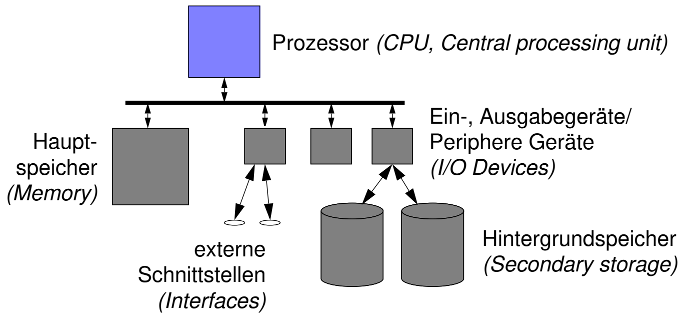
F Prozesse und Nebenläufigkeit

- Einordnung



F.1 UNIX — Prozeßverwaltung

1 Prozeß

- Ausführung eines Programms in der durch die Prozeßdatenstrukturen definierten Umgebung

2 Datenstrukturen zur Ausführung eines Programms

- Programmkode und Programmdaten werden in **Segmenten *(Regions)*** organisiert
- abhängig vom gerade auszuführenden Programm, können diese während der Lebenszeit eines Prozesses gewechselt werden

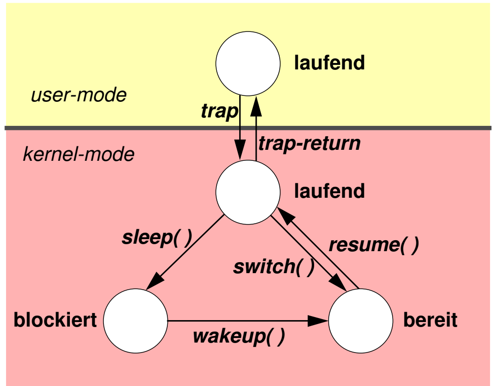
2 Datenstrukturen zur Ausführung eines Programms (2)

- **Text-Segment**
  Maschineninstruktionen des Programms
  (in der Regel schreibgeschützt, wird von mehreren Prozessen gemeinsam benutzt)
- **Daten-Segment**
  Daten des Programms (global und *static*), dynamisch erweiterbar
  (*malloc(3), sbrk(2)*)
- **Stack-Segment**
  lokale Daten und Aufrufparameter von Funktionen
  sowie Sicherungsbereiche für Registerinhalte und Rücksprungadressen
  — wächst bei Bedarf
- **shared Daten-Segment (optional)**
  gemeinsamer Datenbereich für mehrere Prozesse

3 Multiprogramming, Scheduling

- UNIX erlaubt die (quasi-)gleichzeitige Abwicklung mehrerer Prozesse
- Umschaltung zwischen Prozessen durch **Prozeßwechsel *(context switching)***
- Prozeßwechsel erfolgen
  - wenn Prozesse warten müssen (z. B. auf E/A), oder
  - nach einer bestimmten Laufzeit — **Zeitscheibe *(time slice)***
- die Entscheidung, welcher Prozeß als nächstes den Prozessor zugeteilt bekommt ***(Scheduling)*** erfolgt auf der Basis **dynamischer Prioritäten *(multi-level feedback)***

4 Prozeßzustände



5 Prozeßdatenstrukturen

- Prozeßkontrollblock *(Process control block; PCB)*
  - Datenstruktur, die alle nötigen Daten für einen Prozeß hält.
    Beispielsweise in UNIX:
    - Prozeßnummer *(PID)*
    - verbrauchte Rechenzeit
    - Erzeugungszeitpunkt
    - Kontext (Register etc.)
    - Speicherabbildung
    - Eigentümer *(UID, GID)*
    - Wurzelkatalog, aktueller Katalog
    - offene Dateien
    - ...

6 Prozeßwechsel

- Prozeßwechsel unter Kontrolle des Betriebssystems
  - Mögliche Eingriffspunkte:
    - Systemaufrufe
    - Unterbrechungen
  - Wechsel nach/in Systemaufrufen
    - Warten auf Ereignisse
      (z.B. Zeitpunkt, Nachricht, Lesen eines Plattenblock)
    - Terminieren des Prozesses
  - Wechsel nach Unterbrechungen
    - Ablauf einer Zeitscheibe
    - bevorzugter Prozeß wurde laufbereit
- Auswahlstrategie zur Wahl des nächsten Prozesses
  - *Scheduler*-Komponente

7 Erzeugen von Prozessen

- jeder UNIX-Prozeß kann mit dem Systemaufruf ***fork(2)*** einen neuen Prozeß erzeugen
  - *fork( )* erzeugt eine nahezu identische Kopie des aufrufenden Prozesses
  - der *fork( )* aufrufende Prozeß wird **Vaterprozeß *(parent process)*** genannt
  - der durch *fork( )* neu erzeugte Prozeß wird als **Sohnprozeß *(child process)*** bezeichnet
  - der Sohnprozeß erbt alle Rechte und Einschränkungen vom Vaterprozeß
  - wesentliche Unterschiede zwischen Vater- und Sohnprozeß:

fork-Ergebnis:	Vaterprozeß: *pid* des Sohnes, Sohnprozeß: 0
PID:	Sohnprozeß erhält nächste freie *pid*

7 Erzeugen von Prozessen (2)

- Beispiel:

Vater
```
int p;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
} else if( p != -1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```


7 Erzeugen von Prozessen (3)

- Beispiel:

Vater
```
int p;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
} else if( p != -1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

Kind
```
int p;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
} else if( p != -1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```


8 Ausführen eines Programms

- ein UNIX-Prozeß kann die Ausführung eines Programms durch ein anderes Programm ersetzen — ***exec(2), execve(2)***
  - ◆ durch Laden eines neuen Programms werden die zuvor von dem Prozeß bearbeiteten Programm-Datenstrukturen zerstört
    ➥ **im Erfolgsfall gibt es kein return aus *exec( )***
  - ◆ durch Laden eines neuen Programms entsteht **kein neuer Prozeß**
- *execve( )* werden die Argumente ***(argv)*** und ein Environment ***(envp)*** für das neue Programm mitgegeben
- Details:
  - ◆ programmspezifische Daten des Prozesses (Segmente, Signalbehandlungsfunktionen) werden initialisiert
  - ◆ Zugriffsrechte des Prozesses werden auf Eigentümer/Gruppe der Programm-Datei geändert, wenn bei dieser ein ***s-bit*** gesetzt ist

8 Ausführen eines Programms (2)

- … Details:
  - ◆ alle anderen Prozeßparameter bleiben unverändert (*pid, current working directory, ...*)
  - ◆ **offene Dateideskriptoren bleiben erhalten**, es sei denn, sie wurden als *close-on-exec* (siehe ***fcntl(2)***) markiert
- Programmier-Beispiel:

```
if ( (pid = fork()) < 0 ) {
    perror("fork");
    exit(1)
} else if (pid == 0) {           /* child process */
    execl("/bin/cp", "cp",    "/tmp/a", "/tmp/b", (char *)0);
    perror("exec");
    exit(1);
} else {                         /* parent process */
                ...
}
```

9 Terminieren von Prozessen

- Prozesse terminieren, wenn
  - ◆ sie den Systemdienst ***exit(2)*** aufrufen
  - ◆ ein Signal an den Prozeß zugestellt wurde, für das keine Signalbearbeitungsfunktion vorgesehen ist
- die Startumgebung für C-Programme ruft nach einem *return* aus der Funktion *main* automatisch *exit(0)* auf
- dem *exit*-Aufruf kann ein Status-Wert (1 Byte) mitgegeben werden, der durch den Vaterprozeß abgefragt werden kann (→ *wait(2)*)
- ein Vaterprozeß kann auf das Terminieren von Sohnprozessen warten und deren exit-Status abfragen — ***wait(2)***

10 Systemaufruf *wait( )*

- ein Prozeß kann warten, bis ein Sohnprozeß terminiert oder gestoppt wird und dabei den Status des Sohnprozesses abfragen
  ➥ übliche Arbeitsweise einer Shell bei Vordergrundprozessen
- ***wait(2)*** blockiert den aufrufenden Prozeß so lange, bis ein Sohnprozeß im Zustand ZOMBIE existiert oder ein Sohnprozeß gestoppt wird
  - ◆ *pid* dieses Sohnprozesses wird als Ergebnis geliefert
  - ◆ als Parameter kann ein Zeiger auf einen *int*-Wert mitgegeben werden, in dem der Status (16 Bit) des Sohnprozesses abgelegt wird

Sohnprozeß ist	Status & `0xff00`	Status & `0x00ff`
gestoppt	Signalnummer	`0xff`
terminiert durch *exit( )*	exit-code	`0x00`
terminiert durch Signal	`0x00`	Signalnummer
terminiert durch Signal + *coredump*	`0x00`	Signalnummer `+ 0x80`

10 Systemaufruf *wait( )* (2)

- Beispiel:

Vater

```
int status;
int p, c;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
    exit(1);
} else if( p != -1 ) {
    /* parent */
    c = wait(&status);
} else {
    /* error */
    ...
}
```

Kind

```
int status;
int p, c;
...
p= fork();
if( p == 0 ) {
    /* child */
    ...
    exit(1);
} else if( p != -1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```


11 Systemaufrufe — Überblick

- Prozeß erzeugen

```
pid_t fork( void );
```

- Ausführen eines Programms

```
int execve( const char *path, char *const argv[],
            char *const envp[] );
```

- Prozeß beenden

```
void exit( int status );
```

- Prozeßidentifikator

```
pid_t getpid( void );          /* eigene PID */
pid_t getppid( void );         /* PID des Vaterprozesses */
```

- Warten auf Beendigung eines Kindprozesses

```
pid_t wait( int *statusp );
```