CSC 256/456: Operating Systems

Synchronization Principles I

John Criswell
University of Rochester

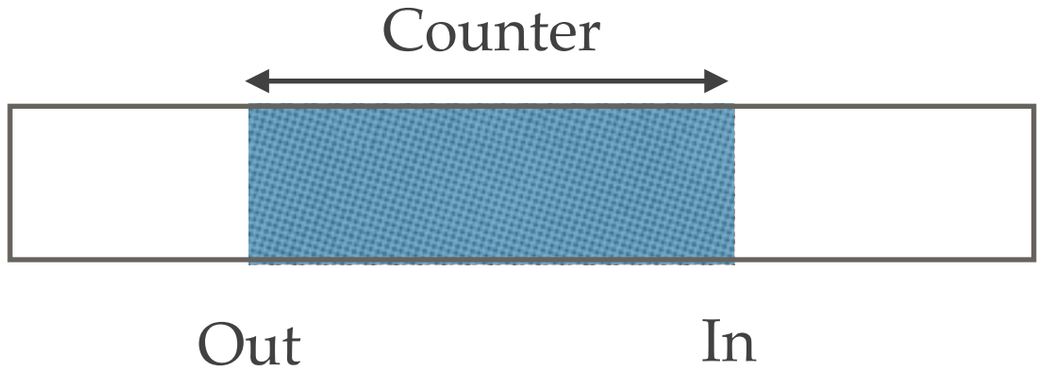
Synchronization Principles

- Background
  - Concurrent access to shared data may result in data inconsistency.
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- The Critical-Section Problem
  - Pure software solution
  - With help from the hardware
- Synchronization without busy waiting (with the support of process/thread scheduler)
  - Semaphore
  - Mutex lock
  - Condition variables

Bounded Buffer

Shared Data

```
typedef struct {…} item;
item buffer[BUFFER_SIZE];
int in=0, out=0;
int counter = 0;
```



Producer process

```
item nextProduced;
while (1) {
    while (counter==BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

Consumer process

```
item nextConsumed;
while (1) {
    while (counter==0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    counter--;
}
```

Bounded Buffer

- Following statements must be performed atomically:
  - counter++;
  - counter--;
- Atomic operation means an operation that completes in its entirety without interruption.

Individual Statements

counter++ sequence

- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

counter— sequence

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;

The Good

counter++

counter—

Counter
4

- register1 = counter;
- register1 = register1 + 1;
- counter   = register1;

- register2 = counter;
- register2 = register2 - 1;
- counter   = register2;

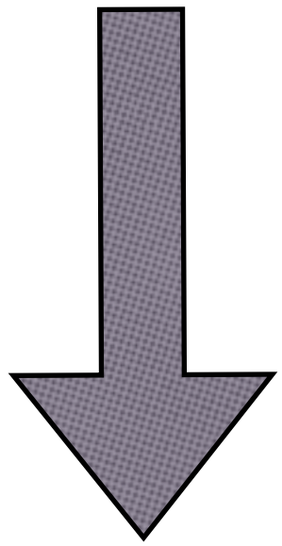
The Good

counter++

counter—

Counter
4

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;



Counter
4

- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

The Other Good

counter++

counter—

Counter
4

- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;

The Other Good

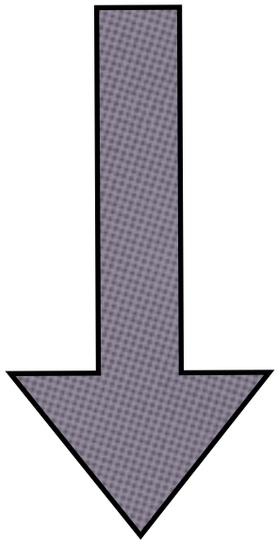
counter++

counter—

Counter
4

- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;



Counter
4

The Ugly

counter++

counter—

Counter
4

- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;

The Ugly

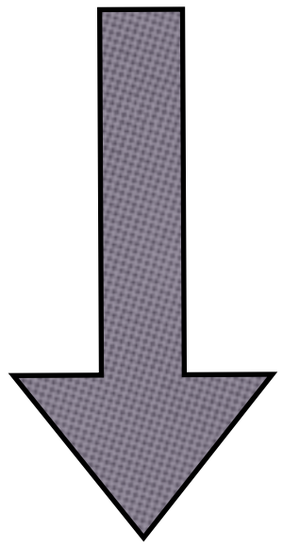
counter++

counter—

Counter
4

- register1 = counter;
- register1 = register1 + 1;
- counter = register1;

- register2 = counter;
- register2 = register2 - 1;
- counter = register2;



Counter
5!

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

- *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.

Requirements for Solutions to the Critical Section Problem

Mutual Exclusion	No two processes are in the critical section at the same time.
Progress	No process outside its critical section can block other processes.
Bounded Waiting	When a process requests to enter its critical section, there is a bound on the time it waits before it is allowed to enter.

Race Condition Solutions

Eliminate Concurrency!

- Disable context switching when in the critical section
  - Only works on a single-processor machines
- Eliminating context switching harder than it looks
  - Software exceptions
  - Hardware interrupts
  - System calls
- Disabling interrupts?
  - Infeasible for user programs since they shouldn't be able to disable interrupts
  - Feasible for OS kernel programs

Critical Section for Two Processes

- Only 2 processes:
  - P0 and P1
- Processes may share some common variables to synchronize their actions.
- Assumption: instructions are atomic and no re-ordering of instructions.
- Critical section does not infinite loop

General Structure of Process

```
do {
    entry_section
        critical_section
    exit_section
        remainder
} while (1);
```

Algorithm 1

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

- Process Pi

```
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 ⇒ Pi ready to enter its critical section

- Process Pi

```
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 Pi

```
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.

Hardware for Synchronization

Need for Synchronization Hardware

- All synchronization systems assume atomic operations
- Hardware can implement atomic operations efficiently
- Modern hardware issues require it
  - Caches and cache coherency
  - Memory consistency models

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 This Solve the Critical Section Problem?

```
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 This Solve the Critical Section Problem?

```
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
```

1.Mutual Exclusion?

Does This Solve the Critical Section Problem?

```
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
```

1. Mutual Exclusion?
2. Progress?

Does This Solve the Critical Section Problem?

```
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
```

1. Mutual Exclusion?
2. Progress?
3. Bounded Waiting?

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
```

Does load-linked/store conditional satisfy the bounded wait requirement?

Why is forgoing the bounded wait requirement okay?

Waiting for Godot…

Question

- In all our solutions, a process enters a loop until entry is granted
- Why is this bad?

Busy Waiting

- In all our solutions, a process enters a loop until 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 to Busy Waiting

- Yield processor
- If you can't avoid busy wait, you must prevent context switch during critical section (disable interrupts while in the kernel).

Recap

- Concurrent access to shared data may result in data inconsistency – race condition.
- The Critical-Section problem
  - Pure software solution
  - With help from the hardware
- Problems with busy-waiting-based synchronization
  - Waste CPU, particularly when context switch occurs while a process is inside critical section
- 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)

Synchronization Mechanisms

Mutex Lock (Binary Semaphore)

- Mutex lock – variable with two states:
  - Locked
  - Unlocked
- Implemented using atomic instructions

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

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

Semaphore

- Synchronization tool that does not require busy waiting
- Integer variable with two atomic operations:
  - wait (or P)
  - signal (or V)

wait(S) or P(S):

```
wait until S > 0;
S—;
```

signal(S) or V(S):

```
S++;
```

Using a Semaphore as a Mutex

```
semaphore mutex = 1;


wait (semaphore);
critical_section;
signal (semaphore);
remainder_section;
```

Semaphore Implementation

- Define a semaphore as a record

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

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

Semaphore operations now defined as (both are atomic):

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

Semaphore Implementation

- Define a semaphore as a record

```
typ
    i
    proc_list *L;
} semaphore;
```

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

How do we make sure wait(S) and signal(S) are atomic?

Semaphore operations now defined as (both are atomic):

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

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);
```

Implement Semaphore Using Mutex Lock

- Data structures:
  - mutex_lock L1, L2;
  - int C;
  - Initialization:
    - L1 = unlocked;
    - L2 = locked;
  - C = initial value of semaphore;

Have we truly eliminated busy waiting?

- 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);
```

Tune in next week for our next exciting episode!

Credits and Disclaimer

- Parts of the lecture slides contain original work from Gary Nutt, Andrew S. Tanenbaum, and Kai Shen. 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).