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/distry 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

Theodore's slides; heavily inspired by Sriram's, Richard's, Cristina's and Zhi Heng's slides.

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 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

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	Part 2: CUDA	Part 3: MPI
Making **single-node CPU** programs faster	Making **single-node GPU-able** programs faster	Making **multi-node** programs faster
Complex tasks that are relatively **less parallelizable** (10s of tasks)	**Simpler tasks** that are relatively **more parallelizable** (millions of tasks even)	**Large, mixed workloads** (slow communication between nodes is worth it due to size / parallelizabilty 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*”)

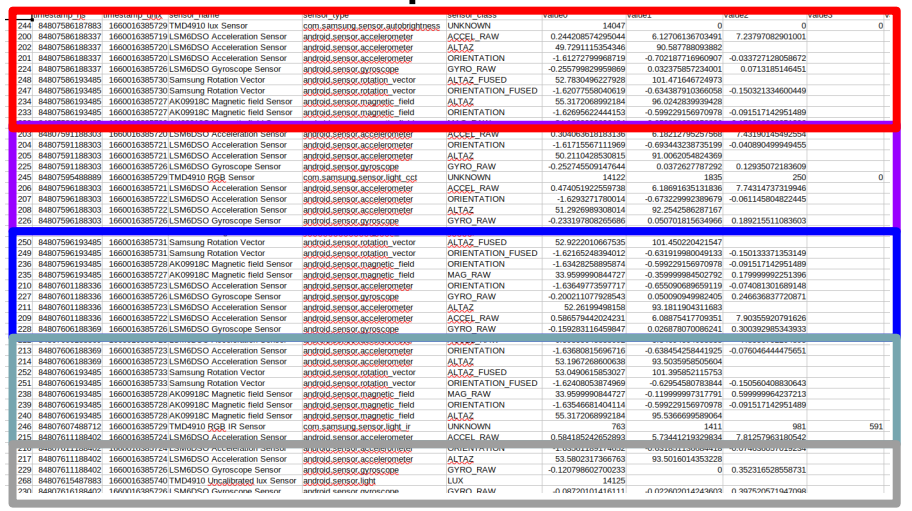
1. Learn how to **decompose problems into tasks** in your sleep

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



Our goals for the semester

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

2. Become **systems detectives**: think *holistically* about performance

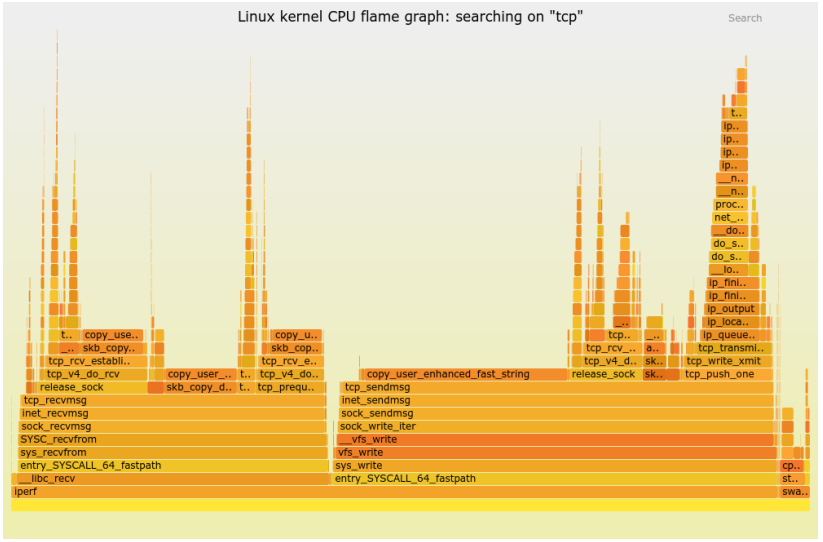
Is it the algorithm? Disk? Cache? Network?
Not enough work? Too much work? Synchronization overhead?

Theory alone is useless here.

```
Performance counter stats for 'dd if=/dev/zero of=test.iso bs=10M count=1':

         21.812281 task-clock                #    0.912 CPUs utilized
                15 context-switches          #    0.001 M/sec
                 2 CPU-migrations            #    0.000 M/sec
             2,805 page-faults               #    0.129 M/sec
        62,025,623 cycles                    #    2.844 GHz                      [51.76%]
         6,299,287 stalled-cycles-frontend   #   10.16% frontend cycles idle     [45.35%]
        24,456,020 stalled-cycles-backend    #   39.43% backend  cycles idle     [45.92%]
        12,655,619 instructions              #    0.20  insns per cycle
                                             #    1.93  stalled cycles per insn [89.86%]
         3,552,630 branches                  #  162.873 M/sec                    [76.11%]
            51,348 branch-misses             #    1.45% of all branches          [60.39%]

       0.023914596 seconds time elapsed
```

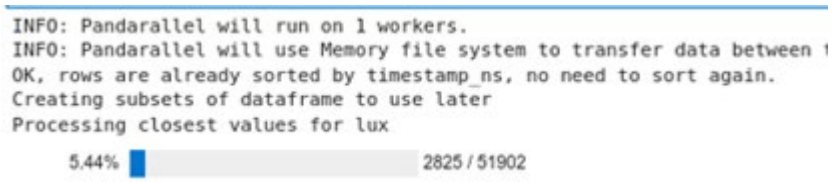


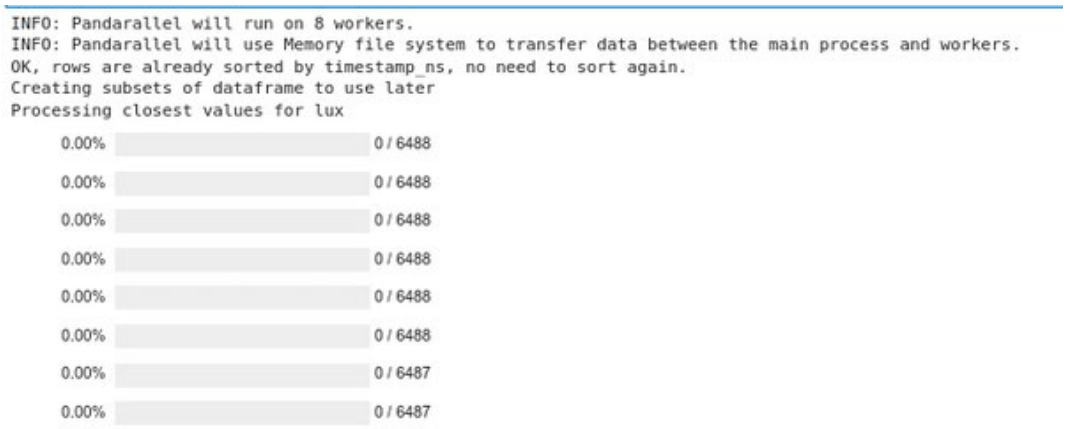
Our goals for the semester

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

3. Have fun!

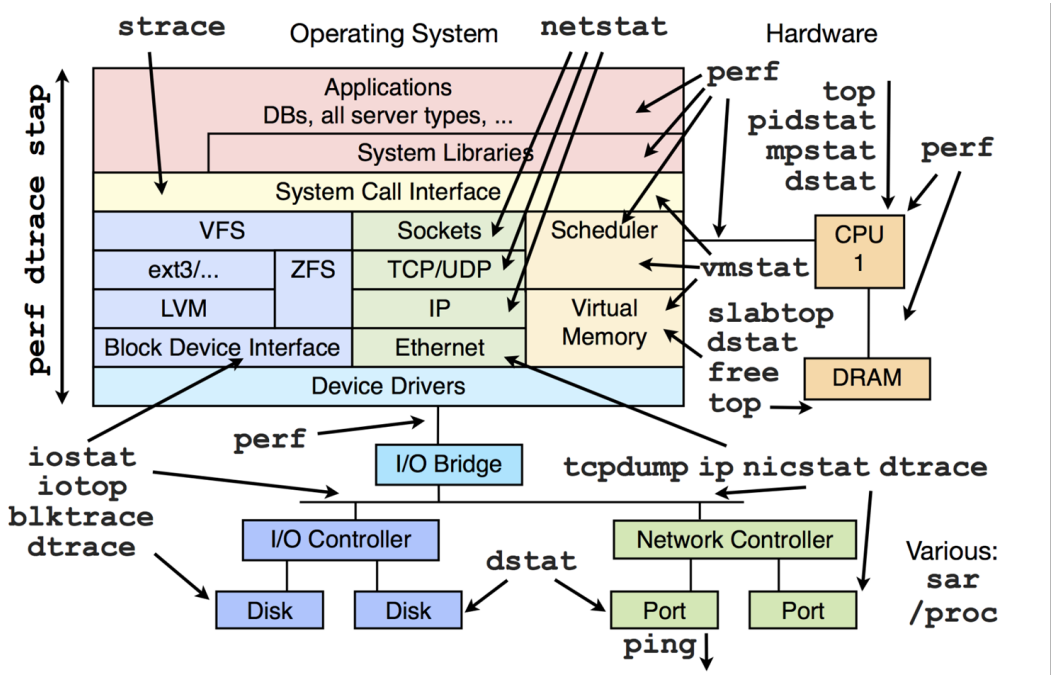
Very satisfying to make a program faster. You will hopefully understand soon :)





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.)

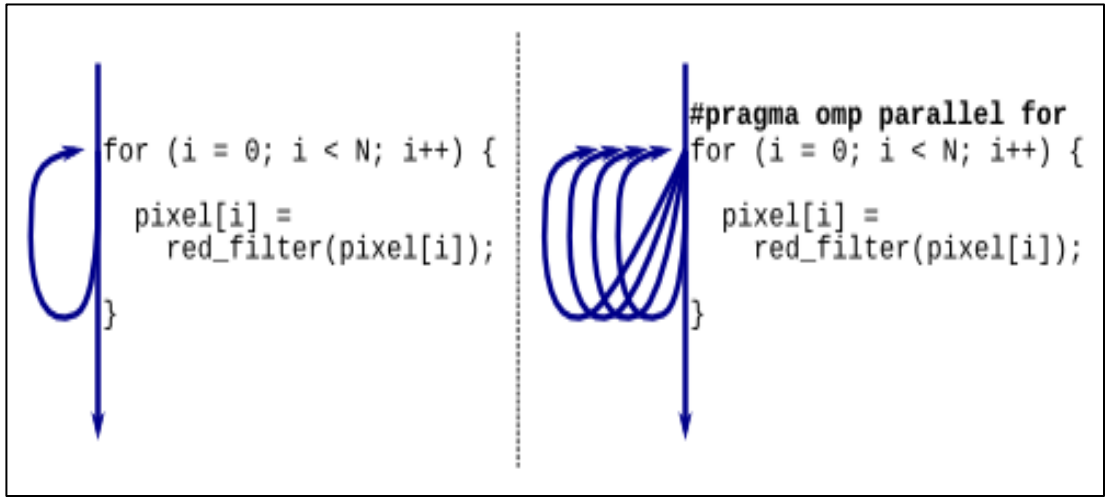
Python

multi.py

C: > Users > Administrator.SHAREPOINTSKY > multi.py > ...

```python
from multiprocessing import Pool
def num(n):
    return n*4
if __name__=='__main__':
    numbers=[3,6,9]
    pool=Pool(processes=1)
    print(pool.map(num,numbers))
```

C/C++



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
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
```

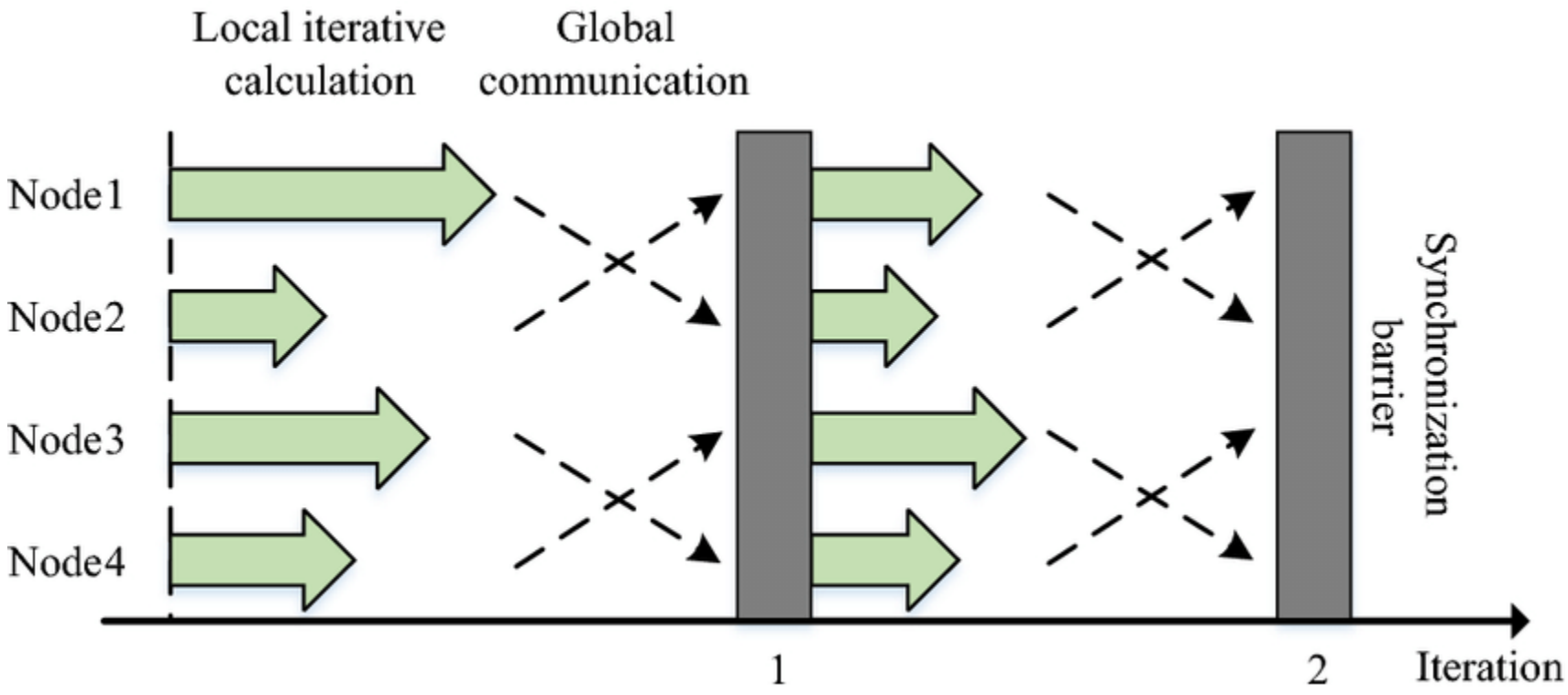
```
INFO: Pandarallel will run on 8 workers.
INFO: Pandarallel will use Memory file system to transfer data between the main process and workers.
OK, rows are already sorted by timestamp_ns, no need to sort again.
Creating subsets of dataframe to use later
Processing closest values for lux
0.00% 0 /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

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

```python
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

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/
  - 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()`

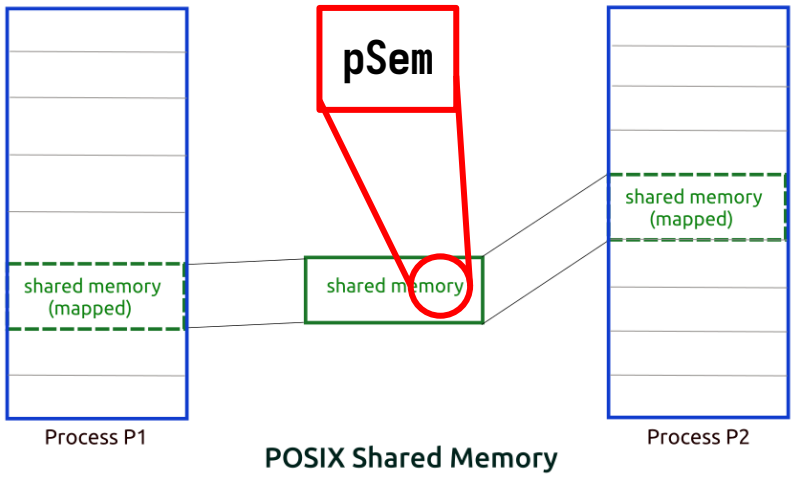
- Parent process *executes*:

```
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:

`ipcs -cm`	Check shm usage on machine
`ipcrm -M <key>`	Delete the memory area
`ls -la /dev/shm`	Check for named semaphores
`rm /dev/shm/<sem>`	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);
```

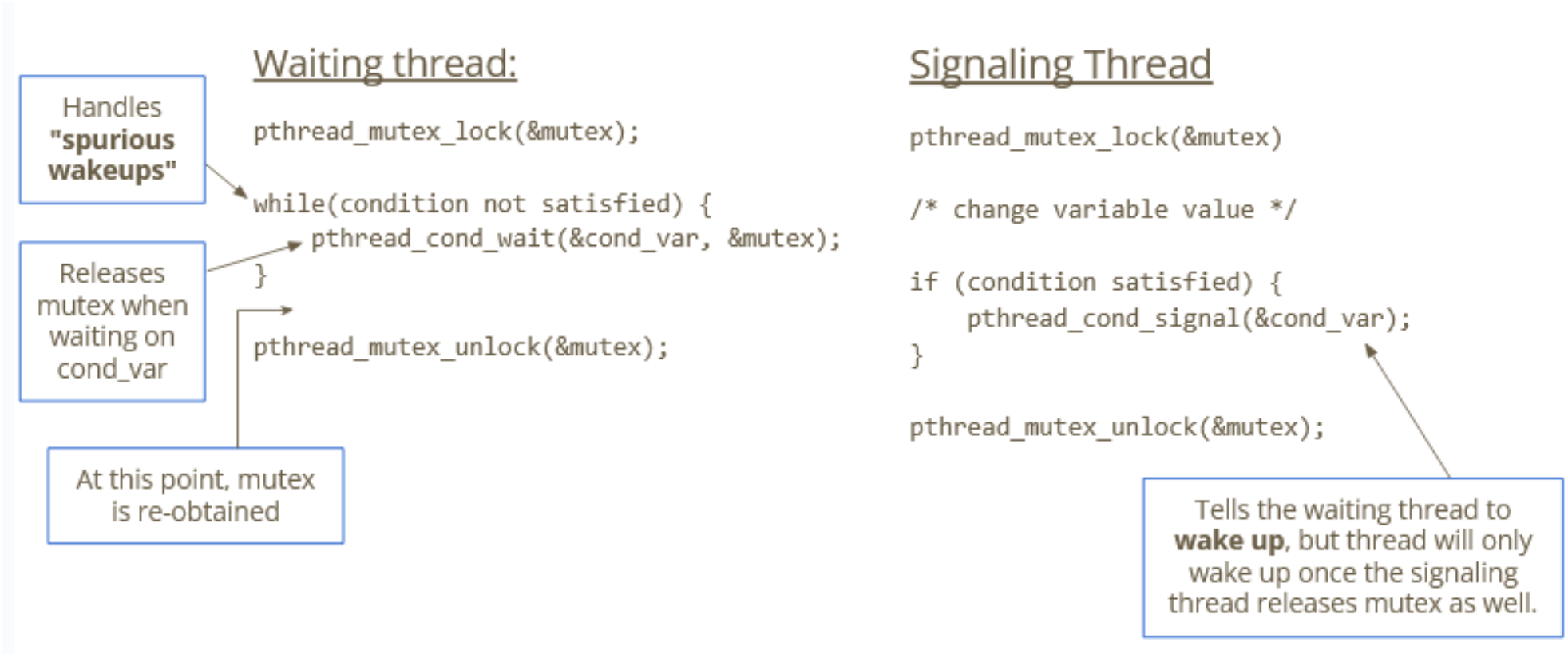
Waiting thread 2

```
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



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
- Email: theo@comp.nus.edu.sg

- Now: PDC Lab Visit (and back again)