OPERATING SYSTEM DESIGN WITH NACHOS

Eslam Mohammed, Mohammed AlMarakby, Sofiane Tanji, Arthur Chardon

grenoble inp-ensimag

31-1-2020

Agenda

- Introduction
- Organization & workload
- Stages Implementation
- Conclusion

Introduction

Our NachOS version supports:

- Various Syscalls.
- User level multi-threading support with synchronization.
- Ability to launch multiple processes.
- A file system that provides multiple file and directory operations.
- Basic networking features.

Organization & workload

- We utilized Git and Gitlab for the sharing and collaboration of code.
- We divided the work for everyone to work on a part he is comfortable with.
- We “tried” to use pair programming.

Organization & workload

Work distribution:

- Step 1: All the Team members.
- Step 2: Sofiane and Eslam.
- Step 3: Mohammed and Eslam.
- Step 4: Mohammed and Eslam.
- Step 5: Sofiane.
- Step 6: Mohammed and Eslam.

Stages Implementation

Stage 2 - Synchronous console and a few syscalls

During this stage:

- Thread safe **SynchConsole**
- Extended the **PutChar** to be employed in **PutString**.
- At Stage 4, we updated the **CopyStringFromMachine()** to read from virtual memory instead of the main one.

Stage 3

Main design principles:

- 1:1 model with kernel threads.
  - pass to the kernel threads routine to be invoked, arguments and exit routine.
- No thread Id re-use.
  - The Id variable is restarted after large number of threads creation to prevent overflow.
- A data structure to track the list of working threads. Facilitates further functionalities like thread join and thread exit.

Stage 3

The Thread in the Address space:

- A bitmap instance to handle the thread positions in the user stack.
- Shift down the bitmap pointer with 8 bytes and allocate two pages per thread.
- Handling large number of threads, 2 options: block the execution Vs. slow down the execution.

Stage 3

Synchronization:

- Thread join implementation: Attach a semaphore acting as a barrier with every thread, only post on it in the thread finish routine.
- Users-level Semaphores: Allocate a defined number of semaphores per address space, and transmit the usual P and V via syscalls to the user program space.
- For protection we block the releasing of a semaphore if it is active “being posted or waited on”

Stage 3

Synchronization *conti.*

- For automatic termination, we pass the exit thread handler to the return address register, so that it is executed when the thread finishes it’s routine.
- All the data structure inside the thread are protected with locks to ensure safety.

To be continued:

Stack protection: Prevent the threads from writing in each others stack memory.

Stage 4

ForkExec() Creates a thread that creates an address space!

- This new address space loads the executable and start the program via **Run()**.
- and the address space has to call **ProcessTerminate()**.
- We do not reuse the same PID, however a new one is assigned.

Stage 4

ProcessTerminate() exits from an address space and deletes it!

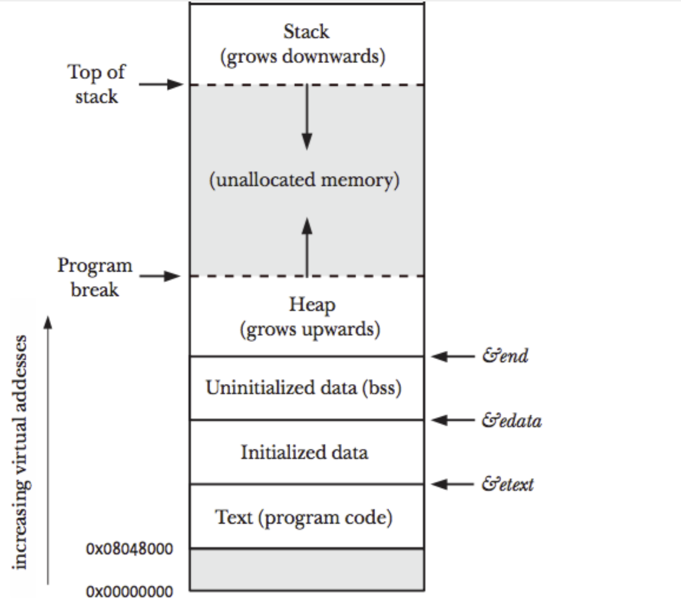
- If there were any user threads inside, the main thread waits.
- We call **Halt()** when it is the last process.
- otherwise, we let the user thread finish.

Stage 4

Brk and Sbrk

- Allocate extra entries in the address space page table, but do not allocate counter-part physical pages and set valid bit to *false* .
- On a call to Sbrk, allocate empty frames and map them to the page table entries point to by brk.
- Update the brk to point to the program break.
- Limit the amount of heap extension per address space.

Stage 4


Stack
(grows downwards)
Top of stack
(unallocated memory)
Program break
Heap
(grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
increasing virtual addesses
0x08048000
0x00000000

Stage 4

To be continued:

- Automatic termination for **ProcessTerminate()**.
  - Return address of the driving kernel thread .
  - Update the stub.
- Nested level hierarchical termination:
  - Implementing a Guardian/Protector relationship between parent/children processes.

Stage 5 - File system implementation

- OS comes with a directory hierarchy.
  - ChangeDirectory, MakeDirectory, RemoveDirectory at the user-level.
- Open multiple files.
  - up to 10 !
- Safely* !

  *almost ...
- Fixed-size files

Stage 5 - Directory hierarchy

- Distinguish directories and files.
- What to do if it’s a directory ?
- . and .. : a necessary step
- ChangeDirectory, MakeDirectory, RemoveDirectory

Stage 5 - Open multiple files

- Store opened files somewhere ...
- Tables of two types !
  - SystemTable - Chained list of opened files
  - Table - Chained list of opened files PER thread

Stage 5 - Issues and improvement

- Path names implementation
- Addressing concurrency issues

Stage 6

A ring topology is implemented based on the routing logic of **farAddr** of the receiver, so we know when to send and when to wait.

Stage 6

- Reliable transfer?
- How could we inform NachOS to resend the message again in case if failure?
- How could we keep the sender thread to make sure the message arrived safely?

Stage 6

Our approach:

- Schedule an interrupt that executes a Time out handler after **TEMPO**.

Stage 6

Our approach:

- Schedule an interrupt that executes a Time out handler after **TEMPO**.
- A synched list **History** that track all the sent messages.

Stage 6

Our approach:

- Schedule an interrupt that executes a Time out handler after **TEMPO**.
- A synched list **History** that track all the sent messages.
- **reliableSend()** send and receive(on mailbox1) acknowledgment at the same time.

Stage 6

Our approach:

- Schedule an interrupt that executes a Time out handler after **TEMPO**.
- A synched list **History** that track all the sent messages.
- **reliableSend()** sends and receives(on mailbox1) acknowledgment at the same time.
- **reliableReceive()** sends an acknowledgment on receiving a message.

Stage 6

Our approach:

- Schedule an interrupt that executes a Time out handler after **TEMPO**.
- A synched list **History** that track all the sent messages.
- **reliableSend()** sends and receives(on mailbox1) acknowledgment at the same time.
- **reliableReceive()** sends an acknowledgment on receiving a message.
- **numTrials** attached in every mail to know how many times this mail is sent.

Stage 6

Problems?

- After the sender thread sends a message it waits on the semaphore when it performs **GET**

We went through hardcore debugging but could not figure it out for time limitations.

Stage 6

Regarding variable sized messages, we have the following approach:

- A **carry** variable that hold the current size(in bytes) of the mail.
- buffer on the receiver side to concatenate all the chunks together.

Thank You!