Example 1: Signal Masking and Races

```
int num_jobs = 0;

int main(int argc, char **argv)
{
    char c;
    char cmdline[MAXLINE];

    /* Install the signal handlers */
    Signal(SIGCHLD, sigchld_handler);  /* Terminated or stopped child */

    /* Execute the shell's read/eval loop */
    while (1)
    {
        /* print the prompt */
        print_prompt();

        /* read command line */
        if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
            app_error("Terminated due to fgets error");

        /* evaluate the command line */
        eval(cmdline);
    }

    exit(0); /* control never reaches here */
}


/*
 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
 *     a child job terminates (becomes a zombie), or stops because it
 *     received a SIGSTOP or SIGTSTP signal. The handler reaps all
 *     available zombie children, but doesn't wait for any other
 *     currently running children to terminate.
 */
void sigchld_handler(int sig)
{
    int status;
    pid_t pid;
    struct job_t *currentJob = NULL;
    int reaping = 1;

    debug("Entering sigchld_handler()\n");

    /* Reap as many children as possible */

    while ( (pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0 )
    {
        num_jobs--;
        remove_from_joblist(pid);
    }

    debug("Returning from sigchld_handler()\n");
    return;
}
```

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    pid_t pid;
    int bg = parseline(cmdline, argv); /* Initialize argv */




    /* Display the current state of the system */

    printf("The current number of jobs is %d\n", num_jobs);
    print_job_list();




    /* Fork and create child */

    pid = fork();




    if ( pid == 0 )     /* I AM THE CHILD */
    {
        /* Start new program */
        if ( execve(argv[0], argv, environ) == -1 )
        {
            debug("Child could not start new program - execve(): %s\n",
                  strerror(errno));
        }

        exit(0);
    }

    else                /* I AM THE PARENT */
    {
        num_jobs++;     /* update the number of jobs */
        add_to_joblist(pid);
    }
    return;

}
```

The main problem here is that the child could change it's state after the parent returns from `fork()`, but before the child gets added to the job list. To make sure that we can track the child's state correctly, we need to block all signals from the child until we have set up a record for it. This means that we should block signals (SIGCHLD, SIGINT, SIGSTP) before we `fork` and only unblock them once our job list is updated. Note that since the child inherits signal masks, we need to unblock signals in the child before we call `execve()`.

Example 2: Never The Same Thing Twice

NOTE: *Adapted from Problem 3, Exam 2, Spring 2004*

Consider the following C programs. (For space reasons, we are not checking error return codes, so assume that all functions return normally.) Assume that `printf()` is unbuffered and that each call to `printf()` executes atomically.

```
/* parent.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAXLEN 32

int main(int argc, char *argv[])
{
   int i, status, limit;
   char num[MAXLEN+1];
   pid_t pid;

   for(i = 0; i < 3; i++)
   {
      if((pid = fork()) == 0) /* child */
      {
      /* convert i into a string for printing */
         snprintf(num, MAXLEN, "%d", i);
         execl("./kid", "./kid", num, NULL);
      }

      /* parent */
      if(i == 1)
      {
         waitpid(pid, &status, 0);
         printf("%d\n", status);
      }
   }
   return 0;
}
```

```
/* kid.c */
#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("%d\n", atoi(argv[1]));
   return 0;
}
```

Assume that parent.c is compiled into `parent` and kid.c is compiled into `kid` in the same directory. List **all** possible outputs (note the newlines) of the following command:

```
> ./parent
```

Since the parent waits for its second child to finish, we must have the output from that child "1", followed by the parent printing the child's status ("0" since the child is assumed to exit normally), followed by the output from the parent's third child ("2"). There are no restrictions placed on the ordering of these events and the parent's first child. That child can come and print "0" anywhere in between the other `printf()` calls. Because there are four places where child 0 can come in, we have four different output sequences:

10, 11, 0, 12

11, 10, 0, 12

11, 0, 10, 12

11, 0, 12, 10