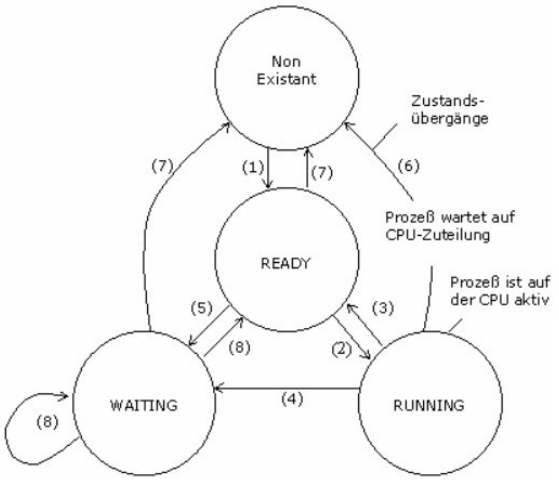
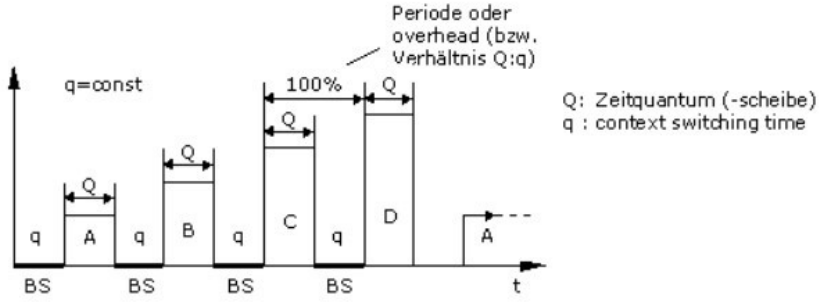
Prozesszustandsmodell

1. Erzeugen eines Prozesses
2. Zuteilen eines Prozessors
3. Entzug des Prozesses (Verdrängung)
4. Selbstsuspendierung
5. Resume (Ereignis eingetroffen ® wecken!)
6. Prozess beendet sich selbst (EXIT)
7. Externes Beenden (KILL)
8. Erhöhen/Erniedrigen der Suspendierungstiefe (auf wie viele Ereignisse muss ein Prozess noch warten, bis er wieder laufen darf)

Anmerkung: Für preemptives Multitasking mindestens Erforderlich: READY/RUNNING



Round Robin Scheduling : Jedem Prozess wird ein Zeitintervall, sein Quantum, zugewiesen, innerhalb dessen er laufen darf. Wenn der prozess am Ende des Quantums noch läuft, wird die CPU entzogen und einem anderen Prozess zugewiesen. Ist das Quantum eines Prozesses abgelaufen, so wird er an das Ende der Liste eingereht. Die einzig interessierende Größe des Round-Robin Verfahrens ist das Quantum. Der Umschaltvorgang von einem zum nächsten Prozess erfordert einen gewissen Zeitaufwand für die Verwaltung. Sichern und laden der Register und der Speicherzuordnungen, Aktualisieren verschiedener Listen und Tabellen, usw. Die Effizient der CPU wird erhöht, wenn das Quantum vergrößert wird.



Q: Eff , t_A (mehr Zeit, bis man wieder "dran" ist)
Q↓: Eff ↓ , t_A ↓ (weniger Zeit, bis man wieder "dran" ist)

Ausgangspunkt: Umschalten von einem Prozess zum anderen:

Effizienz: $E = \frac{Q}{Q+q} \cdot 100\ \%$

Overhead: $h = \frac{q}{Q+q} \cdot 100\ \%$

Zusammenhang zwischen q, Q und Anzahl der Benutzer:

$t_A \le n \cdot (Q+q) - Q$
(n = Anzahl der Prozesse)

$$q \le t_A \le n \cdot (Q+q) - Q$$

Setzt man das Quantum zu klein an, verursacht man eine zu geringe Effizient der CPU durch viele Kontextumschaltungen. Setzt man es zu groß an, verursacht man schlechte Antwortzeiten für kurze interaktive Anforderungen.

Shortest Job Next (SJN): Die Strategie geht davon aus, dass die gesamte Dienstzeit *t* von vornherein bekannt ist. Aus diesem Grund ist sie nur für stapelverarbeitende Prozesse anwendbar. SJN ist eine Strategie ohne Vorrechte, die Prozesse gemäß ihrer gesamten Dienstzeit einordnet.

Priorisiertes Scheduling: eder Prozess erhält eine Priorität und der lauffähige Prozess mit der höchsten Priorität wird aktiviert. Hochpriorisierte Prozesse könnten so die CPU für sich vereinnahmen. Um dieses zu verhindern, könnte ein Scheduler die Priorität eines laufenden Prozesses zu jedem Zeitscheibenende erniedrigen. Fällt seine Priorität unter der des nächsthöheren Prozesses, so erfolgt eine Prozessumschaltung.

Gemischtes Scheduling:

- Einführung von Prioritätsklassen
- Innerhalb der Klassen wird nach Round Robin verfahren
- Zwischen den Klassen wird das priorisierende Scheduling verwendet

Kriterien für globales Scheduling

1. Prozesspriorität
2. Speicherbedarf eines Prozesses (kleine bevorzugen; soll heißen: nicht swappen)
3. Zeit seit der letzten Ein- oder Auslagerung
4. CPU - Zeitbedarf

busy waiting - aktives Warten: nicht effizient. Prozesse voneinander abhängig. Für simple Anwendungen okay, sonst Müll.

Realisierung mit Busmasterinstruktionen: Voraussetzung Hardware Unterstützung

Vorteil: - einfache (elegante) Implementierung
- korrekte Synchronisation

Nachteil: - nicht für alle Prozessoren zugänglich
- immernoch ‚busy waiting'

Deadlocks:

Notwendige Bedingungen für die Entstehung von Deadlocks

- EXKLUSIVITÄT von Ressourcen
- WARTEBEDINGUNG (Prozesse warten auf explizite Freigabe von Ressourcen)
- NICHTENTZIEHBARKEIT von Ressourcen (bereits reservierte Ressourcen können einem Prozess auch zwangsweise nicht entzogen werden)
- BELIEBIGE REIHENFOLGE der Anforderungen (Prozesse können Ressourcen in beliebiger Reihenfolge anfordern)

Strategien zur Behandlung von Deadlocks

*Ignorieren (vogel-strauß-algorithmus)
- in meisten Systemen;
- Bedingung: Absturzgefahr durch Deadlocks weitaus unwahrscheinlicher als Absturz aus anderen Gründen (OS-bedingt)

*Entdecken und beseitigen
a) Automatisch:
  - aktualisieren des Ressourcen-Graphen + sukzessives Eliminieren des Prozesses
b) "intelligent"
  - keine Ressourcen-Graphen
  - Nutzung von TimeOut, Eliminieren von lange inaktiven Prozessen

*deadlock-freie Zuteilung:
  - Prozesse teilen OS bei Start mit, welche Betriebsmittel es benötigen wird
  - OS teilt den Prozessen Ressourcenaccounts zu, welche tabellarisch dazu genutzt werden können, drohende Deadlocks zu verhindern
  - Prozesse lassen sich nun je nach Anforderungen parallel/sequentiell abarbeiten

- In der Datei **/etc/passwd** befindet sich zu jedem autorisierten Benutzer eine Zeile, die Information enthält, wie (user name, Password, Gruppe) in neuer Unix keine Passord wegen Sicherheit.
- **Shell** ist ein Programm, das die Kommandos des Benutzers entgegennimmt, interpretiert und in Systemaufrufe umsetzt, so daß die vom Benutzer geforderten Aktivitäten vom System durchgeführt werden(Unix-Shell nicht Bestandteil des Betriebssystemkerns, sondern ein eigenes Programm).

- Unter Unix gibt es eigentlich **keine Struktur für Dateien**. Eine Datei ist für das System nur eine Folge von Bytes(Die einzigen Ausnahmen sind die Dateiarten, die für die Dateihierarchie und die Identifizierung der Geräte benötigt werden)
- Dateien sind stets in Blöcken von Bytes gespeichert. Dateien Können zumindest theoretisch beliebig lang sein
- **Dateiarten**

1. *Regular Files* (reguläre Dateien, einfache Dateien, gewöhnliche Dateien) Eine solche Datei ist eine Sammlung von Zeichen, die unter den entsprechenden Dateinamen gespeichert sind.
2. *Special Files* (spezielle Dateien, Gerätedateien) Gerätedateien repräsentieren die logische Beschreibung von physikalischen Geräten wie z.B. Bildschirmen, Druckern oder Festplatten.
3. *Directory* (Dateiverzeichnis) Ein Directory enthält wieder Dateien. Es kann neben einfachen Dateien auch andere Dateiarten (wie z.B. Gerätedateien) oder aber auch wiederum Directories enthalten.
4. *FIFO* (first in first out, Named Pipes)dienen der Kommunikation und Synchronisation verschiedener Prozesse.
5. *Sockets* Sockets dienen zur Kommunikation von Prozessen in einem Netzwerk, können aber auch zur Kommunikation von Prozessen auf einem lokalen Rechner benutzt werden.
6. *Symbolic Links* (symbolische Verweise) Symbolische Links sind Dateien, die lediglich auf andere Dateien zeigen.

- In einem Dateinamen sind außer dem Slash (/) und dem NUL-Zeichen alle Zeichen erlaubt(Auch sollte als erstes Zeichen eines Dateinamens nicht +, **-** oder **.** benutzt werden).
- Angenommen, das Working-Directory sei /user1/herbert, dann würde der **relative Pfadname** briefe/finanzamt dem **absoluten Pfadnamen** /user1/herbert/briefe/finanzamt entsprechen.
- **copy1 <datei1 >datei2** [*kopiert datei1 nach datei2*]
- **Elementare E/A-Funktionen** sind in der Headerdatei <unistd.h> deklariert. Wichtige elementare E/A-Funktionen.
    1. ***open*** (Öffnen einer Datei; liefert entsprechenden Filedeskriptor)
    2. ***read*** (Lesen aus einer geöffneten Datei)
    3. ***write*** (Schreiben in eine geöffnete Datei)
    4. ***lseek*** (Positionieren des Schreib-/Lesezeigers in geöffneter Datei)
    5. ***close*** (Schließen einer geöffneten Datei).
- **Prozesse unter Unix**
- Prozess = ein Programm während der Ausführung(Wird das gleiche Programm von unterschiedlichen Benutzern gestartet, so handelt es sich dabei um zwei verschiedene Prozesse, obwohl beide das gleiche Programm ausführen).
- **Prozeß-ID(PID)** :Jedem Prozeß wird vom Betriebssystem eine eindeutige Kennung in Form einer nichtnegativen ganzen Zahl zugewiesen.
- Will ein Prozeß seine PID erfahren, so muß er nur die Systemfunktion ***getpid*** aufrufen.

- **Systemfunktionen zur Prozeßsteuerung**
  1. Kreieren von neuen Prozessen (***fork***)
  2. Prozesse mit anderen Programmcode überlagern (***exec, ...***)
  3. Kommunikation zwischen verschiedenen Prozessen (***pipe ,popen, ...***)
  4. Warten auf die Beendigung von Prozessen (***waitpid,*** …)
- ***fork(): neuen Prozeß zu kreieren. Der neue*** Prozeß ist eine exakte Kopie des aufrufenden Prozesses. Mit Elternprozeß bezeichnet man den aufrufenden und mit Kindprozeß den neu kreierten Prozeß. Die Funktion ***fork*** gibt für den Elternprozeß die nichtnegative PID des neuen Kindprozesses und für den Kindprozeß den Wert 0 zurück.
- Wenn bei der Ausführung einer Systemfunktion ein Fehler auftritt, so liefern viele Systemfunktionen -1 als Rückgabewert und setzen zusätzlich die Variable errno auf einen von 0 verschiedenen Wert.
- Um die Fehlermeldung zu erhalten, die zu einem in errno stehenden Fehlercode gehört, schreibt ANSI C die beiden Funktionen ***perror()*** und ***strerror()*** vor.
- Die Funktion ***perror()*** gibt auf stderr die zum momentan in errno stehenden Fehlercode gehörende Fehlermeldung aus.
- Die Funktion ***strerror()*** liefert die zu einer Fehlernummer (üblicherweise der errno-Wert) gehörende Meldung als Rückgabewert.(gibt zurück: Zeiger auf die entsprechende Fehlermeldung)
- 0 ist die User-ID des besonders privilegierten Superusers, dessen Loginname meist *root* ist. Ein Superuser ha alle Rechte im System.
- **Signale:** sind asynchrone Ereignisse, die erzeugt werden, wenn während einer Programmausführung besondere Ereignisse eintreten.
- Ein Prozess hat drei verschiedene Möglichkeiten, auf das Eintreffen eines Signals zu reagieren:
  1. Ignorieren des Signals
  2. Voreingestellte Reaktion: Für jedes mögliche Signal ist eine bestimmte Reaktion festgelegt.(meist Beendigung des Prozesses)
  3. Ausführen einer eigenen Funktion***:*** Für jedes Signal kann ein Prozeß auch seine eigene Reaktion festlegen. Dazu muß er mit der Funktion signal sogenannte **Signalhandler** (Funktionen) einrichten.
- **Kalenderzeit**: Diese Zeit wird im Systemkern als die Anzahl der Sekunden dargestellt, die seit *00:00:00 Uhr des 1. Januars 1970 (UTC*4*)* vergangen sind. Diese Kalenderzeit, die immer im Datentyp time_t dargestellt wird.
- ***CPU-Zeit:*** Diese Zeit gibt an, wie lange ein bestimmter Prozeß die CPU benutzte. Die CPU-Zeit wird in *clock ticks ("Uhr-Ticks")* pro Sekunde gemessen.(Datentyp clock_t)
- Für einen Prozeß unterhält der Kern drei Zeitwerte:
  1. abgelaufene Uhrzeit seit Start
  2. Benutzer-CPU-Zeit
  3. System-CPU-Zeit

Der Unix-Prozeß

- Prozeß = Programm während der Ausführung
- **Startup-Routine**
  Wird ein Programm vom Kern (mit einer der *exec*-Funktionen) gestartet, so wird immer zuerst eine spezielle Startup-Routine (vor der eigentlichen *main*-Funktion) aufgerufen. Die Startup-Routine sorgt dafür, daß vor dem eigentlichen Aufruf von *main* der Prozeß mit Daten (Kommandozeilenargumente und Environment-Variablen) aus dem Kern versorgt wird.
- Die Prototypdeklaration für ***main*** ist
  `int main(int argc, char *argv[]);`
  - `argc` ist dabei die Anzahl der Argumente auf der Kommandozeile und `argv` ist ein Array von Zeigern auf die einzelnen Argumente.
- **Beendigung eines Unix-Prozesses**
  1. Normale Beendigung
     - normales Beenden der Funktion ***main*** (mit oder ohne **return**)
     - Aufruf der Funktionen ***exit*** oder ***_exit***
  2. Anormale Beendigung
     - Aufruf der Funktion ***abort***
     - durch interne oder externe Signale
- Der Exit-Status für ein Programm ist in folgenden Fällen nicht definiert:
  - Automatische Rückkehr aus der Funktion ***main*** durch Beendigung des Codes.
  - Aufruf von return; in ***main***. /* Keine Angabe eines Rückgabewerts */
  - Aufruf von exit; oder _exit; im Programm.
- **void exit(int *status*); :** Um ein Programm normal zu beenden, wobei zuvor jedoch noch einige »Aufräumarbeiten« durchgeführt werden (wie z.B. alle noch nicht auf Dateien geschriebenen Pufferinhalte **a**uch wirklich physikalisch schreiben).
- **void _exit(int *status*);** : Um ein Programm normal zu beenden, wobei jedoch keinerlei »Aufräumarbeiten« wie bei *exit* durchgeführt werden.
- **Evironment-Liste** : Jeder Unix-Prozeß besitzt seine eigene Umgebung (*environment*). Diese Environment liegt in Form einer Liste vor, die ihm von der Startup-Routine übergeben wird. Die Environment-Liste ist – wie die Argumenten-Liste (argv) – ein Array von Zeigern au Strings.
- Die Adresse dieser Environment-Liste ist immer in der globalen Variablen environ enthalten:
  extern char **environ;
- Zugriff auf die ganze Environment-Liste
  1. *Zugriff über die globale Variable* environ.
  2. *Zugriff über ein drittes Argument in der main-Funktion.*
- **Dynamisches Anfordern von Speicherplatz**
  - ***void* *malloc**(size_t *groesse*); **:** reserviert (allokiert) einen Speicherbereich mit *groesse* Bytes. Die Bytes dieses Speicherbereichs haben keine definierten Werte als Inhalt.
  - ***void* *calloc**(size_t *anzahl*, size_t *groesse*); : reserviert (allokiert) einen Speicherbereich für *anzahl* Objekte mit *groesse* Bytes. Alle Bytes dieses Speicherbereichs werden dabei mit dem Wert 0 initialisiert.
  - void ***realloc**(void **zgr*, size_t *neuegroesse*); **:** verändert die Größe eines bereits zuvor allokierten Speicherbereichs **(***zgr* ist seine Anfangsadresse) auf *neuegroesse* Bytes.

alle drei geben zurück: Adresse des allokierten Speicherbereichs (bei Erfolg); NULL bei Fehler

- ***void free(void *zgr);*** *:* zur Freigabe von dynamisch angefordertem Speicherplatz.
  Falls für zgr eine Adresse eines Speicherbereichs angegeben wird, der nicht zuvor mit malloc, calloc oder realloc allokiert wurde, oder wenn die für zgr angegebene Adresse auf einen Speicherbereich zeigt, der zuvor mit free(zgr) oder realloc(zgr,0) wieder freigegeben wurde, dann liegt laut ANSI C undefiniertes Verhalten vor. In der praktischen Anwendung kann dies katastrophale Folgen für den Prozeß haben, da die ganze Speicherverwaltung inkonsistent wird.

Die Prozeßsteuerung

1. **pid_t getpid(void);** gibt zurück: PID(Process ID) des aufrufenden Prozesses
2. **pid_t getppid(void);** gibt zurück: PPID(Parent Process ID) des aufrufenden Prozesses
3. **pid_t getuid(void);** gibt zurück: reale User-ID des aufrufenden Prozesses
4. **pid_t geteuid(void);** gibt zurück: effektive User-ID des aufrufenden Prozesses
5. **pid_t getgid(void);** gibt zurück: reale Group-ID des aufrufenden Prozesses
6. **pid_t getegid(void);** gibt zurück: effektive Group-ID des aufrufenden Prozesses

Unix-Prozeßhierarchie

- **Scheduler-Prozeß mit PID 0** Dieser Systemprozeß ist Teil des Kerns und erhält normalerweise die PID 0. Er wird oft auch mit swapper bezeichnet.
- **init-Prozeß mit PID 1** Genau wie der Scheduler-Prozeß wird der init-Prozeß niemals beendet. Obwohl der init-Prozeß mit Superuser-Rechten läuft, ist er doch – anders als der Scheduler-Prozeß ein Benutzerprozeß und kein Systemprozeß im Kern.
- **pagedaemon mit PID 2** (auf manchen Systemen) Auf manchen Systemen, die mit virtuellen Speichern arbeiten, wird dieser spezielle Prozeß kreiert. Er ist dabei für das Paging im virtuellen Speicher zuständig. Genau wie der swapper ist der pagedaemon ein Systemprozeß im Kern.

Kreieren von neuen Prozessen

- **pid_t fork(void);**
  gibt zurück: 0 im Kindprozeß; Prozeß-ID des Kindprozesses im Elternprozeß; -1 bei Fehler
- Ein typisches Codestück für die Kreierung eines neuen Prozesses sieht oft wie folgt aus:

```
switch ( rueckgabe=fork() ) {
        case -1:
                /* Fehlermeldung, daß fork-Aufruf nicht erfolgreich war */
        break;
        case 0:
                /* Code fuer den Kindprozeß */
        break;
        default:
                /* Code fuer den Elternprozeß */
        break;
        }
```

- Für einen nicht erfolgreichen ***fork***-Aufruf gibt es zwei Gründe:
  1. Es existieren bereits zu viele Prozesse.
  2. Das obere Limit von Prozessen (`CHILD_MAX` aus `<limits.h>`) ist für die reelle User-ID bereits ausgeschöpft.
- Unterschiede zwischen Eltern- und Kindprozeß
  1. Rückgabewert von ***fork*** (0 bei Kind, PID des Kindprozesses bei Elternprozeß)
  2. unterschiedliche PIDs (Prozeß-IDs)
  3. unterschiedliche PPIDs (Parent Prozeß-Ids)
  4. Dateisperren des Elternprozesses werden nicht an den Kindprozeß vererbt.
  5. Eingeschaltete Zeitschaltuhren des Elternprozesses (mittels ***alarm***) werden beim Kindprozeß ausgeschaltet.
  6. Hängende (noch nicht zugestellte) Signale des Elternprozesses werden nicht an den Kindprozeß vererbt.

- Typische Anwendungen für fork()
  1. Ein Programm soll zu einem Zeitpunkt gleichzeitig zwei verschiedene Codestücke ausführen
  2. Ein Prozeß (wie eine Shell) möchte ein anderes Programm ausführen. Bei diesem Anwendungsfall ruft der Kindprozeß unmittelbar nach der Rückkehr aus ***fork*** die Funktion ***exec***
- Die Funktion ***vfork()*** hat den gleichen Prototyp wie ***fork*** und kreiert ebenso wie ***fork*** einenKindprozeß. Anders als ***fork*** kopiert ***vfork*** nicht den Adreßraum des Elternprozesses, sondern läßt dem Kindprozeß den Adreßraum des Elternprozesses mitbenutzen, bis der Kindprozeß ***exec*** oder ***exit*** aufruft..
- **vfork()** garantiert wird, daß der Kindprozeß zuerst (also vor dem Elternprozeß) abläuft, bis er entweder exec oder exit aufruft. Es ist darauf hinzuweisen, daß dies zu einem Deadlock führen kann, wenn der Kindprozeß auf Aktionen des Elternprozesses wartet, bevor er exec oder exit aufruft.

Warten auf Beendigung von Prozessen

- In jedem Fall kann der Elternprozeß den Beendigungsstatus eines Kindprozesses mit einer der beiden Funktionen ***wait*** und ***waitpid*** erfahren.
- **Verwaiste Kindprozesse :** Wenn ein Elternprozeß sich beendet, bevor alle seine Kindprozesse beendet sind, so wird der init-Prozeß der Elternprozeß von allen dessen Kindprozessen. Der Kern setzt dies um, indem er bei jeder Beendigung eines Prozesses die PID aller aktiven Prozesse überprüft. Besitzt ein noch aktiver Prozeß als PPID die PID des gerade beendeten Prozesses, so erhält er als neue PPID die Nummer 1 (Prozeß-ID von init). So ist immer sichergestellt, daß jeder Prozeß einen Elternprozeß hat.
- **Zombie-Prozesse :** Kindprozesse, die sich beendet haben, ohne daß der Elternprozeß auf sie wartete, werden in Unix als *Zombies* bezeichnet und beim Kommando ps wird für ihren Zustand Z ausgegeben.
  der Kern hält sich über jeden beendeten Prozeß eine gewisse Menge an Informationen , so daß der Elternprozeß auch nachträglich diese Information mittels einer der beiden Funktionen ***wait*** oder ***waitpid*** erfragen kann. Die dabei vom Kern aufgehobene Information umfaßt mindestens die PID, Beendigungsstatus und verbrauchte CPU-Zeit des beendeten Prozesses.
  Wenn verwaiste Kindprozesse, die als neuen Elternprozeß den init-Prozeß zugeordnet bekamen, sich beenden, so ruft init automatisch eine der ***wait***- Funktionen auf, um ihren Beendigungsstatus zu erfragen. So ist sichergestellt, daß init- Kindprozesse niemals Zombies werden und das System nicht unnötig belasten.
- **pid_t wait(int ****status***);**
  pid_t waitpid(pid_t *pid*, int ****status*, int *optionen*);** beide geben zurück: Prozeß-ID (bei Erfolg); 0 (WNOHANG wurde angegeben und kein Kindprozeß aktiv); -1 (bei Fehler)

- Ein Aufruf der Funktionen ***wait*** oder ***waitpid*** kann folgendes Verhalten nach sich ziehen:
  1. Sofortige Rückkehr von ***wait*** bzw. ***waitpid*** mit dem Beendigungsstatus eines Kindprozesses, wenn ein Kindprozeß sich bereits früher beendet hat und der Kern nur auf die Abholung des Beendigungsstatus dieses Zombieprozesses wartet.
  2. Sofortige Rückkehr mit Fehler, wenn keine Kindprozesse existieren.
  3. Blockierung des aufrufenden Prozesses, wenn alle Kindprozesse immer noch aktiv sind.

- ***wait*** und ***waitpid*** unterscheiden sich in den drei folgenden Punkten:
  1. ***wait*** wartet nur auf die nächste Beendigung eines beliebigen Kindprozesses, während ***waitpid*** auf die Beendigung eines bestimmten Kindprozesses warten kann.
  2. ***wait*** kann den aufrufenden Prozeß blockieren, bis sich ein Kindprozeß beendet, während bei ***waitpid*** mit einer Option die Blockierung des aufrufenden Prozesses unterbunden werden kann.
  3. ***waitpid*** unterstützt anders als ***wait*** die Jobkontrolle
- Beide Funktionen geben bei Erfolg die Prozeß-ID des Kindprozesses zurück, der sich beendet hat. Bei Fehler geben beide Funktionen -1 zurück. Mögliche Fehlersituationen sind dabei bei ***wait*** und ***waitpid***, daß kein Kindprozeß existiert oder daß ***wait*** bzw. ***waitpid*** durch ein Signal unterbrochen wurden. Bei ***waitpid*** führt zusätzlich die Nicht-Existenz eines angegebenen Prozesses oder einer- Prozeßgruppe oder aber, daß es sich bei der angegebenen ID nicht um einen Kindprozeß handelt, zu einem Fehler.
- Beide Funktionen schreiben den Beendigungsstatus an die Adresse, die mit dem Argument `status` übergeben wird. Falls der Aufrufer nicht an dem Beendigungsstatus interessiert ist, kann er für `status` einen `NULL`-Zeiger angeben.
- Das Argument pid legt bei ***waitpid*** fest, auf was zu warten ist

1. pid == -1 : Auf Beendigung eines beliebigen Kindprozesses warten; identisch zu *wait*
2. pid > 0 : Auf Beendigung des Kindprozesses mit der Prozeß-ID pid warten.
3. Pid == 0 : Auf Beendigung eines Kindprozesses warten, dessen Prozeßgruppen-ID gleich der Prozeßgruppen-ID des aufrufenden Prozesses ist.
4. Pid < -1 Auf Beendigung eines Kindprozesses warten, dessen Prozeßgruppen-ID gleich dem absoluten Wert von pid ist.

- Wenn man ein Programm erstellt, in dem man einen Kindprozeß kreiert, auf dessen Ende man nicht warten möchte, man aber auch gleichzeitig verhindern möchte, daß dieser Kindprozeß ein Zombie wird, so muß man ***fork*** zweimal aufrufen.

- ***wait3*** und ***wait4 :*** Um auf das Ende eines Prozesses zu warten und dann bei Prozeßende zu erfahren, welche Ressourcen vom beendeten Prozeß und allen seinen Kindprozessen benutzt wurden
- ***wait4*** entspricht weitgehend der Funktion ***waitpid***. Der einzige Unterschied ist, daß bei ***wait4*** zusätzlich Informationen über die Ressourcenbenutzung (viertes Argument usage) des beendeten Prozesses zurückgegeben werden.
- ***wait3*** gleicht andererseits der Funktion ***wait4***, erlaubt es dem Aufrufer aber nicht festzulegen, auf welchen Kindprozeß zu warten ist.

- Synchronisationsprobleme zwischen Elternund Kindprozessen **:**
  - Synchronisationsprobleme (race condition) zwischen Eltern- und Kindprozessen treten immer dann auf, wenn Eltern- und Kindprozesse voneinander abhängig sind.
    - Eine einfache Möglichkeit für den Kindprozeß auf die Beendigung des Elternprozesses zu warten, ist die Angabe der folgenden Schleife (Polling-Methode)

      ```
      while (getppid() != 1)
      sleep (1);
      ```

      (-Sie verbraucht CPU-Zeit
      -nur eingesetzt werden kann, wenn sich der Elternprozeß nach Ausführung der Aktion beendet.)

    - Synchronisation von Eltern- und Kindprozeß mit Signalen.

Die exec-Funktionen

- Ein exec-Aufruf bewirkt keine Änderung der Prozeß-ID, da nicht wie bei fork ein neuer Prozeß kreiert wird, sondern nur die Segmente (Textsegment, Datensegment, Heap und Stack) des aktuellen Prozesses durch das neue Programm überschrieben werden.
- Es stehen sechs verschiedene ***exec***-Funktionen zur Verfügung.

1. int execl(const char *pfadname, const char *arg0, ... /* NULL */);
2. int execv(const char *pfadname, char *const argv[]);
3. int execle(const char *pfadname, const char *arg0, .../* NULL, char *const envp[] */);
4. int execve(const char *pfadname, char *const argv[], char *const envp[]);
5. int execlp(const char *dateiname, const char *arg0, ... /* NULL */);
6. int execvp(const char *dateiname, char *const argv[]);

alle sechs geben zurück: -1 bei Fehler; keine Rückkehr (bei Erfolg)

- die Funktion *execve*, stellt unter Linux einen Systemaufruf dar. Die restlichen Funktionen sind in der C-Bibliothek implementiert und rufen ihrerseits *execve* auf.
- **l** (list) , **v** (*vector*), **p** (path) : diese Funktion einen Dateinamen (und nicht einen Pfadnamen) als Argument erwartet. , **e** (*environment*) Die Funktion erwartet die Environment-Liste als Vektor (envp[]) und benutzt nicht die aktuelle Environment.
- Interpretation des Dateinamens (bei execlp und execvp)
- Wenn eine Funktion als Argument einen Dateinamen erwartet (*execlp* und *execvp*), dann gilt folgendes:
  1. Enthält der beim Aufruf angegebene Dateiname einen Slash (/), so wird er als Pfadnameinterpretiert,
  2. andernfalls wird in den PATH-Directories nach einer ausführbaren Datei diesen Namens gesucht. Wird eine solche Datei gefunden, so wird sie für den Fall, daß sie Maschinencode enthält, direkt gestartet. Enthält sie keinen Maschinencode, so wird angenommen, daß es sich um Shellskript handelt und zur Ausführung dieser Datei wird / bin/sh aufgerufen.
- **Die Funktion system**
  - ➢ intern die drei Funktionen ***fork***, ***exec*** und ***waitpid*** aufruft
  - ➢ int **system**(const char **kdozeile*); Zurückgabe von
    1. Beendigungsstatus der Shell im Format von waitpid, wenn alle drei Funktionen (fork, exec und waitpid), die von system aufgerufen wurden, erfolgreich ausgeführt werden .
    2. -1, wenn das interne fork fehlschlug, oder der interne waitpid-Aufruf einen anderen Fehler als EINTR geliefert hat.
  - ➢ Der Vorteil der Verwendung von *system* gegenüber einer eigenen Nachbildung eines *system*-Aufrufs mittels *fork*, *exec* und *waitpid* ist, daß *system* die erforderlichen Fehler und Signalbehandlung von sich aus durchführt.

Signale

- Signale sind sogenannte *Interrupts* (Unterbrechungen), die von der Hardware oder Software erzeugt werden, wenn während einer Programmausführung besondere Ausnahmesituationen auftreten, wie z.B. Division durch 0 oder Drücken der Programmabbruchtaste (*Strg-C* oder *DELETE*) durch den Benutzer.
- Bei dem Signalkonzept richtet ein Prozeß sogenannte ***Signalhandler*** ein, indem er dem Kern sagt: *Wenn dieses bestimmte Signal auftritt, dann tue bitte folgendes!* Solche Signalhandler lassen sich mit der Funktion ***signal*** einrichten.
- **void (*signal(int *signr*, void (**sighandler*)(int)))(int); :** gibt zurück: Adresse des zuvor eingerichteten Signalhandlers
  Mit der Funktion signal kann man dem Kern mitteilen, was zu tun ist, wenn ein bestimmtes Signal auftritt.
  Signr: legt die Nummer des Signals fest, für das ein Signalhandler einzurichtenist.
  Sighandler: gibt die Adresse der Funktion an, die aufzurufen ist, wenn das Signal signr auftritt. Es bestehen drei verschiedene Möglichkeiten der Angabe:
  1. Signal ignorieren (Angabe: SIG_IGN):für alle Signale außer SIGKILL und SIGSTOP möglich.
  2. Default-Aktion einrichten (Angabe: SIG_DFL): meistens die Default-Aktion ist Beendigung des Prozesses.
  3. Signal abfangen (Angabe: *Adresse einer Funktion*)
- Signale SIGKILL und SIGSTOP können nicht abgefangen werden. Der kern führt immer die Standardaktionen aus.
- Wenn ein **Prozeß mit fork** einen Kindprozeß erzeugt, so erbt der Kindprozeß alle eingerichteten Signalhandler des Elternprozesses, da bei der Kreierung des Kindprozesses immer automatisch die Adressen der Signalhandler-Routinen mitkopiert werden.
- Wenn ein Prozeß ein neues Programm **mit der *exec*-Funktion** startet, so wird außer bei den zu ignorierenden Signalen bei allen anderen Signalen die Default-Aktion eingerichtet. da die Adressen der Signalhandlerfunktionen für das neue Programm keine Gültigkeit mehr haben.
- **Sigpending():**Zum Blockieren von Signalen oder zum Erfragen von hängenden Signalen
- Zu jedem Signal gibt es einen symbolischen Namen, der immer mit `SIG` beginnt und für eine Nummer steht, wie z.B. der Name **`SIGINT`** für das Signal, das generiert wird, wenn der Benutzer die Programmabbruchtaste (*Strg-C*) drückt.
- **Probleme mit der signal-Funktion:**
  1. Erfragen des aktuellen Signalstatus ohne Änderung nicht möglich
  2. Zeitspanne zwischen dem Auftreten eines Signals und dem daraus resultierenden Aufruf der *signal*-Funktion. in dieser Zeit kann erneut das gleiche Signal geschickt werden.
  3. Endlosschleifen beim Warten auf das Eintreten von Signalen
- Um Signalmengen repräsentieren zu können, benötigt man einen eigenen Datentyp. Datentyp sigset_t und die folgenden fünf Funktionen zur Manipulation von Signalmengen vor.
  1. ***Sigemptyset:*** entfernt alle Signale aus der Signalmenge, auf die set zeigt.
  2. ***Sigfillset :*** fügt alle vorhandenen Signale zu der Signalmenge hinzu, auf die set zeigt.
  3. ***sigaddset :*** ein einzelnes Signal zu dieser Signalmenge hinzufügen
  4. ***sigdelset :*** ein einzelnes Signal aus dieser Signalmenge entfernen.
  5. ***Sigismember:*** kann man erfragen, ob das Signal signr in der Signalmenge enthalten ist.

- **int sigaction(int signr, const struct sigaction *neu_handler, struct sigaction *alt_handler);** gibt zurück: 0 (bei Erfolg); -1 bei Fehler : Um für ein Signal einen Signalhandler neu einzurichten oder den momentan eingerichteten Signalhandler zu erfragen oder auch beides.

- **int sigpending(sigset_t *_smenge_);** gibt zurück: 0 (bei Erfolg); -1 bei Fehler : Um die Menge von Signalen zu erfragen, deren Zustellung blockiert ist und die momentan hängen Die Menge der momentan hängenden Signale schreibt die Funktion ***sigpending*** an die Adresse `smenge`.
- Signalhandler sollten also grundsätzlich nur Reentrant-Funktionen (Funktionen, die problemlos während ihrer Ausführung wieder aufgerufen werden dürfen) aufrufen.
- int **raise**(int *signr*); : Mit der Funktion ***raise*** kann sich ein Prozeß selbst ein Signal schicken.
- int **kill**(pid_t *pid*, int *signr*); Um anderen Prozessen ein Signal zu schicken
  Wird beim Aufruf von ***kill*** für das Argument `signr` die 0 (*Null-Signal*) angegeben, so sendet ***kill*** kein Signal, sondern führt lediglich eine Prüfung durch, ob an den betreffenden Prozeß oder die Prozeßgruppe ein Signal geschickt werden kann. Das Nullsignal wird meist geschickt, um zu überprüfen, ob ein bestimmter Prozeß noch existiert. Falls der betreffende Prozeß nämlich nicht mehr existiert, so liefert ***kill*** als Rückgabewert -1 und setzt `errno` auf `ESRCH`.
- **void abort(void);** abort kehrt niemals zurück : bewirkt eine anormale Programmbeendigung. Abort schickt dem aufrufenden Prozeß das Signal `SIGABRT`. Dieses Signal sollte niemals von einem Prozeß ignoriert werden.
- ***setjmp*** und ***longjmp*** dagegen ist ein Rücksprung zu einem indirekten Aufrufer möglich.
- Mit den beiden Funktionen ***setjmp*** und ***longjmp*** ist es möglich, aus einer beliebig tief in der Aufrufhierarchie befindlichen Funktion an einen zuvor durchlaufenen und markierten Punkt (***setjmp***) – über mehrere Ebenen hinweg – zurückzukehren (***longjmp***)
- die zwei Funktionen ***sigsetjmp*** und ***siglongjmp*** , die man immer bei nicht-lokalen Sprüngen aus Signalhandlern (anstelle von ***setjmp*** und ***longjmp***) verwenden sollte. (Der einzige Unterschied ist das zusätzliche Argument erhalte_smask bei der Funktion sigsetjmp)
- **int pause(void);** Um einen Prozeß zu suspendieren, bis ein Signal eintrifft
- **unsigned int alarm(unsigned int sekunden);** gibt zurück: 0 oder Anzahl der Sekunden, bis eine zuvor eingerichtete Zeitschaltuhr abläuft.
  - ➢ Wenn die mit ***alarm*** eingerichtete Zeitschaltuhr abgelaufen ist, wird das Signal SIGALRM generiert. Wird dieses Signal ignoriert oder nicht abgefangen, so beendet sich der Prozeß, was die voreingestellte Default-Aktion für dieses Signal ist.
  - ➢ Das Argument sekunden gibt an, in wieviel Sekunden die Zeitschaltuhr ablaufen und das Signal SIGALRM generiert werden soll. Es ist dabei zu beachten, daß diese mit sekunde angegebene Zeit für den Ablauf der Zeitschaltuhr nicht immer genau eingehalten werde kann, denn zwischen dem Generieren des Signals durch den Kern und der Zustellung an den Prozeß vergeht weitere Zeit, welche vor allen Dingen durch Verzögerungen bei Prozessor-Scheduling nicht ganz unerheblich sein kann.
  - ➢ Eine typische Anwendung für ***alarm*** ist das Festlegen einer oberen Zeitgrenze für eineAktion, die blockiert werden kann.

Pipes und FIFOs

Pipes

- Pipes besitzen grundsätzlich die beiden folgenden Eigenschaften:
  1. Pipes können nur zwischen Prozessen eingerichtet werden, die einen gemeinsamen Vorfahren (Elternprozeß, Großelternprozeß) haben. (Kinderprozeß erbt Pipes von Eltern)
  2. Pipes sind halbduplex, die Daten immer nur in eine Richtung fließen können. So kann z.B. ein Prozeß, der eine Pipe zum Schreiben eingerichtet hat, in diese Pipe nur schreiben, während der Prozeß auf der anderen Pipe-Seite nur aus dieser lesen kann. Soll die Kommunikation auch in umgekehrter Form stattfinden, muß hierfür eine weitere Pipe eingerichtet werden.
- FIFOs und benannte Stream Pipes (haben Einschränkung 1 nicht) und Stream Pipes (haben Einschränkung 2 nicht).
- **int pipe(int *fd*[2]);** gibt zurück: 0 (bei Erfolg); -1 bei Fehler : Um eine Pipe einzurichten
  `fd[0]` geöffneter Filedeskriptor zum Lesen aus der Pipe
  `fd[1]` geöffneter Filedeskriptor zum Schreiben in die Pipe
- Wenn eine Seite einer Pipe geschlossen wird, so gelten die beiden folgenden Regeln:
  1. Beim Lesen aus einer Pipe, deren Schreibseite geschlossen wurde, nachdem alle Daten aus dieser Pipe gelesen wurden, liefert ***read*** 0
  2. Beim Schreiben in eine Pipe, deren Leseseite geschlossen wurde, wird das Signal `SIGPIPE` (gebrochene Pipe) generiert. Sowohl beim Ignorieren als auch beim Abfangen des Signals (nach der Rückkehr aus dem Signalhandler) liefert ***write*** einen Fehler als Rückgabewert, wobei `errno` auf `EPIPE` gesetzt wird.
- **FILE *popen(const char ****kdozeile*****, const char ****typ*****);** gibt zurück: Dateizeiger (bei Erfolg); `NULL` bei Fehler : zu einem schon existierenden Programm eine Pipe einrichten. Die Funktion ***popen*** richtet also zwischen dem aufrufenden Prozeß und dem Programm `kdozeile`, das gestartet wird, eine Pipe ein. Für `typ` ist entweder »`r`« (für Lesen aus der Pipe) oder »`w`« (für Schreiben in die Pipe) anzugeben. Der Rückgabewert ist ein Dateizeiger für diese Pipe.
- ***popen*** nimmt dem Programmierer in diesem Fall viel Arbeit ab, indem es die folgenden Aktionen ausführt:
  1. Einrichten einer Pipe mit pipe
  2. Kreieren eines Kindprozesses mit fork
  3. Schließen der nicht benutzten Seiten der Pipe in Eltern- und Kindprozeß
  4. Überlagern des Kindprozesses durch ein Shellprogramm mit einem exec-Aufruf, um die entsprechende kdozeile ausführen zu lassen
  5. Warten auf die Beendigung des Kommandos kdozeile
- Eine mit ***popen*** eingerichtete Pipe sollte mit ***pclose*** wieder geschlossen werden. ***Pclose*** wartet auf die Beendigung von `kdozeile` und liefert den Beendigungsstatus von `kdozeile` als Rückgabewert.

Benannte Pipes (FIFOs)

- Während normale Pipes nur zwischen Prozessen verwendet werden können, wenn ein gemeinsamer Vorfahre die entsprechende Pipe kreiert hat, können die sogenannten FIFOs zwischen beliebigen Prozessen zum Austauschen von Daten benutzt werden.
- int **mkfifo**(const char **pfadname*, mode_t *modus*); gibt zurück: 0 (bei Erfolg); -1 bei Fehler : Um eine benannte Pipe zu kreieren
- ***mkfifo*** legt im Dateisystem eine Datei mit dem Namen `pfadname` an. Diese Datei ist keine normale Datei, sondern eine *FIFO*. Eine *FIFO* ist eine der verschiedenen Dateiarten.

- Nachdem man eine FIFO mit ***mkfifo*** kreiert hat, kann man diese mit ***open*** öffnen. Dann können die für normale Dateien angebotenen elementaren E/A-Funktionen (***read***, ***write***, ***close***, ***unlink*** usw.) für die FIFO verwendet werden.
- Während Pipes auf Shellebene nur für lineares Pipelining verwendet werden können, können FIFOs auch für nicht-lineares Pipelining verwendet werden.

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Hamzih Abukhashab

Osama Alabaji

System Programmierung 1. Laboraufgabe

Ws 2018/19

2.1

Exec Familie: die Auswahl eines Buchstabes wird so definiert:

Buchstabe Bedeutung e Es wird ein Environment liste (Umgebungsvariablen) als Vektor erwartet. l Es werden die Kommandozeilenargumente in Form einer Liste erwartet. v Es werden die Kommandozeilenargumente in Form eines Vektors erwartet. p Es wird ein Dateiname und nicht ein Pfadname als Argument zum Aufruf des Programms erwartet.

Beispiel:

Besonderheit von lp ist, dass das erste Argument zu execlp() nicht ein absoluter Dateiname sein muss, wie dies bei der Listenform von execle() der Fall ist.

```
/* exec1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd. h.>
int main (void) {
   return execlp("/bin/ls","ls","-l",getenv("HOME"),NULL);
}

execvp()  downtop
```

Ähnlich wie eben execlp(), nur werden die Argumente als Vektor übergeben.

```
/* exec2.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd. h.>
int main (void) {
   char *argumente[4];
   argumente[0] = "ls";
   argumente[1] = "-l";
   argumente[2] = getenv("HOME");
   argumente[3] = NULL;

   return execvp ("/bin/ls", argumente);
}
```

2.2

2.2.1. Welchen Wert liefert der Aufruf fork() zurück? Welche Bedeutung hat dieser für Parent u. Child?

Bei erfolgreichem Fork ist Rückgabewert für den Parent die PID des Child-Prozesses, der Rückgabewert an den Child ist 0. Bei einem Fehler wird dem Parent -1 ausgegeben. (Es wurde kein Child-Prozess erstellt.)

2.2.2. Was realisieren die Aufrufe wait() und waitpid()? Welche Aufrufvarianten gibt es? Suchen Sie mit Hilfe des Manuals nach verwandten Aufrufen. Wann erscheint der Einsatz welches Aufrufes bzw. welcher Aufrufvariante sinnvoll?

Beide Aufrufe pausieren den Prozess, bis ein Child-Prozess einen bestimmten Status erhält. Wait wird eingesetzt, wenn nur ein Child-Prozess existiert. Wenn mehrere Child-Prozesse existieren, wird WaitPid verwendet. Der Parameter PID bestimmt dabei den Child-Prozess, auf den gewartet wird.

2.2.3.Welche Möglichkeiten hat der Parent, verschiedene beendete Child-Prozesse zu unterscheiden?
Beim Beenden eines Child-Prozesses wird die PID übergeben.

2.2.4. In welchem Zustand befindet sich ein Child-Prozess, der zeitlich vor dem Erreichen des wait-Aufrufes seines Parent-Prozesses terminiert, warum und für wie lange?

Der Child-Prozess befindet sich dann im Zombie-Zustand. Er wird erst aus der Prozess-Liste entfernt, wenn der Parent-Prozess beendet wird.

2.2.5.

a. Wann und warum wird der exit-Aufruf verwendet?

Am Ende eines Prozesses wird exit aufgerufen, damit der Speicher entleert wird (z.B. temporäre Dateien werden gelöscht).

b. Ist dessen Gebrauch zwingend?

Exit muss nicht explizit aufgerufen werden.

c. Welche Aufrufvarianten kennen Sie und was bewirken diese exakt?

1) Indirekter Aufruf durch Rückkehr aus der main-Funktion -> Programm wird ohne Exit-
   Status beendet
2) Direkter Aufruf von Exit -> Programm wird mit Exit-Status beendet
3) Return in main-Funktion -> Programm wird mit Exit-Status beendet

d. Liefert der Aufruf einen Wert zurück?
Nein

2.2.6. Welches Format hat die Exit-Status-Variable, die beim Terminieren über den wait-Aufruf an den Parent übergeben wird, und wie muss dieses bei deren Auswertung berücksichtigt werden?

Exit-Status ist ein Integer. 1 gilt als fehlerhafter Ablauf des Prozesses, 0 gilt als erfolgreicher Ablauf.

3.3 Ermitteln Sie mit Hilfe eines einfachen C-Programms die tatsächliche Anzahl der auf Ihrem System pro User bzw. pro User-Prozess maximal möglichen Child-Prozesse. Ist diese konstant? Begründung?

je, nach dem Benutzer es wird limitiert standardmäßig auf 709, trotzdem dies kann angepasst und gecheckt werden mithilfe des ulimit Befehles.

```
ls3523:~ # ulimit -a
core file size          (blocks, -c) 0 data seg
size           (kbytes, -d) unlimited file size
(blocks, -f) unlimited pending signals
(-i) 4091 max locked memory       (kbytes, -l)
32 max memory size         (kbytes, -m)
unlimited open files                    (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
stack size              (kbytes, -s) 8192 cpu time
(seconds, -t) unlimited max user processes
(-u) 4091 virtual memory          (kbytes, -v)
unlimited file locks                    (-x) unlimited
```

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Hamzih Abukhashab

Osama Alabaji

System Programmierung 3. Laboraufgabe

We 2018/19

1.Systemaufrufe

Systemaufruf	Funktion	Parameter
kill	Beendet einen Prozess oder sendet Signal	ProzessID(s), optional Signal
signal	Sendet ein Signal	Signal und Handler
sigaction	Verändert die Reaktion auf ein Signal	Neues und altes Signal sowie Handler
alarm	Sendet SIGALARM an den Prozess	Anzahl der Sekunden bis Alarm
sigpause	Wartet auf ein Signal	Signalmaske
setjmp	Sichert Stack für nicht lokales Goto	Pointer zu Buffer
longjmp	Nicht lokaler Jump zu einem gesicherten Kontext	Pointer zu Buffer
sigsetjmp	Sichert Stack für nicht lokales Goto mit Signalmaske	Pointer zu Buffer und Signalmaske
siglongjmp	Nicht lokaler Jump zu einem gesicherten Kontext	Pointer zu Buffer und Signalmaske
system	Führt Shell-Kommando aus	Auszuführendes Shell-Kommando

2. Siehe Code Dateien

```
MacBook-Air:L3 gio$ ./com

Befehl eingeben:
Loop gestartet.
Bitte im Hauptprogramm eine Aktion auslösen! (20 s)
pstat
  UID   PID  PPID        F CPU PRI NI       SZ    RSS WCHAN     S             ADDR TTY           TIME CMD   STIME
  501 23394  1021     4006   0  31  0  4276944    616 -       S+                  0 ttys001    0:00.01 ./com  1:49PM

Befehl eingeben:
pong
Pausiert.
SIGUSR1 empfangen.
PID = 23395, PPID = 23394
Antwort des Child-Prozesses
Befehl eingeben:
whoare
Eltern PID = 23394
Kind PID = 23395
Befehl eingeben:
send

Befehl eingeben:
send sig

Befehl eingeben:
Befehl nicht erkannt.
Befehl eingeben:
Signal 14 nicht erkanntSignal 14 nicht erkanntBitte im Hauptprogramm eine Aktion auslösen! (20 s)
pong
Pausiert.
SIGUSR1 empfangen.
PID = 23395, PPID = 23394
Antwort des Child-Prozesses
Befehl eingeben:
quit
Auf Befehl terminiert.
```

4 Labor

4.1 Vergleichen von Dateien

Schreiben Sie ein Shell-Skript zum zeilenweisen Vergleich von C-Quellcode mit folgenden Eigenschaften:

- Das Skript soll wie ein Kommando arbeiten, Parameter sind die zwei Namen der zu vergleichenden Programmdateien: `compfile name1 name2`.
- Sollten die ersten 1-100 Zeilen Kommentarzeilen sein, so gebe man diese für beide zu vergleichende Programmdateien aus (dort stehen meistens Infos über die jeweiligen Programme!). Alle anderen Kommentarzeilen sollen ignoriert werden.
- Sollten sich 2 Programmzeilen NUR durch das Vorhandensein von Tabulatoren oder Leerzeichen unterscheiden, so ignoriere man die Unterschiede.
- Sollten in einer der Programmdateien Leerzeilen vorhanden sein, sollen diese ebenfalls ignoriert werden.
- Sollten in der einen oder anderen Programmdatei Quelltextzeilen lediglich hinzugefügt worden sein (also eingeschoben), sollte dies ebenfalls erkannt werden. Achten Sie darauf, daß nicht der nachfolgende Quelltext als unterschiedlich gekennzeichnet wird.
- Die Ausgabe sollte wahlweise auf den Bildschirm (fensterweise) oder in eine Datei ausgegeben werden können (> dateiname).

Hinweis: Nutzen Sie alle zur Verfügung stehenden UNIX-Standardwerkzeuge, die auf der Basis regulärer Ausdrücke arbeiten.

Shared Memory / Semaphore

Einführung in die parallele Programmierung
mit Shared Memory und Semaphore unter Unix/Linux

Lehrfach

Verteilte Applikationen/Client-Server

1 Grundlagen

1.1 Einleitung

Eine der einfachsten Möglichkeiten zur Programmierung verteilter Anwendungen ist die Erzeugung von mehreren Prozessen. Sind die Aufgaben der Anwendung parallelisierbar bzw. verteilbar, können mehrere Prozesse erzeugt und auf den Prozessoren parallel ausgeführt werden. Die Kommunikation und Synchronisation der Prozesse untereinander können dabei über verschiedene Mechanismen erfolgen. Die heutigen Betriebssysteme enthalten mehrere verschiedene Mechanismen zur Kommunikation und Synchronisation von verteilten Prozessen.

In diesem Versuch soll der Umgang mit den IPC-Mechanismen Shared Memory und Semaphore vorgestellt werden. Diese wurden zusammen mit den Message Queues in der System V-Linie von UNIX eingeführt.

1.2 Shared Memory

Eine Möglichkeit der Kommunikation zwischen verteilten Prozessen ist die Nutzung eines gemeinsamen Speicherbereichs, auf den die Prozesse Lese- und Schreibzugriff haben. Dies ist die schnellste Form der Interprozesskommunikation (IPC). Durch die Nutzung eines gemeinsamen Speicherbereichs werden Kopiervorgänge zwischen verschiedenen Puffern vermieden, wie sie bei anderen IPC-Mechanismen wie Pipe, FIFO und Message Queues vorkommen. Da diese Kopiervorgänge zwischen Kernel- und Userspace stattfinden (**intercontext copy**), sind sie im Vergleich zu Shared Memory zeitaufwendig. Neben dem Vorteil der Schnelligkeit besitzt die Kommunikation über einen gemeinsamen Speicherbereich jedoch folgenden Nachteil: Der Zugriff auf den Shared-Memory-Bereich ist ungeschützt (nicht synchronisiert) und deshalb vom Programmierer zu synchronisieren.

1.3 Prozesssynchronisation

Bei der Kommunikation mehrerer Prozesse über gemeinsame Daten, können zeitkritische Abläufe entstehen. Diese entstehen, wenn die Endergebnisse von der zeitlichen Reihenfolge der Lese- und Schreibzugriffe abhängen. Um diese zeitkritischen Abläufe zu vermeiden, muss der Zugriff auf die gemeinsamen Daten so koordiniert werden, dass zu einem bestimmten Zeitpunkt **nur ein** Prozess Zugriff auf die Daten hat (exklusiver Zugriff).

Der Teil des Programms der auf die gemeinsamen Daten zugreift, wird als kritischer Bereich bezeichnet. Um zeitkritische Abläufe zu verhindern, müssen die Prozess-Abläufe so synchronisiert werden, dass zu einem bestimmten Zeitpunkt nur ein Prozess in seinem kritischen Bereich arbeiten kann. Dies wird durch wechselseitigen Ausschluss der Prozesse (mutual exclusion) während des Zugriffs bewerkstelligt (Zugriffssynchronisation).

Zur Synchronisation von Prozessen können unter Unix/Linux verschiedene Mechanismen zum Einsatz kommen:

- **Signale**

Bei der Verwendung von Signale besteht die Vorgehensweise dabei im Senden von Signalen zur Zugriffsregelung zwischen den Prozessen. Die Synchronisation mit Hilfe von Signalen ist eher unüblich, da unter gewissen Bedingungen eine korrekte Arbeitsweise nicht garantiert werden kann.

- **Dateien**

Die Synchronisation mit Hilfe von Dateien ist eine einfache Form der Synchronisation und kann leicht umgesetzt werden. Dabei wird eine Sperrdatei (Lockfile) im Dateisystem angelegt, um einen wechselseitigen Ausschluss zu ermöglichen. Ein Prozess legt vor dem Eintritt in den kritischen Bereich eine Sperrdatei an. Ist die Sperrdatei vorhanden, müssen die anderen Prozesse warten. Verlässt der Prozess den kritischen Bereich, wird die Sperrdatei vom Prozess gelöscht. Für diese Art der Synchronisation werden nur Funktionen für den Dateizugriff benötigt. Damit ist diese Synchronisationsart mit allen Betriebssystemen die ein Dateisystem zur Verfügung stellen möglich. Bei dieser Methode stellt das Betriebssystem nur die Grundfunktion für den Zugriff auf die Dateien zur Verfügung.

- **Semaphore**

Die Semaphore wurden 1965 vom Holländer E. W. Dijkstra eingeführt. Diese sind nichtnegative Integervariablen auf die zwei **atomare** Operationen ausgeführt werden. Sie werden als P()- und V()-Operation bezeichnet. Die P()-Operation überprüft ob der Wert des Semaphors größer Null ist und erniedrigt diesen, wenn dies der Fall ist. Ist der Wert Null, wird der aufrufende Prozess blockiert. Die V()-Operation erhöht den Wert des Semaphors und weckt gegebenenfalls ein Prozess auf, der wegen einer P()-Operation auf diesem Semaphor blockiert wurde. Nach einer V()-Operation, bei der ein blockierter Prozess geweckt wurde, bleibt der Wert des Semaphors Null. Die Auswahl des zu weckenden Prozesses kann zufällig, prioritätsabhängig oder nach dem First-In-First-Out-Prinzip geschehen. Bei dem zufälligen Prinzip kann es zu einer unfairen Auswahl der Prozesse kommen. In diesem Fall kann es zum Verhungern eines Prozesses kommen.

1.4 Programmierung

1.4.1 Allgemeines zu Shared Memory und Semaphore

Obwohl sich diese beiden IPC-Mechanismen relativ stark unterscheiden, ist die Verwaltung derer IPC-Objekte im Kern recht einheitlich. Es werden jeweils zwei Parameter (ein sog. *Schlüssel* (key) und eine *Kennung* (id)) benutzt, die bei beiden IPC-Mechanismen benutzt werden.

Der *Schlüssel* ist vom Datentyp **key_t**, der in den meisten Systemen als **long** in der Datei `<sys/types.h>` definiert ist. Der Schlüssel wird vom Kern beim Erstellen eines neuen IPC-Objektes mit einer Kennung verbunden. Dadurch wird auch nichtverwandten Prozessen die Möglichkeit gegeben, mit dem Schlüssel auf das IPC-Objekt zuzugreifen. Da die Kennung erst beim Erstellen des IPC-Objektes generiert wird, ist sie nur verwandten Prozessen bekannt. Anderen Prozessen muss diese erst durch andere Kommunikationsmechanismen mitgeteilt werden. Es kann auch IPC-Objekte geben, die keinen Schlüssel zugeordnet bekommen. Solche werden als privat bezeichnet. Um ein privates IPC-Objekt einzurichten, muss anstelle des Schlüssels die Konstante `IPC_PRIVATE` angegeben werden.

Die *Kennung* wird vom Kern beim Erstellen eines neuen IPC-Objektes vergeben und ist eine nichtnegative Zahl. Die Kennungen werden bei jedem Einrichten hochgezählt, wobei frei gewordene niedrige Kennungen nicht, wie das bei den Filedeskriptoren der Fall ist, gleich wieder benutzt werden. Erst wenn der maximal mögliche Wert erreicht ist, wird wieder bei Null begonnen.

Zu jedem IPC-Objekt existiert eine `ipc_perm` Struktur, die in `<sys/ipc.h>` definiert ist. Diese Struktur enthält Informationen über Eigentumsverhältnisse und Zugriffsrechte.

```
struct ipc_perm {
   key_t key;     /*Schlüssel*/
   uid_t uid;     /*effektive User-ID des Eigentümers*/
   gid_t gid;     /*effektive Group-ID des Eigentümers*/
   uid_t cuid;    /*effektive User-ID des Einrichters */
   gid_t cgid;    /*effektive Group-ID des Einrichters*/
   ushort mode;   /*Zugriffsrechte                    */
   ulong pad1;
   ulong seq;     /*Sequenznummer                    */
   ulong pad2;
}
```

Beim Einrichten des IPC-Objekts, wird die Struktur bis auf `seq` initialisiert und die User-ID sowie die Group-ID des Eigentümers sind mit denen des Einrichters identisch. Später können IDs und Zugriffsrechte verändert werden. Die Zugriffsrechte auf IPC-Objekte sind dem file access mode für Dateien ähnlich, nur dass es bei IPC-Objekten keine `execute`-Berechtigung gibt.

1.4.2 Shared Memory

Neben der allgemeinen `ipc_perm`-Struktur wird für das IPC-Objekt noch eine spezielle Struktur im Kern angelegt. Für das Shared Memory ist es die `shmid_ds`-Struktur.

```
struct shmid_ds {
   struct  ipc_perm shm_perm;/*Struktur für die Zugriffsrechte etc.*/
   size_t  shm_segsz;     /*Größe des Speicherbereichs*/
   time_t  shm_atime;     /*letzter attach-Zeitpunkt  */
   time_t  shm_dtime;     /*letzter detach-Zeitpunkt  */
   time_t  shm_ctime;     /*Zeit der letzten Änderung */
   pid_t   shm_cipd;      /*PID des Einrichters       */
   pid_t   shm_lpid;      /*PID des letzten shmop-   Aufrufers*/
   ulong   shm_nattch;    /*wie viele Prozesse sind angebunden*/
}
```

Die `mode`-Komponente in der `ipc_perm`-Struktur wird beim Shared Memory um zwei zusätzliche Flags erweitert. Hinzu kommen `SHM_LOCKED` und `SHM_DEST`. Die Konstante `SHM_LOCKED` verhindert das Auslagern von Speicherseiten des Shared Memory und `SHM_DEST` legt fest, dass das Shared Memory nach der letzten detach-Operation automatisch freigegeben wird.

Shared Memory erstellen bzw. öffnen

Als erstes wird ein Shared Memory Objekt angelegt bzw. geöffnet. Für die Erstellung wird die Funktion `shmget` benutzt. Folgende Aufrufkonvention wird benutzt:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

int shmget (key_t key, size_t shmsize, int shmflg)
```

Auf den Parameter Schlüssel wurde im vorherigen Abschnitt näher eingegangen. Die Größe gibt die minimale Größe des anzufordernden Speicherbereichs an. Wird ein bereits existierender Bereich geöffnet, kann für die Größe Null angegeben werden. Das Argument Flag ist eine Kombination aus verschiedenen Konstanten, die die Zugriffsrechte sowie das Verhalten bei der Erstellung des Speicherbereichs beeinflussen. Tabelle 1 zeigt die möglichen Flag-Werte für die Zugriffsrechte.

Symbolik	Wert	Beschreibung
SHM_R	0400	lesbar für Eigentümer
SHM_W	0200	schreibend für Eigentümer
SHM_R 3	0040	lesbar für Gruppe
SHM_W 3	0020	schreibend für Gruppe
SHM_R 6	0004	lesbar für alle
SHM_W 6	0002	schreibend für alle

Tabelle 1: Flags für shmget

Die Flagkonstante `IPC_CREAT` und `IPC_EXCL` spielen bei der Erstellung eines neuen Speicherbereichs eine große Rolle. Mit `IPC_CREAT` wird sichergestellt, dass ein neuer Bereich erstellt und nicht ein bereits vorhandener geöffnet wird. Existiert schon ein Bereich mit gleichem Schlüssel, wird ein Fehler zurückgegeben und `errno` auf `EEXIST` gesetzt.

Der Rückgabewert ist bei erfolgreicher Ausführung die Kennung des Speicherbereichs und im Fehlerfall -1. Mit der Kennung kann der Speicherbereich angebunden werden. Tritt ein Fehler auf, kann `errno` auf folgende Werte gesetzt sein:

EINVAL: Die angegebene Größe des Segments ist kleiner als die minimal bzw. größer als die maximal erlaubte Segment-Größe oder ein neues Segment wurde mit einem bereits existierenden Schlüssel erstellt, aber die angegebene Größe ist größer als die des Segments.

EEXIST: Wenn das Segment schon existiert und `IPC_CREAT – IPC_EXCL` gesetzt war.

EIDRM: Segment ist zum Löschen markiert oder ist bereits gelöscht.

ENOSPC: Alle möglichen Shared Memory Kennungen sind vergeben oder es kann kein Segment der gewünschten Größe allokiert werden, ohne die systemweiten Limits zu überschreiten.

ENOENT: Es existiert kein Segment mit diesem Schlüssel und `IPC_CREAT` war nicht gesetzt.

EACCES: Der Prozess hat keine Zugriffsrechte auf das Shared Memory.

ENOMEM: Es konnte kein Speicher für die Verwaltung des Shared Memory allokiert werden.

Speicherbereich anbinden

Durch shmget wird ein Speicherbereich erst angelegt bzw. geöffnet. Um diesem benutzen zu können, muss eine Speicheradresse bekannt sein. Um diese zu bekommen, muss der Speicherbereich angebunden werden. Hierzu wird folgende Funktion benutzt:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
```

`void *shmat (int shmid, void *shmaddr, int shmflg)`

Der erste Parameter ist die Kennung, die von `shmget` zurückgegeben wurde. Für den zweiten Parameter ergeben sich folgende Möglichkeiten:

- `shmaddr==NULL` Der Kern legt automatisch die Adresse fest
- `shmaddr! =NULL`
  - `SHM_RND`-Flag ist gesetzt. Der Bereich wird an der festgelegten und um den in der `SHMLBA`-Konstanten festgelegten Wert abgerundete Adresse angebunden.
  - `SHM_RND`-Flag ist nicht gesetzt. Der Bereich wird an der übergebenen Adresse angebunden.

Die `SHMLBA`-Konstante steht für low boundary address multiple und hat als Wert eine Zweierpotenz. Die Shared Memory Segmente müssen an solchen Speichergrenzen beginnen. In der momentanen Implementierung, ist die `SHMLBA`-Konstante gleich der Speicherseitengröße des Systems.

Als dritter Parameter können Flags angegeben werden, welche die Anbindung beeinflussen. Eine Übersicht dieser Flags zeigt Tabelle 2.

CMD	Wert	Beschreibung
SHM_RDONLY	010000	nur lesend anbinden
SHM_RND	020000	Adresse auf SHMLBA runden

Tabelle 2: Flags für shmat

Von praktischer Bedeutung und aus Portabilitätsgründen, ist als Adresse `NULL` zu übergeben. Als Flagkonstante gibt es neben `SHM_RND` auch noch `SHM_RDONLY`, die den Speicherbereich nur für den lesenden Zugriff anbindet.

Bei erfolgreicher Ausführung wird die Adresse des angebundenen Speicherbereichs zurückgegeben. Im Fehlerfall wird eine -1 zurückgegeben und `errno` kann auf folgende Werte gesetzt sein:

EINVAL: Ungültige Shared Memory Kennung, ungültiger Adresswert oder die Adresse ist nicht nach Speicherseitengrenzen ausgerichtet und das Kommando `SHM_RND` ist nicht gesetzt.

EACCES: Der Prozess hat keine Zugriffsrechte.

ENOMEM: Es konnte kein Speicher für den Deskriptor und den Speicherseitentabellen allokiert werden.

Bei einem fork()-Systemaufruf erbt das Kind das angebundene Segment. Wird der Prozess mit einem exec()-Aufruf überlagert oder mit einem exit()-Aufruf beendet, wird das Segment automatisch gelöst.

Kommandofunktion

Um das Shared Memory zu löschen oder den Status abzufragen bzw. zu ändern, steht dem Programmierer die Funktion *shmctl* zur Verfügung.

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid_ds *puffer)
```

Das *cmd*-Argument spezifiziert das Kommando. In der Tabelle 3 sind alle möglichen Kommandos aufgeführt.

CMD	Wert	Beschreibung
IPC_RMID	0	Löschen
IPC_SET	1	Setzen der UID/GID und der Zugriffsrechte
IPC_STAT	2	Abfragen des Status
IPC_INFO	3	Info (nur unter GNU)
SHM_LOCK	11	Sperren
SHM_UNLOCK	12	Sperre aufheben
SHM_STAT	13	Status über das Segment
SHM_INFO	14	Informationen über das Segment

Tabelle 3: Kommandos für shmctl

IPC_RMID Löschen des Shared Memory. Sollten noch Prozesse diesem Speicherbereich nutzen, ist das Segment nach dem Aufruf nicht wirklich gelöscht. Das wirkliche Löschen des Speicherbereichs geschieht erst wenn der letzte Prozess, der den Bereich nutzt, beendet wird oder die Anbindung aufhebt. Die Information über die Anzahl der noch an den Speicherbereich angebundenen Prozesse entnimmt der Kern von der Komponente `shm_nattch` aus der `shmid_ds`-Struktur. Unabhängig davon, ob `shm_nattch` gleich Null ist, wird die Kennung des Bereiches sofort gelöscht, um neue Anbindungen zu verhindern.

IPC_SET Setzen der Zugriffsrechte und der Eigentümer UID/GID. Dies erfolgt durch die Übergabe einer `shmid_ds`-Struktur, wobei nur `uid`, `gid` und `mode` berücksichtigt werden. Das Setzen ist nur möglich, wenn die effektive UID des Prozesses gleich der `cuid`, `uid` oder die des Superusers ist.

IPC_STAT Die Statusinformationen werden an die Adresse *puffer* geschrieben.

IPC_INFO Auslesen von Informationen über das entsprechende Shared Memory. Die Informationen werden in der Struktur `shminfo` abgelegt, deren Adresse *puffer* entspricht. Die Struktur hat folgenden Aufbau: 1cm

```
struct shminfo {
   int  shmmax; /*max. Anzahl eines Segments in Bytes*/
   int  shmmin; /*max. Groesse des Segments          */
   int  shmmni; /*max. Anzahl von Shared Memories im System*/
   int  shmseg; /*max. Anzahl von Segmenten je Prozess*/
   int  shmall; /*max. Anzahl von Segmenten           */
}
```

SHM_LOCK Durch dieses Kommando wird der Speicherbereich gesperrt. Es kann nur vom Superuser ausgeführt werden.

SHM_UNLOCK Mit diesem Kommando wird ein mit `SHM_LOCK` gesperrtes Shared Memory wieder freigegeben. Auch dies kann nur vom Superuser ausgeführt werden.

SHM_STAT Gibt den Status über das Shared Memory Segment aus.

SHM_INFO Gibt Informationen über das Shared Memory Segment aus.

Im Fehlerfall wird eine -1 zurückgegeben und `errno` kann folgende Werte annehmen:

EINVAL: Die Parameter *shmid* und *cmd* haben keine gültigen Werte.

EFAULT: Als Kommando ist `IPC_SET` oder `IPC_STAT` gesetzt, aber auf die Adresse, die mit *puffer* übergeben wurde, ist kein Zugriff möglich.

EIDRM: Wenn `shmid` als gelöscht markiert ist.

EPERM: Tritt bei den Kommandos `IPC_SET` oder `IPC_STAT` auf, wenn die effektive UID des Prozesses nicht des Einrichters, Besitzers oder eines Superusers entspricht.

EACCES: Wenn `IPC_STAT` gesetzt wurde, aber kein Lesezugriff für das Shared Memory erlaubt ist.

EOVERFLOW: Tritt bei dem Kommando `IPC_STAT` auf, wenn der Wert der UID oder GID zu groß ist, um ihn in der bei `puffer` übergebenen Struktur abzuspeichern.

Lösen des Shared Memory

Um die Anbindung eines Speicherbereichs aufzuheben, wird die Funktion `shmdt` benutzt. Die Funktion wird wie folgt aufgerufen:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

int shmdt (void *adresse)
```

Die von `shmat` zurückgegebene Adresse muss übergeben werden. Es ist zu beachten, dass das Shared Menory dadurch nicht aus dem System entfernt wird, außer es wurde schon vorher mit `shmctl` gelöscht. Bei fehlerhafter Ausführung ist der Rückgabewert gleich -1 und `errno` wird auf folgenden Wert gesetzt:

EINVAL: An der angegebenen Adresse ist kein Shared Memory angebunden.

1.4.3 Semaphore

Wie auch beim Shared Memory gibt es bei der Verwaltung der Semaphore im System neben der allgemeinen `ipc_perm`-Struktur noch eine spezielle für die Semaphore. Die Semaphore werden mit folgender Struktur verwaltet:

```
struct semid_ds {
   struct  ipc_perm sem_perm;/*Struktur für die Zugriffsrechte*/
   struct  sem *sem_base;    /*Adresse des 1. Semaphor*/
   ushort  sem_nsems;        /*Anzahl d. Semaphore in d.  Menge*/
   time_t  sem_otime;        /*letzter semop-Aufruf   */
   time_t  shm_ctime;        /*letzte Änderung        */
}
```

Semaphormenge erstellen bzw. öffnen

Semaphore werden in Form von Mengen bearbeitet. Um eine Semaphormenge zu erstellen oder zu öffnen, steht die Funktion `semget` zur Verfügung:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

int semget (key_t schluessel, int semnum, int semflg)
```

Der Parameter Schlüssel hat die gleiche Funktion wie bei Shared Memory. Jeder Prozess, der den Schlüssel kennt, kann die Semaphormenge öffnen. Der Parameter `nsems` gibt die Anzahl der Semaphore in der Menge an. Wird eine schon existierende Semaphormenge geöffnet, kann für `nsems` Null angegeben werden. Das Flag-Argument hat die gleichen Werte wie bei Shared Memory, aber mit anderen Konstantennamen. Tabelle 4 zeigt eine Übersicht der Konstanten. Die Konstanten `IPC_CREAT` und `IPC_EXCL` erfüllen die gleiche Funktion wie bei Shared Memory.

Symbolik	Wert	Beschreibung
SEM_R	0400	lesbar für Eigentümer
SEM_A	0200	änderbar für Eigentümer
SEM_R>>3	0040	lesbar für Gruppe
SEM_A>>3	0020	änderbar für Gruppe
SEM_R>>6	0004	lesbar für alle
SEM_A>>6	0002	änderbar für alle

Tabelle 4: Flags für semget

Der Rückgabewert der Funktion ist im Fehlerfall -1 und bei erfolgreicher Ausführung die ID der Semaphormenge. Im Fehlerfall kann `errno` folgende Werte annehmen:

EINVAL: Die Anzahl der gewünschten Semaphore (`semnum`) ist kleiner Null, größer als das Systemlimit oder die Anzahl stimmt nicht mit der Semaphormenge überein, wenn schon eine Menge mit diesem Schlüssel existiert.

EEXIST: Es existiert schon eine Semaphormenge mit dem Schlüssel und `IPC_CREAT` und `IPC_EXCL` waren gesetzt.

ENOENT: Es existiert keine Semaphormenge mit dem Schlüssel und `IPC_CREAT` war nicht gesetzt.

ENOMEM: Das System hat nicht genug Speicher für die neuen Strukturen.

EACCES: Es existiert eine Semaphormenge mit dem Schlüssel, aber der Prozess hat keine Zugriffsberechtigung.

ENOSPC: Die Systemlimits für die max. Anzahl von Semaphormengen oder die max. Anzahl von Semaphore wurden überschritten.

Semaphoroperationen

Um Semaphoroperation auszuführen, wird folgende Funktion benutzt.

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

int semop (int semid, struct sembuf semoparray[], size_t nsops)
```

Der erste Parameter ist die ID der Semaphormenge, die von der `semget`-Funktion zurückgegeben wurde. Das zweite Argument ist ein Feld von `sembuf`-Strukturen. Diese Struktur gibt die auszuführende Aktion an. Die `sembuf`-Struktur hat folgenden Aufbau:

```
struct sembuf {
   ushort  sem_num; /*Nr. des Semaphors in der Menge*/
   short   sem_op;  /*Operation                     */
   short   sem_flg; /*Flag                        */
}
```

FLAG	Wert	Beschreibung
SEM_UNDO	0x1000	Operation rückgängig bei exit
IPC_NOWAIT	0x4000	nicht blockierend

Tabelle 5: Flags für Semaphoroperation

Bei einer fehlerhaften Ausführung ist der Rückgabewert -1 und `errno` kann folgende Werte annehmen:

EINVAL: Die Semaphormenge existiert nicht, `semid` ist kleiner Null oder `nsops` hat einem negativen Wert.

E2BIG: Die Anzahl der auszuführenden Operationen (`nsops`) ist größer als die max. Anzahl pro Systemaufruf.

EAGAIN: Die Operation wurde nicht durchgeführt, weil der Semaphorwert nicht Null und das Flag `IPC_NOWAIT` gesetzt ist.

ENOMEM: Das `SEM_UNDO` Flag war bei einigen Operationen gesetzt, aber das System konnte keinem Speicher für die `undo`-Strukturen allokieren.

EACCES: Der Prozess hat keine Zugriffsrechte für die Semaphormenge.

EFAULT: Es konnte nicht auf die Adresse, die durch `semoparray` übergeben wurde, zugegriffen werden.

EFBIG: Für einige Operationen ist der Wert `semnum` kleiner Null oder größer bzw. gleich der Anzahl der Semaphore in der Menge.

EIDRM: Die Semaphormenge ist gelöscht.

EINTR: Tritt auf, wenn das Flag `IPC_NOWAIT` nicht gesetzt ist und während des Wartens auf den Semaphorwert Null der Prozess ein Signal empfängt, welches er abfängt.

ERANGE: Für einige Operationen ist die Summe von Semaphorwert und Semaphoroperation größer als der systemabhängige Maximalwert.

Bei einer Überlagerung des Prozesses wird die `sem_undo`-Struktur nur durch einem `execve`-Aufruf weitergegeben. Es ist zu beachten, dass die `sem_undo`-Struktur bei einem fork()-Aufruf nicht an das Kind vererbt wird.

Kommandofunktionen

Wie bei Shared Memory gibt es auch für die Semaphormengen eine Kommandofunktion.

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>

int semctl (int semid, int semnum, int cmd)
```

Das erste Argument ist die ID der Semaphormenge und das zweite gibt den Semaphor in der Menge an, auf die ein Kommando ausgeführt werden soll. Der dritte Parameter gibt das Kommando an, was in Form einer Zahl oder einer definierten Konstante angegeben werden kann. Tabelle 6 zeigt eine Übersicht zur Verfügung stehender Kommandos.

CMD	Wert	Beschreibung
IPC_RMID	0	Löschen
IPC_SET	1	Setzen der UID/GID und der Zugriffsrechte
IPC_STAT	2	Abfragen des Status
IPC_INFO	3	Info (nur unter GNU)
GETPID	11	PID vom letzten Prozess der semop
GETVAL	12	Semaphorwert auslesen
GETALL	13	alle Semaphorwerte auslesen
GETNCNT	14	Anzahl von Prozessen die auf Semaphorwert größer 0 warten
GETZCNT	15	Anzahl von Prozessen die auf Semaphorwert gleich 0 warten
SETVAL	16	Semaphorwert setzen
SETALL	17	alle Semaphorenwerte setzen
SEM_STAT	18	Status der Semaphormenge
SEM_INFO	19	Information über die Semaphormenge

Tabelle 6: Semaphor-Kommandos

IPC_RMID, **IPC_SET**, **IPC_STAT**, **IPC_INFO** Diese Gruppe von Kommandos erfüllen die gleichen Aufgaben, wie bei Shared Memory bereits beschrieben. Nur die Struktur für die Informationen von `IPC_INFO` hat einen anderen Aufbau.

```
struct seminfo {
   int  semmap;/*max. Einträge in der Semaphormenge   */
   int  semmni;/*max. Anzahl von Semaphormengen       */
   int  semmns;/*max. Anzahl von sem_undo-Strukturen im System*/
   int  semmsl;/*max. Anzahl von Semaphore pro Kennung*/
   int  semopm;/*max. Anzahl von Operationen je semop-Aufruf  */
   int  semume; /*max. Anzahl von sem_undo-Einträge je Prozess */
   int  semusz; /*Größe der sem_undo-Struktur         */
   int  semvmx; /*max. Wert eines Semaphors          */
   int  semaem; /*max. Wert für eine sem_undo-Struktur*/
}
```

Die Einträge `semmap`, `semume`, `semusz` und `semaem` haben unter Linux keine Bedeutung, da sie ignoriert werden.

GETPID Gibt die PID des letzten Prozess zurück, der auf diesem Semaphor eine semop-Aufruf ausführte.

GETVAL Ermittelt den Wert eines einzelnen Semaphors.

GETALL Dieses Kommando ist gleich dem Vorangegangenen, mit dem Unterschied, dass alle vorhandenen Semphoren in der Menge abgefragt werden.

GETNCNT Gibt die Anzahl der Prozesse zurück, die auf eine Erhöhung des Semaphorwertes warten.

GETZCNT Gibt die Anzahl der Prozesse zurück, die darauf warten, dass der Semaphorwert gleich Null wird.

SETVAL Mit diesem Befehl wird ein bestimmter Semaphor in der Menge mit einem bestimmten Wert gesetzt.

SETALL Dieses Kommando erlaubt es, alle Semaphore in der Menge mit einem bestimmten Wert zu setzen.

SEM_STAT Gibt den Status der Semaphormenge zurück.

SEM_INFO Gibt Informationen über die Semaphormenge aus.

Der Rückgabewert der Systemfunktion ist bei fehlerfreier Ausführung ein nichtnegativer ganzzahliger Wert und im Fehlerfall -1. Je nach Kommando hat der Rückgabewert eine andere Bedeutung. In Tabelle 7 sind diese aufgeführt. Kommandos die nicht aufgeführt sind, geben eine Null als Wert zurück.

Kommando	Rückgabewert
GETNCNT	Anzahl der auf eine Erhöhung des Semaphorwertes wartenender Prozesse
GETPID	Prozess-PID des letzten semop ausführenden Prozesses
GETVAL	Semaphorwert
GETZCNT	Anzahl der Prozesse die auf den Semaphorwert 0 warten

Tabelle 7: Rückgabewerte der Semaphorkommandos

Folgende Fehler können bei dieser Funktion auftreten:

EINVAL: Ungültiger Wert für `cmd` oder `semid`.

EFAULT: Auf die für die Datenstruktur übergebene Adresse ist kein Zugriff möglich.

EIDRM: Die Semaphormenge ist gelöscht.

EPERM: Tritt bei den Kommandos `IPC_SET` oder `IPC_RMID` auf, wenn der Prozess keine Rechte zum Ausführen dieser Kommandos hat.

EACCES: Der Prozess hat keine Zugriffsrechte.

ERANGE: Die Kommandos `SETALL` oder `SETVAL` sind gesetzt und der zu setzende Wert für den Semaphor ist kleiner Null oder größer als der systemabhängige Maximalwert für ein Semaphor.

2 Kontroll- und Vorbereitungsfragen

1. Was bedeuten die Begriffe „Verklemmen“ und „Verhungern“ im Zusammenhang mit parallelen Prozessen?
2. Was versteht man unter einer Wettkampf-Bedingung (race condition)?
3. Erklären Sie den Unterschied zwischen blockierendem und aktivem Warten. Bei welchen der vorgestellten Synchronisationsformen wird welche Methode benutzt?
4. Was ist ein Semaphor?
5. Bei der Synchronisation mittels Semaphoren können bei unerwarteter (externer) Beendigung eines Prozesses Verklemmungen (deadlocks) entstehen. Wie können diese unter UNIX/LINUX verhindert werden.
6. Wie werden Shared Memory- und Semaphor-Objekte gelöscht?
7. Wie kann unter UNIX/LINUX eine Prozess-Blockierung bei einer Semaphor-Operation verhindert werden?
8. Mit welchem Shell-Kommando können Informationen über die im System vorhandenen Shared Memory- und Semaphor-Objekte, ermittelt werden?

3 Durchführung

1. Schreiben Sie zwei Programme mit den Namen **`showmem`** und **`fillmem`**. Das Programm **`showmem`** soll ein Shared Memory anlegen und es mit einem beliebigen ASCII-Zeichen initialisieren. Danach soll es **N** neue Prozesse erzeugen, die dann mit dem Programm **`fillmem`** überlagert werden. Nach der Prozesserzeugung soll **`showmem`** den Inhalt des Shared Memory auf Tastendruck ausgeben. Die Anzahl der zu erzeugenden Prozesse **N** soll als Argument übergeben werden. Die Größe des Shared Memory sollte sich nach der Größe/Auflösung des Text-Bildschirms (-Fensters) richten, wobei die letzte Zeile zur besseren Übersicht weggelassen werden soll.
2. Die erzeugten **`fillmem`**-Prozesse sollen das Shared Memory mit jeweils ***verschiedenen*** ASCII-Zeichen füllen. Um die unsynchronisierten Zugriffe besser zu verdeutlichen, soll der Prozess nach dem er ein Zeichen in den Speicherbereich geschrieben hat, eine zufällige Zeit aktiv warten.
3. Implementieren Sie in den Programmen **`showmem`** und **`fillmem`** eine Zugriffssynchronisation mit Hilfe einer Sperrdatei (Lock-File).
4. Implementieren Sie in den Programmen **`showmem`** und **`fillmem`** eine Zugriffssynchronisation mithilfe der Unix-Semaphor-Operationen.

Hinweise zur Programmierung

1. Für die einzelnen Aufgaben sind separate Verzeichnisse und Makefiles zu erstellen.
2. In den ersten Zeilen des Quelltextes ist ein Kommentartext mit Name, Matr.-Nr., Programmname, Lehrveranstaltung, Versuch und Versuchsaufgabe einzufügen.
3. Bei Eingabe falscher oder ungültiger Parameter, ist dem Benutzer eine usage-Meldung auszugeben, die die Aufrufkonventionen beschreibt.
4. Die usage-Meldung sowie auftretende Fehler sind über die Standardfehlerausgabe auszugeben.
5. Die Quelltexte müssen mit der Compileroption `-Wall -ansi -pedantic` ohne Fehler und Warnungen kompilierbar sein.

Literatur

[1] Helmut Herold. *Linux/Unix-Systemprogrammierung*. Addison-Wesley, 2004.

[2] Andrew S. Tanenbaum, Marten van Steen. *Verteilte Systeme*. Prentice Hall, 2003.

Anlage UNIX Kommandos

Befehl	Option	Bedeutung
pwd		Zeigt das aktuelle Verzeichnis an (also in welchem man sich gerade befindet)
ls		Listet Verzeihnisinhalt auf
ls	-l	Listet Verzeichnisinhalt auf und zeigt Dateiattribute Tabelarisch mit an
ls	-a	Zeigt Verzeichnisinhalt vollständig (also mit versteckten Dateien) an
cd [pfad]		Wechselt Verzeichnis nach "pfad"
cd ..		Wechselt in übergeordnetes Verzeihnis
mkdir [name]		Erstellt Verzeichnis mit dem Namen "name"
rm		Lösch Verzeichnis (funktioniert so nur wenn Verzeichnis leer ist)
rm	-r	Löscht Verzeichnis rekursiv (also mit Inhalt)
rm	-f	Löscht Verzeichnis ohne überprüfen auf etwaige Systemdateien.
cp [quelle][ziel]		Kopiert "Quelle" nach "Ziel"
cp [quelle][ziel]	-r	Kopiert rekursiv (also Ordner)
mv [quelle][ziel]		Verschiebt quelle nach Ziel
mv [alter_name][neuer_name]		Benennt Datei oder Ordner um
ln [verknüpfungsziel][pfad+name der verknüpfung]	-s	Symbolischer Link = Verknüpfung welche auf "Verknüpfungsziel" verweist
locate [dateiname]		Sucht nach "dateiname"
grep [Begriff]		Filtert Ausgaben auf "Beriff"

***cat :**
Syntax: cat [optionen] datei [datei]
Beschreibung: Ausschreiben der angegebenen Dateien auf die Standardausgabedatei <stdout>, bei Dateiumlenkung können mehrere Dateien zu einer verknüpft werden.
Optionen:

-n Ausgabe mit Zeilennumerierung
-s keine Meldung bei nicht existierenden Dateien
-u ungepufferte Ausgabe
-v für nicht darstellbare Zeichen werden Ersatzdarstellungen ausgegeben

Beispiel(e):

-cat .cshrc die Datei .cshrc wird auf den Bildschirm ausgegeben, wenn sie sich im aktuellen Verzeichnis befindet
-cat>neue_datei es wird eine neue Datei mit dem Namen neue_datei im aktuellen Verzeichnis erzeugt, die von der Tastatur solange Zeichen aufnimmt,bis ein Dateiende (^Z) eingegeben wird

***tail:** Ausgabe der letzten Zeilen einer Datei.
***find:** Sucht nach Dateien mit vorgegebenen Charakteristika.
***mkdir:** Legt einen neuen Dateikatalog an.

***mkfs:** Legt eine neue initiale Dateistruktur auf einem Datenträger an.
***rm:** Löscht eine Datei (Referenz) aus dem Katalog.
***rmdir:** Löscht einen Dateikatalog.
***cp:** Kopiert Dateien.
***chgrp:** Ändert die Gruppennummer einer Datei.
***chmod:** Erlaubt die Zugriffsrechte (Mode) einer Datei zu ändern.
***chown:** Ändert den Besitzereintrag einer Datei.
***In:** Gibt einer Datei einen weiteren Namen (Namensreferenz).
***mv:** Ändert den Namen einer Datei.

***exit:** Beendet eine Kommandoprozedur oder die Shell.
***login:** Abmelden und erneut Anmelden.
***logout:** Meldet bei der csh einen Benutzer ab.
***newgrp:** Ändern der Gruppennummer.
***passwd:** Ändern oder Eintragen des Paßwortes.
***pwd:** Liefert den Namen des aktuellen Arbeitskatalogs.

***locate:** Sucht nach Kommandos zu bestimmten Stichworten.
***man:** Gibt Teile des UNIX-Manuals aus.
***whatis:** Sucht Bedeutung eines Kommandos.

***kill:** Abbrechen eines im Hintergrund laufenden Kommandos.
***killall:** Abbruch aller aktiven Prozesse eines Benutzers.

***pas:** Aufruf des PASCAL-Compilers .

***mknod:** Schafft einen neuen Geräteeintrag oder liegt eine FIFO-Datei an.
***ps -e :** alle aktive Processe mit PID.
***ps -l :** aktive Processe mit PID, PPID, UID, GID,...
***kill PID:** eine Process beenden

Locate: sucht nach Datei Name
find: Sucht nach Dateien mit vorgegebenen Charakteristika.
which:wo ein bestimmtes Programm in Linux installiert ist.
*ipcs: Zeigt an, wieviele Nachrichtenpuffer (Message Queues), Semaphorbereichen und Tabellen für Shared memory im System aktuell existieren.

echo | date >> filename : schreibt das datum und Uhrzeit am ende des Files.

Basissystemfunktionen

Systemaufrufe (system calls) für pipeling:

<fork> Dynamisches Erzeugen eines Prozesses, child gleicht parent bis auf PID (process identifikation)

<exec> [l,p,le] Ausführen eines Programms mit Übergabeparametern, exec(Name+Argumente); nur der Programmcode wird dabei überlagert

<wait> Warten auf das Ende eines Childprozesses p=wait(&status) p ist PID des beendeten Prozesses mit Status

<exit> Beenden eines Prozesses mit Status exit(Status); Status wird zur Fehlerbehandlung verwendet

<getpid> Ermittelt eigene PID innerhalb eines Prozesses, pid = getpid();

<getppid> Ermittelt PID des Parents

Systemcalls für file management:

<creat> Erstellt normale Datei, fd=creat(name,mode);
fd (file deskriptor), mode r/w/x

<open> Öffnet vorhandene Datei, fd=open(name,mode); gilt auch für Geräte

<read> Liest aus einer Geöffneten Datei, n=read(fd,buffer,nbytes); n gibt die Anzahl der gelesenen Bytes zurück

<write> Schreibt in eine geöffnete Datei, n=write(fd,buffer,nbytes)

<lseek> Setzt Positionszeiger in geöffneter Datei (Longseek), pos=lseek(fd,offset,whence), liefert Position zurück, Offset bei relativen Sprung, whence ist der Bezugspunkt (0 Anfang, 1 aktuell, 2 Ende)

<stat> Liefert Dateistaus s=stat(fd,statbuff)

<pipe> Legt pipe an (Kommunikationskanal zu anderen Prozessen), p=pipe(*fd); fd[0] lesen, fd[1] schreiben

<ioctl> Ändert Dateieigenschaften offener Dateien (ctl = control), s= ioctl(fd,request,argument); request ist ein Kommando an ein Gerät

Systemcalls für signal management:

<kill> Sendet Signal an einen anderen Prozess kill(PID,SIG);
<signal> Abfangen und reagieren auf ein Signal u= signal(SIG,func);
func ist Signal handler/-catcher
<alarm> Löst einen Alarm aus t= alarm(seconds);
<pause> Blockiert die Prozessausführung s= pause();

Signale: SIGALM(Wecksignal), SIGBUS(Bus error), SIGFPE(floating point error), SIGHUP(parent beendet), SIGILL(illegale Instruktion), SIGINT (Tastatur Interrupt), SIGKILL(Prozess schließen), SIGTERM("softes" killen), SIGQUIT(Prozessabbruch), SIGUSRX(Signal von User X)

System calls für directory management:

<link> Legt zweiten Verzeichniseintrag für eine Datei an, p=link(name1,name2);
<unlink> Löscht einen Verzeichniseintrag
<mount> Fügt ein Gerät hinzu s=mount(special device,name,rwflg)
<umount> Koppelt ein Gerät ab s= umount(spezial device);
<sync> Schreibt Cacheinhalt auf die Festplatte s=sync();
<chdir> Prozess wechselt in ein neues Verzeichnis
<chroot> Ändert das sichtbare Stammverzeichnis chroot(name);

System calls für security management:

<chmod> Legt einen neuen Zugriffsmodus fest n= chmod(name,mode);
<getuid> Stellt User ID fest () nach login, uid= getuid();
<getgid> Stellt Group ID fest
<setuid> Setzt eine User ID
<setgid> Setzt eine Group ID
<chown> Ändert die Benutzerrechte auf eine Datei s=chown(name,uid,gid);
(s=umask(complmask), setzt die default creation mask)

System calls für time management:

<time> Liefert die vergangene Zeit seit dem 1.1.1970 seconds= time();
<stime> Setzen der Systemzeit s=stime(tp);
<utime> Datei updaten p=utime(name,timepar);
<times> Gibt die vergeudete Zeit eines Prozesses zurück, s=times(buffer);

Systemprogrammierung

16.1.19

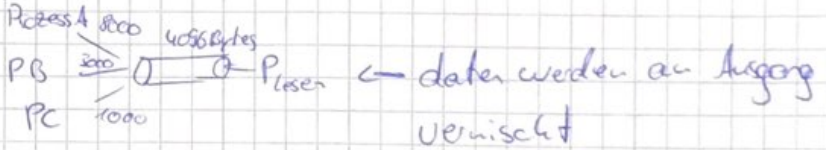
Signal Altklausuren Aufgaben schreiben

- Linux Schnittstelle
  - ↳ Datei kopiert
  - ↳ Datei neu anlegt
  - ↳ Verzeichnisse anlegt
  - ↳ Anzeigen von Prozessen (-aktiv)
  - ↳ 16 wichtigsten Kommandos
  - ↳ suchen nach Datei
  - ↳ compilieren
  - ↳ chmod

- Prozess Management
  - ↳ Was ist ein Prozess
  - ↳ Wie erzeugt man ein Prozess
  - ↳ Synchronisation
  - ↳ alle Vorbereitungsfragen von Laboren lernen
  - ↳ was ist fork
  - ↳ wait
  - ↳ waitpid
  - ↳ welche Parameter
  - ↳ exit Status überprüfen
  - ↳ wo finde ich der exit Status
  - ↳ exec (-was macht er
  - ↳ Parameter lernen
  - ↳ alle 6 ~~Parameter~~ exec lernen
  - ↳ wie übergibt man environment

- Weisen / Zombies
- Datensegment / Codesegment bei exec aufruf
- wird bei exec ein neuer Prozess erzeugt ? Nein

- Pipelining
  - ↳ Pipeaufruf
  - ↳ Fifos erstellen
  - ↳ Fifos / Pipes ← Unterschiede
  - ↳ Vor/Nachteile von Pipe/Fifos
  - ↳ wie wird der parent Benachrichtigt wenn Leiner liest und so
  - ↳ SIGPIPE wann wird das gesendet
  - ↳ ~~irgendwas von~~
    dup / dup2 ← wozu nutzt man das
  - ↳ Named Pipe nur ein Deskriptor
  - ↳ Bei Pipe haben wir 2 Deskriptoren
  - ↳ wie groß ist der maximale Puffer bei Pipes
    - ↳ bei größeren Portionen können Daten vermischt werden



← daten werden am Ausgang vermischt

  - ↳ Signal Handler
  - ↳ Signalmaske ← hängende Signale
  - ↳ Signalmaske bei exit / fork

- IPC
  - ↳ Semaphoren
  - ↳ Bus Master Instruktionen
  - ↳ Dijkstra Semaphoren

- Zeitscheibenverfahren
- Effizienz, Overhead berechnen
- kurze Antwortzeit → viel Overhead

Priorisierten Scheduling
- statisch / dynamische Prioritäten → Vor-/Nachteile
- können Prozesse verhungern
- Deadlocks

- Lösung von Peterson

- echte Parallelität
- Multicoresystem / Multisinglesystem
- Ist Busy Waiting schlecht? → wann ist Busy Waiting gut?

- zu 90% nichts rechnen

- Zombies / Orphans ansehen
- Vermeiden von Zombies → wie macht man das