SOLUTION

Multiple-Choice (5 points each)
Pick the one **best** answer for each question.
(there may be more than one {arguably} correct answer)
Please print your answers in UPPER-CASE on the line provided

1) A file `/usr/bin/scr.sh` has execute permission and its first line is `#!/bin/sh -vx`. We `cd /usr/bin ; ./scr.sh foo`. Where does `/bin/sh` see the string `foo`?

A) argv[0]
B) argv[1]
C) argv[2]
D) argv[3]

Unit 3, pp. 5-6 explain how the argument vector of the #! interpreter is constructed. argv[0] of course is the usual argv[0] of the interpreter, i.e. argv[0]="sh". Since the #! line has the optional argument following the path to the interpreter binary, argv[1]="-vx". Now the full pathname of the script file itself is passed (so the interpreter can open it and start reading the commands) so argv[2]="/usr/bin/scr.sh". Finally, the remaining argument vector of the script invocation is appended, and argv[3]="foo"

2) Consider the idealized model of a UNIX filesystem presented in Unit #2 where a directory is a list of (name,inode) pairs. Let us say we sample the first byte of the name field for all of the directories in the volume and count unique byte code values. What is the maximum possible number of unique values?

A) 254
B) 255
C) 256
D) 257

Unit 2, pg. 7 and extensive in-class discussion: The only two characters which are NOT valid in a path component are forward slash (because it is the delimiter) and NUL (because path name arguments, e.g. to open system call, are NUL-terminated). Therefore there are 254 valid characters.

3) On an x86-32 system running a Linux kernel, an interrupt handler for the disk I/O controller is running:

A) When the handler concludes, the CPU's User/Supervisor bit will revert to USER
B) The handler should use a blocking reader lock before accessing a data structure that might also be written to by a system call handler
C) The handler may set the NEED_RESCHED flag
D) The kernel will demand-page-in the handler's text region

Unit 7, pp. 9-10 explain kernel control paths. Interrupt handlers can run at almost any time, including "on top of" active kernel code such as system call or page fault handlers. The user/supervisor bit will only revert to user mode if the interrupt handler interrupted user-mode code. If it interrupted kernel code, then the kernel code resumes after the interrupt handler, with Supervisor privilege of course. The same section talks about deferred context switch with the NEED_RESCHED flag. The handler often sets this flag when it feels that the task which it interrupted might be a candidate for a task switch (e.g. because it just woke up a "better" task). (C) is the correct choice. An interrupt handler can not use any kind of blocking lock, because it could then deadlock against the kernel code that it interrupted. Say the system call handler had just obtained the corresponding writer lock to the shared data structure and then the interrupt came. The interrupt handler would be blocked waiting for the reader lock, which could never be satisfied because it is held by the interrupted system call handler! Choice (D) was intended as a farce: kernel code is ALWAYS resident.

4) Consider the page table structure of X86-64 (**64 bit**) running a Linux kernel

A) The 256th entry of the PGD (counting from 0) represents the start of kernel virtual memory
B) There are 1024 entries in each Page Table
C) Physical Addresses are 64 bits wide
D) Each entry in a PMD maps to one PUD

The key phrase here, repeated twice and **bolded, no less**, *was 64-bit. Yes, in the X86-32 page table structure, the page tables have 1024 entries. But in 64-bit, they have only 512 entries. Unit 5, pp. 6-7 detail that the kernel/user dividing line in virtual address space is at the half-way point, which is the pgd[256] entry (counting from pgd[0]), i.e. user va space is 128TB and kernel space is also 128TB. This differs from the 32-bit model where the split is 3GB / 1 GB. The PTE only has room for a 40-bit PFN which equates to a 52-bit Physical Address (not 64). That's still 4PB of physical memory, which will continue to be an outrageously generous amount for many years. In the 64-bit 4-level page table, each PGD entry maps to one PUD which maps to PMD and finally to PT. Choice (D) is reversed.*

5) Examine the short program below. A parent process picks up the exit status of this program via the `wait` system call. What is it?

```
main()
{
        kill(getpid(),SIGINT);
        return 0;
}
```

A) 0
B) -1
C) 2
D) undefined

It was clarified during the exam that the program was invoked with the "usual conditions" meaning in particular that the signal handling table is at default values. One can always send oneself a signal, since by definition the sending process and the receiving process will have the same uid. getpid() will never fail (if it does, the world is ending anyway) so the kill system call 100% works. The process dies immediately from a signal #2. Since the process exit status consists of the signal number in the LSB (or 0 in the LSB if the process exited voluntarily and the return value in the upper byte), then wait status is 2.

6) Examine program `MC #1`. What is the last message printed to stderr?

A) Nothing is ever printed
B) `counter = 1`
C) `All done`
D) It is indeterminate

Unit 4, pp. 7-9 basically give this as an example! Because we longjmp out of the signal handler (not sigsetjmp/siglongjmp) the signal mask at the point of returning from setjmp the "second time" (return value !=0) is what it was during the handler. Since sa_flags=0 we know that the default behavior of masking the signal being handled happened (set sa_flags= SA_NOMASK if you don't want this). When the program starts, first the signal handler is established. Then we take the "first time" through setjmp and wind up at the fork. The child and parent run "simultaneously". Although it is not defined which runs first (or both on a multi-cpu system), this has no bearing on the problem, which is entirely deterministic. The parent enters an endless loop just beyond GOTO_IS_EVIL: The child sends multiple SIGINTs. The parent's signal handler is invoked, it increments counter to 1 (initial value is 0 because it is a bss variable) and then longjmps. Now we are in the "second time" leg of setjmp. The message counter = 1 is printed and we goto back to the endless loop. SIGINT is still masked. Subsequent SIGINTs from the child are never taken and the parent never exits the endless loop. The child concludes and exits (producing no output). Therefore (B) is correct.

7) The Page Frame Reclamation Algorithm (PFRA)

A) Sets the PTE A (Accessed) bits as it scans

B) Tends to steal the most recently used pages

C) Swaps out anonymous pages back to their mapped files

D) Is moody, depending on how shallow the pool is

Unit 5, pp. 37-39 describe the PFRA. It CLEARS (never sets) the A bits as it scans (the A bits are SET, never cleared by hardware). Based on this, it develops a notion of the LEAST recently used pages, and these are stolen first. Anonymous pages (if dirty) are paged-out to swap, because they have no mapped files. The PFRA is "moody" ("adaptive") in altering its page reclamation aggressiveness depending on how low the free page pool is getting.

8) Assume the PTE on an X86-32 Linux system is laid out with flag bits: PRWXDAS, followed by 5 unused bits, followed by the 20-bit PFN. The P flag is the most significant bit [this is a distortion of reality but it is how we did the examples in class]. You are browsing the page table entries of a process and you encounter one with value 0x00001234. What can you conclude?

A) When it is paged-in, it will occupy PFN 0x01234

B) It is an anonymous page

C) It corresponds to offset 0 within inode #0x1234

D) Copy-on-write is in play here

Unit 5, pp. 35-37 show the different states of a PTE and what they mean. In this case, clearly the P(resent) bit it is OFF. The virtual page is NOT mapped in. Copy-on-write only applies to virtual pages that are mapped-in and (temporarily) sharing a page frame. The inode and offset are stored in the vm_area_struct, not the PTE. When a page fault occurs at this address, indeed a new page frame will be allocated to satisfy it, but the kernel doesn't do "assigned seating" like some kind of airline. The page frame assigned will be whichever is at the head of the free list at the time. At that time, the PFN field of the PTE will be set to the correct value, but while the PTE is invalid, the PFN field is used by the Linux kernel to remember where it swapped out an anonymous page (it encodes the swap area # and swap slot # as described in notes). Ans: (B)

P R W X D A S

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0

9) Look at the assembly program below which is executed on X86-32 Linux. What is the value of the %eax register at end of the code snippet?

```
start:    movl      $0xFFFFFFFF,%eax
          movl      $0,%ebx
          movl      $1,%ecx
          int       $0x80
```

A) EINVAL
B) ENOSYS
C) 0
D) None of the above

The eax register holds the system call number. The system call entry assembly code which is in Unit #7 and which we went through line-by-line in class compares this against the highest system call number. It happens to be an unsigned comparison, so 0xFFFFFFFF is seen as 4294967295. Now, Linux has a lot of system calls, but not that many! And if you thought it was a signed number, -1 isn't a valid number either. So we are going to get an ENOSYS. Now here's a slightly tricky twist which we did go over in class, but I still gave half-credit for (B) even though it is wrong: System call return values are negative for error, positive or zero for success, and if negative, then the value is -errno. So upon return from the system call, the value in eax is actually -ENOSYS. (D) is the correct answer

10) A thread within a multi-threaded X86-32 Linux process makes a blocking system call which calls `schedule`. The scheduler picks another thread in the same process as the `next`. What does **not** change?

A) CR3 register
B) Kernel stack pointer
C) TSS
D) `current`

A multi-threaded program, from the Linux kernel standpoint, consists of multiple tasks, each of which is assigned a distinct pid (this becomes a thread id (tid) at user level), and therefore a distinct kernel stack and current (task_struct). The TSS would also change since the kernel stack is going to change. But the CR3 register controls address space and it remains the same.

11) Oh no, your `/home` filesystem (volume) is out of disk space (attempts to create new files fail with ENOSPC and `df` reports 0% free). A quick scan shows a file `/home/me/WOW.tgz` consuming 1TB. You run `rm /home/me/WOW.tgz` and then take another `df`. The amount of free space has not changed. This indicates

A) You did not have write permission on `/home/me/WOW.tgz`
B) There is another symlink resolving to `/home/me/WOW.tgz`
C) The file is open
D) You are not the owner (uid) of `/home/me`

This is a classic example of the "ghost file" (Unit 2, pp. 11-12): the resources are not freed because the file is open somewhere. You do not need write permission on a file to delete it. You do need write permission on /home/me, but you could have that even if you are not the uid owner, either via group or other permissions having the W bit on. While a HARD link would keep the inode link count above 0 and thus not delete the file, a SYMLINK does not behave this way.

12) A volume which is formatted as an EXT3 filesystem with journalling enabled is under heavy activity creating and writing to files when the system suddenly loses power. After reboot

A) `fsck` will need to examine the entire inode table, free map and each directory
B) transactions with a BEGIN record and no matching COMMIT record will be replayed
C) manual intervention by the sysadmin will be required to mount the volume again
D) some data that programs believed were written successfully might be lost

Unit 2, pp. 39-41 talk about journalling filesystems. They obviate the need for a long and tedious integrity check after a crash, so !A. Only complete transactions are "replayed" (written again to the disk). Without the matching COMMIT record, we don't know if we have the entire transaction, so we can't replay a potentially incomplete and corrupted transaction. Manual intervention might be required in cases of extreme corruption but is not an expected result. The correct answer is (D). The journal has several modes, and while all protect the metadata, only the most conservative (and low-performance) mode protects the actual data. Even with this, programs would need to explicitly fsync() before close or use the O_SYNC flag to receive an iron-clad guarantee that successful system call returns mean the data are really on the disk.

13) Give an example of a pseudo-filesystem on Linux

A) /proc
B) NFS
C) EXT4
D) VFAT

Because of a typo, this inadvertently became a trick question. The lecture notes define Network File Systems (such as NFS and SMBFS) as a sub-category of pseudo-filesystems. Choice (B) was supposed to read NTFS, the native format of Windows. Likewise VFAT is a DOS/Windows format. Both of these are "alien" filesystems but Linux will recognize them. They are not pseudo filesystems since they correspond to actual on-disk volumes. /proc is a true pseudo-filesystem where its entries (which we explored extensively in class) don't correspond to any thing on any volume anywhere. Instead they are "files" and "directories" that represent things such as per-process file descriptor tables. NFS, a network file system, is a way of attaching a "real" filesystem on a remote file server via the network.

14) We wish to have a producer/consumer FIFO in shared memory accessed by multiple threads or processes. Which synchronization primitive would be appropriate for coordinating the full/empty sleep/wakeup conditions?

A) semaphore
B) spin lock
C) seqlock
D) mutex

Only the semaphore provides sleeping and waking up based on a condition. We could use semaphores to represent the number of empty slots and the number of pending FIFO data. spin locks and mutexes prevent simultaneous access but can not keep track of conditions. A seqlock is an "optimistic sync" method for improving mutex-like performance when the chances of and penalties for actual conflict are low.

Problem LA1 (20 points)

```
1         int i=15;
2         int u;
3         char a[8000];
4
5         int main(int argc,char **argv)
6         {
7           int l1,l2;
8                   i++;
9                   l2=a[10];
10                  switch(l1=fork())
11                  {
12                   case -1:perror("this won't happen");return -1;
13                   case 0:
14                           a[20]='X';
15                           l2++;
16                           break;
17                   default:
18                           i=a[5000];
19                           f(i);
20                           break;
21                  }
22             HERE_IT_IS:
23                  /* Execution has reached this point in both parent and child */
24                  for(;;)
25                          ;
26        }
27
28        void f(int x)
29        {
30          char la[4000];
31                  la[3999]=x;
32        }
```

Consider the program above, which compiles and executes without any errors. Consider

the point in time where both parent and child process have reached the point marked `HERE_IT_IS`. Assume that:

- This is an X86-32 Linux system: int, long and pointers are 4 bytes and the page size is 4096 bytes.
- The compiler has not "optimized away" any operations. All variable accesses take place as written in the C code.
- The text, data, bss and stack regions were placed by the compiler at the virtual addresses shown in the partially completed diagram below. Any code which the C library ran before `main` consumed no more than one page of stack space. At entry to `main`, as shown, that one page from BFFFF000- BFFFFFFF is mapped in. The call to `fork` does not cause the stack to exceed the single page. However, the call to function `f` clearly does because of the large local variable `la`. Assume that only one additional page is required, i.e. during execution of `f`, the stack is greater than 4096 but less than 8192 bytes.
- The text and data regions are one page long and the bss region is two pages, as shown below.
- Global variables are assigned sequential increasing addresses within their respective regions based on the order in which they are declared in the C source code.
- Physical memory is plentiful and therefore the PFRA is not running.

Draw the abstract (disregarding the actual 2-level structure of the page table) Page Table Entries for the parent and child at the point under consideration, representing a Present PTE by an arrow leading from the virtual page to the physical page and annotated with the letters R, W and/or X representing the page protection bits of the PTE. Do not concern yourself with Access, Dirty, Supervisor or other PTE bits. A page in a virtual address space which is NOT Present is represented by the lack of a PTE edge. Note that you are not being asked to diagram the actual bytes within the physical pages, variable values or addresses (although you are free to do so if it helps you visualize the problem). To illustrate what is being asked, *one sample* PTE has been drawn *for the parent process only* mapping to one physical page (page frame). Represent each additional **unique** page frame with its own box under the "Page Frames" section, similar to how these diagrams were drawn in unit #5. Draw shared page frames, if any, by multiple PTE arrows pointing to the same PFN box. You need to complete the diagram for the rest of the parent's PTES and all of the child's PTEs!

For clarity, line numbers have been added to the source code listing and I will annotate the page frames with letters. The initial conditions given in the problem statement plus the "giveaway" of the initial PTE establish that the stack has been active prior to main (because of the C library start-up routines) but has never been deeper than one page. Of course, the text region is active by virtue of the program running. Its page of virtual address space was demand-paged in to page frame "A" and would have R-X PTE flags. At line 8, variable i which is in the data region is written to. The data page gets faulted in, flags RW-, frame B. At line 9, the variable a in the BSS region is read. Because this is only a read, and bss bytes are initialized to 0, the Linux kernel optimizes by linking this virtual page to the global shared page frame filled with 0 bytes, which I denoted "0". It sets up copy-on-write by setting the PTE flags to R--.

OK, now we fork. The virtual address space is duplicated into the child and initially, all page frames are shared with copy-on-write set up. For "A" in the text region, since write is never allowed, the page frame will remain shared. B and 0 are temporarily shared but

with PTE flags marked down to R--. The child at line 14 writes to a[20] which is in the first bss page. This breaks the sharing. A new page frame "C" must be allocated and copied from the 0 page frame, then the PTE in the child for address 0804A000 is RW-. At line 15, the child writes to its stack, which breaks the sharing of frame X and Y is allocated. The parent will also write to the stack by virtue of making a function call, and since we were in the first page of the stack, the PUSH of the return address, etc. takes place in that page, so the parent's PTE at BFFFF000 is clearly also RW-. It is not deterministic if the parent or child breaks the sharing of frame X first, so possibly the child is pointing to X and the parent to Y. This is inconsequential since the problem did not ask for this.

The parent at line 18 writes to its data page. The original page frame B gets copied to a new frame D and the parent's PTE is swung over to that and upgraded back to RW-. The child continues to use B. It is arguable if the child's PTE for 08049000 is R-- or RW- at this point. I would say the former, because the kernel is not going to do a reverse mapping search to find out that there is only one remaining PTE link to B. The parent at this same line is also reading a[5000] which is in the second page of BSS. A new PTE R-- is created in the parent only from 0804B000 to the shared 0 page. Finally, the parent has called function f which according to the problem statement grows the stack into the second page. The write to la[3999] at line 31 clearly writes to that page which starts at BFFFE000 and therefore the parent gets a new PF "Z" with RW- PTE.

Problem SA1 (10 points)

```
/* Assume this program is spawned with only fds 0,1 and 2 open and
 * the signal handling table at default values.
 * System pipe buffer size is 64K */
main()
{
  char buf[8192];
  int a[2],i;
  int status;
          pipe(a);  /* Assume this syscall succeeds */
          switch(fork())  /*        ditto                */
          {
            case 0:
                    for(i=0;i<65536;i++)
                              write(a[1],buf,8192);
                    return 0;
          }
          if (wait(&status)== -1)
                    perror("OMFG Wait failed!!!!");
          else
                    fprintf(stderr,"Child returned %d\n",status);
          return 0;
}
```

What does this program print? **Explain your answer** for full credit!

Nobody ever reads from the read side of the pipe. The pipe has a capacity of 64K as stated but the child tries to write 512K. (There is no SIGPIPE or EPIPE because the read side of the pipe is still open -- in both processes actually). The write system call blocks and the child never exits. The parent is stuck in the wait system call and never gets out of it since the child is stuck. Nothing prints.

Code Fragment MC#1

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
#include <setjmp.h>

jmp_buf jb;
int counter;

void handler(int sn)
{
         counter++;
         longjmp(jb,1);
}

main()
{
 struct sigaction sa;
 int ppid,i;
         sa.sa_handler=handler;
         sigemptyset(&sa.sa_mask);
         sa.sa_flags=0;
         sigaction(SIGINT,&sa,NULL);
         if (setjmp(jb)!=0)
         {
                  fprintf(stderr,"counter = %d\n",counter);
                  if (counter>=1000)
                  {
                           fprintf(stderr,"All done\n");
                           exit(0);
                  }
                  goto GOTO_IS_EVIL;
         }
         ppid=getpid();
         switch(fork())
         {
           case 0:
                  for(i=0;i<1000;i++)
                           kill(ppid,SIGINT);
                  exit(0);
         }
 GOTO_IS_EVIL:
         for(;;)
                  ;
}
```