

# Multithreading C++11.

## 1-р хэсэг

F.CS306 ПАРАЛЛЕЛ ПРОГРАММЧЛАЛ – Лекц 2

Г.ГАНБАТ

[ganbatg@must.edu.mn](mailto:ganbatg@must.edu.mn)

# Хичээлийн агуулга

- ❖ C++ хэлний орчин үеийн C++11 болон C++14 хувилбараар multithread-тэй програм бичиж сурах
- ❖ Multithreading ба Multiprocessing
- ❖ Thread хуулах, Thread-үүд нэгтэх
- ❖ Thread-үүдийг зогсооход анхаарах зүйлс
- ❖ Буцаах утгыг удирдах

# Multithreading ба Multiprocessing

- ❖ **Multithreading:** Ашиглагдаагүй нөөцийг сүл зогсооос зайлсхийх зорилгоор нэг эсвэл олон цөмийн кэш, RAM зэрэг техник хангамжийн нөөцийг хамтран эзэмших.
- ❖ **Multiprocessing:** Тооцооллыг хурдасгахын тулд янз бүрийн CPU цөмийн хувьд регистр, арифметик логик үйлдлийн нэгж (ALUs) гэх мэт нөөцийг гүйцэт ашиглах зорилгоор олон тооцоолох нэгж (Жны: CPU цөмүүд) дээр програмыг зэрэгцээгээр ажиллуулах.
  - ❖ Дундын нөөцийн хэрэглээг бүрэн хязгаарладаггүй, том хэмжээний процессуудад сокет болон MPI ашиглан хэрэгжүүлдэг

# Multithreading

- ❖ Thread-үүд нь дундын санах ойн хөнгөн механизм
  - ❖ Thread-үүд нь системийн нэг процесийн санах ойн хэсэгт параллел ажилладгаараа бие даасан системийн процессуудаас (хуваарилагдсан санах ой болон сокет зэрэг хүнд сувагчлалтай мэдээлэл солилцоон дээр ажилладаг) ялгаатай
- ❖ Бүх thread эцэг процесийн нөөцийг дундаа хэрэглэнэ.
  - ❖ *Давуу тал:* Дундын регистр болон дундын массив ашигласнаар thread хоорондын мэдээлэл солилцоо хоцролт багатай, хөнгөн байна
  - ❖ *Сул тал:* Нэг thread нь нөгөө thread-ийн өгөгдлийг хялбар тандах положмтой

# Thread хуулах (spawning)

- ❖ Программ хангамжийн thread нь системийн процесийн мастер thread-ээр дурын тооны хуулбарт хувилагдана.
- ❖ Хувилагдсан хуулбарын доторх thread-үүдийг рекурсив байдлаар хувилах боломжтой
- ❖ Параллел ажиллах thread-уудын тоог системийн физик цөмүүдийн хэмжээтэй тэнцвэржүүлэх барих шаардлагатай.
  - ❖ **Oversubscription:** Thread-ийн тоо нь их байвал үйлдлийн систем өртөг ихтэй орчин сэлгэлт (context switch) хэрэглэн дарааллуулан ажиллуулна.

# Thread-ҮҮДИЙГ ТӨГСГӨХ

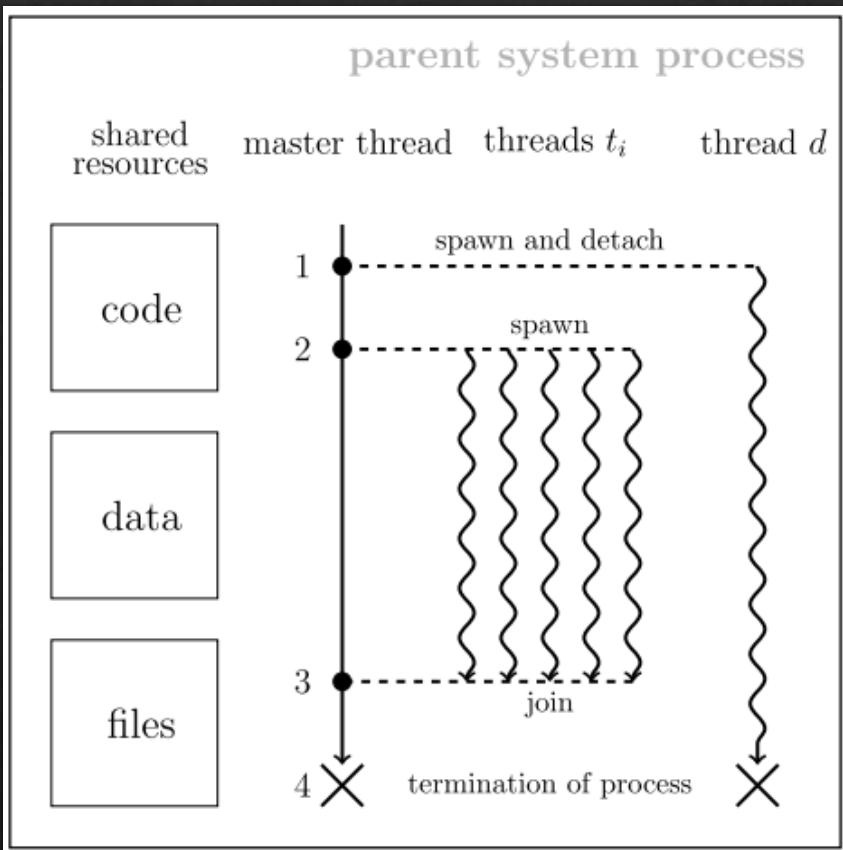
Мастер thread-ийн зааврын дараалал нь хуулбар thread-үүдийн ажлын гүйцэтгэлээс үл хамааран үндсэн функцийн төгсгөл хүртэл үргэлжилдэг. Олон хуулбарыг зогсоох аргууд:

1. Бүх хуулбар thread ажиллаж дуустал мастер thread дотор хүлээнэ. Энэ хүлээлт нэгтэлт (joining) хийснээр дуусна.
2. Нэг thread-ийг тусгаарлана. Мастер процесс дуусахдаа хуулбаруудыг хүлээлгүйгээр тасалдаг.
  - ❖ Муу тохиолдол: дутуу бичигдсэн файл эсвэл сүлжээгээр илгээсэн гүйцэд биш мессежүүдтэй програм дуусч болно
  - ❖ Ямар нэгэн синхрончлолгүй тусгаарлагдсан thread-ийн жишээ программын ажиллагааг хянаж, лог файлд бичилт хийдэг thread

# Thread-үүдийг төгсгөхөд анхаарах зүйлс

- ❖ Thread бүр нэг удаа л нэгтгэх эсвэл тусгаарлах боломжтой
- ❖ Тусгаарлагдсан thread нэгтгэх боломжгүй, эсвэл эсрэгээрээ
- ❖ Нэгтгэсэн эсвэл тусгаарласан thread-ыг дахин ашиглах боломжгүй.
- ❖ Тухайн түвшний бүх thread-ийг нэгтгэсэн эсвэл тусгаарласан байна.

# Multithread програмын жишээ



1.  $d$  thread-ыг хувилаад тусгаарлана. Энэ thread өөрөө дуусах эсвэл программ дуусах хүртэл ажиллана.
2. Таван  $t_i$  thread өгөгдлийг параллелиар боловсруулна.
3. Мастер thread тэдгээрийг нэгтгэж дуустал хүлээнэ.
4. Мастер болон бүх тусгаарласан thread дууссанаар программ ажиллаж дуусна.

# Multithreaded program

```
void say_hello(uint64_t id) {
    std::cout << "Hello from thread: " << id << std::endl;
}
int main(int argc, char * argv[]) {
    const uint64_t num_threads = 4;
    std::vector<std::thread> threads;
    for (uint64_t id = 0; id < num_threads; id++)
        threads.emplace_back(say_hello, id);
    for (auto& thread: threads)
        thread.join();
}
```

- ❖ Эхлээд thread-ийн заагчид зориулж стандарт сангийн *std::thread* объектууд агуулсан *std::vector* төрлөөр санах ой нөөцлөнө
- ❖ *id* аргументтайгаар *say\_hello* методыг ажиллуулдаг *num\_threads* ширхэг thread-д хувилан vector threads-д хадгалж байна.
- ❖ vector threads-ын *push\_back* функцээр thread объектуудыг шилжүүлэх боломжтой: *threads.push\_back(std::thread(say\_hello, id));*
- ❖ Дараа нь хандахад уян хатан байх тул нэгтгэх шатанд thread-ийн заагчдийг хадгалах хэрэгтэй

# Compile

```
g++ -O2 -std=c++11 -pthread hello_world.cpp -o hello_world
```

- ❖ *-O2*
  - ❖ код стандарт оновчлолыг идэвхжүүлнэ.
- ❖ *-std=c++11*
  - ❖ нь C++11 дэмжих боломжтой
- ❖ *-pthread*
  - ❖ *PThreads* санд суурилсан multithreading-д боломжийг нэмнэ

Үр дүн:

```
Hello from thread: 3  
Hello from thread: 1  
Hello from thread: 0  
Hello from thread: 2
```

# БУЦААХ УТГЫГ УДИРДАХ

Thread-үүд дурын аргументтай функцийг ажиллуулаад утгууд буцаана. Харин Thread объект нь буцаах утганд шууд хандахыг дэмждэггүй. Ямар нэг эргэх холбоогүйгээр олон thread ачаалах үеийн fire-and-forget сценарын хувьд уламжлалт код хязгаарлагдмал юм. Тиймээс дараах 3 аргыг хэрэгжүүлдэг:

- ❖ Уламжлалт арга
- ❖ Promises, Futures хэрэглэх арга
- ❖ Асинхрон арга

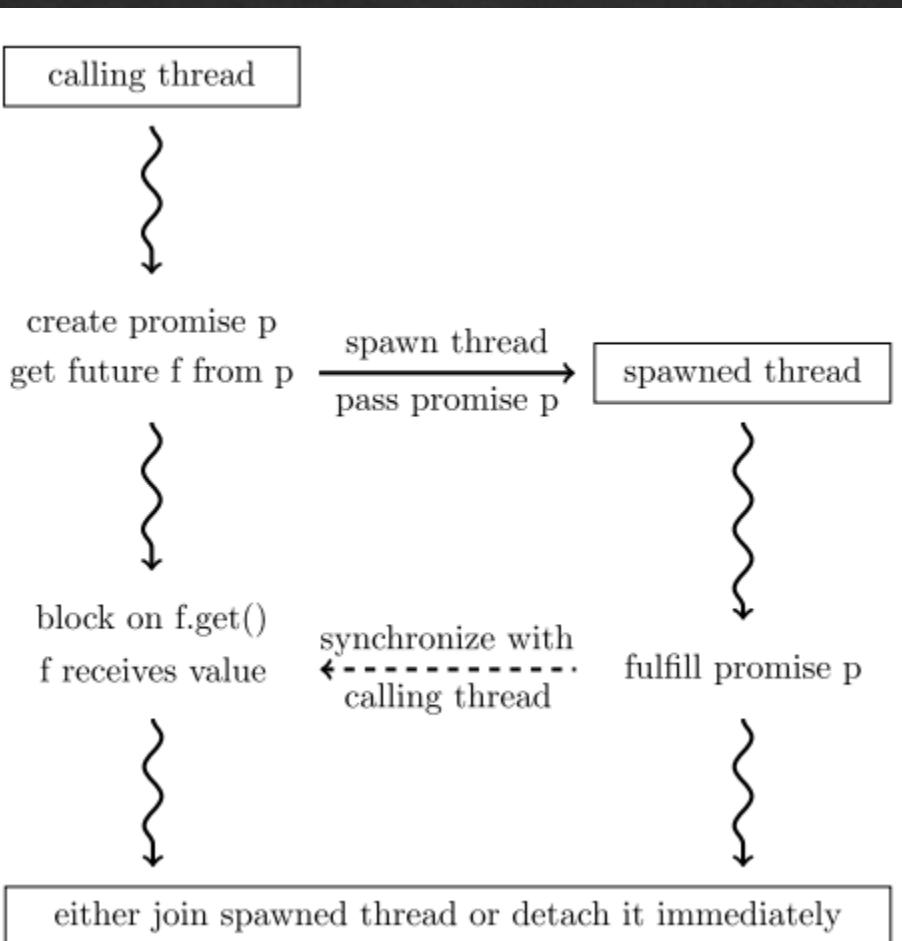
# Уламжлалт арга

- ❖ Бүх thread санах ойг multithreading сценаарын дундаа ашигладаг
  - ❖ Заагчийг үр дүнгийн утганд холбох
  - ❖ Тооцоолсон утгыг холбогдох санах ойд бичих
- ❖ Thread боломжит чөлөөлөгдсөн санах ойд ажиллах тул сегментчлэлийн алдаа үүсгэх боломжтой
- ❖ Уралдааны нөхцөл (*race condition*)-д орохгүйн тулд thread ажиллах явцад үр дүнгийн нь утгуудыг мастер thread-тэй холбоотой байлгах шаардлагатай.

# Promises, Futures хэрэглэх арга

- ❖ Асинхрон ажиллагааны шинж чанарыг хангасан C++-ын буцаах утгын механизм.
- ❖  $s = (p, f)$  харилцан холбоотой хослолыг тодорхойлно.
  - ❖  $p$ : promise, зөвхөн мэдээлэл солилцох шатанд 1 удаа утга олгоно
  - ❖  $f$ : future, promise-оор мэдээлэл солилцсоны дараа хандана
- ❖ Мастер thread болон хуулбар thread хоёрын хооронд синхрончлох механизм болгон хэрэглэх  $p, f$ -ын хамааралыг байгуулна.
- ❖ Promise хэзээ ч ажиллахгүй бол future-ийг унших гэж оролсоноор түгжрэл (*deadlock*) үүсэх боломжтой

# $p, f$ -ын хамааралыг хэрэгжүүлэх



1. Т тусгай төрөлтэйгөөр `std::promise<T> p;` гэж зарлаад дараа нь түүнтэй уялдуулж `std::future f = p.get_future();` хуваарилан `s = (p, f)`-г үүсгэнэ.
2. Дуудах функцын *rvalue*-аар `std::promise && p` гэж `p`-г дамжуулна. Үүний тулд `std::move()` хэрэглэн мастер thread-ээс хуулбар thread-рүү `p` шилжсэн байна.
3. Хуулбар thread-ийн их бие дотор `p.set_value(some_value)`-аар `p` утга олгоно.
4. Төгсгөлд нь мастер thread дотор `f` ийг `f.get()` хэрэглэн уншина. Мастер thrad `f`д `p` мэдээлэл солилцтол энэ ажиллагааг зогсоодог.

# Асинхрон арга

- ❖ *std::async* команд нь хуваарилагдсан thread-ийн болон мастер thread-ийн аль нэгийг хэрэглэн даалгаврыг (task) биелүүлдэг.
- ❖ Даалгавар үүсгэх: *auto future = std::async(fibo, id);*
- ❖ Анхаарах асуудлууд:
  1. *std::async* функц дуудсанаар thread хуулбарлагдсан гэсэн үг биш. Ажиллагаа нь thread дуудалтаар хэрэгжинэ.
  2. *future.get()* гэж харгалзах future-д хандахгүй бол даалгаврын ажиллагаа saatаж болно.
  3. Харгалзах future-ийн хүрээнд анхаарахгүй бол ялгаатай даалгавруудын ажиллагаа дараалж ажиллах болно.

Баярлалаа.