**UNIX/LINUX SYSTEMS CALLS AND LIBRARY FUNCTIONS**

|  |  |
| --- | --- |
| perror – print a system error message | perror(message) 🡪 gibt „message“ (String) aus, gefolgt vom Grund für den Fehler. |
| wait – wait for process to change state  wait(&status) 🡪 speichern von Status-Information in der int-Variable worauf status zeigt. | Wird von Prozessen aufgerufen, um solange zu warten, bis Statusänderungen in Kindprozessen auftreten. 🡪 Informationen über den Kindprozess, dessen Status geändert hat, erhalten.  State change in child: terminated, stopped by a signal, resumed by a signal.  Wenn **KEIN WAIT** ausgeführt wird, dann verbleibt der terminierte Kindprozess im **ZOMBIE-STATUS**. 🡪 wait for child = release resources |
| fork – create a child child process | Erstellt einen neuen Prozess mit genau einem Thread, indem der aufrufende Prozess (**Der Thread, der fork() aufruft**) dupliziert wird. Der Kindprozess erbt unter anderem File-Descriptoren von offenen Dateien, Mutex-Stati, den gesamten virtuellen Adressbereich des Elternprozesses. |
| abort – cause **abnormal** process termination | Aufforderung zur Beendigung eines Prozesses.  First unblocks the **SIGABRT** signal, then raises that signal for the calling process. |
| exit – cause **normal** process termination |  |

**UNIX MANUAL SECTIONS**

The standard sections of the manual include:

|  |  |
| --- | --- |
| 1 | User Commands |
| 2 | System Calls |
| 3 | C Library Functions |
| 4 | Devices and Special Files |
| 5 | File Formats and Conventions |
| 6 | Games et. Al. |
| 7 | Miscellanea |
| 8 | System Administration tools and Deamons |

Distributions customize the manual section to their specifics, which often include additional sections.

# Aufgabe 1 – Betriebssystemressourcen anzeigen (top, htop)

**~$ top –u remi –d 1**

**~$ htop –u remi –d 1**

Toggle threads display -🡪 **H**

# Aufgabe 2 – Prozesse mit fork() erzeugen

i vor fork: 5

… wir sind die Eltern ‚PID‘ mit i=4 und Kind ‚pid‘,

Unsere Eltern sind ‚ParentPID‘

… ich bin das Kind ‚PID‘ mit i=6, meine Eltern sind ‚ParentPID‘

. . . . . und wer bin ich ?

. . . . . und wer bin ich ?

# Aufgabe 3 – Prozesse mit execl() erzeugen und ausführen

b) Der Kindprozess gibt „. . . . . und wer bin ich ?“ nicht aus.

Die C-Library-Funktion execl() ersetzt das aktuelle Prozess-Image mit einem neuen Prozess-Image.

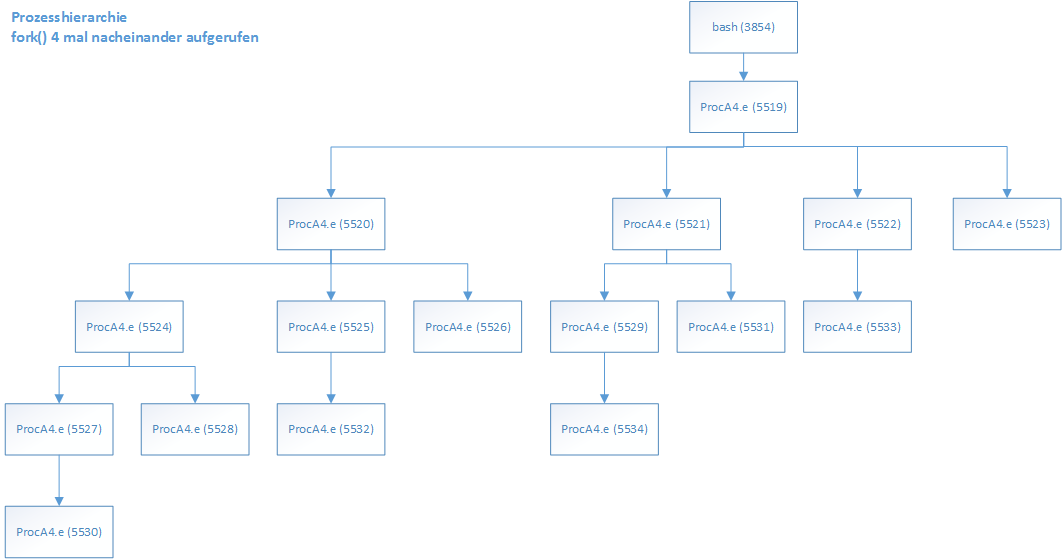
c) return value ist kleiner als 0 🡪 perror gibt den Grund des Fehlers aus: No such file or directory

d) change permissions of a file: **chmod -x**  
ChildProgA3.e vor Manipulation: -rwxrwxr-x

ChildProgA3.e nach Manipulation: -rw-rw-r- -

return value ist kleiner als 0 🡪 perror gibt den Grund des Fehlers aus: Permission denied

# Aufgabe 4 – Prozesshierarchie



Bash-Output des Programmes ProcA4.e:

PID: 5519 PPID: 3854

PID: 5520 PPID: 5519

PID: 5524 PPID: 5520

PID: 5527 PPID: 5524

PID: 5521 PPID: 5519

PID: 5525 PPID: 5520

PID: 5530 PPID: 5527

PID: 5528 PPID: 5524

PID: 5522 PPID: 5519

PID: 5532 PPID: 5525

PID: 5529 PPID: 5521

PID: 5523 PPID: 5519

PID: 5534 PPID: 5529

PID: 5531 PPID: 5521

PID: 5533 PPID: 5522

PID: 5526 PPID: 5520

# Aufgabe 5 – Zeitlicher Ablauf

CPU-Affinität 🡪 **CPU 0**

Nicht deterministisch; der Scheduler plant die Prozesse ein.

O(1)-Scheduler-Systemtakt unter Architektur x86 ist standardmässig 1000 Hz 🡪 1 ms

Interne (statische) Prioritäten für normale Timesharing-Prozesse reichen von **100 bis 139**.

Der Scheduler bevorzugt Prozesse mit höherer Priorität (100 = höchste).

Es wird versucht Prozesse gleicher Priorität werden solange abzuarbeiten, bis das Quantum von allen Prozessen der Liste verbraucht ist. Erst danach wird eine Prozessliste mit geringerer Priorität bearbeitet.

Das Quantum eines Prozesses wird von der periodischen Scheduling-Funktion bei jedem Tick reduziert, solange, bis es den Wert 0 erreicht hat.

Die Quantumsberechnung hängt von der statischen Priorität ab. Je höher der Wert der Priorität (also je niedriger die statische Priorität ist), umso kleiner ist das Quantum.

Falls die statische Priorät ist:

Falls die statische Priorät ist:

Neben der statischen Priorität und des Quantums verwaltet der O(1)-Scheduler für jeden Prozess einen Bonuswert.

Einen Bonus erhalten Prozesse, die viel schlafen (warten auf Ein-/Ausgabe).

**Fazit**: der Scheduler bevorzugt Ein-/Ausgabe-intensive (interaktive) Prozesse. Rechenintensive Prozesse verbrauchen ihr Quantum schnell. Ob ein Prozess interaktiv ist, hängt im Wesentlichen von seiner durchschnittlichen Schlafzeit und seiner Priorität ab.

# Aufgabe 6 – Verwaiste Kindprozesse

b) Wenn der Elternprozess terminiert, erbt der Kindprozess eine neue Parent-Prozess-ID (in diesem Fall war es die PID von init)

c) Der Elternprozess warten nicht auf Statusänderung des Kindprozesses 🡪 Der Kindprozess terminiert vor dem Elternprozess, d.h. er verwaist.

# Aufgabe 7 – Zombies unter Unix

c) wait() gibt die Prozess-ID des terminierten Kindprozesses zurück. Bei Fehler 🡪 -1

wait(NULL) speichert keine Statusinformation.

# Aufgabe 8 – wait() und waitpid()

See section 7 of man pages for **signal (overview of signals)**

Actions:

|  |  |
| --- | --- |
| Term | Default action is to terminate the process |
| Core | Default action is to terminate the process and dump core |

b)

**~$ ./ProcA8.e 1**

\*\*\* I am the child having job nr. 1 \*\*\*

Child exits on signal 11

Child exits with core dump 128

Signal 11 unter x86-Architektur bedeutet SIGSEGV – Invalid memory reference, **Action: Core**

core dump: a file containing an image of the process’s memory at the time of termination.

core dump einschalten: ulimit –c unlimited

core dump file lesen mit: gdb –c core ChildProgA8.e

**CORE DUMP**

Reading symbols from ChildProgA8.e...done.

[New LWP 11092]

Core was generated by `ChildProgA8.e 1'.

Program terminated with signal SIGSEGV, Segmentation fault.

#0 0x000000000040070f in main (argc=2, argv=0x7fff626183f8)

at ChildProgA8.c:37

37 case 1: \*a = i; // force segmentation error

c)

**~$ ./ProcA8.e 2**

\*\*\* I am the child having job nr. 2 \*\*\*

Child exits on signal 30

Child exits with core dump 0

Signal 30 unter x86-Architektur bedeutet SIGPWR – Power failure, **Action: Term**

d)

**~$ ./ProcA8.e 3**

\*\*\* I am the child having job nr. 3 \*\*\*

Child exits on signal 6

Child exits with core dump 128

Signal 6 unter x86-Architektur bedeutet SIGABRT – Abort signal from abort(3), **Action: Core**

Der Kindprozess wird in einer abnormalen Weise terminiert.

**CORE DUMP**

Reading symbols from ChildProgA8.e...done.

[New LWP 2645]

Core was generated by `ChildProgA8.e 3'.

Program terminated with signal SIGABRT, Aborted.

#0 0x00007fe418e2df20 in \_\_nanosleep\_nocancel ()

at ../sysdeps/unix/syscall-template.S:81

81 ../sysdeps/unix/syscall-template.S: No such file or directory.

e)

**~$ ./ProcA8.e 4**

\*\*\* I am the child having job nr. 4 \*\*\*

. . . child is playing

. . . child is playing

. . . child is playing

. . . child is playing

. . . child is playing

. . . child is playing

. . . child is playing

. . . child is playing

. . . child is playing

Child has exited with 'exit(222)'

waitpid(pid, &status, WNOHANG) retourniert 0 solange der Status des Kindprozesses mit der Prozess-ID pid sich nicht ändert.

**WEXITSTATUS(status)**

returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit(3) or \_exit(2) or as the argument for a return statement in main(). This macro should only be employed if WIFEXITED returned true.

The exit() function causes normal process termination and **the value of status & 0377 is returned to the parent.**

# Aufgabe 9 – Prozessräume

a)

Hallo, I am on the way to fork now, ……look: I am the parent

clear ?

Hallo, I am on the way to fork now, ……look: I am the child

clear ?

Der String „Hallo, I am on the way to fork now, ……lo“ bleibt im Buffer des Elternprozesses, weil nicht geflushed wird, z.B. mit **fflush(stdout)** oder **‚\n‘** am Stringende.

Der Kindprozess erbt den Buffer des Elternprozesses.

b)

c) Eltern- und Kindprozess schreiben in die Datei AnyOutPut.txt. Der Kindprozess erbt den File-Desktriptor der geöffneten Datei vom Elternprozess.

Inhalt der Datei AnyOutPut.txt

|  |  |  |
| --- | --- | --- |
| **Run #1** | **Run #2** | **Run #3** |
| Mami 1  Mami 2  Mami 3  Mami 4  Mami 5  Mami 6  Mami 7  Mami 8  Mami 9  Mami 10  Mami 11  Fritzli 1  Fritzli 2  Fritzli 3  Mami 12  Mami 13  Mami 14  Mami 15  Fritzli 4  Fritzli 5  Fritzli 6  Fritzli 7  Fritzli 8  Fritzli 9  Fritzli 10  Fritzli 11  Fritzli 12  Fritzli 13  Fritzli 14  Fritzli 15 | Mami 1  Mami 2  Mami 3  Mami 4  Mami 5  Mami 6  Mami 7  Fritzli 1  Mami 8  Fritzli 2  Mami 9  Fritzli 3  Mami 10  Fritzli 4  Mami 11  Fritzli 5  Mami 12  Fritzli 6  Mami 13  Fritzli 7  Fritzli 8  Mami 14  Mami 15  Fritzli 9  Fritzli 10  Fritzli 11  Fritzli 12  Fritzli 13  Fritzli 14  Fritzli 15 | Mami 1  Mami 2  Mami 3  Mami 4  Mami 5  Mami 6  Fritzli 1  Fritzli 2  Fritzli 3  Fritzli 4  Mami 7  Fritzli 5  Mami 8  Mami 9  Mami 10  Mami 11  Mami 12  Fritzli 6  Fritzli 7  Fritzli 8  Fritzli 9  Mami 13  Fritzli 10  Mami 14  Fritzli 11  Mami 15  Fritzli 12  Fritzli 13  Fritzli 14  Fritzli 15 |

# Aufgabe 10 – Threads

a)

b) Ein ProcA10-Prozess schläft und zwei ProcA10-Threads sind im Running-State.

Unter Linux mit der pthreads-Implementation haben Threads eine eigne PID.

pthread\_join() wartet auf Beendigung des Threads ab. Wenn nicht gewartet wird, dann wird der Prozess normal terminiert und mit ihm alle seine Threads.

# Aufgabe 11 – Bestimmung der Rechenzeit

b)

**~$ ./ProcA11\_1.e**

Elapsed: 1.590

User CPU-time: 0.740

System CPU-time: 0.020

**~$ ./ProcA11\_1.e 1**

Elapsed: 10.280

User CPU-time: 0.750

System CPU-time: 0.010

Die Verweilzeit erhöht sich, da der rechenintensive Prozess vom Scheduler weniger bevorzugt wird.

# Aufgabe 12 – Daemon

a)

**PlapperMaul.e**

Console schliessen 🡪 Programm PlapperMaul.e terminiert

Log out 🡪 Alle User-Programme terminieren, auch PlapperMaul.e

b)

**MrTimeDaemon.e**

Console schliessen 🡪 Daemon MrTimeDaemon.e läuft weiterhin

Log out 🡪 Daemon MrTimeDaemon.e läuft weiterhin

Solange Prozess-ID 1 (init) läuft, läuft auch Kindprozess MrTimeDaemon.

c)

\*\*\* daemon is already running \*\*\*

Das Programm kann nur einmal gestartet werden.

f)

1. - aktuellen Prozessen forken.  
 - Elternprozess (PID > 0) terminieren mit exit(0)

2. - der Prozess mit ID 1 (init) wird der neue Elternprozess des Kindprozesses

- signal(SIGINT, SIG\_IGN) 🡪 unterbindet Beendigung des Prozesses mit CTRL-C  
 (Interrupt from keyboard)  
 - signal(SIGQUIT, SIG\_IGN) 🡪 unterbindet quit from keyboard  
 - signal(SIGHUP, SIG\_IGN) 🡪 Hangup detected on controlling terminal or death of controlling  
 process

3. - setsid() macht den Kindprozess zum Session-Leader (und Prozessgruppen-Leader)  
 - in sicheres (festes, nicht montiertes) Verzeichnis des Systems wechseln  
 - umask(0) 🡪 setzt Dateierzeugungsmaske auf 0, d.h. der Daemon kann beliebige  
 Zugriffsrechte beim Öffnen von Dateien verlangen.  
- damit der Daemon nur einmal gestartet werden kann, wird ein Lockfile aufgesetzt.

g)

bind failed : Address already in use

Kann nicht ein zweites Mal die Adresse localhost auf dasselbe Socket binden.

See man 2 bind:

It is normally necessary to assign a local address using bind() before a SOCK\_STREAM socket may receive connections