Literately Programming a Round Robin Scheduler

Hans-Georg Eßer (*h.g.esser@gmx.de*)
Universität Mannheim

August 24, 2008

Contents

This is my first attempt at "Literate Programming" [Knu84, Ram94], using `noweb` (`http://www.cs.tufts.edu/~nr/noweb/`). I have fully documented my Python implementation of a Round Robin scheduler which I used in the lab work of my Operating Systems lectures in the summer term 2008.

1 A Process Execution Simulator

The program reads a configuration file that holds information on processes for a virtual system—each line of the configuration file describes one process. A sample line looks like this:

```
3:2,5,1,5,2,-1
```

and it has the following meaning: The process specified by this line should

- begin execution at system time 3 (`3:`),
- compute for 2 time units, then block (on I/O) for 5 time units, compute another 1 time unit, block for 5 time units, compute further 2 units (`2,5,1,5,2,`)
- and finally terminate (`-1`).

Several lines in the configuration file will thus declare the behavior of several processes, and it is the scheduler's task to execute these processes in some order, possibly in a preemptive manner.

The program ⟨*scheduler.py* 2⟩ consists of an initialization routine, a primary loop in which the scheduler selects and executes processes, and a closing display of execution statistics, and before those come some constant and variable definitions and function declarations:

2 ⟨*scheduler.py* 2⟩≡
  ⟨*python header* 18c⟩
  ⟨*user defined constants* 15b⟩
  ⟨*variable definitions* 3a⟩
  ⟨*function declarations* 3c⟩
  ⟨*main program* 18b⟩

1.1 Some data structures

The program uses some global variables which we initialize here:

- `tasks` is the process list; it can be regarded as a list of process control blocks (PCBs). What such a PCB will look like, becomes clear in the following section on initializing the process list. It begins life as an empty list.
- `current` holds the process ID (PID) of the current process. It is initialized as −1 (saying: there is no current process).
- `trace` hold a log of process execution; see definition of `log_to_trace` (page 11).
- `runqueue` and `blocked` are two queues (lists again) holding process IDs of processes which are currently ready/running or blocked. Processes which have not entered the system yet and processes who have left it (terminated) will occur in neither.
- `cputime` is the current time, the value of the virtual clock which starts at 0.
- `finished` is a boolean variable that says whether the simulator can stop or not.

3a ⟨*variable definitions* 3a⟩≡ (2) 7a ▷

```
tasks = []      # task list; list of PCBs
current = -1    # current process' ID
trace = []      # trace/log of process execution
runqueue = []   # runqueue (with process IDs)
blocked  = []   # blocked queue (with process IDs)
cputime = 0     # initialize the clock
finished = 0    # if 1, all processes have terminated
```

1.2 Initializing the process list

Let us start with the most simple code parts: parsing the configuration file and creating the data structures which hold this information.

The `init()` function will read the configuration data from a file whose name was supplied as a command line argument – thus it will need the `argv` function from the `sys` module. Since it will also check for a wrong filename, it additionally requires the `exit` function from the same module.

3b ⟨*init function* 3b⟩≡ (3c)

```
def init ():
  from sys import argv
  from sys import exit
  ⟨open and read configuration file 4a⟩
  ⟨parse configuration data 4b⟩
  return
```

Defines:
`init`, used in chunk 18b.

3c ⟨*function declarations* 3c⟩≡ (2) 6a ▷

```
⟨init function 3b⟩
```

where the configuration file is opened and then read with the standard Python file method `readlines()`:

4a ⟨*open and read configuration file* 4a⟩≡ (3b)
```
  try:
    filename = argv[1]
    f = open (filename, "r")
    lines = f.readlines()
    f.close ()
  except:
    print "Error: requires filename of the process config file"
    exit()
```

The whole opening, reading and closing is embedded in a `try` block so that any error occurring while trying to open or read the file will be caught; in that case a warning message is generated and the program terminates via the `exit()` call.

As a next step the read lines are parsed: The configuration strings are first split at the colon, so after the first `split()` the starting time and the rest are separated. Then the remaining string is split again, using the comma as delimiter symbol. Notice that initially all data are strings, so an explicit conversion to integers is necessary for each datum, using the `int()` function. After this decomposition the variable `behavior` contains a list of integers (including as last element the `-1` terminator. It is an error if a line does not end with `-1`, though this is not checked. Finally the function `create_process()` is called with the start time and the behavior variables as arguments—how `create_process()` treats these data will be shown in chunk ⟨*create process* 8a⟩.

4b ⟨*parse configuration data* 4b⟩≡ (3b)
```
  for l in lines:
    if l == "\n": return
    (starttime,times) = l[:-1].split(":")
    starttime = int(starttime)
    times = times.split(",")
    behavior=[]
    for t in times:
      behavior.append(int(t))
    create_process (starttime, behavior)
```
Uses `create_process` 8a.

Before we can look at the process creation, we have to talk about the internal data structures of the simple scheduler simulator. The task list `tasks` is a Python list that contains Python dictionaries[1], where each dictionary holds the complete information about one of the processes, whether it has not yet started execution, is in the mid of it, or has already terminated. Dictionaries can be augmented at any time, so there is no need to predefine or fill all possible elements. For the purpose of generating an initial process entry in the task list, the following fields in the dictionary will be sufficient:

- `starttime`: this is the starting time of the process. the lowest possible number is 0. If no other processes exist on the system at time t and the start time of a process is t, then the scheduler is expected to pick this process and execute its first "instruction" at time t. The start time is the first field of a configuration file line.
- `behavior`: this holds a list of iterating CPU (compute) and I/O (blocked) times – the first element after initialization is always a CPU time value. The meaning of this first value being 0 is that the process starts with an I/O cycle.[2] If this array starts with two 0 elements, it is considered a syntactically wrong entry; a user should always remove leading double-zero entries.
- `firstruntime`: the time, when the process was run for the first time – initialized to -1 for all new processes.
- `cputime`: amount of time the process has spent using the CPU.
- `iotime`: amount of time the process has spent waiting for I/O. (Notice that other processes may be executed during the waiting time, or the CPU may idle.) After the process terminates, the sum of CPU and I/O time will be the total time of process existence in the system.
- `status`: This is a status flag which will be explained in the next subsection.
- `usedquant`: This field holds the amount of time that the (currently active) process has spent using the CPU (sind its last selection by the scheduler).
- `endtime`: The time at which the last CPU cycle finished. Convention: A process that starts at time 0 and uses one unit of CPU time ends at time 0, not 1.

Fields of the directory are accessed through square brackets, where the field name has to be surrounded by double quotes ("): So the `starttime` field of the `task` dictionary is `task["starttime"]`.[3]

Since we attempt to treat the dictionary fields as private variables (in an object oriented sense), we define several functions for accessing (reading and modifying) these fields. In general, to read the *property* field of a process with PID *pid*, we will define

```
get_property (pid):  return tasks[pid]["property"]
```

and similarly for setting a field we use `set_...` procedures:

```
set_property (pid,value):  tasks[pid]["property"] = value
```

[1]see `http://docs.python.org/tut/node7.html`, section 5.5

[2]Whether this is conceptually useful or not, shall not matter for this purpose.

[3]It would be possible to omit the double quotes if each field name were treated as an integer constant, e. g. by defining `starttime=1`, `behavior=2` ..., and then using `task[starttime]` etc.

In some cases a simple increment procedure is defined that will first read a value, add 1, and then write it back:

inc_*property* (*pid*): set_*property* (*pid*, get_*property* (*pid*) + 1)

Last in the list is the function `dec_behavior_head` which takes the first element in the behavior list and substracts 1 (assuming the value is ≥ 1). The function `remove_behavior_head` removes the head element of the `behavior` list; it is called whenever a CPU or I/O phase ends and the process transitions to `S_BLOCKED` or `S_READY` state, respectively.

6a ⟨*function declarations* 3c⟩+≡ (2) ◁3c 6b▷
```
def set_behavior (pid, b_list):  tasks[pid]["behavior"] = b_list
def set_starttime (pid, t):      tasks[pid]["starttime"] = t
def set_cputime (pid, t):        tasks[pid]["cputime"] = t
def set_iotime (pid, t):         tasks[pid]["iotime"] = t
def set_status (pid, status):    tasks[pid]["status"] = status
def set_firstruntime (pid, t):   tasks[pid]["firstruntime"] = t
def set_endtime (pid, endtime):  tasks[pid]["endtime"] = endtime
def set_firstruntime (pid,t):    tasks[pid]["firstruntime"] = t
def set_usedquant (pid,t):       tasks[pid]["usedquant"] = t

def get_behavior (pid):    return tasks[pid]["behavior"]
def get_starttime (pid):   return tasks[pid]["starttime"]
def get_endtime (pid):     return tasks[pid]["endtime"]
def get_cputime (pid):     return tasks[pid]["cputime"]
def get_iotime (pid):      return tasks[pid]["iotime"]
def get_status (pid):      return tasks[pid]["status"]
def get_firstruntime(pid): return tasks[pid]["firstruntime"]
def get_usedquant(pid):    return tasks[pid]["usedquant"]

def inc_cputime (pid):   set_cputime (pid, get_cputime(pid)+1)
def inc_iotime (pid):    set_iotime (pid, get_iotime(pid)+1)
def inc_usedquant (pid): set_usedquant (pid, get_usedquant(pid)+1)

def dec_behavior_head (pid):    tasks[pid]["behavior"][0] -= 1
def remove_behavior_head (pid):
  set_behavior( pid, get_behavior(pid)[1:] )
```

We also define a simple function that returns the next available process ID – in our model process IDs start with 0, and each new process gets the next number as ID; so it is simply the current length of the process list. However we also provide a global variable `proccount` so that we need not calculate the list length.

6b ⟨*function declarations* 3c⟩+≡ (2) ◁6a 7c▷
```
def get_freepid():
  global proccount
  proccount+=1
  return proccount-1
```

As a side effect the `get_freepid()` function increases the number of processes – so it should immediately be followed by the creation of the belonging process.

The `proccount` variable has to be initialized at the program start:

7a ⟨*variable definitions* 3a⟩+≡ (2) ◁3a 7b▷
```
proccount = 0;   # number of processes in the system
```

1.2.1 Process status

As promised, we now talk about the process status. In our simple model a process may be in one of four states:

- active: The process is currently holding the CPU ressource.
- ready: The process waits for the CPU to become available (more precisely: for the scheduler to elect it).
- blocked: The process is currently waiting for the end of an I/O operation. After being unblocked it becomes ready immediately.
- done: The process has terminated.

We define integer constants for these four states which will be used throughout the program:

7b ⟨*variable definitions* 3a⟩+≡ (2) ◁7a
```
# status constants
S_ACTIVE=1
S_READY=2
S_BLOCKED=3
S_DONE=0
```

As a tool for the process list function to be defined later, we provide a simple function that translates the numbers into meaningful strings:

7c ⟨*function declarations* 3c⟩+≡ (2) ◁6b 8b▷
```
def statusname (i):
  try:
    return ["Done","Active","Ready","Blockd"][i]
  except:
    return "Error"
```

1.2.2 Process creation

Now we have all the tools and data structures we need to create a new process from a line in the configuration file. Remember that during the evaluation of these lines in ⟨*parse configuration data* 4b⟩, a function `create_process` is called several times, with two arguments: the starting time (an integer) and the process behavior, a list of computing and I/O wait times, ending with -1. We define `create_process` as follows:

8a ⟨*create process* 8a⟩≡ (8b)

```
def create_process( starttime, behavior ):
  pid = get_freepid()
  task={}; tasks.append ( task )  # empty task
  set_starttime( pid, starttime )
  set_behavior( pid, behavior )
  set_firstruntime( pid, -1 )     # has never run yet
  set_cputime( pid, 0 )
  set_iotime( pid, 0 )
  set_status( pid, S_READY )
  set_usedquant( pid, 0 )
  if behavior[0]==0:
    # process starts with I/O
    set_behavior( pid, get_behavior(pid)[1:] )
    set_status(pid, S_BLOCKED)
  set_endtime(pid,-1)  # process not finished yet
  return pid
```

Defines:
`create_process`, used in chunk 4b.

We have to add this new function to the list of function declarations:

8b ⟨*function declarations* 3c⟩+≡ (2) ◁7c 8c▷
⟨*create process* 8a⟩

1.3 Helper functions

Since the complete program will simulate the scheduling component of an operating system, some helper functions are needed, and we introduce them now.

First, we define an activator function which, given a process ID, will walk through the process list, searching for the currently active process and change its state from `S_ACTIVE` to `S_READY`; then it will set the state of the process with the given ID to `S_ACTIVE` (assuming it was `S_READY` before).

8c ⟨*function declarations* 3c⟩+≡ (2) ◁8b 9▷

```
def activate (pid):
  for t in tasks:
    if t["status"] == S_ACTIVE: t["status"] = S_READY
  tasks[pid]["status"] = S_ACTIVE
```

Defines:
`activate`, used in chunk 12.

Next, we declare a process list tool `ps` that enumerates all the currently existing processes – note that by "existing", we mean that start time of the process is $\leq$ the current CPU time. So while a process may be listed in the process list variable `tasks`, it need not appear in the output of `ps`.

9 ⟨*function declarations* 3c⟩+≡ (2) ◁8c 10▷

```
def ps ():
  global tasks,proccount,cputime,runqueue,blocked
  print "PID | Sta | End | CPU | I/O | Status | Verhalten"
  for pid in range(0,proccount):
    if cputime >= get_starttime(pid)-1:
      task = tasks[pid]
      print "%3d | %3d | %3d | %3d | %3d | %6s |" % (pid,
        get_starttime(pid), get_endtime(pid),
        get_cputime(pid), get_iotime(pid),
        statusname( get_status(pid) ) ),
      print task["behavior"]
  print "Runqueue:",runqueue,"  Blocked:",blocked
  print
  return
```

Defines:
`ps`, used in chunks 12 and 18b.

In the end, before the program terminates, it should print some information about the behavior of the process. It will use the following function `stats` for this purpose. The turnaround time (column `TurnAro`) is the difference end time (column `End`) − arrival time (column `Arrival`), where the arrival time is the first value in the configuration file entry. Thus the turnaround time describes how long a process remained in the system. In contrast to the arrival time, the first-run time (column `Start`) is the time when the process had its first CPU cycle. The CPU time (column `CPU-tme`) is the amount of computing time that the process used – it is independent of the scheduling decisions.

It is interesting to also note the quotient of turnaround time and CPU time: The closer it gets to 1, the better the scheduler's decisions have been from this process' point of view.

10 ⟨*function declarations* 3c⟩+≡ (2) ◁9 11▷

```
def stats():
  global cputime
  print
  print "Endzeit: %d" % cputime
  print "Trace:",trace

  print "Laufzeiten:"
  print "PID Arrival CPU-tme Start   End     TurnAro  Quotient"
  print "-----------------------------------------------------"

  for pid in range(0,proccount):
    stime  = get_starttime(pid)
    etime  = get_endtime(pid)+1
    frtime = get_firstruntime(pid)
    cputime = get_cputime(pid)
    if cputime != 0:
      ratio = (etime-stime+0.0)/cputime
    else:
      ratio = -1
    print "%3d %7d %7d %7d %7d %7d %9.4f" % (pid,
      stime, cputime, frtime, etime, etime-stime, ratio)
```

Defines:
`stats`, used in chunk 18b.

This will display an output such as

```
Endzeit: 91
Trace: [0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
 3, 3, 3, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2,
 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0, 0, 2, 2, 2, 2, 2, 2,
 2, 'END']
Laufzeiten:
PID Arrival CPU-tme   Start     End TurnAro  Quotient
-----------------------------------------------------
  0       0       6       0      84      84   14.0000
  1       0      30       2      64      64    2.1333
  2       0      25      17      91      91    3.6400
  3       0      30      32      82      82    2.7333
```

The trace that was shown in the output above is a list that describes what happened at what point in time – there are three entry types:

- a process ID – the process with the given ID was executed.
- `NONE` – no process was executed (because all processes were blocked while waiting for the end of an I/O operation).
- `END` – all processes have finished. This entry will only appear as the last one.

The trace is generated while executing the main loop ⟨*main loop* 12⟩:

11 ⟨*function declarations* 3c⟩+≡ (2) ◁10 14▷

```
  def log_to_trace(status):
    global trace
    if status == -1: status = "NONE"
    elif status == -2: status = "END"
    trace.append(status)
```

2 The Main Loop

The basic idea behind the simulator is that in a loop the scheduler ⟨*scheduler* 16⟩ is called repeatedly and then the system executes one instruction of the process that was chosen by the scheduler.

If a process was blocked in this step, its waiting time has to be updated (decreased by 1); and if the waiting time goes down to 0, the I/O phase is over and the process becomes ready again. (This is done in `update_blocked_processes`.)

Depending on the result (`current`) which the scheduler function `schedule` returns, there are three possibilities:

- The scheduler has selected a process for execution (`current` > 0), in that case this process is activated and executed (one step), and the waiting times of the blocked processes are updated.

  Notice that this description models a preemptive scheduler. If a non-preemptive scheduler is to be implemented, the scheduler will have to "stick" with a decision it made once, i. e., it has to re-select the same process again and again until it terminates or changes to an I/O phase.

- The scheduler found no ready processes (but there are still unfinished processes in the system): `current` $= -1$. In this case only the waiting times are updated; no process executes.

- The system is done (`current` $= -2$). Nothing remains to to.

12 ⟨*main loop* 12⟩≡ (18b)

```
while not finished:
  ⟨check for new processes 13⟩
  current = schedule()
  log_to_trace (current)
  if current >= 0:
    print "Time: %4d, Executing PID: %3d [%3d]" % (cputime,
      current,get_cputime(current))
    activate (current)
    run_current()               # run current process
    update_blocked_processes()  # decrement wait times
  elif current == -1:
    print "Time: %4d, all blocked" % cputime
    update_blocked_processes()  # decrement wait times
  else:
    finished = 1
    break
  ps()                          # show process list
  cputime += 1                  # increment CPU time
```

Uses `activate` 8c, `ps` 9, `run_current` 14, `schedule` 16, and `update_blocked_processes` 15a.

where in each cycle the system also checks whether new processes have appeared (those with a start time that is equal to the current CPU time) – for each such process there is also a check whether it starts with a blocked phase.

13 ⟨*check for new processes* 13⟩≡ (12)
```
for pid in range(0,proccount):
  if get_starttime(pid) == cputime:
    if get_status(pid)==S_BLOCKED:
      blocked.append(pid)
    else:
      runqueue.append(pid)
```

2.1 Running a process and handling blocked processes

Once a process is blocked, it is moved into the special queue `blocked` (and removed from the runqueue). This happens in the `run_current` function which otherwise is responsible for running the current process.

Running the process consists of the following tasks:

- checking whether a process is run for the first time ever – if so, the current time is saved in the PCB structure via `set_firstruntime` for later referral.
- reducing the first element of the `behavior` list by 1 (using `dec_behavior_head`) and increasing the process' cputime (`inc_cputime`).
- if this leads to the head element of `behavior` becoming 0, it means that the process has finished the recent CPU cycle and moves to an I/O wait phase (or terminates). The function determines which of these is the case by first removing the old head element (now 0) and then looking at the new head: If it is non-negative, an I/O phase follows; otherwise it can only be -1 which means termination of the process.
- If the process becomes blocked the status must be set to `S_BLOCKED`, the process is removed from the runqueue and added to the blocked queue.
  If the process terminates, its status is set to `S_DONE`, it is removed from the runqueue and the current time is saved in its `endtime` element using `set_endtime`.

14 ⟨*function declarations* 3c⟩+≡ (2) ◁11 15a▷

```
def run_current():
  global current, cputime, runqueue, blocked
  if get_firstruntime(current) == -1:
    set_firstruntime(current, cputime)
  dec_behavior_head (current)
  inc_cputime(current)
  if get_behavior(current)[0] == 0:
    # current CPU cycle is over; possibly become blocked
    remove_behavior_head (current)
    set_status(current,S_BLOCKED)
    runqueue.remove(current)
    if get_behavior(current)[0] == -1:
      # process terminates
      set_status(current,S_DONE)
      set_endtime(current,cputime)
      if current in runqueue: runqueue.remove(current)
    else:
      # I/O phase
      blocked.append(current)
  return
```

Defines:
`run_current`, used in chunk 12.

Information about blocked processes must be updated as well. This happens in the `update_blocked_processes` function which is called from the ⟨*main loop* 12⟩ just after `run_current`. It goes through the blocked list, and for each process found does the following:

- it reduces the remaining I/O wait time (which for a blocked process is also in the `behavior` head element, so this can be done with `dec_behavior_head`, as in `run_current`).
- Next the I/O wait time (`iotime`) is incremented for the later analysis.
- Similar to the transation from `S_ACTIVE` to `S_BLOCKED` here the transition from `S_BLOCKED` to `S_READY` must be handled: When the head element of `behavior` has reached value 0, that head is removed. Depending on the next element (it being non-negative or -1), the process status is set to either `S_ACTIVE` or `S_DONE`. In both cases the process is removed from the blocked queue (it is no longer blocked), and it is appended to the runqueue if it becomes ready. If the process terminates, the end time is saved.

15a ⟨*function declarations* 3c⟩+≡ (2) ◁14 18a▷

```
def update_blocked_processes():
  global current, cputime, runqueue, blocked
  for pid in blocked:
    if cputime >= get_starttime(pid) and pid != current:
      dec_behavior_head (pid)
      inc_iotime(pid)
      if get_behavior(pid)[0] == 0:
        remove_behavior_head(pid)
        set_status(pid,S_READY)
        blocked.remove(pid)
        if get_behavior(pid)[0] == -1:
          set_status(pid,S_DONE)
          set_endtime(pid,cputime)
        else:
          runqueue.append(pid)
  return
```

Defines:
`update_blocked_processes`, used in chunk 12.

3 The Round Robin Scheduler

Now we come to the heart of this program – the implementation of the Round Robin Scheduler. The main concept of this scheduler is time slicing: The CPU is virtualized by letting each process assume it holds the CPU for itself alone, and the scheduler lets the system switch between processes regularly. The interval length that determines how long each chosen process may go on computing before a context switch is called quantum length, and in our implementation we define it by setting the constant `RR_QUANT` (Round Robin Quantum):

15b ⟨*user defined constants* 15b⟩≡ (2)

```
RR_QUANT=15
```

Defines:
`RR_QUANT`, used in chunk 17a.

In the "real world" a Round Robin scheduler depends on the computer allowing for preemptive scheduling, i. e. having an internal clock and timer interrupts. In our model the main program (the simulator) can be seen as generating an interrupt after every process instruction (or time cycle in which all processes were waiting for I/O completion). In a real system clock interrupts occur less frequently, and the scheduler is activated even less frequently, but in our model the scheduler is run after every instruction.

1. The Round Robin scheduler first checks whether the currently active process has used up its quantum – if so, it is removed from the head of the runqueue and appended at its end: We use a simple FIFO queue for the processes. If a quantum length of infinity (or simply large enough) were chosen, the scheduler would degenerate to the non-preemtive FIFO scheduler. This first check is implemented in ⟨*sched check current may continue* 17a⟩. Notice that it is necessary to also check whether `current` is -1 – in that case there was no current process (which may mean the system was just started or all processes are blocked).

2. Next we check whether there are any processes left to execute – they may be found in either the runqueue or the blocked queue. That is done in ⟨*sched check runqueue and blockedqueue* 17b⟩. A last thing to check – only in the case that both queues are empty – is whether there are "future processes", that is: those which have a start time in the future. If so, the scheduler returns `-2`.

3. Now, if we reach the next step, there is at least one process in either the runqueue or the blocked queue; if the runqueue is non-empty we return the head of that list; otherwise we return `-1`, indicating that all processes are blocked.

16 ⟨*scheduler* 16⟩≡ (18a)

```
def schedule():
  # Round Robin Scheduler
  global current, tasks, runqueue, blocked, current, cputime
  ⟨sched check current may continue 17a⟩
  ⟨sched check runqueue and blockedqueue 17b⟩
  # choose head of runqueue or return -1 if it is empty
  if (runqueue != []):
    return runqueue[0]
  else:
    return -1
```

Defines:
`schedule`, used in chunk 12.

The current process will typically be marked active – that is the first thing the scheduler checks in this step. If it is not active it can either be done or blocked; it cannot be in the ready state – `get_status(current)` can never be `S_READY`, since only the scheduler resets the active state, and there is no other transition to `S_READY` but from `S_ACTIVE`, except for process creation, and the current process is not a newly created process.

Finding out whether the current (and active) process may continue is then just looking at the quantum used so far – the condition `get_usedquant(current) < RR_QUANT-1` must hold. If so, the used quantum is incremented, and the scheduler returns the current process' ID.

If not, the current process has used up its quantum and a new process must be chosen. In preparation, the used quantum length is reset to 0 (for the next run of this process), and the process is removed from the runqueue's head and appended to its tail. The variable `current` is then set to the runqueue's (new) head. Note that here it cannot happen that the runqueue is empty, because the formerly active process has just been appended to its tail. In general the runqueue could be empty, but that would mean that the old current process was not active and thus we would not run through this code.

17a ⟨*sched check current may continue* 17a⟩≡ (16)
```
  if (current!=-1) and (get_status(current) == S_ACTIVE):
    if get_usedquant(current) < RR_QUANT-1:
      inc_usedquant(current)
      return current
    else:
      set_usedquant(current,0)
      runqueue.remove(current)
      runqueue.append(current)
      current=runqueue[0]
      return current
```
Uses `RR_QUANT` 15b.

If both the runqueue and the blocked queue are emptied, all processes which existed before, have terminated and left the system – so it may be over. But there may also be processes who have not started yet. This is checked by calling `futureprocesses(cputime)`: If the returned list is empty, there are no future processes, and the system will stop.

17b ⟨*sched check runqueue and blockedqueue* 17b⟩≡ (16)
```
  if (runqueue == []) and (blocked == []):
    # vielleicht kommt noch einer?
    if futureprocesses(cputime) == []: return -2
```
Uses `futureprocesses` 18a.

The function `futureprocesses` is defined here and added to the list of function declarations. For a given time `t`, the function returns a list of all processes with a start time that is greater than the given time.

18a ⟨*function declarations* 3c⟩+≡ (2) ◁15a

```
def futureprocesses(t):
  # gibt alle PIDs zurueck fuer Prozesse, die nach Zeit t starten
  global proccount
  fp = []
  for pid in range(0,proccount):
      if get_starttime(pid) > t: fp.append(pid)
  return fp
⟨scheduler 16⟩
```

Defines:
`futureprocesses`, used in chunk 17b.

So far we have not defined the ⟨*main program* 18b⟩ section that consists of initialization, the main loop and printing the statistics – here it is:

18b ⟨*main program* 18b⟩≡ (2)

```
init ()
print "Number of tasks: ", proccount
ps()
⟨main loop 12⟩
stats()
```

Uses `init` 3b, `ps` 9, and `stats` 10.

What remains is the Python header which starts with the typical "hash-bang" line for finding the interpreter, followed by a coding description (to allow for german characters such as äöüß) and then some version and copyright information.

18c ⟨*python header* 18c⟩≡ (2)

```
#!/usr/bin/env python
# -*- coding: iso-8859-15 -*-

# scheduler.py v1.1 (Praktikum, Arbeitsblatt 6)
#
# Vorlesung Betriebssysteme
# Hans-Georg Eßer, Hochschule München
# hans-georg.esser@hm.edu
```

4 Discussion

This scheduling simulator was used in an exercise for computer science students taking a course on operating systems. The simulator was provided with a different scheduler (non-preemptive FCFS) that had to be modified, and the original version was written in Python from scratch (without the *literate programming* approach). The original code was modified and cleaned-up, e. g. direct access to PCB structures was replaced with access via `get_`, `set_`, `dec_` and `inc_` procedures.

I've made the attempt to document the new code as fully as seemed useful. I will put this document up for discussion on the lecture website, though I assume that only few of my former course participants will look at it.

When "tangling" and "weaving" this document, the output is ca. 7 KByte of Python code and 50 KByte of LaTeX documentation, where the original source only contained a few lines of explaining comments. So now there is substantial documentation which may help students of upcoming OS courses better understand the concept of this simulator.

A Usage Instructions

In order to work, *noweb* requires the existence of a LaTeX environment, as well as the *noweb* package itself. The author, Norman Ramsey, provides Debian, RPM and tar.gz versions of this software package on his homepage [Ram06]. A test installation on an OpenSuse 11.0 Linux system showed no dependencies on other packages, however when trying to use the *noweave* program, it turned out that one additional package was required: *iconx*, an "Executor for Icon, a high-level programming language". The Debian repositories carry an *iconx* deb package (e.g. `ftp://ftp.debian.org/debian/pool/main/i/icon/` that could easily be converted into an rpm package using *alien* (`http://kitenet.net/programs/alien/`).

This document and the program file `scheduler.py` can be generated using the following script:

```
#!/bin/bash
# generate sched-rr10.dvi (documentation)
noweave -index -delay -latex sched-rr10.nw  > sched-rr10.tex
latex sched-rr10.tex
bibtex sched-rr10
latex sched-rr10.tex
# xdvi sched-rr10.dvi

# generate scheduler.py (program) and run a test
notangle -Rscheduler.py sched-rr10.nw  > scheduler.py
notangle -Rtest.dat sched-rr10.nw  > test.dat
chmod a+x scheduler.py
./scheduler.py test.dat
```

The test file `test.dat` is also part of this noweb document and is defined here:

19 ⟨*test.dat* 19⟩≡

```
  0:2,5,2,5,2,-1
  0:30,-1
  0:18,5,7,-1
  0:30,-1
```

For creating the *dvi* file it is also necessary to have the *bibtex* file with the references (`lit.bib`).

References

[Eße08] Hans-Georg Eßer. Betriebssysteme I, Unterlagen zur Vorlesung, 2008. `http://hm.hgesser.de/bs-ss2008/`.

[Knu84] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27:97–111, 1984.

[Ram94] Norman Ramsey. Literate Programming Simplified. *IEEE Software*, pages 97–105, September 1994.

[Ram06] Norman Ramsey. noweb web page, 2006. `http://www.cs.tufts.edu/~nr/noweb/`, accessed 2008/08/24.