

Massimo Mennuni - S304731

Analisi Comparativa tra OS161 e altri Sistemi Operativi Open-Source All'avanguardia per Sistemi Embedded e Computer general purpose

Indice

1. Introduzione	3
2. Sistemi operativi opensource analizzati	4
2.1 xv6	4
2.2 Nachos	5
2.3 MINIX	5
2.4 Pintos	6
2.5 Xinu	7
1.1 Redox	8
1.2 FreeRTOS	9
2. Analisi comparativa rispetto OS161	10
3. Approfondimento su Pintos	11
3.1 Avvio del Sistema Operativo	11
3.2 Debugging tools	12
3.3 Development tools	13
3.4 Codice di Pintos	13
3.4.1 Loader	13
3.4.2 Mappa della memoria	14
3.4.3 Thread	14
3.4.4 Thread switch	16
3.4.5 Sincronizzazione	17
3.4.6 Allocazione di memoria	19
3.4.7 Indirizzamento virtuale	22
3.4.8 Page Table	23
3.4.9 Hash table	23
3.4.10 Scheduler	24
4. Modifiche al codice	24
4.1 Modifica del codice di test	24
4.1 Debug Test	25
4.2 Analisi delle carenze di Pintos	25
5. Conclusioni	28

1. Introduzione

L'analisi qui presentata è volta a comparare le principali caratteristiche dei seguenti sistemi operativi opensource rispetto a OS161 (sistema operativo didattico adottato dal corso di Programmazione di Sistema 2022/2023).

Il panorama dei sistemi operativi opensource è molto ampio ed è difficile fornire una disamina esaustiva di tutti quelli presenti. In questo capitolo vengono introdotti quelli che sono stati considerati più interessanti fornendo i riferimenti alle fonti da cui sono state raccolte le diverse informazioni.

In particolare nel prossimo capitolo sono stati analizzati superficialmente i seguenti sistemi operativi:

- xv6
- Nachos
- MINIX
- Pintos
- Xinu
- Redox
- FreeRTOS

Mentre il capitolo successivo è dedicato ad un approfondimento su Pintos.

2. Sistemi operativi opensource analizzati

Tra le principali caratteristiche valutate in questa prima fase ci sono:

- Piattaforme HW supportate
- Linguaggi di programmazione adottati
- System Calls
- Meccanismi di sincronizzazione
- Memoria virtuale e Memory Management Unit (MMU)
- Implementazione di diversi algoritmi di scheduling

Oltre a questi aspetti in ogni paragrafo dedicato ad un sistema operativo sono riportati altri aspetti ritenuti interessanti nella comparazione rispetto a OS161.

2.1 xv6

Il sistema operativo Xv6 è un sistema operativo leggero progettato come strumento didattico ed è basato su Unix Version 6 (sistema operativo sviluppato da Bell Labs, rilasciato per la prima volta nel maggio del 1975 e mantenuto attivo fino al 1985).

Xv6 è stato sviluppato per supportare principalmente le piattaforme HW basate su architettura x86: può essere tuttavia eseguito su QEMU, consentendone quindi la portabilità su tutti i sistemi in cui può girare QEMU in modalità emulazione x86.

Xv6 è scritto in C, come la maggior parte dei sistemi operativi di derivazione Unix.

Fornisce alcune systems calls di base tra cui:

- Supporto alla gestione di thread e processi (fork, exit, wait, kill, getpid, sleep, exec, sbrk)
- Supporto alla gestione dei file (open, write, read, close)
- Supporto alla gestione del filesystem (dup, pipe, chdir, mkdir, mknod, fstat, stat, link, unlink)

Supporta i seguenti meccanismi di sincronizzazione:

- semafori per la gestione delle risorse condivise e sincronizzazione tra i processi

La gestione della memoria:

- Implementa una forma di paging semplice

Gli algoritmi di scheduling dei processi:

- xv6 utilizza un algoritmo di scheduling round-robin semplice, in cui ogni processo ottiene un tempo di CPU uguale prima di passare al successivo.

Riferimenti:

Website	Wikipedia.org	Google Scholar (Articles)	Source code
https://pdos.csail.mit.edu/6.828/2022/xv6.html	https://en.wikipedia.org/wiki/Version_6_Unix	https://cs326.cs.usfca.edu/assignments/book-riscv-rev3.pdf	https://github.com/mit-pdos/xv6-public

2.2 Nachos

Nachos è un sistema operativo sviluppato da University of California, Berkeley. Nachos è l'acronimo di *Not Another Completely Heuristic Operating System*, ovvero un sistema operativo che cerca di evitare di adottare strategie basate su approssimazioni o decisioni prese in base a informazioni incomplete come fanno altri sistemi operativi per semplificare e velocizzare alcuni dei loro processi.

Nachos supporta nativamente piattaforme HW basate su MIPS, ma può essere eseguito attraverso un emulatore su piattaforme differenti.

Questo sistema operativo è scritto principalmente in C++, anche se alcune parti del codice sono scritte in C o assembly quando è richiesta un'interazione di basso livello con l'HW. Esiste anche una versione di Nachos scritta in Java.

Nachos è un sistema operativo didattico che punta a bilanciare semplicità e completezza non entrando in tutti gli aspetti, come ad esempio nella gestione della memoria e dei processi per permettere un graduale avvicinamento da parte degli studenti ai concetti sottostanti senza entrare in dettagli che renderebbero l'apprendimento più complesso.

Fornisce solo alcune systems calls di base e molte di queste con implementazioni molto semplici tra cui:

- Apertura, chiusura, lettura e scrittura di file (Create, Open, Write, Read, Close)
- Creazione e terminazione thread e processi (halt, exit, exec, join, yield)

La gestione del filesystem è molto basica, non prevede nessuna struttura gerarchica, tutti i file si trovano in una stessa directory.

Supporta tutti i principali meccanismi di sincronizzazione:

- semafori
- Locks

La gestione della memoria virtuale è semplice e prevede l'uso di una singola page table.

Gli algoritmi di scheduling dei processi si basano sull'assegnazione di un time slice randomico ad ogni thread e la pianificazione è di tipo "non-preemptive", non può quindi essere interrotta dal kernel fino al termine della sua esecuzione.

Riferimenti:

Website	Wikipedia.org	Google Scholar (Articles)	Source code
https://homes.cs.washington.edu/~tom/nachos/	https://es.wikipedia.org/wiki/NachOS	https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b33959031d950de596586a4c0b259af292f60041	[NOT OFFICIAL] https://github.com/daviddwlee84/OperatingSystem/tree/master/Nachos/nachos-3.4

2.3 MINIX

Minix è un sistema operativo scritto da Andrew S. Tanenbaum, tra la fine degli anni ottanta e l'inizio degli anni novanta per le piattaforme Intel ed è stato usato come riferimento per la progettazione di Linux da parte di Linux Torvald.

Tra le caratteristiche principali di Minix c'è l'utilizzo di microkernel, che prevede l'organizzazione delle diverse funzioni del sistema operativo per moduli e lasciando al kernel solo le funzionalità essenziali come la gestione dei processi e la comunicazione tra di loro. La modularità incide in modo rilevante nella facilità di manutenzione del software a discapito delle prestazioni.

Un'altra caratteristica rilevante di MINIX è il self-healing, cioè la capacità di auto ripararsi grazie all'utilizzo del microkernel che permette di isolare le funzionalità non core del sistema operativo eseguendole come processi utente (es. driver dei periferici), prevenendo in questo modo la diffusione degli errori: se un processo fallisce non influenzerà gli altri processi e/o il kernel. L'isolamento dei processi e la gestione degli errori critici sono pensati in Minix per garantire una maggiore resilienza del sistema.

Minix supporta molte piattaforme HW differenti: x86, ARM

Questo sistema Operativo è scritto principalmente in C, alcune parti sono scritte in Assembly, quando queste richiedono un controllo di basso livello dell'interazione con l'HW.

Minix supporta un insieme completo di System Calls, essendo un sistema operativo completo e non "didattico".

Per la sincronizzazione tra thread MINIX supporta tutti i principali costrutti di sincronizzazione:

- Spinlock
- Mutex
- Semafori
- RwLock
- condvar

La gestione della memoria virtuale prevede un sistema avanzato con supporto per:

- Paginazione
- Gestione degli errori di pagina
- Allocazione dinamica della memoria

Relativamente agli algoritmi di scheduling dei processi, Minix implementa un meccanismo denominato MLFQ – *Multy Level Feedback Queue* - che schedula i processi con la stessa priorità in modalità round-robin, mentre per evitare la starvation dei processi con meno priorità, viene ridotta la priorità dei processi con maggiore priorità ogni volta che consumano un quanto di tempo.

Website	Wikipedia.org	Google Scholar (Articles)	Source code
https://www.minix3.org/	https://simple.wikipedia.org/wiki/MINIX	https://research.vu.nl/ws/portalfiles/portal/42190059/complete+dissertation.pdf	https://git.minix3.org/index.cgi?p=minix.git

2.4 Pintos

Pintos è un progetto didattico sviluppato presso l'Università di Stanford. È probabilmente uno dei sistemi operativi opensource più diffusi per scopi didattici. Adotta un approccio del kernel minimale, che implementa solo le funzionalità essenziali ed è derivato da Nachos.

Pintos può essere eseguito su piattaforme x86, utilizzando il simulatore Bochs che crea un ambiente virtuale per Pintos.

Questo sistema operativo è scritto principalmente in C; alcune funzioni sono scritte in Assembly quando richiedono un controllo di basso livello dell'HW.

È incluso un supporto per le System Calls di base quali:

- apertura e chiusura di file
- lettura e scrittura
- creazione e terminazione di processi.

Previene la presenza di tutti i meccanismi di sincronizzazione di base:

- Mutex
- Semafori
- Condition variable

La gestione della memoria virtuale di base è basata sull'allocazione di segmenti senza paginazione. L'associazione pagina/frame avviene sommando un base address all'indirizzo logico.

La schedulazione dei processi/thread avviene, nella versione base, attraverso un meccanismo di round-robin: sono però definite (senza impetrazione) tutte le funzioni necessarie per gestire lo scheduling in base alla priorità dei thread.

Website	Wikipedia.org	Google Scholar (Articles)	Source code
http://www.stanford.edu/class/cs140/projects/pintos/pintos.html	https://en.wikipedia.org/wiki/Pintos	https://benpfaff.org/papers/pintos.pdf	http://www.stanford.edu/class/cs140/projects/pintos/pintos.tar.gz

2.5 Xinu

Xinu (acronimo ricorsivo di *Xinu Is Not Unix*) è un sistema operativo didattico progettato per l'apprendimento dei principi dei sistemi operativi. È noto per la sua semplicità e chiarezza nel codice e offre funzionalità come la gestione dei processi, la gestione della memoria e la gestione dei dispositivi.

È costruito in modo da garantire una elevata portabilità su piattaforme differenti: x86, ARM, MIPS, Power PC e altre. Il kernel modulare consente di aggiungere o rimuovere funzionalità facilmente, senza influire sul resto del sistema. Xinu è anche noto per la sua efficienza e per la capacità di garantire prestazioni elevate anche per sistemi real time.

Xinu supporta diverse piattaforme HW, tra cui: MIPS, Raspberry Pi, x86, ARM, PowerPC, Sparc e può inoltre essere portato su altre piattaforme con uno sforzo ragionevole per piattaforme simile a quelle citate.

Questo sistema operativo è scritto principalmente in C, alcune parti sono scritte direttamente in Assembler per ottimizzare l'efficienza del codice in base all'architettura HW su cui viene eseguito.

Xinu include un insieme di system call di base per:

- Gestione dei processi/thread (create(), kill(), getpid(), getprio(), gettime(), sleep(), secume(), suspend(), wait(), signal(), receive(), send(), sleepms())
- Non viene fornito nessun supporto per la gestione dei file

Relativamente ai meccanismi di sincronizzazione tra thread, Xinu fornisce tutti i principali sistemi di sincronizzazione:

- Mutex
- Semafori

La memoria virtuale è gestita attraverso un meccanismo di paginazione con pagine a dimensione fissa di 4KB, appoggiandosi ad una Page Table per ogni processo, con algoritmo di sostituzione LRU (Least Recently Used) e protezione delle pagine relativamente agli accessi in lettura, scrittura o esecuzione.

I thread vengono schedulati in modo preemptive (possono essere interrotti dal kernel) in modalità round-robin, assegnando un quanto di tempo ad ogni thread e gestendo la priorità dei thread.

Nel caso in cui nessun thread sia presente nella coda dei Ready, Xinu, può bloccare il sistema in attesa di un evento o dell'interazione da parte dell'utente, riducendo in questo modo ulteriormente il consumo di energia.

Website	Wikipedia.org	Google Scholar (Articles)	Source code
https://xinu.cs.purdue.edu/	https://en.wikipedia.org/wiki/Xinu	https://api.pageplace.de/preview/DT0400.9781498712446_A36322972/preview-9781498712446_A36322972.pdf https://www.cs.purdue.edu/homes/dxu/cs503/notes/part7.pdf	https://github.com/xinu-os/xinu

1.1 Redox

Redox è un sistema operativo Open-source che si basa su un micro-kernel sviluppato in Rust, progettato per garantire elevati standard di sicurezza e affidabilità e fornire un'alternativa moderna ad altri sistemi operativi.

L'architettura è modulare, in modo da gestire facilmente l'aggiunta o la rimozione di funzionalità specifiche senza alterare la stabilità del sistema nel suo insieme e per favorire le attività di sviluppo e manutenzione del software.

Supporta diverse architetture hardware: workstation, server o dispositivi embedded.

Fornisce un ambiente di sviluppo integrato (IDE) chiamato Ion che supporta la scrittura, il debug e l'esecuzione di applicazioni per il sistema operativo Redox.

Redox supporta le piattaforme HW i686, ARM64, MIPS e offre un set di system calls simili a quelle di altri sistemi operativi. Tra queste ci sono:

- Supporto alla gestione dei file (open(), close(), read(), write())
- Supporto alla gestione dei thread (fork(), exec(), exit(), wait())
- Supporto alla Inter process communication (pipe())
- Supporto al filesystem (chdir(), getcwd(), mkdir(), unlink(), stat())

Relativamente ai meccanismi per la sincronizzazione tra thread Redox offre di default:

- Mutex
- Condition variable
- Semafori
- Atomic operation
- Barriers (sincronizzazione di gruppo di thread)

La memoria virtuale è gestita attraverso il paging su richiesta (solitamente con pagine da 4KB): ogni processo mantiene il proprio spazio di indirizzamento virtuale e ne mappa l'associazione con la corrispondente frame fisica all'interno di una page table. In questo modo i processi sono isolati tra di loro non potendo in alcun modo accedere allo spazio di indirizzamento di memoria di un altro processo.

La scheduling dei thread di Redox è di tipo round-robin, ogni dieci "tick" viene preso il thread successivo nella coda dei ready.

Website	Wikipedia.org	Google Scholar (Articles)	Source code
https://www.redox-os.org/	https://en.wikipedia.org/wiki/Redox_(operating_system)	https://books.google.com/books?hl=it&lr=&id=GJP-DwAAQBAJ&oi=fnd&pg=PA62&dq=redox+micro-	https://github.com/redox-os/redox

		kernel&ots=VfnunXzOx4&sig=yY8Rjl2_ICiRQYldUZfmnopGuXkg	
--	--	--	--

1.2 FreeRTOS

FreeRTOS (*Real-Time Operating System*) è un open source progettato dal MIT ed orientato principalmente per i sistemi embedded e per la gestione dei processi Real Time, cioè per quelle applicazioni che richiedono che certe operazioni vengano eseguite entro determinati limiti temporali.

Un'altra caratteristica fondamentale di FreeRTOS è la portabilità, supporta infatti diverse architetture Hardware, da microcontrollori (CPU orientate all'I/O più che all'elaborazione di informazioni) ai DSP (Digital Signal Processor) e alle FPGA (Field-Programmable Gate Array).

FreeRTOS supporta diverse architetture HW, tra queste ARM (Arduino, raspberry Pi, STM32, ...), x86, MIPS, PowerPC, Xtensa (piattaforma HW usata per microcontrollori e SoC, system on chip).

Questo sistema operativo è scritto in C, ma offre tuttavia interfacce API che possono essere richiamate da altri linguaggi di programmazione quale ad esempio C++.

FreeRTOS è utilizzato in moltissimi ambiti, dai dispositivi IoT, ai sistemi per l'automazione industriale, all'ambito elettro-medicale.

La schedulazione dei thread avviene in modalità preemptive, i thread vengono pianificati in base alla loro priorità, senza che questi possano essere bloccati da thread a priorità minore.

Essendo un sistema orientato ai microcontroller, che non dispongono del supporto HW necessario per la gestione della memoria virtuale, FreeRTOS permette l'allocazione della memoria dinamica attraverso API per l'allocazione e la liberazione di blocchi di memoria.

FreeRTOS è progettato per essere "leggero" e relativamente alla sincronizzazione tra thread offre solo il supporto per:

- semafori
- mutex

Website	Wikipedia.org	Google Scholar (Articles)	Source code
https://www.freertos.org/	https://en.wikipedia.org/wiki/FreeRTOS	https://www.embeddedrelated.com/documents/190225_FreeRTOS_Embedded_World.p	https://github.com/FreeRTOS/FreeRTOS-Kernel

2. Analisi comparativa rispetto OS161

I sistemi operativi descritti brevemente nei paragrafi precedenti, pur essendo un ridotto sottoinsieme di tutti quelli open source presenti, evidenziano principalmente che:

- **Unix Like:** la stragrande maggioranza dei sistemi operativi opensource sono derivanti da Sistemi Operativi Unix Like, questo perché i sistemi operativi proprietari come Windows pur mettendo a disposizione delle versioni “education” del proprio sistema operativo non rendono il codice in essi contenuti OpenSource, ne permettono di sovrascrivere delle API di sistema, perché questo potrebbe compromettere la stabilità del sistema nel suo insieme.
- **C:** Il linguaggio di programmazione di riferimento per la stragrande maggioranza dei sistemi operativi opensource è il C.
- **Kernel monolitico VS micro-kernel:** in base agli obiettivi del sistema operativo, questo può essere implementato adottando un approccio a kernel monolitico (come Linux) che favorisca le performance a discapito della manutenibilità e della resilienza del sistema, rispetto a sistemi a micro-kernel, che permettono di gestire come moduli esterni (che possono girare anche a livello utente) alcune delle funzionalità tipiche offerte dal sistema operativo. Questo permette di garantire un kernel di dimensioni più piccole e modulare che può essere portato su piattaforme con minori risorse disponibili (sistemi embedded/IoT).
- **Specializzazione:** Redox e FreeRTOS mostrano come le scelte architetturali del sistema operativo possano essere funzionali a scopi specifici (Sicurezza, Sistemi Real Time) a discapito della compatibilità con il software sviluppato per i sistemi operativi general purpose.
- **Sistemi Operativi didattici:** rispetto a OS161, Xv6, Nachos, Pintos e Xinu offrono una baseline di servizi disponibili differente, spesso nella documentazione del sistema operativo vengono anche indicati gli step formativi che deve superare chi intende approfondire il funzionamento dei sistemi operativi open source.

Nella tabella qui sotto sono riassunte le caratteristiche principali dei sistemi operativi analizzati:

Sistema Operativo	Piattaforme HW	Linguaggio di programmazione	System Calls	Meccanismi di sincronizzazione	Gestione della memoria virtuale	Scheduling
OS/161	MIPS	C	reboot _time	e Spinlock e Semafori	Segment Allocation	Round-robin
xv6	HW simulato QEMU	C	Processi File Filesystem	Semafori Spinlock Mutex Atomix Instruction	Page Table Kernel Page Table	Round-robin
Nachos	MIPS	C++	Processi File	Semafori Locks	Single Page Table	Non-preemptive Random Time slice
MINIX	x86, ARM	C	Set completo	Spinlock Semafori Mutex RwLock condvar	Per process Page Table with Dynamic Allocation	Round Robin MLFQ – Multy Level Feedbak queue
Pintos	x86 HW simulato QEMU	C	System Handler Calls non implementato	Semafori Mutex Condvar	Segment Allocation	Round-robin
Xinu	MIPS, ARM, x86 HW simulato QEMU	C	Supporto alla gestione dei Processi	Mutex, semafori	Page Table con Free list (first fit kernel & last fit user)	Round-robin
Redox	I686 ARM64	Rust	Set completo	Mutex RwLock Semafori Atomic Instruction Condvar Barriers	Non documentata ufficialmente, in alcuni articoli si fa riferimento al Demand Paing	Round-robin
FreeRTOS	Arm V8 Risc V	C	Set completo	Semafori Mutex	No Virtual Memory management	Priority Pre-emptive Scheduling

3. Approfondimento su Pintos

Per poter approfondire il funzionamento del sistema operativo Pintos, ho preparato una VM con Ubuntu 16.04 che, attraverso Qemu, permetta di simulare un ambiente i386 su cui far girare Pintos. A dispetto di quello che potrebbe sembrare, creare un ambiente di sviluppo per Pintos è tutt'altro che semplice:

- Il codice sorgente messo a disposizione dalla Stanford University non è customizzato di default per funzionare con Qemu, ma con Bochs.
- Il compilatore GCC incluso nelle versioni più recenti di Ubuntu non è adatto alla piattaforma i386 per cui sono scritti i sorgenti di Pintos.
- Molte delle guide presenti in rete, incluse quelle fornite dalla Stanford University, non includono informazioni relativamente alle dipendenze dei software che vengono installati.

Per ovviare questi problemi alla radice, dopo svariati tentativi, ho optato per seguire le indicazioni della guida https://www.researchgate.net/publication/359393118_Pintos_Installation_on_Ubuntu_1604_32-bit che si appoggia a Ubuntu 16.04 TLS e che fornisce informazioni abbastanza precise su come modificare i file di configurazione di Pintos per permettere l'avvio dello stesso in modo corretto.

I binari del sistema operativo sono stati copiati nella cartella `/os/bin`.

Anche in questa guida non tutte le modifiche necessarie ai file di configurazione sono tracciate, per questa ragione, solo per quanto riguarda le modifiche ai file di Pintos, ho seguito anche le indicazioni fornite nella guida https://piazza.com/class/profile/get_resource/itgbdzpqj6417o/iuzt04qzagh40p.

Per accedere alla macchina virtuale lo username è `max` e la password `max123`.

3.1 Avvio del Sistema Operativo

La guida ufficiale, di Pintos è disponibile nei seguenti formati:

- HTML: <https://www.scs.stanford.edu/20wi-cs140/pintos/pintos.html>
- PDF: <https://web.stanford.edu/class/cs140/projects/pintos/pintos.pdf>

La dimensione di Pintos è inferiore al megabyte e la struttura delle sue directory è la seguente:

Directory	Descrizione
Src	radice contiene tutte le altre
src/threads	Kernel directory
src/userprog	Directory dei programmi utente
src/vm	Virtual memory
src/filesys	Filesystem
src/devices	Interfacce di I/O (tastiera, timer, dischi, ...)
src/lib	Set ridotto delle Librerie standard del C
src/lib/kernel	Librerie standard del C usate solo nel kernel
src/lib/use	Librerie standard del C usate solo negli User Program
src/tests	Test del funzionamento delle modifiche richieste dai progetti della Stanford University
src/examples	Programmi utente di prova

Per verificare il corretto funzionamento di Pintos la guida suggerisce di lanciare il comando `pintos run alarm-multiple`. Questo comando:

- Avvia Pintos su Qemu
- Lancia una serie di thread paralleli
- Fornisce, terminata l'esecuzione, alcune informazioni statistiche e quindi spegne la macchina

```
max@max-VirtualBox:~/os/bin$ more test.txt
qemu -hda /tmp/DECIWySeXe.dsk -m 4 -net none -serial stdio
Pilo hda1
Loading.....
Kernel command line: run alarm-multiple
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 674,201,600 loops/s.
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) Thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
```

Pintos permette di modificare la configurazione del Kernel o del virtual HW attraverso dei parametri passati da riga di comando preceduti da "--". Per visualizzare l'elenco delle opzioni è sufficiente lanciare il comando `pintos` senza nessun parametro.

```
max@max-VirtualBox:~/os/bin$ pintos
pintos, a utility for running Pintos in a simulator
Usage: pintos [OPTION...] -- [ARGUMENT...]
where each OPTION is one of the following options
and each ARGUMENT is passed to Pintos kernel verbatim.
Simulator selection:
  -b, --bochs          (default) Use Bochs as simulator
  -q, --qemu           Use QEMU as simulator
  -p, --player         Use VMware Player as simulator
Debugger selection:
  --no-debug          (default) No debugger
  --monitor           Debug with simulator's monitor
  -g, --gdb           Debug with gdb
Display options: (default is both VGA and serial)
  -v, --no-vga        No VGA display or keyboard
  -s, --no-serial      No serial input or output
  -t, --terminal      Display VGA in terminal (Bochs only)
```

3.2 Debugging tools

Durante la scrittura delle nuove funzionalità di Pintos poter debuggare il proprio software ricopre un ruolo fondamentale. Gli strumenti messi disposizione sono:

- **printf()**: la funziona di stampa su standard output, può essere richiamata da qualsiasi punto del Kernel. Una strategia che si può adottare è quello di mettere in diverse parti del codice dei `printf()` per segnalare le parti di codice raggiunte.
- **ASSERT**: l'uso delle assert permette di intercettare rapidamente i problemi. Il loro uso è finalizzato a verificare il valore dei parametri di ingresso di una funzione o degli invarianti, mandando il Kernel in panic nel caso in cui la condizione restituisca false. Il messaggio d'errore include la posizione nel codice dell'asserzione che è fallita e il backtrace delle funzioni che sono state richiamate prima del panic.
- **MACRO**: sono presenti anche altre macro che sono funzionali alla soppressione dei warning generati dal compilatore:
 - o **UNUSED**: sopprime il warning relativo ad eventuali parametri non utilizzati in una funzione.
 - o **NO_RETURN**: sopprime il warning relativo ad eventuali funzioni che non restituiscono mai un valore.
 - o **NO_INLINE**: sopprime il warning relativo alla mancanza della funzione in linea.
 - o **PRINTF_FORMAT**: indica al compilatore che la funzione accetta come parametro di input una stringa nel formato `printf()`, permettendogli di controllare la validità del tipo dei parametri successivi.
- **BACKTRACE**: quando il Kernel va in panico viene stampata una backtrace che include il punto dove si trovava il programma quando si è verificato e di come è arrivato a quel punto (prendendo le informazioni dallo stack), cioè un elenco di indirizzi (esadecimali) delle funzioni che erano running quando il Kernel è andato in panic. Per tradurre gli indirizzi nel numero di riga si può usare una funzione chiamata `backtrace`, nella quale vengono passati come parametri il binario del Kernel (`kernel.o`) e gli indirizzi. La funzione restituisce il nome della funzione dove si trova l'indirizzo e i numeri di riga di ogni indirizzo passato.
- **GDB**: è possibile eseguire Pintos in modalità supervisionata da GDB attraverso l'opzione `-gdb`, e aprendo un secondo terminale su cui eseguire `pintos-gdb kernel.o`, dopodiché eseguire il comando `target remote localhost:1234`. GDB è un tool di debug e reverse engineering che permette di definire breakpoint e controllare l'avanzamento del programma verificando ad ogni step lo stato dei registri e della memoria (variabili).

Inoltre, tra le opportunità messe a disposizione dal comando `pintos` a supporto delle attività di debug, si possono usare le seguenti opzioni:

- **-j**: permette di definire delle interruzioni a intervalli di tempo fissi e riproducibili per poter effettuare attività di debug.
- **-r**: permette di eseguire Pintos in una modalità più realistica per quanto riguarda le interruzioni temporali.

3.3 Development tools

Per sviluppare il codice di Pintos si possono sfruttare i seguenti tool:

- **tags**: All'interno del file *Makefile*, possono essere definiti dei tags da utilizzare per ricercare più rapidamente all'interno del codice variabili globali o funzioni. Gli editor di testo utilizzati comunemente in ambiente unix (vi o emacs) permettono di effettuare ricerche sui tag.
- **cscope**: è un programma che fornisce un indice delle funzioni dichiarate in un programma, cercando tutti i punti dove una certa funzione viene chiamata. La dichiarazione degli indici avviene all'interno del file *Makefile*: attraverso i comandi *makefile cscope* e *cscope* è possibile navigare il codice.
- **git**: è il sistema di controllo del codice che permette di gestire il versioning del codice di Pintos, mantenendo traccia dei cambiamenti e permettendo di coordinare le attività di sviluppo tra più persone.

3.4 Codice di Pintos

In questo paragrafo viene descritta l'organizzazione delle principali componenti del software di Pintos.

Il codice di Pintos è scritto in C seguendo il GNU coding standard, per facilitarne la lettura e l'aggiornamento nel corso del tempo (rif. <http://www.gnu.org/prep/standards/toc.html>)

Tra le librerie del C incluse nel codice sorgente, sono state deliberatamente escluse alcune funzioni per la gestione delle stringhe notoriamente pericolose (soggette ad esempio ad attacchi di tipo format string), quali ad esempio: strcpy, strncpy, strcat, strncat, strtok, sprintf, vsprintf.

3.4.1 Loader

Il codice del loader si trova all'interno del file *threads/loader.S*: è scritto in assembler e viene richiamato direttamente dal bios che lo carica dal MBR – Master Boot Record. Quando viene trovata una partizione bootable, all'inizio della partizione si trova il Kernel. Il suo scopo è quello di caricare il Kernel in memoria, estrarre l'informazione dell'*entry point* dall'intestazione del file del Kernel, dopodiché cedere il controllo al Kernel stesso. Il Kernel di Pintos non può essere superiore a 512KB.

La entry point del Kernel di Pintos è la funzione *start()*, che si trova sotto *threads/start.S* e si occupa di impostare la cpu a 32-bit in protected mode e di chiedere al bios la dimensione della RAM (MAX 64MB), impostando questo valore all'interno della variabile globale *init_ram_pages*.

L'indirizzo iniziale della tabella delle pagine (0) è associato all'indirizzo fisico *LOADER_PHYS_BASE*, pari a 0xc000 0000 (3GB). Il Kernel viene mappato all'indirizzo 0xc002 0000, all'interno del range gestito dalla page table.

Una volta attivata la page table, vengono impostati i registri di controllo della CPU per l'attivazione della modalità protetta, la paginazione e i registri di segmento.

Gli *interrupt in modalità protetta* non sono ancora attivi, successivamente viene chiamata la funzione *main()* che rappresenta l'avvio vero e proprio del Kernel.

Il main si occupa di richiamare altri moduli di inizializzazione di Pintos (terminano con *_ini()*), il cui contenuto è scritto all'interno del file *module.c* e relativo header *module.h*:

- **bss_init()**: BSS rappresenta un segmento di memoria che viene inizializzato a tutti zero, usato dal C quando si dichiara una variabile senza inizializzazione in modo che assuma di default il valore zero.

- `read_command_line()`: si occupa di fare il parsing degli argomenti passati al Kernel, leggendo le opzioni all'inizio della riga di comando e i comandi da eseguire alla fine.
- `thread_init()`: inizializza il sistema creando una struttura per i thread e il sistema di lock acquire/release. Inizializza la console e stampa il messaggio iniziale.
- `pallocc_init()`: inizializza l'allocatore del Kernel, restituisce una o più pagine alla volta.
- `mallocc_init()`: inizializza l'allocatore di blocchi di memoria in dimensione arbitraria.
- `paging_init()`: imposta una tabella delle pagine per il Kernel .
- `intr_init()`: inizializza la interrupt descriptor Table.
- `timer_init()`: inizializza l'handler per la gestione degli interrupt temporizzati.
- `kbd_init()`: inizializza l'handler per la gestione degli interrupt dalla tastiera.
- `input_init()`: fonde i segnali di input della seriale e della tastiera.

Dopo l'inizializzazione della gestione degli interrupt viene chiamata la funzione di inizializzazione dei thread, `thread_start()`, che crea il thread "idle" e abilita le interruzioni.

Da questo momento sono possibili delle interruzioni per segnali che arrivano dalle porte seriali di I/O, per questo viene richiamata la funzione `serial_init_queue()`. La funzione `timer_calibrate()`, calibra il funzionamento del timer per la gestione dei delay brevi.

Alla fine del boot viene stampato un messaggio d'errore e viene richiamata la funzione `run_actions()`, che esegue le azioni indicate nella command line che ha richiamato il Kernel. Se tra le opzioni del Kernel viene indicato anche -q successivamente all'esecuzione di `run_actions()` viene richiamata anche la funzione `shutdown_power_off()`, che spegne la macchina, diversamente la funzione `main()` chiamerà la `thread_exit()` che permette l'esecuzione di qualsiasi altro thread nella coda dei running.

3.4.2 Mappa della memoria

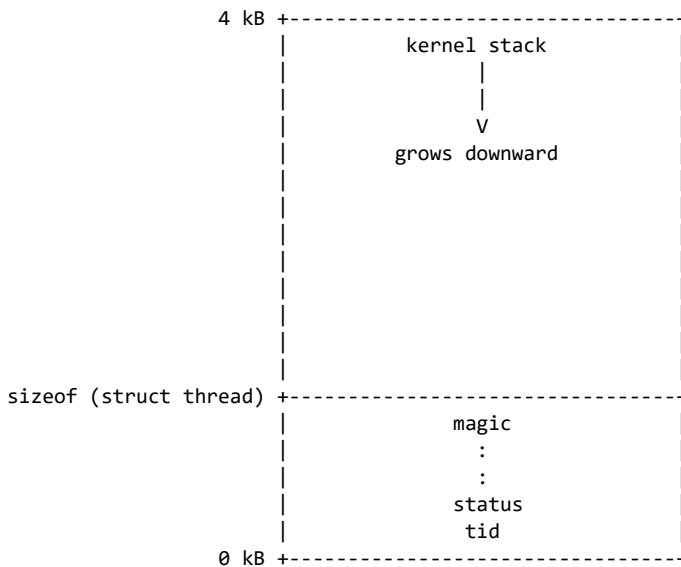
La memoria di Pintos complessivamente indirizzabile è pari a 64MB e può essere rappresentata in questo modo:

RANGE	OWNER	DESCRIZIONE
00000000--000003ff	CPU	Real mode interrupt table
00000400--000005ff	BIOS	Miscellaneous data area
00000600--00007bfff	--	--
00007c00--00007dfff	Pintos	Loader
0000e000--0000effff	Pintos	Stack for loader; kernel stack and struct thread for initial kernel thread
0000f000--0000ffff	Pintos	Page directory for startup code
00010000--00020000	Pintos	Page tables for startup code.
00020000--0009ffff	Pintos	Kernel code, data, and uninitialized data segments
000a0000--000bffff	Video	VGA display memory.
000c0000--000effff	Hardware	Reserved for expansion card RAM and ROM
000f0000--000fffff	BIOS	ROM BIOS
00100000--03ffffff	Pintos	Dynamic memory allocation

3.4.3 Thread

Il Kernel di Pintos rappresenta i thread attraverso la data structure `struct thread` dichiarata all'interno del file `threads/thread.h`. Questa struttura rappresenta un thread o uno user process.

Le struct thread sono poste all'inizio della propria pagina di memoria, mentre il resto della pagina viene utilizzato per lo stack del thread che cresce verso il basso verso il fondo della pagina



Questa scelta architetturale comporta che la struct thread non possa essere troppo grande e lo stack non possa crescere troppo, perché comprometterebbe lo stato del thread: di conseguenza, [le funzioni del kernel non dovrebbero MAI allocare strutture di grandi dimensioni](#) come variabili locali non statiche, ma utilizzare l'allocazione dinamica attraverso malloc() o palloc_get_page().

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid; /* Thread identifier. */
    enum thread_status status; /* Thread state. */
    char name[16]; /* Name (for debugging purposes). */
    uint8_t *stack; /* Saved stack pointer. */
    int priority; /* Priority. */
    struct list_elem allelem; /* List element for all threads list. */
    /* Shared between thread.c and synch.c. */
    struct list_elem elem; /* List element. */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir; /* Page directory. */
#endif
    /* Owned by thread.c. */
    unsigned magic; /* Detects stack overflow. */
};
```

A seguire una descrizione dei principali campi della struct thread:

- **tid**: Thread ID (univoco)
- **status**: descrive lo stato del thread e può assumere i seguenti valori:
 - o THREAD_RUNNING running
 - o THREAD_READY non running ma pronti a essere eseguiti
 - o THREAD_BLOCKED in attesa di qualche evento
 - o THREAD_DYING in fase di distruzione

- **name**: nome del processo/thread.
- ***stack**: ogni thread ha il proprio puntatore allo stack.
- **priority**: assume valori tra PRI_MIN (0) e PRI_MAX (63), nella versione iniziale di Pintos questo campo è ignorato durante lo scheduling dei processi.
- **Allelem**: collega il thread alla lista di tutti i thread, in modo che possa essere usato per iterare dalla *thread_foreach()*.
- **Elem**: lista doppio linkata di elementi che viene usata per i thread ready (pronti ad essere eseguiti) e quelli in wait su un semaforo *sema_down()*.
- ***pagedir**: puntatore alla process Page Table. (controllato dal flag USERPROG).
- **magic**: viene impostato sempre a THREAD_MAGIC e serve per identificare situazioni di buffer overflow visto che il thread frame e lo stack risiedono nella stessa pagina.

All'interno del file *threads/thread.c* sono implementate diverse funzioni pubbliche a supporto dei thread. Le principali sono:

- **void thread_init(void)**: inizializza il thread, creando la struct thread del thread iniziale di Pintos.
- **void thread_start(void)**: richiamata dal main() per avviare lo scheduler, crea il thread di idle, abilita gli interrupt, e quindi anche lo scheduler che viene richiamato dal "time interrupt".
- **void thread_tick(void)**: viene richiamato dal timer interrupt ad ogni "tick", usato per le statistiche dei thread. Attiva lo scheduler in modo ricorrente.
- **void thread_print_stats(void)**: richiamato durante lo shutdown di Pintos, stampa le statistiche dei thread
- **tid_t thread_create(const char *name, int priority, thread_func *func, void *aux)**: crea un nuovo thread e restituisce il tid. Riceve come parametri di ingresso il nome del thread, la sua priorità e la funzione che deve eseguire. Alloca una pagina per la struct thread e lo stack. Il thread è inizializzato in stato di blocco, verrà "sbloccato" appena prima del ritorno.
- **void thread_block(void)**: mette il thread corrente in stato di Block, il thread non verrà più eseguito finché non verrà richiamata la *thread_unblock()*.
- **void thread_unblock(struct thread *thread)**: sblocca un thread che precedentemente era stato bloccato spostandolo nella coda dei ready. Questo meccanismo viene usato quando un evento che il thread stava attendendo si è verificato.
- **struct thread *thread_current(void)**: restituisce il thread attuale.
- **tid_t thread_tid(void)**: restituisce il tid (thread ID) del thread corrente. Equivale a *thread_current()->tid*
- **const char *thread_name (void)**: restituisce il nome del thread corrente. Equivale a *thread_current()->name*
- **void thread_exit (void)**: determina l'uscita dal thread senza fornire alcun ritorno.
- **void thread_yield (void)**: rilascia la CPU allo scheduler, che sceglie un altro thread dalla lista dei ready. Il thread potrebbe anche essere nuovamente quello corrente.
- **void thread_foreach (thread_action_func *action, void *aux)**: itera su tutti i thread eseguendo l'azione *action()* su ognuno dei thread.

3.4.4 Thread switch

La funzione *schedule()* p è responsabile del cambio del thread in esecuzione.

È una funzione definita in *threads/thread.c* e viene chiamata dalle tre funzioni pubbliche che possono modificare il thread in esecuzione:

- *thread_block()*
- *thread_exit()*
- *thread_yield()*

Prima di richiamare `schedule()`, queste funzioni [disabilitano gli interrupt](#).

Le azioni svolte da `schedule()` sono:

- registrare nella variabile `cur` il thread corrente.
- esegue [next_thread_to_run\(\)](#) per determinare il prossimo thread da eseguire mettendolo nella variabile `next`.
- chiama [switch_threads\(\)](#), per effettuare lo switch vero e proprio.

[Switch_threads\(\)](#) è scritta in assembly all'interno del file `threads_switch.S` e si occupa di:

- Salvare i registri nello stack
- Salvare lo stack pointer nell'apposito campo della struct thread
- Salvare nel registro stack pointer il valore del nuovo thread
- Ripristinare i valori dei registri del nuovo thread recuperandoli dal suo stack.

A seguire viene richiamata la funzione [thread_schedule_tail\(\)](#), che si occupa di segnare il nuovo thread come "running" se il thread precedente era in stato di dying e liberare la pagina della struct thread e dello stack del processo precedente.

Quando un thread viene [eseguito per la prima volta](#), dopo che `thread_create()` ha creato un nuovo thread, per consentirne l'avvio corretto è necessario creare tre "false stack" che possano essere richiamati dalle funzioni `switch_threads()`, `switch_entry()` e `kernel_thread()`.

3.4.5 Sincronizzazione

La condivisione di risorse tra thread deve essere gestita attentamente per non generare comportamenti inattesi e difficilmente riproducibili. Pintos offre diverse primitive di sincronizzazione e diversi meccanismi a garanzia della sincronizzazione tra i diversi threads.

Disabilitazione interrupt

Rappresentano il modo più semplice per sincronizzare diversi thread, di fatto impedendo che in un certo periodo di tempo il thread corrente possa essere interrotto nella sua esecuzione.

Questo meccanismo è molto grezzo e non dovrebbe essere utilizzato, specialmente nelle architetture multi core dove, in ogni momento, sono in esecuzione più thread anche se vengono disabilitati gli interrupt. Inoltre questo approccio sarebbe inefficace quando si verifica un interrupt non mascherabile (NMI).

Per disabilitare le interruzioni si usano le funzioni definite all'interno di [threads/interrupt.h](#):

- [enum intr_level intr_get_level \(void\)](#): restituisce lo stato corrente degli interrupt (INTR_OFF o INTR_ON)
- [enum intr_level intr_set_level \(enum intr_level level\)](#): attiva e disattiva gli interrupt e restituisce lo stato precedente.
- [enum intr_level intr_enable \(void\)](#): attiva gli interrupt e restituisce lo stato precedente
- [enum intr_level intr_disable \(void\)](#): disattiva gli interrupt e restituisce lo stato precedente

Semafori

I semafori sono dei sistemi di sincronizzazione che permettono di garantire l'accesso controllato a risorse condivise. Il funzionamento di questo strumento è collaborativo e richiede la presenza di più thread che cooperano tra loro.

Il numero delle risorse è rappresentato da un numero intero aggiornato atomicamente da due funzioni:

- `P()` – rappresenta la wait, cioè l'attesa che il valore del semaforo torni ad essere positivo.
- `V()` – rappresenta la signal, ovvero l'incremento del valore del semaforo e comporta il risveglio di uno dei thread in attesa.

In Pintos i semafori sono descritti all'interno del file [thread/synch.h](#).

```
struct semaphore
{
    unsigned value;           valore attuale del semaforo
    struct list waiters;      Lista dei thread in attesa
};
```

Le principali funzioni relative ai semafori sono:

- `void sema_init (struct semaphore *sema, unsigned value)`: inizializza il semaforo passato per reference.
- `void sema_down (struct semaphore *sema)`: decrementa il valore del semaforo passato per reference, aspettando se il valore non è positivo.
- `bool sema_try_down (struct semaphore *sema)`: prova a decrementare il valore del semaforo senza attendere, restituendo true se il decremento è andato a buon fine, false se il valore del semaforo è già zero. Di fatto questa funzione inserita all'interno di un loop utilizza un approccio di tipo busy wait, cioè con attesa che consuma cicli di CPU e andrebbe utilizzato solo quando si ha la ragionevole certezza che l'attesa sarà breve
- `void sema_up (struct semaphore *sema)`: incrementa il valore del semaforo. Se ci sono dei thread in attesa per questo semaforo viene svegliato uno di loro.

I semafori di Pintos sono costruiti per disabilitare gli interrupt e bloccare e sbloccare i thread attraverso le funzioni `thread_block()` e `thread_unblock()`. La lista dei thread in attesa è gestita attraverso una linked list.

Locks

I lock sono dei tipi particolari di Semaforo con valore iniziale pari a 1. Le funzioni up e down vengono chiamate `release()` e `acquire()`.

Questo tipo di struttura viene utilizzata per garantire l'accesso esclusivo (un thread alla volta) ad una determinata risorsa. In questo senso i lock introducono il concetto di Owner, ovvero il lock è posseduto da un thread per volta e nessun altro ne può acquisire il possesso finché lo stesso thread non lo rilascia.

I lock di Pintos non sono ricorsivi, quindi lo stesso thread non può acquisire più volte lo stesso lock (anche se ne è già il possessore).

I locks, come gli altri meccanismi di sincronizzazione sono descritti all'interno del file [thread/synch.h](#).

```
struct lock
{
    struct thread *holder;      thread che possiede il lock
    struct semaphore semaphore; semaforo (binario => max 1)
};
```

Le funzioni principali per la gestione dei lock sono:

- `void lock_init (struct lock *lock)`: inizializza il lock, non necessariamente il thread owner.
- `void lock_acquire (struct lock *lock)`: acquisisce il lock da parte del thread corrente, se il lock è già posseduto da un altro thread attende senza consumare CPU.

- `bool lock_try_acquire (struct lock *lock)`: prova ad acquisire il lock, senza attendere. Restituisce true se è riuscito ad acquisire il lock e false se il lock è già posseduto da un altro thread. Inserendo questa funzione all'interno di un loop si genera un busy waiting, cioè un'attesa che consuma cicli di CPU, da evitare se non si hanno garanzie che l'attesa sia breve.
- `void lock_release (struct lock *lock)`: il lock viene rilasciato da parte del thread che lo possiede in questo momento.
- `bool lock_held_by_current_thread (const struct lock *lock)`: funzione per verificare se il thread corrente possiede il lock oppure no.

Monitors

I monitor sono delle forme di sincronizzazione più complesse dei normali semafori e dei lock e sono costituiti da:

- dati da sincronizzare
- lock
- e una o più condition variable

Prima di accedere all'area dei dati protetti, il thread deve prima acquisire il lock del monitor. Una volta ottenuto il lock, il thread può accedere in lettura e scrittura ai dati di sincronizzazione.

Ogni condition variable è associata al verificarsi di un evento, i thread che acquisiscono il lock possono quindi mettersi in attesa del verificarsi di una delle condizioni controllate dalle condition variable.

Quando la condizione si verifica, il thread che ha determinato il cambiamento risveglia uno o tutti i thread in attesa per quella condizione, quindi rilascia il lock.

Le condition variable sono definite all'interno del file `threads/sync.h`.

```
struct condition
{
    struct list waiters;           Lista dei thread in attesa
};
```

mentre i monitor non sono definiti nella versione iniziale di Pintos.

Le condition variable sono sempre associate a un LOCK che ne garantisce l'accesso in modalità esclusiva. Le funzioni principali sono:

- `void cond_init (struct condition *cond)`: crea la condition variable.
- `void cond_wait (struct condition *cond, struct lock *lock)`: il thread si mette in attesa dell'evento controllato dalla condition variable.
- `void cond_signal (struct condition *cond, struct lock *lock)`: risveglia un thread in attesa del verificarsi dell'evento rappresentato dalla condition variable. Il lock serve solo per "simmetria" con la funzione di wait, ma non è usato all'interno del codice. Per la signal non è necessario acquisire il lock.
- `void cond_broadcast (struct condition *cond, struct lock *lock)`: risveglia tutti i thread in attesa sulla condition variable.

3.4.6 Allocazione di memoria

Pintos utilizza due diversi allocatori di memoria, uno per pagine, l'altro per blocchi di dimensione arbitraria.

Page Allocator

L'allocatore per pagine è dichiarato all'interno del file `threads/palloc.h` permette di allocare la memoria in unità di pagine, la maggior parte delle volte viene utilizzato per allocare una pagina per volta, ma permette anche di allocare più pagine in modo contiguo.

Questo allocatore divide la memoria in due pool differenti chiamati: `Kernel pool` e `User pool`. Di default, ogni pool, ha dimensione pari alla metà della memoria disponibile superiore a 1MB. Le richieste di allocazione possono attingere da un pool o dall'altro, non esiste un vincolo stringente, ma i processi utente dovrebbero attingere dallo user pool, mentre il Kernel dal Kernel pool.

Le pagine allocate da entrambi i pool vengono tracciate su una `bitmap`, che viene scansionata quando vengono richieste allocazioni multiple di più pagine per trovare un'area contigua. La strategia adottata è di tipo `first fit`.

Questo tipo di approccio porta a frammentazione esterna, il che potrebbe rendere impossibile allocare il numero di pagine contigue richieste, mentre le richieste di singole pagine non possono fallire per problemi di frammentazione.

Quando una pagina viene liberata, tutti i suoi bite vengono sovrascritti con il valore `0xcc` per permettere di identificare più facilmente l'utilizzo di puntatori a pagine che sono già state liberate (dangling pointer) durante le fasi di debug.

Le funzioni del page allocator sono:

- `void *palloc_get_page (enum palloc_flags flags)`: permette di allocare una pagina di memoria
- `void *palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)`: permette di richiedere l'allocazione di `page_cnt` pagine di memoria contigue.
- `void palloc_free_page (void *page)`: libera una pagina di memoria.
- `void palloc_free_multiple (void *pages, size_t page_cnt)`: libera N pagine di memoria.

L'enumeratore passato come parametro `palloc_flag`, può assumere i seguenti valori:

- `PAL_ASSERT`: se le pagine non possono essere allocate, il Kernel va in panic. Usato durante l'inizializzazione del Kernel, non dovrebbe essere usata dagli user process
- `PAL_ZERO`: prima di restituire la pagina questa viene inizializzata tutta a zero, altrimenti il contenuto delle pagine è imprevedibile.
- `PAL_USER`: ottiene le pagine dal pool utente. Se non viene impostato, tutte le pagine vengono allocate dal pool del Kernel.

```
struct pool
{
    struct lock lock;           Lock per l'accesso esclusivo
    struct bitmap *used_map;    bitmap delle pagine usate
    uint8_t *base;             puntatore alla base del pool
};
```

L'allocatore di memoria è abbastanza semplice e nella configurazione iniziale non prevede di mantenere traccia del thread che ha richiesto l'allocazione di una determinata pagina, ma solo il fatto che la pagina sia stata assegnata. Se il thread non rilascia la memoria questa rimane occupata fino al riavvio del Kernel.

```
void *
palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)
{
```

```

struct pool *pool = flags & PAL_USER ? &user_pool : &kernel_pool;
void *pages;
size_t page_idx;
if (page_cnt == 0)
    return NULL;
lock_acquire (&pool->lock);
page_idx = bitmap_scan_and_flip (pool->used_map, 0, page_cnt, false);
lock_release (&pool->lock);

if (page_idx != BITMAP_ERROR)
    pages = pool->base + PGSIZE * page_idx;
else
    pages = NULL;
if (pages != NULL)
{
    if (flags & PAL_ZERO)
        memset (pages, 0, PGSIZE * page_cnt);
}
else
{
    if (flags & PAL_ASSERT)
        PANIC ("palloc_get: out of pages");
}
return pages;
}

```

Block Allocator

L'allocatore per blocchi è dichiarato all'interno di `threads/malloc.h`, e permette di allocare memoria di qualsiasi dimensione. È costruito come uno strato superiore a quello dell'allocatore per pagine usando le pagine del pool del kernel.

Il block allocator adotta due strategie differenti:

- **Blocchi $\leq 1\text{KB}$** : vengono arrotondati alla potenza di 2 superiore più vicina e raggruppati in una pagina dedicata all'allocazione di oggetti di quella dimensione (es. 512Byte).
- **Blocchi $> 1\text{KB}$** : in questo caso l'arrotondamento viene fatto aggiungendo un piccolo overhead alla dimensione richiesta e cercando il numero di pagine superiore più vicino. L'allocatore chiederà al page allocator di allocare pagine contigue.

In entrambi i casi la differenza tra lo spazio richiesto per l'allocazione e la dimensione reale del blocco viene sprecata, generando frammentazione interna. Tuttavia, considerando che la maggior parte delle richieste di allocazione per pagina avviene per variabili di dimensione minore a quella della pagina stessa, questo approccio permette di ridurre il numero di pagine realmente utilizzate.

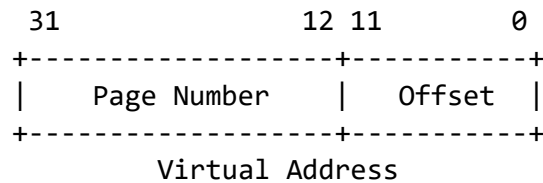
Anche in questo caso, quando un blocco viene liberato, tutti i byte vengono sovrascritti con il valore 0xcc, per permettere di identificare più semplicemente in fase di debug la presenza di dangling pointers.

Le funzioni usate dall'allocatore hanno lo stesso nome e interfaccia delle funzioni standard del C:

- void *malloc (size_t size)
- void *calloc (size_t a, size_t b)
- void *realloc (void *block, size_t new_size)
- void free (void *block)

3.4.7 Indirizzamento virtuale

Lo spazio di indirizzamento virtuale a 32 bit viene diviso in 20 bit per il numero di pagina, 12 bit per l'offset.



La memoria virtuale in Pintos è divisa in due regioni:

- User virtual memory
- Kernel virtual memory

All'interno del file threads/vaddr.h sono definite le funzioni e le macro a supporto della memoria virtuale.

Le macro:

- PGSHIFT pari a 0 indica il primo bit dell'offset
- PGBITS pari a 12 rappresenta il numero di bit dell'offset
- PGMASK rappresenta la Maschera per ottenere il numero della pagina
- PGSIZE dimensione della pagina $2^{12} = 4\text{KByte}$

Le funzioni:

- unsigned pg_ofs (const void *va): restituisce l'offset da un indirizzo virtuale.
- uintptr_t pg_no (const void *va): restituisce il numero di pagina da un indirizzo virtuale.
- void *pg_round_down (const void *va): restituisce l'indirizzo iniziale della pagina di un indirizzo virtuale.
- void *pg_round_up (const void *va): restituisce l'indirizzo iniziale della pagina successiva di un certo indirizzo virtuale.

Il confine tra queste due aree è definita dalla macro PHYS_BASE, Il cui valore di default è (3GB) 0xc000 0000, potrebbe essere cambiata a qualsiasi multiplo di un 1GB. Il Kernel virtuale parte da zero fino a PHY_BASE, mentre la memoria virtuale utente è da 3GB fino alla fine dello spazio di indirizzamento a 32 BIT (4GB).

Per distinguere un indirizzo di memoria virtuale Kernel da uno user sono state definite due funzioni:

- bool is_user_vaddr (const void *va)
- bool is_kernel_vaddr (const void *va)

Considerando che il processore 80x86, non fornisce meccanismi per accedere direttamente alla memoria dato un indirizzo fisico, Pintos aggira questo problema mappando la memoria virtuale del kernel uno a uno con la memoria fisica usando un "relocator register".

L'indirizzo della memoria fisica è pari a PHYS_BASE + l'indirizzo della memoria virtuale.

Esistono due funzioni che permettono di passare da un indirizzo virtuale a uno fisico del Kernel e viceversa:

- void *ptov (uintptr_t pa)
- uintptr_t vtop (void *va)

3.4.8 Page Table

Il file [pagedir.c](#), contiene l'interfaccia per la hardware page table del 80x86, chiamata [page directory](#).

Per creare, distruggere e attivare la page table sono definite tre funzioni. La versione base del codice di Pintos utilizza già queste funzioni dove necessario:

- uint32_t *pagedir_create (void)
- void pagedir_destroy (uint32_t *pd)
- void pagedir_activate (uint32_t *pd)

mentre usano o aggiornano il mapping tra pagina e frame all'interno della page table, flushando la TLB se necessario:

- `bool pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool writable)`: aggiunge alla page directory il mapping tra user page e la frame identificata dal kernel virtual address page, setta anche il flag di lettura/scrittura o solo lettura.
- `void *pagedir_get_page (uint32_t *pd, const void *uaddr)`: permette di cercare la frame associata ad un determinato “user address” nella page directory. Se esiste restituisce l’indirizzo, altrimenti null.
- `void pagedir_clear_page (uint32_t *pd, void *page)`: imposta la pagina come “non presente” nella page directory, gli accessi successivi falliranno.

L'hardware del 80x86 fornisce supporto nell'implementazione degli algoritmi di replacement attraverso due bit associati alle page table entry. Per ogni pagina acceduta viene settato l'**accessed bit**, mentre ad ogni scrittura viene settato il **dirty bit**. La CPU non resetta mai i valori del accessed bit o del dirty bit, ma il sistema operativo può farlo attraverso un set di funzioni:

- ```
- bool pagedir_is_dirty (uint32_t *pd, const void *page)
- bool pagedir_is_accessed (uint32_t *pd, const void *page)
- void pagedir_set_dirty (uint32_t *pd, const void *page, bool value)
- void pagedir_set_accessed (uint32_t *pd, const void *page, bool value)
```

```

31 12 11 9 6 5 2 1 0
+-----+-----+-----+-----+-----+-----+-----+
| Physical Address | AVL | |D|A| |U|W|P|
+-----+-----+-----+-----+-----+-----+-----+

```

Per creare una entry all'interno della page table sia nello spazio utente che Kernel o per accedere al contenuto di una PTE (Page Table Entry) sono disponibili le seguenti funzioni:

- uint32\_t pte\_create\_kernel (uint32\_t \*page, bool writable)
- uint32\_t pte\_create\_user (uint32\_t \*page, bool writable)
- void \*pte\_get\_page (uint32\_t pte)

### 3.4.9 Hash table

Pintos fornisce anche strutture per creare delle hash table all'interno di [lib/kernel/hash.c](#). Queste possono essere usate includendo il file header associato [lib/kernel/hash.h](#).

Nella versione iniziale di Pintos non vengono usate le hash table. L'uso delle hash table è funzionale a rendere più rapida la traduzione di una pagina in una frame.

### 3.4.10 Scheduler

L'obiettivo principale di uno scheduler è quello di bilanciare le diverse esigenze di esecuzione dei thread. I thread che effettuano molto I/O richiedono tempi di risposta rapidi per poter mantenere impegnati i dispositivi di I/O, ma sfruttano pochissimo il tempo di esecuzione della CPU. Per i thread computazionali invece il tempo di CPU è molto più importante, ma non richiedono tempi di risposta rapidi.

L'obiettivo di uno scheduler è quindi quello di cercare di soddisfare entrambe queste necessità e tutte quelle che si collocano nel mezzo tra questi due estremi.

Questo è possibile attraverso un sistema denominato multilevel feedback queue e alla valutazione di informazioni statistiche trascorsi un certo numero di "tick" (intervallo di tempo predefinito)

Questo tipo di schedulatore non è incluso nella versione standard di pintos.

## 4. Modifiche al codice

Dopo aver effettuato il setup di Pintos e studiato un po' più in profondità la sua architettura è giunto il momento di fare qualche piccola modifica al software del Kernel per verificare il funzionamento della VM e la possibilità di modificare il codice del sistema operativo.

### 4.1 Modifica del codice di test

Per questa ragione ho effettuato una piccola modifica file `threads/init.c` introducendo una `printf` all'interno del main, subito dopo che è terminato il boot e giusto prima che venga eseguita l'azione passata come parametro nella command line del Kernel.

```
printf ("Boot complete.\n");

/* Test di modifica del kernel di pintos */
printf("### ARGOMENTO PASSATO DA RIGA DI COMANDO: %s ###\n",*argv);

/* Run actions specified on kernel command line. */
run_actions (argv);

/* Finish up. */
shutdown ();
thread_exit ();
}
```

Per compilare le modifiche è sufficiente posizionarsi nella cartella `/os/pintos/src/threads/` ed eseguire nuovamente il comando `thread`, dopodiché copiare i file binari compilati (in questo caso `kernel.o`, `kernel.bin` e `loader.bin` si trovano all'interno della cartella `./build`), all'interno della cartella `/os/bin/`.

L'esecuzione del comando `pintos run alarm-multiple` restituirà quindi un output leggermente differente

```
max@max-VirtualBox:~/os/bin$./pintos run alarm-multiple
qemu -hda /tmp/SVnPtVAWzh.dsk -m 4 -net none -serial stdio
WARNING: Image format was not specified for '/tmp/SVnPtVAWzh.dsk' and probing guessed raw.
 Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
 Specify the 'raw' format explicitly to remove the restrictions.
PiLo hda1
Loading.....
Kernel command line: run alarm-multiple
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer... 353,484,800 loops/s.
Boot complete.
ARGOMENTO PASSATO DA RIGA DI COMANDO: run
Executing 'alarm-multiple':
(alarm-multiple) begin
```

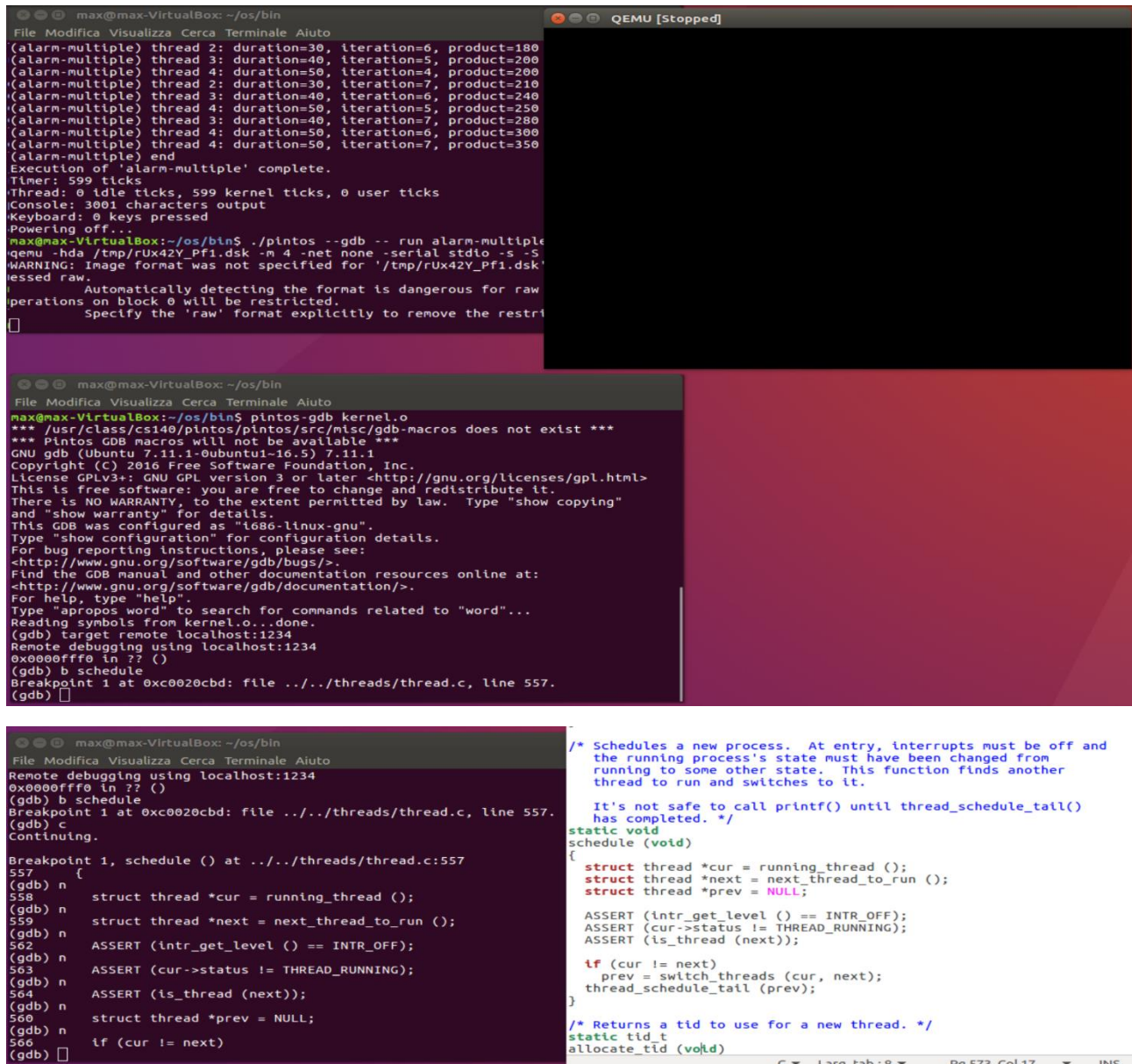


## 4.1 Debug Test

Prima di iniziare con le modifiche vere e proprie testare il funzionamento di GDB. Come indicato nel paragrafo dedicato ai tool di debug è possibile lanciare Pintos in modalità supervisiones. Per farlo è necessario:

- 1) Lanciare Pintos con l'opzione `--gdb`.
- 2) Aprire un secondo terminal nel quale eseguire il comando `pintos-gdb kernel.o`
- 3) Eseguire il comando `target remote localhost:1234`

Oltre a questi step per testare il funzionamento della fase di debug si può mettere un breakpoint sulla funzione `schedule()`, con il comando `b schedule`, seguito da `c` (per continuare l'esecuzione fino al prossimo break point).



```
max@max-VirtualBox: ~/os/bin
File Modifica Visualizza Cerca Terminale Aiuto
(alarm-multiple) thread 2: duration=30, iteration=6, product=180
(alarm-multiple) thread 3: duration=40, iteration=5, product=200
(alarm-multiple) thread 4: duration=50, iteration=4, product=200
(alarm-multiple) thread 2: duration=30, iteration=7, product=210
(alarm-multiple) thread 3: duration=40, iteration=6, product=240
(alarm-multiple) thread 4: duration=50, iteration=5, product=250
(alarm-multiple) thread 3: duration=40, iteration=7, product=280
(alarm-multiple) thread 4: duration=50, iteration=6, product=300
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
(alarm-multiple) end
Execution of 'alarm-multiple' complete.
Timer: 599 ticks
Thread: 0 idle ticks, 599 kernel ticks, 0 user ticks
Console: 3001 characters output
Keyboard: 0 keys pressed
Powering off...
max@max-VirtualBox:~/os/bin$./pintos --gdb -- run alarm-multiple
qemu -hda /tmp/rUx42Y_Pf1.dsk -m 4 -net none -serial stdio -s -S
WARNING: Image format was not specified for '/tmp/rUx42Y_Pf1.dsk'
and 'raw' format is deprecated.
Automatically detecting the format is dangerous for raw
disks, as operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restriction.
max@max-VirtualBox:~/os/bin$ pintos-gdb kernel.o
*** /usr/class/cs140/pintos/pintos/src/misc/gdb-macros does not exist ***
*** Pintos GDB macros will not be available ***
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel.o...done.
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x00000000 in ?? ()
(gdb) b schedule
Breakpoint 1 at 0xc0020cbd: file ../../threads/thread.c, line 557.
(gdb) c
Continuing.
Breakpoint 1, schedule () at ../../threads/thread.c:557
557 {
(gdb) n
558 struct thread *cur = running_thread ();
(gdb) n
559 struct thread *next = next_thread_to_run ();
(gdb) n
562 ASSERT (intr_get_level () == INTR_OFF);
(gdb) n
563 ASSERT (cur->status != THREAD_RUNNING);
(gdb) n
564 ASSERT (is_thread (next));
(gdb) n
560 struct thread *prev = NULL;
(gdb) n
566 if (cur != next)
(gdb)

/* Schedules a new process. At entry, interrupts must be off and
the running process's state must have been changed from
running to some other state. This function finds another
thread to run and switches to it.

It's not safe to call printf() until thread_schedule_tail()
has completed. */
static void
schedule (void)
{
struct thread *cur = running_thread ();
struct thread *next = next_thread_to_run ();
struct thread *prev = NULL;

ASSERT (intr_get_level () == INTR_OFF);
ASSERT (cur->status != THREAD_RUNNING);
ASSERT (is_thread (next));

if (cur != next)
prev = switch_threads (cur, next);
thread_schedule_tail (prev);
}

/* Returns a tid to use for a new thread. */
static tid_t
allocate_tid (void)
```

## 4.2 Analisi delle carenze di Pintos

A questo è necessario individuare quale funzione mancante nella versione attuale di Pintos deve essere sviluppata.

Uno dei limiti strutturali più grandi di Pintos è relativo alla dimensione dello stack associata ad ogni thread. Per ragioni legate al design del sistema operativo, la dimensione massima dello stack è inferiore ai 4KB (dimensione di una pagina – dimensione della struct thread).

La catena delle chiamate che porta all’inizializzazione di un nuovo thread e quindi anche all’assegnazione a questo di uno stack è la seguente:

```
thread_create
|
|-----> palloc_get_page (da sostituire con palloc_get_multiple)
|
|-----> init_thread (non rilevante per questa modifica)
|
|-----> allocate_tid (non rilevante per questa modifica)
|
|-----> intr_disable (non rilevante per questa modifica)
|
|-----> alloc_frame x stackframe (non variazioni, fa riferimento a "t")
|
|-----> alloc_frame x switch_entry (non variazioni, fa riferimento a "t")
|
|-----> alloc_frame x switch thread (non variazioni, fa riferimento a "t")
|
|-----> intr_set_level (non rilevante per questa modifica)
|
|-----> thread_unblock (non rilevante per questa modifica)
```

Tra le altre funzioni dello scheduler che si occupano di allocare/deallocare le pagine di memoria utilizzate dei thread ci sono:

```
thread_schedule_tail
|
|-----> palloc_free_page (da sostituire con palloc_free_multiple)
```

Per controllare la dimensione dello stack in termini di numero di pagine ho introdotto una costante nel file thread.h

```
/* Massimo Mennuni - Luglio 2023
 Define per gestione dimensione stack superiore a 1 pagina
*/
#define STACK_PAGES 4
```

Successivamente ho modificato la thread\_create, sostituendo la chiamata alla funzione palloc\_get\_page con la chiamata palloc\_get\_multiple.

```
tid_t
thread_create (const char *name, int priority,
 thread_func *function, void *aux)
{
 struct thread *t;
 struct kernel_thread_frame *kf;
 struct switch_entry_frame *ef;
 struct switch_threads_frame *sf;
 tid_t tid;
 enum intr_level old_level;

 ASSERT (function != NULL);

 /* Allocate thread. */
 /* Massimo Mennuni - Luglio 2023
 Modifiche gestione stack > di una pagina
 */
 t = palloc_get_multiple (PAL_ZERO, STACK_PAGES);
 // t = palloc_get_page (PAL_ZERO);
```

Simmetricamente ho modificato anche la `thread_schedule_tail`, per consentire la liberazione di tutte le pagine allocate per lo stack.

```
/* If the thread we switched from is dying, destroy its struct
thread. This must happen late so that thread_exit() doesn't
pull out the rug under itself. (We don't free
initial_thread because its memory was not obtained via
palloc().) */
if (prev != NULL && prev->status == THREAD_DYING && prev !=
initial_thread)
{
 ASSERT (prev != cur);
/* Massimo Mennuni Luglio 2023
Modifiche dimensione stack
*/
 palloc_free_multiple (prev, STACK_PAGES);
// palloc_free_page (prev);
}
}
```

Dopo aver verificato attraverso una sessione di debug che ad ogni `thread_create` vengono allocate 4 pagine per lo stack, come ulteriore verifica inserisco una `printf` dell'indice delle bitmap sia nella funzione di allocazione che in quella che le libera

```
void *
palloc_get_multiple (enum palloc_flags flags, size_t page_cnt)
{
 struct pool *pool = flags & PAL_USER ? &user_pool : &kernel_pool;
 void *pages;
 size_t page_idx;

 if (page_cnt == 0)
 return NULL;

 lock_acquire (&pool->lock);
 page_idx = bitmap_scan_and_flip (pool->used_map, 0, page_cnt,
false);
 lock_release (&pool->lock);

 /* Massimo Mennuni Luglio 2023
 stampo l'indice delle pagine per osservare l'evoluzione
 durante l'avvio*/

 printf("### allocated page_idx: %zu ###\n", page_idx);

 return pages;
}

void
palloc_free_multiple (void *pages, size_t page_cnt)
{
 struct pool *pool;
 size_t page_idx;

 ASSERT (pg_ofs (pages) == 0);
 if (pages == NULL || page_cnt == 0)
 return;

 if (page_from_pool (&kernel_pool, pages))
 pool = &kernel_pool;
 else if (page_from_pool (&user_pool, pages))
 pool = &user_pool;
 else
 NOT_REACHED ();

 page_idx = pg_no (pages) - pg_no (pool->base);

 /* Massimo Mennuni - Luglio 2023
 stampo l'indice delle pagine quando il thread viene chiuso*/

 printf("### freed page_idx: %zu ###\n", page_idx);
}
```

L'output sotto mostrato mostra come da un certo momento in avanti, tutte le volte che viene creato un thread

```
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
allocated page_idx: 6
allocated page_idx: 7
allocated page_idx: 8
allocated page_idx: 12
allocated page_idx: 16
allocated page_idx: 20
allocated page_idx: 24
freed page_idx: 8
freed page_idx: 12
freed page_idx: 16
freed page_idx: 20
freed page_idx: 24
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
```

## 5. Conclusioni

La versione Base di Pintos include molte funzioni avanzate se confrontata con la versione iniziale di OS161, ciononostante molte funzionalità di base dei sistemi operativi unix based spesso non sono presenti, tra queste ad esempio:

- Gestione bilanciata della schedulazione dei thread tra esigenze di computazione e I/O
- La possibilità di eseguire programmi utente
- Una gestione della memoria virtuale
- La presenza di un file system
- Gestione delle cache nelle operazioni di I/O
- La presenza di una shell per interagire con il sistema operativo
- Una gestione delle system calls attraverso un handler ed un numero associato ad ogni syscall

Nonostante le differenze è evidente tuttavia la radice comune tra i due sistemi operativi confrontati, dovuta al fatto che entrambi sono sistemi operativi Unix based scritti in C, molte delle scelte architetturali e delle loro realizzazioni sono per questo motivo quasi coincidenti.