School of ECE, UNIST
CSE251 (System Programming)
Spring 2019

Instructor:
Hyungon Moon

Final, 8:00 - 10:00 (120min) Jun 13, 2019

Name (In English):	
Student ID:	

Instruction:

1. Submit all the sheets.
2. Write your student ID at every page, bottom left.
3. Your answers should be printed at the designated locations, and written with an **inerasable black or blue** pen.
4. You are not allowed to go to the rest room and come back.

Question	Points	Score
1	18	
2	10	
3	16	
4	18	
5	18	
6	20	
Total:	100	

Enjoy!

1. **General Topics**

Answer short questions.

(a) [14 points] State if each statement is true or false (2pt each).

(1) Some memory bugs may not result in Segmentation Fault. (**True**)

(2) When a program accesses a virtual memory page, the OS kernel checks the access to determine if it results in page fault or not. (**False**)

(3) When a program tries to divide an integer by zero, which is impossible, it invokes the OS kernel to handle the exception. (**False**)

(4) When a program needs to use OS-provided services such as process creation or file writes, the program invokes the OS kernel. (**True**)

(5) Once a program setup a potential destination with `setjmp`, it can jump to the program point whenever it needs with `longjmp`. (**False**)

(6) A Linux program can communicate with another process either on the same or different machine using a socket. It is a file descriptor that a processes can invoke `read` or `write` system calls to interact with. (**True**)

(7) A web server can handle only one connection per each port. (**False**)

(b) [4 points] Modern computer systems typically use multi-level page tables instead of single-level ones, even though it increases TLB miss penalty. What is the benefit of having such multi-level tables? Explain with some quantitative numbers assuming page size of 4kB and virtual address space size of 4GB, and a process using only one 4kB page.

Solution:
By using multi-level tables, we can significantly reduce the amount of memory required for page tables. If all pages are 4kB and we have 4GB of virtual address space, there are 2^20 to be translated, each requires 4 bytes. As a result, one process needs 2^24B, or 16MB just for the page tables, which is huge. My using multi-level tables, for a process using only one virtual page, we only need to allocate one table at each level, which is typically much smaller than 16MB.
Criteria:

- Mentioning the benefit of memory sizes: 1pt.
- Correct discussion with numbers: 3pt.

2. [10 points] **Threaded Echo Server**

   Identify and explain the two potential memory leaks found in this program, and describe how we can fix with the corresponding line numbers.

```
 1 int echo(int);
 2 void* thread(void *vargp) {
 3     int connfd = *((int *)vargp);
 4 
 5     // echo the incoming bytes back to the client.
 6     echo(connfd);
 7     close(connfd);
 8     return NULL;
 9 }
10 
11 int main(int argc, char ** argv) {
12     int listenfd, *connfdp;
13     socklen_t clientlen;
14     struct sockaddr_storage clientaddr;
15     pthread_t tid;
16     listenfd = Open_listenfd(); // opens a listening socket for us.
17     while(1) {
18         clientlen = sizeof(struct sockaddr_storage);
19         connfdp = malloc(sizeof(int));
20         *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
21         pthread_create(&tid, NULL, thread, connfdp);
22     }
23 }
```

Solution:

- The memory allocated by the master thread at line 19 is never freed. To fix, one way is to add the following line between line 3 and 4.

```
free(vargp);
```

- The master thread, which is running an infinite loop, is not reaping its children, resulting in memory leak. A possible fix is to detach the thread as the master is not using the return values. To do so, we can add this line to between line 3 and 4.

```
pthread_detach(pthread_self());
```

Criteria: 5 points each, 2 for identify and 3 for fix.

3. Exceptional Control Flow

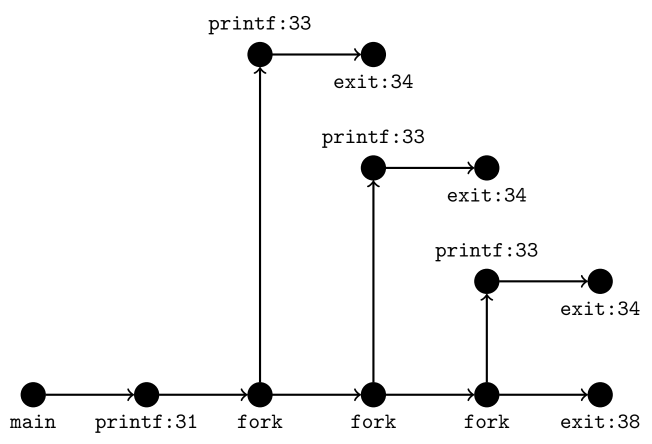
Answer the questions about the following program that creates and reaps child processes.

```c
#include<errno.h>
#include <stdio.h>
#include <sys/wait.h>
#include <string.h>
#define N 3

const char* reaped = "Reaped!\n";
const char* err = "Error\n";

int exited;
void sigchld_handler(int sig) {
    int olderrno = errno;
    if ((waitpid(-1, NULL, 0)) < 0) {
        write(1,err,strlen(err));
    }
    write(1,reaped,strlen(reaped));
    exited += 1;
}

int main(void) {
    int i = 0;
    struct sigaction action, action_old;
    pid_t pid;
    exited = 0;
    memset(&action,0,sizeof(struct sigaction));
    action.sa_handler = sigchld_handler;
    sigemptyset(&action.sa_mask);
    sigaction(SIGCHLD,&action,&action_old);

    for(i = 0; i < N; i+= 1) {
        printf("Forking: %d\n",i);
        if((pid = fork()) == 0) {
            printf("child: %d\n",i);
            exit(100+i);
        }
    }
    int prev = exited;
    while(exited < N);
    exit(0);
}
```

Question 3 continues on the next page. . .

(a) [8 points] Depict the process graph that shows the partial ordering of program statements. Represent all and only the `printf`, `fork`, `exit` and the beginning of `main` functions as the nodes in the graph. To distinguish multiple calls from each other, append the line number at the end, e.g., `printf:31`.

Solution:



(b) [8 points] The program may or may not terminate. Explain why and describe how we can fix it with the corresponding code lines to be changed.

Solution:
The handler in the program reaps only one child when it is called. This may leave some zombie children because multiple exiting child may call the handler only once, as signals are not queued.
As a fix, we should replace the signal handler as follows.

```c
void sigchld_handler(int sig) {
    while ((waitpid(-1, NULL, 0)) > 0) {
        exited += 1;
    }
}
```

4. **Virtual Address Translation**

In this question, you are going to present the steps of address translation, with these assumptions.

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not 4-byte words).
- Virtual addresses are **12** bits wide.
- Physical addresses are **9** bits wide.
- The page size is **32** bytes.
- The TLB is **4-way** set associative with **8** total entries.

For all subproblems, assume the TLB looks like this at the beginning:

Set	Tag	PPN	Valid	Tag	PPN	Valid
0	0x14	0x6	1	0x10	0x3	1
1	0x11	0x1	1	0x12	0x5	1

This is a part of page table. All the numbers are hexadecimal, but the prefix `0x` is omitted for space reasons.

VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid
00	b	1	10	c	1	20	3	1
01	-	0	11	-	0	21	-	0
02	3	1	12	d	1	22	3	1
03	-	0	13	-	0	23	1	1
04	7	1	14	1	1	24	7	1
05	-	0	15	-	0	25	5	1
06	9	1	16	4	1	26	9	1
07	0	0	17	f	0	27	0	0
08	-	0	18	-	0	28	6	1
09	-	0	19	-	0	29	1	1
0a	6	1	1a	2	1	2a	0	1
0b	7	1	1b	3	1	2b	1	1
0c	8	1	1c	4	1	2c	2	1
0d	9	1	1d	5	1	2d	3	1
0e	a	1	1e	6	1	2e	4	1
0f	b	1	1f	7	1	2f	5	1

For all questions (on the next page), either answer with an hexadecimal number (e.g., `0x04`) or the letter `Y` or `N` if instructed. If there is a field whose value is unknown from the given information, fill it with `Unknown`.

Your task is to fill out the tables. Each entry gives you 1 point.

Question 4 continues on the next page...

(a) [6 points] Virtual address: 0x4ac.

Parameter	Value
TLBI	**0x1**
TLBT	**0x12**
VPN	**0x25**
TLB hit (Y/N)	**Y**
Page Fault (Y/N)	**N**
PPN	**0x5**

(b) [6 points] Virtual address: is 0x4c9.

Parameter	Value
TLBI	**0x0**
TLBT	**0x13**
VPN	**0x26**
TLB hit (Y/N)	**N**
Page Fault (Y/N)	**N**
PPN	**0x9**

(c) [6 points] Virtual address: 0x139.

Parameter	Value
TLBI	**0x1**
TLBT	**0x04**
VPN	**0x09**
TLB hit (Y/N)	**N**
Page Fault (Y/N)	**Y**
PPN	**Unknown**

5. **Dynamic Memory Allocation**

Answer these questions about heap implementations. Assume the word size is 4 bytes and the size of address is 4 bytes.

(a) [6 points] Compared to the implicit free list, explicit free list requires some chunks to have more metadata. Explain why the additional metadata does not increase the internal fragmentation if the size of smallest chunk (including all metadata) is 32 bytes.

Solution: Unlike the size metadata, the pointers for explicit free list are not required for the allocated block. If the smallest memory chunk is 32 bytes, a free chunk always has at least 16 bytes of free memory block which is supposed to be used by the calling program when it is allocated. By reusing the free space, a `malloc` implementation can compose free lists without additional space overhead.

(b) [6 points] Explain why it is reasonable to avoid creating or maintaining smaller chunks. (e.g., smaller than 32 bytes). In other words, explain why it is inefficient to allocate small dynamic memory.

Solution: For each chunk of memory a `malloc` implementation should maintain some metadata, whose size remains constant even if the chunks is small. If a program allocates many small chunks, the ratio of metadata to the in-use memory becomes high, and as a result, the utilization will become low.

(c) [6 points] In class, we discussed the reasons why it is hard and expansive to implement garbage collection for languages like C/C++. State and explain one of the reasons.

Solution:
The key idea of possible answers:

- GC needs to distinguish the pointers from the other data, which is not easy in C/C++ in which casting between pointers and data is allowed.
- Given a pointer, GC needs to determine the exact base address and the size of heap chunk enclosing the pointer, which is not easy with typical memory allocators for C/C++.

6. **Synchronization and Parallelism**

Answer the questions about this problematic implementation of a parallel addition toy example. The program is supposed to print the sumation of all numbers from 1 through 1000000. Answer the questions on the next page.

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <errno.h>
long gsum = 0;
long num_integers = (1000 * 1000);
long num_integers_per_thread;
sem_t mutex;

void* thread(void* vargp) {
    long myid = *((long*)vargp);
    long start = myid * num_integers_per_thread + 1;
    long end = start + num_integers_per_thread;
    long i;
    for(i = start; i < end; i+=1) {
        sem_wait(&mutex);
        gsum += i;
        sem_post(&mutex);
    }
    return NULL;
}

int main(int argc, char** argv) {
    size_t i = 0;
    assert(argc == 2);
    long num_threads = atoi(argv[1]);
    pthread_t* tid = malloc(sizeof(pthread_t)*num_threads);
    long* myid = malloc(sizeof(long)*num_threads);
    sem_init(&mutex, 0, 1);
    num_integers_per_thread = num_integers / num_threads;
    for(i = 0; i < num_threads; i+=1) {
        myid[i] = i;
        pthread_create(&tid[i], NULL, thread, &myid[i]);
    }
    for(i = 0; i < num_integers % num_threads; i += 1) {
        gsum += num_integers - i;
    }

    printf("result: %ld\n",gsum);

    return 0;
}
```

Question 6 continues on the next page...

(a) [8 points] The developer wanted to print the result at line 41, but it won't. Explain why it doesn't and how to fix it, by describing the code that should be added, modified or deleted with the corresponding line numbers.

Solution: The master (main) thread does not wait for the completion of the worker threads, which results in race at the variable `gsum`. A possible fix is to add the following code snippet to between line 36 and line 37. If added after 37, the program again suffers from the race.

```
for (i = 0; i < num_threads; i+= 1) {
    pthread_join(tid[i],NULL);
}
```

Criteria: 4pt for problem, 4pt for fix.

(b) [8 points] Another problem is in its performance. Even assuming that the cost of single addition operation is very expensive and we can ignore the cost of initializing processes or creating threads, the execution time of the program does not become shorter as we give it more threads. Explain why, and describe the fix with the corresponding line numbers.

Solution: The addition operation at each thread cannot be executed simultaneously because they all use one semaphore to avoid race at `gsum`. To avoid such a synchronization, an option is to perform the additions locally at each thread and let the master thread to collect them. For example, we can replace the line from 17 through 21 with this to calculate the sums locally.

```
for(i = start; i < end; i+=1) {
    mysum += i;
}
*((long*)vargp) = mysum;
```

At master thread, between line 36 and 37, after the code we added earlier, we can insert this.

```
for(i = 0; i < num_threads; i+= 1) {
    gsum += myidsum[i];
}
```

(c) [4 points] Assume that when we run the program with one thread, it takes 200μs for thread function and the total execution time is 1ms. What is the lowerbound of the execution time we have 10 and 500 cores and run 1 thread on each core? Fill out the table and describe the way you derived the answers.

Cores	1	10	500
Execution Time	1ms	**0.82ms**	**0.8004ms**

Question 6 continues on the next page...

Solution: Use the Amdahl's Law. For 10 cores case,
$\frac{0.2}{10} + 0.8 = 0.82$
For 500 cores case,
$\frac{0.2}{500} + 0.8 = 0.8004$
Criteria:

- 1 pt per blank.
- 1 pt per explanation.

The end of questions.