cs 111 Operating Systems Principles, Winter 2011

Midterm *Solutions*

You have 110 minutes to complete this midterm.

Please make sure you have all 11 pages of the midterm.

Write your name on this cover sheet **and on any sheet you detach from the booklet**. Use the backs of the pages if you need to.

In order to receive full credit you must answer the questions precisely. Some questions may be harder than others. Read them all through first and attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Write down your reasoning if you're not sure; this will make it easier for us to give you partial credit.

OPEN BOOK, OPEN NOTES, NO COMPUTER, NO PHONES

I (xx/30)	II (xx/25)	III (xx/20)	IV (xx/25)	Total (xx/100)

Name:

System call reference (you may use others)

`pid_t fork()` — Create a new process. You may assume this always succeeds.
`void exit(int status)` — Exit with status `status`.
`pid_t getpid()` — Return current process's ID.
`pid_t getppid()` — Return parent process's ID.
`unsigned sleep(unsigned sec)` — Block for `sec` seconds (or a signal arrives).
`pid_t waitpid(pid_t p, int *s, int flags)` — Block until child process `p` exits, then store `p`'s exit status in `*s` and return `p`. Return `-1` if `p` doesn't exist, isn't a child, or another `waitpid` has already collected `p`'s status. If `flags == WNOHANG`, don't block: return `0` if `p` is still running. If `p == -1`, collect any child.
`int kill(pid_t p, int signo)` — Send signal `signo` to process `p`. `signo == SIGKILL` forces `p` to exit (with status `-256`).
`int simple_sigaction(int signo, void (*handler)(int))` — Install a signal handler `handler` for signal `signo`. Example `signo`s: `SIGCHLD` (a child process has exited), `SIGPIPE` (broken pipe—write to pipe with no readers).
`int pipe(int fd[2])` — Create a pipe: writes to `fd[1]` may be read from `fd[0]`.
`int close(int fd)` — Close a file descriptor.
`int dup2(int oldfd, int newfd)` — Make `newfd` a copy of `oldfd`.
`int execvp(const char *program_name, const char *argv[])` — Replace the current process image with a fresh execution of the given program with the specified arguments. Use the `PATH` environment variable to find the program binary.

I Epistemology

1 [**5 points**]. True or false?

A. T / [✓ F] The `getpid` system call can block.

B. [✓ T] / F The `read` system call can block.

C. [✓ T] / F The `write` system call can block.
In particular, `write` to a full pipe will block.

D. [✓ T] / F The `open` system call can block.

E. [✓ T] / F The `waitpid` system call can block.

2 [**5 points**]. True or false?

A. T / [✓ F] Timer interrupts are required to implement cooperative multitasking.
They are required to implement preemptive *multitasking.*

B. T / [✓ F] Timer interrupts are required to implement alternating parallelism.

C. T / [✓ F] A process receiving a signal is an instance of simultaneous parallelism.
The signal is more like an interrupt—it interrupts the process's normal code.

D. T / [✓ F] A Turing machine models simultaneous parallelism.
However, a multi-tape *Turing machine models simultaneous parallelism by having multiple heads and tapes.*

E. T / [✓ F] Race conditions cannot happen in a system with alternating parallelism.
Signals are a counterexample.

3 [**5 points**]. True or false?

A. T / [✓ F] A function call is a type of protected control transfer.

B. [✓ T] / F A system call is a type of protected control transfer.

C. T / [✓ F] A context switch is a type of protected control transfer.
Context switches can happen as a result of system calls or *for other reasons.*

D. T / [✓ F] `add` is an example of a dangerous instruction.

E. [✓ T] / F `lidt` is an example of a dangerous instruction.
`lidt` changes the IDT (interrupt descriptor table) register, which should be modified only by the kernel.

4 [**5 points**]. True or false (in a typical modern operating system)?

A. T / [✓ F] An isolated process can change any processor register.

B. [✓ **T**] / **F** An isolated process can change any memory in its stack.

C. **T** / [✓ **F**] An isolated process can change any memory in its address space (assuming there is enough physical memory).
As we saw in class, parts of the kernel also appear in a process's address space, and program instructions are often read-only.

D. [✓ **T**] / **F** An isolated process can change parts of its process descriptor using a system call.
Think `exit`.

E. [✓ **T**] / **F** An isolated process can change parts of its process descriptor using an `add` instruction.
Context switches save the process's registers in its descriptor; `add` changes the process's registers.

5 [**5 points**]. True or false?

A. [✓ **T**] / **F** If a process can write the kernel's memory, this violates a safety property of process isolation.

B. **T** / [✓ **F**] If a process can turn off timer interrupts, this violates a safety property of process isolation.
It violates a liveness property (that something good eventually happens).

C. **T** / [✓ **F**] Lowering context switch time lowers utilization.
It raises *utilization.*

D. [✓ **T**] / **F** Lowering context switch time does not affect robustness.

E. **T** / [✓ **F**] Protected control transfer requires a trap instruction (e.g. `int`).
As Microsoft found, an illegal instruction also works!

6 [**5 points**]. True or false?

A. [✓ **T**] / **F** WeensyOS 1 uses cooperative multitasking.

B. [✓ **T**] / **F** WeensyOS 1 processes have their own stacks.

C. **T** / [✓ **F**] WeensyOS 1 supports arbitrarily many simultaneously running processes.
It supports at most 15.

D. **T** / [✓ **F**] WeensyOS 1 processes are isolated.

E. [✓ **T**] / **F** WeensyOS 1 processes use memory-mapped I/O.
They print to the console via direct access to console memory.

II Debugging

7 [**10 points**]. WeensyOS 1's extra credit problem #7 asked you to "Introduce a `sys_kill(`*`pid`*`)` system call by which one process can force another process to exit." This system call should return 0 on success and −1 on error. This was *after* you changed `sys_wait` to block (probably using a wait queue). Here's an attempted implementation:

```
case INT_SYS_KILL: {
    pid_t p = current->p_registers.reg_eax;    // get process ID to kill
    proc_array[p].p_state = P_ZOMBIE;          // kill it
    proc_array[p].p_exit_status = 0;           // arbitrary exit status
    current->p_registers.reg_eax = 0;          // return value 0 indicates success
    run(current);                              // return to caller
}
```

There are at least four problems with this code. Briefly describe at least **two** of them.

1. *`p` is not bounds-checked; if the user passes a value < 0 or $\geq$ `NPROCS`, memory corruption will ensue.*
2. *The current state of `proc_array[p]` isn't checked. If that slot is empty, the system call should fail—or at the least the process's state shouldn't be turned to `P_ZOMBIE`.*
3. *If the process kills itself, it will run nevertheless (since the `run(current)` will run the process regardless of its state).*
4. *This code does not wake up the process's wait queue.*

8 [**10 points**]. Lab 1 asked you to implement pipes and command execution. Here's an attempted `command_exec` implementation. Its author, Eddie Dumbo, hasn't implemented reading from pipes yet (or `cd`, or `exit`, or file redirection).

```
static pid_t command_exec(command_t *cmd, int *pass_pipefd) {
    pid_t pid = -1;          // process ID for child
    int pipefd[2];           // file descriptors for this process's pipe
    // Create a pipe, if this command is the left-hand side of a pipe.
    // Return -1 if the pipe fails.
    if (cmd->controlop == CMD_PIPE) {
        if (pipe(pipefd) < 0) {
            perror("pipe");
            return -1;
        }
    }
    // Fork the child and execute the command in that child.
    pid = fork();
    if (pid == 0) {
        if (cmd->controlop == CMD_PIPE)
            pipefd[1] = STDOUT_FILENO;
        execvp(cmd->argv[0], cmd->argv);
    }
    // Return the child process ID.
    return pid;
}
```

There are at least four problems with this code when considering just normal command execution and writing to pipes. Briefly describe at least **two** of them.

1. *`execvp` isn't checked for errors. If the user enters an invalid command, the child will return to the command loop, leaving two copies of the shell process.*
2. *The assignment to `pipefd[1]` doesn't actually change the file descriptor table. Dumbo meant `dup2(pipefd[1], STDOUT_FILENO)`.*
3. *`pipefd[1]` isn't closed in the parent shell.*
4. *`pipefd[0]` isn't closed in the child.*
5. *`fork` isn't checked for errors (though this might be OK since our system call guide said you could assume `fork` doesn't fail).*

9 [**5 points**]. William Hazlitt is benchmarking an operating system with the following characteristics:

Context switch time	1 ms
System call time	0.25 ms for any system call
Timer interrupt frequency	100 Hz (that is, 100 intr/s, or 1 intr/10 ms)

He is running a pipeline `A | B` whose processes run this code:

```
A
while (1) {
    write(STDOUT_FILENO, "A", 1);
}
```

```
B
char buf;
while (1) {
    read(STDIN_FILENO, &buf, 1);
}
```

There are also N other processes, each running an infinite loop.

Mr. Hazlitt observes a write-to-read latency of 12.5 ms per character, and a throughput of 80 characters per second (= 1 character/12.5 ms).

What is N? Show your work (only partial credit for guessing the right number).

For explicitness we will work out this problem in detail.

The `A` process writes 1 character at a time to a pipe, and the `B` character reads 1 character at a time from the pipe.

We need to make some assumptions to make progress. Specifically, let's assume that the operating system runs processes in some fixed order, such as $ABX_1X_2 \ldots$ or $BX_1AX_2 \ldots$, where the X_is represent infinite loop processes. Without loss of generality, we can reorder the processes so that A comes first. Then any possible order has the form $AX_1 \ldots X_M BX_{M+1} \ldots X_N$, where M infinite loop processes run between A and B and $(N - M)$ infinite loop processes run between B and A.

We can also assume that A fills the pipe each time it runs. If the pipe's ring buffer has capacity c, it takes c system calls to fill the pipe. B will then empty the pipe with c system calls of its own.

Now we can analyze the schedule as follows. All parts of the schedule count towards throughput, but the latency measurement is between a single `write` *and the corresponding* `read`*: for latency, only the first half of the schedule counts.*

Action	***Time (ms)***	***Latency?***
A calls `writec`	$0.25c$	*Yes*
A context switch	1	*Yes*
M infinite loop processes, each running for a timer interrupt plus context switch	$(10+1)M$	*Yes*
B calls `readc`	$0.25c$	*No*
B context switch	1	*No*
$N - M$ infinite loop processes	$(10+1)(N-M)$	*No*
And, repeat.		

Adding up:

$$\textit{Latency} = 0.25c + 1 + 11M \textit{ ms}.$$

For throughput, we need to remember that the schedule writes c characters total.

$$\begin{aligned}\textit{Throughput} &= c/(0.25c + 1 + 11M + 0.25c + 1 + 11(N - M)) \\ &= c/(0.5c + 2 + 11N) \textit{ characters/ms}.\end{aligned}$$

Look at latency first, since that equation is simpler. We know that latency is 12.5 ms, so:

$$\begin{aligned}12.5 &= 0.25c + 1 + 11M, \textit{ so} \\ 11.5 &= 0.25c + 11M.\end{aligned}$$

It should be easy to see that $M = 1$, $c = 2$ is a solution. (There is another solution—$M = 0$, $c = 46$—but that solution does not make sense, because $c = 46$ system calls would take 11.5 *ms to execute, more than the timer interrupt frequency.)*

Plugging those results into throughput:

$$\begin{aligned}1/12.5 &= 2/(0.5 \times 2 + 2 + 11N) = 2/(3 + 11N), \textit{ so} \\ 3 + 11N &= 2 \times 12.5 = 25, \textit{ so} \\ N &= 2.\end{aligned}$$

For your information, this problem required you to solve Diophantine equations*: indeterminate equations where the unknown variables are constrained to be integers. Diophantine equations can be easy, as these were, or amazingly hard. For example, consider the well-known Diophantine equation $x^n + y^n = z^n$. Fermat's Last Theorem, one of the most famous problems in mathematics history, stated that no solutions for this equation existed with $n > 2$ (and x, y, z positive); it remained unproven for 358 years.*

III Free Processes

10 [**20 points**]. Bowie Cutlery has invented a new version of Unix he calls Punix. In Punix, process creation is so cheap that Bowie uses helper processes to implement other systems concepts!

Implement a user-level blocking mutual exclusion lock based on helper processes. (Such a lock might be useful for a multi-threaded process.) For full credit, **you must not use atomic instructions** like `compare_and_swap` or `swap`. However, for partial credit, simply define the mutex using such instructions.

You may assume that no external code calls `waitpid()`. Thus, helper process statuses will never be collected by other code. **You may also assume that process IDs are not reused.**

```
typedef struct {

    volatile pid_t p;

} mutex_t;


void mutex_init(mutex_t *m) {

    m->p = fork();
    if (m->p == 0) exit(0);

}
```

```
void mutex_acquire(mutex_t *m) {

    int s;
    while (waitpid(m->p, &s, 0) == -1) /* */;
    m->p = fork();
    while (m->p == 0) sleep(1000000);

}


void mutex_release(mutex_t *m) {

    kill(m->p, SIGKILL);

}
```

There are many solutions; ours, which uses helper processes in arguably the simplest way, is above.

Any blocking mutual exclusion lock requires some blocking operation. For processes, we know that `waitpid` can block when waiting for a child to exit.

Mutual exclusion also requires that at most one thread can have a lock in the acquired state at any instant. We therefore need a system call that gives at-most-once behavior. `read` from a pipe gives at-most-once behavior because it reads characters in first-in, first-out order without duplicates or omissions: if one character is in a pipe, and two processes or threads call `read` at the same time, only one of them will read the character. And you can do this problem with a pipe to the helper process. But `waitpid` also gives at-most-once behavior, because when a process dies, its status is collected at most once (the zombie is destroyed once the status is collected, so future `waitpid` attempts on that process ID will fail).

These insights lead to the lock above. The locked state is represented by a living helper process, the unlocked state by a zombie helper process. To release the lock we kill the helper process. To acquire the lock, we use `waitpid` to try to collect the helper process's state. If the *helper is a zombie (the lock is unlocked), `waitpid` will succeed; we create a new helper process to represent the locked state. If multiple threads try to acquire a lock at once, only one of them will succeed (`waitpid` will return `m->p`); the rest of them will return an error since the zombie has been destroyed. The error indicates that some other thread grabbed the lock and we should retry. Finally, to initialize the lock, we create a thread that immediately exits, since the lock should start out in the unlocked state.*

IV Zombie Prevention

George Romero doesn't like zombie processes. He suggests that we replace the `fork` system call with a new system call, `fork_status`:

`pid_t fork_status(int *status)` — Create a new process. When the child process exits, its exit status is stored automatically in the parent's `*status`. The child process descriptor is then totally destroyed (the child does not become a zombie).

George also suggests that `fork_status` makes it possible to implement `waitpid` entirely at user level, with no kernel support.

11 [**10 points**]. Implement user-level functions `zfork` and `zwaitpid_nohang` that behave respectively like `fork` and `waitpid` with `WNOHANG` (that is, polling `waitpid`), but are based on `fork_status`.

You may use other system calls (but not `fork` and `waitpid`). **You will need to define a global data structure** to store child process statuses. You may assume that processes are single threaded (i.e. no synchronization problems), that `INT_MIN` is never a valid exit status, that the process never runs out of memory, and that `fork_status` always succeeds. Handle other error conditions by returning `-1`.

We'll write out full code for the global data structure, but also accepted linked list pseudocode, or (for example) an array plus the assumption that there are never more than N active children at a time. We take advantage of the infinite memory available by never freeing state.

Global data structure:

```
typedef struct node {
    struct node *next;
    pid_t p;
    int status;
} node_t;
node_t *children;
```

```
pid_t zfork() {

    node_t *n = (node_t *) malloc(sizeof(node_t));
    n->status = INT_MIN;
    n->p = fork_status(&n->status);
    n->next = children;
    // add child---but the new child has no children yet!
    children = (n->p == 0 ? NULL : n);
    return n->p;

}

pid_t zwaitpid_nohang(pid_t p, int *s) {
    assert((p == -1 || p > 0) && s != NULL);

    node_t *n = children;
    while (n) {
        if (n->p == p && n->status == INT_MIN)
            // asked for specific child that hasn't exited yet
            return 0;
        else if (n->p == p || (p == -1 && n->status != INT_MIN)) {
            // collect "zombie"
            p = n->p;             // remember pid in case p == -1
            *s = n->status;
            n->status = INT_MIN;  // clear node
            n->p = 0;
            return p;
        } else
            n = n->next;
    }
    // no child has exited yet, or specific child doesn't exist
    return p == -1 ? 0 : -1;

}
```

12 [**5 points**]. Implement a user-level function `zwaitpid` that emulates `waitpid` using `zwaitpid_nohang`. Your emulated version must return the same values as a true `waitpid` implementation—for instance, if called with `flags == 0`, it must not return until process `p` exits—but might or might not have different performance characteristics and utilization. **No race conditions allowed.**

```
pid_t zwaitpid(pid_t p, int *s, int flags) {
    assert((p == -1 || p > 0) && s != NULL && (flags == 0 || flags == WNOHANG));

    while (1) {
        pid_t x = zwaitpid_nohang(p, s);
        if (x != 0 || flags == WNOHANG) return x;
    }

}
```

For what it's worth, this problem didn't require that you got problem 11 right, or that you understood `fork_status`; all you needed was to understand the signature and behavior of `zwaitpid_nohang`.

13 [**5 points**]. Implement a user-level function `risky_zwaitpid` that emulates `waitpid` using `zwaitpid_nohang`. Your emulated version must return the same values as a true `waitpid` implementation **and must have similar performance characteristics and utilization.** However, **your code may contain a race condition** involving signals. (You may assume `zwaitpid_nohang` has no internal performance problems.)

```
pid_t risky_zwaitpid(pid_t p, int *s, int flags) {
    assert((p == -1 || p > 0) && s != NULL && (flags == 0 || flags == WNOHANG));

    while (1) {
        pid_t x = zwaitpid_nohang(p, s);
        if (x != 0 || flags == WNOHANG) return x;
        sleep(100000);
    }

}
```

Our solution assumes that dying processes send `SIGCHLD` to their parents as usual, and that the parent has not ignored `SIGCHLD`. The `SIGCHLD` will interrupt the `sleep` system call. The "critical section" is the body of the `while` loop, except for the `return` statement.

14 [**5 points**]. If your solution to Problem 13 contains a race condition, return to your code and circle the minimal "critical section": the smallest set of C statements where, if those statements are never interrupted, there will be no race condition. (If you circle a system call, that only includes the user-level instructions that prepare for and make the system call, not the kernel's system call implementation.) If it does not contain a race condition, write "no race" below and explain why not on the reverse.