Documentation and design choices for Yaeos

Andrea Berlingieri Stefano Andriolo Federico Rachelli
Igor Ershov

A.A. 2017/2018

Contents

Data structures

Pcb List

The pcb list is kept using a series of recursive functions.

The function used to inster a new process in the queue is called insertprocQ. Firstly it checks if there are elements in the queue, replacing the head if there are none, then it checks if the new process has a higher priority than the head, and places it before if its has , or it proceeds recursevly on the list if not.
Y

With headProcQ you can get the first element of the list, without removing it, to also remove it you can use removeProcQ instead.
You can remove a specific process using outProcQ. This function will check the head of the list against the desired process, if its a match it will remove and return it , else it will proceed recursevly on the next element on the list.

To apply a function to all the processes in the list you can use forallProcQ, it will run the function against the first element of the list and proceed recursevly on the next one.

Free Pcb List

Its a list of free pcbs and its kept using a series of recursive functions.

With freePcb the pcb passed as an argument is made the head of the list.
With initPcbs the list will be initialized to contain all the elements of the pcbFree_table. This function will be called only once when the data is being initialized.
With allocPcb a new pbc with all the parameters set to null will be created and will be made the head of the free pcb list.

Tree

Active Semaphore Hash Table

The semaphores in YAEOS are implemented with the use of an integer variable and a semaphore descriptor. The semaphore descriptor contains a reference to the semaphore it represents and a pointer to a queue of PCBs blocked on that semaphore. The semaphore descriptors currently in use are mantained in a hash table.

There is a fixed number of semaphore descriptors, each of which is statically allocated in the kernel and, if not being used at the moment, is kept in a list of available descriptors, that can be used when a semaphore descriptor is needed.

This means that there's a limit to the number of total semaphores that can be used concurrently.

A semaphore's value is contained in the integer variable associated with it, and a semaphore is identified by the address of that same variable. This address is used as an argument to the hash function to retrieve its semaphore descriptor in an efficient way.

The chosen hash function is the multiplicative hash. Despite the computational cost due to the multiplication, this function has the pro of being independent of the size of the hash table. The constant used for the multiplication was suggested by Donald Knuth and should be good in most situations.

The queue of PCBs blocked on a semaphore is a priority queue and the hashtable is a chained hash table with linked lists.

PCB design

The PCB contains the following fields:

- `p_next`: Pointer for the next element of the list.
- `p_parent, p_first_child, p_sib`: Pointers for tree hierarchy of processes.
- `p_s`: CPU state. Useful to save the state of a process to resume it at a later time.
- `p_priority`: Process' priority. The scheduler can increase the priority according to aging policy, up to the maximum limit of MAXPRIO.
- `old_priority`: We keep track of the original priority of the process, i.e. the one the process was created with. This can be restored at the appropriate time.
- `p_semKey`: A pointer to the semaphore the process is blocked on (if any).
- `waitingOnIo`: 1 if process is waiting for I/O, 0 otherwise.
- `childSem`: Semaphore used to block the process when it's waiting for a child to terminate. This semaphore will be V'ed by the child which terminates first.
- `usertime`: Process' time spent in user mode
- `kerneltime`: Process' time spent in kernel mode
- `lasttime`: Last TOD marker. We use it for keep track of last time the process was started. Useful to keep track of the process' user time
- `wallclocktime`: Process' creation TOD. Useful for calculating the wallclocktime

- **`sysbk_new, sysbk_old, tlb_new, tlb_old, pgmtrap_new, pgmtrap_old`**: Per-trap handlers

Scheduler

The YAEOS scheduler is a priority-based scheduler with aging and a timeslice. The aging period is 10ms, whereas the duration of a timeslice is 3ms. Conceptually, the dispatcher is like a Singleton Object with some "member fields" to keep the status of the system and some "member functions" to interact with it. The "member fields" of the scheduler are the ready queue, a priority list for the processes that are ready to be started, a pointer to the currently running process, the number of processes in the ready queue, the number of active processes and the number of softblocked processes. The so called "active processes" are all the processes that are not waiting on a device semaphore (these include the ready processes), whereas the softblocked processes are the ones waiting on a device semaphore. The "member functions" of the scheduler are the dispatch function, the suspend function, the increasePriority function, the restoreRunningProcess function, the insertInReady function, the three functions for processes' time accounting (userTimeAccounting, kernelTimeAccounting and freezeLastTime) and the passup function.

Parts of the OS call the methods of the scheduler they need to and handle the counters directly. Some aspects of the of the loading of a new process, like the saving of the state of a process in its pcb and some adjustments to the pc of a process, are done only in the scheduler module, so they are abstracted by the scheduler methods. Still, there is not a single function that checks the state of the system and takes the right scheduling decision.

There may be, at some point, no processes in the ready queue. This could be a normal situation, as there could be some processes that are blocked on some device or on the pseudoclock, or there could be no processes to execute at all, or there might be a deadlock. In the first case the dispatch function puts the system in a waiting state, with all interrupts unmasked, waiting for an interrupt from a device or from the timer to unblock a process. In the second case the system is shut down. Finally in the third case a PANIC instruction is executed.

Interrupts

Interrupts are handled by a function that, based on the device that caused the interrupt, takes the appropriate handling decisions. In pretty much every case, except for the timer, the handler determines the device that caused the interrupt, unblocks a process from the semaphore associated with it, acknowledges the

interrupt and saves the return status from the device in the a1 register of the state of the pcb of the process that was waiting on the device. The terminal devices are slightly different in that they are "double devices": they act like a pair of devices, one of which is the receiver and the other is the transmitter. The determination of the device that caused the interrupt is made by scanning each interrupt line and each bit in the bitmap associated with a particular interrupt line according to their precedence order (as defined by the YAEOS specification).

Interrupts are handled one at a time: when an interrupt is generated the interrupt handler routine handles it and then gives back control to processes or, if there are no processes to give control to, it goes in a waiting state to wait for an interrupt to wake up some process. In both cases the interrupts are unmasked. If during the handling of the first interrupt others have been generated then, as soon as the interrupts are unmasked, uarm gives control back to the interrupt handler routine, that handles the new interrupt. This effectively creates a loop in which all interrupts are handled before giving control back to processes.

The timer interrupts are handled based on the cause of the interrupt. This is maintained in a separate variable that is properly set when the timer is updated. If the cause of the interrupt was the pseudoclock tick then all the processes blocked on the pseudoclock semaphore are unblocked. If the cause was the aging tick, then all processes (if any) in the ready queue have their priority incremented. Otherwise a timeslice expired and the running process needs to be suspended and a process dispatch needs to take place. This is handled by the scheduler.

The pseudoclock and aging clock ticks are handled in the same way: a global variable for each keeps the count of ticks. Based on that it's possible to calculate the next deadline for the pseudoclock or aging tick. If it's close enough or it has been passed then the next cause for the timer interrupt will be set accordingly and the appropriate actions will be taken. These include the increment of the tick counter.

The update of the timer is done as follows: first the deadline of the next pseudoclock tick is calculated. If this is less than the timeslice duration then the cause of the next interval timer's interrupt will be the pseudoclock tick. If the deadline has been passed the timer is not set: this way its interrupt is not ackowledged and so it is handled again the moment the interrupts are unmasked. Otherwise, the timer is set to trigger an interrupt on the next pseudoclock tick. The same thing is done for the aging tick if the pseudoclock deadline is further away in the future than the duration of a timeslice. If the aging tick is also too far in the future to worry about it then the timer is set to the timeslice duration and the cause is set accordingly.

Handlers

Pgm Trap Handler

Tries to passup the trap to a higher level handler using the passup function , if its unsuccesful it terminates the process and dispatches a new one.

Tlb Handler

Tries to passup the tlb to a higher level handler using the passup function , if its unsuccesful it terminates the process and dispatches a new one.

Syscall and Breakpoint Handler

The YAEOS Syscall and breakpoint handler only supports the first 10 syscalls for processes in kernel mode, if the cause of the exception is a breakpoint or the process is in user mode it tries to pass it to a higher level handler, or a pgm trap handler with the cause "Reserved Instruction" if the process in user mode tries to use a reserved syscall, using the passup function and if unsuccesful it terminates the process and dispatches a new one. If the process is in kernel mode and the exception is a syscall the handler checks which syscall is being called, sets the appropriate parameters to the respective function and calls it, storing any return values in the "a" registers of the process. If the handler completes its job succesfully without passing up, it restores the caller if there is a running process, or it dispatches a new one if there isn't.

Passup helper function

Checks if the process has a higher level handler stored in its corrisponding new area (sysbk, tlb or pgmtrap) , if there is one the current state will be saved in the old area and the new area will be loaded, if there isn't the function will return a failure.

Initialization nucleus

The first operations executed by the kernel are the initialization of data structures, variables, and NEW areas for interrupts and traps as well as the creation of the first PCB.

First of all, NEW areas for interrupts and traps are initialized following the specification. For each interrupt and trap there's a constant which points to the address of the NEW area that will be loaded every time that interrupt or trap is raised. A function handler for each interrupt or trap is specified, and its address

is stored in the program counter variable of the NEW area. The stack pointer value is set ro RAM_TOP, all interrupts are disabled and the execution mode is set to privileged.

The pcbFree list and the ASL are initialized using the default methods. Before the initialization of the first PCB, the other variables are set with the default values. These variables are the number of PCBs in the ready queue, blocked on devices semaphores for I/O and the number of active onesi (that is not softblocked). Pseudoclock ticks count, aging ticks count are set to zero and the pseudoclock semaphore is initialized to zero. All the devices semaphores are set to zero. The two variables which represent the high and low part of the TOD when the initialization is done are set using the values returned by getTODLO() and getTODHI(). These values are useful to set the pseudoclock and aging ticks.

The last action to be performed before the schduler is called is the initialization of the first PCB. All variables except for the state variable will contain the default values. The state of the PCB is set according to the specification, so the stack pointer points to RAM_TOP - FRAMESIZE (the second last memory frame), the program counter contains the address of the *test* function defined in p1test; all interrupts are enabled, the execution mode is privileged, and all the bits of the CP15 coprocessor are set to zero to disable the virtual memory. The createProcess syscall is invoked to finally create the first PCB.

The last action performed by the nucleus is the call to the dispatch function of the scheduler. From this point onward the execution flow will return to the nucleus only when an interrupt occurs or a trap is raised, and the operating system is, at this point, actually started.