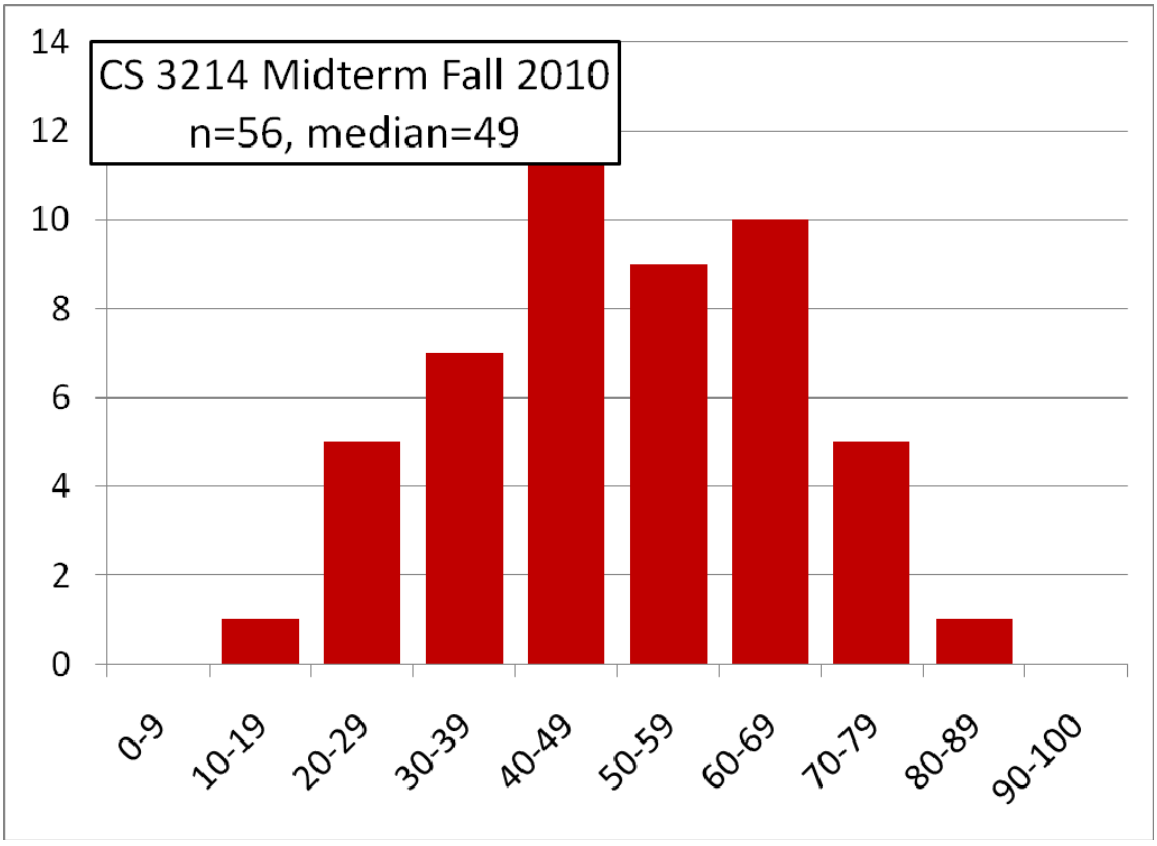
CS 3214 Midterm Solution

51 students took the midterm. The table below shows for each problem who graded it, as well as statistics about each problem. If you have a question about your score, please contact the person who graded the problem first before contacting your instructor.

Problem	1	2	3	4	Total
Min	3	0	0	2	10
Max	20	25	22	26	84
Median	9	15	12	15	49
Average	9.6	14.2	11.3	14.4	49.5
StDev	3.8	7.2	6.1	5.9	16.1
Grader	a),b) McQuain c) Puran	a) Puran b) Back	Xiaomo	a-c) Scott d) McQuain e) Puran	

Students who received a score of 40 or less are at risk of failing the class (even if the project minimum requirements are met) and must show improvement in the final exam as well as in the concurrent exercises.



Solutions are shown in this style.
Grading comments are shown in this style.

1. Compiling and Linking (25 pts)

a) (2 pts) *Prototypes.* During project 3, your teammate added a function `give_terminal_to()` to the esh.c file like so:

```
void give_terminal_to(pid_t pgrp, struct termios *pg_tty_state) {
    …
}
```

When you compiled the project, you saw:

```
$ make
cc -Wall -Werror -Wmissing-prototypes -g   -c -o esh.o esh.c
cc1: warnings being treated as errors
esh.c:196: warning: no previous prototype for ‘give_terminal_to’
make: *** [esh.o] Error 1
```

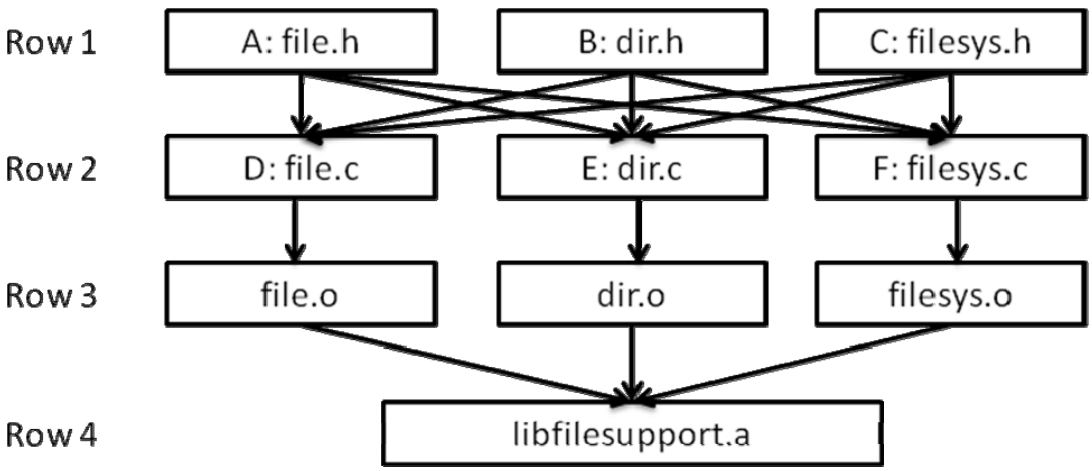
Your teammate then changed the code to be:

```
void give_terminal_to(pid_t pgrp, struct termios *pg_tty_state);
void give_terminal_to(pid_t pgrp, struct termios *pg_tty_state) {
   …
}
```

Discuss the merits of this idea! Address whether the code now compiles or not, and whether your teammate suggested the correct approach! If not, describe the approach that should be used!

The code compiles now, but the suggested fix is utterly bogus. Either the function should be defined static (if it is used only in esh.c), or else the prototype belongs in a header file such as esh.h

b) *Static Linking.* Consider a static library built from multiple .c and multiple .h files with the arrangement shown below:



i. (3 pts) Rows 1 and 2, 2 and 3, and 3 and 4 are connected by arrows that denote the relationship of these files during the build process. For each connection, name the tool that connects the files in each pairs of rows! You may use Unix shorthand names.

a. Rows 1 to 2

C preprocessor (cpp or built into cc1, or gcc -E)

b. Rows 2 to 3

The C compiler (cc1 or gcc)

c. Rows 3 to 4

The archiver (ar)

A lot of students made the mistake to name the linker (ld) for rows 3 to 4. The linker builds executables from object files (.o) and libraries (.a archives containing .o files).

ii. (18 pts) Consider the following C declarations and definitions. For each declaration or definition, list all possible locations where a programmer may legally place them so that any program using libfilesupport.a will link and function correctly! Use the letters shown in the figure, i.e., A for file.h, B for dir.h, and so on! If a declaration needs to appear, or may appear in multiple files, say "A and B" or "A or B" as appropriate. In addition, list those files where an experienced programmer following best practices would place those declarations or definitions!

Fill your answers into the table on the next page!

Declaration/Definition	Legal Placements	Best Practice
`struct file;`	A – F	Usually A, but could be repeated in B and C.

`struct file { // an open file`
`  struct list_elem elem;`
`  // link elem for open file list`
`  off_t pos;`
`  // current read/write offset`
`};`
	A – F	A
		
		(or D if struct is private to file.c)

`struct file *file_open (const char *name);`	A – F	A (prefix matches file name)
`static struct file * get_file_helper (int fd);`	A – F	D or E or F, as needed

`static off_t file_get_pos (struct file *f) {`
`  return f->pos;`
`}`
	A – F	A

`static struct list open_file_list;`	D or E or F	Likely D
`static int open_dir_count;`	D or E or F	Likely E
`extern struct disk *filesys_disk;`	A – F	Likely C
`struct disk *filesys_disk;`	A – F (weak/BSS symbol)	Likely F

The entries in this table are shown under the assumption that all object files of the libfilesupport.a library might be included in an executable the linker creates when linking with libfilesupport.a (recall that when linking with a static library, the linker will include only those object files from the library that provide definitions for symbols that are still in the 'unresolved' list at the time the library is processed).

Most entries could be in any file and still legally link and work because they either represent just declarations (which don't define symbols and thus don't cause linker conflicts) or represent definitions of weak symbols (e.g. filesys_disk). (Though some combinations may not compile, such as including file_get_pos's definition in multiple .h files that are included in the same .c file). The definitions

of static variables (open_file_list and open_dir_count) cannot be placed in header files because then the linker will create multiple copies of these variables.

Regarding best practice: a forward declaration such as 'struct file' would usually be found in file.h, but it's not uncommon to forward declare such a struct independently elsewhere. Its definition would be found in file file.h (or file.c if the struct is used only in file.c). file_open's declaration should go in file.h which declares the public functions file.c exports. file_get_pos() is likely best placed in file.h since it operates on struct file; placing it in file.h allows the compiler to inline the function. Static variables open_dir_count and open_file_list would likely be used in dir.c and file.c, respectively. filesys_disk's declaration should go in a header file (provided this variable is accessed elsewhere, which would be the only reason to have an extern declaration), and its definition should go in the .c file that logically owns it.

c) (2 pts) *Dynamic Linking.* You and a friend discuss the benefits and drawbacks of dynamic linking, specifically with position-independent shared object libraries. Your friend claims that programs that call dynamically linked functions generally run slightly slower than programs that are statically linked, even when one ignores the one-time cost of loading the dynamic library at runtime.
Is he or she correct? Justify your opinion!

Yes, this is correct. Even after the dynamic linking process, each invocation of a dynamically linked function involves an indirect jump through a pointer in the global offset table.

Additional notes: note that the question specifically asked about position-independent shared object libraries. It is also possible to relocate the executable and binary at run time (in essence, performing the same process that a regular linker does at link time when linking statically), but that will require updates to the process's and libraries text and data segments, thus preventing them from being shared across processes. For this reason, it's not commonly done.

Second, please do not misinterpret this loss of performance as a major concern. For most applications, the performance drawback is negligible – it is a concern only for high-performance applications that invoke functions in their most frequently executed code portions. Those are typically statically linked for this reason, if not inlined.

A lot of students pointed out that there is overhead when the library is dynamically loaded, but the question explicitly stated to ignore the one-time costs of dynamic loading.

2. Execution and Optimization (25 pts)

The following questions relate to how programs are compiled and optimized for IA32.

a) (12 pts) *Understanding IA32 Assembly Code.*
Consider the following function compiled using gcc –O2 –S:

```
pws3:
    pushl   %ebp
    movl    %esp, %ebp
    movl    12(%ebp), %ecx
    movl    16(%ebp), %edx
    leal    (%edx,%ecx), %eax
    imull   %ecx, %edx
    imull   8(%ebp), %eax
    popl    %ebp
    addl    %edx, %eax
    ret
```

Write a C version of this function!

This function computes the pairwise sum of its 3 integer arguments:

```
int
pws3(int a, int b, int c)
{
    return a * b + a * c + b * c;
}
```

We accepted any C function that returns the pairwise sum of its 3 int (or long) arguments.

b) (13 pts) *Optimizing Sparse Matrix-Vector Multiply*. Sparse matrices are those in which the majority of elements is zero. In many numerical applications, special techniques are used that avoid storing those zeros. For instance, the COO format stores the nonzero elements of a sparse matrix using tuples (row, column, value). These tuples may be stored in multiple one-dimensional arrays, one per component. For example, the matrix:

$$A = \begin{pmatrix} 1 & 0 & 0 & 2 \\ 5 & 1 & 0 & 0 \\ 0 & 0 & 7 & 0 \\ 0 & 0 & 4 & 1 \end{pmatrix}$$ would be stored as $$AA = (1 \quad 2 \quad 5 \quad 7 \quad 4 \quad 1 \quad 1)$$ $$ROWS = (0 \quad 0 \quad 1 \quad 2 \quad 3 \quad 1 \quad 3)$$ $$COLS = (0 \quad 3 \quad 0 \quad 2 \quad 2 \quad 1 \quad 3)$$

because A[0][0] = 1, A[0][3] = 2, and so on. The dimension of the matrix is given by N=4; the number of nonzeros is nnz=7 in this example. The following function accepts any matrix in COO format and computes the algebraic product of this matrix with a dense vector b:

```
double *
spmv(double *AA, int *ROWS, int *COLS, int nnz, double *b, int N)
{
    double * r = calloc(N, sizeof(double));
    int i = 0;
    for (i = 0; i < nnz; i++)
        r[ROWS[i]] += AA[i] * b[COLS[i]];

    return r;
}
```

This question examines the locality properties of this function.

i. (10 pts) Fill in the following table, describing what kind of locality you can expect for the memory accesses through the array variables used in this function during a single invocation. Use suitable terms to describe each variable's locality, such as stride, regularity/irregularity, and reuse!

	Spatial Locality	Temporal Locality
AA	Good – regular accesses with a stride of 1	Bad - No reuse
ROWS	Good – regular accesses with a stride of 1	Bad - No reuse
COLS	Good – regular accesses with a stride of 1	Bad - No reuse
b	Potentially irregular (depending on matrix structure and order of nonzeros);	Reuse depends on matrix structure – each b[i] is reused as many times as there are nonzero elements in column i
r	Potentially irregular (depending on the order in which nonzeros are stored)	Reuse depends on order of nonzeros

Some students interpreted the exhibited locality only when the function is run on the given example. In this case, the answer should be the same (or similar) to the one given above. Common mistakes included to discuss the locality of accesses to the pointer variables (rather than the memory these pointers refer to), and assuming that the function would be called multiple times (in which case AA, ROWS, and COLS are reused). The problem stated explicitly to consider only a single invocation. Some students interpreted the lack of reuse as good temporal locality. It's bad – caches pay off only when elements are reused.

ii. (3 pts) How could the locality of this function be improved?

Keeping the nonzero elements sorted by row will result in perfect temporal locality of the accesses to r[], and unless the matrix has empty rows, also perfect spatial locality with regular accesses with stride 1. Further sorting the elements within each row by column will improve b's spatial and temporal locality, depending on the structure of the matrix. Many matrices are banded in that the nonzero form bands that are parallel to the diagonal. (This sorting is commonly done in COO representation, though the code shown does not assume it, and the example provided showed ROWS/COLS not sorted.)
These appear to be the only obvious choices for COO. In general, it is difficult to improve the locality of sparse matrix operations. Interested students are referred to

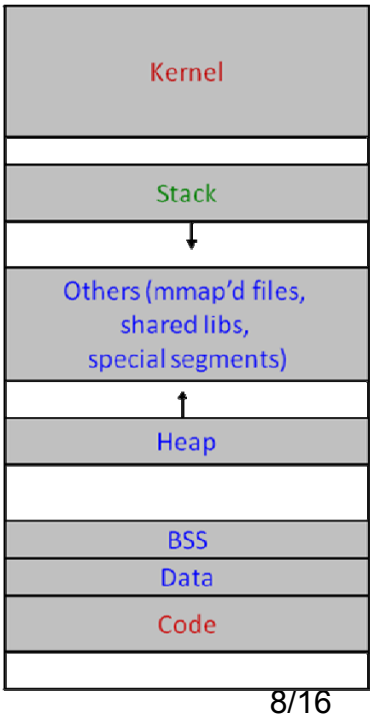
Belgin, M., Back, G., and Ribbens, C. J. 2009. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proceedings of the 23rd international Conference on Supercomputing* (Yorktown Heights, NY, USA, June 08 - 12, 2009). ICS '09. ACM, New York, NY, 100-109. DOI= http://doi.acm.org/10.1145/1542275.1542294

Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*(Reno, Nevada, November 10 - 16, 2007). SC '07. ACM, New York, NY, 1-12. DOI= http://doi.acm.org/10.1145/1362622.1362674

Some of you proposed techniques such as using local temp variables or the use of multiple accumulators. The use of local temp variables requires a different matrix representation that tracks where the nonzero of each row start (called CSR). The use of multiple accumulators may improve performance, but the question asked how to improve locality. Since multiple accumulators do not change the memory access pattern, locality is not affected.

3. Variables and Addresses (20 pts)

Address Space Layout in Windows. In lecture, we had discussed the address space layout of a typical IA32 process executing on Linux (shown on the right). In this question, you are asked to infer the address space layout of a process executing on 32-bit Windows XP. To that end, the following C program was compiled using the Cygwin GCC compiler and executed on a 32-bit Windows machine. (Microsoft's C compiler would produce similar results, except for printf()). Assume that printf is dynamically linked!




```
#include <stdio.h>
#include <stdlib.h>

int y = 42;
int *z;
```

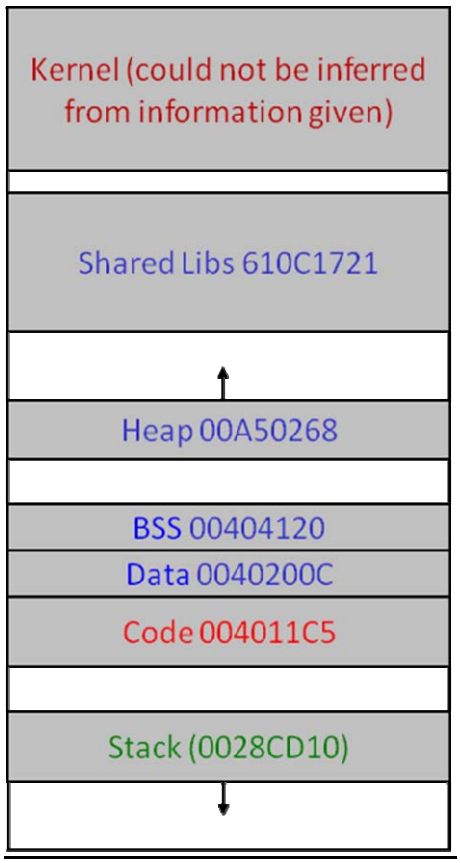
```
void *get_real_printf_address() {
    char * printf_code = (char *) printf;
    /* objdump -d shows printf as:
    00401310 <_printf>:
      401310:  ff 25 bc 50 40 00  jmp *0x4050bc
     */
    void ** printf_ptr_location = (void **) &printf_code[2];
    void ** printf_ptr = (void **) * printf_ptr_location;
    return * printf_ptr;
}

int main(int ac, char *av[]) {
    int x = 22;

    printf("&x\t= %08X\n", &x);
    printf("&y\t= %08X\n", &y);
    printf("&z\t= %08X\n", &z);
    z = malloc(200);
    printf("z\t= %08X\n", z);
    printf("main\t= %08X\n", main);
    printf("printf\t= %08X\n", get_real_printf_address());
}
```

This program produced this output:

```
$ ./where.exe
&x      = 0028CD10
&y      = 0040200C
&z      = 00404120
z       = 00A50268
main    = 004011C5
printf  = 610C1721
```

a) (6 pts) Based on this information, draw a sketch of the address space layout of a 32-bit Windows process!

Based on the information provided, the following relative locations of the different segments could be made out. (The addresses are shown for illustration only and were not required for full credit.)



An easy question if you understood which variables belong to which segment.

b) (14 pts) *Mystery GCC Option.* Consider this function discussed in lecture:

```
#include <stdio.h>

void echo(void)
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

When compiled with a certain option, gcc created this code:

```
echo:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $20, %esp
    movl    %gs:20, %eax
    movl    %eax, -8(%ebp)
    xorl    %eax, %eax          # clear canary from %eax
    leal    -12(%ebp), %ebx
    movl    %ebx, (%esp)
    call    gets
```

```
    movl    %ebx, (%esp)
    call    puts
    movl    -8(%ebp), %eax
    xorl    %gs:20, %eax
    je  .L3
    call    __stack_chk_fail
.L3:
    addl    $20, %esp
    popl    %ebx
    popl    %ebp
    ret
```

i. (8 pts) Examine this code and explain what this option does! You should explain both its purpose and how it accomplishes this purpose!

The option is `-fstack-protector-all`, see http://gcc.gnu.org/onlinedocs/gcc-4.2.3/gcc/Optimize-Options.html

GCC emits additional code in order to detect stack overflow attacks. Upon entering the procedure, it writes a canary value to -8(%ebp). Before the procedure returns, the value at -8(%ebp) is checked. If the value changed, the program will assume a stack overflow attack occurred and will call __stack_chk_fail, which then aborts the process. Here's what the user will see

```
$ ./stack
Type a string:a long string
a long string
*** stack smashing detected ***: ./stack terminated
Aborted
```

ii. (3 pts) Circle or underline all relevant instructions in the code above!

The added instructions are shown in red.

We didn't require that you circled the xor %eax, %eax instruction. It is used to clear the canary value from %eax so that it is not accidentally leaked.

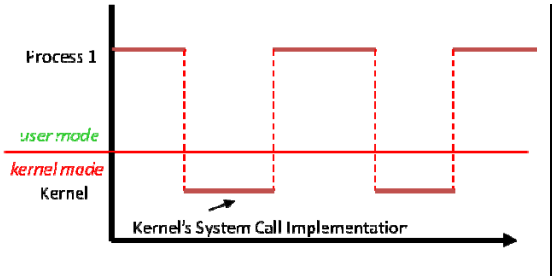
iii. (3 pts) For this code to be effective, what kind of value must be stored at address %gs:20?

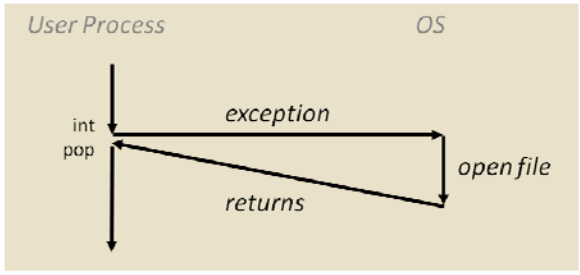
It must be a value an attacker cannot easily guess; typically, a random value is generated when a program starts. If the attacker knew the value, they could construct a stack overflow attack that leaves this value intact.

4. Processes in Unix (30 pts)

The following questions relate to how processes execute on Unix.

a) (4 pts) *System Calls.* The following two diagrams illustrate system calls. The first one is taken from the lecture slides, the second one from the text book.





Both illustrations show a user process and the OS/kernel, but the textbook authors (right) chose a different way of arranging the figure than the lecture slide (left). Provide a brief rationale for each approach at illustrating system calls!

i. Rationale for approach on left

The approach on the left shows the actual physical time line, illustrating that the execution of a process consists of user-mode and kernel-mode portions.

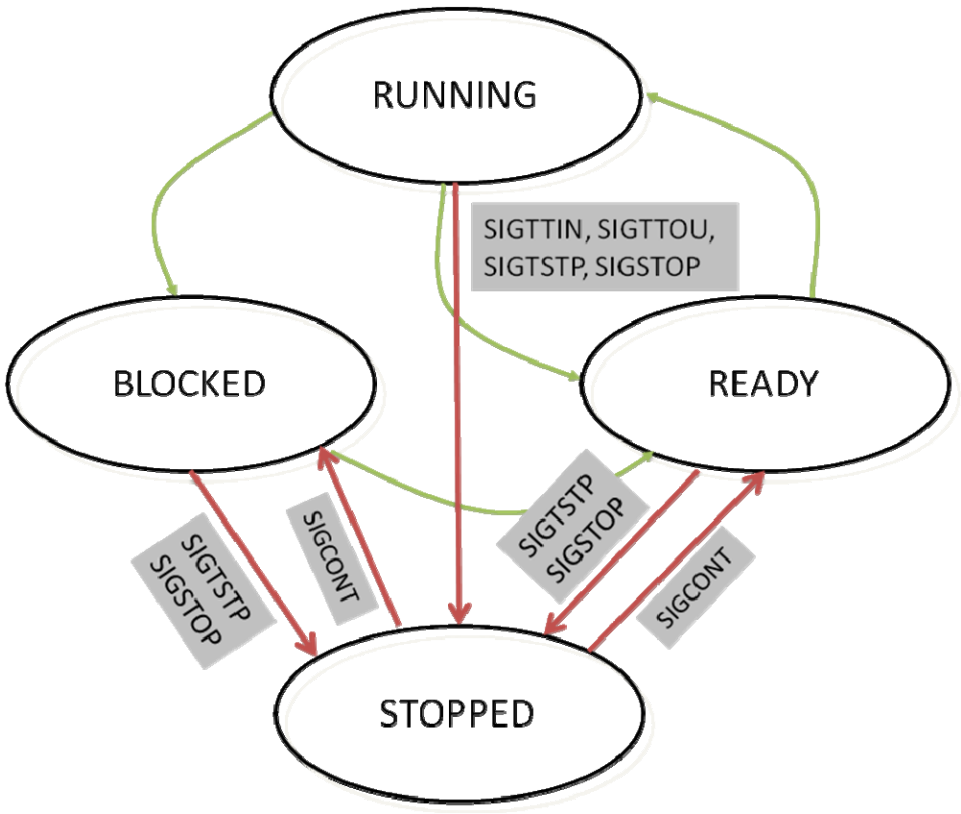
ii. Rationale for approach on right

The approach on the right illustrates that a user process's logical control flow continues directly after the system call, making it appear to the process as if a system call is simply a procedure call.

We awarded partial credit for pointing out relevant details shown in either figure, such as the fact that system calls are invoked via a trap/exception mechanism and involve a transition from user to kernel mode.

b) (4 pts) *Process States*. In lecture, we had discussed a simplified process state diagram consisting of the states RUNNING, READY, and BLOCKED. In Project 3, you saw that Linux processes can enter an additional state, STOPPED.

Complete the state diagram below and show which transitions to/from the STOPPED state are possible! Label each transition and describe which action or system call may cause the transition! If a transition can be caused by multiple reasons, provide at least 2! You do not need to label the transitions already shown.



The figure above shows which signals are involved in the transition. We also accepted if you provided the key strokes and/or shell commands that trigger the corresponding transitions. ^Z for SIGTSTP, needing exclusive access for SIGTTOU/SIGTTIN, issuing the 'stop' command for SIGSTOP, and issuing the 'fg' or 'bg' commands to send SIGCONT to stopped jobs to get them to continue.

It's not possible to transition from the stopped state directly into the running state. Note that SIGTTIN/SIGTTOU are caused by actions a process itself takes, so the process must have been running. On the other hand, SIGTSTP and SIGSTOP can be triggered by external events – the affected process may be RUNNING, READY, or BLOCKED.

Note that this diagram is still simplified – real OS use two separate states for STOPPED to remember whether a BLOCKED process was stopped or a READY/RUNNING process: say STOPPED/BLOCKED and STOPPED/READY. BLOCKED processes will transition to STOPPED/BLOCKED, and READY/RUNNING processes to STOPPED/READY. When asked to continue with SIGCONT, they'll return to the BLOCKED and READY states, respectively. The reason is that a user's decision to continue blocked processes cannot unblock them unless the event they were waiting for has also arrived, which is independent of the user's job control decisions. The arrival of the event while a stopped process is blocked would move the process from the STOPPED/BLOCKED state to the STOPPED/READY state.

You can easily observe process states and their relationship to job control using the 'ps' command. Processes that are marked 'R' are READY or RUNNING, those marked 'S' are BLOCKED (which Linux calls 'Sleeping'), and those STOPPED are marked 'T'.

c) (8 pts) *time*. The /usr/bin/time command can be used to determine how long a Unix command takes to execute as well as additional statistics about its execution. The command takes the name and command line arguments of an arbitrary program as input, and passes those command line arguments to the program it executes. For instance, below are examples that show how to time the execution of 'sleep 1.5' and 'echo Hello World'.

```
$ /usr/bin/time sleep 1.5
0.00user 0.00system 0:01.50elapsed (rest elided)
$ /usr/bin/time echo Hello World
Hello World
0.00user 0.00system 0:00.00elapsed (rest elided)
```

In this exercise, you should implement a simplified version of time which only prints the real (wall clock) time elapsed during the execution of any command. The output of your command shall be:

```
$ ./time sleep 1.5
Took 1 sec 502470 usec.
$ ./time echo Hello World
Hello World
Took 0 sec 960 usec.
```

Complete time.c below!

```
// time.c
#include <stdio.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/wait.h>

int
main(int ac, char *av[])
{
    struct timeval start, end, diff;
    gettimeofday(&start, NULL);

    /* Complete this part.  For simplicity,
       you do not need to show error checking. */

    if (!fork())
        execvp(av[1], av + 1);

    wait(NULL);

    gettimeofday(&end, NULL);
```

```
    timersub(&end, &start, &diff);
    printf("Took %ld sec %ld usec.\n",
            diff.tv_sec, diff.tv_usec);
    return 0;
}
```

In my opinion, this simple example shows the elegance of Thompson's and Ritchie's original design of Unix.

d) (4 pts) *File Descriptors*. In addition to redirecting stdout (file descriptor 1) and stdin (file descriptor 0), the bash shell and many other shells also support redirecting stderr (file descriptor 2). Many programs report error messages to stderr. In bash, this is possible using the syntax 2>&1, which redirects stderr to stdout. For instance,

```
$ bash -c "ls doesnotexist >error 2>&1"
```

produces a file named `error` with this content:

```
ls: doesnotexist: No such file or directory
```

If you examined a system call trace of bash and its children, which system calls would you find that implement this redirection functionality? Include both system calls to redirect stdout to `error` and to redirect stderr to stdout! Your solution should include the parameters given to those system calls, using variables as necessary!

open("error", …) = X
dup2(X, 1)
dup2(1, 2)

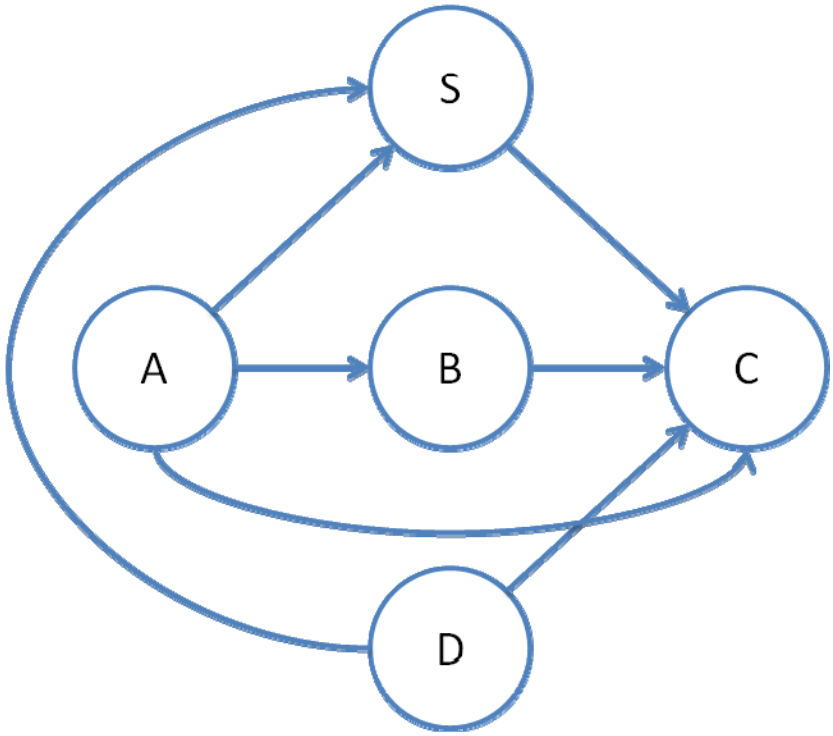
where 'X' is a file descriptor chosen by the OS.

e) (10 pts) *Concurrency.* Consider the following program:

```
void sigchld_handler(int s) {
    OUTPUT(S);
}

int main()
{
    signal(SIGCHLD, sigchld_handler);
    esh_signal_block(SIGCHLD);
    if (fork() != 0) {
        OUTPUT(A);
        esh_signal_unblock(SIGCHLD);
        OUTPUT(B);
        waitpid(-1, NULL, 0);
        OUTPUT(C);
    } else {
        OUTPUT(D);
```

```
    }
}
```

OUTPUT(X) outputs the character 'X'. esh_signal_block() and esh_signal_unblock() (implementation not shown) block and unblock SIGCHLD, respectively. In the chart below, connect two circles with an arrow if it is guaranteed that the character shown in the first circle is printed before the character shown in the second circle. For instance, an arrow from B → D would indicate that B is always printed before D.



A happens before B, which happens before C because of the textual order in the parent process. D happens before C since the parent will not progress past waitpid() unless the child has exited. D happens before S since SIGCHLD is not sent until after the child has exited. A happens before S because SIGCHLD is blocked while A is output, delaying SIGCHLD even if the child has already exited. S happens before C since even if the child exits after the parent has reached waitpid(), the signal will be delivered when waitpid() returns, before the process does anything else. A happens before C because of transitivity.

Any other arrows are wrong. There are no guarantees with respect to the order of A and D, B and D, or B and S.

We didn't deduct for not showing A→C as long as A→B and B→C were shown. Similarly, we didn't deduct if D→C wasn't shown as long as D→S and S→C were shown.