Assignment 1

Satvik Chauhan (Y9521) , Pankaj More (Y9402)

- Changes in Nachos
  - Changes in Address Space
    - Nachos was a single process system , so it used the whole of its memory for the address space of a single process .
    - Every address space consist of pagetable which is made up of pages .
    - The first page of the address space starts with virtual address 0 ,but it has some mapping to the physical address on the main memory.
    - In the original nachos both virtual address and the physical address were same (equals 0) as it had to deal with only one process .
    - The class Bitmap from Bitmap.h was used to create a bitmap for the pages in the physical memory and to find the empty physical address while creating a new address space.
    - Initially after creating the new address space , the whole physical memory was wiped out to 0 . So we had to change it so that it only assigns 0 to its own address space bits.
    - We had to translate the virtual address to the corresponding physical address while copying any file to the address space.
    - To handle the specific need of the Fork system call in which we had to copy the current address space to a newly created address space we used *constructor overloading* to create a new address space constructor deep copies the address space from the address space pointer passed to it.
    - We had to move the pagetable creation step from the original constructor to the Load function . So it creates new address space when Load function is called and assigns required amount of pages for it .
    - The Address space destructor has to take care to free up the memory and mark the corresponding pages in the pagetable bitmap as dirty so that the space can be reused.
    - We also created a class element called id which assigns a unique id to each process based on the starting page of its address space.
  - System calls
    - *Fork*
      - Fork creates a new kernel level thread

  - Copies the current address space to the address space of the new thread .
  - Also keep the registers to be used by the child process .
  - Puts child on the ready queue and starts the child process.
  - It returns 0 in the child process and the *PID* of the child in the parent process.
- There are two implementations of the exec system call described below
  - *Exec2*
  - To be used with Fork system call only .
  - It replaces the current address space with the program to be loaded .
  - Resets all the registers and thus moves the program counter to the beginning .
  - The control returns to the parent process after the child process finishes.
  - *Exec*
  - An amalgamation of the above Exec2 and Fork calls .
  - Instead of calling Fork and then Exec2 , We can directly call Exec with the executable of the new process and it directly loads and starts the new process.
  - The control returns to the parent process after the child process finishes .
- *Exit*
  - Exit system call is called automatically whenever a program exits .
  - Handles the cleanup of the system .
  - It stops the thread if it is not the main thread .
  - Since the last thread can not stop itself , so we can not just finish the last *main* thread .
  - So if the pid of the thread calling main is 0 (0 means it is the main thread) it calls nachos Halt instead of finishing the thread.
- SysStats
  - SysStats internally calls SysStatsCall() function defined in exception.cc .
  - The structures ProcInfo and SysInfo as given in the question have been defined in machine/stats.h with a few modifications (for optimisation and convienince) which are only internal to the code. The external api for calling it is consistent with the question.

- A global mysysinfo structure is created as a member of the kernel object and it can be accessed as kernel->mysysinfo .
- Since the tick variables are already defined and are working in stats.h , our corresponding ticks in SysInfo structure are just pointers to those pre-defined ticks. When the kernel is initialised , and kernel->stats is created , we setup the corresponding pointer references in Kernel::Initialize() .
- We create the array of ProcInfos during Thread creation. When Thread::Thread() constructor is called, we simply create an new ProcInfo structure and add it to the proc array in mysysinfo and setup the corresponding pointer references for thread name and thread status.
- kernel->mysysinfo->proc refers to an array of ProcInfo structures which store the status of each procces name and status.
- Our Nachos user processes are nothing but Thread objects defined in /threads/threads.h .
- When a process is just created , its status is JUST_CREATED.
- When the Nachos System is running the process , it sets its status to RUNNING.
- When the process is Yield , its status becomes READY.
- When the process is finished/killed , it becomes BLOCKED.
- When SysStats system call is made , it prints all the processes (both running and killed) along with their status. Moreover , it also prints out the relevant ticks details on the console.

– Assignment1.c

  * Assignment1.c uses Exec2 system call which comfirms to the spec given in the question.
  * It basically calls fork and execute on each programs in the progs array.

– otherExec.c

  * It uses the Exec system call. It is different from the one asked in question.
  * It does a thread fork and then loads the program simultaneously on the same Exec system call.