

# Homework 4 - Computer Systems

Adil Hydari

## Q1: Signal Masking in Parent Process

### 1.1 Normal Execution

The parent process starts by constructing a signal mask that excludes four specific signals. By blocking **SIGTSTP** (terminal stop), **SIGTERM** (termination request), **SIGINT** (keyboard interrupt), and **SIGFPE** (arithmetic error), it ensures that any critical sections of code are not interrupted. This mask applies from process start to the first explicit handler invocation.

### 1.2 Handler on SIGFPE, SIGHUP, SIGCHLD

When `sig_handler` is invoked for **SIGFPE**, **SIGHUP** (hangup), or **SIGCHLD** (child status change), it inherits the existing mask. This prevents re-entry or interruption by those same signals while the handler runs, ensuring a consistent recovery procedure.

### 1.3 Handler on SIGILL or SIGQUIT

If an illegal instruction (**SIGILL**) or quit request (**SIGQUIT**) arrives, the handler temporarily adds **SIGABRT** (abort) and **SIGQUIT** itself to the mask, on top of the original four. This extended mask protects the handler from nested aborts or quits, allowing it to log diagnostics or perform cleanup safely.

### 1.4 Handler on SIGUSR1

A separate handler `sig_usr1` addresses **SIGUSR1**. During its execution, the mask is extended to block **SIGABRT** and retain **SIGTERM**, plus the four standard signals. This guarantees that user-defined operations complete without unexpected termination.

## Q2: Signal Handling in Parent Process

### 2.1 sig\_handler via sigaction

Using `sigaction`, the parent sets a persistent handler for:

- **SIGILL**: catches illegal instruction faults.
- **SIGINT**: handles keyboard interrupts when unmasked.
- **SIGQUIT**: intercepts user-initiated quits.

This handler remains installed until explicitly changed, ensuring consistent behavior on each arrival.

### 2.2 sig\_handler via signal (one-shot)

A one-shot installation via `signal` is used for:

- **SIGFPE**: arithmetic exceptions.
- **SIGHUP**: hangup notifications.
- **SIGTSTP**: terminal stops.
- **SIGCHLD**: child termination or stop.

After handling one occurrence, the disposition resets to default, allowing subsequent signals to use their default action.

### 2.3 sig\_usr1 via sigaction

A persistent handler is installed for **SIGUSR1**, enabling the process to perform custom work on every delivery of the user-defined signal.

## Q3: Signal Masking in Child Process

### 3.1 Normal Execution

The child process begins by blocking five signals to guard initialization:

- **SIGABRT**: aborts.

- **SIGTSTP**: stops.
- **SIGTERM**: terminations.
- **SIGSEGV**: segmentation faults.
- **SIGILL**: illegal instructions.

This mask ensures safe setup and shared-resource protection.

### 3.2 Handler on SIGFPE, SIGHUP, SIGURG

When handling **SIGFPE**, **SIGHUP**, or **SIGURG** (urgent I/O), the handler retains the full five-signal mask, avoiding nested faults.

### 3.3 Handler on SIGQUIT, SIGILL, SIGINT, SIGTSTP

Invoking `sig_handler` for these signals adds **SIGABRT** and reaffirms **SIGTSTP**, while continuing to block **SIGTERM**, **SIGSEGV**, and **SIGILL**. This combination prevents simultaneous control-flow interrupts.

### 3.4 Handler on SIGUSR1

The `sig_usr1` handler in the child blocks **SIGABRT**, **SIGTSTP**, and **SIGTERM**. It also keeps **SIGSEGV** and **SIGILL** masked, ensuring critical cleanup can complete.

## Q4: Actions from Signals (Lines 150–156, 183 and 157, 180–188)

The code explicitly sends signals on certain lines; here’s what happens.

### 4.1 Sent to Child

- Line 150 (**SIGUSR1**): delivered immediately to `sig_usr1` via `sigaction`.
- Line 151 (**SIGQUIT**): caught by `sig_handler` (persistent).
- Line 152 (**SIGILL**): currently masked, so remains pending.

- Line 153 (**SIGURG**): handled once by `sig_handler` installed via `signal`.
- Line 154 (**SIGINT**): caught by persistent `sig_handler`.
- Line 155 (**SIGSEGV**): blocked, pending delivery.
- Line 156 (**SIGTSTP**): masked by all handler masks, pending.
- Line 183 (**SIGFPE**): handled once; subsequent SIGFPE defaults after handler removal.

## 4.2 Sent to Parent

- Line 157 (**SIGFPE**): currently masked, so pending.
- Line 180 (**SIGUSR1**): not sent (child's PID=0).
- Line 181 (**SIGQUIT**): caught by `sig_handler` (persistent).
- Line 182 (**SIGCONT**): default action resumes the process.
- Line 184 (**SIGILL**): delivered to `sig_handler`.
- Line 185 (**SIGINT**): masked, pending.
- Line 186 (**SIGCHLD**): handled once by one-shot `sig_handler`.
- Line 187 (**SIGHUP**): handled once, then resets.
- Line 188 (**SIGTSTP**): masked, pending.

## Q5: Scenario 2 (All Signals Sent)

Now assume both parent and child exchange every signal from lines 150–165 and 180–197.

## 5.1 Child Receive Actions

Repeated deliveries mirror Q4. Key differences:

- Second SIGURG (line 158) arrives after removal of one-shot handler and is ignored.
- Subsequent SIGFPE, SIGHUP, SIGCHLD behave by default (terminate or ignore) once the one-shot handler is gone.

## 5.2 Parent Receive Actions

Follows Q4 with:

- Re-delivery of SIGHUP (line 164) and SIGFPE (line 165) now cause default termination.
- SIGCHLD at line 193 uses default ignore.
- Persistent handlers continue catching SIGQUIT and SIGILL.

## 5.3 Pending Signals

After all sends complete:

- **Parent:** SIGFPE, SIGINT, SIGTSTP remain in the waiting mask.
- **Child:** SIGILL, SIGSEGV, SIGTSTP remain pending.

## Q6: Terminating Signals

Any signal with default disposition *terminate* or *core dump* can kill the process when unhandled or after a one-shot handler is removed. Notable examples in this program include: SIGKILL, SIGBUS, SIGTRAP, SIGSYS, SIGIOT; plus repeated SIGFPE or SIGHUP once their custom handlers have executed.

## Q7: Threads and Signals (Problem 4B)

### 7.1 Q7.1: First Thread into func

Thread scheduling is non-deterministic. Regardless of which thread starts first,

1. The initial call to `func` computes denominator safely.
2. The second call sees , so denominator zero, raising **SIGFPE**. No handler exists yet, so that thread terminates.
3. The third call uses denominator one and completes normally.

### 7.2 Q7.2: Part `/* B */` Lines

**Line C (SIGSEGV to tid2, 3 times)** Thread 2 has a persistent handler for **SIGSEGV** via `sigaction`. Each of the three signals invokes the handler, producing three lines of output: “Caught signal no = 11”.

**Line D (SIGINT to tid1, 3 times)** Thread 1 inherited **SIG\_IGN** for **SIGINT** before creation. Each delivered **SIGINT** is ignored, yielding no output.

### 7.3 Q7.3: External `kill(pid,sig)` Signals

**Q7.3.1: SIGSEGV (11)** Any thread not currently blocking **SIGSEGV** may handle it. The OS picks the first available; that thread’s re-installing handler prints “Caught signal no = 11” on each of the three kills.

**Q7.3.2: SIGQUIT (3)** Threads install `sig_func` or `sig_func2` at different times:

- If a thread without a custom handler receives **SIGQUIT**, default disposition terminates that thread.
- If `tid3` or `main` (after its handler set) receives it, they print “Caught signal no = 3” and continue.

**Q7.3.3: SIGBUS (7)** The first installation uses `sig_func`, then overrides with `sig_func2`. Each delivery invokes the active handler, printing “Caught signal no = 7”.

## Q8: Multi-threaded Sieve (Problem 5A)

Implement two threads:

- **Sieve thread:** Allocate array , mark every multiple of each prime (starting at ) as composite, stopping at . This yields all primes in time.
- **Reversible-prime thread:** Iterate through marked primes. For each prime , reverse its decimal digits to get ; check . If is also prime, print .

Threads share read-only and write-only regions and synchronize only on startup and shutdown.

## Q9: Fibonacci Threads (Problem 5B)

Original solution:

- Main thread reads .
- Worker thread fills shared array .
- Main thread `pthread_joins`, then prints all values.

To print values incrementally as they arrive:

1. Protect the shared array and an index counter with a mutex.
2. Worker, after computing , locks the mutex, increments a count, signals a condition variable, then unlocks.
3. Main, in a loop for to , locks the mutex, waits on the condition until count*i*, prints , unlocks.

This allows the parent to output each Fibonacci number immediately when it becomes available.