Bug tipici dovuti alla concorrenza (codici che in una situazione single-threaded non danno problemi ma che in una situazione multi-threaded potrebbero dare problemi)

Esempio 1:

```
struct foo *p;

void f() {
        ...
        if (p!=NULL) {
                p->bar = 3;
        }
}
```

Se ci sono altri thread all'interno del processo, un altro thread potrebbe andare a mettere a null quel puntatore; p potrebbe essere == 0.
Questo bug si chiama **violazione dell'atomicità.**

Esempio 2:

```
/* some pointer type */ glob_thread = NULL;

void init()
{
        ...
        glob_thread = create_thread(thread_func, ...);
        ...
}

void thread_func(...)
{
        ...
        foo = glob_thread->bar;
        ...
}
```

Ci sono due thread, uno che ha chiamato la funzione create_thread, e l'altro creato. Quando arrivo alla linea di codice di thread_func, il thread chiamato potrebbe non essere ancora stato inizializzato.
Questo bug si chiama **violazione dell'ordine.**

C'è un'altra categoria di bug chiamata **deadlock**
Supponiamo che ci siano due mutex nel programma (m1 e m2); un thread acquisisce il mutex m1, poi acquisisce il mutex m2 e poi fa qualcosa ecc. Il secondo thread acquisisce il mutex m2, poi acquisisce il mutex m1 e poi fa qualcosa ecc.

```
Thread-A lock(m1); lock(m2);
Thread-B lock(m2); lock(m1);
```

Cosa può succedere?
Si possono incrociare i flussi. Il thread-A non può proseguire perché sta aspettando che qualcuno rilasci m2, e thread-B non può proseguire (e rilasciare m2) perché aspetta m1.

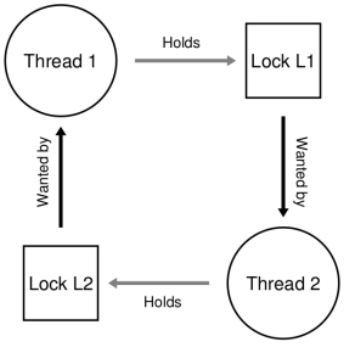

Figure 32.2: **The Deadlock Dependency Graph**

Le condizioni per arrivare ad un deadlock sono:

- La mutua esclusione: se non abbiamo la mutua esclusione non abbiamo il deadlock, ma è difficile ignorare e non usare la mutua esclusione
- Hold-and-wait: i thread mantengono le risorse acquisite mentre ne richiedono/aspettano altre
- No preemption: le risorse non possono essere tolte
- Circular wait: ci deve essere un'attesa circolare

Se rinunciamo ad una di queste preveniamo il deadlock.
I due modi per mantenere la correttezza del programma ed evitare il deadlock:

- Fare in modo che i lock siano sempre acquisiti tutti insieme (se un thread a bisogno di m1 e m2 allora acquisisce entrambi) -> no hold-and-wait
- Do un ordine ai lock e se li acquisisco sono obbligato ad acquisirli in ordine -> no circular wait

SICUREZZA:
Il sistema operativo è quello che gestisce l'hardware e tiene isolati i processi.
Per sviluppare un sistema operativo sicuro, dobbiamo costruirlo su una base sicura -> se il processore non è sicuro, ha delle falle, sicuramente anche il sistema operativo non potrà essere sicuro
Che cosa vuol dire sicuro?
In generale ci sono tre obbiettivi che vanno preservati per ottenere un sistema operativo sicuro:

- La confidenzialità, la segretezza (possono accedere ai dati solo i rispettivi proprietari)
- L'integrità (un altro utente non può verificare il contenuto di un mio file, anche accidentalmente)
- La disponibilità (che il sistema risponde) -> un attacco contro la disponibilità, Denial of Service (DoS) o Distributed Denial of Service (DDoS), fa si che il sistema venga sovraccaricato da richieste farlocche tanto che non riesce a rispondere agli utenti legittimi

Quando un processo fa una system call, il kernel deve valutare la richiesta che viene fatta

- Controlla se è sensata (se il numero passato corrisponde ad una system call e se i parametri sono validi)

- Rispetta la politica di sicurezza che dice chi può fare cosa:
    - L'entità che fa la richiesta, chiamata principal o subject
    - Le richieste sono relative ad un oggetto (l'object)
    - Hanno una modalità di accesso (lettura, scrittura, ecc.)
    - Nel valutare se eseguire o meno una system call, si deve considerare anche il contesto (ad esempio Linux, Android, ecc.)

In Android, ogni app funziona come se fosse un utente Unix diverso
Un utente non parla direttamente con il kernel, ma interagisce con il sistema e lancia i processi che fanno le system call -> deve esserci un'associazione fra utenti e processi e un modo per verificare l'identità degli utenti
Nei sistemi di autenticazione ci sono le cosiddette **AAA** -> <u>authentication</u> (verifica dell'identità), <u>authorization</u> (applicazione della politica di sicurezza, decidere se accettare o rifiutare le richieste) e <u>accounting</u> (fare il logging di quello che succede e gestire il consumo delle risorse)

Identificazione e autenticazione

Argomenti molto vasti
Caso più comune: gli utenti si identificano con uno username, e si autenticano con una password
All'interno dei sistemi Unix ci sono due file che contengono per ogni utente le sue informazioni; in particolare:

- L'user ID (UID), che è un numero che identifica l'utente
- La password

I file sono **/etc/passwd** dove c'è tutto tranne la password, e **/etc/shadow** dove c'è la password

L'UID zero-> root, amministratore del sistema, quindi i processi che girano con UID0 sono privilegiati
Quindi

$$UID = 0 \text{ privilegiati; } UID \neq 0 \text{ non-privilegiati}$$

Autorizzazione

Ci sono due approcci:

- Access control list
    - Ogni oggetto contiene una lista di coppie subject/access
    - Versione ottimizzata Unix in 9 bit

- capabilities (che non c'entrano con le linux capabilities)
    - sono delle chiavi che racchiudono un oggetto in accesso e ci danno il loro privilegio (object/access)

indipendentemente dall'uso di acl o capabilities, chi decide chi può e chi non può accedere ad una risorsa?

- il proprietario, si parla di dac -> discretionary access control (controllo degli accessi discrezionale dove il proprietario del file decide chi può accedere ecc.)
- controllo di accessi imposto da regole, mac -> mandatory access control

un **principio** importante è quello **del minimo privilegio** -> in ogni momento, ciascuna identità (quindi ciascun programma, cliente) dovrebbe avere i permessi minimi per eseguire i suoi compiti. se un programma ha più permessi del necessario, vuol dire che un attaccante può fare più cose

ogni processo ha associato tre user id (uid):

- real uid -> proprietario del processo, chi può uccidere il processo (kill)
- effective uid -> utilizzato per verificare la politica di sicurezza; identità usata per determinare i permessi di accesso a risorse condivise
- saved uid

esiste una funzione setuserid che permette a un processo di modificare l'effective uid, facendolo diventare come il real o come il saved

- se il chiamante della funzione è privilegiato, può modificarli tutti e tre

di solito, quando si utilizza la funzione execve per eseguire un programma, non cambia gli id del processo chiamante; se però il file eseguibile ha un bit dal nome set-userid, allora l'effective uid e il saved uid diventano uguali a quelli del proprietario del file

chroot -> system call che ci permette di modificare il significato di / quando vengono risolti i percorsi di file -> normalmente / fa riferimento alla root del file system; con chroot dico "per questo processo e per i suoi figli il file-system non inizia dalla root ma da una cartella che decido sia la nuova root" -> limita i danni

- permette di creare le cosiddette **chroot jail** che permettono di limitare la visibilità del file-system