Model Checking Process Scheduling over Multi-core Computer System with MSVL

Xinfeng Shu[1](✉) and Zhenhua Duan[2]

[1] School of Computer Science,
Xi'an University of Posts and Communications, Xi'an 710061, China
shuxf@xupt.edu.cn

[2] Institute of Computing Theory and Technology,
Xidian University, Xi'an 710071, China
zhhduan@mail.xidian.edu.cn

Abstract. To solve the problem that software testing cannot meet the verification needs of process scheduling over multi-core computer system, a model checking approach with MSVL is adapted to verify the correctness of process scheduling. Firstly, the grammar of MSVL is briefly introduced; further, a general model supporting the most commonly used scheduling algorithms for multi-core computer is formalized by MSVL program; finally, as a case study, the safeness of the processes scheduler with the earliest deadline first(EDF) scheduling algorithm over a 2-core CPU is verified with MSV toolkit, which indicates the proposed approach can effectively solve the verification problems of process scheduling over multi-core CPU.

Keywords: Process scheduling · System modeling · Verification · Model checking

1 Introduction

Process scheduler is a critical component of the modern operating system. Its correctness is one of the most important factors affecting the safety and reliability of the computer system. How to ensure the correctness of the process scheduling is of great importance to the system developer. Currently, testing is the most important means to check the correctness of process scheduling. However, the method can only show the existence of the errors, but can not prove their absence. What's more, the executions of the processes over multi-core CPU become really parallel, and each process proceeds in a unpredictable speed under the influence of other processes. Hence, the classical testing method can hardly meet the verification needs of process scheduling under such circumstance.

This research is supported by Natural Science Foundation of Education Bureau of Shaanxi Province, China (No.11JK1037, No.15JK1678), Natural Science Foundation of Shaanxi Province, China (No.2015JM6359), the NSFC Grant No.61133001, No.61202038, No.61272117, No.61272118 and No.61322202.

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2015, LNCS 9559, pp. 103–117, 2016.
DOI: 10.1007/978-3-319-31220-0_8

In the past decade, quite some researches have been down in formal specification and verification of process scheduling with various approaches. In [1], the authors employ the μCRL process algebra to model the scheduling and apply the μCRL model checker toolset to find the near-optimal solutions. In [2], the author uses Promela to describe a job-shop scheduling problems and adds Branch-and-Bound techniques to the LTL property to find a solution effectively. In [3], the real-time system with a preemptive scheduling policy is modeled as a scheduling time Petri Net and the timed properties are verified using HyTech, an automatic tool for the analysis of embedded systems.

Projection Temporal Logic (PTL) [4–6] is an interval based first order temporal logic with a key temporal construct, $(P1, ..., Pm)prjQ$, and supports both finite and infinite time. Further, Propositional Projection Temporal Logic (PPTL), the propositional subset of PTL, has the expressiveness power of the full regular expressions [7], which can be used to specify the properties of the hardware or software systems to be verified.

The language Modeling, Simulation and Verification Language (MSVL) is an executable subset of PTL with framing technique [8,9]. It supports the commonly used data types (e.g., char, integer, pointer, ...), data structures (e.g., array, user-defined structure, list, ...), boolean and arithmetic expressions. Besides, MSVL provides many powerful programming statements, such as frame, if, while, parallel ($||$), ..., and has ability to model, simulate and verify the concurrent and reactive systems within a same logical system [10].

In recent years, MSVL has been successfully applied to verify the typical hardware and software systems such as virtual memory and embedded operating system [11–15]. This paper extends the application of MSVL based model checking approach to verify the process scheduling in a multi-core CPU computer system. To this end, a general system model supporting the typical periodic and non-periodic processes as well as the most commonly used scheduling algorithms is formalized by MSVL program. Then, an example is given to illustrate how this method works.

The rest of paper is organized as follows. In the next section, the language MSVL is briefly introduced. In Sect. 3, the system model of process scheduling over multi-core CPU is formalized. In Sect. 4, a case study is given to show how the proposed method works. Finally, conclusions are drawn in Sect. 5.

2 Modeling, Simulation and Verification Language

The language MSVL is an executable subset of PTL [5]. The grammar of MSVL consists of expressions and statements, where expressions can be regarded as the PTL terms, and statements as the PTL formulas [9,11].

Expression. Let D denote the data domain, which includes integers, string, lists, sets, etc. The expression e and boolean expression b of MSVL are defined inductively as follows:

$$e ::= d \mid x \mid \bigcirc e \mid f(e_1, \ldots, e_m)$$
$$b ::= true \mid false \mid e_1 = e_2 \mid \rho(e_1, ..., e_m) \mid \neg b \mid b_1 \wedge b_2$$

where $d \in D$ is a constant; $x \in V$ is a variable; f is a function and ρ is a predicate both defined over D.

Statement. The elementary statements in MSVL are defined as follows:

(1) Immediate Assign	$x \Leftarrow e$
(2) Unit Assignment	$x := e$
(3) Conjunction	$S_1 \ and \ S_2 \overset{\text{def}}{=} S_1 \wedge S_2$
(4) Selection	$S_1 \ or \ S_2 \overset{\text{def}}{=} S_1 \wedge S_2$
(5) Next	$next \ S \overset{\text{def}}{=} \bigcirc P$
(6) Always	$always \ S \overset{\text{def}}{=} \Box P$
(7) Termination	$empty \overset{\text{def}}{=} \neg \bigcirc true$
(8) Skip	$skip \overset{\text{def}}{=} \bigcirc \varepsilon$
(9) Sequential	$S_1 ; S_2 \overset{\text{def}}{=} (S_1, S_2) \ prj \ empty$
(10) Local	$exist \ x : S \overset{\text{def}}{=} \exists \, x : S$
(11) State Frame	$lbf(x)$
(12) Interval Frame	$frame(x)$
(13) Projection	$(S_1, \ldots, S_m) \ prj \ S$
(14) Condition	if b then S_1 else $S_2 \overset{\text{def}}{=} (b \rightarrow S_1) \wedge (\neg b \rightarrow S_2)$
(15) While	while b do $S \overset{\text{def}}{=} (b \wedge S)^{\star} \wedge \Box(\varepsilon \rightarrow \neg b)$
(16) Await	$await(b) \overset{\text{def}}{=} \bigwedge_{x \in V_b} frame(x) \wedge \Box(\varepsilon \leftrightarrow b)$
(17) Parallel	$S_1 \| S_2 \overset{\text{def}}{=} ((S_1 ; true) \wedge S_2) \vee (S_1 \wedge (S_2 ; true)) \vee (S_1 \wedge S_2)$

where x is a variable, e is an arbitrary expression, b is a boolean expression, and $S_1, \ldots, S_m, S$ are all MSVL statements. The immediate assignment $x \Leftarrow e$, unit assignment $x := e$, $empty$, $lbf(x)$ and $frame(x)$ are basic statements, and the left composite ones.

The immediate assignment $x \Leftarrow e$ can be defined as $x = e \wedge p_x$, where subformula $x = e$ denotes the value of variable x is equal to that of the arithmetic expression e, and atomic proposition p_x is the assignment flag for variable x. The unit assignment $x := e$ can be defined as $\bigcirc x = e \wedge \bigcirc p_x$, which means the value of variable x in the next equals to that of e and the assignment flag p_x holds in the next state.

The state frame statement $lbf(x)$ means that the value of x in the current state is equal to that in the previous state if there is no assignment to x, i.e., its assignment flag p_x does not hold; and the interval statement $frame(x)$ means that variable x always keeps its old value in the previous state if variable x is not assigned with a new value. The means of the left statements can be found in [10,11] and hence are omitted here.

To use MSVL to formal verification, a model checking tool named MSV toolkit has been developed. It has three modes, i.e., modeling, simulation and verification, where simulation finds a model for the MSVL program, while modeling finds all models, and verification is based on the model checking approach.

3 Modeling of Process Scheduling over Multi-core CPU

Figure 1 gives a sketch of process scheduling in a typical multi-core computer system. The hardware resource of the system contains a m-core CPU($m \geq 1$), and $n(n \geq 1)$ I/O devices of all kinds. The process queues manage all the processes waiting for the CPU or other I/O device. Whenever a resource is free, the Process Scheduler will select an appropriate process with a standing request for the resource and put it to execute. The computer system allows many processes to execute concurrently, however, a resource can only be occupied by one process at any time.

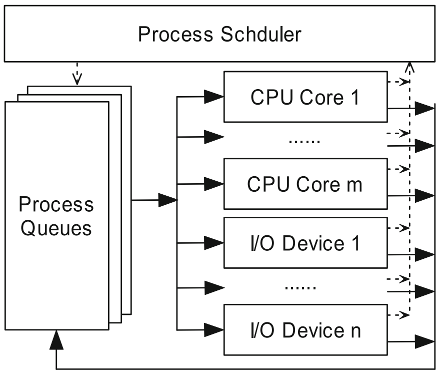


Fig. 1. Sketch of process scheduling over Multi-core CPU

3.1 Resource Description

Since one kind of hardware may have many physical devices, e.g., CPU may have more than 2 cores, we introduce the concept of logical device for the convenience of resource allocation. The name of the logical device is in fact the kind name of the corresponding physical device. Moreover, we assign each logical (physical) device a unique non-negative integer to identify each device. Thus, a process only needs to apply for logical devices during its execution, and the scheduler will allocate the specific physical ones to it and enable the process to run.

Let N_0 and N_1 be the set of non-negative and positive integers respectively. The logical and physical device can be depicted as follows.

Definition 1 (Physical Device). A physical device *pdev* is a tuple defined as

$$pdev ::= (id, status, ldid, cproc),$$

where $id \in N_0$ is the device identifer; $status \in \{0, 1\}$ is the device status (0 denotes idle, 1 means busy); *ldid* is the identifier of the corresponding logical device; *cproc* is the process currently using the device.

Definition 2 (Logical Device). A logical device *ldev* is a tuple defined as

$$ldev ::= (id, type, dlist, pque),$$

where $id \in N_0$ is the device identifer; $type \in \{0, 1\}$ is the device type (0 denotes exclusive, and 1 means shareable); *dlist* is the list of the physical devices belonging to the logical device; *pque* is the queue of the processes standing for the device.

3.2 Process Description

Intuitively, the execution of a process is in fact alternatively using different resource for a certain time in sequence according to the process code. Thus, from the view point of using the resources, the process code can be depicted by a sequence of resource requirement. To this end, we employe *Resource Requirement List* (RRL) to describe the code of a process.

Definition 3 (Resource Requirement List). Let *id* be the identifier of a logical device, and $t \in N_1$ be the time of the device to use. The resource requirement section (RRS) *sec*, composed resource requirement section (CRRS) *cse* and resource requirement list (RRL) *rrl* are defined inductively as follows:

$$\begin{aligned} sec &::= (id, t) \\ rss &::= sec \mid rss_1, rss_2 \\ cse &::= (rss, t) \\ rrl &::= rss \mid cse^n \mid cse^\omega \end{aligned}$$

where RRS (id, t) denotes needing device *id* for t units of time; CRRS (rss, t) $(t \geq \sum_{sec \in rss} sec.t)$ is used to describe the resource requirement of a periodic task, which means the period lasts t units of time and the resource requirement within a period is rss. $cse^n (n \in N_1)$ and cse^ω are used to describe the finite and infinite periodic tasks respectively. The former represents the CRRS *cse* will be repeat for n times, and the later stands for the CRRS *cse* will be repeat for infinitely many times.

Now we give two examples to show how RRL works to describe a process. Without loss of generality, we suppose the computer is equipped with the devices as shown in Table 1. If the RRLs of process P_1 and P_2 are as follows:

$$\begin{aligned} rrl_1 &= (1, 10), (4, 5), (1, 2), (2, 10) \\ rrl_2 &= ((1, 1), (3, 2), (1, 2), 10)^\omega \end{aligned}$$

then, the rrl_1 of process P_1 denotes P_1 will use the devices CPU, HD, CPU, PRT with the lasting time 10, 5, 2, 10 in sequence. Further, rrl_2 means the process P_2 is a infinite periodic task with time period 10, and moreover, it needs CPU, NET and HD for 1, 2 and 2 units of time respectively in a task period.

Table 1. Check list between logical devices and physical ones

Logical Device			Physical Device	
ID	Name	Type	ID	Name
1	CPU	1	1	CPU Core 1
			2	CPU Core 2
2	PRT	0	3	Laser Printer
3	NET	0	4	Network Card
4	HD	0	5	Hard Disk

In consideration of modeling needs for the typical scheduling algorithms, we employ a 8-tuple to describe a process defined as follows.

Definition 4 (Process). The process *proc* is define as:

$$proc ::= (pid, rrl, ldid, pro, stime, ltime, tslice, dline)$$

where $pid, ldid, pro, stime, ltime, tslice, dline \in N_0$; pid is the identifier of the process; rrl is the resource requirement list; $ldid$ is the identifier of the logical device that the process is currently standing for; pro denotes the priority; $stime$ records the start time when the process begins to apply for the current resource; $ltime$ stores the left required time for current device; $tslice$ records the time slice used in a time sharing system; $dline$ keeps the deadline of the current period and it takes effect only if the process is a periodic task.

3.3 System Modeling

The modeling strategy of process scheduling over multi-core computer system is depicted in Fig. 2. The system model consists of three parts, i.e., Scheduler, Physical Devices and System Clock. Each of them will be formalized by a sub-program with MSVL in the following subsections. For simplicity, we ignore the execution time of the Scheduler.

Roughly speaking, the task of the Scheduler is to allocate resource for the waiting process. Firstly, the Scheduler keeps on waiting for the scheduling signal *sigSch*, which denotes there exist some processes applying for system resources. In case of *sigSch* received, it sets variable $cStatus = 0$ to pause the System Clock and allocates needed resources to the standing processes according to scheduling algorithm. Then, the Scheduler sends signal *sigRun* to the allocated resources and enable them to run. Moreover, it set variable $cStatus = 1$ to resume System

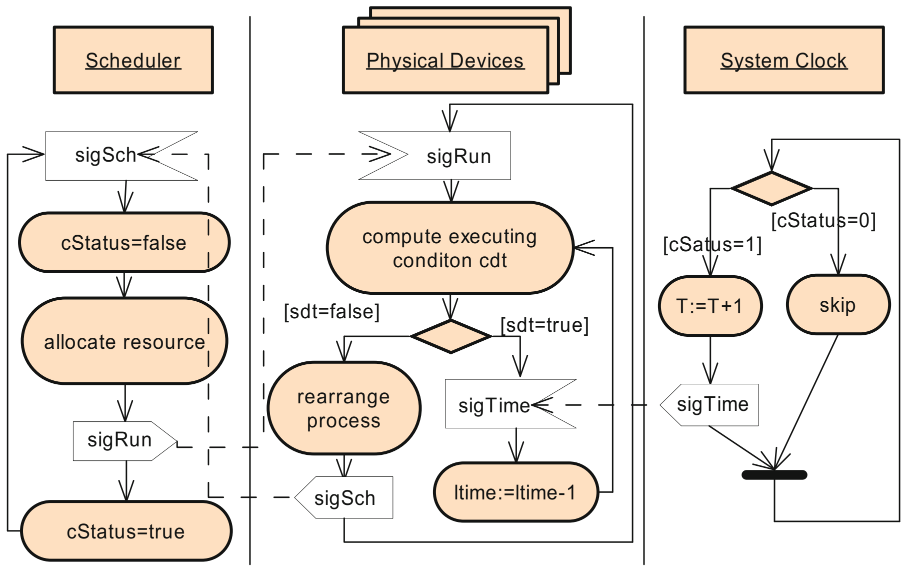


Fig. 2. Activity diagram of the system model

Clock again. After that, the Schedule continues to wait for the occurrence of another scheduling event.

For each physical device, we design a subprogram to describe its running. The execution of a process on a device is abstracted as the decreasing left required time of the resource under the driven of the signal *sigTime* from the System Clock. Initially, the physical device keeps idle and waits for the coming of signal *sigRun*. When the signal *sigRun* arrives, the device computes the executing condition *cdt* (the details can be found in Subsect. 3.3). If *cdt* holds, the device decreases the left required time *ltime* by 1 on the arriving of signal *sigTime* and continues to run the process; otherwise, it rearranges the process int.o the waiting process queue according to the left resource requirement and send a signal *sigSch* to the Scheduler. Finally, the device goes on waiting for another execution.

The task of System Clock is to keep the synchronization among physical devices like a real computer does. It just keeps on increasing the system time T and sending signal *sigTime* to each physical device in case of $cStatus = 1$.

In the following, we firstly employ MSVL structures to describe the data structures used in system modeling, and then use MSVL functions to simulate the running of Physical Device, Scheduler and System Clock.

Data Structures. We use linked list to manage the data of logical devices, physical devices and processes. According to the Definitions 1 and 2, the types of list nodes for physical device and logical device are defined respectively with MSVL as follows.

```
struct pdev_t                 // node definition for physical device
{
    int id and                // physical device id
    int status and            // device status:0 for free and 1 for busy
    int ldid and              // corresponding logical device id
    process_t *cproc and      // pointer of the process using the device
    pdev_t *nexts             // pointer of the next node
};
struct ldev_t              // node definition for logical device
{
    int id and             // logical device id
    int type and           // device type:0 for exclusive and 1 for shareable
    pdev_t *dlist and      // physical device list
    process_t *pque and // queue of processes standing for the device
    ldev_t *nexts          // pointer of the next node
};
```

According to the Definition 3, the types of list nodes for resource requirement section and resource requirement list are defined as follows.

```
struct sec_t              // node definition for resource requirement section
{
    int id and            // logical device id
    int t and             // time required
    sec_t *nexts          // pointer of the next node
};
struct rrl_t              //node definition for resource requirement list
{
    sec_t *secs and       // pointer of the resource requirement sections
    int ptime and         // 0 for non periodic task, otherwise the period time
    int reptimes and      // 0 denotes the task is an infinite periodic task,
                          //   otherwise, the number of periods to be repeated
    rrl_t *nexts          // pointer of the next node
};
```

The list node type for processes (Definition 4) is defined as follows.

```
struct process_t           //node definition for process
{
    int pid and            // identifier of the process
    rrl_t *rrl and         // resource requirement list
    int ldid and           // identifier of the logical device
    int pro and            // priority
    int stime and          // start time to wait for the device
    int ltime and          // left required time
    int tslice and         // time slice
    int dline and          // deadline of the current period
    process_t *nexts       // pointer of the next node
};
```

Table 2. Priority functions of typical scheduling algorithms

Algorithm name	Function definition
First Come First Service(FCFS)	$Hp(P_i, P_j) \overset{\text{def}}{=} P_i.stime < P_j.stime$
Shortest Process First(SPF)	$Hp(P_i, P_j) \overset{\text{def}}{=} P_i.ltime < P_j.ltime$
Highest Response Ratio Next (HRRN)	$Hp(P_i, P_j) \overset{\text{def}}{=} (T - P_i.stime)/P_i.ltime < (T - P_j.stime)/P_j.ltime$
Highest Priority Next(HPN)	$Hp(P_i, P_j) \overset{\text{def}}{=} P_i.pro < P_j.pro$
Round Robin(RR)	$Hp(P_i, P_j) \overset{\text{def}}{=} P_i.st < P_j.st$
Multilevel Feedback Queue (MFQ)	$Hp(P_i, P_j) \overset{\text{def}}{=} (P_i.pro = P_j.pro$ and $P_i.pro < P_j.pro)$ or $(P_i.pro < P_j.pro)$
Rate Monotonic(RM)	$Hp(P_i, P_j) \overset{\text{def}}{=} P_i.ltime < P_j.ltime$
Earliest Deadline First(EDF)	$Hp(P_i, P_j) \overset{\text{def}}{=} P_i.dline < P_j.dline$

Scheduler Modeling. The priority of a process to acquire CPU or I/O devices is decided by the scheduling algorithm. Let $Hp(P_i, P_j)$ be the priority function between processes P_i and P_j which are both waiting for a same resource. Based on the data items of the process, it is not hard to write out the priority function for a commonly used scheduling algorithms. Table 2 gives the definitions of the priority functions for the typical scheduling algorithms, where variable T records the current system time. One can chooses the corresponding priority function in case of scheduling verification.

Let $ldSet, sigSch, cStatus$ be the variables saving the header pointer of the logical device list, scheduling signal and status of System Clock respectively. Further, we use the attribute *status* of physical device to keep its running signal. Moreover, we employ boolean variable *alPrem* to denotes whether the shared resource allows to be preempted according to the scheduling algorithm. Note that, before calling function *Sch*, the processes in the waiting queue of a logical device must be arranged on the descending ordered of priority according to the scheduling algorithm.

According to the strategy depicted in Fig. 3, the function *Sch* for the Scheduler in MSVL is as follows.

```
function Sch(ldev_t *ldSet, int *sigSch, int alPrem, int *cStatus)
{
    frame (pLogDev, pPhyDev, fPhyDev, rtn ) and (
        ......;                      /* Define and initialize local variables*/
        while (true) {
            await(*sigSch=1) ;          /*Wait for the scheduling signal*/
            *sigSch:=0 and *cStatus:=0;    /*Pause the system clock */
            pLogDev:=ldSet;
            while (pLogDev!=NULL) {  /*Process each logical device*/
                pPhyDev:=pLogDev→dlist;
                /*Firstly, allocate the idle physical devices to processes*/
                while( pPhyDev!=NULL and pLogDev→pque!=NULL ) {
                    ......; /* allocate the idle device*/
                };
                /*Then, preempt device from the process with lower priority*/
                if(alPrem=1 and pLogDev→type=1) then { /*shareable device*/
                    fPhyDev:=pLogDev→dlist;
                    while( fPhyDev !=NULL and pLogDev→pque!=NULL){
                        pPhyDev:=pLogDev→dlist→nexts;
                        while (pPhyDev!=NULL){     /* compare the priority*/
                            rtn:=Hp(pPhyDev, fPhyDev, rtn);
                            if (0=rtn) then {
                                fPhyDev:=pPhyDev
                            }; pPhyDev:=pPhyDev→nexts
                        };
                        ......;      /* preempt the device*/
                    }
                };
                pLogDev:=pLogDev→nexts;
            }
            *cStatus:= 1; /* Resume the System Clock */
        }
    )
};
```

In function *Sch*, if the scheduling signal *sigSch* arrives (i.e., $*sigSch := 1$), the Scheduler firstly turns off the scheduling signal (i.e., $*sigSch := 0$) and pauses the System Clock (i.e., $*cStatus := 0$). Then, the Scheduler traverses each logical device and allocates the corresponding physical devices to the waiting processes. For a given logical device, the Scheduler firstly allocates its idle physical devices to the standing processes. After that, if scheduling algorithm allows preemption (i.e., $alPrem = 1$) and the logical device is shareable (i.e., pLogDev→type=1) as well as there still exists some waiting processes, the Scheduler will preempt some physical devices from the process with lower priority and allocate them to the waiting higher priority processes. Function $Hp(pPhyDev, fPhyDev, rtn)$ compares the priorities between the two given processes. If the priority of the former one is above than or equal to that of the second one, it returns 1, otherwise 0.

Device Modeling. Let $dev, ldSet, sigSch, sigTime$ be the variables saving physical device, the header pointer of the logical device list, scheduling signal and system time signal respectively. The activities of *dev* given in Fig. 3 are formalized with a MSVL function *PhyDev* as follows.

```
function PhyDev(pdev_t *dev, ldev_t *ldSet, int *sigSch, int *sigTime)
{
    frame( devID, rtn) and (
        ......;                 /* Define and initialize local variables*/
        while (true) {
            await(dev→status=1) ;      /*Wait for the running signal*/
            while(dev→cproc→ltime> 0 and dev→cproc→tslice> 0) {
                await(*sigTime=1);            /*Wait for system clock*/
                ......; /*Decrease left required time and time slice by 1*/
            };
            if(dev→cproc→ltime> 0) then { /*Run out of time slice*/
                AddToLogDev(ldSet, dev→cproc, dev→cproc→ldid )
            }else {
                /* Compute the next resource requirement */
                rtn:=NextReqDev(dev→cproc, &devID, rtn);
                if(rtn> 0) then {
                    AddToLogDev(ldSet, dev→cproc, devID)
                }
            }
            dev→status:=0 and *sigSch:=1;  /*Set scheduling signal*/
        }
    )
};
```

In function *PhyDev*, if the running signal arrives (i.e., dev→status=1), the device begins to repeatedly decrease the left required time and time slice of current process by 1 until either of which equals to 0. Subsequently, if the current time slice runs out (i.e., dev→cproc→ltime> 0), the process is added back to the corresponding waiting queue by calling function *AddToLogDev*. Otherwise, the device computes the next resource requirement by calling function *NextReqDev* and adds the process to the waiting process queue of logical device *devID*. Finally, the device sets its status to idle (i.e., dev→status:=0) and sends the scheduling signal to the Scheduler (i.e., *sigSch:=1).

System Clock Modeling. The task of System Clock is relative simple. It just keeps on increasing the system time T and generating signal $sigTime$ to synchronize physical devices in case of $cStatus = 1$. Let $T, cStatus, sigTime$ be the variables saving the system time, the status of System Clock and system time signal respectively. Function *SysClock* formalizing the System Clock is as follows.

```
function SysClock(int *T , int *cStatus, int *sigTime)
{
    while (true) {
        if(*status=1) then { /*Run out of time slice*/
            *T:=*T+1 and *sigTime:=1
        }else {
            *sigTime:=0 and skip
        }
    }
};
```

3.4 Specification of the Whole System

Let $sigSch, sigTime, cStatus, T$ be the variables saving the scheduling signal, system time signal, system clock status and system time respectively. Further, let $ldArray, pdArray, prArray, secArray, rrlArray$ be the arrays keeping the data of logical devices, physical devices, processes, resource requirement sections as well as resource requirement lists respectively. Since all the physical devices, the Scheduler and the System Clock run concurrently in the computer system, the whole process scheduling over a multi-core computer system can be described by the following MSVL function.

```
function System()
{
    frame(sigSch, sigTime, T, cStatus, ldArray, pdArray, prArray,
        secArray, rrlArray) and (
        ......; //Definitions for local variables
        Init(ldArray, pdArray, prArray, secArray, rrlArray);
        Sch(ldArray, &sigSch, 1, &sigRun, &cStatus) ||
          $||_{pdev \in pdArray}$PhyDev(pdev, ldArray, &sigSch, &sigRun, &sigTime)
          || SysClock(&T, &cStatus, &sigTime)
        }
    )
};
```

where function *Init* initiates the sets of logical devices, physical devices as well as processes. The function will be specified according to the computer system to be verified.

4 Verification Example

In this section, we employ the system model formalized above to verify the process scheduling over a multi-core computer system.

4.1 System Description

Without loss of generality, let the hardware of the computer consist of a 2-core CPU, and 3 different I/O devices named by Laser Printer, Network Card and Hard Disk which only support exclusive access mode. The check list between logical devices and physical devices is given in Table 1. Moreover, the scheduler of the computer selects the algorithm of Earliest Deadline First (EDF), a preemptive real-time scheduling algorithm, to select a process.

Suppose 3 processes P_1, P_2 and P_3 are read to run and their original resource requirement lists are as follows:

$$P_1 : ((4,3),(1,5),(2,3),12)^{10}$$
$$P_2 : ((1,4),(3,3),(4,2),10)^{\omega}$$
$$P_3 : ((3,4),(1,8),(2,5),18)^{\omega}$$

where P_1 is a finite periodic task, P_2, P_3 are both infinite periodic tasks. After being loaded into the memory, each process will be assigned to the standing process queue belonging to the logical device it initially applies for.

Thus, function *Init* of the system model are defined as follows.

```
function Init(ldev_t ldArray[ ], pdev_t pdArray[ ], process_t prArray[ ],
        sec_t secArray[ ], rrl_t rrlArray[ ])
{
   frame(i) and (
      int i and i<==0 and empty;
      while (i< 10) {
         ......; /*Initialize each array into a linked list respectively*/
      };
      ldArray[0].id<== 1 and ldArray[0].dlist<==&pdArray[0] and
         ldArray[0].type<== 1 and pdArray[1].nexts<==NULL and
         ldArray[0].pque<==NULL and empty;
      ......; /*Initialize the other logical devices*/
      pdArray[0].id<== 1 and pdArray[0].cproc<==NULL and
         pdArray[0].status<== 0 and pdArray[0].ldid<== 1 and empty;
      ......; /*Initialize the other physical devices*/
      prArray[0].pid<== 1 and prArray[0].ldid<== 4 and
         prArray[0].stime<== 0 and prArray[0].ltime<== 0 and
         prArray[0].rrl<==&rrlArray[0] and rrlArray[1].nexts<==NULL and
         prArray[0].tslice<== 0 and prArray[0].dline<== 0 and empty ;
      ......; /*Initialize other processes and their resource requirements*/
   )
};
```

After completing the modeling program, we execute the modeling program in the MSV interpreter and analyze the result. The screenshot result of modeling is given in Fig. 3.

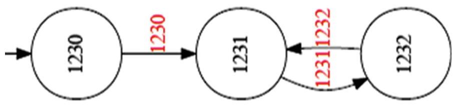
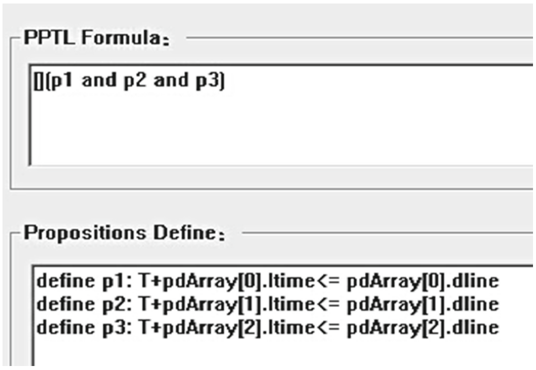


Fig. 3. The result of system modeling

4.2 System Verification

In the following, we use the MSV toolkit to verify the safety property of the process scheduling over the computer, i.e., the task within a period finish successfully before the coming of deadline for each process. It is equivalent to verify that for any process *prc*, the left required resource time *ltime* plus the system



```
[](p1 and p2 and p3)
```

```
define p1: T+pdArray[0].ltime<= pdArray[0].dline
define p2: T+pdArray[1].ltime<= pdArray[1].dline
define p3: T+pdArray[2].ltime<= pdArray[2].dline
```

Fig. 4. The PPTL formula to be verified

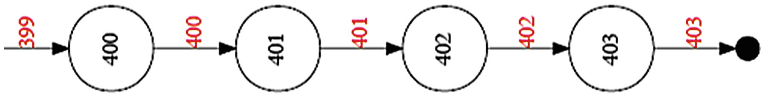


Fig. 5. The result of verification

time T must be less than or equal to the deadline *dline* of current period at any time. The property is depicted in MSVL as follows:

$$safeSch \overset{\text{def}}{=} \bigwedge_{prc \in prArray} always(T + prc.ltime \leq prc.dline).$$

The definition of the property in PPTL formula is depicted in Fig. 4. The result of the verification is given in Fig. 5, which shows property $safeSch$ is not satisfied at state 403.

5 Conclusion

In this paper, we introduce the MSVL based model checking method to verify the safeness of process scheduling over multi-core computer system. A general model for process scheduling is formalized that it can be easily used to verifying all kinds of commonly used scheduling algorithms and typical periodic/nonperoid tasks, and the only work need to do is just adjust necessary parameters. However, the execution time of the process scheduler is ignored. In the future, we will improve the system model to be more close to a real computer. Besides, we will extend the application of the method to more area, such as embedded system, operating system, cloud computing, etc.

References

1. Wijs, A., van de Pol, J., Bortnik, E.M.: Solving scheduling problems by untimed model checking. STTT **11**(5), 375–392 (2009)
2. Ruys, T.C.: Optimal scheduling using branch and bound with SPIN 4.0. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 1–17. Springer, Heidelberg (2003)
3. Lime, D., Roux, O.H.: Formal verification of real-time systems with preemptive scheduling. Real-Time Syst. **41**(2), 118–151 (2009)
4. Duan, Z.: An extended interval temporal logic and a framing technique for interval temporal logic programming. Ph.D. thesis, University of Newcastle Upon Tyne, May 1996
5. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2005)
6. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. Acta Inf. **45**(1), 43–78 (2008)
7. Tian, C., Duan, Z.: Expressiveness of propositional projection temporal logic with star. Theor. Comput. Sci. **412**(18), 1729–1744 (2011)
8. Duan, Z., Koutny, M.: A Framed Temporal Logic Programming Language. J. Comput. Sci. Technol. **19**, 333–344 (2004)
9. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Sci. Comput. Program. **70**(1), 31–61 (2008)
10. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 167–186. Springer, Heidelberg (2008)
11. Wang, M., Duan, Z., Tian, C.: Simulation and verification of the virtual memory management system with MSVL. In: Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 360–365, May 2014
12. Cui, J., Duan, Z., Tian, C., Zhang, N., Zhou, C.: Model Checking μ C/OS-III Multi-task System with TMSVL. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) Formal Methods and Software Engineering. Lecture Notes in Computer Science, vol. 9407, pp. 187–200. Springer, Switzerland (2015)
13. Yu, Y., Duan, Z., Tian, C., Yang, M.: Model checking C programs with MSVL. In: Liu, S. (ed.) SOFL 2012. LNCS, vol. 7787, pp. 87–103. Springer, Heidelberg (2013)
14. Bin, Y., Duan, Z., Tian, C.: Bounded model checking of traffic light control system. Electr. Notes Theor. Comput. Sci. **309**, 63–74 (2014)
15. Ma, Q., Duan, Z., Zhang, N., Wang, X.: Verification of distributed systems with the axiomatic system of MSVL. Formal Asp. Comput. **27**(1), 103–131 (2015)