1 Lezione del 15-04-25

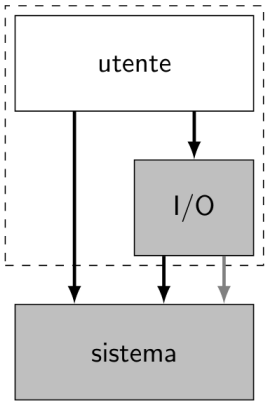
1.1 Gestione di primitive di I/O quasi atomiche nel kernel reale

Abbiamo che quanto abbiamo visto finora sulle primitive di I/O non corrisponde propriamente al kernel reale: avevamo infatti il problema di dover realizzare primitive *"quasi"* atomiche, nel senso che potevano essere interrotte dalle primitive semaforiche (che usavano per sincronizzazione e mutua esclusione), ma avevano anche accesso alle strutture dati sistema (in particolare le code processi), per cui bisogna fare attenzione nella loro implementazione di non toccare parti sensibili, in modo da evitare errori di difficile rilevazione.

Possiamo migliorare leggermente questa architettura separando il **modulo** sistema (inteso come un insieme di file compilati e collegati insieme) dal modulo di I/O, cioè stabilendo la seguente distinzione:

- **Modulo sistema**, implementato in `sistema.cpp` e `sistema.s` (parti C++ e assembler);
- **Modulo di I/O**, implementato in `io.cpp` e `io.s` (come sopra).

Avremo quindi che la parte I/O dipenderà dalla parte sistema, ma non avrà accesso a tutte le sue struttura dati: ad esempio non avra accesso alle code processi. Si andrà quindi a creare la seguente struttura:



dove le frecce nere indicano primitive accessibili a livello utente e le frecce grigie indicano primitive accessibili a livello sistema.

Prima di capire il perché di questa distinzione vediamo di capire a quale livello lavora il modulo di I/O.

Abbiamo visto in 8.1.1 come il registro CS ci permette di definire 4 livelli, o *ring*, e di come storicamente i ring intermedi (1 e 2) venivano usati per i driver (quindi, per i nostri scopi, i moduli di I/O): Oggi, però, questa distinzione non esiste più (assieme

CS	Ring	Tipo
00	Ring 0	Kernel (sistema)
01	Ring 1	Driver *(non più supportato)*
10	Ring 2	/ /
11	Ring 3	Utente

alla memoria segmentata, si perde anche questa funzionalità del registro CS), e gli unici

livelli permessi sono 0 (che abbiamo assunto come livello sistema) e 3 (che abbiamo assunto come livello utente).

Dovremo quindi porre il modulo di I/O in livello sistema, in modo che questo possa avere accesso alle funzioni privilegiate di cui vogliamo privare l'utente. L'unico vantaggio dell'architettura adottata sarà quindi quello di impedire al modulo di I/O di intaccare le strutture dati sensbili lato sistema. Ad esempio, non si potranno nemmeno usare erroneamente le `salva_stato()` o `carica_stato()` nelle primtive di I/O, in quanto queste non saranno nemmeno definite nel modulo di I/O.

1.1.1 Processi driver

Avremo quindi che i driver saranno rappresentati da processi, detti **processi esterni**: le primitivi accessibili solo da livello sistema dal modulo di I/O saranno appunto le primitive per la creazione di processi esterni (la `activate_pe()`), e altre primitive utili al modulo di I/O (a cui l'utente non dovrà avere accesso). Queste saranno Il sistema avrà quindi il compito di definire, su richiesta del modulo di I/O **handler** caricati nella IDT, che si occuperanno di gestire le interruzioni esterne semplicemente indirizzandole ai processi di I/O relativi, creati anch'essi dal modulo di I/O.