



UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

**Progetto e realizzazione di una  
estensione VSCode per il debugging di  
un nucleo multiprogrammato**

Relatore:

**Prof: Giuseppe Lettieri**

**Prof: Luigi Leonardi**

Candidato:

**Francesco Mignone**

---

ANNO ACCADEMICO 2023/2024

# Abstract

Questo elaborato descrive la progettazione e l'implementazione di un'estensione per VS Code che facilita il debugging del nucleo multiprogrammato didattico. Gli obiettivi principali dell'estensione includono la possibilità di impostare e gestire breakpoints, visualizzare variabili in tempo reale e eseguire codice passo-passo. Per raggiungere questi obiettivi, l'estensione utilizza il Debug Adapter Protocol (DAP) per interfacciarsi con strumenti di debugging tradizionali come GDB.

Il lavoro presentato in questa tesi comprende un'analisi dettagliata dei requisiti funzionali e non funzionali, l'analisi dell'architettura dell'estensione e l'implementazione delle principali funzionalità. lo scopo dell'estensione è semplificare significativamente il processo di debugging del nucleo, offrendo un'interfaccia utente intuitiva e funzionalità avanzate di debugging.

Vengono inoltre discusse le limitazioni dell'estensione e vengono proposte possibili direzioni per futuri miglioramenti, tra cui l'aggiunta di ulteriori funzionalità e l'ottimizzazione delle prestazioni.

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Contesto e motivazione . . . . .	3
<b>2</b>	<b>Ambiente e strumenti</b>	<b>5</b>
2.1	Il debugger - GDB e QEMU . . . . .	5
2.1.1	Quick emulator - QEMU . . . . .	5
2.1.2	GNU Debugger - GDB . . . . .	5
2.2	L'architettura del debugger di VS Code . . . . .	6
2.2.1	Debug Adapter Protocol e Debug Adapter . . . . .	7
2.2.2	Logpoints . . . . .	11
2.3	Webview di VS Code . . . . .	11
2.4	Requisiti dell'estensione . . . . .	13
<b>3</b>	<b>Implementazione</b>	<b>15</b>
3.1	Build e connessione al nucleo . . . . .	17
3.2	nucleo_vscode.py . . . . .	19
3.3	Estensione Nucleo Debug . . . . .	22
3.3.1	Creazione dell'estensione . . . . .	22
3.3.2	Caricamento dell'estensione . . . . .	22
3.3.3	Richiesta del comando GDB . . . . .	23
3.3.4	Webview . . . . .	25
<b>4</b>	<b>Utilizzo del debugger</b>	<b>29</b>
4.1	Breakpoint e Logpoint . . . . .	30
4.2	Azioni di debug . . . . .	31
4.3	Pannello di sinistra . . . . .	32
4.4	Informazioni aggiuntive . . . . .	33
<b>5</b>	<b>Conclusione e sviluppi futuri</b>	<b>35</b>
<b>6</b>	<b>Sviluppo e contributi</b>	<b>37</b>
6.1	Aggiunta di comandi . . . . .	38
6.1.1	nucleo_vscode.py . . . . .	40

INDICE	1
--------	---

---

7 Ringraziamenti	41
------------------	----



# Capitolo 1

## Introduzione

TODO: Forse emglio togliere la subsection

### 1.1 Contesto e motivazione

Il nucleo di un sistema operativo è il componente software fondamentale che gestisce le risorse hardware e software. A causa della sua complessità e della sua importanza critica, il debugging del nucleo è un compito altamente specialistico che richiede strumenti potenti e una profonda comprensione del sistema. Tuttavia, gli strumenti tradizionali come GDB e QEMU possono essere difficili da utilizzare, soprattutto per i nuovi sviluppatori o per coloro che preferiscono interfacce grafiche più intuitive.

Visual Studio Code (VSCode) è un editor di codice sorgente open-source sviluppato da Microsoft. VS Code è diventato molto popolare tra gli sviluppatori grazie alla sua facilità d'uso e alla sua capacità di supportare molti linguaggi di programmazione grazie alla sua vasta gamma di estensioni disponibili. Tuttavia, al momento della scrittura di questa tesi, non esiste un'estensione dedicata per il debugging del nucleo didattico su VSCode.

L'obiettivo di questa tesi è sviluppare un'estensione per VS Code che renda il debugging del nucleo più accessibile e intuitivo, permettendo agli sviluppatori di beneficiare dell'ambiente user-friendly di VSCode senza rinunciare alla potenza degli strumenti tradizionali.



# Capitolo 2

## Ambiente e strumenti

### 2.1 Il debugger - GDB e QEMU

#### 2.1.1 Quick emulator - QEMU

QEMU é un emulatore open-source, permette di emulare l'architettura di un processore. Permette quindi l'utilizzo di vari sistemi operativi ad un livello di virtualizzazione Kernel-Based (Kernel Virtual Machine - KVM) con il beneficio di prestazioni vicine all'hardware. Per il nostro utilizzo QEMU emula un sistema x86-64.

#### 2.1.2 GNU Debugger - GDB

GNU Debugger (GDB) è un debugger portatile, permette quindi di testare e effettuare il debug di programmi. Eseguire il programma in questo ambiente controllato permette al programmatore di tenere traccia dell'esecuzione e monitorare le risorse al fine di individuare un eventuale malfunzionamento nel codice. Per la realizzazione dell'estensione utilizzeremo la funzione di debug remoto per connetterci ad un socket di sistema utilizzato da QEMU per il debug. GDB utilizza delle chiamate di sistema chiamate `process trace` (`ptrace`).

#### Breakpoints

Un breakpoint permette al programma in esecuzione all'interno di un debugger di interrompere il flusso in un determinato punto. Si realizzano sostituendo all'istruzione, alla quale si vuole fermare l'esecuzione, una speciale istruzione la quale solitamente invia un segnale SIGTRAP, il quale verrà catturato dal debugger. Il procedimento di sostituzione è eseguito dal debugger stesso prima di avviare l'esecuzione, nel caso di GDB il programmatore deve eseguire il comando `break [arg]` dove l'argomento può essere la specifica linea di codice o un simbolo.



### Continue e Stop

Tramite i comandi `continue` e `stop` possiamo rispettivamente, a seguito di un interruzione, continuare la normale esecuzione del codice oppure interrompere l'esecuzione del programma.

### Step Over

Permette di proseguire alla prossima istruzione senza entrare nei componenti interni dell'istruzione a cui siamo fermi attualmente.

### Step Into

Rende possibile seguire il codice riga-per-riga entrando anche nei componenti interni e subroutine.

### Step Out

Quando all'interno di una subroutine e si vuole risalire al chiamante, il comando `stepOut` permette di far continuare l'esecuzione fino a ritornare all'istruzione successiva del chiamante.

### Analisi delle variabili

Durante l'esecuzione vi può essere la necessità di osservare come il valore o il tipo di una variabile cambi. Inoltre è possibile cambiare il valore delle variabili ad esecuzione avviata.

### Call Stack

Il call stack, o program stack, è una struttura che permette di raccogliere informazioni su tutte le subroutine di un programma in esecuzione. Tale struttura è utile per tenere traccia di quale routine ha il controllo del flusso di istruzioni e a chi deve restituire tale controllo al termine della propria esecuzione.

### Comandi personalizzati

Ulteriore funzione di GDB è la possibilità di estendere le funzionalità, tramite script in python, come la creazione di comandi personalizzati.

## 2.2 L'architettura del debugger di VS Code

Tipicamente se si vuole creare un debugger e la sua interfaccia grafica bisogna implementare l'intera applicazione. Microsoft ha realizzato un protocollo che

permette di comunicare con i debugger: il Debug Adapter Protocol (DAP). VSCode, o altri applicativi, tramite il DAP si interfaccia non direttamente al debugger ma a un attore intermedio, il Debug Adapter (DA) il quale si occupa di trasformare le richieste dell'applicazione in comandi per il debugger di destinazione.

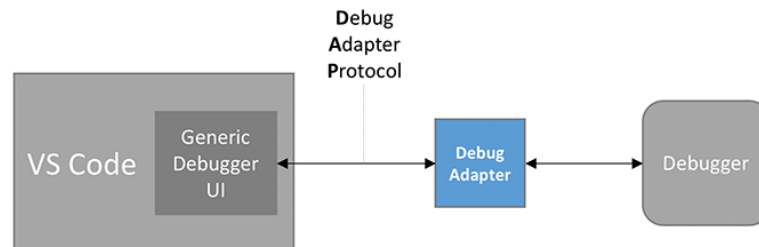


Figura 2.1: DAP e DP

Rende possibile quindi la realizzazione di una generica interfaccia di debug la quale poi si occupa di comunicare con uno o più DP. Inoltre i Debug Adapters possono essere riutilizzati in diversi ambienti di sviluppo, eliminando la necessità di crearne uno specifico per ogni esigenza.

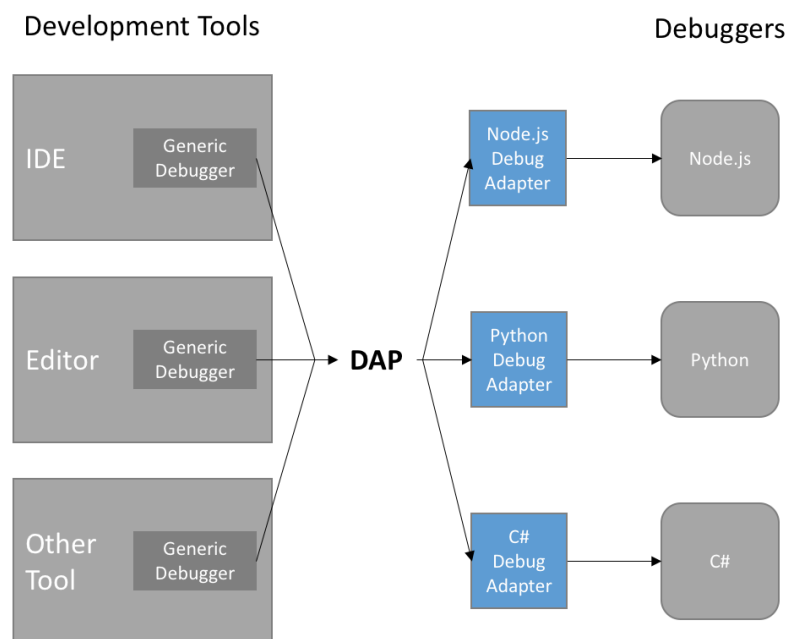


Figura 2.2: Ambienti di sviluppo multipli

### 2.2.1 Debug Adapter Protocol e Debug Adapter

Analizziamo come avviene la connessione e scambio di messaggi tra l'applicativo e il debugger. Gli strumenti di sviluppo possono interagire con il Debug

Adapter in due modi:

- Modalità a singola sessione: l'applicazione avvia una sessione di debug singola e comunica attraverso `stdin` e `stdout`. Alla fine della sessione, il Debug Adapter viene terminato
- Modalità a sessioni multiple: l'applicazione di debug si connette a un debugger già avviato in precedenza e si disconnette al termine della sessione.

Il DAP supporta molte funzionalità, ciascuna rappresentata da una "capacità". Quando inizia una sessione di debug, lo strumento di sviluppo invia una richiesta di inizializzazione per capire le funzionalità dell'adattatore. Dopo l'inizializzazione, il Debug Adapter è pronto per accettare richieste di avvio o collegamento.

### Breakpoint

Lo strumento di sviluppo gestisce i breakpoint inviando le informazioni di configurazione all'adattatore prima dell'esecuzione del programma. Quando il programma si ferma, l'adattatore, solitamente, invia un evento di stop con il motivo e l'id del thread. Lo strumento di sviluppo richiede i thread e lo stacktrace, e tramite essi risalire alle variabili.

### Inizio della sessione di debug

Dopo aver stabilito una connessione, lo strumento di sviluppo comunica con l'adattatore tramite il protocollo di base. Il protocollo di base é implementato tramite lo scambio di messaggi composti da un'intestazione e un contenuto, chiamati `ProtocolMessage`.

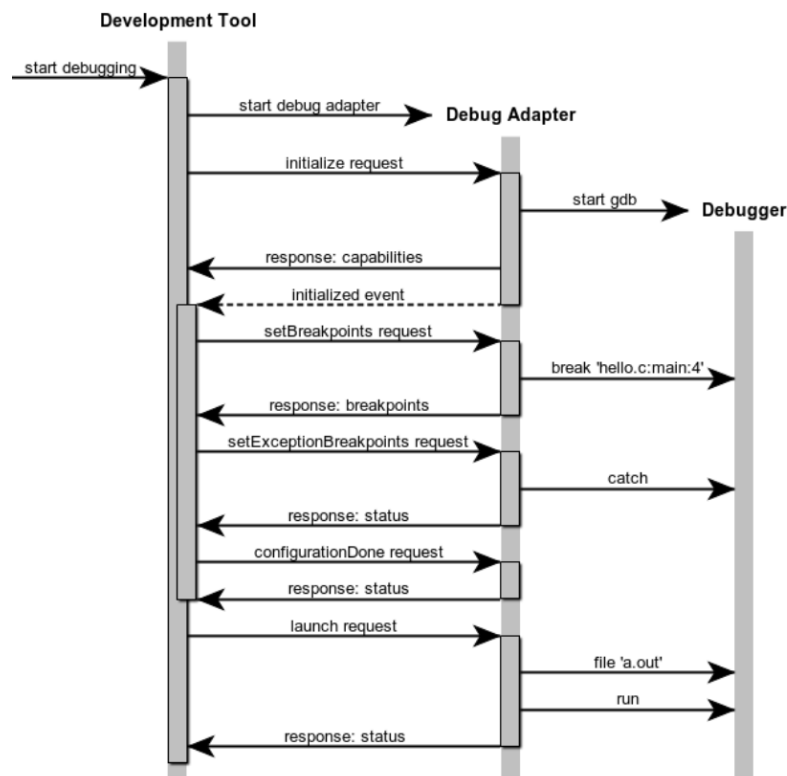


Figura 2.3: Esempio di avvio di una sessione di debug[[Microsoft\(2024a\)](#)]

Quando inizia una sessione di debug, lo strumento di sviluppo deve comunicare con l'adattatore di debug che implementa il Protocollo di Adattatore di Debug (Debug Adapter Protocol, DAP). Il protocollo di base é implementato tramite lo scambio di messaggi composti da un'intestazione e un contenuto, chiamati `ProtocolMessage`.

---

```
1 interface ProtocolMessage {
2     /**
3      * Sequence number of the message (also known as message ID). The 'seq'
4      * for
5      * the first message sent by a client or debug adapter is 1, and for each
6      * subsequent message is 1 greater than the previous message sent by that
7      * actor. 'seq' can be used to order requests, responses, and events,
8      * and to
9      * associate requests with their corresponding responses. For protocol
10     * messages of type 'request' the sequence number can be used to cancel
11     * the
12     * request.
13     */
14     seq: number;
15
16     /**
17     * Message type.
18     * Values: 'request', 'response', 'event', etc.
19     */
20     type: 'request' | 'response' | 'event' | string;
21 }
```

---

Figura 2.4: ProtocolMessage[Microsoft(2024b)]

### Termine della sessione di debug

Il processo per terminare la sessione è diverso a seconda di come si è avviata la sessione, "avviato" o "agganciata":

- debugger "avviato": se il Debug Adapter implementa la richiesta di interruzione, allora la sessione viene terminata correttamente. Se non dovesse essere supportata la sessione continua a essere attiva fino a quando il debugger stesso non invia il comando di terminazione forzata
- debugger "agganciato": l'applicazione di debug invia una richiesta di disconnessione al Debug Adapter. Questo permette al debugger di cessare la connessione con l'applicativo e continuare l'esecuzione

se la sessione di debug termina, e il Debug Adapter è opportunamente configurato, un messaggio di corretta terminazione di sessione viene inviato all'applicazione di debug del programmatore.

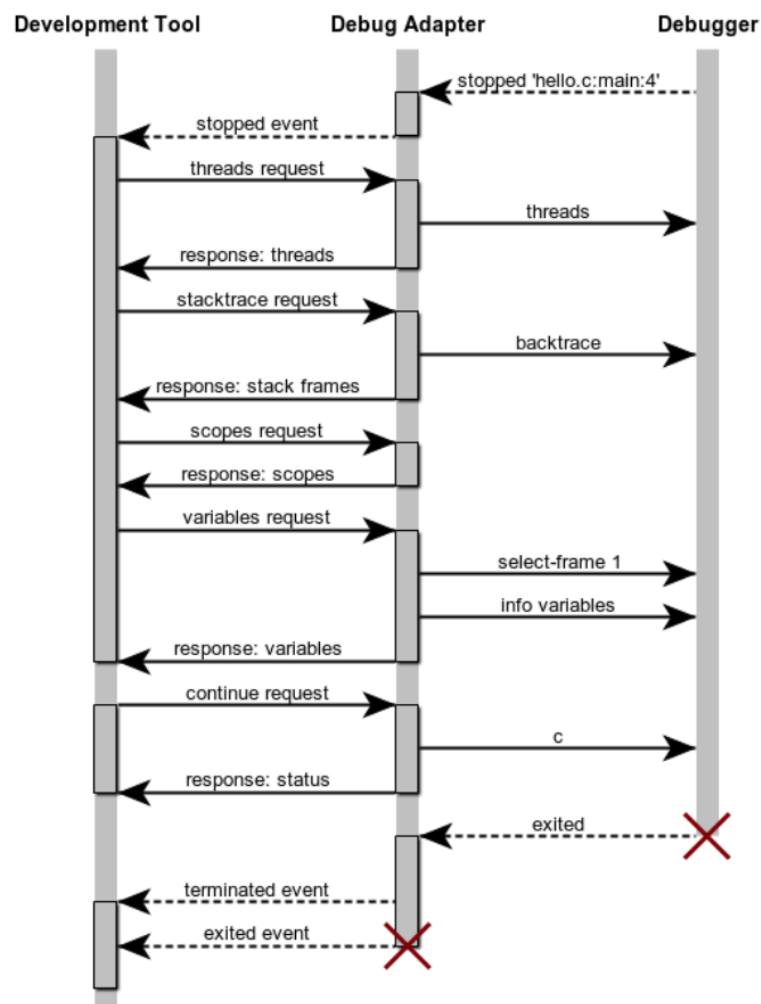


Figura 2.5: Esempio terminazione di una sessione di debug[Microsoft(2024a)]

### 2.2.2 Logpoints

I logpoint sono una variante dei breakpoint. Permettono, senza interrompere l'esecuzione, di controllare il valore di una o più variabili e mostrando il risultato nell'console di debug di VSCode. Sono molto utili per evitare aggiungere codice di log all'interno del programma.

## 2.3 Webview di VS Code

VSCode mette a disposizione la possibilità di creare nuove schede nelle quali un utente può visualizzare contenuti personalizzati. Le webview sono molto simili a `iframe` e sono capaci di renderizzare qualsiasi contenuto HTML e scambio di informazioni tra la webview e VSCode.

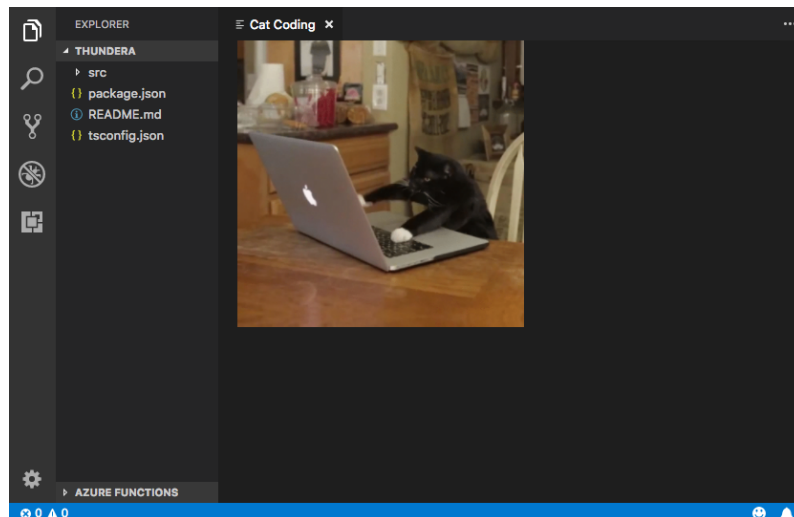


Figura 2.6: Esempio di una webview in VSCode

## 2.4 Requisiti dell'estensione

Le funzionalità richieste per la prima versione dell'estensione sono l'implementazione delle operazioni di base di debug: continue, stop, step over, step into, step out e l'analisi delle variabili. Per informazioni specifiche sul nucleo é richiesto di mostrare una lista dei processi attualmente in esecuzione.





## Capitolo 3

# Implementazione

Per realizzare la nostra estensione andremo a estendere le funzionalità di base del debugger di VSCode. I comandi di base verranno gestiti dal debugger di VSCode dopo aver configurato opportunemente il collegamento con il GDB. La nostra estensione si occuperà di inviare comandi per la visualizzazione dei processi attualmente in esecuzione e mostrare i dati in una webview di VSCode.

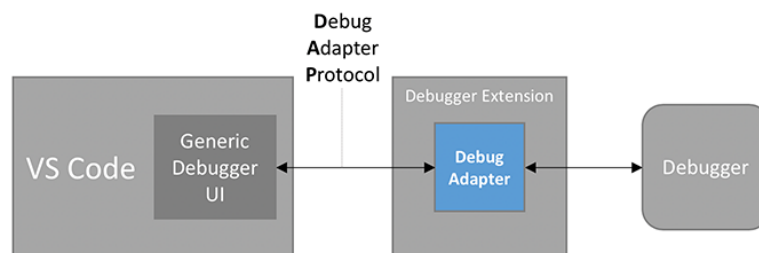


Figura 3.1: Estensione del DP

Prima di instaurare un collegamento con il GDB vi è la necessità di compilare ed eseguire il nucleo nella macchina QEMU. VSCode mette a disposizione il file `./vscode/task.json` all'interno della directory di lavoro. Il file permette di dichiarare dei comandi che permettono di automatizzare una moltitudine di operazioni.

### task.json

Il file `task.json`, come si può evincere dall'estensione, è in formato `JSON`. I campi che lo compongono sono spiegati esaustivamente all'interno della pagina dedicata della documentazione di VSCode [Microsoft(2024c)]. In particolare si è configurato il file per eseguire i comandi di `compile` e `boot` presenti nell'ambiente di sviluppo Linux.

---

## compilazione

---

```
1 {  
2   "label": "Compila il nucleo",  
3   "type": "shell",  
4   "command": "compile",  
5   "options": {},  
6   "problemMatcher": [  
7     "$gcc"  
8   ],  
9   "group": "build",  
10  "presentation": {  
11    "echo": true,  
12    "reveal": "always",  
13    "focus": false,  
14    "panel": "shared",  
15    "showReuseMessage": false,  
16    "clear": true  
17  }  
18 }
```

---

Figura 3.2: Compilazione

- **label**: identifica univocamente il nome del task all'interno dell'ambiente
- **type**: come deve essere interpretato il campo **command**, in questo caso come un comando di shell
- **command**: il comando effettivo da eseguire
- **group**: definisce il gruppo a cui il task appartiene all'interno di VSCode
- **presentation**: sono istruzioni per VSCode su come mostrare l'output del comando all'utente

### Avvio in modalità debug

```
1 {
2     "label": "Avvia debug",
3     "type": "shell",
4     "isBackground": true,
5     "command": "boot -g",
6     "dependsOn": "Compila il nucleo",
7     "problemMatcher": [
8         {
9             "pattern": [
10                {
11                    "regexp": ".",
12                    "file": 1,
13                    "location": 2,
14                    "message": 3
15                }
16            ],
17            "background": {
18                "activeOnStart": true,
19                "beginsPattern": "INF",
20                "endsPattern": "."
21            }
22        }
23    ]
24 }
```

Figura 3.3: Avvio in modalità debug

Il task di avvio in modalità debug di QEMU del nucleo richiede un campo aggiuntivo per segnalare a VSCode la terminazione del task. QEMU viene avviato in modalità debug e resta in attesa fino alla connessione del GDB. Il campo `problemMatcher` è configurato in modo tale da attendere che QEMU segnali l'attesa del GDB tramite la riga di output `INF Attendo collegamento da gdb`.

## 3.1 Build e connessione al nucleo

Il prerequisito per il funzionamento dell'estensione è il collegamento al GDB. Per configurarlo è stato creato il file di configurazione `launch.json` all'interno della cartella `./.vscode` dell'ambiente di lavoro.

## lauch.json

---

```
1 "configurations": [  
2   {  
3     "name": "Launch nmd",  
4     "type": "cppdbg",  
5     "request": "launch",  
6     "program": "${workspaceFolder}/build/sistema",  
7     "cwd": "${workspaceFolder}",  
8     "miDebuggerPath": "gdb",  
9     "stopAtEntry": true,  
10    "preLaunchTask": "Avvia debug",  
11    "stopAtConnect": true,  
12    "setupCommands": [  
13      {"text": "cd ${workspaceFolder}"},  
14      {  
15        "text": "source .gdbinitvscode",  
16        "ignoreFailures": true  
17      },  
18    ],  
19  }  
20 ]
```

---

Figura 3.4: launch.json

- **name**: identifica univocamente il nome della configurazione del debugger
- **type**: richiede la tipologia di debugger da utilizzare
- **request**: richiede di lanciare una nuova istanza del debugger
- **program**: indica quale eseguibile deve lanciare il debugger
- **cwd**: imposta il percorso di lavoro del debugger
- **miDebuggerPath**: è l'eseguibile del debugger da avviare
- **preLaunchTask**: richiede quali task eseguire prima di lanciare la connessione
- **setupCommands**: è la lista dei comandi da eseguire all'avvio del debugger

## .gdbinitvscode

GDB permette inoltre di caricare dei file di configurazione al cui interno sono definiti dei comandi di GDB, viene utilizzato per impostare le varie funzioni di visualizzazione, caricamento di ulteriori simboli o caricamento degli script personali.

---

```
1 set $MAX_LIV=4
2 set $MAX_SEM=1024
3 set $SEL_CODICE_SISTEMA=8
4 set $SEL_CODICE_UTENTE=19
5 ...
6 add-symbol-file /home/studenti/CE/lib/ce/boot.bin
7 add-symbol-file build/io
8 add-symbol-file build/utente
9 set arch i386:x86-64:intel
10 target remote gdb-socket
11 ...
12 break sistema.s:start
13 continue
14 delete 1
15 source debug/vscode_nucleo.py
```

---

Figura 3.5: .gdbinitvscode

## 3.2 nucleo\_\_vscode.py

Il file `nucleo_debug.py` contiene la definizione di un nuovo comando per GDB: `process list`. Il comando é stato creato modificando la struttura dello script `nucleo.py`.

---

```

1 def process_dump(proc, indent=0, verbosity=3):
2     write_key("livello", colorize('col_usermode', "utente") if
        proc['livello'] == gdb.Value(3) else colorize('col_sysmode',
        "sistema"), indent)
3     write_key("corpo", dump_corpo(proc), indent)
4 ...
5
6 ...
7     write_key("rip", "{:>18s} {}".format(rip_s[0], " ".join(rip_s[1:])),
        indent)
8     if (verbosity > 2):
9         write_key("cs", dump_selector(readfis(stack + 8)), indent)
10        write_key("rflags", dump_flags(readfis(stack + 16)), indent)
11        write_key("rsp", "{:#18x}".format(readfis(stack + 24)), indent)
12        write_key("ss", dump_selector(readfis(stack + 32)), indent)
13 ...
14 ...
15        gdb.write(colorize('col_proc_hdr', "-- prossima istruzione:\n"),
        indent)
16        show_lines(gdb.find_pc_line(rip), indent)
17    if len(toshow) > 0:
18        if verbosity > 2:
19            gdb.write("\x1b[33m-- campi aggiuntivi:\x1b[m\n", indent)
20            for f in toshow:
21                write_key(f.name, proc[f], indent)
22 ...
23
24 ...
25 class ProcessList(gdb.Command):
26 ...
27     def __init__(self):
28         super(ProcessList, self).__init__("process list", gdb.COMMAND_DATA)
29
30     def invoke(self, arg, from_tty):
31         for pid, proc in process_list(arg):
32             gdb.write("==> Processo {}\n".format(pid))
33             process_dump(proc, indent=4, verbosity=0)
34 ...

```

---

Figura 3.6: nucleo.py

Il codice é stato modificato in modo tale da costruire come output un oggetto [JSON](#)

---

```

1  ...
2
3  def process_dump(pid, proc, indent=0, verbosity=3):
4      proc_dmp = {}
5      proc_dmp['pid'] = pid
6      proc_dmp['livello'] = "utente" if proc['livello'] == gdb.Value(3) else
          "sistema"
7  ...
8
9  ...
10     pila_dmp = {}
11     pila_dmp['start'] = "{:016x} \u279e {:x}):\n".format(vstack, stack)
12     pila_dmp['cs'] = dump_selector(readfis(stack + 8))
13     pila_dmp['rflags'] = dump_flags(readfis(stack + 16))
14     pila_dmp['rsp'] = "{:#18x}".format(readfis(stack + 24))
15     pila_dmp['ss'] = dump_selector(readfis(stack + 32))
16     proc_dmp['pila_dmp'] = pila_dmp
17  ...
18
19  ...
20     cr3 = toi(proc['cr3'])
21     proc_dmp['cr3'] = vm_paddr_to_str(cr3)
22
23     # proc_dmp['nex_ist'] = show_lines(gdb.find_pc_line(rip), indent)
24     if len(toshow) > 0:
25         campi_aggiuntivi = {}
26         for f in toshow:
27             campi_aggiuntivi[f.name] = str(proc[f]),
28             proc_dmp['campi_aggiuntivi'] = campi_aggiuntivi
29
30     return proc_dmp
31  ...
32
33  ...
34  class ProcessList(gdb.Command):
35  ...
36
37  ...
38     def invoke(self, arg, from_tty):
39         out = {}
40         out['command'] = "process_list"
41         out['process'] = []
42         for pid, proc in process_list(arg):
43             out['process'].append(process_dump(pid, proc, indent=4,
                verbosity=0))
44         with open('myfile.txt', 'w') as f:
45             f.write(json.dumps(out))
46             gdb.write(json.dumps(out) + "\n")
47  ...

```

---

Figura 3.7: nucleo\_vscode.py



Il comando `process list` restituisce la lista di tutti i processi in esecuzione e le informazioni relative al contesto del processo.

## 3.3 Estensione Nucleo Debug

Il funzionamento generale dell'estensione si basa sull'ascoltare quando VSCode avvia una sessione di debug. Una volta avviato il debug viene caricata una webview, questa viene aggiornata ad intervalli regolari lanciando i comandi personalizzati di GDB e formattando in HTML il risultato ottenuto dal debugger.

### 3.3.1 Creazione dell'estensione

Dopo aver inizializzato l'ambiente di sviluppo per un'estensione di VSCode grazie a `yeoman`, si procede alla configurazione al file di descrizione dell'estensione `package.json`. All'interno si possono definire vari campi tra cui:

- `name`: identifica univocamente il nome dell'estensione
- `commands`: vettore di eventuali comandi che fornisce l'estensione
- `activationEvents`: vettore di eventi da monitorare ai quali l'estensione viene caricata

in particolare `activationEvents` è stato popolato con l'evento `onDebugResolve:cppdbg`, che permette di caricare l'estensione solo quando viene attivato il debug con tipo `cppdbg`.

### 3.3.2 Caricamento dell'estensione

Al momento del caricamento dell'estensione VSCode dal file principale `extension.ts` chiama la funzione `activate()` al quale interno posizioniamo il codice di inizializzazione.

---

```
1 ...
2
3 export function activate(context: vscode.ExtensionContext) {
4     vscode.debug.onDidStartDebugSession( ()=>{
5         NucleoInfo.createInfoPanel(context.extensionUri);
6     });
7
8     vscode.debug.onDidTerminateDebugSession(() =>{
9         NucleoInfo.currentPanel?.dispose();
10    });
11
12 }
13
14 ...
```

---

Figura 3.8: activate()

- `vscode.debug.onDidStartDebugSession`: permette di monitorare quando viene avviata una sessione di debug e chiamare una funzione che inizializza la webview contenente le informazioni del nucleo
- `vscode.debug.onDidTerminateDebugSession`: permette di monitorare quando viene terminata la sessione di debug e chiamare la funzione per pulire le schede aperte dall'estensione

### 3.3.3 Richiesta del comando GDB

All'interno del costruttore di `NucleoInfo` impostiamo un intervallo tramite `setInterval(...)`, che si occuperà di gestire tutte le richieste dei comandi personalizzati di GDB tramite `customCommand(...)` e aggiornare la webview.

---

```
1 private constructor(panel: vscode.WebviewPanel, extensionUri: vscode.Uri) {
2     this._panel = panel;
3     this._extensionUri = extensionUri;
4
5     // Set the webview's initial html content
6     this._update();
7
8     // Listen for when the panel is disposed
9     // This happens when the user closes the panel or when the panel is
    closed programmatically
10    this._panel.onDidDispose(() => this.dispose(), null,
        this._disposables);
11    ...
12
13    ...
14
15    const session = vscode.debug.activeDebugSession;
16    const updateInfo = async () => {
17        this.process_list = await this.customCommand(session, "process
            list");
18
19        const infoPanel = this._panel.webview;
20        infoPanel.html = this._getHtmlForWebview();
21    };
22
23    interval = setInterval(updateInfo, 500);
24 }
```

---

Figura 3.9: NucleoInfo.constructor

### customCommand()

---

```
1 private async customCommand(session: typeof
    vscode.debug.activeDebugSession, command: string, arg?: any){
2     if(session) {
3         const sTrace = await session.customRequest('stackTrace', {
            threadId: 1 });
4         if(sTrace.stackFrames[0] === undefined){
5             return;
6         }
7         const frameId = sTrace.stackFrames[0].id;
8
9         // Build and exec the command
10        const text = '-exec ' + command;
11        let result = session.customRequest('evaluate', {expression: text,
            frameId: frameId, context:'hover'}).then((response) => {
12            return response.result;
13        });
14        return result
15    }
16 }
```

---

Figura 3.10: NucleoInfo.customCommand()

La funzione `customCommand(...)` preleva il frame di esecuzione del nucleo da GDB e dopo aver costruito il comando da eseguire crea una richiesta al Debug Adapter tramite il metodo `.customRequest`, il quale istanzia una classe `ProtocolMessage` definita dal Debug Adapter Protocol di tipo `request` per richiedere l'esecuzione del comando da parte del GDB.

### 3.3.4 Webview

Dopo aver recuperato tutte le informazioni del nucleo necessarie viene chiamato il metodo `_getHTMLForWebview()`.

---

```

1 private _getHtmlForWebview() {
2     const scriptPathOnDisk = vscode.Uri.joinPath(this._extensionUri,
3         'src/webview', 'main.js');
4     ...
5     ...
6     let sourceDocument = `
7         <!DOCTYPE html>
8         <html lang="en">
9             <head>
10                 <meta charset="UTF-8">
11                 <meta name="viewport" content="width=device-width,
12                     initial-scale=1.0">
13                 <link href="${stylesResetUri}" rel="stylesheet">
14                 <link href="${stylesMainUri}" rel="stylesheet">
15                 <link href="${codiconsUri}" rel="stylesheet" />
16                 <title>Info Nucleo</title>
17             </head>
18             <body>
19                 {{{processList}}}
20                 <script src="${scriptUri}"></script>
21             </body>
22         </html>
23     `;
24
25     let template = Handlebars.compile(sourceDocument);
26     return template({ processList: this.formatProcessList() });
27 }

```

---

Figura 3.11: NucleoInfo.\_\_getHTMLforwebview

Per costruire la pagina HTML che verrà caricata dalla webview utilizziamo la libreria di templating [handlebars](#). Prima di poter utilizzare i dati, essi devono essere convertiti in una struttura JSON e creato il codice HTML per la visualizzazione, utilizziamo la funzione creata appositamente [formatProcessList\(\)](#), non riportata per la sua lunghezza ma disponibile nella repository del progetto [[mignone\(2024\)](#)].

Il codice HTML generato viene assegnato alla webview che si preoccupa di renderizzare la pagina ottenendo il seguente risultato

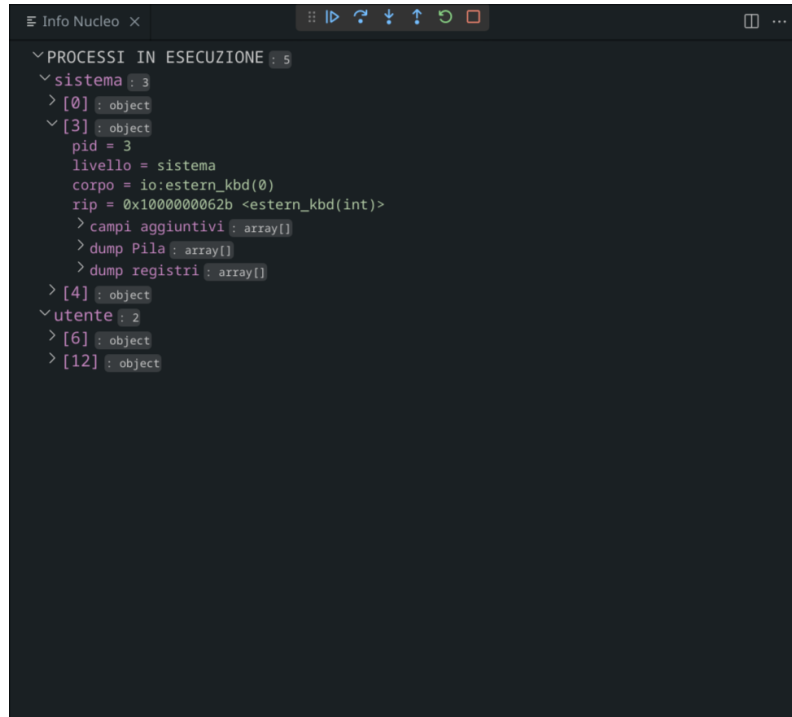


Figura 3.12: Lista dei processi in esecuzione



# Capitolo 4

## Utilizzo del debugger

Introduciamo ora come é possibile utilizzare l'ambiente di debug di VSCode per eseguire il debug del nucleo. È possibile avviare l'interfaccia di debug tramite l'apposita scheda 4.1 (pin 1) e poi dopo aver selezionato la configurazione `launch nmd` dal menù a tendina 4.1 (pin 2) si avvia la sessione premendo il tasto di avvio collocato accanto allo stesso menù.

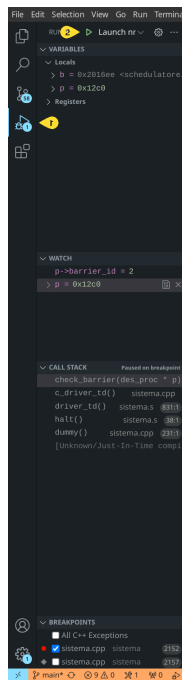


Figura 4.1: Azioni per avviare il debug

oppure premendo il tasto `F5` sulla tastiera.

l'interfaccia che ci viene presentata é la seguente



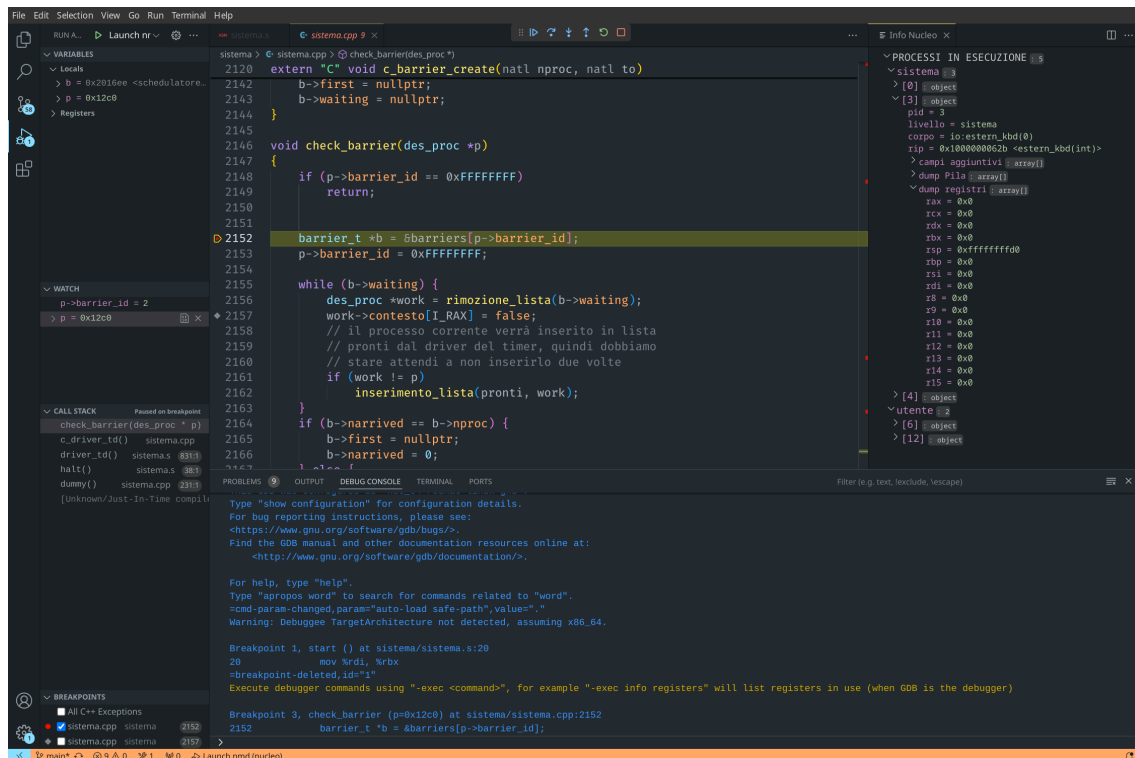


Figura 4.2: Interfaccia di debug di VSCode

Analizziamo le varie finestre che ci vengono proposte.

## 4.1 Breakpoint e Logpoint

All'interno di VSCode vi é la possibilità di inserire breakpoint cliccando sul lato sinistro della riga di codice interessata, VSCode stesso si occuperà poi di comunicare al Debug Adapter la richiesta di inserimento del breakpoint.

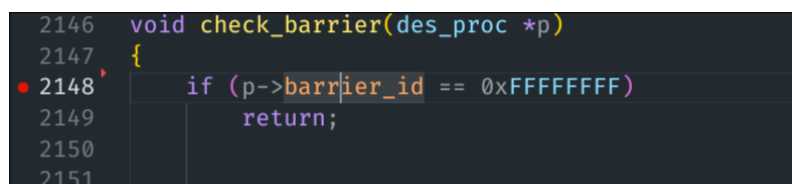


Figura 4.3: Esempio di un breakpoint in VSCode

Similarmente si può inserire un logpoint cliccando con il tasto destro del mouse e selezionando la dicitura logpoint. All'interno del box di testo si possono aggiungere le variabili da osservare tramite `{{variable}}`

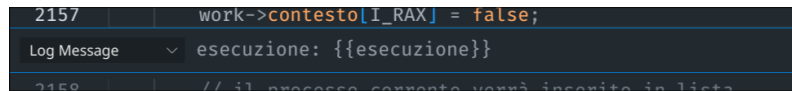


Figura 4.4: Esempio di logpoint

una volta ripresa l'esecuzione del codice possiamo vedere l'output nella console di debug.

## 4.2 Azioni di debug

VSCode mette a disposizione una barra di funzioni 4.5(pin 1) per permettere all'utente di ispezionare il codice tramite i comandi di step over, step in e step out. È possibile anche eseguire azioni come continue, stop e il riavvio della sessione di debug.

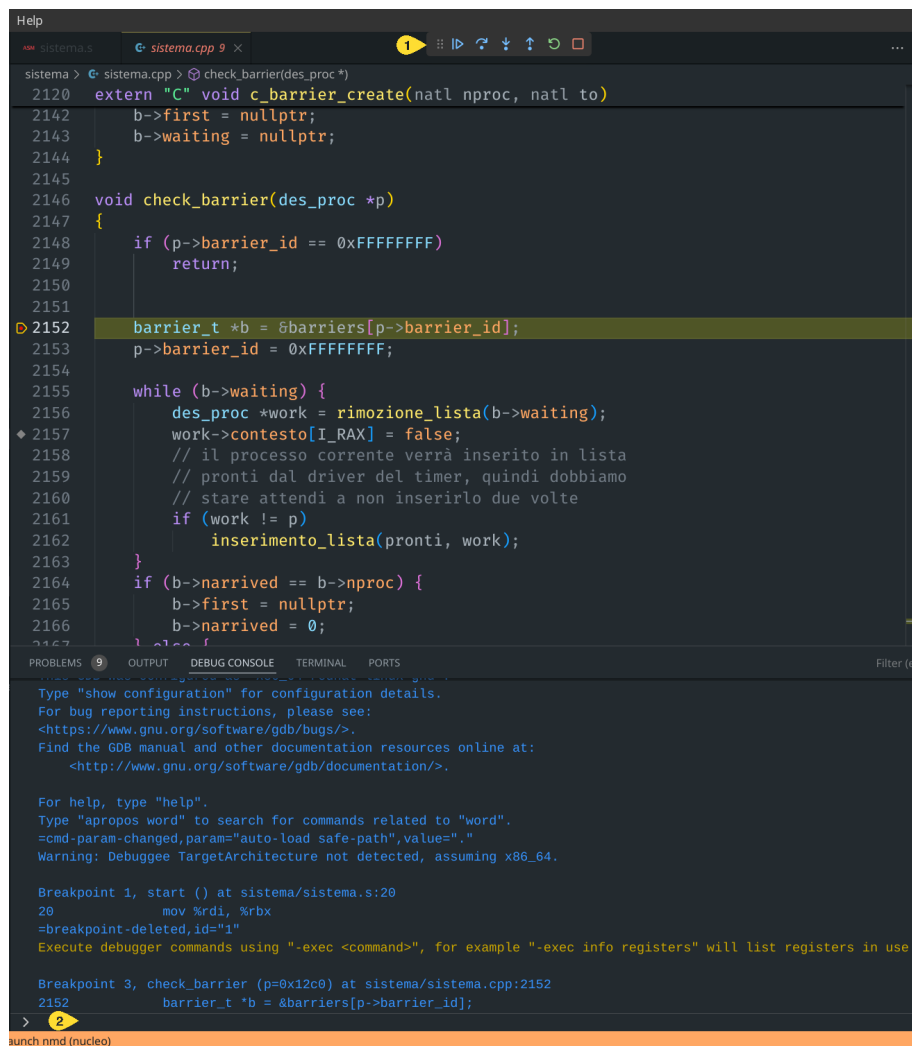


Figura 4.5: Azioni e linea di comando

Inoltre é possibile inviare comandi di GDB direttamente dalla linea di comando 4.5(pin 2) tramite il comando `-exec [GDB command]`.

## 4.3 Pannello di sinistra

Il pannello di sinistra permette di analizzare lo stato delle variabili locali 4.6(pin 2) e lo stato dei registri 4.6(pin 3).

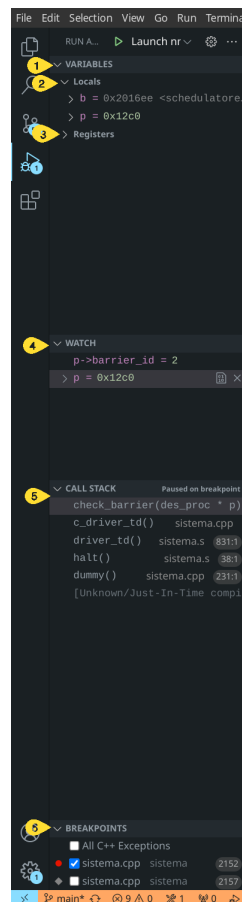


Figura 4.6: Azioni e linea di comando

## Watch

Nel pannello dedicato 4.6(pin 4) possiamo visualizzare le variabili messe sotto osservazione. L'aggiunta può avvenire tramite il pulsante "+" oppure selezionando con il cursore la variabile o l'espressione da analizzare e cliccando con il tasto destro dal menù a tendina selezionare "Add to watch" come mostrato in figura.

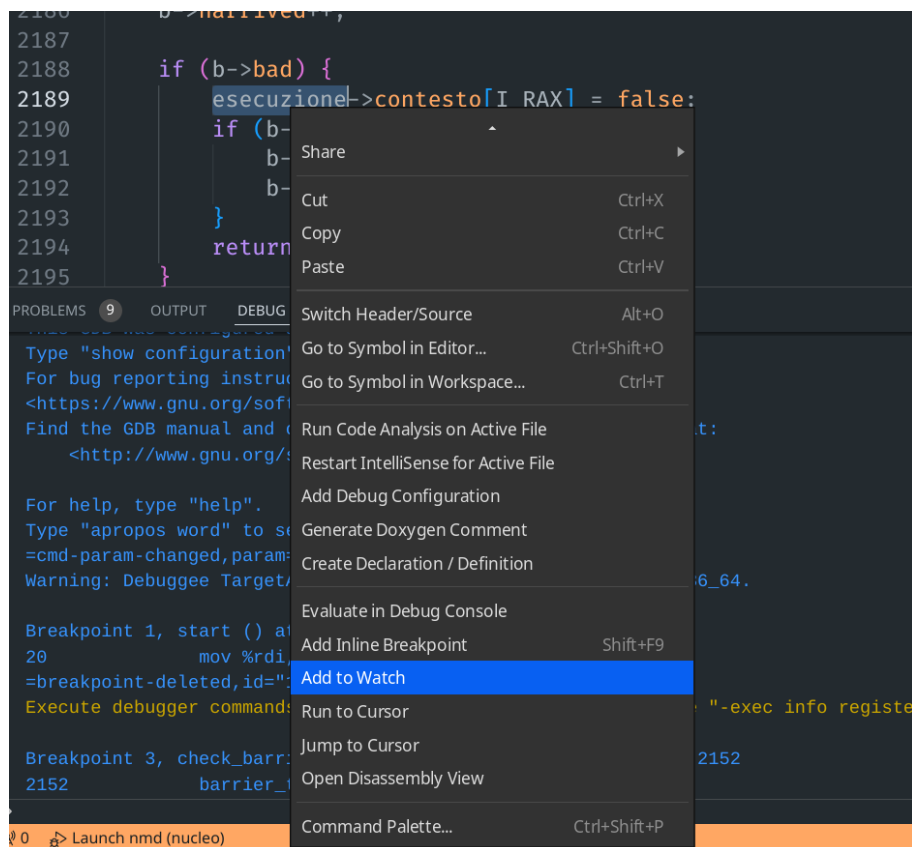


Figura 4.7: Aggiunta al watch di una variabile

## Call stack e breakpoints

È inoltre possibile visualizzare informazioni riguardanti il call stack 4.6(pin 5) e gestire i breakpoint presenti tramite il pannello dedicato 4.6(pin 6)

## 4.4 Informazioni aggiuntive

Sulla parte destra della finestra di debug troviamo la scheda dedicata alle informazioni aggiuntive del nucleo. In questo caso mostra solo i processi in esecuzione

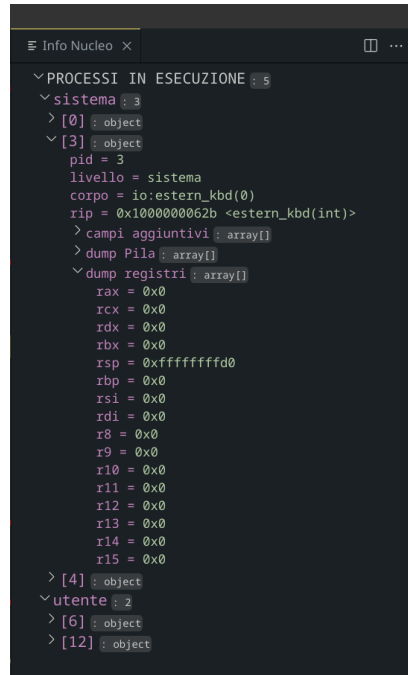


Figura 4.8: Informazioni aggiuntive sul nucleo

## Processi in esecuzione

Le informazioni relative ai processi vengono mostrate tramite una lista con elementi selezionabili. Ogni elemento della lista é formato da una rappresentazione tramite **chiave:valore** e un eventuale campo adiacente con eventuali informazioni sulla tipologia della variabile o oggetto. I processi sono stati divisi per convenienza tra processi di sistema e processi utente.

## Capitolo 5

# Conclusione e sviluppi futuri

L'utilizzo estensione rende semplice approcciarsi al debug e in particolare al debug del nucleo didattico. Tuttavia l'estensione non è completa, vi sono numerose funzionalità che possono essere implementate per ampliare le informazioni relative al nucleo come informazioni riguardanti la memoria virtuale o stato dell'APIC.



## Capitolo 6

# Sviluppo e contributi

In questo capitolo spiego come é possibile aggiungere nuove funzionalità all'estensione. È possibile sviluppare l'estensione tramite la macchina virtuale fornita durante il corso, tuttavia si consiglia vivamente di installare il nucleo sulla propria macchina seguendo le istruzioni fornite dal Professor Lettieri[Lettieri(2024a)], come editor é molto indicato VSCode stesso poiché é configurato per debug e esecuzione delle estensioni.

### Installazione

Per prima cosa bisogna impostare l'ambiente di sviluppo per l'estensione:

- scaricare la repository del progetto **Nucleo-Debugger**
- navigare all'interno della repository
- creare la propria branch di sviluppo della feature da implementare tramite **git**
- installare le dipendenze: **yarn** e **node**
- inizializzare l'ambiente con il comando **yarn install** e una volta terminato lanciare il comando **yarn global add vsce** per installare il pacchetto di VSCode per creare il file di installazione dell'estensione
- aprire VSCode con il comando **code .**

Se eventualmente l'estensione per il debug del nucleo dovesse essere presente tra le estensioni di VSCode bisogna rimuoverla.

### Debug e esecuzione

È possibile caricare l'estensione in un environment separato da quello di sviluppo tramite proprio il debugger di VSCode premendo il tasto **F5**. Una volta



completata la compilazione verrà avviata una nuova finestra di VSCode dove apriamo la cartella di un nucleo, in questo caso possiamo usare `prova-test` e avviare a sua volta il debug tramite `F5`.

## 6.1 Aggiunta di comandi

Per aggiungere comandi bisogna modificare il file `NucleoInfo.ts` nelle sezioni indicate:

- se vi é la necessità di ricevere dati da GDB bisogna dichiarare una variabile di supporto.

---

```
1 ...
2   public process_list: any | undefined;
3   // declare your new GDB response variable
4   // public command_VAR: any | undefined;
5 ...
```

---

Figura 6.1: Variabile di supporto per i dati

- Per eseguire un comando é necessario chiamare la funzione `.customCommand(args ...)` e se si devono ricevere dati assegnarli alla relativa variabile di appoggio.

---

```
1 const updateInfo = async () => {
2   this.process_list = await this.customCommand(session, "process list");
3
4   // Insert your command
5   // this.VAR = this.customCommand(session, "COMMAND");
6
7   const infoPanel = this._panel.webview;
8
9
10  infoPanel.html = this._getHtmlForWebview();
11 };
```

---

Figura 6.2: Richiesta di esecuzione del comando

- Per aggiornare la webview bisogna aggiungere all'HTML già presente il proprio tramite una funzione di appoggio che si occupa di formattare i dati (es. `this.YOUR_FORMATTER()`)

---

```

1 private _getHtmlForWebview() {
2   ...
3
4   ...
5   let sourceDocument = `
6     <!DOCTYPE html>
7     <html lang="en">
8       <head>
9         <meta charset="UTF-8">
10        <meta name="viewport" content="width=device-width,
            initial-scale=1.0">
11        <link href="${stylesResetUri}" rel="stylesheet">
12        <link href="${stylesMainUri}" rel="stylesheet">
13        <link href="${codiconsUri}" rel="stylesheet" />
14        <title>Info Nucleo</title>
15      </head>
16      <body>
17        {{{processList}}}
18
19        //{{varNAME}}
20      ...
21
22      ...
23      let template = Handlebars.compile(sourceDocument);
24      return template({ processList: this.formatProcessList(), varNAME:
        this.YOUR_FORMATTER() });
25  }

```

---

Figura 6.3: Costruzione della pagina HTML

- il formattatore deve creare il codice HTML da incorporare nella funzione `_getHtmlForWebview()` per mostrare correttamente i dati

---

```

1 private YOUR_FORMATTER() {
2   ...
3   document = `
4
5     // Creazione del codice HTML contenente i dati
6
7     `
8   ...
9   let template = Handlebars.compile(document);
10  return template({var: parsed_command_VAR});
11 }

```

---

Figura 6.4: Implementazione di un formattatore

come spiegato in seguito, le strutture per lo scambio di informazioni sono a discrezione del programmatore, tuttavia si consiglia l'utilizzo del formato **JSON** per facilitare la visualizzazione tramite il tool di templating `handlebars` [Katz(2024)].

### 6.1.1 `nucleo_vscode.py`

All'interno del file `nucleo_vscode.py` bisogna implementare il nuovo comando richiesto dall'estensione. Per l'effettiva implementazione del comando dipende dalla funzione che si vuole aggiungere, si faccia riferimento alla guida per lo scripting di GDB [Free Software Foundation(2024)] e al file `./debug/nucleo.py` all'interno di una delle versioni del nucleo presenti sul sito del Professor Lettieri [Lettieri(2024b)].

Capitolo 7

Ringraziamenti



# Bibliografia

- [Free Software Foundation(2024)] Inc Free Software Foundation. Debugging with gdb: the gnu source-level debugger, 2024. URL <https://sourceware.org/gdb/current/onlinedocs/gdb.html/Python-API.html>.
- [Katz(2024)] Yehuda Katz. Handlebars: Minimal templating on steroids, 2024. URL <https://handlebarsjs.com/>.
- [Lettieri(2024a)] Giuseppe Lettieri. Calcolatori elettronici, 2024a. URL <https://calcolatori.iet.unipi.it/>.
- [Lettieri(2024b)] Giuseppe Lettieri. Calcolatori elettronici, 2024b. URL <https://calcolatori.iet.unipi.it/appelli.php>.
- [Microsoft(2024a)] Microsoft. What is the debug adapter protocol?, 2024a. URL <https://microsoft.github.io/debug-adapter-protocol/overview>.
- [Microsoft(2024b)] Microsoft. What is the debug adapter protocol?, 2024b. URL <https://microsoft.github.io/debug-adapter-protocol/specification>.
- [Microsoft(2024c)] Microsoft. Integrate with external tools via tasks, 2024c. URL <https://code.visualstudio.com/Docs/editor/tasks>.
- [mignone(2024)] Francesco mignone. Vscode nucleo debugger, 2024. URL <https://github.com/ilnerdchuck/VSCoDe-Nucleo-Debugger/blob/main/nucleo-debugger/src/nucleoweb.ts>.