CMPSC 473 Project 1

Generated by Doxygen 1.9.1

Chapter 1

CMPSC 473 Project 1

Jinyu Liu(liu.jinyu@psu.edu), Hongyu Guo(hbg5147@psu.edu)

1.1 Main Data Structures

1.1.1 Queues

The priority queue is the data structure all queues are based on. It is implemented on a linked list. It takes a priority argument in its enqueue function to allow flexibility. Entries with smaller priority numbers have higher priorities and ties are broken with the thread_id where lower ids have higher priority.

1.1.1.1 arrival_queue

The arrival_queue holds threads that will arrive in the future. Its entries are ordered by arrival time

1.1.1.2 ready_queue

The ready queue(s) holds threads that are waiting for the CPU. Its behavior changes depending on the type of CPU scheduler specified.

1.1.1.3 io_queue

The IO queue holds the threads that are performing or waiting to perform IO. It is FCFS and the head of the queue is the thread currently doing IO.

1.1.1.4 semaphore's blocked_queues

The blocked queue holds threads that are blocked by this semaphore. It is FCFS.

1.1.2 ProgramControlBlock

The ProgramControlBlock are the block that contain information about each thread in the system. Most importantly, the allow_to_run conditional that is used to control the thread's execution.

1.1.3 Semaphore

Contain the value of semaphore and queue of threads blocked by the semaphore.

1.2 Theory of Operation

The program mostly runs single-threaded with only one thread actively running at any time to make sure to race conditions occurs while operating on the various data structures and to make scheduling easy. Each thread is blocked using its own conditional in its ProgramControlBlock, this allows the schedule function to choose which thread to execute next by signaling only the conditional blocking the specific thread. Once one thread is unblocked it runs until another call to schedule where control is transferred to another thread.

1.2.1 The start_check() Function

It is the job of the start_check() function to get the disorganized threads into this state where only one is running at a time. It enqueues the calling thread onto the arrival queue and blocks the calling thread on its conditional. The once the last thread calls start_check, start_check will make the first call to schedule and set in motion the scheduling operations.

1.2.2 The schedule() Function

The schedule function is heart of the scheduler. It decides which thread to transfer control to. The main goal of the schedule function is ensure that the wall_clock is not advanced until every thread has completed all operations in the current tick. To do this it priorities operations that do not advance the wall clock. It choose threads in the following order of priority.

1. Any thread in the arrival_queue that arrived at the current time.
2. Any thread in the io_queue that completed IO at the current time.
3. The thread in the ready_queue that is next in line for the CPU.

Once the schedule function finds the next thread to run according to the priority order above, It will transfer control to that thread.

However it is possible that the CPU is idle and all threads are either blocked or will arrive in the future. In this case we would need to skip time forward. Since threads blocked by semaphores require other threads to execute to be come unstuck, we don't need to consider them when figuring out how much to skip time forward by. Thus we only need to consider the following.

- The next thread in the arrival_queue that will arrive at some point in the future
- The thread currently doing IO that will complete at some point in the future

If the schedule function can not find any thread to run at the current time, it will find the earlier of the two from the list above, skip time to when that event happens, and transfer control to that thread. At this point schedule should have found a thread to give control to, if it still can't find a thread, it means there is a deadlock or bug, so it will raise an assertion error and terminate the program.

1.2.3 The cpu_me() function

It puts the thread on the ready_queue and then calls schedule. Once the scheduler decides it should have the CPU it will increment the wall clock by 1 by calling tick(1) to represent the 1 unit of time passes while it is using the CPU. it will return the current wall_time as required.

1.2.4 The io_me() function

It puts the thread on the io_queue and then calls schedule. Once the IO is completed, the scheduler will release the thread, and it will return the current wall_clock.

1.2.5 The P() function

It will decrement the value of the specified semaphore. If the value is negative, it will put the calling thread on the blocked_queue of the semaphore and call schedule to transfer control to another thread. After either it deciding not to block or the thread gets released by a corresponding call to V(), it will return the current wall_time.

1.2.6 The V() function

It will increment the value of the specified semaphore. If there are thread being blocked, it will moved one thread from blocked_queue to arrival_queue, to be run next time schedule is called.

1.2.7 The end_me() function

It will call schedule to transfer control to another thread, unless it is last thread. If it is the last thread, it will clean up all memory allocations, and exit.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

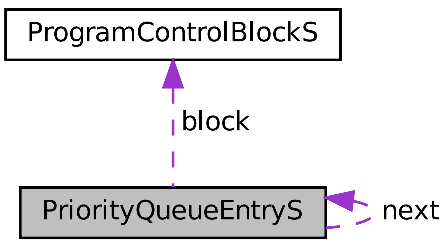
Data Structure Documentation

4.1 PriorityQueueEntryS Struct Reference

An Entry in the Priority Queue.

`#include <scheduler.h>`

Collaboration diagram for PriorityQueueEntryS:



Data Fields

- int priority

  priority of the current entry
- struct PriorityQueueEntryS * next

  pointer to data
- ProgramControlBlock * block

  pointer to next entry, or Null if this is the last entry

4.1.1 Detailed Description

An Entry in the Priority Queue.

The documentation for this struct was generated from the following file:

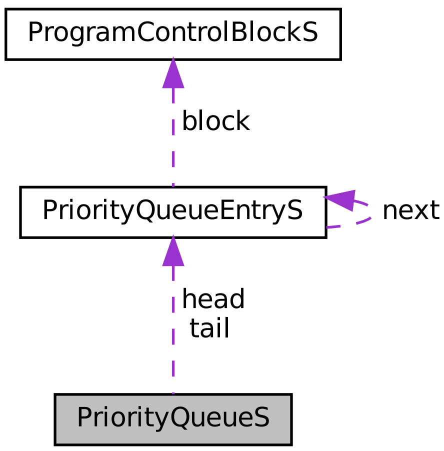
- scheduler.h

4.2 PriorityQueueS Struct Reference

An Priority Queue for ProgramControlBlocks.

`#include <scheduler.h>`

Collaboration diagram for PriorityQueueS:



Data Fields

- PriorityQueueEntry ∗ head
  head of the queue
- PriorityQueueEntry ∗ tail
  tail of the queue

4.2.1 Detailed Description

An Priority Queue for ProgramControlBlocks.

The documentation for this struct was generated from the following file:

- scheduler.h

4.3 ProgramControlBlockS Struct Reference

Holds information about each thread in the system.

```
#include <scheduler.h>
```

Data Fields

- int thread_id

  Unique id of the thread.
- int cpu_burst_length

  How long has the current cpu burst lasted.
- bool started

  Has the thread been started (specifically has it called startup_check() yet)
- int arrival_time

  The wall_clock time whe the thread requested the current activity.
- int remaining_time

  How much longer in ticks do the current activity require.
- int remaining_time_quanta

  How many ticks remain in the threads CPU time quanta allocation.
- int priority_level

  The priority level of the thread (only used in MLFQ)
- pthread_cond_t allow_to_run

  A conditional used to block and unblock execution of this thread.

4.3.1 Detailed Description

Holds information about each thread in the system.

The documentation for this struct was generated from the following file:

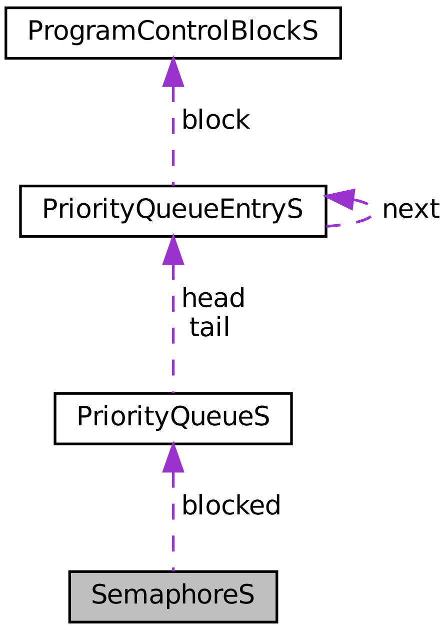
- scheduler.h

4.4 SemaphoreS Struct Reference

A Semaphore.

```
#include <scheduler.h>
```

Collaboration diagram for SemaphoreS:



Data Fields

- int value

  the current value of the semaphore
- PriorityQueue blocked

  a queue for holding threads blocks by this semaphore

4.4.1 Detailed Description

A Semaphore.

The documentation for this struct was generated from the following file:

- scheduler.h

4.5 thread_struct Struct Reference

Data Fields

- pthread_t **p_t**
- int **tid**
- char **line** [MAX_LINE_LEN]

The documentation for this struct was generated from the following file:

- main.c

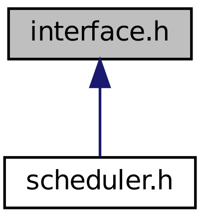
Chapter 5

File Documentation

5.1 interface.h File Reference

Scheduler Interface definitions.

This graph shows which files directly or indirectly include this file:



Macros

- #define **MAX_NUM_SEM** 10

Enumerations

- enum **sch_type** { **SCH_FCFS** = 0 , **SCH_SRTF** = 1 , **SCH_MLFQ** = 2 }

Functions

- void **init_scheduler** (enum sch_type scheduler_type, int thread_count)
- int **cpu_me** (float current_time, int tid, int remaining_time)
- int **io_me** (float current_time, int tid, int duration)
- int **P** (float current_time, int tid, int sem_id)
- int **V** (float current_time, int tid, int sem_id)
- void **end_me** (int tid)

5.1.1 Detailed Description

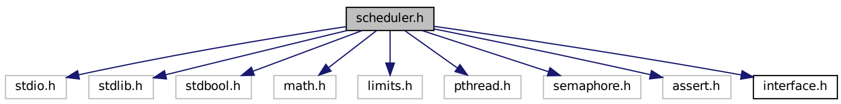
Scheduler Interface definitions.

5.2 scheduler.h File Reference

Scheduler function and data structures.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <limits.h>
#include <pthread.h>
#include <semaphore.h>
#include <assert.h>
#include "interface.h"
```

Include dependency graph for scheduler.h:



Data Structures

- struct ProgramControlBlockS

  Holds information about each thread in the system.
- struct PriorityQueueEntryS

  An Entry in the Priority Queue.
- struct PriorityQueueS

  An Priority Queue for ProgramControlBlocks.
- struct SemaphoreS

  A Semaphore.

Macros

- #define MLFQ_LEVELS sizeof(MLFQ_TIME_QUANTUM) / sizeof(int)

  The number of levels to use in MLFQ.

Typedefs

- typedef struct ProgramControlBlockS ProgramControlBlock

  Shorthand for struct ProgramControlBlockS.
- typedef struct PriorityQueueEntryS PriorityQueueEntry

  Shorthand for struct PriorityQueueEntryS.
- typedef struct PriorityQueueS PriorityQueue

  Shorthand for struct PriorityQueueS.
- typedef struct SemaphoreS Semaphore

  Shorthand for struct SemaphoreS.

Functions

- void PriorityQueue_init (PriorityQueue ∗queue)

  Initialize the PriorityQueue.
- void PriorityQueue_enqueue (PriorityQueue ∗queue, ProgramControlBlock ∗block, int priority)

  Enqueue a ProgramControlBlock onto the queue according to priority.
- ProgramControlBlock ∗ PriorityQueue_dequeue (PriorityQueue ∗queue)

  Returns the first entry from the queue and removes it from the queue.
- ProgramControlBlock ∗ PriorityQueue_peek (PriorityQueue ∗queue)

  Returns the first entry from the queue without removing it from the queue.
- bool PriorityQueue_is_empty (PriorityQueue ∗queue)

  Returns wether the queue is empty or not.
- void PriorityQueue_destroy (PriorityQueue ∗queue)

  Removes and deallocate all entries in the queue.
- void request_P (int arrival_time, int thread_id, int sem_id)

  Semaphore post implementation.
- void request_V (int arrival_time, int thread_id, int sem_id)

  Semaphore wait implementation.
- void request_cpu (int arrival_time, int thread_id, int remaining_time)

  Request to use the CPU.
- void request_io (int arrival_time, int thread_id, int duration)

  Request to use IO.
- void start_check (int arrival_time, int thread_id)

  Performs startup synchronization.
- void schedule (int thread_id)

  Performs scheduling operations.
- void tick (int num_ticks)

  Ticks the global clock, this should update current IO operations as well.
- void add_to_ready_queue (ProgramControlBlock ∗block)

  Add the specified ProcessControlBlock to the ready queue.
- ProgramControlBlock ∗ pick_from_ready_queue ()

  Gets the next process in line the be run in the ready queue.
- ProgramControlBlock ∗ pick_from_arrival_queue ()

  Gets the next process to arrive from the arrival queue.
- ProgramControlBlock ∗ pick_from_io_queue ()

  Gets the a process that has just completed its IO operation. If such a process is available.
- int next_arrival ()

  Gets how many ticks until the next process arrives.
- int next_io_queue_completion ()

  Gets how many ticks until the current IO operation is finished.
- void print_ready_queue ()

  DEBUG: print the ready queue.
- void print_future_queue ()

  DEBUG: print the future queue.
- void print_io_queue ()

  DEBUG: print the io queue.
- void clean_up ()

  Cleans up before exit.

Variables

- int num_processes

  total number of processes
- int num_started_processes

  number of started processes
- int num_running_processes

  number of running processes
- ProgramControlBlock * program_control_blocks

  Array holding ProgramControlBlocks of each thread.
- int wall_clock

  global simulation time
- pthread_mutex_t execution_lock

  Mutex to ensure only one thread is active at any time.
- PriorityQueue arrival_queue

  the arrival queue
- PriorityQueue * ready_queue

  the Ready queue(s)
- PriorityQueue io_queue

  the IO Queue
- Semaphore * semaphores

  all Semaphores
- enum sch_type scheduler_type

  The scheduler type.
- ProgramControlBlock * current_cpu_program

  The currently executing thread on the cpu.

5.2.1 Detailed Description

Scheduler function and data structures.

Author

Jinyu Liu (jzl6359@psu.edu)

Hongyu Guo (hbg5147@psu.eud)

Date

2022-10-10

5.2.2 Macro Definition Documentation

5.2.2.1 MLFQ_LEVELS

```
#define MLFQ_LEVELS sizeof(MLFQ_TIME_QUANTUM) / sizeof(int)
```

The number of levels to use in MLFQ.

Calculated from the length of the array MLFQ_TIME_QUANTUM

5.2.3 Function Documentation

5.2.3.1 add_to_ready_queue()

```
void add_to_ready_queue (
            ProgramControlBlock * block )
```

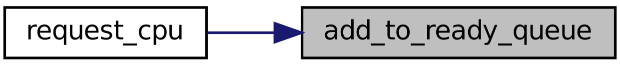
Add the specified ProcessControlBlock to the ready queue.

Exact implementation is scheduler_type dependent.

Parameters

block	the block to enqueue into the ready queue

Here is the caller graph for this function:

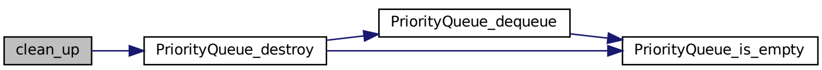


5.2.3.2 clean_up()

```
void clean_up ( )
```

Cleans up before exit.

Deallocates any memory allocated and deinitialize all data structures. This should be called by the last thread just before exiting end_me() Here is the call graph for this function:



5.2.3.3 next_arrival()

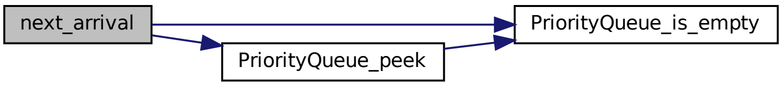
```
int next_arrival ( )
```

Gets how many ticks until the next process arrives.

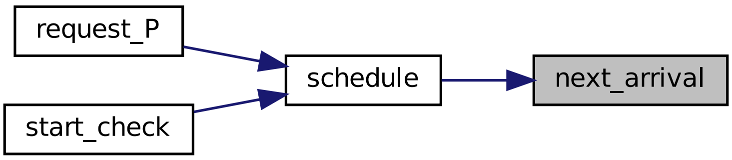
Returns

int ticks until the next process arrives, -1 if no process is available

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.3.4 next_io_queue_completion()

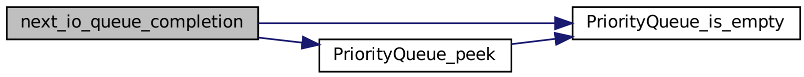
```
int next_io_queue_completion ( )
```

Gets how many ticks until the current IO operation is finished.

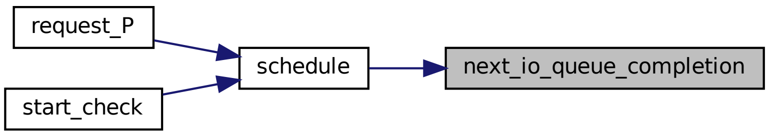
Returns

int ticks until the current IO operation finishes, -1 if no IO is in progress

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.3.5 pick_from_arrival_queue()

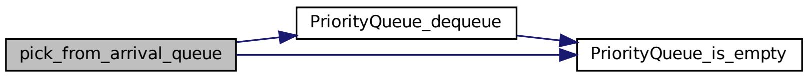
`ProgramControlBlock* pick_from_arrival_queue ( )`

Gets the next process to arrive from the arrival queue.

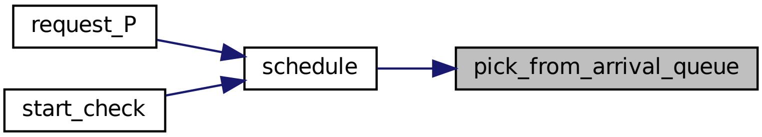
Returns

ProgramControlBlock∗ the next process to arrive from the arrival queue.

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.3.6 pick_from_io_queue()

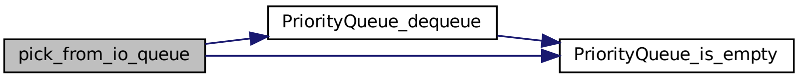
`ProgramControlBlock* pick_from_io_queue ( )`

Gets the a process that has just completed its IO operation. If such a process is available.

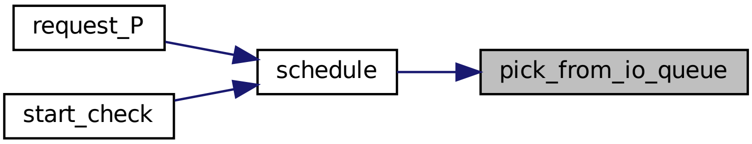
Returns

ProgramControlBlock∗ a process that just completed IO, if available. NULL otherwise

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.3.7 pick_from_ready_queue()

```
ProgramControlBlock* pick_from_ready_queue ( )
```

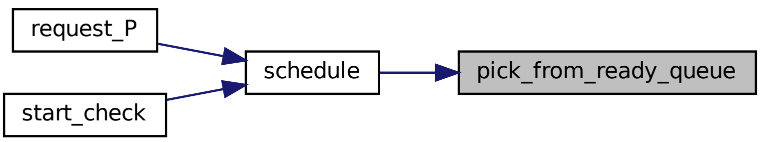
Gets the next process in line the be run in the ready queue.

The behavior of this function is dependent on the scheduler_type.

Returns

ProgramControlBlock*

Here is the caller graph for this function:



5.2.3.8 PriorityQueue_dequeue()

```
ProgramControlBlock* PriorityQueue_dequeue (
            PriorityQueue * queue )
```

Returns the first entry from the queue and removes it from the queue.

Do not call this function with an empty queue. Use PriorityQueue_is_empty() first to make sure queue is not empty.

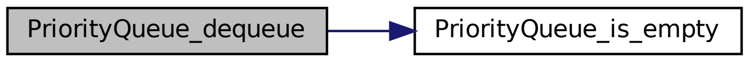
Parameters

queue	the queue to operate on

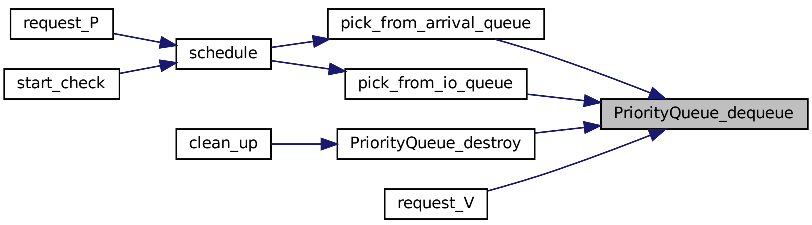
Returns

ProgramControlBlock∗ the first entry in the queue

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.3.9 PriorityQueue_destroy()

```
void PriorityQueue_destroy (
            PriorityQueue * queue )
```

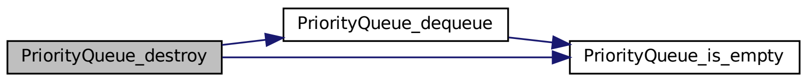
Removes and deallocate all entries in the queue.

Note this does not deallocate the blocks, just the PriorityQueueEntries

Parameters

queue	the queue to destroy

Here is the call graph for this function:



Here is the caller graph for this function:

5.2.3.10 PriorityQueue_enqueue()

```
void PriorityQueue_enqueue (
            PriorityQueue * queue,
            ProgramControlBlock * block,
            int priority )
```

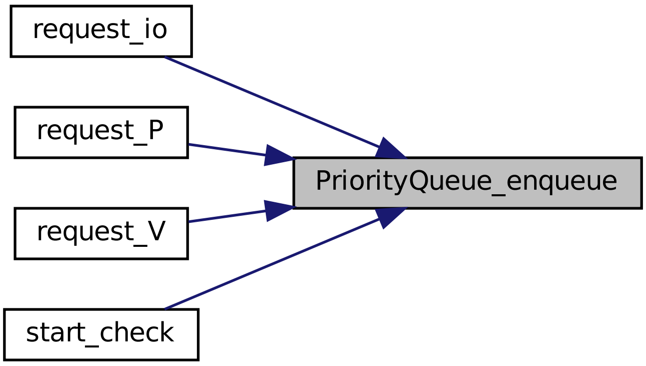
Enqueue a ProgramControlBlock onto the queue according to priority.

Ties are broken using the block's thread_id with lower thread ids getting higher priority.

Parameters

queue	the queue to operate on
block	the block to enqueue
priority	the priority to assign to this block

Here is the caller graph for this function:



5.2.3.11 PriorityQueue_init()

```
void PriorityQueue_init (
            PriorityQueue * queue )
```

Initialize the PriorityQueue.

This should be called before any other operations are made to the queue

Parameters

queue	queue to operate on

5.2.3.12 PriorityQueue_is_empty()

```
bool PriorityQueue_is_empty (
            PriorityQueue * queue )
```

Returns wether the queue is empty or not.

This should be called to ensure the queue is not empty before calling PriorityQueue_dequeue() or PriorityQueue_peek()

Parameters

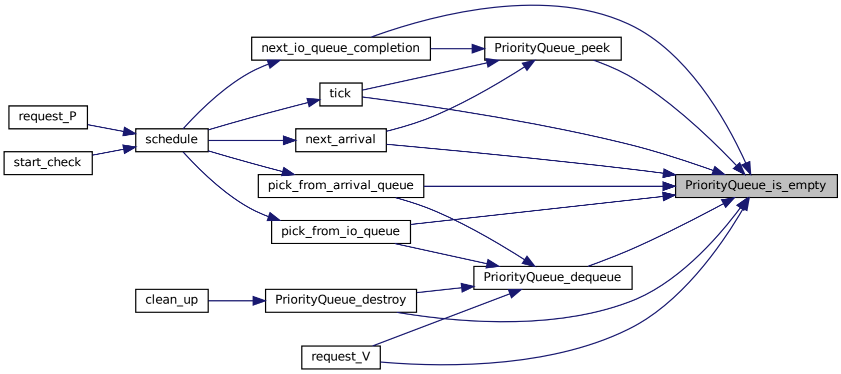
queue	the queue to operate on

Returns

true the queue is empty

false the queue has at least one entry

Here is the caller graph for this function:



5.2.3.13 PriorityQueue_peek()

```
ProgramControlBlock* PriorityQueue_peek (
            PriorityQueue * queue )
```

Returns the first entry from the queue without removing it from the queue.

Do not call this function with an empty queue. Use PriorityQueue_is_empty() first to make sure queue is not empty.

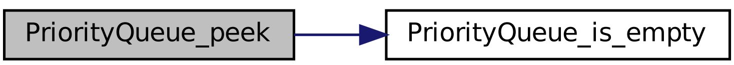
Parameters

queue	the queue to operate on

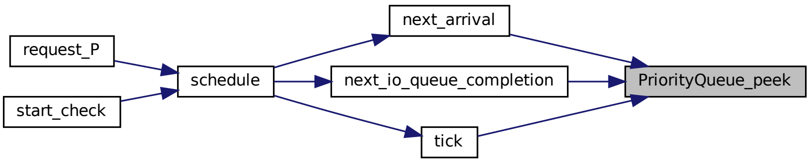
Returns

ProgramControlBlock∗ the first entry in the queue

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.3.14 request_cpu()

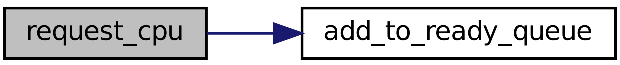
```
void request_cpu (
            int arrival_time,
            int thread_id,
            int remaining_time )
```

Request to use the CPU.

Parameters

arrival_time	wall_clock time of arrival
thread_id	process requesting the CPU
remaining_time	how many more consecutive ticks will this process need the cpu

Here is the call graph for this function:



5.2.3.15 request_io()

```
void request_io (
            int arrival_time,
            int thread_id,
            int duration )
```

Request to use IO.

Parameters

arrival_time	wall_clock time of arrival
thread_id	process requesting the IO
duration	how long will the IO operation take.

Here is the call graph for this function:

5.2.3.16 request_P()

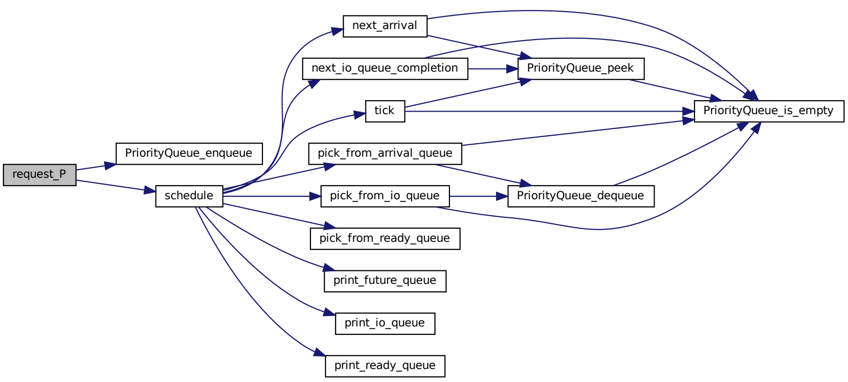
```
void request_P (
            int arrival_time,
            int thread_id,
            int sem_id )
```

Semaphore post implementation.

Parameters

arrival_time	The current wall_clock time
thread_id	The id of the calling thread
sem_id	The id of the semaphore to operate on

Here is the call graph for this function:



5.2.3.17 request_V()

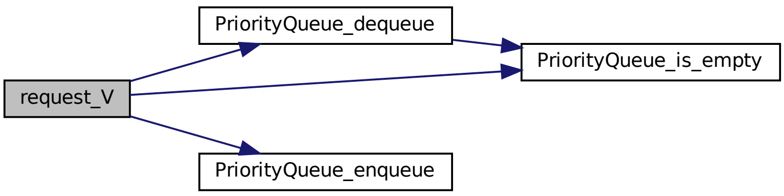
```
void request_V (
            int arrival_time,
            int thread_id,
            int sem_id )
```

Semaphore wait implementation.

Parameters

arrival_time	The current wall_clock time
thread_id	The id of the calling thread
sem_id	The id of the semaphore to operate on

Here is the call graph for this function:



5.2.3.18 schedule()

```
void schedule (
            int thread_id )
```

Performs scheduling operations.

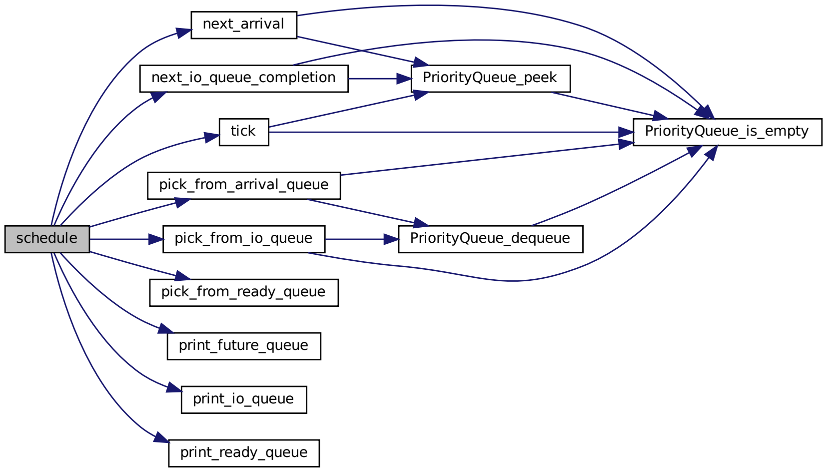
This program should be called in cpu_me, io_me, V, and end_me as long as there is still more operations to perform.

If it this decided that this thread should continue to execute then this function returns immediately, other wise it will signal the thread to execute next and then block itself until another call to schedule decides that this thread should run next.

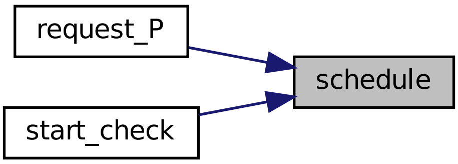
Parameters

thread←_id	the id of the calling thread

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.3.19 start_check()

```
void start_check (
            int arrival_time,
            int thread_id )
```

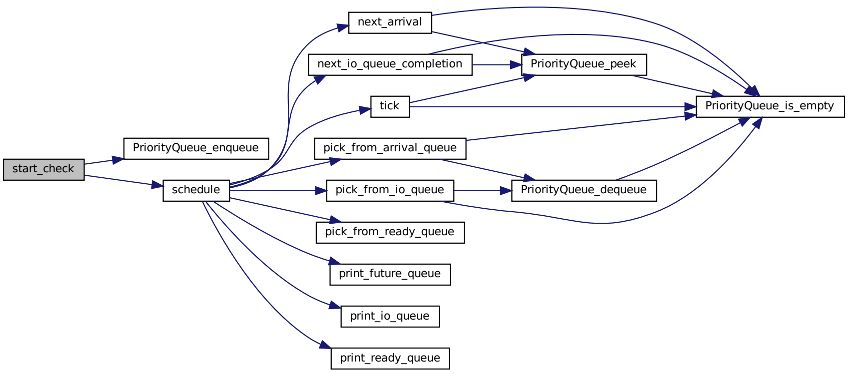
Performs startup synchronization.

Does nothing once the thread has been started. Call this before doing anything else in cpu_me(), io_me(), P(), V().

Parameters

arrival_time	current wall_time
thread_id	id of the calling thread

Here is the call graph for this function:



5.2.3.20 tick()

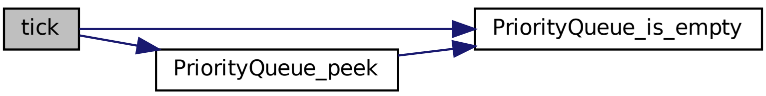
```
void tick (
            int num_ticks )
```

Ticks the global clock, this should update current IO operations as well.

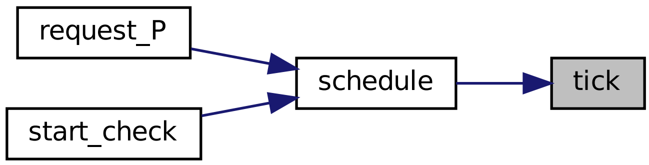
Parameters

num_ticks	number of ticks to skip forward

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.4 Variable Documentation

5.2.4.1 arrival_queue

```
PriorityQueue arrival_queue [extern]
```

the arrival queue

This queue holds all threads that will arrive in the future according to the curent simulation time(wall_clock).

pick_from_arrival_queue() and next_arrival() operate on this queue.

5.2.4.2 current_cpu_program

```
ProgramControlBlock* current_cpu_program [extern]
```

The currently executing thread on the cpu.

This is null if cpu is idle.

5.2.4.3 execution_lock

```
pthread_mutex_t execution_lock [extern]
```

Mutex to ensure only one thread is active at any time.

All threads acquires this in their first call to startup_check(). They will temporarily relinquish it when blocked by their respective ProgramControlBlockS::allowed_to_run conditional. Finally they release it near the end of end_me()

5.2.4.4 io_queue

```
PriorityQueue io_queue [extern]
```

the IO Queue

Holds threads that are preforming or waiting to perform io. The head is the thread that is currently performing IO while all other threads are waiting

5.2.4.5 ready_queue

```
PriorityQueue* ready_queue [extern]
```

the Ready queue(s)

A set of queues to hold threads waiting for their turn on the cpu. The exact operations of this queue is dependent on the scheduler type

add_to_ready_queue() and pick_from_ready_queue() operates on this queue

5.2.4.6 scheduler_type

```
enum sch_type scheduler_type [extern]
```

The scheduler type.

as given in init_scheduler()

5.2.4.7 wall_clock

```
int wall_clock [extern]
```

global simulation time

Do not change this value except using the tick() function

Index