Università degli Studi dell'Insubria
Dipartimento di Scienze teoriche e applicate

Corso di laurea in Informatica

Una nuova primitiva di creazione dei processi per sistemi Unix

Relatore
Prof. Simone Tini

Laureando
Andrea Pappacoda
Matricola 749726

Anno accademico 2023/2024

Sommario

Nei sistemi Unix la creazione dei processi è stata storicamente effettuata attraverso la famiglia di funzioni `fork()` ed `exec()`. In questa tesi verrà introdotta e analizzata la funzione `spawn()`, una nuova primitiva fondamentalmente diversa che, con un design più semplice, mira a risolvere le problematiche legate alla natura di `fork()`.

Indice

1 Perché `spawn()`

Prima di poter parlare del motivo per cui `spawn()` sia una chiamata di sistema desiderabile, è necessario capire quali problematiche la classica funzione `fork()` comporti, e come diverse parti dell'ecosistema di sistemi Unix stiano gradualmente considerando l'adozione dell'interfaccia qui discussa.

1.1 Problematiche di `fork()`

La sua più grande forza è anche la sua più grande debolezza: la capacità di `fork` di duplicare lo spazio di indirizzamento di un processo è una semplice soluzione al problema di personalizzare la creazione di un processo figlio, ma che comporta numerosi problemi implementativi:

- La maggior parte degli usi di `fork` è quello di lanciare immediatamente un nuovo programma con la funzione `exec`. Concettualmente, dover duplicare il processo corrente invece che eseguire direttamente un processo nuovo risulta confusionario, soprattutto ai principianti [65].
- L'utilizzo di `fork` in combinazione con altri sistemi di parallelismo può causare problemi, spesso sottili e difficili da rilevare. In particolare, bisogna prestare particolare attenzione ad utilizzare `fork` in combinazione con i thread e i segnali Unix [32]. Queste difficoltà, ad esempio, hanno portato Python 3.12 a deprecare l'uso di `fork` in ambienti multi-threaded [19], con l'intenzione di cambiare la primitiva predefinita di parallelismo basato su processi da `fork` a `spawn` in Python 3.14 [18].
- Per implementare `fork` in maniera efficiente, l'intero sistema operativo dev'essere architettato intorno ad essa. Come descritto in «A fork() in the road» [5], per ottenere una *personalità* POSIX soddisfacente, il sistema operativo sperimentale K42 ha dovuto subire una riprogettazione totale che ha fatto convergere il design del sistema a un design Unix-like, a scapito delle altre personalità. Altri sistemi più comuni, come Cygwin [4] e Hurd [24], sono vittime di questo problema.
- Anche sfruttando memoria Copy-on-Write (CoW), le prestazioni possono essere non eccellenti in casi di `fork` di processi molto grandi, come, ad esempio, web browser o programmi che fanno uso di cache propria [15].

È quindi chiaro come `fork`, nonostante abbia avuto un incredibile successo — soprattutto considerato che, come affermato da Ritchie stesso in *The Evolution of the Unix Time-sharing System* [63], sia stata implementata in un paio di giorni e con una manciata di linee di assembly — presenti delle criticità non indifferenti, e che c'è bisogno di una primitiva alternativa.

1.2 Interesse in `spawn()`

L'alternativa più intuitiva ai problemi di `fork` è quella di creare una singola primitiva che lanci direttamente un nuovo processo, senza duplicare quello corrente. In effetti, non è una idea rivoluzionaria. Lo pensò Ritchie nel 1996 per il sistema Unix [63], ma non venne fatto per mancanza della primitiva `exec`, quella che mette in esecuzione un nuovo programma. Altri sistemi operativi, invece, forniscono esclusivamente una primitiva unificata, tra cui Windows con `CreateProcess` [10].

Anche all'interno del mondo Unix, però, l'interesse verso `spawn` esiste, e negli ultimi tempi si è dimostrato più vivo che mai.

Lo standard POSIX [29], già nel dicembre 1999, standardizzò la funzione `posix_spawn()` attraverso la rettifica POSIX.1d-1999 [28]. La funzione era pensata per permettere a piccoli sistemi *embedded* senza una MMU — quindi impossibilitati ad implementare `fork` efficientemente — di soddisfare i requisiti dello standard ed essere certificati *POSIX-compliant*. Oggi, però, il design di `posix_spawn` è apprezzato per la sua capacità di risolvere i problemi tecnici di `fork`.

Un esempio del rinnovato interesse in `spawn` è osservabile nel mondo Linux. Nel 2022, lo sviluppatore Josh Triplett propose una nuova system call chiamata `io_uring_spawn` [16]. Come indicato dal nome, la system call unisce il sottosistema `io_uring` [9] al modello di creazione di processi di una funzione `spawn`. Tra le ragioni citate da Triplett per l'introduzione di tale interfaccia, ci sono miglioramenti prestazionali e riduzione di complessità, specialmente in ambienti multi-threaded. La system call è di basso livello, ed è pensata per essere utilizzata come primitiva di creazione dei processi all'interno della già citata `posix_spawn` al posto di `clone` [14]. Purtroppo, però, il codice di Triplett non è mai stato integrato nel kernel Linux.

Un secondo esempio, sempre originatosi nel mondo Linux, è quello della funzione di libreria `pidfd_spawn` [8], aggiunta alla libreria C GNU nella versione 2.39, rilasciata il 31 gen. 2024 [27]. A differenza di `io_uring_spawn`, questa non è una nuova primitiva di creazione di processi implementata a livello kernel, ma un'alternativa all'interfaccia *user-space* `posix_spawn` che fa uso dei *PID file descriptors* di Linux, i quali permettono di evitare alcune corse critiche che possono verificarsi in seguito all'utilizzo di numeri interi globali come identificatori per i processi (PID). L'utente principale di questa nuova funzione è systemd [68], il gestore del sistema e dei servizi Linux. systemd, inoltre, ha abbandonato l'uso di `fork` per l'avvio di servizi a partire dalla versione 255 [69], rilasciata a fine 2023, optando per l'uso di `posix_spawn`, così da rendere più affidabile l'avvio di servizi in situazioni *near-OOM*; l'architettura è mostrata in figura 1.

Anziché eseguire direttamente il programma desiderato, systemd lancia un programma intermedio, `systemd-executor`, il quale riceve da `PID1` le operazioni da applicare all'ambiente di esecuzione del figlio prima di chiamare

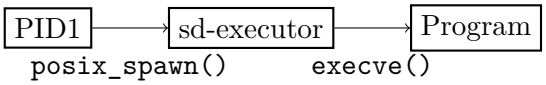

Figura 1: Modello di spawn di systemd

`exec` e lanciare effettivamente il processo desiderato. La motivazione per questo esecutore intermedio è che `spawn` non offre tutta la flessibilità di `fork`: se tra `fork` ed `exec` è possibile modificare l'ambiente di esecuzione del figlio utilizzando un numero arbitrario di funzioni (limitandosi a quelle definite in *signal-safety* [32] nel caso di programmi multi-threaded), `spawn` è al contrario limitata alle opzioni che questa può ricevere come argomenti. Nonostante il prototipo di `posix_spawn` sia piuttosto esaustivo, questo non è sufficiente per i bisogni fuori dal comune di systemd, ragion per cui `systemd-executor` implementa quello che può essere chiamato un Domain-Specific Language per l'impostazione dell'ambiente di programmi lanciati con `spawn`. Questo dimostra che, con un poco di ingegno, `spawn` può essere utilizzata anche dove la flessibilità di `fork` sembra impareggiabile.

Spostandosi all'infuori di Linux ma rimanendo nel contesto di Unix, è interessante notare come in Solaris [40], il sistema proprietario di Oracle, sia stata implementata con successo una system call `spawn` [13] nella sua ultima release. Come intuibile da quanto da quanto detto in precedenza, la nuova primitiva ha migliorato per il sistema operativo le performance, specialmente in contesti multi-threaded. Questo perché, prima dell'introduzione di `spawn`, la funzione di libreria `posix_spawn` di Solaris era implementata usando `vfork`. `vfork`, a differenza di `fork`, crea un secondo processo che condivide la memoria del padre — non avvengono quindi copie di memoria, e la `vfork` di processi con uno spazio di indirizzamento ampio rimane performante. Il rovescio della medaglia è però che, ovviamente, due processi diversi non possono essere eseguiti simultaneamente senza sincronizzazione all'interno dello stesso address space, e il processo padre è quindi messo in attesa fino all'esecuzione di `exec` da parte del figlio, o fino alla sua terminazione. La `posix_spawn` di Solaris (e di Linux!), quindi, aveva prestazioni specialmente poco ottimali in processi facenti un pesante uso di thread. Come se non bastasse, se il processo figlio va a modificare, anche minimamente, lo spazio di memoria in condivisione col padre, si va in una situazione di *undefined behaviour*, la quale può portare alla terminazione anomala del processo padre, o peggio, a sottili bug che possono poi essere sfruttati da utenti malintenzionati per creare falle di sicurezza [17].

È quindi chiaro che, neanche con *vfork*, è possibile implementare un'interfaccia di creazione di processi efficiente. Non esiste alternativa all'aggiunta di una nuova primitiva `spawn` al kernel del sistema operativo.

2 Perché OpenBSD

Il sistema operativo sul quale andare a implementare l'interfaccia `spawn()` è stato scelto in base a una serie di criteri, alcuni più stringenti di altri.

Un lavoro del genere ha bisogno in primis di essere eseguito su un sistema operativo *Open Source* [73], ovvero che permetta la libera consultazione e modifica del suo codice sorgente. Questo elimina alcuni sistemi Unix proprietari come macOS e Solaris [40] — quest'ultimo, come descritto nella sezione 1, è tra l'altro l'unico sistema Unix a possedere una system call analoga a quella descritta nella presente tesi.

Il secondo criterio è legato alla complessità del kernel da modificare. Un kernel più piccolo è intrinsecamente più abbordabile rispetto a un kernel ampio e in continuo sviluppo, caratteristica che si presta di più alla sperimentazione. Di conseguenza, Linux è escluso, essendo il kernel probabilmente più ampio, pareggiato solo dal kernel Windows NT. Un altro aspetto tecnico che porta all'esclusione di Linux è che la sua primitiva di creazione dei processi, `clone()` [14], ha una interfaccia più intricata che permette di modificare la fase di creazione del figlio in maniera granulare, ma pur sempre clonando il processo padre; prendere `clone()` come riferimento avrebbe quindi introdotto complessità aggiuntiva.

Il terzo requisito riguarda la popolarità e la stabilità del sistema operativo. Modificare un kernel usato in produzione permette di misurare e analizzare l'impatto in carichi di lavoro reali, fattore cruciale per `spawn()` in quanto uno dei suoi vantaggi principali sono i guadagni prestazionali. Sistemi operativi come GNU Hurd [23], Ares [70, 12], Redox [62] e altri sistemi operativi sperimentali [75, 66] sono quindi esclusi.

Analizzati i requisiti, rimangono soltanto alcuni sistemi operativi, tutti della famiglia BSD: OpenBSD [74], FreeBSD [71] e NetBSD [72]. Tra questi, OpenBSD è stato scelto per diversi motivi:

- Gli sviluppatori pongono enfasi sulla coerenza del sistema e sulla qualità della sua documentazione, il che rende la fase di studio del sistema più semplice.
- Lo sviluppo di numerose funzioni e componenti del sistema operativo è spesso *paper-driven* [39], e una tesi documentante le specifiche di una nuova system call `spawn()` sarebbe quindi ben accetta.
- Tra i vari sistemi operativi convenzionali, OpenBSD è l'unico che accetta *breaking changes* [61] al fine di minimizzare la dimensione del sistema operativo, così da eliminare complessità e ridurre la superficie d'attacco. Questo minimalismo ha il comodo effetto di rendere il kernel relativamente più semplice da comprendere.

3 Ambiente di sviluppo

Una volta scelto il sistema operativo sul quale lavorare, è necessario creare al suo interno un ambiente di sviluppo facilmente riproducibile, così che qualunque lettore della presente tesi, autore compreso, possa in un qualsiasi momento ricreare lo stesso ambiente di sviluppo e riprendere il lavoro così come condotto dal sottoscritto.

Nonostante OpenBSD sia uno dei sistemi operativi Open Source più utilizzati al mondo, i suoi sviluppatori adottano un sistema di sviluppo "vintage"; non sono intenzionati a cambiare ciò che funziona per loro, a meno di vantaggi significativi. Ad esempio, il Version Control System utilizzato per la gestione dell'intero codice sorgente di kernel, shell e port, è Concurrent Versions System (CVS) [11], nonostante sia un VCS considerato ormai obsoleto e soppiantato da Git [22]. Non è quindi una sorpresa il fatto che l'ambiente di sviluppo scelto per lo svolgimento di questo progetto sia anch'esso, per certi versi, "vintage" e minimalista, così da allinearsi agli sviluppatori veri e propri. Un *setup* minimale è anche una necessità: il campo di applicazione principale di OpenBSD è quello dei server, quindi il supporto ad applicazioni "moderne" come editor di codice o IDE grafici è piuttosto limitato.

Vediamo quindi le componenti dell'ambiente.

3.1 OpenBSD 7.5

Il primo passaggio necessario è quello di installare il sistema operativo OpenBSD. L'installazione dell'intero sistema operativo è richiesta anche per la sola compilazione del suo kernel — questo perché il suo *build system* consiste in *Makefile* facenti uso di estensioni specifiche del programma `make` di OpenBSD [49].

L'installazione di OpenBSD non è particolarmente difficoltosa se si ha un minimo di dimestichezza nel campo dell'amministrazione di sistema, specialmente se si ha già installato un sistema Linux o BSD in passato. La procedura di installazione è spiegata nel dettaglio in *OpenBSD FAQ - Installation Guide* [51] e in *INSTALL.amd64* [30]. La cosa a cui prestare più attenzione è che si utilizzi il file con estensione `.img` e non `.iso`, dato che quest'ultima ha problemi ad essere eseguita in alcuni virtualizzatori.

È consigliata, ma non richiesta, l'installazione dell'ambiente grafico X11 [20] abilitando `xenodm` [60].

Una volta installato OpenBSD, saranno già disponibili alcuni strumenti di sviluppo come gli editor Vi [46] e mg [50]. È indubbiamente possibile usare uno qualunque dei due per lavorare sul codice C del kernel, ma personalmente ho optato per un editor leggermente più avanzato che potesse offrire funzionalità utili come *syntax highlighting* e suggerimenti durante la stesura del codice: Vim [38].

3.2 Vim 9.1

3.2.1 Installazione

L'editor Vim è uno degli editor di più diffusi al mondo, se non il più diffuso. Basti pensare che è pre-installato su ogni principale distribuzione Linux e su macOS [2]. Essendo estremamente *portable* è possible installarlo con semplicità su OpenBSD attraverso il suo sistema di gestione dei pacchetti software, documentato nella pagina di manuale *packages* [52].

I pacchetti software installabili su OpenBSD derivano dal più complesso sistema dei *ports* [54], il quale contiene sia il codice sorgente di tutto il software di terze parti installabile che i *Makefile* da utilizzare per la loro compilazione e installazione. Sebbene sia possibile scaricare e compilare manualmente il codice degli applicativi desiderati, il metodo consigliato per aggiungere programmi di terzi al sistema è usare il comando `pkg_add` [53]; ad esempio, per installare Vim, è sufficiente il comando mostrato nel listing 1.

Il nome del pacchetto da installare, `vim`, è seguito da `--no_x11` per specificare che, tra le varie versioni di Vim disponibili, quella da installare è la variante senza integrazione col sistema grafico X11; questo perché per lavorare sul codice sorgente del kernel è necessario essere l'utente `root`, e i processi in esecuzione come utente `root` hanno spesso problemi a collegarsi a X11 [3].

```
# pkg_add vim--no_x11
```

Listing 1: Installazione di Vim con `pkg_add`

3.2.2 Configurazione

Una volta installato, è consigliabile procedere con la sua configurazione. In questo modo, sarà possibile ottenere nell'editor alcune delle funzionalità intelligenti degli IDE, come suggerimenti, errori in linea e navigazione avanzata.

Il modo più conveniente per beneficiare di queste funzionalità è fare affidamento al Language Server Protocol (LSP) [35]. L'obiettivo di LSP è quello di separare dall'editor di codice la conoscenza di un particolare linguaggio di programmazione — gli autori di un linguaggio sviluppano un *server* che è in grado di comprenderne la semantica e fornire documentazione e funzionalità di navigazione, mentre gli autori di un editor di testo implementano un *client* all'interno di esso in grado di parlare a un server qualunque attraverso un protocollo omogeneo e standardizzato indipendente dal linguaggio in uso.

L'implementazione lato server più completa di LSP per il linguaggio C è *clangd* [43], ed è anch'essa installabile dai pacchetti di OpenBSD, come mostrato nel listing 2.

```
# pkg_add clang-tools-extra
```

Listing 2: Installazione di clangd con `pkg_add`

clangd non necessita di alcuna configurazione, al contrario di Vim. L'editor, infatti, non implementa alcun client LSP. La soluzione è ricorrere al sistema di *plugin* di Vim, installando il plugin "lsp" per Vim 9 [33]. La procedura è semplice, ed è descritta nel listing 3.

```
$ mkdir -p ~/.vim/pack/downloads/opt
$ cd ~/.vim/pack/downloads/opt
$ ftp -o - \
    https://github.com/yegappan/lsp/archive/refs/heads/main.tar.gz \
    | pax -rz
$ mv lsp-main lsp
```

Listing 3: Download di "lsp" nella cartella dei pacchetti di Vim

L'ultimo passaggio consiste in istruire Vim all'uso effettivo del plugin "lsp". Ciò avviene attraverso un file di configurazione chiamato `vimrc`, da creare all'interno della cartella `~/.vim` creata in precedenza. Il file mostrato nel listing 4 include la configurazione di default di Vim, così da non sovrascriverla, per poi aggiungere alla variabile `lspServers` il server clangd, se presente. Durante la fase di setup del plugin, questa lista viene passata a "lsp" con la funzione `LspAddServer`, così che il plugin sappia con quale server comunicare per fornire funzionalità intelligenti per il linguaggio C. Infine, alcuni *key bindings* di Vim vengono ri-mappati così che alcuni comandi standard come `gd` [37] e `CTRL-]` [36] facciano affidamento al server LSP per la navigazione e i salti nel codice.

3.3 Bear

Al fine di dare suggerimenti e offirre funzionalità di navigazione all'interno della *codebase* del kernel, clangd deve conoscerne la struttura. Normalmente questo avviene attraverso dei file parte del build system — ad esempio, nel mondo Java, la struttura di un progetto può essere espressa attraverso file Maven [34].

Quando si parla di un kernel, però, la situazione è molto più complessa. La loro compilazione fa uso, nella quasi totalità dei casi, di sistemi di compilazione *ad-hoc*, sviluppati nel tempo in base alle esigenze del sistema operativo stesso. Non esiste quindi editor di codice o IDE al mondo che,

```
vim9script

source $VIMRUNTIME/defaults.vim

packadd! lsp

final lspServers: list<dict<any>> = []

if executable('clangd')
    lspServers->add({
        filetype: ['c', 'cpp'],
        path: 'clangd',
        args: ['--background-index']
    })
endif

autocmd User LspSetup g:LspAddServer(lspServers)

au User LspAttached nnoremap <buffer> <C-]> <Cmd>LspGotoDefinition<CR>
au User LspAttached nnoremap <buffer> gd <Cmd>LspGotoDeclaration<CR>
au User LspAttached nnoremap <buffer> K <Cmd>LspHover<CR>
au User LspAttached nnoremap <buffer> <F2> <Cmd>LspRename<CR>
```

Listing 4: Configurazione del plugin “lsp”

preso il kernel di OpenBSD come spazio di lavoro, sia in grado di fornire funzionalità intelligenti.

L’unica soluzione per capire come effettivamente il progetto sia strutturato, e come i vari file sorgente interagiscano tra di loro, è quello di “sbirciare” il processo di compilazione per estrapolarne una rappresentazione che risulti più comoda da gestire per strumenti come clangd.

Bear [6] è un software che fa esattamente questo. È in grado di interporsi tra il sistema di compilazione e le chiamate al compilatore, estrarre i comandi di compilazione rilevanti e salvarli all’interno di un *JSON Compilation Database* [31], indipendente dal sistema di compilazione, il quale verrà poi letto da clangd.

Purtroppo Bear non è disponibile nel sistema dei *packages*, ed è quindi necessaria la sua compilazione da sorgente. Per quanto teoricamente questo sia un compito semplice da eseguire, su OpenBSD la sua compilazione è estremamente complessa a causa della mancanza di una versione di gRPC [26] facilmente installabile come pacchetto, in quanto dipendenza di Bear. Compilare gRPC su OpenBSD è specialmente difficile perché gli sviluppatori di tale software non supportano ufficialmente il sistema operativo, e la

compilazione non è quindi possibile senza una lunga configurazione delle opzioni del suo sistema di compilazione e senza l'applicazione di *patch* al suo codice sorgente. Per evitare di prolungare in maniera eccessiva questo paragrafo con istruzioni dettagliate, è stata preparata una versione pre-compilata di Bear scaricabile all'indirizzo `https://andrea.pappacoda.it/bear_openbsd75_amd64.tar.gz`, installabile con i comandi mostrati nel listing 5.

```
$ cd /tmp
$ ftp -o - \
    https://andrea.pappacoda.it/bear_openbsd75_amd64.tar.gz \
    | pax -rz
# pkg_add abseil-cpp fmt protobuf
$ cd bear_openbsd75_amd64
# make install
```

Listing 5: Installazione della versione pre-compilata di Bear

3.4 Ottenimento del codice sorgente

Il processo di compilazione di OpenBSD è standardizzato e relativamente semplice. Tutto avviene all'interno della cartella `/usr/src`, creata in fase di installazione del sistema operativo. La procedura è documentata nella pagina di manuale *release* [56], ma verrà riprodotta qui per completezza.

Prima di iniziare, se intendiamo eseguire la procedura con un utente diverso dall'utente `root`, è necessario aggiungere l'utente in uso al gruppo `wsrc`. Questo è possibile usando il comando *user* [58], seguito da un riavvio del sistema, come mostrato nel listing 6.

```
# user mod -G wsrc exampleuser
# shutdown -r now
```

Listing 6: Aggiunta di *exampleuser* al gruppo `wsrc`

Il primo passo è quello di scaricare il codice sorgente, sia di kernel che user space; questo perché, quando viene aggiunta una system call, è anche necessario aggiungere un *wrapper* per essa all'interno della libreria C, così che questa possa essere chiamata — in OpenBSD, a partire dalla release 7.5, non è possibile invocare system call se non mediante la libreria C [64, 1]. È possibile ottenere gli archivi contenenti tale codice eseguendo i comandi illustrati nel listing 7.

Una volta scaricati gli archivi, questi vanno estratti nella cartella apposita `/usr/src`, come mostrato nel listing 8.

```
$ cd /tmp
$ ftp https://cdn.openbsd.org/pub/OpenBSD/7.5/src.tar.gz \
    https://cdn.openbsd.org/pub/OpenBSD/7.5/sys.tar.gz
```

Listing 7: Download del codice sorgente di OpenBSD

```
$ cd /usr/src
$ pax -rzf /tmp/src.tar.gz
$ pax -rzf /tmp/sys.tar.gz
```

Listing 8: Estrazione del codice sorgente di OpenBSD

3.5 Creazione del compilation database

Dopo l'estrazione del codice, è necessario effettuare una compilazione completa del kernel usando Bear, così da rendere possibile a clangd l'analisi del codice sorgente. Una volta eseguiti i comandi nel listing 9, verrà creato un file chiamato `compile_commands.json`. Le prime due invocazioni di `make` creeranno, rispettivamente, la cartella di compilazione e il file di configurazione del kernel, mentre la terza invocazione, effettuata da `bear`, ne effettuerà la compilazione totale. Non sarà necessario generare nuovamente il file ad ogni compilazione, bensì verrà utilizzato senza modifiche per tutte le attività di sviluppo.

```
$ cd /sys/arch/$(machine)/compile/GENERIC.MP
# make obj
# make config
# bear --force-wrapper -- make
```

Listing 9: Compilazione del kernel usando Bear

La cartella contenente il sorgente del kernel è la sotto-cartella `sys` di `/usr/src`, e serve quindi copiare il JSON Compilation Database in essa. Il listing 10 mostra come fare.

Terminate queste operazioni, la composizione dell'ambiente di sviluppo è completata, ed è possibile passare alla fase di sviluppo.

4 Operazioni preliminari

Prima di poter scrivere una system call complessa come `spawn`, bisogna in primo luogo capire come creare una chiamata al kernel. In questa sezione, verrà illustrato come creare una system call chiamata `spawn` la quale, dato un numero intero passato da parte dell'utente, stampa quest'ultimo nel log di sistema (*syslog* [57]).

```
# cp \
    /sys/arch/$(machine)/compile/GENERIC.MP/compiles_commands.json \
    /usr/src/sys/
```

Listing 10: Salvataggio del compilation database in `/usr/src/sys`

4.1 Implementazione di una system call

In OpenBSD esistono diverse convenzioni relative alle system call. Generalmente, ogni chiamata vive nel suo file sorgente C, contenuto nella sotto-cartella `kern` di `sys` (da intendersi relativa alla gerarchia `/usr/src`), e segue una nomenclatura del tipo *kern_nomesyscall.c*.

All'interno di esso, risiede la procedura che implementa la system call, la quale è comunemente chiamata *sys_nomesyscall*.

Proseguiamo quindi con la creazione di `sys_spawn` all'interno di un nuovo file `kern_spawn.c`. Intuitivamente, verrebbe da creare una funzione avente un prototipo dichiarante un singolo intero, la quale chiama poi la funzione *log* [48] per passarlo al sistema di logging generale, come mostrato nel listing 11.

```
#include <sys/syslog.h>

int
sys_spawn(int num)
{
    log(LOG_WARNING, "spawn: ho ricevuto il numero %d\n", num);
    return 0;
}
```

Listing 11: Approccio naïf alla creazione di una system call

Purtroppo, però, questo non è possibile. Tutte le system call hanno infatti un prototipo omogeneo, e la definizione dei parametri da esse letti è situata in un file separato, `kern/syscalls.master` [67]. Il file `syscalls.master` agisce come "registro" contenente ogni system call fornita dal kernel; di ognuna, tiene traccia di nome, prototipo e caratteristiche. La procedura usuale per registrare una nuova chiamata nel kernel è la sostituzione di una system call contrassegnata come *OBSOL* (obsoleta) con la definizione della nuova funzione. In OpenBSD 7.5 la syscall `oquota` è definita come obsoleta, e può quindi essere rimpiazzata con `sys_spawn`. Il listing 12 mostra esattamente le modifiche apportate al file.

Una volta registrata la system call in `syscalls.master`, è necessario apportare modifiche alla sua definizione in `kern_spawn.c`. In OpenBSD, tutte le system call hanno un prototipo omogeneo, e l'accesso ai parametri

```
diff --git a/sys/kern/syscalls.master b/sys/kern/syscalls.master
index 0400b8ea8d4..5b46727a2aa 100644
--- a/sys/kern/syscalls.master
+++ b/sys/kern/syscalls.master
@@ -286,7 +286,7 @@
 147    STD             { int sys_setsid(void); }
 148    STD             { int sys_quotactl(const char *path, int cmd, \
                            int uid, char *arg); }
-149    OBSOL           oquota
+149    STD             { int sys_spawn(int num); }
 150    STD NOLOCK      { int sys_ypconnect(int type); }

 ; Syscalls 151-180 inclusive are reserved for vendor-specific
```

Listing 12: Rimpiazzare `oquota` con `sys_spawn`

come registrati avviene attraverso l'uso di una macro chiamata *SCARG*. È inoltre richiesta la scrittura dei parametri come registrati all'inizio della definizione della system call. È più immediato comprendere il procedimento con un esempio, mostrato nel listing 13.

```c
#include <sys/proc.h>
#include <sys/syscallargs.h>
#include <sys/syslog.h>
#include <sys/systm.h>
#include <sys/types.h>

int
sys_spawn(struct proc* p, void* v, register_t* retval)
{
    struct sys_spawn_args /* {
        syscallarg(int)    num;
    } */ *uap = v;
    int num = SCARG(uap, num);
    log(LOG_WARNING, "spawn: ho ricevuto il numero %d\n", num);
    return 0;
}
```

Listing 13: Una semplice system call funzionante

Nel listing 13 rispetto al listing 11 sono stati introdotti numerosi costrutti che vale la pena descrivere. Per prima cosa, il prototipo di `sys_spawn` presenta tre parametri, `p`, `v` e `retval`:

- Il primo parametro della funzione contiene le informazioni relative

al processo *user-space* che ha invocato la chiamata a sistema. La definizione di `struct proc` è nell'header `proc.c`.

- `v` è un puntatore gestito dal kernel utilizzabile per accedere ai dati passati da *user-space*. Per convenzione, non è mai usato direttamente, bensì utilizzato per la definizione di un puntatore `uap` del tipo `struct sys_spawn_args`, definito dal sistema di compilazione in base a quanto dichiarato in `syscalls.master` e scritto nell'header `syscallargs.h`.

- Infine, `retval` è utilizzato per restituire a *user-space* un valore numerico. Questo parametro "di uscita" è necessario in quanto il valore proprio di ritorno della funzione viene utilizzato per segnalare errori durante l'esecuzione della system call, ponendolo nella variabile `errno`, familiare a ogni programmatore C. Il tipo `register_t` è definito in `types.h`.

Grazie a questo prototipo, il parametro `num` che intendiamo leggere dal chiamante viene estratto dal puntatore `uap` attraverso la macro `SCARG`, definita in `systm.h`. Il resto della funzione è invariato.

Per ultimare la definizione della system call, bisogna rendere partecipe della sua esistenza il sistema di compilazione. Ciò è molto semplice: è sufficiente porre il nome del file utilizzato per la definizione di `sys_spawn` in `conf/files` [7], come mostrato nel listing 14.

```
diff --git a/sys/conf/files b/sys/conf/files
index fd76e9934e9..a6aac2c87ce 100644
--- a/sys/conf/files
+++ b/sys/conf/files
@@ -723,6 +723,7 @@ file kern/kern_intrmap.c            intrmap
 file kern/kern_sensors.c
 file kern/kern_sig.c
 file kern/kern_smr.c
+file kern/kern_spawn.c
 file kern/kern_subr.c
 file kern/kern_sysctl.c
 file kern/kern_synch.c
```

Listing 14: Dichiarazione di `kern_spawn.c` nel build system

Con questa ultima aggiunta, la creazione della system call è completata. Per renderla disponibile, è sufficiente ripetere i comandi del listing 9, ma rimuovendo l'invocazione di `bear` e aggiungendo il comando `make install` così da installare il nuovo kernel appena compilato.

4.2 Realizzazione di un wrapper per una system call nella libreria C

Come citato nella sezione 3.4, in OpenBSD non è possibile invocare una system call senza passare per la libreria C di sistema (libc). Di conseguenza, per chiamare la funzione `sys_spawn` appena definita, bisogna aggiungere un *wrapper* all'interno della libc, che deleghi l'invocazione della syscall.

La procedura, anche qui, è standardizzata all'interno del mondo OpenBSD. La sua libc, infatti, contiene una semplice infrastruttura in grado di creare automaticamente *stub* corrispondenti a una particolare chiamata a sistema; è sufficiente dichiarare il nome della system call in un paio di file.

Il primo file da modificare è `Symbols.list`, contenuto nella sotto-cartella `lib/libc` di `/usr/src`. In esso sarà necessario aggiungere il nome del simbolo relativo alla system call, come mostrato nel listing 15.

```
diff --git a/lib/libc/Symbols.list b/lib/libc/Symbols.list
index 251760f812e..784dd5b835b 100644
--- a/lib/libc/Symbols.list
+++ b/lib/libc/Symbols.list
@@ -221,6 +221,7 @@ _thread_sys_sigprocmask
 _thread_sys_sigsuspend
 _thread_sys_socket
 _thread_sys_socketpair
+_thread_sys_spawn
 _thread_sys_stat
 _thread_sys_statfs
 _thread_sys_swapctl
@@ -427,6 +428,7 @@ sigsuspend
 sigwait
 socket
 socketpair
+spawn
 stat
 statfs
 swapctl
```

Listing 15: Aggiunta di `sys_spawn` in `Symbols.list`

Successivamente, è necessario l'uso della macro *PROTO_NORMAL* per informare la libc del file all'interno del quale la funzione verrà dichiarata, e infine istruire il sistema di compilazione alla generazione effettiva dello *stub* modificando l'apposito *Makefile*, `sys/Makefile.inc`. Questi due passaggi sono illustrati, rispettivamente, nei listing 16 e 17. In questo caso, `spawn` verrà aggiunta nell'header `unistd.h` — la sua posizione non è importante,

ma è stato scelto questo file in quanto è dove risiedono anche le funzioni `fork` ed `exec`.

```
diff --git a/lib/libc/hidden/unistd.h b/lib/libc/hidden/unistd.h
index 74918f59675..d433f56e4cf 100644
--- a/lib/libc/hidden/unistd.h
+++ b/lib/libc/hidden/unistd.h
@@ -151,6 +151,7 @@ PROTO_NORMAL(setthrname);
 PROTO_NORMAL(setuid);
 PROTO_DEPRECATED(setusershell);
 /*PROTO_CANCEL(sleep);*/
+PROTO_NORMAL(spawn);
 PROTO_DEPRECATED(strtofflags);
 PROTO_DEPRECATED(swab);
 PROTO_NORMAL(swapctl);
```

Listing 16: Informare la libc che `spawn` è definito in `unistd.h`

```
diff --git a/lib/libc/sys/Makefile.inc b/lib/libc/sys/Makefile.inc
index e833a28924c..4ca86c0dda0 100644
--- a/lib/libc/sys/Makefile.inc
+++ b/lib/libc/sys/Makefile.inc
@@ -72,7 +72,7 @@ ASM=  __semctl.o __thrsigdivert.o \
     setreuid.o setrlimit.o setrtable.o setsid.o setsockopt.o \
     settimeofday.o setuid.o shmat.o shmctl.o shmdt.o \
     shmget.o shutdown.o sigaltstack.o socket.o \
-    socketpair.o stat.o statfs.o swapctl.o symlink.o symlinkat.o \
+    socketpair.o spawn.o stat.o statfs.o swapctl.o symlink.o symlinkat.o \
     sysarch.o sysctl.o \
     thrkill.o truncate.o \
     unlink.o unlinkat.o \
```

Listing 17: Generazione dello *stub* di `spawn`

Terminate le modifiche, è possibile ri-compilare e installare la nuova versione della libreria C, con l'usuale coppia di comandi `make` e `make install` (da eseguirsi all'interno della sotto-cartella `lib/libc`).

4.3 Uso della nuova system call

La chiamata `spawn` è ora presente all'interno del sistema: è implementata nel kernel ed è accessibile dalla libreria C. Per renderla accessibile anche ai programmi scritti dall'utente, però, è necessario aggiungere la sua dichiarazione in un header file che verrà poi incluso dall'utente. Come affermato

nella sezione 4.2, il file di intestazione scelto è `unistd.h`. Ci basta quindi modificare `/usr/include/unistd.h` e aggiungere il prototipo in questione. Questo è illustrato nel listing 18.

```
diff --git a/include/unistd.h b/include/unistd.h
index b87979ebbff..ae3f8d08455 100644
--- a/include/unistd.h
+++ b/include/unistd.h
@@ -341,6 +341,7 @@ int  execvp(const char *, char *const *);
 int     execvpe(const char *, char *const *, char *const *);
 #endif
 pid_t   fork(void);
+int     spawn(int);
 long    fpathconf(int, int);
 char   *getcwd(char *, size_t)
                __attribute__((__bounded__(__string__,1,2)));
```

Listing 18: Aggiunta del prototipo di `spawn` in `include/unistd.h`

In seguito, includendo l'header modificato in un programma C, si potrà accedere alla chiamata di sistema. Nel listing 19 è illustrato un programma C che fa uso della nuova `spawn`, e nel listing 20 viene mostrato l'esito dell'esecuzione di tale programma.

```c
#include <unistd.h>

int main(void) {
    spawn(42);
    return 0;
}
```

Listing 19: Programma C che fa uso della nuova syscall `spawn`

```
$ cc -o example example.c
$ ./example
$ grep 'spawn:' /var/log/messages
spawn: ho ricevuto il numero 42
```

Listing 20: Esecuzione del programma C illustrato nel listing 19

```
int
sys_fork(struct proc *p, void *v, register_t *retval)
{
    return fork1(p, FORK_FORK, child_return, NULL,
        retval, NULL);
}

int
sys_vfork(struct proc *p, void *v, register_t *retval)
{
    return fork1(p, FORK_VFORK|FORK_PPWAIT, child_return, NULL,
        retval, NULL);
}
```

Listing 21: Implementazione di `sys_fork` e `sys_vfork`

5 Sviluppo sul kernel

Per sviluppare la vera primitiva di creazione dei processi `spawn`, ho dovuto analizzare a studiare le porzioni di codice relative alle primitive di creazione di processi attuali, `fork` ed `exec`, oltre che a documentarmi e capire quale sarebbe stato il modo migliore di creare una nuova system call, come illustrato nelle sezioni precedenti.

In OpenBSD, le funzioni di libreria `fork` ed `exec` sono definite, rispettivamente, dalle system call `fork1` ed `execve`. Vediamo quindi come queste sono implementate.

5.1 Studio di `fork1()`

A dire il vero, `fork1` [47] non è una system call, ma è una funzione implementata a livello kernel usata per fornire le due system call `fork` e `vfork`. Detto ciò, osservata la definizione di `sys_fork` e `sys_vfork` presente nel listing 21, è ragionevole dire che la vera procedura implementante la creazione di processi del kernel è `fork1` — tutto viene delegato ad essa.

Non dovrebbe sorprendere il fatto che `fork1` sia molto complessa — essa gestisce infatti ogni operazione e procedura necessaria alla creazione del nuovo processo. Non ha quindi senso riportare nella presente tesi l'interezza del suo codice, ma mi limiterò a descriverne le parti più rilevanti ai fini della creazione della primitiva alternativa `spawn`.

La funzione esegue prima di tutto alcuni controlli di routine necessari a garantire il buon funzionamento del sistema. Ad esempio, si assicura che il processo chiamante non stia superando il limite di creazione di processi imposto globalmente.

Le parti più interessanti sono situate dopo questi controlli. Viene prima di tutto allocata l'area di memoria adibita alla memorizzazione del Process Control Block (PCB), con la funzione `uvm_uarea_alloc` [59], all'interno della quale vengono copiati alcuni dati appartenenti al PCB del padre.

Le divergenze tra `fork` e una ipotetica `spawn` non tardano ad arrivare. Il passaggio immediatamente successivo, infatti, è quello di allocare ed inizializzare un nuovo processo a partire dal processo padre. Altri dati più specifici del processo effettivamente in esecuzione, come il puntatore agli argomenti a riga di comando (`argv`) e alle variabili di ambiente (`env`), vengono associati al nuovo processo. Oltre a questi "metadati" associati al processo viene anche creato il vero e proprio *address space* (spazio di indirzzamento) del processo figlio prendendo come riferimento quello del padre, usando la funzione `uvmspace_fork` [59]. Entrambi questi passaggi non sono appropriati per un'operazione di `spawn`; un processo lanciato con essa, infatti, non deve condividere col padre né nome né argomenti passati alla sua esecuzione, tantomeno lo spazio di indirizzamento.

Il terzo e ultimo passaggio di rilevanza presente in `fork1` è definito dalla funzione `cpu_fork`. Questa procedura, definita in maniera diversa per ogni architettura hardware supportata da OpenBSD, si occupa di copiare ed aggiornare i restanti dati del PCB del padre nel figlio, e di metterlo in stato di *ready*.

Il lavoro di `fork1` è quindi essenzialmente terminato. Le operazioni restanti sono relative a raccogliere statistiche di sistema, notificare altri processi potenzialmente interessati dell'avvenuta creazione del figlio e ad integrarsi con il sottosistema di *tracing*.

5.2 Studio di `sys_execve()`

Proprio come `fork1`, nonostante l'idea di `exec` sia concettualmente semplice — mutare il programma in esecuzione — la sua implementazione è parecchio intricata. Al fine di semplificarne la descrizione, questa sezione esclude alcuni dei suoi dettagli.

La prima azione compiuta dalla system call è l'arresto dell'esecuzione dei thread del processo invocante la `exec`. Questo perché, nel suo normale caso d'uso, un processo potrebbe contenere diversi thread al momento dell'invocazione, e la loro esecuzione in contemporanea in una fase così delicata potrebbe causare corse critiche. I thread vengono solo fermati, e non distrutti, per permettere la gestione del caso in cui la syscall generi un errore prima del reset dello spazio di indirizzamento del processo. Nella sezione 5.3 vedremo perché questa problematica non si verifica in `spawn`.

Gran parte del codice successivo prepara quello che nei commenti del codice sorgente viene chiamato *exec package*. Questo "pacchetto" contiene diverse informazioni sul processo da lanciare, tra cui il suo nome, l'interprete da usare (ad esempio, `/bin/sh` in caso di script per la shell di sistema),

gli argomenti da passare al nuovo processo e una serie di *vmspace-building commands*, ovvero una serie di operazioni da eseguire su porzioni dello spazio di indirizzamento del nuovo processo necessarie per la sua creazione.

Questi comandi, i *vmcmds*, sono implementati attraverso l'uso di *callback*, espresse nel linguaggio C mediante puntatori a funzione. Nel listing 22 è riportata la struttura dati che li definisce.

```
/*
 * the exec_vmcmd struct defines a command description to be used
 * in creating the new process's vmspace.
 */
struct exec_vmcmd {
    int     (*ev_proc)(struct proc *p, struct exec_vmcmd *cmd);
                            /* procedure to run for region of vmspace */
    u_long  ev_len;         /* length of the segment to map */
    u_long  ev_addr;        /* address in the vmspace to place it at */
    struct  vnode *ev_vp;   /* vnode pointer for the file w/the data */
    u_long  ev_offset;      /* offset in the file for the data */
    u_int   ev_prot;        /* protections for segment */
    int     ev_flags;
};
```

Listing 22: Definizione della struttura `exec_vmcmd`

Se da un lato l'uso di una struttura dati *self-referencing*, ovvero facente riferimento a sé stessa attraverso l'uso di puntatori, sia indubbiamente una soluzione raffinata ed elegante, dall'altro rischia di introdurre parecchi grattacapo in situazioni di debug. Si rimanda alla sezione 6 per una discussione nel dettaglio di questo approccio.

Una volta popolati i *vmcmd*, la procedura esegue il reset dello spazio di memoria del processo, per poi applicarvi le modifiche e operazioni specificate nel set di *vmcmd* dell'exec package. A questo punto, dato che lo spazio di indirizzamento è stato mutato, gli eventuali thread del processo (precedentemente fermati) vengono eliminati.

Se i comandi vengono applicati con successo, il processo contenente l'immagine di un nuovo programma è quasi pronto per essere rimesso in esecuzione. Vengono eliminate le strutture dati temporanee utilizzate per la messa in moto del programma, vengono "sparpagliati" alcuni frammenti di memoria del nuovo processo in modo da rendere la disposizione dello spazio di indirizzamento meno prevedibile per un possibile utente malevolo, e infine viene restituito il flusso d'esecuzione ai processi *user-space*.

5.3 Composizione di `sys_spawn()`

Una volta acquisita familiarità con le precedenti system call, la composizione di `sys_spawn` è un compito concettualmente diretto, quantomeno nella sua variante più basilare (che non permette la gestione di tutti gli argomenti opzionali della funzione `posix_spawn` citata precedentemente).

È infatti possibile vedere `sys_spawn` come una concatenazione delle operazioni eseguite da `fork1` e `sys_execve` in maniera atomica, senza lasciare il *kernel-space* fino a che il nuovo programma non è pronto all'esecuzione. Grazie a questa atomicità, `spawn` non richiede l'esecuzione di diverse operazioni intermedie, che possono essere rimosse. Vediamo quindi nel dettaglio queste differenze.

La prima differenza sostanziale, già accennata nella sezione 5.1, riguarda la copia dello spazio di indirizzamento del processo padre nel processo figlio. Essa avviene all'interno di `fork1` attraverso la funzione `uvmspace_fork`. Anziché popolare lo spazio di memoria con i dati del padre, `spawn` necessita l'allocazione di memoria pulita, da riempire più tardi nella fase di exec. Fortunatamente, il kernel di OpenBSD fornisce una funzione che fa esattamente questo: `uvmspace_alloc` [59]. `uvmspace_alloc` ottiene e inizializza parzialmente un address space virtuale, che può poi essere "riempito" con quanto desiderato.

Le altre divergenze si verificano invece nella fase di exec. Nella sezione 5.2 si è visto che normalmente è necessario sospendere tutti i thread appartenenti al processo che vuole cambiare la sua immagine. Questo non si rende necessario in `sys_spawn`, in quanto il processo, appena creato e vuoto, non ha avuto la possibilità di creare e mettere in esecuzione alcun thread.

Anche alcune operazioni sullo spazio di indirizzamento del processo da modificare diventano ridondanti. Per esempio, `uvmspace_exec` dissocia la memoria del processo per poi re-inizializzarla — un lavoro totalmente inutile durante una `spawn`, dato che lo spazio di memoria è già pulito e inizializzato.

Per il resto, è possibile che qualcuno dei *vmcmd* descritti in precedenza necessiti di essere modificato o rimosso, ma le caratteristiche restanti di `sys_execve` sono applicabili a `sys_spawn` in maniera diretta.

Purtroppo, però, questa implementazione di `spawn` deve ancora fare i conti con la realtà dell'ambiente circostante: nessuno, prima d'ora, aveva previsto nel kernel una system call simile, e questo porta a numerose problematiche descritte nella prossima sezione.

6 Problematiche riscontrate

Sebbene il codice di `fork1` e `sys_execve` sia, almeno superficialmente, compatibile con `spawn`, il resto del kernel di OpenBSD non condivide questa caratteristica.

Una serie di supposizioni disseminate in numerose parti del kernel, inesperienza da parte del sottoscritto e scelte implementative discutibili da parte degli sviluppatori del sistema operativo non hanno permesso alla system call `spawn` di mettere in discussione l'egemonia di `fork`.

6.1 `KASSERT(p == curproc)`

La problematica forse più importante riscontrata durante lo sviluppo, come anticipato dal titolo, è la supposizione che l'operazione di exec venga innescata dal processo che deve essere modificato.

Nel modello di creazione dei processi "fork & exec", la richiesta di mutare un processo per far sì che esegua un nuovo programma può avvenire solo dal processo stesso. Affinché un processo *proc* possa eseguire un programma `prog`, *proc* deve chiamare la funzione `exec` — in altre parole, il chiamante della system call corrisponde sempre al processo modificato.

In un modello di creazione "spawn", invece, succede che se un processo *proc1* vuole lanciare un programma `prog`, il kernel crea un processo *proc2*, nuovo, sul quale mettere in esecuzione `prog`. In questo scenario, il chiamante della system call è *proc1*, ma il processo modificato è *proc2*.

È quindi ragionevole che un kernel modellato attorno al paradigma "fork & exec" richieda che la corrispondenza tra processo chiamante e modificato sia sempre verificata, indipendentemente dalla motivazione. Potrebbe essere un semplice controllo di integrità, o un requisito imprescindibile per il corretto funzionamento del sistema.

Nel kernel di OpenBSD viene verificata tale corrispondenza in numerose parti del codice relativo a `sys_execve`, specialmente in alcune funzioni invocate come parte dei *vmcmd* descritti in precedenza. Un esempio di tale controllo è riportato nel listing 23.

Non è chiaro se questi controlli siano vitali per il buon funzionamento del sistema, in quanto non sembra esserci documentazione a riguardo. Rimuovere queste asserzioni ha portato alla manifestazione di alcuni errori di memoria, *"EFAULT"*, i quali suggeriscono che la corrispondenza di *p* e *curproc* sia effettivamente richiesta dalle routine di gestione della memoria.

6.2 Difficoltà di debugging

La scoperta di *bug* e problemi in fase di sviluppo non è una rarità; è anzi una parte vera e propria dello sviluppo software stesso.

Lavorare alla risoluzione dei bug individuati in questo progetto, però, è stato particolarmente gravoso a causa del dominio di sviluppo.

Un kernel è un software speciale. Esso infatti è responsabile del funzionamento della macchina nella sua totalità — in un certo senso, è il programma usato per eseguire gli altri programmi.

```
int
VOP_OPEN(struct vnode *vp, int mode, struct ucred *cred, struct proc *p)
{
    struct vop_open_args a;
    a.a_vp = vp;
    a.a_mode = mode;
    a.a_cred = cred;
    a.a_p = p;

    KASSERT(p == curproc);

    if (vp->v_op->vop_open == NULL)
        return (EOPNOTSUPP);

    return ((vp->v_op->vop_open)(&a));
}
```

Listing 23: Controllo della corrispondenza tra *p* e *curproc* con un assert

A differenza dei normali software, non è possibile lanciare il kernel all'interno di un *debugger* per ispezionarne lo stato e individuare difetti da correggere. Per eseguire il debug di un kernel, il debugger dev'essere parte del kernel stesso.

OpenBSD fornisce un debugger, *ddb* [45], che presenta una interfaccia a riga di comando simile al debugger *GDB* [21]. La sua utilità, però, è stata limitata.

Padroneggiare l'utilizzo un debugger come `ddb` non è semplice, soprattutto senza una precedente esperienza il debugger `gdb`. Il suo corretto funzionamento, inoltre, richiede che il kernel venga compilato in modalità debug attraverso la modifica di un file di configurazione documentato nella pagina di manuale *config* [44]. Sono venuto a conoscenza di questo file di configurazione solamente al termine del mio lavoro di sviluppo, e non mi è quindi stato possibile utilizzare `ddb` nel pieno delle sue funzionalità.

In mancanza di un debugger, l'unico modo per individuare problemi nel codice è ricorrere a quello che viene chiamato *printf-style debugging*, ovvero utilizzare istruzioni di stampa a schermo per l'analisi dello stato del programma.

Per quanto utile e immediato, questo tipo di debugging non ha tardato a mostrare i suoi limiti. Come anticipato nella sezione 5.2, il ricorrente uso di *callback* e puntatori a funzione da parte di alcune routine rende pressoché impossibile avere un quadro completo di quello che sta accadendo. Ad esempio, nel listing 24 vengono invocate alcune callback con l'espressione `(*vcp->ev_proc)(p, vcp)`. Individuare quale funzione venga effettivamente

chiamata così da capire quale di essa restituisca un errore è impossibile senza un debugger. Stampare il puntatore `vcp->ev_proc`, ovviamente, mostra a schermo soltanto un mero indirizzo di memoria, non facilmente riconducibile alla funzione corrispondente.

```
int
exec_process_vmcmds(struct proc *p, struct exec_package *epp)
{
    struct exec_vmcmd *base_vc = NULL;
    int error = 0;
    int i;

    for (i = 0; i < epp->ep_vmcmds.evs_used && !error; i++) {
        struct exec_vmcmd *vcp;

        vcp = &epp->ep_vmcmds.evs_cmds[i];

        if (vcp->ev_flags & VMCMD_RELATIVE) {
            vcp->ev_addr += base_vc->ev_addr;
        }
        error = (*vcp->ev_proc)(p, vcp);
        if (vcp->ev_flags & VMCMD_BASE) {
            base_vc = vcp;
        }
    }

    kill_vmcmds(&epp->ep_vmcmds);

    return (error);
}
```

Listing 24: La funzione `exec_process_vmcmds` richiama diverse funzioni attraverso dei puntatori

Un altro limite di questo approccio è costituito dalle caratteristiche delle funzioni di stampa stesse, `log` [48] e `uprintf` [55]. La prima, usata nella sezione 4.1 per la system call esemplificativa, è utile per registrare alcuni aspetti del flusso di esecuzione del kernel per poi consultarli in un secondo momento. La seconda, `uprintf`, stampa il messaggio passatole come argomento all'interno del terminale attualmente in uso, proprio come la classica `printf` della libreria C.

Perché l'output sia visibile a schermo, però, il kernel deve poter lasciare in esecuzione il processo utente in controllo del terminale. Questo è un problema in quanto, se un errore fatale si verificasse immediatamente dopo la chiamata a `uprintf`, il messaggio da essa stampato non avrebbe il tempo

di comparire all'interno del terminale, e l'informazione, spesso di vitale importanza, verrebbe persa.

6.3 Problematiche sociali

La terza categoria di problemi riguardanti questo progetto appartiene a problemi non puramente tecnici, ma sociali.

La funzione `fork`, nonostante le sue criticità, è una delle caratteristiche che hanno definito Unix. È quindi poco sorprendente come diversi sostenitori degli ideali Unix tendano a difendere `fork` quando comparata a `spawn`; è possibile osservare questo fenomeno in forum online, mailing list e thread di commenti fatti ad alcuni degli articoli citati in precedenza.

Questo attaccamento a `fork` potrebbe essere uno dei motivi per cui, chiedendo chiarimenti su alcune scelte implementative del kernel di OpenBSD sulla mailing list di sviluppo, non ho avuto alcun riscontro [41].

D'altronde, la lista *openbsd-tech* alla quale è stata rivolta la mia email è altamente tecnica e viene solitamente usata per la distribuzione e discussione di *patch*. È anche possibile che la mail sia stata ignorata perché non accompagnata da codice già integrabile nel sistema operativo.

7 Passi futuri

Nonostante le problematiche incontrate, avere `spawn` come system call rimane un obiettivo che vale la pena perseguire.

Il primo passo per muoversi verso un futuro in cui `spawn` possa vivere all'interno del kernel di OpenBSD è quello di disaccoppiare il processo chiamante dal processo da modificare nella funzione `exec`. Tale cambiamento sarebbe una miglioria di per sé, senza considerare `spawn`, e consentirebbe in un secondo momento di aggiungere la nuova primitiva senza una delle problematiche maggiori discusse nella presente tesi.

Gli sviluppatori di OpenBSD, dalla loro parte, continuano a migliorare il codice dell'area del kernel qui discussa, applicando ottimizzazioni prestazionali [42] e semplificazioni utili al debug [25].

Un altro obiettivo che vale la pena perseguire, sia in OpenBSD che altrove, consiste nel *porting* di software già esistente, riscrivendolo per fare uso di `posix_spawn` invece che `fork`. Rispetto a implementare una system call, è un compito più semplice, e avere un gran quantitativo di software facente uso di una interfaccia `spawn` non ottimizzata stimola il bisogno della creazione della primitiva a livello kernel, con conseguenti guadagni prestazionali considerevoli in diversi applicativi.

Guardando ad altri sistemi operativi, la situazione è simile. C'è volontà di adottare delle primitive migliori, ma c'è ancora lavoro da fare.

In conclusione, spero che questa tesi possa dare una piccola spinta al mondo dei sistemi operativi Unix verso l'adozione della semplice `spawn`, e

che abbia cambiato la percezione del lettore nei confronti della venerabile `fork`.

Ringraziamenti

Voglio dedicare questo ultimo spazio della mia tesi a tutte quelle persone che, in un modo o nell'altro, l'hanno resa possibile.

Ringrazio prima di tutti il professor Tini, relatore della tesi e tutor dell'attività di tirocinio, per avermi permesso di dedicarmi a questo dominio di ricerca sui sistemi operativi che trovo particolarmente interessante. Insieme a lui, i miei ringraziamenti vanno anche a tutti i professori che hanno dedicato il loro tempo a formarmi durante il corso di studi, lasciando tutti, a modo proprio, un marchio indelebile che va oltre a quanto scritto sui loro syllabi.

Ci tengo a ringraziare i miei colleghi e amici del corso di Informatica, grazie ai quali l'università non è stato solo un posto di crescita professionale, ma anche personale. Questi vanno in particolar modo ad Alberto, Filippo e Jacopo che mi hanno accompagnato per tutti i tre anni degli studi. Ringrazio anche i miei amici del corso di Scienze Biologiche, che hanno riempito i momenti ricreativi dell'ultimo anno con le loro passioni e personalità, arricchendomi con le loro conoscenze e prospettive tanto diverse dalle mie.

Mi è obbligatorio ringraziare anche il mio amico Edo, che mi ha da sempre stimolato a potenziare le mie competenze in ambito informatico, e che ha fornito preziosi consigli linguistici che hanno reso questa tesi indubbiamente più gradevole e coerente.

Infine, ringrazio la mia famiglia, che mi ha permesso — e caldamente consigliato — di intraprendere questo percorso di studi, e che mi ha supportato, incondizionatamente, in ogni momento.

Acronimi

CoW Copy-on-Write. 2

CVS Concurrent Versions System. 6

DSL Domain-Specific Language. 4

IDE Integrated Development Environment. 6–8

LSP Language Server Protocol. 7, 8

MMU Memory Management Unit. 3

OOM Out Of Memory. 3

PCB Process Control Block. 19

VCS Version Control System. 6

Riferimenti bibliografici

[1] Daroc Alden. «OpenBSD system-call pinning». In: *LWN.net* (31 gen. 2024). URL: `https://lwn.net/Articles/959562/` (cit. a p. 10).

[2] Apple. *Open Source Releases*. Ver. 15. URL: `https://opensource.apple.com/releases/` (visitato il giorno 16/11/2024) (cit. a p. 7).

[3] ArchWiki. *Running GUI applications as root*. 28 Set. 2024. URL: `https://wiki.archlinux.org/title/Running_GUI_applications_as_root` (cit. a p. 7).

[4] Cygwin authors. «Cygwin Overview». In: *Cygwin User's Guide*. 9 Apr. 2024. Cap. Process Creation. URL: `https://cygwin.com/cygwin-ug-net/highlights.html#ov-hi-process` (cit. a p. 2).

[5] Andrew Baumann et al. «A fork() in the road». In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 17th Workshop on Hot Topics in Operating Systems. HotOS '19 (Bertinoro, Italy, 13–15 mag. 2019). New York, NY, USA: ACM, 13 mag. 2019. DOI: `10.1145/3317550.3321435`. URL: `https://www.microsoft.com/en-us/research/publication/a-fork-in-the-road/` (cit. a p. 2).

[6] *Bear. Generate a compilation database for clang tooling*. Ver. 3.1.3. 28 Ago. 2023. URL: `https://github.com/rizsotto/Bear` (cit. a p. 9).

[7] *conf/files*. Ver. 1.730. 3 Feb. 2024. URL: https://github.com/openbsd/src/blob/22a98bd7aa1a829cdeacdbc8ee7f1585cc21f19a/sys/conf/files (cit. a p. 14).

[8] Jonathan Corbet. «Race-free process creation in the GNU C Library». In: *LWN.net* (1 set. 2023). URL: https://lwn.net/Articles/943022/ (cit. a p. 3).

[9] Jonathan Corbet. «Ringing in a new asynchronous I/O API». In: *LWN.net* (15 gen. 2021). URL: https://lwn.net/Articles/776703/ (cit. a p. 3).

[10] *CreateProcess function. processthreadsapi.h*. Win32 API. 2 Set. 2023. Cap. Processes and threads. URL: https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa (cit. a p. 3).

[11] *CVS. Concurrent Versions System*. Ver. 1.11.23. 8 Mag. 2008. URL: https://cvs.nongnu.org (cit. a p. 6).

[12] Drew DeVault. «The Helios microkernel». In: *Drew DeVault's blog* (13 giu. 2022). URL: https://drewdevault.com/2022/06/13/helios.html (cit. a p. 5).

[13] Casper Dik. «posix_spawn() as an actual system call». In: *Oracle Solaris Blog* (12 feb. 2018). URL: https://blogs.oracle.com/solaris/post/posix_spawn-as-an-actual-system-call (cit. a p. 4).

[14] Drew Eckhardt e Michael Kerrisk. *clone. create a child process*. Linux man-pages. Ver. 6.8. 2 Mag. 2024. Cap. 2. URL: https://man7.org/linux/man-pages/man2/clone.2.html (cit. alle pp. 3, 5).

[15] ecree. «Microsoft research: A fork() in the road. Comment». In: *LWN.net* (11 apr. 2019). URL: https://lwn.net/Articles/785715/ (cit. a p. 2).

[16] Jake Edge. «Introducing io_uring_spawn». In: *LWN.net* (20 set. 2022). URL: https://lwn.net/Articles/908268/ (cit. a p. 3).

[17] Rich Felker. «vfork considered dangerous». In: *EWONTFIX* (21 ott. 2012). URL: https://ewontfix.com/7/ (cit. a p. 4).

[18] Python Software Foundation. *multiprocessing. Process-based parallelism*. Python documentation. Ver. 3.13. 7 Nov. 2024. Cap. Concurrent Execution. URL: https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods (cit. a p. 2).

[19] Python Software Foundation. *os.fork. Fork a child process*. Python documentation. Ver. 3.13. 7 Nov. 2024. Cap. os. URL: https://docs.python.org/3/library/os.html#os.fork (cit. a p. 2).

[20] X.Org foundation et al. *X Window System*. Ver. 11R7.7. 6 Giu. 2012. URL: https://www.x.org/releases/X11R7.7/ (cit. a p. 6).

[21] *GDB. The GNU Project Debugger*. Ver. 15.2. 29 Set. 2024. URL: https://www.sourceware.org/gdb/ (cit. a p. 23).

[22] *Git. Fast, scalable, distributed revision control system*. Ver. 2.47.0. 6 Ott. 2024. URL: https://git-scm.com (cit. a p. 6).

[23] *GNU Hurd*. URL: https://www.gnu.org/software/hurd/ (cit. a p. 5).

[24] *GNU Hurd / glibc / fork*. 17 Mar. 2016. URL: https://www.gnu.org/software/hurd/glibc/fork.html (cit. a p. 2).

[25] Jonathan Gray. *de-macro new_vmcmd()*. 1 Nov. 2024. URL: https://marc.info/?i=ZyTXHpLVPu3kUbFG%20()%20largo%20!%20jsg%20!%20id%20!%20au (cit. a p. 25).

[26] *gRPC. A high performance, open source universal RPC framework*. URL: https://grpc.io (cit. a p. 9).

[27] Andreas K. Hüttel. *The GNU C Library version 2.39 is now available*. 31 Gen. 2024. URL: https://lists.gnu.org/archive/html/info-gnu/2024-01/msg00017.html (cit. a p. 3).

[28] «Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API). Amendment D: Additional Real time Extensions». In: *IEEE Std 1003.1d-1999* (1999). A cura di IEEE. DOI: 10.1109/IEEESTD.1999.91515 (cit. a p. 3).

[29] «POSIX.1-2024». In: The Open Group Standard Base Specifications 8 (14 giu. 2024). A cura di IEEE e The Open Group. DOI: 10.1109/IEEESTD.2024.10555529. URL: https://pubs.opengroup.org/onlinepubs/9799919799/ (cit. a p. 3).

[30] *INSTALL.amd64. INSTALLATION NOTES for OpenBSD/amd64*. Ver. 7.5. 5 Apr. 2024. URL: https://cdn.openbsd.org/pub/OpenBSD/7.5/amd64/INSTALL.amd64 (cit. a p. 6).

[31] *JSON Compilation Database Format Specification. Format for specifying how to replay single compilations independently of the build system*. Ver. 16.0.0. 17 Mar. 2023. URL: https://releases.llvm.org/16.0.0/tools/clang/docs/JSONCompilationDatabase.html (cit. a p. 9).

[32] Michael Kerrisk. *signal-safety. async-signal-safe functions*. Linux man-pages. Ver. 6.8. 2 Mag. 2024. Cap. 7. URL: https://man7.org/linux/man-pages/man7/signal-safety.7.html (cit. alle pp. 2, 4).

[33] Yegappan Lakshmanan et al. *Vim9 lsp. Language Server Protocol (LSP) plugin for Vim9*. URL: https://github.com/yegappan/lsp (cit. a p. 8).

[34] *Maven. Software project management and comprehension tool*. Ver. 3.9.9. 17 Ago. 2024. URL: https://maven.apache.org (cit. a p. 8).

[35] Microsoft. *Language Server Protocol*. Ver. 3.17. 5 Ott. 2022. URL: https://microsoft.github.io/language-server-protocol/ (cit. a p. 7).

[36] Bram Moolenaar et al. *CTRL-]. Jump to the definition of the keyword under the cursor*. Vim reference manual. Ver. 9.1. 1 Ago. 2024. Cap. Tags and special searches. URL: https://vimhelp.org/tagsrch.txt.html#CTRL-%5D (cit. a p. 8).

[37] Bram Moolenaar et al. *gd. Goto local Declaration*. Vim reference manual. Ver. 9.1. 1 Ago. 2024. Cap. Patterns and search commands. URL: https://vimhelp.org/pattern.txt.html#gd (cit. a p. 8).

[38] Bram Moolenaar et al. *Vim. The ubiquitous text editor*. Ver. 9.1. 2 Gen. 2024. URL: https://www.vim.org (cit. a p. 6).

[39] *OpenBSD. Events and Papers*. URL: https://www.openbsd.org/events.html (cit. a p. 5).

[40] *Oracle Solaris. Consistent. Simple. Secure*. URL: https://www.oracle.com/solaris/ (cit. alle pp. 4, 5).

[41] Andrea Pappacoda. *Implementing a spawn(2) system call*. 12 Ott. 2024. URL: https://marc.info/?i=D4TR60HGKEZZ.2P6VEJST7OMU7%20()%20pappacoda%20!%20it (cit. a p. 25).

[42] Martin Pieuchot. *Reduce KERNEL_LOCK() contention in uvn_detach()*. 19 Ott. 2024. URL: https://marc.info/?i=ZxOxE39YN1DhXhLX%20()%20oliva%20!%20grenadille%20!%20net (cit. a p. 25).

[43] The LLVM Project. *clangd. Teach your editor C++*. Ver. 16.0.6. 14 Giu. 2023. URL: https://clangd.llvm.org (cit. a p. 8).

[44] The OpenBSD Project. *config. Kernel configuration options*. OpenBSD manual pages. Ver. 7.5. 14 Dic. 2023. Cap. 4. URL: https://man.openbsd.org/OpenBSD-7.5/config.4 (cit. a p. 23).

[45] The OpenBSD Project. *ddb. Kernel debugger*. OpenBSD manual pages. Ver. 7.5. 5 Feb. 2024. Cap. 4. URL: https://man.openbsd.org/OpenBSD-7.5/ddb.4 (cit. a p. 23).

[46] The OpenBSD Project. *ex, vi, view. Text editors*. OpenBSD manual pages. Ver. 7.5. 12 Feb. 2024. Cap. 1. URL: https://man.openbsd.org/OpenBSD-7.5/vi.1 (cit. a p. 6).

[47] The OpenBSD Project. *fork1. Create a new process*. OpenBSD manual pages. Ver. 7.5. 29 Dic. 2022. Cap. 9. URL: https://man.openbsd.org/OpenBSD-7.5/fork1.9 (cit. a p. 18).

[48] The OpenBSD Project. *log. Log a message from the kernel through the /dev/klog device*. OpenBSD manual pages. Ver. 7.5. 14 Set. 2015. Cap. 9. URL: https://man.openbsd.org/OpenBSD-7.5/log.9 (cit. alle pp. 12, 24).

[49] The OpenBSD Project. *make. Maintain program dependencies*. OpenBSD manual pages. Ver. 7.5. 10 Ago. 2023. Cap. 1. URL: https://man.openbsd.org/OpenBSD-7.5/make (cit. a p. 6).

[50] The OpenBSD Project. *mg. Emacs-like text editor*. OpenBSD manual pages. Ver. 7.5. 16 Ott. 2023. Cap. 1. URL: https://man.openbsd.org/OpenBSD-7.5/mg.1 (cit. a p. 6).

[51] The OpenBSD Project. *OpenBSD FAQ - Installation Guide*. URL: https://www.openbsd.org/faq/faq4.html (cit. a p. 6).

[52] The OpenBSD Project. *packages. Overview of the binary package system*. OpenBSD manual pages. Ver. 7.5. 5 Gen. 2022. Cap. 7. URL: https://man.openbsd.org/OpenBSD-7.5/packages.7 (cit. alle pp. 7, 9).

[53] The OpenBSD Project. *pkg_add. Install or update software packages*. OpenBSD manual pages. Ver. 7.5. 12 Ago. 2022. Cap. 1. URL: https://man.openbsd.org/OpenBSD-7.5/pkg_add.1 (cit. a p. 7).

[54] The OpenBSD Project. *ports. Contributed applications*. OpenBSD manual pages. Ver. 7.5. 7 Set. 2023. Cap. 7. URL: https://man.openbsd.org/OpenBSD-7.5/ports.7 (cit. a p. 7).

[55] The OpenBSD Project. *printf, uprintf. Kernel formatted output conversion*. OpenBSD manual pages. Ver. 7.5. 11 Set. 2022. Cap. 9. URL: https://man.openbsd.org/OpenBSD-7.5/printf.9 (cit. a p. 24).

[56] The OpenBSD Project. *release. Building an OpenBSD release*. OpenBSD manual pages. Ver. 7.5. 25 Dic. 2023. Cap. 8. URL: https://man.openbsd.org/OpenBSD-7.5/release.8 (cit. a p. 10).

[57] The OpenBSD Project. *syslog. Control system log*. OpenBSD manual pages. Ver. 7.5. 31 Mar. 2022. Cap. 3. URL: https://man.openbsd.org/OpenBSD-7.5/syslog.3 (cit. a p. 11).

[58] The OpenBSD Project. *user. Manage user login information on the system*. OpenBSD manual pages. Ver. 7.5. 6 Feb. 2022. Cap. 8. URL: https://man.openbsd.org/OpenBSD-7.5/user.8 (cit. a p. 10).

[59] The OpenBSD Project. *uvm_map. Virtual address space management interface*. OpenBSD manual pages. Ver. 7.5. 9 Dic. 2022. Cap. 9. URL: https://man.openbsd.org/OpenBSD-7.5/uvm_map.9 (cit. alle pp. 19, 21).

[60] The OpenBSD Project. *xenodm. X Display Manager*. OpenBSD manual pages. Ver. 7.5. 30 Ago. 2021. Cap. 1. URL: https://man.openbsd.org/OpenBSD-7.5/xenodm (cit. a p. 6).

[61] Arthur Rasmusson e Louis Castricato. «Why computers suck and how learning from OpenBSD can make them marginally less horrible». In: *Arc Compute Blog* (5 dic. 2019). URL: https://arccompute.com/blog/why-computers-suck-and-how-openbsd-makes-them-marginally-better/ (cit. a p. 5).

[62] *Redox. Your Next(Gen) OS.* URL: https://www.redox-os.org (cit. a p. 5).

[63] Dennis M. Ritchie. *The Evolution of the Unix Time-sharing System.* Bell Laboratories, 1996. URL: https://www.bell-labs.com/usr/dmr/www/hist.pdf (cit. alle pp. 2, 3).

[64] rueda. «Pinning all system calls». In: *OpenBSD Journal* (9 dic. 2023). URL: https://www.undeadly.org/cgi?action=article;sid=20231209115835 (cit. a p. 10).

[65] sarthak. *Why do we need to fork to create new processes?* 11 Giu. 2014. URL: https://unix.stackexchange.com/q/136637 (cit. a p. 2).

[66] *SerenityOS. A graphical Unix-like operating system for desktop computers.* URL: https://serenityos.org (cit. a p. 5).

[67] *syscalls.master.* Ver. 1.257. 26 Gen. 2024. URL: https://github.com/openbsd/src/blob/22a98bd7aa1a829cdeacdbc8ee7f1585cc21f19a/sys/kern/syscalls.master (cit. a p. 12).

[68] *systemd. System and Service Manager.* URL: https://systemd.io (cit. a p. 3).

[69] *systemd 255 released.* 6 Dic. 2023. URL: https://lists.freedesktop.org/archives/systemd-devel/2023-December/049745.html (cit. a p. 3).

[70] *The Ares Operating System. A new operating system under development.* URL: https://ares-os.org (cit. a p. 5).

[71] *The FreeBSD Project. The power to serve.* URL: https://www.freebsd.org (cit. a p. 5).

[72] *The NetBSD Project. Free, fast, secure, and highly portable Unix-like Open Source operating system.* URL: https://www.netbsd.org (cit. a p. 5).

[73] *The Open Source Definition.* Ver. 1.9. 22 Mar. 2007. URL: https://opensource.org/osd (visitato il giorno 16/02/2024) (cit. a p. 5).

[74] *The OpenBSD Project. Only two remote holes in the default install, in a heck of a long time!* Ver. 7.5. 5 Apr. 2024. URL: https://www.openbsd.org (cit. a p. 5).

[75] *The Sortix Operating System. Small self-hosting operating-system aiming to be a clean and modern POSIX implementation.* URL: https://sortix.org (cit. a p. 5).