0/6 Questions Answered

Check-in Quiz 02 pipe(), signals(), critical sections

Q1 pipe()
5 Points

For this problem, read and consider the following code:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
  int pipe_fds[2];
  pipe(pipe_fds);

  /*** 1 ***/

  pid_t child1 = fork();
  if (child1 == 0) {
    // first child
    dup2(pipe_fds[1], STDOUT_FILENO);
    close(pipe_fds[0]);

    /*** 2 ***/

    // should print "hello" to the pipe
    char* args[] = {"echo", "hello", NULL};
    execvp(args[0], args);
    perror("execvp error");
    exit(EXIT_FAILURE);
  } else if (child1 < 0) {
    // error!
    perror("fork error");
    exit(EXIT_FAILURE);
  }

  /*** 3 ***/

  pid_t child2 = fork();
```

```
if (child2 == 0) {
  // second child
  dup2(pipe_fds[0], STDIN_FILENO);
  close(pipe_fds[0]);

  /*** 4 ***/

  // should read from the pipe till EOF
  // print to stdout everything it reads
  char* args[] = {"cat", NULL};
  execvp(args[0], args);
  perror("execvp error");
  exit(EXIT_FAILURE);
} else if (child2 < 0) {
  perror("fork error");
  exit(EXIT_FAILURE);
}

/*** 5 ***/
close(pipe_fds[0]);
close(pipe_fds[1]);

int wstatus;

// incase you haven't seen a do-while loop,
// it always does one iteration of the loop, and then
// it acts like a while loop after the first iteration.
//
// For example, these two do the same thing:
// ----------------------------
// some_function();
// while (condition) {
//   some_function();
// }
// ----------------------------
// do {
//   some_function();
// } while(condition);
// ----------------------------

do {
  if (waitpid(child1, &wstatus, 0) == -1) {
    perror("waitpid error");
    exit(EXIT_FAILURE);
  }
} while (!WIFEXITED(wstatus) && !WIFSIGNALED(wstatus)); /*** 6 ***/

do {
  if (waitpid(child2, &wstatus, 0) == -1) {
    perror("waitpid error");
    exit(EXIT_FAILURE);
```

```
    }
  } while (!WIFEXITED(wstatus) && !WIFSIGNALED(wstatus)); /*** 7 ***/

  /*** 8 ***/

  return EXIT_SUCCESS;
}
```

Q1.1 What is the problem?
1 Point

There is a bug in this code, causing it to have the incorrect behaviour. The intended behaviour is to do the same thing as `echo hello | cat` in the terminal.

What is the kind of error this program faces?

- Parent does not wait for the children properly
- Child2 does not inherit the pipe correctly and cannot read from it
- Child1 does not redirect stdout to the pipe correctly
- Child2 never hits EOF when reading from the pipe
- Child1 never sends EOF when writing to the pipe
- The arguments to execvp are malformed

Save Answer

Q1.2 What line to edit?
1 Point

There is only one line that needs to be edited to get the correct behaviour. Throughout the program there are 8 comments in the style of `/*** 7 ***/` with a varying number in the comment. One of these lines with a comment can be edited so that the program works. Which line is it?

Note that `/*** 6 ***/` and `/*** 7 ***/` are the only ones that are on a line that already has code.

For the other ones, you can still "edit" the line by replacing the line with a line that has code.

```
/*** 1 ***/
/*** 2 ***/
/*** 3 ***/
/*** 4 ***/
/*** 5 ***/
/*** 6 ***/
/*** 7 ***/
/*** 8 ***/
```

Save Answer

Q1.3 How do we fix it?
3 Points

For the selected line above, please provide what the line should look like to have the expected behaviour.

Be sure to follow the same spacing and style as the provided code, gradescope is picky about the formatting for the accepted answer.

If you are submitting, and think you are correct, but think it may be a formatting issue; make a private post on Ed and course staff will help.

Save Answer

Q2 Critical Section

3 Points

In lecture, we defined a critical section slightly incorrectly. The definition didn't properly cover all possible cases that should be considered a critical section. The wording was also made slightly better. Wording that was added/changed are in **bold**.

Here is an updated definition below:

There can be issues when **one or more resources** are accessed concurrently that causes the **program** to be put in an **unexpected**, invalid, or error state.

These sections of code **where these accesses happen**, called critical sections, need to be protected from concurrent accesses happening during it

---

Q2.1 List
2 Points

Consider the example with critical sections we discussed in lecture:

```
// assume this works
void list_push(list* this, float to_push) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) exit(EXIT_FAILURE);
  node->value = to_push;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}

void handler(int signo) {
  list_push(list, NaN);
}

int main(int argc, char* argv[]) {
  signal(SIGINT, handler);
  float f;
  while(list_size(list) < 20) {
    read_float(stdin, &f);
    list_push(list, f);
  }
  // omitted: do stuff with list
```

```
}
```

With this example we also said you could:

- Assume we have implemented a linked list, and it works
- Assume list is an initialized global linked list

In lecture we, said to fix this we could block signals temporarily in `main()` and `handler()` just before they `list_push()` and then unblock right after we return from `list_push()`

Lets say that instead of changing `main()` and `handler()`, we instead change `list_push()` to block signals. Which of the following lines **at minimum** would need to be included in the critical section to have the program work correctly.

Select all that apply:

- [ ] `node->value = to_push;`
- [ ] `node->next = NULL;`
- [ ] `this->tail->next = node;`
- [ ] `this->tail = node;`

Note the call to malloc would also be part of the critical section, but I am just going to tell you that one since it would be hard to know without thinking about how `malloc()` works

Save Answer

Q2.2 Vector
1 Point

A `vector` is data structure that represents a resizable array. For those used to Java, think of it like an `ArrayList`.

Consider the following C snippet that outlines what a `vector` of `float`s

is and how we would push a value to the end of it

```
typedef struct vec_st {
  size_t length;
  size_t capacity;
  float* eles;
} Vector;


void vec_push(Vector* this, float to_push) {
  // assume that we don't have to resize for simplicity
  assert(this->length < this->capacity);

  this->length += 1;  // increment length to include it
  this->eles[this->length - 1] = to_push;  // add the ele to the end
}
```

Is there a critical section in the `vec_push` function? If so, what line(s)?

- There is no critical section in `vec_push`
- There is a critical section, and it is the line `this->length += 1;`
- There is a critical section, and it is the line `this->eles[this->length - 1] = to_push;`
- There is a critical section, and it includes both of the lines mentioned in the above two answers

Save Answer

Q3 Signals

2 Points

In lecture we talked about signal blocking and unblocking. This is a slightly altered version of the code from `delay_sigint.c` lecture code.

The difference being that the lecture code had `main()` wait for `SIGALRM` to go off and then have `main()` unblock `SIGINT`. The edited code below just has `main()` loop infinitely without checking to see if the alarm goes off, and the `handler()`

function that goes off when `SIGALRM` is raised will unblock the `SIGINT` signal.

```
sigset_t mask;
sigset_t old_mask;

void handler(int signo) {
  if (signo == SIGALRM) {
    printf("alarm delivered\n");
    if (sigprocmask(SIG_SETMASK, &old_mask, NULL) == -1) {
      perror("sigprocmask failed, idk how but it did");
      exit(EXIT_FAILURE);
    }
  }
  if (signo == SIGINT) {
    printf("got sigint\n");
    exit(EXIT_SUCCESS);
  }
}

int main() {
  // initialize the set
  if (sigemptyset(&mask) == -1) {
    perror("sigemptyset failed, idk how but it did");
    exit(EXIT_FAILURE);
  }

  // add SIGINT to the set
  if(sigaddset(&mask, SIGINT) == -1) {
    perror("sigaddset failed, idk how but it did");
    exit(EXIT_FAILURE);
  }

  // block SIGINT
  if (sigprocmask(SIG_BLOCK, &mask, &old_mask) == -1) {
    perror("sigprocmask failed, idk how but it did");
    exit(EXIT_FAILURE);
  }

  // not error checking cause I am too tired
  signal(SIGALRM, handler);
  signal(SIGINT, handler);
  alarm(5);

  // infinitely loop, SIGINT should be able
  // to terminate us after the alarm goes off
  while (true) { }

  return EXIT_SUCCESS;
}
```

Does this work? If not, what is the difference in behaviour?

- This program works the same as the original example lecture code
- If SIGINT is sent before the alarm goes off, it is handled immediately and not delayed
- SIGINT will always be blocked once `main()` blocks it
- If SIGINT is sent after the alarm goes off, it will not be handled correctly
- SIGALRM is not handled correctly, and the handler does not get invoked as expected
- Segmentation Fault or other Memory Error

Save Answer

Save All Answers **Submit & View Submission ›**