Synchronization

Operating System Design – M1 Info

Instructor: Vincent Danjean
Class Assistant: Florent Bouchez

October 12th, 2015

Outline

Surprising Interleaving

```
int count = 0;

void loop(void *ignored) {
  int i ;
  for (i=0 ; i<10 ; i++)
     count++;
}

int main () {
  tid id = thread_create (loop, NULL);
  loop (); thread_join (id);
  printf("%d",count);
}
```

- **What is the output of this program ?**

Surprising Interleaving

```
int count = 0;

void loop(void *ignored) {
  int i ;
  for (i=0 ; i<10 ; i++)
     count++;
}

int main () {
  tid id = thread_create (loop, NULL);
  loop (); thread_join (id);
  printf("%d",count);
}
```

- **What is the output of this program ?**
  Any value between 2 and 20

Surprising Interleaving

```
int count = 0;

void loop(void *ignored) {
  int i ;
  for (i=0 ; i<10 ; i++)
     count++;
}

int main () {
  tid id = thread_create (loop, NULL);
  loop (); thread_join (id);
  printf("%d",count);
}
```

- **What is the output of this program ?**
  Any value between 2 and 20
- **Furthermore, compiler and hardware optimizations break sequential consistency**

  $\leadsto$ **Need to protect such data and critical sections**

Problem Statement

- **n processes** all **competing** to use some **shared data**
- Each process has a code segment, called **critical section**, in which the shared data is accessed

Problem: Ensure **mutual exclusion**

When one process is executing in its critical section, no other process is allowed to execute in its critical section.

```
do {
  entry section()
    critical section
  exit section()
  remainder section
} while (1);
```

Desired Properties

- **Mutual Exclusion**
  - Only one thread can be in critical section at a time
- **Progress**
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in
- **Bounded waiting**
  - Once a thread T starts trying to enter the critical section, there is a bound on the number of times other threads get in
- **Note progress vs. bounded waiting**
  - If no thread can enter C.S., don't have progress
  - If thread A waiting to enter C.S. while B repeatedly leaves and re-enters C.S. *ad infinitum*, don't have bounded waiting

```
do {
  entry section()
    critical section
  exit section()
  remainder section
} while (1);
```

Desired Properties

- **Mutual Exclusion**
  - Only one thread can be in critical section at a time
- **Progress**
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in
- **Bounded waiting**
  - Once a thread T starts trying to enter the critical section, there is a bound on the number of times other threads get in
- **Note progress vs. bounded waiting**
  - If no thread can enter C.S., don't have progress
  - If thread A waiting to enter C.S. while B repeatedly leaves and re-enters C.S. *ad infinitum*, don't have bounded waiting

```
do {
  acquire lock
     critical section
  release lock
    remainder section
} while (1);
```

Outline

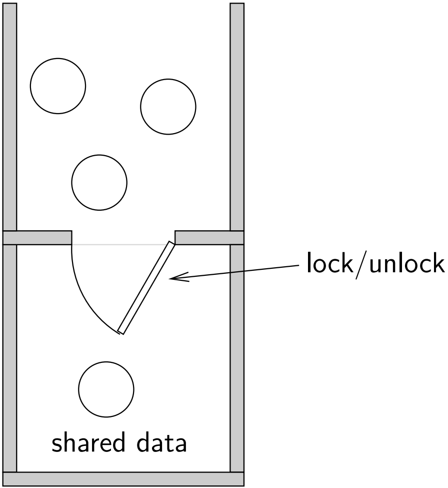
Mutexes

- **Want to insulate programmer from implementing synchronization primitives**
- **Thread packages typically provide mutexes:**

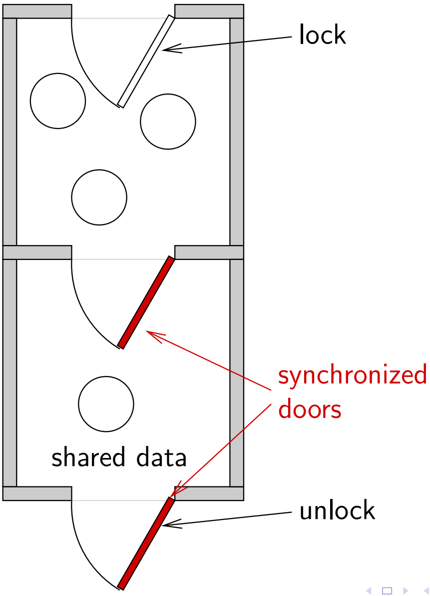
```
void mutex_init (mutex_t *m, ...);
void mutex_lock (mutex_t *m);
int mutex_trylock (mutex_t *m);
void mutex_unlock (mutex_t *m);
```

  - Only one thread acquires `m` at a time, others wait
  - All global data should be protected by a mutex!
- **OS kernels also need synchronization**
  - May or may not look like mutexes

Lock analogy



Lock analogy



Consumer Producer

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */


    while (count == BUFFER_SIZE) {

      /* Do nothing */

    }

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;

  }
}
```

```
void consumer (void *ignored) {
  for (;;) {

    while (count == 0) {

      /* Do nothing */

    }

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;


    /* consume the item in
       nextConsumed */
  }
}
```

Dangerous because of data races

Consumer Producer with Locks

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE) {

      thread_yield ();

    }

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    mutex_unlock (&mutex);
  }
}
```

```


void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    while (count == 0) {

      thread_yield ();

    }

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    mutex_unlock (&mutex);

    /* consume the item in
       nextConsumed */
  }
}
```

What is “wrong” with this solution ?

Consumer Producer with Locks

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and put in
       nextProduced */

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE) {
      mutex_unlock (&mutex);
      thread_yield ();
      mutex_lock (&mutex);
    }

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    mutex_unlock (&mutex);
  }
}
```

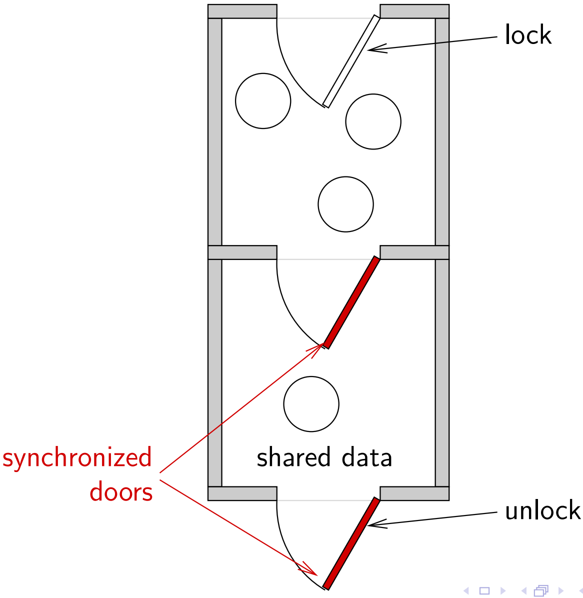
```
void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    while (count == 0) {
      mutex_unlock (&mutex);
      thread_yield ();
      mutex_lock (&mutex);
    }

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    mutex_unlock (&mutex);

    /* consume the item in
       nextConsumed */
  }
}
```

Again... What is “wrong” with this solution ?

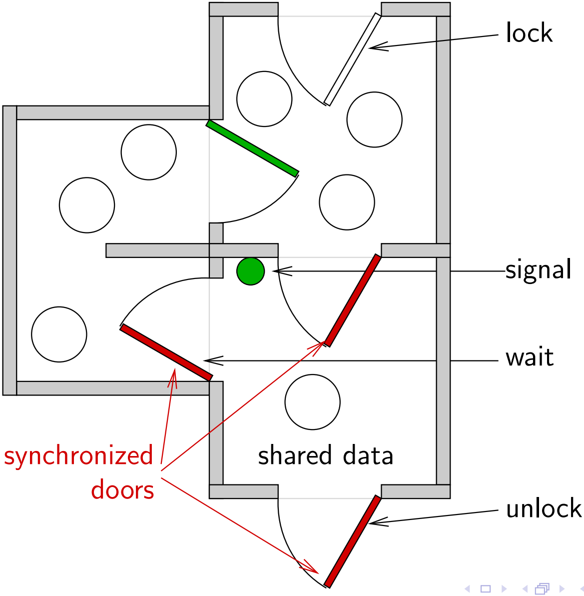
Condition variables

- **Busy-waiting in application is a bad idea**
  - Thread consumes CPU even when can't make progress
  - Unnecessarily slows other threads and processes
- **Better to inform scheduler of which threads can run**
- **Typically done with condition variables**
- `void cond_init (cond_t *, ...);`
  - Initialize
- `void cond_wait (cond_t *c, mutex_t *m);`
  - Atomically unlock `m` and sleep until `c` signaled
  - Then re-acquire `m` and resume executing
- `void cond_signal (cond_t *c);`
  `void cond_broadcast (cond_t *c);`
  - Wake one/all threads waiting on `c`
- **A "condition variable" is a synchronization structure (a queue)**
  It is often associated to a "logical condition" (hence the name)

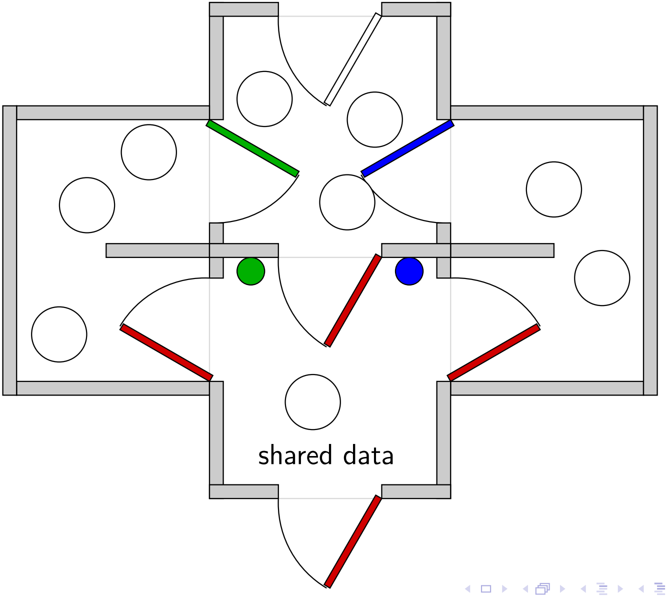
Lock and Condition analogy



Lock and Condition analogy



Lock and Condition analogy



Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and
       put in nextProduced */

    mutex_lock (&mutex);
    if (count == BUFFER_SIZE)
      cond_wait (&nonfull, &mutex);

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    cond_signal (&nonempty);
    mutex_unlock (&mutex);
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    if (count == 0)
      cond_wait (&nonempty, &mutex);

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    cond_signal (&nonfull);
    mutex_unlock (&mutex);

    /* consume the item
       in nextConsumed */
  }
}
```

Does it work with many readers and many writers?

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and
       put in nextProduced */

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE)
      cond_wait (&nonfull, &mutex);

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    cond_signal (&nonempty);
    mutex_unlock (&mutex);
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    while (count == 0)
      cond_wait (&nonempty, &mutex);

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    cond_signal (&nonfull);
    mutex_unlock (&mutex);

    /* consume the item
       in nextConsumed */
  }
}
```

Always put a `while` around the waiting on a condition!

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
  for (;;) {
    /* produce an item and
       put in nextProduced */

    mutex_lock (&mutex);
    while (count == BUFFER_SIZE)
      cond_wait (&nonfull, &mutex);

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
    cond_signal (&nonempty);
    mutex_unlock (&mutex);
  }
}
```

```
void consumer (void *ignored) {
  for (;;) {
    mutex_lock (&mutex);
    while (count == 0)
      cond_wait (&nonempty, &mutex);

    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    cond_signal (&nonfull);
    mutex_unlock (&mutex);

    /* consume the item
       in nextConsumed */
  }
}
```

Beware: this solution does not warrant First Come First Served!

Condition variables (continued)

- **Why must `cond_wait` both release mutex & sleep? Why not separate mutexes and condition variables?**

```
while (count == BUFFER_SIZE) {
  mutex_unlock (&mutex);
  cond_wait(&nonfull);
  mutex_lock (&mutex);
}
```

Condition variables (continued)

- **Why must `cond_wait` both release mutex & sleep? Why not separate mutexes and condition variables?**

```
    while (count == BUFFER_SIZE) {
      mutex_unlock (&mutex);
      cond_wait(&nonfull);
      mutex_lock (&mutex);
    }
```

- **Can end up stuck waiting when bad interleaving**

```
    PRODUCER                         |    CONSUMER
while (count == BUFFER_SIZE){        |
   mutex_unlock (&mutex);            |
                                     | mutex_lock (&mutex);
                                     | ...
                                     | count--;
                                     | cond_signal (&nonfull);
   cond_wait (&nonfull);             |
```

- **With mutex and conditions, we can implement safe (no data race) and efficient (no busy waiting) synchronization)**
  - We will see soon that busy waiting has actually not completely disappeared

Common Synchronization Problems

Deadlock P_0 and P_1 both want to access S and Q that are each protected by a lock:

P_0	P_1
lock(S.lock)	lock(R.lock)
lock(R.lock)	lock(S.lock)
...	...
unlock(R.lock)	unlock(S.lock)
unlock(S.lock)	unlock(R.lock)

Starvation There may be indefinite blocking (e.g., if resuming in LIFO order, if select in priority process that are not swapped, ...)

Priority Inversion Assume we have three process L, M, H with priority $L < M < H$.

1. L acquires resource R
2. H tries to acquire R and is thus blocked
3. M becomes runnable and thereby preempts L
4. Hence, M affects how long H must wait for R

This priority inversion occurs only in systems with more than two priorities. Can be solved by implementing **priority inheritence**

Other thread package features

- **Alerts – cause exception in a thread**
- **Timedwait – timeout on condition variable**
- **Shared locks – concurrent read accesses to data**
- **Thread priorities – control scheduling policy**
  - Mutex attributes allow various forms of *priority donation*

- **Thread-specific global data**
- **Different synchronization primitives (in a few slides)**
  - Monitors
  - Semaphores

Outline

Implementing synchronization

- **User-visible mutex is a straight-forward data structure**

```
typedef struct mutex {
  bool is_locked;          /* true if locked */
  thread_id_t owner;       /* thread holding lock, if locked */
  thread_list_t waiters;   /* threads waiting for lock */

};
```

```
mutex_lock(struct mutex *L) {
  if(L->is_locked) {
    add myself to L->waiters;
    block();
  } else
    L->owner = myself;
  L->is_locked = true;
}
```

```
mutex_unlock(struct mutex *L) {
  pick P from L->waiters;
  L->owner = P;
  L->is_locked = false;
  wakeup(P)
}
```

Implementing synchronization

- **User-visible mutex is a straight-forward data structure**

```
typedef struct mutex {
  bool is_locked;          /* true if locked */
  thread_id_t owner;       /* thread holding lock, if locked */
  thread_list_t waiters;   /* threads waiting for lock */
  lower_level_lock_t lk;   /* Protect above fields */
};
```

```
mutex_lock(struct mutex *L) {
  if(L->is_locked) {
    add myself to L->waiters;
    block();
  } else
    L->owner = myself;
  L->is_locked = true;
}
```

```
mutex_unlock(struct mutex *L) {
  pick P from L->waiters;
  L->owner = P;
  L->is_locked = false;
  wakeup(P)
}
```

- **Need lower-level lock `lk` for mutual exclusion**
  - Internally, `mutex_*` functions bracket code with `lock(mutex->lk)` . . . `unlock(mutex->lk)`
  - Otherwise, data races! (E.g., two threads manipulating `waiters`)
- **How to implement `lower_level_lock_t`?**

Outline

Approach #1: Disable interrupts

- **Only for apps with $n:1$ threads (1 kthread)**
  - On a multiprocessor, the disable interrupt message has to be passed to all processors, which delays entry into each critical section and decreases efficiency.
  - But sometimes most efficient solution for uniprocessors
- **Trick: Masking interrupts is costly. Instead, have per-thread "do not interrupt" (DNI) bit**
- `lock (lk)`: **sets thread's DNI bit**
- **If timer interrupt arrives**
  - Check interrupted thread's DNI bit
  - If DNI clear, preempt current thread
  - If DNI set, set "interrupted" (I) bit & resume current thread
- `unlock (lk)`: **clears DNI bit and checks I bit**
  - If I bit is set, immediately yields the CPU

Outline

Approach #2: Spinlocks

- **Most CPUs support atomic read-[modify-]write**
- **Example:** `int test_and_set (int *lockp);`
  - Atomically sets `*lockp = 1` and returns old value
  - Special instruction – can't be implemented in portable C
- **Use this instruction to implement spinlocks:**

```
#define lock(lock)    while (test_and_set (&lock))
#define trylock(lock) (test_and_set (&lock) == 0)
#define unlock(lock)  lock = 0
```

```
do {
  while(test_and_set(&lock))
      ;
  critical section
  lock = 0;
  remainder section
} while (1);
```

Approach #2: Spinlocks

- **Most CPUs support atomic read-[modify-]write**
- **Example:** `int test_and_set (int *lockp);`
  - Atomically sets `*lockp = 1` and returns old value
  - Special instruction – can't be implemented in portable C
- **Use this instruction to implement spinlocks:**

```
#define lock(lock)     while (test_and_set (&lock))
#define trylock(lock)  (test_and_set (&lock) == 0)
#define unlock(lock)   lock = 0
```

```
do {
  while(test_and_set(&lock))
     ;
  critical section
  lock = 0;
  remainder section
} while (1);
```

- **Satisfies Mutual Exclusion and Progress but not Bounded Waiting**

Bounded Waiting with Test and Set

```
volatile boolean waiting[n]; // Initialized to 0
volatile boolean lock;       // Initialized to 0

do {
  int islocked = 1;
  waiting[i] = 1;
  while (waiting[i] && islocked)
    islocked = test_and_set(&lock);
  waiting[i] = 0;

  critical section

  // look for next process to free
  j = (i+1)%n;
  while((j!=i) && (!waiting[j]))
    j = (j+1)%n;
  // free process j
  if(j==i) lock = 0;
  else waiting[j] = 0;

  remainder section
} while(1);
```

Bounded Waiting with Test and Set

```
volatile boolean waiting[n]; // Initialized to 0
volatile boolean lock;       // Initialized to 0

do {
  int islocked = 1;
  waiting[i] = 1;
  while (waiting[i] && islocked)   // Do we really need islocked ?
    islocked = test_and_set(&lock);
  waiting[i] = 0;

  critical section

  // look for next process to free
  j = (i+1)%n;
  while((j!=i) && (!waiting[j]))
    j = (j+1)%n;
  // free process j
  if(j==i) lock = 0;
  else waiting[j] = 0;

  remainder section
} while(1);
```

Mutex vs. Spinlocks

- **Can you use spinlocks instead of mutexes?**
  - On x86, requires the CPU to lock memory system around read and write
    - Prevents other uses of the bus (e.g., DMA)
  - Usually runs at memory bus speed, not CPU speed
    - Much slower than cached read/buffered write
    - Causes ping-pong cache line migration
  - Wastes CPU, especially if thread holding lock not running
- **Spinlocks are often used by the kernel as a low-level mutex to implement a higher level mutex**
  - **Spinlocks** = busy waiting but the only way to really implement mutual exclusion
  - When the Critical Section is long, this is extremely inefficient
  - Higher-level **mutex** = "few" busy waiting
  - With mutex, busy waiting is limited to the CSs of `lock`,`unlock`,`wait`, `signal`, which are short sections (about ten instructions).
  - Therefore, the CS os mutexes is almost never occupied and busy-waiting occurs rarely and only for a short time
  - On multiprocessor, sometimes good to spin for a bit, then yield

Outline

Kernel Synchronization

- **Should kernel use locks or disable interrupts?**
- **Old UNIX had non-preemptive threads, no mutexes**
  - Interface designed for single CPU, so `count++` etc. not data race
- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**

Kernel Synchronization

- **Should kernel use locks or disable interrupts?**
- **Old UNIX had non-preemptive threads, no mutexes**
  - Interface designed for single CPU, so `count++` etc. not data race
- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**
  - Yes! Can't sleep in interrupt handler, so can't wait for lock
  - So even modern OSes have support for disabling interrupts
  - Often uses DNI trick, which is cheaper than masking interrupts in hardware
- **Modern OS provide both mutex, spinlocks, ability to disable/enable interrupts, and even more (futex,).**

Outline

A few classical synchronization problems

Producer/Consumer

- the bounded-buffer problem, where consumers (resp. producers) need the buffer to be non-empty (resp. non-full) to proceed
- need to work with many producers and many consumers (then the `while` around the `wait(condition, mutex)` is mendatory)
- may or not respect FIFO order

Bank Account

- a bank with the two functions `deposit(amount, account)`, `withdraw(amount, account)`
- a shared bank account where concurrently, the husband calls `withdraw` and the wife calls `deposit`

Reader/Writer

- the same shared memory is concurrently accessed by threads, some reading and some writing
- writers require exclusive access
- there may be multiple readers at the same time
- readers-preference, writers-preference, no thread shall be allowed to starve

Dining Philosophers

- N philophers eating rice around a round table
- only N chopsticks and philophers need to pick left and right chopsticks to eat
- avoid deadlock
- avoid starvation

A few classical synchronization problems (cont'd)

Cigarette Smokers

- you need paper, tobacco, and a match to make a cigarette
- 3 smokers around a table each with an infinite supply of *one* of the three ingredients
- an arbiter randomly select two smokers, takes one item out of their supply and puts it on the table, which enables the third smoker to smoke
- a smoker only begins to roll a new cigarette once he has finished smoking the last one

Sleeping Barber

- a barbershop: a waiting room with n chairs, a barber room with 1 barber
- if there is no customer to be served, the barber takes a nap
- when a customer enters the shop, if all chairs are occupied, then the customer leaves the shop (unshaved!)
- if the barber is busy but chairs are available, then the customer sits on one of the free chairs
- if the barber is asleep, the customer wakes up the barber

Traffic Lights and Intersection

- traffic lights of both ways are synchronized (using red/orange/green helps)
- each car spends a finite (random) time at the intersection
- only cars from of a given way (but both directions) can be in the intersection at the same time

$\rightsquigarrow$ need to forbid some cars to enter the intersection when the light needs to be switched but also needs to let the cars leave the intersection befor switching

- the intersection may contain up to k cars for each direction

$\rightsquigarrow$ need to forbid some cars to enter the intersection

Summary

- Implementing the three versions of the reader-writer problem will be your next programming assignment
- Work the others by yourself
- Often you'll realize the problem needs some extra specification
- In all these problems, fairness (avoiding starvation) may decrease throughput

Outline

Introduction

- **Spinlocks, futex, or signal handling are special “kernel” synchronization mechanisms**
- **Mutex locks and conditions are standard synchronization structures available in any thread library**
- **There are a few other standard synchronization structures that I will present now**

Outline

Semaphore

- Thread packages typically provide *semaphores*.
- A *Semaphore* is initialized with an integer N `void sem_init (sem_t *s, unsigned int value);`
  `void sem_post (sem_t *s);` (originally called V)
  `void sem_wait (sem_t *);` (originally called P)
  `void sem_trywait (sem_t *);`
  `void sem_getvalue (sem_t *);`
- Think of a semaphore as a purse with a certain number of tokens

```
sem_wait(sem_t S) {
  S->value--;
  if(S->value < 0) {
    add myself to S->waiters;
    block();
  }
}
```

```
sem_post(sem_t S) {
  S->value++;
  if(S->value <= 0) {
    pick P from S->waiters
    wakeup(P)
  }
}
```

- Remember real implementations require lower level locking (e.g., spinlocks or interruption management)

Semaphores vs. Mutex

- If $N == 1$, then semaphore is a mutex with `sem_wait` as lock and `sem_post` as unlock
  Yet, is there a difference between a binary semaphore and a lock?
- Could use semaphores to implement conditions. Yet, is there a difference between a post and a signal?
- Can re-write producer/consumer to use three semaphores
- Semaphore `mutex` initialized to 1
  - Used as mutex, protects buffer, in, out...
- Semaphore `full` initialized to 0 ($\approx$ number of items)
  - To block consumer when buffer empty
- Semaphore `empty` initialized to N ($\approx$ number of free locations)
  - To block producer when queue full

Consumer Producer with Semaphores

```
    void producer (void *ignored) {
        for (;;) {
            /* produce an item and put in nextProduced */
            sem_wait (&empty);
            sem_wait (&mutex);
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            sem_post (&mutex);
            sem_post (&full);
        }
    }

    void consumer (void *ignored) {
        for (;;) {
            sem_wait (&full);
            sem_wait (&mutex);
            nextConsumed =  buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            sem_post (&mutex);
            sem_post (&empty);
            /*  consume the item in nextConsumed */
        }
    }
```

Semaphores vs. Mutex

- Semaphores can implement Mutex and vice-versa
- Semaphores allow elegant solutions to some problems (producer/consumer, reader/writer)
- *"One structure to rule them all"* ☺
- Yet...

Semaphores vs. Mutex

- Semaphores can implement Mutex and vice-versa
- Semaphores allow elegant solutions to some problems (producer/consumer, reader/writer)
- *"One structure to rule them all"* ☺
- Yet. . . they are quite error prone
  - If you call `wait` instead of `post`, you'll have a deadlock.
  - If you forgot to protect parts of your code, then you may end up either with a deadlock or a mutual exclusion violation
  - You have "tokens" of different types, which may be harder to reason about
  - **If by mistake you interchange the order of the `wait` and the `post`, you may violate mutual exclusion in an unreproducable way.**
- That is why people have proposed higher-level language constructs

Outline

Monitors [BH][Hoar]

- **Programming language construct**
  - Possibly less error prone than raw mutexes, but less flexible too
  - Basically a class where only one procedure executes at a time

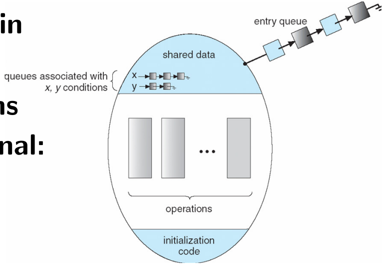
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (...) { ... }
  ...
  procedure Pn (...) { ... }

  Initialization code (..) { ... }
}
```

- **Can implement mutex w. monitor or vice versa**
  - But monitor alone doesn't give you condition variables
  - Need some other way to interact w. scheduler
  - Use *conditions*, which are essentially condition variables

Monitor implementation

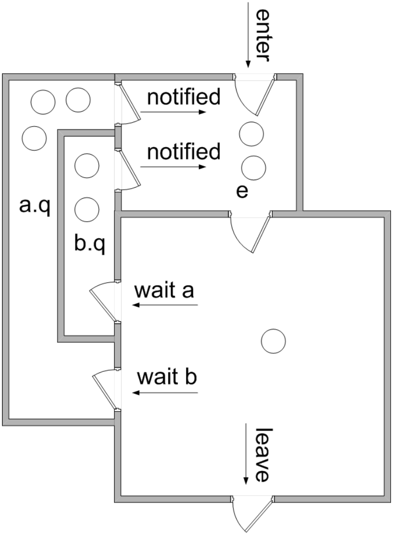
- **Queue of threads waiting to get in**
  - Might be protected by spinlock
- **Queues associated with conditions**
- **Two possibilities exist for the signal:**
  - Signal and wait
  - Signal and continue

  Both have pros and cons

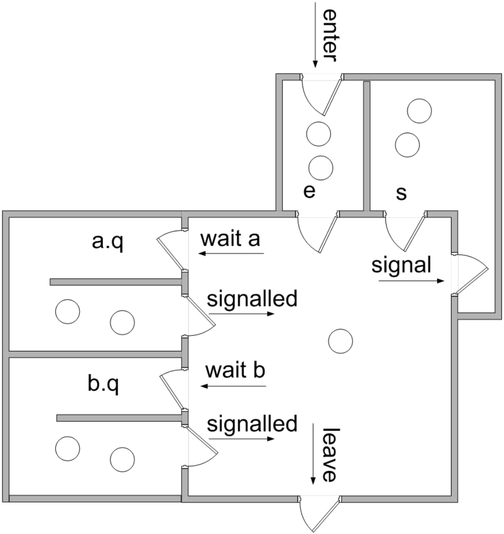


- **Java provide monitors: just add the synchronized keyword. Locks are automatically acquired and release. They have only one waiting queue though.**

Signal and continue analogy



Signal and wait analogy



Recap

- Synchronization structures enable to keep out of race conditions and optimizations breaking sequential consistency
- They may lead to deadlocks and starvation
- Enables to save resources if well used (avoid busy waiting as much as possible)
- Your critical sections should be as short as possible to avoid poor resource usage
- The tradeoff over-protecting/loosening mutual exclusion affects the tradeoff fairness/throughput
- All thread libraries provide Mutexes (locks and conditions) and semaphores. Some languages provide even higher level constructs like monitors.
- Efficient implementations require a lots of different non-standard tricks and tradeoffs (spinlock, interruption, user/kernel space,...)

Outline

The deadlock problem

```
mutex_t m1, m2;

void p1 (void *ignored) {
  lock (m1);
  lock (m2);
  /* critical section */
  unlock (m2);
  unlock (m1);
}

void p2 (void *ignored) {
  lock (m2);
  lock (m1);
  /* critical section */
  unlock (m1);
  unlock (m2);
}
```

- **This program can cease to make progress – how?**
- **Can you have deadlock w/o mutexes?**

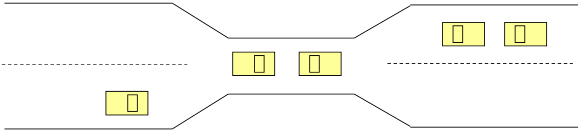
More deadlocks

- **Same problem with condition variables**
  - Suppose resource 1 managed by c_1, resource 2 by c_2
  - A has 1, waits on $c2$, B has 2, waits on $c1$
- **Or have combined mutex/condition variable deadlock:**

```
- lock (a); lock (b); while (!ready) wait (b, c);
  unlock (b); unlock (a);
- lock (a); lock (b); ready = true; signal (c);
  unlock (b); unlock (a);
```

- **One lesson: Dangerous to hold locks when crossing abstraction barriers!**
  - I.e., `lock (a)` then call function that uses condition variable

Deadlocks w/o computers



- **Real issue is resources & how required**
- **E.g., bridge only allows traffic in one direction**
  - Each section of a bridge can be viewed as a resource.
  - If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
  - Several cars may have to be backed up if a deadlock occurs.
  - Starvation is possible.

Deadlock conditions

1. **Limited access (mutual exclusion):**
   - Resource can only be shared with finite users.
2. **No preemption:**
   - once resource granted, cannot be taken away.
3. **Multiple independent requests (hold and wait):**
   - don't ask all at once (wait for next resource while holding current one)
4. **Circularity in graph of requests**

- **All of 1–4 necessary for deadlock to occur**
- **Two approaches to dealing with deadlock:**
  - pro-active: prevention
  - reactive: detection + corrective action

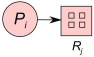
Outline

Prevent by eliminating one condition

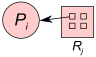
1. **Limited access (mutual exclusion):**
   - Buy more resources, split into pieces, or virtualize to make "infinite" copies.
2. **No preemption:**
   - Threads: threads have copy of registers = no lock
   - Physical memory: virtualized with VM, can take physical page away and give to another process!
   - You can preempt resources whose state can be easily saved and restored later
3. **Multiple independent requests (hold and wait):**
   - Wait on all resources at once (must know in advance, bad resource usage, starvation, ...)
4. **Circularity in graph of requests**
   - Single lock for entire system: (problems?)
   - Partial ordering of resources (next)

Resource-allocation graph

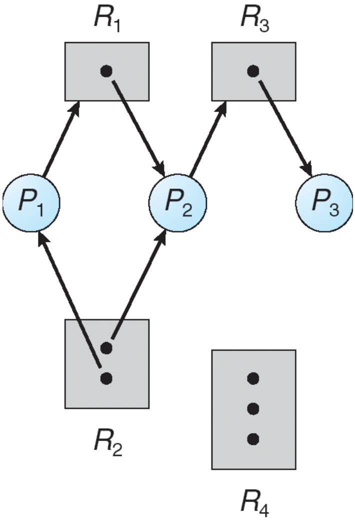
- **View system as graph**
  - Processes and Resources are nodes
  - Resource Requests and Assignments are edges
- **Process:** 
- **Resource w. 4 instances:** 
- P_i **requesting** R_j**:**



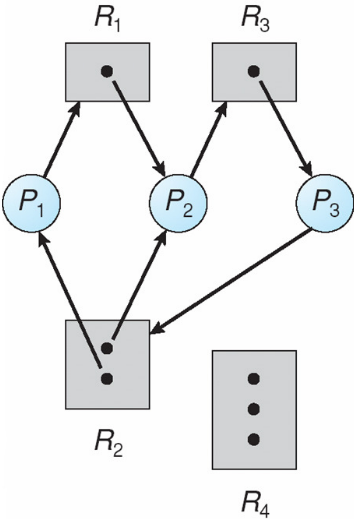
- P_i **holding instance of** R_j**:**



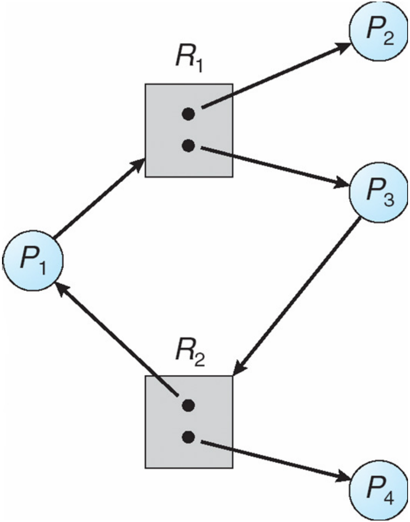
Example resource allocation graph



Graph with deadlock



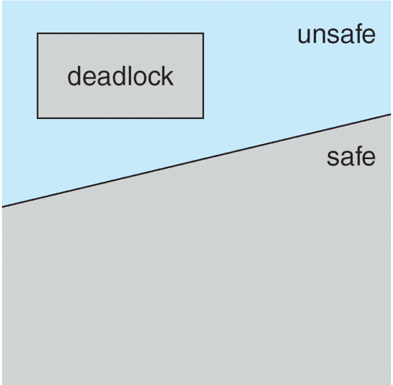
Is this deadlock?



Cycles and deadlock

- **If graph has no cycles $\implies$ no deadlock**
- **If graph contains a cycle**
  - Definitely deadlock if only one instance per resource
  - Otherwise, maybe deadlock, maybe not
- **Prevent deadlock w. partial order on resources**
  - E.g., always acquire mutex m_1 before m_2
  - Usually design locking discipline for application this way

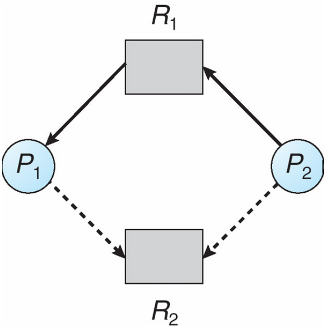
Prevention



- **Determine safe states based on possible resource allocation**
- **Conservatively prohibits non-deadlocked states**

Claim edges

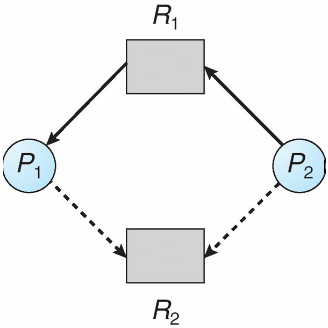
- **Dotted line is claim edge**
  - Signifies process *may* request resource
- **Process should claim edges all at once before requisting them.**



- **Upon request, transform claimed edge into an assignment edge only if does does not create a cycle ($O(n^2)$ algorithm).**
  Otherwise, make it a request edge (even if you "could" allocate resource).

Example: unsafe state

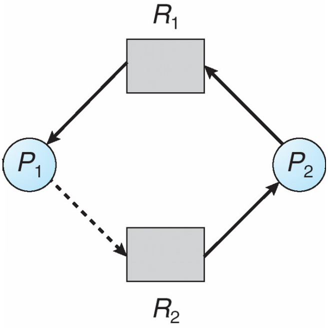
- **Assume that P_1 requests R_2 (**Figure is wrong and should be fixed for next year**).**



- **Note cycle in graph**
  - P_1 might request R_2 before relinquishing R_1
  - Would cause deadlock
- **This techniques works only for systems with a single instance of each resource type. Other situations need more elaborate algorithms (e.g., banker's algorithm)**

Example: unsafe state

- **Assume that P_1 requests R_2 (**Figure is wrong and should be fixed for next year**).**

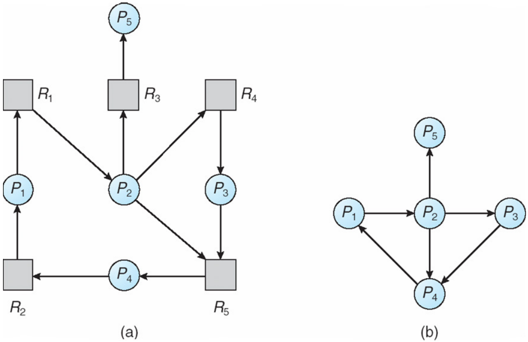


- **Note cycle in graph**
  - P_1 might request R_2 before relinquishing R_1
  - Would cause deadlock
- **This techniques works only for systems with a single instance of each resource type. Other situations need more elaborate algorithms (e.g., banker's algorithm)**

Outline

Detecting deadlock

- **Static approaches (hard)**
- **Program grinds to a halt**
- **Threads package can keep track of locks held:**



Resource-Allocation Graph   Corresponding wait-for graph

- **Again, this techniques works only for systems with a single instance of each resource type. Otherwise use extensions of the banker's algorithm**

Fixing & debugging deadlocks

Detection is costly $\rightsquigarrow$ when ? how often ? is it really worth the effort ? . . .

- **Reboot system (windows approach)**
- **Examine hung process with debugger**
- **Threads package can deduce partial order**
  - For each lock acquired, order with other locks held
  - If cycle occurs, abort with error
  - Detects *potential* deadlocks even if they do not occur
- **Or use transactions. . .**
  - Another paradigm for handling concurrency
  - Often provided by databases, but some OSes use them
  - *Vino* OS used transactions to abort after failures [Seltzer]
  - OS support for transactional memory now hot research topic

Transactions

- **A transaction T is a collection of actions with**
  - *Atomicity* – all or none of actions happen
  - *Consistency* – T leaves data in valid state
  - *Isolation* – T's actions all appear to happen before or after every other transaction T'
  - *Durability** – T's effects will survive reboots
  - Often hear mnemonic *ACID* to refer to above
- **Transactions typically executed concurrently**
  - But *isolation* means must *appear* not to
  - Must roll-back transactions that use others' state
  - Means you have to record all changes to undo them
- **When deadlock detected just abort a transaction**
  - Breaks the dependency cycle
- **Most Lock-free algorithms rely on atomic read-modify-write primitives.**
  Software transactional memories promises standard abstractions for writing efficient non-blocking code.