Simulation and Verification of the Virtual Memory Management System with MSVL

Meng Wang
Institute of ComputingTheory and Technology
Xi'an 710071, P.R.China
Email: wangmnxd@163.com

Zhenhua Duan
Institute of Computing Theory and Technology,
Xi'an 710071, P.R.China
Email: zhhduan@mail.xidian.edu.cn

Cong Tian
Institute of Computing Theory and Technology,
Xi'an 710071, P.R.China
Email: ctian@mail.xidian.edu.cn

***Abstract*—The paging mechanism is widely used in most modern systems to handle the virtual memory. Many page replacement algorithms have been proposed. Therefore, the cor-rectness and reliability of virtual memory management systems become very important. It is essential to formalize and verify the system in a formal way. In this paper, we model the virtual memory management system with MSVL, which is a parallel programming language used for the modeling, simulation and verification of software and hardware systems. Then we employ the model checking approach based on MSVL to verify the interval related properties and periodic repeated properties of the system.**

Keywords— virtual memory, simulation, modeling, verifica-tion

I. INTRODUCTION

When a program is too large, the program cannot run normally since this may lead to the size of required memory greater than that of the available physical memory. In order to solve this problem, the alternative called virtual memory [1] has been developed. The main idea of the virtual memory is that the virtual memory space is divided into many segments and the currently used parts is located in the physical memory and others is in the hard disk. Although the actual size of the physical memory does not increase, it makes user processes have an illusion that there is a large size memory in the system.

The means of demand paging is used widely in the modern virtual storage system [2], [3]. The virtual address space is divided into pages and the corresponding unit of physical memory is a page frame. The size of a page and a page frame is generally from 0.5KB to 64 KB in the modern computer systems.

The virtual memory management system contains two important parts: (1)Memory Management Unit(MMU), which translates from a virtual page address to a physical one. (2)Page Fault Handler(PFH), which reacts by moving the requested page to physical memory [4] when a page fault occurs.

Systems manage virtual pages through the page table space. The page table entry is the unit of the page table. It contains the information whether a page is in the physical memory or not as well as the information of the pattern that the page is accessed and the page frame number which is used to find the corresponding physical page.

The virtual address is sent to the MMU and the MMU translates from a virtual page address to a physical one. The virtual address is divided into the virtual page number and the offset. The MMU gets the page table entry according to the virtual page number. When the page is not located in the physical memory, it will lead to a page fault.

The system checks the validity of the address. It looks for an idle page frame in the physical memory to load the required page if the address is valid. If there is no idle page frame, the page replacement algorithm is implemented to search for a page in the physical memory to swap out. If the chosen page frame is dirty which means it has been modified, it must be written back to the disk. Once the page frame is clean, the required page is loaded into the physical memory.

The system needs to find a page to evict when the page fault occurs and there is no idle page frame. In order to improve the performance of the virtual memory management system, choosing a least used page to evict becomes important. Choosing a frequently used page to move out may bring unnecessary extra overhead, since the page may be accessed soon. There are many page replacement algorithms to solve this problem. We will use the aging algorithm which is an improvement of the Not Frequently Used (NFU) strategy [5].

Although the MMU and PFH can improve the performance of the execution of a program, the required page replacement algorithms or programs may cause errors during its execution. This is critical to a large real time program. Therefore, to ensure the correctness and reliability of the page replacement algorithms or programs is important. Simulation and model checking are widely used approaches for formal verifications.

MSVL(Modeling, Simulation and Verification Language) [6] is a parallel programming language used for modeling,simulation and verification of systems. Systems and properties can all be described with MSVL. In this paper, we use MSVL to formalize the virtual memory management of the operating system, and employ the model checking approach based on MSVL to check the safety, interval related properties and periodic repeated properties of the system.

This paper is organized as follows. Section 2 briefly introduces MSVL which is used to describe the model. In section 3, the virtual memory management is formalized with MSVL. In section 4, the verification results are demonstrated. Finally,

978-1-4799-3776-9/14/$31.00 ©2014 IEEE

the conclusions are summarized in section 5.

II. MODELING, SIMULATION AND VERIFICATION LANGUAGE (MSVL)

The Modeling, Simulation and Verification Language is an executable subset of PTL [6], [7], [8]. It can be used to model, simulate and verify concurrent systems.

Syntax As an executable subset of PTL, MSVL consists of expressions and statements. Expressions can be regarded as PTL terms, and statements as PTL formulas. The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

$$\begin{array}{lll} e & ::= & n \mid x \mid \bigcirc x \mid \ominus x \mid e_0\, op\, e_1 (op ::= + \mid - \mid * \mid / \mid mod) \\ b & ::= & true \mid false \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \wedge b_1 \end{array}$$

where n is an integer and x is a variable. The elementary statements in MSVL are defined as follows:

1. Assignment	$x = e$
2. P-I-Assignment	$x \Leftarrow e$
3. Conjunction	$p \wedge q$
4. Selection	$p \vee q$
5. Next	$\bigcirc p$
6. Always	$\Box p$
7. Termination	empty
8. Sequential	$p\,;\, q$
9. Local variable	$\exists x : p$
10. State Frame	$\mathsf{lbf}(x)$
11. Interval Frame	$\mathsf{frame}(x)$
12. Projection	$(p_1, \ldots, p_m)$ prj q
13. Parallel	$p \parallel q \stackrel{\text{def}}{=} p \wedge (q\,;\, \mathsf{true}) \vee q \wedge (p\,;\, \mathsf{true})$
14. Conditional	$\mathtt{if}\ b\ \mathtt{then}\ p\ \mathtt{else}\ q \stackrel{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
15. While	$\mathtt{while}\ b\ \mathtt{do}\ p \stackrel{\text{def}}{=} (b \wedge p)^* \wedge \Box(\mathsf{empty} \rightarrow \neg b)$
16. Await	$\mathsf{await}(b) \stackrel{\text{def}}{=} (\mathsf{frame}(x_1) \wedge \cdots \wedge \mathsf{frame}(x_h)) \wedge \Box(\mathsf{empty} \leftrightarrow b)$ where $x_i \in V_b = \{x \mid x \text{ appears in } b\}$

where x is a variable, e denotes an arbitrary arithmetic expression, b is a boolean expression, and $p_1, \ldots, p_m$, p and q are MSVL programs. All the statements can be divided into two categories: basic statements and composite ones. The assignment $x = e$, the positive immediate assignment $x \Leftarrow e$, empty, $\mathsf{lbf}(x)$, and $\mathsf{frame}(x)$ are basic statements, and the others composite ones;

The assignment $x = e$ presents that the value of the variable x is equal to that of the arithmetic expression e. The positive immediate assignment $x \Leftarrow e$ can be defined as $x = e \wedge p_x$, where p_x is the assignment flag for variable x. $p \wedge q$ means that p and q are executed concurrently and share all the variables during the execution, while the disjunction statement $p \vee q$ presents p or q are executed. The next statement $\bigcirc p$ means that the program p is executed at the next state. The always statement $\Box p$ presents that p holds in all the states. The termination statement empty means that the current state is the last one of the interval. The sequential statement $p\,;\, q$ means that p holds from now until its termination and then q holds from that time and be executed. The local variable statement $\exists x : p$ permits p to use a local variable x. 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, while $\mathsf{frame}(x)$ means that variable x always keeps its old value over an interval if no assignment to x is encountered. The projection statement $(p_1, \ldots, p_m)$ prj q is executed on different time scales. The parallel statement $p \parallel q$ is another parallel computation, which allows p and q to specify their own intervals. The conditional statement $\mathtt{if}\ b\ \mathtt{then}\ p\ \mathtt{else}\ q$ means that if the boolean condition b is true , then the process p is executed, otherwise the process q is executed. The while statement $\mathtt{while}\ b\ \mathtt{do}\ p$ means that the process p will be executed repeatedly as long as the condition b holds. When the condition b is false, the while statement terminates. The await statement $\mathsf{await}(b)$ means that the process does not terminate until the condition b becomes true.

A model checking tool named MSV has been developed based on MSVL. It has three modes: modeling, simulation and verification.

III. FORMALIZATION OF THE VIRTUAL MEMORY MANAGEMENT SYSTEM

A. Overview of the Virtual Memory Management System

The strategy of the virtual memory management is based on Mach [9], [10]. Fig.1 shows how the virtual memory management works. When a process starts, a mapping between the process and pages needs to be established by the external pager. Once the process is being executed, the virtual addresses are translated to physical ones by the MMU. When the page is not in the physical memory, it leads to a page fault. If a page fault occurs, the page fault handler is invoked to find the virtual page that the process requires. Then a message is sent to the external pager about the needed page. The external pager requests for the page from the disk. Afterwards the required page reaches the external pager and is copied to the external pager's address space. After that, a message about the position is sent to the page fault handler. The page fault handler looks for an idle page frame in the physical memory and if there is no one, it finds a page to evict according to the aging algorithm. Then the MMU handler is requested to load the page to the correct position in the user address space and the process continues to run. The whole process is described by MSVL in order to verify the properties of the system in a formal way.

B. Page Table Entry

In this paper, the virtual address is 16 bits. The size of a virtual page is 1KB. Thus, we need 10 bits to denote the offset and 6 bits to denote the page number which means there are 64 virtual pages in the system. In our system, there are 32 page frames in the physical memory. Therefore, 5 bits are needed to express the page frame number.

To formalize the page table, we denote an integer array $Page[64]$ which has 64 elements to store all the page table entries. Each element of the array is a page table entry and each bit of the element has its given meaning which is described below.

The structure of the page table entry is shown in fig.2.

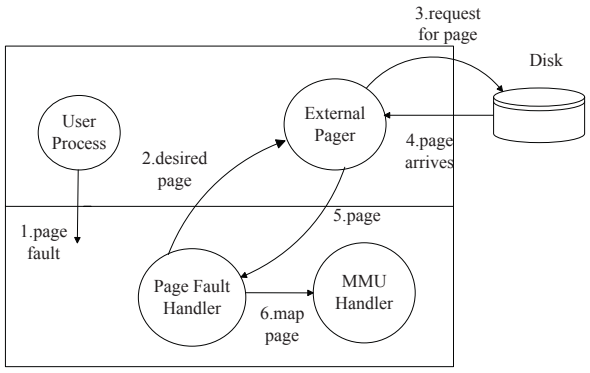

Fig. 1. Virtual Memory Management

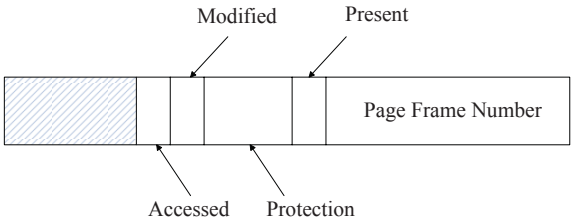

Fig. 2. Page Table Entry

Page Frame Number: it is one of the most important domains. It is used to calculate the physical address and access the needed page in the memory. It can be gotten by $Page[PageNum]\%32$.

Present: 1 bit is used to denote it. When the value is 1, it means the page table entry is effective; otherwise, the corresponding page is not in the physical memory. It can be gotten by $(Page[PageNum]/32)\%2$.

Protection: 3 bits are used to denote it, which represent the types of access to the page. They correspond to reading, writing and executing the page respectively. It can be gotten by $(Page[PageNum]/64)\%8$.

Accessed: 1 bit is used to denote it. This flag of the page table entry is set 1, once the processor accesses the page that the page table entry maps. It can be gotten by $(Page[PageNum]/1024)\%2$

Modified: 1 bit is used to denote it. When the processor performs a writing operation on a page, it will set the flag of the corresponding page table entry 1. It can be gotten by $(Page[PageNum]/512)\%2$.

C. Memory Management Unit (MMU) Handler

Modern multi-user multi-process operating systems require MMU. When the processor enables the MMU, the virtual address will be sent to the MMU instead of the memory bus and the MMU handler translates from the virtual address to a physical one.

Fig.3 shows the working principle of the MMU handler. The input 16-bit virtual address is divided into a 6-bit page number and a 10-bit offset. The page number is used as the page table index to find the corresponding page frame number. If the bit of present is 0, which means the page is not in the physical memory, the MMU triggers a page fault interrupt and the page fault handler reacts to it by moving the requested page to the physical memory; otherwise, it will find the page frame number from the page table and add the page frame number to the offset to get a 15-bit physical address [11].

In our model of the virtual memory management, we denote some main variables. We use va as the virtual address, $phys\_addr$ as the physical address. We use MSVL to formalize the MMU handler as follows:

```
MMU =def
(
  frame(exist) and
  ( PageNum:=va/1024;
    exist:=(Page[PageNum]/32)%2;
    if(exist=0)
    then{page_fault:=1}
    else{ page_fault:=0 and
      phys_addr:=(Page[PageNum]%32)*1024+va%1024
      and TransOk:=1 and
      if((Page[PageNum]/1024)%2=0)
      then{ Page[PageNum]:=Page[PageNum]+1024}
      else{skip}
      }
  )
)
```

Since in our system 10 bits are used to denote the offset, we can get the virtual page number $PageNum$ through va divided by 1024 and the offset through $va\%1024$. According to the value of $PageNum$, we find the corresponding page table entry $Page[PageNum]$. The 6th bit denotes whether the page is in the physical memory, thus we make the $Page[PageNum]$ divided by 32 to get the value of this flag. When the page is in the memory, the page frame number can be obtained by $Page[PageNum]\%32$. The physical address $phys\_addr$ can be calculated according to the offset and the page frame number. Also we should set the accessed flag of this page to be one by adding 1024 to the page table entry $Page[PageNum]$ to denote that the page is accessed in this clock tick, which helps the system to choose a page to swap out when a page fault occurs.

D. Aging Algorithm

The aging algorithm [5] is an improvement of the Not Frequently Used (NFU) page replacement algorithm. Each page is connected to a software counter in the NFU algorithm. The accessed flag of all the pages is added to the corresponding counter at each clock tick. However, the NFU algorithm has its disadvantages that the way to denote the access frequency is not good. Since every time the page is accessed, the counter is added by 1, the effect of access at begin is treated like the access later. This may lead the system to evict a useful page instead of a page that is not used again.

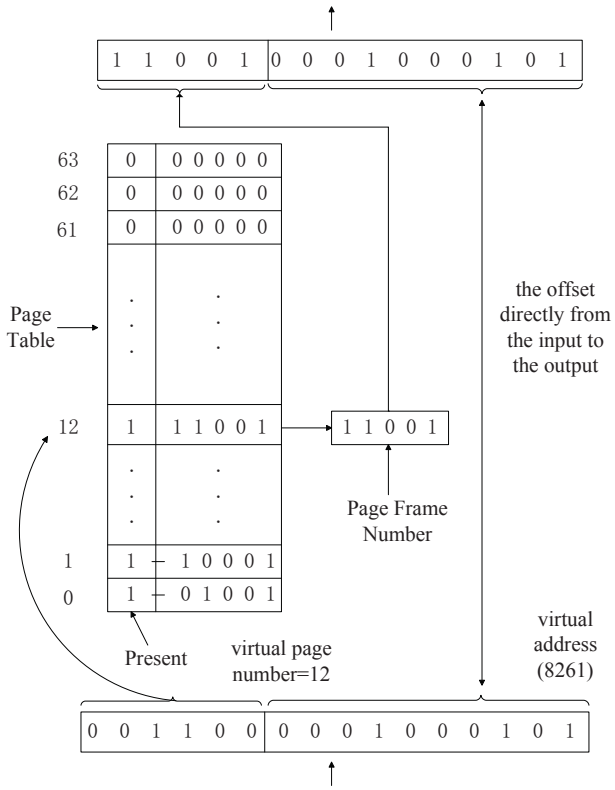

Fig. 3. MMU Handler

To solve this problem, the aging algorithm modifies the way that the accessed flag is added to the counter. Different with the NFU algorithm, before the accessed flag is added to the counter, the value of the counter is shifted right by 1. Afterwards, the accessed flag will be added to the most left bit of the counter.

In this paper, an array $count[32]$ is used to denote each counter corresponding to the pages in the physical memory. At begin, every element of the array is set 0. Another array $memory[32]$ denotes the page frames in the physical memory. The initial value of each element is -1, which means the page frame is idle. When a page is loaded, the value of the element of the array is set the page number. The model of the aging algorithm to find a page to replace is shown as follows:

```
aging ≝
  frame(i,page_index) and
  ( int i and int page_index and skip;
    i:=0 and rep_page:=0 and page_index:=Memory[0];
    while(Page[page_index]/1024=1)
    { i:=i+1 and rep_page:=rep_page+1;
      page_index:=Memory[i]
    };
    while(i<31)
    { page_index:=Memory[i+1];
      if(Count[rep_page]>Count[i+1] and
         Page[page_index]/1024=0)
      then {rep_page:=i+1}
      else{ skip};
      i:=i+1
    }
  )
```

rep_page denotes the page number of the page which will be moved out of the physical memory according to the aging algorithm. We can find the first page which is not accessed in this clock tick but in the physical memory through the first while-loop. In the second while-loop, we traverse the rest of the pages in the memory. According to the value of the array $Count[\ ]$,we can find a page to evict.

After each clock tick, we add the corresponding accessed flag to each element of $Count[\ ]$ according to the replacement algorithm and clean the accessed flag by subtracting 1024 from the page table entry $Page[PgNum]$ which corresponds to the page accessed in this clock tick. The corresponding MSVL program is shown as follows:

```
AddCount ≝
  frame(i,PgNum) and
  (
   int i and int PgNum and skip;
   i:=0 and PgNum:=0;
   for 32 times do
   {
    PgNum:=Memory[i];
    if(PgNum!=-1)
    then{
      Count[i]:=Count[i]/2+((Page[PgNum]/1024)%2)*128;
      if((Page[PgNum]/1024)%2=1)
      then{ Page[PgNum]:=Page[PgNum]-1024}
      else{skip}
    }
    else{skip};
    i:=i+1
   }
  )
```

E. Page Fault Handler

When a page fault occurs, the page fault handler as one of the main parts in the virtual memory management system works [12]. The user process continues to run after its execution. Here, we focus on the page replacement, thus the details of some registers and messages are ignored. The details about how the page fault handler works are listed as follows:

(1) When an invalid access to the page occurs, the system tries to find the desired virtual page. Once the virtual address which triggers the page fault is known, the page fault handler checks the validity of the address. The address space 0x0000∼0xbffff (49151) is the user space and 0xc000∼0xffff is the kernel space. We assume that when the user process accesses an address in the kernel space, the access is invalid. If the address is invalid, the process will be killed. Here we set the variable $proc\_kill$ to be 1 to denote it. If the address is valid, go to (2).

(2) The page fault handler looks for an idle page frame in the physical memory. If there is no one, it executes the page replacement algorithm to find a page to swap out. Here the function $aging(Count, Page, Memory, i)$ will be called, where the value of i denotes the page number of the page that will be replaced. Then the $Count[i]$ should be set 0 again to start counting the frequency of the access to the new page.

(3) When the page frame chosen to move out is dirty, the page fault handler will arrange the page written back to the disk.

(4)Once the page frame is clean, the page fault handler will send a message to the external pager to tell it the desired page. The external pager looks for the position of the required page in the disk and reads it from the disk. The desired page is loaded into the physical memory. Here the corresponding value of $Memory[i]$ is assigned by the required page number $PageNum$, where i is the page frame number that denotes the position where the desired page is loaded.

IV. SIMULATION AND VERIFICATION

As a case study, we simulate how the virtual memory management of the operating system works when there are two processes running in the processor. The case of more processes running in the processor has the same principle.

We define a time sequence by defining a global int variables $Clock$. At first, there is no page in the memory. When the process accesses the address 0x0003, a page fault occurs. The value of $page\_fault$ is set 1. Then the page fault handler function PFH is called. The PFH finds an idle memory and the desired page 0 is loaded into $Memory[0]$. The page table entry $Page[0]$ is set 1312 afterwards, which means the process accesses the page with the read-only mode at this clock tick and the corresponding page frame number is 0.

After a period of time, there will be no idle page frame in the physical memory. The page fault handler uses the aging algorithm to find a page to swap out.

Every process needs to maintain its own page table. In the case of two processes, we define two page tables $Page1[64]$ and $Page2[64]$ each for one process. Each process has its independent address space, which means the same virtual address of different processes is mapped to different physical addresses through MMU and one process cannot access the data of another process. With the function of simulation of the MSV toolkit, we can see the same va is translated into different values in different processes. As shown in fig.4, $va1$ is 3, which is equal to $va2$. However, they are translated into different physical addresses 3 and 1027 respectively.

Not all the virtual addresses can be accessed by the process. By simulation, we can find that the virtual addresses cannot be translated in the following two cases:

(1) The process tries to access the kernel space.

(2) The process accesses more than 32 pages in one clock tick.

The kernel space is 0xc000∼0xffff that means the pages after Page 48 cannot be accessed. As shown in fig.5, when a process accesses the address 49154, the value of $proc\_kill$ is 1, which means the process cannot access the address and is killed by the system. The virtual address is not translated to the corresponding physical address.

There are 32 page frames in the physical memory. Therefore, when the process accesses more than 32 pages in one clock tick, the virtual address cannot be translated to the physical address. In the simulation, the process accesses 33 pages in one clock and the result is shown in fig.6. The value of $proc\_kill$ is 1 , this is to say, the process is killed and cannot access the address.

We can also use MSV toolkit to verify the safety property of the virtual memory management system. As an example, we can treat "the kernel space cannot be accessed by the process" as a safety property. To verify the property, we need to define the following propositions:

define p1: Page[48]=0;
define p2: Page[49]=0;
...;
define p16: Page[63]=0;

We can describe this property with MSVL:

$\Box(p1 \wedge p2 \wedge p3 \wedge ... \wedge p16)$.

An interval related property is verified here.

As we know, when the virtual address is sent to the MMU, the MMU translates from the virtual address to a physical one or triggers a page fault interrupt when the page is not in the physical memory. In our system, it takes one unit of an interval to set the virtual address and the type of the access. Then three units of an interval is taken for the reduction of the MMU. In order to verify the function of the MMU, the following propositions are defined:

define p: TransOk=1;
define q: page_fault=1;

The property can be defined with MSVL as follows:

$len\ 4; (p\ or\ q); true$

The system should satisfy the property every time the process accesses the virtual address. Therefore, we can give a periodic property. After the process of the processor, the virtual address will be translated to a physical address, or the process will be killed by the processor with the reasons that have been mentioned before. To verify the property, we need to define the following propositions:

define p: TransOk=1;
define q: page_fault=1;
define r: proc_kill=1;

The property can be defined with MSVL as follows:

$(len\ 4; (p\ or\ q); true; (p\ or\ r)\ and\ skip)^*; true$

All the properties mentioned above are verified with the MSV toolkit. Since they are all satisfied, the results of verification are similar with each other. For simplicity, we give a figure of the results here. As shown in Fig.7, there is only one node in the figure when each property is verified, that is to say there is no counterexample. Thus all the properties mentioned are satisfied.

At last, a property that the system does not hold is verified. As mentioned before, the process will be killed when it accesses the kernel space. Thus, we can verify whether the

```
State【25】Global: Clock=0 and Count=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] and Memory=[0,0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
and Mode=4 and Modified=0 and Page1=[1312,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] and Page2=
[1313,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] and PageNum=0 and page_fault=0 and phys_addr1=3 and phys_addr2
=1027 and proc_kill=0 and va1=3 and va2=3
```

Fig. 4. The result of simulation (1)

```
State【10】Global: Clock=0 and Count=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] and Memory=[-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
and Mode=4 and Modified=0 and Page1=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] and Page2=
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] and PageNum=48 and page_fault=1 and phys_addr1=nil and phys_addr2
=nil and proc_kill=1 and va1=49154 and va2=nil
```

Fig. 5. The result of simulation (2)

```
State【1377】Global: Clock=1 and Count=[128,192,192,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128] and
Memory=[37,4,6,13,8,10,9,12,11,14,7,17,15,16,20,18,19,21,22,23,24,26,27,25,28,29,31,30,33,35,32,36] and Mode=6 and Modified=1 and Page1=
[288,0,0,0,929,0,290,298,292,294,933,936,295,291,297,300,301,939,303,304,942,945,306,307,948,951,309,310,312,313,315,954,318,316,0,957,319,928,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0] and Page2=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] and PageNum=38 and page_fault=1 and phys_addr1=1
and phys_addr2=0 and proc_kill=1 and va1=38920 and va2=nil
```

Fig. 6. The result of simulation (3)

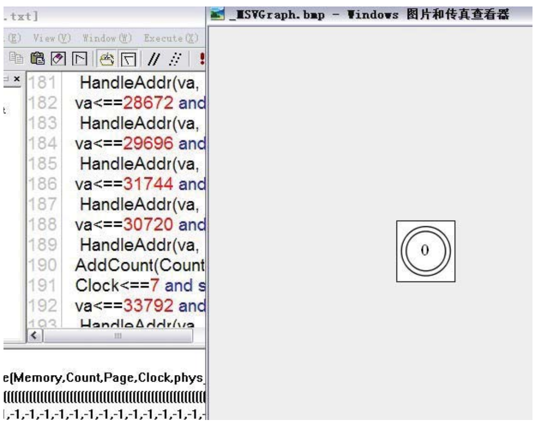

Fig. 7. The result of verification(1)

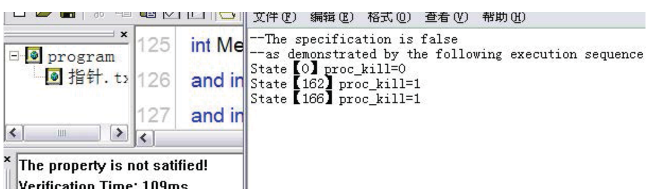

Fig. 8. The result of verification(2)

value of $proc\_kill$ keeps to be 0. To do so, we need to define the following propositions:

define p: proc_kill=0;

The property can be defined with MSVL as follows:

$\Box(p)$

The result of the verification is shown in fig.8. The property is not satisfied, and a counterexample is given. We can see at state 162 the value of $proc\_kill$ is equal to 1, which makes the property not satisfied.

V. CONCLUSION

In this paper, the virtual memory management system is simulated and some properties are verified with MSVL. The system manages the virtual memory by handling the pages and uses the aging algorithm to find a page to swap out when a page fault occurs and there is no idle page. This process is simulated by an MSVL program. Further, we verify the system's properties using model checking approach with the MSV toolkit. In the future we will study more details about how the virtual memory management system works to establish a more accurate model. Furthermore, we will further verify more useful and interesting properties that the system should hold.

REFERENCES

[1] Peter J Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[2] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal pervasive verification of a paging mechanism. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 109–123. Springer, 2008.

[3] Artem Starostin. *Formal verification of demand paging*. PhD thesis, PhD thesis, Saarland University, Computer Science Dept, 2010.

[4] Andre Bensoussan, Charles T Clingen, and Robert C. Daley. The multics virtual memory: concepts and design. *Communications of the ACM*, 15(5):308–318, 1972.

[5] Heikki Paajanen. Page replacement in operating system memory management. 2007.

[6] Zhenhua Duan. *An extended interval temporal logic and a framing technique for temporal logic programming*. PhD thesis, University of Newcastle upon Tyne, 1996.

[7] Zhenhua Duan. *Temporal logic and temporal logic programming*. Science Press, 2005.

[8] Zhenhua Duan and Maciej Koutny. A framed temporal logic programming language. *Journal of Computer Science and Technology*, 19(3):341–351, 2004.

[9] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. 1986.

[10] Richard Rashid, Avadis Tevanian Jr, Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *Computers, IEEE Transactions on*, 37(8):896–908, 1988.

[11] Alexander Vaynberg and Zhong Shao. Compositional verification of a baby virtual memory manager. In *Certified Programs and Proofs*, pages 143–159. Springer, 2012.

[12] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, and Jeffrey Eppinger. *The duality of memory and communication in the implementation of a multiprocessor operating system*, volume 21. ACM, 1987.