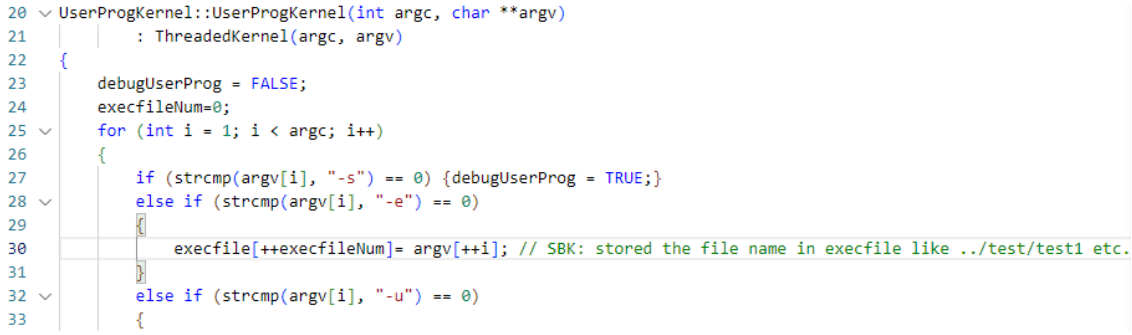
NTU Operating System - Project1

How to parse this project

- You can research userkernel.cc file first. UserProgKernel will parse the command line and store the file you want to execute in execfile variable which defined at userkernel.h.



```
20  UserProgKernel::UserProgKernel(int argc, char **argv)
21          : ThreadedKernel(argc, argv)
22  {
23      debugUserProg = FALSE;
24      execfileNum=0;
25      for (int i = 1; i < argc; i++)
26      {
27          if (strcmp(argv[i], "-s") == 0) {debugUserProg = TRUE;}
28          else if (strcmp(argv[i], "-e") == 0)
29          {
30              execfile[++execfileNum]= argv[++i]; // SBK: stored the file name in execfile like ../test/test1 etc.
31          }
32          else if (strcmp(argv[i], "-u") == 0)
33          {
```

- Then you'll find something interesting.

```
ForkExecute(Thread *t)
{t->space->Execute(t->getName());}
// space and getName() are defined at thread.h
// Execute is defined at addrspace.h
```

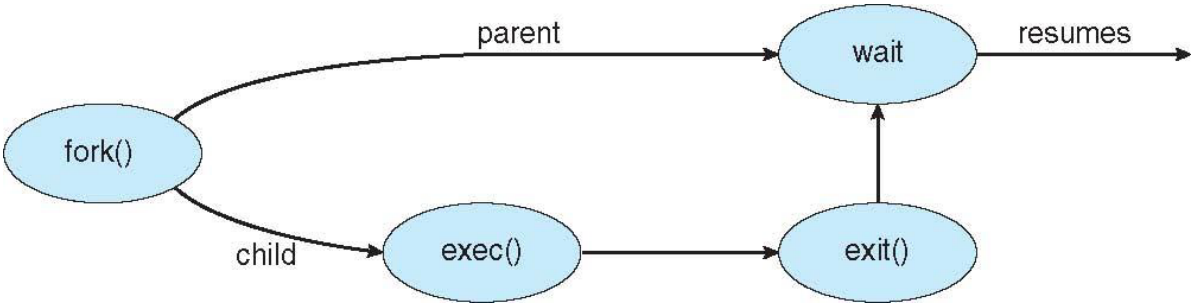
- This is what we've learned on lecture that it'll clone a child process and execute the same program as parent process just like a multi-thread.
- Next, you can observe Run() function

```
void
UserProgKernel::Run()
{
    cout << "Total threads number is " << execfileNum << endl;
    for (int n=1;n<=execfileNum;n++)
    {
        t[n] = new Thread(execfile[n]);
        t[n]->space = new AddrSpace();
        t[n]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[n]);
        cout << "Thread " << execfile[n] << " is executing." << endl;
    }
    ThreadedKernel::Run();
}
```

- Review a very simple concept: Process Representation in Linux

```
// Represented by the C structure task_struct
pid t pid; /* process identifier */
long state; /* state of the process */
unsigned int time slice /* scheduling information */
struct task struct *parent; /* this process's parent */
struct list head children; /* this process's children */
struct files struct *files; /* list of open files */
struct mm struct *mm; /* address space of this process */
```

```
* Another related concept is **Process Creation**
    * Address space
        * Child duplicate of parent
        * Child has a program loaded into it
    * UNIX examples
        * fork() system call creates new process
        * exec() system call used after a fork() to replace the process' memory
space with a new program
```



- After reviewing the concept above, we can continue to answer next question(Please follow the next section -> Q2)

Project Q&A

- Q1: Why the result is not congruent with expected?
    - Because OS has multi-thread concept like the code above and it'll fork child process. When more than 1 process be executed simultaneously without any precondition, it'll happend what we unexpected. And the precondition or you can say the real problem in this case is context switching which is the progress between preocesses switching by CPU.
- Q2: How to solve the issue?(You can include some code and explain it)
    - According to the previous question, NachOS didn't manage memory for executing multi program and this'll cause the current program's page overlap to the others running program.
    - Thus, we have to revise these two program and let the program's virtual memory map to the real memory that no one used.

```
code/userprog/addrspace.cc
code/userprog/addrspace.h
```

1. Add the code in addrspace.cc at the beginning to stored the usage of all physical pages and stored how many free pages can be used now.

```
#define PAGE_OCCU true
#define PAGE_FREE false
bool AddrSpace::PhyPageStatus[NumPhysPages] = {PAGE_FREE};
int AddrSpace::NumFreePages = NumPhysPages;
```

```
class AddrSpace
{
    ...
    private:
        ...
        static bool PhyPageStatus[NumPhysPages]; //stored the usage of all
physical pages
        static int NumFreePages; //stored how many free pages can be used
now
}
```

2. Take off unnecessary mapping

```
AddrSpace::AddrSpace()
{
}
```

3. Allocate Physical Pages - Revise at addrspace.cc AddrSpace::Load function

```
// SBK: Verify if there is enough space page
ASSERT(numPages <= NumPhysPages);       // check we're not trying to run
anything too big -- at least until we have virtual memory
// Allocate
pageTable = new TranslationEntry[numPages];
for(unsigned int i = 0, idx = 0; i < numPages; i++)
{
    pageTable[i].virtualPage = i;
    while(idx < NumPhysPages && AddrSpace::PhyPageStatus[idx] == PAGE_OCCU)
idx++;
    AddrSpace::PhyPageStatus[idx] = PAGE_OCCU;
    AddrSpace::NumFreePages--;
    // Clean the page that'll be used soon
    bzero(&kernel->machine->mainMemory[idx * PageSize], PageSize);
    pageTable[i].physicalPage = idx;
    pageTable[i].valid = true;
    pageTable[i].use = false;
    pageTable[i].dirty = false;
    pageTable[i].readOnly = false;
}
```

4. Change Reading Position - Revise at addrspace.cc AddrSpace::Load function

```
// then, copy in the code and data segments into memory
if (noffH.code.size > 0)
{
    DEBUG(dbgAddr, "Initializing code segment.");
    DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
    executable->ReadAt(&(kernel->machine->mainMemory[pageTable[noffH.code.virtualAddr/PageSize].physicalPage * PageSize + (noffH.code.virtualAddr%PageSize)]), noffH.code.size, noffH.code.inFileAddr);
}
if (noffH.initData.size > 0)
{
    DEBUG(dbgAddr, "Initializing data segment.");
    DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
    executable->ReadAt(&(kernel->machine->mainMemory[pageTable[noffH.initData.virtualAddr/PageSize].physicalPage * PageSize + (noffH.code.virtualAddr%PageSize)]), noffH.initData.size, noffH.initData.inFileAddr);
}
```

When the process is loaded in memory, we must fill in the physicalPage corresponding to pageTable[]. We used linear time to search the first free page and fill it in.
When load it successfully, we can start to execute then we must compute the entry point which is main memory address.
First, computing which pages(i-th page) and times `PageSize` equals **page base**.
Second, **page offset** is `code.address % PageSize`
Finally, `the entry point = page base + page offset`

5. Free Physical Pages

```
// Free the physical page that this program used
for(int i = 0; i < numPages; i++)
{
    AddrSpace::PhyPageStatus[pageTable[i].physicalPage] = PAGE_FREE;
    AddrSpace::NumFreePages++;
}
delete pageTable;
```

- Q3: Experiment result
  *

```
sbk@sbk-virtual-machine:~/NTU/Operating System/nachos-4.0-new/code/userprog$
 ./nachos -e ../test/test1 -e ../test/test2
Total threads number is 2
Thread ../test/test1 is executing.
Thread ../test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:6
return value:0
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 300, idle 8, system 70, user 222
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

- Q4: Discussion
  ■ The most difficaulty that I encountered is previewing most of the background knowledge to complete this project. Most of these knowlege have not been taught yet, so I must be self-learning.