

# HW3 + 4

---

## 1. Given that you can create multiple threads to perform different tasks within a program, explain why you might still need to use fork.

---

### 1. Executing New Programs

`fork()` allows a process to create a child that can replace itself with a completely new program using `exec()`. This capability is essential for tasks like running external commands in a shell or starting new applications.

### 2. Privilege Separation

Forked processes can change their user or group privileges independently, enabling security measures like running less-privileged tasks in isolation. This is crucial for applications that handle sensitive or potentially dangerous operations.

### 3. Debugging and Error Containment

Debugging a multi-threaded program can be more challenging due to the complexities of thread synchronization and shared state.

A program using processes is often easier to debug, as processes do not share memory and their state can be analyzed independently.

### Example Scenarios Where `fork()` Is Essential

- **Web Servers:** Many web servers fork processes to handle incoming client connections. Each forked process is isolated and can handle its own client without interference.
- **Command Line Shells:** Shells like `bash` use `fork()` to spawn processes that execute user commands.
- **Process Pooling:** Applications like databases may use processes rather than threads to handle multiple connections due to stability and security reasons.

## 2. The code snippet below needs to be fixed. Point out the problem. (Assume ... does not do anything wrong).

---

```
int main(void) {
    signal(SIGALRM, sig_alm);
    alarm(60);
    if (setjmp(env_alm) != 0) {
        /* handle timeout */
        ...
    }
    ...
}
static void sig_alm(int signo) {
    longjmp(env_alm, 1);
}
```

- Signal Mask Behavior

POSIX does not specify the effect of `setjmp()` and `longjmp()` on signal masks, it does not specify whether `setjmp()` stores the signal mask to the `jmp_buf`. Any section that rely on signal mask in the snippet may be unpredictable depends on different system's implementation of `setjmp()` and `longjmp()`.

We can fix this issue by using `sigsetjmp()` and `siglongjmp()` if we want to ensure predictable behavior.

## 3. Consider the following code snippet for a bounded queue. Consider multiple POSIX threads that could each execute the `enqueue()` and `dequeue()` functions. Fill out the missing parts (A), (B), ©, and (D) to achieve proper synchronization with conditional

**variable and mutex. (0.5pts). Next, explain why the “while (q->count...” checks in the two functions are crucial.**

---

```
typedef struct {
    int data[MAX_SIZE];
    int front;
    int rear;
    int count;
    pthread_mutex_t mutex;
    pthread_cond_t not_full;
    pthread_cond_t not_empty;
} Queue;

void enqueue(Queue *q, int item) {
    pthread_mutex_lock(&q->mutex);
    while (q->count == MAX_SIZE) {
        _____(A)_____; // Wait when queue is full
    }
    q->data[q->rear] = item;
    q->rear = (q->rear + 1) % MAX_SIZE;
    q->count++;
    _____(B)_____; // Signal that queue is not empty
    pthread_mutex_unlock(&q->mutex);
}

int dequeue(Queue *q) {
    pthread_mutex_lock(&q->mutex);
    while (q->count == 0) {
        _____(C)_____; // Wait when queue is empty
    }
    int item = q->data[q->front];
    q->front = (q->front + 1) % MAX_SIZE;
    q->count--;
    _____(D)_____; // Signal that queue is not full
    pthread_mutex_unlock(&q->mutex);
    return item;
}
```

(A): pthread\_cond\_wait(&q->not\_full, &q->mutex):

The producer thread will wait until the queue is not full, releasing the mutex and blocking until not\_full is signaled.

(B): pthread\_cond\_signal(&q->not\_empty):

After adding an item to the queue, signal the not\_empty condition to notify a waiting consumer thread.

©: `pthread_cond_wait(&q->not_empty, &q->mutex)`:

The consumer thread will wait until the queue has at least one item, releasing the mutex and blocking until `not_empty` is signaled.

(D): `pthread_cond_signal(&q->not_full)`:

After removing an item from the queue, signal the `not_full` condition to notify a waiting producer thread.

**4. Using `mmap()` to perform IPC in a Unix environment is possible. Explain how you would do that with a code snippet. (incomplete code is OK; i.e.,**

## the code does not have to be compiled)

---

```
#define SHM_NAME "/my_shared_memory"
#define SHM_SIZE 1024

int main() {
    int shm_fd;
    void *shm_ptr;

    // Step 1: Create or open shared memory
    shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(EXIT_FAILURE);
    }

    // Step 2: Set the size of shared memory
    if (ftruncate(shm_fd, SHM_SIZE) == -1) {
        perror("ftruncate");
        exit(EXIT_FAILURE);
    }

    // Step 3: Map shared memory into address space
    shm_ptr = mmap(NULL, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (shm_ptr == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    // Example communication: Parent writes, child reads
    pid_t pid = fork();
    if (pid == 0) {
        // Child process
        printf("Child: Waiting for data...\n");
        sleep(2); // Simulate delay
        printf("Child: Received message: %s\n", (char *)shm_ptr);
    } else {
        // Parent process
        printf("Parent: Writing data to shared memory...\n");
        snprintf((char *)shm_ptr, SHM_SIZE, "Hello from parent!");
        wait(NULL); // Wait for the child to finish
    }

    // Cleanup
    munmap(shm_ptr, SHM_SIZE);
    shm_unlink(SHM_NAME);

    return 0;
}
```

The shared memory region created with `mmap()` is accessible to both the parent and child processes after `fork()`. Alternatively, unrelated processes can access the same shared memory by opening the same shared memory object (`shm_open`) and mapping it with `mmap()`.