More Synchronization Mechanisms

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

UNIVERSITY OF CALGARY

Topics

- thread safe classes and monitors
- message passing
- disabling interrupts
- lock variables
- strict alternation, Peterson’s algorithm
- atomic operations, synchronization hardware, spinlocks
- fork-join model & barriers
- priority inversion
- race conditions in processes (filesystems)
- reader-writer lock

Containers and Threads

C++ containers & threads

- C++ containers are not **thread-safe**
  - modifying the same container instance from multiple threads could result in race conditions and UB (undefined behavior)
  - e.g. you cannot have a shared `std::vector` and have multiple threads use it simultaneously in arbitrary ways, such as calling `push_back()`

- does it mean we cannot use C++ container in multi-threaded code?
  - we can, but only if we take proper precautions …

C++ containers & threads

- C++ containers are OK to use in multiple threads in **some circumstances:**
  - access to shared containers is done in protected critical sections
    e.g. access to shared global `std::vector` guarded by a mutex
  - or, each thread modifies a different instance
    e.g. local `std::vector` variables in each thread
  - or, each thread use shared instances in read-only manner (only use `const` methods)
    e.g. shared global `std::vector` variable, used in read-only mode in multiple threads

C++ containers & threads

- thread safety of other types of access depend on the container (BE CAREFUL)
  read https://en.cppreference.com/w/cpp/container (see section on Thread safety)
  - e.g. it's OK to allocate a shared `std::vector` in main thread, and then modify different elements of it in different threads, as long as we don't resize it (shrink or grow)
  - but the same **does not** apply to `std::unordered_map`, since change in values could result in re-balancing the tree, which is not thread-safe

Monitors

Monitors

- a monitor is a higher-level construct compared to mutexes and semaphores
- a monitor is a **programming language construct** that controls access to data shared between threads
- synchronization code automatically added by a compiler and enforced at runtime
- implemented (to some extent) in C#, D, Modula-3, Java, Ruby, Python, …
- can be emulated in C++, by manually implementing **thread-safe classes**

- can be even emulated in C, but requires careful use

Monitors

- a monitor is a **module** that encapsulates
  - private data members that can be shared among threads
  - public methods that operate on these shared data structures
  - monitors can only be accessed via the published methods
  - execution of all public methods is automatically mutually exclusive per instance, using a hidden mutex, which is auto-locked in each method
  - can include condition variables for signalling conditions

- a properly implemented monitor is *virtually impossible* to use in a wrong way, because they are thread-safe

Monitors

- concurrent invocations of any method of the same instance will result in only one thread executing a method, while the other threads will block
- Example (made-up syntax):

```
Monitor counter {
  int c = 0;
  void incr() {
    c = c + 1;
  }
  void decr() {
    c = c - 1;
  }
  int get() { return c; }
}
```

←calling `incr()` or `decr()` from multiple threads would allow one thread in, the rest would block

←think of all bodies of all methods as being critical sections, protected by one mutex

←in C++ you can emulate this by making a private mutex, and locking it at the beginning of every method

Monitors with condition variables

- monitors can have their own condition variables
- CVs declared as part of the module
- only accessible from within the module (private)
- similar to `pthread_cond_t`

```
Monitor counter {
   ...
   condition c1, c2;
   proc() {
      c1.wait();
      c2.signal();
   }
}
```

Thread-Safe Class

Thread-safe C++ class - with mutex & scope guard

```
class Counter {
  long counter = 0;
  std::mutex m;

public:

  void increment() {
    m.lock();
    counter ++;
    m.unlock();
  }

  void decrement() {
    std::lock_guard<std::mutex> guard(m);
    counter --;
  }

  long get() {
    std::lock_guard<std::mutex> guard(m);
    return counter;
  }
};
```

wrap each body of public method with mutexes → thread safe class

scope guard to auto-release mutex on return (or exception...)

```
int main() {
  Counter c;

  auto t1 = std::thread( [&](){
    for( int i = 0 ; i < 10000000 ; i ++ )
      c.increment();
  });

  auto t2 = std::thread( [&](){
    for( int i = 0 ; i < 10000000 ; i ++ )
      c.decrement();
  });

  t1.join();
  t2.join();

  printf("Counter=%ld\n", c.get());
}
```

threads + lambdas

Thread-safe classes vs. semaphores and mutexes

- once a thread-safe class is correctly programmed, access to the protected resource is correct for accessing from all threads
- with semaphores or mutexes, resource access is correct only if all threads that access the resource are programmed correctly
- programming with thread-safe classes → you test/debug the class
- programming with mutexes/semaphores → you test/debug all code using them

Thread-safe classes vs. semaphores and mutexes

- most C++ containers are not thread-safe, e.g. calling `std::vector::push_back()` from multiple threads on the same vector would create a race condition

- Option 1: use mutex to append to vector

```
std::vector<int> v; // shared
std::mutex m;       // shared

// inside every thread
m.lock();
  v.push_back(7);
m.unlock();
```

it is up to the user of the `std::vector` to protect against race conditions

- Option 2: implement thread-safe vector wrapper class

```
class vector_int_ts {
  std::mutex m;
  std::vector<int> v;
public:
  void push_back(int val) {
    m.lock();
    v.push_back(val);
    m.unlock();
  }
}
```

now you can use it like this:

```
vector_int_ts v; // shared

// inside any thread
v.push_back(7);
```

Message Passing

Message Passing

- processes or threads send each other **messages**
  - could work across different computers too, over network
- messages can contain arbitrary data
- delivered messages can be queued up in **mailboxes**
- processes can check contents of mailboxes, take messages out, or wait for messages
- common implementation is MPI (message passing interface)
  - popular in HPC (high-performance-computing)
  - many good tutorials available (google "MPI tutorial")
- you could create your own, e.g. using thread-safe queues

Reminder

Requirements for good race-free solution

Recall:

1. **Mutual exclusion**: No two processes/threads may be simultaneously inside their critical sections (CS).
2. **Progress**: No process/threads running outside its CS may block other processes/threads.
3. **Bounded waiting**: No process/thread should have to wait forever to enter its CS.
4. **Speed**: No assumptions may be made about the speed or the number of CPUs.

General structure

```
non-critical section
entry code
  critical section
exit code
non-critical section
```

Disabling Interrupts

Disabling interrupts (bad idea)

```
non-critical section
disable interrupts
  critical section
enable interrupts
non-critical section
```

Old idea, from back when we had single CPU per computer…

- Each process disables all interrupts just before entering its CS and re-enable them just before leaving the CS.
- Once a process has disabled interrupts, it can examine and update the shared memory without interventions from other processes.

Problems:

- What if a process never re-enables the interrupts?
- On a multi-CPU systems, disabling interrupts affects only one CPU.
- Sometimes used inside kernels, but even that is becoming problematic.

Lock Variables

Software solution 1 – Lock variables (bad idea)

- A shared (`lock`) variable, initialized to 0.
  - `lock == 0` : no process is in CS
  - `lock == 1` : a process is in CS
- A process can only enter its CS if `lock==0`. Otherwise, it must wait.
- Any problems?

```
non-critical section
while (lock == 1) {;}
lock = 1;
  critical section
lock = 0;
non-critical section
```

Software solution 1 – Lock variables (bad idea)

- A shared (lock) variable, initialized to 0.
  - `lock == 0` : no process is in CS
  - `lock == 1` : a process is in CS

- A process can only enter its CS if `lock==0`.
- Otherwise, it must wait.

- **Problem: no mutual exclusion**
  - 1st thread gets past `while`...
  - context switch to 2nd thread
  - 2nd thread enters CS
  - context switch to 1st thread
  - 1st thread enters CS
  - both threads are in their CS

```
non-critical section
while (lock == 1) {;}
lock = 1;
  critical section
lock = 0;
non-critical section
```

Strict Alternation

Software solution 2 – Strict alternation (decent idea)

- two processes alternate entering their critical sections
- shared global variable `turn`, initialized to `0`

Thread 0:

```
while(1) {
  while (turn != 0) {;}
    critical section
  turn = 1;
  non-critical section
}
```

Thread 1:

```
while(1) {
  while (turn != 1) {;}
    critical section
  turn = 0;
  non-critical section
}
```

- mutual exclusion is OK
- other problems?

Software solution 2 – Strict alternation (decent idea)

- two processes alternate entering their critical sections
- shared global variable `turn`, initialized to `0`

```
Thread 0:

while(1) {
  while (turn != 0) {;}
    critical section
  turn = 1;
  non-critical section
}
```

```
Thread 1:

while(1) {
  while (turn != 1) {;}
    critical section
  turn = 0;
  non-critical section
}
```

spinlock

- problem 1: busy waiting – might be OK in some applications
- problem 2: only works for 2 processes
- problem 3: violates progress requirement – faster process blocked by slower process not in CS

Peterson's algorithm

Software solution 3 – Peterson's algorithm (good idea, but…)

- software solution for 2 processes (can be extended to N processes)
- two shared variables:
  - integer **`turn`** – indicates whose turn it is
  - array **`flag[2]`** – indicates who is interested in entering CS, initialized to **`0`**

Thread 0:

```
while(1) {
   flag[0] = TRUE;
   turn = 1;
   while (flag[1] && turn == 1) {;}
      critical_section
   flag[0] = FALSE;

   non-critical-section
}
```

Thread 1:

```
while(1) {
   flag[1] = TRUE;
   turn = 0;
   while (flag[0] && turn == 0) {;}
      critical_section
   flag[1] = FALSE;

   non-critical-section
}
```

Software solution 3 – Peterson's algorithm (good idea, but…)

Thread 0:

```
while(1) {
   flag[0] = TRUE;
   turn = 1;
   while (flag[1] && turn ==
      critical_section
    flag[0] = FALSE;

   non-critical-section
}
```

Thread 1:

```
                {
    flag[1] = TRUE;
    turn = 0;
          (flag[0] && turn == 0) {}
    tical_section
     flag[1] = FALSE;

    itical-section
```

mutual exclusion ✓

progress ✓

bounded waiting ✓

- WARNING: will not work on modern CPUs
  - assumes atomicity, visibility & ordering
  - modern CPUs: out-of-order execution and/or memory reordering make this a no-go solution
  - modern CPUs provide special instructions that could be used to fix this

Atomic Operations

Synchronization hardware

- most modern computer systems provide special hardware instructions that implement useful atomic operations
  - atomic operation is an operation that appears to execute instantaneously with respect to the rest of the system
  - no other threads/processes will be able to change the state, or observe intermediate state, while the atomic operation is happening
- these can be used to create higher level locking mechanisms, such as mutexes
- they can be also used to make lock-free solutions or even wait-free solutions https://en.wikipedia.org/wiki/Non-blocking_algorithm
- examples: compare-and-swap, test-and-set, swap

Compare-and-swap (CAS)

- Compare-And-Swap is an atomic operation supported by most CPUs today, e.g. CMPXCHG on Intel
- general algorithm:
  - compare contents of memory to val1
  - if they are the same, change the memory to val2
  - return the original content of memory

must be atomic !

Pseudocode:

```
int cas(int * mem, int val1, int val2)
{
  int old = *mem;
  if (old == val1) *mem = val2;
  return old;
}
```

must be atomic !

```
int p = 0; // shared

// each thread:
while(1) {
  while( cas(&p,0,1) == 1 ) {;}
  critical section
  p = 0;
  non-critical section
}
```

Compare-and-swap in GCC 4.4+

- gcc provides access to a number of atomic operations, including CAS
- `type __sync_val_compare_and_swap (type *ptr, type oldval, type newval)`
  - atomic compare and swap
  - `type` can be any 8, 16, 32 and 64 bit integers or pointers
  - saves the original value of `*ptr`
  - if the current value of `*ptr==oldval`, then overwrite `*ptr` with `newval`
  - returns the original value of `*ptr`
- `bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval)`
  - same as above, but a bit more convenient
  - returns true if `newval` was written, otherwise returns false

Atomic counter with compare-and-swap (lock free)

- we can implement thread-safe counter with just compare-and-swap

```
void atomic_add( int * counter, int val) {
  int done = 0;
  while( ! done) {
    int curr = * counter;
    done = __sync_bool_compare_and_swap(
       counter, curr, curr + val);
  }
}
```

```
void inc( int * counter) {
  atomic_add( counter, 1);
}

void dec( int * counter) {
  atomic_add( counter, -1);
}
```

- notice the `while` loop loops only if other threads are trying to modify the same counter at the same time, and will usually terminate very quickly
- the program is considered **lock-free** – at least one thread is guaranteed to progress

Atomic counter with atomic integer (lock/wait free)

- if you use a type that supports atomic operations, you can create lock-free programs, or even wait-free programs
- atomic operations easy to use C++11 and above:

```
#include <atomic>

std::atomic<int> counter;
```

```
void inc( std::atomic<int> & counter) {
  counter ++;              // atomic
  counter += 1;            // atomic
  counter = counter + 1; // NOT ATOMIC!!!
}

void dec( std::atomic<int> & counter) {
  counter --;
}
```

- the above code is "wait-free" on most architectures, as each operation will finish in bounded amount of time
- wait-free implies lock-free, but not the other way around
- think of wait-free as programming with atomic operations that can be used without loops

C++, thread-safe class that is lock-free (and likely wait-free)

```
#include <cstdio>
#include <thread>
#include <atomic>

class Counter {
  std::atomic<long> p_counter = {0};
public:
  void increment() { p_counter ++; }
  void decrement() { p_counter --; }
  long get() { return p_counter; }
};
```

note: Counter is a silly class, since we could just use std::atomic<long> c; but it could prevent us from accidentally typing c = c + 1; and it would also allow us to change the implementation later without changing APIs

```
int main() {
  Counter c;
  auto t1 = std::thread( [&] () {
    for( int i = 0 ; i < 10000000 ; i ++ )
      c.increment();
  });
  auto t2 = std::thread( [&] {
    for( int i = 0 ; i < 10000000 ; i ++ )
      c.decrement();
  });
  t1.join();
  t2.join();
  printf("Counter=%ld\n", c.get());
}
```

Test-and-set

- a weaker version of compare-and-swap, only using booleans
- general algorithm
  - remember contents of memory
  - set memory to true
  - return the remembered contents of memory

atomic !

Pseudocode:

```
int tas(int* mem)
{
  int old = * mem;
  * mem = TRUE;
  return old;
}
```

atomic !

```
// shared with all threads:
int p = 0;

// in each thread:
while(1) {
  while( tas(&p) ) {;}
  critical section
  p = 0;
  non-critical section
}
```

Using test-and-set to protect CS

Swap

- another atomic operation that can be used for synchronization
- general algorithm
  - atomically swap contents of two memory locations

Pseudocode:

```
void swap(int* a, int* b)
{
  int tmp = * a;
  * a = * b;
  * b = tmp;
}
```

atomic !

```
// shared with all threads:
int lock = 0;

// in each thread:
while(1) {
  int key = 1; // local var.
  while(key) swap(&lock,&key);
  critical section
  lock = 0;
  non-critical section
}
```

Using swap to protect CS.

Spinlocks

Spinlocks

- another synchronization mechanism
- lightweight alternative to mutex, but read this before using:
  https://www.realworldtech.com/forum/?threadid=189711&curpostid=189723
  TLDR; in general, prefer mutex over spinlock
- implemented using busy waiting loops,
  but only* makes sense on multi-core/multi-cpu systems
- usually implemented in assembly, using **atomic instructions**
- potentially efficient if you know the wait time will be very short
  as re-scheduling is not required

atomic operation - an operation that appears to execute instantaneously w.r.t. to the rest of the system

nothing else will be able to change the state, or observe intermediate state, while the operations is happening

Spinlocks

- you could use spinlocks instead of mutexes to protect critical sections

mutex

```
mutex m;
...

mutex_lock( & m);
  /* critical section */
mutex_unlock( & m);
```

spinlock

```
spinlock s;
...

spin_lock( & s);
 /* (very short) critical section */
spin_unlock( & s);
```

- code using spinlocks might actually run a bit faster, especially if CS is shorter than time slice and/or number of threads <= number of cores

- but spinlocks are less efficient – they tie up the CPU

Spinlock in x86 - https://en.wikipedia.org/wiki/Spinlock

```
locked:                     ; The lock variable. 1 = locked, 0 = unlocked.
     dd      0                ; initialized to 'unlocked'

spin_lock:                                ; procedure to lock the mutex
     mov     eax, 1           ; set the EAX register to 1
     xchg    eax, [locked]    ; atomically swap EAX with the lock variable
                              ; eax = old locked, new locked = 1
     test    eax, eax         ; test whether old locked == 0
     jnz     spin_lock        ; keep spinning until old locked == 0
     ret                      ; the lock was acquired, we are done (locked = 1!!!)

spin_unlock:                  ; procedure to unlock the mutex
     mov     eax, 0           ; set EAX register to 0.
     xchg    eax, [locked]    ; atomically set locked = 0
     ret                      ; lock has been released
```

Spinlock using compare-and-swap

```
void spin_lock(volatile int *p)
{
    while( ! __sync_bool_compare_and_swap(p, 0, 1)) { /* ??? */ }
}

void spin_unlock(volatile int *p)
{
    *p = 0;
}
```

Spinlock using compare-and-swap

```
void spin_lock(volatile int *p) {
    while(!__sync_bool_compare_and_swap(p, 0, 1)) {

        ;                       // option 1 - CPU/bus very busy
        sched_yield();          // option 2 - reschedule thread, lot of CPU overhead
        while(*p) {;}           // option 3 - lot of CPU overhead, less bus overhead
        while(*p) _mm_pause(); // option 4 - less bus overhead with multiple CPUs
                                //            mm_pause is an intrinsic, delaying
                                //            execution of the next instruction
                                //            by a small amount

    }
}

void spin_unlock(volatile int *p) {
    *p = 0;
}
```

Spinlocks in pthreads

```
#include <pthread.h>
int   pthread_spin_init(pthread_spinlock_t * lock, int pshared);
int   pthread_spin_destroy(pthread_spinlock_t * lock);
int   pthread_spin_lock(pthread_spinlock_t * lock);
int   pthread_spin_trylock(pthread_spinlock_t * lock);
int   pthread_spin_unlock(pthread_spinlock_t * lock);
```

Notice the similarity to mutex APIs - making it trivial to switch between mutexes and spinlocks.

Synchronization hardware

Synchronization hardware

- can be used to protect critical sections with mutual exclusion, progress, speed and even bounded waiting

- advantage:
  - avoids system calls
  - can be more efficient if expected wait time is short
  - only makes sense on multi-CPU/core systems

- drawbacks:
  - busy-waiting (spinlocks)
  - no bounded wait (can be added, but requires extra coding)

Bounded waiting with synchronization hardware

- when used correctly, the low level atomic operations such as compare-and-swap, test-and-set, swap can be used to achieve mutual exclusion, progress and speed
- but they are too low level to achieve bounded waiting, especially for more than 2 processes
- bounded waiting can be 'added', for example, by using few more shared variables

```
// we'll use integer for locking
// using test-and-set
int lock;

// array of booleans indicating which
// process wants to enter critical
// section
int waiting[n];

// process/thread ID, starting at 0
int id;

// general structure
while (1) {
    /* ??? entry code ??? */
    /* critical section */
    /* ??? exit code ??? */
    /* remainder section */
}
```

Bounded waiting with synchronization hardware

indicate interest to enter CS

if CS is locked, we wait until someone else gives us a turn

clear indicator

```
while (1) {
    waiting[id] = true;
    while (waiting[id]
           && testAndSet(&lock)) {;}
    waiting[id] = false;

    /* critical section */

    j = (id + 1) % n;
    while ((j != id) && !waiting[j])
        j = (j + 1) % n;
    if (j == id)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */
}
```

find next process trying to enter CS

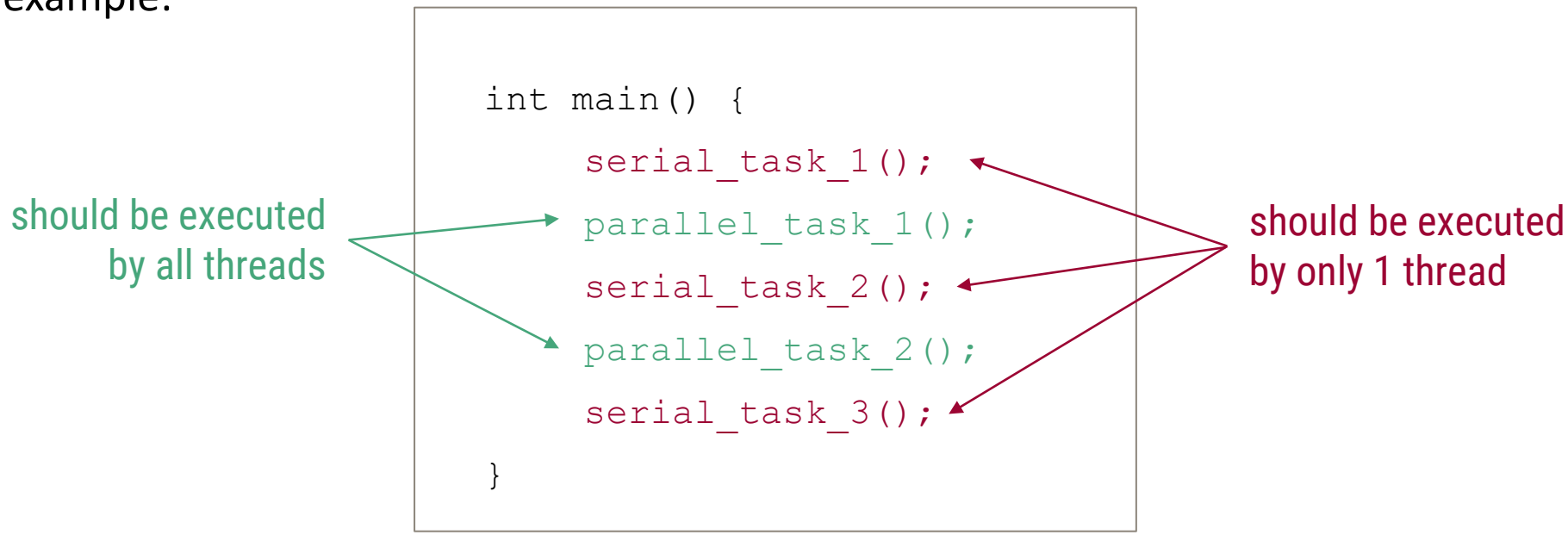
if nobody else waiting, release lock

if we found someone else waiting, let them enter CS, but without releasing lock

Fork-join Model

Fork-join model

- imagine you have program
  - parts of it can run in parallel
  - other parts can only run serially
- for example:

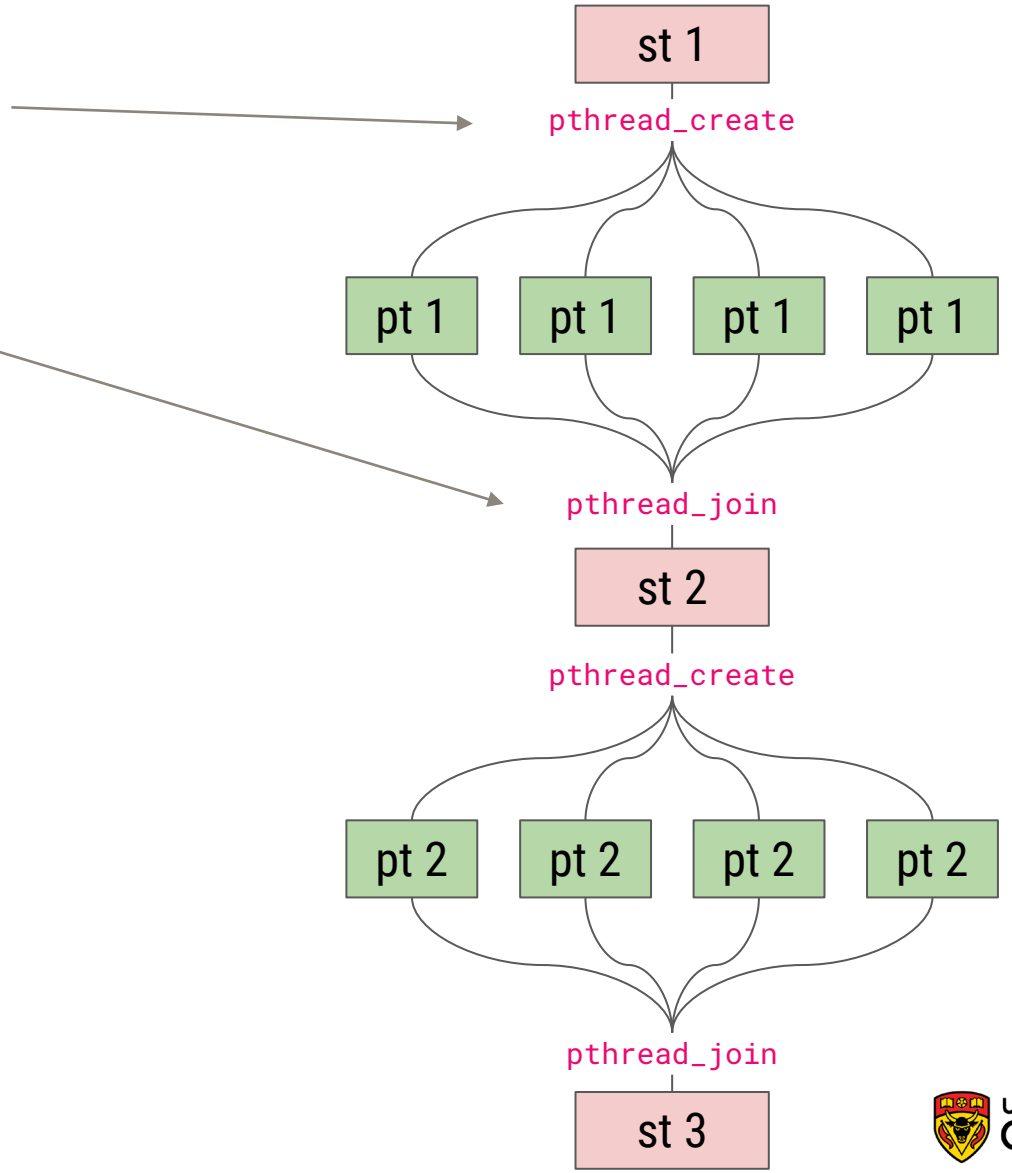


- Q: how do we program this?
  A: we can try `pthread_create()` / `pthread_join()`

Fork-join model (using thread creation and destruction)

- we can **create** threads whenever we want to run something in parallel
- then **destroy** threads when we need to run something in a single thread

```
int main() {
    st1     serial_task_1();
    pt1     parallel_task_1();
    st2     serial_task_2();
    pt2     parallel_task_2();
    st3     serial_task_3();
}
```



Fork-join using pthread_create and pthread_join

```
const int N_THREADS = 5;
const int N_WORK = 5;
```

```
void serial_work() {
  for( int i = 0 ; i < N_WORK ; i ++ ) {
    printf("serial work %d\n", i);
    rand_sleep();
  }
}
```

```
void * parallel_work( void *) {
  for( int i = 0 ; i < N_WORK ; i ++ ) {
    printf("parallel work %d\n", i);
    rand_sleep();
  }
}
```

```
int main() {
  pthread_t thread_id[N_THREADS];

  serial_work();

  for( int i = 0 ; i < N_THREADS ; i ++)
    pthread_create( & thread_id[i], NULL,
      parallel_work, NULL);
  for( int i = 0 ; i < N_THREADS ; i ++)
    pthread_join( thread_id[i], NULL);

  serial_work();

  for( int i = 0 ; i < N_THREADS ; i ++)
    pthread_create( & thread_id[i], NULL,
      parallel_work, NULL);
  for( int i = 0 ; i < N_THREADS ; i ++)
    pthread_join( thread_id[i], NULL);

  serial_work();
}
```

barrier-fork-join-with-pthreadjoin.cpp
works, but **not very efficient** due to creating/destroying threads

Barriers

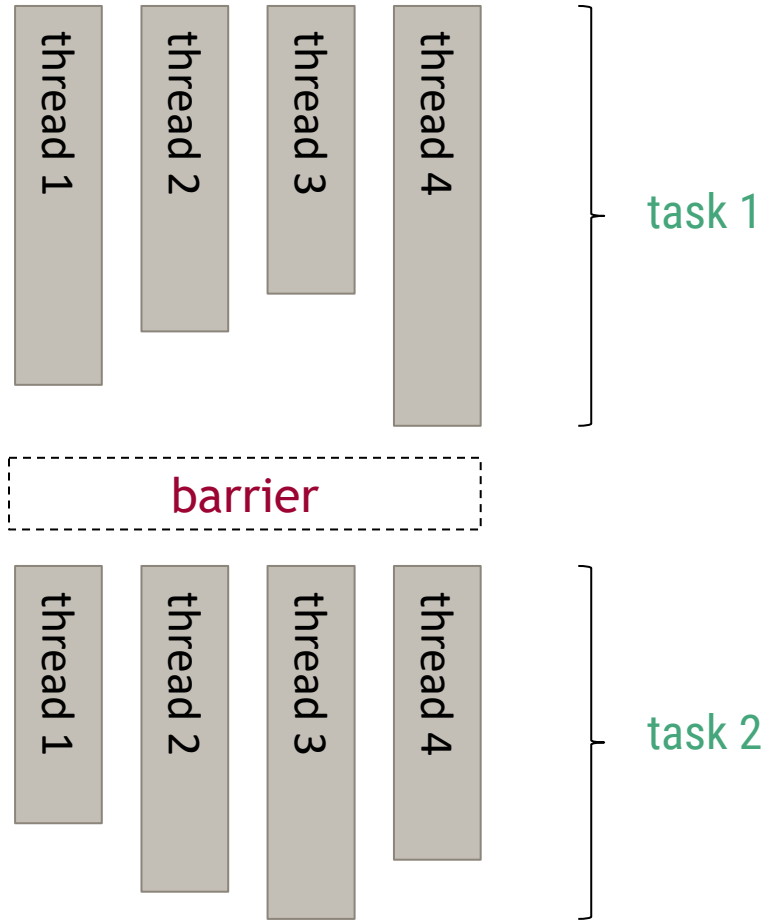
Barriers

- imagine you have a program with multiple threads, and occasionally you need all threads to reach the same point before continuing

  i.e. you want all threads to **block** until all threads reach this point, and then all threads would **continue** again

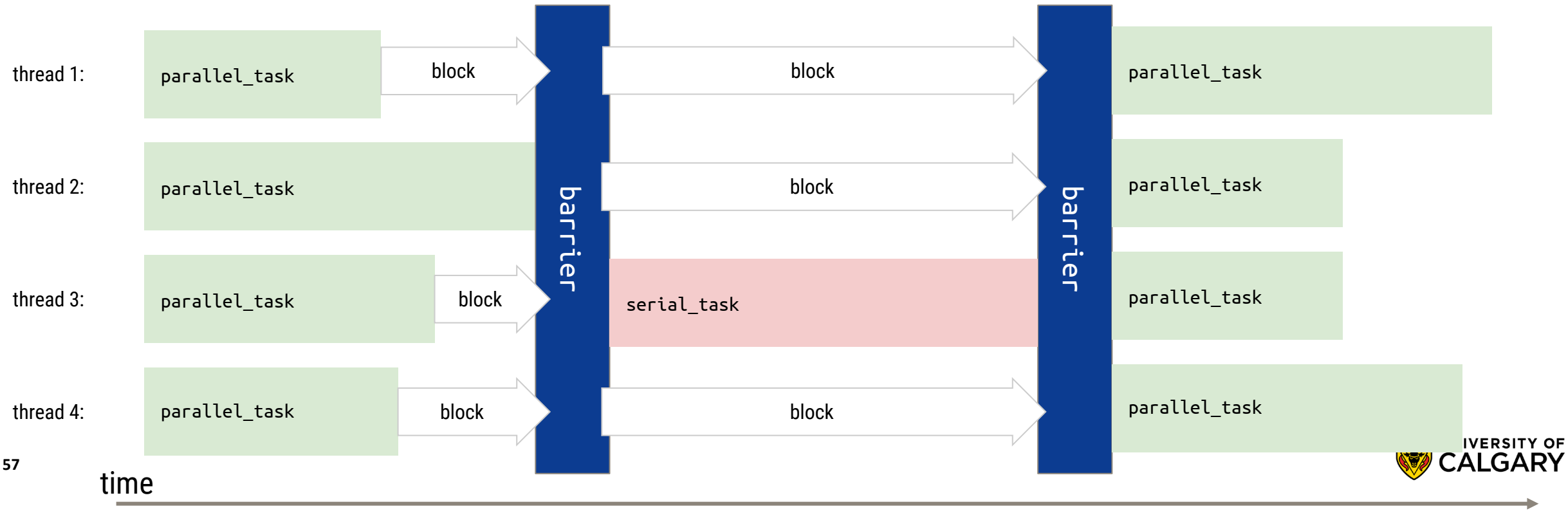
- this is easy to program using barriers
- when a thread executes barrier, the thread waits until the last thread reaches the same barrier, after which all threads are unblocked
- example pseudocode:

```
thread_run() {
   parallel_task_1(); // all threads run this together
   barrier();         // threads wait until all threads reach the barrier
   parallel_task_2(); // all threads run this together
}
```



Fork-join example using barriers

- barriers can also distinguish one of the threads from the rest (the thread is picked arbitrarily)
- this can be used to pick one thread to execute the serial task
- note that we need to **surround the serial task with two barriers**
- example with 4 threads, where thread 3 is (randomly) picked to execute serial task:



pthread_barrier

- `pthread_barrier_t barrier;`
  - declare a barrier object, shared by all threads, e.g. global variable [yuck]
- `pthread_barrier_init( pthread_barrier_t *barrier,`
  `const pthread_barrierattr_t *attr,`
  `unsigned int count)`
  - initialize a barrier object
  - `count` specifies how many threads we are synchronizing (usually total number of threads)
- `int pthread_barrier_wait(pthread_barrier_t *barrier)`
  - first `count-1` threads will block, until `count` threads have called this
  - returns `0` for `count-1` threads, and `PTHREAD_BARRIER_SERIAL_THREAD` for one arbitrary thread
- `pthread_barrier_destroy(pthread_barrier_t *barrier)`
  - cleanup

Fork-join example using pthread_barrier (C)

```
const int N_THREADS = 5;
const int N_WORK = 5;
```

```
void serial_work() {
  for( int i=0 ; i<N_WORK ; i++ ) {
    printf("serial work %d\n", i);
    rand_sleep();
  }
}
```

```
void parallel_work() {
  for( int i=0 ; i<N_WORK ; i++ ) {
    printf("parallel work %d\n", i);
    rand_sleep();
  }
}
```

```
void * run( void *) {
  parallel_work();
  if( pthread_barrier_wait( & barr_id) != 0)
    serial_work();
  pthread_barrier_wait( & barr_id);
  parallel_work();
  if( pthread_barrier_wait( & barr_id) != 0)
    serial_work();
  pthread_barrier_wait( & barr_id);
  parallel_work();
}
```

```
int main() {
  pthread_t thread_id[N_THREADS];
  pthread_barrier_init( & barr_id,NULL,N_THREADS);
  for( int i = 0 ; i < N_THREADS ; i ++)
    pthread_create( & thread_id[i], NULL, run, NULL);
  for( int i = 0 ; i < N_THREADS ; i ++)
    pthread_join( thread_id[i], NULL);
}
```

barrier-fork-join-with-pthreadbarrier.cpp

much more efficient since we are re-using threads

C++ custom barrier (no built-in barrier until C++20)

```
class simple_barrier {

  std::mutex m_;
  std::condition_variable cv_;
  int n_remaining_, count_;
  bool coin_;

public:

  simple_barrier(int count) {
    count_ = count;
    n_remaining_ = count_;
    coin_ = false;
  }

  bool wait() {
    std::unique_lock<std::mutex> lk(m_);
    if( n_remaining_ == 1) {
      coin_ = ! coin_;
      n_remaining_ = count_;
      cv_.notify_all();
      return true;
    }
    auto old_coin = coin_;
    n_remaining_ --;
    cv_.wait(lk, [&](){ return old_coin != coin_; });
    return false;
  }

};
```

barrier-implementation.cpp

Readers/writer lock

Readers/writer lock

- scenario:
  - a single resource is shared among several threads and resource supports **multiple concurrent readers**, but only a **single writer**, e.g. a shared global variable
  - some threads only read the resource, others read and write it

- we can use a **shared-exclusive lock**, or **reader-writer lock**
  - such lock would have two different locking APIs - one for reading, one for writing

- pthread has `pthread_rwlock_*()` , C++17 has `std::shared_mutex`;

- we could use semaphores to implement this ourselves

Readers/writers implementation with semaphores

```
// number of readers currently reading
int readcount = 0;
// used as mutex to protect CS in reader
sem_t cs = 1;
// semaphore to block/unblock writers
sem_t w_only = 1;

writer_lock() {
    sem_wait(w_only); // lock out readers
}

writer_unlock() {
    sem_post(w_only); // allow readers
}
```

```
reader_lock() {
  sem_wait(cs); // enter critical section
    readCount ++; // add reader
    if (readCount == 1) // first reader will
      sem_wait(w_only); // wait for writers
  sem_post(cs); // exit critical section
}

reader_unlock() {
  sem_wait(cs); // enter critical section
    readCount --; // remove reader
    if (readCount == 0) // last reader will
      sem_post(w_only); // let writers write
  sem_post(cs); // exit critical section
}
```

Priority Inversion

Priority inversion

- Scenario: Assume we have three processes, L, M, and H, whose priorities follow the order L < M < H. Assume that process H requires resource R, which is currently being accessed by process L. While H is waiting for L to finish using resource R, M becomes runnable, thereby preempting process L. Now, H has to wait for both M and L to finish. Effectively, H has the lowest priority in this execution.
- This problem is known as priority inversion.
- Possible solution: priority-inheritance
  - As soon as H requests resource R, the process P holding resource R automatically inherits the priority of H if the P has lower priority. Once P releases the resource, its original priority is restored.
  - As a result, we would have the following execution order in the above scenario: L, H, M.

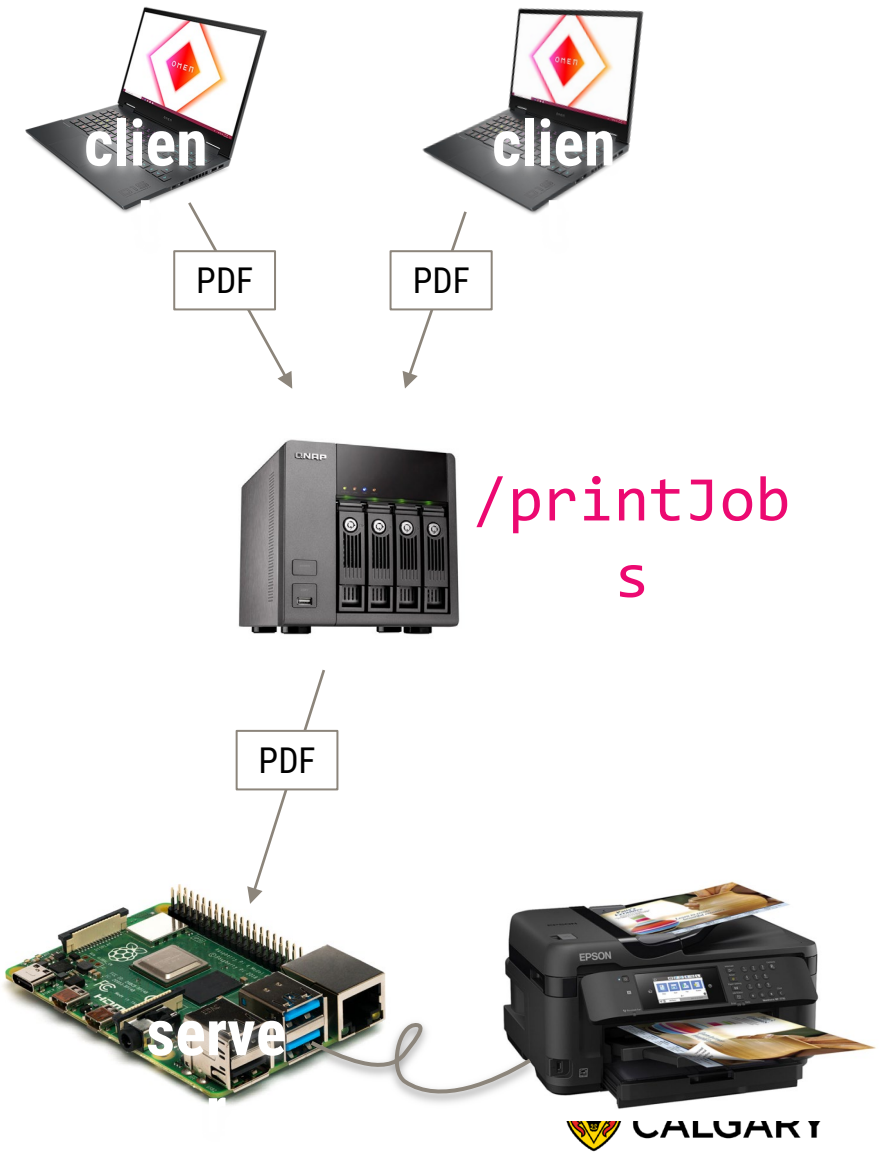
Race conditions between processes

Race conditions between processes

Imagine a simple design of a print server

- assume clients and server have access to a shared `/printJobs` directory (e.g. NFS)
- to print, a client saves a PDF file to `/printJobs`
- a print server monitors `/printJobs`, e.g. by scanning the `/printJobs` directory for `*.pdf` every 5 seconds
- when the server detects a PDF file, it prints it and removes the file

Can you spot a problem with this setup?



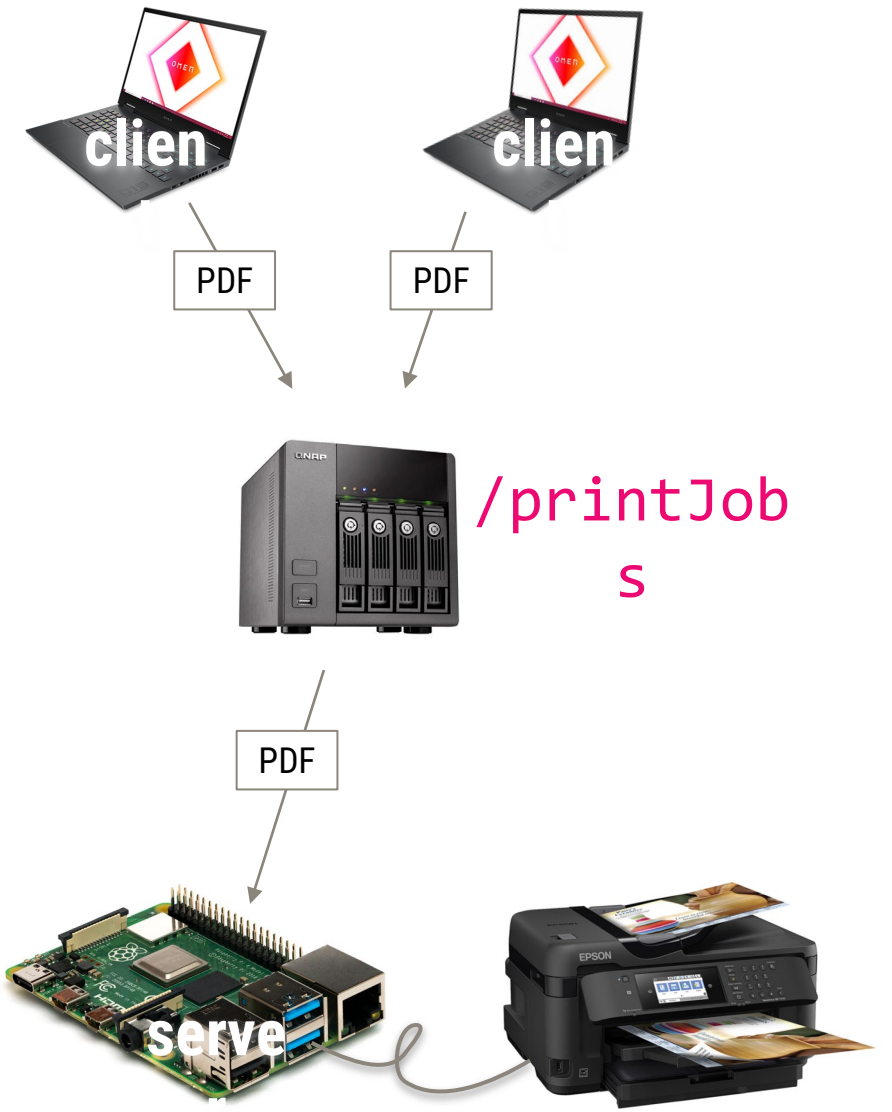
Race conditions between processes

Problems with this design:

- print server might print an incomplete PDF file
- two (or more) clients might decide to write to the same PDF file

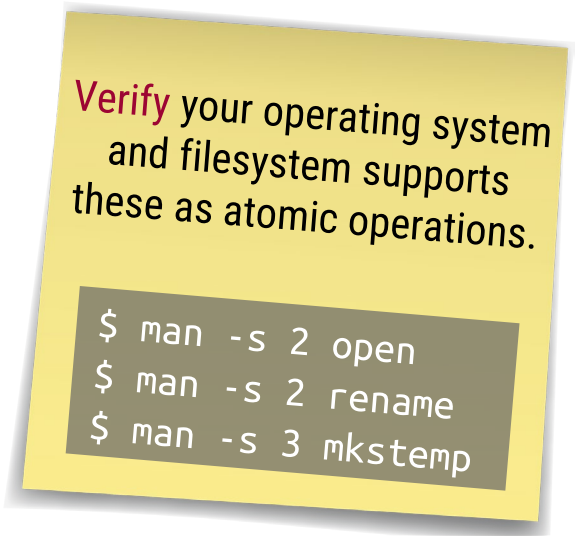
How to fix?

- we need to prevent incomplete PDF files from appearing in the directory
- each client needs to pick a unique filename for PDF



Race conditions between processes

- what assumptions can we make?
- some filesystems support file/directory locking mechanisms, but not all, so we should not rely on those
- however, nearly all filesystems support at least 2 atomic operations:
  - file creation via `open(fname, O_CREAT|O_EXCL, 0644);`
  - file rename via `rename(old_fname,new_fname);`
- many systems also support `mkstemp()` for creating temporary files
- more info: https://tldp.org/HOWTO/Secure-Programs-HOWTO/avoid-race.html



Race conditions between processes

Solution:

- server needs no modification, we only modify how clients print
- to print, client creates a temporary file inside **`/printJobs`**, e.g. file **`job-xxxxxx.tmp`**, where **`xxxxxx`** is random characters
- we can use `mkstemp(3)` for this, or `open(2)` in a loop
- client then writes PDF output to the temporary file
- finally, client renames **`job-xxxxxx.tmp`** to **`job-yyyyyy.pdf`** where **`yyyyyy`** is another string of random characters

```
// creating temp. file
loop:
  xxxxxx = random string
  open(job-xxxxxx.tmp,
       O_CREAT|O_EXCL,0644)
  break if successful
```

```
// renaming temp. file to PDF
loop:
  yyyyyy = random string
  rename( job-xxxxxx.tmp,
          job-yyyyyy.pdf)
  break if successful
```

Event Flags

Event Flags

- **event flags:**
  - a memory word with N bits
    - different events may be associated with different bits in a flag
  - operations:
    - set flag
    - clear flag
    - wait for 1 flag
    - wait for any flag
    - wait for all flags

Review

Review

- Which one of the following achieves mutual exclusion but violates the “progress” requirement?
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - Peterson’s Solution

Onward to … deadlocks

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY