Fakultät für Informatik

MNM TEAM

MUNICH NETWORK MANAGEMENT TEAM

Institut für Informatik

LMU

Systempraktikum im Wintersemester 2009/2010 (LMU):
Vorlesung vom 30.10. – Foliensatz 3

Prozesse (T)
CPU-Zuteilung (Scheduling) (T)
Prozesssteuerung in Unix (P)
Kommunikation über Pipes (P)

Dr. Thomas Schaaf, Dr. Nils gentschen Felde

Prozesse

MNM TEAM

- Programm
  - Vom Linker erzeugt,
  - als ausführbare Datei abgelegt,
  - kann durch Systemaufruf `execve()` zur Ausführung veranlasst werden.
- Mögliche Definitionen für "Prozess"
  - Ein Prozess ist ein in Ausführung befindliches Programm.
  - Ein Prozess ist Eigentümer von Ressourcen/Betriebsmitteln.
  - Ein Prozess ist eine Verwaltungseinheit des Betriebssystems, die ein Programm so ausführt, als ob der Prozessor nur diesem Programm zur Verfügung stünde.

Prozesse

MNM TEAM

- Prozesshierarchie
  - Unix: Prozess-Baum
  - Initialer Prozess (Wurzel): init
  - Jeder Prozess (außer init) hat einen Elternprozess (Parent)
  - Ein Prozess kann einen oder mehrere Kindprozesse (Child) erzeugen
- Waisen und Zombies
  - Waise:
    - Prozess in Ausführung, aber Vaterprozess bereits terminiert
    - Folge: Waisen-Prozess wird vom init-Prozess "adoptiert"
  - Zombie:
    - Prozess eigentlich beendet, aber Terminierungs-Signal (Rückggabewert) vom Vaterprozess nicht angenommen/abgerufen
    - Folge: Prozess als Zombie weiterhin in der Prozessliste
    - belegt keine Ressourcen (Speicher)

Prozesskontrolle

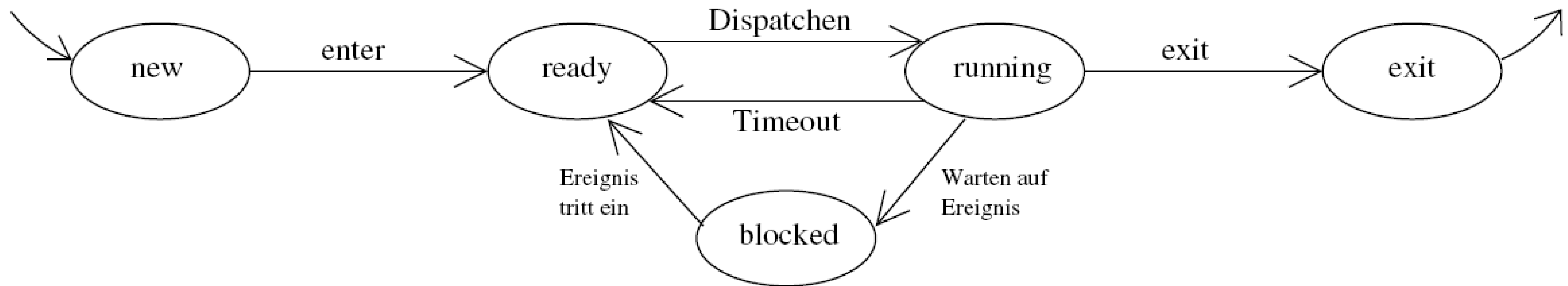
MNM TEAM

- Eigenschaften von Prozessen
  - Prozesse können verschiedene Zustände einnehmen
  - Modellierung durch Zustands-Prozessmodelle (endliche Automaten)
    - Basismodell (Zustände: New/Initialized, Ready/Sleeping, Running, Exit/Terminated)
    - 5-Zustands-Prozessmodell (Basismodell + Blocked/Waiting)
    - 7-Zustands-Prozessmodell (Basismodell + Ready Suspended + Blocked/Waiting + Blocked/Waiting Suspended)
  - Prozesse können nebenläufig sein, d.h.
    - sie rechnen (pseudo-)parallel und
    - sie sind abhängig voneinander (z.B. Erzeuger/Verbraucher-Szenarien).
  - Problem der Nebenläufigkeit
    - Es kann kritische Bereiche geben (z.B. gemeinsam genutzte Variablen/Speicherbereiche)
    - Synchronisation der Prozesse erforderlich

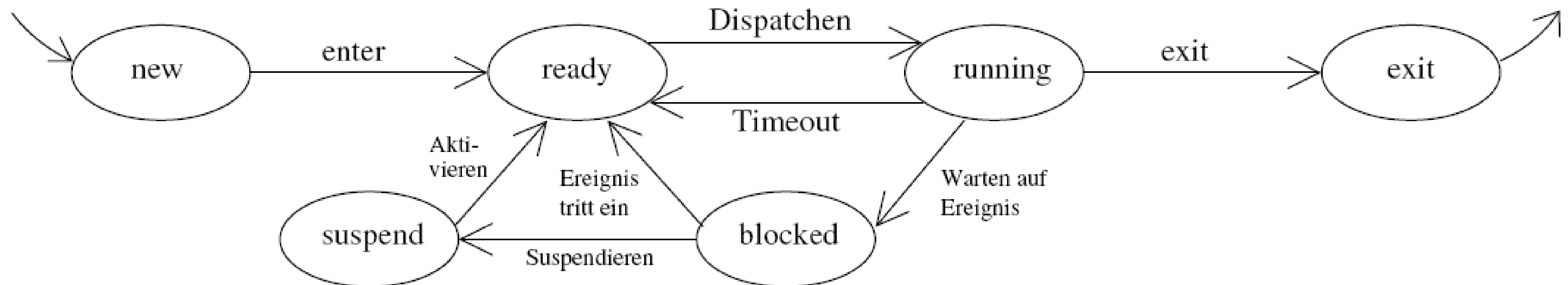
Prozesskontrolle

MNM TEAM

- Beispiel: 5-Zustands-Prozessmodell



- Beispiel: 5-Zustands-Prozessmodell + suspend



Prozesskontrolle

MNM TEAM

- Prozesskontrollblock (PCB)
  - enthält alle Informationen, die zur Beschreibung des aktuellen Ausführungsstatus eines Prozesses benötigt werden
  - Arten von Informationen im PCB:
    - Prozessidentifikations-Informationen
      - Prozess-ID (PID)
      - Elternprozess-ID (PPID)
      - ID des Eigentümers des Prozesses (UID)
    - Prozesszustandsinformationen:
      - Prozesskontext
      - Programm-Statuswort (PSW)
    - Prozesskontrollinformationen:
      - Scheduling-/Zustandsinformationen
      - Datenstrukturen
      - Signale/Nachrichten (zur Interprozesskommunikation)

CPU–Zuteilung (Scheduling)

MNM TEAM

- Scheduling
  - Zuteilung von Prozessor–Rechenzeit an einen Prozess durch das Betriebssystem
  - Klassen von Scheduling–Verfahren:
    - nicht–preemptiv (Run to completion)
    - preemptiv
  - Bewertung von Scheduling–Strategien:
    - Metriken:
      - ➢Antwortzeit
      - ➢Wartezeit
      - ➢Verweilzeit
    - Sonstige Kriterien:
      - ➢Fairness
      - ➢Können Prozesse verhungern?
      - ➢Implementierbarkeit

CPU-Zuteilung (Scheduling)

MNM TEAM

Strategie	Nicht-preemptive Variante?	Preemptive Variante?	Implementierbar?
First Come First Serve (FCFS)			
Shortest Job Next (SJN)			
Shortest Remaining Proceccing Time (SRPT)			
Priority-Scheduling (PS)			
Round Robin			
Multiple Queues (MQ)			

Prozesserzeugung und -ausführung

MNM TEAM

- Informationen über Prozesse
  - C: Funktionen `getpid(), getppid(), getuid(), ...`
  - Shell: Befehle `ps, top`
- Erzeugen von (Kind-)Prozessen
  - C-Befehl `fork()` in <unistd.h>
  - C-Datentyp für Prozess-IDs: `pid_t` in <sys/types.h>
  - Funktionssignatur für `fork()`: `pid_t fork(void);`
    - Rückgabe an aufrufenden Prozess (Parent): Prozess-ID des erzeugten Kindprozesses
    - Rückgabe an erzeugten Prozess (Child): 0

Prozesserzeugung und -ausführung

MNM TEAM

- Erzeugen von (Kind-)Prozessen (Forts.)
  - Was macht `fork()` genau?
    - Anlegen einer identischen Kopie des Elternprozesses
    - Parent und Child haben anfangs den gleichen (aber nicht den selben) PCB, d.h.
      - ➢Kindprozess erbt PCB von Elternprozess
      - ➢Ausnahme: PID und PPID sowie bestimmte Locks, Signale und Zeitwerte
    - Parent und Child sind nach `fork()` prinzipiell vollkommen eigenständige Prozesse
  - Elternprozesse können mittels `wait()` bzw. `waitpid()` gezwungen werden, vor ihrer Terminierung auf Kindprozesse zu warten (erfordert `#include <wait.h>`)

    → siehe Beispiel 4 (fork4.c)

Prozesserzeugung und -ausführung

MNM TEAM

- Beispiel 1: Prozesserzeugung mit `fork()`
  - Wie oft wird "Fork-Test" ausgegeben? (fork1.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {
   pid_t pid;

   pid = fork();
   pid = fork();

   printf("Fork-Test\n");

   return EXIT_SUCCESS;
}
```

Prozesserzeugung und -ausführung

MNM TEAM

- Beispiel 2: Prozesserzeugung mit `fork()`
  - Fallunterscheidung nach `fork()`-Rückgabewerten (fork2.c):

```
#include <...>

int main (void) {
   int x = 5;
   pid_t pid;

   if ((pid = fork()) < 0) {
      fprintf(stderr, "Fehler bei fork().\n");
   } else if (pid == 0) {
      /* Kindprozess */
      printf("Im Kindprozess\n");
      printf("Meine PID = %i\n", getpid());
      printf("Meine Parent-PID = %i\n", getppid());
      printf("Addiere 1: x + 1 = %i\n\n", ++x);
   } else {
      /* Elternprozess */
      printf("Im Elternprozess\n");
      printf("Meine PID = %i\n", getpid());
      printf("Meine Parent-PID = %i\n", getppid());
      printf("Subtrahiere 1: x - 1 = %i\n", --x);
   }
   return EXIT_SUCCESS;
}
```

- Beispiel 3: Prozesserzeugung mit `fork()`
  - Verwendung einer `switch`-Kontrollstruktur (fork3.c):

```
#include <...>

int main (void) {
   pid_t pid;
   switch (pid = fork ()) {
   case -1:
      fprintf(stderr, "Fehler bei fork().\n");
      break;
   case 0:
      /* Kindprozess */
      printf("Im Kindprozess\n");
      printf("Meine PID = %i\n", getpid());
      printf("Meine Parent-PID = %i\n", getppid());
      break;
   default:
      /* Elternprozess */
      printf("Im Elternprozess\n");
      printf("Meine PID = %i\n", getpid());
      printf("Meine Parent-PID = %i\n", getppid());
      break;
   }
   return EXIT_SUCCESS;
}
```

MNM TEAM

- Beispiel 4: Prozesserzeugung mit `fork()`
  - Können Pointer "vererbt" werden? (fork4.c)

```
#include <...>

int main (void) {
   pid_t pid; int x = 1;
   int *ptr = &x;

   pid = fork();
   printf("x = %i, &x = %p\n", x, &x);
   printf("ptr = %p, *ptr = %i\n", ptr, *ptr);

   if (pid > 0) { /* Elternprozess */
     *ptr = *ptr+1;
   }
   else if (pid == 0) { /* Kindprozess */
     sleep(3);
   }
   else { /* Fehler */
     perror("Fehler bei fork."); return -1;
   }

   printf("x = %i, &x = %p\n", x, &x);
   printf("ptr = %p, *ptr = %i\n", ptr, *ptr);
   return EXIT_SUCCESS;
}
```

Ausführung und Ausgabe:

Prozesserzeugung und -ausführung

MNM TEAM

- Die `exec*`-Familie
  - Jedes `exec*`-Kommando dient zur Ersetzung eines Prozess-Images durch ein anderes
  - Varianten:

    `execve(), execl(), execv(), execle(), execvp(), execlp()`
  - Funktionssignaturen: siehe `man`-Pages
  - Rückgabe jeder `exec*()`-Funktion:
    - Fehlerfall: -1
    - Sonst: Keine Rückkehr zum Aufrufer
  - Anwendungsbeispiele:
    - Ausführung eines Konsolenkommandos oder Programms
    - Erzeugen eines Kindprozesses mit `fork()`, dann mit `exec*()` durch ein anderes Programm ersetzen

Überlagerung von Kindprozessen

MNM TEAM

- Beispiel: Überlagerung von Kindprozessen
  - printargs.c:

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

int main(int argc, char **argv) {

   int i;
   for (i=1; i < argc; i++) {
      printf("%i. Argument: %s\n", i, argv[i]);
   }
   return EXIT_SUCCESS;
}
```

Überlagerung von Kindprozessen

MNM TEAM

- Beispiel: Überlagerung von Kindprozessen
  - exec.c:

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

int main (void) {
   pid_t pid;
   switch (pid = fork ()) {
   case -1:
      perror("Fehler bei fork()\n");
      return EXIT_FAILURE;
   case 0:
      execlp("./printargs", "./printargs", "a", "b", "c", NULL);
      break;
   default:
      if (waitpid(pid, NULL, 0) != pid) {
         perror ("Fehler beim Warten auf Kindprozess\n");
         return EXIT_FAILURE;
      }
   }
   return EXIT_SUCCESS;
}
```

Kommunikation über Pipes

- Namenlose Pipes (Unnamed Pipes)
  - Eigenschaften einer Pipe:
    - Unidirektionale Kommunikation
    - Nachrichtenreihenfolge bleibt erhalten (FIFO)
    - Aufbau von Pipes nur zwischen "verwandten" Prozessen
  - Einrichten einer Unnamed Pipe:

```
int pipe(int fd[2]);
```

    - `fd[0]`: Filedeskriptor zum Lesen (Leseseite)
    - `fd[1]`: Filedeskriptor zum Schreiben (Schreibseite)
    - Rückgabe bei Erfolg 0, im Fehlerfall -1
  - Schließen eines Pipe-Filedeskriptors:

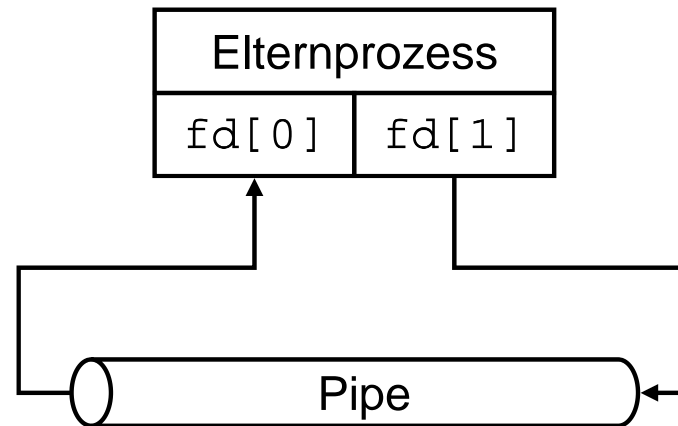
```
close(fd[n]);
```

  - Kindprozesse erben Pipes des Elternprozesses

Kommunikation über Pipes

MNM TEAM

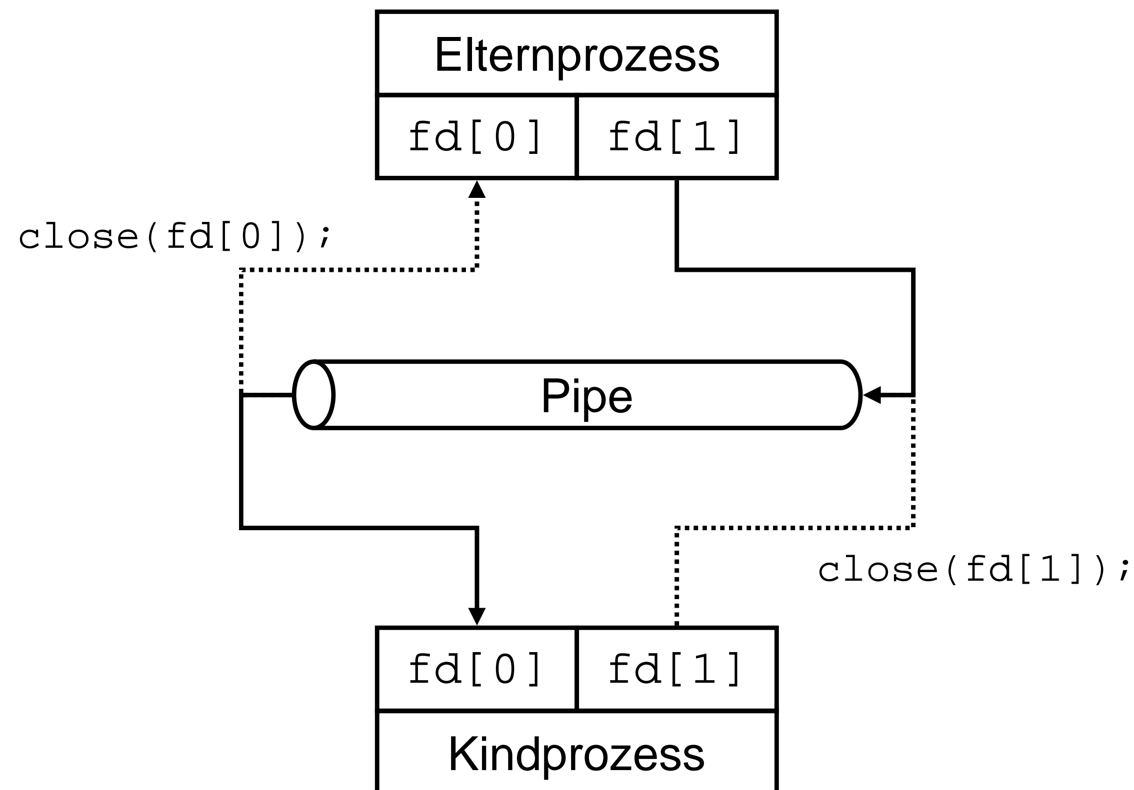
- Pipe zum "Selbstgespräch"



Kommunikation über Pipes

MNM TEAM

- Pipe zur Interprozesskommunikation (nach (!) `fork()`)



Kommunikation über Pipes

MNM TEAM

- Beispiel: Namenlose Pipes (pipes1.c)

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <limits.h>

int main() {
   pid_t pid;
   int fd[2], n = 5;
   char string[] = "abc\n\n";

   if (pipe(fd) < 0) {
      perror ("Fehler beim Einrichten der Pipe.");
      exit(EXIT_FAILURE);
   }

   if ((pid = fork ()) < 0) {
      perror ("Fehler bei fork().");
      exit(EXIT_FAILURE);
   }
```

Kommunikation über Pipes

MNM TEAM

- Beispiel: Namenlose Pipes (pipes1.c) (Forts.)

```
 else if (pid > 0) {
      /*im Elternprozess */
      close(fd[0]); //Leseseite schließen
      if ((write (fd[1], string, n)) != n) { //In Schreibseite schreiben
         perror ("Fehler bei write().");
         exit (EXIT_FAILURE);
      }
      /* Warten auf den Kindprozess */
      if ((waitpid (pid, NULL, 0)) < 0) {
         perror ("Fehler beim Warten auf Kindprozess.");
         exit (EXIT_FAILURE);
      }
   }
   else {
      /*im Kindprozess */
      close(fd[1]); //Schreibseite schließen
      n = read (fd[0], string, PIPE_BUF); //Leseseite auslesen
      if ((write (STDOUT_FILENO, string, n)) != n) {
         perror ("Fehler bei write().");
         exit (EXIT_FAILURE);
      }
   }
   exit (EXIT_SUCCESS);
}
```

Zusammenfassung

MNM TEAM

- Prozesse
- Prozesskontrolle, PCB und Zustandsmodelle
- Prozesshierarchie
- Waisen und Zombies
- CPU-Zuteilung (Scheduling)

- Prozesserzeugung und -ausführung
- Überlagern von Kindprozessen
- Namenlose Pipes zur Interprozesskommunikation