PintOS

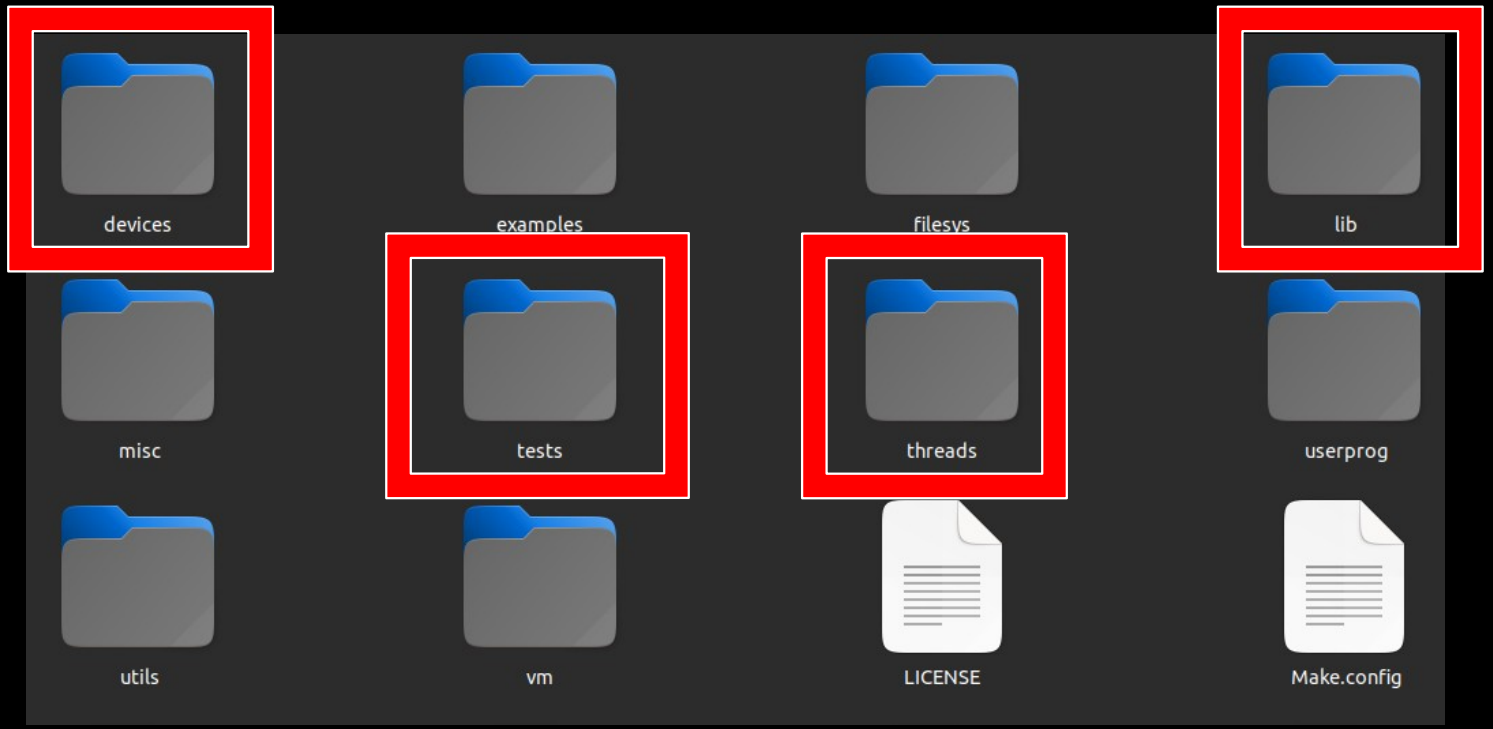
monitores

- Gabriel Ferreira da Silva
- Gabriel Vinicius
- Kailane Felix
- Arthur Santos
- Filipe Maciel
- Gabriel Pierre
- João Coutinho
- Emanoel Thyago
-

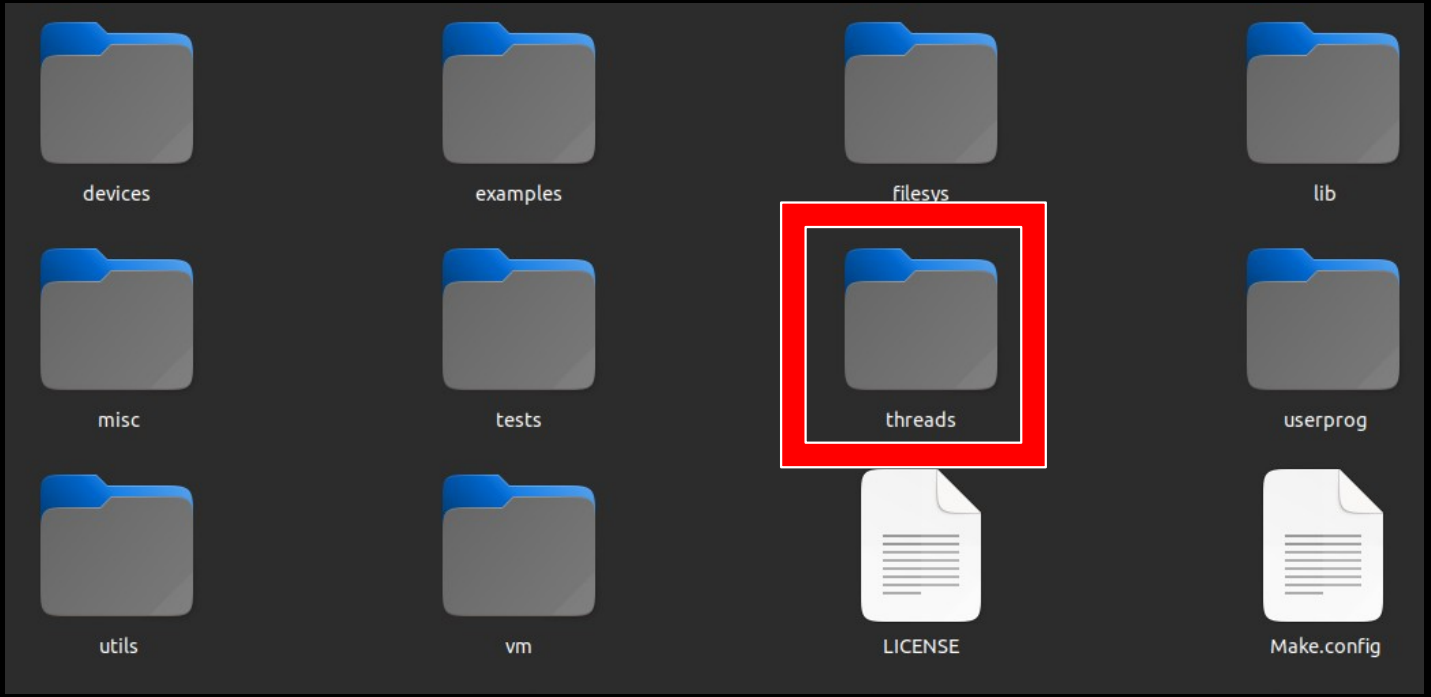
pintos

- Pintos é um OS pequeno e fácil de manipular
- É um projeto introdutório para sistemas operacionais maiores e complexos
- Temos arquivos em assembly, perl, shell e C. Porem precisaremos manipular apenas os arquivos C.
- Usaremos qemu como nosso emulador padrão

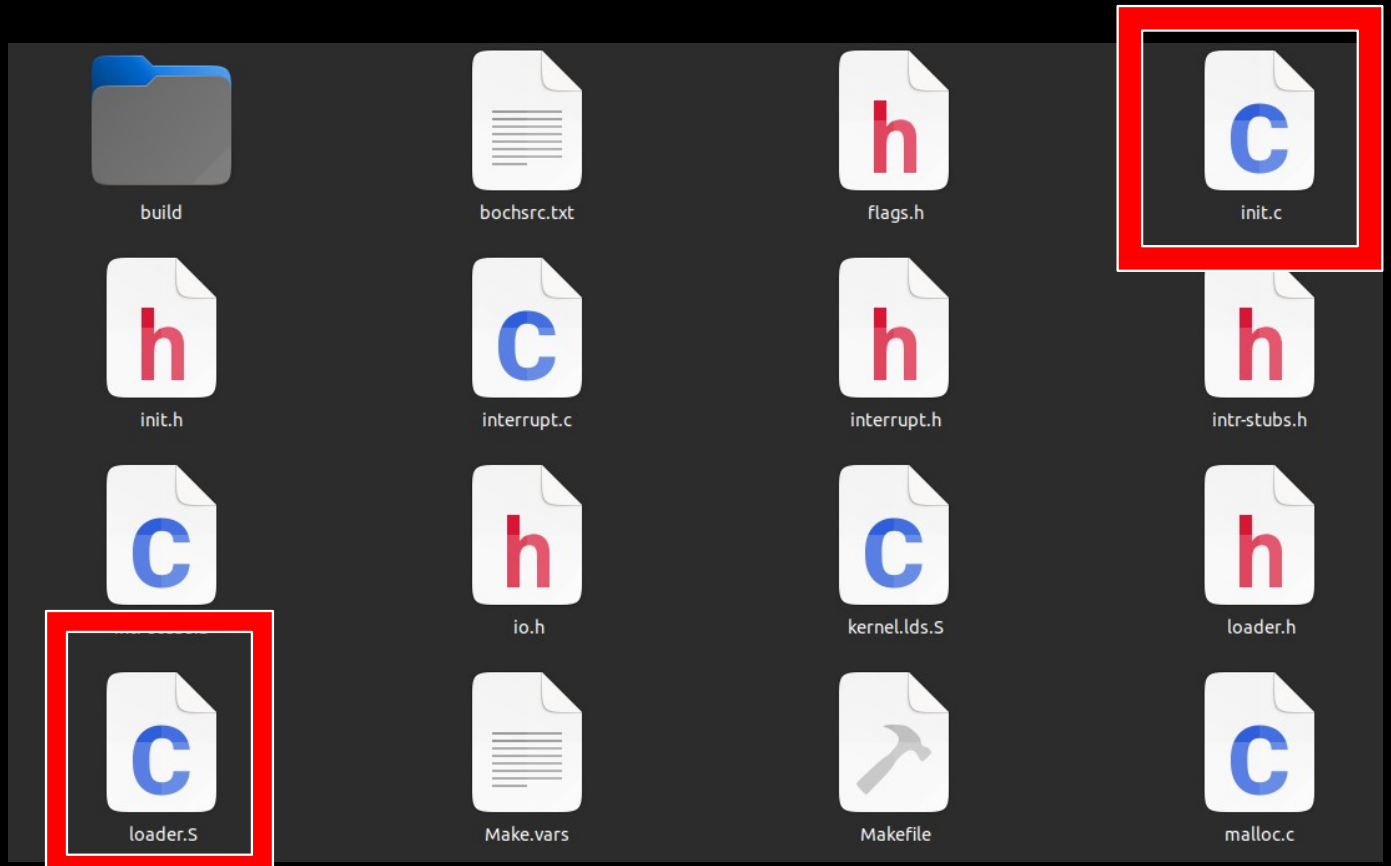
arquivos importantes



arquivos importantes

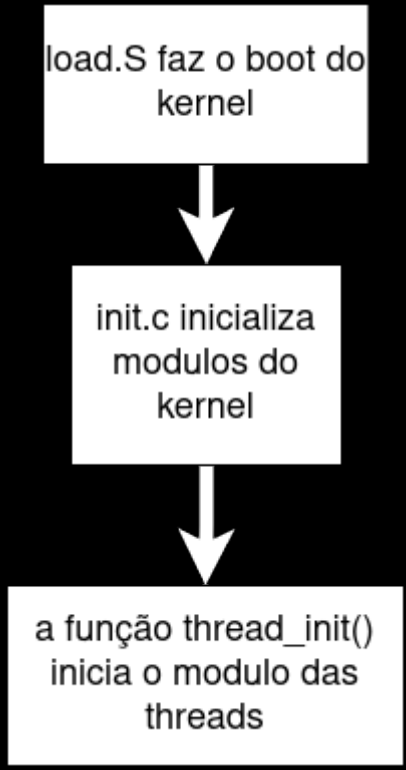


Arquivos importantes



Arquivos importantes

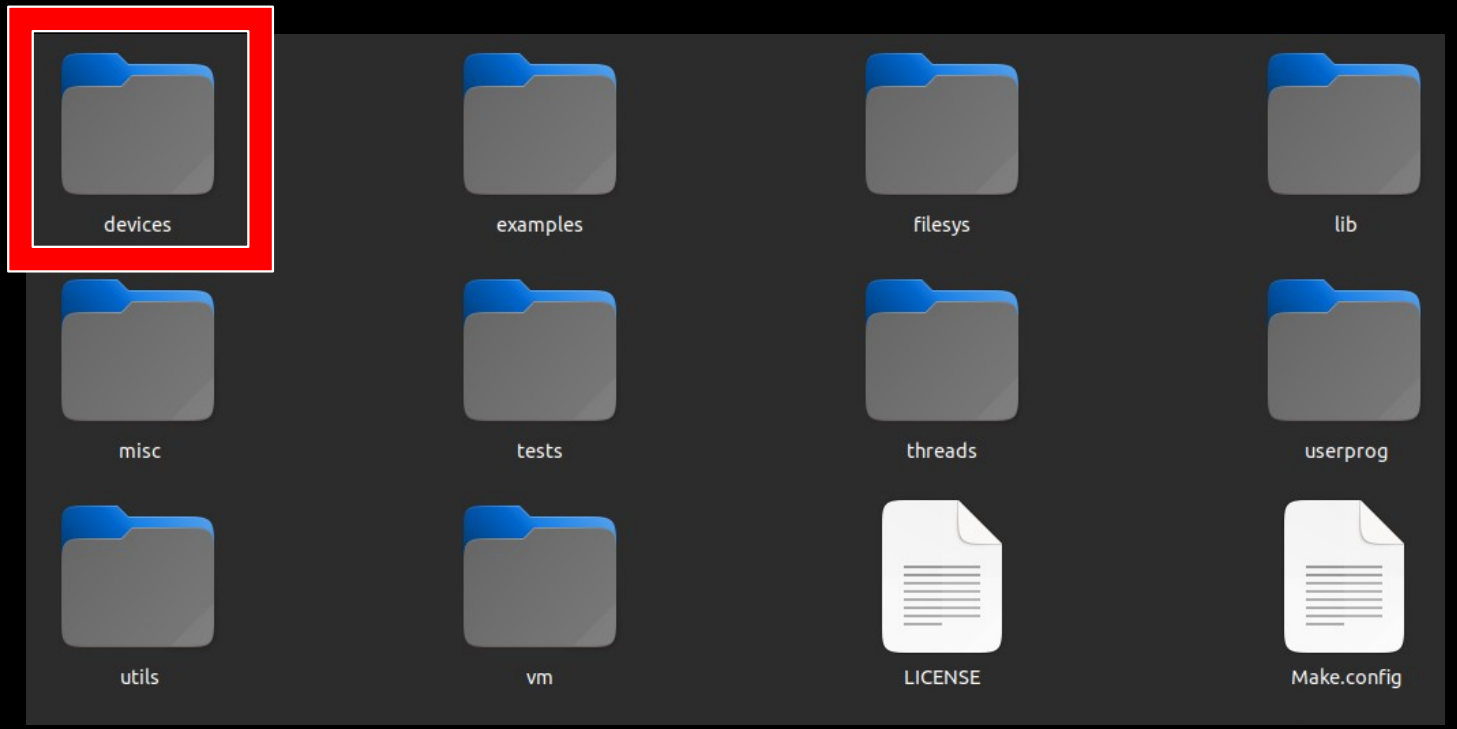
- Load.S esta escrito em assembly que faz o boot e carrega o kernel do pintos
- O primeiro programa a ser executado realmente é o init.c
- Na função main() em Init.c vários componentes do kernel são inicializados incluindo as threads



Arquivos importantes

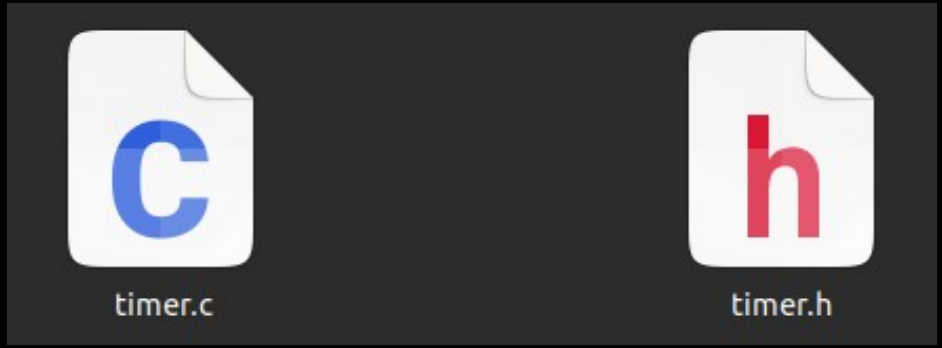
```
74
75 /* Pintos main program. */
76 int
77 main (void)
78 {
79   char **argv;
80
81   /* Clear BSS. */
82   bss_init ();
83
84   /* Break command line into arguments and parse options. */
85   argv = read_command_line ();
86   argv = parse_options (argv);
87
88   /* Initialize ourselves as a thread so we can use locks,
89      then enable console locking. */
90   thread_init ();
91   console_init ();
92
93   /* Greet user. */
94   printf ("Pintos booting with %'"PRIu32" kB RAM...\n",
95           init_ram_pages * PGSIZE / 1024);
96
```

arquivos importantes



devices/timer.c

- Em devices temos os arquivos de timer.c/timer.h
- Aqui estão funções que pintos usa para controle de tempo.



devices/timer.c

Aqui estão definidas algumas constantes importantes.

- TIMER_FREQ: é o número de ticks por segundo.
- No pintos esta definido que 100 ticks de clock é igual a um segundo

```
11 /* See [8254] for hardware details of the
12
13 #if TIMER_FREQ < 19
14 #error 8254 timer requires TIMER_FREQ >=
15 #endif
16 #if TIMER_FREQ > 1000
17 #error TIMER_FREQ <= 1000 recommended
18 #endif
19
20 /* Number of timer ticks since OS booted.
21 static int64_t ticks;
22
```

devices/timer.c

- A função timer_sleep será uma das primeiras a serem modificadas no projeto.
- Ela esta num modo busy wait, fazendo com que ela e todas as outras threads não executem
- Precisamos modifica-la de tal forma que apenas a thread que a chamou durma e as outras não.

```
87 /* Sleeps for approximately TICKS t
88    be turned on. */
89 void
90 timer_sleep (int64_t ticks)
91 {
92   int64_t start = timer_ticks ();
93
94   ASSERT (intr_get_level () == INTR
95   while (timer_elapsed (start) < ti
96     thread_yield ();
97 }
98
```

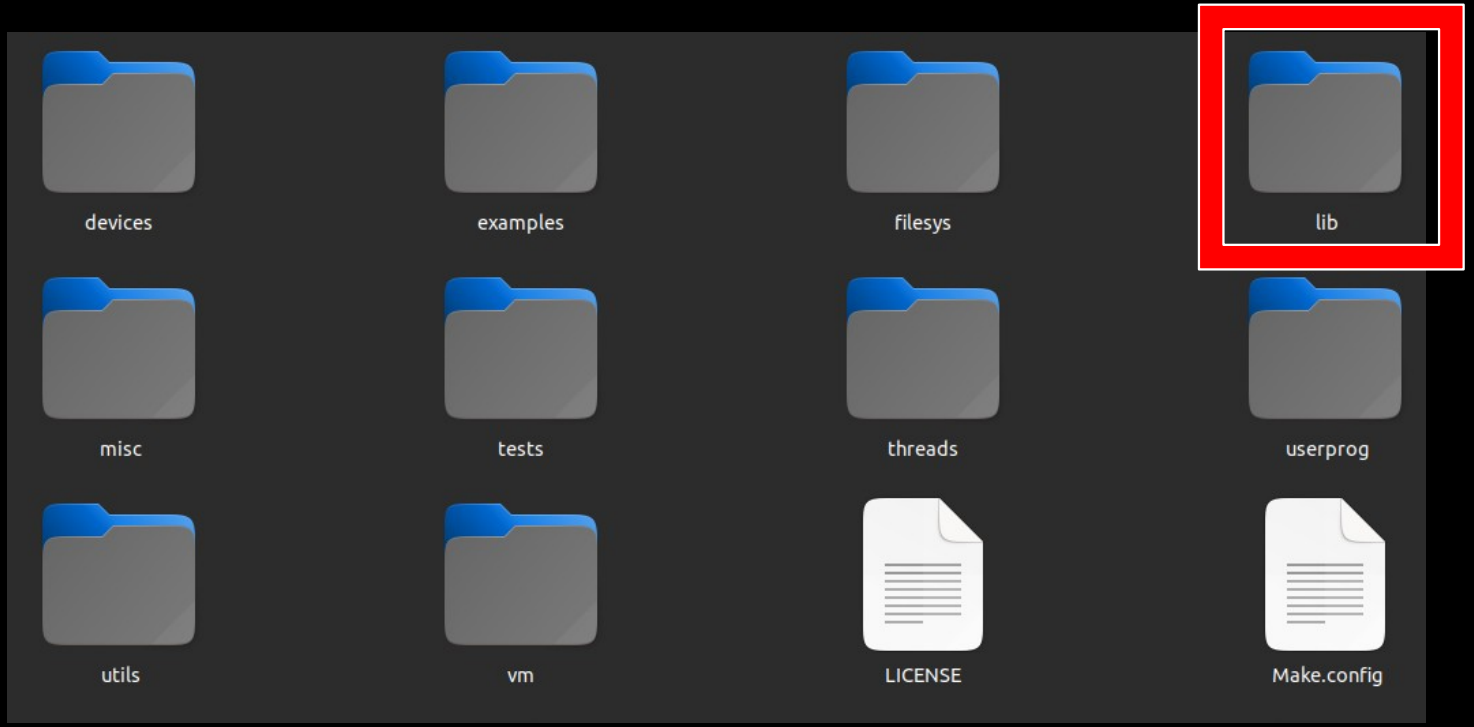
devices/timer.c

Timer_interrupt () atualiza o numero de ticks no OS

- A função thread_tick() está definida em thread.c e é responsável por atualizar parâmetros das threads dependentes do tempo.

```
170 static void
171 timer_interrupt (struct
172 {
173   ticks++;
174   thread_tick ();
175 }
```

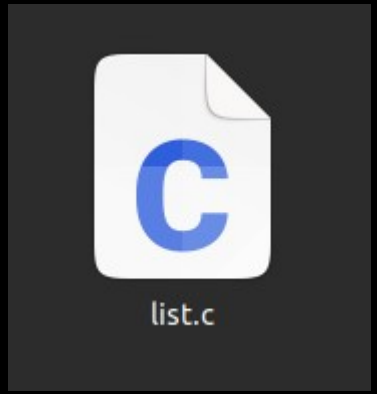
Arquivos importantes



lib/kernel/list.c

Na pasta lib/kernel temos o arquivo list.c/ list.h

- Nele há uma implementação de linked list que serão usadas diversas vezes durante o projeto.



Como criar uma lista

- Crie uma estrutura do tipo list_elem. Ela será o elemento base da lista
- Crie uma estrutura do tipo list.

```
struct list_elem elem;
```

```
static struct list ready_list;
```

Como criar uma lista

- Inicie a lista com função list_init
- Agora pode-se usar a lista normalmente

```
list_init (&ready_list);
```

Exemplos de funções em list.c

- list_push_back (struct list *list, struct list_elem *elem)

  Coloca um item *elem no fim de uma lista *list

- list_push_front (struct list *list, struct list_elem *elem)

  Coloca um item *elem no inicio de uma lista *list

- list_next (struct list_elem *elem)

  dado um elemento *elem retorna o elemento seguinte a ele. Útil quando se precisa iterar sobre uma lista.

- list_end (struct list *list)

  retorna o ultimo elemento da lista *list

Exemplos de funções em list.c

- list_entry(LIST_ELEM, STRUCT, MEMBER)

  essa função recebe:

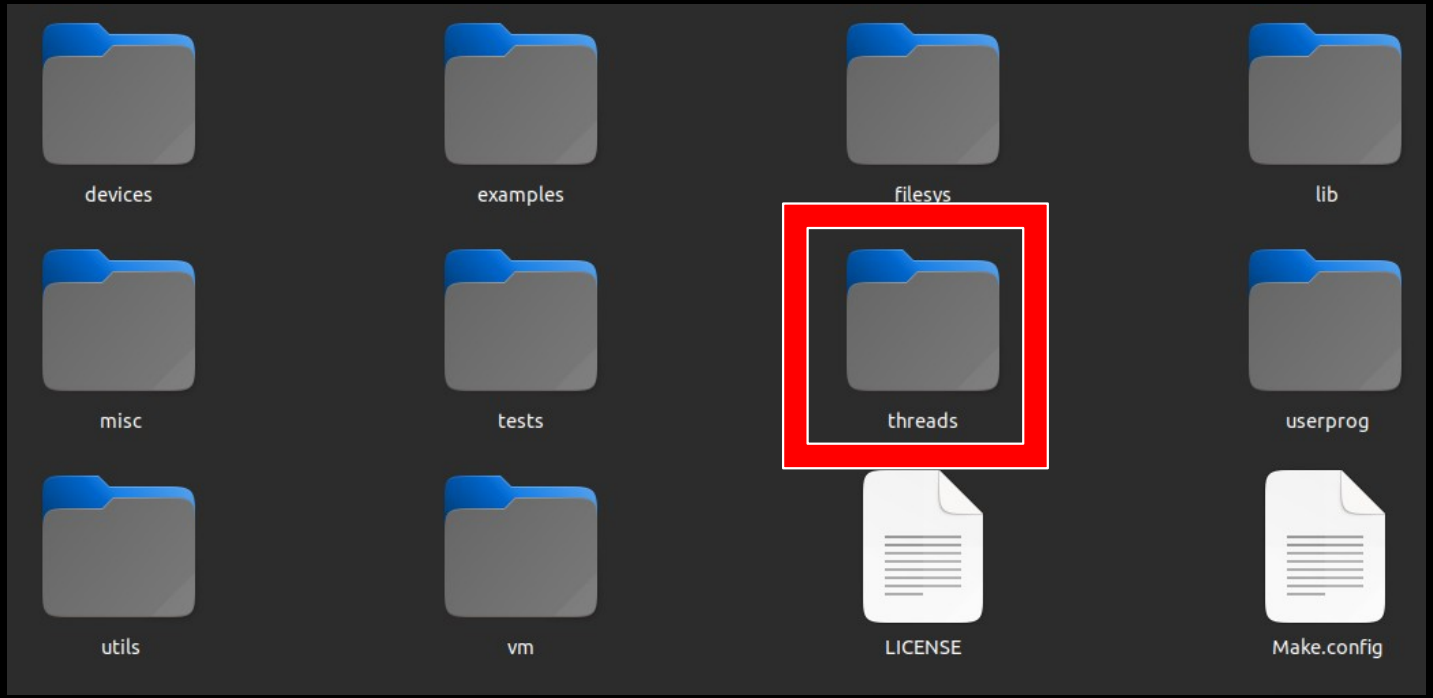
  - Uma estrutura list_elem
  - O tipo de estrutura que envolucra uma list_elem menber
  - Uma list_elem member

  e retorna a instância da estrutura que envolucra o list_elem menber:

- Exemplo de uso:

```
struct thread *t = list_entry (e, struct thread, allelem);
```

arquivos importantes



threads/sync.c

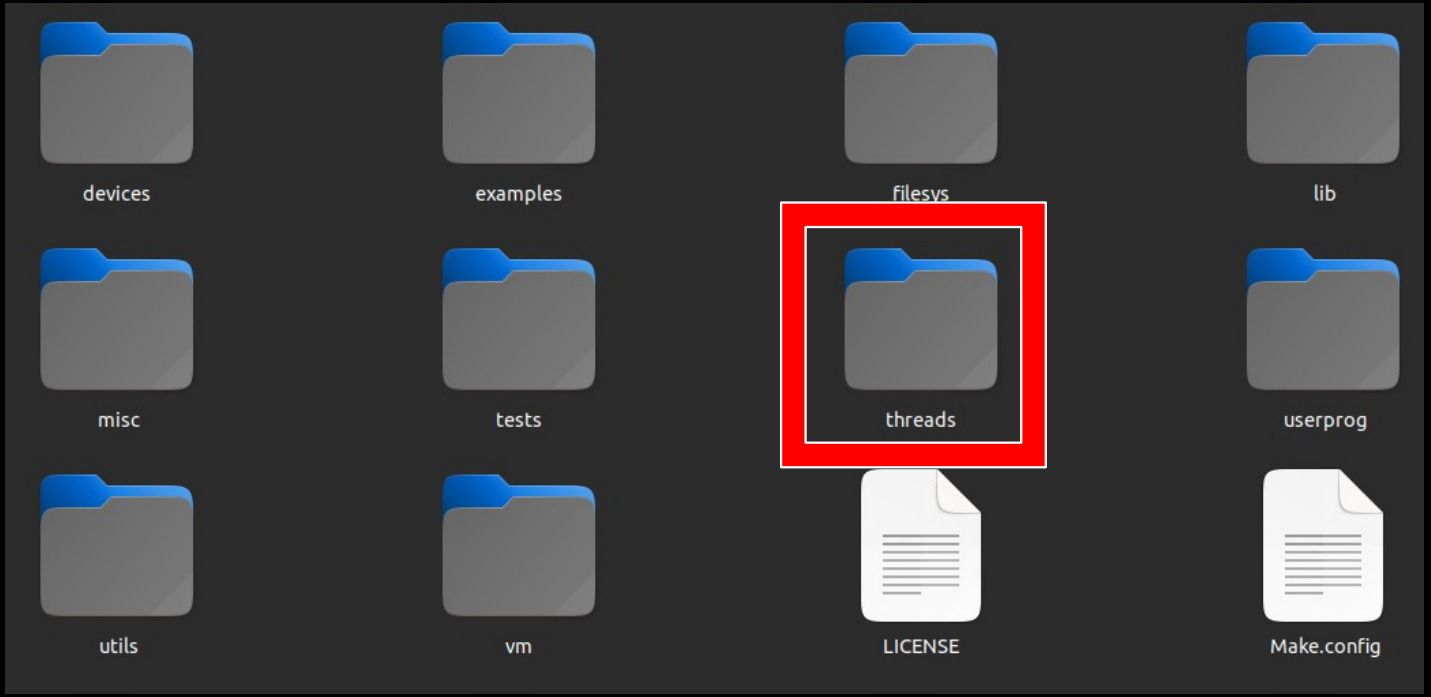
- Em sync.c estão as funções que lidam com locks, semáforos e monitores

threads/sync.c

- Em alguns testes será exigido atualização de threads bloqueada que usam blocks

```
228 void
229 lock_release (struct lock *lock)
230 {
231   ASSERT (lock != NULL);
232   ASSERT (lock_held_by_current_thread (lock));
233
234   lock->holder = NULL;
235   sema_up (&lock->semaphore);
236 }
237
```

arquivos importantes



threads/Thread.c

- É nesses arquivos que a maior parte do projeto será implementado
- Aqui estão as estruturas e funções para se manipular threads

threads/Thread.h

- A estrutura thread: fornece as informações básicas para se manipular threads como.
  - Tid: thread identity
  - Name
  - Allelem: elemento para a lista que contem todas as threads
  - Elem: elemento para a lista que contem as threads no estado ready

```
83 struct thread
84   {
85     /* Owned by thread.c. */
86     tid_t tid;
87     enum thread_status status;
88     char name[16];
89     uint8_t *stack;
90     int priority;
91     struct list_elem allelem;
92
93
94     struct list_elem elem;
95
```

threads/Thread.h

- É recomendável colocar o header de toda função feita em thread.c em thread.h

```
121
122 void thread_block (void);
123 void thread_unblock (struct thread *);
124
125 struct thread *thread_current (void);
126 tid_t thread_tid (void);
127 const char *thread_name (void);
128
129 void thread_exit (void) NO_RETURN;
130 void thread_yield (void);
131
```

thread.c: funções importantes

- thread_init( )
  - Inicia os locks
  - Inicia as listas
  - Cria a thread main
- É recomendável colocar/alterar aqui qualquer configuração inicial que for feita: por ex: criar listas, iniciar variáveis globais e etc...

```
89 void
90 thread_init (void)
91 {
92   ASSERT (intr_get_level () == INTR_OFF);
93
94   lock_init (&tid_lock);
95
96
97   list_init (&ready_list);
98
99
100
101   list_init (&all_list);
102
103   /* Set up a thread structure for the running thread. */
104   initial_thread = running_thread ();
105   init_thread (initial_thread, "main", PRI_DEFAULT);
106   initial_thread->status = THREAD_RUNNING;
107   initial_thread->tid = allocate_tid ();
108 }
```

thread.c: funções importantes

- thread_create( )
- Cria uma thread nova
- Chama a função init_thread( ) para inicializar a nova thread
- Chama a função thread_unblock( ) para desbloquear a nova thread

```
172 tid_t
173 thread_create (const char *name, int priority,
174                thread_func *function, void *aux)
175 {
176   struct thread *t;
177   struct kernel_thread_frame *kf;
178   struct switch_entry_frame *ef;
179   struct switch_threads_frame *sf;
180   tid_t tid;
181   enum intr_level old_level;
182
183   ASSERT (function != NULL);
184
185   /* Allocate thread. */
186   t = palloc_get_page (PAL_ZERO);
187   if (t == NULL)
188     return TID_ERROR;
189
190   /* Initialize thread. */
191   init_thread (t, name, priority);
```

thread.c: funções importantes

- thread_current( )
- Retorna a thread sendo executada atualmente

```
270 struct thread *
271 thread_current (void)
272 {
273   struct thread *t = running_thread ();
274
275   /* Make sure T is really a thread.
276      If either of these assertions fire, then your thread may
277      have overflowed its stack.  Each thread has less than 4 kB
278      of stack, so a few big automatic arrays or moderate
279      recursion can cause stack overflow. */
280   ASSERT (is_thread (t));
281   ASSERT (t->status == THREAD_RUNNING);
282
283   return t;
284 }
```

thread.c: funções importantes

- init_thread( )
- Chamada por thread_create para Inicializar uma nova thread
- É recomendável configurar novas threads aqui e não em thread_create
- É aqui que os parâmetros da thread são atribuídos
- Após definidos os parâmetros a thread é colocada na lista all_list

```
469 static void
470 init_thread (struct thread *t, const char *name, int priority)
471 {
472   ASSERT (t != NULL);
473   ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
474   ASSERT (name != NULL);
475
476   memset (t, 0, sizeof *t);
477   t->status = THREAD_BLOCKED;
478   strlcpy (t->name, name, sizeof t->name);
479   t->stack = (uint8_t *) t + PGSIZE;
480   t->priority = priority;
481   t->magic = THREAD_MAGIC;
482   list_push_back (&all_list, &t->allelem);
483 }
```

thread.c: funções importantes

- thread_block( )
- Bloqueia a thread atual
- Atualiza o status da thread atual para bloqueado e retira a thread atual da ready_list

```
228 void
229 thread_block (void)
230 {
231   ASSERT (!intr_context ());
232   ASSERT (intr_get_level () == INTR_OFF);
233
234   thread_current ()->status = THREAD_BLOCKED;
235   schedule ();
236 }
```

thread.c: funções importantes

- thread_unblock( )
- Recebe uma thread *t e desbloqueia ela
- Coloca ela novamente na ready_list

```
246 void
247 thread_unblock (struct thread *t)
248 {
249   enum intr_level old_level;
250
251   ASSERT (is_thread (t));
252
253   old_level = intr_disable ();
254   ASSERT (t->status == THREAD_BLOCKED);
255   list_push_back (&ready_list, &t->elem);
256   t->status = THREAD_READY;
257   intr_set_level (old_level);
258 }
```

thread.c: funções importantes

- thread_yield( )
- Inicia a troca da thread atual pela próxima thread a ser executada
- Coloca a thread atual no fim da ready_list
- Chama a função schedule() para realizar a troca

```
316 void
317 thread_yield (void)
318 {
319   struct thread *cur = thread_current ();
320   enum intr_level old_level;
321
322   ASSERT (!intr_context ());
323
324   old_level = intr_disable ();
325   if (cur != idle_thread)
326     list_push_back (&ready_list, &cur->elem);
327   cur->status = THREAD_READY;
328   schedule ();
329   intr_set_level (old_level);
330 }
```

thread.c: funções importantes

- schedule( )
- Troca a thread atual para a próxima thread a ser executada

```
565 static void
566 schedule (void)
567 {
568   struct thread *cur = running_thread ();
569   struct thread *next = next_thread_to_run ();
570   struct thread *prev = NULL;
571
572   ASSERT (intr_get_level () == INTR_OFF);
573   ASSERT (cur->status != THREAD_RUNNING);
574   ASSERT (is_thread (next));
575
576   if (cur != next)
577     prev = switch_threads (cur, next);
578   thread_schedule_tail (prev);
579 }
```

thread.c: funções importantes

- schedule( )
- O que define qual vai ser a próxima thread a ser executa é a next_ thread_to_run( )

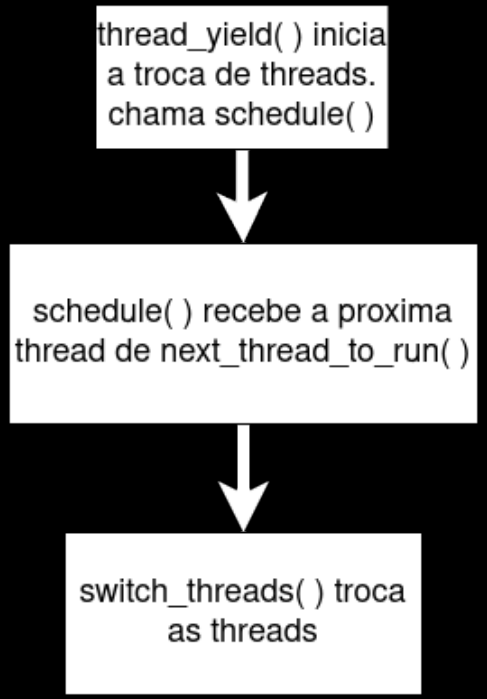
```
565 static void
566 schedule (void)
567 {
568   struct thread *cur = running_thread ();
569   struct thread *next = next_thread_to_run ();
570   struct thread *prev = NULL;
571
572   ASSERT (intr_get_level () == INTR_OFF);
573   ASSERT (cur->status != THREAD_RUNNING);
574   ASSERT (is_thread (next));
575
576   if (cur != next)
577     prev = switch_threads (cur, next);
578   thread_schedule_tail (prev);
579 }
```

thread.c: funções importantes

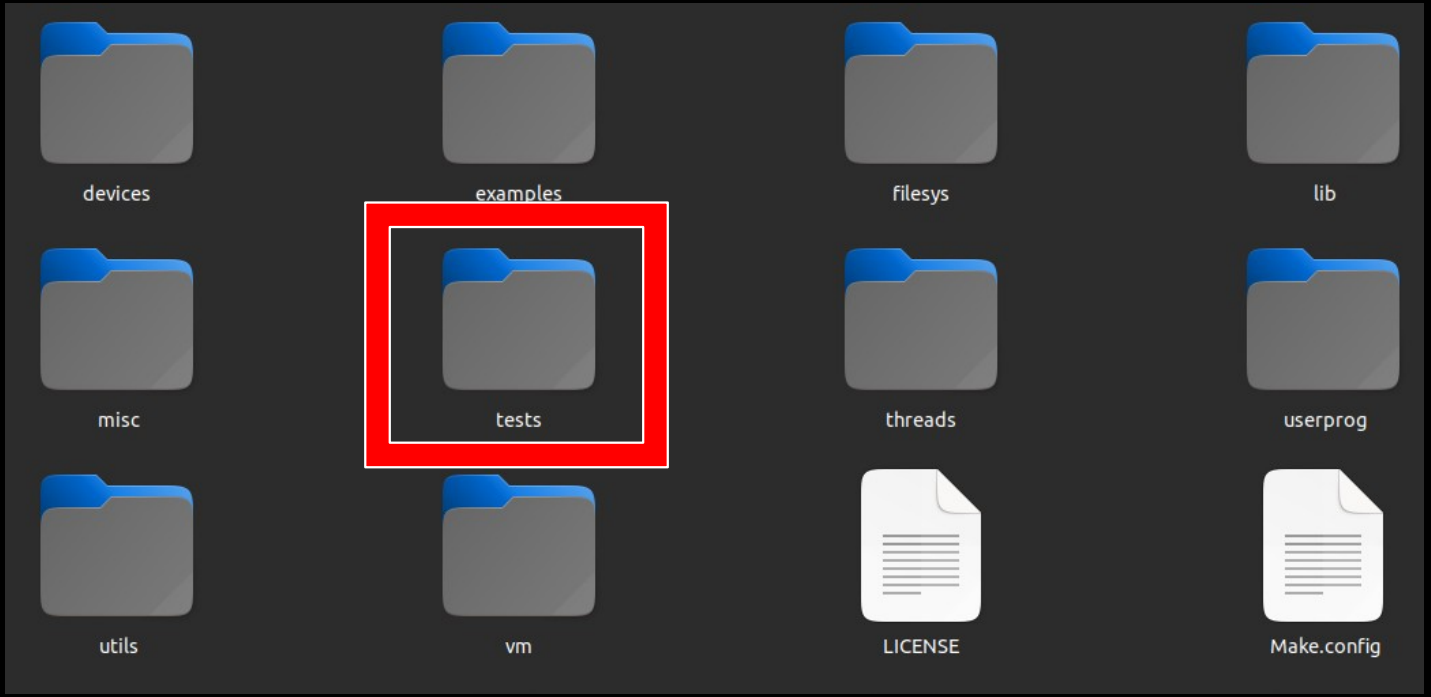
- next_thread_to_run( )
- Essa é a função que define qual a próxima thread a ser executada
- Nos a alteraremos para returna uma thread baseada em prioridade, tempo de execução ou qualquer outro parâmetro que desejarmos.

```
503 static struct thread *
504 next_thread_to_run (void)
505 {
506   if (list_empty (&ready_list))
507     return idle_thread;
508   else
509     return list_entry (list_pop_front (&ready_list), struct thread, elem);
510 }
```

thread.c: funções importantes



arquivos importantes



test/threads

- Nessa pasta estão todos os testes usados para avaliar o projeto
- É recomendavel ler os testes e entender o que eles fazem
- Não se deve alterar esses arquivos
- Usamos o comando make check para iniciar todos os testes

avaliação

- Apenas os testes do alarm-clock e advanced scheduler serão levados em conta na avaliação
- A nota será baseado na proporção de testes que passaram pelo total de testes

```
pass tests/threads/mlfqs-block
pass tests/threads/alarm-single
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
FAIL tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
FAIL tests/threads/priority-change
FAIL tests/threads/priority-donate-one
FAIL tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL tests/threads/priority-donate-lower
FAIL tests/threads/priority-fifo
FAIL tests/threads/priority-preempt
FAIL tests/threads/priority-sema
FAIL tests/threads/priority-condvar
FAIL tests/threads/priority-donate-chain
pass tests/threads/mlfqs-load-1
pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
13 of 27 tests failed.
```

obs

- Pintos como muitos outros OS não suporta operações de ponto flutuante
- Usamos uma biblioteca para converter float em int e vice versa como indica o tutorial do pintos.

```
 8
 9
10
11 #define FLOAT_SHIFT_AMOUNT 17
12 #define F mypow(14)
13 #define Q 14
14
15 int mypow(int p);
16 typedef int float_type;
17
18 #define FLOAT_CONST(A) ((float_type)(A * F))
19 #define FLOAT_ADD(A,B) (A + B)
20 #define FLOAT_ADD_MIX(A,B) (A + (B * F))
21 #define FLOAT_SUB(A,B) (A - B)
22 #define FLOAT_SUB_MIX(A,B) (A - (B * F))
23 #define FLOAT_MULT_MIX(A,B) (A * B)
24 #define FLOAT_DIV_MIX(A,B) (A / B)
25 #define FLOAT_MULT(A,B) ((float_type)(((int64_t) A) * B / F))
26 #define FLOAT_DIV(A,B) ((float_type)((((int64_t) A) * F) / B))
27 #define FLOAT_INT_PART(A) (A >> FLOAT_SHIFT_AMOUNT)
28 #define FLOAT_ROUND(A) (A >= 0 ? ((A + (1 * mypow(Q - 1))) / F ) \
29         : ((A - (1 * mypow(Q - 1))) / F))
30
31
32
33
```