CSE120
Principles of Operating Systems

Prof Yuanyuan (YY) Zhou
Synchronization: Semaphore
Synchronization Needs

- Two synchronization needs
  - **Mutual exclusion**
    - Whenever multiple threads access a shared data, you need to worry about “protection” for mutual exclusion
  - **Coordination** (one wait for the other to finish something, e.g. produce the data, free the buffer, etc)
typedef struct __lock_t {
  int flag;
  int guard;
  queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
  m->flag = 0;
  m->guard = 0;
  queue_init(m->q);
}

void lock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1) {
    // acquire guard lock by spinning
    if (m->flag == 0) {
      m->flag = 1; // lock is acquired
      m->guard = 0;
    } else {
      queue_add(m->q, gettid());
      m->guard = 0;
      park();
    }
  }
}

void unlock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1) {
    // acquire guard lock by spinning
    if (queue_empty(m->q))
      m->flag = 0; // let go of lock; no one wants it
    else
      unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
  }
}
Higher-Level Synchronization

- We looked at using locks to provide mutual exclusion.
- Those locks work, but they have some drawbacks when critical sections are long:
  - Spinlocks – inefficient.
- Instead, we want synchronization mechanisms that:
  - Block waiters.
  - Leave interrupts enabled inside the critical section.
- Look at two common high-level mechanisms:
  - Semaphores: binary and counting.
  - Monitors and condition variables.
- Use them to solve common synchronization problems.
Semaphores

- Semaphores are an **abstract data type** that provide mutual exclusion to critical sections

- Semaphores can also be used as atomic counters
  - More later

- Semaphores are **integers** that support two operations:
  - **P(semaphore)**: decrement, block the calling thread until semaphore is open, i.e. the integer is greater than 0
    - Also **Wait()**, or down()
  - **V(semaphore)**: increment, allow another thread to enter. If there is a thread is blocked, wake up this thread.
    - Also **Signal()**, or up()
  - That's it! No other operations – not even just reading its value – exist

- Semaphore safety property: the semaphore value is always greater than or equal to 0
Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes.
- When \( P() \) is called by a thread:
  - If semaphore is open (\( >0 \)), thread continues
  - If semaphore is closed (\( ==0 \)), thread blocks on queue
- Then \( V() \) opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread by increment the counter
- In other words, \( V() \) has “history” (c.f., condition vars later)
- This “history” is a counter
Semaphore Functionality (NOT implementation)

P(Semaphore s)

    If (s==0) blocked in a queue;  /* wait until s>0 */
    s=s-1;

}

V(Semaphore s)

    s=s+1;
    if (someone is waiting in a queue) wakeup one from the queue;

}

Init(Semaphore s, int v)

    s=v;

}
Semaphore Types

- Semaphores come in two types
- **Binary** semaphore (value can be only 1 or 0, some referred it as mutex)
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
    - similar to locks with a subtle difference: the former can “remember” when you do $V(sem)$ when $sem=0$ vs do unlock(l) if no thread has the lock
- **Counting** semaphore
  - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore
  - Number of threads determined by the semaphore “count”
- You can use one type to implement the other
Counter Semaphore

2 lanes

B. Komazec, 2012
Semaphore Animation Video

- [https://www.youtube.com/watch?v=PQ5aK5wLCQE](https://www.youtube.com/watch?v=PQ5aK5wLCQE)

- In this video, wait() is P() and signal is V()
Using Semaphores

- Mutex is similar to our locks, but semantics are different

```c
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    P(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    V(S);
    return balance;
}
```

Threads block

P(S);
balance = get_balance(account);
balance = balance – amount;
V(S);

critical section

put_balance(account, balance);
V(S);

It is undefined which thread runs after a signal

... V(S);

... V(S);

CSE 120 – Synchronization Semaphore
Exercise

- Using semaphores to allow robots to attend an exam:
  - Only 10 seats, but 100 robots
  - If a robot comes to the classroom, if there is an available seat, it takes the seat; otherwise, wait outside unless another robot leaves the room;
  - Every robot sits in the seat for only 30min to finish the exam, and then leaves the room
  - Implement the code (steps) for every robot to follow
Classic Synchronization problems

- We’ve looked at a simple example for using synchronization
  - Mutual exclusion while accessing a bank account
- Now we’re going to use semaphores to look at more interesting examples
  - Readers/Writers
  - Bounded Buffers
  - Santa clause problem (youtube video)
Readers/Writers Problem

- Readers/Writers Problem:
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers but only one writer

- How can we use semaphores to control access to the object to implement this protocol?
- Use three variables
  - int readcount – number of threads reading object
  - Semaphore mutex – control access to readcount
  - Semaphore w_or_r – exclusive writing or reading
Semaphore w_or_r=1;

Reader{
    P(w_or_r); // lock out writers
    read;
    V(w_or_r); // up for grabs
}

writer {
    P(w_or_r); // lock out readers
    Write;
    V(w_or_r); // up for grabs
}

Does it work?
Why?
Semaphore w_or_r=1;
int readcount; //record #readers

Reader{
    readcount++;
    if (readcount == 1){
        P(w_or_r); // lock out writers
    } 
    read;
    readcount--; 
    if (readcount == 0){
        V(w_or_r); // up for grabs
    }
}

writer {
    P(w_or_r); // lock out readers
    Write;
    V(w_or_r); // up for grabs
}
Readers/Writers Real Solution

- Use three variables
  - int readcount – number of threads reading object
  - Semaphore mutex – Guard access to readcount
  - Semaphore w_or_r – exclusive writing or reading
Readers/Writers

// number of readers
readcount = 0;

// mutual exclusion to readcount
Semaphore mutex = 1;

// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    P(w_or_r); // lock out readers
    Write;
    V(w_or_r); // up for grabs
}

reader {
    P(mutex); // lock readcount
    readcount ++; // one more reader
    if (readcount == 1)
        P(w_or_r); // synch w/ writers
    V(mutex); // unlock readcount
    Read;
    P(mutex); // lock readcount
    readcount --; // one less reader
    if (readcount == 0)
        V(w_or_r); // up for grabs
    V(mutex); // unlock readcount
}

I will give you 2-3 minutes to discuss it with someone next to you
w_or_r provides mutex between readers and writers, and also between multiple writers.

Why do readers use mutex (binary semaphore)?

What if V() is above “if (readcount == 1)”?

Why do we need “if(readcount==1)”?

Why do we need “if(readcount==0)”?
But it still has a problem

// number of readers
readcount = 0;

// mutual exclusion to readcount
Semaphore mutex = 1;

// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    P(w_or_r); // lock out readers
    Write;
    V(w_or_r); // up for grabs
}

reader {
    P(mutex); // lock readcount
    readcount ++; // one more reader
    if (readcount == 1)
        P(w_or_r); // synch w/ writers
    V(mutex); // unlock readcount
    Read;
    P(mutex); // lock readcount
    readcount --; // one less reader
    if (readcount == 0)
        V(w_or_r); // up for grabs
    V(mutex); // unlock readcount
}
Problem: Starvation

- What if a writer is waiting, but readers keep coming, the writer is starved
Bounded Buffer

- **Problem:** There is a set of resource buffers shared by producer and consumer threads
  - **Producer** inserts resources into the buffer set
    - Output, disk blocks, memory pages, processes, etc.
  - **Consumer** removes resources from the buffer set
    - Whatever is generated by the producer

- **Producer and consumer execute at different rates**
  - No serialization of one behind the other
  - Tasks are independent (easier to think about)
  - The buffer set allows each to run without explicit handoff
Bounded Buffer (2)

- Use three semaphores:
  - **empty** – count of empty buffers
    - Counting semaphore
    - empty = N – (np-nc)
  - **full** – count of full buffers
    - Counting semaphore
    - np - nc = full
  - **mutex** – mutual exclusion to shared set of buffers
    - Binary semaphore
Bounded Buffer (3)

Semaphore mutex = 1;  // mutual exclusion to shared set of buffers
Semaphore empty = N;  // count of empty buffers (all empty to start)
Semaphore full = 0;    // count of full buffers (none full to start)

producer {
    while (1) {
        Produce new resource;
        P(empty);  // wait for empty buffer
        P(mutex);  // lock buffer list
        Add resource to an empty buffer;
        V(mutex);  // unlock buffer list
        V(full);   // note a full buffer
    }
}

c consumer {
    while (1) {
        P(full);   // wait for a full buffer
        P(mutex);  // lock buffer list
        Remove resource from a full buffer;
        V(mutex);  // unlock buffer list
        V(empty);  // note an empty buffer
        Consume resource;
    }
}
Bounded Buffer (4)

Consumer decrements FULL and
Blocks when buffer has no item
Since the semaphore FULL is at 0

Producuer decrements EMPTY
and blocks when buffer is full
since the semaphore is at 0
Bounded Buffer (5)

- Why need the mutex at all?
- Where are the critical sections?

- What happens if operations on mutex and full/empty are switched around?
  - The pattern of P/V on full/empty is a common construct often called an interlock

- Why V(full) and V(empty)?

- Producer-Consumer and Bounded Buffer are classic examples of synchronization problems
Youtube Video for Bounded Buffer

- https://www.youtube.com/watch?v=GvfjiA9jkTs
Possible Deadlocks with Semaphores

Example:

P0
share two mutex semaphores S and Q
S:= 1; Q:=1;

P(S);                  P(Q);
P(Q);                  P(S);

…..                            ……

V(S);                    V(Q);
V(Q);                    V(S);
Be Careful When Using Semaphores

// Violation of Mutual Exclusion
V(mutex);
critical section
P(mutex);

// Deadlock Situation
P(mutex);
critical section
P(mutex);

// Violation of Mutual Exclusion
critical section
V(mutex);
Semaphore Summary

- Semaphores can be used to solve any of the traditional synchronization problems
- However, they have some drawbacks
  - They are essentially shared global variables
    - Can potentially be accessed anywhere in the program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
    - Note that I had to use comments in the code to distinguish
  - No control or guarantee of proper usage
- Sometimes hard to use and prone to bugs
Monitors

• A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why is this an advantage?

• A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures

• A monitor protects its data from unsynchronized access

• It guarantees that threads accessing its data through its procedures interact only in legitimate ways
Monitor Semantics

- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    - So the monitor has to have a wait queue...
  - If a thread within a monitor blocks, another one can enter

- What are the implications in terms of parallelism in monitor?
Account Example

Monitor account {
    double balance;
    double withdraw(amount) {
        balance = balance – amount;
        return balance;
    }
}

- Hey, that was easy
- But what if a thread wants to wait inside the monitor?

When first thread exits, another can enter. Which one is undefined.

Threads block waiting to get into monitor

withdraw(amount)
    balance = balance – amount;
withdraw(amount)
withdraw(amount)
return balance (and exit)
balance = balance – amount
return balance;
balance = balance – amount
return balance;
A condition variable is associated with a condition needed for a thread to make progress

Monitor M {
    ... monitored variables
    Condition c;

    void enter_mon (...) {
        if (extra property not true) wait(c);  \ waits outside of the monitor's mutex
        do what you have to do
        if (extra property true) signal(c);  \ brings in one thread waiting on condition
    }
}
Condition Variables

- Condition variables support three operations:
  - **Wait** – release monitor lock, wait for C/V to be signaled
    - So condition variables have wait queues, too
  - **Signal** – wakeup one waiting thread
  - **Broadcast** – wakeup all waiting threads

- Condition variables *are not* boolean objects
  - “if (condition_variable) then” … does not make sense
  - “if (num_resources == 0) then wait(resources_available)” does
  - An example will make this more clear
Condition Variables

- Condition variables are NOT conditions
  - So don’t EVER do:
    - if (conditionaVariable){
      
      ...

    }

- Instead, it is a way for one thread to wait (if some resource is not available), and some other thread to wake it up once the resource becomes available
  - Sleep
  - Wake (also called as “signal”)
  - Wakeall (or “signalAll”)

10/10/18
Condition Variable & Lock

- Condition variable doesn’t replace lock, instead it compliments lock

**Sleep**(condition, lock) or **Wait**(condition, lock)
  - First release the lock, put the thread into the queue of the condition, if waking up, re-acquiring the lock
  - Once sleep returns, it is awaken by some other thread, and it also holds the lock

**Wake**(condition) or **Signal**(condition): Wake up a thread waiting on the condition (queue)
  - Some systems use a different name such as “**Signal(condition)**” or “**Notify(condition)**”

**Wakeall**(condition) or **Broadcast**(condition)
  - Wake all the thread waiting on the condition (queue)
  - Some systems may use a different name such as “**SignalAll(condition)**, or **NotifyAll(condition)**"
Monitor bounded_buffer {
    Resource buffer[N];
    // Variables for indexing buffer
    // monitor invariant involves these vars
    Condition not_full; // space in buffer
    Condition not_empty; // value in buffer

    void put_resource (Resource R) {
        if (buffer array is full)
            wait(not_full);
        Add R to buffer array;
        signal(not_empty);
    }

    Resource get_resource() {
        if (buffer array is empty)
            wait(not_empty);
        Get resource R from buffer array;
        signal(not_full);
        return R;
    }
} // end monitor

- What happens if no threads are waiting when signal is called?
  - Signal is lost
Monitor bounded_buffer {
    Condition not_full;
    …other variables…
    Condition not_empty;
    void put_resource () {
        …wait(not_full)…
        …signal(not_empty)…
    }
    Resource get_resource () {
        …
    }
}
Condition Vars != Semaphores

- Monitor with Condition variables != semaphores
  - But they can implement each other
- Access to the monitor is controlled by a lock
  - `wait()` blocks the calling thread, and gives up the lock
    - To call `wait`, the thread has to be in the monitor (hence has lock)
    - `Semaphore::P` just blocks the thread on the queue
  - `signal()` causes a waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - `Semaphore::V()` increases the semaphore count, allowing future entry even if no thread is waiting
    - Condition variables have no history
Signal Semantics

- There are two flavors of monitors that differ in the scheduling semantics of signal():
  - **Hoare** monitors (original)
    - signal() immediately switches from the caller to a waiting thread
    - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
    - Signaler must restore monitor invariants before signaling
  - **Mesa** monitors (Mesa, Java)
    - signal() places a waiter on the ready queue, but signaler continues inside monitor
    - Condition is not necessarily true when waiter runs again
      - Returning from wait() is only a hint that something changed
      - Must recheck conditional case
Hoare vs. Mesa Monitors

- **Hoare**
  
  ```
  if (empty)
    wait(condition);
  ```

- **Mesa**
  
  ```
  while (empty)
    wait(condition);
  ```

- **Tradeoffs**
  - Mesa monitors easier to use, more efficient
    - Fewer context switches, easy to support broadcast
  - Hoare monitors leave less to chance
    - Easier to reason about the program
Monitor Readers and Writers

- Write with just wait() will be safe, maybe not “live” - why?
  - Starvation

Monitor RW {
    int nr = 0, nw = 0;
    Condition canRead, canWrite;

    void StartRead () {
        while (nw != 0) do wait(canRead);
        nr++;
    }

    void EndRead () {
        nr--;
        if (nr==0) signal(canWrite)
    }

    void StartWrite {
        while (nr != 0 || nw != 0) do wait(canWrite);
        nw++;
    }

    void EndWrite () {
        nw--;
        signal(canWrite);
        signal(canRead);
    }
} // end monitor
Monitor Readers and Writers

- Is there any priority between readers and writers?
- What if you wanted to ensure that a waiting writer would have priority over new readers?
Summary

- **Semaphores**
  - P()/V() implement blocking mutual exclusion
  - Also used as atomic counters (counting semaphores)
  - Can be inconvenient to use

- **Monitors**
  - Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
    - Only one thread can execute within a monitor at a time
    - Relies upon high-level language support

- **Condition variables**
  - Used by threads as a synchronization point to wait for events
  - Inside monitors, or outside with locks
Dining Philosophers: an intellectual game

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- Possible deadlock?
- How to prevent deadlock?
Does it solve the Dining Philosophers Problem?

```c
#define N 5 /* number of philosophers */

void philosopher(int i) /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think(); /* philosopher is thinking */
        take_fork(i); /* take left fork */
        take_fork((i+1) % N); /* take right fork; % is modulo operator */
        eat(); /* yum-yum, spaghetti */
        put_fork(i); /* put left fork back on the table */
        put_fork((i+1) % N); /* put right fork back on the table */
    }
}
```
Dining Philosophers Solution

#define N 5  /* number of philosophers */
#define LEFT  (i+N-1)%N  /* number of i’s left neighbor */
#define RIGHT (i+1)%N  /* number of i’s right neighbor */
#define THINKING 0  /* philosopher is thinking */
#define HUNGRY 1  /* philosopher is trying to get forks */
#define EATING 2  /* philosopher is eating */

typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i) {  /* i: philosopher number, from 0 to N-1 */
    while (TRUE) {  /* repeat forever */
        think();  /* philosopher is thinking */
        take_forks(i);  /* acquire two forks or block */
        eat();  /* yum-yum, spaghetti */
        put_forks(i);  /* put both forks back on table */
    }
}
Dining Philosophers Solution

void take_forks(int i) /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex); /* enter critical region */
    state[i] = HUNGRY; /* record fact that philosopher i is hungry */
    test(i); /* try to acquire 2 forks */
    up(&mutex); /* exit critical region */
    down(&s[i]); /* block if forks were not acquired */
}

void put_forks(i) /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex); /* enter critical region */
    state[i] = THINKING; /* philosopher has finished eating */
    test(LEFT); /* see if left neighbor can now eat */
    test(RIGHT); /* see if right neighbor can now eat */
    up(&mutex); /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
The Sleeping Barber Problem

- N customer Chair
- One barber can cut one customer’s hair at any time
- No customer, goes to sleep
The Sleeping Barber Solution (1)

```c
#define CHAIRS 5

typedef int semaphore;

semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;
/* # chairs for waiting customers */
/* use your imagination */
/* # of customers waiting for service */
/* # of barbers waiting for customers */
/* for mutual exclusion */
/* customers are waiting (not being cut) */
```
The Sleeping Barber Solution (2)

```c
void barber(void)
{
    while (TRUE) {
        down(&customers); /* go to sleep if # of customers is 0 */
        down(&mutex); /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers); /* one barber is now ready to cut hair */
        up(&mutex); /* release 'waiting' */
        cut_hair(); /* cut hair (outside critical region) */
    }
}
```
The Sleeping Barber Solution (3)

```c
void customer(void) {
    down(&mutex); /* enter critical region */
    if (waiting < CHAIRS) { /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers); /* wake up barber if necessary */
        up(&mutex); /* release access to 'waiting' */
        down(&barbers); /* go to sleep if # of free barbers is 0 */
        get_haircut(); /* be seated and be serviced */
    } else {
        up(&mutex); /* shop is full; do not wait */
    }
}
```

Solution to sleeping barber problem.
Santa Clause Problem

- http://www.youtube.com/watch?v=pqO6tKN2lc4