CS330: Operating Systems
Mid-semester Examination Solution Sketch

- Please be concise in your answers and write legibly.
- Do not write unnecessarily long answers. Longer answers usually lead to lower scores.
- Every answer must be accompanied by adequate explanation. Any answer (correct or incorrect) without explanation will not receive any credit.
- This is an open-book/open-note examination.

1. How many processes does the following C program create? Express your answer in terms of `N`. Explain your answer. **(5 points)**

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

int main(void)
{
        int i;

        for (i=0; i<N; i++) fork();

        return 0;
}
```

Solution: Let $f(i)$ be the number of processes that execute iteration `i`. Therefore, $f(i+1) = 2f(i)$ for all $i \geq 0$ with $f(0) = 1$. We need to calculate $f(N)$. It is straightforward to prove that $f(N) = 2^N$.

2.(a) The fork system call consumes a significant amount of time to execute because the parent needs to copy the valid portion of its address space into child's address space. Suppose the set of valid addresses of the parent process at the time of executing a fork system call is denoted by S. An address location is said to be valid if the parent process has used that address in some way before executing the fork system call. As part of the fork system call, the parent process copies the set S into child's address space. After executing the fork system call, among the addresses in S, the addresses that the parent process reads from are captured in the set R_P and the addresses that the parent process writes to are captured in the set W_P. Similarly, we define R_C and W_C for the child process. Therefore, $R_P \cup W_P = R_C \cup W_C = S$. Note, however, that a process can read from and write to the same address i.e., $R_P \cap W_P$ or $R_C \cap W_C$ may not be empty. Suppose at the time of executing the fork system call, the parent process is aware of the sets R_P, W_P, R_C, W_C. In that case, instead of copying the entire set S when executing the fork system call, is it possible to derive a subset of S, copying which would guarantee correct behavior of the fork system call? Exactly specify the minimal subset that suffices. Explain. **(8 points)**

Solution: For the child's correctness, we must copy all variables modified by the parent and read by the child. Symmetrically, all variables modified by the child and read by the parent must be copied. So, the minimal subset is $(W_P \cap R_C) \cup (W_C \cap R_P)$. If we did not have the knowledge of R_P and R_C, the minimal subset would be $W_P \cup W_C$. I have given full credit for this answer as well.

2.(b) Let us refer to the optimized implementation of the fork system call discussed in part (a) as optfork. Suppose that for a given program X which executes one fork system call, optfork carries out a total of N copy operations from the parent address space to the child's address space. In other words, the minimal subset has cardinality N for the program X. In realistic scenarios, the parent process does not know the sets R_P, W_P, R_C, W_C simply because it cannot see the accesses that will happen in future after the fork system call. Discuss a mechanism that can achieve N copy operations from the parent address space to the child's address space when X is executed assuming that the sets R_P, W_P, R_C, W_C are not known to the parent at the time of executing the fork system call. **(12 points)**

Solution: It is easy to achieve $N \leq |W_P \cup W_C|$. To start off, everything is shared between the parent and the child right after the fork. Whenever the parent or the child tries to write to a shared variable, a copy is created for the child. Achieving $N \leq |(W_P \cap R_C) \cup (W_C \cap R_P)|$ requires additional effort. In the solution described here, we will make sure that N achieves this bound in terms of copies done from the parent's space to the child's space. However, within each space there may be additional copies necessary. To start off, everything is shared between the parent and the child right after the fork. Whenever the parent tries to modify a shared variable, this variable is entered into a table T_P maintained in the parent space along with its current value. The variable is marked "tainted" at this time. Whenever the child tries to read a variable tainted by the parent, a copy is created for the child by looking up the table T_P. Similarly, whenever the child tries to modify a shared variable, this variable is entered into a table T_C maintained in the child space along with its current value. The variable is marked "tainted" at this time. Whenever the parent tries to read a variable tainted by the child, a copy is created for the child and old value of this variable is copied from table T_C into parent's space. Modifying a tainted variable leaves the variable tainted and nothing special needs to be done.

3.(a) In UNIX, it is recommended that a parent call `wait` for each of its children. What is the reason behind this recommendation? **(5 points)**

Solution: Not calling `wait` leaves the terminated child entries in the process table in zombie state. Eventually, the process table may get filled up with such zombie entries. The reason why the kernel cannot remove these entries even after the children have finished is that it does not know if the parent will call `wait` in future at some point. Calling `wait` allows the kernel to free the process table entry of a zombie child and avoids cluttering the process table. Please see the discussion in Section 7.4 of Chapter 7 of "The Design of the UNIX Operating System" by M. J. Bach for more details and an example.

3.(b) In certain programs, it may not be convenient or may not be necessary for the parent to call `wait` for each of its children. In such programs, how does the parent process inform the UNIX kernel that it will not call `wait`? How does the UNIX kernel use this information? **(4+1 points)**

Solution: The parent process can inform the kernel that it would like to ignore SIGCHLD by calling `signal(SIGCHLD, SIG_IGN)`. The kernel uses this information to clean up the process table entry of a child as soon as it terminates. Please see the discussion in Section 7.4 of Chapter 7 of "The Design of the UNIX Operating System" by M. J. Bach for more details and an example.

4.(a) In UNIX shared memory, it is recommended that there be a matching `shmdt` call for every `shmat` call.

What is the reason behind this recommendation? **(4 points)**

Solution: The `shmdt` call is responsible for decrementing the attach count. If the attach count does not drop to zero, a shared memory region cannot be freed even if a call to `shmctl` was made with `IPC_RMID` command. Therefore, the impact of not calling `shmdt` is that the global region table gets cluttered with idle entries (just like zombies in the process table).

4.(b). The `shmdt` call specifies the starting byte pointer to the shared memory region as an argument. Why doesn't it use the shared memory region descriptor as an argument instead? **(6 points)**

Solution: There are two reasons. First, a process may have multiple attach points for the same shared memory region and each `shmdt` call must correspond to one particular attach point. Second, the shared memory region descriptor may have been removed by a prior `shmctl` call by one of the sharing processes.

5. One of the steps in UNIX signal handling is to reset the signal handler array entry i before a process calls the handler for signal i. As a result, if this process wishes to catch signal i in future, it will have to register the handler in entry i through another call to the `signal()` function. Can you think of any reason why the signal handler array entry is reset before calling the signal handler function? (Hint: Giving the user an option to handle this signal through a different handler in future is not the correct answer.) Suggest changes to the way UNIX handles signals so that the signal handler array entry is not reset before calling the signal handler function, but the associated problem is also avoided. Remember that signal handler functions must execute in the user mode under all circumstances. **(3+7 points)**

Solution: If the handler array entry is not reset, there is a possibility of receiving another signal while executing the signal handler. If this keeps happening recursively, the process stack will keep growing due to nested calls to the signal handler (equivalent to an unbounded recursion) and eventually, the process will run out of virtual memory and get terminated. To avoid this problem, the early implementations of UNIX used to reset the handler array entry before executing the signal handler.

One possible way to avoid this problem is to buffer all signals received during the execution of a signal handler in an array separate from the usual signal array. So, these signals remain masked while a process executes a signal handler. At the end of the signal handler, these signals are moved to the actual signal array. In this procedure, the only tricky part is to know when the signal handler ends. Since the signal handler executes in the user mode, the kernel has no way to know when the signal handler ends. To resolve this, a small change needs to be done in the way the process user mode stack is set up by the kernel before invoking the signal handler. Currently, the stack is set up such that on returning from the signal handler, the process will be executing the code in the user mode where it left off. Instead, now the stack has to be set up such that after executing the signal handler, the process returns to a special function in the kernel mode which copies the buffered signals into the actual signal array. On returning from this special function, the process will be again executing in the user mode and now it will execute the usual user mode code where it was before handling the signal.

6.(a) What possible outputs do you expect from the following segment of C program running on UNIX? Explain. **(5 points)**

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <signal.h>
#include <sys/time.h>
#include <setjmp.h>
```

```
#define CHILD_STACK 16384

int i;
jmp_buf env;

int f (void *arg)
{
        longjmp(env, 2);
}

int main(void)
{
        void *child_stack = malloc(CHILD_STACK);
        i=setjmp(env);
        if (i==0) {
                clone (f, child_stack+CHILD_STACK, SIGCHLD, NULL);
                wait(NULL);
        }
        else i++;
        printf("%d, ", i);
        return 0;
}
```

Solution: 3, 0,

6.(b) What outputs do you expect to see with and without the `sleep(1)` statement in the following C program running on UNIX? Explain. **(5+5 points)**

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <signal.h>
#include <sys/time.h>
#include <setjmp.h>
#define CHILD_STACK 16384

int i;
jmp_buf env;

int f (void *arg)
{
        i++;
}
int main(void)
{
        void *child_stack = malloc(CHILD_STACK);
        i=setjmp(env);
```

```
        if (i==0) {
                clone (f, child_stack+CHILD_STACK, SIGCHLD | CLONE_VM, NULL);
                longjmp(env,2);
                printf("%d, ",i);
                wait(NULL);
        }
        else {
                i++;
        }
        sleep(1);
        printf("%d, ", i);
        return 0;
}
```

Solution: Several possibilities exist. However, if we assume that the parent keeps running until it needs to switch to a sleep state, it is easy to see that without the `sleep` statement, the outcome is 3. When the `sleep` is inserted, the outcome is 4. I will accept other solutions as well backed by proper reason.