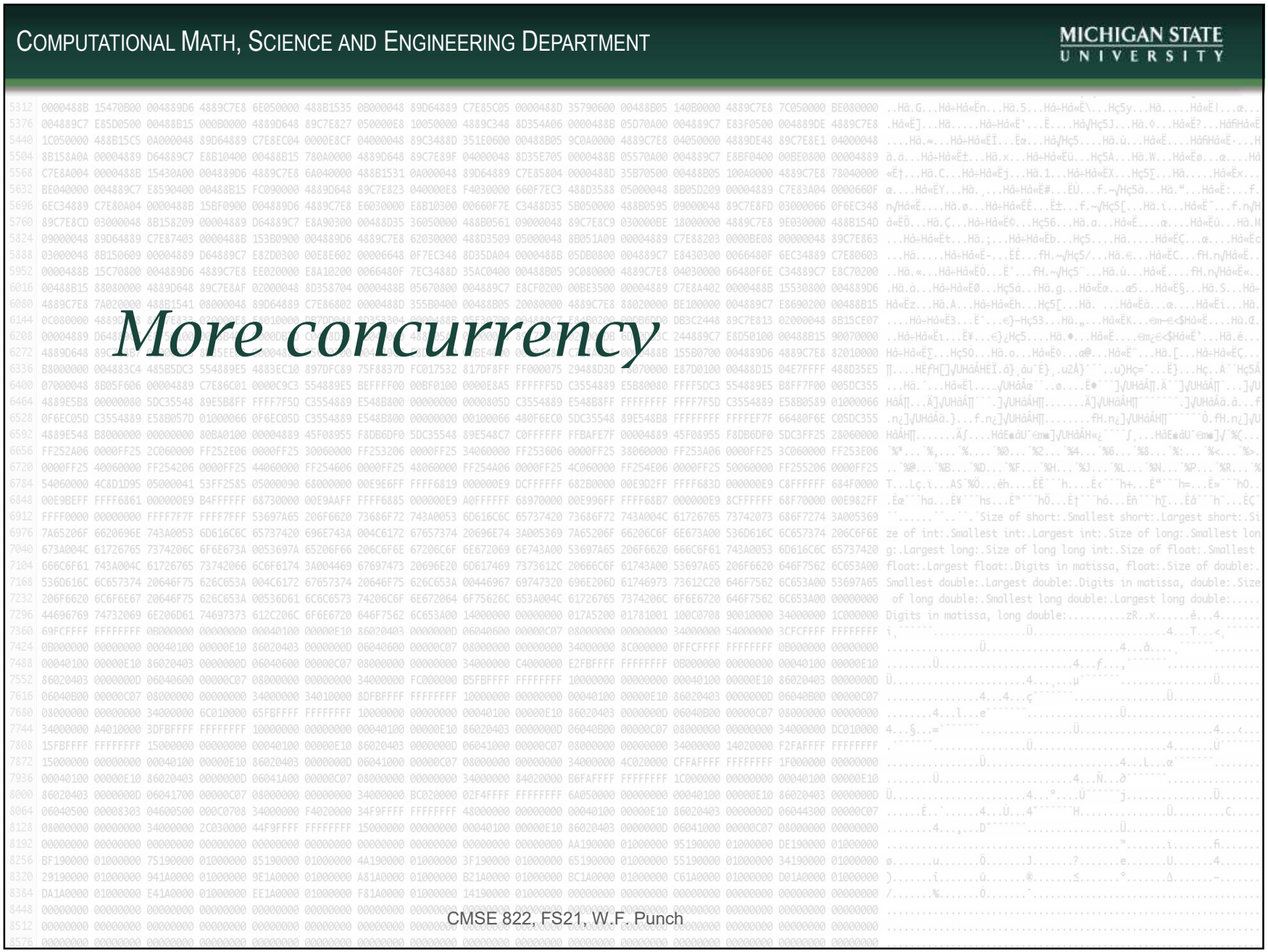


More concurrency

A long list

Because concurrency is such an important topic there are a number of different types (in C++ and elsewhere).

I'll list them here, but just briefly and you can investigate more on your own if you are interested.

mutex variety

- timed_mutex: lock to a timepoint
- recursive_mutex: allow the same thread to recursively call itself
  - for every lock called, an unlock must be called before the mutex is released
  - only the same thread can make the multiple calls. Other threads wait
- recursive_timed_mutex: combination of the two above

more

- shared_mutex: two ways to lock:
  - exclusive (using lock, try_lock). Only 1 thread.
  - shared (using lock_shared, try_lock_shared). More than 1 thread can use it.
- shared_timed_mutex
  - combo of shared and timed.

semaphores

Semaphore (c++-20) has a count of the number of threads that can share access:

- acquire grants access and decrements the count. When the count hits 0, acquire blocks
- release ups the count, unblocks.
- not tied to one thread. The acquire thread can be different from release.

lock variety

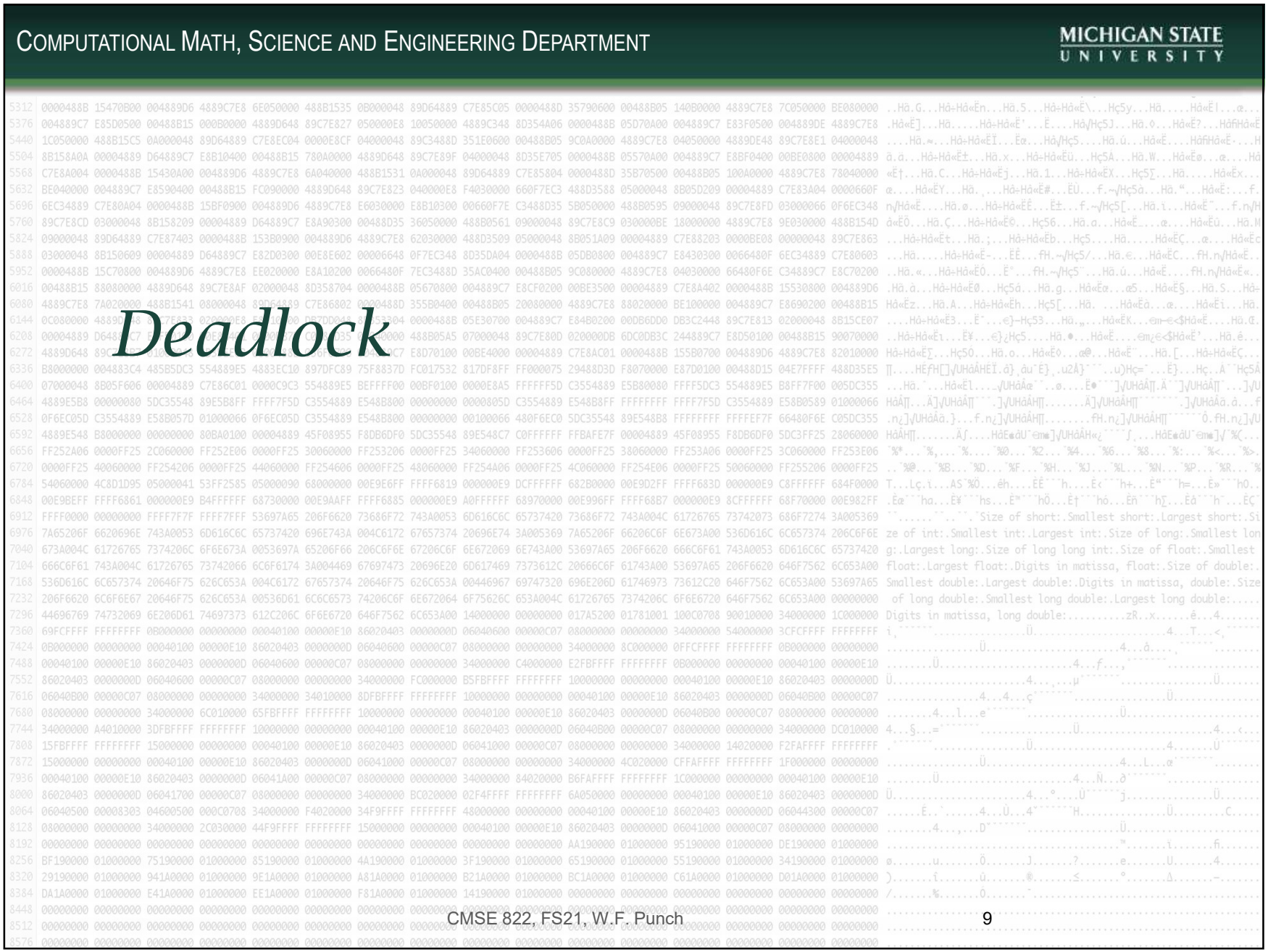
- unique_lock: more general purpose lock_guard
  - can be constructed without locking
  - can be unlocked
- scoped_lock: a lock_guard to grab multiple mutexs before moving forward

barrier

- C++-20, requires g++/clang++ 11 or greater
  - has a count such that all threads reaching the barrier wait until the count reaches 0.

condition variable

- is a way to have a thread wait and to notify other threads when processing can proceed
  - wait is on a condition (a Boolean function)
  - can notify one or many threads
  - think reader/writer or produce/consumer as natural fits.



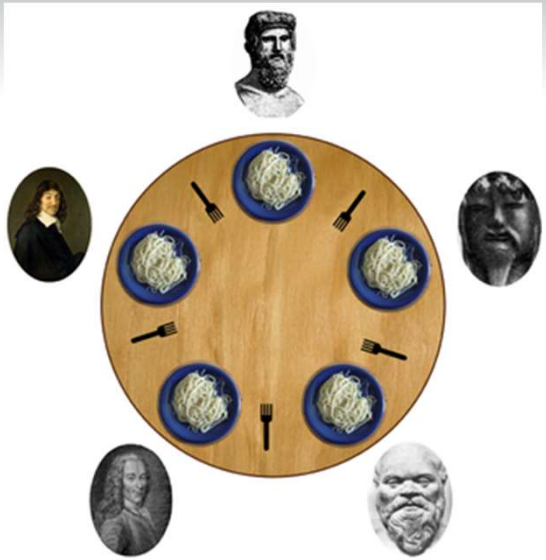
Deadlock

Deadlock?

When two (or more) processes are stuck waiting for access to a resource held by a different process.

If true, we have reached an impasse. No deadlocked process can continue unless another process releases a resource.

Dining Philosophers



- you need two forks to eat
- you can only grab a fork to your direct left or right
- you need to alternate eating and thinking. When thinking (not eating) you must put down both forks

Four Conditions for Deadlock

- Coffman et. al. 1971
- Necessary conditions for deadlock to exist:
  - **Mutual Exclusion**
    - At least one resource must be held is in non-sharable mode
  - **Hold and wait**
    - There exists a process holding a resource, and waiting for another
  - **No preemption**
    - Resources cannot be preempted
  - **Circular wait**
    - There exists a set of processes $\{P_1, P_2, \ldots P_N\}$, such that
      - P_1 is waiting for P_2, P_2 for P_3, …. and P_N for P_1

All four conditions must hold for deadlock to occur

Common causes

- self deadlock
- lack of mutex ordering
- starvation

selflock example

- how to fix this?

order example

- how to fix?

barrier/starvation

- how to fix

Operation	Effect
mutex *m*	Default constructor; creates an unlocked mutex
m.~***mutex***()	Destroys the mutex (must not be locked)
m.lock()	Locks the mutex (blocks for lock; error if locked and not recursive)
m.try_lock()	Tries to lock the mutex (returns true if lock successful)
m.try_lock_for(*dur*)	Tries to lock for duration *dur* (returns true if lock successful)
m.try_lock_until(*tp*)	Tries to lock until timepoint *tp* (returns true if lock successful)
m.unlock()	Unlocks the mutex (undefined behavior if not locked)
m.native_handle()	Returns a platform-specific type native_handle_type for nonportable extensions

Table 18.7. Operations of Mutex Classes, If Available

Operation	Effect
lock_guard `lg(`*m*`)`	Creates a lock guard for the mutex *m* and locks it
lock_guard `lg(`*m*`,adopt_lock)`	Creates a lock guard for the already locked mutex *m*
`lg.~`***lock_guard***`()`	Unlocks the mutex and destroys the lock guard

Table 18.8. Operations of Class `lock_guard`