1 System Calls

1.1 Allgemeines

Zum Zugriff auf Funktionen des Kernels benutzen Prozesse (dies beinhaltet sowohl Anwendungen als auch Kernelmodule) System Calls. Ein System Call ist der Aufruf einer Kernelfunktion über Interrupt 0x30. Dabei gelten folgende Konventionen:

- Die Funktionsnummer wird im Register *eax* übergeben.
  Fehlerverhalten: Ein Prozeß, der eine undefinierte Funktionsnummer aufruft, wird sofort beendet.
- Die Parameter werden in umgekehrter Reihenfolge der C-Deklaration der aufzurufenden Funktion auf den Stack abgelegt, so daß der vorderste Parameter auf dem Stack oben ist.

  Die Parameter bleiben für den Aufrufer erhalten (und der Aufrufer ist entsprechend auch dafür verantwortlich, die Parameter nach dem Funktionsaufruf wieder von Stack zu entfernen).
- Ein Rückgabewert wird im Register *eax* übergeben. Bei einer Funktion ohne Rückgabewert ist der Inhalt von *eax* nach dem Funktionsaufruf undefiniert.

1.2 Speicherverwaltung

1.2.1 allocate

Funktion: 1 – allocate

C-Prototyp: `void* allocate(size_t size, int flags)`

Parameter:

`size`	Größe des zu reservierenden Speicherbereichs in Bytes. Wenn die Größe nicht durch 4096 teilbar ist, wird auf das nächste Vielfache von 4096 aufgerundet.
`flags`	Spezifiziert zusätzliche Anforderungen an den Speicherbereich: Bit 0: Für DMA tauglich Bit 1: Auslagerung verhindern Bit 2: Ausführbar Bit 3: Beschreibbar

Rückgabewert: Pointer auf den Anfang des reservierten Speicherbereichs

Fehlerverhalten: –

1.2.2 free

Funktion: 2 – free

C-Prototyp: `void free(void* ptr)`

Parameter:

`ptr`	Pointer auf den Anfang einer für den aufrufenden Prozeß reservierten Speicherseite

Rückgabewert: –

Fehlerverhalten: Wenn *ptr* nicht auf den Anfang einer Speicherseite verweist oder die Speicherseite nicht reserviert oder nicht dem aufrufenden Prozeß zugeordnet ist, wird der aufrufende Prozeß beendet.

1.3 Prozeßverwaltung

1.3.1 create_process

Funktion: 3 – create_process

C-Prototyp:
```
word create_process(void* image_base, size_t image_size,
dword initial_eip, dword uid)
```

Parameter:

`image_base`	Pointer auf den Anfang der geladenen Binary
`image_size`	Größe der geladenen Binary
`initial_eip`	Startpunkt der Ausführung relativ zu *image_base*
`uid`	Benutzer-ID, unter der der neue Prozeß laufen soll.

Rückgabewert: Die PID des neu erzeugten Prozesses bei Erfolg, ansonsten false

Fehlerverhalten: Wenn *uid* von aufrufendem und zu erzeugendem Prozeß nicht übereinstimmen, und der aufrufende Prozeß nicht UID 0 hat, wird false zurückgegeben und kein neuer Prozeß erzeugt.
Wenn *eip* > *image_size*, wird false zurückgegeben und kein neuer Prozeß erzeugt.
Wenn der Speicherbereich von *image_base* bis *image_base* + *image_size* nicht dem aufrufenden Programm zugeordnet ist, wird der aufrufende Prozeß beendet und kein neuer Prozeß erzeugt.

1.3.2 fork

Noch nicht definiert.

1.3.3 exit_process

Funktion: 5 – exit_process

C-Prototyp: `void exit_process(dword pid, int returncode`

Parameter:

`pid`	PID des zu beendenden Prozesses, 0 für den aufrufenden Prozeß
`returncode`	Rückgabewert des Prozesses

Rückgabewert: –

Fehlerverhalten: Wenn der zu beendende Prozeß und der aufrufende Prozeß nicht dieselbe UID haben, wird der aufrufende Prozeß beendet.

Beendet einen Prozeß. Dazu gehören insbesondere auch die Freigabe aller vom zu beendenden Prozeß belegter Ressourcen (d.h. Ports und Speicherbereiche). Außerdem wird ein Event ausgelöst, durch das Module auf das Prozeßende reagieren und möglicherweise weitere Ressourcen freigeben können.

1.3.4 sleep

Funktion: 6 – sleep

C-Prototyp: `void sleep(word milliseconds)`

Parameter:

`milliseconds`	Anzahl der Millisekunden, die der Prozeß pausieren soll

Rückgabewert: –

Fehlerverhalten: –

1.3.5 get_uid

Funktion: 7 – get_uid

C-Prototyp: `int get_uid(dword pid)`

Parameter:

`pid`	PID des Prozesses, dessen UID zurückgegeben werden soll, 0 für den aufrufenden Prozeß

Rückgabewert: UID des gegebenen Prozesses

Fehlerverhalten: Wenn kein Prozeß mit der übergebenen PID existiert, wird 0 zurückgegeben.

1.3.6 set_uid

Funktion: 8 – set_uid

C-Prototyp: `int set_uid(dword pid, dword uid)`

Parameter:

`pid`	PID des Prozesses, dessen UID geändert werden soll, 0 für den aufrufenden Prozeß
`uid`	Zuzuweisende UID

Rückgabewert: true bei Erfolg, false sonst

Fehlerverhalten: Wenn der aufrufende Prozeß nicht die UID 0 besitzt, wird die UID nicht geändert und false zurückgegeben.
Wenn kein Prozeß mit der übergebenen PID existiert, wird die UID nicht geändert und false zurückgegeben.

1.3.7 request_port

Funktion: 9 – request_port

C-Prototyp: `int request_port(word port, int timeout)`

Parameter:

`port`	Portnummer, die reserviert werden soll
`timeout`	Zeit in Millisekunden, die auf eine Freigabe des Ports gewartet werden soll, wenn der Port von einem anderen Prozeß reserviert ist. 0 für sofortigen Abbruch, wenn der Port besetzt ist.

Rückgabewert: true, wenn der Port reserviert werden konnte, false sonst

Fehlerverhalten: –

1.3.8 release_port

Funktion: 10 – release_port

C-Prototyp: `void ReleasePort(word port)`

Parameter:

`port`	Portnummer des freizugebenden Ports

Rückgabewert: –

Fehlerverhalten: Wenn der freizugebende Port nicht vom aufrufenden Prozeß reserviert ist, wird der aufrufende Prozeß beendet.

1.4 Interprozeßkommunikation

1.4.1 set_rpc_handler

Funktion: 50 – set_rpc_handler

C-Prototyp: `void set_rpc_handler(void (*rpc_handler)())`

Parameter:

`rpc_handler`	Pointer auf die Funktion, die RPC-Aufrufe verarbeitet

Rückgabewert: –

Fehlerverhalten: –

Der RPC-Handler wird als `void rpc_handler()` übergeben, da sein Aufruf nicht der C-Aufrufkonvention entspricht, tatsächlich erhält er Parameter. Er ist folglich wahrscheinlich nur in Assembler implementierbar.

Insbesondere zu beachten ist für den RPC-Handler, daß er die Maschine vor dem Rücksprung wieder in den Originalzustand versetzen muß, d.h. insbesondere, daß alle Register gesichert und zurückgeschrieben werden müssen. Andernfalls ist nach dem Rücksprung in den eigentlichen Programmcode ein Absturz des Prozesses wahrscheinlich.

Der RPC-Handler findet folgende Situation auf dem Stack vor:

(%esp)	*data_size*	Größe der zusätzlich übergebenen Daten
4(%esp)	*caller_pid*	Prozeß-ID des aufrufenden Prozesses
8(%esp)	*data*	Zusätzlich übergebene Daten
...		
8+*data_size*(%esp)		Rücksprungadresse

1.4.2 rpc

Funktion: 51 – rpc

C-Prototyp: `void rpc(dword callee_pid, dword data_size, char* data)`

Parameter:

`callee_pid`	PID des Prozesses, in dem eine Funktion aufgerufen werden soll
`data_size`	Größe der zu übergebenden Daten
`data`	Zeiger auf die zu übergebenden Daten

Rückgabewert: –

Fehlerverhalten: Wenn *data_size* größer als 4096 Bytes ist, wird der aufrufende Prozeß beendet. Wenn der aufzurufende Prozeß keinen RPC-Handler installiert hat, wird der aufrufende Prozeß beendet.

1.4.3 add_interrupt_handler

Funktion: 52 – add_interrupt_handler

C-Prototyp: `void add_intr_handler(dword intr)`

Parameter:

`intr`	Interrupt, über den der aufrufende Prozeß informiert werden soll. Der RPC-Handler erhält bei einer Interrupt-Information ein Datenpaket von 4 Bytes, das die Interruptnummer als dword enthält.

Rückgabewert: –

Fehlerverhalten: –