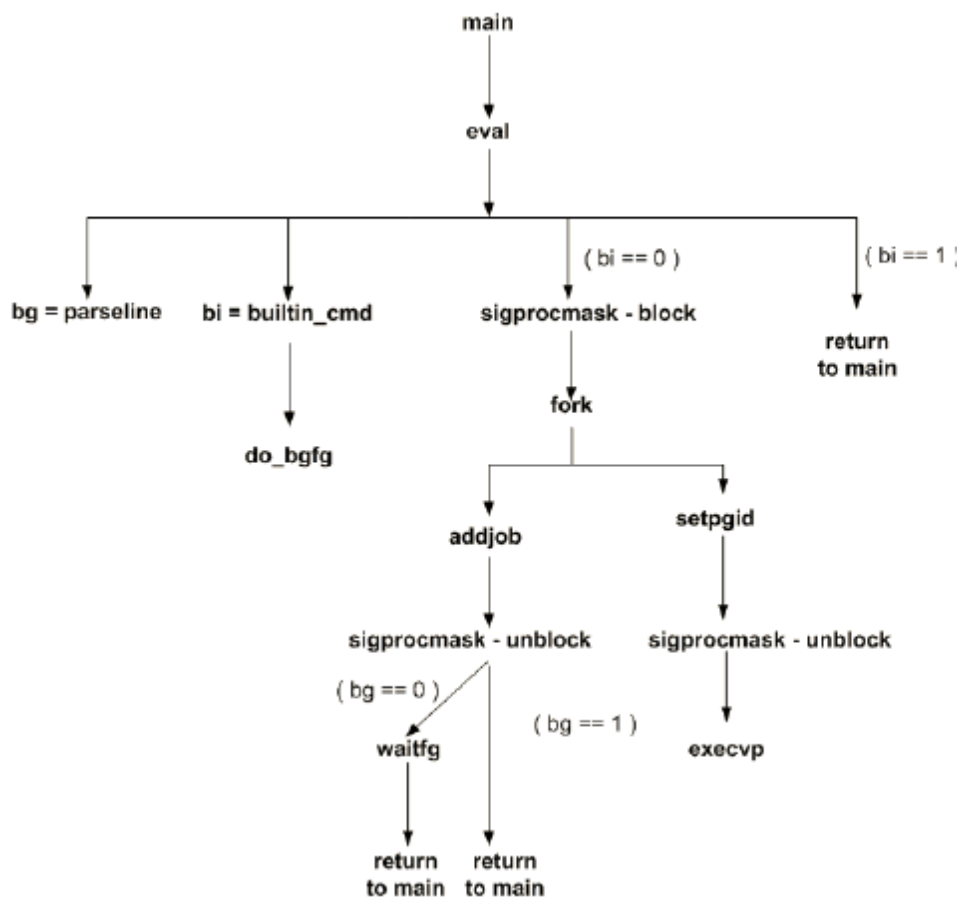


# tsh – implementation guide

For the shell project, you are given an initial incomplete program. Some functions are already implemented or partially implemented. Others are empty and for you to complete. You may add additional functions if you wish, but the provided functions when completed should suffice.

Note: this guide is provided just for reference. Remember, you have to implement the full specification of tsh detailed in the project handout and then turnin your solution via moodle, by April 11.

Here is a call graph of the functions provided.



## Shell Implementation - Step 1

### 1. builtin\_cmd:

`builtin_cmd` should return 1 for all these "builtin" commands.

1. quit

Can just call `exit(0)` for now. Later a check should be made if there are any background jobs and exit only if there are none; otherwise, if there are any background jobs, print a message and return.

2. jobs

Can call the provided `listjobs` function to print background jobs (if any).

3. fg and bg

Call `do_bgfg`, but don't implement `do_bgfg` yet.

Also until you implement code for commands that are not builtin (e.g until you fork, etc), `builtin_cmd` can return 1 for all commands, preventing eval from forking.

2. eval

1. call `parseline`

2. if there were **no** items on the command line return to main

3. call `builtin_cmd` to check for builtin commands and return to main if `builtin_cmd` returns 1 (yes it was a builtin\_cmd). Initially `builtin_cmd` can always return 1 whether the command is builtin or not. So eval need not add the code for calling fork, etc. yet.

## Tests

### Run tsh

1. Enter blank lines: should print the prompt again
2. quit command (should terminate the shell)
3. ctrl-d (should terminate the shell)
4. jobs (should print nothing at this point)
5. fg, bg (should print nothing at this point)

## Shell Implementation - Step 2

In the previous step only (some) builtin commands were implemented and no child processes were created by the shell. Next add to the shell the following functionalities:

1. Error Checking for fg and bg (but no execution yet; just error checks)
2. Create child process to execute non-builtin commands (but running in the foreground only; no background processes, yet).
3. Wait for each child to finish (but NOT by calling `waitpid`)
4. Completely remove the child process; that is, *reap* the child process so that it is completely removed from the Linux process table. (The `SIGCHLD` signal handler will call `waitpid` to do this after a child finishes.)

## Error checking for fg and bg

The `do_bgfg` function gets the array of command line arguments:

```
void do_bgfg(char *argv[]);
```

This function will only be called if `argv[0]` is either "fg" or "bg", but its argument might be incorrect or missing.

### Missing argument

The `argv` array will contain a NULL entry at the index after the last actual command line argument. So `argv[1]` will be NULL if the argument for "fg" or "bg" is missing. If the command is "fg" and the argument is missing, you should print this message and return.

```
fg command requires PID or %jobid argument
```

### Incorrect argument type

If the argument to "fg" or "bg" is NOT missing, it is in `argv[1]`.

The argument should either be the pid (an integer) or `%jid` (% followed immediately by the job id integer). E.g.,

```
fg 1400
```

or

```
fg %2
```

But any other kind of argument would be an error. E.g.

```
fg a
```

or

```
fg [2]
```

If the argument to "fg" is not a pid or `%jid`, then print this message and return.

```
fg: argument must be a PID or %jobid
```

### Correct argument type, but pid or job id doesn't exist

If the argument to `fg` is an integer, it should be the pid of an existing job. But you need to check if it is in the job table.

Use the provided helper function

```
struct job_t *getjobpid(struct job_t *jobs, int pid);
```

This function returns the NULL pointer if no job exists for pid. For example, if the argument to "fg" is 12345 and there is no job with pid = 12345,

```
p = getjobpid(jobs, pid)
```

would return NULL.

In that case do\_bgfg should print this message and return:

```
(12345): No such process
```

Similarly, if the argument is of the form %jid, check if this jobs is in the job table with the function:

```
struct job_t *getjobjid(struct job_t *jobs, int jid);
```

If the argument to "fg" is %2 but there is no job with jid = 2, then print this message and return:

```
fg %2: No such job
```

## Create Child Process to execute non-builtin commands

If the command was not builtin,

1. create a child process; using fork
2. child executes; use execvp
3. if execvp fails, child prints a message and exits.

For example, if there is no program or command named "blob" and execvp fails, the child should print

```
blob: Command not found
```

4. The shell should wait for the child by calling the waitfg function:

```
void waitfg(pid_t pid);
```

See below for how to provide a *temporary* implementation of the waitfg function. (The implementation should NOT call waitpid.)

## Temporary waitfg implementation

The waitfg function should wait until the foreground process changes state (either terminates or is stopped) and then return. But the comment before *waitfg* warns that you

should not use the `waitpid` function for this, but just simply loop until the job state changes. The only function that should call `waitpid` is the `sigchld_handler`. However, at this step jobs are not being added to the job table where the job state is to be recorded and maintained by the shell. But a *temporary* implementation of `waitfg` can be provided that doesn't call `waitpid`. Just call the `pause` function like this:

```
pause( );
```

This will block until a signal is received. When the foreground process changes state, the `SIGCHLD` signal is sent to the shell. This will cause the `pause` function to return. Don't try to run jobs in the background yet. This means there will be only one child process at a time. With this restriction, using the `pause()` function works to wait for that child. Once background jobs are allowed, the implementation of `waitfg` will need to be implemented as stated in the *waitfg* comment.

### The `sigchld_handler` function

Recalling that signals are *not queued*, the `sigchld_handler` needs to loop calling `waitpid` until all children who have changed state (i.e., terminated or stopped) have been waited for. If there are additional children who have not yet changed state, the `sigchld_handler` should not wait for those children.

So the third parameter (options paramter) passed to `waitpid` needs to set the following in addition to the default of returning information when a child *terminates*:

- `WNOHANG` - return 0 immediately if additional children exist, but none have changed state yet
- `WUNTRACED` - also return pid if a child has been stopped (not just if a child has terminated)

## Shell Implementation - Step 3

1. [Implement `sigint\_handler` and `sigtstp\_handler`](#)
2. [Finish implementing `sigchld\_handler`](#)
3. [Finish `eval`](#)
4. [Change implementation of `waitfg`](#)
5. [Finish implementing `do\_bgfg`](#)
6. [Update the `quit` command](#)

### Implement `sigint_handler` and `sigtstp_handler`

These signal handlers should forward the signal to the foreground process job if there is one; else do nothing.

1. Use the `fgpid` function to get the pid of the foreground job. This function return 0 if there is currently no foreground job.

2. If there is a foreground job, use the kill system call to send the signal to the process group for the foreground job.

Note 1: The group id of the foreground job is the pid stored in the job table. Why?

Note 2: How do you send the signal to all processes in the foreground job process group?

### Finish implementing sigchld\_handler

If waitpid returns the pid of a child,

1. Check if it terminated normally. If so delete it from the job table. This will change all fields of the entry for the job. In particular, it changes the state entry to be UNDEF.
2. Otherwise, check if the process was terminated by a signal. If so, delete the job from the job table and print a message including the signal number that caused termination. (See the tests or run the reference shell to see the exact format and content of the message.)
3. Otherwise, check if the process was stopped. In this case, print a message (see the tests for the exact format and content) and change the state to ST, but don't delete the job.

### Finish eval

1. Declare a signal set (type sigset\_t) variable and initialize it to be empty (sigemptyset function)
2. Add the SIGCHLD signal to the signal set
3. If the command is not builtin, block SIGCHLD before forking a child (sigprocmask function). Why?
4. Child should create a new process group with itself as the group leader (setpgrp function) before calling execvp.
5. Parent should add the child to the job list and then unblock the SIGCHLD signal. Why?

### Change implementation of waitfg

1. Get a pointer to the entry in the job table for the pid passed to waitfg.
2. Loop while the state of this entry is FG. Call sleep to sleep for 1 second between checks.

When the state is no longer FG, return.

### Finish implementing do\_bgfg

1. Get the entry for the job from the job table

2. If the state is ST and the command is bg, change the state to BG and send SIGCONT signal to the process group. How?
3. If the state is ST and the command is fg, change the state to FG, send SIGCONT signal to the process group (how?) and wait (by calling waitfg).
4. If the state is BG and the command is fg, change the state to FG and wait (by calling waitfg)
5. Print appropriate messages (see the expected output from tests for the exact content of messages).

## Update the quit command

Check if there are stopped jobs (how?). If so print a message, but don't exit. Just return 1.

---

### Note:

The tsh.c program already has a nifty way of turning on/off print statements, which is useful for debugging. For example, if you wanted to print a message for debugging purposes every time the sigchld\_handler executes, include this code in sigchld\_handler:

```
if (verbose) {  
    printf("... your message  ...\n");  
}
```

Then if you want debugging messages like this printed, start the shell with the -v option:

```
tsh -v
```

This option will set the value of verbose to be 1. Otherwise, without the -v option, the value of verbose is 0. The global int variable verbose is already declared in the tsh.c file (outside any function) and so it can be used as above from any function in the tsh.c file.