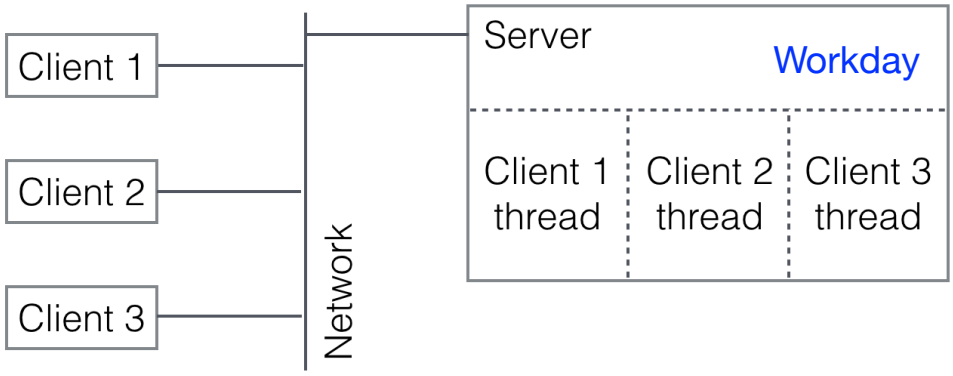
Concurrent Programming (I)

Overview

- A concurrent program is a program designed to have *two or more* execution contexts. Such a program is said to be multithreaded, since more than once execution context can be *active simultaneously*.
- Although the programs used in this course so far are all single-threaded, multithread programming is widely used in daily life.

- Do you know any examples of concurrent programming?
  - One typical example is the networks. Take Workday Student for instance, what we use on a web browser is the front end of Workday system. We call it the client side. There is a database system on the other side of the network managing all kinds of information, such as students' information, faculty's information, courses' information, and etc. We all the database system the server side, which is the back end of the Workday system.
  - Every time a student logs into Workday, a new thread is created for the student. So that the server side must be able to handle multithreads.
  - Please note that all these threads share the same database systems. A database consists of a number of tables. A table is actually a piece of memory. It is very possible that several users want to access the same piece of memory at the same time: read the information or modify the information stored at that piece of memory (the remaining seats of a course).
  - How to make those uses can access to the same piece of memory successfully and get the correct information. These make concurrent programming challenging but interesting.



- Two fundamental problems:
  - Race condition
  - Deadlock

Race Condition

- **Race condition** is one of the **fundamental problems** that can occurs while **executing** different threads **asynchronously**.
- A race condition occurs when the meaning of a program depends upon the order in which each thread accesses a shared variable.
- Take post-increment for example. **Post-increment** is **not atomic**. It actually takes three machine instructions:
  1. fetch the value from memory
  2. calculate the new value
  3. write the new value back to the memory

- Suppose there are two threads, A and B, executing the post-increment statement simultaneously, and suppose the initial value of i is 0.
- **What are the possible results we can get?**
  - If A and B execute the three steps in the second order, the value of i is 2 after finishing executing.
  - All other possible orders lead to the value of i is 2.

`i++`

1. fetch i value, v, from memory
2. calculate new value v' based on v
3. store v' back to the memory allocated to i

thread A, B

`i = 0`

A(1) B(1) A(2) A(3) B(2) B(3)	A(1) A(2) A(3) B(1) B(2) B(3)	A(1) A(2) B(1) B(2) A(3) B(3)
`i = 1`	`i = 2`	`i = 1`

Dinning Philosophers

- It is **unrealistic and fanciful**. But the **synchronization behavior** that **it models can happen in real system**.
- The **description** of this problem is that:
  - A collection of **N philosophers** sits at a round table, where N > 1.
  - **N chopsticks** are placed on the table, one between each pair of adjacent philosophers.

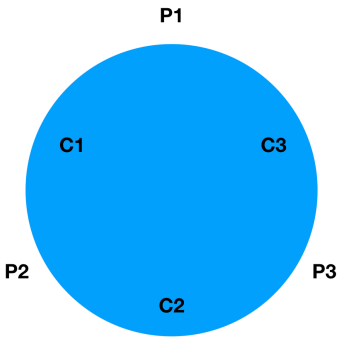
  - No philosopher can eat unless he has **two chopsticks**.
  - If a philosopher doesn't eat, the he/she is back to sleep.
- Obviously, adjacent philosophers cannot eat at the same time. Each philosopher alternately eats and sleeps, waiting when necessary for the requisite chopsticks before eating.
- Our task is to **write code simulating** the dining philosophers so that no philosopher starves.
- An obvious protocol that each philosopher might follow is:

```
while (true) {
  grab left chopstick;
  grab right chopstick;
  eat;
  release left chopstick;
  release right chopstick;
  sleep;
}
```

- **What is the problem of this program? How could the problem happen?**
  - This code leads to a deadlock.
  - Now assume that action of the philosophers are perfectly interleaved: the first philosopher grabs his left chopstick, then the second philosopher grabs his left chopstick, and so in until the Nth philosopher grabs his left chopstick. Then the first philosopher tries to grab his right chopstick, the second philosopher tries to grab his right chopstick, and so on. they all have to wait because no right chopstick is available and they all starve.

Deadlocks

- A deadlock occurs when a ring of threads comes to a point in the program where *each needs a resource from the next thread in the ring* in order to continue. No thread can provide the resource because each thread is waiting for another thread.
- **What are the possible solutions to this problem?**
  - Theoretical computer scientists have proven that there is **no deterministic uniform solution** to this problem. (By **uniform**, we mean that every philosopher **executes exactly the same code with no access to identifying state information** such as the name of the "current" philosopher.) But many non-uniform solutions exists.
  - For example, we could **number the philosophers**. **Even numbered** philosophers **ask for the left chopstick first, odd numbered** ones **ask for the right chopstick first**.
  - Another common solution to this sort of deadlock is to **order the resources** (in this case chopsticks) and **force the processes** (philosophers) **to grab chopsticks in ascending order**. This solution is very general and is



- widely used in practice.
  - Consider the case where we have 3 philosophers: P1, P2, P3. Then we order the chopsticks C1, C2, C3, as shown in the figure on the right.
  - Now, no matter what, all the philosophers will be able to eat. For instance, if P2 gets C1, and P3 gets C2, P1 must wait until C1 is free (grabbing in ascending order). So P3 will get C3 (since there will be no contention), and finish eating. This will release C2 and C3, allowing P2 to get C2 and finish eating. Finally, this will release C1, allowing P1 to get C1 and C3 (since there will be no further contention) and finish eating.

Threads

- In a multithreaded program, **all threads execute the same piece of code**. They **share the heap**, but each thread **has its own stack frame**.
- **main() function** comprise a **single, default thread**. **Threads other than the default one** can be **created by programmers**.

Threads in C

- To create a multithreaded C program, you need to know these basic:
  - First, include the header file pthread.h
  - Second, a worker function should be defined. A worker function is a C routine that the thread will execute once it is created.
  - Third, the thread attribute should be initialized. You can use NULL for the default values. (attr: scope, joinable, size and etc.)
  - Now you can create a thread by using the pthread_create function.
  - After create threads, you can use pthread_join to suspend execution of the calling thread until the target thread terminates.
  - Or you can use pthread_exit to terminates the calling thread.
  - When compiling, remember to include the pthread library. You may no need this on Mac. If you use other machines, double check if the library path has been set already.

- **pthread** (POSIX thread): threads that use the POSIX standard programming interface
  - #include <pthread.h>
  - Define a worker function: a C routine that the thread will execute once it is created
    - void *foo (void *args) () {}
  - Initialize pthread_attr_t: you can use NULL for the default values
    - pthread_attr_t attr;
    - pthread_attr_init (attr);

- Create a thread
  - pthread_t thread;
  - pthread_create (&thread, &attr, foo, arg);
- Thread management
  - pthread_join (thread, status); //suspend execution of the calling thread until the target thread terminates
  - pthread_exit (status); // terminates the calling thread
- Compiling: using -pthread

- Show the `helloThreads.c`, and run the code. Comment out the join for loop, run the program again, and ask students why the results are different. [pthread_join will suspends the calling thread (main thread) until the target thread terminates.]

```c
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 5

typedef struct {
     int id;
} threadInfo;

void *hello_thread(void *threadinfo) {
     threadInfo *ti = (threadInfo *) threadinfo;
     printf("Thread %d saying Hello!\n", ti->id);
     pthread_exit(NULL);
}

int main () {
     int i;
     threadInfo ti[NUM_THREADS];
     pthread_t thread[NUM_THREADS];

     // Set up the parameters for each thread
     for (i = 0; i < NUM_THREADS; i++)
       ti[i].id = i;

     // Get the threads going
     for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&(thread[i]), NULL, hello_thread, &(ti[i]));

     // Join up with them. This will wait until they are done.
     for (i = 0; i < NUM_THREADS; i++)
       pthread_join(thread[i], NULL);

     return 0;
}
```



- with join for loop, the output is several “saying Hello!”
- without join for loop, the output is empty.