SMP, Threading und die klassische IPC

Matthias Sattler
Unix-AG, Universität Kaiserslautern
sattler@unix-ag.uni-kl.de

V1.0 – 18. Mai 1997

Zusammenfassung

Der Betrieb von mehreren Prozessoren unter der Kontrolle eines Betriebssystemes rückt seit den jüngsten Entwicklungen auf dem Hardwaresektor immer mehr in den Bereich des Möglichen auch für den privaten Entwickler. Linux bietet hier eine hervorragende Entwurfsplattform. Der Vortrag stellt Konzepte des SMP sowie die verschiedenen Kommunikationsparadigmen vor, die hiermit einhergehen.

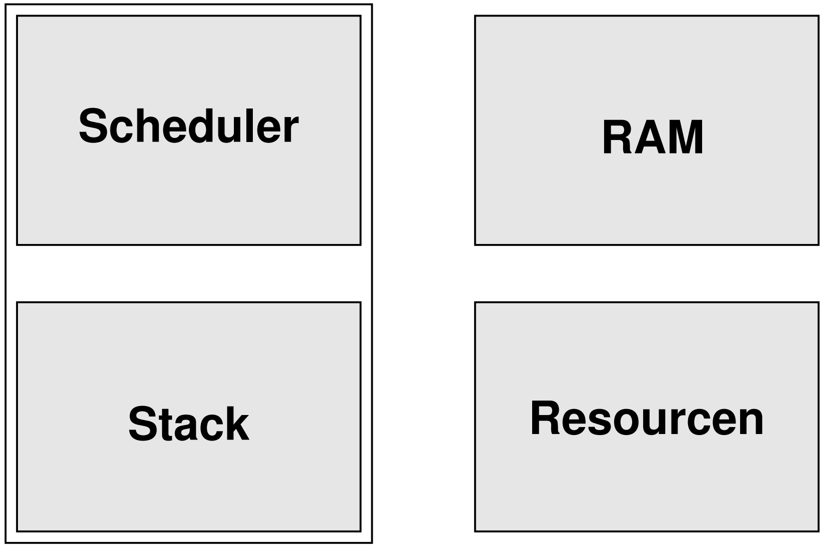
Übersicht

1. InterProcess Communication (IPC)
   - Einführung und Begriffsdefinition
   - Überblick über IPC in Linux
   - Eigenschaften der IPC in Linux
   - Portabilität (Unix, OS/2, Windows NT)
   - Beispiele (Pipe, Thread)
   - Ausblick (Integration, Standards)
   - Literaturhinweise und Empfehlungen
2. Symmetric MultiProcessing (SMP)
   - Einführung und Begriffsdefinition
   - Implementierung im Linux Kern
   - IPC im Zusammenspiel mit SMP
   - Ausblick (Erweiterungen, Standards)

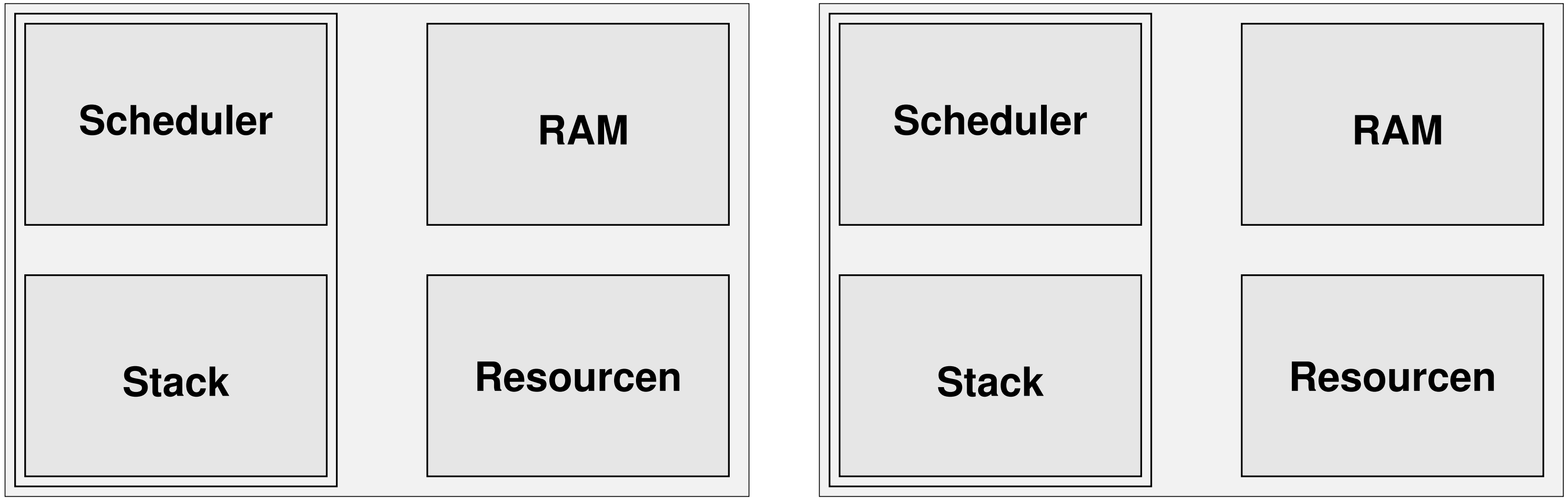
IPC: Einführung und Begriffsdefinition

- Verwendung eines neuen abstrakten Ansatzes, um die grundlegenden Gemeinsamkeiten der IPC in der Fuelle der einzelnen Verfahren nicht aus den Augen zu verlieren
- Die InterProcess Communication IPC ist die Kommunikation zwischen verschiedenen Ausführungsebenen
- IPC wird realisiert durch gemeinsame Resourcen oder Botschaften
- Klassische IPC steht für die IPC, die in Unix Systemen schon lange verbreitet ist (Pipes, Named Pipes, Shared Memory, Signals und Semaphores)
- Begriffe (oft etwas abweichend von der üblichen Definition):
  - Execution Context (Prozessorzeit und Stack)
  - Process (Execution Context und Resourcen)
  - Thread (Execution Context mit geteilten Resourcen)

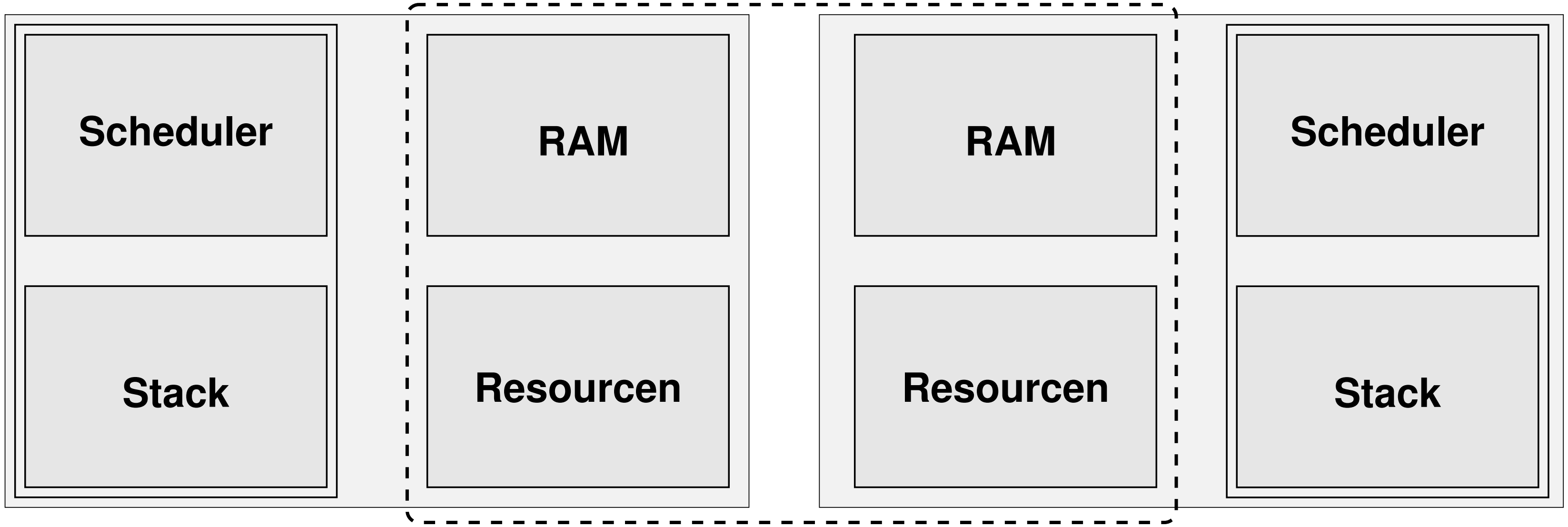
Execution Context



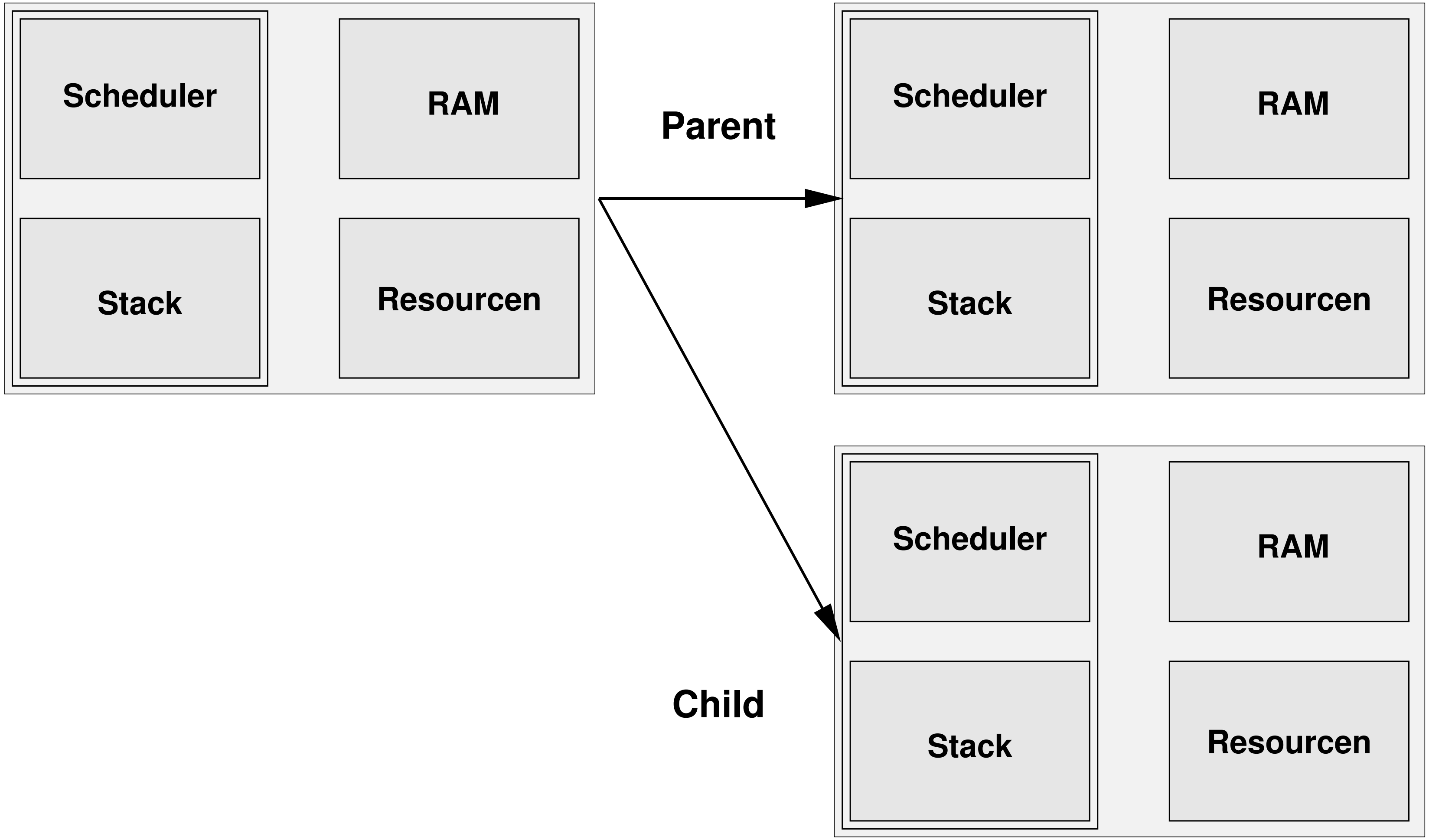
Prozeß


Scheduler
RAM
Stack
Resourcen
Scheduler
RAM
Stack
Resourcen

Thread


Scheduler
Stack
RAM
Resourcen
RAM
Resourcen
Scheduler
Stack

Beziehungen zwischen Unix-Prozessen



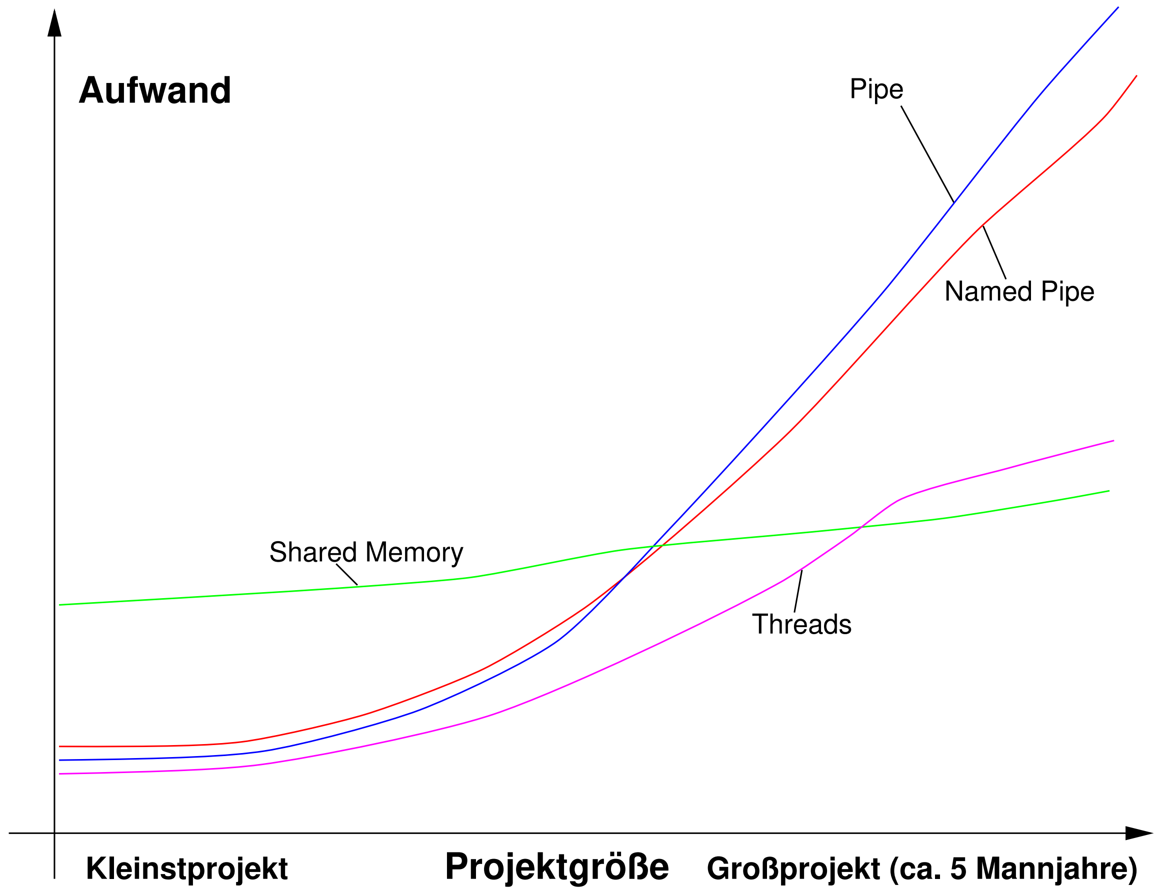
Überblick über IPC in Linux

- In Linux verfügbare IPC:
  - Pipes (Anforderungen und Einsatzmöglichkeiten)
  - Named Pipes (Unterschiede und Gemeinsamkeiten zur Pipe)
  - Shared Memory (Erzeugung und Anwendung)
  - Signals und Semaphores
  - Threads als Bestandteil der IPC
- Möglichkeiten und Beschränkungen der aktuellen Implementierung der IPC

Eigenschaften der IPC unter Linux

- Schnelle IPC (Shared Memory, Threads)
- Einfache IPC (Pipes, Named Pipes)
- Programmieraufwand und Effizienz
- Mögliche Entwicklungen der IPC unter Linux
- Signals und Semaphores als unterstützende Werkzeuge zur Synchronisierung der IPC

Implementationsaufwand in Bezug zur Projektgröße



Portabilität Unix, OS/2, Windows NT

Portabilität Unix

- Ohne Probleme übertragbar:
  - Grundkonzept des Prozesses
  - Pipes und Named Pipes
  - Signals und Semaphores
- Mit geringem Aufwand portierbar:
  - Shared Memory
- Mit größerem Aufwand portierbar:
  - Threads

Portabilität Unix nach OS/2 und Windows NT

Vorbemerkung: Windows 3 IPC = DOS IPC + Messages + Clipboard

- Mit geringem Aufwand portierbar:
  - Named Pipes
  - Signals
  - Semaphores
- Mit mittlerem Aufwand portierbar:
  - Prozeßerzeugung
  - Pipes
  - Shared Memory
- Mit größerem Aufwand portierbar:
  - Threads

Beispiel für Pipes

```
int main(void)
{
    int n, fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (pipe(fd) < 0)
        fatal("pipe error");

    if ((pid = fork()) < 0)
        fatal("fork error");
    else if (pid > 0) {         /* parent */
        close(fd[0]);
        write(fd[1], "hello, world!\n", 13);
    } else {                    /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        puts(line);
    }
```

Beispiel für Threads

```
/* The classic producer-consumer example, implemented with semaphores.
   All integers between 0 and 9999 should be printed exactly twice,
   once to the right of the arrow and once to the left. */

#include <stdio.h>
#include "pthread.h"
#include "semaphore.h"

#define BUFFER_SIZE 16

/* Circular buffer of integers. */

struct prodcons {
  int buffer[BUFFER_SIZE]; /* the actual data */
  int readpos, writepos;   /* positions for reading and writing */
  sem_t sem_read;          /* number of elements av. for reading */
  sem_t sem_write;         /* number of locations av. for writing */
};

/* Initialize a buffer */

void init(struct prodcons * b)
{
  sem_init(&b->sem_write, 0, BUFFER_SIZE - 1);
  sem_init(&b->sem_read, 0, 0);
  b->readpos = 0; b->writepos = 0;
}
```

Beispiel II: `put()` und `get()`

```
/* Store an integer in the buffer */

void put(struct prodcons * b, int data)
{
  /* Wait until buffer is not full */
  sem_wait(&b->sem_write);
  /* Write the data and advance write pointer */
  b->buffer[b->writepos] = data;
  b->writepos++;
  if (b->writepos >= BUFFER_SIZE) b->writepos = 0;
  /* Signal that the buffer contains one more element for reading */
  sem_post(&b->sem_read);
}

/* Read and remove an integer from the buffer */

int get(struct prodcons * b)
{
  int data;
  /* Wait until buffer is not empty */
  sem_wait(&b->sem_read);
  /* Read the data and advance read pointer */
  data = b->buffer[b->readpos];
  b->readpos++;
  if (b->readpos >= BUFFER_SIZE) b->readpos = 0;
  /* Signal that the buffer has now one more location for writing */
  sem_post(&b->sem_write);
  return data;
}
```

Beispiel III: `producer()` und `customer()`

```
/* A test program: one thread inserts integers from 1 to 10000,
   the other reads them and prints them. */

#define OVER (-1)

struct prodcons buffer;

void * producer(void * data)
{
  int n;
  for (n = 0; n < 10000; n++) {
    printf("%d --->\n", n);
    put(&buffer, n);
  }
  put(&buffer, OVER);
  return NULL;
}

void * consumer(void * data)
{
  int d;
  while (1) {
    d = get(&buffer);
    if (d == OVER) break;
    printf("---> %d\n", d);
  }
  return NULL;
}
```

Beispiel IV: Hauptprogramm

```
int main()
{
  pthread_t th_a, th_b;
  void * retval;

  init(&buffer);
  /* Create the threads */
  pthread_create(&th_a, NULL, producer, 0);
  pthread_create(&th_b, NULL, consumer, 0);
  /* Wait until producer and consumer finish. */
  pthread_join(th_a, &retval);
  pthread_join(th_b, &retval);
  return 0;
}
```

Ausblick (Integration und Standards)

- Posix IPC: Obermenge aller hier beschriebener IPC aller beschriebener Systeme.
- Standards: Unix System V (Pipes, Named Pipes, Shared Memory, Streams, Signals, Semaphores)
- Threads: OS/2
- Linux: Auswahl der korrekten Bibliotheken (libc, X11, ...)

SMP

Einführung und Begriffsdefinitionen:

- MP (MultiProcessing): Im betrachteten Bereich bezeichnet es die Anwendung CPUs gleicher Bauart, die sich den Zugriff auf alle übrigen Resourcen teilen.
- SMP (Symmetrisches MultiProcessing): Konkurrenz der CPUs um die Resourcen.
- ASMP (Asymmetrisches MultiProcessing): Verwaltung der Resourcen durch eine CPU.

Implementation im Linuxkern

- Threadfähigkeit des Linuxkerns der aktuellen Benutzerversion
- Unterstützung zur Zeit für die Intel- und die Sparc-Architektur

IPC im Zusammenspiel mit SMP

- Eine gute Verwendung der vorhandenen IPCs bringt die Ausnutzung aller CPUs mit sich
- Anforderungen an die IPC unter SMP:
  - Der Zugriff auf eine IPC Resource darf nur von einer CPU gleichzeitig erfolgen
- Verteilbarkeit von Threads aus SMP

Ausblick

- Verfeinerte SMP-Fähigkeit zukünftiger Linuxkerne
- Viele komplexe Probleme lassen sich (wenigstens teilweise) sinnvoll in parallel lauffähige Probleme unterteilen
- Intel SMP Standard

Literaturhinweise und Empfehlungen

InterProcess-Communication:

- W. Richard Stevens, 1992, **Advanced Programming in the UNIX Environment**.
- Brian W. Kernighan, Dennis M. Ritchie, 1990, **Programmieren in C** Zweite Ausgabe, ANSI C
- Bryan O'Sullivan, 1996, **Answers to frequently asked questions for comp.programming.threads**