1 Virtual memory

1.1 The segmentation approach

1.1-1

Convert 0x71 to binary:

```
0x71 = 0111 0001
```

Then we can know that

- The top 2 bits (`01`) indicate segment is `01` → Heap segment.
- The lower 6 bits (`110001`) is `0x31` (decimal `49`) is the offset.
- Heap segment size is 50 bytes → valid offsets range from 0 to 49.
- Offset 49 (`0x31`) is within 0-49, so valid.

Thus, the physical address is:

```
Physical Address = Heap Base + Offset = 0x200 + 0x31 = 0x231
```

Ans: 0x231

1.1-2

```
0x11 = 0001 0001
```

- `00` → Code segment
- offset → `010001` is `0x11` (dec `17`)
- Code segment size is 32 bytes → offset is valid

Thus, the physical address is: `0x111 (base) + 0x11 (offset) = 0x122`.

Ans: 0x122

1.1-3

- Code segment: from `0x111` (dec `273`) up to `0x130` (dec `304`) , +31 bytes
- Heap segment: from `0x200` (dec `512`) up to `0x231` (dec `561`) , +49 bytes

Thus, `0x130 + 1` ~ `0x200` is unused → 207 bytes

Ans: 207 bytes

1.2 The paging approach

1.2-1

TLB is just for speed up, without TLB, it would need an additional memory access to read PDE and PTE

Ans: False

1.2-2

- In PTE: Indicates whether the referenced page is legitimately in physical memory (valid = 1) or not present/invalid (valid = 0). If invalid, an access triggers a page fault or similar exception.
- In TLB entry: Specify if the TLB entry itself is usable for translation. If valid=0, the TLB must ignore that entry.
- In PDE: Indicates whether that portion of the address space has a valid second-level page table.

1.2-3

- linear page table
  - We must have one big page table covering all 4 GB.
  - Total pages in the virtual space: 4 GB / 4 KB = 2^20 = 1,048,576 pages.
  - Each page requires a PTE of 32-bit long ⇒ total page-table size is

    ```
    1,048,576 * 4 bytes = 4MB
    ```
- 2-level paging
  - page directory: 4 KB
    - contain 1024 PDE
    - each PDE point to an second-level page
    - each second-level page contain 1024 PTE (2^{12} / 2^2 bytes)
  - 1 page for code, 1 page for heap, 2 pages for stack
    - 4 pages ⇒ can put in only one second-level page
    - Thus, only one PDE is needed, which means only one PDE is valid
    - Thus, total cost is 1 page directory + 1 second-level page:

      ```
      4KB + 4KB = 8KB
      ```

Thus, the space saving is 4MB - 8KB

Ans: 4MB - 8KB

> 4,194,304 bytes - 8,192 bytes = 4,186,112 bytes

1.2-4

- num of page = 16KB / 4KB = 4 ⇒ top 4 bits are for the VPN.
- page size is 4KB ⇒ offset = 12 bits ⇒ 16 ~ 12 bits are for the VPN.

1.2-4-(a)

```
0x012a = 0001 0010 1010
```

- Split the address:
  - offset = lower 12 bits = `0001 0010 1010` (decimal 298).
  - VPN = upper 4 bits = (`0x012a >> 12`) = `0x0`.
- Look for VPN=0x0 and ASID=0 with valid=1
  - We find `VPN=0x0, PFN=0x10, valid=1, prot=rw-, ASID=0`
- Construct the physical address:
  - `PFN=0x10` ⇒ PFN = `0x10` = `0001 0000`.
  - Physical address = `(PFN << 12) + offset`.
  - `PFN << 12` = `0001 0000 0000 0000 0000` = `0x10000`
  - Add offset `0x12a` ⇒ `0x10000 + 0x12a` = `0x1012a`.

Answer: 0x1012a

1.2-4-(b)

16 KB vitrual-address ⇒ 0x0000 ~ 0x3FFF

Thus, 0x5123 is out of range

Ans: Segmentation fault

1.3

ChatGPT:

- Segmentation
  - Pros: Can match logical program units (code, data, stack) naturally; can reduce internal fragmentation if each segment matches exact size.
  - Cons: External fragmentation is likely; each segment must be allocated contiguously in physical memory; can cause segmentation faults if offset is out of range.
- Paging
  - Pros: Eliminates external fragmentation; simpler memory allocation in fixed-size pages; easy to move pages around or swap them out.
  - Cons: Internal fragmentation (the last part of each page may go unused); multi-level page tables can become large and add overhead (extra memory lookups); TLB misses add latency.

ChatGPT typically covers the main conceptual advantages and disadvantage, maybe it can discuss more deeply like hybrid approach and the real-world usage

2 Working with xv6

Modify the `runcmd` function in `sh.c`:

```
  case BACK:
    bcmd = (struct backcmd*)cmd;
    if(fork1() == 0) {
      // Double-fork to run the actual background command.
      int bgpid = fork1();
      if(bgpid == 0) {
        // Grandchild: execute the actual background job.
        runcmd(bcmd->cmd);
        exit(0);
      }

      // Reaper child: wait for the grandchild (the real bg job).
      wait(0);

      // Now the background command has finished. Print the done message:
      // We know the PID was 'bgpid'.
      // For the NAME, we just re-print the first argument of bcmd->cmd
(common approach).
      // If bcmd->cmd is an execcmd, we can cast and read its argv[0].
      int silent = 0;
      char *progname = 0;
      if(bcmd->cmd->type == EXEC) {
        struct execcmd *ec = (struct execcmd*)bcmd->cmd;
        if(ec->argv[0]) {
          progname = ec->argv[0];

          if(strcmp(progname, "osleep") == 0) {
            // Looking for -s
            for(int i = 1; ec->argv[i] != 0; i++){
              if(strcmp(ec->argv[i], "-s") == 0){
                silent = 1;
                break;
              }
            }
          }
        }
      }

      if(!silent)
        printf("%d Done %s\n", bgpid, progname);
    }
```

Just do one more fork to generate an process to handle the print usage. Generally, it just wait it and get the PID and program name from the `bcmd->cmd`. I think the command should always be `EXEC` type, but not 100% sure though.

There is an flag `silent` to determine if the `printf` should be execute, it would only be unflagged for the command `osleep -s time &`.

For the `osleep` command:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/fcntl.h"

int
main(int argc, char *argv[])
{
  // Expect usage:
  //   osleep -s <seconds> &
  // or
  //   osleep -v <seconds> &
  // or
  //   osleep <seconds> &   (choose a default mode, e.g. silent)
  if(argc < 2){
    fprintf(2, "Usage: osleep [-s | -v] <seconds>\n");
    exit(1);
  }

  int timeIndex = 1;
  if((strcmp(argv[1], "-s") == 0) || (strcmp(argv[1], "-v") == 0)){
    if(argc < 3){
      fprintf(2, "Usage: osleep [-s | -v] <seconds>\n");
      exit(1);
    }
    timeIndex = 2;
  }

  int ticks = atoi(argv[timeIndex]);
  if(ticks < 1){
    fprintf(2, "osleep: invalid sleep time\n");
    exit(1);
  }

  sleep(ticks);
  exit(0);
}
```

The implmentation was simple, just parse the command, find the `-s` and `-v` flag, then call `sleep`.