https://www.opensuse-forum.de/thread/40397-su-oder-sudo-und-der-ganze-shell-und-login-wirrwarr/?postID=120515#post120515

Shells? Und wenn ja, wieviele?

Es gibt unzählige Shells in der Unix/Linux Welt. Welche davon in einer Distri verfügbar sind, kann man sich oft mit `cat /etc/shells` angucken. Diese Liste ist nur ein sehr kleiner Ausschnitt der unzähligen Shells und es ist nicht klar, ob manche davon erst zu installieren sind. Ganz sicher werden NICHT alle gelistet die es gibt. (Es gäbe vogelwilde Kommandos, um herauszufinden, welche installiert sind, aber so viel wollen wir gar nicht wissen...).

Die Standardshell bei openSUSE ist jedenfalls die `bash`, wie bei den allermeisten Distris.
Mit `chsh` kann man sich als User eine andere Shell zuordnen, wenn sie denn lauffähig installiert ist. `chsh` erlaubt es einem User die eigene Zeile in der /etc/passwd zu ändern, ohne Root-Rechte haben zu müssen.

Die meisten Shells sind dazu da, entweder Eingaben des Users auszuführen, oder stattdessen Scripte aufzurufen. Manche shells, wie z.B. `/sbin/nologin` sind spezialisiert auf nur einen Job. Trägt man in eine Userzeile im Shellfeld `/sbin/nologin` ein (siehe `man 5 passwd`), so kann sich dieser User nicht mehr am System anmelden, und wird höflich darüber informiert. (erstellt man dazu noch die optionale Datei `/etc/nologin.txt`, so wird dieser Text dem User auch noch vor die Nase gehalten.). Auch von diesen spezialisierten Shells gibt es einige. Und viele Programme bringen noch ihre eigene Shell mit. Sogar `zypper sh` ist eine interaktive Shell, innerhalb derer man alle zypper- Kommandos ausführen kann. Sehr nützlich, wenn man mal seine Paket und Repolisten bereinigen möchte.

Ich schreibe im Weiteren NUR über die `bash`, auch wenn viele Shells ganz ähnlich, und oft sogar gleich, arbeiten, so gibt es stellenweise doch subtile bis hin zu sehr drastischen Unterschieden.

Der Name Shell ist das englische Wort für Muschel. Wie eine Muschel legt sich die Shell, unser Kommandointerpreter, um alles, was der Kernel zur Verfügung stellt und macht Dienste und Programme dem User einfach bedienbar. Also relativ einfach. Die erste Shell gab es schon 1965, und viele Shells wurden anfangs der 70er geboren und werden seit dieser Zeit konstant weiterentwickelt.

Und was hat nun `su` und `sudo` damit zu schaffen?

Weil ein Linux nun mal von Haus aus ein Multitasking und Multiuser System ist, sind diese Shells ziemlich komplex, da sie nicht nur Programme ausführen können. Dazu müssen sie alle Sicherheitsvorgaben beachten können und sich entsprechend verhalten können. Jeder User soll ja -egal wieviele Rechte er auf dem System hat- seine Jobs immer mit den dafür benötigten Rechten ausführen können, und möglichst auch kein Jota mehr als wirklich dafür benötigte Rechte.
Dazu wird für jedes Programm ein Prozess erzeugt, dem eine Umgebung (ein `env`ironment) zugordnet ist. Und noch Verwaltungsinformationen des Kernels. Die bestehen aus Rechten und Laufzeitinformationen. (Man kann einzelnen Prozessen geringere Proiorität einräumen, damit wichtige Prozesse ungestört ihren Job erledigen können, ohne von unwichtigen Prozessen unterbrochen zu werden. (siehe `man nice`).
Einige Envrionmentvariablen können vom User geändert werden, andere nicht. (Es gibt noch ein paar spezielle Variablen, die jeweils eine Sonderfunktion zur Verfügung stellen. So gibt ein `echo $SECONDS` die Anzahl der Sekunden aus, die seit dem Start dieser Shellinstanz vergangen sind, oder ein `echo $RANDOM` eine zufällige Integerzahl zwischen 0 und 32767.)
Der Befehl `env` hat zwei Aufgaben. Ohne Argument gibt er schlicht alle Umgebungsvariablen aus, die für den User relevant sind, aber nicht alle, die existieren. Einige sind so fix, dass sie nicht einmal angezeigt werden. Ruft man `env` mit einem Programmnamen auf, so durchsucht er das dem User zugängliche Envrionment nach genau diesem angegebenen Befehl und versucht ihn dann aufzurufen.

Damit wir jetzt Prozesse starten können, die in einer anderen Prozessumgebung laufen, gibt es die beiden Tools `su` und `sudo`. Der Unterschied zwischen beiden liegt schon im Namen `su` heßt **S**witch**U**serkontext und `sudo` **S**witch**U**serkontextAnd**DO**. Die Arbeitsweise ist sehr ähnlich, aber es gibt einen gravierenden Unterschied. Um den verstehen zu können, müssen wir noch einen Überblick haben, was eine Shell beim

Aufruf eines Programmes alles erledigt, und welche Dinge dafür relevant sind. Weshalb wir uns kurz angucken,

wie eine Shell generell arbeitet.

Die `bash` kennt viele Modi, in denen sie arbeiten kann, und sie berücksichtigt beim Aufruf viele sogenannte "Dotfiles". Die dienen per traditioneller Konvention der Konfiguration. Laut dem Sourcecode der `bash` Die kennt folgende
"Konfigurationsdateien": `/etc/profile` , `/etc/bash.bashrc` , `~/.bash_profile` , `~/.bash_login` und zu guter Letzt gibt es noch ein Script, das ausgeführt wird, wenn die bash beendet wird: `~/.bash_logout` .
Letztlich sind alle diese Dateien gültige Shellscripte, die einfach "gesourced" werden. Steht in irgendeinem Shellscript eine Zeile, wie `source /Pfad/zu/irgendeinem/Shellscript` (oder die dazu synonyme Kurzform `. /Pfad/zu/irgendeinem/Shellscript`), so wird diese Zeile durch den gesamten Inhalt dieses Scripts ersetzt, also somit anstelle dieser Sourcezeile ausgeführt.

Es gibt so viele Dateien, weil ja völlig verschiedene Shellszenarien bedient sein wollen. Die Umgebung einer Shell hängt nicht nur von den Rechten des jeweiligen Users ab, sondern natürlich auch noch von anderen Faktoren. Ein User, der direkt vor dem Keyboard eines Linuxrechners sitzt, hat eine andere Shellumgebung, wie ein User, der sich via ssh über das Netzwerk einloggt. Es gibt noch ein paar Umstände, die zu berücksichtigen wären, wie z.B. ein System, auf dem teilweise User in einer `chroot` Umgebung arbeiten, andere nicht.

Und weil das immer noch nicht kompliziert genug ist, sourced die Bash noch ganze Verzeichnisse, wie z.B. `/etc/bash_completion.d/*` , in dem jede Menge Shellscripts liegen, die in der Shell dann Completion-Funktionen bereitstellen.
Und natürlich kann jeder User noch selbst kräftig rumsourcen.
Es ist also nicht ganz klar, ich welcher Reihenfolge welche Dateien bei der Konfiguration einer gerade aufgerufenen Shell mitspielen.
Außerdem könnte es auch noch sein, dass ein User ein Script gänzlich ohne ein Terminal laufen lässt, sich also weder über eine Loginshell, sei die lokal oder remote gerufen. Ein "detached"er Hintergrundprozess. `screen` wäre da so ein typisches Beisiel dafür.

Wäre nun das alles leicht verstehbar und sofort nachprüfbar, so wird die Sache vollends verwirrend,

weil jede Shell verschiedene Modi kennt.

Die Bash versteht, wie die allermeisten GNU- Tools lange und kurze Optionen (-l --lange-option),
Wir hätten da zu bieten:

1. **Der POSIX Mode**
   POSIX == **P**ortable**O**perating**S**ystem**I**nterface(e**X**change) ist eine Programmierschnittstelle, die sicherstellen soll, dass Programme auf einem kompatiblen System auch problemlos laufen und die Daten dadurch ebenfalls austauschbar werden. Die Bash stellt in diesem Modus für Shellscripte eine leicht eingeschränkte Syntax zur Verfügung und verhält sich in subtilen Kleinigkeiten etwas anders, als im normalen Bash- Modus. Linux Distributionen sind nicht ganz POSIX- konform, aber weitestgehend. Man kann diesen Modus einschalten indem man die Environmentvariable POSIXLY_CORRECT auf 1 setzt, die Option -p oder --posix angibt, oder als Shebangzeile nicht `#!/bin/bash` , sondern `#!/bin/sh` angibt.
2. **Der Restricted Mode**
   tut genau, was er sagt. root kann die Befehle angeben, die der User in dieser Shell aufrufen kann, oder eben nicht.
3. **Loginshell**
   Wenn ein User sich einloggt, ist seine erste Shell die, die meist in der `/etc/passwd` angegeben ist. (Man kann diese Infos auch über andere Mechanismen z.B. LDAP, PAM oder sonstwie managen).
4. **Interaktiver oder nichtinteraktiver Modus**
   Wenn die Bash ein Script ausführt, läuft sie nichtinteraktiv, ist mit ihr aber ein Keyboard (mit Bildschirm) verbunden läuft sie interaktiv in einem Terminal.

Und diese Modi kann man natürlich auch noch fast nach Belieben mischen. Außerdem spucken noch die Konfiguration anderer Loginmechanismen, wie z.B. beim Login via SSH, wo die Konfiguration des SSHD

Dämons ein gewaltiges Wörtchen bei der Konfiguration der aufzurufenden Shell mitredet.

Relativ einfach kann man zwar für jeden Modus angeben welche Konfigdateien in welcher Reihenfolge verarbeitet werden, aber root und User fummeln ja immer drin rum.
Man wird also auf jeden System eine ganze Menge an Dateien durchlesen müssen, um wirklich sicher festzustellen, was, wann, wo passiert.

Was passiert nun mit `su` und `sudo`?

Gibt man in einem Terminal `su` oder `sudo` ein, so wird eine weiter Shell gestartet. Bei `su` eine interaktive, bei `sudo` eine nicht interaktive. Also schon mal zwei leicht verschiedene Umgebungen. Zudem folgt openSUSE der reinen Lehre. Der Befehl `sudo` ist wirklich nur dazu da einen Befehl auszuführen, der ganz genau samt seinen Optionen und Argumenten festgelegt ist. (Ein Befehl kann selbstverständlich seinerseit ein Shellscript sein). Die Zuordnung von ganz exakt definierten Kommandos zu Users erfolgt in der Datei `/etc/sudoers` und kann bei openSUSE auch leicht mit YaST erledigt werden.

Im Gegensatz zu Ubuntu, wo es einen root- Account nach einer Standardinstallation nicht einmal gibt. Dort kann man NUR mit sudo Root-Rechte bekommen. Dort ist die Datei `/etc/sudoers` so eingerichtet, dass jeder sudo- Aufruf vernünftig zu Root-Rechten führt.
Bei openSUSE ganz bestimmt NICHT.
Auch wenn die Unterschiede sehr subtil sind, und es eine richtige Wissenschaft werden kann, die jeweils gültige exakte Shellumgebung herauszufinden.

Aber man kann das auf

eine sehr einfache Grundregel zum Umgang mit `su` und `sudo` reduzieren:

1. ### unter openSUSE
   Finger weg von sudo!!! Das Wort aus dem Alphabet streichen!!! Es sei denn man hötte die /etc/sudoers selbst sauber dafür konfiguriert.
   Nur und ausschließlich eines, der drei völlig synoymen Kommandos verwenden: `su -` oder `su -l` oder `su --login`
2. ### unter Ubuntu
   `su` aus dem Wortschatz streichen und immer `sudo` vor jeden Befehl setzen. Oder erst einmal einen Root- Account hinzufügen, alle Rechte zuteilen und dann....

Nachtrag:
Einen Punkt, den ich ganz vergessen habe, ist, dass eine Shell auch bestimmte Variablen auswertet oder setzt, je nachdem in welchem Terminal sie aufgerufen wird. Das verkompliziert die Sache noch einmal ungemein. Denn auch Terminals gibt es wie Sand am Meer und die Konfigurationen davon uferlos.