## Synchronization Mechanisms &Problems





## 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( &a->semaphore );          /* get exclusive access to the account        */

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

    v( &a->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 pipe_semaphore = { 0, 0, 0 }; /* count = 0; pipe empty */
char buffer[BUFSIZE]; int read_ptr = 0, write_ptr = 0;

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

void pipe_write_string( char *buf, int count ) {
        while( count-- > 0 ) {
                buffer[write_ptr++] = *buf++;    /* store next character        */
                if (write_ptr >= BUFSIZE)    /* circular buffer wrap        */
                        write_ptr -= BUFSIZE;
                v( &pipe_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 = PTHEAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHEAD_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(&empty, &mutex);
            put(fifo, msg[i]);
            pthread_cond_signal(&fill);
            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(&fill, &mutex);
                                    msg[i] = get(fifo);
                                    pthread_cond_signal(&empty);
                                    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);
            sem_wait(&mutex);
            put(fifo, msg[i]);
            sem_post(&mutex);
            sem_post(&full);
    }
}
                        void consumer( FIFO *fifo, char *msg, int len ) {
                            for( int i = 0; i < len; i++ ) {
                                    sem_wait(&full);
                                    sem_wait(&mutex);
                                    msg[i] = get(fifo);
                                    sem_post(&mutex);
                                    sem_post(&empty);
                            }
                        }
```



## Object Level Locking

- mutexes protect <u>code</u> critical sections
  - brief durations (e.g. nanoseconds, milliseconds)
  - other threads operating on the same data
  - all operating in a single address space
- persistent objects are more difficult
  - critical sections are likely to last much longer
  - many different programs can operate on them
  - may not even be running on a single computer
- solution: lock objects (rather than code)



## Whole File Locking

**int flock(*fd*, *operation*)**

- supported *operation*s:
  - LOCK_SH … shared lock (multiple allowed)
  - LOCK_EX … exclusive lock (one at a time)
  - LOCK_UN … release a lock
- lock is associated with an open file descriptor
  - lock is released when that file descriptor is closed
- locking is purely advisory
  - does not prevent reads, writes, unlinks

## Advisory vs Enforced Locking

- Enforced locking
  - done within the implementation of object methods
  - guaranteed to happen, whether or not user wants it
  - may sometimes be too conservative
- Advisory locking
  - a convention that “good guys” are expected to follow
  - users expected to lock object before calling methods
  - gives users flexibility in what to lock, when
  - gives users more freedom to do it wrong (or not at all)
  - mutexes are advisory locks



## Ranged File Locking

**int lockf(*fd, cmd, offset, len*)**

- supported *cmds*:
  - F_LOCK … get/wait for an exclusive lock
  - F_ULOCK … release a lock
  - F_TEST/F_TLOCK … test, or non-blocking request
  - *offset/len* specifies portion of file to be locked
- lock is associated with a file descriptor
  - lock is released when file descriptor is closed
- locking may or may not be enforced
  - depending on the underlying file system



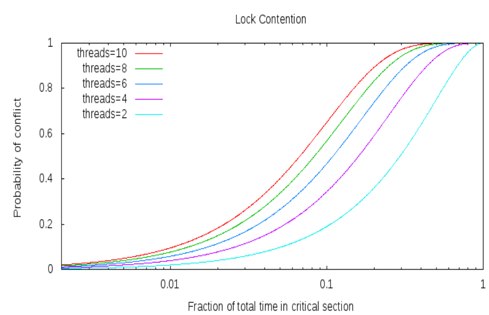
## Cost of not getting a Lock

- protect critical sections to ensure correctness
- many critical sections are very brief
  - in and out in a matter of nano-seconds
- blocking is much more (e.g. 1000x) expensive
  - micro-seconds to yield, context switch
  - milliseconds if swapped-out or a queue forms
- performance depends on conflict probability

  $C_{expected} = (C_{get} * (1-P_{conflict})) + (C_{block} * P_{conflict})$



## Probability of Conflict






## Convoy Formation

- in general

  $P_{conflict} = 1 - (1 - (T_{critical} / T_{total}))^{threads-1}$

  (nobody else in critical section at the same time)
- unless (or until) a FIFO queue forms

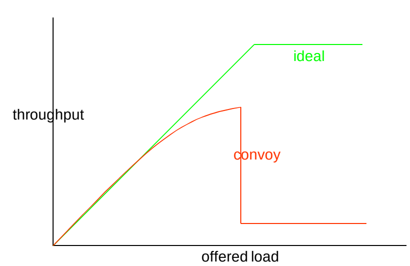
  $P_{conflict} = 1 - (1 - ((T_{wait} + T_{critical}) / T_{total}))^{threads}$

  if $T_{wait} >> T_{critical}$ , $P_{conflict}$ rises significantly
- if $T_{wait}$ exceeds the mean inter-arrival time

  the line becomes permanent, parallelism ceases, (cheap) $T_{critical}$ is replaced by (expensive) $T_{wait}$



## Performance: resource convoys



## Contention Reduction

- eliminate the critical section entirely
  – eliminate shared resource, use atomic instructions
- eliminate preemption during critical section
  – by disabling interrupts … not always an option
  – avoid resource allocation within critical section
- reduce time spent in critical section
  – reduce amount of code in critical section
- reduce frequency of critical section entry
  – reduce use of the serialized resource
  – reduce exclusive use of the serialized resource
  – spread requests out over more resources



## Reducing Time in Critical Section

- eliminate potentially blocking operations
  – allocate required memory before taking lock
  – do I/O before taking or after releasing lock
- minimize code inside the critical section
  – only code that is subject to destructive races
  – move all other code out of the critical section
  – especially calls to other routines
- cost: this may complicate the code
  – unnaturally separating parts of a single operation



## Reduce Time or Preemption

```
int List_Insert(list_t *l, int key) {
    pthread_mutex_lock(&l->lock);
    node_t new = (node_t*) malloc(sizeof(node_t));
    if (new == NULL) {
            perror("malloc");
            pthread_mutex_unlock(&l->lock);
            return(-1);
    }
    new->key = key;
    new->next = l->head;
    l->head = new;
    pthread_mutex_unlock(&l->lock);
    return 0;
}
```

```
int List_Insert(list_t *l, int key) {
    node_t new = (node_t*) malloc(sizeof(node_t));
    if (new == NULL) {
            perror("malloc");
            return(-1);
    }
    new->key = key;
    pthread_mutex_lock(&l->lock);
    new->next = l->head;
    l->head = new;
    pthread_mutex_unlock(&l->lock);
    return 0;
}
```



## Reduced Use of Critical Section

- can we use critical section less often
  – less use of high-contention resource/operations
  – batch operations
- consider "sloppy counters"
  – move most updates to a private resource
  – costs:
    - global counter is not always up-to-date
    - thread failure could lose many updates
  – alternative:
    - sum single-writer private counters when needed



## Non-Exclusivity: read/write locks

- reads and writes are not equally common
  – file read/write: reads/writes > 50
  – directory search/create: reads/writes > 1000
- only writers require exclusive access
- read/write locks
  – allow many readers to share a resource
  – only enforce exclusivity when a writer is active
  – policy: when are writers allowed in?
    - potential starvation if writers must wait for readers



## Spreading requests: lock granularity

- coarse grained - one lock for many objects
  – simpler, and more idiot-proof
  – greater resource contention (threads/resource)
- fine grained - one lock per object (or sub-pool)
  – spreading activity over many locks reduces contention
  – dividing resources into pools shortens searches
  – a few operations may lock multiple objects/pools
- TANSTAAFL
  – time/space overhead, more locks, more gets/releases
  – error-prone: harder to decide what to lock when

## Partitioned Hash Table

```
int Hash_Insert(hash_t *h, int key) {
    int bucket = key % h->num_buckets;
    list_t *l = &h->lists[bucket];
    return List_Insert(l, key);
}
```

- Each list_t is still protected by a lock
  - but contention has been greatly reduced
- Partitioning function must be race-free
  - no critical-section to protect
  - per partition load depends on request randomness



## Mid-Term Exam

- When
  - Thursday, the full 110 minute period
- Value
  - 15% of course grade
- Form and content
  - 10 multi-part, brief-answer questions
    - covering all lectures and reading to date
    - based on key learning objectives
  - one hard extra credit question
    - similar to those on part II of the final



# Supplementary Slides

## Example: P and V

```
void v(struct semaphore *s) {
    struct proc_desc *p = 0;

    save = intr_enable( ALL_DISABLE );
    while ( TestAndSet( &s->sem_lock ) );
    s->sem_count++;
    if (p = get_from_queue( &s->sem_queue)) {
      p->runstate &= ~PROC_BLOCKED;
    }
    s->sem_lock = 0;
    intr_enable( save );
    if (p)
      reschedule(p);
}
```

```
void p(struct semaphore *s) {
    struct proc_desc *p = 0;
    for (;;) {
       save = intr_enable( ALL_DISABLE );
       while ( TestAndSet( &s->sem_lock ) );
       if (s->sem_count > 0) {
            s->sem_cont--;
            s->sem_lock = 0;
            intr_enable( save );
            return;
       }
       add_to_queue( &s->sem_queue, myproc );
       myproc->runstate |= PROC_BLOCKED;
       s->sem_lock = 0;
       intr_enable( save );
       yield();
    }
}
```

| | | | | | | |
|---|---|---|---|---|---|---|
| **process A** | | | P | P | | WAKE |
| **process B** | | V | | | V | |
| **Semaphore** | | | | | | |
| lock | NO | YES | YES | YES | YES | YES |
| count | 0 | 1 | 0 | | 1 | 0 |
| queue | 0 | | | A | | |
| **int disable** | NO | YES | YES | YES | YES | YES |



## Example: Producer/Consumer

```
char pipe_read_char() {

        p (&pipe_semaphore );
        c = buffer[read_ptr++];

        if (read_ptr >= BUFSIZE)
                read_ptr -= BUFSIZE;
        return(c);
}
```

```
void pipe_write_string( char *buf, int count ) {

    while( count-- > 0 ) {
            buffer[write_ptr++] = *buf++
            if (write_ptr >= BUFSIZE)
                    write_ptr -= BUFSIZE;

            v( &pipe_semaphore );
    }
}
```

| | | | | | |
|---|---|---|---|---|---|
| **process A** | | | WRITE "abc" | | |
| **process B** | | READ | | WAKE | READ |
| **buffer** | | | a b c | | |
| read_ptr | 0 | | | 1 | 2 |
| write_ptr | 0 | | 3 | | |
| **sem count** | 0 | | 3 | 2 | 1 |



## 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

## Where to put the locking

- there is a choice about where to do locking
  - A ,B require serialization, and are called by C,D
  - should we lock in objects (A,B) or in callers (C,D)
- OO modularity says: as low as possible (in A,B)
  - correct locking is part of correct implementation
- but as high as necessary (in C,D)
  - locking needs may depend on how object is used
  - one logical transaction may span many method calls
  - in such cases, only the caller knows start/end/scope



## Performance of Locking

- Locking typically performed as an OS system call
  - Particularly for enforced locking
- Typical system call overheads for lock operations
- If they are called frequently, high overheads
- Even if not in OS, extra instructions run to lock and unlock



## Eliminating Critical Sections

- Eliminate shared resource
  - Give everyone their own copy
  - Find a way to do your work without it
- Use atomic instructions
  - Only possible for simple operations
- Great when you can do it
- But often you can't



## Locking Costs

- Locking called when you need to protect critical sections to ensure correctness
- Many critical sections are very brief
  - In and out in a matter of nano-seconds
- Overhead of the locking operation may be much higher than time spent in critical section



## Performance: lock contention

- The riddle of parallelism:
  - parallelism: if one task is blocked, CPU runs another
  - concurrent use of shared resources is difficult
  - critical sections serialize tasks, eliminating parallelism
- What if everyone needs to use one resource?
  - one process gets the resource
  - other processes get in line behind him (convoy)
  - parallelism is eliminated; B runs after A finishes
  - that resource becomes a *bottle-neck*