NachOS Subproject 3 : Multithreading

Goal

- During step 2 NachOS launches a single user program with a single execution flow
  - one process
  - one thread
- During subproject 3 we will continue having one single process but will implement multiple threads

Example of a POSIX multithreaded program

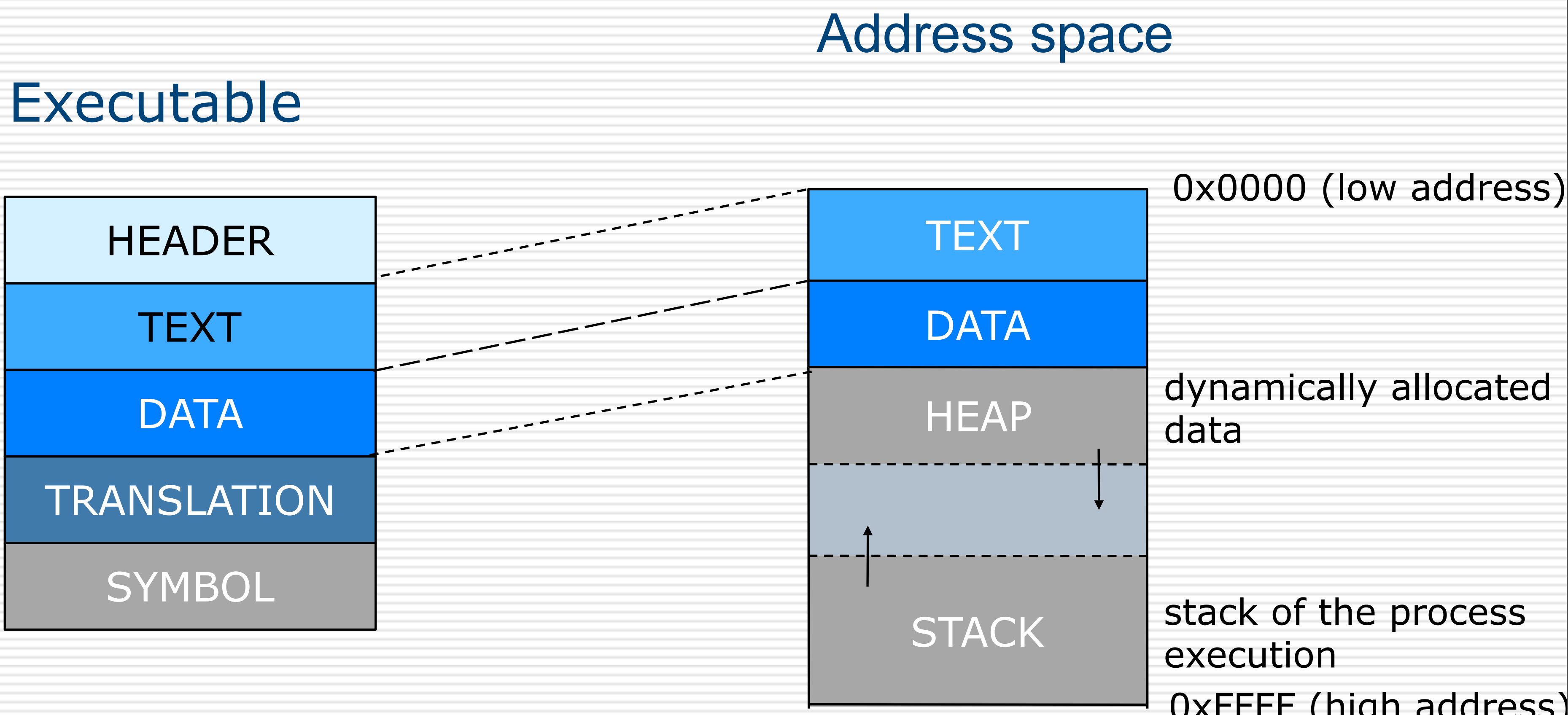
Example Code - Pthread Creation and Termination

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Processes vs threads

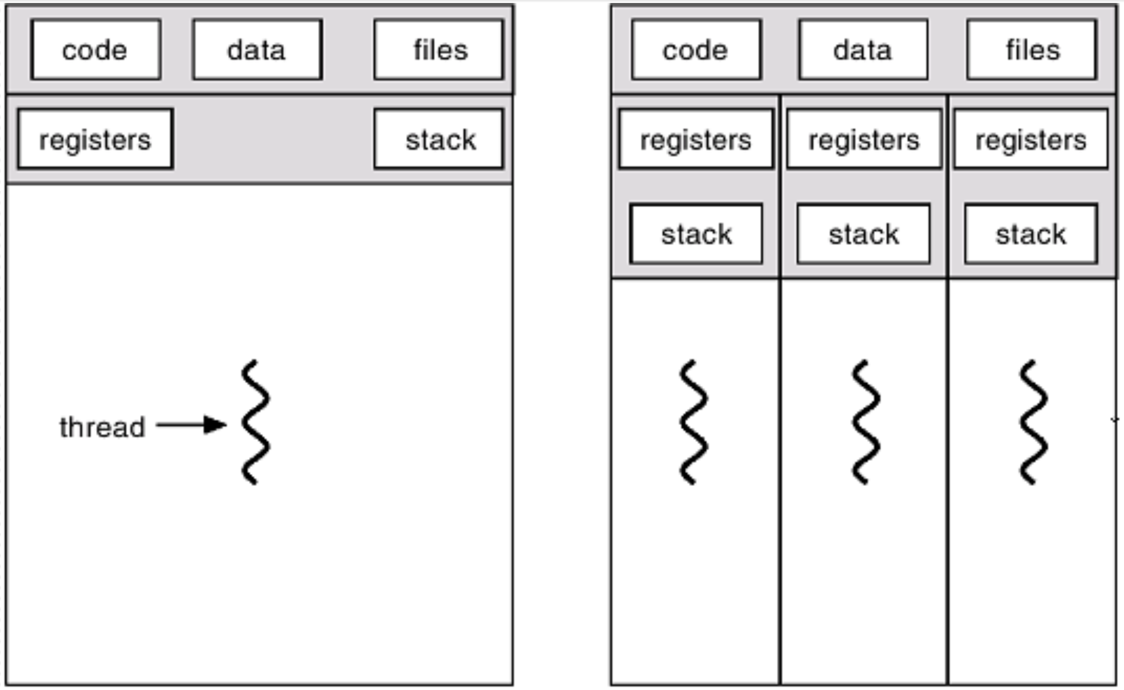
- Do you remember what a process is?
  - What is its structure in memory?
  - What information is managed?
  - What about the cost of process-related functions?
- Do you remember what a thread is?
  - What is the motivation behind threads?
  - What is specific to threads and what is shared?

Process Structure



What about threads' address space?

- Threads share the text zone(code), the data zone, the system resources allocated to the process
- They manage their own registers and stack



single-threaded multithreaded

The `AddressSpace` Class (userprog)

```
#ifndef ADDRSPACE_H
#define ADDRSPACE_H

#include "copyright.h"
#include "filesys.h"

#define UserStackSize       1024    // increase this as necessary!

class AddrSpace
{
  public:
    AddrSpace (OpenFile * executable);  // Create an address space,
    // initializing it with the program
    // stored in the file "executable"
    ~AddrSpace ();      // De-allocate an address space

    void InitRegisters ();  // Initialize user-level CPU registers,
    // before jumping to user code

    void SaveState ();      // Save/restore address space-specific
    void RestoreState ();   // info on a context switch

  private:
      TranslationEntry * pageTable; // Assume linear page table translation
    // for now!
    unsigned int numPages;  // Number of pages in the virtual
    // address space
};

#endif // ADDRSPACE_H
```

The `AddressSpace` Class (2)

```
AddrSpace::AddrSpace (OpenFile * executable) {
    ...
    executable->ReadAt (...); ...
    // how big is address space?
    size = noffH.code.size + noffH.initData.size +
    noffH.uninitData.size + UserStackSize; // we need to increase the
                                           // size to leave room for the stack
    numPages = divRoundUp (size, PageSize);
    size = numPages * PageSize;

    ASSERT (numPages <= NumPhysPages); // check we're not trying
                                       // to run anything too big --
    ...
    // then, copy in the code and data segments into memory
    if (noffH.code.size > 0){
     ...
    }
    if (noffH.initData.size > 0) {
     ...
    }
}
```

The AddressSpace Class (3)

```
void
AddrSpace::InitRegisters ()
{
    int i;

    for (i = 0; i < NumTotalRegs; i++)
        machine->WriteRegister (i, 0);

    // Initial program counter -- must be location of "Start"
    machine->WriteRegister (PCReg, 0);

    // Need to also tell MIPS where next instruction is, because
    // of branch delay possibility
    machine->WriteRegister (NextPCReg, 4);

    // Set the stack register to the end of the address space, where we
    // allocated the stack; but subtract off a bit, to make sure we don't
    // accidentally reference off the end!
    machine->WriteRegister (StackReg, numPages * PageSize - 16);
    }
```

The system call `UserThreadCreate`

```
void print(int i) {
  PutInt(i);
  if (i % 2)
    PutString("Je suis un nombre impair\n");
  else
    PutString("Je suis un nombre pair\n");
}

int main() {
      PutString("Début du main…\n");
      UserThreadCreate(print,12);
      UserThreadCreate(print,23);
}
```

Standard treatment for the system call

- Define the flag
- Implement it in the assembler code
- Define a new handler for `UserThreadCreate`
  - `do_UserThreadCreate`
  - creation of a new kernel thread (`Fork`)
    - Attention! `Fork` does not do much...You should initialize correctly the address space and the registers

thread->Fork

```
void
Thread::Fork(VoidFunctionPtr func, int arg) {
    DEBUG('t', "Forking thread \"%s\" with func = 0x%x,
       arg = %d\n",
              name, (int) func, arg);
    StackAllocate(func, arg);
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

StackAllocate

```
void
Thread::StackAllocate (VoidFunctionPtr func, int arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof
   (int));
   ...
    machineState[PCState] = (int) ThreadRoot;
    machineState[StartupPCState] =
                            (int) InterruptEnable;
    machineState[InitialPCState] = (int) func;
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = (int) ThreadFinish;
}
```

So…

- `Fork` is for kernel threads
  - allocates physical memory
  - manages the basic thread management mechanism : call of the function, the passing of arguments, call of the return function
  - it is up to you to manage the user threads that are to share the same address space
- To do so, you should execute `Fork` taking the function `StartUserThread` as a paremeter
  `Fork(StartUserThread, ARG)`
  - the latter will correctly initialize the user thread (address space and stack)
  - you will need to pass the initial arguments (function to be executed by the thread ans its arguments) to this `StartUserThread` function (ARG). You should encapsulate the two data in order to pass them as a single argument.

The central point to manage is...

- user threads' stacks
  - When a user thread is cretaed yo should make sure that its stack does not overlap (delete) the stack of another thread
  - start with a simple test that creates only one thread
    - i.e a pg with two threads : the main and one created with `UserThreadCreate`
  - to manage multiple threads you should manage the available memory space => manage the available memory slots for stacks

Let us draw it...

- The memory space
- The first thread (main)
  - AddressSpace
  - StackReg?
  - PCReg?
- The 2nd thread
- The 3d thread
- ...
- Well, now there s no more space, so what about the nth thread?
- Now, the 2nd one is finished and there is a (n+1)th thread?

Implementation detail

- For stack management you will need the `BitMap` class

Thread Termination

- What happens if main terminates while the other threads are still running?
  - For threads created with `UserThreadCreate`, termination should be managed by `UserThreadExit`
  - `main` should not kill user threads
  - you will certainly need to count threads per address space (per process)
    - need for a semaphore

Compilation Issue for Using Semaphor in AddressSpace

- In synch.h, delete #include "thread.h".
- in network/post.cc, add #include "thread.h"

```
#include "copyright.h"
#include "post.h"
#include "thread.h"
#include <strings.h>
```