Nr. 1b)

1. a) Beim mehrfach Schicken wird einfach der Handler mehrmals aufgerufen und gibt in unserem Fall die entsprechende Signal-nummer (hier 14) aus, solange der Prozess dabei nicht abgebrochen wird.

1. b) Beim gesendeten Befehl SIGPFE (illegale mathematische Operation) sieht man in der Kommandozeile nichts, da es nicht abgefangen wird, aber Python wird vom Betriebssystem beendet und man bekommt in der GUI eine Fehlermeldung, dass das Programm unerwartet beendet wurde. Beim Befehl SIGPWE bekommt man mit jobs die Meldung „Fehler in Stromversorgung“ und das Programm wird angehalten.

2.

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

3. Hier Signal Kill: Nein, es wird nur dann ein Prozess gekillt, wenn ein lokaler Prozess oder der Kernel selbst das Signal sendet. Es gibt keinen Standard-Daemon, dem man über remote anweisen kann, lokal ein Signal zu senden. Wenn man sich selbst an dem Rechner authentifiziert, forkt man in eine neue Shell, dann wäre es aber nicht mehr remote.

4. a) Idee: Handler schreiben, der nichts macht.

4. b) Seit Python 3 sind Signalkonstanten als Enum definiert. Entsprechend können wir mit einer foreach-Schleife über alle Signale iterieren und so einen default-handler konstruieren (siehe Quelltext). SIGKILL und SIGSTOP können nie abgefangen werden, deswegen schmeißen wir sie aus dem Set über das wir iterieren.

Nr. 2: 1. Ablauf Hardware Interrupts:

- Hardware signalisiert auf Hardware-Ebene (Signalleitung ändert Spannung) dass etwas passiert
- CPU beendet Befehl den sie gerade bearbeitet
- nach jedem Befehl wird geschaut, ob sich am Strompegel etwas geändert hat
- Flag wird gesetzt in einem Register (Interruptregister) = > CPU ist jetzt im Interrupt-Modus
- CPU muss Hardware signalisieren, dass Interrupt angenommen wurde (zweite Signalleitung)
- Hardware legt auf Datenbus die Adresse des Interrupthandlers (beim Start des Programms initialisiert)
- CPU liest vom Datenbus die Adresse und springt in den Interrupthandler
- Register mit denen gerade gearbeitet wurden, werden vom Interrupthandler gesichert => Kontextwechsel
- Befehlszähler von CPU wird auf die Anfangsadresse vom Interrupthandler gesetzt
- Abarbeitung des Interrupts
- Rückabwicklung des Kontextwechsels
- Verlassen der Interruptregister
- Scheduler noch einmal aufrufen, um neuen hoch priorisierten Prozess zu berücksichtigen
- normalen Prozess fortsetzen

2. Der Interrupt-Vektor ist zusammengesetzt aus einem oberen und einem unteren Teil. Der obere Teil ist die Adresse der Interrupt-Handler-Tabelle und der untere Teil ist der Adressteil vom Peripherie-Gerät. Der untere Adressteil wird von der Hardware auf den Datenbus gelegt und von der CPU gelesen. Beide Teile zusammen ergeben die Interrupt-Hanlder-Adresse. (siehe Folien Interrupt 9)

3. Ein Flag wird im Interrupt-Register gesetzt, sobald ein Interrupt festgestellt wird. Je nachdem wird ein Interrupt aktiviert oder deaktiviert, also Interrupts zulässt oder sperrt.

Nr. 3

1. Ausführreihenfolge ist gleich dem Skript.

2.

Create_process()	Prozess erstellen, in Prozessliste eintragen - Bekommt PID, Startzeit - Wird auf Ready gesetzt - behavior[0, 2, 4, …] sind CPU Zeiten - behavior[1, 3, 5, …] sind I/O Zeiten - Wenn behavior[0] == 0 (entspricht keiner CPU Zeit) dann mit I/O beginnen und blocked setzen - An tasks anhängen (append) - Am Ende: return PID
Run_current()	Aktuellen Prozess eine Zeiteinheit rechnen lassen - Wenn noch nie gelaufen (firstruntime == -1): → firstruntime setzen mit current & cputime - Verbleibende Rechenzeit reduzieren (head_behavior) - cputime erhöhen - wenn Prozess fertig ist oder blockiert, wird er aus der runqueue entfernt - wenn Prozess nicht fertig ist, wird er der blocked-Liste hinzugefügt
Update_blocked_process()	Prozess nur bearbeiten, wenn er bereits im System ist und wenn er nicht gerade erst in die blocked queue geschoben wurde - Wenn PID nicht aktuelle ist: → verbleibende Wartezeit reduzieren, I/O-Time erhöhen - Wenn Wartezeit auf 0: → aktuelle I/O-Phase abgeschlossen, → Status von dem Aktuellen auf Ready setzen & aus blocked-Liste löschen - in Abhängigkeit davon, ob nach der I/O noch CPU-Phase benötigt werden, Prozess entweder auf Ready oder Done setzen