The Standard Concurrency Model

COMS W4995-02

Prof. Stephen A. Edwards
Fall 2002
Columbia University
Department of Computer Science

The Standard Concurrency Model

Provided by most operating systems, both timesharing and real-time.

> Multiple, unsynchronized threads of control updating shared memory simultaneously.

How to make their collective behavior determinstic or at least more disciplined?

(Most material taken from Silberschatz, Galvin, and Gagne, *Operating Systems Concepts*).

Races: Two Simultaneous Writes

Thread 1	**Thread 2**
count = 3	count = 2

At the end, does *count* contain 2 or 3?

Races: A Read and a Write

Thread 1	**Thread 2**
if (count == 2)	count = 2
return TRUE;	
else	
return FALSE;	

If *count* was 3 before these run, does Thread 1 return TRUE or FALSE?

Read-modify-write: Even worse

Consider two thread trying to execute *count* += *1* and *count* += *2* simultaneously.

Thread 1	**Thread 2**
tmp1 = count	tmp2 = count
tmp1 = tmp1 + 1	tmp2 = tmp2 + 2
count = tmp1	count = tmp2

If *count* is initially 1, what outcomes are possible?

Must consider all possible interleavings.

Read-modify-write: Interleaving 1

Thread 1	**Thread 2**
tmp1 = count (=1)	
	tmp2 = count (=1)
	tmp2 = tmp2 + (=3)
	count = tmp2 (=3)
tmp1 = tmp1 + 1 (=2)	
count = tmp1 (=2)	

Read-modify-write: Interleaving 2

Thread 1	**Thread 2**
	tmp2 = count (=1)
tmp1 = count (=1)	
tmp1 = tmp1 + 1 (=2)	
count = tmp1 (=2)	
	tmp2 = tmp2 + 2 (=3)
	count = tmp2 (=3)

Read-modify-write: Interleaving 3

Thread 1	**Thread 2**
tmp1 = count (=1)	
tmp1 = tmp1 + 1 (=2)	
count = tmp1 (=2)	
	tmp2 = count (=2)
	tmp2 = tmp2 + 2 (=4)
	count = tmp2 (=4)

The Critical-Section Problem

Similar considerations apply to any shared resouce (e.g., all types of I/O).

Most things need their states updated *atomically* to ensure an invariant.

Classically, this problem is cast as the *Critical-Section Problem*.

The Critical-Section Problem

Given a system of n processes $P_1, \ldots, P_n$, each of which contains a segment of code called a *critical section*, ensure that no two processes are ever executing their critical sections simultaneously.

Any reasonable solution must guarantee three properties:

1. Safety: If P_k is in its critical region, none of $P_{i \neq k}$ are.
2. Fairness: If $P_{j_1}, \ldots, P_{j_k}$ have all requested to enter their critical regions, one of them will immediately.
3. Bounded Fairness: Only a bounded number of other processes may enter their critical regions once P_k signals its request to do so.

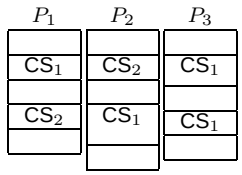
Comments on Critical Sections

There should be one critical section per shared resource.

Multiple shared resources mean multiple sets of critical sections.

Classical problem statement only considers single shared resource.

Actual programs often have many different critical section groups:



The Two-Process Case

Thread 1

```
for (;;) {
  wait_to_enter();
  /* Critical section */
  signal_exit();
  /* other code */
}
```

Thread 2

```
for (;;) {
  wait_to_enter();
  /* Critical section */
  signal_exit();
  /* other code */
}
```

A Solution?

Thread 1

```
for (;;) {
  while (turn != 1);
  /* Critical section */
  turn = 2;
  /* other code */
}
```

Thread 2

```
for (;;) {
  while (turn != 2);
  /* Critical section */
  turn = 1;
  /* other code */
}
```

Does this work?

A Solution?

Thread 1

```
for (;;) {
  while (turn != 1);
  /* Critical section */
  turn = 2;
  /* other code */
}
```

Thread 2

```
for (;;) {
  while (turn != 2);
  /* Critical section */
  turn = 1;
  /* other code */
}
```

This forces execution of the two critical regions to alternate (guarantees safety), but violates fairness.

If thread 2's other code does not terminate, thread 1's critical region can only execute once more. *That's not fair.*

The "After You" Solution

```
bool at_door[2];
int turn;

for (;;) {
  at_door[0] = true;
  turn = 1; /* "After you" */
  /* Wait while friend at door and I said "after you" last */
  while (at_door[1] && turn == 1);

  /* Critical section */

  at_door[0] = false;
  /* other code */
}
```

Basic idea: writes to *turn* must be sequential.

Scenarios

If you're the only one at the door,

1. You arrive at the door and wait.
2. You say "after you" to no one in particular.
3. You immediately notice nobody else is there and proceed through the door.

Scenarios

If your friend arrives at nearly the same time,

1. You both arrive at the door and wait.
2. You and your friend both say "after you" at about the same time.

3a. Your friend says "after you" last, so you proceed through the door.

3b. You say "after you" last, so your friend proceeds through the door.

Multiple-processs Solutions

Problem: What if three people try to go through the door? Previous algorithm only determine who says "after you" *last* (obviously, other person said it first).

We need an alternative.

The Bakery Algorithm

Model: 

(well, almost: no central sign)

Processes (try to) take turns choosing number, then defer to the process with the lowest number.

One wrinkle: doesn't guarantee unique numbers.

In the case of a tie, lowest-numbered *process* goes first.

The Bakery Algorithm

```
bool choosing[N];  /* True when process choosing */
int n[N];          /* Number held by process */

for (;;) {
  choosing[i] = true;
  n[i] = 1 + max(n[0], ..., n[N-1]);
  choosing[i] = false;
  for ( j = 0 ; j < N ; j++ ) {
    while (choosing[j]); /* wait while choosing */
    while (n[j] != 0 &&     /* wait while other */
           (n[j] < n[i] || /* process is lower */
             (n[j] == n[i] && j < i)));

  /* Critical section */

  n[i] = 0; /* Discard number */
  /* other code */
}
```

Synchronization

Basic problem boils down making some operation *atomic*.

"After you" relies on writes to the "turn" variable to be atomic (sequentialized).

The Bakery Algorithm relies on atomic reads and writes to elements of the "number" array.

Processors designed for multiprocessing provide atomic instructions:

- test-and-set (e.g., 68000)
- swap (e.g., x86, SPARC)

Implemented with a special, uninterruptable bus cycle.

Test-and-set

Pseudocode for the atomic test-and-set operation:

```
bool test_and_set(bool *flag) {
  bool r = *flag;
  *flag = true;
  return r;
}
```

How to use test-and-set to implement mutual exclusion:

```
bool lock;

for (;;) {
  while (test_and_set(&lock));
  /* Critical section */
  lock = false;
  /* other code */
}
```

Atomic Swap

Pseudocode for atomic swap:

```
void swap(bool *a, bool *b) {
  bool t = *a;
  *a = *b;
  b = t;
}
```

Implementing mutual exclusion:

```
bool lock;

for (;;) {
  bool key = true;
  while (key == true) swap(&lock, &key);
  /* Critical section */
  lock = false;
  /* other code */
}
```

Semaphores

General solution to the shared resource problem

Due to Dijkstra, mid 1960s.

Limit resource usage to "no more than k processes"

(You choose k—Mutual exclusion a special case)

Implementation requires atomic operations

Semaphores

```
int resouces_available = 1;  /* Shared */

void wait(int &s) {
  while (s <= 0);  /* Wait until resource available */
  s--;  /* Claim a resource */
}

void signal(int &s) {
  s++;  /* Relinquish a resource */
}
```

Some authors use Dijkstra's original notation: "wait" was "P" (*proberen*, "to test") and "signal" was "V" (*verhogen*, "to increment").

Using Semaphores

Straightforward:

```
int my_sem = 1;

for (;;) {
  wait(my_sem);

  /* Critical section */

  signal(my_sem);

  /* Other code */
}
```

Implementing Sempahores

Tests and updates of **s** must be atomic.

On single-processor system, disabling interrupts suffices.

```
uni_wait(int &s) {
  disable_int();
  while (s <= 0) {
   enable_int();
   disable_int();
  }
  s--;
  enable_int();
}

uni_signal(int &s) {
  disable_int();
  s++;
  enable_int();
}
```

This trick doesn't work on multiprocessor systems because they run multiple processes simulataneously.

Busy Waiting

Disadvantage of all these schemes is that a process must repeatedly check a condition when blocked:

```
void wait(int &s) {
  while (s <= 0);
  s--;
}
```

```
for (;;) {
  while (test_and_set(&lock));
  lock = false;
}
```

```
for (;;) {
  at_door[0] = true; turn = 1;
  while (at_door[1] && turn == 1);
  at_door[0] = false;
}
```

Busy Waiting

Semaphores that busy wait sometimes called *spinlocks*

Acceptable in multiprocessor systems whose processes have very short critical sections. Few wasted cycles.

Wasteful on a single processor: test repeated when nothing changes.

Alternative is a more event-driven approach:

Put a process to sleep while it waits and wake it only when something changes.

Blocking Semaphores

```
typedef struct {
  int value;    /* # of resources/blocked processes */
  struct process *L; /* Linked list of processes */
} semaphore;

void wait(semaphore S) {
  S.value--;  /* < 0 counts blocked processes */
  if (S.value < 0) {
    add_this_process_to(S.L);/* we're blocked */
    block();                 /* stop process */
  }
}

void signal(semphore S) {
  S.value++;       /* one fewer blocked processes */
  if (S.value <= 0) { /* at least one still blocked */
    struct process P = select(S.L);
    resume(P);  /* unblock chosen process */
  }
}
```

Ways to Misuse Semaphores

```
semphore S, Q;

wait(S);          wait(Q);
wait(Q);          wait(S);
```

Deadlock!

```
signal(S);        signal(Q);
signal(Q);        signal(S);
```

Lesson: Always acquire and release multiple locks in the same order in all processes.

Ways to Misuse Semaphores

Replacing **signal** with **wait**:

```
semaphore S;

wait(S); /* OK */
/* Critical Section */
wait(S); /* Deadlock */
```

Ways to Misuse Semaphores

Reversing calls to **signal** and **wait**:

```
semaphore S;

signal(S); /* Oops: Allows other processes access */
/* Critical Section? (Not mutually exclusive) */
wait(S);
```

Program might appear to work correctly.

Critical Regions

High-level construct to help avoid certain stupid errors. (Hypothetical language).

Declare a shared variable ***v*** of type ***T*** that must be accessed within the body of a **region** statement.

***v*: shared *T*;**

Region statement that blocks while other region is running or the predicate doesn't hold:

region *v* when (*expr*) { *body* }

Bodies of **region** statements guaranteed atomic.

Critical Region Queue Example

```
shared struct buffer {  /* Accessed in a region */
  int pool[N];          /* Buffer data */
  int count, in, out;
};

void insert(int i) {
  region buffer when (buffer.count < N) {
    buffer.pool[buffer.in] = i;
    buffer.in = (buffer.in + 1) % N;
    buffer.count++;
  }
}
int remove() {
  region buffer when (buffer.count > 0) {
    i = buffer.pool[buffer.out];
    buffer.out = (buffer.out + 1) % N;
    buffer.count--;
  }
  return i;
}
```

Monitors: Java's Mechanism

Each Java object has a semaphore-like lock.

The **synchronized** keyword guarantees mutual exclusion:

```
Counter c = new Counter;

synchronized(c) {  /* Acquire c's lock */
  c.count();       /* Critical section */
}                  /* Release c's lock */
```

Shared objects/variables not noted: programmer responsible for ensuring shared objects are synchronized appropriately.

Java Monitors

In an OO language, obvious thing is to lock “this”:

```
class Counter {
  private long cnt; /* Java long R/W not atomic */
  public void count() {
    synchronized (this) {
      cnt++;
    }
  }
}
```

Java provides a frequently-used shorthand:

```
class Counter {
  private long cnt;
  public synchronized void count() {
    cnt++;
  }
}
```