Quiz #2

April 17, 2025

Section: 1 or 2 **Student ID:** **Name:**

Q1. What is a difference between non-preemptive scheduling and preemptive scheduling? (1point)

Non-preemptive scheduling: Running process is not interrupted
Preemptive scheduling: Running process can be interrupted (scheduling may occur while process is running)

Q2. What is the problem of starvation in CPU scheduling? Also provide the solution. (1point)

Starvation in operating systems occurs when **a process waits indefinitely to gain access to a needed resource because other higher-priority processes continuously preempt it**. This typically happens in scheduling algorithms that favor certain processes (e.g., shortest job first or priority scheduling), leading lower-priority or long-waiting processes to be perpetually delayed. The main problem with starvation is that it violates fairness and can cause critical tasks to never complete. A common solution is **aging**, a technique that gradually increases the priority of waiting processes over time, ensuring they eventually get scheduled and preventing indefinite postponement.

Q3. Illustrate a Gantt chart for the following situations. (Show the process IDs and time interval) (1point)

- Run the process with the highest priority first (a lower number means higher priority).
- Process with the same priority run in a round-robin manner with a time quantum of 2msec, in increasing order of process ID
- Five processes (P1 to P5) arrive at the same time.

Process ID	Burst Time	Priority
P1	5	1
P2	2	2
P3	3	2
P4	7	1
P5	3	3

Gantt chart

P1	**P4**	**P1**	**P4**	**P1**	**P4**	**P4**	**P2**	**P3**	**P3**	**P5**

0 2 4 6 8 9 11 12 14 16 17 20

Q4. The following source code is about shared memory. Fill in the blanks below. (1point)

* `shm_pro.c`

```
#include ...

int main() {
    const int SIZE = 4096;
    const char *name = "OS";
    const char *message0 = "Studying ";
    const char *message1 = "Operating Systems ";
    const char *message2 = "Is Fun!\n";

    int shm_fd;
    void *ptr;
`
    [      (1)      ](name, O_CREAT | O_RDWR,
                      0666);

    [  (2)  ](shm_fd,SIZE);

    [      (3)      ](0, SIZE,
                       PROT_READ | PROT_WRITE,
                       MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Map failed\n");
        return -1;
    }

    sprintf(ptr,"%s",message0);
    ptr += strlen(message0);
    sprintf(ptr,"%s",message1);
    ptr += strlen(message1);
    sprintf(ptr,"%s",message2);
    ptr += strlen(message2);

    return 0;
}
```

* `shm_con.c`

```
#include ...

int main()
{
    const char *name = "OS";
    const int SIZE = 4096;

    int shm_fd;
    void *ptr;
    int i;

    [      (1)      ](name, O_RDONLY, 0666);
    if (shm_fd == -1) {
        printf("shared memory failed\n");
        exit(-1);
    }

    [      (3)      ](0, SIZE, PROT_READ,
                       MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Map failed\n");
        exit(-1);
    }

    printf("%s\n",(char *)ptr);

    if ([  (4)  ](name) == -1) {
        printf("Error removing %s\n",name);
        exit(-1);
    }

    return 0;
}
```

* Hint: fork(), wait(), execvp(), signal(), shm_open(), shm_unlink(), ftruncate(), mmap(), sprint(), pipe(), mkfifo()

(1) : shm_fd = shm_open

(2): ftruncate

(3): ptr = (char*) mmap

(4): shm_unlink

Q5. Try to predict the output of the code below for the given command. (1point)

* Source code: `wspipe.c`

```
#include ...
…
#define READ_END    0
#define WRITE_END   1
#define BUFFER_SIZE 1024

ssize_t read_line(int fd, char *buffer, size_t max_length);

int main(int argc, char *argv[]) {
    pid_t pid;
    int fd[2];
    int i;
    char buffer[BUFFER_SIZE];
    char *command, *word;
    int total_count = 0;

    if (argc < 3) {
        printf("Usage: %s <command> <word> \n", argv[0]);
        return 0;
    }

    command = argv[1];
    word = argv[2];

    if (pipe(fd) == -1) {
        fprintf(stderr,"Pipe failed");
        return 1;
    }

    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork failed");
        return 1;
    }

    if (pid > 0) {
        close(fd[WRITE_END]);
        i = 1;
        char *ptr, *cur;
        while (read_line(fd[READ_END], buffer, BUFFER_SIZE-1) != 0) {
            cur = buffer;
            while ((ptr = strstr(cur, word)) != NULL) {
                total_count++;
                cur = ptr + strlen(word);
            }
            i++;
        }

        close(fd[READ_END]);
        wait(NULL);
        printf("Result for '%s': %d\n", word, total_count);
    } else {
        close(fd[READ_END]);
        dup2(fd[WRITE_END], STDOUT_FILENO);
        close(fd[WRITE_END]);

        // Just execute the ‘command’
        execl("/bin/sh", "sh", "-c", command, NULL);
        exit(1);
    }

    return 0;
}

ssize_t read_line(int fd, char *buffer, size_t max_length) {
    ssize_t num_read = 0;
    char c;
    while (read(fd, &c, 1) == 1 && num_read < max_length) {
        buffer[num_read++] = c;
        if (c == '\n')
            break;
    }
    buffer[num_read] = '\0';
    return num_read;
}
```

* Result

```
$ gcc wspipe.c -o wspipe
$ ./wspipe 'cat wspipe.c' count
```

```
Output?
```

Result for 'count': 3