Operating Systems — Synchronization

INF107

Stefano Zacchiroli

2023

The Critical Section Problem

The Producer-Consumer Concurrency Scenario

- Classic paradigm for cooperating processes:
  - **Producer** process produces information that is consumed by a **consumer** process
  - Intermediate *buffer* used to pass produced stuff from producer to consumer
- Two variations:
  - **Unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is nothing to consume
  - **Bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if buffer is full
    - Consumer waits if there is nothing to consume

Bounder-Buffer — Shared-Memory Implementation

- Let’s provide an implementation of bounded-buffer producer-consumer
- Using **shared-memory** IPC between producer and consumer
  - E.g., two threads in the same process

Shared data:

```c
#define BUFFER_SIZE 10
typedef struct {
    /* ... */
} item;

item buffer[BUFFER_SIZE];  /* circular buffer of items */
int in = 0;                /* location for next produced item */
int out = 0;               /* location for next consumed item */
int counter = 0;           /* number of available items */
```

Bounder-Buffer — Shared-Memory Implementation (cont.)

Producer code:

```
while (true) {
    next_produced = /* produce an item */
    while (counter == BUFFER_SIZE) ; /* buffer is full, wait */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer code:

```
while (true) {  /* consume an item */
    while (counter == 0) ; /* buffer is empty, wait */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in next_consumed */
}
```

Bounder-Buffer — Shared-Memory Implementation (cont.)

Producer code:

```
while (true) {
    next_produced = /* produce an item */
    while (counter == BUFFER_SIZE) ; /* buffer is full, wait */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer code:

```
while (true) {  /* consume an item */
    while (counter == 0) ; /* buffer is empty, wait */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in next_consumed */
}
```

Q: is this implementation correct?

Race Condition — Example

1. `counter++` could be implemented in hardware as:
   - `register1 = counter`
   - `register1 = register1 + 1`
   - `counter = register1`
2. `counter--` as:
   - `register2 = counter`
   - `register2 = register2 - 1`
   - `counter = register2`

- Consider the following execution interleaving of instructions (1) and (2) with “count = 5” initially:
  - S0: producer execute register1 = counter `{register1 = 5}`
  - S1: producer execute register1 = register1 + 1 `{register1 = 6}`
  - S2: consumer execute register2 = counter `{register2 = 5}`
  - S3: consumer execute register2 = register2 – 1 `{register2 = 4}`
  - S4: producer execute counter = register1 `{counter = 6}`
  - S5: consumer execute counter = register2 `{counter = 4}`

Race Condition — Definition

- Processes can execute concurrently
- May be interrupted at any time, *partially* completing execution
- Concurrent access to shared data may result in data inconsistency. More formally:

Definition (Race Condition)

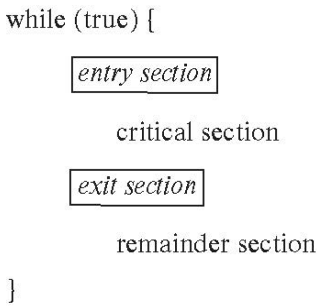
A **race condition** is a situation where several processes access and manipulate the same data concurrently and *the outcome of the execution depends on the particular order* in which the access takes place.

- Corollary: the *correctness* of a program affected by race conditions depend on the execution order of its concurrent parts.
  - Order which is, in general, unpredictable : - (
- Maintaining data consistency requires **mechanisms to ensure the orderly execution of cooperating processes**

We will study some of these mechanisms in this lecture.

The Critical Section Problem

- Consider system with n processes $\{P_0, P_1, \dots, P_{n-1}\}$
- Each process has a **critical section** (CS) segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - Desired property: *when one process in its critical section, no other process may be in its own*
- The **critical section problem** is to design a protocol that enforces this property
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**
- If we can solve this, we can avoid all race conditions
  - Iff we box all concurrent accesses to shared data into critical sections!



```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

Figure: general structure of process P_i

The Critical Section Problem (cont.)

Requirements for acceptable solutions to the critical section problem:

1. **Mutual Exclusion**: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress**: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter next is taken only by processes not executing in their remainder section and cannot be postponed indefinitely
3. **Bounded Waiting**: 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

Assumptions:

- Each process executes at a *nonzero speed*
- No assumption concerning *relative speed* of the processes

Software Solutions

Software Solutions

- A **software solution** to the critical section problem is one that requires no special support from the operating system or the hardware to work.

Software Solution 1:

- Two-process solution: P_0, P_1.
  - Notation: we will refer to P_i as *the other* process w.r.t. P_j and vice-versa
- Assumption: `load` **and** `store` machine instructions are **atomic**; i.e., they cannot be interrupted
- The two processes **share one variable**: `int turn;`
- The variable `turn` indicates whose turn it is to enter the critical section next
- Initially, `turn = i` (for some i, doesn't matter which)

Software Solution 1 (cont.)

Algorithm for process P_i:

```
while (true) {
    while (turn == j) ;

      /* critical section */

    turn = j;

      /* remainder section */
}
```

Software Solution 1 (cont.)

Algorithm for process P_i:

```
while (true) {
    while (turn == j) ;

      /* critical section */

    turn = j;

      /* remainder section */
}
```

Q: is this correct?

Software Solution 1 (cont.)

Algorithm for process P_i:

```
while (true) {
    while (turn == j) ;

      /* critical section */

    turn = j;

      /* remainder section */
}
```

Q: is this correct?

- Mutual exclusion is preserved
  - P_i enters critical section only if:
    - `turn = i`, and
    - and `turn` cannot be both 0 and 1 at the same time
- But. What about the Progress requirement?
- What about the Bounded-waiting requirement?

Peterson's Solution

- Two-process solution: P_0, P_1.
- Assumption: `load` **and** `store` machine instructions are **atomic**; i.e., they cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `boolean flag[2];`
- The variable `turn` indicates whose turn it is to enter the critical section next
- The `flag` array is used to indicate if a process is ready to enter the critical section.
  - `flag[i] = true` → process Pi is ready to enter

Peterson’s Solution (cont.)

Algorithm for process P_i:

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j) ;

      /* critical section */

    flag[i] = false;

      /* remainder section */
}
```

Intuition:

- A process can “book” its CS access using `flag` and can only enter when booked and it is its turn.
- Before entering, a process sets the turn to the *other* process, so if the other process have been waiting it will always get a change to enter before one that has just left the CS.

Correctness of Peterson's Solution

It is provable (cf. OS Book, Chapter 6) that the three CS requirement are met:

1. Mutual exclusion is preserved
    - P_i enters CS only if: either `flag[j] = false` or `turn = i`
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Peterson's Solution vs Modern Hardware

- Problem: Peterson's Solution is not guaranteed to work on modern hardware architectures.
- To improve performance, processors and compilers may **reorder instructions** that have no dependencies.

Example

- Two threads share the data: `boolean flag = false; int x = 0;`
- Thread 1: `while (!flag) ; print x;`
- Thread 2: `x = 100; flag = true;`
- What is the expected output? 100

Peterson’s Solution vs Modern Hardware

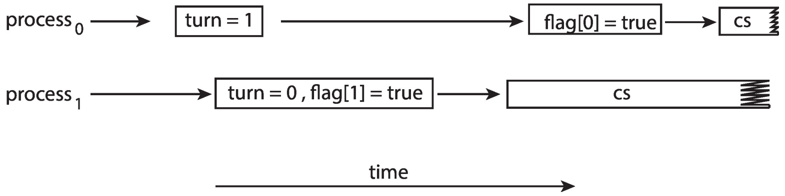
- Problem: Peterson’s Solution is not guaranteed to work on modern hardware architectures.
- To improve performance, processors and compilers may **reorder instructions** that have no dependencies.

Example

- Two threads share the data: `boolean flag = false; int x = 0;`
- Thread 1: `while (!flag) ; print x;`
- Thread 2: `x = 100; flag = true;`
- What is the expected output? 100
- However, since `flag` and `x` are independent of each other, thread 2 instructions can be reordered to become: `flag = true; x = 100;` (!)
- If this occurs, the output may be 0 (!!)

Peterson’s Solution vs Modern Hardware (cont.)

- The effect of instruction reordering on Peterson’s Solution can be disastrous:[1]

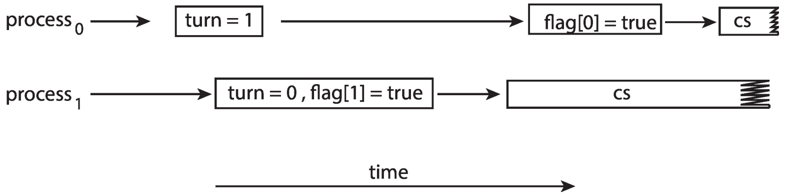


- This allows both processes to be in their critical section at the same time!

[1] Reminder of Peterson’s for P_1: `flag[1] = true; turn = 0; while(flag[0] && turn == 0);`

Peterson’s Solution vs Modern Hardware (cont.)

- The effect of instruction reordering on Peterson’s Solution can be disastrous:[1]



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson’s solution works correctly on modern hardware we must use memory barrier instructions (e.g., `"x = 100; memory_barrier(); flag = true;"`)
  - A **memory barrier** is an instruction that forces any change in memory to be propagated (“flushed”, made visible) to all other processors
  - When a memory barrier is performed, the system ensures that all loads/stores are completed before any subsequent load/store operations are performed (even in presence of reordering)

[1] Reminder of Peterson’s for P_1: `flag[1] = true; turn = 0; while(flag[0] && turn == 0);`

Atomic Instructions

Hardware Support for Synchronization

- Modern hardware provides support for synchronization
  - With it, we can do better than with pure-software solutions to the critical section problem
- **Special hardware instructions** that allow to either test-and-modify the content of a memory word, or to swap the contents of two words **atomically** (= cannot be interrupted during execution)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction

The test_and_set Instruction

- Behavior equivalent to:

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

- Properties:
  - **Executed atomically**
  - Return the original value referenced by target
  - Set the new value of target reference to true

Solution Using `test_and_set`

- Shared boolean variable `lock`, initialized to `false`
- Solution:

```
do {
    while (test_and_set(&lock)) ;

        /* critical section */

    lock = false;

        /* remainder section */
} while (true);
```

- Does this solve the critical section problem?

Solution Using `test_and_set`

- Shared boolean variable `lock`, initialized to `false`
- Solution:

```
do {
    while (test_and_set(&lock)) ;

        /* critical section */

    lock = false;

        /* remainder section */
} while (true);
```

- Does this solve the critical section problem?
- Not completely.
  - It implements (1) mutual exclusion, and (2) progress, but not (3) bounded waiting.
  - One fast process can make an unlucky slow process wait indefinitely and never enter its own CS.

The compare_and_swap Instruction

- Definition:

```c
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected) {
        *value = new_value;
    }
    return temp;
}
```

- Properties:
  - **Executed atomically**
  - Return the original value referenced by `value`
  - Set the new value of `target` reference to `new_value` only if previous value was `expected`
    - That is, the swap only takes place *conditionally*

Solution Using `compare_and_swap`

- Shared integer `lock`, initialized to 0
- Solution:

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0) ;

      /* critical section */

    lock = 0;

      /* remainder section */
}
```

- Does this solve the critical section problem?

Solution Using `compare_and_swap`

- Shared integer `lock`, initialized to 0
- Solution:

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0) ;

      /* critical section */

    lock = 0;

      /* remainder section */
}
```

- Does this solve the critical section problem?
- Not completely.
- As before it provides mutual exclusion without satisfying bounded waiting.

Full Solution Using `compare_and_swap`

```c
while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;

      /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;

     /* remainder section */
}
```

`compare_and_swap` on Modern CPUs

- On Intel x86 architectures, the assembly language statement `cmpxchg` is used to implement `compare_and_swap`.
- The `lock` prefix is used to lock the bus while the destination operand is being updated.
- The general form of this instruction hence appears as:

```
lock cmpxchg <destination operand>, <source operand>
```

Atomic Variables

- Typically, instructions such as compare_and_swap are used as **building blocks for other synchronization tools**.
- One tool is an **atomic variable** that provides atomic (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let sequence be an atomic variable of type integer
  - Let increment() be an operation available on the atomic variable sequence
  - The command: increment(&sequence); ensures sequence is incremented without interruption
- increment() can be implemented on top of compare_and_swap like this:

```
void increment(atomic_int *v) {
    int temp;
    do {
        temp = *v;
    } while (temp != (compare_and_swap(v, temp, temp + 1));
}
```

- Modern programming languages and compilers (e.g., C++20, Rust) provide atomic variables.

Mutex Locks

Mutex Locks

- Previous solutions are complicated and generally inaccessible to high-level applications programmers
- OS designers build easier-to-use software tools to solve the CS problem
- Simplest is **mutex lock** (“mutex” for “MUTual EXclusion”)
  - Boolean variable indicating if lock is available or not
- Protect a critical section by:
  - First `acquire()` a lock
  - Then `release()` the lock
- Using a mutex lock (application point of view):

```
while (true) {
    acquire();

        /* critical section */

    release();

        /* remainder section */
}
```

Mutex Locks (cont.)

- Implementation of `acquire()` and `release()` (OS point of view):

```
acquire() {
    while (!available) ; /* busy wait */
    available = false;
}
```

```
release() {
    available = true;
}
```

- The OS must guarantee that `acquire()` and `release()` are executed atomically
  - Can do so using hardware support like `compare_and_swap`
- Problem: **busy wait**, upon `acquire()` we constantly check if the lock has become available
  - Waste of resources: CPU, battery
  - Also called **spinlock** as the CPU keeps “spinning” waiting for the lock

Semaphores

Semaphores

- Synchronization tool—introduced by Edsger Dijkstra in 1962—that provides more sophisticated (e.g., than mutex locks) ways for processes to synchronize their activities.
- A **semaphore** S is an integer variable
- Can only be accessed via two **atomic operations**
  - `wait(S)` — originally and often still called `P(S)` from Dutch *proberen*, “to test”
  - `signal(S)` — or `V(S)` from *verhogen*, “to increment”
- Definitions:

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

- Two variants:
  - **Binary semaphore**: integer value can range only between 0 and 1
    - Intuition: analogous to a mutex lock
  - **Counting semaphore**: integer value can range over an unrestricted domain
    - Intuition: count the number of resources available from a finite pool

Semaphore Usage

- With semaphores we can solve various synchronization problems
- Example (1): **solution to the CS problem**
  - Semaphore called `mutex` initialized to 1 (intuitive meaning: CS available)
  - Each process:

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

Semaphore Usage

- With semaphores we can solve various synchronization problems
- Example (1): **solution to the CS problem**
  - Semaphore called `mutex` initialized to 1 (intuitive meaning: CS available)
  - Each process:

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

- Example (2): two processes P_1 and P_2 want to **ensure that action A_1 by P_1 happens before action A_2 by P_2**
  - Semaphore `sync` initialized to 0 (intuitive meaning: "A_1 has not happened yet")
  - Processes:

```
P1() {
    A1;
    signal(sync);
}
```

```
P2() {
    wait(sync);
    A2;
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute the `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
  - We know how to address this, but it would be nice to do so without **busy waiting**

Semaphore Implementation with no Busy Waiting

- **Each semaphore** is associated to a **waiting queue**
  - In the kernel, known to the scheduler
- Two basic operations:
  - **Block**: place the process invoking the operation on the appropriate waiting queue
    - It will stop executing → no busy wait
  - **Wakeup**: remove one of processes in the waiting queue (usually, but not necessarily, in FIFO order) and place it in the ready queue

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

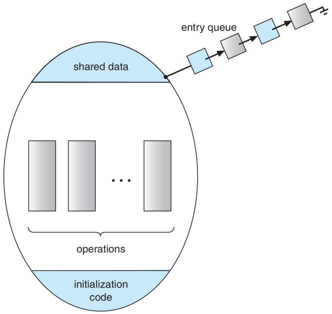
Monitors

Problems with Semaphores

- While the atomicity of operations is guaranteed, programmers can still (and do) shoot themselves in the foot with semaphores
  - E.g., inverse `wait()` … `signal()` around a CS → race condition
  - E.g., `wait()` one time too many → process blocks forever
- These—and others—are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- **Abstract Data Type**, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- **Avoid by construction** certain classes of misuse of synchronization tools



- Monitor components:
  - Shared data
  - Operations (= “methods”) that processes can invoke
  - Initialization code, executed upon monitor creation
  - Waiting queue of processes willing to execute monitor code (when another process is executing monitor code)
- Implementation
  - One semaphore `mutex` per monitor instance
  - Wrap the body of each operation P in a `wait(mutex)` … `signal(mutex)` critical section

Java Monitors

- *Monitor* is an **abstract notion**, which can be implemented in a high-level programming languages and provided as a concrete **programming tool**
- For example, it is implemented in Java using classes and the `synchronized` keyword

```java
public class BoundedBuffer<E> {
    // Shared data:
    private static final int BUFFER SIZE = 5;
    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
        // Initialization code
    }
    public synchronized void insert(E item) {
        // Java runtime guarantees that only one thread at a time will execute *any* synchronized
        // method on a given class *instance*. (Others will wait for completion.)
    }
    public synchronized E remove() {
        // ...
    }
}
```

Liveness

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a more general set of properties that *a system* must satisfy to ensure making progress.
    - Indefinite waiting is an example of a liveness failure.
    - Other common cases/patterns exist.

Deadlock

Definition (Deadlock)

Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Example: Let S and Q be two semaphores initialized to 1

```
P0() {
    wait(S);
    wait(Q);
    /* ... */
    signal(S);
    signal(Q);
}
```

```
P1() {
    wait(Q);
    wait(S);
    /* ... */
    signal(Q);
    signal(S);
}
```

- Consider if P_0 executes `wait(S)` and P_1 `wait(Q)`.
- When P_0 executes `wait(Q)`, it must wait until P_1 executes `signal(Q)`.
- However, P_1 is waiting until P_0 execute `signal(S)`.
- Since these `signal()` operations will never be executed, P_0 and P_1 are **deadlocked**.

Deadlock Characterization

More generally, deadlock can arise if four conditions hold simultaneously:

1. **Mutual exclusion:** only one thread at a time can use a resource
2. **Hold and wait:** a thread holding at least one resource is waiting to acquire additional resources held by other threads
3. **No preemption:** a resource can be released only voluntarily by the thread holding it, after that thread has completed its task
4. **Circular wait:** there exists a set $\{T_0, T_1, \ldots, T_{n-1}\}$ of waiting threads such that $\forall i, 0 \leq i \leq n-1, T_i$ is waiting for a resource that is held by $T_{(i+1) \bmod n}$

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - **Deadlock prevention** (*static*)
    - Invalidate one of the four necessary conditions for a deadlock
    - E.g., invalidate "hold and wait" → Require each thread to request and be allocated all resources before execution begins; or allow a thread to request resources only if it has none allocated
    - E.g., invalidate "circular wait" → Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration

- **Deadlock avoidance**
  - Requires that the system has some additional *a priori* information available, e.g., the maximum number of resources of each type that a thread may need
  - The deadlock-avoidance algorithm *dynamically* examines the resource-allocation state to ensure that there can never be a circular-wait condition

Methods for Handling Deadlocks (cont.)

- Allow the system to enter a deadlock state and then **recover**
  - Dynamically detect that a deadlock has occurred
    - Maintain a graph of available resources as well as who wants/waits for them
    - Detect a circular wait loop
  - Recovery
    - Process termination: kill one of the threads involved
    - Resource preemption: take back an allocated resource (“rollback”)

- **Ignore the problem** and pretend that deadlocks never occur in the system
  - Most OS do this, at least in the general case.
  - They can still detect limited forms of deadlocks around individual resources and prevent them by refusing to give a resource to the last requesting process that will induce a deadlock.

Liveness (cont.)

Other forms of liveness violations:

- **Starvation:** indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority inversion:** Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - 3 processes with priorities $L < M < H$
  - L holds a lock that H wants
  - M preempts L
  - M ($< H$), who is not involved in the lock at all, impacts how much H has to wait to obtain the lock (which should not happen with priority-based scheduling)
  - Solution: **priority-inheritance protocol**: L temporarily inherits H priority until lock release

Reading List

You should study on books, not slides! Reading material for this lecture is:

- Silberschatz, Galvin, Gagne. Operating System Concepts, Tenth Edition:
  - Chapter 6: Synchronization Tools
  - Chapter 7: Synchronization Examples
  - Chapter 8: Deadlocks

Credits:

- Some of the material in these slides is reused (with modifications) from the official slides of the book Operating System Concepts, Tenth Edition, as permitted by their copyright note.