneously

	Reader	Writer
Reader	OK	No
Writer	No	No

# Possibile soluzione al problema dei lettori/scrittori

---

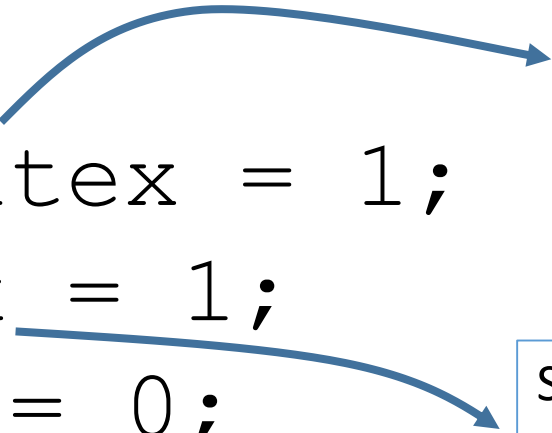
- Più lettori possono accedere contemporaneamente al database
- Gli scrittori devono avere accesso esclusivo al database
- **I lettori hanno precedenza sugli scrittori** → Se uno scrittore chiede di accedere mentre uno o più lettori stanno accedendo al database, lo scrittore deve attendere che i lettori abbiano finito → possibile stato di attesa indefinita (starvation) sugli scrittori

# Possibile soluzione al problema dei lettori/scrittori

---

Strutture dati condivise:

```
semaphore rw_mutex = 1;  
semaphore mutex = 1;  
int read_count = 0;
```



Semaforo di mutua esclusione per  
gli scrittori usato anche dal primo  
e dall'ultimo lettore

Semaforo di mutua esclusione per  
gestire gli aggiornamenti della  
variabile `read_count`

# Possibile soluzione al problema dei lettori/scrittori

---

```
while (true) {  
    wait(rw_mutex);  
    . . .  
    /* esegui l'operazione di scrittura */  
    . . .  
    signal(rw_mutex);  
}
```

CS {

**Figura 7.3** Struttura generale di un processo scrittore.

# Problema dei lettori/scrittori

```
while (true) {  
    CS {  
        wait(mutex);  
        read_count++;  
        if (read_count == 1)  
            wait(rw_mutex);  
        signal(mutex);  
        . . .  
        /* esegui l'operazione di lettura */  
        . . .  
        CS {  
            wait(mutex);  
            read_count--;  
            if (read_count == 0)  
                signal(rw_mutex);  
            signal(mutex);  
        }  
    }  
}
```

primo lettore

ultimo lettore

**Figura 7.4** Struttura generale di un processo lettore.

# Lock lettura-scrittura

---

Le soluzioni al problema dei lettori-scrittori sono state generalizzate su alcuni sistemi in modo da fornire **lock di lettura-scrittura**.

I **lock di lettura-scrittura** sono utili:

1. dove si identificano i processi che si limitano alla lettura di dati condivisi e quelli che si limitano alla scrittura di dati condivisi
2. dove si prevedono più lettori che scrittori

# Problema dei filosofi a cena

Il *problema dei filosofi a cena* (*dining philosophers*) è considerato un classico problema di sincronizzazione, perché rappresenta una vasta classe di problemi di **controllo della concorrenza**, in particolare i problemi caratterizzati dalla necessità di assegnare varie risorse a diversi processi evitando **situazioni di stallo** e **d'attesa indefinita**

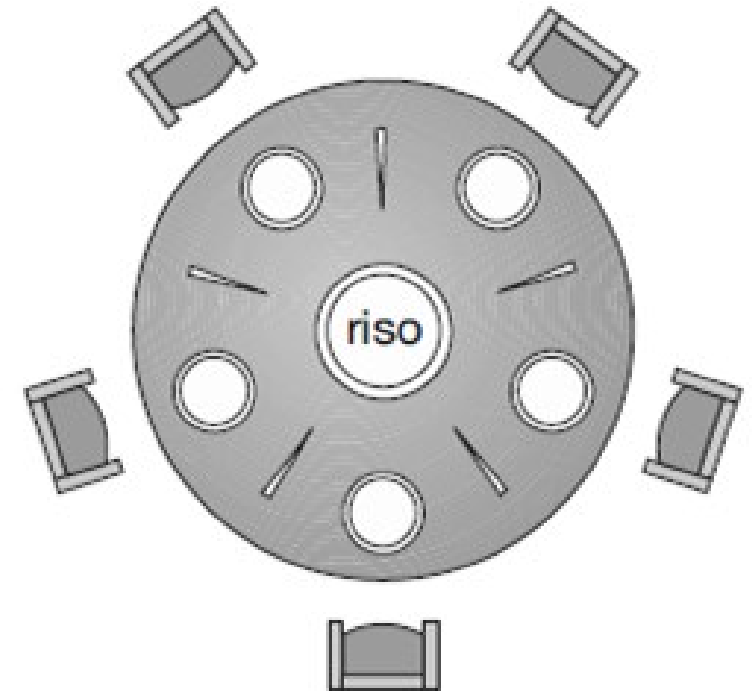


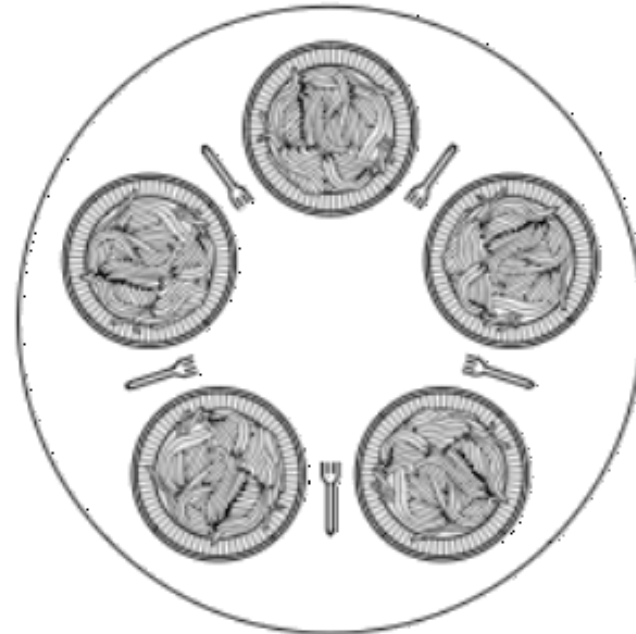
Figura 7.5 Situazione dei cinque filosofi (dining philosophers).



# Definizione del problema dei filosofi a cena

---

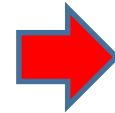
- N philosophers and N forks
- Philosophers eat/think
- Eating needs 2 forks
- Pick up one fork at a time



# Soluzione con uso di semafori

- Un semaforo per ogni forchetta o bacchetta (chopstick)  
semaphore chopstick[5]
- Tutti gli elementi di chopstick sono inizializzati a 1

```
while (true) {  
    wait(chopstick[i]);  
    wait(chopstick[(i+1) % 5]);  
    . . .  
    /* mangia */  
    . . .  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    . . .  
    /* pensa */  
    . . .  
}
```



Questa soluzione garantisce  
che non vi siano due vicini che  
mangino contemporaneamente

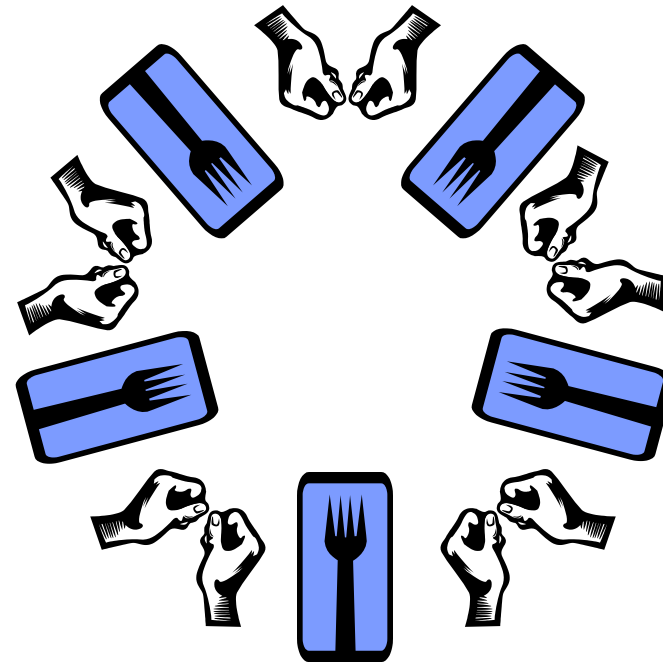


è sufficiente?

**Figura 7.6** Struttura del filosofo *i*.

# Esecuzione

```
while (true) {  
    wait(chopstick[i]);  
    wait(chopstick[(i+1) % 5]);  
    . . .  
    /* mangia */  
    . . .  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    . . .  
    /* pensa */  
    . . .  
}
```

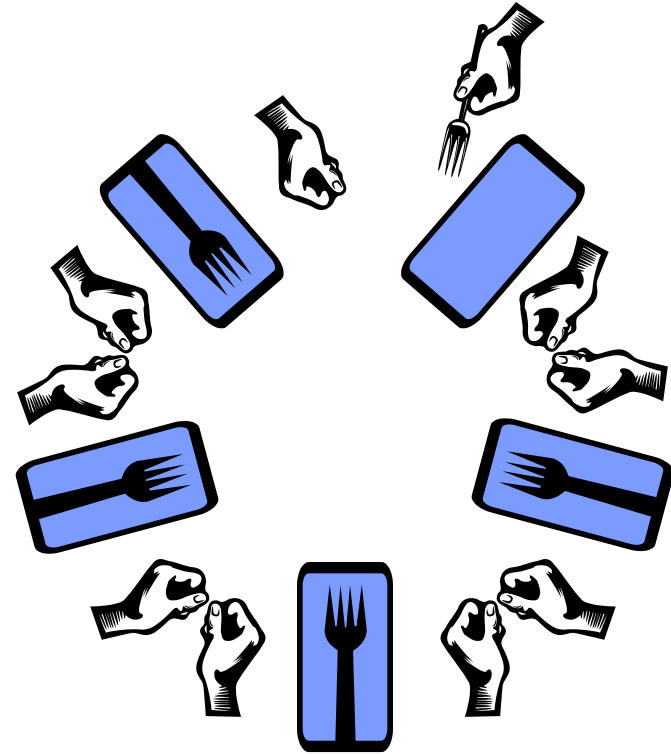


**Figura 7.6** Struttura del filosofo *i*.

# Esecuzione



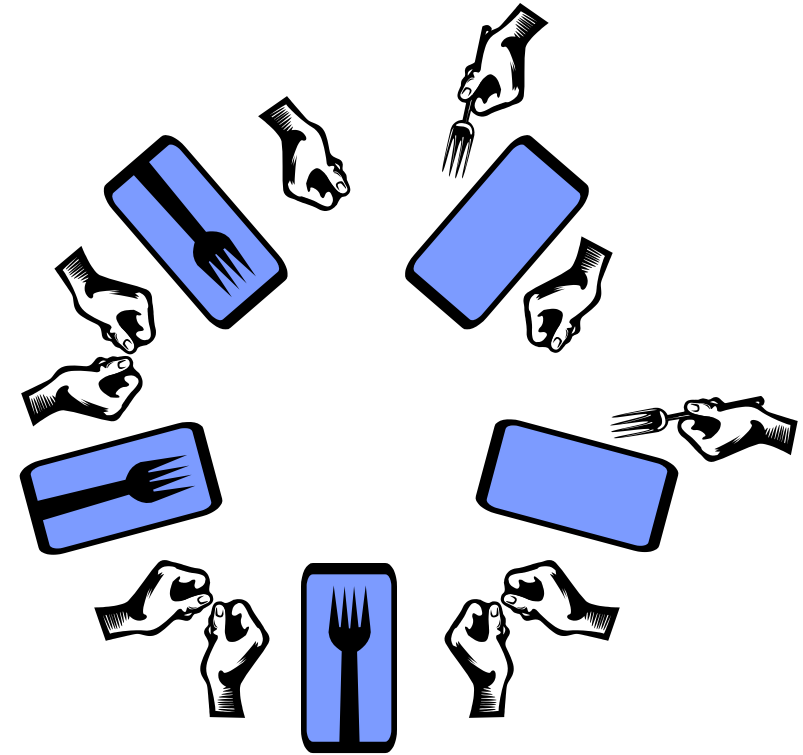
```
while (true) {  
    wait(chopstick[i]);  
    wait(chopstick[(i+1) % 5]);  
    . . .  
    /* mangia */  
    . . .  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    . . .  
    /* pensa */  
    . . .  
}
```



**Figura 7.6** Struttura del filosofo *i*.


# Esecuzione

```
→ while (true) {  
    wait(chopstick[i]);  
    wait(chopstick[(i+1) % 5]);  
    . . .  
    /* mangia */  
    . . .  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    . . .  
    /* pensa */  
    . . .  
}
```



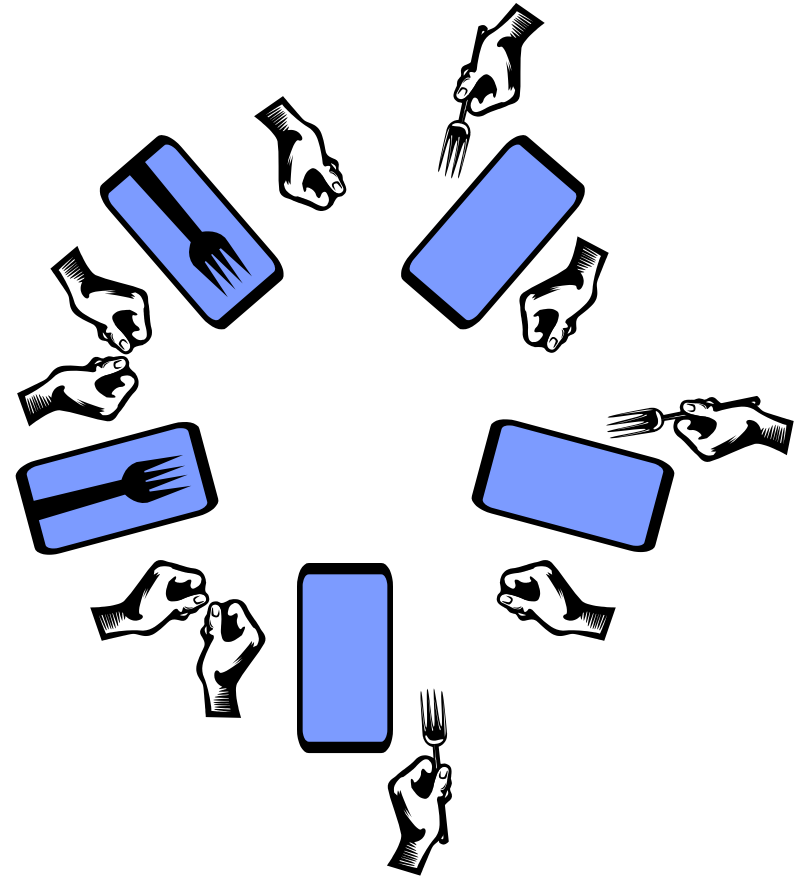
**Figura 7.6** Struttura del filosofo *i*.

# Esecuzione




```
while (true) {  
    wait(chopstick[i]);  
    wait(chopstick[(i+1) % 5]);  
    . . .  
    /* mangia */  
    . . .  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    . . .  
    /* pensa */  
    . . .  
}
```

**Figura 7.6** Struttura del filosofo *i*.

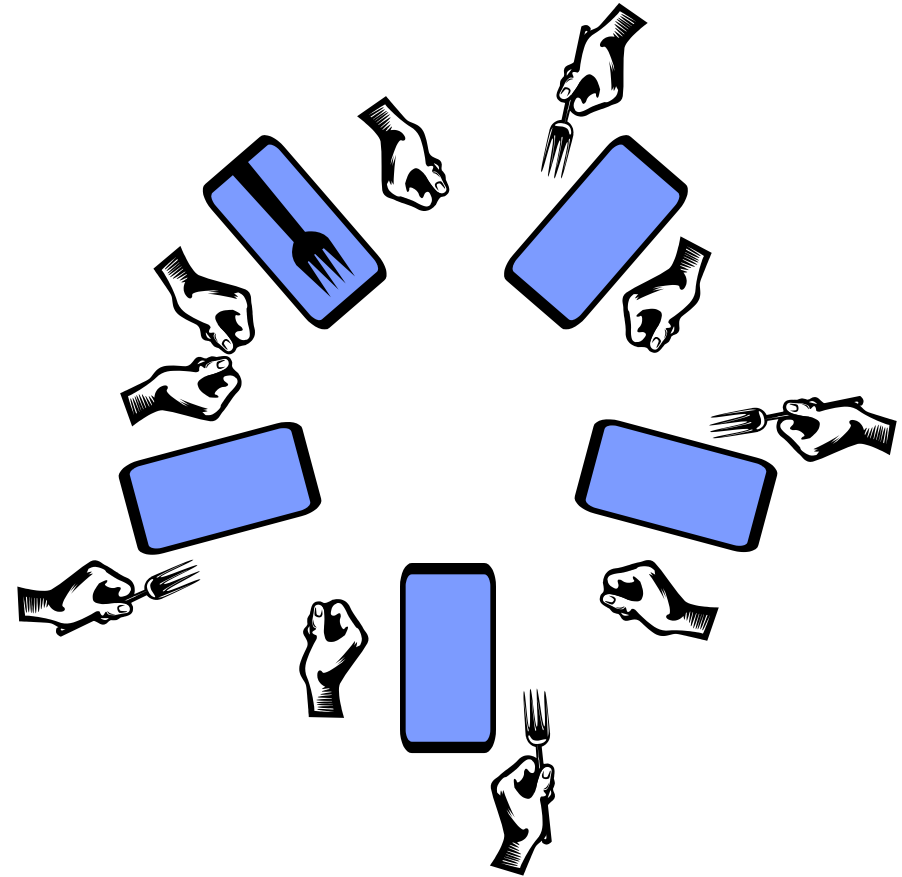


# Esecuzione

```
while (true) {  
    wait(chopstick[i]);  
    wait(chopstick[(i+1) % 5]);  
    . . .  
    /* mangia */  
    . . .  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    . . .  
    /* pensa */  
    . . .  
}
```



**Figura 7.6** Struttura del filosofo *i*.

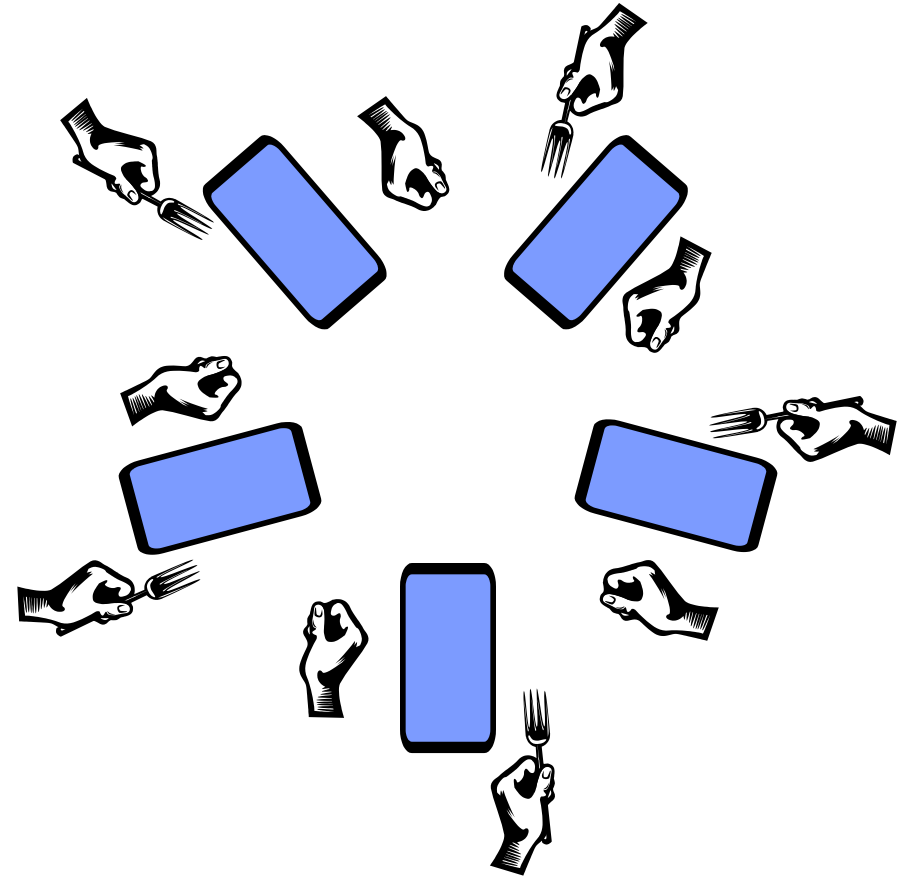




# Esecuzione

```
while (true) {  
    → wait(chopstick[i]);  
      wait(chopstick[(i+1) % 5]);  
      . . .  
      /* mangia */  
      . . .  
      signal(chopstick[i]);  
      signal(chopstick[(i+1) % 5]);  
      . . .  
      /* pensa */  
      . . .  
}
```

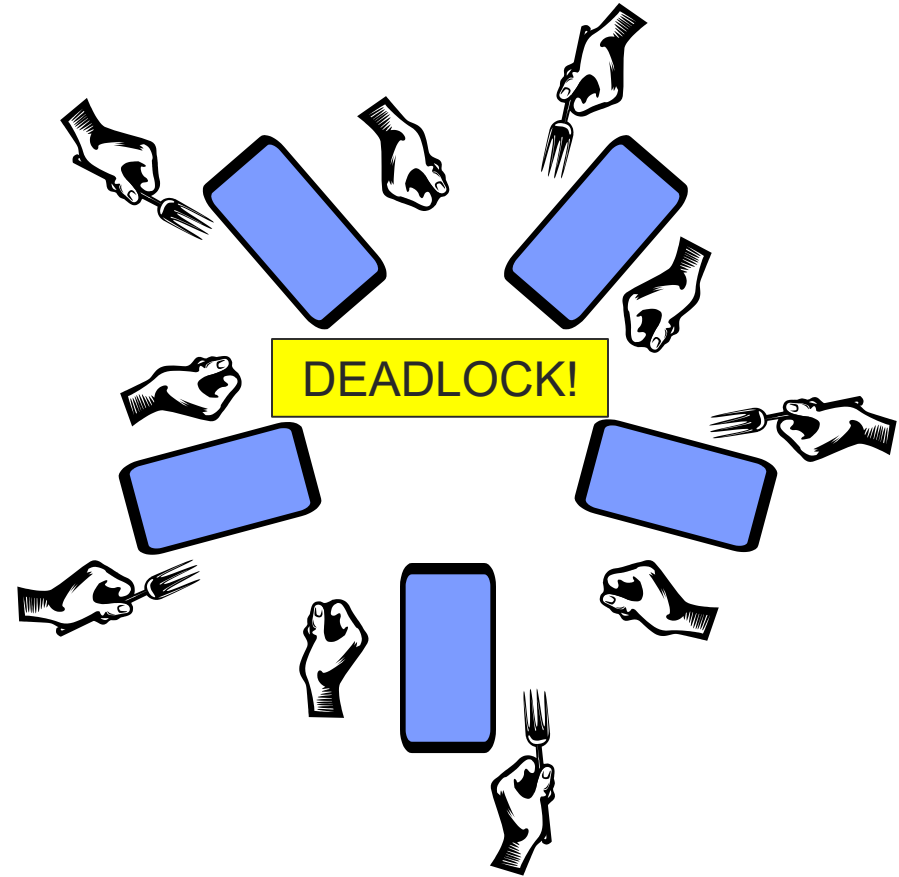
**Figura 7.6** Struttura del filosofo *i*.



# Esecuzione

```
while (true) {  
    wait(chopstick[i]);  
    wait(chopstick[(i+1) % 5]);  
    . . .  
    /* mangia */  
    . . .  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    . . .  
    /* pensa */  
    . . .  
}
```

**Figura 7.6** Struttura del filosofo *i*.



# Problema dei cinque filosofi

## Soluzione per mezzo di monitor

3 stati:

- Thinking
- Hungry
- Eating

`condition self`  
per ritardare un  
filosofo quando ha  
fame

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    Initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Figura 7.7 Una soluzione con monitor al problema dei cinque filosofi.

# Soluzione per mezzo di monitor

---

- La soluzione per mezzo di semafori impone che un filosofo possa prendere entrambe le bacchette solo quando sono disponibili
- Assicura che due vicini non mangino contemporaneamente
- Non provoca situazioni di stallo
- Non garantisce che sia evitata l'attesa indefinita

# Sincronizzazione in Windows

---

Il sistema operativo Windows ha un **kernel multithread** che offre anche il supporto alle applicazioni in tempo reale e alle architetture multiprocessore

## All'interno del kernel

Sistema monoprocessore →  
disabilita temporaneamente le  
interruzioni

Sistema multiprocessore →  
spinlock

## Fuori dal kernel

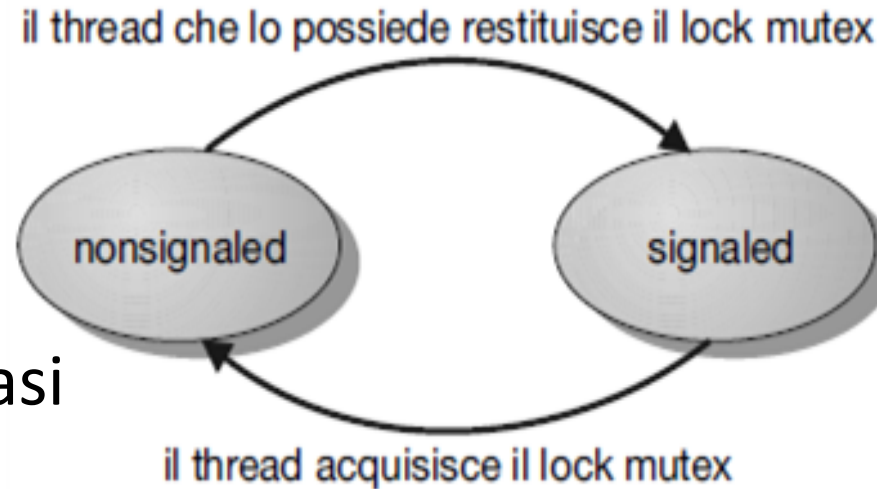
Per la sincronizzazione fuori dal kernel, Windows offre gli **oggetti dispatcher**, che permettono ai thread di sincronizzarsi servendosi di diversi meccanismi, inclusi **lock mutex**, **semafori**, **eventi** e **timer**.

# Oggetti dispatcher

---

## stato nonsignaled

l'oggetto non è disponibile e qualsiasi thread che tenta di accedervi viene bloccato



## stato signaled

l'oggetto è disponibile e un thread che tenta di accedere all'oggetto non viene bloccato

Figura 7.8 Oggetto dispatcher di tipo mutex.

# Sincronizzazione in Linux

---

## Livello Kernel

La tecnica di sincronizzazione più semplice nel kernel di Linux è **l'intero atomico**, rappresentato mediante il tipo di dato opaco `atomic_t`



Gli interi atomici sono particolarmente efficienti in situazioni in cui deve essere aggiornata una variabile intera, per esempio un contatore, in quanto non risentono dell'overhead dei meccanismi di lock.



# Sincronizzazione in Linux

---

## Livello Kernel

Linux fornisce anche **spinlock** e **semafori** (nonché la variante lettore-scrittore di questi due meccanismi) per implementare i lock a livello kernel.

# Sincronizzazione POSIX

---

L'**API POSIX** è a disposizione dei programmatori a **livello utente** e non fa parte di alcun particolare kernel

**API Pthreads** utilizzate per la creazione e la **sincronizzazione di thread** da parte degli sviluppatori su sistemi UNIX, Linux e macOS

# Lock mutex

---

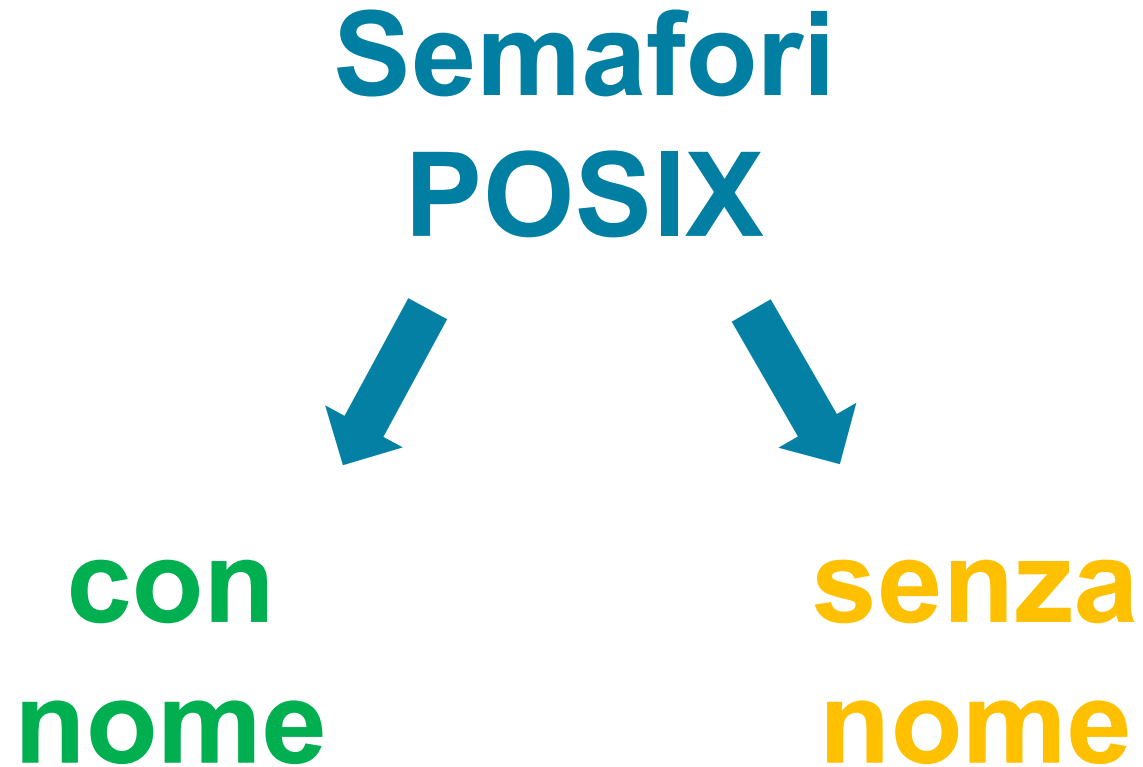
I **lock mutex** rappresentano la tecnica di sincronizzazione fondamentale in ambiente Pthreads



proteggono le **sezioni critiche** del codice

# Semafori POSIX

---



# Semafori con nome POSIX

---

Il vantaggio dei **semafori con nome** è che più processi non correlati possono facilmente utilizzare un semaforo comune come meccanismo di sincronizzazione, facendo semplicemente riferimento al **nome del semaforo**.

Sia i sistemi Linux sia quelli macOS forniscono **semafori POSIX con nome**

# Semafori senza nome POSIX

---

Un **semaforo senza nome** viene creato e inizializzato mediante la funzione `sem_init()`

I semafori POSIX senza nome usano le stesse operazioni di quelli con nome, cioè

`sem_wait()`

e

`sem_post()`

# Variabili condizionali POSIX

---

Le **variabili condizionali in Pthreads** usano il tipo di dato `pthread_cond_t` e vengono inizializzate mediante la funzione `pthread_cond_init()`

Per l'**attesa** su una variabile condizionale viene usata la funzione `pthread_cond_wait()`

# Variabili condizionali POSIX

---

Il seguente codice mostra come un thread può aspettare il verificarsi della condizione `a == b` utilizzando una variabile condizionale Pthreads:

```
pthread_mutex_lock(&mutex);  
while(a != b)  
    pthread_cond_wait(&cond_var, &mutex);  
pthread_mutex_unlock(&mutex);
```



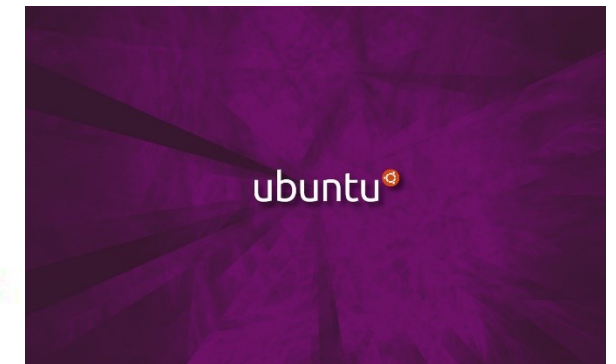
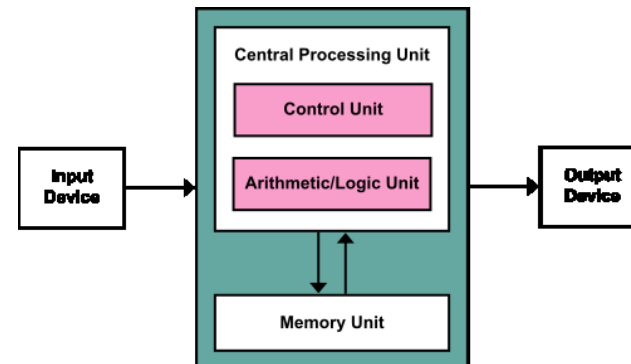
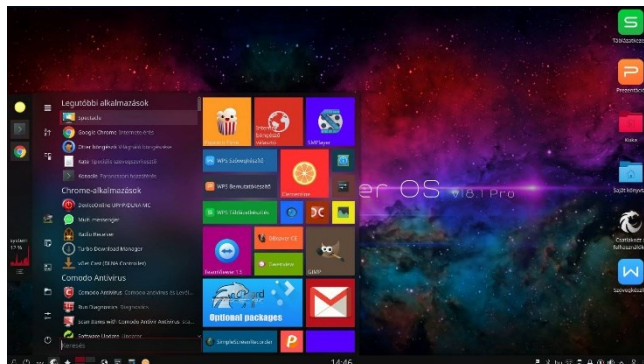
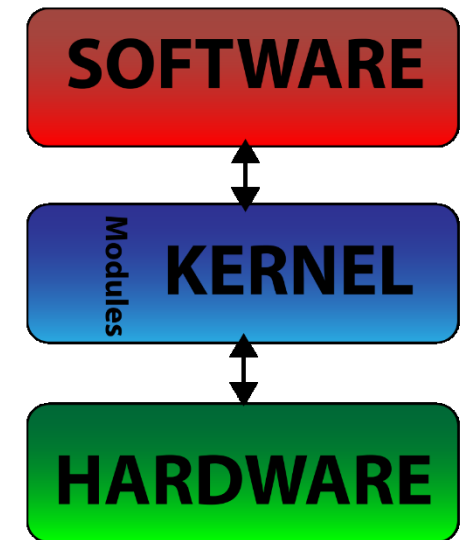
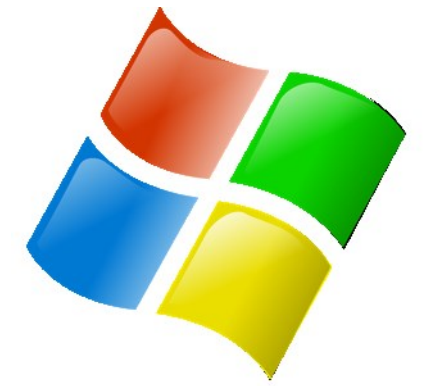


**UNIVERSITÀ DEGLI STUDI  
DELLA BASILICATA**

## *Corso di Sistemi Operativi*

# Esempi di sincronizzazione

Docente:  
**Domenico Daniele  
Bloisi**



# Esame

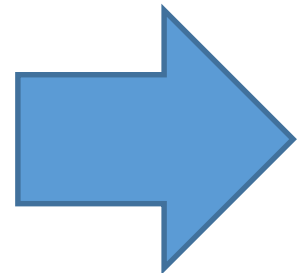
---

- Il voto finale viene conseguito svolgendo un esame scritto con tre domande a risposta aperta (max 5 punti per ogni risposta) e 2 esercizi (max 7,5 punti per ogni esercizio).
- Gli studenti possono chiedere di svolgere un progetto BONUS facoltativo per ottenere fino a tre punti che verranno sommati al voto ottenuto durante l'esame scritto.

# Progetti BONUS

---

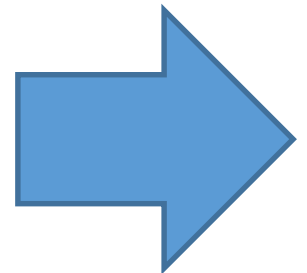
- Il progetto può essere svolto individualmente o in gruppo
- Il numero massimo di studenti in un gruppo è 2
- Il progetto dovrà essere realizzato in C/C++ in ambiente GNU/Linux



# Progetti BONUS

---

- Il codice dovrà essere disponibile su un repository Git (per esempio, GitHub, GitLab, Bitbucket, ...)
- Nel caso di lavori in gruppo, il repository dovrà avere contributi da tutti i membri del gruppo (verificabili tramite analisi delle operazioni di commit)



# Progetti BONUS

---

- Insieme al codice dovrà essere consegnata una presentazione (10-15 slide) con la descrizione del lavoro effettuato
- Un esempio di progetto BONUS è disponibile sulla home page del corso

# Git

---

Git /git/ is a distributed revision control and source code management (SCM) system with an emphasis on speed



# SCM - motivation

---

- Sources sharing across networks
- User signature on each revision
- More advanced features (local/remote repository, branching...)





# SCM - motivation

---

- Each revision is stored on the repository
- Rollback to a working version (after a disaster update) it's blazing fast

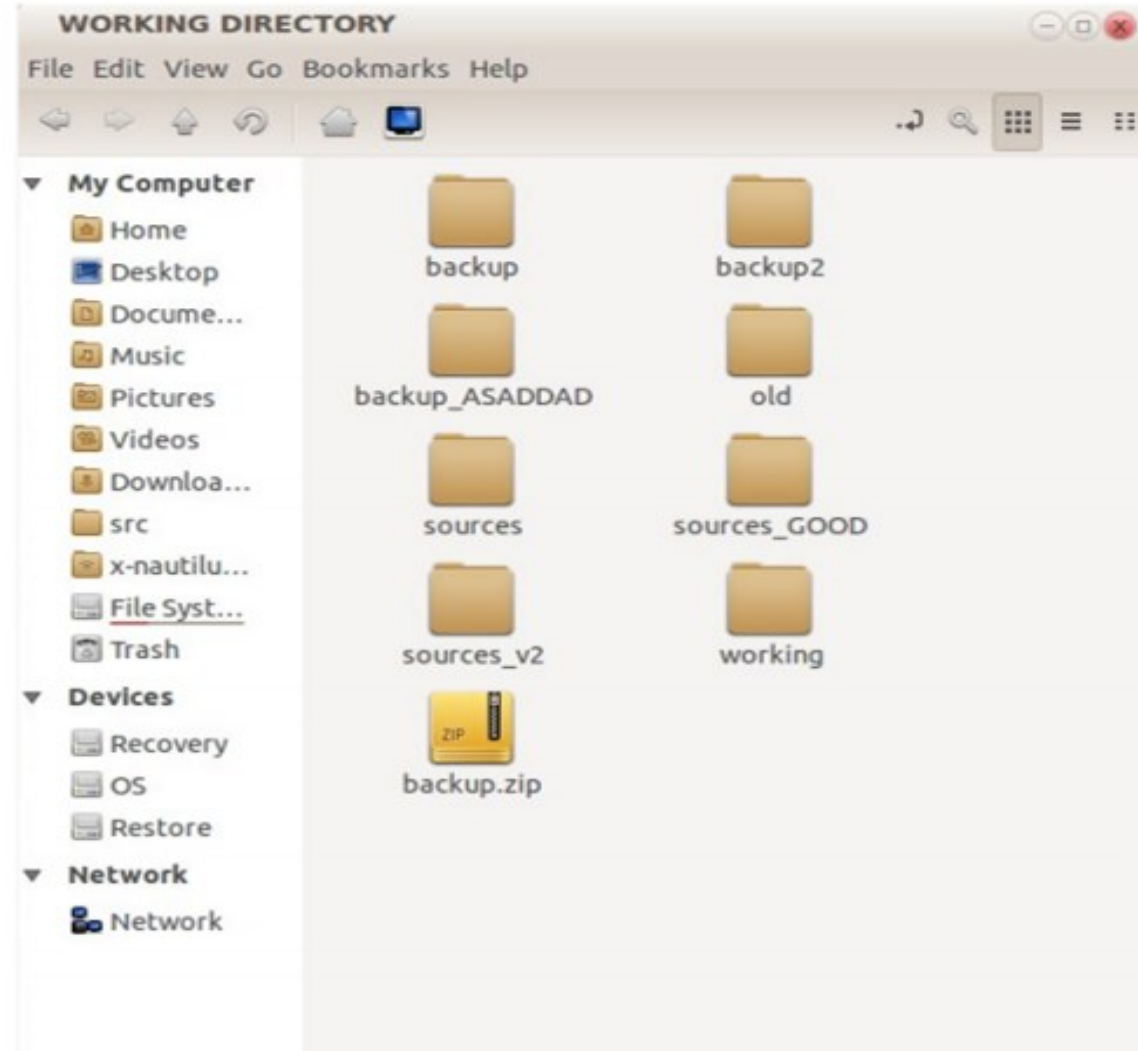




# SCM - motivation

---

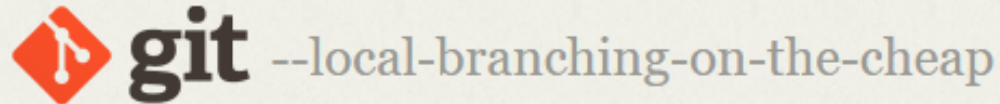
Vogliamo evitare di avere diverse versioni del codice non ordinate



# Installare Git su Linux

```
sudo apt-get install git
```

git-scm.com



Search entire site...

Git is a **free and open source** distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

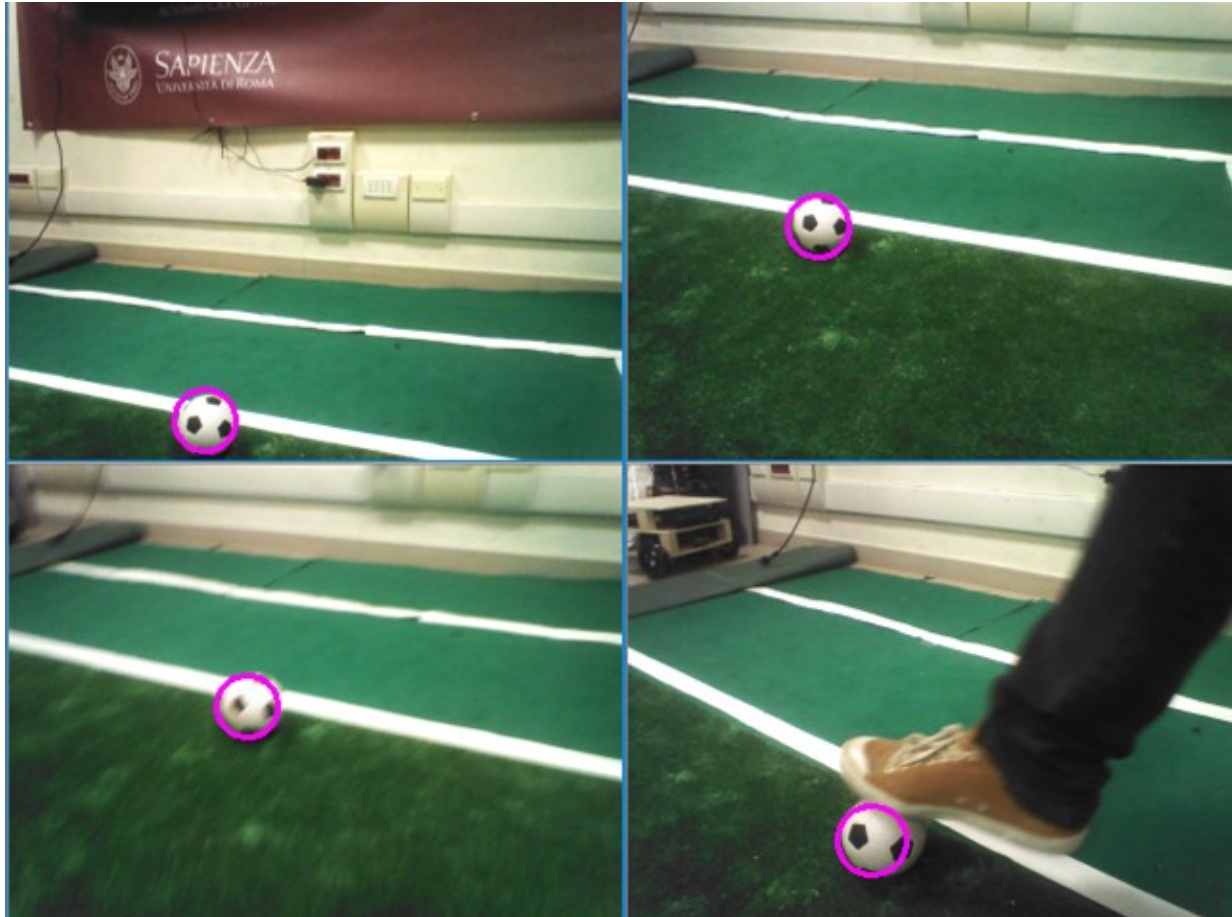
Git is **easy to learn** and has a **tiny footprint with lightning fast performance**. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like **cheap local branching**, convenient **staging areas**, and **multiple workflows**.



# Ottenere codice con Git

---

```
git clone https://github.com/dbloisi/detectball.git
```



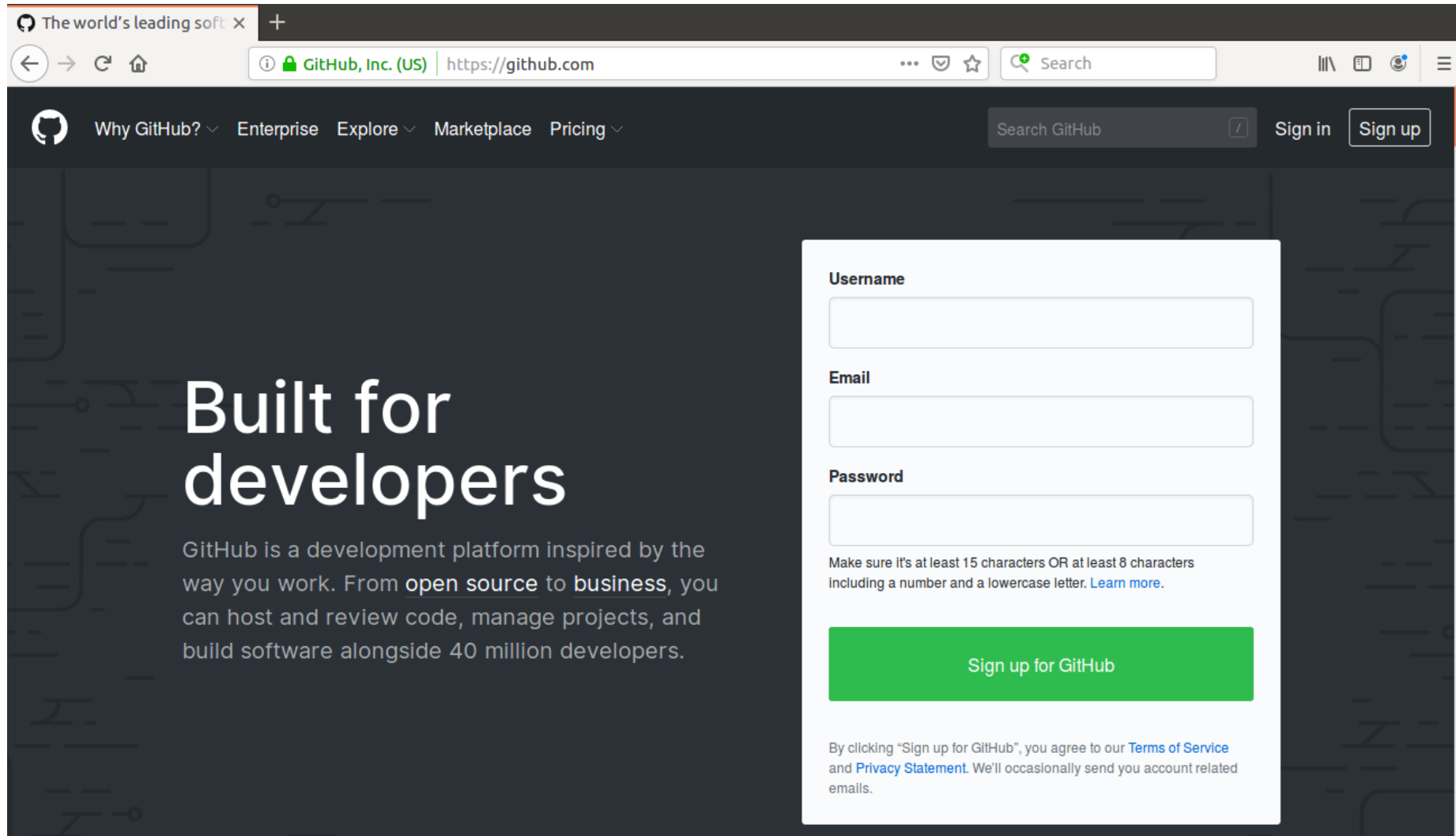
# GitHub

---

- Online git repository
- Free for open source projects



# github.com

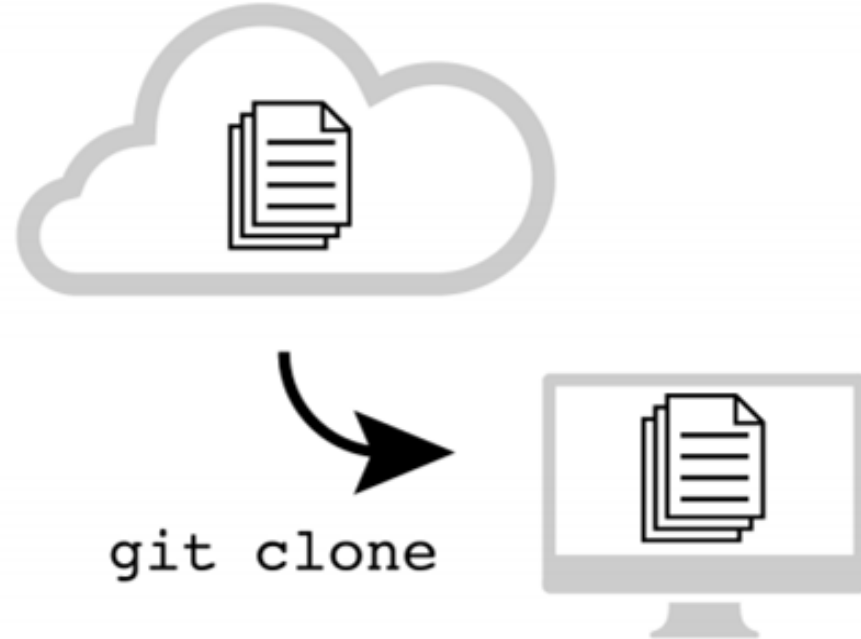


# clone

---

**git clone \$URL**

copy the whole  
repository and it's  
story on the local  
machine



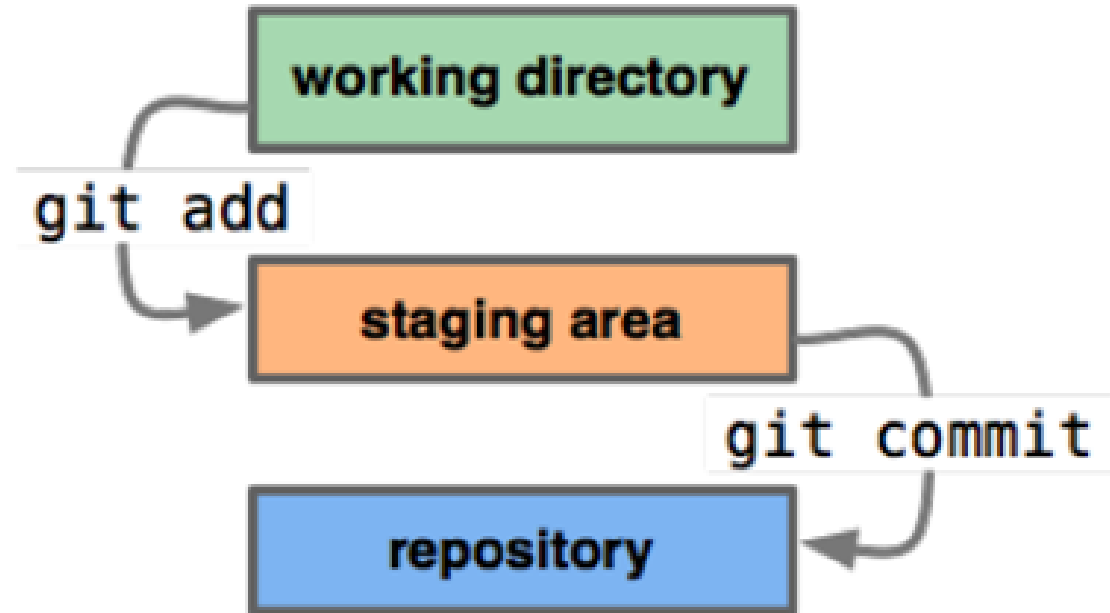
# add e commit

---

**git add \$FILE**

**git commit \$MESSAGE**

the file new release is confirmed and locked in the local repository.

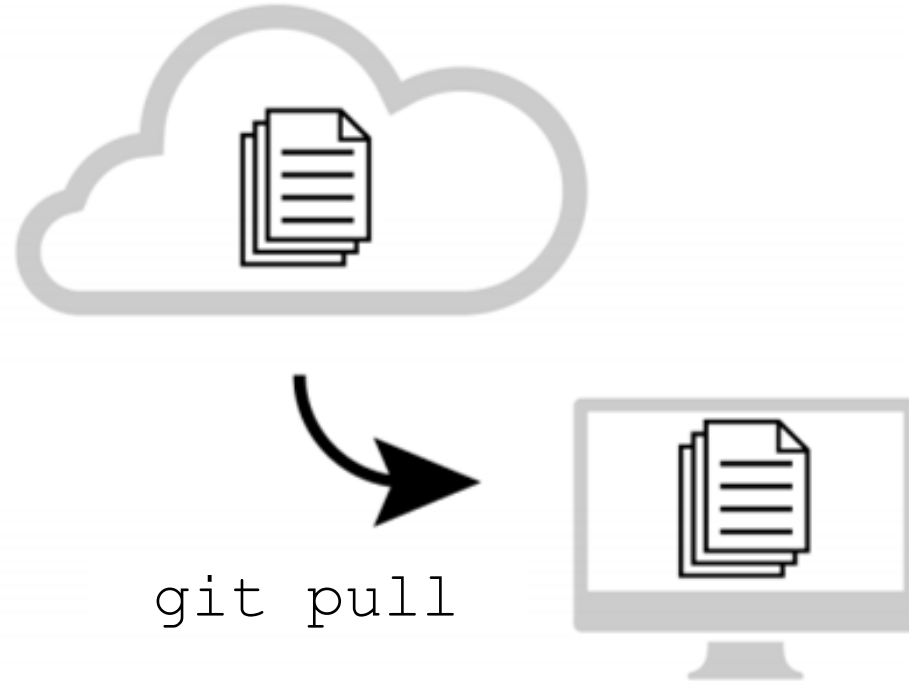


# pull

---

## **git pull**

downloads the  
updated files from the  
remote repository



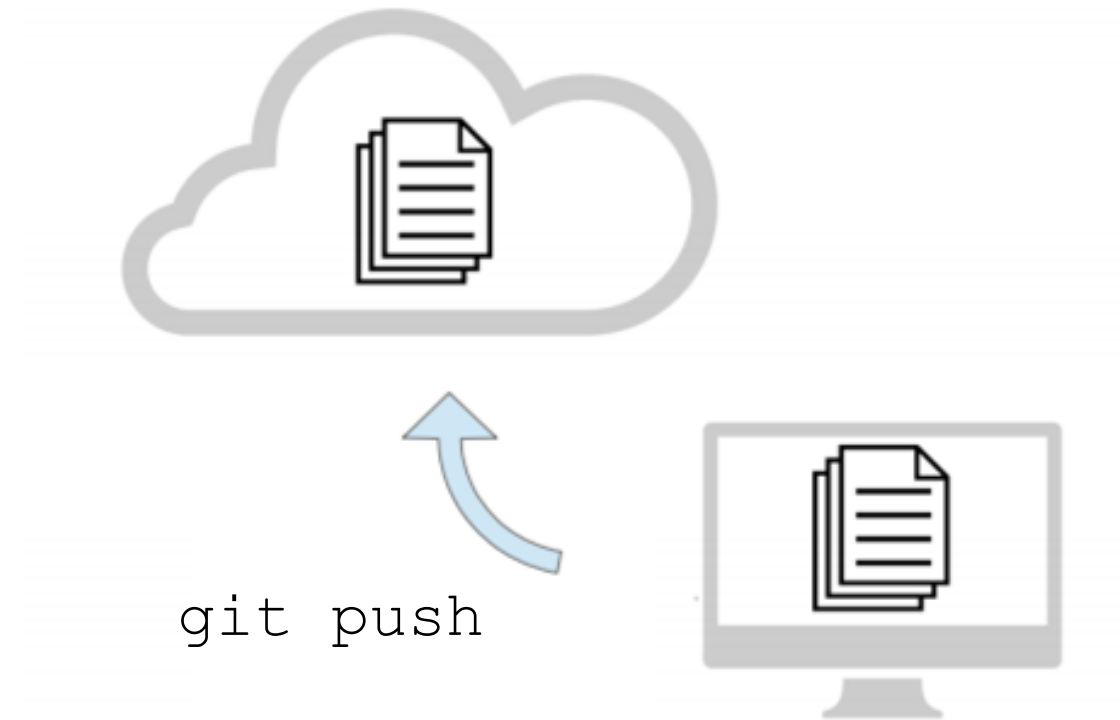


# push

---

## **git push**

sends the committed  
files to the remote  
repository

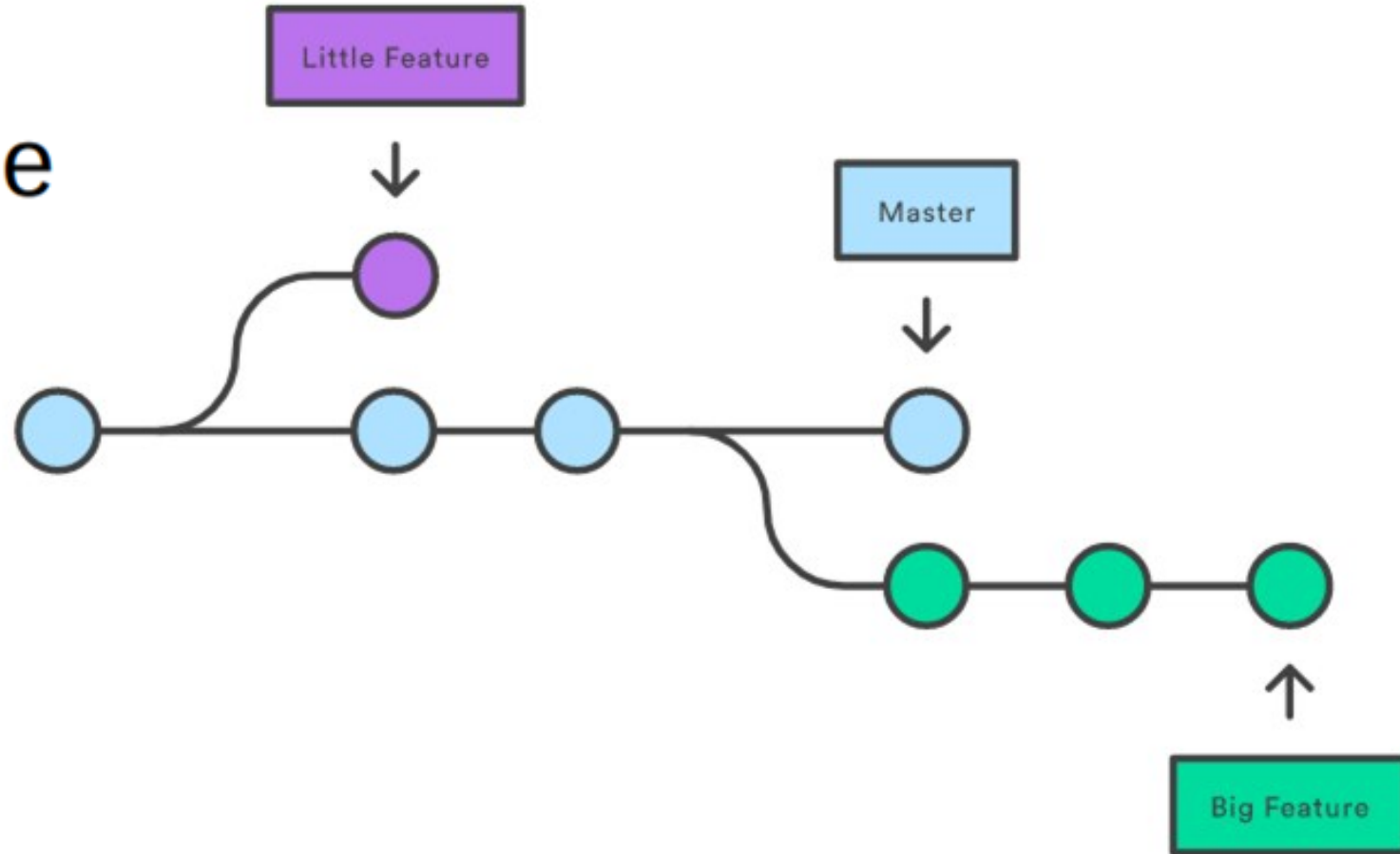


# branch

---

## git branch

list all available  
branches

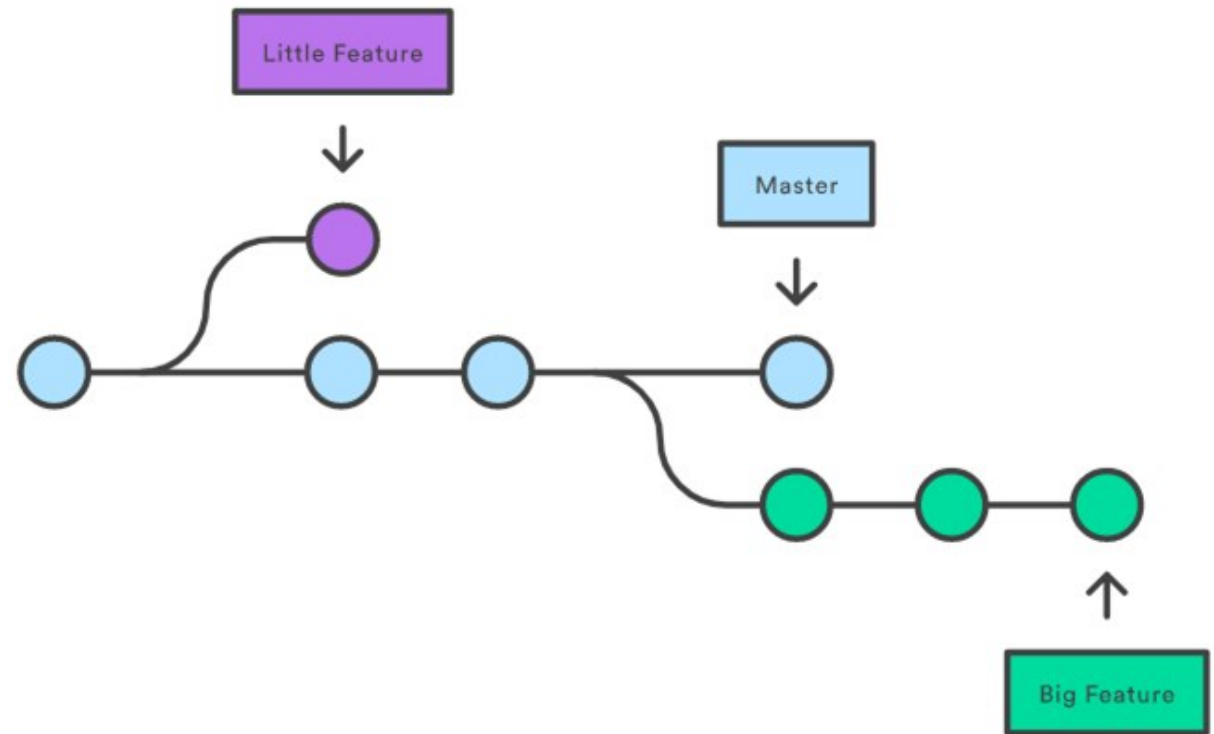


# checkout

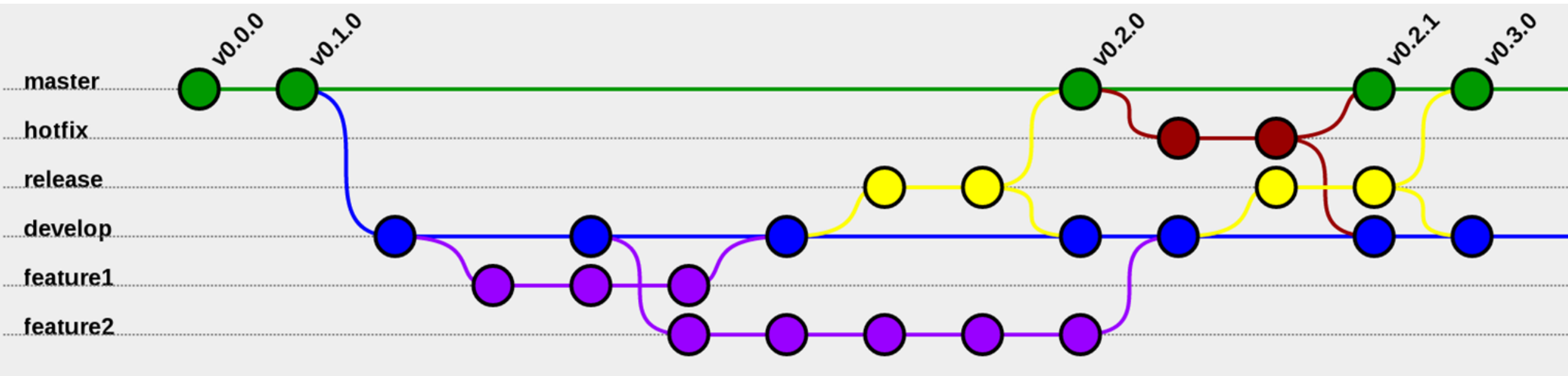
---

**git checkout \$BRANCHNAME**

switch from current branch to  
\$BRANCHNAME



# Esempio



# Sleeping Barber

- Customers
  - N chairs for waiting
- Barber
  - Can cut one customer's hair at any time
  - No waiting customer => barber sleeps
- Customer enters
  - If all waiting chairs full, customer leaves
  - If barber asleep, wake up barber and get hair cut
  - Otherwise (barber is busy), wait in a chair



# Soluzione - Sleeping Barber

```
#define CHAIRS 5
```

```
semaphore customers, barbers;
```

```
mutex lock
```

```
int waiting
```

```
barber {
```

```
    while (TRUE) {
```

→ Sleep if no customers

```
        semWait(customers);
        mutexLock(lock);
        waiting = waiting-1;

        → One barber is ready to cut hair
        semSignal(barbers);
        mutexUnlock(lock);
        cutHair();
    }
```

```
}
```

```
customer {
```

```
    mutexLock(lock);
```

Wake up barbers →

```
    if (waiting < CHAIRS) {
        waiting = waiting+1;
        semSignal(customers);
        mutexUnlock(lock);

        → Wait for barber
        semWait(barbers);
        getHaircut();
    }
    else {
        → If no free chairs, leave
        mutexUnlock(lock);
    }
```

If no free chairs,  
leave

# Dati condivisi - Sleeping Barber

```
#define CHAIRS 5
semaphore customers, barbers;
mutex lock
int waiting
```

```
barber {
    while (TRUE) {
        semWait(customers);
        mutexLock(lock);
        waiting = waiting-1;
        semSignal(barbers);
        mutexUnlock(lock);
        cutHair();
    }
}
```

What is the shared data?  
What part protects the shared data?

```
customer {
    mutexLock(lock);
    if (waiting < CHAIRS) {
        waiting = waiting+1;
        semSignal(customers);
        mutexUnlock(lock);
        semWait(barbers);
        getHaircut();
    } else {
        mutexUnlock(lock);
    }
}
```

# Dati condivisi

What is the shared data?  
What part protects the shared data?



```
barber {  
    while (TRUE) {  
        semWait(customers);  
        mutexLock(lock);  
        waiting = waiting-1;  
        semSignal(barbers);  
        mutexUnlock(lock);  
        cutHair();  
    }  
}
```

Shared  
data

```
#define CHAIRS 5  
semaphore customers, barbers;  
mutex lock  
int waiting  
  
customer {  
    mutexLock(lock);  
    if (waiting < CHAIRS) {  
        waiting = waiting+1;  
        semSignal(customers);  
        mutexUnlock(lock);  
        semWait(barbers);  
        getHaircut();  
    }  
    else {  
        mutexUnlock(lock);  
    }  
}
```

Shared  
data



# Coda di attesa limitata

```
#define CHAIRS 5
semaphore customers, barbers;
mutex lock
int waiting
```

```
barber {
    while (TRUE) {
        semWait(customers);
        mutexLock(lock);
        waiting = waiting-1;
        semSignal(barbers);
        mutexUnlock(lock);
        cutHair();
    }
}
```

What guarantees that not too many customer are waiting?

```
customer {
    mutexLock(lock);
    if (waiting < CHAIRS) {
        waiting = waiting+1;
        semSignal(customers);
        mutexUnlock(lock);
        semWait(barbers);
        getHaircut();
    }
    else {
        mutexUnlock(lock);
    }
}
```


Limits  
number of  
customers

Too many customers?  
Then leave!

# Il semaforo barbiere


```
#define CHAIRS 5
semaphore customers, barbers;
mutex lock
int waiting

barber {
    while (TRUE) {
        semWait(customers);
        mutexLock(lock);
        waiting = waiting-1;
        semSignal(barbers);
        mutexUnlock(lock);
        cutHair();
    }
}
```

 Signal one customer at a time

What guarantees that there is only one customer in the chair?

```
customer {
    mutexLock(lock);
    if (waiting < CHAIRS) {
        waiting = waiting+1;
        semSignal(customers);
        mutexUnlock(lock);
        semWait(barbers);
        getHaircut();
    }
    else {
        mutexUnlock(lock);
    }
}
```

 Wait on barber

# Il semaforo cliente

```
#define CHAIRS 5
semaphore customers, barbers;
mutex lock
int waiting
```

```
barber {
    while (TRUE) {
        semWait(customers);
        mutexLock(lock);
        waiting = waiting-1;
        semSignal(barbers);
        mutexUnlock(lock);
        cutHair();
    }
}
```

What guarantees that the barber doesn't miss a customer?

```
customer {
    mutexLock(lock);
    if (waiting < CHAIRS) {
        waiting = waiting+1;
        semSignal(customers);
        mutexUnlock(lock);
        semWait(barbers);
        getHaircut();
    }
    else {
        mutexUnlock(lock);
    }
}
```

# Cigarette Smokers Problem

---

- There are four processes in this problem: three smoker processes and an agent process.
- Each of the smoker processes will make a cigarette and smoke it. To make a cigarette requires tobacco, paper, and matches.
- Each smoker process has one of the three items. I.e., one process has tobacco, another has paper, and a third has matches.
- The agent has an infinite supply of all three.
- The agent places two of the three items on the table, and the smoker that has the third item makes the cigarette.
- Synchronize the processes.

# Idee per progetti BONUS

---

- Implementazione in C usando i semafori di Pthreads per il problema sleeping barber
- Implementazione in C usando i semafori di Pthreads del problema dining philosophers
- Implementazione in C usando i semafori di Pthreads del problema cigarette smokers