

## Initial CS3210 Questions

0 surveys completed

0 surveys underway

## Have you taken CS2106 (Operating Systems)?

Yes, already completed it

Taking it this semester

Haven't taken yet

Are you familiar with using SSH (either in the command line or using your editor)

Yes

Not too familiar, but I can figure it out

Not too familiar, and I will need help

## How is your familiarity with C++?

Enough knowledge to be workable

Dabbled in C++ a bit

I know C but never touched C++

I can barely use C

I really never used C

## How do you edit code remotely?

vim

emacs

nano

VSCode

Other editor(s) not mentioned here

I have never done this

## What OS/distro are you using (and how do you do UNIX-y things if not UNIX?)

Windows + WSL

Windows + PuTTY SSH

Windows + (non-PuTTY) SSH

Mac OS

Ubuntu / Debian / Other Debia

Fedora

Arch

NixOS

Others

# CS3210 Lab 1

Processes, Threads, and Synchronization Basics

# About myself

Theodore Leebrant

(Theodore/Theo is good!)

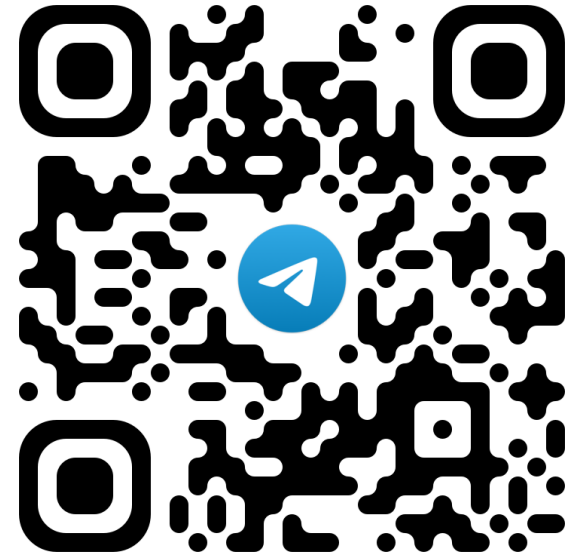
- Computer Science + Mathematics
  - was a PL nerd, mostly with Rust
- Plays too much Final Fantasy XIV
- Out-of-tutorial communication:
  - Email: [theo@comp.nus.edu.sg](mailto:theo@comp.nus.edu.sg) for consults, questions
  - Telegram: details later
  - Will reply messages within 24 hours – *except 2 days before deadlines*



# Quick admin stuff

## Telegram group:

- Things that are **OK**
  - Banter
  - Interesting finds
  - Lab cluster issues / requests
- Things that are **not OK**
  - Code debugging help  
*(unless you spot something funny in our code)*



Slides will be uploaded after every session

Anonymous feedback: [bit.ly/feedback-theodore](https://bit.ly/feedback-theodore)

Why are we here?

Make things go **fast**

(latency)

Make *things* finish *quickly*

(throughput)

Make things go fast **efficiently**

(energy efficiency)

# CS3210 from 30,000 feet in the sky

## Part 1: OpenMP

Making single-node  
CPU programs faster

Complex tasks that are  
relatively less parallelizable (10s  
of tasks)

## Part 2: CUDA

Making single-node  
GPU-able programs faster

Simpler tasks that are relatively  
more parallelizable  
(millions of tasks even)

## Part 3: MPI

Making multi-node  
programs faster

Large, mixed workloads  
(slow communication between  
nodes is worth it due to size /  
parallelizability of work)

The complexity: how to **harness** this power effectively?

# Our goals for the semester

(not just “finish the tutorials & labs” and “pass your exams”)

## 1. Learn how to decompose problems into tasks in your sleep

sometimes a curse: cannot unsee this :)

### Task parallelism

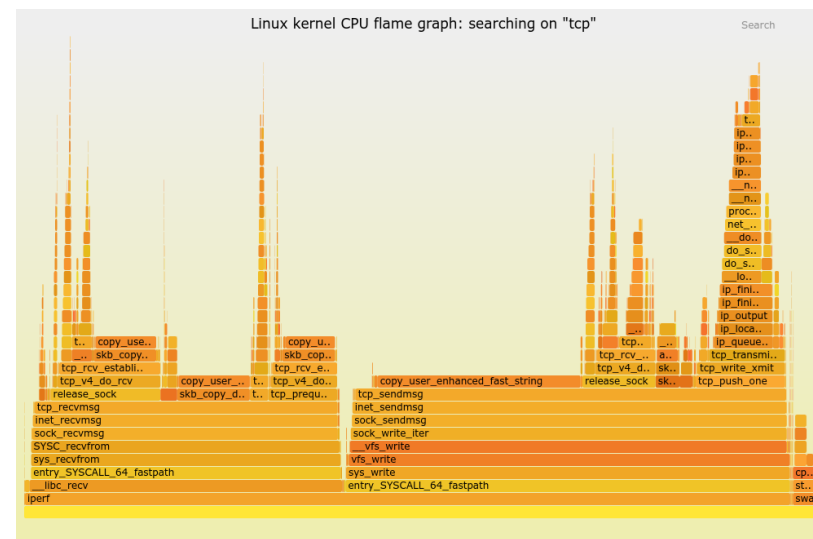
```
/*
scheduler table - all regular tasks apart from the fast_loop()
should be listed here, along with how often they should be called
(in 10ms units) and the maximum time they are expected to take (in
microseconds)
*/
static const AP_Scheduler::Task scheduler_tasks[] PROGMEM = {
  { update_GPS,          2,    900 },
  { update_nav_mode,     1,    400 },
  { medium_loop,         2,    700 },
  { update_altitude,    10,   1000 },
  { fifty_hz_loop,       2,    950 },
```

### Data parallelism

id	sensor_id	sensor_name	sensor_type	sensor_unit	value	value	value	value
204	84807596188785	1660016385729 TMD4910 lux Sensor	com.samsung.sensor.auxlightness	UNKNOWN	14047	0	0	0
200	84807596188337	16600163857191 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ACCEL_RAW	0.244208574295044	6.12706136703491	7.23797082901001	
202	84807596188337	16600163857201 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ALTAZ	49.729115354346	90.567788093882		
201	84807596188337	16600163857201 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ORIENTATION	-1.6127279966719	-0.70218771996067	-0.033727128058672	
224	84807596188337	16600163857261 LSM6DSO Gyroscope Sensor	android.sensor.gyroscope	GYRO_RAW	-0.25579829669699	0.032373657234001	0.0713185146451	
248	84807596193485	1660016385730 Samsung Rotation Vector	android.sensor.rotation_vector	ALTAZ_FUSED	52.7830496227928	101.471646724973		
247	84807596193485	1660016385730 Samsung Rotation Vector	android.sensor.rotation_vector	ORIENTATION_FUSED	-1.62077558040619	-0.634387910365058	-0.150321334600449	
234	84807596193485	1660016385727 AK09918C Magnetic field Sensor	android.sensor.magnetic_field	ALTAZ	95.317206992184	96.0243293692428		
223	84807596193485	1660016385727 AK09918C Magnetic field Sensor	android.sensor.magnetic_field	ORIENTATION	-1.62665622444153	-0.59929156970978	-0.091517142951489	
203	84807596188337	1660016385721 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ACCEL_FUSED	0.34943301818138	6.18244776257958	7.43219143812054	
204	84807596188303	1660016385721 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ALTAZ	-1.61715567111969	-0.693443238735199	-0.040890499949455	
205	84807596188303	1660016385721 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ORIENTATION	50.2110428530815	91.0062054824369		
225	84807596188303	1660016385726 LSM6DSO Gyroscope Sensor	android.sensor.gyroscope	GYRO_RAW	-0.252745590147644	0.0378277787292	0.12935072183609	
245	84807596488889	1660016385729 TMD4910 RGB Sensor	com.samsung.sensor.light_csi	UNKNOWN	14122	1835	250	0
206	84807596188303	1660016385721 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ACCEL_RAW	0.474051922559739	6.18891635131836	7.74314737319946	
207	84807596188303	1660016385722 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ORIENTATION	-1.6262217180014	-0.673229923898479	-0.081145804822445	
208	84807596188303	1660016385722 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ALTAZ	51.292698308014	92.2542586287167		
226	84807596188303	1660016385726 LSM6DSO Gyroscope Sensor	android.sensor.gyroscope	GYRO_RAW	-0.233197808265686	0.050701815634966	0.189215511083603	
250	84807596193485	1660016385731 Samsung Rotation Vector	android.sensor.rotation_vector	ALTAZ_FUSED	52.9222010667535	101.450220421547		
249	84807596193485	1660016385731 Samsung Rotation Vector	android.sensor.rotation_vector	ORIENTATION_FUSED	-1.62165248394012	-0.631919980048133	-0.150133371353149	
236	84807596193485	1660016385728 AK09918C Magnetic field Sensor	android.sensor.magnetic_field	ORIENTATION	-1.63426288999874	-0.59929156970978	-0.091517142951489	
235	84807596193485	1660016385727 AK09918C Magnetic field Sensor	android.sensor.magnetic_field	MAG_RAW	33.9599990844727	-0.359999954502792	0.1799999251396	
210	84807601188336	1660016385723 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ORIENTATION	-1.63649773597717	-0.65500069659119	-0.074081301689148	
227	84807601188336	1660016385726 LSM6DSO Gyroscope Sensor	android.sensor.gyroscope	GYRO_RAW	-0.200211077928543	0.050000049965426	0.246636837720871	
211	84807601188336	1660016385723 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ALTAZ	52.76194948158	93.1811904311883		
209	84807601188336	1660016385722 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ACCEL_RAW	0.58657842024231	6.08875417709351	7.90355620791626	
228	84807601188369	1660016385726 LSM6DSO Gyroscope Sensor	android.sensor.gyroscope	GYRO_RAW	-0.156263116459847	0.026878070086241	0.30032985343933	
213	84807606188369	1660016385723 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ORIENTATION	-1.63680815696716	-0.638454258441825	-0.070046444476551	
214	84807606188369	1660016385723 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ALTAZ	53.1967268000038	93.5029595202064		
252	84807606193485	1660016385730 Samsung Rotation Vector	android.sensor.rotation_vector	ALTAZ_FUSED	51.0406115852027	101.395852115753		
251	84807606193485	1660016385730 Samsung Rotation Vector	android.sensor.rotation_vector	ORIENTATION_FUSED	-1.6240058374969	-0.62954580783844	-0.150560408830643	
238	84807606193485	1660016385728 AK09918C Magnetic field Sensor	android.sensor.magnetic_field	MAG_RAW	33.9599990844727	-0.19999997317791	0.599999942427213	
239	84807606193485	1660016385728 AK09918C Magnetic field Sensor	android.sensor.magnetic_field	ORIENTATION	-1.63456681404114	-0.59929156970978	-0.091517142951489	
240	84807606193485	1660016385728 AK09918C Magnetic field Sensor	android.sensor.magnetic_field	ALTAZ	55.317206992184	96.536695959064		
246	84807607488712	1660016385729 TMD4910 RGB IR Sensor	com.samsung.sensor.light_ir	UNKNOWN	763	1411	981	591
215	84807611188402	1660016385724 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ACCEL_RAW	0.584185242652993	5.73441215230934	7.8125796319542	
216	84807611188402	1660016385724 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ORIENTATION	-1.63601102974402	-0.63361120084412	-0.074368021625324	
217	84807611188402	1660016385724 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	ALTAZ	53.5802317366783	93.5016014353228		
229	84807611188402	1660016385726 LSM6DSO Gyroscope Sensor	android.sensor.gyroscope	GYRO_RAW	-0.12078680700233	0	0.352316528583731	
268	84807615457883	1660016385740 TMD4910 Unbalanced lux Sensor	android.sensor.light	LUX	14125			
7401	84807616158407	16600163857361 LSM6DSO Acceleration Sensor	android.sensor.accelerometer	GYRO_RAW	-1.06720101418111	-0.0790000142436031	0.967000071947008	

(not just “finish the tutorials & labs” and “pass your exams”)

Is it the algorithm? Disk? Cache? Network?  
Not enough work? Too much work? Synchronization overhead?  
**Theory alone is useless here.**

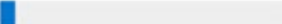


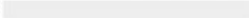
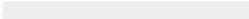
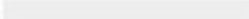
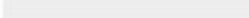
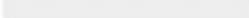
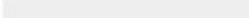
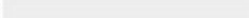
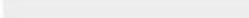
# Our goals for the semester

(not just “finish the tutorials & labs” and “pass your exams”)

## 3. Have fun!

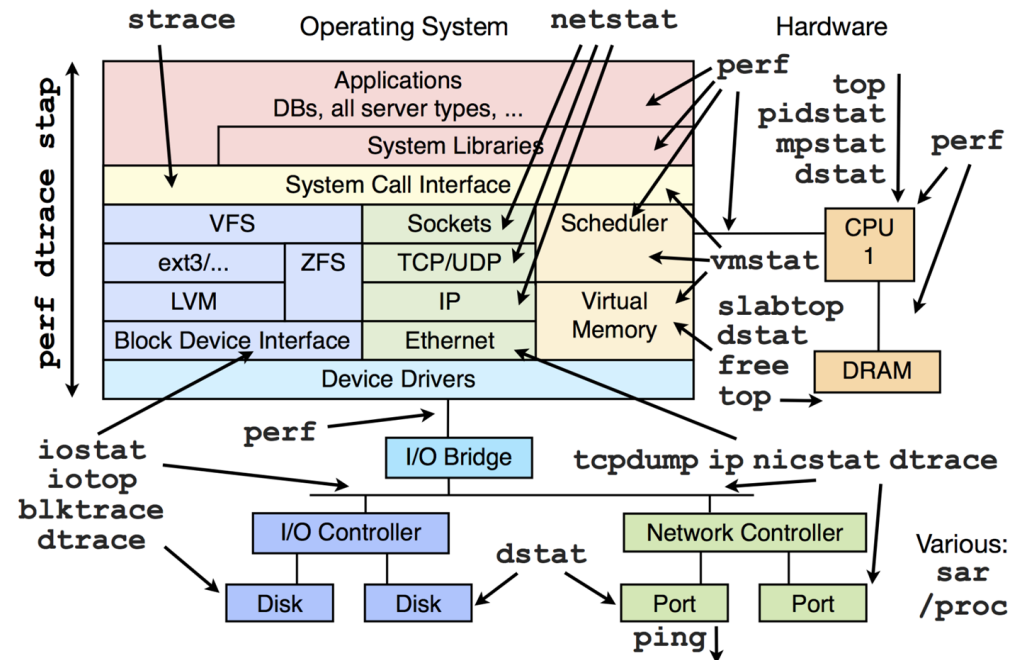
Very satisfying to make a program faster. You will hopefully understand soon :)

```
INFO: Pandarallel will run on 1 workers.  
INFO: Pandarallel will use Memory file system to transfer data between  
OK, rows are already sorted by timestamp_ns, no need to sort again.  
Creating subsets of dataframe to use later  
Processing closest values for lux  
5.44%  2825 / 51902
```

```
INFO: Pandarallel will run on 8 workers.  
INFO: Pandarallel will use Memory file system to transfer data between the main process and workers.  
OK, rows are already sorted by timestamp_ns, no need to sort again.  
Creating subsets of dataframe to use later  
Processing closest values for lux  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6487  
0.00%  0 / 6487
```

# No such thing as stupid questions: **ever**

- “Systems” are made up of many, many parts
- Infinite learning process
- Please feel free to bring anything up
- Personal promise: all questions will be treated equally



# General Lab Workflow

- For you:
  - Labs/Tutorials are for you to learn. **No stress.**  
Only a small percentage of grade. Please experiment!
  - Feel free to talk, communicate, **work together!**  
**Submit separately, state who you worked with.**
  - First tutorial slot challenges...
- Me:
  - I will probably not give direct answers, but I will guide :)
  - Will be hovering around, try not to be conscious of it, trying to help
  - Might interrupt with solutions and interesting behaviours / questions
  - Will start at :05 and finish in ~1h, but will stay back for questions



# CS3210 Lab 01

Processes / Threads / Synchronization

# Things to note for labs

- ‘C-style’ C++ programming language
  - C++ *only*! Even if you want to write C...
  - [Lab 1] Use only the pthread library.
    - (i.e. don't use `std::mutex`, `std::thread`, ...)
- **Do the lab exercises in order!**
- Submit the necessary exercises
  - Lab 1 deadline: next week, 2 Sept, **1400hrs**



# Why synchronization for lab 1?

(because anyone can solve embarrassingly parallel problems.)

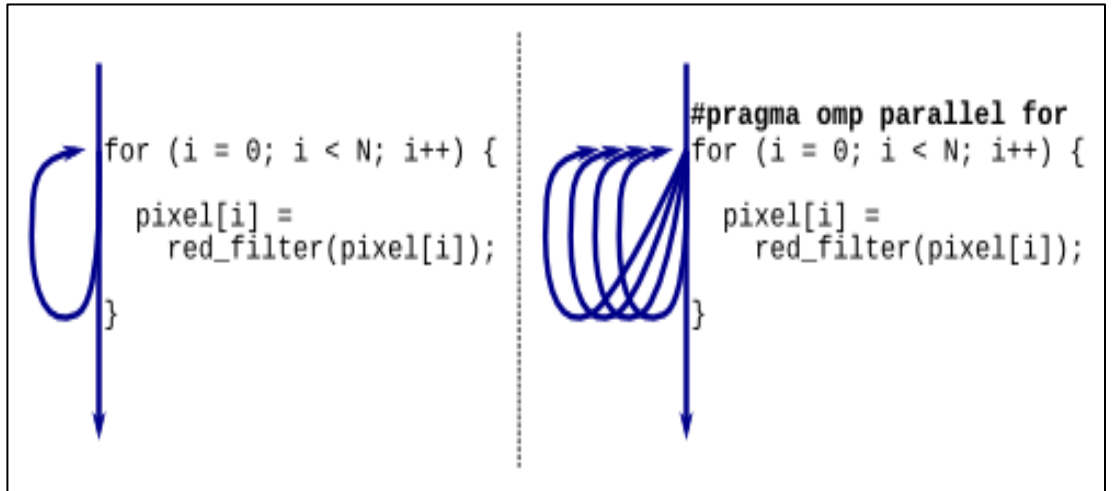
Python

```
multi.py  X
C: > Users > Administrator.SHAREPOINTSKY > multi.py > ...
1  from multiprocessing import Pool
2  def num(n):
3      return n*4
4  if __name__ == '__main__':
5      numbers=[3,6,9]
6      pool=Pool(processes=1)
7      print(pool.map(num,numbers))
```

C/C++

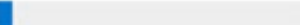
```
for (i = 0; i < N; i++) {
    pixel[i] =
        red_filter(pixel[i]);
}

#pragma omp parallel for
for (i = 0; i < N; i++) {
    pixel[i] =
        red_filter(pixel[i]);
}
```

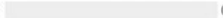
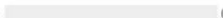



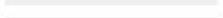

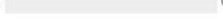


# Why synchronization for lab 1?

(because anyone can solve embarrassingly parallel problems.)

```
INFO: Pandarallel will run on 1 workers.  
INFO: Pandarallel will use Memory file system to transfer data between 1  
OK, rows are already sorted by timestamp_ns, no need to sort again.  
Creating subsets of dataframe to use later  
Processing closest values for lux  
5.44%  2825 / 51902
```

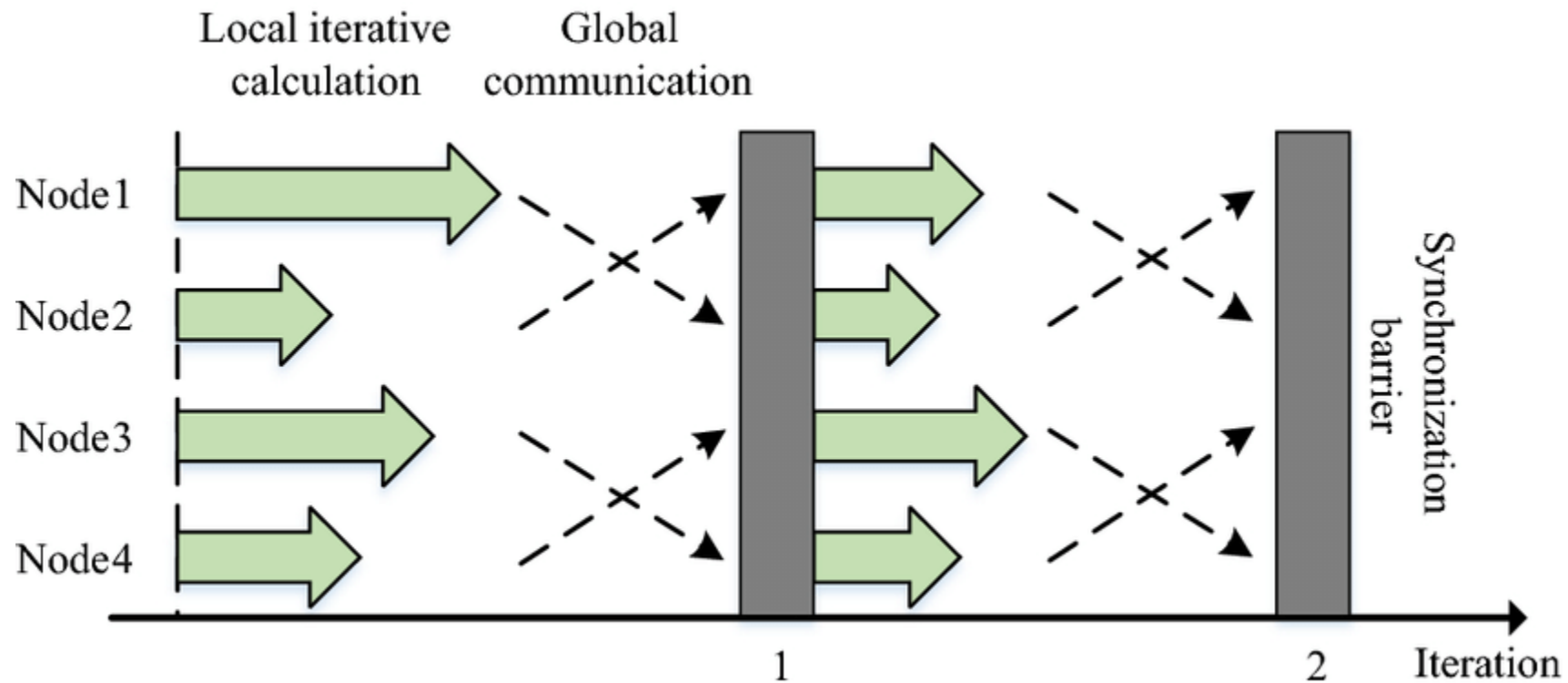
```
new_cols = df_filtered.apply(  
    get_closest_impt_values_v2,  
    axis=1,  
    result_type="expand",  
)
```

```
INFO: Pandarallel will run on 8 workers.  
INFO: Pandarallel will use Memory file system to transfer data between the main process and workers.  
OK, rows are already sorted by timestamp_ns, no need to sort again.  
Creating subsets of dataframe to use later  
Processing closest values for lux  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6488  
0.00%  0 / 6487  
0.00%  0 / 6487
```

```
new_cols = df_filtered.parallel_apply(  
    get_closest_impt_values_v2,  
    axis=1,  
    result_type="expand",  
)
```

# Why synchronization for lab 1?

Anyone can solve embarrassingly parallel problems;  
the difficult part is coordinating between tasks, a.k.a. synchronization!



# Lab 1 Tasks

- Connect to lab machines, download code
- ex1: processes with fork/wait
- ex2: pthreads ordering, counter variable
- Bonus: Semaphore usage
- ex3: race condition
- ex4: pthread joining
- ex5: pthread mutexes
- ex6: condition variables (starts to get challenging)
- ex7: producer-consumer: pthreads (for submission)
- ex8: producer-consumer: processes and semaphores (for submission)
- ex9: explaining ex7 and ex8 (for submission)
- Visiting the PDC lab downstairs

# Let's get started!

Ask for help if you can't SSH!

For advanced users: generate SSH key, copy your key to the pdc machines, set up your SSH config file...

[nus-cs3210.github.io/student-guide/accessing](https://nus-cs3210.github.io/student-guide/accessing)

1. Get SoC ID
2. Check email for PDC Lab username & password
3. Get SoC VPN (not for *right now*, but for accesses outside SoC)
4. SSH to `soctf-pdc-xxx.d1.comp.nus.edu.sg`
  - xxx is 001-003 or 009-011; don't crowd 001 :')
5. Download lab code into the machines and start the lab :D
  - `wget https://www.comp.nus.edu.sg/~srirams/cs3210/L1_code.zip`
  - `unzip L1_code.zip`

**For VSCode users, SSH into the node with bash/powershell first to change your password before using VSCode to connect!**

# [Extra] Quality-of-life for SSH

- Consider adding the following to `~/.ssh/config`

```
Host pdc-003
  HostName soctf-pdc-003.d1.comp.nus.edu.sg
  User [insert your lab id here]
  ForwardAgent yes
```

Now you can just type  
`ssh pdc-003` to connect :)

- Generate your SSH key:  
`ssh-keygen -t ed25519`
  - Add your public key (.pub) to [pdc.comp.nus.edu.sg/accounts/profile/](https://pdc.comp.nus.edu.sg/accounts/profile/)
  - Modify your config:

```
Host pdc-003
  HostName soctf-pdc-003.d1.comp.nus.edu.sg
  User [insert your lab id here]
  ForwardAgent yes
  IdentityFile ~/.ssh/id_ed25519
```



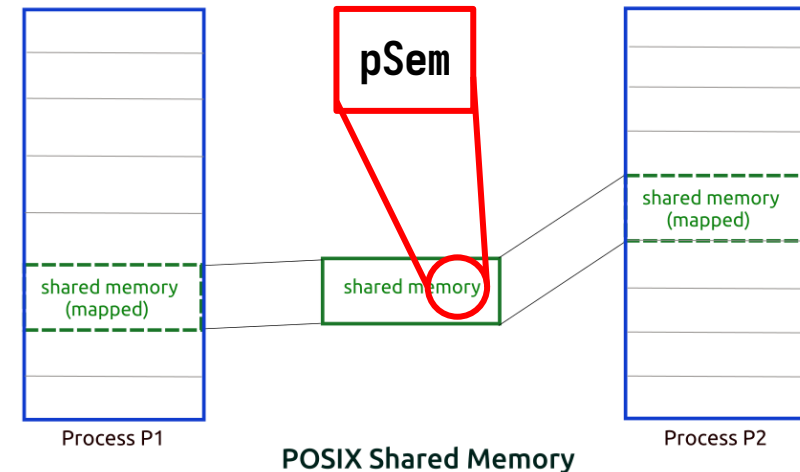
# Named vs. Unnamed Semaphores

- Named semaphores (created with `sem_open`) are "automatically shared between processes" - how?
  - Processes duplicate their memory spaces on `fork()`
- Parent process executes:  

```
sem_t* sem = sem_open("pSem", O_CREAT | O_EXCL, 0644, value);
```
- So, shouldn't the semaphore be private to each process?

# Named vs. Unnamed Semaphores

- Shouldn't the semaphore be private to each process?
- OS auto-maps semaphore into shared memory for us!
  - It even saves it as a virtual file
- Try:
  - `./semaph`
  - `Ctrl+Z` after entering sem value
  - `ipcs; ls /dev/shm`
  - `fg; Ctrl+C; ipcs; ls /dev/shm;`
  - `ipcrm -M ..`



# Shared Memory Usage

- You may have allocated things in shared memory within your process and terminated it prematurely before cleanup.
- To resolve:

<code>ipcs -cm</code>	Check shm usage on machine
<code>ipcrm -M &lt;key&gt;</code>	Delete the memory area
<code>ls -la /dev/shm</code>	Check for named semaphores
<code>rm /dev/shm/&lt;sem&gt;</code>	Remove named semaphores

# Binary Semaphores vs. pthread Mutexes

- What is the difference?

# Binary Semaphores vs. pthread Mutexes

- What is the difference?
  - Hint: who can unlock mutexes?

# Condition Variables: Why?

- Semaphores can be too generic to map nicely to all problems
  - Increment / Decrement integer
- Mutex: suffers from similar issues

# Condition Variables: Why?

Can you spot anything wrong with this code?

## Waiting thread

```
while(true) {  
    pthread_mutex_lock(&mutex);  
    if (condition is satisfied) {  
        // Do something when cond satisfied  
        done = true;  
    }  
    pthread_mutex_unlock(&mutex);  
    if (done) break;  
}
```

## Signalling thread

```
pthread_mutex_lock(&mutex);  
  
/* change variable value to make condition  
satisfied */  
  
pthread_mutex_unlock(&mutex);
```

# Condition Variables: Why?

Let's shuffle things a little bit to make it better

## Waiting thread

```
pthread_mutex_lock(&mutex);
if (condition not satisfied) {
    pthread_mutex_unlock(&mutex);
    while (condition not satisfied) { //wait }
    pthread_mutex_lock(&mutex);
}
// Do something, condition satisfied

pthread_mutex_unlock(&mutex);
```

## Signalling thread

```
pthread_mutex_lock(&mutex);

/* change variable value to make condition
satisfied */

pthread_mutex_unlock(&mutex);
```



# Condition Variables: Why?

What happens if there are two waiting threads?

## Waiting thread 1

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

1. Both waiting threads are waiting

## Waiting thread 2

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

## Signalling thread

```
pthread_mutex_lock(&mutex);
/* change variable value to satisfy cond */
pthread_mutex_unlock(&mutex);
```

# Condition Variables: Why?

What happens if there are two waiting threads?

## Waiting thread 1

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

2. Signalling thread  
changes the  
variable value

## Waiting thread 2

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

## Signalling thread

```
pthread_mutex_lock(&mutex);
/* change variable value to satisfy cond */
pthread_mutex_unlock(&mutex);
```

# Condition Variables: Why?

What happens if there are two waiting threads?

## Waiting thread 1

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

3. Both waiting threads exit the busy wait; T1 gets the mutex

## Waiting thread 2

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

## Signalling thread

```
pthread_mutex_lock(&mutex);
/* change variable value to satisfy cond */
pthread_mutex_unlock(&mutex);
```

# Condition Variables: Why?

What happens if there are two waiting threads?

## Waiting thread 1

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

4. T1 does something  
that invalidates the  
condition

## Waiting thread 2

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

## Signalling thread

```
pthread_mutex_lock(&mutex);
/* change variable value to satisfy cond */
pthread_mutex_unlock(&mutex);
```

# Condition Variables: Why?

What happens if there are two waiting threads?

## Waiting thread 1

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

5. T1 releases mutex,  
T2 gets the mutex

## Waiting thread 2

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

## Signalling thread

```
pthread_mutex_lock(&mutex);
/* change variable value to satisfy cond */
pthread_mutex_unlock(&mutex);
```

# Condition Variables: Why?

What happens if there are two waiting threads?

## Waiting thread 1

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

6. T2 gets to the critical section but condition might no longer be satisfied!

## Waiting thread 2

```
1 pthread_mutex_lock(&mutex);
2 if (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

## Signalling thread

```
pthread_mutex_lock(&mutex);
/* change variable value to satisfy cond */
pthread_mutex_unlock(&mutex);
```

# Condition Variables: Why?

Fixing the 'sniping' behaviour

## Waiting thread 1

```
1 pthread_mutex_lock(&mutex);
2 while (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

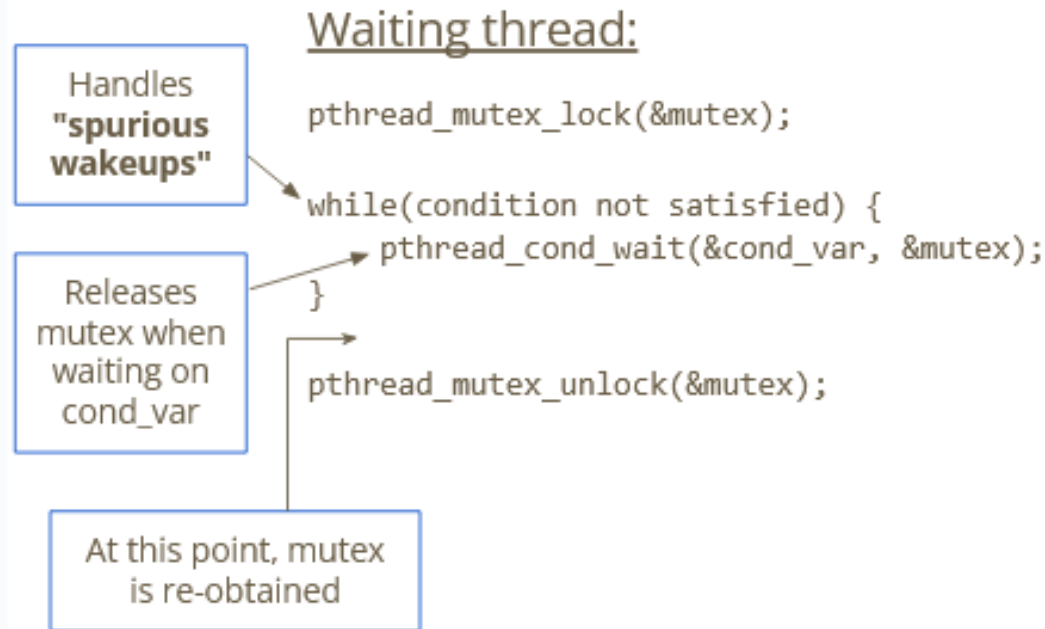
## Waiting thread 2

```
1 pthread_mutex_lock(&mutex);
2 while (condition not satisfied) {
3     pthread_mutex_unlock(&mutex);
4     while (condition not satisfied) { //wait }
5     pthread_mutex_lock(&mutex);
6 }
7 // Do something, condition satisfied
8
9 pthread_mutex_unlock(&mutex);
```

## Signalling thread

```
pthread_mutex_lock(&mutex);
/* change variable value to satisfy cond */
pthread_mutex_unlock(&mutex);
```

# Condition Variables



## Signaling Thread

```
pthread_mutex_lock(&mutex)

/* change variable value */

if (condition satisfied) {
    pthread_cond_signal(&cond_var);
}

pthread_mutex_unlock(&mutex);
```

The diagram shows the code for a signaling thread. It starts with `pthread_mutex_lock(&mutex)`, followed by a comment `/* change variable value */`, then an `if` statement: `if (condition satisfied) { pthread_cond_signal(&cond_var); }`, and finally `pthread_mutex_unlock(&mutex);`. An annotation box at the bottom right has an arrow pointing to the `pthread_cond_signal` call.

Tells the waiting thread to **wake up**, but thread will only wake up once the signaling thread releases mutex as well.



# Condition Variables: Spurious Wakeups

- Why do we not use **if** statements for checking condition in the waiting thread?

## Waiting thread:

```
pthread_mutex_lock(&mutex);

if (condition not satisfied) {
    pthread_cond_wait(&cond_var, &mutex);
}

// do something to maybe make condition invalid

pthread_mutex_unlock(&mutex);
```

## Signaling thread:

```
pthread_mutex_lock(&mutex)

// do something to make condition satisfied

if (condition satisfied) {
    pthread_cond_signal(&cond_var);
}

pthread_mutex_unlock(&mutex);
```

# Condition Variables: Spurious Wakeups

- Spurious wakeup: occurs when a thread wakes up from waiting and finds that the condition is still unsatisfied. (Credit: Wikipedia)
- It can happen due to (but is not limited to):
  - Implementation
    - the thread blocked suddenly wakes up even though no signal or broadcast occurs
  - Race conditions
    - what happens if two threads wake up at the same time?

# Summary

- Connecting to lab machines
- Processes and threads
- Mutexes / Semaphores / Condition Variables

# End of Lab 1

- Telegram group: scan on right
- Slides will be uploaded after every session
- Feedback: [bit.ly/feedback-theodore](https://bit.ly/feedback-theodore)
- Email: [theo@comp.nus.edu.sg](mailto:theo@comp.nus.edu.sg)
- Now: **PDC Lab Visit** (and back again)

