Operating Systems

Complutense University of Madrid
2015-2016

Lab assignment #3

Processes and threads: scheduling and synchronization

J.C. Sáez

Contents

OS

Contents

Introduction

Objectives

- The main goal of the assignment is to learn how to implement different scheduling algorithms in a realistic simulation environment
  - The provided simulator is a multithreaded program
- Students will also have a chance to get familiar with various tools/mechanisms for the creation of multithreaded programs:
  - POSIX threads library
  - POSIX semaphores
  - Mutexes
  - Condition variables

Introduction

Reminder: Processes vs. threads

- 2 processes (parent - child) *do not share memory*
  - The OS creates a copy of the parent's memory image for the child
  - Each process accesses its own memory image
- 2 threads (from the same process) *share all memory* (except for the stack)

Do the exercises/Check the examples

- Help us to understand the subject ...
- ... and questions about them may be found in the test

Contents

OS

Command-line options

Terminal

```
debian:P3 osuser$ ./schedsim
No input file was provided
Usage: ./schedsim -i <input-file> [options]
debian:P3 osuser$ ./schedsim -h
List of options:
-h: Displays this help message
-n <cpus>: Sets number of CPUs for the simulator
-m <nsteps>: Sets the maximum number of simulation steps (default 50)
-s <scheduler>: Selects the scheduler for the simulation (default RR)
-d: Turns on debug mode
-p: Selects the preemptive version of SJF or PRIO (only if they are selected with -s)
-t <msecs>: Selects the tick delay for the simulator (default 250)
-q <quantum>: Set up the timeslice or quantum for the RR algorithm
-l <period>: Set up the load balancing period (specified in simulation steps)
-L: List available scheduling algorithms
debian:P3 osuser$ ./schedsim -L
Available schedulers:
RR
SJF
debian:P3 osuser$
```

Task input files: syntax

Examples

- Several sample input files can be found in the *examples* directory
- New input files can be built easily by following the syntax

Terminal

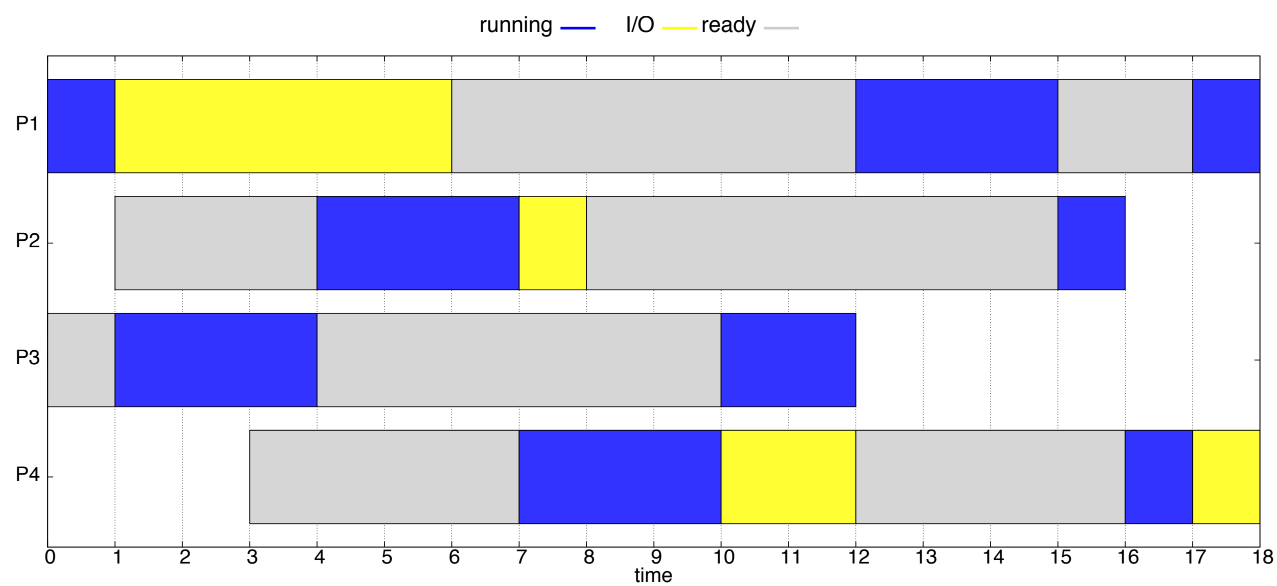
```
$ cat examples/example1.txt
P1 1 0 1 5 4
P2 1 1 3 1 1
P3 1 0 5
P4 1 3 3 2 1 1
```

- One line per task
- Column 1: task name
- Column 2: priority (*the lower the value* → *the higher the priority*)
- Column 3: arrival time
- Subsequent columns: *CPU burst - I/O burst - CPU burst - ...*

Example: RR on a system with one CPU

Terminal

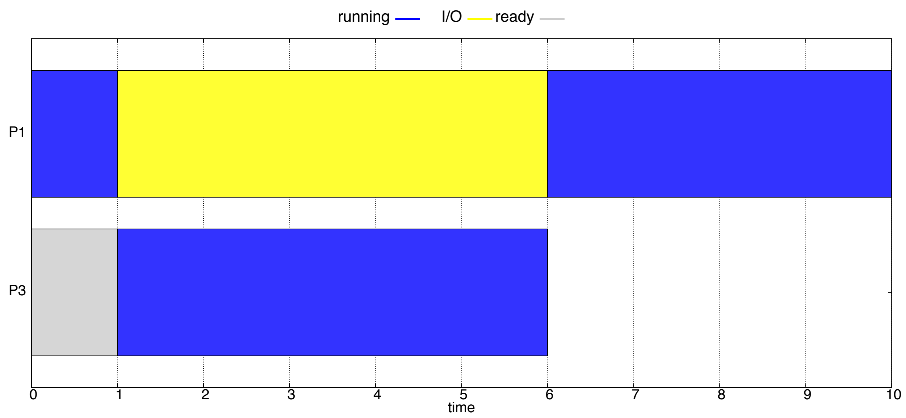
```
debian:P3 osuser$ ./schedsim -i examples/example1.txt
Statistics: task_name=P3 real_time=12 user_time=5 io_time=0
Statistics: task_name=P2 real_time=15 user_time=4 io_time=1
Statistics: task_name=P1 real_time=18 user_time=5 io_time=5
Statistics: task_name=P4 real_time=15 user_time=4 io_time=3
Simulation completed
Closing file descriptors...
debian:P3 osuser$ cd ../gantt-plot
debian:P3 osuser$ ./generate_gantt_chart ../schedsim/CPU_0.log
debian:P3 osuser$ cd -
debian:P3 osuser$ gnome-open CPU_0.eps
```

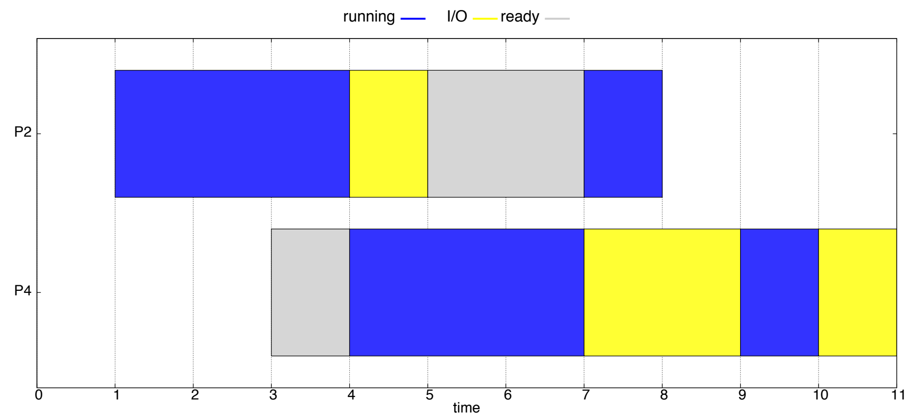


Example: RR on a system with 2 CPUs

Terminal

```
debian:P3 osuser$ ./schedsim -i examples/example1.txt -n 2
Statistics: task_name=P3 real_time=6 user_time=5 io_time=0
Statistics: task_name=P2 real_time=7 user_time=4 io_time=1
Statistics: task_name=P1 real_time=10 user_time=5 io_time=5
Statistics: task_name=P4 real_time=8 user_time=4 io_time=3
Simulation completed
Closing file descriptors...
debian:P3 osuser$ cd ../gantt-plot
debian:P3 osuser$ ./generate_gantt_chart ../schedsim/CPU_0.log
debian:P3 osuser$ ./generate_gantt_chart ../schedsim/CPU_1.log
debian:P3 osuser$ cd -
debian:P3 osuser$ gnome-open CPU_0.eps
debian:P3 osuser$ gnome-open CPU_1.eps
```





Example: Debugging mode

- The debugging mode (`-d` switch) enables to see what happens on each simulation cycle
  - The cycle period can be established as follows: "`-t <milisecs>`"

Terminal

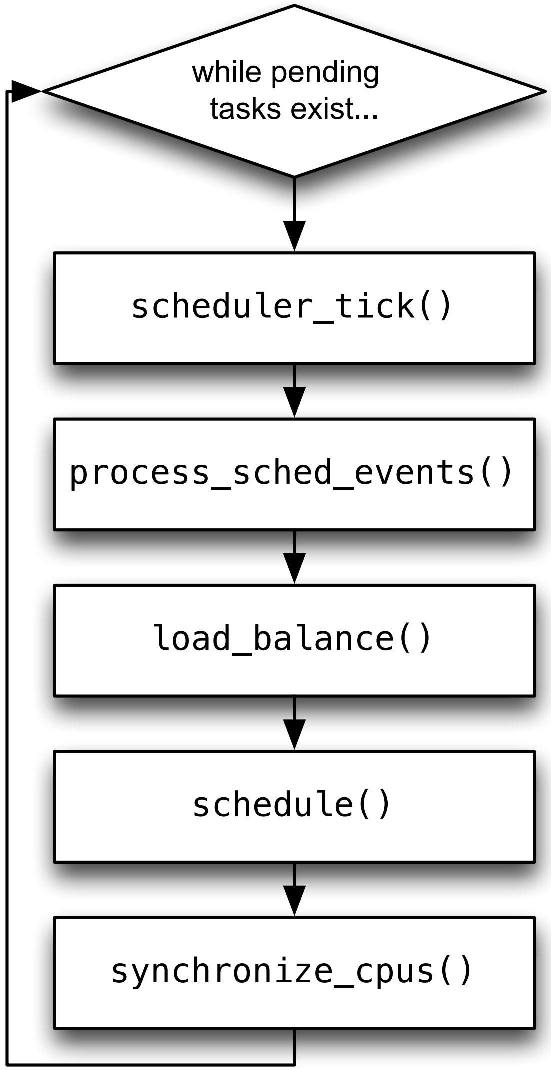
```
debian:P3 osuser$ ./schedsim -i examples/example1.txt -d -t 1000
==== TASK P1 ===
Priority: 1
Arrival time: 0
Profile: [ 1 5 4 ]
=================
==== TASK P2 ===
Priority: 1
Arrival time: 1
Profile: [ 3 1 1 ]
...
Scheduler initialized. Press ENTER to start simulation.

CPU 0:(t0): New task P1
CPU 0:(t0): New task P3
CPU 0:(t0): P1 running
CPU 0:(t1): Task P1 goes to sleep until (t6)
CPU 0:(t1): New task P2
CPU 0:(t0): Context switch (P1)<->(P3)
CPU 0:(t1): P3 running
...
```

Design of the scheduling simulator

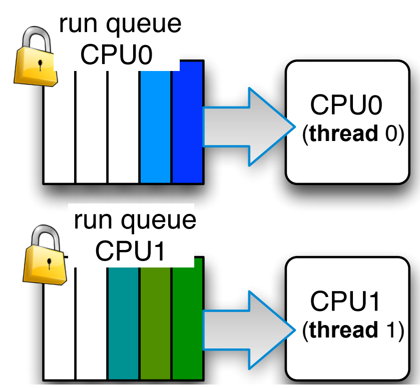
Features 2 components

1. Generic scheduler (`sched.c`)
   - Performs generic actions on each simulation cycle
     - Balances the load across CPUs
     - Updates tasks' state as well as the real time, I/O time, ...
2. Scheduling classes
   - 2 classes (RR and SJF) in the initial version of the simulator
     - Implemented in the `sched_rr.c` and `sched_sjf.c` files
   - Each class implements a specific scheduling algorithm
     1. Selects the next task to run
     2. Decides when to *preempt* (evict) a task from the CPU
     3. Manages per-CPU run queues
   - New classes (algorithms) can be easily added to the simulator
     - Each class implements the `struct sched_class` interface
     - The *active* scheduling class is selected when launching the simulator (Example: `./schedsim -s SJF -i examples/example1.txt`)

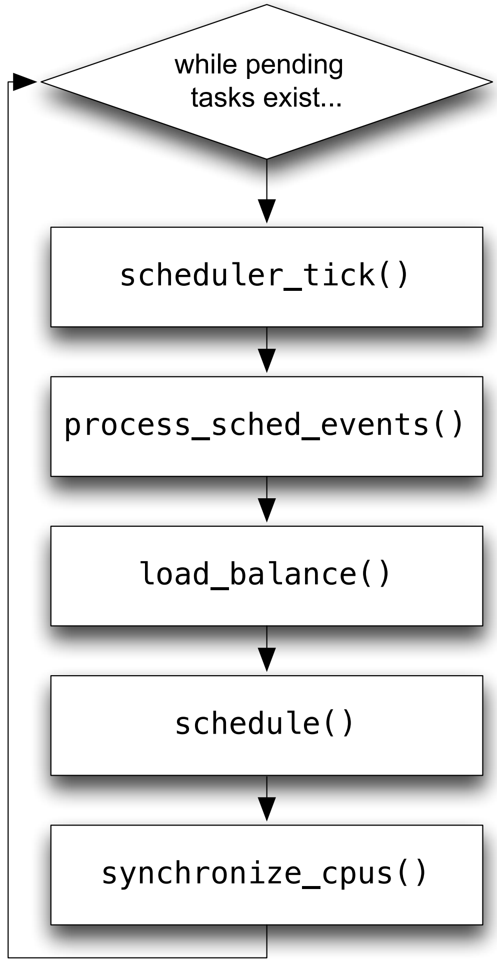
Simulation cycle (I)



- An actual thread exists for each simulated CPU
- Each thread executes this loop while pending tasks exist (`sched_cpu()`)
- One loop iteration is performed for each timer *tick* (simulation cycle)
  - Each CPU (thread) manages a queue with tasks in the ready state (aka *run queue*)
  - A lock is associated with each *run queue* to serialize accesses from multiple CPUs



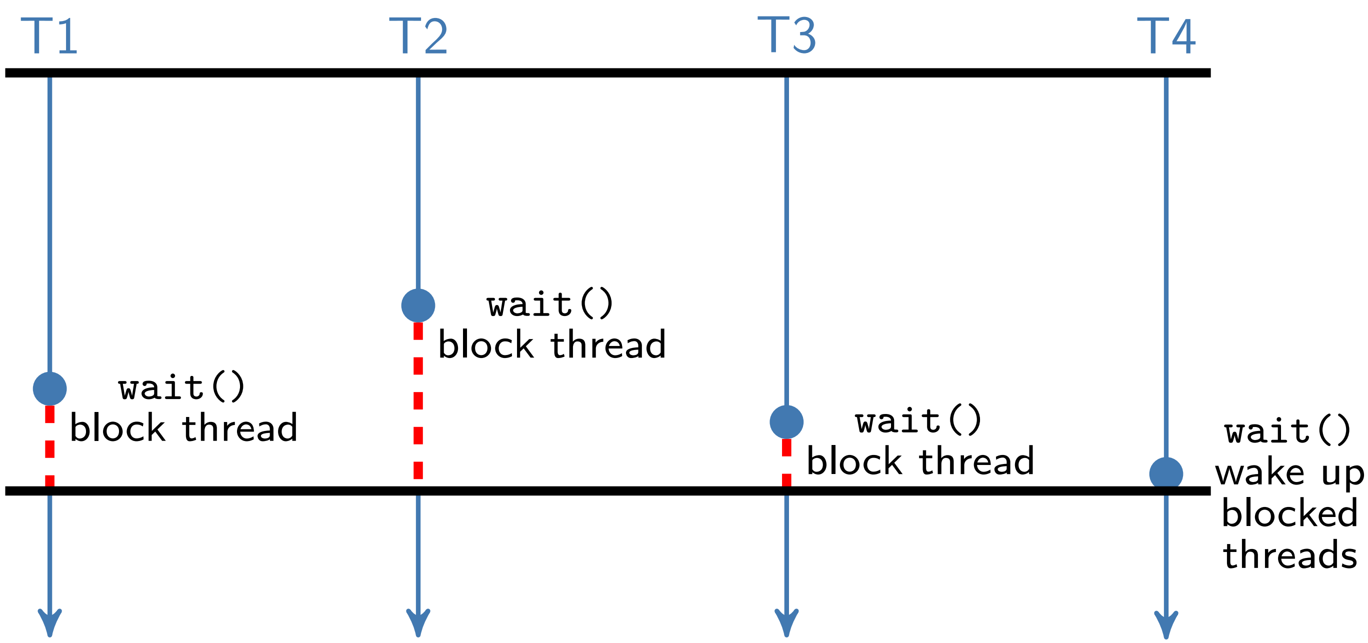
Simulation cycle (II)



1. *Tick* processing
   - Invoke the `task_tick()` operation of the SC
   - The class may request a preemption of the currently running task
2. Wake up blocked (or newly created) tasks. These tasks will be "ready" in the next simulation cycle
3. Load balance across CPUs if necessary
4. If the CPU is idle or a preemption was requested on this CPU:
   - Select the next task to run (`pick_next_task()` operation of the SC)
   - Is selected task $\neq$ currently running task $\rightarrow$ context switch
5. Wait until the remaining CPUs complete the simulation cycle
   - A synchronization barrier is used

Synchronization barrier

- Synchronization mechanism with an atomic operation: **`wait()`**
  - In creating a barrier, we need to specify how many threads will synchonize with it
- All threads invoke **`wait()`** to synchronize with one another at a given point in the application's code



Implementing synchronization barriers

- By default, the simulator uses the POSIX threads' barrier implementation (`pthread_barrier_t`)
- As part of the assignment, students will have to create an alternative implementation for the barrier

Possible implementation

- 2 Counters
  - `max_threads`: # of threads that use the barrier
  - `nr_threads_arrived` : # of threads that arrived at the barrier
- 1 Mutex (to serialize access to counters)
- 1 Condition variable (to block threads)

Implementing synchronization barriers

```
typedef struct {
    pthread_mutex_t mutex; /* Barrier lock */
    pthread_cond_t cond; /* Condition variable where threads remain blocked */
    int nr_threads_arrived; /* Number of threads that reached the barrier */
    int max_threads;     /* Number of threads that synchronize at the barrier.
                            (This value is set up upon barrier creation, and must
                                 not be modified afterwards) */
}custom_barrier_t;
```

Potential deadlock if the following events occur

1. `max_threads - 1` threads are blocked in the barrier (`wait()`)
2. The last thread *LT* arrives at the barrier:
   - prepare barrier for the next invocation of `wait()`:
     `nr_threads_arrived=0;`
   - wake up blocked threads
3. *LT* invokes `wait()` again and updates `nr_threads_arrived` before that all awaken threads have a chance to return from `wait()`

Implementing synchronization barriers

- To overcome this issue, the barrier implementation must distinguish between "even" and "odd" barriers
  - To make this happen, a `cur_barrier` field (0 or 1) and private counters for each case (even and odd) are maintained
  - The last thread that reaches the barrier must update `cur_barrier`

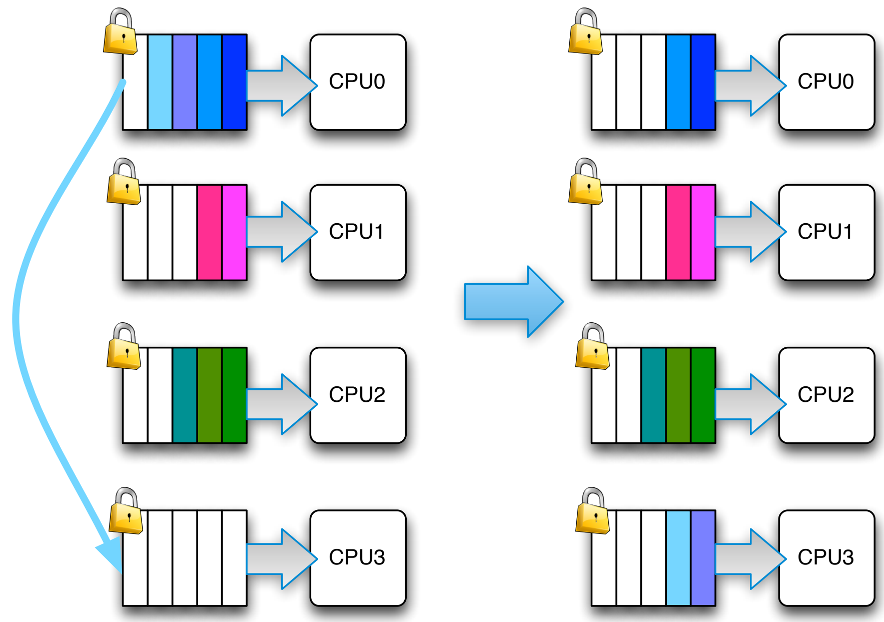
Robust implementation

```
typedef struct {
   pthread_mutex_t mutex; /* Barrier lock */
   pthread_cond_t cond; /* Condition variable where threads remain blocked */
   int nr_threads_arrived[2]; /* Number of threads that reached the barrier.
                               [0] Counter for even barriers, [1] Counter for odd
                                   barriers */
   int max_threads;     /* Number of threads that synchronize with the barrier.
                               (This value is set up upon barrier creation, and must
                                   not be modified afterwards) */
   unsigned char cur_barrier; /* Field to indicate whether the current barrier is
          an even (0) or an odd (1) barrier */
}sys_barrier_t;
```

Load balancing

- Load balancing is done in a distributed fashion from each CPU
  - Invoked periodically or when CPU is idle
  - `load_balance()` function

- The thread in the highest loaded CPU attemps to move tasks to the least loaded CPU
- The thread in the least loaded CPU attemps to *steal* tasks from the highest loaded CPU



- It may happen in parallel: potential *deadlock*
  - Specific solution, similar to the *Dining Philosophers* problem
  - The run queue lock of the CPU with the highest number is always acquired first

Task descriptor

task_t

```
typedef struct{
    int task_id;
    char task_name[MAX_TASK_NAME];
    exec_profile_t task_profile; /* Task behavior */
    int prio;                   /* Priority */
    task_state_t state;         /* Task state */
    ...
    int runnable_ticks_left;    /* Number of ticks the task
                                   has to complete till blocking or exiting */
    ...
    bool on_rq;                /* flag to indicate if the task is on the rq or not */
    unsigned long flags;        /* generic flags field */
    void* tcs_data;            /* Pointer enabling a scheduling class
                                         to store private data if needed */
    ...
}task_t;
```

Flags associated with a task (sched.h)

```
#define TF_IDLE_TASK 0x1 /* Active for the idle task */
/* Enable to indicate that the task must be inserted at the beginning
   of the task list rather than at the end */
#define TF_INSERT_FRONT 0x2
```

run queue (one for each CPU)

runqueue_t

```
typedef struct{
    slist_t tasks;     /* runnable task queue (doubly-linked list) */
    task_t* cur_task;   /* Pointer to the currently running task */
    task_t idle_task; /* This CPU's idle task */
    bool need_resched;  /* This flag must be set to TRUE when the
                           sched class wants to preempt the current
                               task */
    int nr_runnable;     /* Number of runnable task in this CPU
                           -> Note that current is not on the RQ */
    int next_load_balancing;
    void* rq_cs_data;
    pthread_mutex_t lock; /* Runqueue lock*/
}runqueue_t;
```

Doubly-linked lists

Implementation of task lists (slist_t)

```
/* Basic operations */
void* head_slist (slist_t* slist); /* Returns first item on the list */
void* tail_slist (slist_t* slist); /* Returns last item on the list */
int is_empty_slist (slist_t* slist); /* Returns !=0 if list is empty */
int size_slist (slist_t* slist);  /* Returns number of items on the list */
void remove_slist (slist_t* slist, void* item); /* Removes item from the list */
void insert_slist (slist_t* slist, void* item); /* Inserts item on the list */

/* Operations on sorted lists
 * (a comparison function must be passed as the third parameter.
 *  Check example in sched_sjf.c)
 */
void sorted_insert_slist(slist_t* slist, void* object, int ascending,
       int (*compare)(void*,void*));
void sorted_insert_slist_front(slist_t* slist, void* object, int ascending,
    int (*compare)(void*,void*));
```

Scheduling class interface

sched_class

```
typedef struct sched_class {
   /* Initialization/destruction (usually blank) */
   void (*sched_init)(void);
   void (*sched_destroy)(void);
   /* Invoked when creating a new task */
   void (*task_new)(task_t* t);
   /* Invoked when a task completes */
   void (*task_free)(task_t* t);
   /* Returns an dequeues from the run queue the next task to run on the given CPU.
      If the run queue is empty, it returns NULL. */
   task_t* (*pick_next_task)(runqueue_t* rq,int cpu);
   /* Invoked to insert a task into the queue
     - if task just woke up o just entered the system (runnable==0)
         -> then update nr_running field in the run queue
   */
   void (*enqueue_task)(task_t* t,int cpu, int runnable);
   /* Tick processing for the currently running task (T)
     - If necessary, it triggers T's preemption (rq->need_resched=TRUE;)
      * In doing so, the generic scheduler will then invoke the pick_next_task()
           operation
     - If T goes to sleep or completes, task_tick() updates the number of runnable
           tasks assigned to the CPU
   */
   void (*task_tick)(runqueue_t* rq,int cpu);
   /* Returns and dequeues a task from this CPU (to be migrated to another CPU) */
   task_t* (*steal_task)(runqueue_t* rq,int cpu);
}sched_class_t;
```

Implementing a new scheduling algorithm

Steps to add a new scheduling algorithm

1. Implement new scheduler in a new .c file
   - File name: `sched_<scheduler_name>.c`
   - Implement the interface of a scheduling class (`sched_class`)
2. Modify Makefile to compile the new .c file
3. Register new scheduler in `sched.h`

Example: Creating the FCFS scheduler (1/4)

Add new `sched_fcfs.c` file

```
#include <sched.h>

static task_t* pick_next_task_fcfs(runqueue_t* rq,int cpu) { … }

static void enqueue_task_fcfs(task_t* t,int cpu, int runnable) { … }

static void task_tick_fcfs(runqueue_t* rq,int cpu) { … }

static task_t* steal_task_fcfs(runqueue_t* rq,int cpu) { … }


/* Instantiante the interface:
   operation=associated-function
*/
sched_class_t fcfs_sched={
    .pick_next_task=pick_next_task_fcfs,
    .enqueue_task=enqueue_task_fcfs,
    .task_tick=task_tick_fcfs,
    .steal_task=steal_task_fcfs,
};
```

Example: Creating the FCFS scheduler (2/4)

Modify Makefile to compile `sched_fcfs.c`

```
TARGET=schedsim
SOURCES=main.c sched.c slist.c barrier.c \
       sched_rr.c sched_sjf.c sched_fcfs.c

OBJECTS=$(patsubst %.c,%.o,$(SOURCES))
MY_INCLUDES=.
HEADERS=$(wildcard $(MY_INCLUDES)/*.h)
OS=$(shell uname)
LDFLAGS=-lpthread
#CFLAGS=-g -Wall
CFLAGS=-g -Wall -DPOSIX_BARRIER


...
```

Example: Creating the FCFS scheduler (3/4)

Register new scheduler in sched.h

```
/* Scheduling class descriptors */
extern sched_class_t rr_sched;
extern sched_class_t sjf_sched;
extern sched_class_t fcfs_sched;

/* Numerical IDs for the available scheduling algorithms */
enum {
    RR_SCHED,
    SJF_SCHED,
    FCFS_SCHED,
    NR_AVAILABLE_SCHEDULERS
};

typedef struct sched_choice {
    int sched_id;
    char* sched_name;
    sched_class_t* sched_class;
}sched_choice_t;

/* This array contains an entry for each available scheduler */
static const sched_choice_t available_schedulers[NR_AVAILABLE_SCHEDULERS]={
    {RR_SCHED,"RR",&rr_sched},
    {SJF_SCHED,"SJF",&sjf_sched},
    {FCFS_SCHED,"FCFS",&fcfs_sched}
};
```

Example: Creating the FCFS scheduler (4/4)

- Retrieve the list of available schedulers (with the -L option) to make sure that the scheduler has been registered correctly

Terminal

```
debian:P3 osuser$ make clean
...
debian:P3 osuser$ make
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c main.c -o main.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c sched.c -o sched.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c slist.c -o slist.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c barrier.c -o barrier.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c sched_rr.c -o sched_rr.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c sched_sjf.c -o sched_sjf.o  -Wall
gcc  -g -Wall -DPOSIX_BARRIER  -I. -c sched_fcfs.c -o sched_fcfs.o  -Wall
gcc  -o  schedsim main.o sched.o slist.o barrier.o sched_rr.o sched_sjf.o
 sched_fcfs.o -lpthread
debian:P3 osuser$ ./schedsim -L
Available schedulers:
RR
SJF
FCFS
debian:P3 osuser$
```

Contents

Mandatory part

Changes in the simulator

1. Create the **FCFS** scheduler (non-preemptive)
   - Must be implemented in a new file: `sched_fcfs.c`
   - Very similar implementation as that of RR (FCFS + *timeslices*)
2. Create a **preemptive priority-based scheduler**
   - Must be implemented in a new file: `sched_prio.c`
   - Similar implementation as that of the preemptive SJF scheduler (`sched_sjf.c`)
3. Implement a synchronization barrier using mutexes and condition variables
   - Complete the implementation in the `barrier.c` file (`sys_barrier_init()`, `sys_barrier_destroy()` and `sys_barrier_wait()` functions in the `#else` code path)
   - Modify the *Makefile* to avoid the definition of the `POSIX_BARRIER` macro

Mandatory part (script)

Shell script `test.sh`

- The script will simulate a given input file under all the implemented schedulers and all possible CPU counts (until a given maximum CPU count)
  - The specification of the script can be found on the lab assignment's instruction sheet

Two additional features of the BASH shell must be used

1. `for` loops (check syntax in the slides "Introduction to the BASH shell")
2. "`read`" builtin command to read a line from the standard input and store it in a variable

Terminal

```
debian:P3 osuser$ read variable
line of text typed with the keyboard
debian:P3 osuser$ echo $variable
line of text typed with the keyboard
debian:P3 osuser$
```

Assignment submission

- Deadline: January 8, 8:55am
- To hand in an assigment for this course, one ".zip" or ".tar.gz" archive must be uploaded to the Virtual Campus
  - The archive must include all the necessary files to build the program (source + Makefile).
  - Don't forget to run "make clean" before generating the archive
  - File name:

    L<lab_number>_P<PC_number>_A<assignment_number>.tar.gz

Contents of the compressed file (.zip or .tar.gz)