NACHOS

Sannara Ek
Juan Vazquez
Shakeel Ahmad Sheikh

MoSIG M1 - GROUP E

Project Management

- For steps 2, 3 and 4, the basic parts were done concurrently
- Shakeel: Additional parts of Step2, Thread Join, Process Exit
- Sannara: User Semaphores, Thread Join,
- Juan: Files systems, Process Join

Outline

- ★ Multithreading
  - Thread Creation
  - Stack Management
  - Thread Join
  - Thread Exit
  - User Semaphore
- ★ Multiprocessing
  - Process Creation (ForkExec)
  - Exit of process
  - Waiting on Processes (UserWaitPID)
- ★ Filesystem
  - Directory Tree
  - Open Files Table and concurrent access
  - Increasing the file size

Thread Creation

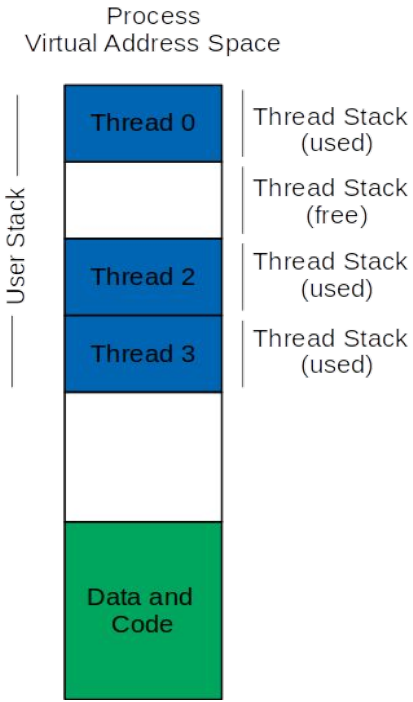
- Thread id is unique across its address space
- Thread id is not being used even after the termination of a thread.
- One to One mapping between NachOS user level threads and kernel level threads.
- Inside a process threads share the same address space but has its own registers and stack.

Supporting Structure:

- A counter, *tidcounter*,
- A counter, *livethreads*,
- A bit map, *stackFrames*
- An array of semaphores, called *createdThreads* used for ThreadJoin

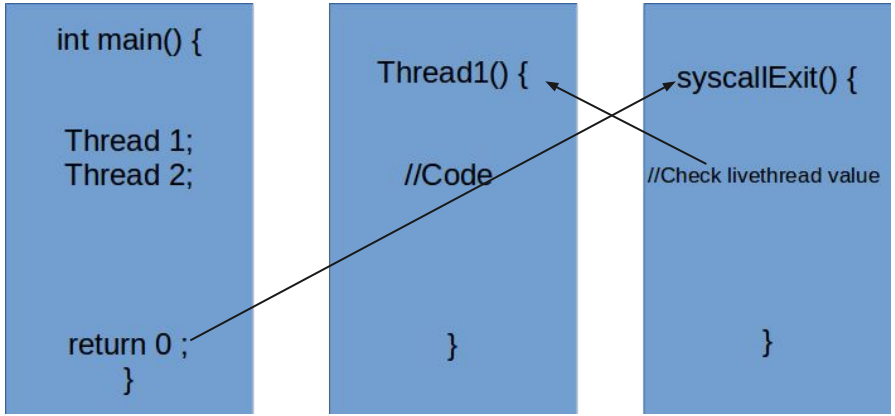
Stack Management

- *UserStackSize*
- User Stack is divided into frames
- bitmap *stackFrames*, returns an integer value the frame number



Thread Exit

- What happens to child threads if the main thread returns i.e calls syscallExit() ?
- Threads won't have a chance to complete in this case.
- Synchronised handler for syscallExit()
- *Livethreads* Counter



Thread Join

- *UserThreadJoin* System call
- *createdThreads:* Array of semaphores

```
thread1(void* a)
{
  ...
  UserThreadExit();
}
int main()
{
  ...
  tid1=ThreadCreate(thread1);
  UserThreadJoin(tid1);
  ...
```

tid1 = 3

Case 1: Thread 1 finishes before main executes UserThreadJoin

Initial status of *creathedThreads*

2	3	4
0	0	0

Thread1 with tid = 3 finishes:
creathedThreads[3]->V()

2	3	4
0	1	0

Main thread reaches Join:
creathedThreads[3]->P()
Main thread continues

2	3	4
0	0	0

Case 2: Main executes UserThreadJoin before thread 1 finishes

Initial status of *creathedThreads*

2	3	4
0	0	0

← *CreatedThreads Array*

Main thread reaches Join:
creathedThreads[3]→P()
Main thread waits

2	3	4
0	0	0

← *CreatedThreads Array*

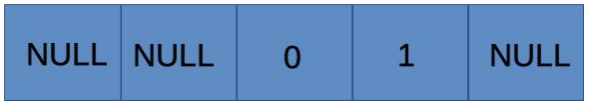
Thread1 with tid = 3 finishes:
creathedThreads[3]->V()
Main thread will continue

2	3	4
0	1	0

← *CreatedThreads Array*

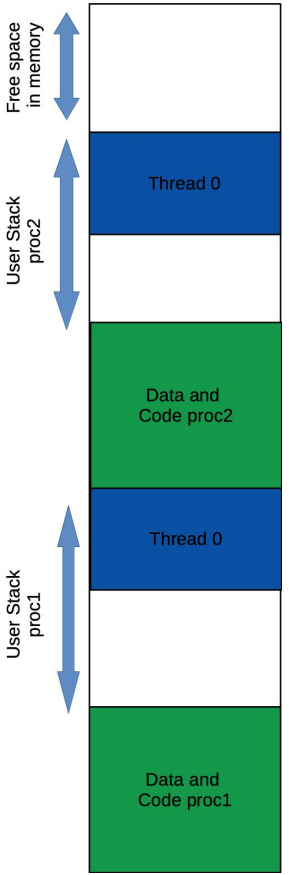
User Semaphores

- A new **sem_t** data type that is based on ints which are used to represent semaphoresID
- These IDs are used as the index in an array.
- Functionalies & Implementation method
    - SemInit(sem_t *arg1, int arg2): *UserSemaphores[semID]* with the value from arg2
    - SemP(sem_t* arg): Calls the kernels wait on *UserSemaphores[semID]* on the index obtained from arg.
    - SemV(sem_t* arg): Call the kernels post on *UserSemaphores[semID]* on the index obtained from arg.



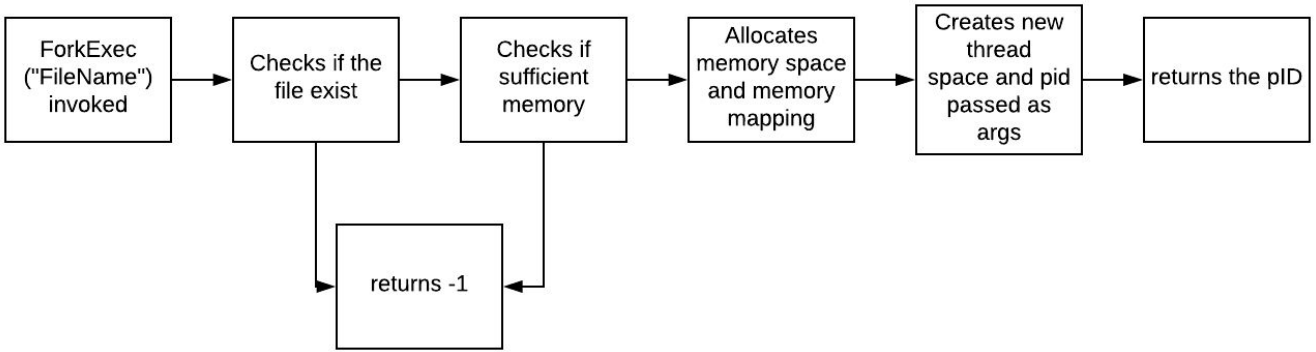
Fork Exec (Multiprocessing)

- Process are created through the system call ForkExec()
- Supporting Structure:
  - A counter, *procounter*,
  - A counter, *livepro*,
  - A bit map, *frameProvider*
  - A identifier, *pro*



Fork Exec

1. Implementations:
    a. Retrieves the file name passed obtained from the argument and attempts to open it with the function fileSystem->Open(Filename)
    b. Checks if the the file else was executable
    c. Checks if there is valid free memory through a function called CheckPhysicalSpace
    d. An object called *space*, data typed addrSpace, will be declared for the new process
    e. The construction of addrSpace for the new process maps frames in the physical memory to the page table and declares them as used through function *GetEmptyFrame*
    f. The new thread is then created with fork, given, a pointer to *space* and to the value of *procounter as an argument*.



Managing processes exits

- Extended system call *Exit*
- A counter, *livepro*, is incremented and decremented every time a process starts and finishes
- *livepro* is checked If its value is at 0.
- If false, the ending process only releases its resources and memories
- No Halt is called

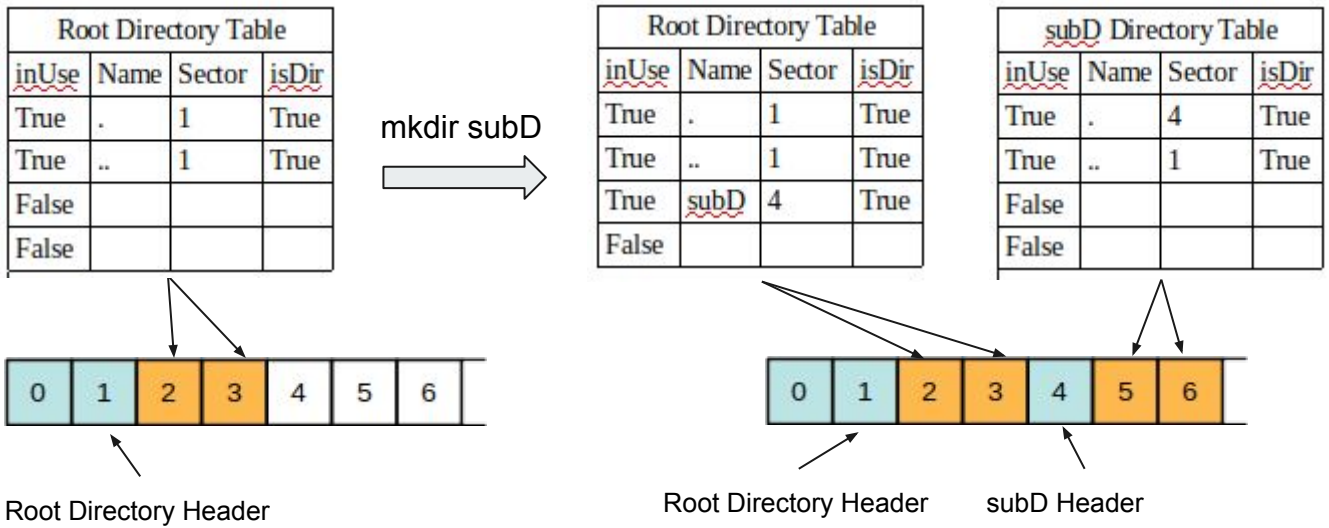
UserWaitpID

- Mechanism user is similar to *userThreadJoin* which was spoken earlier
- An array of semaphores is used with process ID as indexes.
- *UserWaitPid(pid)* will invoke a semaphore wait at the index of pid in the semaphore array, which causes the the calling process to wait until the desired process has finishes its work and call a semaphore post.

File System

- Directory tree
- Multiple open files
- Concurrent access to files
  - A file can be opened multiple times inside a process
  - A file can not be open in mutiple processes at the same time
- Maximun supported file size: 119,75KB

File System: Directory Tree



Executing *cd subD* from root directory:

- The global variable *currentSector* is set to 4;
- The global variable *directoryFile* point to subD file

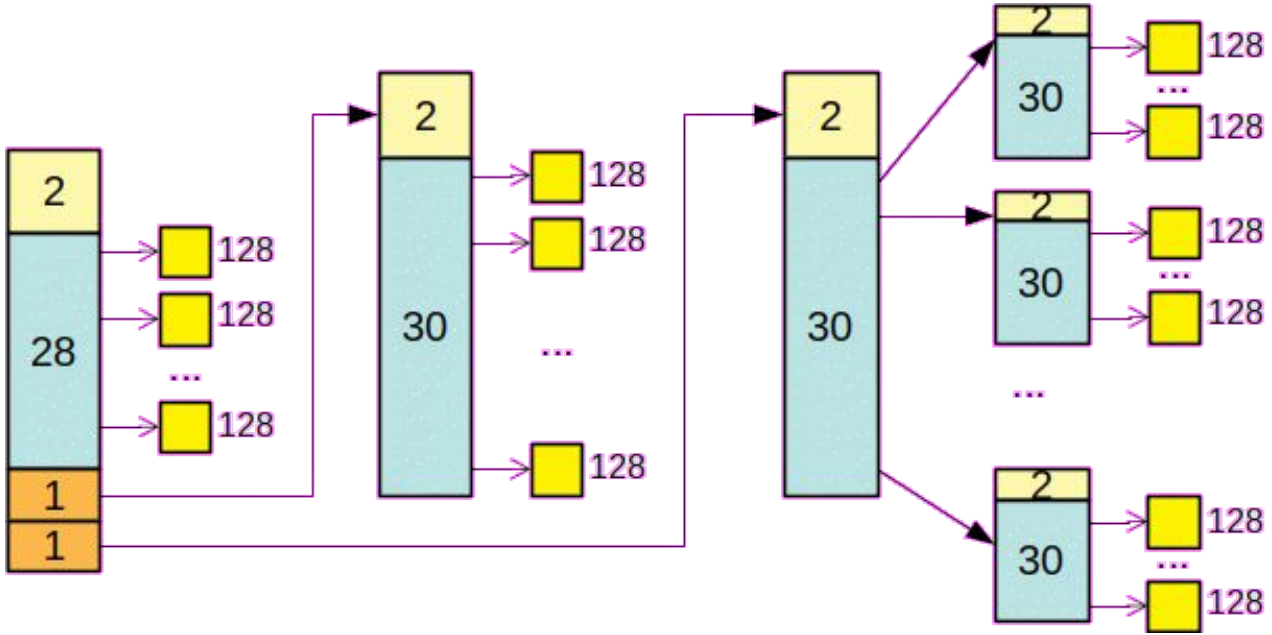
Open Files Table and Concurrent Access to Files

- Two different processes cannot open the same file
- A file can be opened by different threads running in the same process.
- Two open files tables are implemented: one sistem-wide table, and one for each process
- Each open() creates a new entry in the per-process open files table.
    - It may create an entry in the system-wide open files table
- close() frees the corresponding entry in the per-process open files table.
    - It may free the corresponding entry in the system-wide open files table.

Per Process Open Files Table			
	inUse	OpenFile*	name
0			
1			
2			

System-wide Open Files Table		
inUse	name	pid

Increasing the Maximum File Size



Max Size = $(28+30+30^2) \times 128 = 119{,}75KB$

Conclusion

1. Working on this project made us really understand how an OS fundamentally works.
2. Our project has some limitations, but we had a tight time schedule. We also have some ideas how to fix them.
3. Git was a good tool to learn and explore.

Any Questions?