CSCI 4061: Introduction to Operating Systems
Fall 2024
Midterm Exam 2 – Practice Questions

1. **Virtual Memory**

   Assume we have a process running on a computer with 8-bit memory addresses and a virtual memory system featuring 32-byte pages. The page table for this process is as follows. Recall that pages have specific permission settings which determine if read (R), write (W), or execute (X) operations are allowed on data contained in this page.

Page #	**Valid?** `0/1`	**Frame #**	**Permissions**
`111`	`1`	`010`	R, W
`110`	`1`	`100`	R, W
`101`	`0`	`Null`	N/A
`100`	`0`	`Null`	N/A
`011`	`0`	On Disk	R
`010`	`1`	`011`	R, W
`001`	`1`	`111`	R, X
`000`	`1`	`110`	R, X

   (a) What is the benefit of allowing some pages, such as page `011` in the table above, to reside on disk rather than in physical memory?

   (b) Which portion of the process's virtual address space (stack, heap, data, or code) is most likely stored in virtual pages `111` and `110`? Explain why.

(c) What is the number of offset bits used in memory address on this computer? Explain.

(d) Consider each of the following operations involving virtual addresses. For each operation:

- If the operation is valid and can proceed without issue, translate the virtual memory address to the corresponding physical memory address.
- If the operation references a page on disk, write "Page Fault".
- If the operation references an address that is not part of an allocated page, write "Segmentation Fault".
- If the operation violates a page's permission settings, write "Protection Fault".

i. Write to address `11001110`

ii. Write to address `00110100`

iii. Read from address `00010100`

iv. Execute instruction at address `00100100`

v. Read from address `01100000`

vi. Write to address `01011100`

2. **Signals**

(a) Consider the following C program, which installs a handler for the `SIGINT` signal.

```
#include <stdio.h>
#include <signal.h>
#include <sys/stat.h>
#include <unistd.h>

#define BUFSIZE 512

void handle_signal(int signo) {
    struct stat statbuf;
    if (stat("test.txt", &statbuf) == -1) {
        perror("Failed to stat test.txt");
    } else {
        printf("test.txt size: %ld\n", statbuf.st_size);
    }
}

int main() {
    struct sigaction sact;
    sact.sa_handler = handle_signal;
    sact.sa_flags = SA_RESTART;
    if (sigfillset(&sact.sa_mask) == -1) {
        perror("sigfillset");
        return 1;
    }
    if (sigaction(SIGINT, &sact, NULL) == -1) {
        perror("sigaction");
        return 1;
    }

    char buf[BUFSIZE];
    if (getcwd(buf, BUFSIZE) == NULL) {
        perror("getcwd");
        return 1;
    }
    printf("Current directory is: %s\n", buf);
    return 0;
}
```

i. What is one problem that may occur with this code due to the implementation of the signal handler above? Explain a sequence of events that induces this problem.

ii. What is a second problem that may occur with this code due to the implementation of the signal handler above? Explain a sequence of events that induces this problem.

(b) Consider the following C code, which installs a handler for the `SIGINT` signal.

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int x = 0;
int y = 0;

void handle_signal(int signo) {
    x = 4;
    y = 8;
}

int main() {
    struct sigaction sact;
    sact.sa_handler = handle_signal;
    sact.sa_flags = SA_RESTART;
    if (sigfillset(&sact.sa_mask) == -1) {
        perror("sigfillset");
        return 1;
    }
    if (sigaction(SIGINT, &sact, NULL) == -1) {
        perror("sigaction");
        return 1;
    }

    x = 15;
    y = 16;
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

i. What is one set of values for `x` and `y` that may be printed out by the program? What sequence of events causes these values to be printed?

ii. What is a second set of values for `x` and `y` that may be printed out by the program? What sequence of events causes these values to be printed? If there is no second possibility, simply write "N/A" below.

iii. What is a third set of values for `x` and `y` that may be printed out by the program? What sequence of events causes these values to be printed? If there is no third possibility, simply write "N/A" below.

3. IPC With Pipes

You decide you'd like to analyze how quickly new processes get created and scheduled to run by the operating system. Timestamps on Posix systems are represented by `struct timespec` values and we use the `clock_gettime` function to obtain the current time.

For example, if we want to store the current time in a `struct timespec` variable `ts`, we can write the following code:

```
struct timespec ts;
if (clock_gettime(CLOCK_REALTIME, &ts) == -1) {
    perror("clock_gettime");
}
```

Your task is to implement the following function to measure the start times of `n` new processes:

```
int measure_start_times(struct timespec *timestamps, unsigned n)
```

The parameters for the function are as follows:

- `struct timespec *timestamps`: The starting address of an array of `struct timespec` elements. You may assume this memory has been properly allocated before this function is called.
- `n`: The number of process start time measurements to collect (and the length of the array referenced by the `timestamps` parameter).

The `measure_start_times` function should start `n` unique child processes. The first action each child takes is obtaining the current system time by calling `clock_gettime`. Then, this timestamp should be reported back to the parent using a pipe.

The `measure_start_times` function returns `0` on success or `-1` on error. If the function succeeds, then the array referenced by the `timestamps` parameter should be filled with the `n` timestamp values obtained by the `n` child processes.

Your solution should not collect any data from child processes until all children have been started, allowing child processes to execute in parallel.

Your solution should perform proper error handling for the `clock_gettime` function but does not need to check any other function calls for errors. If any child process encounters an error, then `measure_start_times` should indicate an error by returning `-1`.

WRITE YOUR SOLUTION ON THE NEXT PAGE ⟶

```
int measure_start_times(struct timespec *timestamps, unsigned n) {

}
```

4. **Shared Memory**

Note: This is identical to the previous question except that you will need to use shared memory rather than pipes.

You decide you'd like to analyze how quickly new processes get created and scheduled to run by the operating system. Timestamps on Posix systems are represented by `struct timespec` values and we use the `clock_gettime` function to obtain the current time.

For example, if we want to store the current time in a `struct timespec` variable `ts`, we can write the following code:

```
struct timespec ts;
if (clock_gettime(CLOCK_REALTIME, &ts) == -1) {
    perror("clock_gettime");
}
```

Your task is to implement the following function to measure the start times of `n` new processes:

```
int measure_start_times(struct timespec *timestamps, unsigned n)
```

The parameters for the function are as follows:

- `struct timespec *timestamps`: The starting address of an array of `struct timespec` elements. You may assume this memory has been properly allocated before this function is called.
- `n`: The number of process start time measurements to collect (and the length of the array referenced by the `timestamps` parameter).

The `measure_start_times` function should start `n` unique child processes. The first action each child takes is obtaining the current system time by calling `clock_gettime`. Then, this timestamp should be stored in a region of memory shared by the parent process and all child processes.

The `measure_start_times` function returns `0` on success or `-1` on error. If the function succeeds, then the array referenced by the `timestamps` parameter should be filled with the `n` timestamp values obtained by the `n` child processes.

Your solution should allow child processes to execute in parallel.

Your solution should perform proper error handling for the `clock_gettime` function but does not need to check any other function calls for errors. If any child process encounters an error, then `measure_start_times` should indicate an error by returning `-1`.

WRITE YOUR SOLUTION ON THE NEXT PAGE ⟶

```
int measure_start_times(struct timespec *timestamps, unsigned n) {

}
```

Strings

```
size_t strlen(const char *s);
char *strcpy(char *dest, const char *src);
char *strtok(char *str, const char *delim);
int strcmp(const char *s1, const char *s2);
```

Memory

```
void *memset(void *s, int c, size_t n);
void *memcpy(void *dest, const void *src, size_t n);
void *malloc(size_t size);
void free(void *ptr);
```

`stdio` Operations

```
FILE *fopen(const char *pathname, const char *mode);
int fclose(FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int fseek(FILE *stream, long offset, int whence);
long ftell(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
```

Process Creation and Management

```
pid_t getpid();
pid_t getppid();
pid_t fork();
int execl(const char *pathname, const char *arg, ..., NULL);
int execlp(const char *file, const char *arg, ..., NULL);
int execle(const char *pathname, const char *arg, ..., NULL, char *const envp);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Environment Variables

```
char *getenv(const char *name);
int setenv(const char *name, const char *value, int overwrite);
int unsetenv(const char *name);
```

I/O System Calls

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
int close(int fd);
```

File System Operations

```
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
DIR *opendir(char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
void rewinddir(DIR *dirp);
```

File System Operations (Continued)

```
int link(const char *oldpath, const char *newpath);
int unlink(const char *path);
int symlink(const char *target, const char *linkpath);
```

Memory-Mapped I/O

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Signals

```
int kill(pid_t pid, int sig);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(sigset_t *set, int signum);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
int pause();
int sigsuspend(const sigset_t *sigmask);
int sigwait(const sigset_t *sigmask, int *signo);
```

Pipes

```
int pipe(int filedes[2]);
int mkfifo(const char *path, mode_t mode);
```

Multiplexed I/O

```
int poll(struct pollfd *fds, int nfds, int timeout);
```

Shared Memory

```
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

Socket Programming

```
int getaddrinfo(const char *node, const char *service,
            const struct addrinfo *hints, struct addrinfo *res);
int socket(int domain, int type, int protocol);
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
            const struct sockaddr *dest_addr, socklen_t addrlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
            struct sockaddr *src_addr, socklen_t *addrlen);
```