This exam is 75 minutes. Stop writing when “time” is called. You must turn in your exam; we will not collect it. Do not get up or pack up in the final ten minutes. The instructor will leave the room 78 minutes after the exam begins and will not accept exams outside the room.

There are 15 problems in this booklet. Many can be answered quickly. Some may be harder than others, and some earn more points than others. You may want to skim all questions before starting.

This exam is closed book and notes. You may not use electronics: phones, tablets, calculators, laptops, etc. You may refer to ONE two-sided 8.5x11” sheet with 10 point or larger Times New Roman font, 1 inch or larger margins, and a maximum of 55 lines per side.

If you find a question unclear or ambiguous, be sure to write any assumptions you make.

Follow the instructions: if they ask you to justify something, explain your reasoning and any important assumptions. Write brief, precise answers. Rambling brain dumps will not work and will waste time. Think before you start writing so that you can answer crisply. Be neat. If we can’t understand your answer, we can’t give you credit!

If the questions impose a sentence limit, we will not read past that limit. In addition, a response that includes the correct answer, along with irrelevant or incorrect content, will lose points.

Don’t linger. If you know the answer, give it, and move on.

Write your name and NetId on this cover sheet and on the bottom of every page of the exam.
I  C programming (18 points total)

1. [8 points] Consider the following C program:

```c
#include <stdio.h>

void f(int p, int q) {
    p = q;
}

void g(int* p, int q) {
    p = &q;
}

void h(int* p, int q) {
    *p = q;
}

int main() {
    int a, b;
    a = 9;
    b = 2;
    f(a, b);
    printf("%d %d\n", a, b);
    a = 9;
    b = 2;
    g(&a, b);
    printf("%d %d\n", a, b);
    a = 9;
    b = 2;
    h(&a, b);
    printf("%d %d\n", a, b);
    return 0;
}
```

What does the program print?

9 2
9 2
2 2
2. [10 points] We modify the WeensyOS process_t structure to add a field, p_stake, which contains an arbitrary positive integer; there is no relationship between a process’s p_stake and its p_pid. Your job is to write a function that returns the pid of the process with the highest p_stake value, out of all runnable processes. Assume that there is at least one runnable process.

Fill in the function below, and note that the definitions after the function may be helpful. For full credit, your code must be in syntactically correct C.

```c
// A process descriptor for each process.
// Note that proc_array[0] is never used;
// The first application process descriptor is proc_array[1].
static process_t proc_array[NPROCS];

int highest_stake_runnable_proc()
{
    int pid, maxstake_pid;
    int maxstake = 0;

    // FILL THIS IN; your code will need to iterate over the proc_array

    return maxstake_pid;
}
```

Here are some relevant definitions:

```c
// Process state type
typedef enum procstate {
    P_EMPTY = 0, P_RUNNABLE, P_BLOCKED, P_ZOMBIE
} procstate_t;

// Process descriptor type
typedef struct process {
    int p_pid; // Process ID
    int p_stake; // <-- NEW FIELD
    registers_t p_registers; // registers, stack location, EIP, etc.
    procstate_t p_state; // Process state; defined above
    int p_exit_status; // Process's exit status
} process_t;
```

Name: Solutions NYU NetId:
II I/O, Devices, threads (35 points total)

3. [5 points] Recall the function `swtch()` that we saw in class: it performs a context switch for user-level threads. Consider the following statement: “`swtch()` makes a system call.”

Is this statement True or False? Justify using one sentence.

False. `swtch()` is implemented entirely in user space; there is no involvement from the operating system.

4. [5 points] Assume an operating system (OS) that presents a completely synchronous I/O interface to processes. This means: (1) if a system call requires the OS to perform I/O, that system call blocks until the I/O is complete; and (2) there is no way for a process to ask the OS whether a given call would block. (As a technical point, there is no memory mapping of files.) Assume a process that has four user-level threads.

How many I/O requests from this process can be pending at the OS at once? Explain your answer in no more than two sentences.

1 request. The operating system does not “see” the user-level threads: it sees the process as a unit. Because any I/O-inducing system call blocks the entire process, there is no way for a single process to achieve I/O parallelism in this setup.

5. [8 points] We grade True/False questions with positive points for correct items, 0 points for blank items, and negative points for incorrect items. The minimum score on this question is 0 points. To earn exactly 1 point on this question, cross out the question and write SKIP.

Circle True or False for each item below:

- **True / False** Bugs in a kernel can cause the kernel to crash. True.
- **True / False** A buggy device driver running in a non-buggy kernel can cause the kernel to crash. True. The device driver generally runs in the address space of the kernel, and can corrupt its memory.
- **True / False** A buggy process running on a non-buggy kernel can cause the kernel to crash. False. The kernel is supposed to be engineered so that, regardless of what a process does, it cannot crash the kernel.
- **True / False** Interacting with devices through memory-mapped I/O requires the device driver to use special stack manipulation instructions. False.
- **True / False** In class, we saw examples of primitive device drivers. True.
- **True / False** In a machine that supports Direct Memory Access (DMA), devices have the ability to load from and store to the machine’s RAM. True.
6. [4 points] Fill in the blank: We said in class that there is a tradeoff between polling and 
interrupts.
7. **[13 points]** Consider a disk with the following characteristics:

- The disk has 6 platters (and 6 corresponding heads); changing which head is active has zero cost
- The disk makes 128 rotations per second
- Each sector is 4096 bytes
- There are 256 sectors per track (we are ignoring the fact that the number of sectors per track varies on a real disk)
- There are 1024 tracks per platter
- The track-to-track seek time is 0 milliseconds
- The average seek time for a read is 10.5 ms; for a write it is 12 ms.
- The maximum seek time is 16 ms.
- Ignore the time to transfer the bits from the disk to memory; that is, once the disk head is positioned over the sector, the transfer happens instantaneously.

For the questions below, you will generally want to be working with powers of 2. To that end, here is a potentially helpful table:

<table>
<thead>
<tr>
<th>Power</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^6$</td>
<td>64</td>
</tr>
<tr>
<td>$2^7$</td>
<td>128</td>
</tr>
<tr>
<td>$2^8$</td>
<td>256</td>
</tr>
<tr>
<td>$2^9$</td>
<td>512</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>1024</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Power</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{11}$</td>
<td>2048</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>4096</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>8192</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>16384</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>32768</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>65536</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>131072</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>262144</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>524288</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>1048576</td>
</tr>
</tbody>
</table>

What is the storage capacity of the disk in bytes or gigabytes? Explain briefly (for example by showing your work).

**Answer:** 6 GB. 6 platters/disk * 1024 tracks/platter * 256 sectors/track * 4096 bytes/sector = $6 \cdot 2^{10} \cdot 2^8 \cdot 2^{12} = 6 \cdot 2^{30}$ bytes = 6 GB

What is the sequential transfer bandwidth, expressed in bytes/second or megabytes/second? Explain briefly (for example by showing your work). Hint: the disk reads one track in a rotation.

128 MB/s. We modeled the track-to-track seek time and the I/O bus overhead as negligible, so we look at the number of bytes in a track and how long it takes to read a track (itself given by the rotation speed): 128 rot/sec * 1 track/rot * 256 sectors/track * 4096 bytes/sector = $2^7 \cdot 2^8 \cdot 2^{12} = 2^{27}$ bytes/sec = 128 MB/s.
Now, assume that we have two disks of the kind above. We arrange to make them exact copies of each other (this is RAID-1; there is no striping). Notice that, to obtain a logical sector, the system can read from either disk. The question below depends on your answer to the prior two questions. If you did not answer either or both of those, then use “C GB” and “T MB/s” as respective stand-ins.

If the RAID system is at capacity (that is, storing the maximum amount of data), what is the minimum time in seconds to read all logical sectors into memory? (Assume that there is enough RAM to store all of the data.) Explain briefly.

24 seconds. The system can, in parallel, read the first half of the data from disk 1 and the second half of the data from disk 2. So the answer is \((C/2)/T = (6 \cdot 2^{30}/2)/2^{27} \text{s} = 3 \cdot 8 \text{s} = 24 \text{s}.

III Virtual memory, paging, page replacement (30 points total)

8. [6 points] A computer has a 32-bit virtual address space, 24-bit physical addresses, and 1 KB pages. (You saw this combination on the homework; if you are curious about the differently-sized virtual and physical addresses, imagine that the machine handles 32-bit pointers but only accommodates 16 MB of RAM.) Note that 1 KB = $2^{10}$ bytes. Below, you do not need to show your work.

What is the maximum number of virtual addresses per process?
$2^{32} = 4G$, because addresses are 32 bits.

How many bits is the VPO (virtual page offset)?
10, because pages are 1KB.

How many bits is the VPN (virtual page number)?
22, because the remainder of the bits reference a virtual page.

How many bits is the PPO (physical page offset)?
10. Same as VPO.

How many bits is the PPN (physical page number)?
14, because the remainder of the physical address bits reference a physical page.

9. [5 points] This question is about page replacement. Assume that there are 4 physical frames; these start out empty. Assume the following sequence of $2N$ references to virtual pages, where $N > 4$:

$$1, 2, \ldots, N - 1, N, N, N - 1, \ldots, 2, 1,$$

(As usual, a reference either hits the cache or else causes a cache fill; in the latter case, if the cache entry was occupied, an eviction occurs.)

(a) State the optimal (highest) number of cache hits, over all cache replacement policies, and
(b) State a policy that achieves this optimum number. You do not need to justify your answers.

4, achievable via OPT, LRU, and FIFO (but not, say, MRU [most recently used]).

One way to get this answer is to simulate OPT since we know that OPT is optimal.

Another is to intuit that since each page is referenced only twice, only the second reference can generate a hit, and furthermore, this can happen for only 4 entries. A more formal version of this reasoning is as follows; we will restrict to implementable policies. Claim: 4 is an upper-bound. Consider the viewpoint of a single cache entry; that entry cannot experience more than one hit, total, over the lifetime of this sequence (if it experiences, say, two hits, that would mean that the cache entry would have “seen” a pattern like $x, \ldots, x, \ldots, y, \ldots, y$, which is impossible, given the reference sequence.) Since there are 4 cache entries total, the upper-bound is 4, as claimed. Meanwhile, 4 is actually achievable (by several policies), and so the bound is tight; hence, 4 is optimal.

The point of this example is that, if there is not a lot of repetition in the access pattern, then caching does not do a lot of good. In this case, it gets us from $2N$ misses to $2N - 4$ misses; if $N$ is large, this difference is minor.
10. [13 points] Below, you will draw the high-level page table structure of a single WeensyOS process after it is running (you created this structure in exercises 2 and 5 of lab 5). By high-level, we mean that your drawing should accurately depict the number of level-1 and level-2 page tables that a WeensyOS process uses, and how these objects connect to each other; use arrows to indicate which entries in which objects point to which other objects. You do not need to depict the format or contents of a page table entry (beyond the specified arrows and connections). Terminology note: The terms “level-1 page table” and “page directory” are synonymous.

Draw the WeensyOS process page table structure in the space below:

There should be a one L1 page table, with the 1st (zeroth) entry pointing to a single L2 page table.

As a WeensyOS kernel developer, you decide that you want 8 KB (= $2^{13}$ bytes) of physical memory, starting at physical address $0xA0000$, to appear in the address space of WeensyOS process 1. The memory should appear starting at virtual address $0x2fd000$, and the process should be permitted to store to these virtual addresses. Below, you will write a line of code to create the required mapping. (You can imagine that your code is invoked when WeensyOS sets up process 1.) You do not need to handle pageinfo bookkeeping.

Create the required mapping, and note that useful definitions follow:

```c
// after the next line, 'pagetable' points to process 1's level-1 page table
x86_pagetable* pagetable = proc_array[1].p_pagetable;

// YOUR CODE HERE. USE vm_memory_map(). ONLY ONE LINE IS NEEDED.
```

Here are some useful definitions:

```c
// vm_memory_map(pagetable, pa, va, sz, perm)
// Maps virtual address range '[va,va+sz)' to '[pa,pa+sz)'
// in 'pagetable', with permissions 'perm'
vm_memory_map(x86_pagetable* pagetable, uintptr_t pa,
              uintptr_t va, size_t sz, int perm);

#define PTE_P ((unsigned long) 1) // entry is Present
#define PTE_W ((unsigned long) 2) // entry is Writeable
#define PTE_U ((unsigned long) 4) // entry is User-accessible

#define PAGESHIFT 12 // log_2 (size of page in bytes)
#define PAGESIZE (1 << PAGESHIFT) // Size of page in bytes
```

```c
vm_memory_map(pagetable, 0xA0000, 0x2fd000, PAGESIZE*2, PTE_P|PTE_W|PTE_U);
```
11. [6 points] This question is about how software can simulate the Accessed bit. Recall that the Accessed bit, sometimes called PTE_A in code, is one of the bits in a page table entry on the x86 (other bits include PTE_P, PTE_U, etc.). Let \( P \) be a running process; let \( T \) be \( P \)'s page table (the full structure, which you depicted on the previous page); and let \( e \) be a given entry in \( T \). The CPU automatically sets the Accessed bit in \( e \) when \( P \) performs a load or store to the virtual page that corresponds to \( e \). Assume that the kernel clears the Accessed bit in all entries in \( T \) when \( P \) is context switched out.

Now, posit an architecture that is identical to the x86 except that the CPU does not automatically set the Accessed bit. In this alternate, the kernel can do whatever it wants with this bit; the hardware completely ignores it.

**In this alternate, how can the kernel simulate the Accessed bit?** **Hint:** the mechanism of page faults is essential. Use no more than three sentences. By “simulate”, we mean that the kernel should arrange for a process’s Accessed bits (in the process’s page table entries) to be set exactly as if the process were executing on an x86 CPU.

When a process is context switched in, mark the entries in all page tables as not present (clear PTE_P); this requires that the kernel store the information about which pages truly are not present. When a process loads or stores to a page, there will be a page fault. If the page access is valid (according to the information about which pages are truly present), the kernel sets the Accessed bit, sets PTE_P, and restarts the process.
IV File systems and feedback (17 points total): this is the last page

12. [5 points] We often take our computer’s files for granted, but they are an abstraction created on top of hardware. Which entity creates the abstraction of a file?

Circle the BEST answer:

A The disk
B The CPU
C The north bridge
D The user-level program
E The user-level threading package
F The file system software
G The SSD

F.

13. [5 points] In the classic Unix file system, the i-node is an imbalanced tree.

What is the advantage to making the tree imbalanced? Limit your answer to two sentences.

Access to short files (5120 or fewer bytes) requires no traversals to fetch metadata.

14. [5 points] In class and in the reading, we studied FFS (the Fast File System).

State one optimization introduced by FFS, not including the buffer cache and not including bitmaps. Use a maximum of one sentence:

Multiple answers possible here.

15. [2 points] This is to gather feedback. Any answer, except a blank one, will get full credit.

Please state the topic or topics since midterm 1 that have been least clear to you.

Please state the topic or topics since midterm 1 that have been most clear to you.