Assignment 4 Report

Aditya Trivedi, 190101005

Topic: Process Management in Linux

Link to Video : https://youtu.be/5_Exv98s00k
Link to PPT:
https://iitgoffice-my.sharepoint.com/:p:/g/personal/atrivedi_iitg_ac_in/EcNdonBeK_ZLtKzm8LkBP3kB0RJuJXwc8TfrL9v5ih_0gQ?e=QHgXpa

Table of Contents

Introduction :

Before we start discussing the management of processes, we must know how a process is represented in different OSes. Here we will be only focussing on xv6 and Linux.

In xv6, a process is represented by a struct proc

```
struct proc {
 uint sz;                     // Size of process memory (bytes)
 pde_t* pgdir;                // Page table
 char *kstack;                // Bottom of kernel stack for this
process
 enum procstate state;        // Process state
 int pid;                     // Process ID
 struct proc *parent;         // Parent process
 struct trapframe *tf;        // Trap frame for current syscall
 struct context *context;     // swtch() here to run process
 void *chan;                  // If non-zero, sleeping on chan
 int killed;                  // If non-zero, have been killed
 struct file *ofile[NOFILE];  // Open files
 struct inode *cwd;           // Current directory
 char name[16];               // Process name (debugging)
};
```

and the context for each process (used when a switch occurs) is stored as follows :

```
struct context {
 uint edi;
 uint esi;
 uint ebx;
 uint ebp;
 uint eip;
};
```

In Linux each process is represented by a task_struct data structure (task and process are terms that Linux uses interchangeably). The task vector is an array of pointers to every task_struct data structure in the system. It is a very large structure and the source code for its definition is given below :
https://github.com/torvalds/linux/blob/master/include/linux/sched.h

To summarise, it has the following subfields :

1. State
2. Scheduling Information
3. Identifiers
4. Inter-Process Communication
5. Links
6. Times and Timers
7. File system
8. Virtual memory
9. Processor Specific Context

What exactly is Process Management
Broadly, process management refers to the creation, scheduling, switching and termination of processes. Let us compare each in xv6 and Linux.

Creation :
It is similar in both the OSes, both utilise the fork() function to create new processes. This function, as studied in class, allocates new memory for the child and copies the parents content into it. Linux avoids naive copying by using **copy-on-write** which saves both RAM and time while copying these contents.
Additionally, Linux has support for kernel-level threading. This means each process is associated with one or more lightweight “threads”. This exploits multi-processor systems and leads to parallelism.

Switching & IPC (Inter-Process Communication) :
Both the OSes use a process table and maintain the context using a stack or process control block. It is however much advanced in Linux. While for IPC, xv6 uses the simplest primitive possible, pipes, i.e., unidirectional flows. While Linux has advanced techniques such as shared memory, queues for messages, etc.

Scheduling :
xv6 has a very naive scheduling algorithm that is similar to (but not exactly) like round-robin. This is easy in implementation but not the most efficient. While Linux uses an advanced algorithm called CFS. This will be explained further ahead. A brief history about Linux below, reveals that there has been ongoing research and attempt to improve scheduling which is why it is also a focus of my discussion.

Version - Method

1.2 - circular queues & round-robin
2.2 - scheduling classes and policies
2.4 - O(N) scheduler based on time epochs
2.6 - O(1) scheduler based on runnable queues – active, expired,
- ❖ more scalable
- ❖ metrics to determine whether tasks were I/O-bound or processor-bound.
- ❖ a large mass of code needed to calculate heuristics

2.6.21 - patch from Con Kolivas - Rotating Staircase Deadline Scheduler (RSDL)
- ❖ incorporated fairness with bounded latency

Termination :

Both the OSes use the exit() system call to terminate a process. Furthermore, terms such as orphan, zombie, etc mean the same in both contexts.

Missing features in xv6 :

The following features are the ones that can be implemented in xv6 and are relevant to our discussion:
1. Kernel-thread process
2. An advanced scheduling algorithm such as CFS

1. Kernel-Thread Processes :

In xv6, threads and processes are synonymous, i.e., each process has one thread mapped to it. While in Linux and modern operating systems, usually every process is associated with multiple threads which leads to parallelism and can exploit multi-core chips.

For now, we introduce threads by implementing one-to-one process threads. This means each process has an associated thread. Upon this, we can build multi-thread processes, which will exploit parallelism.

We will implement 2 functions :
- create_thread()
- assoc_thread()

these will use 2 system calls : clone() and join().

To maintain synchronisation and ensure the integrity of shared data, we will also add a mutex mechanism. This will include :

- Create_mutex()
- Lock_mutex()
- UnLock_mutex()

Implementation

Firstly a thread struct will be defined and declared in threads.h. Then the clone and join system calls will be added, which will be used to create and associate threads with processes.

```
struct my_thread {
    int pid;
    int used;
    void * ustack;
};

//We are going initialize the table
int thread_table_init = 0;
struct my_threads threads[MAX_THREADS];
```

Clone : Its job is to direct the thread to point to the user stack, receive the func to the appropriate register and pass the arguments of that func as well. The PC is also given a fake address of 0xFFFFFFFF. It finally sets the state of that thread's PID to RUNNABLE while using the acquire() and release() lock functions.

Join : It is similar to wait(), additionally, it makes sure that the thread will release itself and kill the process.

Pseudocodes :

```
clone(func, arg, stack){
     new_proc = allocate_process()
     new_proc.initialise() // set parent, size, stack, trapframe
     new_proc.CopyContents(ParentProc)
     new_proc.Registers = GiveFunctionToRun(func, stack)
     move nw_proc to top of stack and pass the provided arguments
     // identical to fork
     for(x = 0; x < NOFILE; x++){
          if(proc->ofile[x]){
               newProcess->ofile[x] = filedup(proc->ofile[x]);
          }
       }
     Set PID
```

```
        lock(process_table);
        Set Process as RUNNABLE
        unlock(process_table);
        return PID
}
```

join(stack){

```
        havekids = 0
        lock(process_table)
        for(each process P in table){
                if(P.parent != global_proc || P.isNotThread) continue
                havekids = 1
                P.initialise()
                release(process_table)
                return P.pid
        }
        if(havekids == 0 || global_proc.isKilled){
                release(process_table)
                return -1
        }
}
```

create_thread(function, arguments){

```
        Initialise the thread_table
        // init
        int pid;
        newStack = malloc(KSTACKSIZE);
        // using clone()
        pid = clone(function, arguments, newStack);
        for each thread in table{
                if( thread.used == 0){
                        thread.pid = pid
                        thread.ustack = newStack
                        thread.used = 1
                }
        }
        return pid
}
```

assoc_thread(){

```
        int temp;
        for(each thread in table){
                if(thread.isUsed){
                        temp = join(thread.stack)
                        if(temp > 0){ // legal
                                for(each thread2 in table){
                                        if(thread2.isUsed
                 && thread2.pid == temp){
```

```
                                        // remove it
                                        free(stack)
                                        thread2.clear()
                                         // sets pid, stack, used to
                                   0/false/NULL
                                   }
                              }
                              break
                         }
                    }
               }
               return temp;
          }
```

The mutex portion is simple, we add the declaration of mutex_lock in defs.h. We define the struct in types.h. Now the following function are defined in spinlock.c.

Pseudocodes :

```
create_lock(){
     mutex_lock new_lk = new Lock()
     return new_lk
}

Lock_mutex(mutex_lock *lk, proc* curr){
     while(lk.isLocked){ // abstracted
          // wait
     }
     // now we have acquired
     lk.owner = curr // setting owner in case we need it
}

unLock_mutex(mutex_lock *lk){
     lk.release() // abstracted
}
```

2. CFS - Completely Fair Scheduler :

This scheduling algorithm is based on fairness between all processes. Here all the participating processes receive fair CPU time. If there are P processes, each will receive 1/P fraction of CPU time.

To determine the balance of time a process has already run, a virtual runtime for each process is maintained in the form of vruntime variable. Whenever there is a switch, if some process ran for T amount of time, then its vruntime will be incremented by T.

The smaller the vruntime, higher its need for the processor. Thus, whenever there is a yield or switch, process with least runtime is given priority to run next. The scheduler works in following steps :

1. Choose process with least vruntime
2. Compute the timeslice dynamically
3. Set timer according to timeslice
4. Begin execution

Intially each process has explicit priority and equal timeslice. Over time the timeslice is computed dynamically. And the priority decays.

Choosing process with least vruntime :

We maintain all active processes (their vruntimes) in a red-black tree. It is a height balanced tree which ensures that all updates, insertions, and deletions occur in O(logP) time, where P is the total number of processes.
Structure of red black tree :

- Each node is a process with the key value being vruntime
- For any node, any node in left sub tree has lesser vruntime than any child of right subtree
- It is height balanced and self balancing.

Thus in our case, we require the process with least runtime, this can be obtained in O(1) if cached, instead of O(logP) by maintaining a pointer to left most node in the RB Tree. Hence for this portion, we need a get_least_vruntime_node() function.

So whenever we call scheduler :

- ❖ Find out process with least vruntime
    - ➢ if cached in pointer, get in O(1)
    - ➢ else, use get_least_vruntime_node(), in O(logP)
- ❖ If this is the process we are currently running, no switch needed
- ❖ Else swapin this new process
- ❖ If current process is still pending, aka, runnable, swap it out and insert it back into the tree. This again takes only O(logP) time.

This was vanilla CFS, however we can also include user and system defined priorities. This priority is used in an weighted manner to determine runtime. If a process ran for T units of time, its vruntime will be updated as follows :

vruntime += T*prio;

where prio is an integer representing the priority.
This means for high priority processes, (smaller prio values), the time runs slower.

These priorities lie from 0 to 139, where only 100-139 are for users, while rest are reserved for real time processes. Also another value called the niceness values is used to determine priority:

for normal processes: prio = 20 + NI (NI is nice and ranges from -20 to 19)
for real time processes: prio = - 1 - real_time_priority (real_time_priority ranges from 1 to 99)

This ensures that high priority real time processes do not get starved.

To incorporate this, we will also need to add a priority property to each process and update vruntime accordingly.

The CFS is used in linux because it dynamically allots timeslices and ensures balance of fairness. Let us see how it avoids starvation of IO bound processes. In any naive scheduling algorithm such as round robin, once an IO bound process leaves for IO, it gets its processor time after a hefty interval. But here, since the CPU time is a short burst - the vruntime is incremented by a smaller value, hence it stays in the left of our red black tree.

Calculating timeslice dynamically, and the concept of latencies is out of scope for this discussion, hence we only keep above mentioned points in our implementation.

Implementation :

In a previous assignment we implemented a scheduler by modifying :

- scheduler() function in proc.c
- proc struct in proc.h
- set_priority() function in proc.c
- Makefile

We also added a yield() system call by modifying :

- syscall.h
- syscall.c
- user.h
- usys.S

Here too we add priority and vruntime in proc struct.
We create a Red Black Tree by making a struct RB_Tree_Node. And add the following associated functions :

createNode()
deleteNode()
insertNode()
get_least_vruntime_node()

These are very standard functions. Now the node of the red black tree has key = vruntime and a pointer to a process in the proc table.

In the scheduler, we have following pseudo code :

```
if (process switch){
      currProc = proc // save the current process

      repProc  = get_least_vruntime_node() // determine replacement
      if(repProc == currProc){
            // do nothing
      }
      else{
            swapOut(curProc)
            swapIn(repProc)
      }
}
```

Now also we must update the vrutime as follows

```
if (process.timeslice is expired){
      lock(process)
      vruntime += timeslice
      unlock(process)
}
```

These are the important pseudocodes, rest is similar to what we have already done as a course assignment, modifying Makefile, adding declarations in .h files, etc.

One can also add a sanity test to verify theoretical calculations.

This concludes the explanation, comparison and implementation of 2 new features in xv6.

References :

https://www.cs.columbia.edu/~junfeng/13fa-w4118/lectures/l07-proc-xv6.pdf
https://developer.ibm.com/tutorials/l-completely-fair-scheduler/
https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf
https://www.baeldung.com/linux/process-vs-thread
http://www.science.unitn.it/~fiorella/guidelinux/tlk/node45.html
https://www.cse.iitb.ac.in/~mythili/os/notes/old-xv6/xv6-process.pdf

Note :
Following topics are in greater detail in the report and online, compared to PPT.

1. Technicalities of assoc_thread and create_thread
2. Explanation of Red Black Tree, a standard data structure
3. Concept of NICE priorities

Many functions (outside the scope of discussion) are also **abstracted**