Synchronization

Raju Pandey
Department of Computer Sciences
University of California, Davis
Winter 2005

Concurrency

- Reasons for concurrency:
  - Multiple applications
    - Multiprogramming
  - Structured application
    - Application can be a set of concurrent processes
  - Operating-system structure
    - Operating system is a set of processes or threads

- Difficulties due to concurrency:
  - Sharing of global resources
  - Operating system managing the allocation of resources optimally
  - Difficult to locate programming errors
Temporal relations

- Instructions executed by a single thread are totally ordered
  - $A < B < C < \ldots$
- Absent synchronization, instructions executed by distinct threads must be considered unordered / simultaneous
  - Not $A < A'$, and not $A' < A$

Example

```
main()
A
B

pthread_create()
foo()
A'

C
```

$Y$-axis is “time.”

Could be one CPU, could be multiple CPUs (cores).

- $A < B < C$
- $A' < B'$
- $A < A'$
- $C == A'$
- $C == B'$
Syntax for Process Creation

- **cobegin/coend**
  - syntax: cobegin C1 // C2 // ... // Cn coend
  - meaning:
    - All Ci may proceed concurrently
    - When all terminate, the statement following cobegin/coend continues

```
cobegin
    Time_Date //
    Mail //
    Edit; cobegin
        (Compile; Load; Execute) //
        Edit; cobegin
        Print // Web
    coend
coend
coend;
```

---

Process Interactions

- **Competition: The Critical Problem**
  - ```
    x = 0;
    cobegin
    p1: ...
    x = x + 1;
    ...
    //
    p2: ...
    x = x + 1;
    ...
    coend
  ```
  - x should be 2 after both processes execute
  - Interleaved execution (due to parallel processing or context switching):
    ```
    p1: R1 = x;
    R1 = R1 + 1;
    x = R1;
    ...
    p2: ...
    R2 = x;
    R2 = R2 + 1;
    ...
    ```
  - x has only been incremented once; The first update (x=R1) is lost.
**Critical Sections / Mutual Exclusion**

- Sequences of instructions that may get incorrect results if executed simultaneously are called **critical sections**
- (We also use the term **race condition** to refer to a situation in which the results depend on timing)
- **Mutual exclusion** means "not simultaneous"
  - $A < B$ or $B < A$
  - We don't care which
- Forcing mutual exclusion between two critical section executions is sufficient to ensure correct execution – guarantees ordering
- One way to guarantee mutually exclusive execution is using **locks**

---

**Critical sections**

Critical sections is the "happens-before" relation

- $T_1$ and $T_2$
- Possibly incorrect
- Correct
- Correct
When do critical sections arise?

- One common pattern: Read-modify-write of
  - A shared value (variable)
  - In code that can be executed concurrently
    (Note: There may be only one copy of the code (e.g., a procedure), but it can be executed by more than one thread at a time)
- Shared variable:
  - Globals and heap-allocated variables
  - NOT local variables (which are on the stack)
    (Note: Never give a reference to a stack-allocated (local) variable to another thread, unless you’re superhumanly careful ...)

Example: buffer management

- Threads cooperate in multithreaded programs
  - to share resources, access shared data structures
    - e.g., threads accessing a memory cache in a web server
  - also, to coordinate their execution
    - e.g., a disk reader thread hands off blocks to a network writer thread through a circular buffer
Example: shared bank account

- Suppose we have to implement a function to withdraw money from a bank account:

```c
int withdraw(account, amount) {
    int balance = get_balance(account); // read
    balance -= amount; // modify
    put_balance(account, balance); // write
    spit out cash;
}
```

- Now suppose that you and your S.O. share a bank account with a balance of $100.00
  - what happens if you both go to separate ATM machines, and simultaneously withdraw $10.00 from the account?

- Assume the bank’s application is multi-threaded
- A random thread is assigned a transaction when that transaction is submitted
Interleaved schedules

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:

  ```
  balance = get_balance(account);
  balance -= amount;
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  spit out cash;
  put_balance(account, balance);
  spit out cash;
  ```

- What's the account balance after this sequence?
  - who's happy, the bank or you?
- How often is this sequence likely to occur?

Other Execution Orders

- Which interleavings are ok? Which are not?

```c
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  spit out cash;
}
```
Morals:
- Interleavings are hard to reason about
  - We make lots of mistakes
  - Control-flow analysis is hard for tools to get right
- Identifying critical sections and ensuring mutually exclusive access is ... “easier”

Terms related to concurrency
- **Critical section**: Section of code within a program/process that requires access to shared resource and which cannot be accessed while another process is in a corresponding section of code
- **Mutual Exclusion**: Requirement that when a process is in a critical section that accesses a shared resource, no other process may be in a critical section that access any of those resources
- **Race conditions**: A situation in which multiple threads or processes read or write a shared data item and the final result depends on the relative timing of their execution
- **Deadlock**: Situation where two or more processes are unable to proceed because each is waiting for other to do something
- **Livelock**: Situation where 2 or more processes continuously change their state in response to changes in others without doing any useful work
- **Starvation**: A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.
- **Fairness**: A constraint that ensures every process gets to run
# Process Interaction

<table>
<thead>
<tr>
<th>Degree of Awareness</th>
<th>Relationship</th>
<th>Influence the one process has on another</th>
<th>Potential control problem</th>
</tr>
</thead>
</table>
| Process unaware of each other        | Competition           | • Results of one process independent of the action of others  
• Timing of process may be affected | • Mutual exclusion  
• Deadlock (renewable resources)  
• Starvation |
| Process indirectly unaware of each other | Cooperation by sharing | • Results of one process may depend on information obtained from others  
• Timing of process may be affected | • Mutual exclusion  
• Deadlock (renewable resources)  
• Starvation  
• Data coherence |
| Processes directly aware of each other | Cooperation by communication | • Results of one process may depend on information obtained from others  
• Timing of process may be affected | • Deadlock (consumable resources)  
• Starvation |

**Requirements for Mutual Exclusion**

- **mutual exclusion**
  - at most one thread is in the critical section
- **progress**
  - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
- **bounded waiting** (no starvation)
  - if thread T is waiting on the critical section, then T will eventually enter the critical section  
  o assumes threads eventually leave critical sections
- **performance**
  - the overhead of entering and exiting the critical section is small with respect to the work being done within it
Mutual Exclusion: Hardware Support

• Interrupt Disabling
  ▪ Disabling interrupts guarantees mutual exclusion
  ▪ Processor is limited in its ability to interleave programs
  ▪ Disadvantages:
    o Responsiveness of system reduced
    o May lose some important interrupts
    o Does not work on multi-processing systems
      disabling interrupts on one processor will not guarantee mutual exclusion

While (true) {
  disable interrupts
  critical section
  enable interrupts
  remainder
}

Mutual Exclusion: Hardware Support

• Special Machine Instructions
  ▪ Performed in a single instruction cycle
  ▪ Access to the memory location is blocked for any other instructions

  **Test and Set**

  ```java
  boolean testset (int i) {
    if (i == 0) {
      i = 1;
      return true;
    } else return false;
  }
  ```

  **Exchange Instruction**

  ```java
  void exchange (int register, int memory) {
    int temp;
    temp = memory;
    memory = register;
    register = temp;
  }
  ```
Mutual Exclusion (Test and Set)

```c
const int n = /* number of processes */
int bolt;
void P(int i) {
    while (true) {
        while (! test_and_set (bolt))
            /* do nothing */
        /* Critical Section */
        bolt = 0;
        /* Reminder */
    }
}
void main ()
{
    bolt = 0;
    cobegin
        P(1); P(2); ...; P(n);
    coend
}
```

Mutual Exclusion (Exchange)

```c
const int n = /* number of processes */
int bolt;
void P(int i) {
    int keyi;
    while (true) {
        keyi = 1;
        while (keyi != 0)
            exchange(keyi, bolt);
        /* Critical Section */
        exchange(keyi, bolt);
        /* Reminder */
    }
}
void main ()
{
    bolt = 0;
    cobegin P(1); P(2); ...; P(n); coend
}
```
Mutual Exclusion Machine Instructions

- **Advantages**
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

- **Disadvantages**
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs it, the higher priority process will obtain the processor to wait for the critical region

Software: Algorithm 1

- Use a single “turn” variable:
  ```c
  int turn = 1;
  cobegin
      p1: while (1) {
          while (turn==2); /*wait*/
          CS1; turn = 2; program1;
      }
      p2: while (1) {
          while (turn==1); /*wait*/
          CS1; turn = 1; program1;
      }
  // ...
  ```

- Violates blocking requirement;
- p1 can block p2 even if it is not inside critical section
Software Solutions: Algorithm 2

- Use two variables to indicate intent:

  ```
  int c1 = 0, c2 = 0;
  cobegin
  p1: while (1) {
      c1 = 1;
      while (c2); /*wait*/
      CS1; c1 = 0; program1;
  }
  p2: while (1) {
      c2 = 1;
      while (c2); /*wait*/
      CS1; c2 = 0; program2;
  }
  ```

- What if they access c1 and c2 at the same time?

Software Solutions: Algorithm 3

- Similar to #2, but reset intent variable each time:

  ```
  int c1 = 0, c2 = 0;
  cobegin
  p1: while (1) {
      c1 = 1;
      if (c2) c1 = 0;
      else {CS1; c1 = 0; program1}
  }
  p2: while (1) {
      c2 = 1;
      if (c1) c2 = 0;
      else {CS1; c2 = 0; program2}
  }
  ```

- What if p1 and p2 operate at same speed: livelock
- What if p2 always checks c1 after c2 has been set to 1: fairness
Software Solutions: Algorithm 4 (Peterson)

- Like #2 but use a "turn" variable to break a tie:
  ```
  int c1 = 0, c2 = 0, WillWait;
  cobegin
      p1: while (1) {
          c1 = 1;
          WillWait = 1;
          while (c2 && (WillWait==1)); /*wait*/
          CS1; c1 = 0; program1;
      }
      p2: while (1) {
          c2 = 1;
          WillWait = 2;
          while (c1 && (WillWait==2)); /*wait*/
          CS1; c2 = 0; program2;
  }
  ```

- Does it guarantee mutual exclusion?
- What about deadlock? What about livelock? Fairness?

Locks

- A lock is a memory object with two operations:
  - `acquire()`: obtain the right to enter the critical section
  - `release()`: give up the right to be in the critical section
- `acquire()` prevents progress of the thread until the lock can be acquired
- (Note: terminology varies: acquire/release, lock/unlock)
Locks: Example

**Acquire/Release**

- Threads pair up calls to `acquire()` and `release()`
  - between `acquire()` and `release()`, the thread holds the lock
  - `acquire()` does not return until the caller “owns” (holds) the lock
    - at most one thread can hold a lock at a time
  - What happens if the calls aren’t paired?
  - What happens if the two threads acquire different locks?
    - (granularity of locking)
Using locks

```c
int withdraw(account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    release(lock);
    spit out cash;
}
```

- What happens when green tries to acquire the lock?

Spinlocks

- How do we implement locks? Here’s one attempt:

```c
struct lock_t {    
    int held = 0;
};
void acquire(lock) {    
    while (lock->held);    
    lock->held = 1;
}
void release(lock) {    
    lock->held = 0;
}
```

- Why doesn’t this work?
  - where is the race condition?
Implementing locks (cont.)

• Problem is that implementation of locks has critical sections, too!
  • the acquire/release must be atomic
    o atomic == executes as though it could not be interrupted
    o code that executes "all or nothing"

• Need help from the hardware
  • atomic instructions
    o test-and-set, compare-and-swap, ...
  • disable/reenable interrupts
    o to prevent context switches

Spinlocks redux: Hardware Test-and-Set

• CPU provides the following as one atomic instruction:

```c
bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

• Remember, this is a single atomic instruction ...
Implementing locks using Test-and-Set

- So, to fix our broken spinlocks:

```c
struct lock {
    int held = 0;
}
void acquire(lock) {
    while(test_and_set(&lock->held));
}
void release(lock) {
    lock->held = 0;
}
```

- mutual exclusion? (at most one thread in the critical section)
- progress? (T outside cannot prevent S from entering)
- bounded waiting? (waiting T will eventually enter)
- performance? (low overhead)

Reminder of use ...

```c
int withdraw(account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    release(lock);
    spit out cash;
}
```

- How does a thread blocked on an “acquire” (that is, stuck in a test-and-set loop) yield the CPU?
  - calls yield( ) (spin-then-block)
  - there’s an involuntary context switch
Problems with spinlocks

- Spinlocks work, but are horribly wasteful!
  - if a thread is spinning on a lock, the thread holding the lock cannot make progress
    - You’ll spin for a scheduling quantum
  - (pthread_spin_t)

- Only want spinlocks as primitives to build higher-level synchronization constructs
  - Why is this okay?

- We’ll see later how to build blocking locks
  - But there is overhead – can be cheaper to spin
  - (pthread_mutex_t)

Another approach: Disabling interrupts

```c
struct lock {
  
void acquire(lock) {
    cli(); // disable interrupts
  }

void release(lock) {
    sti(); // reenable interrupts
  }
```

Problems with disabling interrupts

- Only available to the kernel
  - Can’t allow user-level to disable interrupts!
- Insufficient on a multiprocessor
  - Each processor has its own interrupt mechanism
- “Long” periods with interrupts disabled can wreak havoc with devices

- Just as with spinlocks, you only want to use disabling of interrupts to build higher-level synchronization constructs

Problems with suggested solution

- Difficult to understand and verify
- Solution is applicable only in cases when processes are competing for resources
- Busy waiting
  - Slows down overall system
- Fairness is not guaranteed or enforced

- Solution: Language/System primitives
  - Semaphores
  - Event synchronization
  - Monitors and others
A **Semaphore** is a *non-negative integer, s* (how many tasks can proceed simultaneously), and *two indivisible operations*:

- **P(s)**, often written **Wait(s)**; think “Pause”:
  - “P” from “passaren” (“pass” in Dutch) or from “prolagan,” combining “proberen” (“try”) and “verlagen” (“decrease”).
  - \[ \text{while} \ (s<1)/^*/ \text{wait}^*/; \ s=s-1 \]
- **V(s)**, often written **Signal(s)**; think of the “V for Victory” 2-finger salute:
  - “V” from “vrigeven” (“release”) or “verhogen” (“increase”).
  - \[ s=s+1; \]

```c
struct semaphore {
    int count;
    queueType queue;
}

void semWait(semaphore s) {
    s.count --;
    if (s.count < 0) {
        place this process in s.queue;
        block this process;
    }
}

void semSignal(semaphore s) {
    s.count++;
    if (s.count <= 0) {
        remove process P from s.queue;
        place process P on ready list;
    }
}
```
Binary Semaphore Primitives

```c
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};
void semWaitB(binary_semaphore s) {
    if (s.value == one) s.value = zero;
    else {
        place this process in s.queue;
        block this process;
    }
}
void semSignal(binary_semaphore s) {
    if (s.queue.is_empty())
        s.value = one;
    else {
        remove process P from s.queue;
        place process P on ready list;
    }
}
```

Mutual Exclusion with Semaphores

```c
semaphore s;
const int n = /* number of processes
void P(int i) {
    while (true) {
        semWait(s);
        /* critical section */
        semSignal(s);
        /* remainder */
    }
}
void main() {
    s.count = 1; /* initialize the semaphore */
    cobegin P(1); P(2); ...; P(n); coend
}
Producer/Consumer Problem

- Processes: producer and consumer
  - One or more producers are generating data and placing these in a buffer
  - A single consumer is taking items out of the buffer one at time
- Sharing:
  - A common buffer
- Synchronization problem:
  - Only one producer or consumer may access the buffer at any one time

```
Producer
while (true) {
    /* produce item v */
    b[in] = v;
    in++;
}

Consumer
while (true) {
    while (in <= out)
        /* do nothing */;
    w = b[out];
    out++;
    /* consume item w */
}
```
Solution 1

```c
int n; binary_semaphore s = 1; binary_semaphore delay = 0;

void producer() {
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n == 1)
            semSignalB(delay);
        semSignalB(s);
    }
}

void consumer() {
    semWaitB(delay):
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n == 0)
            semWaitB(delay);
    }
}

void main() {
    n = 0; cobegin producer(); consumer(); coend
}
```

Solution 2

```c
int n; binary_semaphore s = 1; binary_semaphore delay = 0;

void producer() {
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n == 1)
            semSignalB(delay);
        semSignalB(s);
    }
}

void consumer() {
    int m;
    semWaitB(delay):
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m == 0)
            semWaitB(delay);
    }
}

void main() {
    n = 0; cobegin producer(); consumer(); coend
}
```
Solution 3

semaphore n = 0; semaphore s = 1;

void producer() {
    while (true) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}

void consumer() {
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}

void main() {
    cobegin producer(); consumer(); coend
}

Solution to bounded buffer problem

const int sizeofbuffer = /* buffers size */
semaphore n = 0; semaphore s = 1;
semaphore e = sizeofbuffer;

void producer() {
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}

void consumer() {
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}

void main() {
    cobegin producer(); consumer(); coend
}
Motivation

- Semaphores are:
  - Powerful but low-level abstractions
    - Programming with them is highly error prone
    - Such programs are difficult to design, debug, and maintain
  - Not usable in distributed memory systems
- Need higher-level primitives: Based on semaphores or messages
- Monitors (Hoare, 1974)
  - Follow principles of abstract data type (object-oriented) programming:
    - A data type is manipulated only by a set of predefined operations
  - A monitor is
    1. A collection of data representing the state of the resource controlled by the monitor, and
    2. Procedures to manipulate that resource data

Monitors

- Implementation must guarantee:
  1. Resource accessible only by monitor procedures
  2. Monitor procedures are mutually exclusive
- For coordination, monitors provide:
  - `c.wait`: Calling process is blocked and placed on waiting queue associated with condition variable `c`
  - `c.signal`: Calling process wakes up first process on `c` queue
  - “condition variable” `c` is not a conventional variable
    - `c` has no value
    - `c` is an arbitrary name chosen by programmer to designate an event, state, or condition
    - Each `c` has a waiting queue associated
    - A process may “block” itself on `c` -- it waits until another process issues a signal on `c`
Hoare Monitors

- After \texttt{c.signal}, there are 2 ready processes:
  - The calling process which did the \texttt{c.signal}
  - The process which the \texttt{c.signal} “woke up”
- Which should continue?
  (Only one can be executing inside the monitor!)
- Hoare monitor:
  - Woken-up process continues
  - Calling process is placed on high-priority queue

Effect of \texttt{wait}

Effect of \texttt{signal}
Monitor-based solution for Bounded buffers

```c
monitor boundedbuffer;
char buffer[N]; int nextin, nextout; int count;
cond notfull, notempty;
void append(char x) {
    if (count == N) notfull.wait(); /* buffer is full*/
    buffer[nextin] = x; nextin = (nextin+1) mod N;
    count++;
    notempty.signal();
}
void take(char x) {
    if (count == 0) notempty.wait(); /* emptt buffer */
    x = buffer[nextout];
    nextout = (nextout+1) mod N;
    count--;
    notfull.signal();
}
void init() {
    nextin = 0; nextout = 0; count = 0;
}
```

Monitor-based solution for Bounded buffers

```c
void producer() {
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}

void consumer() {
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}

void main() {
    cobegin producer(); consumer(); coend
}
```
Readers/Writers Problem

- Processes: reader and writers
  - Readers: read file
  - Writers: write to file
- Sharing: common file
- Synchronization constraints:
  - Any number of readers may simultaneously read the file
  - Only one writer at a time may write to the file
  - If a writer is writing to the file, no reader may read it
  - Prevent starvation of either process type (variation 1)
    - If Rs are in CS, a new R must not enter if W is waiting
    - If W is in CS, once it leaves, all Rs waiting should enter
      (even if they arrived after new Ws)

Reader/Writer Solution

```c
int readcount; semaphore x = 1, wsem = 1;

void reader() {
    while (true) {
        semWait(x);
        readcount++;
        if (readcount == 1)
            semWait(wsem);
        semSignal(x);
        READUNIT();
        semWait(x);
        readcount--;
        if (readcount == 0)
            semSignal(wsem);
        semSignal(x);
    }
}

void writer() {
    while (true) {
        semWait(wsem);
        WRITEUNIT();
        semSignal(wsem);
    }
}

void main() {
    readcount = 0;
    cobegin producer(); consumer(); coend
}
```

Readers have priority
Reader/Writer Solution

```c
int readcount, writecount;
semaphore x = 1, y=1, z=1, wsem = 1, rsem=1;

void reader() {
    while (true) {
        semWait(z);
        semWait(rsem);
        semWait(x);
        readcount++;
        if (readcount == 1)
            semWait(wsem);
        semSignal(x);
        semSignal(rsem);
        semSignal(z);
        READUNIT();
        semWait(x);
        readcount--;
        if (readcount == 0)
            semSignal(wsem);
        semSignal(x);
    }
}

void writer() {
    while (true) {
        semWait(y);
        writecount++;
        if(writecount == 1)
            semWait(rsem);
        semSignal(y);
        semWait(wsem);
        WRITEUNIT();
        semSignal(wsem);
        semWait(y);
        writecount--;
        if (writecount == 0)
            semSignal(rsem);
        semSignal(y);
    }
}

void main() {
    readcount = writecount = 0;
cobegin producer(); consumer(); coend
}
```

Solution using monitor

```c
monitor Readers_Writers {
    int readCount=0, writing=0;
    condition OK_R, OK_W;
    start_read() {
        if (writing || !empty(OK_W)) OK_R.wait;
        readCount = readCount + 1;
        OK_R.signal;
    }
    end_read() {
        readCount = readCount - 1;
        if (readCount == 0) OK_W.signal;
    }
    start_write() {
        if ((readCount != 0)||writing) OK_W.wait;
        writing = 1;
    }
    end_write() {
        writing = 0;
        if (!empty(OK_R)) OK_R.signal;
        else OK_W.signal;
    }
}
```