Project 2 Initial Design Document

Fang Zhang,
Qiushi Huang,
Zhengyang Song,
Zhaopeng Yan

May 5, 2015

1 File system calls (creat, open, read, write, close, unlink)

1.1 Design

First we need to complete the handleSyscall function based on the given system call id. Based on the arguments every system call needs we declare the corresponding arguments for the new handle functions.

Also note the similarity between *open* and *create*, we just use an extra bool variable to indicate whether it is *create* or *open*.

We use a stack of integers to indicate the unused file descriptors so far, which is initialized to be 2 to 16 (since 0, 1 are reserved for special use) in *load* function.

We also use an array of length 16 (since the number of file descriptors is just 16) to indicate the files opened so far, which is initialized to be 0 as standard input and 1 as standard output as required.

The main idea is as follows: for create, open we use FileSystem.open; for read, write, close we use OpenFile.read, OpenFile.write, OpenFile.close; for unlink we use FileSystem.remove.

1.2 Variables and functions

- int handleOpen(int namePtr, boolean create)
- int handleRead(int fileDescriptor, int bufferPtr, int count)
- int handleWrite(int fileDescriptor, int bufferPtr, int count)
- int handleClose(int fileDescriptor)
- int handleUnlink(int namePtr)
- OpenFile[] openedFiles
- Stack<Integer> unusedFileDesc

Algorithm 1: UserProcess: handleOpen

Input: namePtr, create
Output: fd

```
1  if ununsedFileDesc is empty then
2      return -1
3  get file name from virtual memory using namePtr
4  if name is null then
5      return -1
6  let file to be fileSystem.open parametered by name and create
7  if file is null then
8      return -1
9  pop the top fd from unusedFileDesc
10 mark openedFile[id] to be file
11 return fd
```

Algorithm 2: UserProcess: handleRead

Input: fd, bufferPtr, count
Output: res

```
1  if fd ¡ 0 or fd ¿ 15 or count ¡ 0 then
2      return -1
3  if openedFiles[fd] is null then
4      return -1
5  initialize a buffer of size count
6  read count bytes to buffer from openedFiles[fd]
7  if res == -1 then
8      return -1
9  write the content of buffer to bufferPtr
10 if number of bytes written is less than numebr of bytes read then
11     return -1
12 return res
```

Algorithm 3: UserProcess: handleWrite

Input: fd, bufferPtr, count
Output: res

```
1  if fd ¡ 0 or fd ¿ 15 or count ¡ 0 then
2      return -1
3  if openedFiles[fd] is null then
4      return -1
5  initialize a buffer of size count
6  read count bytes to buffer from bufferPtr
7  if res == -1 then
8      return -1
9  write the content of buffer to openedFiles[fd]
10 if number of bytes written is less than numebr of bytes read then
11     return -1
12 return res
```

Algorithm 4: UserProcess: handleClose

Input: fd
Output: 0

```
1  if fd ¡ 0 or fd ¿ 15 or count ¡ 0 then
2      return -1
3  if openedFiles[fd] is null then
4      return -1
5  close openedFiles[fd]
6  set openedFiles[fd] to be null; push the fd to unusedFileDesc
7  return  0;
```

Algorithm 5: UserProcess: handleUnlink

Input: namePtr
Output: 0 or -1

```
1  get name from vrtual memory using namePtr
2  if name == null then
3      return -1
4  remove name from file system
5  if remove succeeded then
6      return 0
7  else
8      return -1;
```

1.3 Pseudo-code

1.4 Correctness constraints

- No system call can raise an exception in the kernel.
- halt() can only be invoked by 'root' process (for task 3).
- System calls return -1 when error, else return as indicated in syscall.h
- File descriptor 0, 1 initially is for standard input and output, but user process can close these descriptors.
- If open() returns a OpenFile, then it can be accessed by user process.
- Each process can open up to 16 concurrent files, each with a unique file descriptor. A file descriptor can be reused if the associated file is closed.

1.5 Test strategy

- Create a new file, then close it. Open the file, write something into it, then read it out, close it. Delete it at last. Test whether all the system calls are operating normally.
- Try to open a not exist file, see what will happen.
- Open up to 14 files in one process, test whether function well, test whether the 0 and 1 are for standard input and out seperately. Close them one by one, check whether the file descriptors are released timely.
- Try to open up to 15 files(standard input and output not included), see what will happen.
- Make sure that no exceptions are thrown during all above.

2 Multiprogramming

2.1 Design

We allocate pages for a new process and initialize the pageTable of that process when loadSection() is called. When we use virtual address, we first use the pageTable of the current process to translate it into physical address use getPPN(). Therefore, we can use it to implement read and write process.

2.2 Variables and functions

- unusedPPN: public static Stack¡Integer¿ in userkernel, save the unused ppn and it can be allocated to every process.
- unusedPPNLock: lock the unusedPPN.
- UserPoccess::getPPN(int vpn, bool write): input a vpn, use it to get the ppn and return ppn. If write is true but the ppn is readonly then return -1.

- UserProcess::readvirtualMemory():input vpn, translate it into ppn, then load the memory into data using the ppn.
- UserProcess::writevirtualMemory():input vpn, translate it into ppn, then wirte the data into memory using the ppn.
- Userprocess::loadsections():initialize the pagetable. First get the pagesize of the process, then map every vpn into ppn using the unusedppn in userkernel and save it in the pagetable.
- Userprocess::unloadsections():free the ppn. Push it into unusedppn.
- UserKernel::initialize(): add the code to initialize the unusedPPN and unusedPPNLock.

2.3 Pseudo-code

Algorithm 6: UserPoccess::getPPN(vpn, write)

Input: vpn, write
Output: ppn
1 **if** *vpn ¡ 0 —— vpn ¿= numpages* **then**
2 return -1;
3 get entry for pagetable according to vpn;
4 use entry to get ppn;
5 **if** *write and the page is readonly* **then**
6 return -1;
7 return ppn;

2.4 Correctness constraints

- when a process loads successfully, it will get enough pages to use.
- different process will use different phsical address.

2.5 Test strategy

- Run program which use lots of memory to check whether it allocated correctly.
- run multiple user processes to check whether the memory will overlap.

3 System calls for process management (exec, join, exit)

3.1 Design

Since only the first process can call halt(), so we need a int field to indicate the process id(the smaller the earlier). Use numCreated to denote the number of processes created so far to generate an id for the new one.

Algorithm 7: UserPoccess::readvirtualMemory(vaddr,data,offset,length)

Input: $vaddr, data, offset, length$

get memory from machine and convert $vaddr$ into vpn and pageoffset;
ppn = getPPN(vpn,false);
if *ppn ¡ 0* **then**
    return 0;
use ppn and pageoffset to get the address $paddr$;
res = min($length$, pagesize - pageoffset);
copy memory with address paddr into data with offset $offset$ and amount res;
while *doesn't copy length bits memory* **do**
    vpn++;
    ppn = getPPN(vpn,false);
    if *ppn ¡ 0* **then**
        return res;
    use ppn and pageoffset = 0 to get the address $paddr$;
    amount = min($length$ - res, pageSize);
    copy memory with address paddr into data with offset $offset + res$ and $amount$;
    $res = res$ + amount;
return res;

Algorithm 8: UserPoccess::writevirtualMemory(vaddr,data,offset,length)

Input: $vaddr, data, offset, length$

get memory from machine and convert $vaddr$ into vpn and pageoffset;
ppn = getPPN(vpn,false);
if *ppn ¡ 0* **then**
    return 0;
use ppn and pageoffset to get the address $paddr$;
res = min($length$, pagesize - pageoffset);
copy data with offset $offset$ into memory with address paddr and amount res;
while *doesn't copy length bits memory* **do**
    vpn++;
    ppn = getPPN(vpn,false);
    if *ppn ¡ 0* **then**
        return res;
    use ppn and pageoffset = 0 to get the address $paddr$;
    amount = min($length$ - res, pageSize);
    copy data with offset $offset + res$ into memory with address paddr and amount res;
    $res = res$ + amount;
return res;

Algorithm 9: Userprocess::loadsections()

```
1  require lock;
2  if numpages ¿ the number of pages remain in memory then
3      release the lock and close it;
4      return false;
5  for s = 1; s ≤ number of sections ; s++ do
6      get the coff section;
7      for i = 1; s ≤ section's length ; i++ do
8          vpn = section.getfirstvpn() + i;
9          get an unused ppn from userkernel;
10         initialize the pagetable[vpn] with vpn and ppn;
11         loadpage();
12 for i = 1; s ≤ numpages ; i++ do
13     if pagetable[i] is not defined then
14         get an unused ppn from userkernel;
15         initialize the pagetable[i] with i and ppn;
16 release the lock;
17 return true;
```

Algorithm 10: Userkernel::unloadsections()

```
1  require lock;
2  return all the ppn it used to userkernel;
3  release the lock;
```

Algorithm 11: UserKernel::initialize()

```
1  ......;
2  get the phsical page number g from the mechine.
3  push 0 to g into unusedPPN;
```

Use numRunning to denote the number of processes running in order to call terminate timely. In order to change them synchronized, we also dispatch a lock to them each.

Use a new field exitStatus to denote the exit status.

Use a userProcess parent to record the parent of it.

Use Uthread thread to denote the thread the machine actually runs.

Use exeception to record whether the process exits abnormal.

Use a map between id and Process processTable to record the child processes of this, in case join will be called. Also there is a lock associated.

3.2 Variables and functions

- int id
- int exitStatus
- parent
- thread
- exception
- numCreated
- numCreatedlock
- numRunning
- numRunninglock
- processTable
- processTablelock
- int handleExit(int status)
- int handleJoin(int processId, int statusPtr)
- int handleExec(int filePtr, int argc, int argvPtr)
- int handleHalt()
- boolean execute(String name, String[] args)

3.3 Pseudo-code

3.4 Correctness constraints

- No system call can raise an exception in the kernel.
- Only the root process which its id is 0 can call machine.halt().
- A process can only join its child process.
- every process should have an unique process id.
- When a process exits, it should clean up any state associates with it.
- The last process will call terminate().

Algorithm 12: UserProcess: handleHalt

Input: None
Output: 0 or -1

```
1 if the calling process id is not 0 then
2     return -1
3 halt the machine
4 check whether the machine is halted by Lib.assert
5 return 0;
```

Algorithm 13: UserProcess: handleExec

Input: filePtr, argc, argvPtr
Output: child.id or -1

```
1  get file name from virtual memory using filePtr
2  if name is null then
3      return -1
4  new String array of length argc
5  for i=0; ijargc; i++ do
6      new a buff of 4 bytes
7      load ith argument to buff from virtual memory
8      if load failed then
9          return -1
10     load that from buff to args[i]
11     if load failed then
12         return -1
13 create a new process child
14 set this process as the parent of new process
15 call execute(name, args)
16 if call failed then
17     return -1
18 return child.id
```

Algorithm 14: UserProcess: execute

Input: name, args
Output: true or false

```
1 load the name and args into this process
2 if load failed then
3     return false
4 acquire and numRunninglock
5 increase numRunninglock
6 release the numRunninglock
7 create a new thread based on this process
8 fork the thread to run
9 return true;
```

Algorithm 15: UserProcess: handleJoin

```
   Input: processID, statusPtr
1  acquire processTablelock
2  get the child process using processID
3  release processTablelock
4  if child == null or child.parent != this then
5  |  return -1
6  child.thread.join()
7  child.parent = null
8  copy the exitStatus into statusPtr
9  return child.exception ? 0 : 1
```

Algorithm 16: UserProcess: handleExit

```
   Input: status
   Output: 0
1  for all opened file f in opendedFiles do
2  |  if f is not null then
3  |  |  close f
4  release the resources taken up by the process
5  set the exitStatus to be status
6  acquire the numRunninglock
7  decrease the numRunning; if numRunning is 0 then
8  |  terminate the kernel
9  release the numRunninglock
10 finish this process
11 check it is successfully finished by calling Lib.assert
12 return 0;
```

3.5 Test strategy

- run a process to execute processes and join them. Test the execute() and join() and whether the last process will call terminate().
- run a process to join a process which is not its child and check whether it will return -1.
- run several processes and exit them normal or abnormal and check whether them clean up all the associate state.
- run several processes and call halt() to check whether only the root process can call halt().

4 Lottery Scheduler

4.1 Design

Based on original PriorityScheduler, we extends it into LotteryScheduler, with modified pickNextThread() function by using random function to implement lottery, modified Donation() function by using collect all tickets priority.

4.2 Variables and functions

- LotteryScheduler extends PriorityScheduler;
- LotteryQueue extends ThreadQueue;
- ThreadState extends PriorityScheduler.ThreadState;
- LotteryQueue.waitQueue; hashset of <ThreadState, Integer>;(use iterator to pick)
- LotteryQueue.sum; sum of all tickets;
- ThreadState.waitQueue; linkedlist of <LotteryQueue>;(use iterator to donate)

4.3 Pseudo-code

Algorithm 17: LotteryQueue.pickNextThread()

```
1 total← 0;
2 if waitQueue.size= 0 then
3 |   return null;
4 num←random().nextInt(TicketsSum+1);
5 while num > total && waitQueue.hasNext() do
6 |   result←waitQueue.next();
7 |   total+ =waitQueue.get(result).tickets;
8 return result;
```

Algorithm 18: ThreadState.Donation()

```
1  disable interrupt;
2  sum←priority;
3  queueSum← 0;
4  for any owned LotteryQueue lq do
5  |  queueSum = lq.getSum();
6  |  if Integer.MAX_VALUE − sum < queueSum then
7  |  |  return Integer.MAX_VALUE;
8  |  sum+ =queueSum;
9  enable interrupt;
10 return sum;
```

4.4 Correctness constraints

- #Tickets can be very large, and not exceed to MAX_VALUE;
- Effective priority should be sum of all donated tickets plus its own;

4.5 Test strategy

- test whether probability distribution is right;
- test situation sum of tickets exceeding MAX_VALUE;
- test small number scheduler without donation;
- test large number scheduler without donation;
- test small number scheduler with donation;
- test large number scheduler with donation;