xv6 Operating system project

Implemented files:

- *in kernel:*

1. Scheduler.h
2. Scheduler.c
3. RBTree.h
4. RBTree.c

- *in user:*

5. chsched.c

Changed files of xv6 operating system:

- *in kernel:*

1. proc.h
2. proc.c
3. syscall.h
4. syscall.c
5. sysproc.c

- *in user:*

6. user.h
7. usys.pl

In this project, the default scheduler of xv6 operating system is changed. Implemented scheduling algorithms are **Completely Fair Scheduler** (*CFS*) and **Shortest Job First** (*SJF*). Both of them are using Red-Black trees as queues for ready processes. This data structure is implemented in *RBTree.c* (and *RBTree.h*). Red-Black tree is statically allocated and every tree has *NPROC* (which is maximum number of processes) nodes and maximum number of trees is *NCPU* (one tree per processor). Every tree has array of nodes and array of memory free entries with pointer to first free index in array, both of them are statically allocated. When node is inserted in tree it is inserted at index where pointer of free memory entries is pointing. When node is removed from tree, index where it was allocated is released and added to linked list of free memory entries. Other things are implemented by standard rules of Red-Black tree organisation.

Compltely Fair Scheduler is using total execution time (in OS this variable is called *measuredTime*) of process as it's priority for scheduling. Execution time is represented as number of time quants (where one quant is time between two timer interrupts in CPU) while a process was executing on CPU. Maximum execution time, or maximum number of time quants

while process can be executing, is calculated in get method of this scheduler as amount of time quants while process was waiting in ready queue divided by number of processes that are still waiting to be executed (waiting in ready queue). If there are too many processes in waiting queue, maximum execution time is set to 1 (as default value).

Shortest Job First is using predicted burst time of process as it's priority for scheduling. Predicted burst time is calculated dynamiclly using exponential average:

$$\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_n$$

Where tn is acutal burst time (measuredTime variable in OS) and τ_n is last predicted burst time. τ_{n+1} is next predicted burst time. Parameter α is smoothing factor, if it's zero this formula only uses lastly predicted burst time and if it's one then this formula uses only last burst (execution) time. In this OS value for this parameter goes from 0 to 100, and at the end whole equation is divided by 100. This way we use whole numbers every time and avoid working with floating point numbers. To avoid rapid convergence of predicted burst time, every time τ_{n+1} is being calculated we multiply t_n by 128 (the original idea was to multiply it by 100 so we could use first two decimals, but in light of avoiding complex mulitply operation, it has been chosen to use 128 as multiply factor so that way t_n can be just shifted 7 times in left). SJF algorithm has two realisations – preemptive and non-preemptive. If it's preemtive, time slice (or time quant) for execution is set to 1 (default value) and every time an interrupt from timer occurs context is changed and CPU starts executing another process. While at non-preemptive vairant of this algorithm we do not have any problems calculating next predicted burst time, at preemptive variant there is one. If we would calculate next burst time every time context has been changed, we would use 1 as t_n value and τ_{n+1} would converge to 1, not to actual execution time of process between I/O (or Event) blocking operations as it would be in non-preemptive variant. To overcome this flaw in put method of scheduler is added flag *wasBlocked* which indicates if process has come back from blocking operation (like I/O or Event). If *wasBlocked* is set to one, in put method we calculate next prediction of burst time and if it's not we just put process in ready queue without changing it's prirorty (which is next predicted burst time). This way prediction for next burst time is calculated only when process returns from some blocking operation.

Beside these two scheduling algotihms in scheduler are implemented global put and get operations which are actually called every time. These operations choose from which processor's ready queue (or to which processor's ready queue) process shoud be removed (or inserted). To determine this, global scheduler operations have 3 categories of CPUs: *EMPTY CPU*, *BALANCED CPU* and *OVERLOADED CPU*. Using these categories we perform **load balancing** between processors.

- *EMPTY CPU* is state of processor when there are no processes waiting in ready queue (this state is set to all processors when system is booting.
- *BALANCED CPU* is state of processor where total amount of processes that were put in ready queue of this processor is at least two times bigger than number of processes that are still waiting in ready queue of this processor (this way we measure usage of CPU).
- *OVERLOADED CPU* is state of processor which ready queue is not empty and total amount of processes that were put in ready queue of this processor isn't at least two times bigger than number of processes that are still waiting in ready queue of this processor.

Also, global put operation takes care about **process affinity** (process affinity is number of CPU which process was last time exectued at). By putting process in ready queue of processor at which process was lastly executed, we are trying to achieve to lower penalty miss in CPU cache (and possibly TLB unit as well). In scenario where there are processors that are EMPTY or BALANCED and processor, which process has affinity to, is *OVERLOADED* then affinity of process is annuled and process is being scheduled like it is new. In every other scenario, affinity of process is respected. Global put and get methods do some more operations which are not mentioned here.

Also, in this project is implemented one system call (***chsched***) and one user program, with same name as system call, that can be called from command line. This system call (***chsched***) is used for changing between these two scheduling algorithms and it can be accessed through user program. User program takes arguments from command line and forwards them to system call. Arguments that can be set are:

- **name of algortihm** (CFS / SJF - *in capital letters*)

- **alpha (α) value** (value in range [0, 100] – *this argument is only for SJF*)

- **type of alogithm** (preemptive (1) / non-preemptive (0) – *this argumet is only for SJF*)

This project is made within course "Operating systems 2" at School of Electrical Engineering, University of Belgrade.

Project made by: Arsenije Simonović