Design Doc for Project 2

Instructed by *Xu Wei*

Chen Xiaoqi, Ku Lok Sun, Wu Yijie, Zhang Hanrui

Due on April 22, 2015

TASK I: File System

Description

creat, open, read, write, close, and unlink, documented in syscall.h.

Requirement

```
1. We need to check the users' argument passed to kernel
       1. the users' argument passed to kernel should be legitimate
   2. Only root can invoke halt()
     3. Returns values defined in test/syscall.h
       4. Return -1 on error
  2. If there's exception, the user process should be terminated cleanly
     1. Use UserProcess.readVirtualMemory and UserProcess.writeVirtualMemory;
   string arguments are null-terminated and shorter than 256 bytes
     2. Support 16 fd
       3. Allow closing stdin/stdout
       4. Return error if  ThreadedKernel.fileSystem.open() failed
       5. fd can be reused; sane fd# for different process may mean different file
```

Solution

Constant

1. STDIN = 0
2. STDOUT = 1
3. MAXFD = 16

Class variable

1. fileRecords: hashMap, filename -> (open count, calledUnlink)

FileDescriptor

1. FileName
2. File
3. FileRecord

UserProcess() in UserProcess.java

```
...
fds = new array of FileDescriptor Objects
stdin = UserKernel.console.openForReading()
stdout = UserKernel.console.openForWriting()

# stdin and stdout have no file name and no file record
fds[STDIN] = new FileDescriptor(null, stdin, null)
fds[STDOUT] = new FileDescriptor(null, stdout, null)
...
```

complete handleSyscall(int syscall, int a0, int a1, int a2, int a3) by implementing the following handler:

handleHalt() in UserProcess.java

```
process = UserKernal.currentProcess();
if process is first process (root process)
  Machine.halt()
else
  return -1;
```

handleCreat(a0) in UserProcess.java

```
filename = get string from virtual address a0
# find an unuse file descriptor
fdn = -1
for i in 0..15
  if fds[i] == null
    fdn = i
    break
if fdn == -1
  # No empty slot
  return -1;

record = fileRecords[filename]
if record != null
  if record.calledUnlink
    return -1
```

```
# file name checked by file system
# Create file, clean the file if it exists (set file length to 0)
file = filesystem.open(filename, true)
if file == null
  # filename is invaild or
  # Too much file opened
  return -1

if record = null
  fileRecords[filename] = new FileRecord(1,false)
  record = fileRecords[filename]
else
  record.count += 1

fds[fdn] = new FileDescriptor(filename, file, record)

return fdn;
```

handleOpen(a0) in UserProcess.java

```
filename = get string from virtual address a0

fdn = -1
for i in 0..15
  if fds[i] == null
    fdn = i
    break
if fdn == -1
  # No empty slot
  return -1;

record = fileRecords[filename]
if record != null
  if record.calledUnlink
    return -1

# file name checked by file system
# Create file, clean the file if it exists (set file length to 0)
file = filesystem.open(filename, true)
if file == null
  # filename is invaild or
  # Too much file opened
  return -1

if record = null
  fileRecords[filename] = new FileRecrd(1,false)
  record = fileRecords[filename]
else
  record.count += 1
```

```
fds[fdn] = new FileDescriptor(filename, file, record)

return fdn;
```

handleRead(fileDescriptor, a1, count) in UserProcess.java

```
buffer = get string from virtual address a1

# prevent index out of bound
if fileDescriptor >= MAXFD
  return -1

# Get file
file = fds[fileDescriptor].file

# invalid file descriptor
if f == null
  return -1

len = f.read(p,buffer,count)

return len
```

handleWrite(fileDescriptor, a1, count) in UserProcess.java

```
# prevent index out of bound
if fileDescriptor >= MAXFD
  return -1

# Get file current position
file = fds[fileDescriptor].file

# invalid file descriptor
if f == null
  return -1
# p seems always valid, p is internal state

len = write buffer to vitual address a1

return len
```

handleClose(fileDescriptor) in UserProcess.java

```
# prevent index out of bound
if fileDescriptor >= MAXFD
  return -1
```

```
fd = fds[fileDescriptor]

# invalid fd
if fd == null
  return -1

fd.file.close()
fds[fileDescriptor] = null

record = fd.record
if record.count == 1
  if fd.name != null
    fileRecords[fd.name] = null
    if record.calledUnlink
      if !filesystem.remove(fd.name)
        return -1
else
  record.count -= 1
return 0
```

handleUnlink(a0) in UserProcess.java

```
filename = get string from virtual address a0

record = fileRecords[filename]
if record == null
  if !filesystem.remove(fd.name);
    return -1
else
  record.calledUnlink = true
return 0
```

Test Case

1. disk full
2. no string to read
3. invalid file descriptor, (hard code, > 16)
4. read an write on same file? (use open, not creat to get fd)
5. Cyclic reuse of disk; shall not full if file is unlinked properly. 100000*16 file creat/unlink.
6. Cyclic reuse of fd; 1000*16 open/close fd pair, write should not write to wrong file.
7. Different program opens different files; 1000*16 concurrent opening
8. Write after close should cause exit(); unclosed handle should be closed after exit() or exception.
9. Unlink the file while another process opened and is reading the file.

Questions

1. Do processes share the same stdin and stdout with parent?
2. Can different user process share the same file object in User Kernel?
3. Is unlink blocking call?
4. If process A opened file F and then fork(). B is the children. Does B share the same file object of F? Will there be two file object of F?

TASK II: Multiprogramming

Description

Multiple user processes.

Requirement

1. Allocating the machine's physical memory so that different processes do not overlap in their memory usage.

```
1. Allocate a fixed number of pages for the process's stack (8 pages)
2. a global linked list of free physical pages
3. Use synchronization where necessary when accessing the global linked list:
4. NOT acceptable to only allocate pages in a contiguous block
```

1. Make UserProcess.readVirtualMemory and UserProcess.writeVirtualMemory work with multiple user processes

```
1. Maintain the pageTable for each user process
2.The field TranslationEntry.readOnly should be set to true if the page is
coming from a COFF section which is marked as read-only.
3. always return the number of bytes transferred.
```

1. Modify UserProcess.loadSections() so that it allocates the number of pages that it needs.

    i. Based on the size of the user program
    ii. Set up the pageTable structure for the process so that the process is loaded into the correct physical memory pages.
    iii. exec() should return an error, if the new user process cannot fit into physical memory.

Solution

1. UserKernel class.

    - use static LinkedList to hold free pages, initialize all physical pages as free:

      ```
      initialize(){
        ...
        for(i=0;i<number of pysical pages;++i){
          add i to free page list
        }

      }
      ```

    - implement method `allocateFreePage()` and `releaseFreePage(int)` for UserProcess. The interrupt should be disabled when allocating free pages:

      ```
      allocateFreePage(){
        returnPage = -1
        disable interrupt
        if list of free pages not empty
        returnPage = first page in the list
        restore interrupt
        return returnPage
      }

      releaseFreePage(int pn){
        // may need to check argument
        disable interrupt
        add pn to the list of free pages
        restore interrupt
      }
      ```

2. UserProcess class.

    - use page table (array of TranslationEntry) for each user process

    - read/write at virtual address instead of physical address, use page table to implement `readVirtualMemory` and `writeVirtualMemory`:

      ```
      readVirtualMemory(vaddr){
        get vpn (virtual page number) from vaddr
        entry = pageTable[vpn]
        if entry is invalid, return -1
        get ppn (physical page number) from entry
        paddr = ppn * pageSize + offset of vaddr
      ```

```
  if paddr exceed memory length, return 0
  copy to data buffer starting from paddr
  mark entry as used
  return actual copy length
}

writeVirtualMemory(vaddr){
  get vpn from vaddr
  entry = pageTable[vpn]
  if entry is invalid or read-only, return -1
  get ppn from entry
  paddr = ppn * pageSize + offset of vaddr
  copy from data buffer to memory starting from paddr
  mark entry as used and dirty
  return actual copy length
}
```

- modify `load`, `loadSections` and `unloadSections` for allocating pages:

```
load(){
  ... (calculate the number of pages needed)
  ...
  create new pageTable
  for(int i=0;i<number of pages;++i){
    pageTable[i] = TLE that map i to new allocated page
  }
  ... (load sections)
}
```

```
loadSections(){
  ...
  for any section in program{
    startPos = first virtual page number of section
    for(int i=0;i<section.length;++i){
      entry = pageTable[startPos+i]
      endtry.readOnly = section.readOnly
      load entry.ppn to the section at position i
    }
  }
}
```

```
unloadSections(){
  close program
  release all pages in pageTable
  clear pageTable
}
```

Questions:

1. Are we allowed to disable interrupt? Or just allowed to use semaphore/condition variables?
2. It seems no need to synchronize when release pages since no other process can access that page. Is that right?

Test Case:

1. Stress test the paging system: A=B={a_ij=i+j}, calculate C=A*B, A,B,C in M_N*N, N=2000.
2. Test paging overlap by starting 2 or 4 processes, each writing process ID into memory for 10000 times (with sleeping/yielding), then read out all content and check if any is corrupt.
3. Test paging swap in and swap out by starting 10000 processes, each has one page remembering process ID in memory, and check if memory corrupt. XOR the number to 1 every time to test if the page is saved correctly.
4. Test the maximum size of physical memory allocation not causing an error; check if the limit is stable every time; compare with actual design limit and see if it's consistent.

TASK III: System Calls

Description

exec, join, and exit. (We will also implement `rand()` for user program to test)

Requirement

```
1. Use readVirtualMemory and readVirtualMemoryString to pass data between kernel and u
2. Bulletproof
3. A process should have a global unique ID; allow checking next ID to assign
4. Fork/join
      1. Parent and Children should not directly share memory or fd
      2. Only parent can join children; not grandparent
      3. Parent can join even children exit abnormally
5. Exit
      1. Upon exit, system do housekeeping (cleanup memory, close fd, etc)
      2. Exception also cause exit
      3. The exit code should pass to join() call
      4. Abnormal exit code is <0
      5. The last exit calls Kernel.kernel.terminate() to halt the machine
```

Solution

We mainly modify UserProcess.java. Following modifications are all done in class UserProcess.

Add variables in UserProcess.

1. taskCounter: A counter of total number of process ever runned;
2. taskPool: A hash table that maps process id of each existing task to its UserProcess object;
3. id: Obviously, the process id;
4. parent: Parent process, null if no parent;
5. childrenPool: Stores id's of children of the process;
6. threadPool: Stores threads within the process;
7. joiningPool: Stores threads waiting for current process to join, maps statusAddr to the thread joining to.

```
public static int taskCounter
public static map<int, UserProcess> taskPool

private int id
private UserProcess parent
private set<int> childrenPool
private set<KThread> threadPool
private map<int, KThread> joiningPool
```

In public UserProcess(), add:

```
id = ++taskCounter
parent = null
childrenPool = new set<int>()
threadPool = new set<KThread>()
joiningPool = new map<int, KThread>()
```

We modify function execute such that when a new thread is forked, it is immediately inserted into the thread pool.

In public boolean execute(String name, String[] args), replace:

```
new UThread(this).setName(name).fork()
```

with:

```
UThread thread = new UThread(this).setName(name)
```

```
threadPool.insert((KThread)thread)
thread.fork()
```

The above part of modification will also be used when implementng fork().

To set parent field of a process, we need a function setParent.

Add function:

```
void public setParent(UserProcess process):
  parent = process
```

To handle exit syscall, we

1. Wake up all the threads waiting for the exiting process;
2. Finish all its threads;
3. Free all its resources, including opened files;
4. Remove its id from its parent's children pool and the global task pool;
5. Shut the machine down if it is the last process running.

```
private int handleExit(int status):

  for (addr, thread) in joiningPool:
    writeInt(addr, status)
    thread.ready()

  for thread in threadPool:
    thread.finish()

  unloadSections()
  # Close all file
  for i in 0..15:
    handleClose(i)

  parent.childrenPool.remove(id)
  taskPool.remove(id)

  if taskPool.size() == 0:
    Kernel.kernel.terminate()

  return 0
```

To handle exec syscall, we

1. Create a new process of the correct class and set its parent to be the current process;
2. Read the file name and arguments from memory via translation;
3. Call UserProcess.execute to get it running

4. If succeeded, insert the id of the new process to the children pool of current process;
5. Otherwise return -1 indicating an error.

```
private int handleExec(int fileAddr, int argc, int argv):

  newProcess = newUserProcess()
  newProcess.setParent(this)

  String name = readString(fileAddr)
  String[] args = readParameters(argc, argv)

  int flag = (int)(newProcess.execute(name, args))

  if flag:
    childrenPool.insert(newProcess.id)
    return 0
  else:
    return -1
```

To handle join syscall, we

1. Get the current thread;
2. If the process to join is not a child of current process, declare an illegal syscall, kill current process and return -1;
3. Otherwise, get the process to join, insert the address to place return value and current thread to joining pool of the process to join;
4. Sleep current thread.

```
private int handleJoin(int id, int statusAddr):

  thread = KThread.currentThread

  if id not in childrenPool:
    handleExit(1)
    return -1

  UserProcess joining = taskPool[id]
  joining.joiningPool.insert((statusAddr, thread))
  thread.sleep()

  return 0
```

Test Case

1. Put some statement after exit(0), make sure exit(0) function normally.
2. Joining a children twice.

3. exit(0) twice. Join a children who exit() twice
4. fork() then exit parent; children should run normally.
5. fork and join forming a chain; after 1000 level, the final children exit(), then the greatest parent should exit normally, at last.
6. A children is joined and then caused exception; parent should continue normally.
7. Run multiple processes. Let them call exit(0) in random order and check whether the machine terminates after the last process exits.
8. Basic join cases. Let A legal join happen when there are two or more processes running.
9. Multiple thread cases. Let one or more of the threads of the parent process join the child. Will be done after fork() implemented.
10. Let a thread join a process that is not its child. Check whether it is correctly handled, i.e., the joining process gets killed.
11. Basic exec cases. Call exec on legal executable files and check whether it runs well and whether its parent is correctly set.
12. Call exec on files that do not exist. Check if the return value is correct.
13. Call exec on illegal files that do exist. Check if the return value is correct.

TASK IV: Lottery Scheduler

Description

extends PriorityScheduler; must do priority donation.

Requirement

```
1. Ticket transfer
      1. Waiting thread transfer ticket to threads they waited for
      2. Ticket count is the sum of owning ticket and all waiter's ticket, not maximum
2. Pick a thread which held the lottery
3. Capacity
      1. Ticket number may be large; the actual ticket sum less than Integer.MAX_VALUE
      2. Individual priority in [0,Integer.MAX_VALUE]
      3. Scheduler should be efficient, not O(total ticket) time
4. Implement LotteryScheduler.increasePriority() and decreasePriority() for process
```

solution

New Constant

1. lotteryLimit = Integer.MAX_INTEGER

2. sum of lottery

Override Constant

1. priorityMinimum = 1
2. priorityMaximum = Integer.MAX_INTEGER

Override Method

1. Change all method using PriorityQueue to LotteryQueue

increasePriority()

```
if sum of lottery < lotteryLimit
  return super()
else
  return false
```

setPriority(thread, priority)

```
if sum of lottery > lotteryLimit - priority
  super(thread, lotteryLimit - lottery)
else
  super(thread, priority)
```

pickNextThread()

```
ticket = random(1, sum of lottery) # Include boundary
it = waitQueue.iterator();
next = it.next();
while(ticket > 0){
  thread = it.next();
  ticket -= thread.getThreadState(next).getEffectivePriority();
}
return thread;
```

ThreadQueue

protected class LotteryQueue extends PriorityQueue getEffectivePriority()

```
if (transferPriority == false) {
  return currentPriority;
}
if (dirty) {
    effective = currentPriority;
    for ( it = waitQueue.iterator(); it.hasNext();)
        effective += it.next().getEffectivePriority()
    dirty = false;
}

return effective;
```

protected class ThreadState is same as that one in priority scheduler.

Question

1. Should we set the lottery of the transferred process to zero?

Test Case

```
1. Priority Inversion
2. Big slow low-priority process blocking the resource needed by bursty,
```

higher priority process, causing live lock. The system should eventually give more time slice to slow process. 3. In above case, with priority donation there should not be live lock at all. 4. Stress test: creating 10000 processes, with donation in chain, causing total lottery count to n^2. 5. Accuracy test: create 10000 process half with priority 2 and half with priority 1, each only increment a counter on disk (causing block); calculate statistics after some time, check average and variance of wake-up count of two classes.