

# Verteilte Systeme

...für C++ Programmierer

Prozesse

by

Dr. Günter Kolousek

- ▶ Programm besteht aus Anweisungen
- ▶ Prozess (POSIX: portable operating system interface)
  - ▶ entsteht bei der Ausführung eines Programmes
  - ▶ hat einen oder mehrere Threads
  - ▶ hat eigenen virtuellen Adressraum (→ MMU)
  - ▶ hat Rechte auf Ressourcen (Dateien, Geräte)
  - ▶ hat keinen Zugriff auf Ressourcen eines anderen Prozesses (Hauptspeicher, offene Dateien, benutzte Geräte)
- ▶ Kenndaten
  - ▶ PID und PPID, UID und GID, EUID und EGID
  - ▶ PRI, Zustand (running, runnable, sleeping, stopped, traced),...

## Gründe für Parallelisierung auf Prozessebene

- ▶ Entwicklungsaufwand gering
  - ▶ wenn mehrfaches Starten eines Prozesses als Parallelisierung ausreicht
- ▶ Verteilung auf mehrere Rechner einfach
  - ▶ aber keine Abhängigkeiten erforderlich
- ▶ Robustheit
  - ▶ durch Speicherschutz (auch gegen Angriffe)

# Prozess – 3

- ▶ Eltern-Kind-Beziehung
  - ▶ Jeder Prozess hat einen Elternprozess (außer dem “init”-Prozess)
  - ▶ Jeder Prozess kann mehrere Kindprozesse haben
- ▶ Basiskommunikation mit Umwelt
  - ▶ Umgebungsvariablen
  - ▶ Kommandozeilenargumente
    - ▶ → `argc, argv`
  - ▶ `stdin, stdout, stderr`
    - ▶ → `cin, cout, cerr`
  - ▶ Exit-Code
    - ▶ `return` von `main`, `exit`
  - ▶ Signale

# Prozess – 4

Prozess besteht aus

- ▶ Textsegment (Programmcode): `.text`
  - ▶ schreibgeschützt
- ▶ Datensegment (Benutzerdaten)
  - ▶ initialisiert, schreibgeschützt: `.rodata`
  - ▶ initialisiert, nicht schreibgeschützt: `.data`
  - ▶ nicht initialisiert: `.bss`
  - ▶ heap
- ▶ Stacksegment
- ▶ Shared-Memory-Segment

# PID & PPID

```
#include <iostream> // pid.cpp  
// not part of standard C; part of POSIX.1  
// therefore it is not named: <cunistd>  
#include <unistd.h>
```

```
using namespace std;
```

```
int main() {  
    cout << "pid: " << getpid() << endl;  
    cout << "ppid: " << getppid() << endl;  
    cout << "uid: " << getuid() << endl;  
    cout << "euid: " << geteuid() << endl;  
}
```

# Prozesse und Shell unter Linux

- ▶ Vordergrund vs. Hintergrund (meist &)
- ▶ ps
  - ▶ ps
  - ▶ ps -e ... alle Prozesse
  - ▶ ps -f ... “full format”
  - ▶ ps -L ... mit Threads
- ▶ pstree
- ▶ top bzw. htop

# Prozesse und Shell unter Linux – 2

- ▶ jobs
  - ▶ CTRL-Z...kill -TSTP xxx
- ▶ bg
  - ▶ bg %1
  - ▶ bg xxx...kill -CONT xxx
- ▶ kill
  - ▶ kill xxx...beenden (wie kill -TERM xxx)
  - ▶ kill -KILL xxx...keine Widerrede!



# Starten eines Prozesses

```
#include <iostream> // clone.cpp
// not part of standard C → *not* <cunistd>
#include <unistd.h> // fork
using namespace std;
int main() {
    cout << "just before forking...";
    fork();
    cout << "after fork()!" << endl;
}
```

```
just before forking...after fork()!
just before forking...after fork()!
```

# Starten eines Prozesses – 2

- ▶ `fork()` dupliziert Prozess!
  - ▶ inklusive Puffer, deshalb flushen:

```
#include <iostream>    // clone2.cpp
#include <unistd.h>
using namespace std;
int main() {
    cout << "just before forking..."<< endl;
    fork();
    cout << "after fork()!" << endl;
}
```

```
just before forking...
after fork()!
after fork()!
```

# Starten eines Prozesses – 3

```
#include <iostream>    // fork.cpp
#include <unistd.h>     // sleep
#include <cstdlib>      // quick_exit
using namespace std;
int main() {
    auto pid{fork()};
    if (pid == 0) {
        cout << "child is waiting... " << flush;
        sleep(10);  cout << "done" << endl;
        quick_exit(EXIT_SUCCESS);
    } else {
        cout << "child pid is " << pid << endl;
    }
    cout << "parent terminates" << endl;
}
```

# Starten eines Prozesses – 4

- ▶ Ausgabe

```
child pid is 28227
parent terminates
child is waiting...
<after 10 seconds>
done
```

- ▶ Semantik von fork

- ▶ Kindprozess: Duplikat inkl. Register, offener Dateien,...
- ▶ beide Prozesse liefern Rückgabewert von fork
  - ▶ Kindprozess erhält 0
  - ▶ Vaterprozess erhält pid des Kindprozesses
- ▶ Textsegment wird nicht kopiert
- ▶ Daten-, Stack- und Heapsegment: Copy on Write

- ▶ `quick_exit`: Destruktoren von Obj. mit Lebensdauer automatisch, statisch, threadlokal → **kein** Aufruf!

# Starten eines Prozesses – 5

```
#include <iostream>    // waitpid.cpp
#include <unistd.h>
#include <cerrno>       // errno
#include <cstdlib>       // exit
#include <sys/wait.h>    // waitpid
using namespace std;
int main() {
    pid_t pid;
    pid = fork();
    if (pid == -1) {
        cerr << "forking failed: " << errno << endl;
        exit(EXIT_FAILURE);
    }
```

# Starten eines Prozesses – 6

```
if (pid == 0) {  
    cout << "child process here!" << endl;  
    sleep(3);  
    quick_exit(EXIT_SUCCESS);  
} else {  
    cout << "child pid is " << pid << endl;  
    cout << "waiting for child..." << endl;  
    int status;  
    waitpid(pid, &status, 0); // 0...options  
    cout << "child terminated w/ exit code "  
        << status << endl;  
    exit(EXIT_SUCCESS);  
}  
}
```

# Starten eines Prozesses – 7

Ausgabe:

```
child pid is 30587  
waiting for child...  
child process here!  
child terminated w/ exit code 0
```

# Exit codes

```
if (pid == 0) {  
    quick_exit(123); // <---  
} else {  
    cout << "waiting for child..." << endl;  
    int status;  
    waitpid(pid, &status, 0); // 0...options  
    cout << "child terminated w/ exit code "  
        << status << endl;  
    exit(EXIT_SUCCESS);  
}  
}
```

waiting for child...

child terminated w/ exit code 31488



# Exit codes – 2

```
if (pid == 0) {  
    quick_exit(123); // <---  
} else {  
    cout << "waiting for child..." << endl;  
    int status;  
    waitpid(pid, &status, 0); // 0...options  
    if (int s{WIFEXITED(status)}; s)  
        cout << "child terminated w/ exit code "  
            << WEXITSTATUS(status) << endl;  
    else  
        cout << "term. neither by exit nor by return"  
            << endl;  
    exit(EXIT_SUCCESS);  
} }
```

waiting for child...

child terminated w/ exit code 123

# Exit codes – 3

Es werden Zusatzinformationen im Statuscode abgelegt, wenn dieser ungleich 0 ist!

- ▶ `WIFEXITED(status)` ...  $\neq 0$ , wenn normal beendet
- ▶ `WEXITSTATUS(status)` ... der Statuscode selber
- ▶ `WIFSIGNALED(status)` ...  $\neq 0$ , wenn mit einem Signal beendet
- ▶ `WTERMSIG(status)` ... die Nummer des Signals
- ▶ ... siehe man page von `waitpid()`

# Semantik des Elternprozesses

- ▶ Ruft Elternprozess `waitpid` nicht auf, dann
  - ▶ bleibt Kindprozess als so genannter “Zombie” bestehen, wenn sich dieser vor dem Elternprozess beendet.
- ▶ Beendet sich Elternprozess vor Kind, dann
  - ▶ bezeichnet man den Kindprozess als “orphaned” (verwaist)
  - ▶ “init”-Prozess mit PID 1 übernimmt diesen Kindprozess als Elternprozess
  - ▶ beendet sich dann Kindprozess → “init” wird `waitpid` aufrufen, d.h. kein Zombie!

# Zombie

```
#include <iostream> // zombie.cpp
#include <unistd.h>
#include <cstdlib>
using namespace std;
int main() {
    auto pid{fork()};
    if (pid == 0) { quick_exit(EXIT_SUCCESS); }
    cout << "child: " << pid << endl;
    sleep(60);
}
```

```
$ zombie&
```

```
child: 31715
```

```
$ ps 31715
```

PID	TTY	STAT	TIME	COMMAND
31715	pts/2	Z	0:00	[zombie] <defunct>

# Zombie – 2

```
#include <iostream>    // zombie2.cpp
#include <unistd.h>
#include <cstdlib>
#include <csignal>    // signal
#include <sys/wait.h>
using namespace std;
pid_t pid;

void signal_handler(int signal) {
    int status;
    sleep(10);
    waitpid(pid, &status, 0);    // auch nullptr mögl
    cout << "child's end awaited" << endl;
}
```

# Zombie – 3

```
int main() {  
    signal(SIGCHLD, signal_handler);  
    pid = fork(); // starts just one child  
    if (pid == 0) { quick_exit(EXIT_SUCCESS); }  
    cout << "child: " << pid << endl;  
    sleep(60);  
}
```

```
$ zombie2 &
```

```
child: 29776
```

```
$ ps 29776
```

PID	TTY	STAT	TIME	COMMAND
29776	pts/2	Z	0:00	[zombie2] <defunct>

```
$ child's end awaited
```

```
Job 1, 'zombie2 &' hat beendet
```

# Verwaist...

```
#include <iostream> // orphan.cpp
#include <unistd.h>
#include <cstdlib>
using namespace std;
int main() {
    auto pid = fork();
    if (pid == 0) {
        cout << "parent: " << getppid() << endl;
        sleep(5);
        cout << "parent: " << getppid() << endl;
        quick_exit(EXIT_SUCCESS); }
    sleep(3);
}
```

parent: 5662

parent: 1

# Signale

```
#include <iostream>    // signal.cpp
#include <unistd.h>
#include <csignal>
using namespace std;
int pid;

void signal_handler(int signal) {
    cout << "ignoring signal!" << endl;
}

int main() {
    signal(SIGTERM, signal_handler);
    auto i=3;
```



# Signale – 2

```
while (i) {  
    cout << i * 5 << " seconds left..." << endl  
    sleep(5); // will be interrupted by signal  
    --i; }  
cout << "terminating myself" << endl;  
}
```

```
$ signal&  
15 seconds left...  
$ kill %1  
ignoring signal!  
10 seconds left...  
$ 5 seconds left...  
terminating myself  
Job 1, 'signal&' hat beendet
```

# Signale – 3

```
#include <iostream> // killit.cpp
#include <csignal>
using namespace std;
int pid;

int main() {
    int pid;
    cout << "pid: ";
    cin >> pid;
    // use 'raise' for sending signals to
    // the current running process, otherwise
    // use 'kill'. Anyway, it's easier to
    // remember just one system call
    kill(pid, SIGKILL);
}
```

# Signale – 4

Wichtige Signale sind:

SIGHUP (1) hangup

SIGINT (2) interrupt (Keyboard; CTRL-C)

SIGKILL (9) beenden!!

SIGSEGV (11) Adressbereichsfehler!!

SIGTERM (15) beenden

SIGCHLD (17) Ein Kind hat sich beendet!

# execl

```
#include <iostream>    // execl.cpp
#include <unistd.h>     // execl
using namespace std;

int main() {
    // path of the executable
    // name which will be used in process table
    // 0-terminated list of arguments
    execl("/usr/bin/date", "date",
          "--iso-8601", nullptr);
}
```

2021-09-20

liefert -1 zurück, wenn nicht erfolgreich und die globale Variable  
errno wird gesetzt!

# Fehler in C

- ▶ Im C-API werden Fehler oft dadurch angezeigt, dass ein Fehlercode zurückgeliefert wird.
- ▶ Oft wird die globale Variable `errno` mit einem Wert belegt.
  - ▶ `#include <cerrno>`
- ▶ Die zu `errno` gehörige Fehlermeldung kann mittels der Funktion `strerror` ermittelt werden:

```
cerr << strerror(errno) << endl;
```

  - ▶ `#include <cstring>`

# Umgebungsvariable

```
#include <iostream>    // printenv.cpp
#include <cstdlib>      // getenv
using namespace std;

int main() {
    const char* env_shell{getenv("SHELL")};
    if (env_shell)
        cout << env_shell << endl;
    else
        cout << "SHELL not set" << endl;
}

/usr/bin/fish
```

# Interprocess Communication (IPC)

- ▶ Pipes
  - ▶ byte-weise
- ▶ FIFO
  - ▶ wie pipe, aber Name
- ▶ File-locking
- ▶ Message Queues
  - ▶ wie FIFO, aber Nachrichten-basiert
- ▶ Semaphore
- ▶ Shared Memory
- ▶ Memory Mapped Files
- ▶ Sockets