Operating Systems

Synchronization

1

Background

- Threads inherently provide easy access to shared data
- Concurrent access to shared data may result in data inconsistency
  - Implicit and explicit conflicts in shared access
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Providing these mechanisms is a critical function in a concurrent environment

2

Producer / Consumer

- Cooperating process model
  - Separation of concerns using multiple processor threads working in concert
- One thread is running a function to create values and put them in a variable, and another thread is consuming them
- Standard design pattern / problem
- Can consider a bounded buffer or an unbounded queue

3

Producer / Consumer

```
/* ex4.c - main, produce, consume */
#include <xinu.h>

void produce(void), consume(void);

int32 n = 0; /* external variables are shared by all processes */

/*------------------------------------------------------------------------
 * main -- example of unsynchronized producer and consumer processes
 *------------------------------------------------------------------------
 */

void main(void) {

        resume( create(consume, 1024, 20, "cons", 0) );
        resume( create(produce, 1024, 20, "prod", 0) );
}
```

4

Producer / Consumer

```
/*-----------------------------------------------------------------------
 * produce -- increment n 2000 times and exit
 *-----------------------------------------------------------------------
 */

void produce(void) {
        int32 i;

        for( i=1; i<=2000; i++ )
                n++;
}

/*-----------------------------------------------------------------------
 * consume – print n 2000 times and exit
 *-----------------------------------------------------------------------
 */

void consume(void) {
        int32 i;

        for( i=1 ; i<=2000 ; i++ )
                printf("The value of n is %d \n", n);
}
```

5

Synchronization

- Sharing memory leads to race conditions
  - Two processes doing load – manipulate - store can result in corruption
- This can occur in the case of process preemption, but even a voluntary yield could leave values in registers, which are stored when a process stops running
  - Even at non-obvious times as an optimizing compiler tries to keep values in registers for reuse

6

Printing Your Own Money

```
int balance;

int
withdraw(int amount) {
  balance = balance – amount;
  return amount;
}

Pseudo-assembly:
      ld      balance, %r2
      sub     %r2, %r1, %r2
      st      %r2, balance
```

7

Concurrent Execution with Preemption

Starting Balance = 300

Bonnie: %r1 = 100 | **Clyde: %r1 = 100**

```
Bonnie:                          Clyde:
ld    balance, %r2
                                 ld    balance, %r2
                                 sub   %r2, %r1, %r2
sub   %r2, %r1, %r2
st    %r2, balance
                                 st    %r2, balance
```

8

Producer / Consumer

- One thread is running a function to create new items and put them in a buffer and another thread is consuming them
- We can do so by having an integer *count* that keeps track of the number of full buffers
  - Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

9

Producer

```
while (true) {

     /* produce an item, put in nextProduced*/
      while (count == BUFFER_SIZE)
            ; // do nothing
      buffer [in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      count++;
}
```

10

Consumer

```
while (true)  {
    while (count == 0); // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

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

11

Avoiding Busy Waiting

- Busy waiting with “spin locks” isn’t always ideal
  - It uses CPU time
- We often need to put executing processes or threads to sleep
  - Try taking a nap while you wait…
  - Assuming that something is there to wake you up
- Depends on a scheduling entity

12

Counter-example: Kernel Debugging

- The previous examples use putc (and printf) to display to the CONSOLE
- These are fine once things are working, but they depend on various OS components
  - like interrupts
- But if you're debugging those very components, what can you do?
- The answer is polled I/O
  - Does not require interrupts to be working

13

Kputc and Kprintf

- The functions kputc() and kprintf() are used in kernel debugging
- kputc() takes a character and
  - Disables interrupts
  - waits for the CONSOLE device to be idle
  - sends a character to CONSOLE
  - Restores interrupts to their previous state
- This means that all processing stops until the character has been displayed
- kprintf() builds on this to provide formatted output

14

Sleep and Wakeup

```
#define N 100                                  /* number of slots in the buffer */
int count = 0;                                 /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          item = produce_item( );              /* generate next item */
          if (count == N) sleep( );            /* if buffer is full, go to sleep */
          insert_item(item);                   /* put item in buffer */
          count = count + 1;                   /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);    /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          if (count == 0) sleep( );            /* if buffer is empty, got to sleep */
          item = remove_item( );               /* take item out of buffer */
          count = count − 1;                   /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                  /* print item */
     }
}
```

Producer-consumer problem

15

Semaphores

- Consider "lost wakeup calls"
- Semaphore - *bearing a sign*
  - An apparatus for giving signals by the disposition of flags, lanterns, etc.
  - Term coined by Dijkstra
- 0 indicating that no wakeups were saved
  - Value is > 0 if wakeups are pending
- Two ops, *wait and signal*
  - Or **P** and **V**
- Calling *down* (or **P**) on a 0-valued semaphore puts the process to sleep

16

Semaphores

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - P() and V()
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    }
  - signal (S) {
    S++;
    }

17

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks or a mutex
- Can implement a counting semaphore S with a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);
    Critical Section
    signal (S);
- Xinu implements counting semaphores
  - wait() and signal()

18

P/C with Semaphores (ex6.c)

```
/* code */
#include <xinu.h>
sid32 mutex; /* assume initialized with semcreate
              * e.g. mutex = semcreate(1);
              */
int32 shared[100];  /* array shared by processes */
int32 n = 0;          /* count of items in the array */

/*------------------------------------------------------------------
 * additem -- obtain excl. access to array ary and add an item to it
 *------------------------------------------------------------------
 */
void additem( int32 item ) /* item to add to array ary */
{
        wait(mutex);
        shared[n++] = item;
        signal(mutex);
}
```

19

Semaphore Implementation

- Must guarantee that no two processes can execute in `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

20

Synchronization Hardware

- Most systems provide hardware support for critical section code
- Uniprocessors disable interrupts
  - Currently running code will execute without preemption
- Not desirable on multiprocessor systems
  - Not a scalable approach
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable

21

Test-and-Set

- Write to a location and return the previous value atomically

```
int TestAndSet (int *target, int newval) {
    int oldval;

    oldval = *target;
    *target = newval;
    return (oldval);

}
```

22

TSL – Test and Set Lock

- Generally implemented as an instruction called TSL

```
enter_region:

  tsl reg, flag        ; flag is copied into reg and set to 1

  cmp reg, #0          ; Was flag zero at enter_region?

  jnz enter_region     ; Jump to enter_region if flag != 0

  ret                  ; flag was 0, now 1.  lock successful.

leave_region:
  move flag, #0        ; store 0 in flag
  ret                  ; return to caller
```

23

Swap Operation

- Atomically swap two values

```
bool Swap (int *a, int *b) {
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;

    return (true);

}
```

24

Compare-And-Swap (CAS) Instruction

- CAS only performs the swap if the target (*p) is equal to the expected old value

```
bool CompareAndSwap (int *p, int old, int new) {

    if (*p != old) { return (false); )
    *p = new;
    return (true);

}
```

25

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
while (true) {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );

            //    critical section

    lock = FALSE;

            //    remainder section

}
```

26

Lock and Unlock with TSL

```
mutex_lock:
    TSL REGISTER,MUTEX        | copy mutex to register and set mutex to 1
    CMP REGISTER,#0           | was mutex zero?
    JZE ok                    | if it was zero, mutex was unlocked, so return
    CALL thread_yield         | mutex is busy; schedule another thread
    JMP mutex_lock            | try again later
ok: RET | return to caller; critical region entered


mutex_unlock:
    MOVE MUTEX,#0             | store a 0 in mutex
    RET | return to caller
```

27

Load-Link / Store-Conditional

- Load-Link returns a value from a memory region
- Store Conditional will only store to the memory region if no updates have occurred to that memory region in the meantime
- Together these operations implement an atomic read-modify-write operation
- This is a stronger guarantee than the CAS operation, which will not detect if the value has been changed and restored to its original value in the meantime
- Depending on the implementation, LL/SC may fail even if the region has not been modified

28

Issues with Busy Waiting

- These mutual exclusion solutions require busy waiting
- "Spinning" on a lock waiting or polling a variable for it to become available wastes CPU time
- The book asserts that no process should use the CPU while waiting for another process
- In multi-processor/core systems, busy waiting can improve responsiveness

29

Priority Inversion

- Busy waiting has other side-effects as well, such as the "priority inversion" problem:
  - Consider a pair of processes with higher and lower priority
  - The higher priority process gets to run whenever it is ready, but the lower priority process holds the lock
  - The lower-priority process never gets to run to release the lock
  - This assumes there is no "aging" – a temporary increase in priority while waiting to run

30

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (integer or pointer to Process/Thread structure)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

31

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){
     value--;
     if (value < 0) {
             add this process to waiting queue
             block();  }
}
```

- Implementation of signal:

```
signal (S){
        value++;
        if (value <= 0) {
                remove a process P from the waiting queue
                wakeup(P);  }
}
```

32

Pthreads Synchronization

- Pthreads specification provides mutexes
- Another POSIX specification (POSIX.1b) provides semaphores
- Mutex = binary semaphore (as opposed to counting semaphore)
- Counting semaphores can be constructed from mutexes

33

Synchronization

- Mutual exclusion (locks)
  - Ensure certain operations on certain data can be performed by only one process at a time
  - Area that only one thread can enter at a time
  - No ordering guarantees
- Event synchronization
  - Ordering of events to preserve dependences
    - e.g. producer —> consumer of data

34

Pthread Mutexes

- Create a mutex

```
pthread_mutex_t mutex;
pthread_mutexattr_t *attr = NULL;

int res = pthread_mutex_init( &mutex, attr);
```

– returns 0 on success, an error code otherwise

35

Pthread Mutex Attributes

- Attributes can include
- PTHREAD_PROCESS_SHARED
  - Vs PTHREAD_PROCESS_PRIVATE
- #ifndef _POSIX_THREAD_PROCESS_SHARED
  - Then you can't do it
  - Why?
- PTHREAD_MUTEX_ERRORCHECK
  - Checks for attempts to relock a mutex and indicates and error
- PTHREAD_MUTEX_RECURSIVE
  - Allows the same thread to recursively lock a mutex
  - Requires corresponding number of unlocks

36

Simple Pthread Mutex Example

```
pthread_mutex_t mutex;
pthread_mutexattr_t *attr = NULL;

res = pthread_mutex_init(&mutex,
  attr);

res = pthread_mutex_lock(&mutex);
// do things
res = pthread_mutex_unlock(&mutex);
```

37

Mutexes in Pthreads - Summary

- pthread_mutex_t *mutexp;
- pthread_mutex_attr_t *attr;

- int pthread_mutex_init(mutexp, attr);
- int pthread_mutex_lock(mutexp);
  - Suspends the calling thread if necessary
- int pthread_mutex_unlock(mutexp);
- int pthread_mutex_trylock(mutexp);
  - returns EBUSY if the mutex is already locked
  - Think carefully about when to use this
- int pthread_mutex_destroy(mutexp);

38

POSIX Semaphores

- Defined as part of the POSIX real-time extensions (POSIX.1b)
  - System V also has semget(), semop()
- int sem_init(sem_t *sem, int pshared, unsigned int value);
- int sem_wait(sem_t * sem);
- int sem_trywait(sem_t * sem);
- int sem_post(sem_t * sem);
- int sem_getvalue(sem_t * sem, int * sval);
- int sem_destroy(sem_t * sem);

39

Condition Variables

- Pthreads also provides “condition variables”
- Condition variables allow threads to synchronize based on another thread’s activity
  - Rather than just controlling access
- Condition variables are of the type pthread_cond_t
- They are used in conjunction with mutex locks
- Condition variables can eliminate the need for polling
  - Which itself might require locking a mutex…

40

pthread_cond_init()

- Creating a condition variable

```
int pthread_cond_init(
      pthread_cond_t *cond,
     const pthread_condattr_t *attr);
```

  - returns 0 on success, an error code otherwise
  - **cond**: output parameter, condition
  - **attr**: input parameter, attributes (default = NULL)

41

pthread_cond_wait()

Waiting on a condition variable

```
int pthread_cond_wait(
      pthread_cond_t *cond,
     pthread_mutex_t *mutex);
```

returns 0 on success, an error code otherwise
cond: input parameter, condition
mutex: input parameter, associated mutex

42

pthread_cond_signal()

Signaling a condition variable

```
int pthread_cond_signal(
         pthread_cond_t *cond;
```

returns 0 on success, an error code otherwise
cond: input parameter, condition

"Wakes up" one thread out of the possibly many threads waiting for the condition

The thread is chosen non-deterministically

43

pthread_cond_broadcast()

Signaling a condition variable

```
int pthread_cond_broadcast(
         pthread_cond_t *cond;
```

returns 0 on success, an error code otherwise
cond: input parameter, condition

"Wakes up" ALL threads waiting for the condition

May be useful in some applications

44

Condition Variable: example

- Multiple threads waiting until a counter reaches a maximum value

```
pthread_mutex_lock(&lock);
while (count < MAX_COUNT) {
  pthread_cond_wait(&cond,&lock);
}
pthread_mutex_unlock(&lock)
```

  - Locking the lock so that we can read the value of count without the possibility of a race condition
  - Calling **`pthread_cond_wait()`** in a while loop to verify the condition
  - When going to sleep the **`pthread_cond_wait()`** function implicitly releases the lock
  - When waking up the **`pthread_cond_wait()`** function implicitly acquires the lock
  - The lock is unlocked after exiting from the loop

45

pthread_cond_timed_wait()

Waiting on a condition variable with a timeout

```
int pthread_cond_timedwait(
      pthread_cond_t *cond,
     pthread_mutex_t *mutex,
     const struct timespec *delay);
```

returns 0 on success, an error code otherwise
`cond`: input parameter, condition
`mutex`: input parameter, associated mutex
`delay:` input parameter, timeout (same fields as the one used for gettimeofday)

46

Linux: Futex

- Linux implements a Fast Userspace Mutex (futex) that operates on a variable stored in userspace
- Programs manipulate the variable using atomic operations
- The kernel maintains a queue and when the variable is under contention processes invoke a system call to block execution
  - Otherwise unnecessary
- Windows calls it WaitOnAddress

47