

PintOS

Panoramica, Confronto con
OS/161 e Implementazione di
funzionalità aggiuntive

Sofia Longo S310183 – Politecnico di Torino

PintOS

Quadro Generale

Kernel di sistema operativo per l'architettura x86 sviluppato a scopo didattico da Ben Pfaff^[1] nel 2004.

Supporta:

- Interrupt,
- Kernel threads,
- Loading e run di programmi utente,
- RR scheduler,
- File system basilare
- Primitive di sincronizzazione
- Due allocatori di memoria
- Page table

[2]

[1] [The Pintos Instructional Operating System Kernel](#)

[2] [Browsable source code](#)



PintOS

Quadro Generale

A differenza degli altri sistemi operativi didattici, PintOS è in grado di girare sia su hardware fisico che su ambienti di simulazione/emulazione come Bochs e QEMU.

Qui, si predispone una Virtual Machine

- con Ubuntu 16.04;
- si installa QEMU;
- si modificano opportunamente i file di configurazione di PintOS per appoggiarsi su QEMU (default è Bochs).

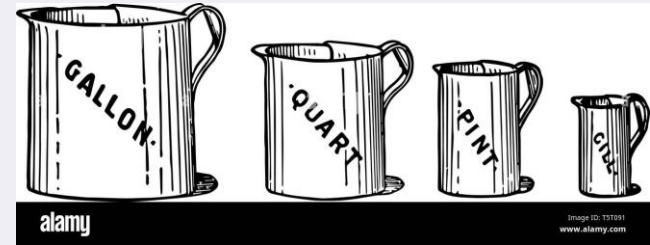


PintOS

Curiosità sul Nome

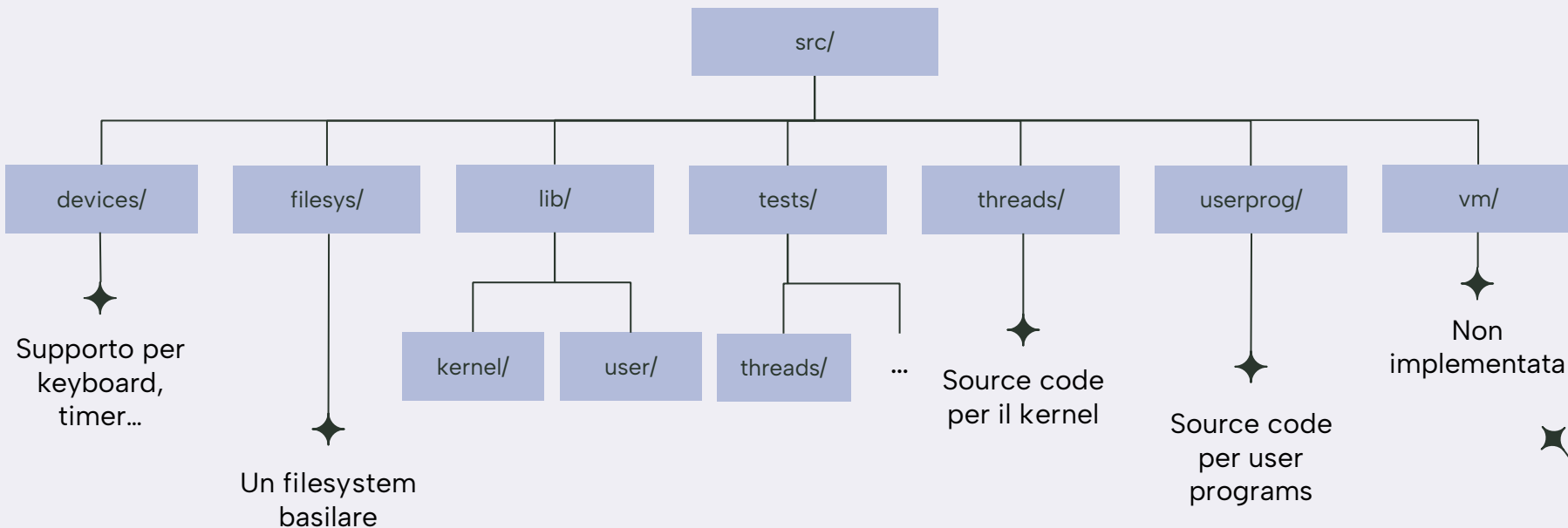
Dalle parole di Ben Pfaff:

- PintOS sostituisce NachOS, e come i nachos, i fagioli Pinto sono una varietà di fagiolo messicano molto comune
- Una pinta è un'unità di misura che caratterizza una piccola quantità
- "Come i guidatori dell'omonima macchina (Ford Pinto – la macchina assassina), anche gli studenti hanno problemi con i blow-up"



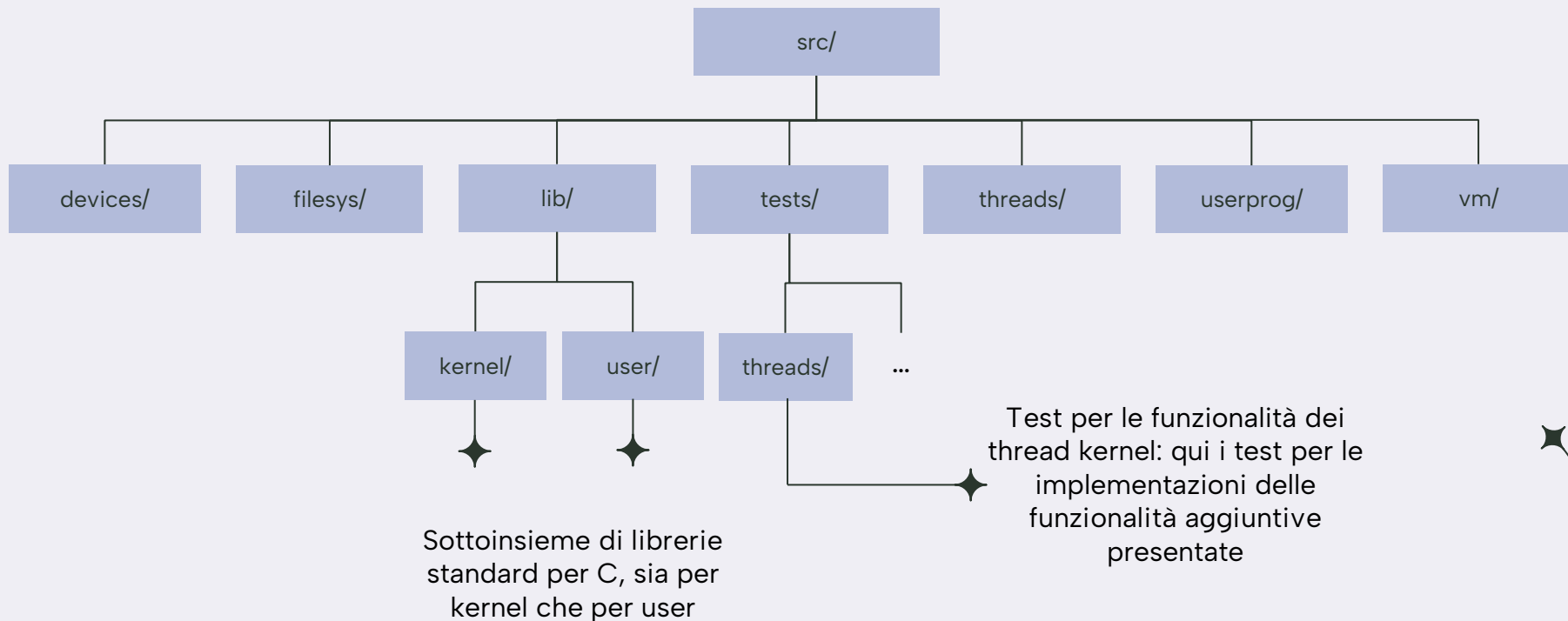


Struttura del codice





Struttura del codice



PintOS

Build + Run

✦ Build PintOS

```
~ pintos/ $ cd src/threads  
~ pintos/src/threads $ make
```

make lancia due comandi: `cd build/` e `make all`

✦ Run PintOS

```
pintos -- run nome-test
```



PintOS

Build + Run

✦ Debug PintOS

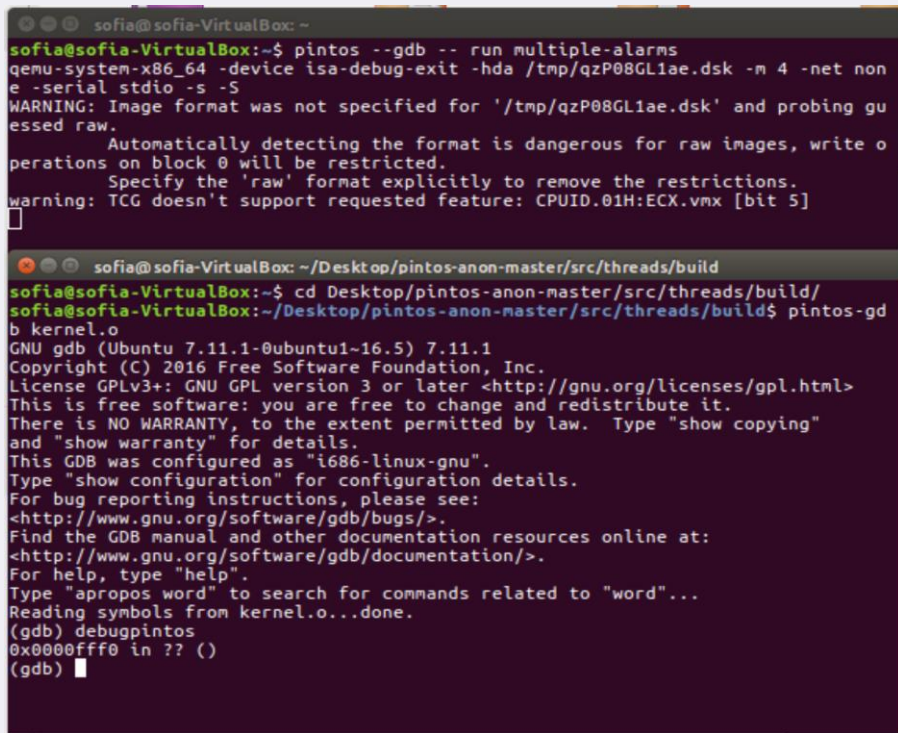
```
pintos --gdb -- run nome-test
```

In un altro terminale

```
cd src/threads/build  
pintos-gdb kernel.o
```

Una volta avviata la sessione di debug

```
debugpintos
```



The image shows two terminal windows from a virtual machine named 'sofia-VirtualBox'. The top window shows the execution of the 'pintos' command with various flags: 'pintos --gdb -- run multiple-alarms qemu-system-x86_64 -device isa-debug-exit -hda /tmp/qzP08GL1ae.dsk -m 4 -net none -serial stdio -s -S'. It displays a warning about the image format and instructions on how to specify the 'raw' format. The bottom window shows the execution of 'pintos-gdb kernel.o' in the directory '/Desktop/pintos-anon-master/src/threads/build'. It displays the GNU gdb version (7.11.1) and the GNU GPL license information.

```
sofia@sofia-VirtualBox: ~  
sofia@sofia-VirtualBox:~$ pintos --gdb -- run multiple-alarms  
qemu-system-x86_64 -device isa-debug-exit -hda /tmp/qzP08GL1ae.dsk -m 4 -net none  
-serial stdio -s -S  
WARNING: Image format was not specified for '/tmp/qzP08GL1ae.dsk' and probing gu  
essed raw.  
Automatically detecting the format is dangerous for raw images, write o  
perations on block 0 will be restricted.  
Specify the 'raw' format explicitly to remove the restrictions.  
warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]  
[  
sofia@sofia-VirtualBox: ~/Desktop/pintos-anon-master/src/threads/build  
sofia@sofia-VirtualBox:~$ cd Desktop/pintos-anon-master/src/threads/build/  
sofia@sofia-VirtualBox:~/Desktop/pintos-anon-master/src/threads/build$ pintos-gd  
b kernel.o  
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) debugpintos  
0x0000fff0 in ?? ()  
(gdb) █
```


PintOS

Build + Run

✦ Debug PintOS

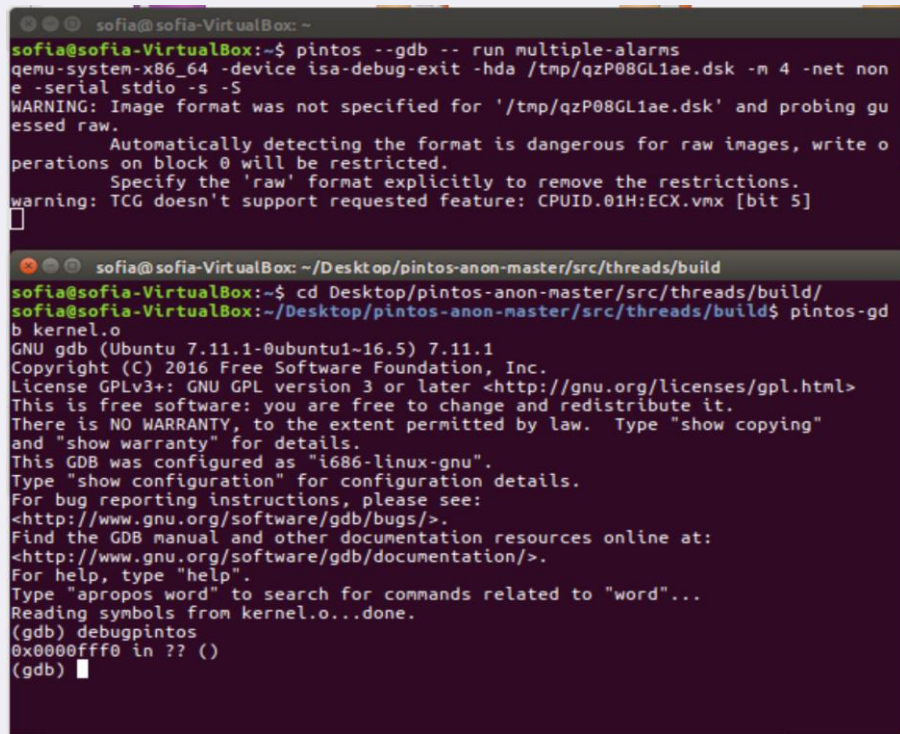
```
pintos --gdb -- run nome-test
```

In un altro terminale

```
cd src/threads/build  
pintos-gdb kernel.o
```

Una volta avviata la sessione di debug

```
debugpintos
```



```
sofia@sofia-VirtualBox: ~  
sofia@sofia-VirtualBox:~$ pintos --gdb -- run multiple-alarms  
qemu-system-x86_64 -device isa-debug-exit -hda /tmp/qzP08GL1ae.dsk -m 4 -net none -serial stdio -s -S  
WARNING: Image format was not specified for '/tmp/qzP08GL1ae.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.  
warning: TCG doesn't support requested feature: CPUID.01H:ECX.vmx [bit 5]  
[  
sofia@sofia-VirtualBox: ~/Desktop/pintos-anon-master/src/threads/build  
sofia@sofia-VirtualBox:~$ cd Desktop/pintos-anon-master/src/threads/build/  
sofia@sofia-VirtualBox:~/Desktop/pintos-anon-master/src/threads/build$ pintos-gdb kernel.o  
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) debugpintos  
0x00000000 in ?? ()  
(gdb) █
```

Ho sviluppato due script per avviare una sessione di debug più velocemente:

```
drun.sh e gdb.sh
```

Due Parti



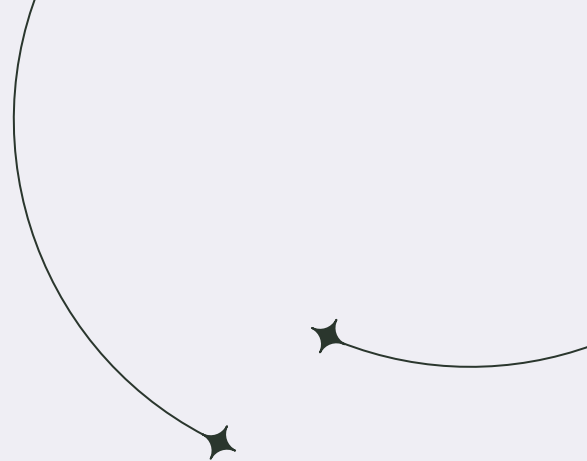
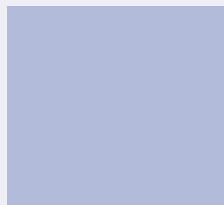
Panoramica di PintOS

- ◆ Moduli del kernel
- ◆ Confronto con OS161



Nuove funzionalità

- ◆ Politica Best Fit nel Page Allocator
- ◆ Priority Scheduler con Priority Shift



Panoramica di PintOS

Moduli del kernel e confronto con OS161

Indice

01 Load + Init

Loading del kernel e
inizializzazione

02 Thread

Supporto per i thread

03 Sincronizzazione

Primitive di sincronizzazione:
interrupt, semafori, lock, monitor,
optimization barrier

04 Allocazione della Memoria

Page allocator e Block allocator

05 Indirizzi Virtuali

Struttura di un virtual address

06 Page Table

Struttura e formato delle entry



01

Load + Init

Load + Init

Nel loading del kernel e sua inizializzazione sono coinvolti 4 file tutti in src/threads/:

1

`load.S`

2

`start.S`



Loading e inizializzazione a basso livello



3

`init.h`

4

`init.c`



Avvio del kernel: inizializzazione ad alto livello



Load + Init

`load.S`

`start.S`

`init.h`

`init.c`

Il loader viene caricato dalla prima sezione dell'hard disk: Master Boot Record

Pintos usa 128B aggiuntivi del MBR per gestire gli argomenti del kernel da cmd.

Rimangono poco più di 300B per il loader, che quindi deve essere scritto in assembly.

L'assembly utilizzato qui è:

Intel x86 Assembly

Utilizzato per il boot – situato nel settore di avvio principale – lungo 512B – contiene:

- Boot loader: `load.S`
- Tabella delle partizioni (64B)
- Signature (2B)

Ora obsoleto – qui si usa ancora

Load + Init

load.S

start.S

init.h

init.c

Il loader viene caricato dalla prima sezione dell'hard disk: Master Boot Record

È il primo file ad essere caricato in memoria:

1. Localizza il kernel leggendo la tabella delle partizioni e cerca una partizione d'avvio del tipo utilizzato per PintOS
2. Salta allo start.S

Load + Init

`load.S`

`start.S`

`init.h`

`init.c`

Fa l'inizializzazione a basso livello del kernel, scritto in assembly.

1. Fa lo switch da real mode a protected mode
2. Chiede al BIOS la dimensione della RAM (max 64MB) e salva in pagine in `init_ram_size`
3. Abilitare la linea A20 dell'address bus
4. Crea una page table base

Una pagina:
4kB

Load + Init

`load.S`

`start.S`

`init.h`

`init.c`

5. Mappa indirizzi virtuali e fisici [dettagli più avanti]
6. Setta dei registri della CPU (protected mode on, paging on, set segment reg...)
7. Disabilita gli interrupt
8. Salta a `main()` in `init.c`

Load + Init

`load.S`

`start.S`

`init.h`

`init.c`

Fa l'inizializzazione ad alto livello: da qui in poi è tutto in C.

1. Inizializza a 0 l'area BSS (Block Started by Symbol) [limiti in `_start_bss` e `_end_bss`]
2. Fa reading e parsing degli argomenti da command line
3. Inizializza l'esecuzione corrente come un thread con `thread_init()` per poter usare i lock
4. Inizializza la console con `console_init()`

Load + Init

`load.S`

`start.S`

`init.h`

`init.c`

5. Inizializza gli allocatori di memoria:

a) `pallocc_init(usr memory limit)` inizializza il kernel page allocator e prende come parametro il massimo numero di pagine che un utente può prendere per volta.

b) `malloc_init()` inizializza l'allocatore che assegna memoria di dimensioni arbitrarie

6. `paging_init()` fa il set up di una page table per il kernel

Load + Init

load.S

start.S

init.h

init.c

7. Si occupa dell'interrupt handling

a) `intr_init()` sistema la interrupt descriptor table per prepararla a fare interrupt handling

b) `timer_init` e `kbd_init()` preparano per gestire gli interrupt di timer e tastiera

c) `input_init()` fa il merge dell'input da seriale e quello da tastiera in un unico stream

d) `exception_init()` e `syscall_init()` sono predisposte per occuparsi di preparare il sistema alla ricezione di interrupt dagli user programs

Load + Init

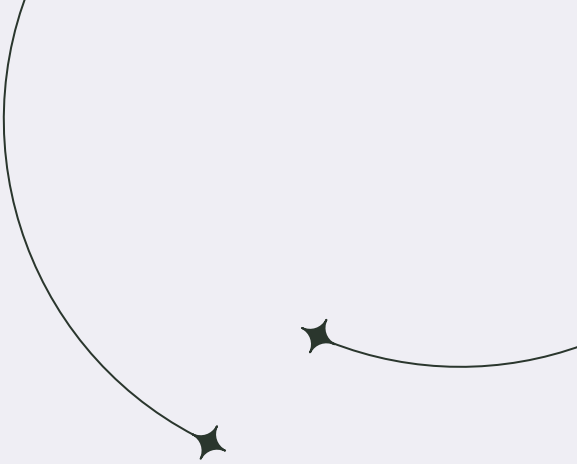
load.S

start.S

init.h

init.c

8. `thread_start()` avvia l'idle thread, lo scheduler e abilita gli interrupt
9. `serial_init_queue()` fa il switch alla modalità interrupt driven serial port I/O
10. `timer_calibrates()` calibra il timer per avere short delays accurati
11. Qualora si abilitasse l'implementazione del file system vanno inizializzati gli IDE disk (qui simulati) con `ide_init()` e il file system stesso con `filesys_init()`
12. `run_actions()` si occupa di eseguire le azioni richieste da command line
13. `shutdown()` spegne eventualmente la macchina



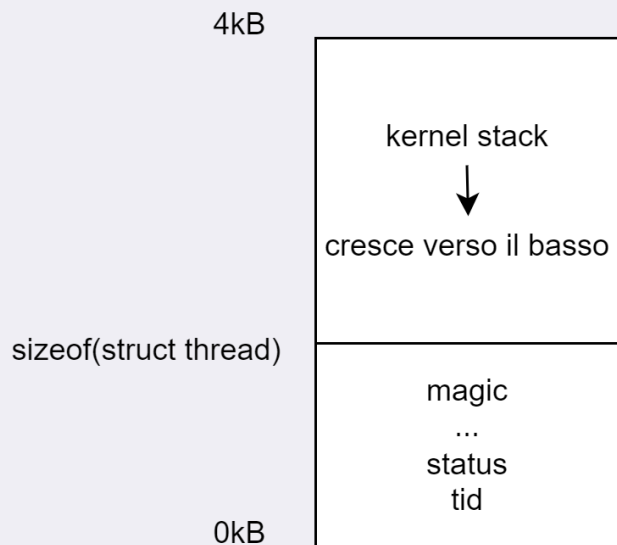
02

Thread

Thread

thread.c

thread.h



Ogni thread ha a disposizione una pagina di memoria (4kB).

La struct thread che lo rappresenta è salvata in basso, lo stack parte dalla fine della pagina e decresce.

thread.c

thread.h

Thread

Struct thread

tid_t tid
enum thread_status status
char name[16]
uint8_t *stack
int priority
struct list_elem allelem
struct list_elem elem
uint32_t *pagedir
unsigned magic

Thread

Struct thread

<code>tid_t tid</code>
<code>enum thread_status status</code>
<code>char name[16]</code>
<code>uint8_t *stack</code>
<code>int priority</code>
<code>struct list_elem allelem</code>
<code>struct list_elem elem</code>
<code>uint32_t *pagedir</code>
<code>unsigned magic</code>

Identificativo del thread

Unico per tutta la lifetime del kernel

I thread vengono numerati a partire da 1 per default, è modificabile

Struct thread

tid_t tid
enum thread_status status
char name[16]
uint8_t *stack
int priority
struct list_elem allelem
struct list_elem elem
uint32_t *pagedir
unsigned magic

Stato del thread espresso con una enum:

- RUNNING vuol dire che thread_current ritorna lui
- READY è nella coda ready
- BLOCKED sta aspettando qualcosa, deve essere sbloccato da una **thread_unblock** o non verrà rischedulato
- DYING: sarà distrutto dallo scheduler dopo il switch al prossimo thread.

thread.c

thread.h

Thread

Struct thread

tid_t tid
enum thread_status status
char name[16]
uint8_t *stack
int priority
struct list_elem allelem
struct list_elem elem
uint32_t *pagedir
unsigned magic

Nome del thread

Usato per ragioni di debug

Thread

Struct thread

tid_t tid
enum thread_status status
char name[16]
uint8_t *stack
int priority
struct list_elem allelem
struct list_elem elem
uint32_t *pagedir
unsigned magic

Puntatore allo stack del thread

Viene salvato qui durante un context switch. Gli altri registri sono salvati nello stack.

Ogni thread in PintOS ha uno stack di poco meno di 4kB.

Struct thread

tid_t tid
enum thread_status status
char name[16]
uint8_t *stack
int priority
struct list_elem allelem
struct list_elem elem
uint32_t *pagedir
unsigned magic

Priorità del thread

lo scheduler non utilizza la priorità per decidere il prossimo thread a cui cedere la CPU, tuttavia è supportata.

Va da da PRI_MIN (= 0) a PRI_MAX (=63).

Struct thread

tid_t tid
enum thread_status status
char name[16]
uint8_t *stack
int priority
struct list_elem allelem
struct list_elem elem
uint32_t *pagedir
unsigned magic

Elemento lista <allelem>

Usato per salvare il thread nella lista di tutti i thread esistenti in quel momento nel kernel (se fa exit() va rimosso).

Per iterare su tutti i thread si usa **thread_foreach()**.

Struct thread

tid_t tid
enum thread_status status
char name[16]
uint8_t *stack
int priority
struct list_elem allelem
struct list_elem elem
uint32_t *pagedir
unsigned magic

Elemento lista <elem>

Usato per lo stesso motivo per inserire in coda dei ready.

Struct thread

tid_t tid
enum thread_status status
char name[16]
uint8_t *stack
int priority
struct list_elem allelem
struct list_elem elem
uint32_t *pagedir
unsigned magic

Riferimento alla page directory del processo

Struct thread

tid_t tid
enum thread_status status
char name[16]
uint8_t *stack
int priority
struct list_elem allelem
struct list_elem elem
uint32_t *pagedir
unsigned magic

Magic number

Sempre settato a `THREAD_MAGIC`, si controlla per detectare lo stack overflow.

Si mette in fondo in modo che se lo stack straborda sia il primo a cambiare.

PintOS vs OS/161

Struct thread in PintOS

<code>tid_t tid</code>
<code>enum thread_status status</code>
<code>char name[16]</code>
<code>uint8_t *stack</code>
<code>int priority</code>
<code>struct list_elem allelem</code>
<code>struct list_elem elem</code>
<code>uint32_t *pagedir</code>
<code>unsigned magic</code>

Struct thread in OS/161

<code>char *t_name</code>
<code>const char *t_wchan_name</code>
<code>threadstate_t t_state</code>
<code>void *stack</code>
<code>struct switchframe *t_context</code>
<code>struct cpu *t_cpu</code>
<code>struct proc *t_proc</code>

Aggiunte

PintOS vs OS/161

Struct thread in PintOS

<code>tid_t tid</code>
<code>enum thread_status status</code>
<code>char name[16]</code>
<code>uint8_t *stack</code>
<code>int priority</code>
<code>struct list_elem allelem</code>
<code>struct list_elem elem</code>
<code>uint32_t *pagedir</code>
<code>unsigned magic</code>

Struct thread in OS/161

<code>char *t_name</code>
<code>const char *t_wchan_name</code>
<code>threadstate_t t_state</code>
<code>void *stack</code>
<code>struct switchframe *t_context</code>
<code>struct cpu *t_cpu</code>
<code>struct proc *t_proc</code>

Differenze

Thread init



```
struct thread *Running_thread(void)
```

Prende lo stack ptr dalla CPU e lo arrotonda all'inizio di una pagina (con un'istruzione assembly passata con asm che è una funzione C) siccome lo stack parte dall'alto e decresce vuol dire che punterà da qualche parte in mezzo alla pagina, arrotondando per difetto si arriva all'inizio della pagina e quindi della struct thread.

```
static void init_thread (struct thread *t, const char *name, int  
priority)
```

Inizializza T come una struct thread che si chiama name. Controlla la validità dei parametri e assegna stato BLOCKED; copia il nome con la funzione strcpy; gli dà come indirizzo dello stack quello a una pagina più; priorità come da parametro; assegna il magic number; disabilita gli interrupt perché deve salvare il thread nella lista di tutti i thread allelem e poi li ripristina.



Thread init



```
static tid_t next_tid allocate_tid (void)
```

In thread.c viene mantenuta una variabile statica **next_tid** (da modificare se si vuole partire diversamente con la numerazione), qui si incrementa la variabile proteggendo l'incremento con **tid_lock**.

```
void thread_init(void)
```

Inizializza il thread system (quindi in generale viene chiamata solo per il primo thread, mentre `init_thread` viene chiamato per creare ogni nuovo thread). Fa un `ASSERT` per assicurarsi che gli interrupt siano off, l'init del lock per prendere il tid, l'init delle liste di ready e quella che mantiene tutti i thread attivi; poi si occupa della struct thread: prende il `running_thread`, lo passa a `init_thread()` e gli assegna status `RUNNING` e un tid con `allocate_tid()`.



Thread start

```
void thread_start(void)
```

Abilitando gli interrupt avvia il preemptive thread scheduling e crea l'idle thread.

```
void thread_tick(void)
```

aggiorna le statistiche del thread: per quanto sta girando quel thread, e se sta girando da più tick di TIME_SLICE che è il tempo massimo per cui si può far girare un solo thread allora forza lo scheduler a intervenire.

```
void thread_print_stats(void) le stampa.
```

Thread start

```
tid_t thread_create(const char *name, int priority, thread_func  
*function, void *aux)
```

Crea un thread che si chiama name, con priorità priority, che esegue function che riceve come unico argomento aux. Controlla che la funzione abbia come signature void thread_func(void* aux), chiede una pagina all'allocatore di pagine, chiama init_thread(...), gli assegna un tid con allocate_tid e lo mette in coda ready con una chiamata a thread_unblock().

```
void thread_block(struct thread *t)
```

Controlla che non si stia processando un interrupt esterno e che gli interrupt siano disabilitati, perchè non può essere interrotta. Poi mette il thread fuori dalla coda di ready impostando lo stato a BLOCKED. Chiama schedule() che trova un altro thread da runnare e switcha al nuovo thread.

Thread start



```
void thread_unblock(struct thread *t)
```

Si assicura che il thread sia valido con `is_thread()`, disabilita gli interrupt, si assicura che stia venendo chiamata su un thread che era bloccato, lo pusha al fondo della coda ready e gli cambia lo stato in READY. Ripristina gli interrupt.

Alla combinazione di `thread_block/thread_unblock` si preferiscono le primitive di sincronizzazione: `thread_block/thread_unblock` agiscono sugli interrupt che non è un modo efficiente di gestire la sincronizzazione.

```
static bool is_thread(struct thread *t) > torna vero se il thread t non è  
nullo e il suo magic number non è alterato, quindi torna vero se si tratta di un thread  
valido.
```

```
struct thread *thread_current(void) > running_thread ma aggiunge due  
sanity check: (1) is_thread() che controlla lo stack overflow, (2) che il thread sia  
RUNNING.
```



Thread start

```
void thread_exit(void) NO_RETURN
```

Rimuove il thread dalla lista dei thread esistenti, setta il suo stato a DYING e chiama `schedule()` che farà girare qualcun altro e il corrente verrà distrutto da `thread_schedule_tail()` che viene chiamata a completamento di un switch. Si assicura di liberare la pagina del thread di cui sta prendendo il posto se questo è DYING e non è l'idle.

```
void thread_yield(void)
```

Il thread corrente pusha se stesso in coda ready, cambia il suo stato da RUNNING a READY e invoca `schedule()`.

Thread start



```
void thread_foreach(thread_action_func *action, void *aux)
```

Itera sulla lista di tutti i thread attivi del kernel e li fa passare in action passando il parametro aux.

Action deve essere del tipo `void thread_action_func(struct thread *thread, void *aux)`.

```
void thread_set_priority(int new_priority) e int  
thread_get_priority(void)
```

Settano e leggono la priorità del thread in modo sicuro perchè passano attraverso `thread_current()`.



Thread switch

```
static struct thread *next_thread_to_run(void)
```

Se la lista ready è vuota restituisce l'idle thread, altrimenti prende il primo della lista. Lo scheduler a questo livello implementa un algoritmo Round Robin.

```
static void schedule (void)
```

Viene chiamata dalle uniche tre funzioni pubbliche nel file thread che hanno necessità di fare switch: `thread_block()`, `thread_exit()`, `thread_yield()`.

Chi chiama deve assicurarsi di aver disabilitato gli interrupt e aver cambiato il proprio stato in qualcosa di diverso da `RUNNING`. Se il prossimo thread in coda ready è valido, opera il switch tra il corrente e il primo della coda tramite `switch_threads(...)`.

Il thread che è stato rimosso viene passato a `thread_schedule_tail()`.

Thread switch

```
switch_threads(struct thread *cur, struct thread *next)
```

È una routine scritta in assembly in switch.S che salva i registri nello stack, lo stack ptr register della CPU dentro al campo stack della struct thread corrente.

Prende dal campo stack della struct thread del thread a cui dare il controllo e lo mette nello stack ptr register della CPU e ripristina i registri del thread tirandoli fuori dallo stack.

```
void thread_schedule_tail(struct thread *prev)
```

Viene chiamata a completamento di uno switch. Setta il thread corrente come RUNNING, inizia un nuovo time slice da cui contare i tick per cui al thread è consentito girare, si assicura di liberare la memoria del thread di cui sta prendendo il posto se questo è DYING e non è l'idle.

PintOS vs OS/161

✦ PintOS

- Un thread viene portato nello stato di ready con `thread_unblock`, si può bloccare esplicitamente con `thread_block`
- Non c'è una `fork`, solo `thread_create` che chiama `thread_unblock` per inserire il thread nella ready list
- Nel context switch si recuperano i registri dallo stack
- Le operazioni vengono protette disabilitando le interruzioni o con semafori creati ad-hoc, o con lock dedicati (come il `tid_lock`)

✦ OS/161

- Un thread viene portato nello stato di ready con `thread_make_runnable`, per essere portato nello stato di `S_SLEEP` deve essere messo in attesa
- `Thread_fork` fa una `thread_create` e una `thread_make_runnable`
- `Thread_switch` si occupa del switch e usa lo `stack_frame` che è un campo della struct `thread`
- Le operazioni vengono protette con lo spinlock della CPU corrente

PintOS vs OS/161

✦ PintOS

- Un thread viene portato nello stato di ready con `thread_unblock`, si può bloccare esplicitamente con `thread_block`
- Non c'è una `fork`, solo `thread_create` che chiama `thread_unblock` per inserire il thread nella ready list
- Nel context switch si recuperano i registri dallo stack
- Le operazioni vengono protette disabilitando le interruzioni o con semafori creati ad-hoc, o con lock dedicati (come il `tid_lock`)

✦ OS/161

- Un thread viene portato nello stato di ready con `thread_make_runnable`, per essere portato nello stato di `S_SLEEP` deve essere messo in attesa
- `Thread_fork` fa una `thread_create` e una `thread_make_runnable`
- Per il switch si usa lo `stack_frame` che è un campo della struct `thread`
- Le operazioni vengono protette con lo spinlock della CPU corrente

PintOS vs OS/161

✦ PintOS

- Un thread viene portato nello stato di ready con `thread_unblock`, si può bloccare esplicitamente con `thread_block`
- Non c'è una `fork`, solo `thread_create` che chiama `thread_unblock` per inserire il thread nella ready list
- Nel context switch si recuperano i registri dallo stack
- Le operazioni vengono protette disabilitando le interruzioni o con semafori creati ad-hoc, o con lock dedicati (come il `tid_lock`)

✦ OS/161

- Un thread viene portato nello stato di ready con `thread_make_runnable`, per essere portato nello stato di `S_SLEEP` deve essere messo in attesa
- `Thread_fork` fa una `thread_create` e una `thread_make_runnable`
- Per il switch si usa lo `stack_frame` che è un campo della struct thread
- Le operazioni vengono protette con lo spinlock della CPU corrente

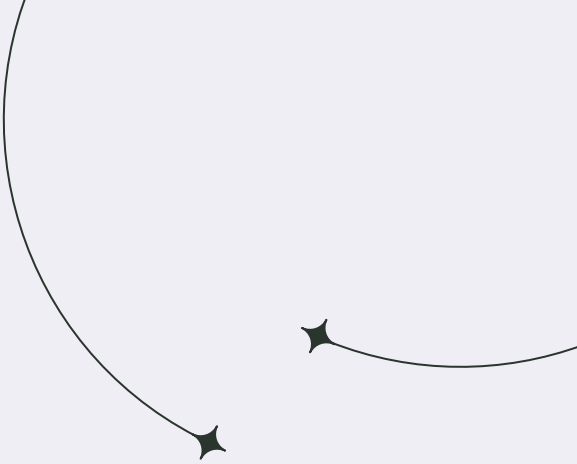
PintOS vs OS/161

✦ PintOS

- Un thread viene portato nello stato di ready con `thread_unblock`, si può bloccare esplicitamente con `thread_block`
- Non c'è una `fork`, solo `thread_create` che chiama `thread_unblock` per inserire il thread nella ready list
- Nel context switch si recuperano i registri dallo stack
- Le operazioni vengono protette disabilitando le interruzioni o con semafori creati ad-hoc, o con lock dedicati (come il `tid_lock`)

✦ OS/161

- Un thread viene portato nello stato di ready con `thread_make_runnable`, per essere portato nello stato di `S_SLEEP` deve essere messo in attesa
- `Thread_fork` fa una `thread_create` e una `thread_make_runnable`
- Per il switch si usa lo `stack_frame` che è un campo della struct `thread`
- Le operazioni vengono protette con lo spinlock della CPU corrente



03

Sincronizzazione



✦ Disabilitare gli Interrupt

✦ Semafori

✦ Locks

✦ Condition Variables: Monitors

✦ Optimization Barriers

Disabilitare gli Interrupt

Quando sono disabilitati nessun thread può interrompere il thread corrente, altrimenti può venire interrotto anche a metà di un'istruzione.

PintOS è un preemptible kernel.

Tradizionalmente i sistemi Unix hanno kernel nonpreemptible: i kernel thread possono interrompersi solo quando sono loro a chiamare lo scheduler esplicitamente.

- `enum intr_level { INTR_ON, INTR_OFF }` denota lo stato degli interrupt nel sistema
- `enum intr_level intr_get_level(void)` ritorna lo stato degli interrupt
- `enum intr_level intr_set_level(enum intr_level level)` setta lo stato degli interrupt a level e ritorna lo stato precedente
- `enum intr_level intr_enable/disable(void)` funzioni che abilitano/disabilitano gli interrupt e ritornano il vecchio stato



Si usano unicamente per sincronizzare i thread kernel con gli interrupt handler esterni che non possono usare le altre forme di sincronizzazione e non possono attendere.

PintOS non gestisce i NMI.

Semafori

È un intero non negativo con due operatori associati che lo modificano in modo atomico:

- **down** (o **P**) che aspetta che il valore diventi positivo per poi decrementarlo,
- **up** (o **V**) che lo incrementa e sveglia un altro thread in attesa se ce ne sono.

I semafori sono implementati internamente tramite l'abilitazione/disabilitazione degli interrupt e la chiamata a **thread block/unblock**.



Semafori

```
void sema_init (struct semaphore *sema, unsigned value)
```

Inizializza il semaforo al valore value.

```
void sema_down (struct semaphore *sema)
```

Finchè il valore del semaforo è 0 ci si mette nella lista di attesa del semaforo e ci si cambia lo stato in BLOCKED. Quando il thread in attesa viene sbloccato decrementa il valore del semaforo.

```
void sema_up (struct semaphore *sema)
```

Fa l'unblock di un thread nella lista (se ce ne sono) e incrementa il semaforo.

```
void sema_try_down/up (struct semaphore *sema)
```

Versioni di down/up in cui si cerca di decrementare il semaforo senza bloccare il thread. Se l'operazione non ha successo si ritorna falso, altrimenti vero.

Non è efficiente da usare in un loop. Sono consigliate le altre versioni o altri approcci.

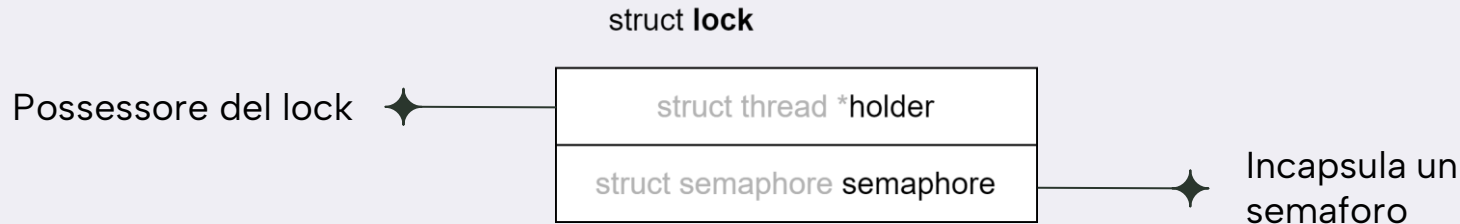
Locks

È un semaforo con valore iniziale 1.

Up e down si chiamano `release` e `acquire`.

La differenza con il semaforo è anche che ha un `owner` che non inferito alla creazione ma all'acquire, e solo chi ha chiamato `acquire` (l'owner) può fare `release`.

Pintos da `errore` se possedendo il lock si richiama `acquire`, non sono implementati i lock ricorsivi.



Locks

```
void lock_init (struct lock *lock)
```

Imposta l'holder a NULL e chiama sema_init con valore di inizializzazione 1.

```
bool lock_held_by_current_thread (const struct lock *lock)
```

Controlla se il lock è posseduto del thread corrente chiamando thread_current e confrontandolo con il campo holder. Non c'è una funzione per controllare se un thread qualsiasi è il possessore del lock, perchè con la concorrenza appena si esce da questa presunta funzione la risposta potrebbe cambiare.

```
void lock_acquire (struct lock *lock)
```

controlla che il thread possessore non sia il chiamante, chiama sema_down e imposta l'owner.

```
void lock_release (struct lock *lock)
```

se il chiamante è anche il possessore rilascia il lock chiamando sema_up.

```
void lock_try_acquire (struct lock *lock)
```

Usa sema_try_down secondo lo stesso principio.

Condition Variables: Monitors

Un monitor consiste di una serie di dati sincronizzati, un lock e una o più condition variable. Prima di accedere ai dati protetti va acquisito il lock (si dice che si è nel monitor).

Quando il thread è nel monitor può accedere liberamente ai dati protetti, quando esce deve rilasciare il lock.

Le condition variable fanno in modo che quando si è nel monitor e bisogna aspettare che qualche condizione si verifichi, ci si possa mettere in attesa sulla corrispondente condvar, e questa si occupa di rilasciare il lock e aspettare che la condizione si avveri.

Se è stato modificato qualche valore coinvolto in una condvar va fatta una signal.

struct **condition**

struct list waiters

Lista di thread in
attesa sulla condvar

Condition Variables: Monitors

```
void cond_init (struct condition *cond)
```

Fa solo l'inizializzazione della lista di thread in attesa.

```
void cond_init (struct condition *cond, struct lock *lock)
```

Si mette in attesa su una condizione, si deve chiamare possedendo il lock.

Rilascia il lock, attende su un semaforo creato con valore 0 e quando riesce a fare down sul semaforo acquisisce nuovamente il lock.

Va ritestata la condizione quando si esce ed eventualmente attendere nuovamente.

```
void cond_signal (struct condition *cond, struct lock *lock UNUSED)
```

Va chiamata possedendo il lock e se qualche thread è in attesa sulla condition variable ne sveglia uno.

```
void cond_broadcast (struct condition *cond, struct lock *lock)
```

Come signal ma li sveglia tutti con più chiamate a cond_signal.

Optimization Barriers

È implementata dalla macro

```
#define barrier() asm volatile ("" : : : "memory")
```

Indica un punto nel codice in cui si desidera limitare l'azione del compilatore, garantendo che alcune ottimizzazioni specifiche non siano applicate oltre quella barriera.

Esempi di ottimizzazioni che si può voler evitare: riordinamento delle istruzioni, soppressione di cicli inefficaci...

Due esempi di utilizzo sono in `too_many_loops()` e `busy_wait()` in `devices/timer.c`.



Optimization Barriers

Alcune considerazioni su soluzioni alternative candidate:

1. Bloccare gli interrupt per non essere bloccati tra due istruzioni può funzionare ma non protegge dal reordering e fa sprecare il tempo dell'esecuzione dell'handler.
2. Si potrebbero dichiarare le variabili in gioco come volatile, che ridimensiona lievemente le ottimizzazioni sulle righe coinvolte perchè il compiler sa che possono essere viste dall'esterno, ma la sintassi di volatile non è ben definita e in Pintos non è usato.
3. I lock non sono una soluzione perchè prevengono gli interrupt ma non il reordering.
4. Pintos però tratta tutte le funzioni definite come extern come una forma limitata di una optimization barrier, e limita l'azione del compiler nelle zone in cui sono usate. Una funzione definita nello stesso modulo o nell'header non gode di questo beneficio.

PintOS vs OS/161

✦ PintOS

- Nei semafori si attende cambiando lo stato del thread con block/unblock e si proteggono le operazioni disabilitando gli interrupt

✦ OS/161

- Nei semafori si attende con wait channel e si proteggono le operazioni con uno spinlock
- Non sono implementati nè lock nè condition variable, anche se il SO fornisce le interfacce.

PintOS vs OS/161

✦ PintOS

- Nei semafori si attende cambiando lo stato del thread con block/unblock e si proteggono le operazioni disabilitando gli interrupt

✦ OS/161

- Nei semafori si attende con wait channel e si proteggono le operazioni con uno spinlock
- Non sono implementati nè lock nè condition variable, anche se il SO fornisce le interfacce.



04

Allocazione della Memoria



Page Allocator

Block Allocator

Page Allocator

Alloca per unità di pagine, si usa al più per concedere una pagina per volta, ma può allocare anche più pagine contigue.

La memoria di sistema è divisa in due pool: kernel e user pool. Ognuno è grande metà dello spazio sopra a 1MB. Se si avvia Pintos con l'opzione `-ul` si può cambiare la proporzione. Allo stato delle cose si alloca solo da quello kernel.

Una bitmap traccia lo stato della memoria, un bit rappresenta una pagina. Una richiesta di n pagine fa una scansione della bitmap e ritorna il first fit.

La limitazione è proprio la frammentazione, nel caso patologico la memoria è vuota per metà ma non possono essere richieste più di una pagina.

Si raccomanda di evitare le allocazioni multiple per questo motivo.

Page Allocator

```
void palloc_init (size_t user_page_limit)
```

Calcola le dimensioni dei due pool e li inizializza con `init_pool` creando la bitmap e posizionandola all'inizio della memoria di ognuno dei due

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

Usa `bitmap_scan_and_flip` della libreria `bitmap.c` per implementare la ricerca first fit di `page_cnt` pagine.

I flag possono essere specificati anche in gruppo:

- **PAL_ASSERT** panica il kernel se non si riesce a fare l'allocazione (user non può far panicare)
- **PAL_ZERO** inizializza la memoria richiesta a 0 prima di ritornarla
- **PAL_USER** se specificato prende dal pool user, sennò dal kernel

Page Allocator

```
void *palloc_get_page(enum palloc_flags flags)
```

Usa la funzione precedente chiamandola con `pagecnt = 1`.

```
void palloc_free_multiple (void* pages, size_t page_cnt)
```

Rileva a quale pool appartiene la pagina, controlla che tutta la parte coinvolta della bitmap fosse a 1 con `bitmap_all` e la setta a 0 con `bitmap_set_multiple`.

Per supporto al debug quando si libera una pagina questa viene azzerata.

```
void palloc_free_page (void* page)
```

Usa la funzione precedente chiamandola con `pagecnt = 1`.



Block Allocator

Può allocare blocchi di qualsiasi dimensione attingendo dal kernel pool.

Si poggia sul page allocator.

Segue due strategie di allocazione distinte a seconda della dimensione richiesta

- < 1kB: arrotonda al $\min\{\text{prossima potenza di } 2, 16\text{B}\}$ e chiede una pagina (chiamata arena), raccoglie tutte le allocazioni di quella dimensione e le destina alla stessa pagina.
- > 1kB: arrotonda al più vicino intero di pagina e consegna quelle pagine.

Quando chiede una arena la divide in blocchi e questi blocchi sono aggiunti alla free list.

Quando nessun blocco della pagina è utilizzato viene fatta la free della pagina.

C'è comunque un po' di spreco nei round up delle allocazioni.

Le allocazioni piccole spesso possono essere soddisfatte con la free list senza richiedere ulteriore memoria. Le allocazioni più grandi di una pagina hanno la stessa debolezza del page allocator.

Block Allocator

struct **desc**

size_t blocks_size
size_t blocks_per_arena
struct list free_list
struct lock lock

struct **arena**

unsigned magic
struct desc *desc
size_t free_cnt

Ogni richiesta di memoria <1kB viene assegnata a un descrittore che si occupa di gestire blocchi di quella dimensione. Se la free_list non è vuota la richiesta viene soddisfatta attingendo a questa.

Altrimenti, si chiede un'altra arena, che viene divisa in blocchi aggiunti alla free_list di quel descriptor e la richiesta viene soddisfatta con uno di questi.

Un'arena senza blocchi in uso viene liberata.

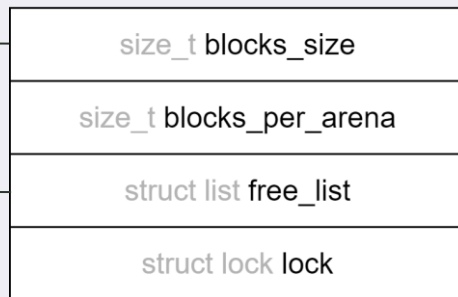
Block Allocator

Dimensione di
un blocco

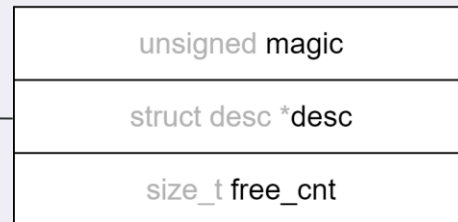
Lista di
blocchi liberi

NULL se l'arena
non possiede un
descriptor

struct **desc**



struct **arena**



Numero di blocchi
per arena

Magic number per
controllare la
validità dell'arena

blocchi liberi o #
pagine in big block

Block Allocator

```
void *malloc (size_t size)
```

Alloca un blocco di **size** bytes secondo il criterio descritto.

```
void *calloc (size_t a, size_t b)
```

Fa una **malloc** ma prende dimensione $a*b$ (controlla prima che la moltiplicazione non faccia overflow nel **size_t** che la contiene) e azzerla la memoria prima di restituirla.

```
void *realloc (void *old_block, size_t new_size)
```

Cerca di riallocare il blocco con dimensione **new_size**. Se non ci riesce ritorna NULL e il vecchio blocco rimane accessibile, se ci riesce potrebbe averlo spostato. Con **old_block = NULL** è equivalente a una **malloc**, con **size = 0** è equivalente a liberare la memoria del blocco.



Block Allocator

```
void *free (void *p)
```

Per motivi di debug durante la free si azzera la memoria.

Si rimette il blocco nella free list e se ci si accorge che la pagina rimane inutilizzata, si chiama la **pallocc_free_page**. Se invece il descrittore era NULL vuol dire che si trattava di un blocco grande che coinvolgeva più pagine: si chiama la **pallocc_free_multiple**.

PintOS vs OS/161

✦ PintOS

- Il meccanismo di allocazione è più sofisticato e prevede la liberazione della memoria.
- Si alloca per blocchi di dimensione arbitraria o per multipli di pagine.
- Si alloca solo da kernel pool.

✦ OS/161

- L'allocatore effettua solo allocazioni contigue di memoria senza mai rilasciarla.
- Si alloca solo per multipli di pagine, non ha il problema di PintOS.
- Sia user che kernel passando per funzioni diverse otterranno memoria da `getppages` che chiama `ram_stealmem`.

PintOS vs OS/161

✦ PintOS

- Il meccanismo di allocazione è più sofisticato e prevede la liberazione della memoria.
- Si alloca per blocchi di dimensione arbitraria o per multipli di pagine.
- Si alloca solo da kernel pool.

✦ OS/161

- L'allocatore effettua solo allocazioni contigue di memoria senza mai rilasciarla.
- Si alloca solo per multipli di pagine, non ha il problema di PintOS.
- Sia user che kernel passando per funzioni diverse otterranno memoria da `getppages` che chiama `ram_stealmem`.

PintOS vs OS/161

✦ PintOS

- Il meccanismo di allocazione è più sofisticato e prevede la liberazione della memoria.
- Si alloca per blocchi di dimensione arbitraria o per multipli di pagine.
- Si alloca solo da kernel pool.

✦ OS/161

- L'allocatore effettua solo allocazioni contigue di memoria senza mai rilasciarla.
- Si alloca solo per multipli di pagine, non ha il problema di PintOS.
- Sia user che kernel passando per funzioni diverse otterranno memoria da `getppages` che chiama `ram_stealmem`.

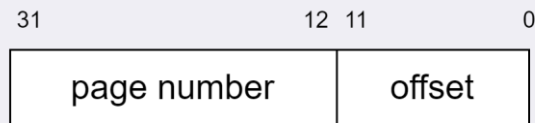


05

Indirizzi Virtuali

Indirizzi Virtuali

Gli indirizzi virtuali in PintOS sono divisi in 20 bit di page number e 12 di offset.



- La macro **PGMASK** è una bit mask che ha tutti i bit dell'offset a 1 e gli altri a 0
- **unsigned *pg_ofs (const void *va)** estrae l'offset dall'indirizzo mettendo in and con la bit mask
- **void *pg_no (const void *va)** estrae il page number allo stesso modo
- **void *pg_round_up (const void *va)** dato un indirizzo ritorna il primo virtual address di quella pagina cioè il page number concatenato con offset 0
- **void *pg_round_down (const void *va)** arrotonda al più vicino page boundary
- **bool is_user/kernel_vaddr (const void *va)** distinguono a chi appartiene quell'indirizzo

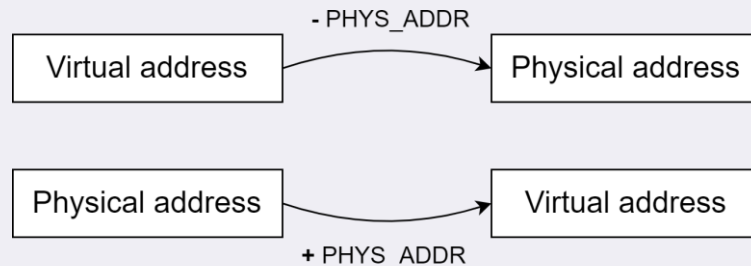
Indirizzi Virtuali

La memoria virtuale kernel/user è divisa dall'indirizzo PHYS_BASE:

- da 0 a PHYS_BASE è user,
- da PHYS_BASE a 4GB è kernel.

PintOS fa una mappatura one-to-one della memoria virtuale e fisica del kernel.

L'indirizzo virtuale PHYS_ADDR corrisponde allo 0 fisico, quindi per trasformare un indirizzo virtuale in uno fisico va sottratto PHYS_ADDR, per trasformare un indirizzo fisico in quello virtuale va aggiunto PHYS_ADDR.



PintOS vs OS/161

✦ PintOS

- La mappatura di indirizzi virtuali e fisici per kernel si fa con addizioni e sottrazioni
- La mappatura per processi utente si fa con una page directory specifica per il processo

✦ OS/161

- La mappatura con page table si applica a kernel e user memory



06

Page Table



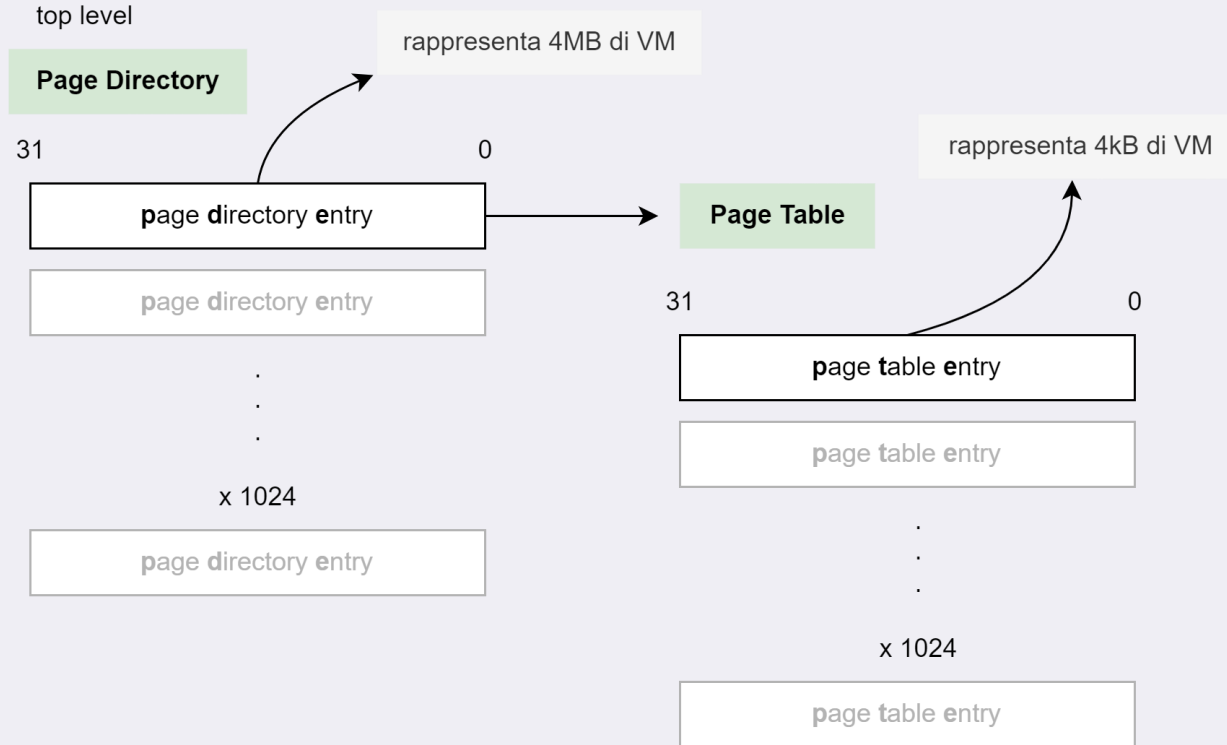
Hardware Page Table Format



Funzioni di Libreria



Hardware Page Table Format





Traduzione indirizzi



La traduzione degli indirizzi avviene in **tre step**:



primi 10 bit più significativi

(1) ricava la page table giusta
dalla page directory

ultimi 12 bit

(3) danno l'offset del dato
nella pagina

secondi 10 bit più significativi

(2) ricava l'indirizzo fisico
della pagina interessata

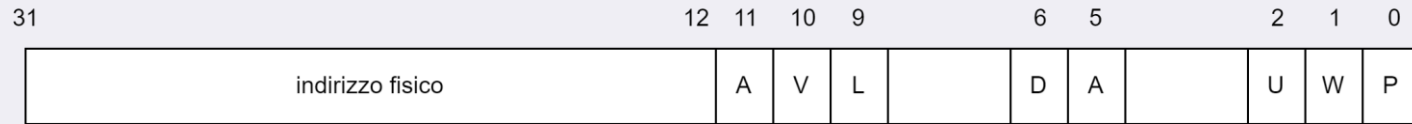
Hardware Page Table Format



Formato pte



Page Table Entry Format



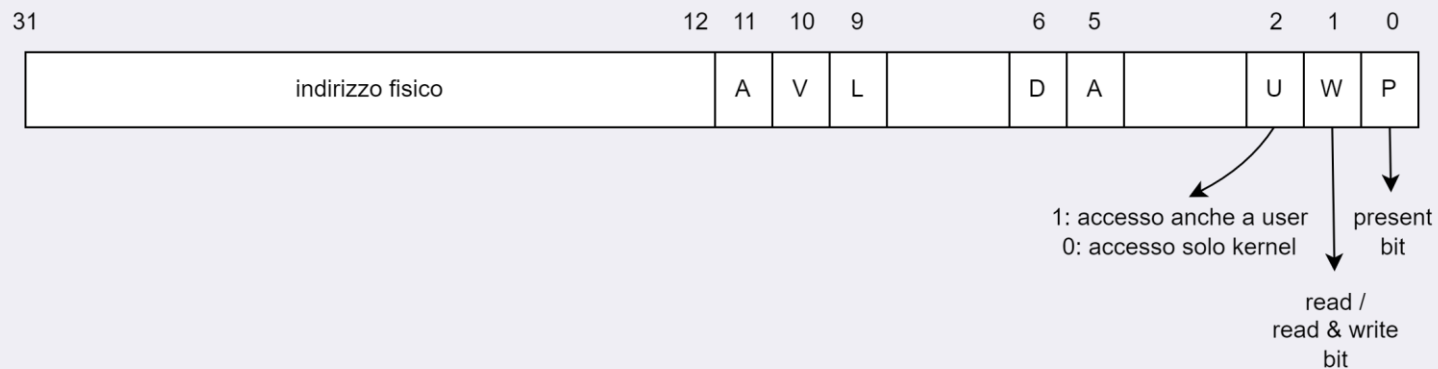
Hardware Page Table Format



Formato pte



Page Table Entry Format



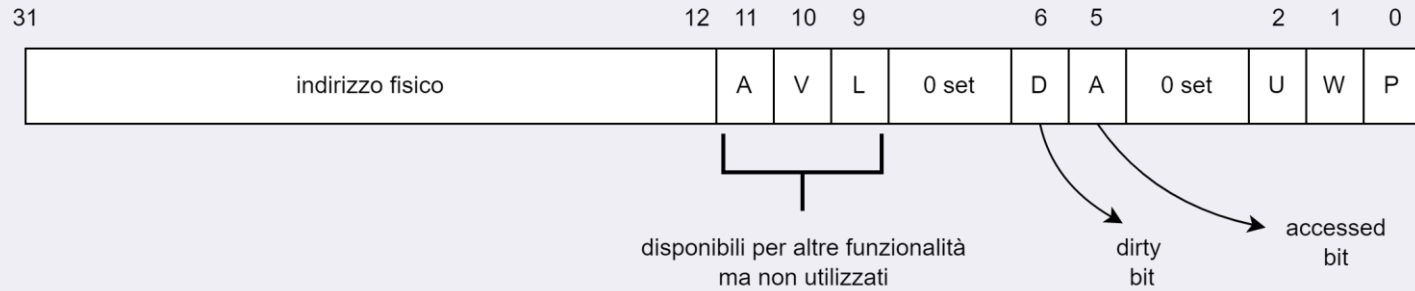
Hardware Page Table Format



Formato pte



Page Table Entry Format





Supporto politiche
di replacing



pagedir.h
pagedir.c

D	A
---	---

I bit D – dirty bit e A – accessed bit sono resi disponibili per l'implementazione di politiche di page replacing.

D > settato a 1 dopo ogni scrittura

A > settato a 1 dopo ogni accesso in lettura o scrittura



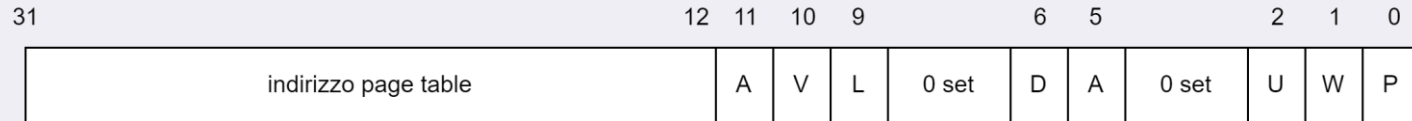
Formato ptd



pagedir.h
pagedir.c

Il formato di una page directory entry è lo stesso di una page table entry con la differenza che i 20 bit più significativi puntano ad una page table.

Page Directory Entry Format





Creazione, distruzione e
attivazione



pagedir.h
pagedir.c



```
uint32_t pagedir_create (void)
```

Chiede una pagina interla per inizializzare la page directory.

```
void pagedir_destroy (uint32_t *pd)
```

Libera la memoria di tutte le risorse che sta mantenendo la pd e la pagina della pd stessa.

```
void pagedir_activate (uint32_t *pd)
```

Posiziona l'indirizzo fisico della pd passata come parametro nel registro page directory base register (PDBR) della CPU e automaticamente la page directory è attiva.



Ispezione e update



`pagedir.h`
`pagedir.c`

```
bool pagedir_set_page (uint32_t *pd, void *upage, void *kpage, bool  
writable)
```

Aggiunge un mapping da user page a kernel page, in lettura e scrittura o solo in lettura a seconda del flag.

```
void *pagedir_get_page (uint32_t *pd, const void *uaddr)
```

Prende uno user address fisico e ritorna (se è mappato) il rispettivo kernel virtual address.

```
void pagedir_clear_page (uint32_t *pd ,void *upage)
```

Invalida la pagina upage, se si cerca di fare accesso a questa pagina ci sarà un page fault ma gli altri bit sono preservati se si volessero leggere postumi per altri motivi.



Gestione delle entry



pte.h

Page table entry

```
uint32_t pte_create_kernel (void *page, bool writable)
```

Prende una pagina e crea una pte che punta a page che è un kernel virtual address.

```
uint32_t pte_create_user (void *page, bool writable)
```

Prende uno user address fisico e ritorna (se è mappato) il rispettivo kernel virtual address.

```
void *pte_get_page (uint32_t pte)
```

Ritorna il kernel virtual address della pagina a cui punta la pte in ingresso.



Gestione delle entry



pte.h

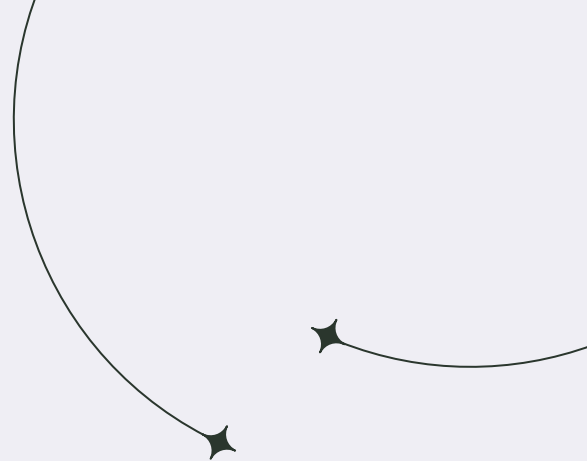
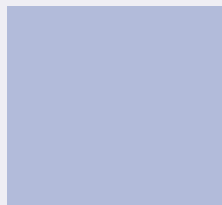
Page directory entry

```
uint32_t pde_create (uint32_t *pt)
```

Ritorna una entry pde che punta alla page table in input, viene marcato come presente, user accessible e read and write.

```
uint32_t *pde_get_pt (uint32_t *pde)
```

Ritorna il kernel virtual address della page table a cui la pde in ingresso punta.



Nuove funzionalità

- ♦ Politica Best Fit nel Page Allocator
- ♦ Priority Scheduler con Priority Shift

Come aggiungere un test

1.

Scrivere la funzione di test
con signature
`void test_func (void)`

2.

Salvare il file in `src/tests/threads/`
con formato del nome
`mine_nometest_test.c`

3.

In `test.h` aggiungere il riferimento
al file creato con
`extern test_func`
`mine_nometest_test`

4.

In `test.c` aggiungere una entry al
dizionario `tests[]` con formato
`{ "nome-cmd",`
`mine_nometest_test }`
per far corrispondere al file un nome
con cui invocare il test da cmd

5.

In `Make.tests`

- Nella sezione `# Test names.`
aggiungere il nome con cui chiamare il
test da cmd
- Nella sezione `# Sources for tests.`
aggiungere il file sorgente da compilare
per il nuovo test



Opzioni di compilazione

Con la compilazione condizionale si abilitano/disabilitano le modifiche da me apportate al codice sorgente. Per attivare un'opzione di compilazione va dichiarata in **src/threads/Make.vars** come **-DNOME_OPZIONE**.

DEBUG_BESTFIT

Abilita le stampe della bitmap per fini di debug e la definizione della funzione di stampa

BESTFIT

Abilita la policy di ricerca Best Fit di cnt pagine su bitmap

DEBUG_PRIORITY

Abilita le stampe della ready list per fini di debug e la definizione della funzione di stampa

PRIORITY

Abilita lo scheduling con priorità e il meccanismo di priority shift.



Best Fit policy

Descrizione

Prima

Per ogni richiesta di allocazione di n pagine viene effettuata una chiamata a **pallocc_get_multiple**.

Questa chiama **bitmap_scan_and_flip** che deve cercare nella bitmap il primo gruppo di pagine libere sufficientemente lungo e marcarlo come occupato per poterlo assegnare.

Viene chiamata **bitmap_scan** per la ricerca, che impiega la funzione **bitmap_contains** per interrompere la ricerca qualora il gap non sia sufficiente per soddisfare la richiesta.

Ora

bitmap_scan_and_flip chiama invece **bitmap_scan_best_fit** che è il cuore della modifica e implementa una ricerca del gap più corto che è in grado di accomodare le pagine richieste.

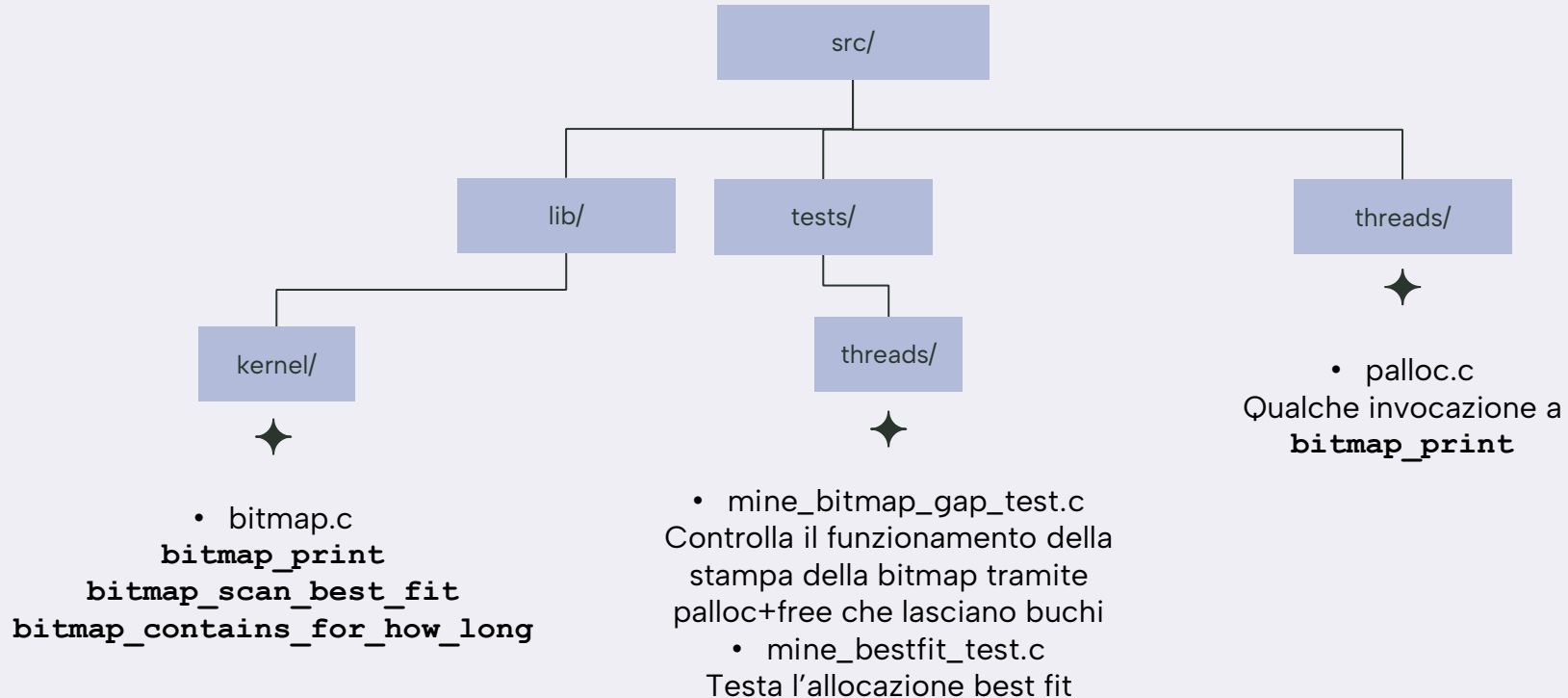
Per questa ricerca è stato necessario estendere **bitmap_contains** con

bitmap_contains_for_how_long che si comporta come prima con l'aggiunta che nel caso in cui il gap rispetti la richiesta, scrive in un parametro passato per indirizzo l'intera estensione del gap individuato.



Best Fit policy

Modifiche nel codice



Best Fit policy

Pseudocodice

```
bool bitmap_contains_for_how_long (const struct bitmap *b, size_t start,
size_t cnt, bool value, size_t *how_long){
    i = 0;
    while (!break_search && entro i limiti della bitmap) {
        if (bitmap[start + i]== value) { break_search = true; }
        else { i++; }
    }
    if (lunghezza del gap >= cnt) {
        *how_long = lunghezza del gap;
        return false; # significa "non ho dovuto interrompere prima del tempo"
    } else {
        return true; # significa "non sono riuscito ad arrivare ad almeno cnt pagine libere"
    }
}
```

Best Fit policy

Pseudocodice

```
size_t bitmap_scan_best_fit (const struct bitmap *b, size_t start, size_t
cnt, bool value) {
    while (i <= last) { # fino all'ultimo bit in cui ha senso cercare cnt pagine consecutive
        if (!bitmap_contains_for_how_long (b, i, cnt, !value, &how_long)) {
            found_at_least_one = true; # trovato almeno un gap >= cnt pagine
            if (how_long < shortest_gap_len) { # è più corto del gap più corto
incontrato fino ad ora?
                shortest_gap_len = how_long;
                shortest_gap_start = i; }
            i += how_long; # salta il gap appena trovato
        } else { i++; } # salta solo al bit successivo
    }
    if (found_at_least_one) { return shortest_gap_start; }
    return BITMAP_ERROR;
}
```

Priority Scheduling

Descrizione

Prima

Lo scheduler segue un algoritmo Round Robin. Seleziona il prossimo thread a cui dare il controllo con **next_thread_to_run** che fa pop dalla testa della ready list e se questa è vuota restituisce l'idle thread.

Ora

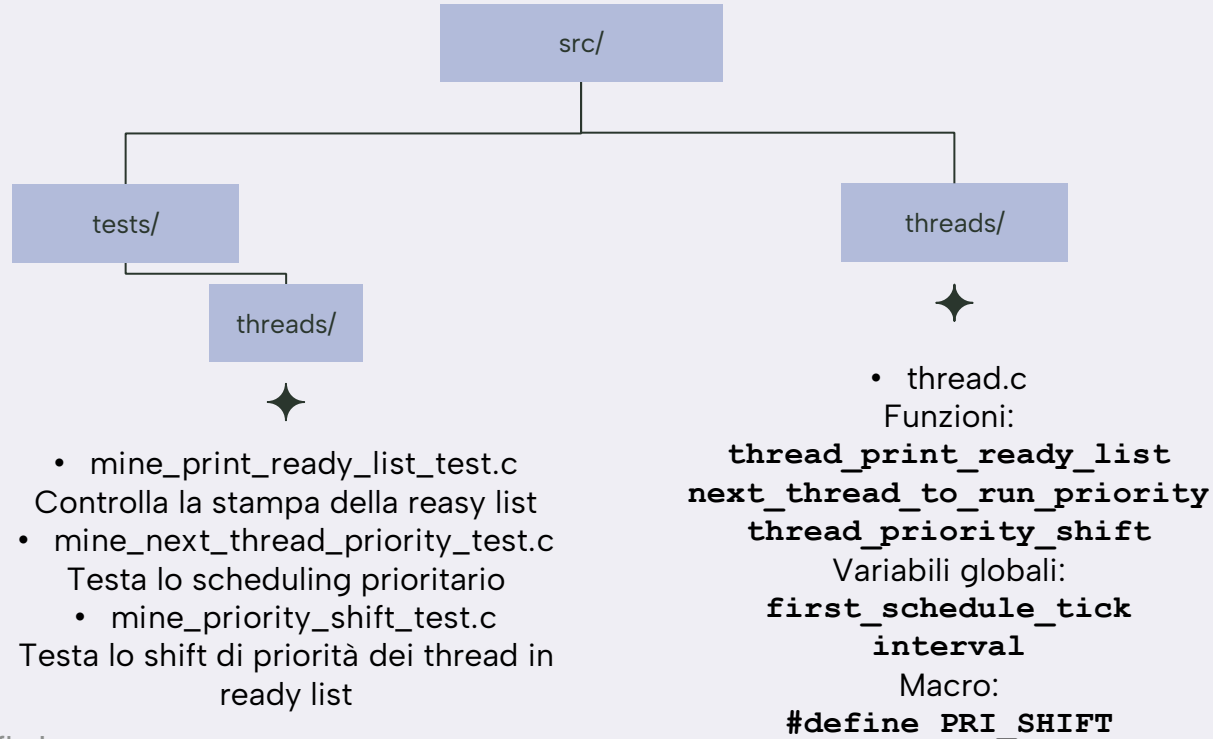
next_thread_to_run è estesa con **next_thread_to_run_priority** che sceglie dalla ready list il primo thread con la priorità più alta in assoluto tra i thread in lista. Dopo averlo selezionato lo rimuove dalla lista.

Per evitare che i thread a priorità bassa rischino la starvation, è stato introdotto un meccanismo di shift della priorità operato da **thread_priority_shift**. Con cadenza temporale customizzabile (da **PRI_SHIFT**), tutti i thread con priorità bassa passano ad avere priorità default, tutti i thread con priorità default passano ad avere priorità alta. L'idle thread è escluso dal priority shift perchè non deve arrivare a competere con altri thread.



Priority Scheduling

Modifiche nel codice



Priority Scheduling

Pseudocodice

```
Static struct thread *next_thread_to_run_priority(void) {  
    # si salva il tick della prima entrata nello scheduler per la logica di shift  
    first_schedule_tick = timer_ticks();  
  
    # se la ready list non è vuota  
    # in un for si cerca il primo thread tra quelli con priorità massima in lista – sarà in:  
    [...] first_max_priority_thread  
    list_remove(first_max_priority_thread);  
  
    # ogni PRI_SHIFT tick operare il priority shift per fare invecchiare i thread  
    if (timer_ticks() > first_schedule_tick + interval*PRI_SHIFT) {  
        thread_priority_shift();  
        interval++; }  
    return first_max_priority_thread;  
}
```

Grazie!

Sofia Longo

s310183@studenti.polito.it



CREDITS: This presentation template was created by [Slidesgo](#), including icons by [Flaticon](#) and infographics & images by [Freepik](#)