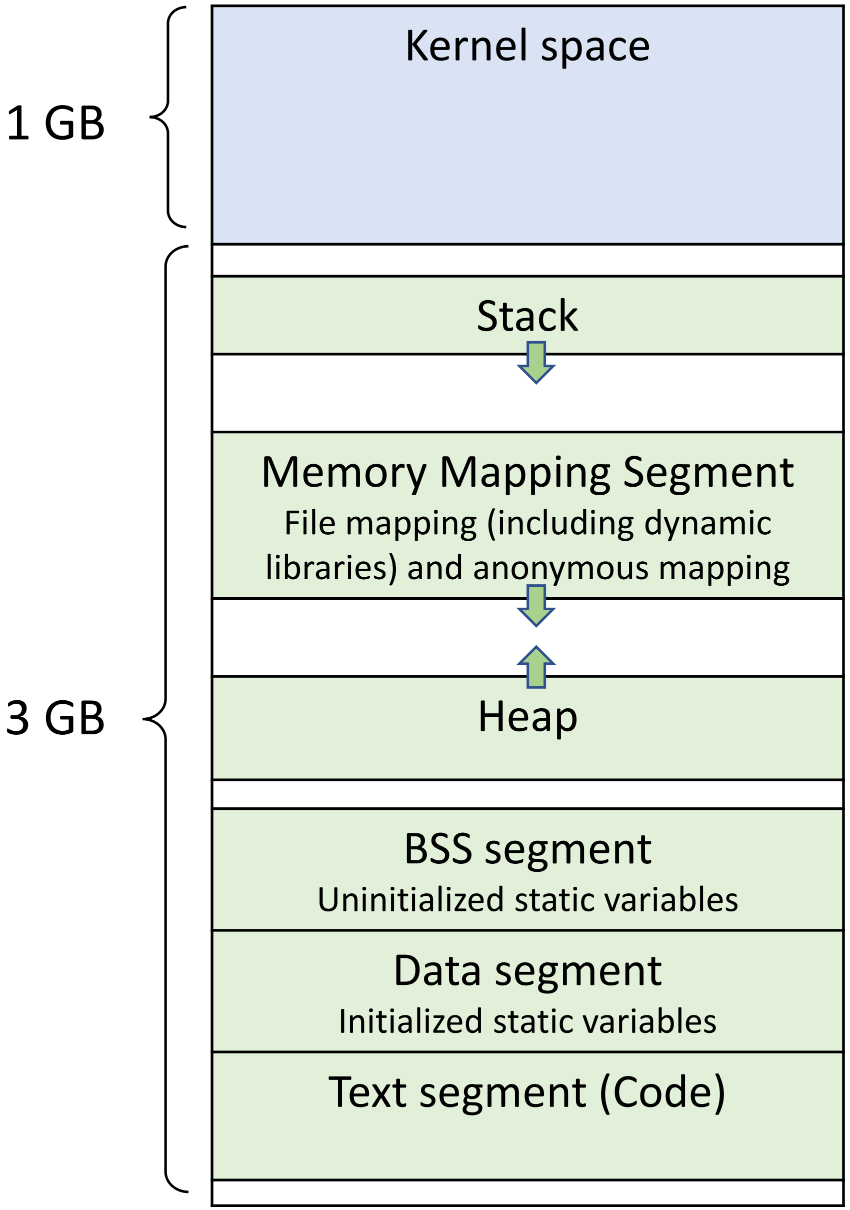
Second semester project

(Memory manager)



1 GB

Kernel space

3 GB

Stack

Memory Mapping Segment

File mapping (including dynamic libraries) and anonymous mapping

Heap

BSS segment

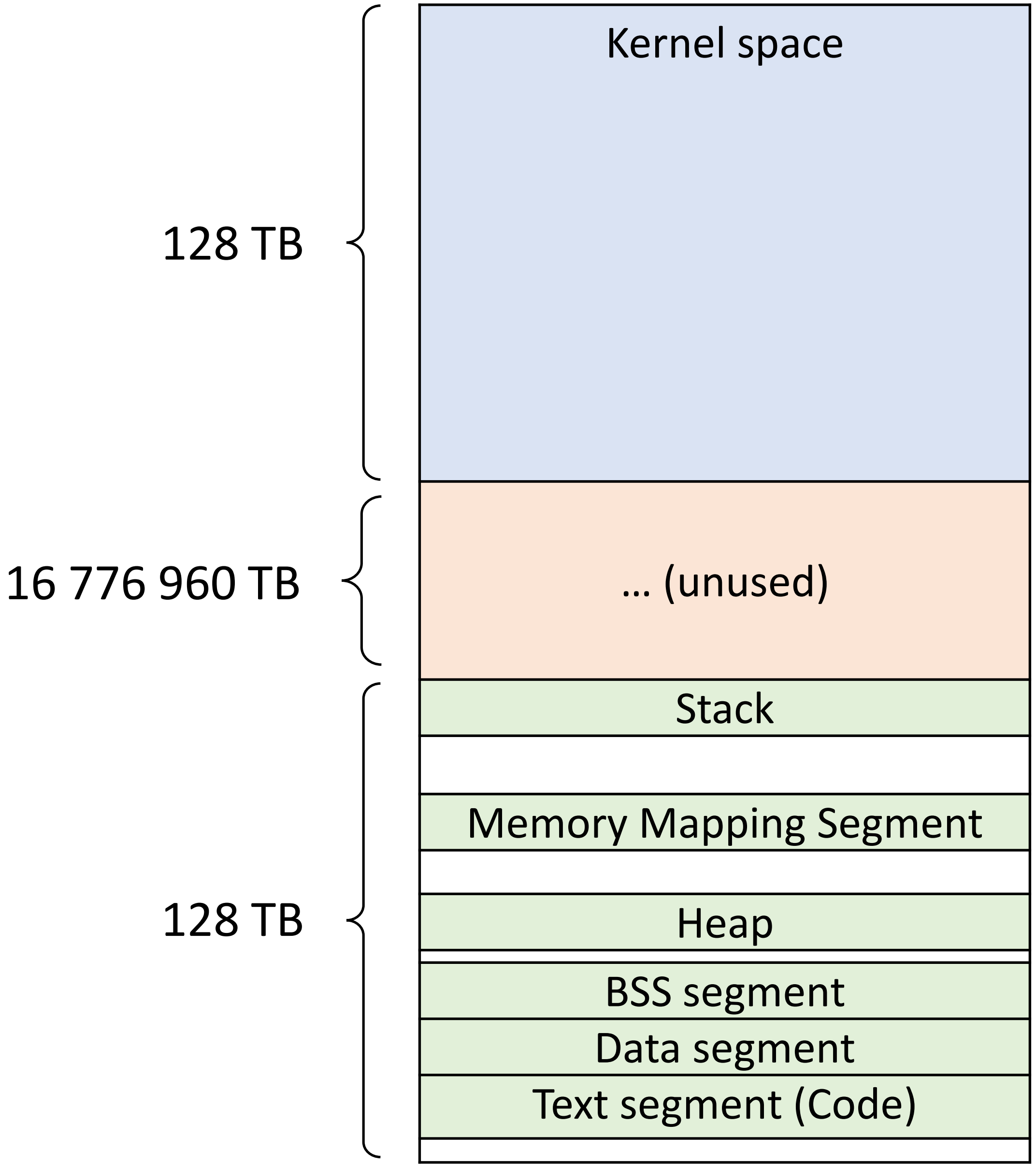
Uninitialized static variables

Data segment

Initialized static variables

Text segment (Code)

32-bit virtual address space



128 TB

Kernel space

16 776 960 TB

… (unused)

128 TB

Stack

Memory Mapping Segment

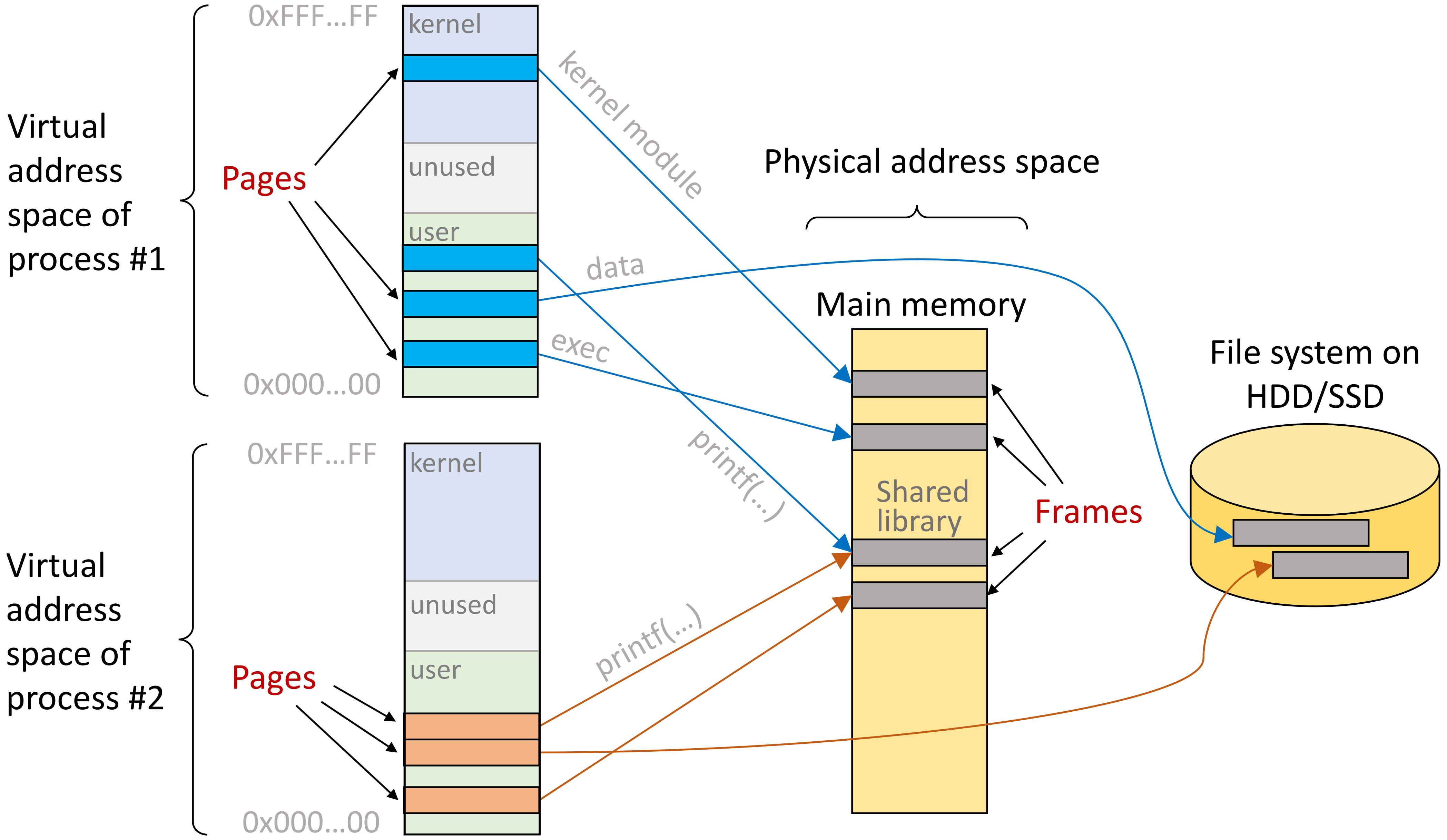
Heap

BSS segment

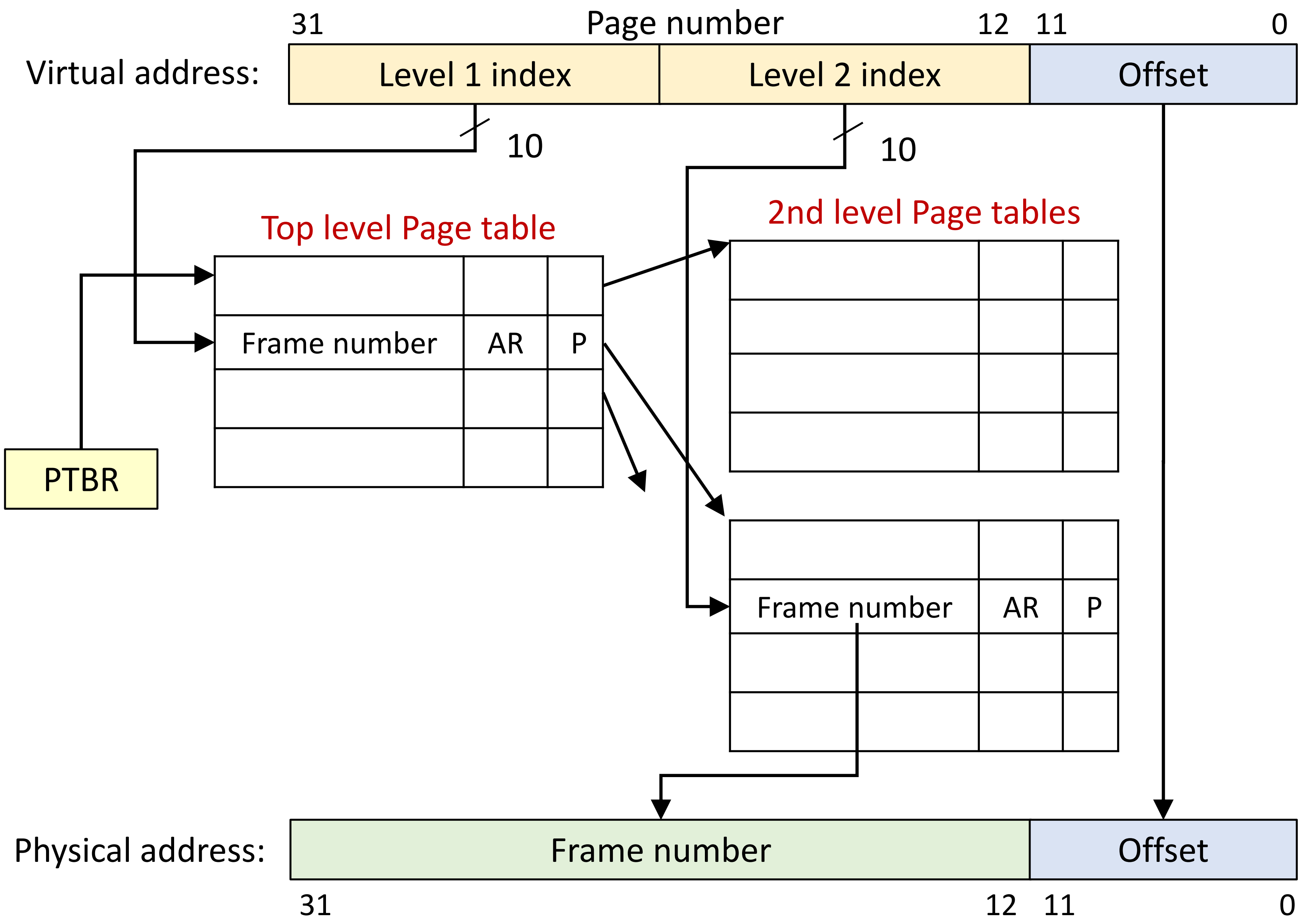
Data segment

Text segment (Code)

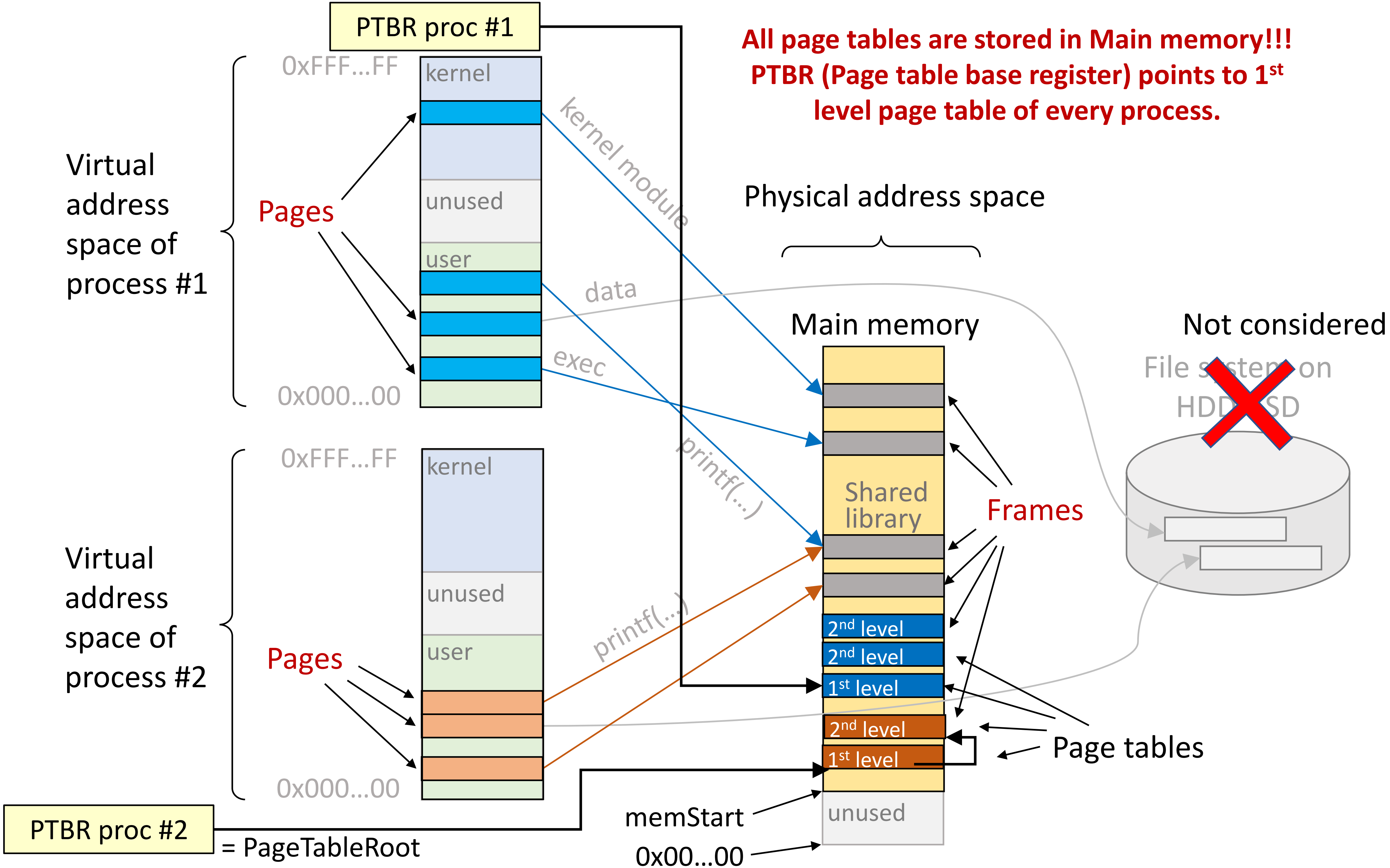
64-bit virtual address space



Virtual address space of process #1
Pages
0xFFF...FF
kernel
unused
user
0x000...00
kernel module
data
exec
printf(...)
Physical address space
Main memory
Shared library
Frames
File system on HDD/SSD
Virtual address space of process #2
Pages
0xFFF...FF
kernel
unused
user
0x000...00
printf(...)


Virtual address:
31
Page number
12
11
0
Level 1 index
Level 2 index
Offset
10
10
Top level Page table
2nd level Page tables
Frame number
AR
P
PTBR
Frame number
AR
P
Physical address:
Frame number
Offset
31
12
11
0

All page tables are stored in Main memory!!!
PTBR (Page table base register) points to 1st level page table of every process.

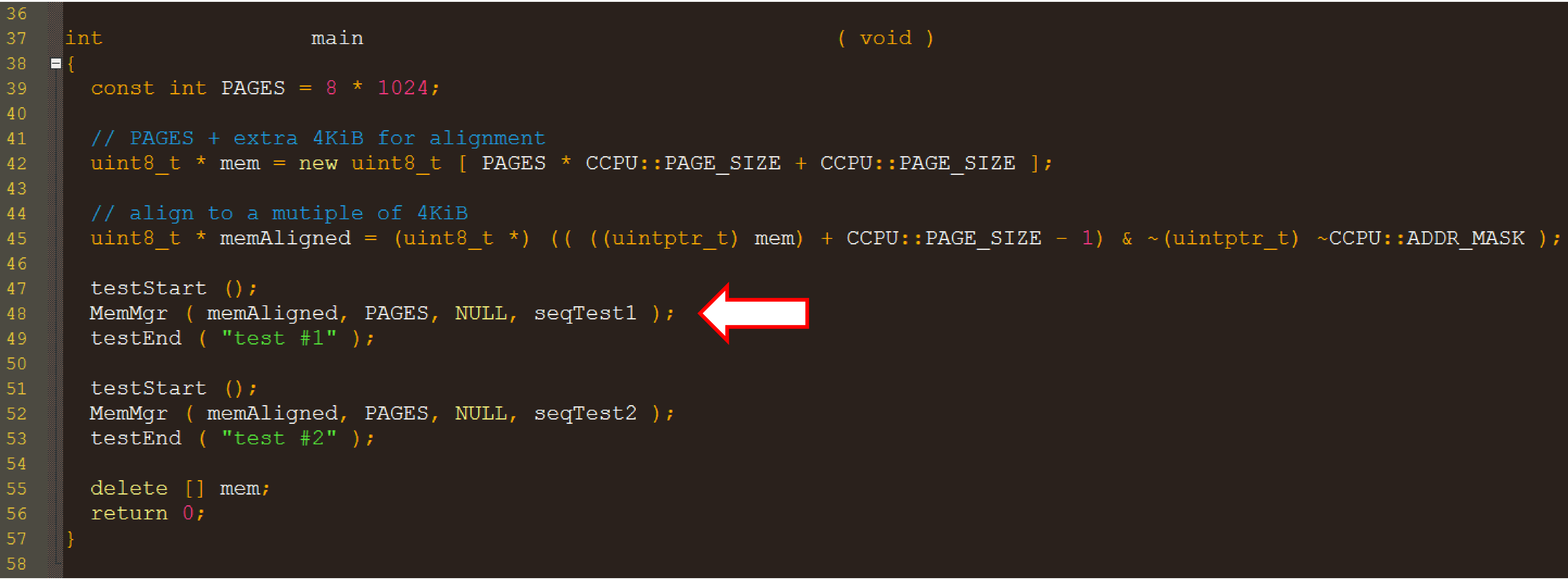


From Progtest:

Start of the simulation

- The simulation will be started when your function **MemMgr** is called. The interface of the function can be seen in the attached files.
- **The function will be passed a memory block to manage** (a pointer to the block + number of pages, each page is 4KiB).
- Next, the function will be passed a pointer-to-function. The pointed function will play the role of "init" process.
- Your implementation will initialize its internal structures, **prepares the first simulated process** (**CCPU instance**, ...), **and calls the "init" function**. The init process will be executed in the main thread (no need to create any extra threads yet). The init function will execute, it may create new simulated processes (using a method from its CCPU instance) and it will access its allocated memory.
- Finally, the function finishes and returns to your implementation and eventually, all other simulated processes terminate. The main thread shall **wait until all simulated processes terminate**, free all allocated resources, and return from MemMgr.

test1.cpp



```cpp
int                   main                              ( void )
{
  const int PAGES = 8 * 1024;

  // PAGES + extra 4KiB for alignment
  uint8_t * mem = new uint8_t [ PAGES * CCPU::PAGE_SIZE + CCPU::PAGE_SIZE ];

  // align to a mutiple of 4KiB
  uint8_t * memAligned = (uint8_t *) (( ((uintptr_t) mem) + CCPU::PAGE_SIZE - 1) & ~(uintptr_t) ~CCPU::ADDR_MASK );

  testStart ();
  MemMgr ( memAligned, PAGES, NULL, seqTest1 );
  testEnd ( "test #1" );

  testStart ();
  MemMgr ( memAligned, PAGES, NULL, seqTest2 );
  testEnd ( "test #2" );

  delete [] mem;
  return 0;
}
```

From Progtest:

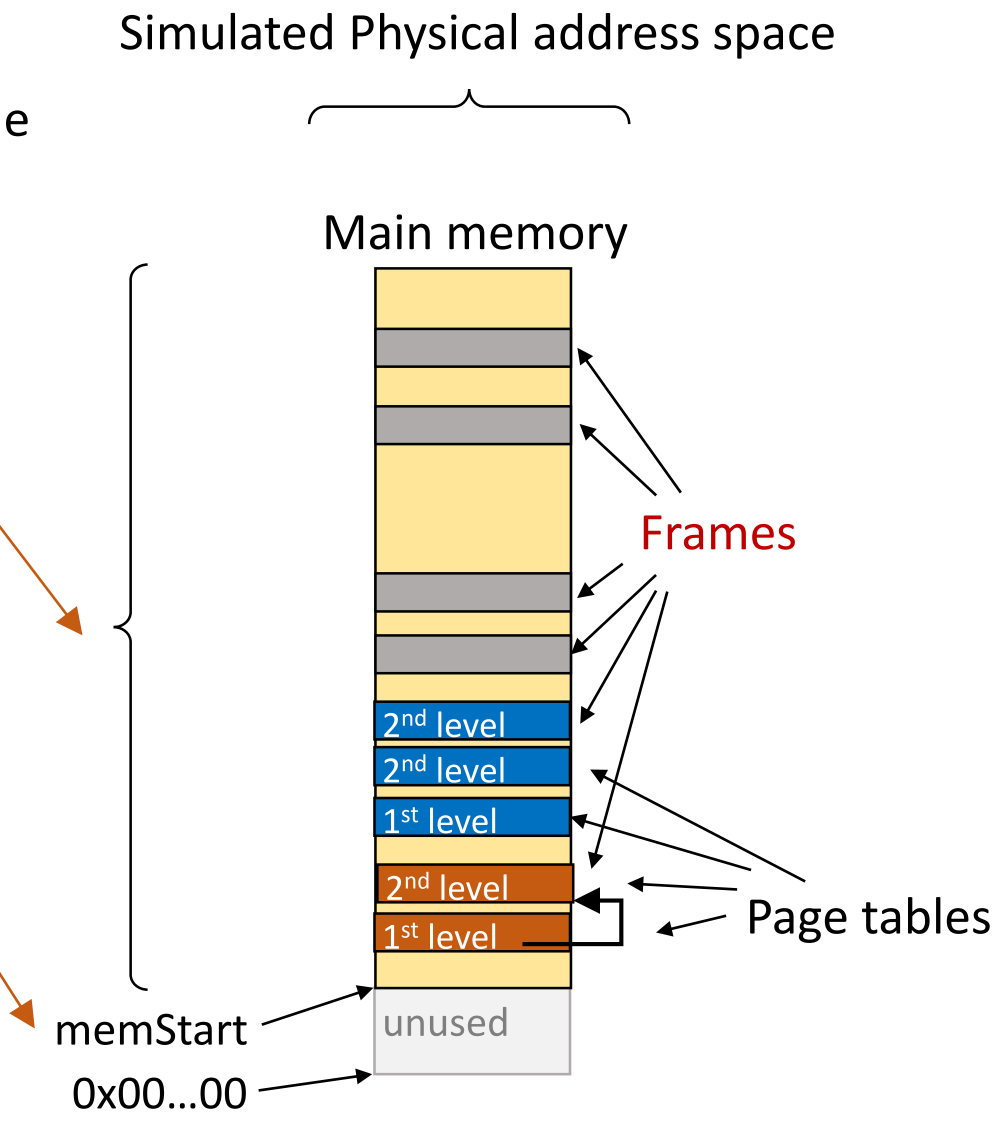
Addresses:

> Our user-space program cannot access real physical addresses and real CPU address mapping. The addresses inside our program (a real OS process) are virtual addresses for the real OS. However, these addresses will be assumed "physical" addresses for the simulation. Next, the "virtual" addresses in our simulation are just 32 bit integers that are passes to the CCPU calls. The CCPU implementation must do the translation.

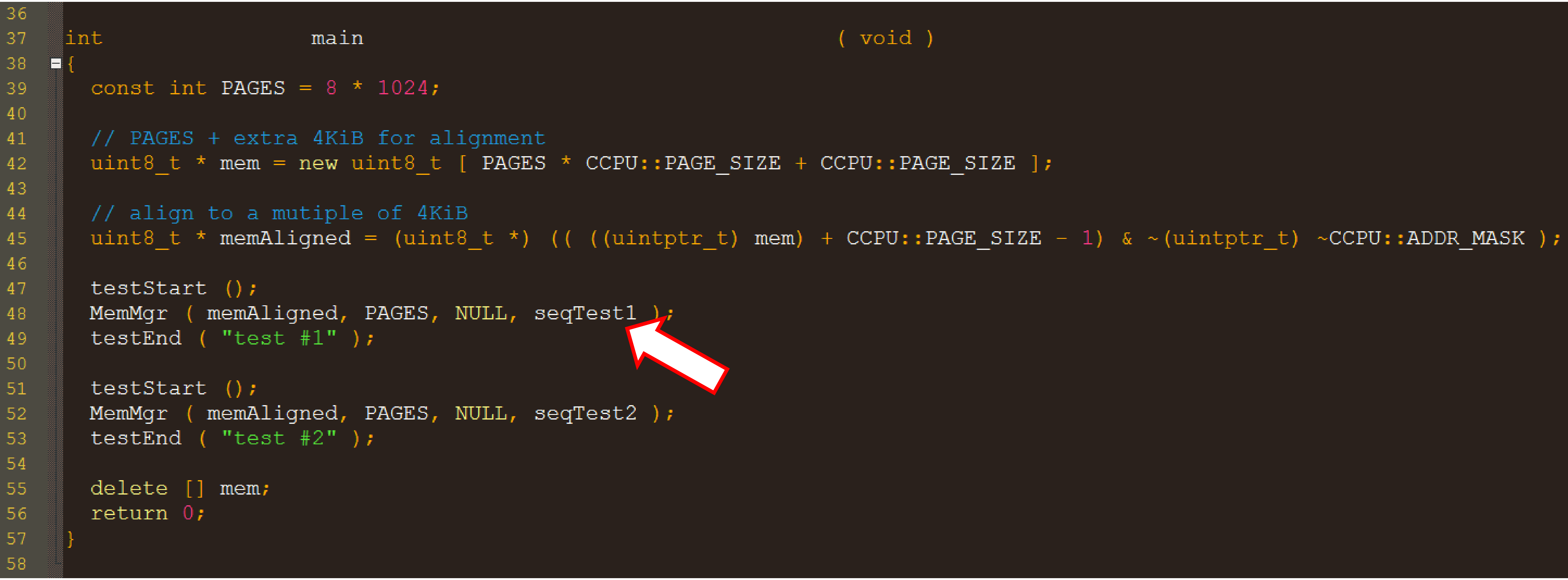
MemMgr (memAligned, PAGES, NULL, seqTest1);

Total number of physical pages in the simulated physical memory

Virtual address of the simulated physical memory



test1.cpp



```

int                main                                         ( void )
{
  const int PAGES = 8 * 1024;

  // PAGES + extra 4KiB for alignment
  uint8_t * mem = new uint8_t [ PAGES * CCPU::PAGE_SIZE + CCPU::PAGE_SIZE ];

  // align to a mutiple of 4KiB
  uint8_t * memAligned = (uint8_t *) (( ((uintptr_t) mem) + CCPU::PAGE_SIZE - 1) & ~(uintptr_t) ~CCPU::ADDR_MASK );

  testStart ();
  MemMgr ( memAligned, PAGES, NULL, seqTest1 );
  testEnd ( "test #1" );

  testStart ();
  MemMgr ( memAligned, PAGES, NULL, seqTest2 );
  testEnd ( "test #2" );

  delete [] mem;
  return 0;
}

```

From Progtest:

Start of the simulation

- The simulation will be started when your function **MemMgr** is called. The interface of the function can be seen in the attached files.
- **The function will be passed a memory block to manage** (a pointer to the block + number of pages, each page is 4KiB).
- Next, the function will be passed a pointer-to-function. The pointed function will play the role of "init" process.
- Your implementation will initialize its internal structures, **prepares the first simulated process** (**CCPU instance**, ...), **and calls the "init" function**. The init process will be executed in the main thread (no need to create any extra threads yet). The init function will execute, it may create new simulated processes (using a method from its CCPU instance) and it will access its allocated memory.
- Finally, the function finishes and returns to your implementation and eventually, all other simulated processes terminate. The main thread shall **wait until all simulated processes terminate**, free all allocated resources, and return from MemMgr.

Thus:

```
void MemMgr(
        void * mem,
        uint32_t totalPages,
        void * processArg,
        void (*mainProcess)(CCPU *, void *))
{
    1.  initialize internal structures (whatever you need)
    2.  prepare the first simulated process (from CCPU instance, i.e. instance of class
        CWhateverNiceNameYouLike : public CCPU -> see solution.cpp)
    3.  call the "init" function, i.e. mainProcess(&instance from step 2, processArg);
    4.  wait until all simulated processes terminate
    5.  free all allocated resources
}
```

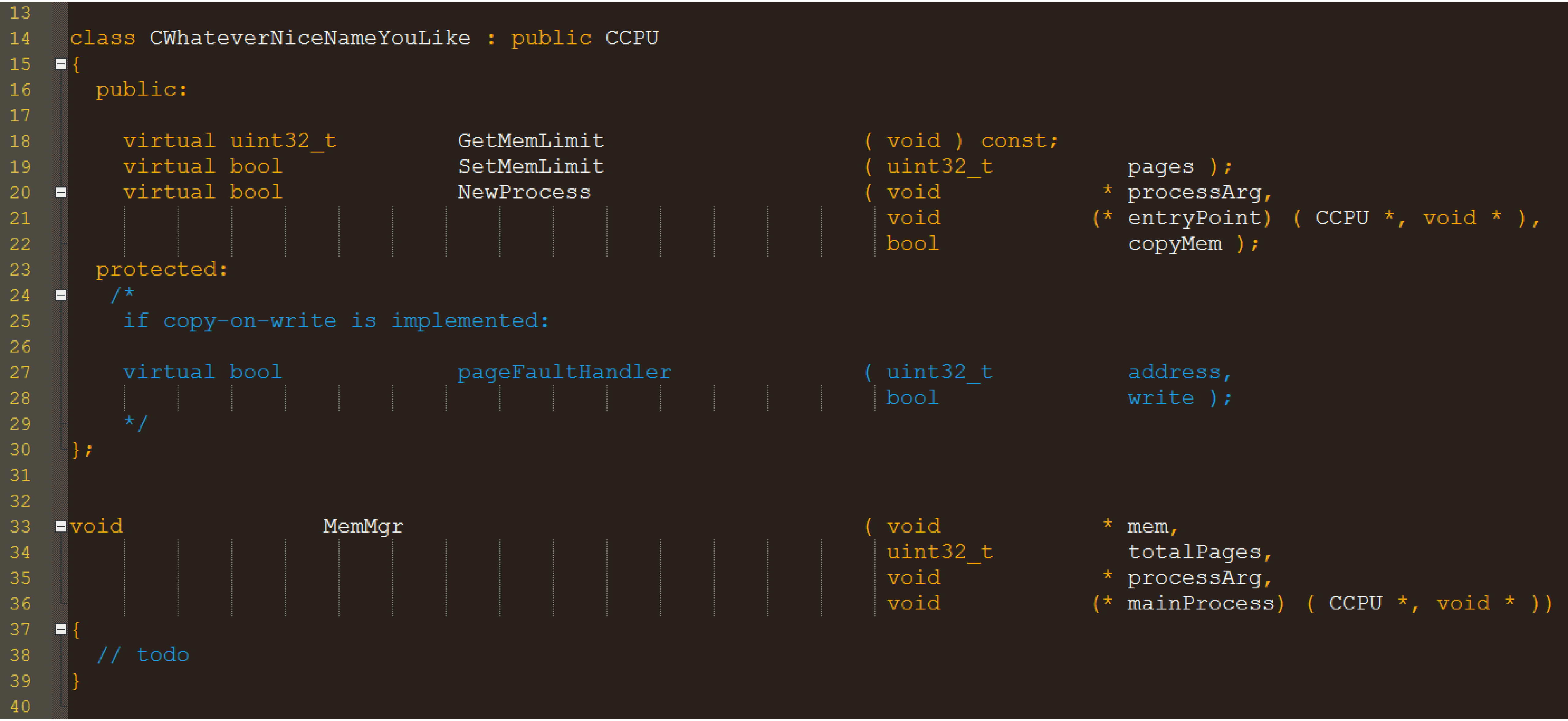
Step 2 (see previus slide)

From Progtest:

CPU

- The CPU's memory management unit (MMU) is implemented in the form of a C++ class CCPU. The class is an abstract class, the implementation is provided in the attached archive (and the same class is included in the testing environment).
- Your implementation will derive a subclass of CCPU and it shall **complete the missing abstract methods**.
- The simulated processes will use an instance of the class (the subclass) to access the memory allocated by the process. The methods will translate the addresses and mediate the memory access.

solution.cpp

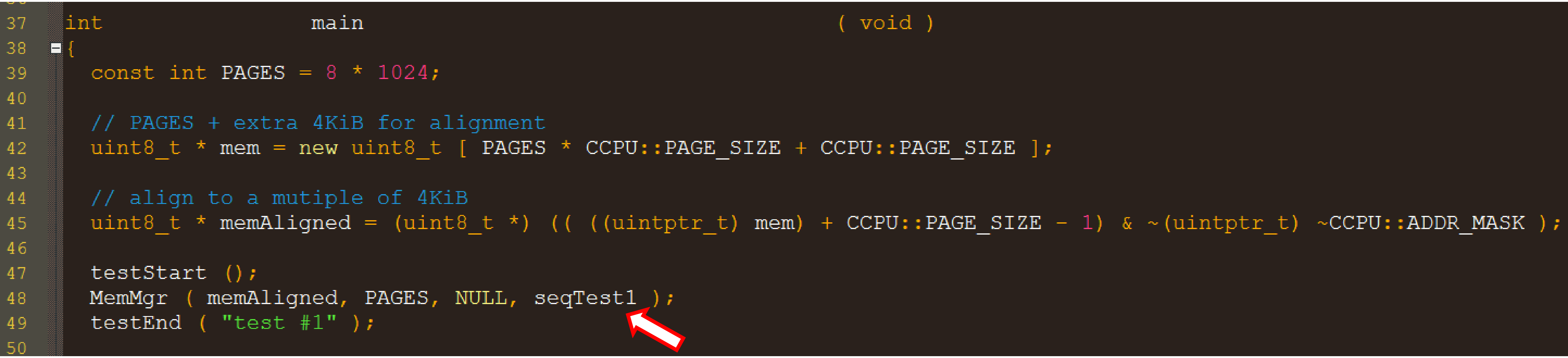


```cpp
class CWhateverNiceNameYouLike : public CCPU
{
  public:

    virtual uint32_t         GetMemLimit                    ( void ) const;
    virtual bool             SetMemLimit                    ( uint32_t          pages );
    virtual bool             NewProcess                     ( void            * processArg,
                                                              void           (* entryPoint) ( CCPU *, void * ),
                                                              bool              copyMem );
  protected:
   /*
    if copy-on-write is implemented:

    virtual bool             pageFaultHandler               ( uint32_t          address,
                                                              bool              write );
    */
};


void               MemMgr                                   ( void            * mem,
                                                              uint32_t          totalPages,
                                                              void            * processArg,
                                                              void           (* mainProcess) ( CCPU *, void * ))
{
  // todo
}
```

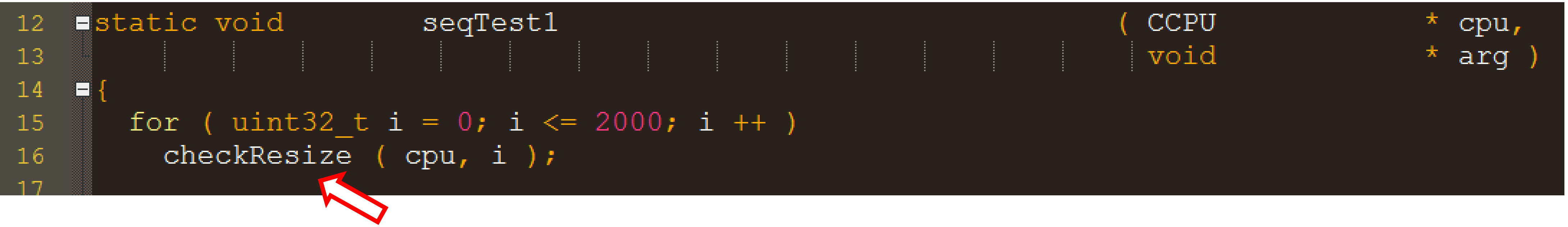
From Progtest:

- Method **GetMemLimit** () returns the total number of pages allocated for the simulated process. This is an abstract method, your derived class must implement it.
- Method **SetMemLimit** (pages) sets the memory limit for the process. The method may be used to either increase or decrease the memory limit. The method returns true to indicate success, or false for failure (e.g. out of memory). The method is abstract, your subclass must override it.

test1.cpp **GetMemLimit (), SetMemLimit ()**

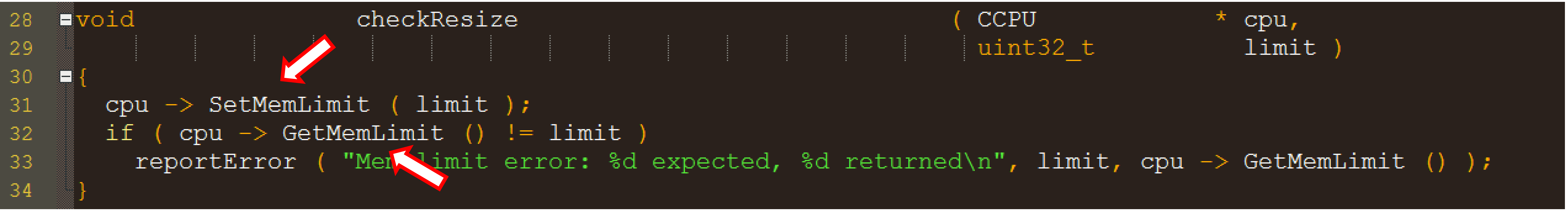


```
int                    main                                      ( void )
{
  const int PAGES = 8 * 1024;

  // PAGES + extra 4KiB for alignment
  uint8_t * mem = new uint8_t [ PAGES * CCPU::PAGE_SIZE + CCPU::PAGE_SIZE ];

  // align to a mutiple of 4KiB
  uint8_t * memAligned = (uint8_t *) (( ((uintptr_t) mem) + CCPU::PAGE_SIZE - 1) & ~(uintptr_t) ~CCPU::ADDR_MASK );

  testStart ();
  MemMgr ( memAligned, PAGES, NULL, seqTest1 );
  testEnd ( "test #1" );

```



```
static void          seqTest1                            ( CCPU           * cpu,
                                                           void           * arg )
{
  for ( uint32_t i = 0; i <= 2000; i ++ )
    checkResize ( cpu, i );

```

test_op.cpp



```
void                    checkResize                                ( CCPU           * cpu,
                                                                    uint32_t         limit )
{
  cpu -> SetMemLimit ( limit );
  if ( cpu -> GetMemLimit () != limit )
    reportError ( "MemLimit error: %d expected, %d returned\n", limit, cpu -> GetMemLimit () );
}
```

From Progtest:

- Method **NewProcess** will create a new simulated process (i.e. **creates a new instance of CCPU, allocates memory, prepares a new thread and executes the function**).
- The parameter is the function to execute in the new thread, a parameter for the function, and a boolean flag whether to copy the address space to the newly created simulated process.
- If the address space is not copied, the newly created process will be assigned 0 pages.
- If copied, the newly created process will have the address space size and contents copied from the caller. The method is abstract, your subclass must override it.
- When creating a new process, the **root page directory must be allocated**. It is not a good idea to allocate a complete all second level page directories (memory limit!). Allocate the root directory and allocate second level directories as needed.
- The implementation of SetMemLimit must update the page directories of the calling process. Before the page directory is updated, check there is enough free pages. If not, return failure and do not update anything. A failure to allocate memory cannot break the whole process or leave it in an undefined state.

Process A:

SetMemLimit (0);

root page directory must be allocated

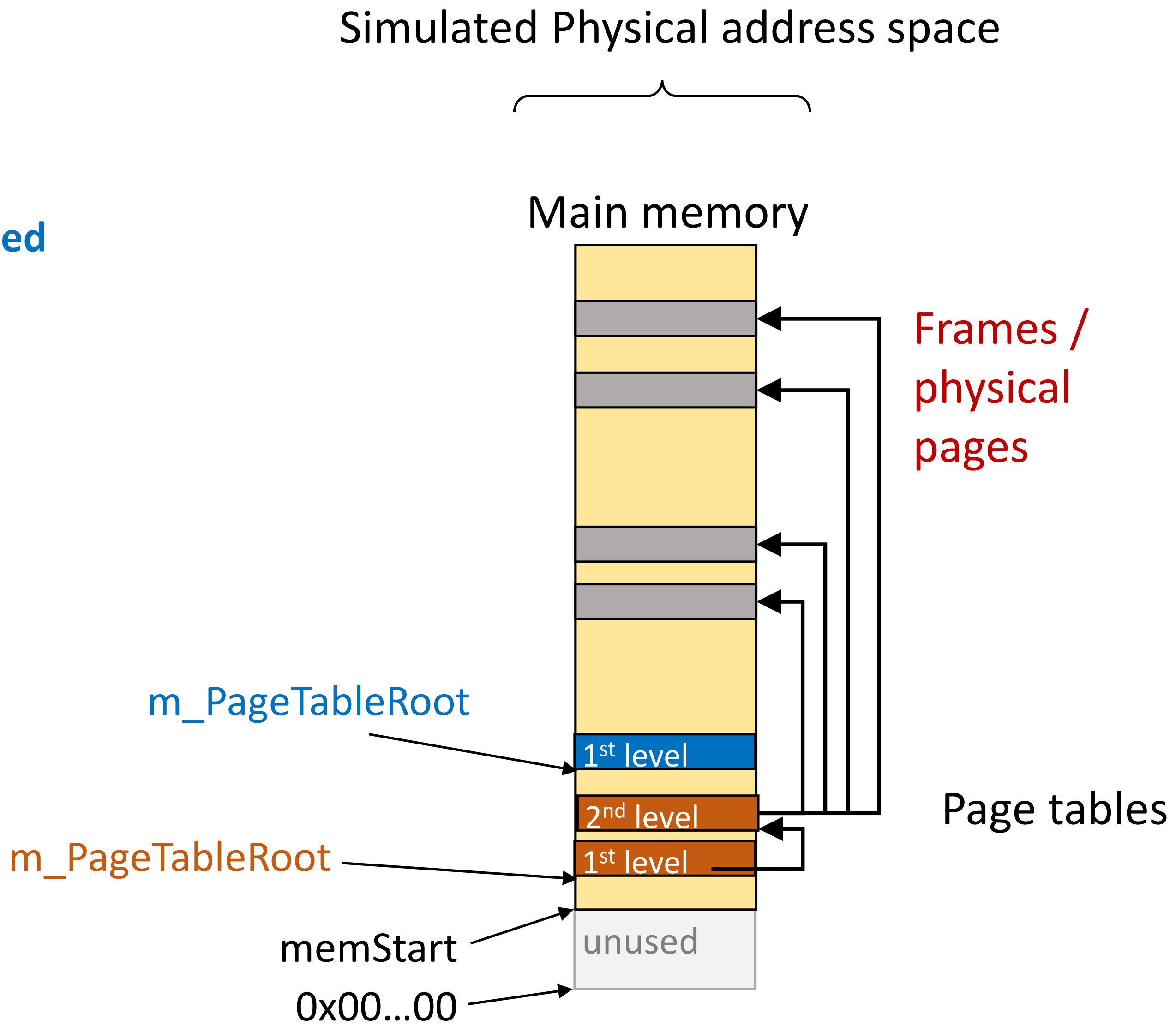
Process B:

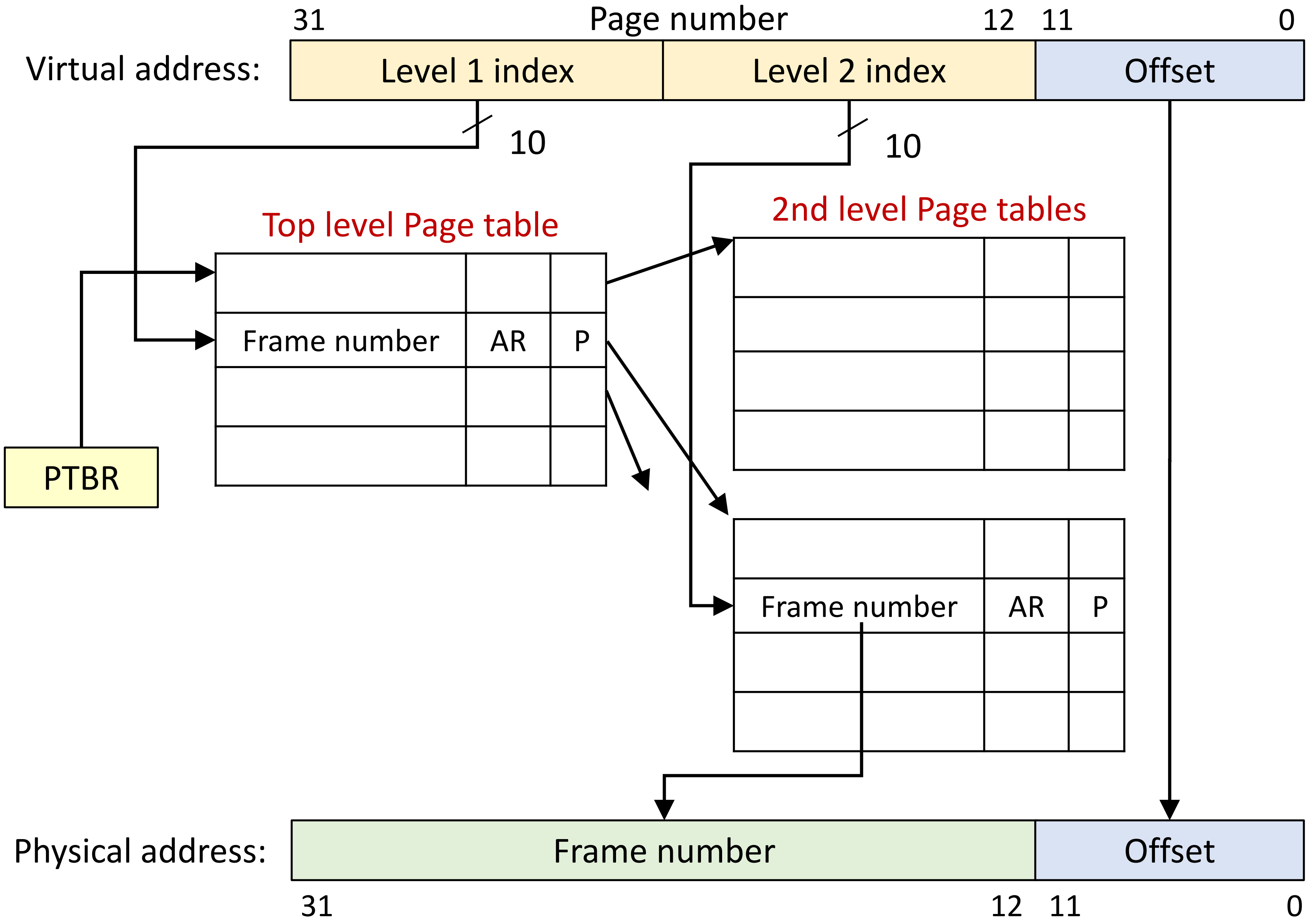
SetMemLimit (4);

root page directory must be allocated

+

Single 2nd level table with 4 valid entries (each entry pointing to the physical page base address)





All you need to do is to allocate as much page tables as needed and set individual page table entries accordingly.

If everything is set correctly, then method virtual2physical will work just fine.

Hint from Progtest:

- Your implementation must keep track of individual pages (used/free). **The easiest way is to use an array**. Since the total number of pages does not change, the size of the array is fixed. It is recommended to build a tree over the used/free array to speed-up the search for free pages.
- The data structure above (free/used flags) shall be placed in the managed memory too (i.e. your implementation may reserve several pages for this purpose). Your implementation is "cheating" if the free/used array is allocated outside of the managed memory pages. Moreover, there may not be enough memory to do so.

Process A:

SetMemLimit (0);

root page directory must be allocated

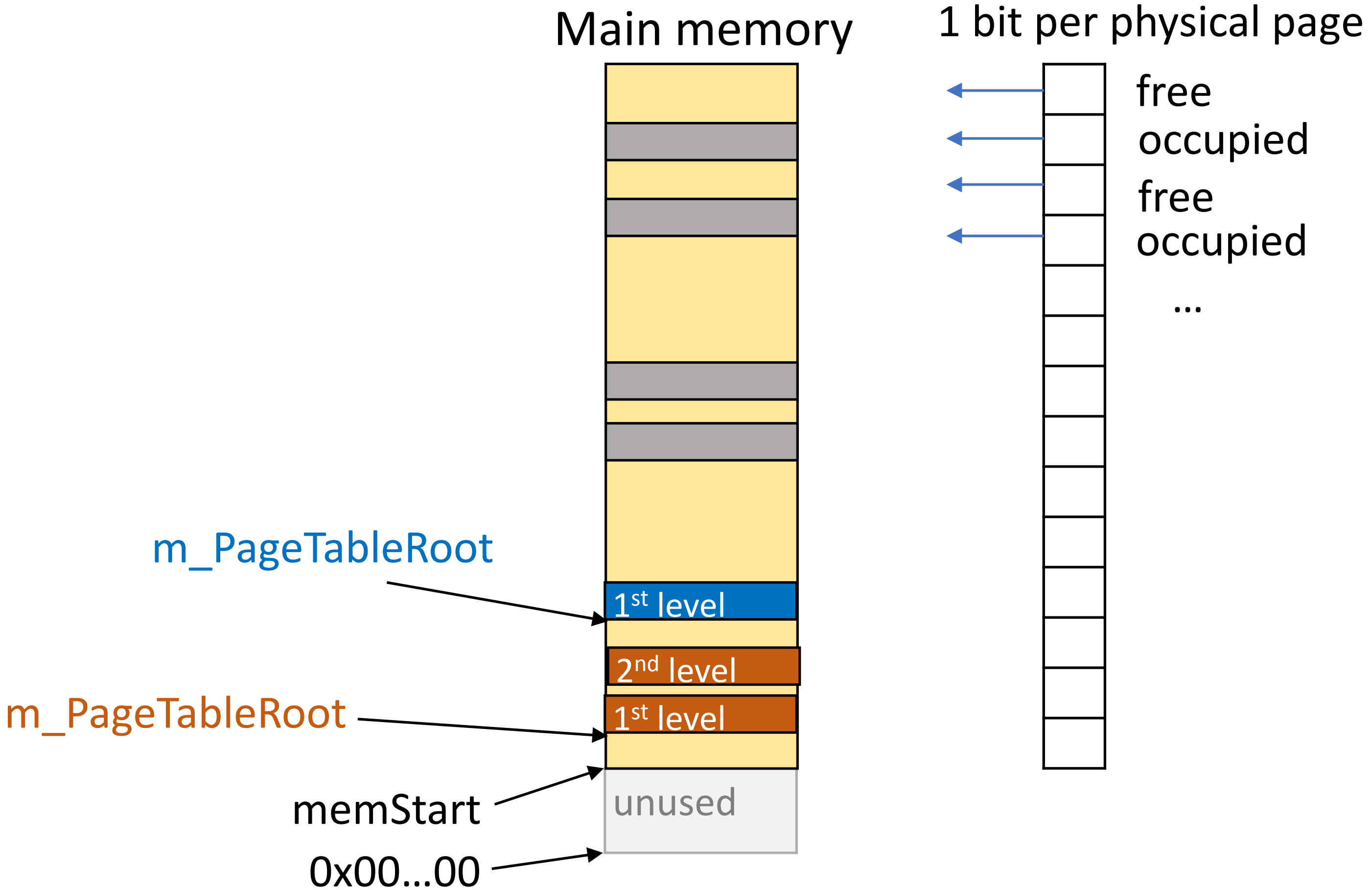
Process B:

SetMemLimit (4);

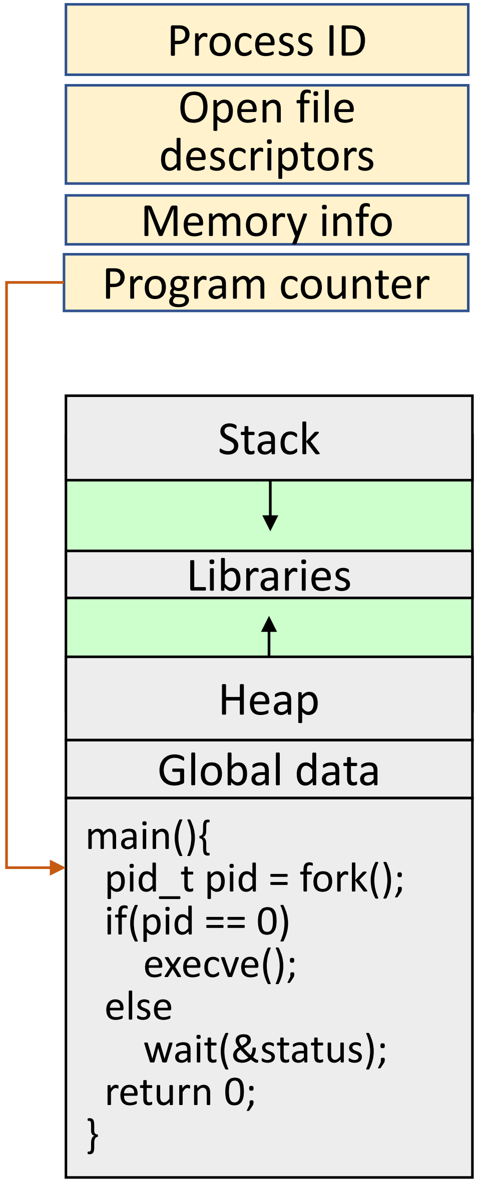
root page directory must be allocated

+

Single 2nd level table with 4 valid entries (each entry pointing to the physical page base address)



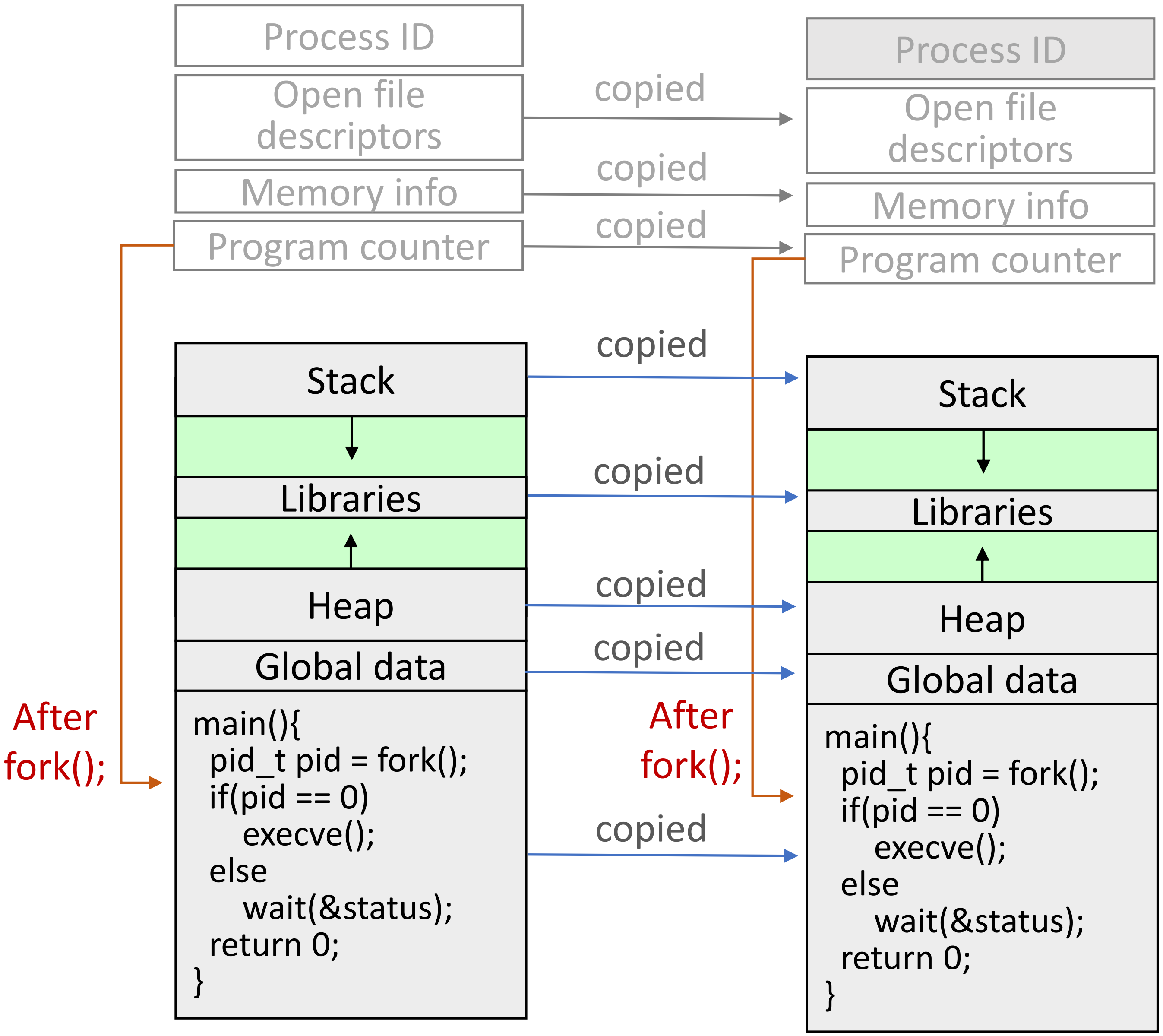
How to get bonus points:



```
main(){
  pid_t pid = fork();
  if(pid == 0)
      execve();
  else
      wait(&status);
  return 0;
}
```

Your task is to copy the address space when NewProcess (…, …, **true**);
is called.

Conceptual view.



```
main(){
  pid_t pid = fork();
  if(pid == 0)
     execve();
  else
     wait(&status);
  return 0;
}
```

- In reality, copy-on-write (COW) technique is used.

IDEA:

- OS just makes a copy of page tables of parent and markes all pages as „read only“ (for both parent and child).
- Each physical page also has a reference counter indicating how many processes are sharing that page. Before the fork, that counter is 1 and, after, it will be 2.
- When either process tries to write to a „read only“ page, it will generate a page fault. That fault is for COW page, thus, OS will create a new „writable“ page, will copy the data and decrement the reference counter.
- If reference counter becomes 1 and other process tries to write to a „read only“ page, it will generate a page fault again. However, this time OS will simply mark the page as „writable“ and „non-shared“.