# CPSC 457

## More synchronization mechanisms

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

# Outline

- monitors, spinlocks, event flags, mailboxes/messages
- mechanisms/algorithms for achieving race-free solutions
  - disabling interrupts
  - lock variables
  - strict alternation
  - Peterson's algorithm
  - synchronization hardware

# Monitors

- a monitor is a higher-level construct compared to mutexes and semaphores
- a monitor is a **programming language construct** that controls access to shared data
  - synchronization code automatically added by a compiler, then enforced at runtime
  - implemented in Concurrent Pascal, C#, D, Modula-3, Java, Ruby, Python, ...
  - can be somewhat emulated in C++ (classes)

    and even to some extent in C (struct + functions or function pointers)

# Monitors

- a monitor is a **module** that encapsulates
  - shared data structures
  - and methods that operate on these shared data structures
  - synchronization is done between concurrent method invocations
- data in monitors can only be accessed via the published methods
- a properly implemented monitor is virtually impossible to use in a wrong way

- another way to look at a monitor:
  - a thread-safe class/object
  - automatic mutual exclusion on every method (via built-in mutex)
  - can include condition variables for signalling conditions

# Monitors

- implement automatic mutual exclusion
- only one thread can execute any monitor method at any time (per instance)
- all other threads would block if they tried
- monitors have their own queues
- Example:

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

← 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();
  }
}
```

# Monitors vs. semaphores and mutexes

- once a monitor 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 monitors → you test/debug the monitor
- programming with mutexes/semaphores → you test/debug all code using them

# Spinlocks

- another synchronization mechanism
- lightweight alternative to mutex
- implemented using busy waiting loops
- usually implemented in assembly, using **atomic operations**
- very efficient if you know the wait time will be very short
  - because no re-scheduling required
  - only make sense on multi-core/CPU systems
- often used inside kernels

**atomic operation** - an operation that appears to execute instantaneously w.r.t. to the rest of the system, i.e. it cannot be *interrupted* by anything else

# Spinlocks

```
mutex m;

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

```
spinlock s;

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

# 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
```

# 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.

# More synchronization mechanisms

- **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

# More synchronization mechanisms

- **message passing**
  - processes send each other **messages**
  - 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)
    - very popular in HPC (high-performance-computing)
    - many good tutorials available (google "MPI tutorial")

# 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 with separate programs

- Scenario - naive print server
  - if a program needs to print something, the program 'submits' the print job by writing a PDF files to the directory `/printJobs`
  - a print server monitors `/printJobs` for jobs to print
    - eg. by scanning the directory for `*.pdf` every N seconds
  - when the server finds a PDF file, it prints it and removes the file
  - if it does not find any PDF files, it sleeps for N seconds
- Can you spot a problem with this?

# Race conditions with separate programs

- Scenario - naive print server
  - if a program needs to print something, the program 'submits' the print job by writing a PDF files to the directory `/printJobs`
  - a print server monitors `/printJobs` for jobs to print
    - eg. by scanning the directory for `*.pdf` every N seconds
  - when the server finds a PDF file, it prints it and removes the file
  - if it does not find any PDF files, it sleeps for n seconds
- Can you spot a problem with this?
  - print server might print an incomplete PDF file
- How do we fix this?
  - we must prevent incomplete PDF files appearing in the directory
  - we need to make sure that when the PDF file appears in the directory, it is complete

# Race conditions with separate programs

- can you design a solution?
- 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
  - file rename

# Race conditions with separate programs

- solution - we modify the programs that print, while print server needs no modification
- when a program needs to print:
  1. it creates a temporary file in **/printJobs** directory, with a random filename, eg. **job-xxxxxx.tmp**, where '**xxxxxx**' is a unique number
  2. it writes the output to this temporary file
  3. when done, it renames **job-xxxxxx.tmp** to **job-yyyyyy.pdf** where '**yyyyyy**' is a another unique number

creating a file with

```
open(pathname,
     O_CREAT|O_EXCL,
     0644)
```

is atomic

- creating random file without a race condition:
  - there is a UNIX utility function `mkstemp()`, although you could make your own easily
- renaming to a random filename — no utility, need to make our own, eg.

```
loop:
  generate random string yyyyyy
  call rename( job-xxxxxx.tmp, job-yyyyyy.pdf) // atomic system call
  if successful, break out of loop
```

# Readers/writers problem

- scenario:
  - a single resource is shared among several threads
  - some threads only read the resource, others only write it
  - the resource supports **multiple concurrent readers**, but only a **single writer**
  - also called **shared-exclusive lock**, or **reader-writer lock**
- can we use semaphores to implement this efficiently?
  - note — pthreads has `pthread_rwlock_*()` functions for just this purpose
- but we can design our own solution using three variables:
  - `int readCount` - contains number of readers reading
  - `semaphore cs` - used as a mutex to control access to critical sections
  - `semaphore w_only` - used as flag representing whether write is available

# Readers/writers problem with semaphores

```
int readcount = 0;
sem_t cs = 1;
sem_t w_only = 1;

writer {
   wait(w_only); // lock out readers
     do_writing();
   signal(w_only); // allow readers
}
```

```
reader {
  wait(cs); // enter critical section
    readCount ++; // add reader
    if (readCount == 1)
      wait(w_only); // wait for writers
  signal(cs); // exit critical section
  do_reading();
  wait(cs); // enter critical section
    readCount --; // remove reader
    if (readCount == 0)
      signal(w_only); // let writers write
  signal(cs); // exit critical section
}
```

Note: first reader locks `w_only`,
last reader releases `w_only`

# Semaphore implemented with mutexes and cond. vars.

```
struct sem {
  pthread_mutex_t mutex;
  int count = 0;
  pthread_cond_t cond;
};
```

```
void down(sem * s) {
  lock(s->mutex);
    while(s->count == 0)
      wait(s->mutex,s->cond);
    s->count --;
  unlock(s->mutex);
}

void up(sem * s) {
  lock(s->mutex);
    s->count ++;
    signal(s->cond);
  unlock(s->mutex);
}
```

# 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 of a process

```
while (1) {
  entry section
    critical section
  exit section
  non-critical section
}
```

# Achieving mutual exclusion

- blocking techniques:
  - mutex, semaphore, monitor, condition variables ✔

- non-blocking techniques:
  - disabling interrupts
  - lock variables
  - strict alternation
  - Peterson's algorithm
  - synchronization hardware
    - spinlocks

# Disabling interrupts

The idea:

- 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.

```
while(1) {
  disable interrupts
    critical section
  enable interrupts
  non-critical section
}
```

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

- A single, 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?

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

# Lock variables

- A single, 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

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

# Lock variables

- A single, shared (lock) variable, initialized to 0.
  - 0 - no process is in CS
  - 1 - a process is in CS
- A process can only enter its CS if lock is 0.

  Otherwise, it must wait.
- **Problem: no mutual exclusion**
- **Solution:**
  - make entering CS atomic operation

    eg. using compare-and-swap
  - or ...

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

# Strict alternation

- two processes alternate entering their critical sections
- global variable: `turn = 0`

```
Thread 1:

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

```
Thread 2:

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

- mutual exclusion is OK
- other problems?

# Strict alternation

- two processes alternate entering their critical sections
- global variable: `turn = 0`

```
Thread 1:

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

```
Thread 2:

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

spinlock

- problem 1: busy waiting (is it always a problem?)
- problem 2: only works for 2 processes
- problem 3: no progress - faster process is blocked by slower process not in its CS

# Peterson's algorithm

- software solution for 2 processes (can be extended to N processes)
  - assumes atomicity, visibility & ordering
  - eg. may fail on some CPUs with out-of-order execution, or memory reordering
- shared integer variable `turn` - indicating whose turn it is
- shared array `flag[2]` - indicating who is interested in entering CS, initialized to **0**

**Process 0:**

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

  non-critical-section
}
```

**Process 1:**

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

  non-critical-section
}
```

# Peterson's algorithm

- software solution for 2 processes (can be extended to N processes)
  - assumes atomicity, visibility & ordering
  - eg. may fail on some CPUs with out-of-order execution, or memory reordering
- shared integer variable `turn` - indicating whose turn it is
- shared array `flag[2]` - indicating who is interested in entering CS, initialized to `0`

**Process 0:**

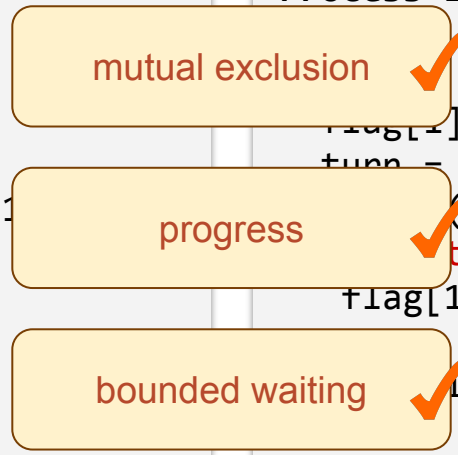
```
while(1) {
  flag[0] = TRUE;
  turn = 1;
  while (flag[1] && turn == 1
     critical_section
   flag[0] = FALSE;

  non-critical-section
}
```

**Process 1:**

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

itical-section
```



# Synchronization hardware

- race conditions are prevented by ensuring that critical sections are protected by locks:
  - a process must acquire a lock before entering a CS
  - a process releases the lock when it exits the CS
- many modern computer systems provide special hardware instructions that implement useful atomic operations
  - these can be used to create atomic locking and unlocking mechanisms
  - e.g. compare-and-swap, test-and-set, swap

# Compare-and-swap (CAS)

- atomic operation used for synchronization
- supported by most CPUs today, eg. CMPXCHG on Intel
- general algorithm:
  - compare contents of memory to val1
  - if they are the same, change the memory to val2
  - return the old contents of memory

must be atomic !

- pseudo-code:

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

must be atomic !

```
int p = 0;

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

# Compare-and-swap in GCC 4.4+

- gcc provides a number of atomic operations, including CAS
- `type __sync_val_compare_and_swap (type *ptr, type oldval, type newval)`
  - atomic compare and swap
  - if the current value of *ptr is oldval, then write newval into *ptr
  - returns the original value of *ptr
- `bool __sync_bool_compare_and_swap (type *ptr, type oldval, type newval)`
  - same as above, but return true if newval was written
- type - can be any 8, 16, 32 and 64 bit integers or pointers

# 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;
}
```

# Test-and-set

- a specialized version of compare-and-swap
  - old hardware had test-and-set
  - newer hardware uses more generalized compare-and-swap
- general algorithm
  - remember contents of memory
  - set memory to true
  - return the old contents of memory

atomic !

- pseudo-code:

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

atomic !

```
int p = 0;

while(1) {
  while( tas(&p) ) {}
    critical section
  p = 0;
  non-critical section
}
```

# Swap

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

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

atomic !

```
int lock = 0;

while(1) {
  int key = TRUE;
  while( key == TRUE ) {
    swap( &lock, &key);
  }
  critical section
  lock = FALSE;
  non-critical section
}
```

# Bounded waiting with synchronization hardware

- when used correctly, the 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', eg. via two shared variables

```
// a CS lock
int lock;
// which processes want to enter CS
int waiting[n];
```

```
int id; // process's ID, starting at 0

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 */
}
```

# 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

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

```
int id; // process's ID, starting at 0

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 */
}
```

# Synchronization hardware

- can be used to implement mutual exclusion, progress, speed and even bounded waiting
- drawbacks:
  - busy-waiting (spinlocks)
  - extra coding (eg. to add bounded waiting)
- advantage:
  - avoids system calls
  - can be more efficient if expected wait time is short
  - only makes sense on multi-CPU/core systems

# Summary

- monitors, spinlocks, event flags, mailboxes/messages
- lock-free solutions:
  - disabling interrupts, lock variables, strict alternation, Peterson's algorithm
  - synchronization hardware: compare-and-swap, test-and-set, swap

- free book !!!

  The Little Book of Semaphores - http://greenteapress.com/wp/semaphores/

Reference: 2.3 (Modern Operating Systems)

5.5, 5.8, 6.5 (Operating System Concepts)

# Review

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

# Questions?