

Prozesserzeugung & Synchronisation

FH Münster

Probevorlesung

Dr.-Ing. Darius Malysiak

Inhalt

I. Motivation / Lernziel

II. Prozesserzeugung

1. Was sind Prozesse?
2. Wie werden Prozesse erzeugt (Linux)?

III. Synchronisation von Prozessen

1. Warum überhaupt Synchronisieren?
2. Möglichkeiten zur Synchronisation

IV. Gemeinsame Übung

I. Motivation / Lernziel

Lernziele:


- Vermittlung der Begrifflichkeiten zu Prozessen und Synchronisationsproblemen.
- Verständnis vermitteln zur Kommunikation zwischen Prozessen.
 - Verständnis vermitteln zu daraus resultierenden Problemen und deren Lösung.
- Erklärung der Zusammenhänge zwischen Theorie und Implementierung in Linux.
- Schaffen einer Basis für das Selbststudium.

II. Prozesserzeugung

1. Was sind Prozesse?

Allgemein gesprochen führt ein Computer System spezifische Aufgaben aus:

- Ein Batch System (z.B. Mainframe) führt Jobs gesammelt aus:
 - Daten Transaktionen
 - Datenprüfungen
 - Daten Analysen
- Ein System mit Time-Sharing (z.B. Desktop Computer) führt Programme oder Tasks aus:
 - Anwenderprogramme
 - Dienste
 - Periodische Aufgaben



Jobs, Tasks,
Programme
müssen im
Computer
dargestellt
werden.

Analogie aus der OOP: Klassen werden durch Objekte „zum Leben erweckt“.

II. Prozesserzeugung

1. Was sind Prozesse?

Eine Aufgabe wird für den Computer in Form eines Programms ausgedrückt

```
#include<stdio.h>
```

```
char a[] = "Hello World\n";  
int i;
```

```
int main() {  
    i=0;  
    printf(a);  
    return i;  
}
```

Programmcode / Programm

Compiler



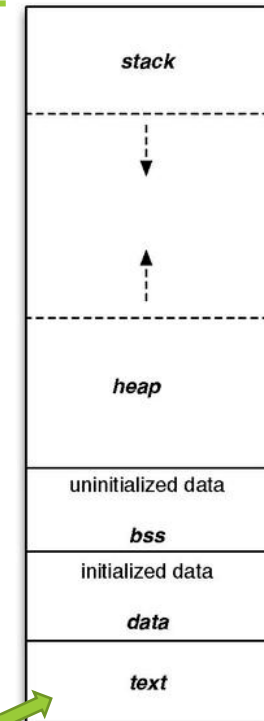
Datei „main“

./main



Virtueller
Adressraum

Prozess



0000000000001139 <main>:

1139: 55

113a: 48 89 e5

113d: 48 8d 05 c0 0e 00 00

1144: 48 89 c7

1147: e8 e4 fe ff ff

114c: b8 00 00 00 00

1151: 5d

1152: c3

push %rbp

mov %rsp,%rbp

lea 0xec0(%rip),%rax

mov %rax,%rdi

call 1030 <puts@plt>

mov \$0x0,%eax

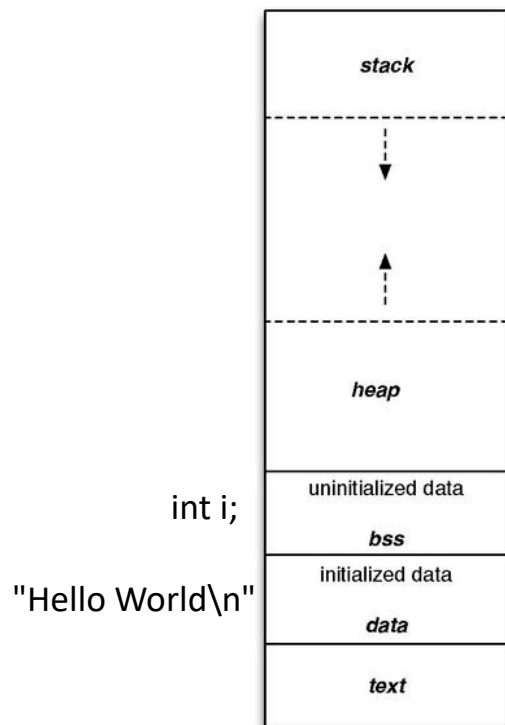
pop %rbp

ret

2004 <_IO_stdin_used+0x4>

II. Prozessorzeugung

1. Was sind Prozesse?



- **Stack**: Hier werden lokale Funktionsvariablen / Funktionsdaten abgelegt, dazu noch Verwaltungsdaten wie z.B. die Rücksprungsadresse. Beim Verlassen einer Funktion werden diese Daten entfernt.
- **Heap**: Hier werden globale Funktionsdaten oder größere Datenmengen abgelegt. Der Speicher wird nicht automatisch verwaltet und kann Fragmentierung entwickeln.
- **BSS**: Hier werden globale / lokale statisch allokierte, jedoch nicht initialisierte Variablen abgelegt. Dieser Bereich wird zur Laufzeit mit entsprechenden Daten befüllt.
- **DATA**: Hier werden globale / lokale statisch allokierte Variablen abgelegt, welche bereits initialisiert sind. Dieser Bereich wird zur Laufzeit mit entsprechenden Daten befüllt, kann jedoch bereits in der ausführbaren Datei betrachtet werden.

II. Prozesserzeugung

2. Wie werden Prozesse erzeugt? (Linux)


Ein Prozess wird unter Linux erzeugt indem:

1. Der aktuell laufende Prozess eine Kopie von sich erstellt. (Fork)
2. Das Betriebssystem die Kopie mit dem Inhalt des entsprechenden Programms befüllt. (Exec)

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1	root	20	0	21216	10464	5248	S	0.0	0.0	3:47.86	/sbin/init
424	root	20	0	31464	5484	3260	S	0.0	0.0	0:03.50	/usr/lib/systemd/systemd-udevd
509	dbus	20	0	9128	3348	2292	S	0.0	0.0	0:01.49	/usr/bin/dbus-daemon --system --address=systemd: --nofork --r
511	root	20	0	48900	5144	4104	S	0.0	0.0	0:00.99	/usr/lib/systemd/systemd-logind
613	root	20	0	7242M	70576	17396	S	1.2	0.1	3h10:10	/usr/bin/dockerd -H fd://
632	root	20	0	7242M	70576	17396	S	0.0	0.1	10:18.93	/usr/bin/dockerd -H fd://
633	root	20	0	7242M	70576	17396	S	0.0	0.1	7:18.18	/usr/bin/dockerd -H fd://
634	root	20	0	7242M	70576	17396	S	0.0	0.1	0:00.20	/usr/bin/dockerd -H fd://
635	root	20	0	7242M	70576	17396	S	0.0	0.1	10:02.27	/usr/bin/dockerd -H fd://
636	root	20	0	7242M	70576	17396	S	0.0	0.1	0:00.00	/usr/bin/dockerd -H fd://
637	root	20	0	7242M	70576	17396	S	0.0	0.1	0:00.00	/usr/bin/dockerd -H fd://
638	root	20	0	7242M	70576	17396	S	0.0	0.1	0:00.39	/usr/bin/dockerd -H fd://
639	root	20	0	7242M	70576	17396	S	0.0	0.1	0:00.26	/usr/bin/dockerd -H fd://
640	root	20	0	7242M	70576	17396	S	0.0	0.1	0:00.23	/usr/bin/dockerd -H fd://
641	root	20	0	7242M	70576	17396	S	0.0	0.1	0:00.01	/usr/bin/dockerd -H fd://
642	root	20	0	7242M	70576	17396	S	0.0	0.1	6:29.24	/usr/bin/dockerd -H fd://
643	root	20	0	7242M	70576	17396	S	0.0	0.1	0:00.00	/usr/bin/dockerd -H fd://

II. Prozesserzeugung

2. Wie werden Prozesse erzeugt? (Linux)

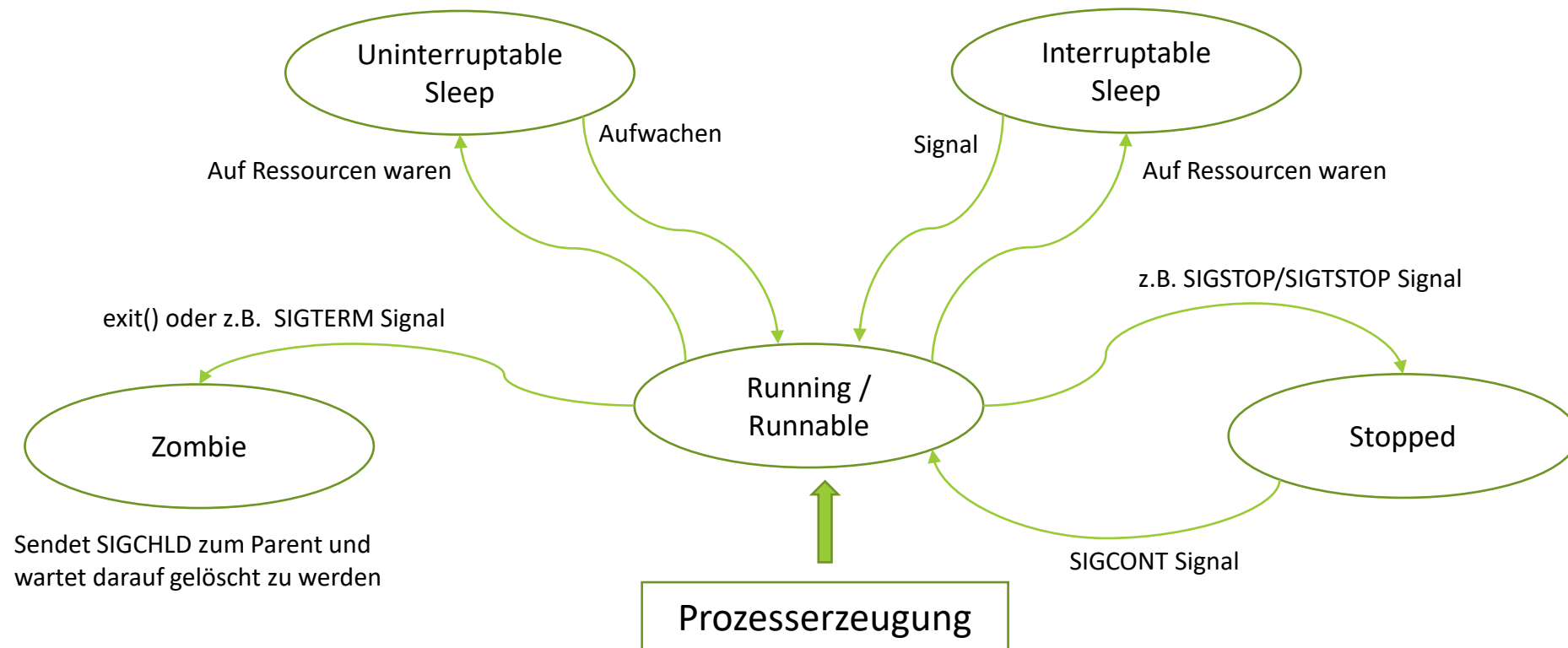
```
int main() {  
    pid_t pid = fork();  Fork  
  
    if (pid == -1) {  
        printf("Failed to fork\n");  
    }  
    else if (pid > 0) {  
        int status;  
        printf("Parent: waiting for child\n");  
        waitpid(pid, &status, 0);  
        printf("Parent: child finished work\n");  
    }  
    else {  
        printf("Child: sleeping for 5s\n");  
        sleep(5);  
        printf("Child: woke up\n");  
    }  
    return 0;  
}
```

- Mit ‚getpid‘ wird die eigene ID (des Elternprozesses) abgefragt.
- Die Funktion ‚fork‘ führt einen Fork und erzeugt eine Kopie von sich als Kind-Prozess.
- Der Elternprozess erhält seine ID zurück, der Kindprozess erhält ‚0‘ zurück.

II. Prozesserzeugung

2. Wie werden Prozesse erzeugt? (Linux)

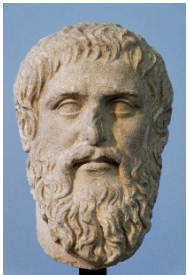
Ein Lebenszyklus eines Prozesses folgt einem Zustandsautomaten



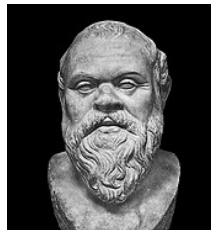
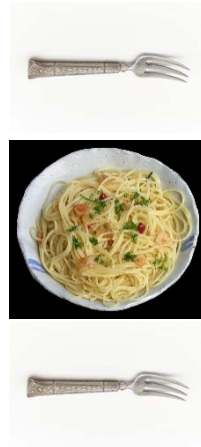
II. Synchronisation von Prozessen

1. Warum überhaupt Synchronisieren?

Deadlocks



Person 1



Person 2

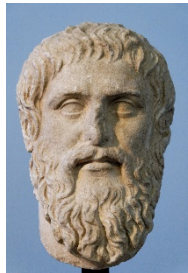
- Beide Personen möchten essen.
- Zum Essen benötigt jede Person zwei Gabeln.
- Sobald eine Person fertig ist legt sie die Gabeln zurück.

Falls beide Personen gleichzeitig ihre jeweils linke / rechte Gabel greifen wird nie gegessen!

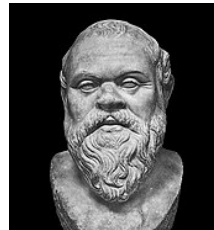
II. Synchronisation von Prozessen

1. Warum überhaupt Synchronisieren?

Korrupte Daten



Person 1



Person 2



S	i	k	t	a	t	e	s	
---	---	---	---	---	---	---	---	--

- Jede Person will ihren Namen ins Textfeld schreiben.
- Jede Person prüft zuvor ob das Feld leer ist und schreibt dann hinein ohne erneut zu prüfen.

Falls beide Personen gleichzeitig prüfen so ist das Feld für sie leer. Selbst wenn die Personen abwechselnd schreiben so ist das Ergebnis nicht nutzbar!

II. Synchronisation von Prozessen

2. Möglichkeiten zur Synchronisation

Deadlocks

```
pthread_t tid[2];
```

```
void* eat(void *arg)
```

```
{
```

```
    sleep(rand() % 2);
```

Kritischer Abschnitt

```
    printf("Person %d eating\n", arg);
```

```
    sleep(5);
```

```
    return NULL;
```

```
}
```

```
int main(void)
```

```
{
```

```
    int err;
```

```
    srand(time(NULL));
```

```
    for(int i=0;i<2;i++)
```

```
    {
```

```
        err = pthread_create(&(tid[i]), NULL, &eat, (void*)i);
```

```
        if (err != 0)
```

```
            printf("\ncan't create thread :[%s]", strerror(err));
```

```
    }
```

```
    pthread_join(tid[0], NULL);
```

```
    pthread_join(tid[1], NULL);
```

```
    return 0;
```


```
}
```

II. Synchronisation von Prozessen

2. Möglichkeiten zur Synchronisation

Deadlocks

```
pthread_t tid[2];  
pthread_mutex_t lock;  
  
void* eat(void *arg)  
{  
    sleep(rand() % 2);  
    pthread_mutex_lock(&lock);    Kritischer Abschnitt  
    printf("Person %d eating\n", arg);  
    sleep(5);  
    pthread_mutex_unlock(&lock);  
    return NULL;  
}
```



Ein Mutex ist eine Datenstruktur mit folgenden Eigenschaften:

- ‚lock‘ sperrt einen Mutex, ‚unlock‘ entsperrt diesen.
- ‚lock‘ und ‚unlock‘ sind atomar.
- Wird versucht ‚lock‘ auf einem bereits gesperrten Mutex aufzurufen so wird an dieser Stelle gewartet bis der Mutex wieder entsperrt ist.

II. Synchronisation von Prozessen

2. Möglichkeiten zur Synchronisation

Deadlocks

```
void* eat(void *arg)
```

```
{
```

```
    sleep(rand() % 2);
```

Eingangsbereich

```
    printf("Person %d eating\n", arg);  
    sleep(5);
```

Kritischer Abschnitt

```
    return NULL;
```

Ausgangsbereich

```
}
```

II. Synchronisation von Prozessen

2. Möglichkeiten zur Synchronisation

Im Allgemeinen muss eine Lösung für den Zugriff auf kritische Blöcke folgende Bedingungen erfüllen:

- Gegenseitiger Ausschluss:
Falls ein Prozess P den kritischen Bereich betreten hat so darf kein anderer Prozess diesen betreten.
- Fortschritt:
Falls kein Prozess den kritischen Bereich betreten hat aber weitere Prozesse diesen betreten möchten, so dürfen nur Prozesse außerhalb des Ausgangsbereich über den Eintritt entscheiden. Die Auswahl des Prozesses darf nicht unendlich lange dauern.
- Endliches Warten:
Falls ein Prozess den kritischen Bereich betreten hat und ein weiterer Prozess diesen betreten möchte so existiert ein Limit für die Anzahl wie oft er ausgewählt werden darf, nach dem Limit muss ein evtl. anderer wartender Prozess ausgewählt werden.

Nur ein Prozess
im kritischen
Bereich

Nur Prozesse vor
dem kritischen
Bereich dürfen
mitentscheiden
wer den
kritischen Bereich
betritt.

Ein Prozess darf
nicht unendlich auf
Eintritt warten.

III. Gemeinsame Übung

Gegeben sind zwei Prozesse P0 und P1:

- i, j sind lokale int Variablen in P0 und P1
- P0 setzt i=0, j=1
- P1 setzt i=1, j=0
- Beide wollen den kritischen Abschnitt betreten.
- bool ready[2] ist eine globale Variable für beide Prozesse.
- int turn ist eine globale Variable für beide Prozesse.
- Zuweisungen für bool und int seien atomar.

Aufgabe: Prüfen Sie die Erfüllung von der Bedingung:

Nur ein Prozess im kritischen Bereich

Für den nebenstehenden Ansatz nach Peterson.

Lösungshinweis: Nehmen Sie an, dass beide Prozesse den kritischen Abschnitt betreten haben und leiten Sie daraus einen Widerspruch ab.

```
//Eingangsbereich
```

```
ready[i] = TRUE;  
turn = j;  
while (ready[j] && turn == j);
```

```
//Kritischer Abschnitt
```


```
ready[i] = FALSE;
```

```
//Ausgangsbereich
```


III. Gemeinsame Übung

Aufgabe: Angenommen P0 und P1 sind beide im kritischen Abschnitt.

Es gilt:

- Ein Prozess betritt den kritischen Abschnitt nur falls:
 - $\text{ready}[j] == \text{false}$ oder $\text{turn} == i$
- $\rightarrow \text{turn} == 0$ für P0 und $\text{turn} == 1$ für P1; das kann nicht sein
- \rightarrow es bleibt also nur $\text{ready}[1] == \text{ready}[0] == \text{false}$:
 - \rightarrow da beide Prozesse noch im kritischen Abschnitt sind folgt, dass P0 und P1 ready jeweils nicht auf TRUE gesetzt haben. 

```
//Eingangsbereich
```

```
ready[i] = TRUE;  
turn = j;  
while (ready[j] && turn == j);
```

```
//Kritischer Abschnitt
```

```
ready[i] = FALSE;
```

```
//Ausgangsbereich
```

III. Gemeinsame Übung

Geht es nicht auch mit einer Variable?

```
//Eingangsbereich
```

```
ready[i] = TRUE;  
while (ready[j]);
```

```
//Kritischer Abschnitt
```

```
ready[i] = FALSE;
```

```
//Ausgangsbereich
```

Deadlock möglich da beide in Endlosschleife verbleiben falls ready[0/1] gleichzeitig auf TRUE gesetzt wurden

```
//Eingangsbereich
```

```
turn = j;  
while (turn == j);
```

```
//Kritischer Abschnitt
```

```
turn = i;
```

```
//Ausgangsbereich
```

Falls P0 als erste dran kommt so hat P1 ihm dies erlaubt. Wenn P0 fertig ist so gibt er P1 den Zugriff auf den kritischen Abschnitt nicht frei, dies passiert erst wenn P0 erneut versucht den kritischen Abschnitt zu betreten.

III. Gemeinsame Übung

Geht es nicht auch mit einer Variable? – Erneuter Versuch

```
//Eingangsbereich
```

```
turn = j;  
while (turn == j);
```

```
//Kritischer Abschnitt
```

```
turn = j;
```

```
//Ausgangsbereich
```

Falls P0 als erste dran kommt so hat P1 ihm dies erlaubt. Wenn P0 fertig ist so gibt er P1 den Zugriff auf den kritischen Abschnitt frei; turn=1.

Falls P0 erneut den kritischen Abschnitt betreten möchte so setzt er turn=1 evtl. zum Zeitpunkt kurz nachdem P1 turn=0 gesetzt hat nachdem P1 fertig war. P0 wartet nun, falls P1 nie wieder versucht den kritischen Abschnitt zu betreten so wartet P0 unendlich lange. → Starvation (unendliches Warten auf eine Resource).

~~Falls P0 als erste dran kommt so hat P1 ihm dies erlaubt.
Wenn P0 fertig ist so gibt er P1 den Zugriff auf den
kritischen Abschnitt nicht frei, dies passiert erst wenn P0
erneut versucht den kritischen Abschnitt zu betreten.~~

Vielen Dank für die Aufmerksamkeit!



Sourcecode + Folien: https://github.com/dmalysiak/lecture_fh_muenster