CENG 2034 - Operating Systems

Week 9: Synchronization Tools

Burak Ekici

May 25, 2023

Outline

Background

- Processes can execute concurrently

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution

Background

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

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

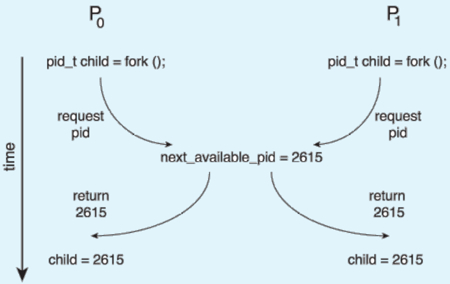
Outline

Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call

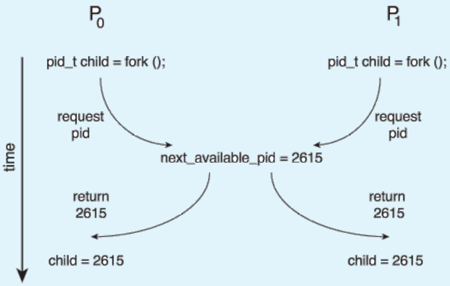
Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (`pid`)



Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (`pid`)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same `pid` could be assigned to two different processes!

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, ..., p_{n-1}\}$

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has critical section segment of code

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots, p_{n-1}\}$
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc.

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots, p_{n-1}\}$
- Each process has 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

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, ..., p_{n-1}\}$
- Each process has 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
- Critical section problem is to design protocol to solve this

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots, p_{n-1}\}$
- Each process has 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
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section

Critical Section

General structure of process P_i

```
while (true)
{
    enrty section

    /* critical section */

    exit section

    /* reminder section */
}
```

Critical Section Problem

Requirements for solution to critical-section problem

Critical Section Problem

Requirements for 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

Critical Section Problem

Requirements for 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 process that will enter the critical section next cannot be postponed indefinitely

Critical Section Problem

Requirements for 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 process 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

Critical Section Problem

Requirements for 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 process 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

Critical Section Problem

Requirements for 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 process 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

Interrupt-based Solution

- Entry section: disable interrupts

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?

Software Solution 1

- Two process solution

Software Solution 1

- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted

Software Solution 1

- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:

```
int turn
```

Software Solution 1

- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:

```
int turn
```

- The variable turn indicates whose turn it is to enter the critical section

Software Solution 1

- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:

  `int turn`
- The variable turn indicates whose turn it is to enter the critical section
- initially, the value of `turn` is set to `i`

Algorithm for Process P_i

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

  /* critical section */

  turn = j

  /* reminder section */
}
```

Correctness of the Software Solution

- Mutual exclusion is preserved

$\boxed{P_i \text{ enters critical section } \texttt{turn = i} \text{ and } \texttt{turn} \text{ cannot be both } \texttt{i} \text{ and } \texttt{j} \text{ at the same time}}$

Correctness of the Software Solution

- Mutual exclusion is preserved

$\boxed{P_i \text{ enters critical section } \texttt{turn = i} \text{ and } \texttt{turn} \text{ cannot be both } \texttt{i} \text{ and } \texttt{j} \text{ at the same time}}$

- What about the Progress requirement?

Correctness of the Software Solution

- Mutual exclusion is preserved

  P_i enters critical section `turn = i` and `turn` cannot be both `i` and `j` at the same time

- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Outline

Peterson's Solution

- 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

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

  /* critical section */

  flag[i] = false

  /* reminder section */
}
```

Correctness of Peterson’s Solution

Provable that the three CS requirement are met:

Correctness of Peterson's Solution

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

$$\boxed{P_i \text{ enters CS only if either } \texttt{flag[j] = false} \text{ or } \texttt{turn = i}}$$

Correctness of Peterson's Solution

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   $\boxed{P_i \text{ enters CS only if either } \texttt{flag[j] = false} \text{ or } \texttt{turn = i}}$

2. Progress requirement is satisfied

Correctness of Peterson's Solution

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   $\boxed{P_i \text{ enters CS only if either } \texttt{flag[j] = false} \text{ or } \texttt{turn = i}}$

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures

Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures
  - To improve performance, processors and/or compilers may reorder operations with no dependencies

Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures
  - To improve performance, processors and/or compilers may reorder operations with no dependencies
- Understanding why it will not work is useful for better understanding race conditions

Peterson’s Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson’s Solution is not guaranteed to work on modern architectures
  - To improve performance, processors and/or compilers may reorder operations with no dependencies
- Understanding why it will not work is useful for better understanding race conditions
- For single-threaded this is fine as the result will always be the same

Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures
  - To improve performance, processors and/or compilers may reorder operations with no dependencies
- Understanding why it will not work is useful for better understanding race conditions
- For single-threaded this is fine as the result will always be the same
- For multithreaded the reordering may produce inconsistent or unexpected results

Example (Modern Architecture)

- Two threads share the data:

```
bool flag = false;
int x = 0;
```

Example (Modern Architecture)

- Two threads share the data:

```
bool flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag);
print x;
```

Example (Modern Architecture)

- Two threads share the data:

```
bool flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag);
print x;
```

- Thread 2 performs

```
x = 100;
flag = true;
```

Example (Modern Architecture)

- Two threads share the data:

```
bool flag = false;
int x = 0;
```

- Thread 1 performs

```
while (!flag);
print x;
```

- Thread 2 performs

```
x = 100;
flag = true;
```

- Q: What is the expected output?
  A: 100

Example (Modern Architecture (cont'd))

- However, since the variables flag and x are independent of each other, the instructions:

```
flag = true;
x = 100;
```

  for Tread 2 may be reordered

Example (Modern Architecture (cont'd))

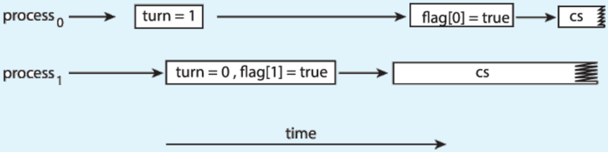
- However, since the variables flag and x are independent of each other, the instructions:

```
flag = true;
x = 100;
```

  for Tread 2 may be reordered
- If this occurs, the output may be 0

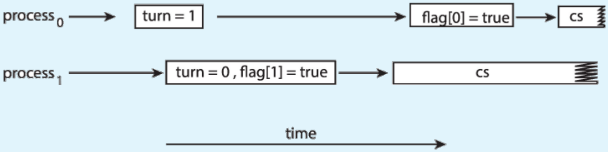
Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



Peterson's Solution Revisited

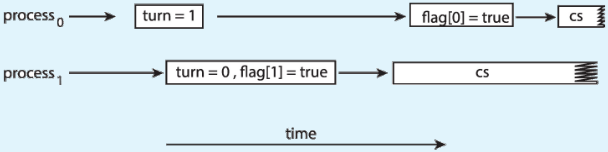
- The effects of instruction reordering in Peterson's Solution



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

Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use Memory Barrier

Outline

Memory Barrier

- Memory model – memory guarantees by computer architecture

Memory Barrier

- Memory model – memory guarantees by computer architecture
- Memory models may be either:

Memory Barrier

- Memory model – memory guarantees by computer architecture
- Memory models may be either:
    - Strongly ordered – where a memory modification of one processor is immediately visible to all other processors

Memory Barrier

- Memory model – memory guarantees by computer architecture
- Memory models may be either:
  - Strongly ordered – where a memory modification of one processor is immediately visible to all other processors
  - Weakly ordered – where a memory modification of one processor may not be immediately visible to all other processors

Memory Barrier

- Memory model – memory guarantees by computer architecture
- Memory models may be either:
  - Strongly ordered – where a memory modification of one processor is immediately visible to all other processors
  - Weakly ordered – where a memory modification of one processor may not be immediately visible to all other processors
- A memory barrier is an instruction that forces any change in memory to be propagated (made visible) to all other processors

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed

Example (Memory Barrier)

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:

Example (Memory Barrier)

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x;
```

Example (Memory Barrier)

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x;
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true;
```

Example (Memory Barrier)

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x;
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true;
```

- For Thread 1 we are guaranteed that the value of `flag` is loaded before the value of x.

Example (Memory Barrier)

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x;
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true;
```

- For Thread 1 we are guaranteed that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
    - Currently running code would execute without preemption

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems

Operating systems using this not broadly scalable

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems

  Operating systems using this not broadly scalable
- We will look at Hardware instructions

Hardware Instructions

Special hardware instructions that allow us to either test-and-modify the content of a word, or to swap the contents of two words atomically (uninterruptedly.)

- Test-and-Set instruction
- Compare-and-Swap instruction

The `test_and_set` Instruction

- Definition

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

The test_and_set Instruction

- Definition

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

- Properties

The `test_and_set` Instruction

- Definition

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

- Properties
  - Executed atomically

The test_and_set Instruction

- Definition

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

- Properties
  - Executed atomically
  - Returns the original value of passed parameter

The test_and_set Instruction

- Definition

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

- Properties
    - Executed atomically
    - Returns the original value of passed parameter
    - Set the new value of passed parameter to true

Solution Using `test_and_set`

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

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);
```

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);
```

- Does it solve the critical-section problem?

The compare_and_swap Instruction

- Definition

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

The compare_and_swap Instruction

- Definition

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

- Properties

The compare_and_swap Instruction

- Definition

```
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

The compare_and_swap Instruction

- Definition

```
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
  - Returns the original value of passed parameter value

The compare_and_swap Instruction

- Definition

```
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
  - Returns the original value of passed parameter value
  - Set the variable `value` the value of the passed parameter `new_value` but only if `*value == expected` is true. That is, the swap takes place only under this condition.

Solution Using `compare_and_swap`

- Shared integer `lock` initialized to 0;

Solution Using compare_and_swap

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

```
while (true)
{
  while (compare_and_swap(&lock, 0, 1) != 0); /* do nothing */

  /* critical section */

  lock = 0;

  /* remainder section */
}
```

Solution Using compare_and_swap

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

```
while (true)
{
  while (compare_and_swap(&lock, 0, 1) != 0); /* do nothing */

  /* critical section */

  lock = 0;

  /* remainder section */
}
```

- Does it solve the critical-section problem?

Bounded-waiting with compare_and_swap

```
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 */
}
```

Outline

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock
  - Then `release()` the lock

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock
  - Then `release()` the lock
- Calls to `acquire()` and `release()` must be atomic

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock
  - Then `release()` the lock
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock
  - Then `release()` the lock
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires busy waiting

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock
  - Then `release()` the lock
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires busy waiting
- This lock therefore called a spinlock

Solution to CS Problem Using Mutex Locks

```
while (true)
{
  acquire lock

  /* critical section */

  release lock

  /* reminder section */
}
```

Outline

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore S – integer variable

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations: `wait()` and `signal()`

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations: `wait()` and `signal()`
  - Originally called `P()` and `V()`

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations: `wait()` and `signal()`
  - Originally called `P()` and `V()`
- Definition of the `wait()` operation

```
wait(S)
{
  while (S <= 0); // busy wait
  S = S - 1;
}
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore *S* – integer variable
- Can only be accessed via two indivisible (atomic) operations: `wait()` and `signal()`
  - Originally called `P()` and `V()`
- Definition of the `wait()` operation

```
wait(S)
{
  while (S <= 0); // busy wait
  S = S - 1;
}
```

- Definition of the `signal()` operation

```
signal(S)
{
  S = S + 1;
}
```

Semaphore (cont'd)

- Counting semaphore – integer value can range over an unrestricted domain

Semaphore (cont’d)

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1 – Same as a mutex lock

Semaphore (cont'd)

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1 – Same as a mutex lock
- Can implement a counting semaphore S as a binary semaphore

Semaphore (cont'd)

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1 – Same as a mutex lock
- Can implement a counting semaphore S as a binary semaphore
- With semaphores we can solve various synchronization problems

Example (Semaphore Usage)

- Solution to the CS Problem

Example (Semaphore Usage)

- Solution to the CS Problem
  - Create a semaphore "`mutex`" initialized to 1

```
wait(mutex);
   CS
signal(mutex);
```

Example (Semaphore Usage)

- Solution to the CS Problem
  - Create a semaphore "`mutex`" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2

Example (Semaphore Usage)

- Solution to the CS Problem
  - Create a semaphore "mutex" initialized to 1

```
wait(mutex);
   CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "synch" initialized to 0

```
P1:
  S1;
  signal(synch);
P2:
  wait(synch);
  S2;
```

Problems with Semaphores

- Incorrect use of semaphore operations:

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex) ... wait(mutex)`

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex) ... wait(mutex)`
  - `wait(mutex) ... wait(mutex)`

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex) ... wait(mutex)`
  - `wait(mutex) ... wait(mutex)`
  - Omitting of `wait(mutex)` and/or `signal (mutex)`

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex) ... wait(mutex)`
  - `wait(mutex) ... wait(mutex)`
  - Omitting of `wait(mutex)` and/or `signal (mutex)`
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

Outline

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization

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

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

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
- Pseudocode syntax of a monitor:

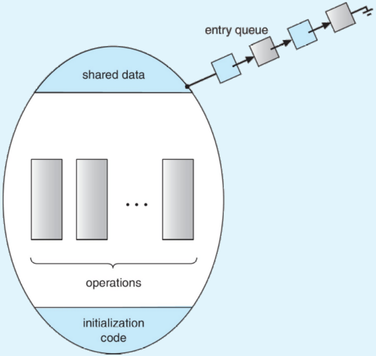
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (...) { ... }

  procedure P2 (...) { ... }

  procedure Pn (...) { ... }

  initialization code (...) { ... }
}
```

Schematic view of a Monitor



Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex
mutex = 1
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex
mutex = 1
```

- Each procedure P is replaced by

```
wait(mutex);
  ...
  body of P;
  ...
signal(mutex);
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex
mutex = 1
```

- Each procedure P is replaced by

```
wait(mutex);
  ...
  body of P;
  ...
signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Condition Variables

- `condition x, y;`

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:

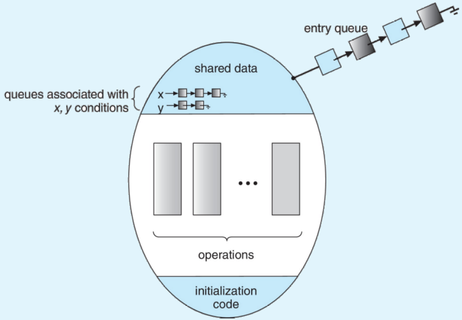
Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`

Monitor with Condition Variables



Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of processes waiting inside the monitor
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;   // (initially  = 1)
semaphore next;    // (initially  = 0)
int next_count = 0; // number of processes waiting inside the monitor
```

- Each function *P* will be replaced by

```
wait(mutex);
  ...
  body of P;
  ...
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

Implementation (Condition Variables)

- For each condition variable x, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

Implementation (Condition Variables)

- For each condition variable x, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation `x.wait()` can be implemented as:

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

Implementation (Condition Variables)

- For each condition variable x, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation `x.wait()` can be implemented as:

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

- The operation `x.signal()` can be implemented as:

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

Resuming Processes within a Monitor

- If several processes queued on condition variable `x`, and `x.signal()` is executed, which process should be resumed?

Resuming Processes within a Monitor

- If several processes queued on condition variable `x`, and `x.signal()` is executed, which process should be resumed?
- FCFS frequently not adequate

Resuming Processes within a Monitor

- If several processes queued on condition variable `x`, and `x.signal()` is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the conditional-wait construct of the form

  `x.wait(c)`

  where:

Resuming Processes within a Monitor

- If several processes queued on condition variable `x`, and `x.signal()` is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the conditional-wait construct of the form

  `x.wait(c)`

  where:

  - c is an integer (called the priority number)

Resuming Processes within a Monitor

- If several processes queued on condition variable `x`, and `x.signal()` is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the conditional-wait construct of the form

  `x.wait(c)`

  where:

  - c is an integer (called the priority number)
  - The process with lowest number (highest priority) is scheduled next

Outline

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.

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

- 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 set of properties that a system must satisfy to ensure processes make progress.

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 set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure.

Liveness

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Liveness

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
`wait(S)`	`wait(Q)`
`wait(Q)`	`wait(S)`
...	...
`signal(S)`	`signal(Q)`
`signal(Q)`	`signal(S)`

Liveness

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
`wait(S)`	`wait(Q)`
`wait(Q)`	`wait(S)`
...	...
`signal(S)`	`signal(Q)`
`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)`

Liveness

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
`wait(S)`	`wait(Q)`
`wait(Q)`	`wait(S)`
...	...
`signal(S)`	`signal(Q)`
`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)`.

Liveness

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
`wait(S)`	`wait(Q)`
`wait(Q)`	`wait(S)`
...	...
`signal(S)`	`signal(Q)`
`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.

Liveness

- Other forms of deadlock:

Liveness

- Other forms of deadlock:
- Starvation – indefinite blocking

Liveness

- Other forms of deadlock:
- Starvation – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended

Thanks! & Questions?