Synchronization Principles

CS 256/456

Dept. of Computer Science, University of Rochester

Race Condition

- **Race condition**:
  - The situation where several processes access and manipulate shared data concurrently.
  - The final value of the shared data and/or effects on the participating processes depends upon the order of process execution – nondeterminism.
- To prevent race conditions, concurrent processes must be **synchronized**.

The Critical-Section Problem

- Problem context:
  - *n* processes all competing to use some shared data
  - Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Find a solution that satisfies the following:
  1. **Mutual Exclusion**. No two processes simultaneously in the critical section.
  2. **Progress**. No process running outside its critical section may block other processes.
  3. **Bounded Waiting/Fairness**. Given the set of concurrent processes, a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical Section for Two Processes

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.
- Assumption: instructions are atomic and no re-ordering of instructions.

Algorithm 1

- Shared variables:
  - `int turn;` initially `turn = 0;`
  - `turn==i` ⇒ P_i can enter its critical section
- Process P_i

```
do {
   while (turn != i) ;
      critical section
   turn = j;
      remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables:
  - `boolean flag[2];` initially `flag[0] = flag[1] = false;`
  - `flag[i]==true` ⇒ P_i ready to enter its critical section
- Process P_i

```
do {
   flag[i] = true;
   while (flag[j]) ;
      critical section
   flag[i] = false;
      remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 3

- Combine shared variables of algorithms 1 and 2.
- Process P_i

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn==j) ;
      critical section
   flag[i] = false;
      remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes. ⇒ called Peterson's algorithm.

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-op (e.g., increment) – returns value and atomically performs op (e.g., increments it)
- Compare-and-swap – compares the contents of two locations and swaps if identical
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value

Synchronization Using Special Instruction: TSL (test-and-set)

```
entry_section:
   TSL R1, LOCK          | copy lock to R1 and set lock to 1
   CMP R1, #0            | was lock zero?
   JNE entry_section     | if it wasn't zero, lock was set, so loop
   RET                   | return; critical section entered

exit_section:
   MOV LOCK, #0          | store 0 into lock
   RET                   | return; out of critical section
```

- Does it solve the synchronization problem?
- Does it work for multiple (>2) processes?

Using ll/sc for Atomic Exchange

- Swap the contents of R4 with the memory location specified by R1

```
try: mov R3, R4      ; mov exchange value
     ll   R2, 0(R1)  ; load linked
     sc R3, 0(R1)    ; store conditional
     beqz R3, try    ; branch if store fails
     mov  R4, R2     ; put load value in R4
```


Solving the Critical Section Problem with Busy Waiting

- In all our solutions, a process enters a loop until the entry is granted ⇒ busy waiting.
- Problems with busy waiting:
  - Waste of CPU time
  - If a process is switched out of CPU during critical section
    - other processes may have to waste a whole CPU quantum
    - may even deadlock with strictly prioritized scheduling (priority inversion problem)
- Solution
  - Avoid busy wait as much as possible (yield the processor instead).
  - If you can't avoid busy wait, you must prevent context switch during critical section (disable interrupts while in the kernel)

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable which can only be accessed via two atomic operations
- Semantics (roughly) of the two operations:

```
wait(S) or P(S):
   wait until S>0;
   S--;

signal(S) or V(S):
   S++;
```

- Solving the critical section problem:

Shared data:

```
semaphore mutex=1;
```

Process *Pi:*

```
wait(mutex);
    critical section
signal(mutex);
    remainder section
```

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
    int value;
    proc_list *L;
} semaphore;
```

- Assume two simple operations:
  - `block` suspends the process that invokes it.
  - `wakeup(P)` resumes the execution of a blocked process `P`.

- Semaphore operations now defined as (both are atomic):

```
wait(S):
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        block;
    }
signal(S):
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
```

How do we make sure wait(S) and signal(S) are atomic? So have we truly removed busy waiting?

Mutex Lock (Binary Semaphore)

- Mutex lock – a semaphore with only two state: locked/unlocked
- Semantics of the two (atomic) operations:

```
lock(mutex):
    wait until mutex==unlocked;
    mutex=locked;

unlock(mutex):
    mutex=unlocked;
```

- Can you implement mutex lock using semaphore?
- How about the opposite?

Implement Semaphore Using Mutex Lock

- Data structures:

```
mutex_lock L1, L2;
int C;
```

- Initialization:

```
L1 = unlocked;
L2 = locked;
C = initial value of semaphore;
```

- *wait* operation:

```
lock(L1);
C --;
if (C < 0) {
    unlock(L1);
    lock(L2);
}
unlock(L1);
```

- *signal* operation:

```
lock(L1);
C ++;
if (C <= 0)
    unlock(L2);
else
    unlock(L1);
```


Classical Problems of Synchronization

- Bounded-Buffer Problem
- Dining-Philosophers Problem

Bounded Buffer Problem

- Shared data

```
buffer;
```

- Producer process

```
while (1) {
    ...
    produce an item in nextp;
    ...
    add nextp to buffer;
    ...
}
```

- Consumer process

```
while (1) {
    ...
    remove an item from buffer to nextc;
    ...
    consume nextc;
    ...
}
```

- Protecting the critical section for safe concurrent execution.
- Synchronizing producer and consumer when buffer is empty/full.

Bounded Buffer Solution

- Shared data

```
buffer;
semaphore full=0;
semaphore empty=n;
semaphore mutex=1;
```

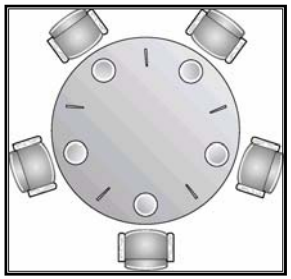
- Producer process

```
while (1) {
    ...
    produce an item in nextp;
    ...
    wait(empty);
    wait(mutex);
    add nextp to buffer;
    signal(mutex);
    signal(full);
    ...
}
```

- Consumer process

```
while (1) {
    ...
    wait(full);
    wait(mutex);
    remove an item from buffer to nextc;
    signal(mutex);
    signal(empty);
    ...
    consume nextc;
    ...
}
```


Dining-Philosophers Problem



- Philosopher i (1 ≤ i ≤ 5):

```
while (1) {
    ...
    eat;
    ...
    think;
    ...
}
```

- eating needs both chopsticks (the left and the right one).

Dining-Philosophers: Possible Solution

- Shared data:

```
semaphore chopstick[5];
```

Initially all values are 1;

- Philosopher i:

```
while(1) {
    ...
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    eat;
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think;
    ...
};
```

Deadlock?

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- Native support for mutual exclusion.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
      . . .
    }
    procedure body Pn (...) {
       . . .
    }
    {
      initialization code
    }
}
```


Condition Variables in Monitors

- To allow a process to wait within the monitor, a `condition` variable must be declared, as

```
condition x, y;
```

- Condition variable can only be used with the operations `wait` and `signal`.
  - The operation

```
x.wait();
```

    means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

  - The `x.signal` operation resumes exactly one suspended process. If no process is suspended, then the `signal` operation has no effect.
- Unlike semaphore, there is no counting in condition variables

Two Semantics of Condition Variables

- Hoare semantics:
  - p_0 executes signal while p_1 is waiting $\Rightarrow$ p_0 immediately yields the monitor to p_1
  - The logical condition holds when P_1 gets to run

```
if (resourceNotAvailable()) Condition.wait();
/* now available ... continue ... */
. . .
```

- Alternative semantics:
  - p_0 executes signal while p_1 is waiting $\Rightarrow$ p_0 continues to execute, then when p_0 exits the monitor p_1 can receive the signal
  - The logical condition may not hold when P_1 gets to run
  - Brinch Hansen ("Mesa") semantics: p_0 must exit the monitor after a signal

Dining Philosophers Solution

```
monitor dp {
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];
    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown (int i) {
        state[i] = THINKING;
        test((i+4)%5);
        test((i+1)%5);
    }

    void test (int i) {
        if (state[(i+4)%5]!=EATING && state[(i+1)%5]!=EATING && state[i] == HUNGRY) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    void init() {
        for (int i=0; i<5; i++)
            state[i] = THINKING;
    }
}
```

Dining Philosophers Alternative Solution

```
monitor dp {
   enum {thinking, eating} state[5];
   condition cond[5];

   void pickup(int i) {
      while (state[(i+4)%5]==eating || state[(i+1)%5]==eating)
         cond[i].wait();
      state[i] = eating;
   }

   void putdown(int i) {
      state[i] = thinking;
      cond[(i+4)%5].signal();
      cond[(i+1)%5].signal();
   }

   void init() {
      for (int i=0; i<5; i++)
         state[i] = thinking;
   }
}
```


Synchronization in Practice

- User program synchronization
  - for threads
  - for processes
- OS kernel synchronization

User Program Synchronization for Threads

- All threads share the same address space
- When there is a need to protect a short critical section
  - software/hardware spin locks (busy waiting is okay)
  - still has the risk of context switch in the middle of critical section
- For complex synchronization
  - semaphore, mutex lock, condition variable, … (busy waiting is NOT okay)
  - may need kernel help
- In pthreads
  - mutex lock and condition variable
  - condition variable must be used together with a mutex lock

Synchronization Primitives in Pthreads

- Mutex lock
  - pthread_mutex_init
  - pthread_mutex_destroy
  - pthread_mutex_lock
  - pthread_mutex_unlock
- Condition variable (used in conjunction with a mutex lock)
  - pthread_cond_init
  - pthread_cond_destroy
  - pthread_cond_wait
  - pthread_cond_signal
  - pthread_cond_broadcast

User Program Synchronization for Processes

- Processes naturally do not share the same address space
- Process synchronization:
  - semaphore
  - shared memory
  - pipes

Mutex Locks: Creation and Destruction

```
pthread_mutex_init(
    pthread_mutex_t * mutex,
    const pthread_mutex_attr *attr);
```

- Creates a new mutex lock

```
pthread_mutex_destroy(
    pthread_mutex_t *mutex);
```

- Destroys the mutex specified by mutex

Mutex Locks: Lock

```
pthread_mutex_lock(
    pthread_mutex_t *mutex)
```

- Tries to acquire the lock specified by mutex.
- If mutex is already locked, then calling thread blocks until mutex is unlocked.

Mutex Locks: UnLock

```
pthread_mutex_unlock(
    pthread_mutex_t *mutex);
```

- If calling thread has mutex currently locked, this will unlock the mutex.
- If other threads are blocked waiting on this mutex, one will unblock and acquire mutex
- Which one is determined by the scheduler

Condition variables: Creation and Destruction

```
pthread_cond_init(
    pthread_cond_t *cond,
    pthread_cond_attr *attr)
```

- Creates a new condition variable cond

```
pthread_cond_destroy(
    pthread_cond_t *cond)
```

- Destroys the condition variable cond.

Condition Variables: Wait

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

- Blocks the calling thread, waiting on cond
- Unlocks the mutex
- Re-acquires the mutex when unblocked

Condition Variables: Signal

```
pthread_cond_signal(
    pthread_cond_t *cond)
```

- Unblocks one thread waiting on cond.
- Which one is determined by scheduler.
- If no thread waiting, then signal is a no-op.

Condition Variables: Broadcast

```
pthread_cond_broadcast(
    pthread_cond_t *cond)
```

- Unblocks all threads waiting on cond.
- If no thread waiting, then broadcast is a no-op.

Use of Condition Variables

- IMPORTANT NOTE: A signal is "forgotten" if there is no corresponding wait that has already occurred
- Use semaphores (or construct a semaphore) if you want the signal to be remembered

TSP (Traveling Salesman)

- Goal:
  - given a list of cities, a matrix of distances between them, and a starting city,
  - find the shortest tour in which all cities are visited exactly once.
- Example of an NP-hard search problem.
- Algorithm: branch-and-bound.

Branching

- Initialization:
  - go from starting city to each of remaining cities
  - put resulting partial path into priority queue ordered by its current length
- Further (repeatedly):
  - take head element out of priority queue
  - expand by each one of remaining cities
  - put resulting partial path into priority queue

Finding the Solution

- Eventually, a complete path will be found
- Remember its length as the current shortest path
- Every time a complete path is found, check if we need to update current best path
- When priority queue becomes empty, best path is found

Using a Simple Bound

- Once a complete path is found, we have a lower bound on the length of shortest path
- No use in exploring partial path that is already longer than the current lower bound
- Better bounding methods exist …

Sequential TSP: Data Structures

- Priority queue of partial paths.
- Current best solution and its length.
- For simplicity, we will ignore bounding.

Sequential TSP: Code Outline

```
init_q(); init_best();
while( (p=de_queue()) != NULL ) {
  for each expansion by one city {
        q = add_city(p);
        if( complete(q) ) { update_best(q) };
        else { en_queue(q) };
  }
}
```


Parallel TSP: Possibilities

- Have each process do one expansion
- Have each process do expansion of one partial path
- Have each process do expansion of multiple partial paths
- Issue of granularity/performance, not an issue of correctness.
- Assume: process expands one partial path.

Parallel TSP: First Cut (part 1)

```
process i:
    while( (p=de_queue()) != NULL ) {
      for each expansion by one city {
            q = add_city(p);
            if complete(q) { update_best(q) };
            else en_queue(q);
      }
    }
```


Parallel TSP: First cut (part 2)

- In de_queue: wait if q is empty
- In en_queue: signal that q is no longer empty

Parallel TSP

```
process i:
    while( (p=de_queue()) != NULL ) {
      for each expansion by one city {
            q = add_city(p);
            if complete(q) { update_best(q) };
            else en_queue(q);
      }
    }
```


Parallel TSP: Critical Sections

- All concurrently accessed shared data must be protected by critical section
- Update_best must be protected by a critical section
- En_queue and de_queue must be protected by the same critical section

Termination condition

- How do we know when we are done?
- All processes are waiting inside de_queue.
- Count the number of waiting processes before waiting.
- If equal to total number of processes, we are done.

Parallel TSP: Mutual Exclusion

```
en_queue() / de_queue() {
   pthread_mutex_lock(&queue);
   …;
   pthread_mutex_unlock(&queue);
}
update_best() {
   pthread_mutex_lock(&best);
   …;
   pthread_mutex_unlock(&best);
}
```


Parallel TSP: Condition Synchronization

```
de_queue() {
   pthread_mutex_lock(&queue);
   while( (q is empty) and (not done) ) {
            waiting++;
            if( waiting == p ) {
                     done = true;
                     pthread_cond_broadcast(&empty);
            }
            else {
                     pthread_cond_wait(&empty, &queue);
                     waiting--;
            }
   }
   if( done )
            return null;
   else
            remove and return head of the queue;
    pthread_mutex_unlock(&queue);
}
```


Disclaimer

- Parts of the lecture slides were derived from those by Willy Zwaenepoel, Kai Shen, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).

Implementing Locks Using Test&Set

- On the SPARC ldstub moves an unsigned byte into the destination register and rewrites the same byte in memory to all 1s

```
_Lock_acquire:
    ldstub [%o0], %o1
    addcc %g0, %o1, %g0
    bne _Lock
    nop
fin:
    jmpl %r15+8, %g0
    nop
_Lock_release:
    st %g0, [%o0]
    jmpl %r15+8, %g0
    nop
```