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

Tesi di Laurea

STUDIO E SPERIMENTAZIONE DEL MICROKERNEL SEL4

STUDYING AND EXPERIMENTING WITH THE SEL4 MICROKERNEL

ELIA MATTEINI

Relatore: *Rosario Pugliese*

Anno Accademico 2022-2023

INDICE

"Inserire citazione"
— Inserire autore citazione

I

INTRODUZIONE

In questa tesi andrò ad affrontare uno studio del *microkernel seL4*, sia da un punto di vista descrittivo sia da un punto di vista più tecnico.

La fase di partenza per la scrittura di questo elaborato è stata quella di svolgere una ricerca sulla letteratura disponibile riguardo a seL4, nonostante sia poca e principalmente fornita dalla *seL4 Foundation* stessa, ma che risulta comunque sufficiente per avere una conoscenza abbastanza approfondita del *microkernel*.

Ho quindi iniziato a farmi un'idea generale dell'argomento attraverso il *whitepaper* [5] di presentazione, il quale a grandi linee dà una visione a trecentosessanta gradi di seL4, entrando in alcune parti anche in ambito specifico e tecnico. Ovviamente in parallelo a questa lettura ho dovuto affiancare un approfondimento sul *kernel* e in generale sui sistemi operativi, in quanto prima di approcciarmi a questa tesi non avevo una conoscenza così specifica sul *kernel* e in particolare sui *microkernel*.

Nel mio percorso universitario avevo già seguito un corso che trattava i sistemi operativi, ma questa esperienza mi ha dato la possibilità di vedere la materia da prospettiva diversa. Quasi tutte le conoscenze che sono state necessarie per intraprendere questa ricerca erano già state acquisite ma spesso, quando si studia a livello teorico, capita che alcuni concetti siano magari chiari, ma fini a se stessi. Indagare nello specifico un singolo sistema mi ha permesso di capire come tutti i vari pezzi cooperino e si incastrino, come in un puzzle per arrivare al disegno finale.

Dopo queste prima fase di analisi mi sono cimentato più sugli aspetti tecnici, guardando ed esaminando quindi le chiamate di sistema, le varie funzioni che sono disponibili nelle API (*Application Programming Interface)*, mettendo anche mano direttamente su di esse. In particolare, sono andato a svolgere dei *tutorial* che la *seL4 Foundation* mette a disposizione appositamente per permettere a chi si approccia al *microkernel*, di prendere confidenza con esso. Questi *tutorial* sono divisi in sezioni e ognuno di essi tratta un argomento diverso; consistono in un programma, in

parte non funzionante, che va prima di tutto compreso e successivamente modificato, in modo tale che funzioni correttamente.

Ovviamente questa seconda fase è stata quella più duratura e più formativa, in quanto è molto complicato, per chi non ha mai programmato a livello *kernel*, fare questo tipo di esperienza. Questi esercizi sono oltretutto strutturati in modo tale da consentire un approfondimento specifico sulla parte trattata da quell'esercitazione; ad esempio l'utilizzo della memoria virtuale: come fare il *mapping*, le varie fasi intermedie e tutte le funzioni che permettono di utilizzarla. In questo stadio ho quindi avuto modo di mettere in pratica il mio bagaglio di conoscenze, vedendo effettivamente come si implementano certi processi.

Nel proseguimento della lettura, seguirà una parte iniziale (capitolo 1) che serve per fornire delle conoscenze di base sui sistemi operativi, così da poter creare un *background* di informazioni, che dovrebbero essere sufficienti per la comprensione del resto della tesi.

Successivamente, col capitolo 2 si entrerà nello specifico del *microkernel* seL4, in cui si troverà una descrizione generale del funzionamento, approfondito tecnicamente in tutte le varie parti di cui è composto.

Nell'ultimo capitolo invece sarà il momento della sperimentazione sul *microkernel*; ci saranno una serie di sorgenti C con vari errori, che dovranno essere corretti in modo da renderli funzionanti. Questi programmi utilizzano oggetti e funzioni creati appositamente per la gestione dei processi e di tutte le funzionalità di seL4.

1

BACKGROUND

In questo capitolo sarà presente una prima sezione che andrà ad offrire le conoscenze di base minime per comprendere cosa sia un sistema operativo e una breve classificazione di essi. Dopodiché seguirà la descrizione di alcuni concetti, come ad esempio lo *scheduling della CPU*, che sono fondamentali per comprendere il resto dell'elaborato.

1.1 COS'È UN SISTEMA OPERATIVO

Un sistema operativo (SO) è un *software* che gestisce le risorse *hardware* e *software* di un sistema di elaborazione, fornendo servizi agli applicativi utente.

In un computer, esso 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 illecito 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 in modo ottimale 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. All'interno di un SO si può isolare una specifica parte di codice, che è quella che permette al *software* di interfacciarsi con l'*hardware* e quindi l'accesso e la gestione delle risorse di un dispositivo. Questa specifica parte si chiama *kernel*, che come suggerisce il nome (nocciolo dall'inglese), rappresenta la parte centrale di un sistema operativo su cui tutto il resto si appoggia.

1.2 ARCHITETTURA *software* DI UN SISTEMA OPERATIVO

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. Quest'ultimi consistono in 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. Tutto ciò comporta dei vantaggi:

- efficienza: motivo principale per cui la maggior parte dei sistemi operativi ancora oggi 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: non avendo una vera e propria strutturazione, essendo il codice tutto in un unico *file* binario, risulta chiaramente più facile 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 ad 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; tutto il resto deve essere gestito da *server* (*daemon*), che operano al di sopra del*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 [9].

Anche questo modello ha dei vantaggi:

- flessibilità: l'inserimento di un nuovo servizio avviene al di sopra del *kernel*, quindi in qualsiasi momento è possibile aggiungere o togliere servizi senza dover modificare il *kernel* stesso;
- sicurezza: minore quantità di codice eseguita in *kernel mode* (quindi minore quantità di *bug* e minore superficie attaccabile) pertanto maggiore sicurezza del sistema; inoltre i servizi 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 operare normalmente e si potrà procedere a riavviare quel singolo servizio;
- semplicità: essendo il codice composto da qualche decina di migliaia di righe di codice (KSLOC) risulta molto più facile da scrivere.

Dall'altro lato ha un grande svantaggio:

- efficienza: dato che ogni servizio è eseguito 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.

In Figura 1 si possono vedere in maniera schematica le differenze tra *kernel* monolitici e *microkernel*.

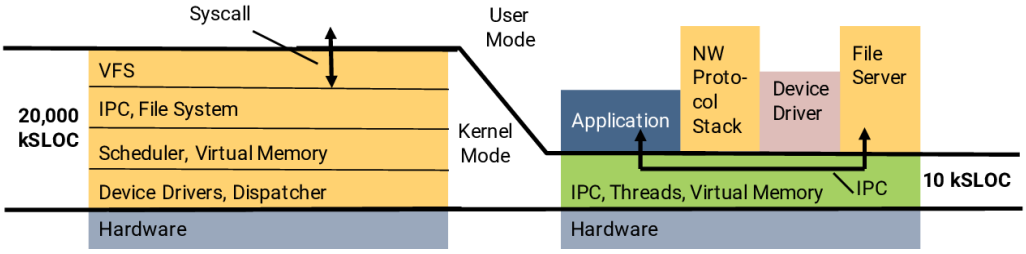


Figura 1: *Kernel* monolitici (sinistra) VS *Microkernel* (destra)

1.3 SCHEDULING DELLA CPU

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. Chiaramente ci sono vari metodi per fare ciò, che prendono il nome di politiche di *scheduling*, i quali si differenziano per modalità e prestazioni. Gli algoritmi che traducono questi metodi si chiamano algoritmi di *scheduling*.

Una particolare politica di *scheduling* rilevante per questo testo è *Round Robin* o *scheduling circolare*: consiste nel determinare un *time slice* (quanto di tempo) nel quale i processi ottengono la CPU. Una volta esaurito questo tempo, il processo viene interrotto e inserito in fondo alla cosiddetta coda dei pronti. In questo modo tutti i processi ottengono la CPU per un tempo massimo stabilito; inoltre è possibile stimare il tempo di attesa prima dell'esecuzione di ciascun processo in base al numero di processi che lo precedono.

1.4 MEMORIA VIRTUALE

Un altro concetto fondamentale quando si parla di sistemi operativi è la gestione della memoria. Il SO deve garantire che ogni programma abbia a disposizione la giusta quantità di memoria necessaria per l'esecuzione, ed inoltre ognuno di essi deve accedere solo alla memoria a lui riservata. Un meccanismo adottato che accomuna quanto appena detto è quello di memoria virtuale.

La memoria virtuale è un meccanismo che permette di simulare uno spazio di memoria centrale (memoria primaria) maggiore di quello fisicamente presente o disponibile, dando l'illusione all'utente di un enorme quantitativo di memoria. Questa tecnica porta con sé diversi vantaggi: uno tra questi la sicurezza dovuta all'isolamento della memoria; la possibilità di condivisione di alcune pagine di memoria tra più processi (es: le pagine contenenti le librerie possono essere usate in contemporanea da più processi senza conflitti) e infine l'ultimo, ma allo stesso tempo il principale vantaggio: avere a disposizione molta più memoria centrale di quella che in realtà è disponibile.

Giustamente viene da chiedersi come tutto ciò sia possibile e il meccanismo alla base è quello di utilizzare una memoria ausiliaria, solitamente la memoria di massa, per allocare una certa parte di memoria che non è stata utilizzata recentemente. Nel momento in cui viene richiesta nuovamente la porzione di dati salvati nella memoria ausiliaria (oppure si

libera spazio nella memoria centrale), i dati relativi vengono prelevati e copiati nuovamente in memoria centrale: questo processo prende il nome di *swapping*.

In presenza di memoria virtuale dunque non parleremo semplicemente di indirizzi di memoria, ma avremo una differenziazione tra indirizzi logici e indirizzi fisici. I programmi lavoreranno solo con indirizzi logici (quindi viene anche facilitata la programmazione) e poi a livello di CPU avverrà un processo di traduzione negli indirizzi fisici.

1.5 *hypervisor*

Un tema che si distacca un po' dai concetti di base dei sistemi operativi, ma che è rilevante per la comprensione del successivo capitolo è quello di *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 viene definito *host machine*, mentre le singole macchina virtuali prendono il nome di *guest machine*. Su ognuna è possibile eseguire un sistema operativo (anche diverso), in modo tale che questi siano isolati tra di loro; inoltre, al contrario di un emulatore, eseguirà la maggior parte delle istruzioni direttamente sulle risorse *hardware* virtualizzate rese disponibili dall'*hypervisor*.

2

SEL4

Entrando adesso più nello specifico, si andrà ad approfondire quello che sarà il caso di studio di questa tesi e cioè *seL4*. Nelle sezioni che seguiranno si avrà una panoramica completa del *microkernel*, trattando singolarmente gli aspetti principali di quest'ultimo. Alla fine di questo capitolo si otterrà un quadro completo di com'è strutturato e delle sue capacità.

2.1 MODELLO STRUTTURALE DI SEL4

SeL4 [5] 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*. SeL4 in particolare è stato sviluppato dal gruppo NICTA, oggi conosciuto con il nome di *Trustworthy System*. Come descritto nel capitolo precedente, essendo seL4 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 nessuno dei servizi che siamo solitamente abituati a trovare in un comune SO, "è solo un sottile involucro attorno all'hardware" [5]. 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). 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. Un esempio pratico è mostrato in Figura 2, 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* e il *file system* per la gestione della memoria secondaria (*hard disk*, supporti rimovibili, ecc.). Le comunicazioni tra le parti saranno gestite da un canale

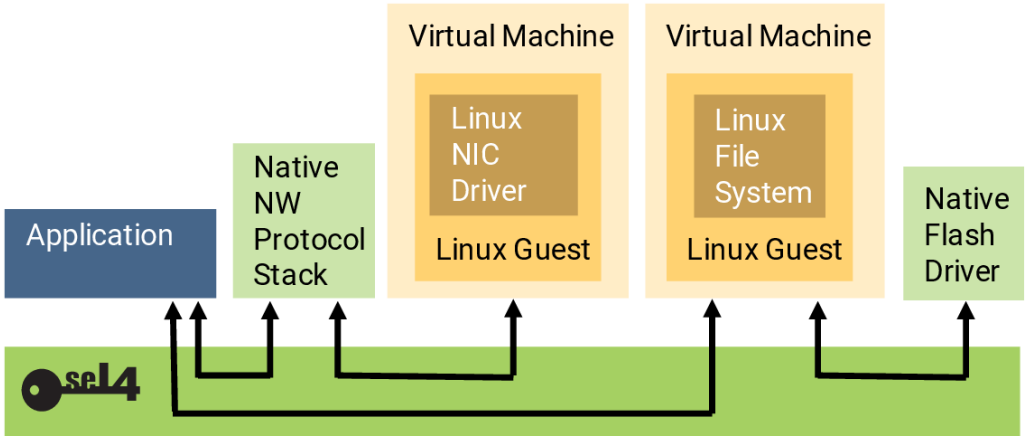


Figura 2: Virtualizzazione del SO Linux per l'integrazione dei servizi di *networking* e *file system*

fornito dal *microkernel*, ma le due macchine virtuali non avranno modo di comunicare tra di loro. Come si vede in figura, anche le comunicazioni tra le varie parti e l'applicazione sono ben delineate e precise: nessun'altra comunicazione al di fuori di quelle indicate dalle frecce è possibile.

2.2 *capability*

Un concetto fondamentale in seL4 è quello di *capability*, che è definita formalmente come un riferimento ad oggetto. Possiamo considerarla anche come un puntatore immutabile, cioè una *capability* che 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, si può pensare alle *capability* come a delle chiavi di accesso estremamente specifiche riguardo su quale entità può accedere ad una particolare risorsa del sistema. Per di più, 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, il che chiaramente 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 ed esecuzione) possono essere eseguite su di esso dai vari utenti (proprietario, gruppo e altri). Tutto ciò implica che ogni *file* con lo stesso *set* di permessi è accessibile ad uno specifico utente. Se ci si colloca nello scenario di voler avviare un programma, di cui non siamo sicuri dell'attendibilità, il quale abbia accesso ad uno e un solo *file* specifico, questo non è possibile perché come può accedere a quel *file*, può accedere anche a tutti gli altri che hanno associati gli stessi permessi.

Con le *capability* questa eventualità non si può presentare perché 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*.

2.2.1 *Proprietà delle capability*

Le *capability* hanno la proprietà di interporsi tra chi crea una *capability* e l'effettivo accesso ad una risorsa: questa proprietà prende il nome di *interposition*. Se un utente dà una *capability* ad un oggetto, esso non è in grado di sapere cosa effettivamente sia quell'oggetto, ma può chiaramente utilizzarlo anche senza sapere che tipo di oggetto sia.

Le *capability* supportano la delega 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*. Questa seconda proprietà si chiama *delegation*.

2.3 *hard real-time systems*

SeL4 è stato sviluppato in modo che possa essere utilizzato anche in casi in cui si sia in presenza di quelli che vengono chiamati *hard real-time system*. 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 comune e semplificato può essere l'autopilota 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 vei-

colo 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 causino il 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.

SeL4 ha alcune caratteristiche che lo rendono adatto in ambiti *hard real-time system*. Infatti, lo *scheduling* dei processi in seL4 è basato sulla priorità. Il *kernel* di sua iniziativa non cambierà mai la priorità di un processo, che è sempre decisa dall'utente.

Inoltre, quando seL4 esegue delle operazioni in modalità *kernel* queste sono esenti da *interrupt*. All'apparenza questo può sembrare grave se non fosse per il fatto che le chiamate di sistema sono tutte brevi. Solo la revoca 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. In aggiunta, ognuna di esse può essere annullata e poi ripresa da quel punto in poi, così da poter gestire degli eventuali *interrupt* in attesa.

Questi due punti appena elencati sono caratteristiche fondamentali per gli *hard real-time system*: *scheduling* dei processi basato sulla priorità, che sia quindi facilmente analizzabile; 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 essere 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*.

2.3.1 *Mixed-Criticality Systems*

Un *mixed-criticality system* (MCS) è un sistema costituito 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, e che questi siano quindi isolati e protetti dai componenti meno critici.

Un approccio classico per questo tipo di sistemi è isolare le criticità sia per quanto riguarda il tempo sia lo spazio. Ciò è noto come *strict time and space partitioning* (TSP). Ma questo implica il dover assegnare staticamente

l'area di memoria, il tempo di esecuzione e quindi lo *scheduling*; per farlo si utilizzano dati misurati precedentemente nel caso pessimo. Essendo sistemi *real-time*, ogni operazione deve avere dei limiti di tempo, quindi un'operazione su cui è stato misurato un tempo di esecuzione di 5 millisecondi (sempre nel caso pessimo) deve avere questa durata, non 4ms né tantomeno 6ms. Chiaramente, determinando staticamente i tempi e gli spazi nel caso peggiore, si è sicuri che questi vengano rispettati. C'è da considerare però che non sempre si presentano dei casi pessimi e si ha quindi uno scarso utilizzo delle risorse; per di più la latenza di un *interrupt* nel caso pessimo, può essere molto costosa.

SeL4 supporta i *mixed-criticality system*. Per quanto riguarda l'isolamento, si è già visto che le *capability*, in termini di spazio, intrinsecamente lo garantiscono. Resta perciò da esaminare il comportamento da un punto di vista temporale.

Il *microkernel* 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 il *time slice* (quanto di tempo) determina per quanto tempo il *kernel* lascerà in esecuzione un *thread*, prima di fermarlo per selezionare un altro processo. Quest'ultimo verrà scelto tra i processi pronti in base alla priorità, con una politica *round-robin* tra i pari 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. Tale politica si chiama *scheduling-context capability*. Quest'ultimo contiene due attributi principali:

1. *time budget*, che sostituisce il *time slice*;
2. *time period*, che determina invece quante volte un *budget* può essere usato per periodo, evitando in questa maniera che un processo monopolizzi la CPU, indipendentemente dalla sua priorità.

2.4 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 stato descritto il funzionamento generale di seL4, con particolare attenzione sulla sicurezza di questo sistema. Chi è dell'ambito, sa già che spesso sicurezza e buone prestazioni 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 ad influire sulle *performance* di un sistema. È di conseguenza lecito domandarsi se questo *microkernel* sia performante o meno.

Nonostante non fosse nelle prerogative dello sviluppo di seL4, questo alla fine, si è rivelato il più performante dei *microkernel* della famiglia L4. Inoltre sono state scritte altre pubblicazioni indipendenti, che mettono a confronto seL4 con altri *microkernel*, per studiarne le *performance*, in particolare Fiasco.OC, Zicron e CertiKOS. Confrontando i costi dell'IPC si può vedere che seL4 ha un bel vantaggio, anche di oltre un fattore 2, rispetto agli altri *microkernel*. Gli articoli che mostrano gli studi delle *performance* sono riportati in [22] per quanto riguarda il confronto tra Fiasco.OC, Zicron e seL4, mentre in [4], si trova il confronto fra CertiKOS e seL4.

3

SPERIMENTAZIONE DI SEL4

In quest'ultimo capitolo viene affrontato il *focus* della tesi, cioè la sperimentazione di *seL4*. Seguiranno tutti gli *step* che sono stati messi in pratica durante questa fase dello studio del sistema: dall'installazione di un sistema operativo in cui poter simulare *seL4*, fino allo sperimentare con mano le varie funzionalità. Ovviamente questo ha richiesto un approfondimento più tecnico e specifico, rispetto a quanto realizzato finora, di alcuni aspetti come la gestione della memoria fisica e virtuale, l'IPC, ecc.

3.1 PREREQUISITI

Come prima cosa ho installato *VirtualBox* (macchina virtuale) sul mio portatile in quanto, come consigliato dalle linee guida fornite da *Trustworthy System* (TS), è ottimale lavorare in ambiente Linux. Inizialmente ho pensato di utilizzare una macchina virtuale con Linux, così da lasciare inalterato il mio computer e comunque avere a disposizione un sistema operativo Linux su cui operare. Andando avanti con il *set-up* del sistema per iniziare a lavorare su seL4, ho però ho incontrato una prima difficoltà: lo spazio nel portatile non era purtroppo adeguato e la macchina virtuale, considerando il sistema operativo e l'installazione dei vari prerequisiti per poter far girare il *microkernel*, cominciava ad occupare una quantità di memoria non trascurabile di GB. Ho di conseguenza 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 sull'SSD esterno collegato via USB.

Per operare su seL4, è necessario avere installati sul sistema programmi che simulino un'architettura su cui farlo eseguire. Per fare ciò è fondamentale installare delle dipendenze (prerequisiti) cioè compilatori, emulatori *software* vari e librerie.

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*) è un emulatore *open-source*, che permette di simulare un'architettura informatica e quindi diversi sistemi operativi, in questo caso, requisito fondamentale perché 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 essenziale 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 22.04.2 LTS, 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* necessaria per la verifica di seL4 contenente, in particolare il sorgente del *kernel*, i *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 e che quindi si procederà alla configurazione di Isabelle posizionandoci 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* darà come risultato:

- la creazione di una cartella per le impostazioni utente di Isabelle;
- l'installazione delle impostazione Isabelle per L4.verified [3], il quale è un *repository* che contiene formalismi per la verifica di seL4;

- il *download* di Scala, Java JDK, PolyML ed altri dimostratori (*prover*) esterni;
- la 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 l'occorrente per poi 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 ci saranno tutte le direttive e le librerie necessarie per eseguire i vari *test* e scaricare anche il *kernel* stesso, attraverso il comando `repo`.

Successivamente è stato necessario creare una cartella `build-x86` di configurazione 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 in seguito, 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*, ecc.) che appariranno nel terminale, concluderà, se tutto è andato a buon fine, con:
`All is well in the universe`
Il che indica che seL4 può essere utilizzato in questo ambiente simulato, come mostrato in Figura 3.

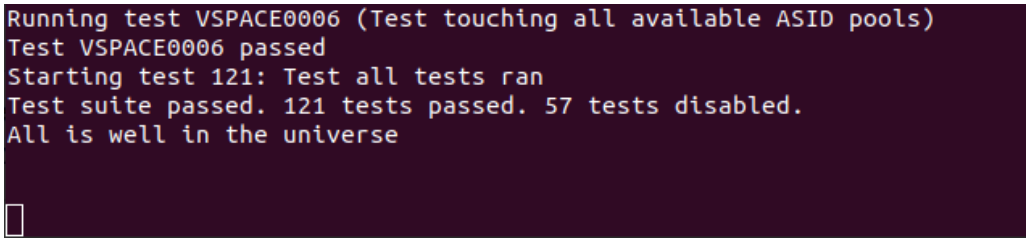


Figura 3: Primo avvio di seL4

3.4 PROGRAMMAZIONE CON LE API LIVELLO *kernel* DI SEL4

Una volta procurati tutti i prerequisiti necessari e appurato che seL4 può essere eseguito senza problemi, è finalmente possibile iniziare a prendere familiarità con il sistema, seguendo *tutorial* forniti dalla *seL4 Foundation* [1]. Tali *tutorial* contengono programmi semicompleti, creati appositamente per sperimentare e far comprendere le funzionalità del sistema, in particolare con le API di seL4 [2].
Come ormai già visto più volte sopra, si otterrà l'ambiente idoneo per eseguire i *tutorial*, attraverso l'uso di `repo`:

```
mkdir sel4-tutorials-manifest
cd sel4-tutorials-manifest
repo init -u https://github.com/seL4/sel4-tutorials-manifest
repo sync
```

Ogni *tutorial* ha un suo *repository* da importare nell'ambiente di lavoro nel quale, tra gli altri *file* e cartelle, c'è (solitamente) un `main.c`, che sarà quello su cui andare ad apportare le modifiche per completare il *tutorial* stesso.

3.4.1 *Capability*

Come già detto nel capitolo precedente, una *capability* è un *token* unico, che dà accesso ad un'entità del sistema, un puntatore con dei diritti di accesso; in seL4 ci sono 3 tipi di *capability*:

1. *capability* che controllano l'accesso ad entità del *kernel*, come i *thread control block* (TCB);
2. *capability* che controllano l'accesso a risorse astratte tipo gli *interrupt*;
3. *untyped capability* che sono responsabili della gestione della memoria.

Tutte le *capability* delle risorse del *kernel* sono date dal processo *root* all'inizializzazione del sistema, un po' come il processo `init` nei sistemi unix, che è padre di tutti i processi. Quando parliamo di *capability* ci sono 3 termini fondamentali: CNode, CSlot e CSpace. Il primo di questi è l'abbreviazione di *Capability-Node*, un oggetto che contiene delle *capability*: si può pensarlo come un vettore (*array*) di *capability*. Ogni elemento dell'*array* è chiamato CSlot (*Capability-Slot*), il quale può avere due stati: `empty` o `full`. Ciò significa, rispettivamente, che il CNode ha una *capability* nulla oppure una *capability* ad una risorsa del *kernel*. Per convenzione il primo CSlot, cioè quello situato alla posizione 0 del vettore, è nullo. Invece un CSpace (*Capability-Space*) è il *range* completo di *capability* accessibile da un *thread*, che può essere composto da uno o più CNode.

Per fare riferimento ad una *capability* ed eseguire operazioni su di essa, è necessario fare un `address` (indirizzamento) della *capability*. Ci sono due modi in seL4: tramite *invocazione* o con *indirizzamento diretto*.

Per quanto riguarda l'invocazione, ogni *thread* ha uno speciale CNode installato nel suo TCB noto come `CSpace root`. Questo può essere nullo, ad esempio quando il *thread* non è autorizzato a invocare nessuna *capability*, o può avere una *capability* ad un noto CNode. Quando si vuole fare un *addressing* di una *capability* attraverso invocazione, un CSlot viene indirizzato implicitamente, invocando il CSpace *root* del *thread* che sta facendo l'invocazione.

Per quanto riguarda il metodo dell'indirizzamento diretto, questo invece permette di specificare il CNode, piuttosto che utilizzare implicitamente il CSpace *root*. Questo tipo di *addressing* è usato principalmente per costruire e manipolare i CSpace, potenzialmente il CSpace di un altro *thread*.

L'esercizio proposto in questa sezione è un programma in linguaggio C con una serie di errori da risolvere, il primo tra questi è nel settaggio del numero di *byte* del CNode:

```
int main(int argc, char *argv[]) {

    /* parse the location of the seL4_BootInfo data structure from
    the environment variables set up by the default crt0.S */
    seL4_BootInfo *info = platsupport_get_bootinfo();

    size_t initial_cnode_object_size = BIT(info->initThreadCNodeSizeBits);
    printf("Initial CNode is %zu slots in size\n",
    initial_cnode_object_size);
    size_t initial_cnode_object_size_bytes = 0; // TODO
    printf("The CNode is %zu bytes in size\n",
```

```
    initial_cnode_object_size_bytes);
```

Chiaramente `initial_cnode_object_size_bytes` non può essere 0, il suo valore altre sì dato sarà dato dal numero degli *slot* del CNode moltiplicato per le dimensioni in *bit* di ognuno di essi: `initial_cnode_object_size * (1u << seL4_SlotBits)`.

Eseguendo nuovamente il codice, questo darà l'errore `Attempted to invoke a null cap`. Ciò accade perché il codice cerca di impostare la priorità del TCB del *thread*, invocando l'ultimo CSlot del CSpace, che però è vuoto:

```
seL4_CPtr first_free_slot = info->empty.start;
seL4_Error error = seL4_CNode_Copy(seL4_CapInitThreadCNode,
           first_free_slot, seL4_WordBits, seL4_CapInitThreadCNode,
           seL4_CapInitThreadTCB, seL4_WordBits, seL4_AllRights);
ZF_LOGF_IF(error, "Failed to copy cap!");
%seL4_CPtr last_slot = info->empty.end - 1;
// TODO

/* set the priority of the root task */
error = seL4_TCB_SetPriority(last_slot, last_slot, 10);
ZF_LOGF_IF(error, "Failed to set priority");
```

Per risolvere dunque il problema è necessario fare un'altra copia della *capability* del TCB all'interno dell'ultimo slot del CNode: viene utilizzato `seL4_CNode_Copy`, che prende come parametri `destination root, slot, depth, source root, slot, depth` e `rights`, dove `depth` indica quanto bisogna attraversare il CNode per arrivare al CSlot e `rights` sono invece i diritti ereditati dalla nuova *capability*; `first_free_slot` è lo *slot* in cui è stata fatta una copia della *capability* del TCB del *thread* iniziale qualche riga di codice sopra:

```
seL4_CNode_Copy(seL4_CapInitThreadCNode, last_slot, seL4_WordBits,
    seL4_CapInitThreadCNode, first_free_slot, seL4_WordBits, seL4_AllRights
    );
```

Rieseguendo il programma non viene più mostrato l'errore precedente, ma è comunque presente un altro errore `first_free_slot is not empty`. Questo avviene perché il codice cerca di spostare `first_free_slot` e `last_slot` in se stesso; ciò non è possibile (perché è già presente una *capability*, cioè se stessa) ed è in realtà un *escamotage* per controllare se un CSlot è vuoto:

```
// TODO
```

```
// check first_free_slot is empty
error = seL4_CNode_Move(seL4_CapInitThreadCNode, first_free_slot,
                        seL4_WordBits, seL4_CapInitThreadCNode,
                        first_free_slot, seL4_WordBits);
ZF_LOGF_IF(error != seL4_FailedLookup, "first_free_slot is not empty");

// check last_slot is empty
error = seL4_CNode_Move(seL4_CapInitThreadCNode, last_slot, seL4_WordBits,
                        seL4_CapInitThreadCNode, last_slot, seL4_WordBits);
ZF_LOGF_IF(error != seL4_FailedLookup, "last_slot is not empty");
```

Perciò per risolvere il problema bisogna eliminare le due *capability*, questo può essere fatto eliminando le due copie delle *capability* usando `seL4_CNode_Delete`, oppure con `seL4_CNode_Revoke` sulla *capability* originale da cui sono state fatte le copie; quest'ultima API elimina tutte le *capability* figlie di essa. Per fare più velocemente, si utilizzerà il secondo metodo, che richiede come parametri il CNode e la posizione dentro di esso in cui andare a recuperare la *capability* (`CNode, index, depth`):

```
seL4_CNode_Revoke(seL4_CapInitThreadCNode, seL4_CapInitThreadTCB,
    seL4_WordBits);
```

L'esercitazione si conclude con la sospensione del *thread* corrente:

```
seL4_TCB_Suspend(seL4_CapInitThreadTCB);
```

Il codice completo del *tutorial* è riportato in [15].

3.4.2 *Gestione della memoria*

Nella sezione precedente sono stati elencati i tipi di *capability* presenti in seL4, al terzo posto nell'elenco si trovano le *untyped capability*, il modo con il quale è possibile gestire la memoria fisica nel *microkernel* seL4.

Ad eccezione di una piccola parte di memoria del *kernel*, tutta la restante è gestita a livello utente. Le *capability* di tutta la memoria fisica disponibile vengono passate al processo *root* come *capability* alla *untyped memory*, che altro non è che un blocco contiguo di memoria fisica con una dimensione ben specifica. Per riassumere, in seL4 si avranno quindi le *untyped capability* che sono *capability* per la *untyped memory*. Inoltre le *untyped capability* possono essere riscritte in oggetti del *kernel* insieme alla *capability*, oppure in ulteriori *untyped capability* più piccole.

Le *untyped capability* hanno anche un *flag* booleano *device*, che indica se

la memoria è scrivibile dal *kernel* oppure no: può essere in un'area non accessibile dal *kernel* o riservata ad altri dispositivi.

In seL4 esiste un unico modo per invocare una *untyped capability*, cioè attraverso l'utilizzo dell'API `seL4_Untyped_Retype`, che serve per creare una nuova *capability* da una *untyped capability*. Nello specifico, questo *retype* darà accesso ad un sottoinsieme della memoria della *capability* di origine, che può essere una *untyped capability* più piccola o può puntare ad un nuovo oggetto con un tipo specifico:

```
seL4_Untyped_Retype(parent_untyped, // the untyped capability to retype
                    seL4_UntypedObject, // type
                    untyped_size_bits,  //size
                    seL4_CapInitThreadCNode, // root
                    0, // node_index
                    0, // node_depth
                    child_untyped, // node_offset
                    1); // num_caps
```

Le *untyped capability* sono riscritte in maniera incrementale seguendo una politica *greedy*, a partire dall'*untyped* invocato. Ogni *untyped capability* mantiene un singolo *watermark*, con gli indirizzi prima di esso non disponibili e quelli successivi liberi. La memoria non può essere liberata fino a che tutti i figli non vengono revocati, laddove i figli non sono altro che le nuove *capability*, che vengono create da una *untyped capability*.

Come per la sezione precedente anche qui è presente un *repository* da scaricare con all'interno un file `main.c`, che una volta compilato e avviato, stampa a video una lista di tutte le *untyped capability* fornite dal processo *root* all'avvio e segnala un errore `Untyped Retype: Requested UntypedItem size too small`. Ciò succede perché il programma sta tentando di creare una *untyped* di dimensione 0:

```
int main(int argc, char *argv[]) {
    /* parse the location of the seL4_BootInfo data structure from
    the environment variables set up by the default crt0.S */
    seL4_BootInfo *info = platsupport_get_bootinfo();


    printf("    CSlot   \tPaddr           \tSize\tType\n");
    for (seL4_CPtr slot = info->untyped.start; slot != info->untyped.end;
    slot++) {
        seL4_UntypedDesc *desc =
                 &info->untypedList[slot - info->untyped.start];
        printf("%8p\t%16p\t2^%d\t%s\n", (void *) slot,
              (void *) desc->paddr, desc->sizeBits,
              desc->isDevice ? "device untyped" : "untyped");
```

```
    }
    seL4_Error error;

    // list of general seL4 objects
    seL4_Word objects[] = {seL4_TCBObject, seL4_EndpointObject,
                           seL4_NotificationObject};
    // list of general seL4 object size_bits
    seL4_Word sizes[] = {seL4_TCBBits, seL4_EndpointBits,
                         seL4_NotificationBits};

    // TODO
    seL4_Word untyped_size_bits = 0; //ERRORE GENERATO QUI
    seL4_CPtr parent_untyped = 0;
    seL4_CPtr child_untyped = info->empty.start;

    // First, find an untyped big enough to fit all of our objects
    for (int i = 0; i < (info->untyped.end - info->untyped.start); i++) {
        if (info->untypedList[i].sizeBits >=
                    untyped_size_bits && !info->untypedList[i].isDevice) {

            parent_untyped = info->untyped.start + i;
            break;
        }
    }
```

Per risolvere questo problema si deve assegnare una dimensione consona alla variabile `untyped_size_bits`. Dato che si deve creare uno spazio per tutti gli elementi di `objects[]` e considerato che la somma di `seL4_EndpointBits` e `seL4_NotificationBits` è inferiore a `seL4_TCBBits`, si può attribuire alla variabile il valore `seL4_TCBBits + 1`. Il +1 fa raddoppiare il numero di *byte*, visto che lo spazio assegnato sarà $2^{\text{seL4\_TCBBits}+1}$ *bit*, i quali sono sufficienti per contenere tutti e tre gli elementi.

Eseguendo di nuovo il programma, questo procederà fino a che non segnalerà un ulteriore errore `Failed to set priority`:

```
// create an untyped big enough to retype all of the above objects from
error = seL4_Untyped_Retype(parent_untyped, seL4_UntypedObject,
    untyped_size_bits, seL4_CapInitThreadCNode, 0, 0, child_untyped, 1);
ZF_LOGF_IF(error != seL4_NoError, "Failed to retype");

// use the slot after child_untyped for the new TCB cap:
seL4_CPtr child_tcb = child_untyped + 1;
// TODO

// try to set the TCB priority
```

```
error = seL4_TCB_SetPriority(child_tcb, seL4_CapInitThreadTCB, 10);
ZF_LOGF_IF(error != seL4_NoError, "Failed to set priority");
```

L'errore viene generato perché `child_tcb` è un CSlot vuoto. Per risolvere è sufficiente assegnare al CSlot una *capability*, creando un *TCB object* da `child_untyped`:

```
seL4_Untyped_Retype(child_untyped, seL4_TCBObject, 0,
    seL4_CapInitThreadCNode, 0, 0, child_tcb, 1);
```

Con questa linea di codice il problema è sì risolto, ma l'esecuzione viene bloccata da un altro errore `Endpoint cap is null cap`:

```
// use the slot after child_tcb for the new endpoint cap:
seL4_CPtr child_ep = child_tcb + 1;
// TODO

// identify the type of child_ep
uint32_t cap_id = seL4_DebugCapIdentify(child_ep);
ZF_LOGF_IF(cap_id == 0, "Endpoint cap is null cap");
```

Tale errore è molto simile al precedente: si sta cercando di identificare un *endpoint* nullo. Quindi per risolvere il problema è va creato un *endpoint object* sempre da `child_untyped` e mettere la *capability* nel CSlot `child_ep`:

```
seL4_Untyped_Retype(child_untyped, seL4_EndpointObject, 0,
    seL4_CapInitThreadCNode, 0, 0, child_ep, 1);
```

Alla fine il programma tenta di allocare tutto il `child_untyped` come *endpoint*, ma fallisce perché tutto lo spazio è stato esaurito dalle allocazioni fatte precedentemente. La soluzione al problema è eseguire una `seL4_CNode_Revoke` (vista sopra) su di esso, in modo che tutto le spazio venga liberato e così facendo il programma termina con successo:

```
// revoke the child untyped
error = seL4_CNode_Revoke(seL4_CapInitThreadCNode, child_untyped,
    seL4_WordBits);

// allocate the whole child_untyped as endpoints
// Remember the sizes are exponents, so this computes 2^untyped_size_bits /
     2^seL4_EndpointBits:
seL4_Word num_eps = BIT(untyped_size_bits - seL4_EndpointBits);
error = seL4_Untyped_Retype(child_untyped, seL4_EndpointObject, 0,
    seL4_CapInitThreadCNode, 0, 0, child_tcb, num_eps);
ZF_LOGF_IF(error != seL4_NoError, "Failed to create endpoints.");
```

```
printf("Success\n");
```

Il codice completo del *tutorial* è riportato in [20].

3.4.3 *Virtual memory management*

SeL4 non fornisce strumenti per la gestione della memoria virtuale al di là delle primitive per la gestione dell'*hardware*; quindi e il servizio di *mapping* della memoria e lo *swapping* devono essere gestiti a livello utente, che ha tutta la libertà di procedere in base alle esigenze del sistema. SeL4 mette dunque a disposizione degli oggetti appositi chiamati *VSpace* (*virtual address space*), simili ai CSpace, che sono composti da oggetti forniti dal *kernel*, che variano in base all'architettura *hardware* (x86_64, RISC-V, ARM).

Per mappare le pagine sono necessari degli *intermediate hardware virtual memory objects*: in pratica si deve creare una struttura intermedia, che varia in base all'architettura. Ad esempio nei sistemi x86_64 per mappare una pagina sono necessari questi 3 oggetti: `seL4_PDPT, seL4_PageDirector,` e `seL4_PageTable`.

Le API di seL4 forniscono varie funzioni per la mappatura della memoria in base all'architettura in cui sta girando seL4. Tutte le funzioni di *mapping* prendono 3 argomenti principali:

- il VSpace in cui mappare l'oggetto;
- l'indirizzo virtuale su cui mappare l'oggetto;
- gli attributi della memoria virtuale che dipendono dall'architettura.

Un esempio di mappatura di un oggetto `seL4_PDPT` ad un certo indirizzo `TEST_VADDR` è:

```
seL4_X86_PDPT_Map(pdpt, seL4_CapInitThreadVSpace, TEST_VADDR,
    seL4_X86_Default_VMAttributes);
```

Una volta che le strutture di paginazione intermedie sono state mappate in un certo *range* di indirizzi virtuali, i *frame* fisici possono essere mappati in quel *range* attraverso l'invocazione del *frame capability*.
Ecco un esempio di mappatura di un *frame*:

```
seL4_X86_Page_Map(frame, seL4_CapInitThreadVSpace, TEST_VADDR, seL4_CanRead
    , seL4_X86_Default_VMAttributes);
```

Come si può notare, questo metodo prende un argomento in più, perché per mappare i *frame* vengono richiesti anche i diritti che determineranno il tipo di mappatura (nell'esempio sopra diritti di sola lettura).

Il *tutorial* di questa sezione fornisce un programma che all'avvio presenta l'errore `Missing intermediate paging structure at level 30`:

```
int main(int argc, char *argv[]) {
    /* parse the location of the seL4_BootInfo data structure from
    the environment variables set up by the default crt0.S */
    seL4_BootInfo *info = platsupport_get_bootinfo();
    seL4_Error error;
    seL4_CPtr frame = alloc_object(info, seL4_X86_4K, 0);
    seL4_CPtr pdpt = alloc_object(info, seL4_X86_PDPTObject, 0);
    seL4_CPtr pd = alloc_object(info, seL4_X86_PageDirectoryObject, 0);
    seL4_CPtr pt = alloc_object(info, seL4_X86_PageTableObject, 0);

  // TODO

  // TODO

    /* map a PDPT at TEST_VADDR */
    error = seL4_X86_PDPT_Map(pdpt, seL4_CapInitThreadVSpace, TEST_VADDR,
    seL4_X86_Default_VMAttributes);

    /* map a read-only page at TEST_VADDR */
    error = seL4_X86_Page_Map(frame, seL4_CapInitThreadVSpace, TEST_VADDR,
    seL4_CanRead, seL4_X86_Default_VMAttributes);
    if (error == seL4_FailedLookup) {
        printf("Missing intermediate paging structure at level %lu\n",
    seL4_MappingFailedLookupLevel());
    }
    ZF_LOGF_IF(error != seL4_NoError, "Failed to map page");
```

L'errore è dovuto al fatto che per mappare una pagina, tutte le strutture di paginazione intermedie devono essere mappate; il valore `30` equivale alla costante `SEL4_MAPPING_LOOKUP_NO_PD`, il che indica che è necessario mappare un oggetto *page directory*, che può essere fatto con l'apposito metodo `seL4_X86_PageDirectory_Map`:

```
seL4_X86_PageDirectory_Map(pd, seL4_CapInitThreadVSpace, TEST_VADDR,
    seL4_X86_Default_VMAttributes);
```

Ricompilando ed eseguendo il codice appare un errore simile al precedente `Missing intermediate paging structure at level 21`, dove il

valore 21 questa volta, indica la costante SEL4_MAPPING_LOOKUP_NO_PT che suggerisce di mappare un oggetto di tipo *page table*:

```
seL4_X86_PageTable_Map(pt, seL4_CapInitThreadVSpace, TEST_VADDR,
    seL4_X86_Default_VMAttributes);
```

Adesso il codice procede mappando la pagina; successivamente però (come si può leggere nel codice sotto riportato) avviene un tentativo di scrittura sulla pagina, che genera un errore perché la pagina era stata mappata in sola lettura seL4_CanRead. L'errore può dunque essere evitato facendo una rimappatura della pagina, questa volta in lettura e scrittura:

```
seL4_X86_Page_Map(frame, seL4_CapInitThreadVSpace, TEST_VADDR,
    seL4_ReadWrite, seL4_X86_Default_VMAttributes);
```

Il *mapping* delle pagine può anche essere disfatto utilizzando unmap sulla pagina o su qualsiasi struttura intermedia di paginazione; in alternativa può essere eseguito eliminando la *capability* finale di qualsiasi struttura di paginazione.
Il codice completo del *tutorial* è riportato in [21].

3.4.4 *Thread*

SeL4, per rappresentare l'esecuzione di un processo e gestirne i tempi di esecuzione, fornisce i *thread*. Essi sono realizzati attraverso *thread control block object* (TCBs), uno per ogni *thread* del kernel.

Come noto, in un SO è lo *scheduler* a decidere quale processo e per quanto tempo può utilizzare la CPU. In seL4, come già visto nel capitolo precedente, la politica di *scheduling* è un'integrazione di *round-robin* e *scheduling a priorità*: lo *scheduler* sceglie i *thread* con maggiore priorità che sono pronti e se ce ne sono con la stessa priorità, questi saranno scelti in ordine FIFO, seconda la politica *round-robin*. La priorità è determinata da un *range* che va da 0 (seL4_MinPrio) a 255 (seL4_MaxPrio). Oltre alla priorità, un TCBs contiene anche un *maximum control priority* (MCP), che serve per controllare che un processo non modifichi la priorità di un altro processo (o di se stesso), impostandola più alta della sua. Quindi un processo che vuole modificare una priorità deve fornire la sua *capability* (di *thread*), in modo da determinare se è autorizzato a impostare quella priorità.

L'esercizio per questa sezione, se fatto partire senza nessuna modifica, inizialmente mostrerà a video una tabella di tutti i TCB (questo è ottenuto

tramite una chiamata di sistema di debug `seL4_DebugDumpScheduler()`) e successivamente lancia un errore `Failed to retype thread: 2` come in Figura 4:

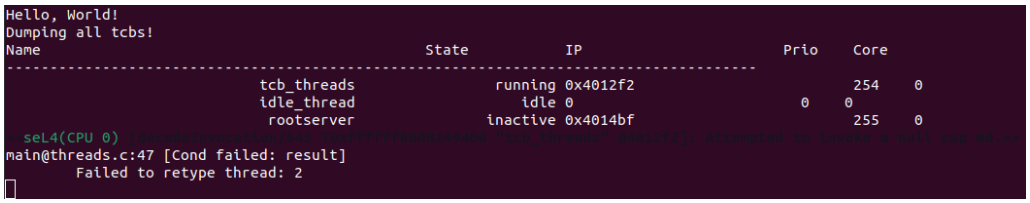


Figura 4: lista TCB

Questo errore avviene perché c'è un'errata invocazione del metodo: `seL4_Untyped_Retype()`.

```
// the root CNode of the current thread
extern seL4_CPtr root_cnode;
// VSpace of the current thread
extern seL4_CPtr root_vspace;
// TCB of the current thread
extern seL4_CPtr root_tcb;
// Untyped object large enough to create a new TCB object

extern seL4_CPtr tcb_untyped;
extern seL4_CPtr buf2_frame_cap;
extern const char buf2_frame[4096];

// Empty slot for the new TCB object
extern seL4_CPtr tcb_cap_slot;
// Symbol for the IPC buffer mapping in the VSpace, and capability to the
    mapping
extern seL4_CPtr tcb_ipc_frame;
extern const char thread_ipc_buff_sym[4096];
// Symbol for the top of a 16 * 4KiB stack mapping, and capability to the
    mapping
extern const char tcb_stack_base[65536];
static const uintptr_t tcb_stack_top = (const uintptr_t)&tcb_stack_base +
    sizeof(tcb_stack_base);

int new_thread(void *arg1, void *arg2, void *arg3) {
    printf("Hello2: arg1 %p, arg2 %p, arg3 %p\n", arg1, arg2, arg3);
    void (*func)(int) = arg1;
    func(*(int *)arg2);
    while(1);
}
```

```
int main(int c, char* arbv[]) {

    printf("Hello, World!\n");

    seL4_DebugDumpScheduler();
  // TODO
    seL4_Error result = seL4_Untyped_Retype(seL4_CapNull, seL4_TCBObject,
    seL4_TCBBits, seL4_CapNull, 0, 0, seL4_CapNull, 1);
    ZF_LOGF_IF(result, "Failed to retype thread: %d", result);
    seL4_DebugDumpScheduler();
```

Come si può evincere, al metodo viene passato un oggetto `seL4_CapNull` come oggetto da riscrivere, che ovviamente genera l'errore. Dunque un modo corretto per ovviare a questo errore è utilizzare gli oggetti creati nelle variabili globali del codice:

```
seL4_Error result = seL4_Untyped_Retype(tcb_untyped, seL4_TCBObject,
    seL4_TCBBits, root_cnode, 0, 0, tcb_cap_slot, 1);
```

Rieseguendo il codice, si noterà che l'errore è risolto e tra la lista dei TCB adesso è presente anche quello appena creato. Dopo aver risolto questo problema, si presenta un errore `Failed to configure thread: 2`, in quanto la configurazione del TCB viene fatta tutta su valori nulli:

```
result = seL4_TCB_Configure(seL4_CapNull, seL4_CapNull, 0, seL4_CapNull, 0,
     0, (seL4_Word) NULL, seL4_CapNull);
ZF_LOGF_IF(result, "Failed to configure thread: %d", result);
```

Il metodo `seL4_TCB_Configure` prende come parametri:

```
seL4_TCB_Configure(tcb, // tcb su cui operare
          cspace_root, // nuovo CSpace root
          cspace_root_data, // opzionale: setta il nuovo CNode
          vspace_root, // nuovo VSpace root
          vspace_root_data, // non ha effetto su x86 e ARM
          buffer, // locazione dell'IPC buffer
          bufferFrame); // IPC buffer
```

Si può quindi procedere alla corretta configurazione del TCB in modo tale da avere lo stesso CSpace e VSpace del *thread* corrente:

```
result = seL4_TCB_Configure(tcb_cap_slot, seL4_CapNull, root_cnode, 0,
    root_vspace, 0, (seL4_Word) thread_ipc_buff_sym, tcb_ipc_frame);
```

Adesso l'errore che si presenta sarà un altro `Failed to set the priority for the new TCB object`: questo accade perché la priorità

data al *thread* ha valore 0:

```
result = seL4_TCB_SetPriority(tcb_cap_slot, seL4_CapNull, 0);
ZF_LOGF_IF(result, "Failed to set the priority for the new TCB object.\n");
seL4_DebugDumpScheduler();
```

Il *thread* corrente ha un MCP di 254 quindi è possibile assegnare questo valore come priorità, ma è necessario anche cambiare il valore `seL4_CapNull` e sostituirlo con il TCB del *thread* corrente `root_tcb`, dopodiché impostare in maniera adeguata i registri iniziali, in particolare il *program counter* e lo *stack pointer*. Ciò è possibile grazie alle *utility* contenute in `libsel4utils`:

```
seL4_UserContext regs = {0};
int error = seL4_TCB_ReadRegisters(tcb_cap_slot, 0, 0, sizeof(regs)/sizeof(
    seL4_Word), &regs);
ZF_LOGF_IFERR(error, "Failed to read the new thread's register set.\n");

// TODO
sel4utils_set_instruction_pointer(&regs, (seL4_Word)new_thread);
// TODO
sel4utils_set_stack_pointer(&regs, tcb_stack_top);
// TODO
error = seL4_TCB_WriteRegisters(tcb_cap_slot, 0, 0, sizeof(regs)/sizeof(
    seL4_Word), &regs);
ZF_LOGF_IFERR(error, "Failed to write the new thread's register set.\n"
                  "\tDid you write the correct number of registers? See
    arg4.\n");
seL4_DebugDumpScheduler();
```

A questo punto si farà partire partire il *thread*, ma con un piccolo aggiustamento nel codice:

```
//resume the new thread
error = seL4_TCB_Resume(seL4_CapNull);
ZF_LOGF_IFERR(error, "Failed to start new thread.\n");
while(1);
return 0;
}
```

Chiaramente il `seL4_TCB_Resume` va fatto sul nostro `tcb_cap_slot` e non su `seL4_CapNull`. Ora il nuovo *thread* viene eseguito e mostra a video `Hello2: arg1 0, arg2 0, arg3 0`.
I valori passati al nuovo *thread* sono tutti 0. Se volessimo però passare al *thread* valori differenti, si potrebbe utilizzare la funzione `sel4utils_arch_init_local_context`, con le dovute modifiche al codice:

```
UNUSED seL4_UserContext regs = {0};
int error = seL4_TCB_ReadRegisters(tcb_cap_slot, 0, 0, sizeof(regs)/sizeof(
    seL4_Word), &regs);
ZF_LOGF_IFERR(error, "Failed to read the new thread's register set.\n"
              "\tDid you write the correct number of registers? See arg4.\n
    ");

sel4utils_arch_init_local_context((void*)new_thread,
                                  (void *)1, (void *)2, (void *)3,
                                  (void *)tcb_stack_top, &regs);
error = seL4_TCB_WriteRegisters(tcb_cap_slot, 0, 0, sizeof(regs)/sizeof(
    seL4_Word), &regs);
ZF_LOGF_IFERR(error, "Failed to write the new thread's register set.\n"
              "\tDid you write the correct number of registers? See arg4.\n
    ");
```

Il codice completo del *tutorial* è riportato in [19].

3.4.5 *IPC*

InterProcess Communication è il meccanismo che utilizza il *microkernel* per sincronizzare lo scambio di piccole quantità di dati e *capability* tra i processi. In seL4 l'IPC è facilitato dal fatto che gli oggetti del *kernel* sono di piccole dimensioni, noti come *endpoint* e fungono da porte per la comunicazione; quindi per mandare e ricevere messaggi IPC bisogna procedere attraverso invocazioni sugli *endpoint*.

I *thread* possono mandare messaggi sugli *endpoint* con la *system call* `seL4_Send` che è bloccante, mentre possono usare `seL4_Recv` per ricevere messaggi.

`seL4_Call` invece è una chiamata di sistema che combina le due precedenti con una differenza: nella fase di ricezione il *thread* che usa questa funzione è bloccato su una *one-time capability* chiamata *reply capability* e non sull'*endpoint* stesso, come avverrebbe normalmente con la `seL4_Recv`. La *replay capability* è contenuta nel TCB del ricevente.

La *system call* `seL4_Reply` invoca la *reply capability*, la quale manderà un IPC che farà risvegliare il processo bloccato. `seL4_ReplyRecv` fa lo stesso, ma invia la risposta e blocca l'*endpoint* fornito in una chiamata di sistema combinata.

Ogni *thread* ha un *buffer* che contiene il *payload* del messaggio IPC composto da dati e *capability*. Il mittente del messaggio specifica la lunghezza e il *kernel* copia questa quantità tra il mittente e il destinatario dell'*IPC buffer*. Quest'ultimo contiene un'area limitata di registri di messaggio

(*message registers*, abbreviato MR), che sono utilizzati per trasmettere dati sull'IPC. Ogni registro ha dimensione di una parola (*word*) (dimensione relativa alla macchina) e la lunghezza massima di un messaggio è contenuta nella costante `seL4_MsgMaxLength`. Per caricare un messaggio all'interno del *buffer* è possibile utilizzare `seL4_SetMR`, mentre per estrarlo `seL4_GetMR`; la quantità di parole che possono entrare in un registro è disponibile nella costante `seL4_FastMessageRegisters`.

Insieme al messaggio, il *kernel* consegna il *badge* dell'*endpoint capability*, sul quale il mittente ha fatto l'invocazione per mandare il messaggio. È possibile assegnare un *badge* all'*endpoint*, utilizzando `seL4_CNode_Mint` oppure `seL4_CNode_Mutate`; una volta che è stato messo il *badge* sull'*endpoint*, questo viene trasferito a tutti i destinatari che ricevono un messaggio su quell'*endpoint*.

SeL4, per codificare la descrizione di un messaggio IPC, usa la struttura dati `seL4_MessageInfo_t`,la quale ha la dimensione di una parola ed è composta dai seguenti campi:

- `length` la quantità di dati nel messaggio;
- `extraCaps` numero di *capability* nel messaggio;
- `capsUnwrapped` marca le *capability unwrapped* del *kernel*;
- `label` dati che verranno trasferiti che non sono stati modificati dal *kernel*.

Come già accennato, insieme ai dati, attraverso l'IPC, è possibile scambiare anche *capability*. In gergo questo viene chiamato *cap transfer*:

```
//Invio di una capability via IPC
seL4_MessageInfo info = seL4_MessageInfo_new(0, 0, 1, 0);
seL4_SetCap(0, free_slot);
seL4_Call(endpoint, info);

//Ricezione di una capability
seL4_SetCapReceivePath(cnode, badged_endpoint, seL4_WordBits);
seL4_Recv(endpoint, &sender);
```

Il numero di *capability* trasferite è codificato nella struttura dati `seL4_MessageInfo_t` come `extraCaps`. Inoltre seL4 può fare la cosiddetta *unwrap* (scartare) delle *capability* sull'IPC: se l'n-esima *capability* nel messaggio si riferisce all'*endpoint* attraverso il quale il messaggio viene inviato, la *capability* viene *unwrapped*: il suo badge viene inserito nell'n-esima posizione dell'IPC *buffer* del destinatario (`caps_or_badges`) e il *kernel* imposta l'n-esimo *bit* nel campo `capsUnwrapped` del `seL4_MessageInfo_t`.

L'unico modo che hanno i processi per comunicare, nei sistemi basati su *microkernel*, è attraverso l'utilizzo dell'IPC. Essendo tutti i servizi a livello utente si può intuire che di questa funzionalità ne verrà fatto un utilizzo massiccio. Quindi è necessario che sia realizzata al meglio e che magari si prevedano scorciatoie per renderla ancora più efficiente, in quanto da essa dipendono le prestazioni dell'intero sistema.

Per soddisfare questa esigenza è stato introdotto il *fastpath*, cioè un cammino nel *kernel* altamente ottimizzato, che garantisce velocità nell'IPC. Per potersi definire tale deve soddisfare cinque condizioni:

- devono essere usate le *system call* `seL4_Call` o `seL4_ReplyRecv`;
- i dati del messaggio devono entrare nel registro `seL4_FastMessageRegisters`;
- i processi devono avere spazi di indirizzi validi;
- non dovrebbero essere trasferite *capability*;
- nessun altro *thread* nello *scheduler*, con priorità superiore a quello sbloccato dall'IPC, può essere in esecuzione.

In questa sezione l'esercizio è un po' diverso. Non c'è un unico *file main* in cui è contenuto tutto il codice, ma c'è un `server.c` e due *client* `client_1.c` e `client_2.c`, i quali manderanno dei messaggi al *server*, che farà da *echo*; tutti i processi hanno accesso ad un unico *endpoint capability*, che fornisce accesso allo stesso *endpoint object*. Al primo avvio si ha questo *output*:

```
Booting all finished, dropped to user space
Client 2: waiting for badged endpoint
Badged 2
Client 1: waiting for badged endpoint
Badged 1
Assertion failed: seL4_MessageInfo_get_extraCaps(info) == 1
Assertion failed: seL4_MessageInfo_get_extraCaps(info) == 1
```

Gli errori sono dovuti al fatto che entrambi i *client* si mettono in attesa, sull'*endpoint* fornito, di un *badged endpoint* tramite *cap tranfer*, che però il *server* non invierà, in quanto esso risponde solo ai messaggi dei *client*:

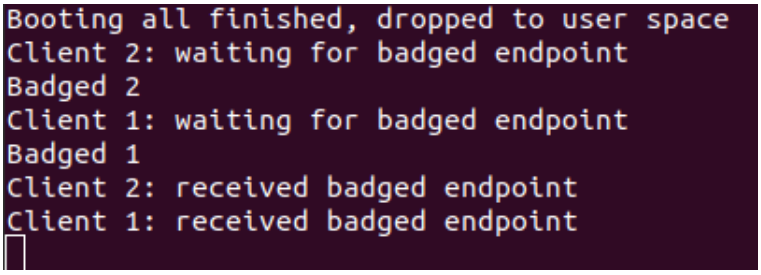
```
// cslot containing IPC endpoint capability
extern seL4_CPtr endpoint;
// cslot containing a capability to the cnode of the server
extern seL4_CPtr cnode;
```

```
// empty cslot
extern seL4_CPtr free_slot;

int main(int c, char *argv[]) {

  seL4_Word sender;
    seL4_MessageInfo_t info = seL4_Recv(endpoint, &sender);
    while (1) {
      seL4_Error error;
        if (sender == 0) {

             /* No badge! give this sender a badged copy of the endpoint */
             seL4_Word badge = seL4_GetMR(0);
             seL4_Error error = seL4_CNode_Mint(cnode, free_slot,
                                seL4_WordBits, cnode, endpoint,
                                seL4_WordBits, seL4_AllRights, badge);
             printf("Badged %lu\n", badge);

             // TODO

             /* reply to the sender and wait for the next message */
             seL4_Reply(info);

             /* now delete the transferred cap */
             error = seL4_CNode_Delete(cnode, free_slot, seL4_WordBits);
             assert(error == seL4_NoError);

             /* wait for the next message */
             info = seL4_Recv(endpoint, &sender);
```

Dunque, per risolvere questo problema si imposterà il *cap transfer* in modo tale che i *client* ricevano il *badged endpoint*:

```
info = seL4_MessageInfo_new(0, 0, 1, 0);
seL4_SetCap(0, free_slot);
```

Compilando e riavviando il programma sembra che non ci siano problemi tranne per il fatto che il sistema si blocca, come si vede nella figura sottostante:



Ciò succede perché al *server* manca l'implementazione della sua funzione di *echo* dei messaggi che gli vengono inviati; tale funzione può essere fatta scorrendo e stampando a video il contenuto dei *message register*.
I `client_1` manda le stringhe "quick", "fox", "over" e "lazy", mentre il `client_2` le stringhe "the", "brown", "jumps", "the" e "dog":

```
for (int i = 0; i < seL4_MessageInfo_get_length(info); i++) {
printf("%c", (char) seL4_GetMR(i));
}
printf("\n");
```

A questo punto però si vedrà stampata a video sempre la stessa parola `the` in *loop* perché il server non manda un *feedback* di risposta al *client* e di conseguenza continua a stampare l'ultima parola ricevuta:

```
for (int i = 0; i < seL4_MessageInfo_get_length(info); i++) {
  printf("%c", (char) seL4_GetMR(i));
}
printf("\n");

// reply to the client and wait for the next message
info = seL4_ReplyRecv(endpoint, info , &sender);
```

Rieseguendo, (l'*output* sarà la stampa a video prima di tutte le parole inviate dal `client_2`, seguite da quelle del `client_1`) si può modificare il codice in modo tale da alternare le stampe dei due *client*, utilizzando `seL4_CNode_SaveCaller` e `free_slot` per salvare le risposte:

```
for (int i = 0; i < seL4_MessageInfo_get_length(info); i++) {
printf("%c", (char) seL4_GetMR(i));
}
printf("\n");

error = seL4_CNode_SaveCaller(cnode, free_slot, seL4_WordBits);
assert(error == 0);
info = seL4_Recv(endpoint, &sender);
for (int i = 0; i < seL4_MessageInfo_get_length(info); i++) {
   printf("%c", (char) seL4_GetMR(i));
}
printf("\n");
seL4_Send(free_slot, seL4_MessageInfo_new(0, 0, 0, 0));

// reply to the client and wait for the next message
info = seL4_ReplyRecv(endpoint, info, &sender);
```

Una volta eseguite tutte le correzioni, l'*output* finale sarà il seguente:

```
Client 2: received badged endpoint
the
Client 1: received badged endpoint
quick
fox
brown
jumps
over
lazy
the
dog
```

Il codice completo del `server` è riportato in [18].
Il codice completo del `client_1` è riportato in [16].
Il codice completo del `client_2` è riportato in [17].

BIBLIOGRAFIA

[1] SeL4 Foundation. Tutorials. URL: https://docs.sel4.systems/Tutorials/. (Cited on page 23.)

[2] Sel4 foundation. Api reference, 2023. URL: https://docs.sel4.systems/projects/sel4/api-doc.html. (Cited on page 23.)

[3] SeL4 Foundation. l4v, 2023. Ultima modifica 19 luglio 2023. URL: https://github.com/seL4/l4v. (Cited on page 21.)

[4] 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 18.)

[5] Gernot Heiser. The sel4 microkernel - an introduction. *The seL4 Foundation*, Revision 1.2, 2020. (Cited on pages 5 and 13.)

[6] JavaTpoint. Hard and soft real-time operating system. URL: https://www.javatpoint.com/hard-and-soft-real-time-operating-system.

[7] Wikipedia, l'enciclopedia libera. Hypervisor, 2023. Ultima modifica 25 luglio 2023. URL: https://en.wikipedia.org/wiki/Hypervisor.

[8] Wikipedia, l'enciclopedia libera. Isabelle (proof assistant), 2023. Ultima modifica 1 marzo 2023. URL: https://en.wikipedia.org/wiki/Isabelle_(proof_assistant).

[9] Wikipedia, l'enciclopedia libera. Kernel, 2023. Ultima modifica 7 giu 2023. URL: https://it.wikipedia.org/wiki/Kernel. (Cited on page 9.)

[10] Wikipedia, l'enciclopedia libera. L4 microkernel family, 2023. Ultima modifica 29 maggio 2023. URL: https://en.wikipedia.org/wiki/L4_microkernel_family.

[11] Wikipedia, l'enciclopedia libera. Memoria virtuale, 2023. Ultima modifica 16 giugno 2023. URL: https://it.wikipedia.org/wiki/Memoria_virtuale.

[12] Wikipedia, l'enciclopedia libera. Operating system, 2023. Ultima modifica 16 luglio 2023. URL: https://en.wikipedia.org/wiki/Operating_system.

[13] Wikipedia, l'enciclopedia libera. Principle of least privilege, 2023. Ultima modifica 2 agosto 2023. URL: https://en.wikipedia.org/wiki/Principle_of_least_privilege.

[14] Wikipedia, l'enciclopedia libera. Qemu, 2023. Ultima modifica 30 giugno 2023. URL: https://en.wikipedia.org/wiki/QEMU.

[15] Elia Matteini. Tutorial capability codice completo, 2023. URL: https://github.com/Elia-dev/Tesi/blob/main/sel4-tutorials-manifest/capabilities/src/main.c. (Cited on page 26.)

[16] Elia Matteini. Tutorial IPC codice completo client_1, 2023. URL: https://github.com/Elia-dev/Tesi/blob/main/sel4-tutorials-manifest/ipc/client_1.c. (Cited on page 41.)

[17] Elia Matteini. Tutorial IPC codice completo client_2, 2023. URL: https://github.com/Elia-dev/Tesi/blob/main/sel4-tutorials-manifest/ipc/client_2.c. (Cited on page 41.)

[18] Elia Matteini. Tutorial IPC codice completo server, 2023. URL: https://github.com/Elia-dev/Tesi/blob/main/sel4-tutorials-manifest/ipc/server.c. (Cited on page 41.)

[19] Elia Matteini. Tutorial threads codice completo, 2023. URL: https://github.com/Elia-dev/Tesi/blob/main/sel4-tutorials-manifest/threads/threads.c. (Cited on page 36.)

[20] Elia Matteini. Tutorial untyped capability codice completo, 2023. URL: https://github.com/Elia-dev/Tesi/blob/main/sel4-tutorials-manifest/untyped/src/main.c. (Cited on page 30.)

[21] Elia Matteini. Tutorial virtual memory management codice completo, 2023. URL: https://github.com/Elia-dev/Tesi/blob/

main/sel4-tutorials-manifest/mapping/src/main.c. (Cited on page 32.)

[22] 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 18.)

[23] Ninja-build. The ninja build system, 2022. Ultima modifica 30 agosto 2022. URL: https://ninja-build.org/manual.html.

[24] A. Silberschatz, P.B. Galvin, and G. Gagne. *Sistemi Operativi - Concetti ed esempi*. Pearson, 10 edition, 2019.