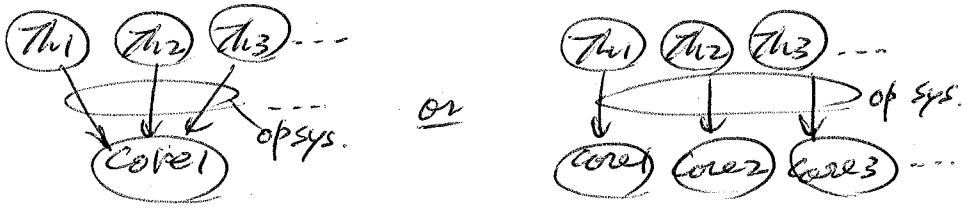
Ch4. C++11 Multithreading

- posix threads (pthreads) — C/C++ based library, Linux/Unix environment
- C++11 threading API — portable (platform-independent code via C++) — Linux/Unix, Windows
  - modern dialects of C++

---

vs.
- multiprocessing — parallelize program over multiple compute units (cores)
- multithreading — shares HW resources (cache, RAM) to avoid idling of unused resources

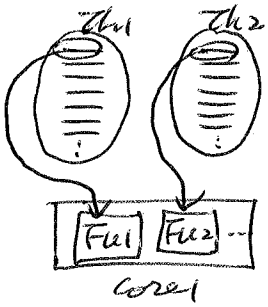
not mutually exclusive



~~ref~~ HW thread (ex) SMT

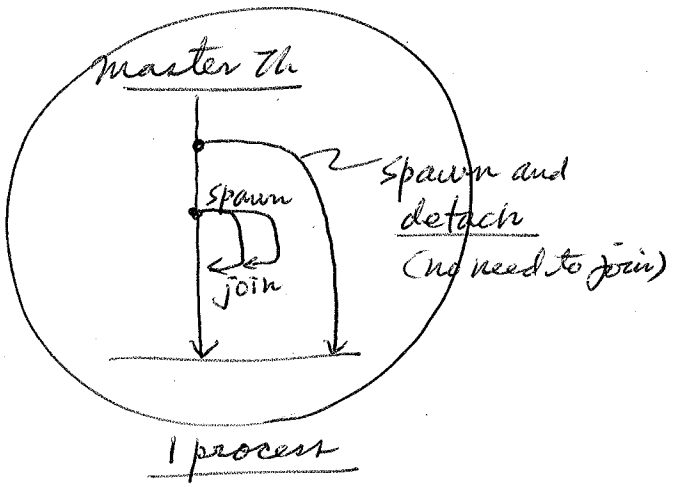
- Conceptual view

- implementation

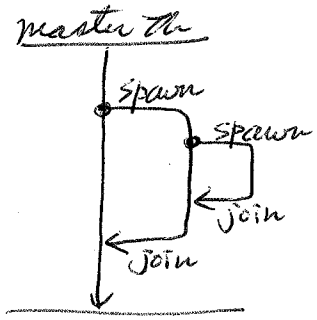


sw thread — component of a prog.

— spawn/join threads



or,



all threads in a process share resources of the process (code/data/file)

Adv: (thread spawning — low cost
Thread comm. — shared-mem. based

disadv: (threads can spy each other (∵ sharing resources)
⇒ security problem.

[# of sw threads > # of cores] — over subscription

currently running threads

vs [join — wait until finish
detach — master thread kills child thread during termination (without waiting)

vs [thread based parallelization — pThread, OpenMP
light-weight shared-mem. mechanism in a single process
process based parallelization — MPI
(independent system processes (distributed-mem.)
heavy weight comm. — channels (Sockets)

— all threads have to be joined or detached within the scope of their declaration, ∄ implicit join

$> g++ -O2 -std=c++11 -pthread p1.cpp ↵
(-O2: code optimize; -std=c++11: C++11)

— Multithreaded Hello program (Listing 4.1) — C++11 version (thread)

```
#include <thread>
int main()
{ ⋮
  vector<thread> threads;        // threads: any name
  ⋮
  for (id=0 ~ num_threads-1)
    threads.emplace_back(slave_func, id);
        // emplace_back: enter a thread to the vector; slave_func: any name; id: th#id
    //or, threads.push_back(thread(slave_func, id));
  ⋮
  for (auto& thread: threads)
    thread.join();
}
```

Assume

```
void slave_func(int id)
{ ...
}
```

ex) threads.emplace_back(A, b, c, d);
(A: func name; b, c, d: 3 args)

```
void A(int b, int c, int d)
{ ...
}
```

↑ vector way thread creation

↓ dynamic array way thread creation

```
⋮
thread* threads = new thread[num_threads];   //needs to delete heap data
for (int id=0 ~ num_threads-1)
  threads[id] = thread(slave_func, id);
⋮
for (int id=0 ~ num_threads-1)
  threads[id].join();
```

— also, stack array is OK

```
thread threads[num_threads];
⋮
```

§4.2 Handling return values (from slave func.)

ex) Fibo#

0, 1, 1, 2, 3, 5, 8, 13, ...

n=0 n=1 n=2 n=3 ...

$a_n = a_{n-1} + a_{n-2}$ (nth Fibo#)

algo

```
double Fibo (int n)
{ double a0 = 0;
  double a1 = 1;
  for (int index=0; indx < n ; index++)
  { double temp = a0;
    a0 = a1;
    a1 += temp;
  }
  return a0;
}
```

(0) a_0 ← ② (1) a_1
① ↓ temp ↗ ③ ($a_1 = a_1 + temp$)

---

1. traditional way — ref. para. way (using pointer)
2. promise/future way
3. packaged task way
4. Asynchronous way.

(Thread slave func can't return value.)

(1) traditional way (ref. para). (pass address)

```
void Fibo ( int n, double* result)     // n: id
{
  ==
  *result = a0;  // content of result pointer.
}

int main(--)
{
  vector <double> results (num_threads, 0);
  for(int id=0 ~ num_threads -1)
    threads.emplace_back( Fibo, id, &(results[id]));
  
}
```

vector init. (size = num_threads, val = 0.)

0, 0, ..., 0 } num_threads

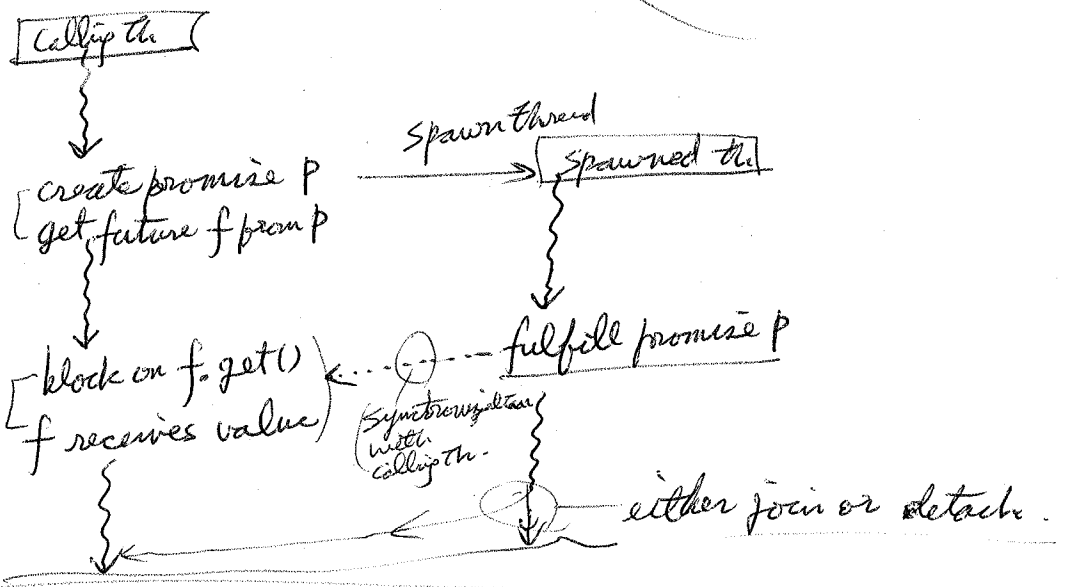
join here

```
for (auto& result : results)
  cout << result << endl;
```

(2) promise/future way

- future fulfills promise
- a kind of synchronization mechanism between master and spawned threads.

object S = (P, f)
- writable view of S (promise)
- readable view of S (future)




```
#include<future>          // for promise/future
void Fibo(int n, promise<double>&& result)   // promise name: P
{
    result.set_value(a0);   // fulfill promise
}
int main(---)
{
    vector<future<double>> results;   // storage for futures
    for(int id=0 ~ num_threads-1)
    {
        promise<double> promise;   // define a promise
        results.emplace_back(promise.get_future());   // assign associated future
        threads.emplace_back(Fibo, id, move(promise));   // move promise to spawned th.
    }
    for(auto& result: results)
        cout << result.get() << endl;   // if received value blocks until receive.
}
```

join

(3) Packaged task way — passing return values using packaged_task (Listing 4.6).
— good for slave func. that communicate one or more values to master thread.

② Simple way — (without using make_task factory)

```
#include<future>
...
double Fibo(int n)
{
  ...
  return a0;
}

int main(...)
{
  ...
  for(id=0~num_threads-1)
  { packaged_task<double(int)> task(Fibo); //create task
    auto future = task.get_future(); // get future
    thread thread(move(task), id); // a thread is created to execute task.
    thread.detach(); //or, join()
    Cout << future.get() << endl; // display output
  }
}
```

or

Using vectors for threads and return values (to store futures)

```
int main(...)
{ ...
  vector<thread> threads;
  vector<future<double>> results;
  for(id=0~num_threads-1)
  { packaged_task<double(int)> task(Fibo);
    results.emplace_back(task.get_future());
    threads.emplace_back(move(task), id);
  }
  for(auto& result: results) Cout << result.get() << endl;
  for(auto& thread: threads) thread.detach(); //or join()
}
```

or

or, using dynamic arrays for threads and return values (to store futures)

```
- int main(---)
{ :
  thread* threads = new thread[num_threads];
  future<double>* results = new future<double>[num_threads];

  for(id=0~num_threads-1)
  { packaged_task<double(int)> task(Fibo);
    results[id] = task.get_future();
    threads[id] = thread(move(task), id);
  }

  for(id=0~num_threads-1)
    cout<< results[id].get()<<endl;

  for(id=0~num_threads-1)
    threads[id].detach(); // or, join()

  delete[] threads;
  delete[] results;
}
```

or,

```
double* results = new double
                  [num_threads];
for(id=0~num_threads-1)
{ packaged_task<double(int)>
                  task(Fibo);
  auto future = task.get_future();
  threads[id] = thread(move(task), id);
  results[id] = future.get();
}
```

double type