Computer Sytems - Solutions to Problems 1–4

Problem 1: Processes, IPC, and Signals

Part 1 (4 points)

1. **Steady state:** A process is "stuck" but not terminated:
   - Blocked on I/O or a resource forever,
   - Running a function (like `g()`) that never returns.
2. **Zombie process:** A process has exited but is not yet `wait`-reaped by its parent. It disappears once the parent (or `init`) calls `wait()`.
3. $$\texttt{wp} = \texttt{wait}(\&\texttt{status}), \quad \texttt{wp} = 1083, \quad \texttt{WIFEXITED(status)} = 1, \quad \texttt{WEXITSTATUS(status)} = 101.$$

   Then process 1000 is the *parent* of process 1083. Process 1083's final call was effectively `exit(101)`.
4. **Pipes:**
   - `read()` after the last writer closes yields 0 (EOF).
   - `write()` when no readers exist raises `SIGPIPE` or returns `EPIPE`.

Part 2 (6 points): Process Tree Sketch
A parent creates three pipes and three children.

- **Child i=0** eventually reads from `fd[0]`, forks subchildren, blocks in `wait` or ends up stuck in `g(0,5)`.
- **Child i=1** writes a value to child 0, reads from `fd[1]`, forks subchildren, finally calls `g(1,5)`.
- **Child i=2** is killed by the parent (`SIGKILL`) and then reaped.

IPC: Child 1 sends an integer to child 0 via `fd[0]`, parent writes to `fd[2]` but kills child 2 before it can read, and parent reaps child 2. All surviving processes remain in functions like `g()` or `wait()`, forming a permanent steady state.

Problem 2: Processes and Pipes

Part 1 (4 points): Explanation

This program creates four children in a loop (`i=0..3`). Each child:

1. Uses a pipe to exchange data,
2. Forks subchildren based on how many bytes are read,
3. Ends by calling `f(...)` (which never returns).

The parent then sends SIGUSR1 to the child i=2, causing it to exit immediately; the parent reaps that child's status. Meanwhile, children i=0,1,3 eventually block in wait() for their own subchildren (or call f(...) themselves). The parent also gets stuck in its second wait(), since no other child exits.

Part 2 (3 points): Final Process Tree

- **Parent**: Created children i=0..3. Reaps only child 2, then blocks in wait().
- **Child i=2**: Receives SIGUSR1 early, calls exit(), then is reaped.
- **Child i=0**: Reads some bytes, forks subchildren (each stuck in f()), and itself ends up blocked in wait() or in f(0,5).
- **Child i=1**: Similar behavior: reads bytes, forks, blocked waiting on subchildren, or calls f(1,5).
- **Child i=3**: Reads bytes, forks subchildren, then blocks in wait().

IPC:

- Child 1 writes a value to child 0,
- The parent writes to child 2's pipe but kills it with SIGKILL,
- Parent also writes data for child 3 to read.

All remaining processes are stuck in either wait() or a never-returning f().

Problem 3: Processes and Pipes

Part 1 (4 points): Explanation

A loop creates four children (i=0..3), each associated with a pipe. The parent writes a PID into each pipe (or closes the pipe, causing the child to read zero). Each child:

1. Reads a pid_t from its pipe,
2. If that read value is positive, sends SIGUSR1 to the indicated process,
3. Calls f(i, pid[i]) (which never returns).

The parent also sets up a signal handler for SIGUSR1 that calls f(0, -3). Hence, any child receiving SIGUSR1 will jump into that handler and remain stuck. The parent tries a wait(NULL) but never reaps any child (none exit), so it remains blocked or otherwise stuck forever.

Part 2 (3 points): Final Process Tree

- **Parent**: Stuck at wait(NULL); no child ever exits.
- **Child 0**: Possibly gets SIGUSR1 from Child 1, then runs handler → f(0, -3) forever.
- **Child 1**: After reading Child 0's PID, does kill(pid[0], SIGUSR1), then calls f(1, pid[0]).
- **Child 2, Child 3**: Typically read zero from their pipes (EOF), do not send signals, and remain in f(2,0) or f(3,0).

Because f() never returns, all processes remain active and blocked in their respective states, forming a permanent steady state.

Problem 4: Processes and Pipes

Code and Behavior:

1. The parent forks a child, then writes two chunks of data to the pipe:
   - `8` bytes initially (`2*sizeof(int)`),
   - Another `8` bytes afterward (`r1` is 8).
2. The child reads data in three calls:
   - `2` bytes first (`r3` = 2),
   - `4` bytes second (`r5` = 4),
   - `3` bytes third (`r6` = 3).
3. After writing, the parent does `waitpid(..., WNOHANG)`: if the child has not exited, it returns `0`.

Values of `r1, r2, r3, r5, r6, r7` and Orphans/Zombies:

- $$r1 = 8, \quad r2 = 8, \quad r3 = 2, \quad r5 = 4, \quad r6 = 3, \quad r7 = 0 \text{ (child not exited yet).}$$
- The child does a final `sleep(20)` unless line D is removed, in which case it may exit first.
  - If the child outlives the parent, it becomes an orphan (no zombies remain).
  - If line D is removed, the child can finish early and be reaped by the parent (`r7` > 0), meaning no orphan at the end.

Summary of Cases:

- **No lines removed (Q1)**: `r1=8, r2=8, r3=2, r5=4, r6=3, r7=0`, one orphan (the child), no zombies.
- **Removing A, B, or C (Q2–Q4)**: Same read/write sizes and `r7=0`; child still outlives the parent, becoming an orphan.
- **Removing D (Q5)**: Child exits faster, so typically `r7` = childPID > 0; no orphan remains.