Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

TITOLO ITALIANO

TITOLO INGLESE

NOME CANDIDATO

Relatore: *Relatore*
Correlatore: *Correlatore*

Anno Accademico 2014-2015

INDICE

"Inserire citazione"
— Inserire autore citazione

1

INTRODUZIONE

In questa introduzione sarà presente una prima parte che andrà a dare le conoscenze di base minime per comprendere cosa sia un sistema operativo e una piccola classificazione, dopodiché seguiranno la descrizione di alcuni concetti che sono fondamentali per comprendere il resto dell'elaborato.

1.1 COS'È UN SISTEMA OPERATIVO

Un sistema operativo (SO) è un insieme di software che gestisce le risorse hardware e software di un sistema di elaborazione fornendo servizi agli applicativi utente.
In un computer quindi il sistema operativo fornisce l'unica interfaccia diretta con l'hardware e in quanto tale ha un accesso esclusivo con il massimo dei privilegi detto *kernel mode*. Questo comporta che una vulnerabilità all'interno del sistema operativo può portare a gravi conseguenze per l'integrità e la sicurezza del sistema, inoltre qualche malintenzionato potrebbe approfittare di questo bug per trarne profitto. Uno degli obiettivi principali di un SO è quindi quello di garantire la sicurezza, ulteriore scopo è l'efficienza: un buon sistema operativo deve saper sfruttare al meglio tutte le risorse che ha a disposizione, dalla gestione della memoria per sfruttare al meglio lo spazio alla schedulazione dei processi per ottimizzare i tempi di esecuzione. Come ultimo obiettivo, ma non per questo meno rilevante, deve rendere il più semplice possibile l'utilizzo del dispositivo su cui è installato. Dalla definizione di SO data a inizio capitolo possiamo isolare una specifica parte di codice che è quella che permette al software di interfacciarsi con l'hardware, quindi l'accesso e la gestione delle risorse di un dispositivo, questa specifica parte si chiama *kernel* che come suggerisci il nome (nocciolo dall'inglese) rappresenta la parte centrale di un sistema operativo su cui tutto il resto si appoggia.

1.2 MICROKERNEL E KERNEL MONOLITICI

Esistono vari modelli strutturali per i sistemi operativi: monolitici, modulari, a livelli, microkernel ed ibridi, ad oggi i più diffusi sono gli ibridi, che combinano i vari modelli tra di loro, ma che in gran parte si basano su sistemi monolitici i quali consistono di un unico file binario statico al cui interno sono definite tutte le funzionalità del kernel e che viene eseguito in un unico spazio di indirizzi, questo comporta dei vantaggi:

- efficienza → motivo principale per cui la maggior parte dei sistemi operativi ancora si basano su kernel in gran parte monolitici, lavorando nello stesso spazio di indirizzi e gestendo tutto attraverso chiamate a procedura il SO risulterà molto reattivo e performante

- semplicità → in quanto non ha una vera e propria strutturazione ma il codice è tutto in un unico file binario risulta chiaramente più semplice da progettare anche se poi l'implementazione risulta difficile

d'altra parte ha anche degli svantaggi:

- inserimento di un nuovo servizio → questo richiede la ricompilazione del kernel, quindi non permette l'inserimento di un nuovo servizio a runtime (problema risolto nei modelli ibridi)

- dimensione → dovendo gestire tutte le principali funzionalità del sistema operativo il kernel sarà composto da milioni di righe di codice sorgente (MSLOC - linux ha circa 20MSLOC) e questo porta direttamente al successivo grosso svantaggio

- sicurezza → maggiore è il numero di righe di codice maggiore sarà il numero di possibili bug, essendo tutto il codice eseguito nello stesso spazio di indirizzi un bug rischia di far bloccare l'intero sistema anche se il problema è molto piccolo e isolato a una minima funzione del kernel.

All'estremo opposto troviamo i *microkernel* che sono composti da un kernel ridotto al minimo indispensabile, che comprende la gestione della memoria, dei processi e della CPU, le comunicazioni tra processi (IPC) e l'hardware di basso livello, mentre tutto il resto deve essere gestito da server (deamon) che operano sopra al kernel, quindi in spazi di indirizzi separati.
I microkernel sono spesso usati in sistemi embedded in applicazioni

mission critical di automazione robotica o di medicina, a causa del fatto che i componenti del sistema risiedono in aree di memoria separate, private e protette.
Anche questo modello ha dei vantaggi:

- flessibilità → l'inserimento di un nuovo servizio avviene al di sopra del kernel quindi in qualsiasi momento è possibili aggiungere o togliere servizi senza dover modificare il kernel.

- sicurezza → minore quantità di codice eseguita in kernel mode (quindi minore quantità di bug e minore superficie attaccabile) maggiore è la sicurezza del sistema, inoltre i servizi sono lavorano in uno spazio di indirizzi differente da quello del kernel di conseguenza se un server (su cui viene eseguito un servizio) smette di funzionare tutto il resto del sistema continua a funzionare normalmente e si potrà procedere a riavviare quel singolo servizio

- semplicità → essendo il codice composto da giusto qualche decina di migliaia di righe di codice (KSLOC) risulta molto più facile da scrivere

e dall'altro lato ha un grande svantaggio:

- efficienza → dato che ogni servizio gira a livello utente l'utilizzo di uno qualsiasi di questi richiede il ricorso a chiamate di sistema che rallentano fortemente l'esecuzione di ogni operazione, motivo principale per cui ancora oggi i sistemi operativi si basano in gran parte su sistemi monolitici.

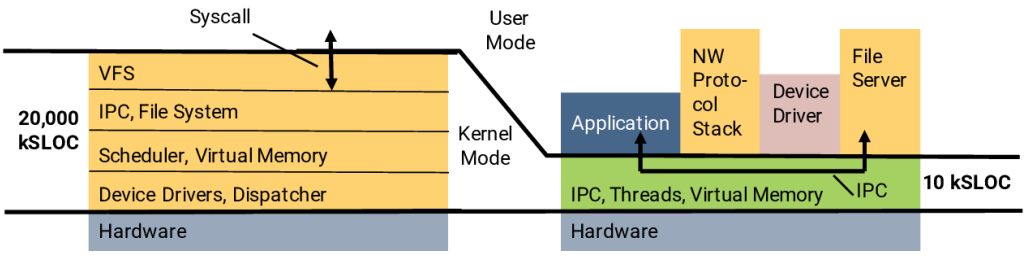


Figura 1: Kernel monolitici (sinistra) VS microkernel (destra)

1.3 SCHEDULING

Proseguendo la lettura verrà nominato lo scheduling: solitamente con il solo termine scheduling si intende quello a breve termine della CPU, cioè la funzionalità che determina quale tra i processi (thread) in attesa della CPU la otterranno, ci sono vari metodi che si differenziano per modalità e prestazioni, gli algoritmi che traducono questi metodi si chiamano politiche di scheduling. Una particolare politica di scheduling rilevante per questo testo è Round Robin o scheduling circolare: consiste nel determinare una quantità di tempo (time slice) nella quale il processo ottiene la CPU al termine del quale il processo viene interrotto e inserito infondo alla lista dei processi in attesa, in questo modo tutti i processi ottengono la CPU per al più un tempo massimo stabilito ed è anche possibile stabilire un tempo massimo di attesa che un processo dovrà attendere in base a quanti processi lo precedono.

1.4 HYPERVISOR

Un *hypervisor*, chiamato anche virtual machine monitor (VMM), è un tipo di sotware/firmware che permette di creare ed eseguire macchine virtuali. Un computer sul quale un hypervisor esegue una o più macchine virtuali prende il nome di host machine mentre le singole macchina virtuali prenderanno il nome di guest machine, su ognuna di queste è possibile far girare un sistema operativo diverso che eseguirà la maggior parte delle istruzioni direttamente sulle risorse hardware virtualizzate rese disponibili dall'hypervisor.

2

SEL4

seL4 fa parte della famiglia dei microkernel L4 che risalgono alla prima metà degli anni ’90 creato da Jochen Liedtke per sopperire alle scarse performance dei primi sistemi operativi basati su microkernel, ad oggi fa parte del Trustworthy System.
Come descritto poco sopra nell’introduzione, seL4 essendo un microkernel, ha un numero di righe di codice sorgente estremamente piccolo e questo è sufficiente per determinare che non è un sistema operativo ma soltanto un microkernel, infatti non fornisce nessun dei servizi che siamo solitamente abituati a trovare su un comune SO, "è solo un sottile involucro attorno all’hardware" [3], tutti i servizi devono essere eseguiti in modalità utente e questi dovranno essere importati ad esempio da sistemi operativi open-source come Linux (oppure scritti da zero). Data questa sua "incapacità" nel fornire servizi all’utente seL4 è anche un *hypervisor*, quindi è possibile eseguire macchine virtuali sulle quali far girare un comune SO che fornirà i servizi non presenti in seL4.

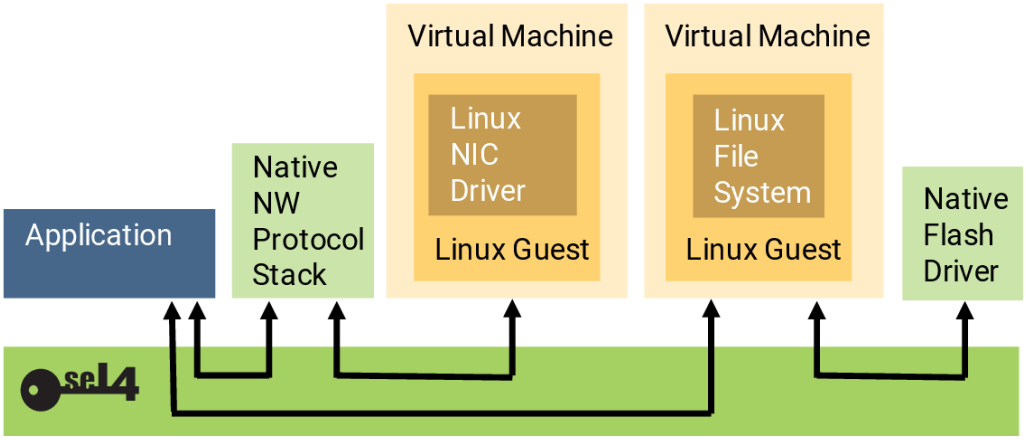


Figura 2: Virtualizzazione del SO Linux per l’integrazione dei servizi di networking e file system

Un esempio pratico può essere quello mostrato in Figura 1, in cui è raffigurato seL4, una generica applicazione e due macchine virtuali (VM) sulle quali viene eseguita una versione ridotta al minimo di Linux, che quindi avrà poco più oltre al servizio che dovrà eseguire. Queste due VM forniranno all'applicazione il servizio di networking ed il file system per la gestione della memoria secondaria (hard disk, supporti rimovibili ecc.), le comunicazioni tra le parti saranno gestite da un canale fornito dal microkernel ma le due macchine virtuali non avranno modo di comunicare tra di loro, non solo questo ma come si vede in figura anche le comunicazioni tra le varie parti e l'applicazione sono ben delineate e precise, nessun'altra comunicazione tra le varie parti è possibile al di fuori di quelle indicate dalle frecce.

2.1 CAPABILITY

Una *capability* è definita formalmente come un riferimento ad oggetto, nel nostro caso specifico possiamo definirla anche come un puntatore immutabile, cioè una capability farà sempre riferimento allo stesso oggetto. SeL4 è un sistema capability-based (basato sulle capability) questo significa che l'unico modo per eseguire un operazione è attraverso l'invocazione di una capability. Ad ognuna di esse, inoltre, sono associati dei diritti di accesso, quindi una capability è un incapsulamento di un riferimento ad oggetto con i diritti ad esso conferiti. Per dare una definizione meno formale possiamo pensare alle capability come a delle chiavi di accesso estremamente specifiche riguardo quale entità può accedere ad una particolare risorsa del sistema. Inoltre permettono di supportare il *principle of least privilege*, principio del privilegio minimo chiamato anche *principle of least authority* PoLA, questo principio implica che ogni modulo deve avere accesso solo ed esclusivamente alle risorse strettamente necessarie al suo scopo. In seL4 quindi i diritti dati ad un componente possono essere ristretti al minimo indispensabile per svolgere il loro lavoro come richiesto dal PoLA e chiaramente questo è un grosso punto a favore per quanto riguarda la sicurezza.
Nei sistemi operativi più comuni tipo Windows o Linux l'accesso alle risorse è gestito dalle *access-control list* (ACL), quindi, nel caso specifico di Linux, ad ogni file viene associato un set di bit che determinano quali operazioni (lettura, scrittura, esecuzione) possono essere eseguite su di esso dai vari utenti (proprietario, gruppo, altri) questo però implica che ogni file del sistema con lo stesso set di permessi è a disposizione di quello specifico utente, quindi se ci mettiamo nello scenario di voler

avviare un programma, di cui non siamo sicuri della sua attendibilità, su uno specifico file questo non è possibile perchè come può accedere a quel file può accedere anche a tutti gli altri che hanno le stesso ACL.
Con le capability questo scenario non si può presentare perchè il kernel consentirebbe un operazione se e solo se chi richiede di eseguire l'operazione ha la "giusta capability" per eseguire l'operazione su quel file.

2.1.1 *Proprietà delle capability*

Interposition → le capability hanno la proprietà di mettersi in mezzo (interpose) tra chi crea una capability e l'effettivo accesso ad una risorsa: se un utente dà una capability ad un oggetto esso non è in grado di sapere cosa effettivamente sia quell'oggetto, può chiaramente utilizzarlo senza però sapere che tipo di oggetto sia.
Delegation → le capability supportano la delegazione dei privilegi tra gli utenti: l'utente X ha un oggetto e vuole dare accesso ad esso anche all'utente Y; X può creare una nuova capability e darla ad Y senza conservare nessun riferimento all'utente X che l'ha creata, la nuova capability può anche avere meno diritti di accesso (esempio solo lettura invece di lettura e scrittura) e inoltre X in qualsiasi momento può revocare l'accesso ad Y distruggendo la capability.

2.2 HARD REAL-TIME SYSTEMS

Un *Hard Real-Time System* è un sistema in cui il mancato rispetto di una scadenza può portare al fallimento dell'intero sistema. Un esempio molto alla mano di tutti e semplificato può essere l'autopilot di un'automobile; un veicolo dotato di un software di guida autonoma richiede la presenza di un numero estremamente elevato di sensori esterni ed interni al veicolo e il computer di bordo deve leggere, elaborare e dare una risposta immediata ad ogni minimo cambiamento di un valore proveniente da questi sensori, se ad un certo punto l'elaborazione di un dato richiede più del tempo dovuto, anche solo di qualche millisecondo, c'è il rischio che questo comporti una serie di ritardi a catena che ad esempio portano al non rilevamento di un oggetto che si sta avvicinando al veicolo, oppure alla mancata correzione della traiettoria e quindi l'abbandono della carreggiata, con conseguenze anche catastrofiche. → Lo scheduling dei processi in seL4 è basato sulla priorità, il kernel di sua iniziativa non

cambierà mai la priorità di un processo, è sempre decisa dall'utente.
→ SeL4 quando esegue delle operazioni in modalità kernel queste sono esenti dagli interrupt, all'apparenza questo può sembrare catastrofico se non fosse per il fatto che le chiamate di sistema sono tutte brevi, solo la revocazione di una capability può richiedere tempi più lunghi ma in presenza di queste operazioni seL4 adotta una politica di divisione dell'esecuzione in sotto operazioni più brevi, ed inoltre ognuna di esse può essere annullata e poi ripresa da quel punto in poi, così da poter gestire degli eventuli interrupt in attesa.
Questi due punti appena elencati sono requisiti fondamentali per gli Hard Real-Time system, scheduling dei processi basato sulla priorità, che sia quindi facilmente analizzabile, e latenza degli interrupt limitata, essendo gli interrupt disabilitati non ci sarà nessuna latenza dovuta al cambio di contesto per gestire subito l'interrupt e dato che le operazioni sono tutte brevi questo non risulta un problema. Per seL4 è stata eseguita una worst-case execution time (WCET), questo vuol dire che è stato determinato un limite superiore di latenza di ogni system call nel caso peggiore e ciò implica anche il caso peggiore di latenza di un interrupt

2.2.1 *Mixed-criticality systems*

Un mixed-criticality system (MCS) è un sistema fatto da più componenti che interagiscono tra di loro e che hanno differenti livelli di criticità, in questi sistemi è imperativo che il fallimento di un componente non influenzi gli altri componenti critici, che questi siano quindi isolati e protetti da componenti meno critici.
Un approccio classico per questo tipo di sistemi è isolare le criticità sia per quanto riguarda il tempo che lo spazio, chiamato anche *strict time and space partitioning* (TSP), ma questo implica dover assegnare staticamente l'area di memoria, il tempo di esecuzione e quindi lo scheduling, e per farlo si utilizzano dati misurati precedentemente nel caso pessimo. Essendo sistemi real-time, come già detto prima, ogni operazione deve avere dei limiti di tempo quindi un operazione a cui è stato misurato, nel caso peggiore, 5 millisecondi deve avere questa durata, non 4ms nè tantomeno 6ms. Chiaramente determinando staticamente i tempi e gli spazi nel caso peggiore siamo sicuri che questi vengano rispettati ma fortunatamente non sempre si presentano dei casi pessimi e quindi porta ad uno scarso utilizzo delle risorse, inoltre la latenza di un interrupt nel caso pessimo può essere molto costosa.
SeL4 supporta il mixed-criticality system: per quanto riguarda l'isolamen-

to abbiamo già visto che le capability, in termini di spazio, intrinsecamente lo garantiscono. Resta quindi da esaminare il comportamento da un punto di vista temporale.
SeL4 normalmente utilizza due parametri per gestire lo scheduling dei processi: la priorità e la quantità di tempo; la *priorità* determina l'ordine di esecuzione dei processi mentre la *quantità di tempo* (time slice) determina quanto tempo il kernel lascerà in esecuzione un thread prima di stopparlo per selezionare un altro processo con più alta priorità con una politica round-robin tra i livelli di priorità.
La versione MCS di seL4 si comporta diversamente, l'accesso al processore viene controllato dalle capability, un componente può ottenere la CPU solo se ha una capability che glielo permette e il tempo di esecuzione è codificato in essa, questa politica si chiama *scheduling-context capabilities*. Lo scheduling-context contiene due attributi principali:

1. *time budget* che sostituisce il vecchio time slice
2. *time period* che determina invece quante volte un budget può essere usato per periodo, in questo modo viene evitato che un processo monopolizzi la CPU indipendentemente dalla sua priorità

2.3 SICUREZZA E PERFORMANCE

Come già detto nelle prime righe di questo capitolo la famiglia dei microkernel L4 nasce per sopperire alle scarse performance dei suoi predecessori, finora è stata fatta una descrizione del funzionamento generale di seL4 con particolare riguardo sulla sicurezza di questo sistema; per chi è dell'ambito sa già che spesso sicurezza e buone performance non vanno molto d'accordo, garantire la sicurezza vuol dire attenersi a regole ben precise e controlli che spesso poi portano a rallentamenti e quindi vanno a influire sulle performance di un sistema, quindi è lecito domandarsi se questo microkernel sia performante oppure no.
Nonostante non fosse nelle prerogative delle sviluppo di seL4 questo, alla fine, si è rivelato il più performante dei microkernel della famiglia L4, inoltre sono state fatte altre pubblicazioni indipendenti che mettono a confronto seL4 con altri microkernel per studiarne le performance, in particolare Fiasco.OC, Zicron [11] e CertiKOS [2], confrontando i costi dell'IPC si può vedere che seL4 ha un bel vantaggio anche di oltre un fattore due rispetto agli altri microkernel.

3

IMPOSTAZIONE DI SEL4

Come primo approccio per arrivare alla scrittura di questa tesi ho innanzitutto fatto una ricerca sulla letteratura che si trova disponibile riguardo a seL4, nonostante sia poca e principalmente fornita da Trustworthy (TS) stesso comunque sufficiente per avere una conoscenza abbastanza approfondita del microkernel.
SeL4 è un sistema open-source dunque lo step successivo è stato quello di scaricare seL4 e sperimentare con mano le funzionalità, ovviamente questo ha richiesto un approfondimento più tecnico e specifico, rispetto a quanto fatto finora, di alcuni aspetti come la gestione della memoria fisica e virtuale, l'IPC ecc. che verranno trattati in questo capitolo.

3.1 PREREQUISITI

Come prima cosa ho installato sul mio portatile VirtualBox in quanto come consigliato dalle guide fornite da TS sarebbe ottimale lavorare in ambiente Linux, non avendo una partizione del portatile con Linux ho inizialmente pensato di utilizzare una macchina virtuale così da lasciare inalterato il mio computer e comunque avere a disposizione un sistema operativo Linux su cui lavorare. Andando avanti con il set-up del sistema per iniziare a lavorare su seL4 però ho incontrato una prima difficoltà che è stata lo spazio: purtroppo lo spazio nel portatile non era tantissimo, la macchina virtuale, considerando il sistema operativo e l'installazione dei vari prerequisiti per poter far girare il microkernel, cominciava ad occupare molto spazio, dunque ho dovuto cercare un'alternativa; Per sopperire al problema mi sono procurato un SSD su cui sono andato a copiare la partizione creata in VirtualBox continuando la sperimentazione sul microkernel lavorando sull'SSD esterno collegato via USB.
Per lavorare su seL4 è necessario avere installato sul sistema dei programmi che simulino un'architettura su cui farlo girare, per fare ciò è necessario installare delle dipendenze (prerequisiti) cioè compilatori,

emulatori software vari e librerie che devono essere installate affinché sia possibile utilizzare seL4.
Prima di tutto ho installato Google repo, così da poter clonare i repository git:

```
sudo apt-get install repo
```

build-essential, cmake, ninja, curl, python e QEMU abbreviazione di Quick EMUlator, un emulatore open-source che permette di emulare un'architettura informatica e che permette di simulare diversi sistemi operativi, in questo caso è fondamentale perchè permette l'esecuzione di seL4:

```
sudo apt-get install build-essential
sudo apt-get install cmake ccache ninja-build cmake-curses-gui
sudo apt-get install libxml2-utils ncurses-dev
sudo apt-get install curl git doxygen device-tree-compiler
sudo apt-get install u-boot-tools
sudo apt-get install python3-dev python3-pip python-is-python3
sudo apt-get install protobuf-compiler python3-protobuf
sudo apt-get install qemu-system-arm qemu-system-x86 qemu-system-misc
pip3 install --user setuptools
pip3 install --user sel4-deps
```

Altro componente fondamentale è CAmkES (component architecture for microkernel-based embedded systems), un framework per realizzare velocemente sistemi multiserver affidabili basati su microkernel:

```
pip3 install --user camkes-deps
curl -sSL https://get.haskellstack.org/ | sh
sudo apt-get install haskell-stack
sudo apt-get install clang gdb
sudo apt-get install libssl-dev libclang-dev libcunit1-dev libsqlite3-dev
sudo apt-get install qemu-kvm
```

Dopodiché sono passato alle dipendenze per l'installazione di Isabelle (theorem prover) che serve per la verifica automatica di sistemi software e hardware:

```
sudo apt-get install \
    python3 python3-pip python3-dev \
    gcc-arm-none-eabi build-essential libxml2-utils ccache \
    ncurses-dev librsvg2-bin device-tree-compiler cmake \
    ninja-build curl zlib1g-dev texlive-fonts-recommended \
    texlive-latex-extra texlive-metapost texlive-bibtex-extra \
    mlton-compiler haskell-stack repo
```

Ancora dipendenze Python e Haskell

```
pip3 install --user --upgrade pip
pip3 install --user sel4-deps

stack upgrade --binary-only
which stack # should be $HOME/.local/bin/stack
stack install cabal-install
```

Con questa serie di comandi bash il sistema operativo Linux, per la precisione Ubuntu VERSIONE, ha tutti i prerequisiti necessari per procedere alla configurazione.

3.2 CONFIGURAZIONE

Lo step successivo è stato quello di recuperare, attraverso repo, la collezione di repository necessari per la verifica di seL4; in particolare contiene il sorgente del kernel, il theorem prover Isabelle/HOL e HOL4 e lo strumento di verifica binaria.

```
mkdir verification
cd verification
repo init -u https://git@github.com/seL4/verification-manifest.git
repo sync
```

A questo punto si avrà quindi una cartella con questa struttura:

```
verification
├── HOL4/
├── graph-refine/
├── isabelle/
├── l4v/
└── seL4/
```

Il che indica che l'importazione delle repository è andata a buon fine, quindi possiamo procedere alla configurazione di Isabelle posizionandosi nella cartella `l4v`:

```
mkdir -p ~/.isabelle/etc
cp -i misc/etc/settings ~/.isabelle/etc/settings
./isabelle/bin/isabelle components -a
./isabelle/bin/isabelle jedit -bf
./isabelle/bin/isabelle build -bv HOL
```

Questa serie di comandi bash daranno come risultato:

- la creazione di una cartella per le impostazioni utente di Isabelle.

- installazione delle impostazione Isabelle per L4.verified [1] il rt-bj6nmgquale è una repository che contiene formalismi per la verifica di seL4.
- download di Scala, Java JDK, PolyML ed altri dimostratori (prover) esterni.
- compilazione del Prover IDE (PIDE) jEdit di Isabelle

3.3 AVVIO DI SEL4

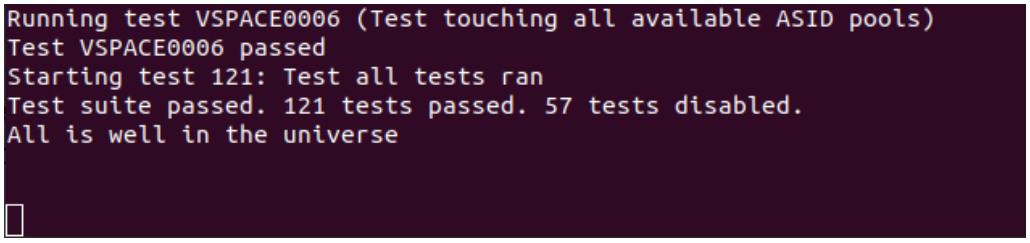
Terminata la prima fase di installazione dei prerequisiti e di configurazione mi sono procurato ciò che servirà poi per eseguire i test delle varie funzionalità di seL4:

```
mkdir seL4test
cd seL4test
repo init -u https://github.com/seL4/sel4test-manifest.git
repo sync
```

Con questi comandi si va a creare una directory `seL4test` al cui interno, attraverso il comando `repo`, che conterrà tutte le direttive e le librerie necessarie per eseguire i vari test (quindi anche il kernel stesso).
Dopodichè è stato necessario creare una cartella `build-x86` di configurazine per QEMU in modo da indicargli il target su cui eseguire le simulazioni:

```
mkdir build-x86
cd build-x86
../init-build.sh -DPLATFORM=x86_64 -DSIMULATION=TRUE
ninja
```

Il comando `ninja`, che si vedrà spesso a seguire, è un assembler che permette di fare il build di sistemi anche complessi molto velocemente.
A questo punto è possibile eseguire il comando `./simulate` che farà partire la simulazione e dopo una lunga serie di test (IPC, chiamate di sistema, thread...) che appariranno nel terminale concluderà, se tutto è andato a buon fine con:
`All is well in the universe`


Running test VSPACE0006 (Test touching all available ASID pools)
Test VSPACE0006 passed
Starting test 121: Test all tests ran
Test suite passed. 121 tests passed. 57 tests disabled.
All is well in the universe

BIBLIOGRAFIA

[1] SeL4 Foundation. l4v, 2023. Ultima modifica 19 luglio 2023. (Cited on page 18.)

[2] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, , and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels, 2016. USENIX Symposium on Operating Systems Design and Implementation. (Cited on page 13.)

[3] Gernot Heiser. The sel4 microkernel - an introduction. *The seL4 Foundation*, Revision 1.2, 2020. (Cited on page 9.)

[4] JavaTpoint. Hard and soft real-time operating system.

[5] Wikipedia, l'enciclopedia libera. Hypervisor, 2023. Ultima modifica 25 luglio 2023.

[6] Wikipedia, l'enciclopedia libera. Isabelle (proof assistant), 2023. Ultima modifica 1 marzo 2023.

[7] Wikipedia, l'enciclopedia libera. Kernel, 2023. Ultima modifica 7 giu 2023.

[8] Wikipedia, l'enciclopedia libera. Operating system, 2023. Ultima modifica 16 luglio 2023.

[9] Wikipedia, l'enciclopedia libera. Principle of least privilege, 2023. Ultima modifica 2 agosto 2023.

[10] Wikipedia, l'enciclopedia libera. Qemu, 2023. Ultima modifica 30 giugno 2023.

[11] Zeyu Mi, Dingji Li, Zihan Yang, XinranWang, , and Haibo Chenh. Skybridge: Fast and secure inter-process communication for microkernels, 2019. EuroSys Conference. (Cited on page 13.)

[12] Rosario Pugliese. Chiedere al prof come citare le sue slide.