Higher Level Synchronization

- 7H Semaphores
- 7I Producer/Consumer Problems
- 7L Higher Level Synchronization

Semaphores – signaling devices

when direct communication was not an option

e.g. between villages, ships, trains

Semaphores - History

- Concept introduced in 1968 by Edsger Dijkstra
  - cooperating sequential processes
- THE classic synchronization mechanism
  - behavior is well specified and universally accepted
  - a foundation for most synchronization studies
  - a standard reference for all other mechanisms
- more powerful than simple locks
  - they incorporate a FIFO waiting queue
  - they have a counter rather than a binary flag

Semaphores - Operations

- Semaphore has two parts:
  - an integer counter (initial value unspecified)
  - a FIFO waiting queue
- P (proberen/test) ... “wait”
  - decrement counter, if count >= 0, return
  - if counter < 0, add process to waiting queue
- V (verhogen/raise) ... “post” or “signal”
  - increment counter
  - if counter >= 0 & queue non-empty, wake 1st proc

using semaphores for exclusion

- initialize semaphore count to one
  - count reflects # threads allowed to hold lock
- use P/wait operation to take the lock
  - the first will succeed
  - subsequent attempts will block
- use V/post operation to release the lock
  - restore semaphore count to non-negative
  - if any threads are waiting, unblock the first in line

Semaphores - for exclusion

```
struct account {
        struct semaphore s;             /* initialize count to 1, queue empty, lock 0  */
        int balance;
        …
};

int write_check( struct account *a, int amount ) {
        int ret;
        p( &s->semaphore );   /* get exclusive access to the account  */

                if ( a->balance >= amount ) {       /* check for adequate funds       */
                        amount -= balance;
                        ret = amount;
                } else
                        ret = -1;

        v( &s->semaphore );   /* release access to the account          */
        return( ret );
}
```

using semaphores for notifications

- initialize semaphore count to zero
  - count reflects # of completed events
- use P/wait operation to await completion
  - if already posted, it will return immediately
  - else all callers will block until V/post is called
- use V/post operation to signal completion
  - increment the count
  - if any threads are waiting, unblock the first in line
- one signal per wait: no broadcasts

Semaphores - completion events

```
struct semaphore data_semaphore = { 0, 0, 0 }; /* count = 0; pipe empty */
struct semaphore space_semaphore = { BUFSIZE, 0, 0 }; /* count=BUFSIZE */
char buffer[BUFSIZE]; int read_ptr = 0, write_ptr = 0;

char pipe_read_char() {
        p (&data_semaphore );           /* wait for input available     */
        c = buffer[read_ptr++];         /* get next input character     */
        v(&space_semaphore);            /* one more empty space        */
        if (read_ptr >= BUFSIZE)        /* circular buffer wrap    */
                read_ptr -= BUFSIZE;
                return(c);
}

void pipe_write_string( char *buf, int count ) {
        while( count-- > 0 ) {
                p(&space_semaphore);    /* wait for free space      */
                buffer[write_ptr++] = *buf++;/* store next character*/
                if (write_ptr >= BUFSIZE)       /* circular buffer wrap    */
                        write_ptr -= BUFSIZE;
                v( &data_semaphore );   /* signal char available    */
        }
}
```


Counting Semaphores

- initialize semaphore count to ...
  - count reflects # of available resources
- use P/wait operation to consume a resource
  - if available, it will return immediately
  - else all callers will block until V/post is called
- use V/post operation to produce a resource
  - increment the count
  - if any threads are waiting, unblock the first in line
- one signal per wait: no broadcasts

Implementing Semaphores

```
void sem_wait(sem_t *s) {
        pthread_mutex_lock(&s->lock);
        while (s->value <= 0)
                pthread_cond_wait(&s->cond, &s->lock);
        s->value--;
        pthread_mutex_unlock(&s->lock);
}
```

```
void sem_post(sem_t *s) {
        pthread_mutex_lock(&s->lock);
        s->value++;
        pthread_cond_signal(&s->cond);
        pthread_mutex_unlock(&s->lock)
}
```


Implementing Semaphores in OS

```
void sem_wait(sem_t *s ) {
        for (;;) {
                save = intr_enable( ALL_DISABLE );
                while( TestAndSet( &s->lock ) );
                if (s->value > 0) {
                        s->value--;
                        s->sem_lock = 0;
                        intr_enable( save );
                        return;
                }
                add_to_queue( &s->queue, myproc );
                myproc->runstate |= PROC_BLOCKED;
                s->lock = 0;
                intr_enable( save );
                yield();
        }
}
```

```
void sem_post(struct sem_t *s) {
        struct proc_desc *p = 0;
        save = intr_enable( ALL_DISABLE );
        while ( TestAndSet( &s->lock ) );
        s->value++;
        if (p = get_from_queue( &s->queue )) {
                p->runstate &= ~PROC_BLOCKED
        }
        s->lock = 0;
        intr_enable( save );
        if (p)
                reschedule( p );
}
```


(locking to solve sleep/wakeup race)

- requires a spin-lock to work on SMPs
  - sleep/wakeup may be called on two processors
  - the critical section is short and cannot block
  - we must spin, because we cannot sleep ... the lock we need is the one that protects the sleep operation
- also requires interrupt disabling in sleep
  - wakeup is often called from interrupt handlers
  - interrupt possible during sleep/wakeup critical section
  - If spin-lock already is held, wakeup will block for ever
- very few operations require both of these

Limitations of Semaphores

- semaphores are a very spartan mechanism
  - they are simple, and have few features
  - more designed for proofs than synchronization
- they lack many practical synchronization features
  - It is easy to deadlock with semaphores
  - one cannot check the lock without blocking
  - they do not support reader/writer shared access
  - no way to recover from a wedged V'er
  - no way to deal with priority inheritance
- none the less, most OSs support them

Using Condition Variables

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
…
pthread_mutex_lock(&lock);
while (ready == 0)
    pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock)
…
                if (pthread_mutex_lock(&lock)) {
                    ready = 1;
                    pthread_cond_signal(&cond);
                    pthread_mutex_unlock(&lock);
                }
```


Bounded Buffer Problem w/CVs

```
void producer( FIFO *fifo, char *msg, int len ) {
    for( int i = 0; i < len; i++ ) {
        pthread_mutex_lock(&mutex);
        while (fifo->count == MAX)
            pthread_cond_wait(&nonfull, &mutex);
        put(fifo, msg[i]);
        pthread_cond_signal(&nonempty);
        pthread_mutex_unlock(&mutex);
    }
}
```

```
void consumer( FIFO *fifo, char *msg, int len ) {
    for( int i = 0; i < len; i++ ) {
        pthread_mutex_lock(&mutex);
        while (fifo->count == 0)
            pthread_cond_wait(&nonempty, &mutex
        msg[i] = get(fifo);
        pthread_cond_signal(&nonfull);
        pthread_mutex_unlock(&mutex);
    }
}
```


Producer/Consumer w/Semaphores

```
void producer( FIFO *fifo, char *msg, int len ) {
    for( int i = 0; i < len; i++ ) {
        sem_wait(&empty_space);
        sem_wait(&mutex);
        put(fifo, msg[i]);
        sem_post(&mutex);
        sem_post(&data_avail);
    }
}
```

```
void consumer( FIFO *fifo, char *msg, int len ) {
    for( int i = 0; i < len; i++ ) {
        sem_wait(&data_avail);
        sem_wait(&mutex);
        msg[i] = get(fifo);
        sem_post(&mutex);
        sem_post(&empty_space);
    }
}
```


Synchronization is Difficult

- recognizing potential critical sections
  - potential combinations of events
  - interactions with other pieces of code
- choosing the mutual exclusion method
  - there are many different mechanisms
  - with different costs, benefits, weaknesses
- correctly implementing the strategy
  - correct code, in all of the required places
  - maintainers may not understand the rules

We need a “Magic Bullet”

- We identify shared resources
  - objects whose methods may require serialization
- We write code to operate on those objects
  - just write the code
  - assume all critical sections will be serialized
- Complier generates the serialization
  - automatically generated locks and releases
  - using appropriate mechanisms
  - correct code in all required places

Monitors – Protected Classes

- each monitor class has a semaphore
  - automatically acquired on method invocation
  - automatically released on method return
  - automatically released/acquired around CV waits
- good encapsulation
  - developers need not identify critical sections
  - clients need not be concerned with locking
  - protection is completely automatic
- high confidence of adequate protection

Monitors: use

```
monitor CheckBook {
    // class is locked when any method is invoked
    private int balance;
    public int balance() {
        return(balance);
    }
    public int debit(int amount) {
        balance -= amount;
        return( balance)
    }
}
```


Evaluating: Monitors

- correctness
  - complete mutual exclusion is assured
- fairness
  - semaphore queue prevents starvation
- progress
  - inter-class dependencies can cause deadlocks
- performance
  - coarse grained locking is not scalable

Java Synchronized Methods

- each object has an associated mutex
  - acquired before calling a *synchronized* method
  - nested calls (by same thread) do not reacquire
  - automatically released upon final return
- static *synchronized* methods lock class mutex
- advantages
  - finer lock granularity, reduced deadlock risk
- costs
  - developer must identify serialized methods

Java Synchronized: use

```
class CheckBook {
    private int balance;
    public int balance() {
        return(balance);
    }
    // object is locked when this method is invoked
    public synchronized int debit(int amount) {
        balance -= amount;
        return( balance)
    }
}
```


Evaluating Java Synchronized Methods

- correctness
  - correct if developer chooses the right methods
- fairness
  - priority thread scheduling (potential starvation)
- progress
  - safe from single thread deadlocks
- performance
  - fine grained (per object) locking
  - some methods may not need the lock at all

Encapsulated Locking

- opaquely encapsulate implementation details
  - explicitly implemented for each class
  - make class easier to use for clients
  - preserve the freedom to change it later
- locking is entirely internal to class
  - search/update races within the methods
  - critical sections involve only class resources
  - critical sections do not span multiple operations
  - no possible interactions with external resources

Client Locking

- Class cannot correctly synchronize all uses
  - critical section spans multiple class operations
  - updates in a higher level transaction
- client-dependent synchronization needs
  - locking needs depend on how object is used
  - client may control access to protected objects
  - client may select best serialization method
- potential interactions with other resources
  - deadlock prevention must be at higher level

Evaluating: Client-Side Locking

- correctness
  - if all clients always do it correctly
- fairness
  - depends on underlying implementation
- progress
  - depends on underlying implementation
- performance
  - potentially very good due to minimal use

Reading and Assignments

Reading:

- Arpaci C32-32.2 … concurrency problems
- Arpaci C32.3 … deadlock
- Kampe: Deadlock Avoidance
- Kampe: Health Monitoring
- Kampe: Priority Inversion

Projects:

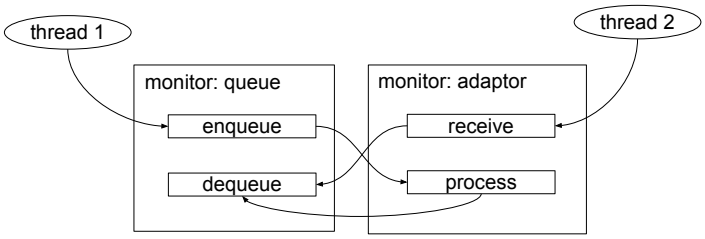
- we will help w/project 4B problems in the lab

Supplementary Slides

Active/Passive - the preemption thing

- standard semaphore semantics are not complete
  - who runs after a V unblocks a P?
  - the running V'er or the blocked P'er
- there are arguments for each behavior
  - gratuitous context switches increase overhead
  - producers and consumers should take turns
  - if we delay P'er, someone else may get semaphore
- preemptive priority-based scheduler can do this
  - reassess scheduling whenever someone wakes up
  - P'ers priority controls who will run after wake-up

nested monitors – example



(nested monitors – simpler isn't safer)

- consider two monitors:
  - QUEUE with methods: enqueue, dequeue
  - ADAPTOR with methods: process, receive
    - where ADAPTORs are implemented with QUEUEs
- possible static deadlocks:
  - QUEUE.enqueue adds entry, calls ADAPTOR.process
  - ADAPTOR.process calls QUEUE.dequeue
- possible dynamic deadlocks:
  - thread 1 calls QUEUE.enque, calls ADAPTOR.process
  - thread 2 calls ADAPTOR.receive, calls QUEUE.enqueue

Monitors: simplicity vs. performance

- monitor locking is very conservative
  - lock the entire class (not merely a specific object)
  - lock for entire duration of any method invocations
- this can create performance problems
  - they eliminate conflicts by eliminating parallelism
  - if a thread blocks in a monitor a convoy can form
- There Ain't No Such Thing As A Free Lunch
  - fine-grained locking is difficult and error prone
  - coarse-grained locking creates bottle-necks

Monitors: implementation

```
monitor generic {
   semaphore mutex = 1;
   … other private data …
// public external entrypoints … all protected by mutex
   public:      method_1(parms) {
                        p(&mutex);
                        _method_1(parms);
                        v(&mutex); }
// real implementations
      _method_1(parms) { … }
}
```


Monitors – Hoare vs Mesa Semantics

- when a process is awakened from a wait …
- Hoare Semantics
  - guarantee that it was the first process to be run
  - guarantee that the mutex is now available
  - simpler semantics, easier for the client
- Mesa Semantics
  - no guarantee about when that process will run
  - another process may have the mutex by then
  - use a loop to wait for the mutex
  - recognizes complexity of parallel computing