How To Run Minecraft Server In The Background Properly

As with nearly all things Unix/Linux, there is more than one solution.

You mentioned you want to be able to launch your server as a foreground process but be able to send it to the background and ... recall it to the foreground at a later time. I'll address answers to those questions here. There is a completely different way of doing things if you want to set up your process so that it launches as a system service (e.g. launching it at boot time). If you regularly know that this is what you want, it is the more accepted (e.g. more operationally mature) way of doing things. I wont address that method here -- you indicated this is not what you want to do.

How to run jobs in the background -- the simple way

The easy way to run any job in the background is to put an & at the end of the line in the command shell when launching the process.

This presume you know that you want to run this in the background when you launching it.

The command shell (terminal window) will issue a job id (in square brackets), followed by the process ID.

I don't have a minecraft server, so I'll simulate running a job in the background using a 'sleep' command (you would use your minecraft script instead of the sleep command):

This starts the job running in the background and note that it was assigned job ID 1 (because it is the first job launched in this command shell) and the process ID was 3958.

When the job finishes the run, it will produce this output (typically the next time you hit the enter key you'll see the message):

Moving foreground jobs to the background

If you launched the job in the foreground but want to move it to the background, you have to interrupt the job using ^ Control+Z.

This stops the job and returns your command prompt. At this point the job is stopped and

neither in the foreground or background. But it is not terminated.

To move it to the background, type:

e.g.

This moves the currently stopped job to continue running in the background.

View the list of background jobs

If you've launched a few jobs and can't remember which job ID is associated with each of them, you can list them by using the jobs command.

In this example, only one job was running. But if multiple jobs were running you would have seen each listed with their job ID (note that this does not display their process ID).

Recalling a background job to the foreground

To return the job to the foreground, use fg %

This brings the task back to the foreground. You can always use ^ Control+Z to stop it and then use bg to return it to the background again.

What happens to job output?

When you run a program in the foreground, any output from that process is typically displayed in your terminal window. So what happens to background processes?

In Unix & Linux, programs normally receive input from stdin (standard input) and send output to stdout (standard output) with the exception of error messages, which are typically sent to stderr (standard error).

By default, stdin is mapped to the terminal keyboard, stdout is mapped to the terminal display, and stderr is also mapped to the terminal display. But you can override those.

If you do not override those, you may see messages from background jobs popping up on your terminal display (in the midst of other things you may be doing) and this can get annoying.

To avoid this, you can redirect the output using the redirect symbol: >

I have a simple script named sayhello.sh with the following:

If I simply run this in the background via ./sayhello.sh & then the word "Hello" will pop up on the terminal in 10 seconds... in the midst of whatever else I might be doing. To redirect this to an output file (to avoid the message from popping up on my display) I can use the redirect >.

This will run the job in the background, but send the output to sayhello.log instead of the terminal display.

That only redirects stdout ... not stderr ... so error messages thrown may still be displayed on the terminal (and... maybe that's what you want so you can be immediately aware of an errors). But often even stderr should be redirected.

It turns out that both stdout and stderr are mapped to numeric values 1 and 2 respectively. So:

is actually the same as

The notation 1> is the same as just using > alone, but it is a bit more obvious (due to the number 1 that we are specifically referring to stdout).

The notation 2> means you want to redirect stderr. I could use

This sends stdout to sayhello.log and stderr to sayhello-errors.log.

If you want both stdout and stderr to go to the same file, there's a short-hand notation:

The first redirect via > says you want to redirect stdout. The second redirect is a bit more complicated 2>&1 which, translated to english says "2 goes to the same place as 1" or... in even more plain english "send my error messages to the same place where I am logging standard output."

Overwritting vs. appending ... or just getting rid of output completely

The redirect directive > normally creates the file and begins writing output to it. But if the file already exists, the directive overwrites the file. Any previous contents (from prior runs) are lost.

You can use the concatenate notation >> to append to the existing file. This will not overwrite the file:

This runs the job in the background but appends output to sayhello.log (if the file already exists... otherwise it will create a new file) and sends stderr to that same file.

If you don't want the output, there is a special file in Unix/Linus named /dev/null. This sends the output to the great bit-bucket in the sky (information is simply discarded.) It's like sending the output to the trash.

The process will run... but will not produce any output (at least not output from stdout or stderr. Other I/O are not impacted by this.

More help

There are other short-hand notations as well. I also didn't go into how to remap stdin (via

All of these (bg, fg, jobs, &, >, etc.) are internal shell commands. They don't have their own documentation. You'll find the documentation embedded in the documentation for your command shell (which, by default, is the BASH shell). This means you can use man bash to read the built-in Linux documentation.
teen time