Betriebssysteme und Systemsoftware *Andrès Montoya, 405409*
Übungsblatt 01 *Til Mohr, 405959*
Tutorium 1 *Marc Ludevid Wulf, 405401*

Aufgabe 1.2

a) Der Kernel-Mode ist ein privilegierter Mode in dem die Module des Kernels ausgeführt werden. Er hat das auschliessliche Zugriffrecht auf den Kernel-Space, den Speicherbereich des Kernels. Der User-Mode hingegen ist für die Ausführung von Anwendungsprogrammen zuständig. Das wechseln zwischen den Modi geschieht über eine klar definierte Schnittstelle. Diese Unterschiedung schützt den Kernel vor Missbrauch durch Anwendungsprogramme. Oft wird der Kernel selbst ebenfalls in verschiedene Ebenen Unterteil um Modularität zu gewährleisten.

b) Der Aufruf einer Trap kann im User-Modus passieren, der Trap selbst wird dann im Kernel-Modus ausgeführt. Aufrufe des Grafikentreibers hingegen sollten nur auf den Kernel möglich sein, da nicht jedes Anwendungsprogramm den Grafikentreiber direkt ansprechen sollte. Dieser Zugriff muss über eine vordefinierte Schnittstelle im Kernel geschehen. Den Zugriff zum Druckertreiber kann man auch aus dem User-Modus erlauben, da das ablegen von einer zu druckenden Datei keine systemkritischen Nebenwirkungen haben kann und somit nicht durch den Kernel kontrolliert werden muss.

c) Es ist sinnvoll Polling einzusetzen wenn das warten auf das Ergebnis eines Prozesses vorhersehbar ist, also das Zeitfenster sehr klein ist, wenn das Ergebnis keine Eile hat, dann kann man durch lange wartezeiten polling verwenden oder wenn man eine hohe Abfragrate braucht, z.B. Ablesen des Status eines I/O-Pins. Der Einsatz von Interrupts ist dann sinnvoll, wenn das Eintreten eines Ereignisses unvorhersehbar ist, aber möglichst schnell behandelt werden soll. Interrupts werden bespielsweise beim Drücken einer Taste auf der Tastatur eingesetzt. So wird keine CPU-Performance unnötig verbraucht.

Aufgabe 1.3

a) Der Pointer z wird auf das 4. Element des Arrays data gesetzt, also 'Z'.

b) Das 4. Element von data wird auf 'H' gesetzt. Damit zeigt nun auch z auf 'H'.

c) Das 2. Element von data wird auf 'S' gesetzt.

d) Der Pointer pi wird auf die Adresse von i gesetzt. Also zeigt pi nun auf i.

e) Diese Anweisung kann man nicht ausführen, da man die Adresse einer Variable nicht setzen kann.

f) A[1] liefert einen Pointer auf den Array 2,3,2. Also ist *A[1] = 2. Daraus folgt, dass &data[*A[1]] der Pointer auf das 2. Elements von data liefert. Davon wird der Wert von *pi = 1 abgezogen. Damit ist z das Element an der 1. Stelle von data, also 'u'.

Aufgabe 1.5

a) Der Grund für scheinbar unterschiedliche Variablenwerte ist auf Überdeckung von globalen Variablen und auf Funktionen, die unerwartete Werde ausgeben, zurückzuführen. Wie in der Aufgabenstellung erklärt, überdecken lokale Variablen globale Variablen mit derselben bezeichnung.

Zeile 1: Hier werden die in der Methode `main` deklarierten und initialisierten lokalen Variablen `a` und `b`, also `a=5` und `b=6`.

Zeile 2: In der Methode `summe` werden nicht die Parameter, sondern die globalen Variablen `a` und `b` ausgegeben, also `a=1` und `b=2`.

Zeile 3: Die Methode `diff` gibt in dieser Methode die lokal deklarierten und initialisierten Variablen `a=4` und `b=3` aus.

Man sieht, bei jeder Ausgabe handelt es sich um komplett verschiedene Variablen.

Aufgabe 1.6

a) Ein **Buffer Overflow** ergibt sich dann, wenn mehr Daten in einen Buffer (z.B. array) geschrieben werden als der Buffer groß ist. Hierdurch können hinter dem Buffer liegende Programmteile (z.B. andere Daten) überschrieben werden.

b) In `password_checker.c` kann man sich mithilfe von Buffer Overflow unberechtigten Zugriff verschaffen. Da `*access` im Arbeitspeicher direkt hinter `*password_input` liegt, kann man mithilfe von geschickten Eingaben wie `"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa1"` (`password_input` und `access` sind im Arbeitsspeicher 32 byte entfernt) `*access` auf `true` setzen und sich so Zugriff verschaffen.

c) Um diese Sicherheitslücke auf diese Art zu verhindern, könnte man einfach zuerst den boolean allocaten, danach den String:

```
bool *access = malloc(sizeof(bool));
char *password_input = malloc(BUFFER_SIZE * sizeof(char));
```

Jedoch kann hier immer noch Buffer Overflow auftreten und womöglich andere Programmteile schädigen. Eine intelligentere Version ist während der String-Eingabe die Anzahl der eingebenen Zeichen mitzuzählen um so einen Buffer Overflow zu verhindern:

```
int i = 0;
char v = getchar();
while(v != '\n') {
        if(i < BUFFER_SIZE) {
                *(password_input + i) = v;
                i++;
        }
        v = getchar();
}
```