Implement multi-process

A. Solution:

Construct and implement 4 object oriented class below:

1. PCB class:

Process Control Block is a data structure that contains information of the process related to it.

```
class PCB
{
private:
   Semaphore* joinsem;          // semaphore for join process
   Semaphore* exitsem;          // semaphore for exit process
   Semaphore* multex;           // exclusive access semaphore
   int exitcode;
   int numwait;                 // the number of join process

public:
   int parentID;                // The parent process's ID

   PCB(int id);                // Constructor
   ~PCB();                      // Destructor

// Load the program has the name is “filename” and the process id is pid
    int Exec(char* name,int pid);          //Create a thread with the name is
filename and the process id is pid
   int GetID();                 // Return the PID of the current process
   int GetNumWait();            // Return the number of the waiting process
    void JoinWait();                // The parent process wait for the child
process finishes
   void ExitWait();              // The child process finishes
     void JoinRelease();              // The child process notice the parent
process
    void ExitRelease();            // The parent process accept to exit the
child process
   void IncNumWait();           // Increase the number of the waiting process
   void DecNumWait();           // Decrease the number of the waiting process
   void SetExitCode(int);       // Set the exit code for the process
   int GetExitCode();           // Return the exitcode
   void SetFileName(char*);     // Set the process name
   char* GetFileName();         // Return the process name
};
```

2. Ptable class:

Used to manage running processes, includes the PCB property which is a process table, has a one-dimensional array structure with a maximum number of 10 elements, each element of the array is of data type PCB.
The constructor function of the class will initialize the parent process (which is the first process) at the 0th position equivalent to the first element of the array. From this process, we will create child processes through the system call Exec().

```
#define MAX_PROCESS 10

class PTable
{
private:
   int psize;
    BitMap *bm;                    // mark the locations that have been used
in pcb
   PCB* pcb[MAX_PROCESS];
    Semaphore* bmsem;               // used to prevent the case of loading 2
processes at the same time

public:
//Constructor initialize the size of the PCB object to store the process
size. Set the initial value to null
    PTable(int size);
    ~PTable();                    // Destructor

   int ExecUpdate(char*);       // Process the syscall SC_Exec
   int ExitUpdate(int);         // Process the syscall SC_Exit
   int JoinUpdate(int);         // Process the syscall SC_Join

    int GetFreeSlot();            // Find the free slot in PTable to save
the new process information
   bool IsExist(int pid);       // Check a process exist or not

   void Remove(int pid);        // Delete the PID from the PTable
   char* GetFileName(int id);   // Return the process name
};
```

3. Sem Class:

Used to manage semaphore: attributes and functions

```
class Sem
{
private:
   char name[50];       // The semaphore name
   Semaphore* sem;     // Create semaphore for management
public:
   // Initial the Sem object, the started value is null
   // Remember to initial the Sem to use
   Sem(char* na, int i)
   {
       strcpy(this->name, na);
       sem = new Semaphore(this->name, i);
   }

   ~Sem()         // Destruct the Sem object
   {
       if(sem)
           delete sem;
   }

   void wait()
   {
       sem->P();   // Conduct the waiting function
   }

   void signal()
   {
       sem->V();   // Release semaphore
   }

   char* GetName() // Return the semaphore name
   {
       return this->name;
   }
};
```

4. STable class:

This is a table describing the semaphore, with a one-dimensional array structure with a maximum number of 10 elements, each element of the array of data type Sem

```
#define MAX_PROCESS 10
class STable
{
private:
  BitMap* bm; // Manage the free slot
  Sem* semTab[MAX_SEMAPHORE];
public:
  // Initial the Sem object, the started value is null
  // Remember to initial the bm object to use
  STable();

  ~STable();
  // Check the semaphore name, create a new one if not already
  int Create(char *name, int init);
  // If the semaphore name already exists, call this->P() to execute it.
If not, report an error in Wait, Signal function
  int Wait(char *name);
  int FindFreeSlot(int id);  // Find an empty slot
};
```

B. Add classes to nachos:

Declare these global variables in **`~code/threads/kernel.h`**

```
Semaphore *addrLock;
Bitmap *gPhysPageBitMap;
STable *semTab;
PTable *pTab;
```

Add these class into the file **~code/thread/kernel.h**

```
class SynchConsoleInput;
class SynchConsoleOutput;
class SynchDisk;
class Semaphore;
```

Include these *.h file into the **~code/thread/kernel.h**

```
#include "bitmap.h"
#include "stable.h"
#include "ptable.h"
```

Update the **~build.linux/Makefile**
For new .h, .cc, .o file simply update the appropriate "_H", "_C", "_O" list below
For example, if you create a class Stable which is declared and implemented in file called "stable.h", stable.cc in the thread subdirectory, you should add:

```
"../thread/stable.h" to THREAD_H
"../thread/stable.cc" to THREAD_C
"../thread/stable.o" to THREAD_O
```

C. Syscall implementation:

1. SC_Exec

Exec system calls use the PCB and Ptable layers to invoke the execution of a new program in a new system thread. Before implement the system call Exec we need to implement the methods:

Exec(char* name, int pid) at the PCB class:

- Call **mutex->P()** to help avoid loading 2 processes at the same time.
- Check if the thread has initialized successfully, if not, not enough memory error messages, call **mutex->V()** and return.
- Set processID of this thread is id.
- Set the parentID of this thread to be the processID of the thread calling Exec
- Call **Fork(StartProcess_2,id)** => We cast the thread to type int, then when processing the StartProcess function we cast the thread to its correct type.
- Returns id.

ExecUpdate(char* name) at the PTable class:

- Call **mutex->P()** to avoid loading 2 processes at the same time.
- Check the validity of the program "name".
- Check the existence of the program "name" by calling the Open method of FileSystem class.
- Compare program name and currentThread name to make sure this program is not called Exec itself.
- Find the empty position in the Ptable table. - If there is an empty slot, create a new PCB with processID being the index of this slot, parentID is processID of currentThread. Mark used.
- Call the method Exec of PCB class
- Call **msem->V()**.
- Return the test results of PCB->Exec.

Exec() syscall:

- Declare the function **int Exec(SpaceID id)**
- Read the address of the program name “name” from register r4. The program name is now in the user space.
- Move the program name from user space to system space:
    - If there is an error, it will report "Cannot open the file" and assign -1 to register r2.
    - If there is no error, call **pTab->ExecUpdate(name)**, return and save the result of executing this method into register r2

2. SC_Join

The syscall Join uses PCB class and Ptable class to perform wait and block based on "**SpaceID id**" parameter. Before implement this system call, we must implement the following methods:

JoinWait() ở lớp PCB:

- Call **joinsem->P()** to switch to block state and stop, and wait for JoinRelease to continue

ExitRelease() ở lớp PCB:

- Call **exitsem->V()** to release the waiting process

JoinUpdate(int id) ở lớp PTable:

- Check the validity of the processID id and check if the process calling Join is the parent of the processID = id:
  - If not, report a reasonable error and return -1. Increment numwait and call JoinWait() to wait for the child process to execute.
- After the child process finished executing, the process was released.
- Handle exitcode.
- Call ExitRelease() to allow the child process to exit

Join() syscall:

- Declare the function **int Join(SpaceID id)**
- Read the id of the process which wants to join from the register r4.
- Call **pTab->JoinUpdate(id)** and save the result in register r2

3. SC_Exit

The syscall Exit uses PCB class and Ptable class to exit the process which it joined. Before implement this system call, we must implement the following methods

JoinRelease() ở lớp PCB:

- Call **joinsem->V()** to release the waiting process, which called JoinWait()

ExitWait() ở lớp PCB:

- Call **exitsem->V()** to switch to block state and stop, and wait for ExitRelease to continue

ExitUpdate(int exitcode) ở lớp PTable:

- Check the validity of the processID id and check if the process calling Join is the parent of the processID = id:
  - If not, report a reasonable error and return -1. Increment numwait and call JoinWait() to wait for the child process to execute.

- After the child process finished executing, the process was released.
- Handle exitcode.
- Call ExitRelease() to allow the child process to exit

Exit() syscall:

- Read the exitStatus from the register r2
- Call **pTab-> ExitUpdate(exitStatus)** and save the result in the register r2

4. SC_CreateSemaphore

Used to create a new semaphore

- Declare the function **int CreateSemaphore(char* name, int semval)**
- Read the “name” address from the register r4
- Read the “semval” value from the register r5
- Transfer the “name” from the user space to system space
- Call **semTab-> Create(name, semval)** to create semaphore. If false, given an error
- Save the result in the register r2

5. SC_Up

The syscall SC_Up uses the class Stable to release the waiting process

- Declare the function **int Up(char* name)**
- Read the “name” address from the register r4
- Transfer the “name” from the user space to system space
- Check if this "name" Semaphore is in the sTab table, if not, given an error
- Call the method **Signal()** of the Stable class
- Save the result in the register r2

6. SC_Down

The syscall SC_Up uses the class Stable to perform the wait operation

- Declare the function **int Down(char* name)**
- Read the “name” address from the register r4
- Transfer the “name” from the user space to system space
- Check if this "name" Semaphore is in the sTab table, if not, given an error
- Call the method **Wait()** of the Stable class
- Save the result in the register r2

D. Test program implementation:

Use the above system calls to implement shell programs, which uses to receive an instruction at a time and execute the corresponding program

- Run a loop with the condition that it stops when the user input the command “exit”
- Ask the user to input the program name

- Check if the program exists on the machine, if not, ask for re-input

Note: You can add extra variable/ method/syscall to serve your solution