3. Deadlocks

Sie haben im Lehrbrief gesehen, dass sich Deadlocks verhindern lassen, indem Sie die Ressourcen (Locks) in eine Reihenfolge bringen und in allen Threads die Ressourcen immer in dieser festen Reihenfolge anfordern. Das macht das gleichzeitige Auftreten von Code-Abschnitten der Form

```
lock (A);                lock (B);
lock (B);                lock (A);
...
unlock (B);              unlock (B);
unlock (A);              unlock (A);
```

unmöglich. Da die Beachtung dieser Regel (wir nennen Sie *Regel A*) umständlich ist, wird ein alternatives Verfahren (*Regel B*) vorgeschlagen: In der Anwendung wird ein zusätzliches Lock `super` definiert, das immer vor dem ersten Aufruf von `lock()` als erstes gelockt werden muss. Wenn `super` gelockt wurde, dürfen die übrigen Ressourcen in beliebiger Reihenfolge gelockt werden. Erst nachdem alle regulären Ressourcen zurück gegeben wurden, darf der Code auch `super` zurückgeben. Die beiden obigen Code-Blöcke sind nach den neuen Regeln also zulässig, erhalten aber die folgende Form (neuer Code **fett**):

```
lock (super);            lock (super);
lock (A);                lock (B);
lock (B);                lock (A);
...
unlock (B);              unlock (B);
unlock (A);              unlock (A);
unlock (super);          unlock (super);
```

a) Warum sorgt *Regel A* immer für Deadlock-Freiheit?

b) Sorgt auch *Regel B* immer für Deadlock-Freiheit?

c) Es ist viel einfacher, beim Programmieren *Regel B* umzusetzen, als *Regel A* umzusetzen. Aber ist dieser Ansatz empfehlenswert?

4. Signale

Betrachten Sie das Programm `signal-pingpong.c` auf der Rückseite. Es erzeugt einen Kindprozess, und die beiden Prozesse nutzen dann den Signal-Mechanismus, um sich gegenseitig im Ping-Pong-Stil Signale zuzusenden. Dabei muss einer den Anfang machen: Das übernimmt der Vaterprozess.

Zum Verständnis: `signal (Nummer, Funktion)` trägt im laufenden Prozess einen Signal-Handler für die angegebene Signalnummer ein, und dieser wird automatisch aufgerufen, wenn ein Prozess mit `kill (ProzessID, Nummer)` dieses Signal an den Prozess schickt.

a) Laden Sie das Programm von der Kurswebseite herunter; der Direkt-Link ist
http://swf.hgesser.de/vb-b1-ss2017/prakt/signal-pingpong.c

b) Testen Sie das Programm unter Linux oder macOS: Sie können es mit

```
gcc -o signal-pingpong signal-pingpong.c
./signal-pingpong
```

übersetzen und starten.

c) Betrachten Sie die Ausgabe und finden Sie heraus, warum

– die Länge der „+“- und „–“-Zeilen zunächst wächst und dann wieder schrumpft,

– die „+“- und „–“-Zeilen immer abwechselnd erscheinen.

d) Warum wird am Schluss nur einer der beiden Prozesse beendet?

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

int meine_pid, partner_pid, pid;
int counter = 1;
char symbol;

void show_symbols (char sym, int count, char *string) {
  // Funktion gibt count-mal das Zeichen sym und als Abschluss string aus
  int i;
  sleep (1);
  for (i = 0;  i < 3*count;  i++)
    printf ("%c", sym);
  printf ("%s", string);
}

void usr1_handler_b (int sig) {
  counter--;
  if (counter < 1)
    exit (0);
  show_symbols (symbol, counter, "\n");
  kill (partner_pid, SIGUSR1);
}

void usr1_handler_a (int sig) {
  counter++;
  if (counter > 4)
    signal (SIGUSR1, usr1_handler_b);  // neuen Handler eintragen
  show_symbols (symbol, counter, "\n");
  kill (partner_pid, SIGUSR1);
}

int main () {
  char message[20];
  printf ("Signal-Handler-Ping-Pong\n");
  signal (SIGUSR1, usr1_handler_a);   // Handler eintragen
  pid = fork ();
  meine_pid = getpid ();
  if (pid == 0) {
    partner_pid = getppid ();  // ID vom Vater
    symbol = '-';
  } else {
    partner_pid = pid;
    symbol = '+';
  }
  printf ("Prozess %d: Mein Partner ist Prozess %d\n", meine_pid, partner_pid);
  if (pid != 0) {
    // starte Ping-Pong
    sleep (5); kill (partner_pid, SIGUSR1);
  }

  sprintf (message, " main() - %d\n", meine_pid);  // baut String zusammen
  for (;;) {
    if (counter <= 1)
      show_symbols ('*', counter, message);
    sleep (5);
  }
}
```