Prozessverwaltung

Pro'zess <m.; -es, -e> *Vorgang*, *Ablauf* [lat.]

Fadenverwaltung ∼inkarnation, ∼deskriptor
☞ Aktivitätsträger, Koroutine

∼**abfertigung** (*process dispatching*)
☞ CPU-Stoß (*CPU burst*) vs. E/A-Stoß (*I/O burst*)

∼**einplanung** (*CPU/process scheduling*)
☞ Klassifikation, Einplanungsebenen und -strategien

☞ Aspekte der Nebenläufigkeit (*race hazards/conditions*)

Prozessinkarnation

- die *Verwaltungseinheit* zur Beschreibung/Repräsentation eines Prozesses
  - der Typ einer Datenstruktur „**Prozessdeskriptor**“ (PD)
    ☞ Prozesskontrollblock (*process control block*, PCB), . . .
    ☞ UNIX Jargon: *proc structure* (von „`struct proc`“)
  - Kopf eines (komplexen) Datenstrukturgeflechts (☞ Objektkomposition)
- das (Software-) *Betriebsmittel* zur Ausführung eines Programms
  - eine Instanz vom Typ „PD“
- die *Identität* für ein sich in Ausführung befindliches Programm
  - das mit einer **Prozessidentifikation** (PID) assoziierte Objekt

Prozessdeskriptor

- Dreh- und Angelpunkt, der alle prozessbezogenen Betriebsmittel bündelt
  - Speicher- und, ggf., Adressraumbelegung †
    ☞ Text-, Daten-, Stapelsegmente (*code, data, stack*)
  - Dateideskriptoren und -köpfe (*inode*) †
    ☞ {Zwischenspeicher,Puffer}deskriptoren, Datenblöcke
  - Datei, die das vom Prozess ausgeführte Programm repräsentiert †
- zentrales Objekt, das Prozess- und Prozessorzustände beschreibt
  - Laufzeitkontext des zugeordneten Programmfadens/Aktivitätsträgers
  - gegenwärtiger Abfertigungszustand (*Scheduling*-Informationen) †
  - anstehende Ereignisse bzw. erwartete Ereignisse †
  - Benutzerzuordnung und -rechte †

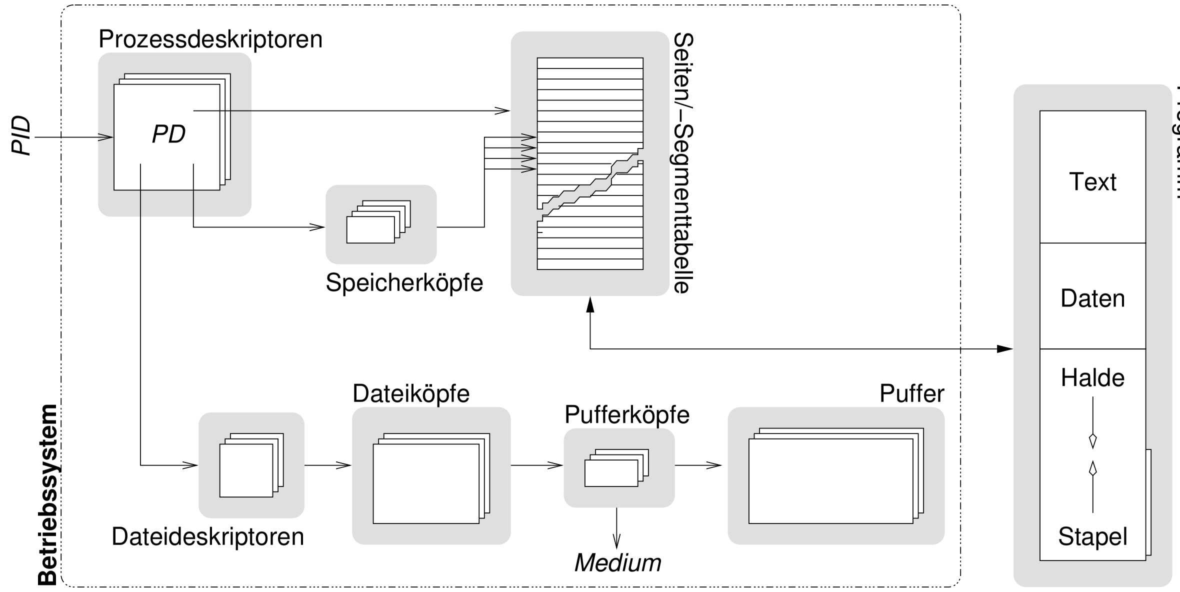
Aspekte der Prozessauslegung

- Aufbau und Struktur des PD ist höchst abhängig von Betriebsart und -zweck:

  † (1) Adressraumdeskriptoren sind nur notwendig im Falle von Systemen, die eine Adressraumisolation erfordern. (2) Für ein Sensor-/Aktorsystem haben Dateideskriptoren/-köpfe wenig Bedeutung. (3) In ROM-basierten Systemen durchlaufen die Prozesse oft immer nur ein und dasselbe Programm. (4) In Einbenutzersystemen ist es wenig sinnvoll, prozessbezogene Benutzerrechte verwalten zu wollen. (5) Bei statischer Ablaufplanung ist die Buchführung von Abfertigungszuständen verzichtbar. (6) Ebenso fällt Ereignisverwaltung nur an bei ereignisgesteuerten und/oder präemptiven Systemen. (7) . . .

- Festlegung auf genau eine Ausprägung grenzt Einsatzgebiete unnötig aus

Generische Datenstruktur „Prozess“



Koroutine — Aktivitätsträger

- ein *autonomer Kontrollfluss* innerhalb eines Programms (z. B. Betriebssystem)

  ☞ **Programm**(kontroll)**faden** (*thread of control*, TOC)

- mit zwei wesentlichen Unterschieden zu herkömmlichen Routinen/Prozeduren:
  1. die Ausführung beginnt immer an der letzten „Unterbrechungsstelle“
     – d. h., an der zuletzt die Kontrolle über den Prozessor abgegeben wurde
     – die Kontrollabgabe geschieht dabei grundsätzlich *kooperativ* (freiwillig)
  2. der Zustand ist *invariant* zwischen zwei aufeinanderfolgenden Ausführungen

- eine Koroutine kann als „zustandsbehaftete Prozedur“ aufgefasst werden

Koroutine (1)

An autonomous program which communicates with adjacent modules as if they were input or output subroutines.

[. . .]

Coroutines are subroutines all at the same level, each acting as if it were the master program. [25] [44]

[44] Koroutinen wurden erstmalig um 1963 in der von Conway entwickelten Architektur eines Fließbandübersetzers (*pipeline compiler*) eingesetzt. Darin wurden Parser konzeptionell als Datenflussfließbänder zwischen Koroutinen aufgefasst. Die Koroutinen repräsentierten *first-class* Prozessoren wie z. B. Lexer, Parser und Codegenerator.

Koroutine (2)

- Koroutinen sind Prozeduren ähnlich, es fehlt jedoch die Aufrufhierarchie:

  Beim Verlassen einer Koroutine geht anders als beim Verlassen einer Prozedur die Kontrolle nicht automatisch an die aufrufende Routine zurück. Stattdessen wird mit einer *resume*-Anweisung beim Verlassen einer Koroutine explizit bestimmt, welche andere Koroutine als nächste ausgeführt wird. [26], S. 49

- ein *programmiersprachliches Mittel* zur Prozessorweitergabe an Prozesse

Routinen vs. Koroutinen

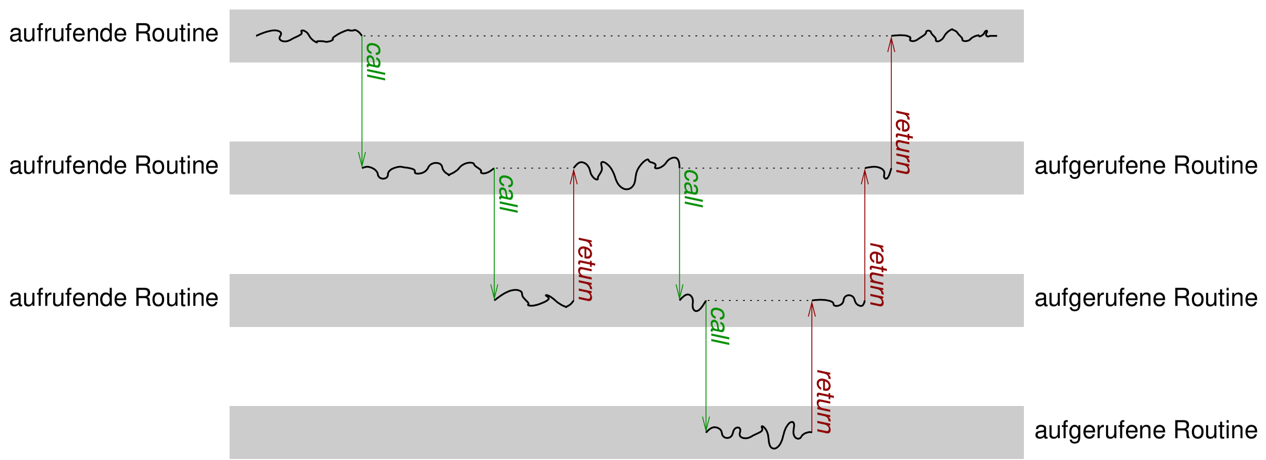
- hinter beiden Konzepten verbergen sich unterschiedliche *Ablaufmodelle*:

  asymmetrische Aktivierung im Falle von Routinen
  - die Beziehung zwischen den Routinen ist *nicht* gleichberechtigt
  - über die Routinen ist eine <u>Aufrufhierarchie</u> definiert

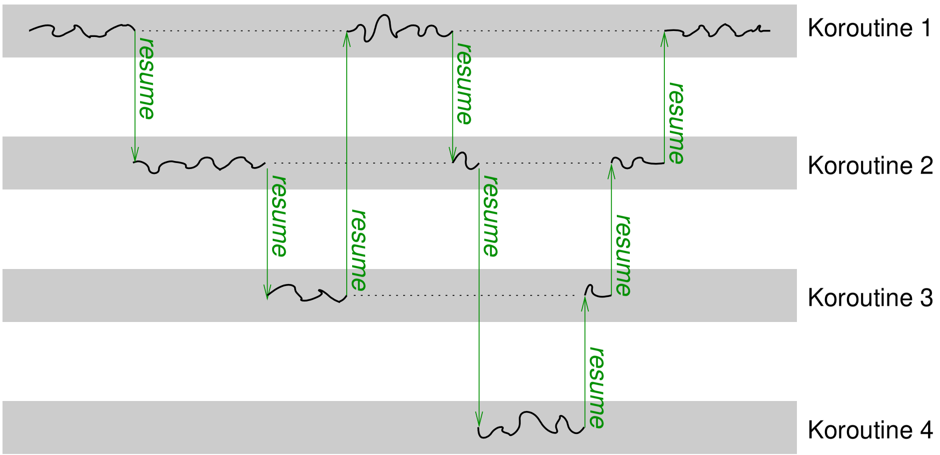
  symmetrische Aktivierung im Falle von Koroutinen
  - die Beziehung zwischen den Koroutinen ist gleichberechtigt
  - über die Koroutinen ist *keine* Aufrufhierarchie definiert

- Routinen werden aufgerufen, um sie zu aktivieren, im Gegensatz zu Koroutinen, die erzeugt, aktiviert und zerstört werden und sich selbst nur deaktivieren

Aufrufhierarchie Routinen



Nebenläufigkeit — Koroutinen



Gemeinsamkeiten — {Kor,R}outinen

- {Kor,R}outinen sind zu reaktivieren, um weiter ausgeführt werden zu können:

  Routine beim *Rücksprung* aus der aufgerufenen Instanz
  Koroutine beim *Suspendieren* der die Kontrolle abgebenden Instanz

- jeder „Aufruf“ hinterlässt seinen „Fußabdruck“ im *Aktivierungsblock*
  - die Rückkehradresse zur aufrufenden {Kor,R}outine
  - die von der {Kor,R}outine belegten Prozessorregister

- der Aufbau des Aktivierungsblocks ist abhängig vom Prozessor bzw. Kompilierer

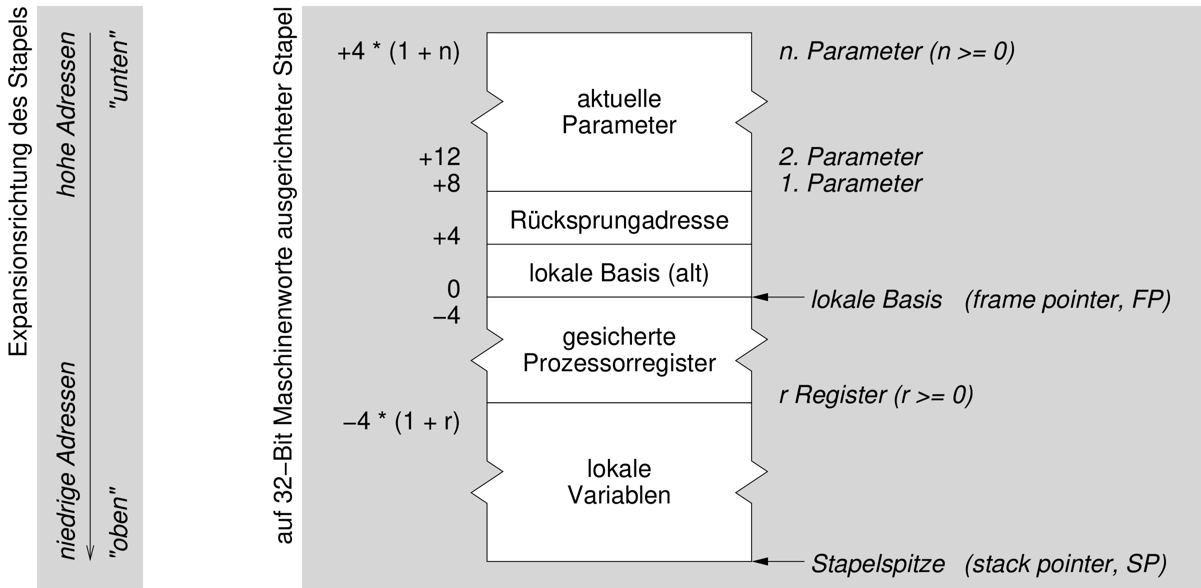
Unterschiede {Kor,R}outinen

eine **Koroutine** besitzt *eigene Betriebsmittel* zur Aktivierungsblockverwaltung

- Art und Anzahl der Betriebsmittel ist sehr problemspezifisch
  - ☞ CPU: CISC (Stapel) vs. RISC (Register und/oder Stapel) — Ebene $_2$
  - ☞ Kompilierer: Laufzeitmodell der Programmiersprache — Ebene $_5$
  - ☞ Anwendungsprogramm: die zu bewältigende Aufgabe — Ebene $_6$
- Verfügbarkeit eigener Betriebsmittel begründet (relative) Unabhängigkeit

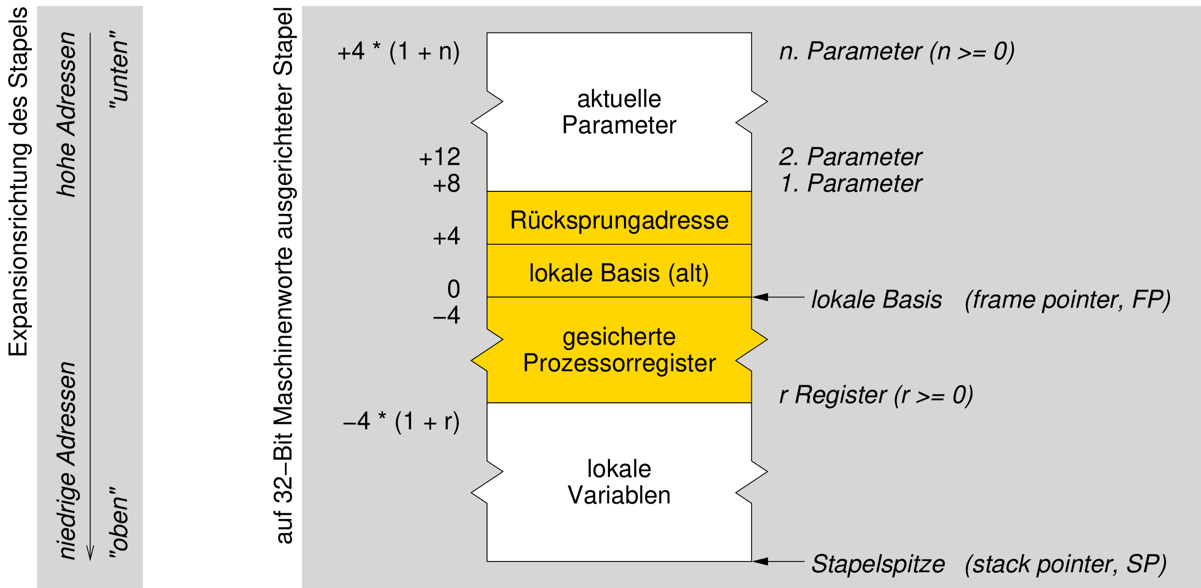
eine **Routine** muss sich diese Betriebsmittel mit anderen Routinen teilen

Aktivierungsblock *activation record*



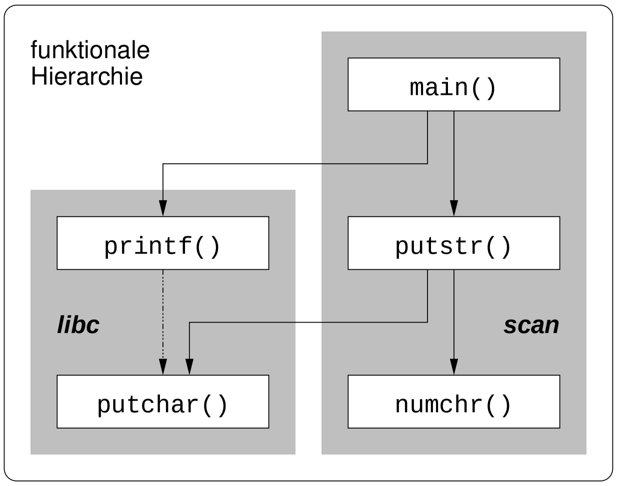
Aktivierungsblock

Laufzeitstatus



Stapelverlauf (6-16 – 6-41)

Fallstudie (1)



`main()` übergibt (beginnend mit `argv[1]`) die Programmargumente nacheinander an `putstr()` und liefert am Ende eine Statistik über die verarbeiteten Zeichen.

`putstr()` übergibt die Zeichen (des aktuellen Parameters) nacheinander an `numchr()` und `putchar()` und bestimmt die Zeichenlänge (des Programmarguments).

`numchr()` klassifiziert ein Zeichen und bestimmt die Häufigkeit je nach Zeichenkategorie.

Stapelverlauf — Fallstudie (2)

```c
#include <stdio.h>

unsigned numchr (char c, unsigned num[])
{
    if (c < '0') num[0]++;
    else if (c <= '9') num[1]++;
    else if ((c >= 'A') && (c <= 'Z')) num[2]++;
    else if ((c >= 'a') && (c <= 'z')) num[3]++;
    else num[4]++;
    return 1;
}

unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        putchar(c);
    }
    return val;
}

int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```


Kontextsicherung (1)

```c
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```

x86 Stapel

Zeiger	Inhalt	Adresse
	argv	0x4280
	argc	0x427c
	return PC	0x4278
FP →	*saved* EBP	0x4274
	saved EDI	0x4270
	saved ESI	0x426c
SP →	*saved* EBX	0x4268
		0x4264
		0x4260
		0x425c
		0x4258
		0x4254
		0x4250
		0x424c
		0x4248
		0x4244
		0x4240
		0x423c
		0x4238
		0x4234
		0x4230
		0x422c
		0x4228
		0x4224

Einrichtung lokaler Variablen (1)

```
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```

x86 Stapel

		argv	0x4280
		argc	0x427c
		return PC	0x4278
	FP →	*saved* EBP	0x4274
idx = EDI		*saved* EDI	0x4270
		saved ESI	0x426c
= EBX		*saved* EBX	0x4268
		num[4]	0x4264
		num[3]	0x4260
		num[2]	0x425c
		num[1]	0x4258
		num[0]	0x4254
	SP →	got	0x4250
			0x424c
			0x4248
			0x4244
			0x4240
			0x423c
			0x4238
			0x4234
			0x4230
			0x422c
			0x4228
			0x4224

Parameterübergabe (1)

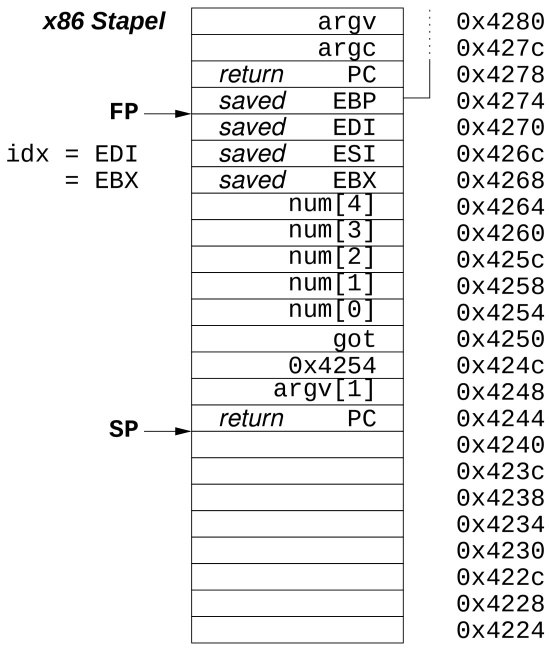
```
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```

x86 Stapel

		argv	0x4280
		argc	0x427c
		return PC	0x4278
	FP →	*saved* EBP	0x4274
idx = EDI		*saved* EDI	0x4270
		saved ESI	0x426c
= EBX		*saved* EBX	0x4268
		num[4]	0x4264
		num[3]	0x4260
		num[2]	0x425c
		num[1]	0x4258
		num[0]	0x4254
		got	0x4250
		0x4254	0x424c
	SP →	argv[1]	0x4248
			0x4244
			0x4240
			0x423c
			0x4238
			0x4234
			0x4230
			0x422c
			0x4228
			0x4224

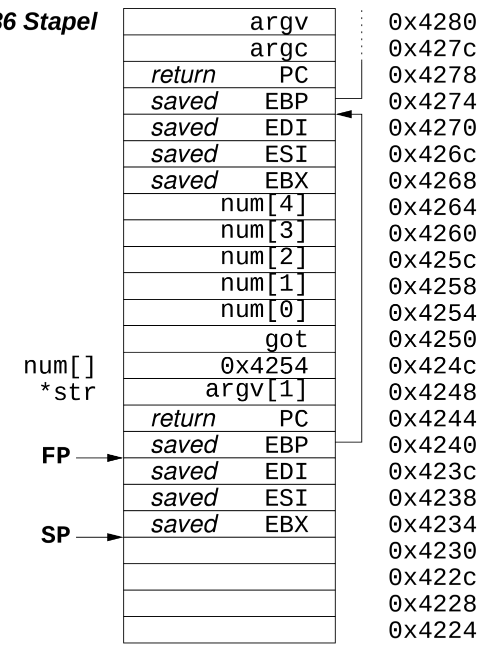
Funktionsaufruf (1)

```c
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```



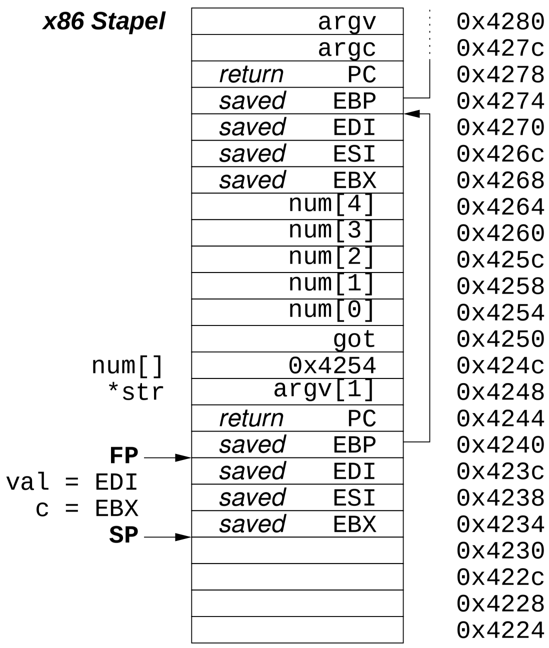
Kontextsicherung (2)

```c
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        putchar(c);
    }
    return val;
}
```



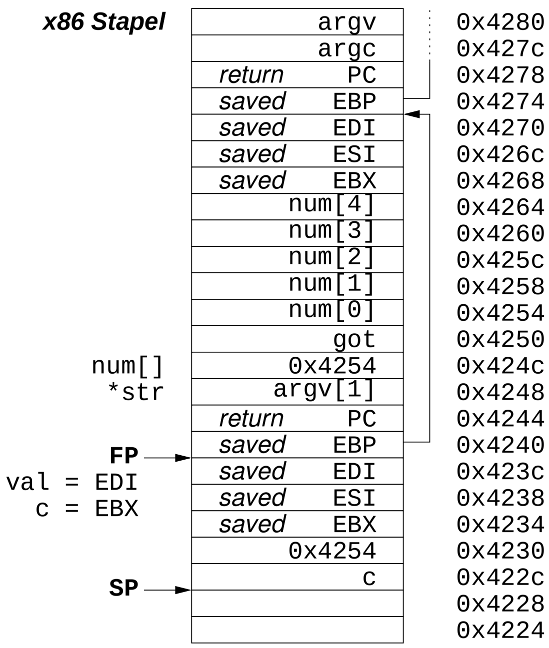
Einrichtung lokaler Variablen (2)

```c
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        putchar(c);
    }
    return val;
}
```



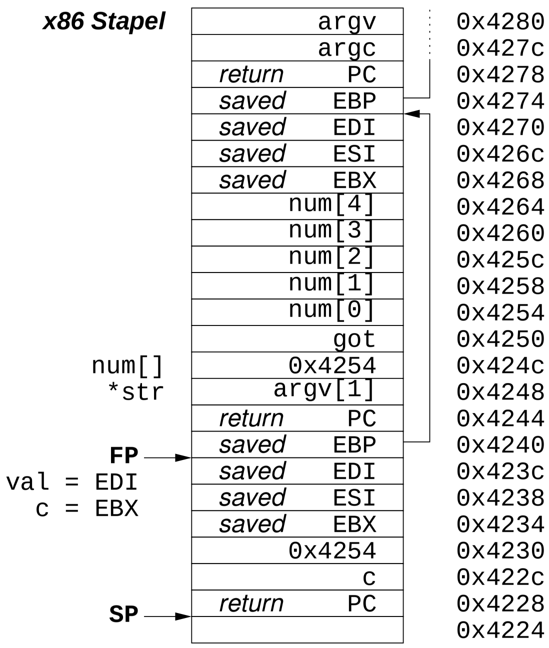
Parameterübergabe (2)

```c
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        putchar(c);
    }
    return val;
}
```



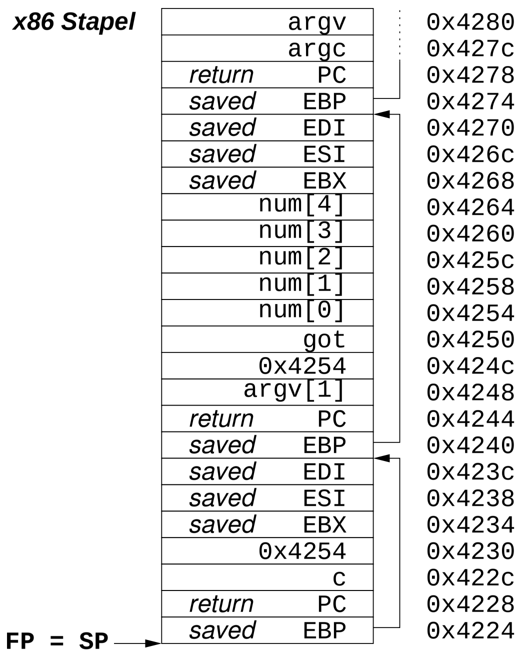
Funktionsaufruf (2)

```
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        putchar(c);
    }
    return val;
}
```



Kontextsicherung (2)

```
unsigned numchr (char c, unsigned num[])
{
    if (c < '0') num[0]++;
    else if (c <= '9') num[1]++;
    else if ((c >= 'A') && (c <= 'Z')) num[2]++;
    else if ((c >= 'a') && (c <= 'z')) num[3]++;
    else num[4]++;
    return 1;
}
```



Kontextwiederherstellung (1)

```
unsigned numchr (char c, unsigned num[])
{
    if (c < '0') num[0]++;
    else if (c <= '9') num[1]++;
    else if ((c >= 'A') && (c <= 'Z')) num[2]++;
    else if ((c >= 'a') && (c <= 'z')) num[3]++;
    else num[4]++;
    return 1;
}
```

x86 Stapel

	argv	0x4280
	argc	0x427c
	return PC	0x4278
	saved EBP	0x4274
	saved EDI	0x4270
	saved ESI	0x426c
	saved EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
	got	0x4250
	0x4254	0x424c
	argv[1]	0x4248
	return PC	0x4244
FP →	*saved* EBP	0x4240
	saved EDI	0x423c
	saved ESI	0x4238
	saved EBX	0x4234
	0x4254	0x4230
	c	0x422c
SP →	*return* PC	0x4228
	saved EBP	0x4224

Rücksprung (1)

```
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        putchar(c);
    }
    return val;
}
```

x86 Stapel

	argv	0x4280
	argc	0x427c
	return PC	0x4278
	saved EBP	0x4274
	saved EDI	0x4270
	saved ESI	0x426c
	saved EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
	got	0x4250
num[]	0x4254	0x424c
*str	argv[1]	0x4248
	return PC	0x4244
FP →	*saved* EBP	0x4240
val = EDI	*saved* EDI	0x423c
c = EBX	*saved* ESI	0x4238
	saved EBX	0x4234
	0x4254	0x4230
SP →	c	0x422c
	return PC	0x4228
	saved EBP	0x4224

Parameterfreigabe (1)

```
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        putchar(c);
    }
    return val;
}
```

x86 Stapel

		Adresse
	argv	0x4280
	argc	0x427c
	return PC	0x4278
	saved EBP	0x4274
	saved EDI	0x4270
	saved ESI	0x426c
	saved EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
	got	0x4250
num[]	0x4254	0x424c
*str	argv[1]	0x4248
	return PC	0x4244
FP →	*saved* EBP	0x4240
val = EDI	*saved* EDI	0x423c
c = EBX	*saved* ESI	0x4238
SP →	*saved* EBX	0x4234
	0x4254	0x4230
	c	0x422c
	return PC	0x4228
	saved EBP	0x4224

Parameterübergabe (3)

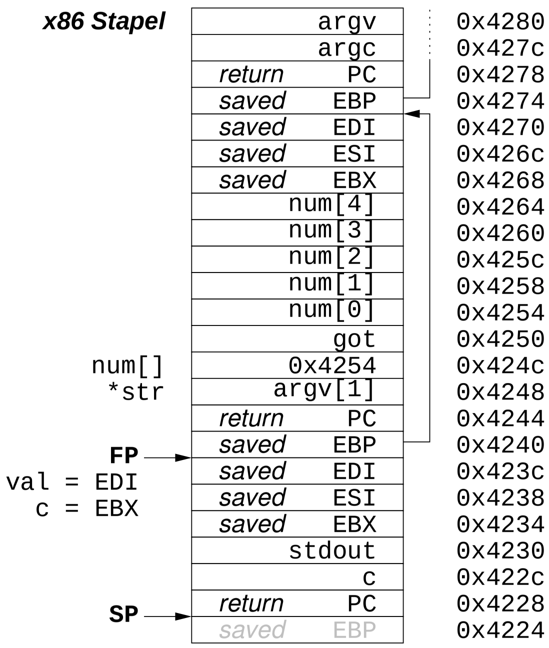
```
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        _IO_putc(c, stdout);
    }
    return val;
}
```

x86 Stapel

		Adresse
	argv	0x4280
	argc	0x427c
	return PC	0x4278
	saved EBP	0x4274
	saved EDI	0x4270
	saved ESI	0x426c
	saved EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
	got	0x4250
num[]	0x4254	0x424c
*str	argv[1]	0x4248
	return PC	0x4244
FP →	*saved* EBP	0x4240
val = EDI	*saved* EDI	0x423c
c = EBX	*saved* ESI	0x4238
	saved EBX	0x4234
	stdout	0x4230
	c	0x422c
SP →	*return* PC	0x4228
	saved EBP	0x4224

Funktionsaufruf (3)

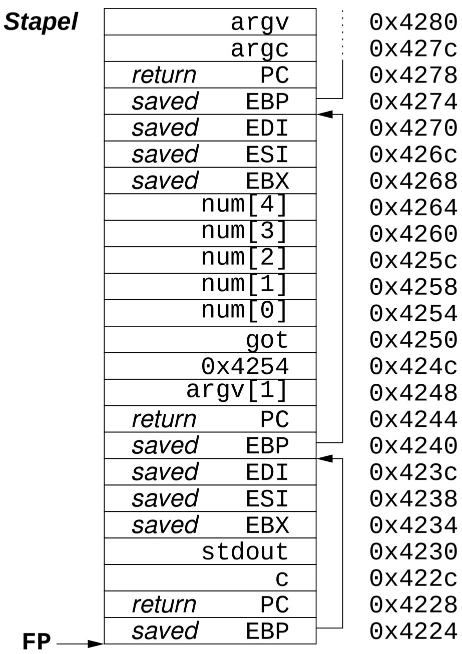
```
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        _IO_putc(c, stdout);
    }
    return val;
}
```



C-Bibliothek

```
int _IO_putc (int c, FILE *stream)
{
    ...
}
```

- `_IO_putc()` gibt das Zeichen (`c`) aus
  - `stream` identifiziert den Ausgabekanal
  - weiterer Stapelauf- und -abbau
- Kontextwiederherstellung, Rücksprung



Rücksprung (2)

```
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        _IO_putc(c, stdout);
    }
    return val;
}
```

x86 Stapel

		Adresse
	argv	0x4280
	argc	0x427c
	return PC	0x4278
	saved EBP	0x4274
	saved EDI	0x4270
	saved ESI	0x426c
	saved EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
	got	0x4250
num[]	0x4254	0x424c
*str	argv[1]	0x4248
	return PC	0x4244
FP	*saved* EBP	0x4240
val = EDI	*saved* EDI	0x423c
c = EBX	*saved* ESI	0x4238
	saved EBX	0x4234
	stdout	0x4230
SP	c	0x422c
	return PC	0x4228
	saved EBP	0x4224

Parameterfreigabe (2)

```
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        _IO_putc(c, stdout);
    }
    return val;
}
```

x86 Stapel

		Adresse
	argv	0x4280
	argc	0x427c
	return PC	0x4278
	saved EBP	0x4274
	saved EDI	0x4270
	saved ESI	0x426c
	saved EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
	got	0x4250
num[]	0x4254	0x424c
*str	argv[1]	0x4248
	return PC	0x4244
FP	*saved* EBP	0x4240
val = EDI	*saved* EDI	0x423c
c = EBX	*saved* ESI	0x4238
SP	*saved* EBX	0x4234
	stdout	0x4230
	c	0x422c
	return PC	0x4228
	saved EBP	0x4224

Kontextwiederherstellung (2)

```
unsigned putstr (char *str, unsigned num[])
{
    char c;
    unsigned val = 0;
    while ((c = *str++)) {
        val += numchr(c, num);
        putchar(c);
    }
    return val;
}
```

x86 Stapel

			Adresse
		argv	0x4280
		argc	0x427c
	return	PC	0x4278
FP →	*saved*	EBP	0x4274
	saved	EDI	0x4270
	saved	ESI	0x426c
	saved	EBX	0x4268
		num[4]	0x4264
		num[3]	0x4260
		num[2]	0x425c
		num[1]	0x4258
		num[0]	0x4254
		got	0x4250
num[]		0x4254	0x424c
*str		argv[1]	0x4248
SP →	*return*	PC	0x4244
	saved	EBP	0x4240
	saved	EDI	0x423c
	saved	ESI	0x4238
	saved	EBX	0x4234
		stdout	0x4230
		c	0x422c
	return	PC	0x4228
	saved	EBP	0x4224

Rücksprung (3)

```
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```

x86 Stapel

			Adresse
		argv	0x4280
		argc	0x427c
	return	PC	0x4278
FP →	*saved*	EBP	0x4274
idx = EDI	*saved*	EDI	0x4270
= EBX	*saved*	ESI	0x426c
	saved	EBX	0x4268
		num[4]	0x4264
		num[3]	0x4260
		num[2]	0x425c
		num[1]	0x4258
		num[0]	0x4254
		got	0x4250
		0x4254	0x424c
SP →		argv[1]	0x4248
	return	PC	0x4244
	saved	EBP	0x4240
	saved	EDI	0x423c
	saved	ESI	0x4238
	saved	EBX	0x4234
		stdout	0x4230
		c	0x422c
	return	PC	0x4228
	saved	EBP	0x4224

Parameterfreigabe (3)

```
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```

x86 Stapel

	argv	0x4280
	argc	0x427c
	return PC	0x4278
FP →	*saved* EBP	0x4274
idx = EDI	*saved* EDI	0x4270
	saved ESI	0x426c
= EBX	*saved* EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
SP →	got	0x4250
	0x4254	0x424c
	argv[1]	0x4248
	return PC	0x4244
	saved EBP	0x4240
	saved EDI	0x423c
	saved ESI	0x4238
	saved EBX	0x4234
	stdout	0x4230
	c	0x422c
	return PC	0x4228
	saved EBP	0x4224

Wiederholung und Ausgabe . . .

```
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```

Der bisherige Stapelverlauf (6-20 – 6-37) wird sich in seinen wesentlichen Zügen und für alle weiteren Programmargumente wiederholen. Die Stapelinhalte variieren dabei in Abhängigkeit von den Werten (d. h., den einzelnen Zeichen) der Argumente.

Im weiteren Verlauf wird das Stapelbild durch die abschließenden Ausgaben stark verändert: `printf()` „strapaziert" den Stapel zutiefst und bewirkt einen entsprechenden Auf- und Abbau.

Freigabe lokaler Variablen

```
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```

x86 Stapel

	argv	0x4280
	argc	0x427c
	return PC	0x4278
FP →	*saved* EBP	0x4274
	saved EDI	0x4270
	saved ESI	0x426c
SP →	*saved* EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
	got	0x4250
	0x4254	0x424c
	argv[1]	0x4248
	return PC	0x4244
	saved EBP	0x4240
	saved EDI	0x423c
	saved ESI	0x4238
	saved EBX	0x4234
	stdout	0x4230
	c	0x422c
	return PC	0x4228
	saved EBP	0x4224

Kontextwiederherstellung (3)

```
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```

x86 Stapel

	argv	0x4280
	argc	0x427c
SP →	*return* PC	0x4278
	saved EBP	0x4274
	saved EDI	0x4270
	saved ESI	0x426c
	saved EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
	got	0x4250
	0x4254	0x424c
	argv[1]	0x4248
	return PC	0x4244
	saved EBP	0x4240
	saved EDI	0x423c
	saved ESI	0x4238
	saved EBX	0x4234
	stdout	0x4230
	c	0x422c
	return PC	0x4228
	saved EBP	0x4224

Rücksprung (4) — Schluss . . .

```
int main (int argc, char *argv[])
{
    unsigned idx = 0;
    unsigned got = 0;
    unsigned num[5] = {0, 0, 0, 0, 0};

    while (--argc)
        got += putstr(argv[++idx], num);
    printf("\n");

    for (idx = 0; idx < 5; idx++)
        printf("%u ", num[idx]);
    printf("\ntotal %u\n", got);

    return 42;
}
```

x86 Stapel (SP → 0x4278)

	argv	0x4280
	argc	0x427c
return	PC	0x4278
saved	EBP	0x4274
saved	EDI	0x4270
saved	ESI	0x426c
saved	EBX	0x4268
	num[4]	0x4264
	num[3]	0x4260
	num[2]	0x425c
	num[1]	0x4258
	num[0]	0x4254
	got	0x4250
	0x4254	0x424c
	argv[1]	0x4248
return	PC	0x4244
saved	EBP	0x4240
saved	EDI	0x423c
saved	ESI	0x4238
saved	EBX	0x4234
	stdout	0x4230
	c	0x422c
return	PC	0x4228
saved	EBP	0x4224

Laufzeitkontext

- der Aktivierungsblock enthält den *Laufzeitstatus* der aufrufenden Routine:
  - die Programmadresse, an der die Routine ihre Ausführung nach erfolgtem Rücksprung weiter fortsetzen wird
    ☞ die Adresse des dem Aufruf nachfolgenden Maschinenbefehls
  - die Inhalte der Arbeitsregister, die von der aufgerufenen Routine im weiteren Verlauf verwendet werden
    ☞ die Inhalte der *nicht-flüchtigen Register*; beim x86: `EBP`, `ESI`, `EDI`, `EBX`
- dieser Status ist *invariant* in Bezug auf die Ausführung aufgerufener Routinen
  - die Anzahl gesicherter nicht-flüchtiger Register ist jedoch variabel
  - abhängig von der aufgerufenen Routine und der virtuellen Maschine

Autonomer Kontrollfluss

- im Gegensatz zur Routine bedeutet die Aktivierung einer Koroutine:
  – Laufzeitstatus- $\left\{\begin{array}{lcr} \textit{Sicherung} & \text{des} & \text{aktiven} \\ \textit{Wiederherstellung} & \text{eines} & \text{inaktiven} \end{array}\right\}$ Kontrollflusses
- d. h., Laufzeitstatus-Sicherung/Wiederherstellung verläuft in zwei Dimensionen:

  vertikal bei Einsprung in und Rücksprung aus einer Routine
  ☞ ohne dabei den aktiven Kontrollfluss zu wechseln (6-16 – 6-41)

  horizontal bei Deaktivierung und Aktivierung eines Kontrollflusses
- Koroutinen repräsentieren gleichberechtigte, autonome Kontrollflüsse

Virtuelle Maschine und Laufzeitstatus von Koroutinen

Ebene $_5$ abstrakter Prozessor „Kompilierer“

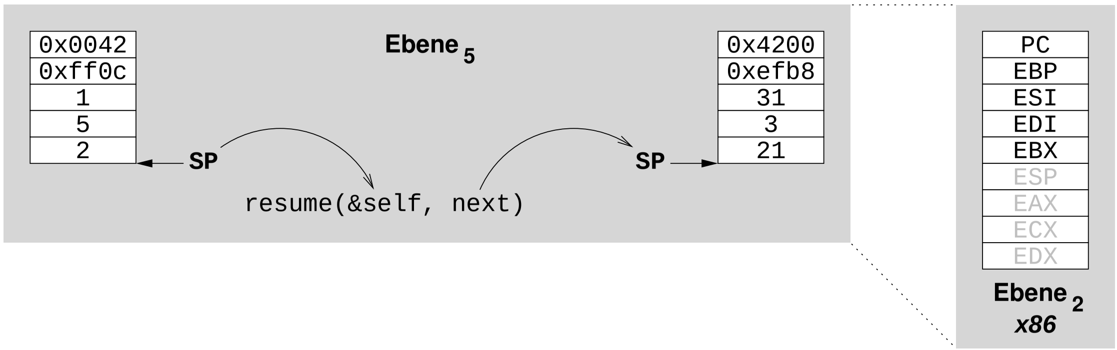
- der „Prozessor“ unterscheidet zwischen zwei Arten von Registern:
  1. *nicht-flüchtige Register*; invariante Inhalte über Prozedurgrenzen hinweg
  2. *flüchtige Register*; sonst
- die nicht-flüchtigen Register speichern den Laufzeitstatus einer Koroutine
- ein Koroutinenwechsel muss die Inhalte „einiger“ CPU-Register austauschen

Ebene $_4$ abstrakter Prozessor „Assemblierer“

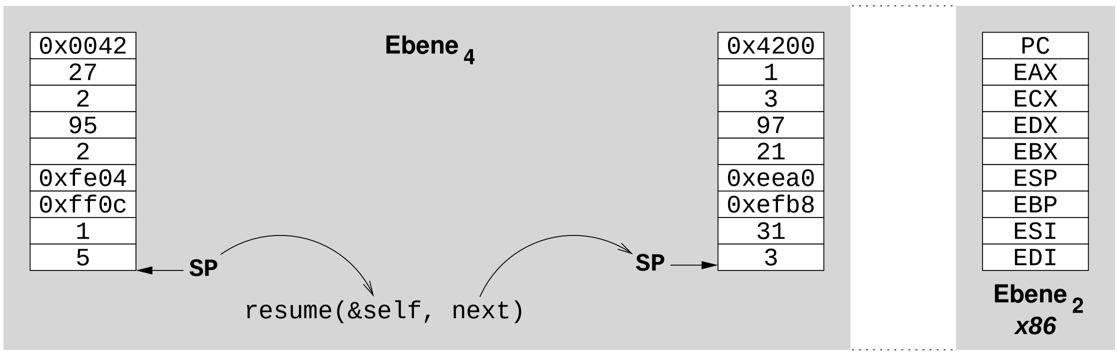
- alle (Ebene $_2$/CPU) Register speichern den Laufzeitstatus einer Koroutine
- ein Koroutinenwechsel muss die Inhalte „aller“ CPU-Register austauschen

☞ Koroutinenwechsel auf Ebene $_5$ (z. B. in C) sind effizienter als auf Ebene $_4$

Kontrollflusswechsel (1) — Ebene $_5$ vs. Ebene $_2$



Kontrollflusswechsel (2) — Ebene $_4$ vs. Ebene $_2$



Kontrollflussverwaltung

`next = branch (stkp, argc)` zweigt einen Kontrollfluss ab

- `stkp` ist Stapelzeiger der zu initialisierenden Koroutine
- `argc` ist die Anzahl zusätzlich zu kopierenden Funktionsparameter
- `next` ist Kontextzeiger ($= 0$ ☞ Koroutine, $\neq 0$ ☞ aufrufende Instanz)

`resume (self, next)` setzt einen anderen Kontrollfluss fort

- `self` ist Referenz auf den Kontextzeiger der abgebenden Koroutine
- `next` ist Kontextzeiger der zu aktivierenden Koroutine

`finite (next)` beendet den laufenden Kontrollfluss; aktiviert `next`

Fallstudie: kooperative (zufällige) Ausgabe von `argv[]`

```
#define STACK_SIZE 1024*16

static char *dad;
static char *son;
static int   arg;

int main (int argc, char *argv[]) {
    son = coroutine(argc);
    if (son) {
        while (argc--) {
            resume(&dad, son);
            printf("argv[%d] -> %s\n",
                arg, (int)argv[arg]);
        }
    }
    return 0;
}
```

```
char* coroutine (int argc) {
    char *stk;
    char *foo;
    int run;

    stk = (char*)malloc(STACK_SIZE);
    if (!stk) return 0;

    foo = branch(&stk[STACK_SIZE], 1);
    if (foo) return foo;

    for (run = 1; ; run++) {
        printf("run %d...", run);
        arg = random() % argc;
        resume(&son, dad);
    }
}
```

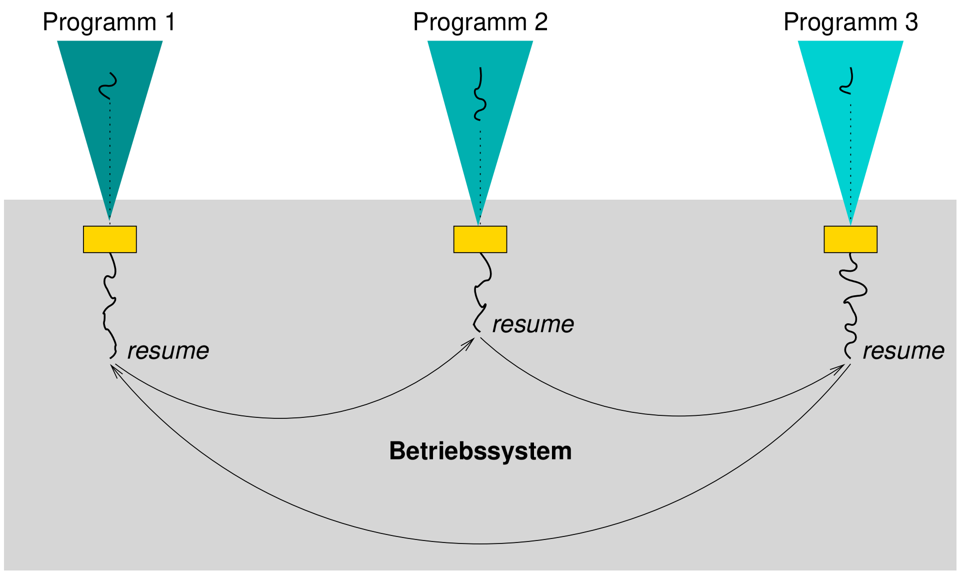
Operationsprinzip von Koroutinen

- die Koroutine „mechanisiert“ den autonomen Kontrollfluss eines Prozesses
  - mit ihr wird die Prozessorzuteilung an ausführbereite Prozesse möglich
  - sie bildet die Grundlage zum Multiplexen der CPU zwischen Prozessen
- das *nicht-sequentielle Operationsprinzip* von Koroutinen ist streng kooperativ
  - eine Koroutine gibt das Betriebsmittel „CPU“ immer nur freiwillig ab
  - die Abgabe ist programmiert, sie muss darüberhinaus erreichbar sein
- Betriebssysteme schützen die CPU vor „unkooperativen Koroutinen“
  - den Koroutinen kann das Betriebsmittel „CPU“ entzogen werden
  - einer Koroutine wird es dadurch unmöglich, die CPU zu monopolisieren

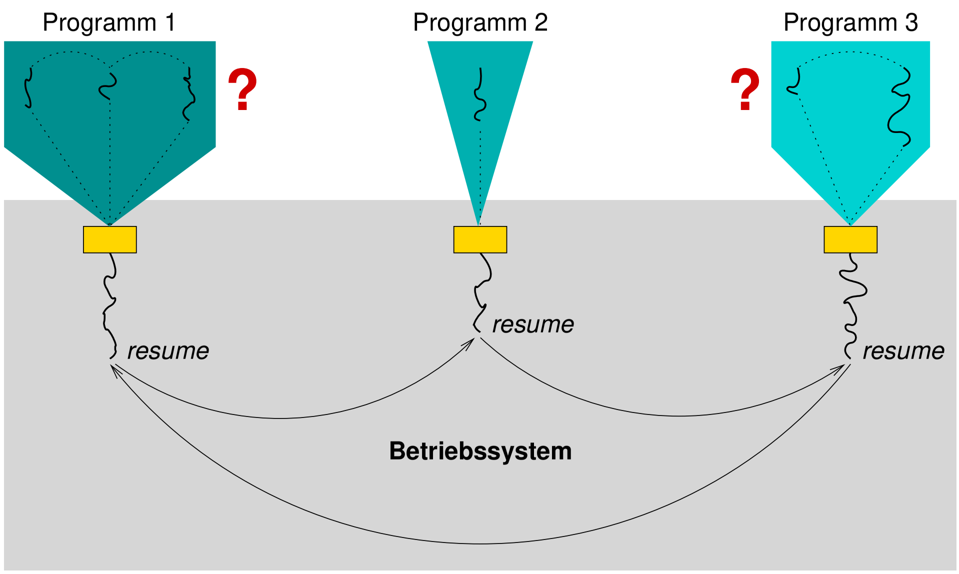
Mehrprogrammbetrieb und Koroutinen

- die Koroutine als „*abstrakter Prozessor*“ zur Ausführung eines Programms:
  - für jedes auszuführende Programm wird genau eine Koroutine bereitgestellt
  - ist eine Koroutine aktiv, läuft auch immer das ihr zugeordnete Programm
  - um ein anderes Programm auszuführen, ist die Koroutine zu wechseln
- die Koroutinen sind als „Aktivitätsträger“ Bestandteil des Betriebssystems
  - für den Laufzeitstatus gibt es pro Koroutine eine *Kontextvariable*
  - jedem Benutzerprogramm ist eine solche Variable zugeordnet
  - die Variablen sind gültig nur innerhalb des Programms „Betriebssystem“
- ein Betriebssystem ist Inbegriff für das *nicht-sequentielle Programm*

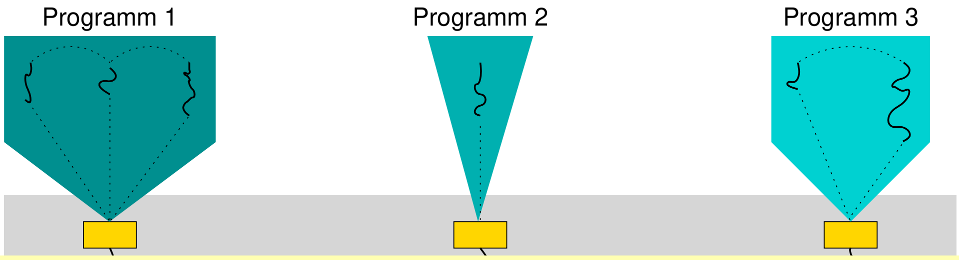
Verarbeitung sequentieller Programme



Verarbeitung (nicht-) sequentieller Programme



Verarbeitung nicht-sequentieller Programme



Programme 1 und 3 sind nicht-sequentielle Programme. Die Verarbeitung aller Kontrollflüsse derartiger Programme durch das Betriebssystem ist nur möglich, wenn im Betriebssystem für jeden Kontrollfluss ein Repräsentant in Form einer Koroutine zur Verfügung steht. Genauso, wie das nicht-sequentielle Programm „Betriebssystem" seine Kontrollflüsse selbst verwalten muss, wenn die CPU dazu unfähig ist, müssen Programme 1 und 3 ihre Kontrollflüsse selbst verwalten, wenn das Betriebssystem dazu unfähig ist.

Koroutinen ≡ Kooperative

- den Gültigkeitsbereich von Koroutinen legt das sie umgebende Programm fest
  - programmübergreifende Koroutinen { -wechsel lassen sich nicht ausdrücken / -auswahl ist nicht praktizierbar }
- die Einplanung (*scheduling*) von Koroutinen ist nicht betriebsmittelorientiert
  - im Vordergrund stehen vielmehr organisatorische/strukturelle Fragen
  - nicht-sequentielle Programme von Koroutinen bilden Kooperativen
  - Koroutinen sind Mittel zum Zweck, Kontrollflüsse zu repräsentieren
- lokale Kooperation ist um „globales Denken und Handeln" zu ergänzen

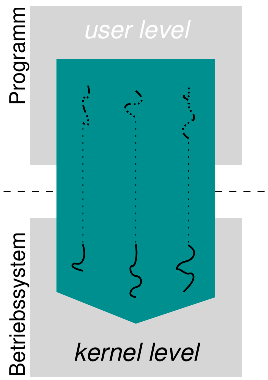
Programmfaden

thread *of control* **1.** der Faden; der Zusammenhang. **2.** durchziehen.

- Einplanungseinheit („*unit of scheduling*“) für die Vergabe der CPU
  - die Ablaufplanung erfolgt $\left\{\begin{array}{l}\text{betriebsmittelorientiert}\\ \text{ereignisgesteuert}\end{array}\right.$
- auch bekannt als **leichtgewichtiger Prozess** („*light-weight process*“, LWP)
  - der sich mit anderen Fäden eine gemeinsame Ausführungsumgebung teilt

Entsprechend der Ebene ihres Vorkommens bzw. ihrer Verwaltung werden Fäden in zwei Hauptgruppen unterteilt: Fäden auf Benutzerbene („Benutzerfäden“) und Fäden auf Kernebene („Kernfäden“).

Fäden Kernebene

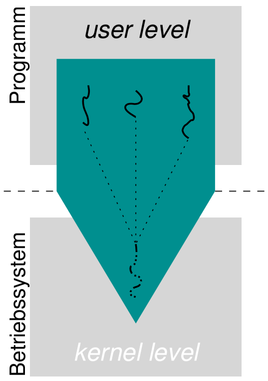


kernel-level thread Prozesskonzept der klassischen Form zur Implementierung nicht-sequentieller Programme

- Prozesse sind Fäden des Programms „Betriebssystem“
- Benutzerfäden werden Kernfäden exklusiv zugeordnet
  - jeder Benutzerfaden besitzt (s)einen Kernfaden
  - nicht jeder Kernfaden „trägt“ einen Benutzerfaden
- Ablaufplanung ist Funktion des Betriebssystem(kern)s

Operationen auf Benutzerfäden sind im Betriebssystemkern implementiert, was zur Konsequenz hat, dass Fadenwechsel innerhalb desselben Benutzerprogramms auch über den Betriebssystemkern verlaufen ☞ Laufzeitaufwand

Fäden — Benutzerebene



user-level thread Prozesskonzept, das Implementierung und Verwaltung von Fäden ganz in „Benutzerhände“ legt

- Prozesse sind mehrfädige (Benutzer-) Programme
- zur Ausführung werden *virtuelle Prozessoren* verwendet
  - die Prozessoren können als Kernfäden realisiert sein
  - der Kern reicht „*scheduler activations*“ nach oben
- Ablaufplanung ist Funktion jedes Benutzerprogramms

Existenz und Anzahl von Benutzerfäden sind dem Betriebssystemkern unbekannt, weshalb über die Auslastung der virtuellen Prozessoren exakte Aussagen nicht getroffen werden können ☞ Prozessorauswahl

Fadenverläufe — Stoßbetrieb (*burst mode*)

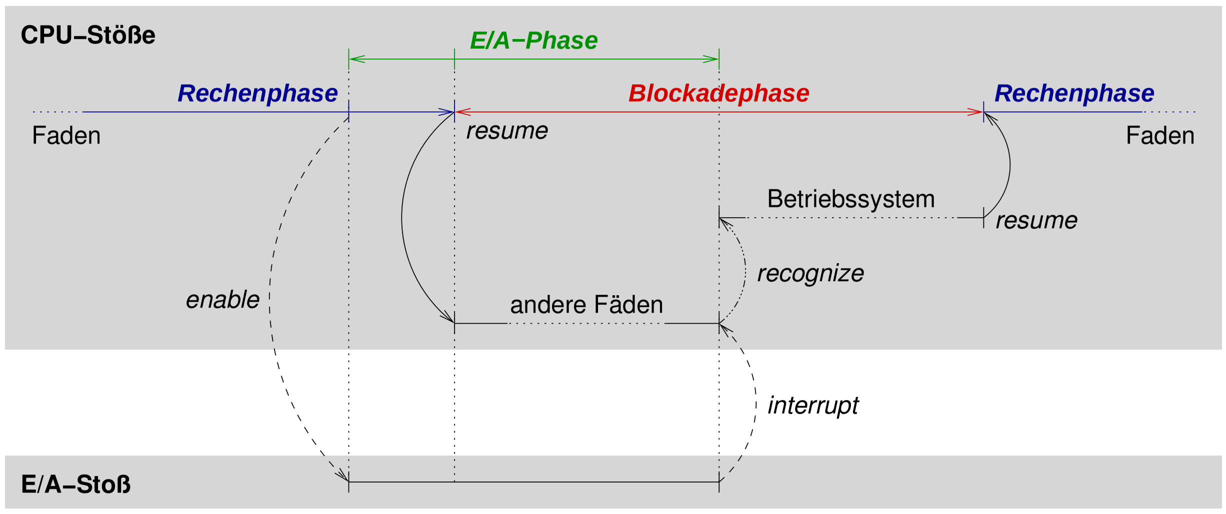
Rechenphase „CPU-Stoß“ (*CPU burst*)

- aktive Phase eines Programmfadens: die CPU führt Instruktionen aus

Blockadephase „E/A-Stoß“ (*I/O burst*)

- inaktive Phase eines Programmfadens, z. B. als Folge von Ein-/Ausgabe:
  - eine E/A-Operation wurde auf Anweisung des Faden in Gang gesetzt
  - der Faden muss die Beendigung der E/A-Operation abwarten
- allgemein: ein Programmfaden erwartet (passiv) ein Ereignis
  - das Ereignis wird von einem anderen Faden signalisiert
  - ein E/A-Gerät kann dabei als „externer Faden“ betrachtet werden

CPU-Stoß vs. E/A-Stoß



Fäden als Mittel zur Leistungsoptimierung

- der CPU-Stoß von Faden_x verläuft parallel[45] zum E/A-Stoß von Faden_y
  – ggf. werden CPU- bzw. E/A-Stöße weiterer Fäden zum „Auffüllen“ genutzt
- die Auslastung wird verbessert durch die Überlappung der verschiedenen Stöße
  – in einem Monoprozessorsystem kann immer nur ein CPU-Stoß aktiv sein
  – parallel dazu können jedoch viele E/A-Stöße (anderer Fäden) laufen
  – als Folge sind CPU und E/A-Geräte andauernd mit Arbeit beschäftigt
- bei nur einen Prozessor (CPU, E/A-Gerät) sind die Fäden zu serialisieren

[45] Ein E/A-Stoß ist zu einem Zeitpunkt zwar genau einem Programmfaden zugeordnet, er wird jedoch von einem separaten „*I/O processor*“ (IOP) ausgeführt — dem E/A-Gerät. Dadurch ergibt sich echte Parallelität auf Stoßebene.