Synchronization Tools
Synchronization Tools

• Background
• The Critical-Section Problem
• Peterson’s Solution
• Synchronization Hardware
• Mutex Locks
• Semaphores
• Monitors
Objectives

• To present the concept of process synchronization.
• To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
• To present both software and hardware solutions of the critical-section problem.
• To examine several classical process-synchronization problems.
• To explore several tools that are used to solve process synchronization problems.
Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
Systems = Objects + Activities

- **Safety** is a property of objects, and groups of objects, that participate across multiple activities.
  - Can be a concern at many different levels: objects, composites, components, subsystems, hosts, ...

- **Liveness** is a property of activities, and groups of activities, that span across multiple objects.
  - Levels: Messages, call chains, threads, sessions, scenarios, scripts workflows, use cases, transactions, data flows, mobile computations, ...
Violating Safety

• Data can be shared by threads
  • Scheduler can interleave or overlap threads arbitrarily
  • Can lead to interference
    • Storage corruption (e.g. a data race/race condition)
    • Violation of representation invariant
    • Violation of a protocol (e.g. A occurs before B)
How does this apply to OSs?

• Any resource that is shared could be accessed inappropriately
  • Shared memory
    • Kernel threads
    • Processes (shared memory set up by kernel)
  • Shared resources
    • Printer, Video screen, Network card, ...

• OS must protect shared resources
  • And provide processes a means to protect their own abstractions
Illustration of the problem:

• Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE) ;
    /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;

}
Consumer

    while (true) {
        while (counter == 0)
            ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
    }
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Start: both threads ready to run. Each will increment the global count.

Shared state  cnt = 0
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state  \( \text{cnt} = 0 \)

\( \text{y} = 0 \)

\( T1 \) executes, grabbing the global counter value into \( y \).
Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

**Shared state**

\[ \text{cnt} = 0 \]

\[ \text{y} = 0 \]

\[ \text{y} = 0 \]

*T1 is pre-empted.*  
*T2 executes, grabbing the global counter value into y.*
Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

**Shared state** \( cnt = 1 \)

T2 executes, storing the incremented \( cnt \) value.
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

*Shared state*

```java
cnt = 1
```

```java
y = 0
```

```
y = 0
```

T2 completes. T1 executes again, storing the old counter value (1) rather than the new one (2)!
But When I Run it Again?
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Start: both threads ready to run. Each will increment the global count.

Copyright 2018 Silberschatz, Gavin & Gagne
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

**Shared state**  
\[\text{cnt} = 0\]

**y = 0**

*T1 executes, grabbing the global counter value into y.*
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state  $\text{cnt} = 1$

$\text{y} = 0$

$T1$ executes again, storing the counter value
**Data Race Example**

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

**Shared state**  
\[ \text{cnt} = 1 \]

**T1 finishes. T2 executes, grabbing the global counter value into y.**  
\[ \text{cnt} = 1 \]
\[ y = 0 \]
\[ y = 1 \]
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state  \(\text{cnt} = 2\)

\(y = 0\)

\(y = 1\)

T2 executes, storing the incremented \(\text{cnt}\) value.
What happened?

• In the first example, \texttt{t1} was preempted after it read the counter but before it stored the new value.
  • Depends on the idea of an \textit{atomic action}
  • Violated an object invariant

• A particular way in which the execution of two threads is interleaved is called a \textit{schedule}. We want to prevent this undesirable schedule.

• Undesirable schedules can be hard to reproduce, and so hard to debug.
Race Condition

• \texttt{counter++} could be implemented as

\begin{verbatim}
register1 = counter
register1 = register1 + 1
counter = register1
\end{verbatim}

• \texttt{counter--} could be implemented as

\begin{verbatim}
register2 = counter
register2 = register2 - 1
counter = register2
\end{verbatim}

• Consider this execution interleaving with “count = 5” initially:

\begin{verbatim}
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}
\end{verbatim}
Question

• If you run a program with a race condition, will you always get an unexpected result?
  • No! It depends on the scheduler
  • ...and on the other threads/processes/etc that are running on the same CPU

• Race conditions are hard to find
Disabling Interrupts

• Doesn’t work for multiprocessors
• Doesn’t permit different groups of critical sections
Synchronization

```c
static int cnt = 0;
struct Mutex lock;
Mutex_Init(&lock);

void run() {
    Mutex_Lock (&lock);
    int y = cnt;
    cnt = y + 1;
    Mutex_Unlock (&lock);
}
```

**Lock**, for protecting

**The shared state**

**Acquires the lock**;

**Only succeeds if not held by another thread**

**Releases the lock**
Java-style synchronized block

```java
static int cnt = 0;
struct Mutex lock;
Mutex_Init(&lock);
void run() {
    synchronized (lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

**Lock, for protecting**
**The shared state**

**Acquires the lock;**
**Only succeeds if not held by another thread**

**Releases the lock**
Applying synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

*Shared state*  
cnt = 0

*T1 acquires the lock*
Applying synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

**Shared state**  
\[ cnt = 0 \]

\[ y = 0 \]

*T1 reads cnt into y*
Applying synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state  \( \text{cnt} = 0 \)  

\( y = 0 \)

\[ \]

**T1 is pre-empted.**

**T2 attempts to acquire the lock but fails because it’s held by T1, so it blocks**

Copyright 2018 Silberschatz, Gavin & Gagne
Applying synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

**Shared state**  \( \text{cnt} = 1 \)

\( y = 0 \)

**T1 runs, assigning to** \( \text{cnt} \)
Applying synchronization

```
int cnt = 0;
it1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
it2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state  

\[
\begin{align*}
\text{cnt} &= 1 \\
y &= 0
\end{align*}
\]

T1 releases the lock  
and terminates
Applying synchronization

int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}

**Shared state**  
\[ cnt = 1 \]

\[ y = 0 \]

*T2 now can acquire the lock.*
Applying synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

**Shared state**

```
cnt = 1
```

```
y = 0
```

```
y = 1
```

*T2 reads cnt into y.*
Applying synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

**Shared state**  \( \text{cnt} = 2 \)

- \( y = 0 \)
- \( y = 1 \)

*\( T2 \) assigns \( \text{cnt} \),
then releases the lock*
Mutexes (locks)

- Only one thread can “acquire” a mutex
  - Other threads block until they can acquire it
  - Used for implementing critical sections
- A critical section is a piece of code that should not be interleaved with code from another thread
  - Executed atomically
- We’ll look at other ways to implement critical sections later …
Mutex Policies

• What if a thread already holds the mutex it’s trying to acquire?
  • Re-entrant mutexes: The thread can reacquire the same lock many times. Lock is released when object unlocked the corresponding number of times
    • This is the case for Java
    • Non-reentrant: Deadlock! (defined soon.)
      • This is the case in GeekOS
  • What happens if a thread is killed while holding a mutex? Or if it just forgets to release it
    • Could lead to deadlock
Java Synchronized statement

• `synchronized (obj) { statements }`
  
• Obtains the lock on `obj` before executing statements in block
  • `obj` can be any Object

• Releases the lock when the statement block completes
  • Either normally, or due to a return, break, or exception being thrown in the block

• Can’t forget to release the lock!
Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called *deadlock*

```java
Object A = new Object();
Object B = new Object();
T1.run() {
    synchronized (A) {
        synchronized (B) {
            ...
        }
    }
}

T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
```
Deadlock

• Quite possible to create code that deadlocks
  • Thread 1 holds lock on A
  • Thread 2 holds lock on B
  • Thread 1 is trying to acquire a lock on B
  • Thread 2 is trying to acquire a lock on A
  • Deadlock!

• Not easy to detect when deadlock has occurred
  • other than by the fact that nothing is happening
Deadlock: Wait graphs

Thread T1 holds lock A

Thread T2 attempting to acquire lock B

Deadlock occurs when there is a cycle in the graph
Wait graph example

T1 holds lock on A
T2 holds lock on B
T1 is trying to acquire a lock on B
T2 is trying to acquire a lock on A
Critical Section Problem

- Consider system of $n$ processes \{\(p_0, p_1, \ldots, p_{n-1}\)\}
- Each process has \textit{critical section} segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- \textit{Critical section problem} is to design protocol to solve this
- Each process must ask permission to enter critical section in \textit{entry section}, may follow critical section with \textit{exit section}, then \textit{remainder section}
Critical Section

• General structure of process $P_i$

```
    do {
        entry section
        critical section
        exit section
        remainder section
    } while (true);
```
Solution to 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 processes that will enter the critical section next 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.
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the $N$ processes.
Two-task Solution

• Two tasks, $T_0$ and $T_1$ ($T_i$ and $T_j$)
• Three solutions presented.
Algorithm 1

• Threads share a common integer variable turn
  • Turn takes values 0 and 1
• Initialize turn to 0
• Entry Section for thread i
  • If turn==i, thread i is allowed to proceed, else yield
• Exit Section for thread i
  • turn==(1-i)
Algorithm 1

• Satisfies mutual exclusion but not progress.
  • Processes are forced to enter their critical sections alternately.
  • One process not in its critical section thus prevents the other from entering its critical section.
Algorithm 2

• Boolean flags to indicate thread’s interest in entering critical section

• Entry Code
  if (t == 0) {
    flag0 = true;
    while(flag1 == true)
      Thread.yield();
  }
  else {
    flag1 = true;
    while (flag0 == true)
      Thread.yield();
  }

• Exit Code
  if (t == 0)
    flag0 = false;
  else
    flag1 = false;

• Initialize
  • Both flags to false
Algorithm 2

• Satisfies mutual exclusion, but not progress requirement.
  • Both processes can end up setting their flag[] variable to true, and thus neither process enters its critical section!
Algorithm 3 Peterson’s Solution

• Combine ideas from 1 and 2
Peterson’s Solution

• Good algorithmic description of solving the problem
• Two process solution
• Assume that the load and store machine-language instructions are atomic; that is, 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
• The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process $P_i$ is ready!
Algorithm for Process \( P_i \)

do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;
        remainder section
} while (true);
Peterson’s Solution (Cont.)

• Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved
     \( P_i \) enters CS only if:
     
     either \( \text{flag}[j] = \text{false} \) or \( \text{turn} = i \)
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met
Algorithm 3

• Meets all three requirements; solves the critical-section problem for two processes.
  • One process is always guaranteed to get into its critical section.
  • Processes are forced to take turns when they both want to get in.
Bakery Algorithm

Critical section for n processes

• Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.

• If processes $P_i$ and $P_j$ receive the same number, if $i < j$, then $P_i$ is served first; else $P_j$ is served first.

• The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
Bakery Algorithm

• Notation $\leq$ lexicographical order (ticket #, process id #)
  • $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$
  • max $(a_0, ..., a_{n-1})$ is a number, $k$, such that $k \geq a_i$ for $i = 0, ..., n - 1$

• Shared data
  
  boolean choosing[n];
  int number[n];

  Data structures are initialized to false and 0 respectively
Bakery Algorithm

do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], ..., number [n – 1])+1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]) ;
        while ((number[j] != 0) && (number[j,j] < number[i,i])) ;
    }
    critical section
    number[i] = 0;
    remainder section
} while (1);
Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
Synchronization Hardware

• Many systems provide hardware support for implementing the critical section code.
• All solutions below based on idea of locking
  • Protecting critical regions via locks
• Uniprocessors – could disable interrupts
  • Currently running code would execute without preemption
  • Generally too inefficient on multiprocessor systems
    • Operating systems using this not broadly scalable
• Modern machines provide special atomic hardware instructions
  • Atomic = non-interruptible
  • Either test memory word and set value
  • Or swap contents of two memory words
Solution to Critical-section Problem Using Locks

do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE".
Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:
  
  ```
  do {
      while (test_and_set(&lock))
      ; /* do nothing */
      /* critical section */
      lock = false;
      /* remainder section */
  } while (true);
  ```

Copyright 2018 Silberschatz, Gavin & Gagne
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;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” == “expected”. That is, the swap takes place only under this condition.
Solution using compare_and_swap

• Shared integer “lock” initialized to 0;
• Solution:

```c
    do {
        while (compare_and_swap(&lock, 0, 1) != 0)
            ; /* do nothing */
        /* critical section */
        lock = 0;
        /* remainder section */
    } while (true);
```
Bounded-waiting Mutual Exclusion with test_and_set

do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);