Prozesse und Threads

Ein Unix-Prozess hat

- IDs (process,user,group)
- Umgebungsvariablen
- Verzeichnis
- Programmcode
- Register, Stack, Heap
- Dateideskriptoren, Signale
- message queues, pipes, shared memory Segmente
- Shared libraries

Jeder Prozess besitzt seinen eigenen Adressraum

Threads existieren innerhalb eines Prozesses

Threads teilen sich einen Adressraum

Ein Thread besteht aus

- ID
- Stack pointer
- Register
- Scheduling Eigenschaften
- Signale

Erzeugungs- und Umschaltzeiten sind kürzer

„Parallele Funktion“

Pthreads

- Jeder Hersteller hatte eine eigene Implementierung von Threads oder „light weight processes“
- 1995: IEEE POSIX 1003.1c Standard (es gibt mehrere “drafts“)
- Definiert Threads in portabler Weise
- Besteht aus C Datentypen und Funktionen
- Header file `pthread.h`
- Bibliotheksname nicht genormt. In Linux `-lpthread`
- Übersetzen in Linux: `gcc <file> -lpthread`

Abstraktion in C++

- Thread interface in C++ seit C++11
- Abstraction für native thread bibliothek
- nutzt unter Linux pthread
- Bessere Intergation in C++ Konzepte

Erzeugen von Threads

- `std::thread` : Kapselt einen Thread und dessen Zustand
- *Opaquer Typ: Genauer Datentyp wird in der Implementierung definiert und implementierungsabhängig.*
- Konstruktion:

```
template< class Function, class... Args >
explicit thread( Function&& f, Args&&... args );
```

  - kopiert alle Daten in thread lokalen Speicher
  - startet `f(args...)` in einem neuen Thread
  - `f` ist ein Callable
- Threads können weitere Threads starten, maximale Zahl von Threads ist implementierungsabhängig

Erzeugen von Threads II

```
1  #include <iostream>
2  #include <string>
3  #include <thread>
4
5  void hello(std::string name)
6  {
7      std::cout << "Hello␣" << name << std::endl;
8  }
9
10 int main ()
11 {
12     // start thread
13     std::thread t(hello, "world");
14
15     ...
16 }
```

Kompilieren: `g++ -std=c++11 -pthread hellofromthread.cpp`

Erzeugen von Threads III

```
1  #include <iostream>
2  #include <string>
3  #include <thread>
4
5  class Say
6  {
7      std::string opener;
8  public:
9      Say (std:string s) : opener(s) {}
10     void operator() (std::string name) {
11       std::cout << opener << name << std::endl;
12     }
13 };
14
15 int main ()
16 {
17     Say say("Hello");
18     // start thread
19     std::thread t(say, "world");
20
21     ...
22 }
```

Warten auf Threads

```
1  #include <iostream>
2  #include <thread>
3
4  void hallo()
5  {
6      std::cout << "Hallo␣Welt" << std::endl;
7  }
8
9  int main ()
10 {
11     std::thread t(hallo);
12 }
```

Ausgabe:

```
1  terminate called without an active exception
2  HaAborted (core dumped)
```

Warten auf Threads

```
1  #include <iostream>
2  #include <thread>
3
4  void hallo()
5  {
6      std::cout << "Hallo␣Welt" << std::endl;
7  }
8
9  int main ()
10 {
11     std::thread t(hallo);
12 }
```

- Am Ende des Programms auf alle threads warten
- `std::thread::join()` wartet auf die Beendigung des threads

Warten auf Threads

```
1  #include <iostream>
2  #include <thread>
3
4  void hallo()
5  {
6      std::cout << "Hallo␣Welt" << std::endl;
7  }
8
9  int main ()
10 {
11     std::thread t(hallo);
12     t.join();
13 }
```

Thread Management Beispiel

```
1  #include <iostream>
2  #include <vector>
3  #include <thread>
4
5  std::vector<double> data;
6
7  void doCalculation(int i) {
8      ...
9  }
10
11 int main ()
12 {
13     std::vector<std::thread> pool(P);
14     // start threads
15     for (i=0; i<P; i++)
16         pool[i] = std::thread(doCalculation, i);
17
18     // wait for threads to finish
19     for(auto & t : pool)
20         t.join();
21
22     return 0;
23 }
```

Übergeben von Argumenten

- Übergeben Argumente werden Kopiert.
- Veränderbare Daten müssen als Referenz übergeben werden

```
1  ...
2
3  void doCalculation(std::vector<double> & v, const int & process) {
4      int myId = process;
5      ...
6  }
7
8  int main () {
9      std::vector<double> v(10000);
10
11     // start threads
12     std::vector<std::thread> pool(P);
13     int i;
14     for (i=0; i<P; i++)
15         pool[i] = std::thread(doCalculation, std::ref(v), std::cref(i));
16     ...
17 }
```

- Vorsicht mit Referenzen!
  - Inhalt von ist möglicherweise verändert bevor Thread liest
  - Stackvariablen ist exisitieren möglicherweise nicht mehr
  - Konkurierende Schreibzugriffe

Übergeben von Argumenten

- Übergeben Argumente werden Kopiert.
- Veränderbare Daten müssen als Referenz übergeben werden

```
1  ...
2
3  void doCalculation(std::vector<double> & v, int process) {
4      int myId = process;
5      ...
6  }
7
8  int main () {
9      std::vector<double> v(10000);
10
11     // start threads
12     std::vector<std::thread> pool(P);
13     int i;
14     for (i=0; i<P; i++)
15         pool[i] = std::thread(doCalculation, std::ref(v), i);
16     ...
17 }
```

- Vorsicht mit Referenzen!
  - Inhalt von ist möglicherweise verändert bevor Thread liest
  - Stackvariablen ist exisitieren möglicherweise nicht mehr
  - Konkurierende Schreibzugriffe

Rückgabe Werte

```
1  #include <iostream>
2  #include <cmath>
3  #include <thread>
4
5  double calc(double a, double b)
6  {
7      return 10*a+b;
8  }
9
10 int main ()
11 {
12     // start thread
13     std::thread t(calc, 10.0, 12.0);
14
15     ...
16 }
```

- Rückgabewerte werden ignoriert → call-by-reference

Rückgabe Werte

```
1  #include <iostream>
2  #include <cmath>
3  #include <thread>
4
5  void calc(double a, double b, double & result)
6  {
7      result = 10*a+b;
8  }
9
10 int main ()
11 {
12     // start thread
13     double result;
14     std::thread t(calc, 10.0, 12.0, std::ref(result));
15
16     ...
17 }
```

- Rückgabewerte werden ignoriert → call-by-reference

Synchronisation: Mutex Variablen

- Mutex Variablen realisieren den *wechselseitigen Ausschluss*
- Erzeugen und initialisieren einer Mutex Variable:

```
std::mutex m;
```

  Mutex Variable ist nach Initialisierung im Zustand frei.
- Eintritt in den kritischen Abschnitt:

```
std::mutex::lock();
```

  Diese Funktion blockiert solange bis man drin ist.
- Versuche in den kritischen Abschnitt einzutreten:

```
std::mutex::try_lock();
```

  Diese Funktion liefert false, wenn das Lock nicht verfügbar war.
- Verlasse kritischen Abschnitt

```
std::mutex::unlock();
```

Beispiel Skalarprodukt

```
1  #include <thread>
2  #include <vector>
3  #include <cmath>
4
5  void sp(double & sum, const std::vector<double> & x, const std::vector<double> & y,
6          std::size_t p, std::size_t P)
7  {
8      double s=0;
9      for (int i=p*x.size()/P; i<std::min((p+1)*x.size()/P; i++) s += x[i]*y[i];
10     sum += s;
11 }
12
13 int main(int argc, char** argv)
14 {
15     const std::size_t N = ...;
16     const std::size_t P = ...;
17     std::vector<double> x(N), y(N);
18
19     // spawn computations
20     double sum = 0.0;
21     std::vector<std::thread> threads(P);
22     for (std::size_t p = 0; p<P; p++)
23         threads[p] = std::thread(sp, std::ref(sum), std::cref(x), std::cref(y), p, P));
24
25     for (std::size_t p = 0; p<P; p++)
26         threads[p].join();
27 }
```

Beispiel Skalarprodukt II

```
1  #include <thread>
2  #include <vector>
3  #include <cmath>
4
5  void sp(double & sum, const std::vector<double> & x, const std::vector<double> & y,
6          std::size_t p, std::size_t P, std::mutex & guard)
7  {
8      double s=0;
9      for (int i=p*x.size()/P; i<std::min((p+1)*x.size()/P; i++) s += x[i]*y[i];
10     guard.lock();
11     sum += s;
12     guard.unlock();
13 }
14
15 int main(int argc, char** argv)
16 {
17     const std::size_t N = ...;
18     const std::size_t P = ...;
19     std::vector<double> x(N), y(N);
20
21     // spawn computations
22     double sum = 0.0;
23     std::mutex guard;
24     std::vector<std::thread> threads(P);
25     for (std::size_t p = 0; p<P; p++)
26         threads[p] = std::thread(sp, std::ref(sum), std::cref(x), std::cref(y), p, P, std::ref(guard)));
27
28     for (std::size_t p = 0; p<P; p++)
29         threads[p].join();
30 }
```

Bedingungs-Variablen

- Warten auf ein Flag. Konzeptionell:

```
while (! flag);
```

- `std::condition_variable()`
  erzeugt eine Bedingungs-Variable. Diese ist nicht kopierbar!
- `std::condition_variable::wait()`
  wartet darauf, dass die Bedingung `true` wird.
- `std::condition_variable::notify_one()` oder
  `std::condition_variable::notify_all()`
  weckt einen / alle wartenden Prozesse auf.

```
1  void thread1(std::condition_variable & cv) {
2      // create some data
3      ...
4      // wakeup thread2
5      cv.notify_one();
6  }
7  void thread2(std::condition_variable & cv) {
8      // wait for thread1
9      cv.wait();
10     // do something with the data
11     ...
12 }
```

Parallele Summe

Nutzen Sie Ihr bisheriges Wissen um das Skalarprodukt mit paralleler Summe aus der Vorlesung zu implementieren.

Alternativer Ansatz: Future

[Baker und Hewitt, 1977]

- Bezeichnet einen Platzhalter für ein Ergebnis, das noch nicht bekannt ist.
- Berechnung findet in einem parallel laufenden Prozesse statt.
- Synchronisation der Prozesse geschieht implizit, durch ein Blockieren, falls das benötigte Future noch nicht vorliegt.
- Eine alternative Abstraktion, zum “Verstecken” von Threads.
- Ebenfalls mit C++-11 nutzbar.

Asynchrone Funktionsaufrufe

- Starten einer Funktion, welcher erst später ausgewertet wird:

```
template< class Function, class... Args >
std::future<typename std::result_of<Function(Args...)>::type>
async( std::launch policy, Function&& f, Args&&... args );
```

- Rückgabewert ist ein `std::future<...>`
- Ausführung kennt zwei Modi
  - `std::launch::async` wertet die Funktion in einem anderen Thread aus
  - `std::launch::deferred` verschiebt die Auswertung, bis der Wert benötigt wird
  - `f` und `args` sind wie bei `std::thread` ein Callable und eine Parameterliste.

std::future

- `std::future<T>`
  wird i.d.R. nicht explizit angelegt.
- `std::future<T>::get()`
  gibt das Ergebnis vom Type `T` zurück. Blockiert, falls dieses noch nicht vorliegt.
- `std::future<T>::wait()`
  wartet auf ein vorliegendes Ergebnis (siehe Bedingungs-Variable).

```
1  #include <future>
2  #include <iostream>
3
4  double calc(int parameter);
5
6  int main()
7  {
8      // spawn calculation
9      std::future<double> result = std::async(std::launch::async,calc,42);
10     // use the results
11     std::cout << "result " << result.get() << std::endl;
12 }
```

Paralleles Skalarprodukt mit Future

?!

Paralleles Skalarprodukt mit Future

```
1  #include <future>
2  #include <vector>
3  #include <random>
4  #include <iostream>
5  #include <algorithm>
6
7  double parallel_sp(const std::vector<double> & x, const std::vector<double> & y,
8                     std::size_t P, std::size_t A, std::size_t B)
9  {
10     if (P == 1) // just one process in the local pool
11         return std::inner_product(x.begin()+A,x.begin()+B,y.begin()+A,0.0);
12     // split the range and call our self recursively
13     std::size_t S = A+(B-A)*(P/2)/P;
14     auto s1 = std::async(std::launch::async, parallel_sp,std::cref(x),std::cref(y),P/2,A,S);
15     auto s2 = std::async(std::launch::async, parallel_sp,std::cref(x),std::cref(y),P-P/2,S,B);
16     return s1.get() + s2.get();
17 }
18
19 int main(int argc, char** argv)
20 {
21     const std::size_t N = std::atoi(argv[1]);
22     const std::size_t P = std::atoi(argv[2]);
23     std::vector<double> x(N), y(N);
24     std::generate(x.begin(), x.end(), drand48);
25     std::generate(y.begin(), y.end(), drand48);
26
27     std::cout << parallel_sp(x,y,P,0,N) << std::endl;
28 }
```

Paralleles Skalarprodukt mit Future

```
1  #include <future>
2  #include <vector>
3  #include <random>
4  #include <iostream>
5  #include <algorithm>
6
7  int main(int argc, char** argv)
8  {
9      const std::size_t N = std::atoi(argv[1]);
10     const std::size_t L = std::atoi(argv[2]);
11     const std::size_t P = 1<<L;
12     std::vector<double> x(N), y(N);
13     std::generate(x.begin(), x.end(), drand48);
14     std::generate(y.begin(), y.end(), drand48);
15
16     // trigger computations
17     std::vector<std::future<double>> part_sum(P);
18     using It = std::vector<double>::iterator;
19     for (std::size_t p=0; p<P; p++)
20         part_sum[p] = std::async(std::launch::async, std::inner_product<It,It,double>,
21             x.begin()+(p*N/P),x.begin()+std::min((p+1)*N/P,N),y.begin()+(p*N/P),0.0);
22     // parallel sum
23     for (std::size_t l=0; l<L; l++)
24     {
25         std::size_t shift = 1<<l;
26         auto Sum = [](std::future<double> a, std::future<double> b) { return a.get() + b.get(); };
27         for (std::size_t p=0; p<P; p+=2*shift)
28             part_sum[p] = std::async(std::launch::async, Sum,
29                 std::move(part_sum[p]), std::move(part_sum[p+shift]));
30     }
31     std::cout << part_sum[0].get() << std::endl;
32 }
```