TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Threads and what can be done about them

Bjoern Doebel
Michael Roitzsch

The plan

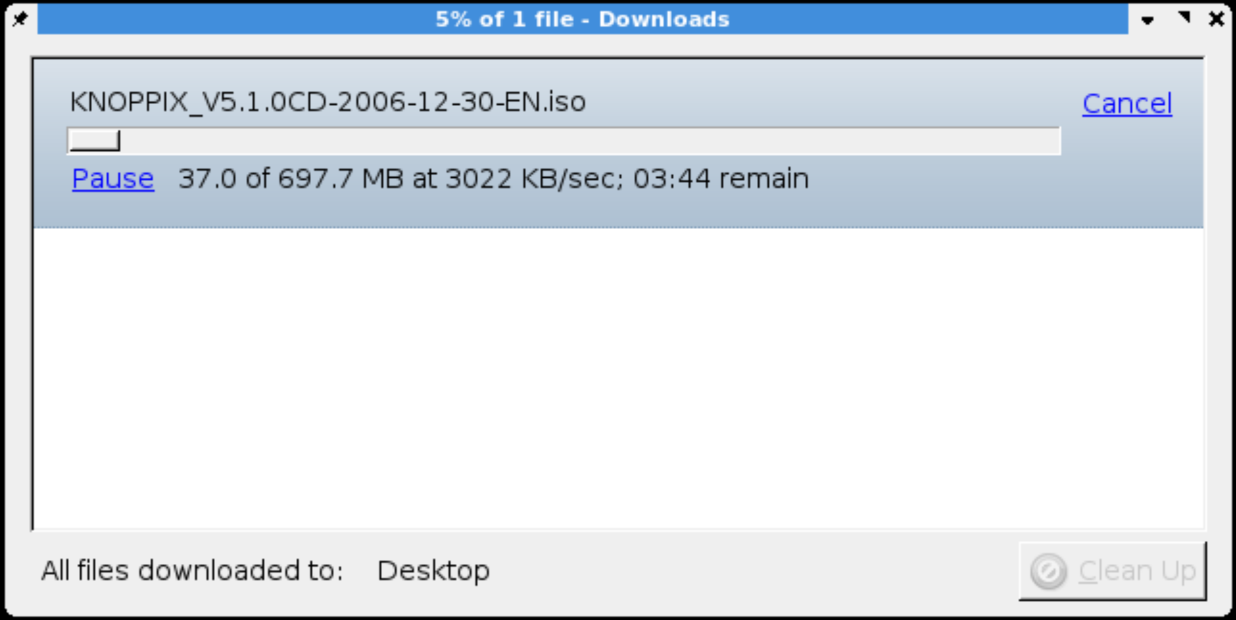
- Introduction to Threads
- PThreads (and also C++ threads)
- Threading problems
  - Race conditions
  - Deadlocks
- Advanced Concepts
  - Producer-consumer scenarios

Processes vs. Threads

- Processes
  - own an Address Space
  - own Resources (files, shared memory areas, ...)
  - represent Security principals
- Threads
  - belong to a process
  - **share address space and resources** with other threads
  - run in **parallel** to other threads
- What is this good for?

Do things in the background

- Scenario: Download large file in a web browser and surf the web during the download.



5% of 1 file - Downloads

KNOPPIX_V5.1.0CD-2006-12-30-EN.iso

Pause 37.0 of 697.7 MB at 3022 KB/sec; 03:44 remain

All files downloaded to: Desktop

Distribute work

- In High Performance Computing (HPC), threads are used to perform the same computations (e.g., execute the same code) on different data.
  - use parallelism to speed up computation
  - makes sense for multi-processor or hyperthreaded machines

How do threads help?

- CPU-bound applications:
  - perform computations in parallel
  - make use of multiple processor cores
  - special libraries exist (OpenMP, MPI, ... - not covered here, take an HPC course for this.)
- I/O-bound applications:
  - increase CPU utilization by running one thread while others are waiting for I/O to be completed
  - even helps on a single CPU core

So this sounds great...

- But threading makes life harder:
  - Execution parallelism
  - Work on shared data

Rule #1 for writing multi-threaded programs:

Don't (ever!) make any assumptions about order of execution or timing behavior!

TECHNISCHE
UNIVERSITÄT
DRESDEN

Thread libraries

- Typically hide low-level implementation details from the user.
- Expose a thread API with functions for
  - thread creation and destruction
  - synchronization mechanisms
  - thread-local storage
- Examples:
  - POSIX Threads (PThreads)
  - Windows Threads
  - C++11 threads

Introduction to PThreads

What are PThreads ?

- Standardized threading programming interface for UNIX (like) systems

- specified by IEEE POSIX 1003.1c standard
- implementations are referred to as POSIX threads, or short “pthread”
- several **draft** pthread **standards**, several **implementations** ...

- implemented with a `pthread.h`
- C programming types and function calls
- typically part of / in conjunction with the libc

pthread basics

- `#include <pthread.h>`
  - defines required structures and functions
- link with `-pthread`
- `pthread_*()`
  - `pthread_self(), pthread_create()`
  - `pthread_equal(), pthread_join()`
  - `...`
- Use man pages or (online) books!
  - man pthreads – overview about pthreads
  - man pthread_create, ... - detailed infos
  - http://www.llnl.gov/computing/tutorials/pthreads

pthread basics – who am I ?!

- `pthread_t`
  - identifies thread within **a** process **uniquely**
- `pthread_t pthread_self(void)`
  - determine thread id of thread calling this function
  - to distinct (parallel) executing threads within a process
- `pthread_equal (pthread_t tid1, pthread_t tid2)`
  - to compare thread ids
  - don't use `tid1 == tid2`

Let's go – create a pthread ...

-
  ```
  int pthread_create (
            pthread_t * tidp,
            pthread_attr_t * attr,
            void * (*startme) (void *),
            void * arg);
  ```
  - tidp -> pointer to memory location where new thread id is to be stored
  - startme -> function to be executed by new thread
  - attr thread attributes (default NULL)
  - arg -> optional argument (NULL)
  - return value (int)
    - success == 0, otherwise contains an error code

Simple example

```
pthread_t ntid;

void * startme (void * arg) {
       /* Put thread code here */
}


int main (int argc, char ** argv) {
    int err;
    err = pthread_create(&ntid, NULL, startme, NULL);
    [...]
}
```

pthread basics - waiting for threads

- a thread can end its execution by
  - return from start routine (startme)
  - calling: `void pthread_exit (void *ptr);`
  - ptr is available to other threads waiting for it
  - Note: exit() terminates the process, pthread_exit() only the thread

- threads can wait for other threads' termination:

  -
    ```
    int pthread_join (pthread_t tid,
                      void **   ptr);
    ```
  - blocks until thread tid terminates
  - ptr contains value passed by thread_exit or by the return from start routine

Exercise 1

- Create 8 threads
  - assign to each thread a number, pass it over to the new thread
  - the new threads have to print out their identifier
  - at end of the thread execution return a value to the main thread
  - let the main thread wait for termination of the 8 threads and print out the return values

Threads in C++11

- #include <thread>
- Type: std::thread
- On your system: wrapper around libpthread

```
void foo() { /* thread func */ }

int main(void) {
   std::thread my_thread(foo);
   my_thread.join();
}
```

Synchronization

Exercise 2

- Add new functions to your list implementation.
- `unsigned int List::count()`
  - return the number of elements in the list by running through the list and counting them on-the-fly
- `unsigned int List::count_fast()`
  - count the number of list elements in a separate field member
  - increment/decrement member during insert()/remove()

- What are the functions' complexities?

Exercise 3

- Scenario: Web server with multiple worker threads. HTTP requests arrive and are stored in a global request list.
  - Start N (1, 2, 5, 10, 30, ...) threads executing `insert_thread()` function.
  - In `insert_thread()` add M (10, 50, 100, ...) elements to your list.
  - Make N and M configurable (compile time constants or command line argument)
  - In `main()` wait for termination of all threads using `pthread_join()`.
  - After all threads finished, print out the return values of `List::count()` and `List::count_fast()`.
  - Run the program multiple times and compare the output to your expectations.

What went wrong?

One of the two race conditions in this code:

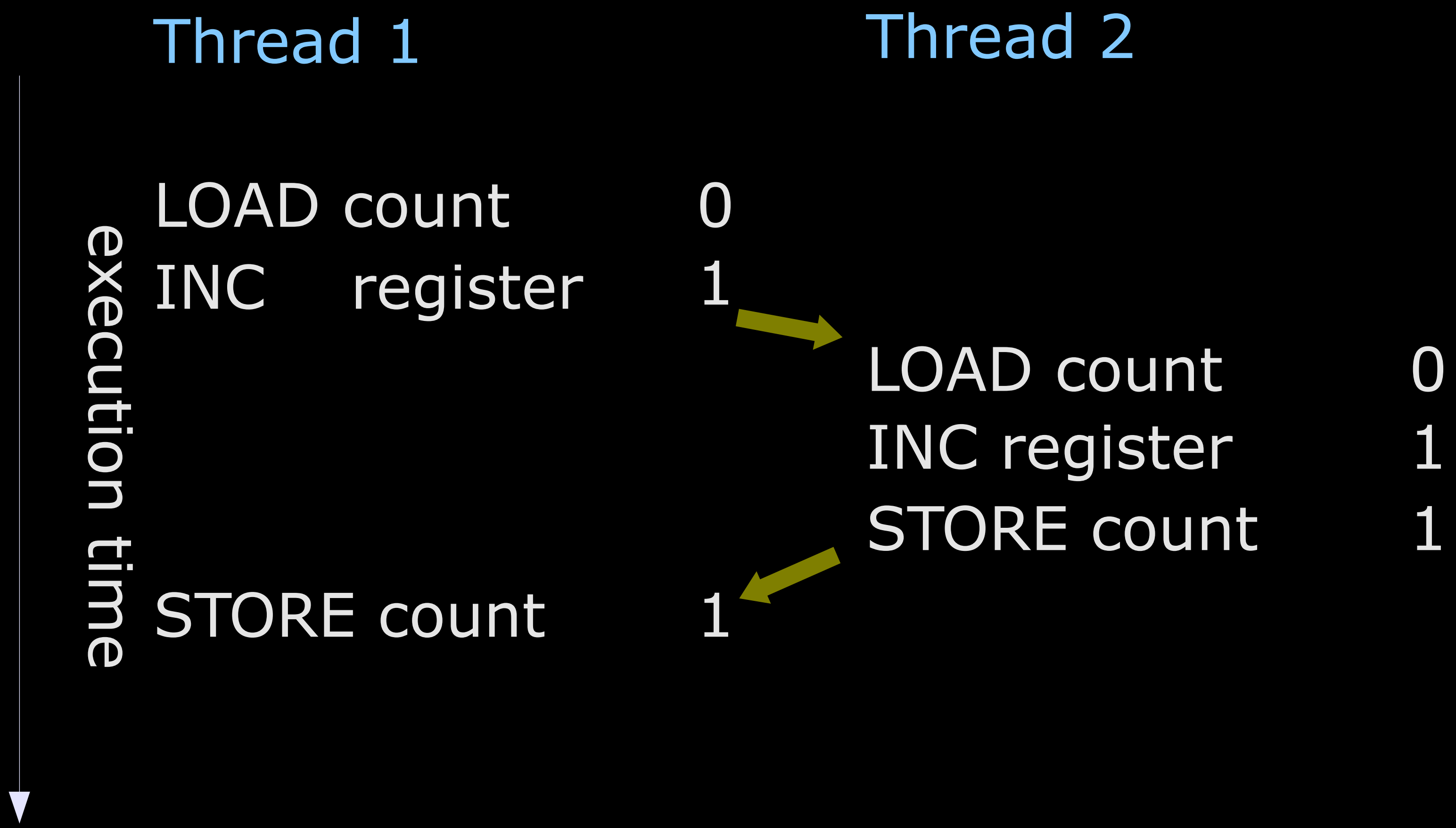
```
void List::insert_head(int *e)
{
      ListElement *next = new
                        ListElement(e);
      next->set_next(_head);
      _head = next;
      ++_elem_count;


}
```

```
LOAD  _elem_count -> REGISTER EAX
INC   REGISTER EAX
STORE EAX               -> _elem_count
```

Doing things in parallel...

count = 0;



Thread 1		Thread 2	
LOAD count	0		
INC register	1		
		LOAD count	0
		INC register	1
		STORE count	1
STORE count	1		

Synchronization

- Sometimes threads executing in parallel need to communicate
  - prevent threads from writing the same data (serialize execution of critical sections)
  - tell other thread about current execution state (e.g., “I'm done with action A...”)

- This is called synchronization and comes in many different flavors.

Mechanisms: Disabling Interrupts

- Synchronize by disabling interrupts so that the scheduler cannot be woken up by a timer interrupt:

```
    asm(“cli”);
    /* execute critical code */
    asm(“sti”);
```

  - Evil or stupid programmers may omit the STI instruction – no scheduling occurs ever again.
  - That's why CLI/STI are privileged HW instructions and are only allowed in privileged (kernel) mode.
  - In real-time systems you need to carefully pay attention to how long you disable interrupts.

Mechanisms: Lock variables

Globally: `int cs_locked = 0;`

In threads:

```
while (cs_locked == 1)
   /* busy wait */
      ;
   cs_locked = 1;       // mark locked
   /* exec. critical code */
   cs_locked = 0;       // mark unlocked
```

Do you see a problem here?

Atomic CMPXCHG

- stands for “compare-and-swap” or “compare-and-exchange”
- hardware instructions of (modern) CPUs
- support for 8/16/32/64 bits
- atomic instruction consists of:
  - reading from a memory location
  - compare with a expected value
  - if compare is successful
    - write back a new value to memory location
  - if compare fails
    - don't write back the new value
    - return instead the value found at the memory location

What is cmpxchg good for ?

- used to alter values in memory atomically
  - which could be read and written by concurrent threads
  - (on SMP machines you have also to add LOCK assembly in conjunction with cmpxchg)
- is a non-privileged command
- can be used by user applications
  - in contrast to cli/sti (disable/enable interrupts)
- can be used to implement locks

Spinlock with CMPXCHG

```
typedef unsigned long lock_t;

void lock(lock_t * lock) {
  lock_t _old   = 0;
  lock_t _new = 1;
  do {
     /* retry until owner is set to 0 (not used)
      * and changing to 1 succeeds
      */
  } while ( ! c_cmpxchg (lock, _new, _old) );
}

void unlock(lock_t * counter) {
  *counter = 0;
}
```

Usage of cmpxchg on x86 with gcc & gas

```
inline int c_cmpxchg (unsigned long *dest,
        unsigned long new_val, unsigned long cmp_val)
{
        unsigned long tmp;
        __asm__ __volatile__ (
            "lock cmpxchg %1,  %3 \n\t"
            :
            "=a"  (tmp)   /* %0 EAX, return val */
            :
            "r"   (new_val),      /* %1 reg, new value */
            "0"   (cmp_val),      /* %2 EAX, compare value */
            "m"   (*dest)/* %3 mem, destination operand */
            :
            "memory", "cc" /* code changes content of memory,
                              and conditional clause register  */
        );
        return tmp == cmp_val;
}
```

Usage of cmpxchg on x86 with gcc & gas

```
inline int c_cmpxchg (unsigned long *dest,
        unsigned long new_val, unsigned long cmp_val)
{
        unsigned long tmp;
        __asm__ __volatile__ (
            "lock cmpxchg %1,  %3 \n\t"
            :
            "=a"  (tmp)   /* %0 EAX, return val */
            :
            "r"   (new_val),      /* %1 reg, new value */
            "0"   (cmp_val),      /* %2 EAX, compare value */
            "m"   (*dest)/* %3 mem, destination operand */
            :
            "memory", "cc" /* code changes content of memory,
                              and conditional clause register  */
        );
        return tmp == cmp_val;
}
```

Spinlock with C11 atomics

```
#include <stdatomic.h>

void lock(atomic_flag *lock)
{
    do {} while (atomic_flag_test_and_set(lock));
}


void unlock(atomic_flag *lock)
{
    atomic_flag_clear(lock);
}
```

Exercise 4

- Use the `lock()` and `unlock()` functions to synchronize list insertion.
  - Design decisions:
    - Where to place the lock?
      - fine-grained locking -> one lock per ListElement
      - coarse-grained locking -> one lock per List
    - How to lock?
      - explicitly upon user request -> public `lock()/unlock()` functions
      - implicitly during insert -> private `lock()/unlock()` functions

Mechanisms: Mutexes / Semaphores

- Previous solution used busy waiting – senselessly burn CPU cycles (but are also practical, e.g., in MP systems).
- Better:
  - find out that critical section is blocked
  - sleep until CS is free (queue of threads)
  - wake up next thread when leaving CS
- Semaphores provide exactly this behavior.
  - counter + wait queue
  - lock(): decrement counter – if <=0, go to sleep
  - unlock():
    - increment counter
    - wake up next thread in queue

PThread mutexes

- pthread_mutex_t : data type for mutexes
- int pthread_mutex_init() : initialize a mutex variable
- int pthread_mutex_lock() : sleep until locking succeeds
- int pthread_mutex_unlock() : as the name says
- int pthread_mutex_trylock() : try to grab lock, return with error on failure
- int pthread_mutex_destroy() : free all resources associated with mutex

Using mutexes

```
pthread_mutex_t mutex;

int main(void)
{
   pthread_mutex_init(&mutex, NULL);

   pthread_mutex_lock(&mutex);
   /* do critical work */
   pthread_mutex_unlock(&mutex);

   pthread_mutex_destroy(&mutex);
}
```

Other mechanisms

- In some cases (e.g., real-time) unbounded sleeping is not a good idea.
- Lock-free synchronization
  - don't use any locks for synchronization, therefore no possibility of deadlocks and livelocks
  - needs careful design
- Wait-free synchronization
  - lock-free mechanisms without sleeping

Mutexes in C++11

- #include <mutex>
- std::mutex (and derivatives)

```
std::mutex mtx;

void thread_func() {
   mtx.lock();
   // critical section
   mtx.unlock();
}
```

C++11: Never forgetting to unlock

- C++: RAII

- std::lock_guard

```
std::mutex mtx;

void thread_func() {
    std::lock_guard<std::mutex> lock(mtx);
    // critical section (with branches
    // and multiple returns
} // release once lock goes out of scope
```

Exercise 5

- Copy your last solution to a new directory.
- Replace the spinlock-based implementation by one using mutexes.

Deadlocks

Exercise 6

- Download 3 (simple) examples
  - svn checkout http://svn.inf.tu-dresden.de/repos/advsysprog/day4/exercise_deadlock

  - understand code
  - what goes wrong?
  - fix it and tell us about the problem.

  - download from ...
  - uncompress, compile, execute
    - (shipped with Makefile)

- lock1 example
  - ok - endless loop of dots
  - ok - sometimes “thread XXXXX exits”
  - stops after some time printing dots, why ?
- lock2 example
  - endless loop of messages:
    - got paper, got printer, print document
  - stops after some time, why ?
- lock3 example
  - endless loop of dots
  - stops after some time, why ?

Deadlocks - conclusion

- A situation where two or more competing actions are waiting for other actions to be finished. Never one of the actions will be finished because the actions are waiting for each other.

- reasons of deadlocks in the examples:
  - multiple locks are requested by different functions/threads in different order
  - forget to release locks (return before unlocking)
  - circular dependencies within thread
    - lock(lock1); lock(lock2); lock(lock1)
      - widespread in/over various functions)

C++11: Solving deadlocks?

- **std::lock_guard / std::unique_lock**
  to avoid forgetting to unlock

- **std::lock(mtx1, mtx2, …)** → “locks given
  the Lockable objects using a deadlock avoidance algorithm”

Livelocks

- In contrast to deadlocks, the threads/processes do something, however don't make progress in respect to a goal (in my words ...)
  - Better definitions in the literature ;-)

```
thread1:                      thread2:
  lock(&lock1);                 lock(&lock2);
  while (1) {                       while (1) {
    if (try_lock(&lock2))             if (try_lock(&lock1))
      do job_X                            do job_X
    else                                else
      do job_1                              do job_2
  }                                   }

thread3: while (1) { if (job_X is finished) exit; else sleep(1);}
```

Advanced Synchronization

So far, so good

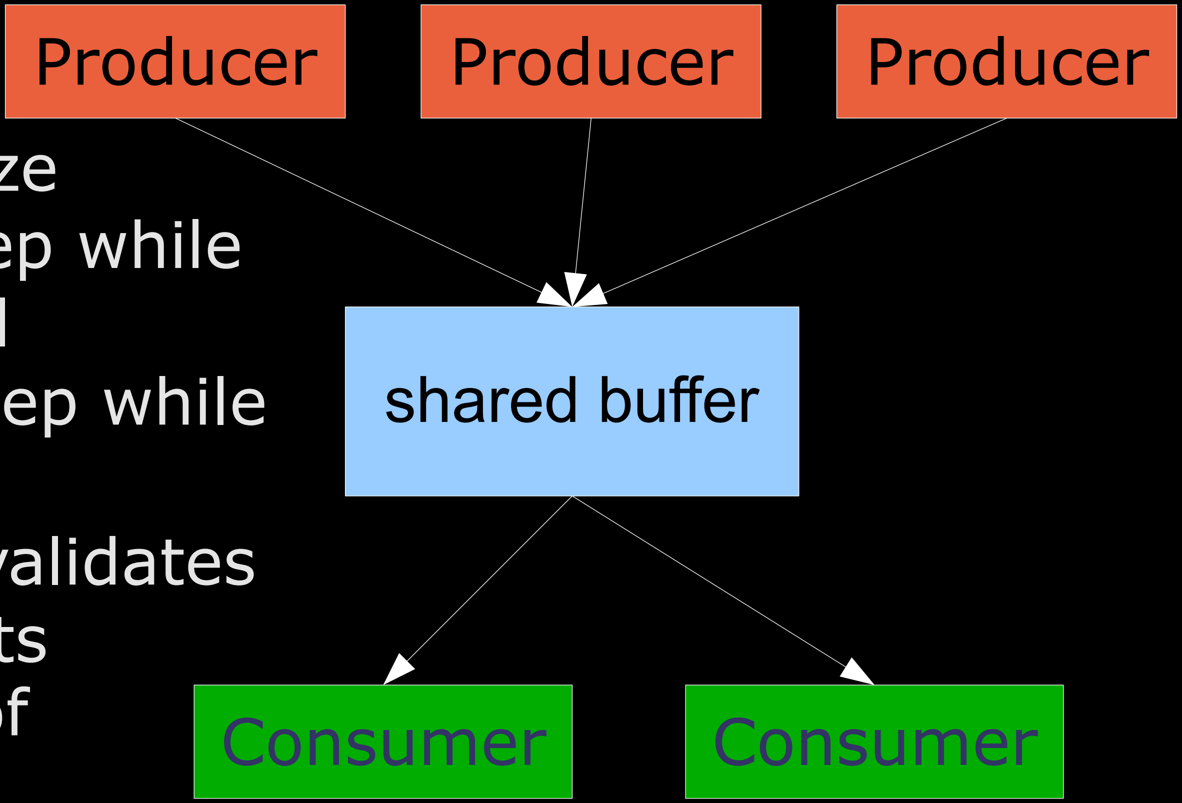
- Up to now:
  - Thread creation
  - Waiting for thread completion
  - Synchronization with mutexes to circumvent race conditions
  - Careful design to avoid deadlocks.

More on synchronization

- There are more scenarios where threads need to communicate:
  - client-server architectures (X.Org, microkernel-based OS) typically use network sockets or other OS-provided IPC mechanism
  - producer-consumer problem: keyboard and mouse drivers “produce” events into a shared buffer, terminal or window manager retrieves events from the buffer
  - reader-writer problem: shared data is written by writers while multiple readers read it in parallel -> need to avoid inconsistent data

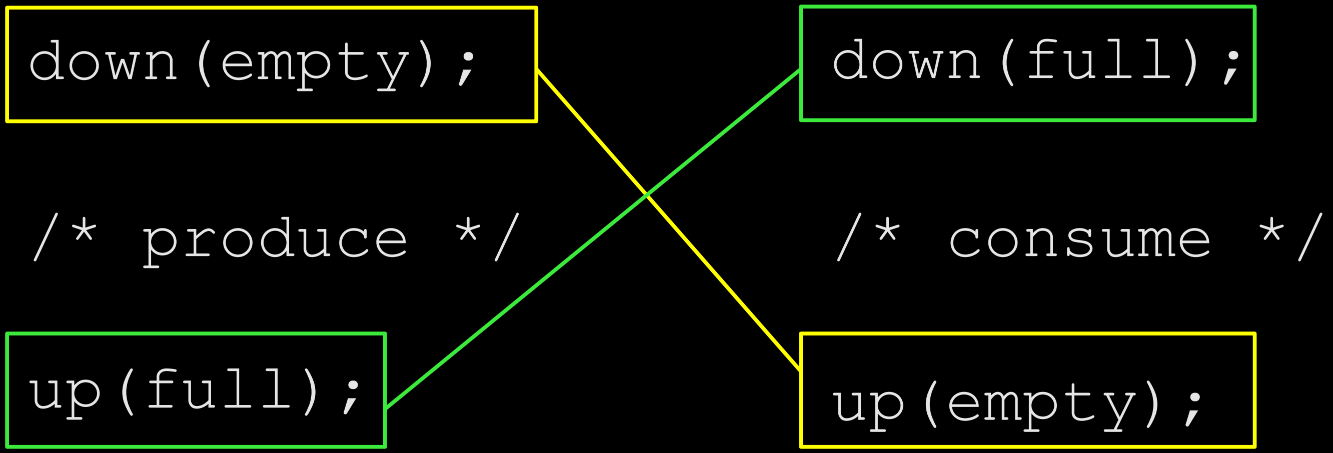
Producer-consumer problem

- fixed buffer size
- producers sleep while no space to fill
- consumers sleep while no data
- consuming invalidates buffer elements
- n:m relation of producers and consumers



Classical solution using semaphores

```
semaphore full(0);
semaphore empty(N);
element_t ringbuffer[N];
```



```
Producer                      Consumer

down(empty);                  down(full);

/* produce */                 /* consume */

up(full);                     up(empty);
```

POSIX Semaphores

- provide system-global and process-local semaphores
- `#include <semaphore.h>`
- `sem_t` datatype
- `sem_init()` - initialize semaphore with a value
- `sem_wait()` - decrement counter, wait if 0 (semaphore_down)
- `sem_post()` - increment counter, wakeup next waiting thread (semaphore_up)

Exercise 7

- Implement a consumer-producer solution using semaphores.
  - Use a ring buffer containing characters.
  - Producer thread writes characters from a string into the buffer.
  - Consumer thread reads string char-wise and prints it after encountering a terminating 0.

Condition variables

- Some people dislike semaphores, because they incorporate multiple concepts:
  - counter
  - synchronization
  - signaling of an event (counter reached a value greater than 0)
- No support for arbitrary signaling conditions.
- Condition variables: objects to synchronize upon
  - sleep on condvar until an event occurs
  - send signals to sleeping thread(s) of a condvar
  - manual checking for conditions

PThread condition variables

- `pthread_cond_t`
- `pthread_cond_init()`
- `pthread_cond_wait()` - sleep until signal occurs
- `pthread_cond_signal()` - wake up one sleeping thread
- `pthread_cond_broadcast()` - wake up **all** sleeping threads

Using condvars

- pthread_cond_t always needs to be combined with a pthread_mutex_t (prevent race condition between sleeping and signalling)

- waiting for an event:

```
pthread_mutex_lock(&mtx);
while (! <condition>)
   pthread_cond_wait(&condvar, &mtx);
pthread_mutex_unlock(&mtx);
```

- signalling an event:

```
if (<condition>)
   pthread_cond_signal(&condvar);
```

C++11: Condition Variables

- #include <condition_variable>

```
std::condition_variable cv;
std::mutex mtx;
bool condition = false;

void waiter() {
   std::unique_lock<std::mutex> ul(mtx);
   cv.wait(ul, []{ return condition });
}
```

C++11: Condition Variables

- #include <condition_variable>

```
std::condition_variable cv;
std::mutex mtx;
bool condition = false;

void creator() {
   std::unique_lock<std::mutex> ul(mtx);
   condition = true;
   cv.notify_all();
}
```

Exercise 8

- Rewrite your semaphore code using condition variables.
  - How many condvars do you need?
  - What are the wakeup conditions?