HW3 + 4

B11605076 Cheng-Ru Hsu

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_alrm);
    alarm(60);
    if (setjmp(env_alrm) != 0) {
        /* handle timeout */
        ...
    }
    ...
}
static void sig_alrm(int signo) {
    longjmp(env_alrm, 1);
}
```

- Signal Mask Behavior

POSIX does not specify the effect of `setjmp()` and `longjmp()` on signal masks, it does not specity wheter `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), (C), 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.

(C): 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.

why the "while (q->count. . . " checks in the two functions are crucial is because `pthread_cond_wait` might have **spurious wake**, we use this while loop to address this issue.

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)

```c
struct mapped {
    unsigned char array[ARRAY_ELEMENTS];
    pthread_mutex_t mutex;
};
int main(void) {
    int rc, fd;
    struct mapped *mapping;
    pthread_mutexattr_t mutexattr;
    pid_t pid;

    fd = open(FILENAME, O_CREAT | O_RDWR, 00600);

    mapping = (struct mapped*)mmap(NULL, sizeof(struct mapped),
              PROT_READ | PROT_WRITE,
              MAP_SHARED, fd, 0);
    close(fd); // we can close the file after we've mapped it

    pthread_mutexattr_init(&mutexattr);
    pthread_mutexattr_setpshared(&mutexattr, PTHREAD_PROCESS_SHARED);
    pthread_mutex_init(&mutexattr->mutex, &mutexattr);

    if ((pid = fork()) == -1) exit(1);
    else if (pid == 0) {
        pthread_mutex_lock(&mapping->mutex);
        // do something
        pthread_mutex_unlock(&mapping->mutex);
    } else {
        pthread_mutex_lock(&mapping->mutex);
        // do something
        pthread_mutex_unlock(&mapping->mutex);
    }

    rc = munmap(mapping, sizeof(struct mapped));

    return 0;
}
```

The shared memory region created with `mmap()` **with flag** `MAP_SHARED` 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 and mapping it with `mmap()` too. The `PTHREAD_PROCESS_SHARED` attribute ensures that the mutex in this shared memory can synchronize access to the shared resource.

MAP_SHARED: share this mapping. Updates to the mapping are visible to other processes mapping the same region. If mapping corresponds to an underlying file, the updates are carried through to the file PTHREAD_PROCESS_SHARED: multiple processes with access to the mutex object can access the mutex, i.e., the mutex is shared between processes