Process Synchronization (Part I)

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Motivation

Background

- Processes can execute concurrently.

Background

- Processes can execute concurrently.
- Concurrent access to shared data may result in data inconsistency.

Background

- Processes can execute concurrently.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Producer-Consumer Problem

- Providing a solution to the producer-consumer problem that fills all the buffers.

Producer-Consumer Problem

- Providing a solution to the producer-consumer problem that fills all the buffers.
- Having an integer counter that keeps track of the number of full buffers.

Producer-Consumer Problem

- Providing a solution to the producer-consumer problem that fills all the buffers.
- Having an integer counter that keeps track of the number of full buffers.
  - Initially, counter is set to 0.

Producer-Consumer Problem

- Providing a solution to the producer-consumer problem that fills all the buffers.
- Having an integer counter that keeps track of the number of full buffers.
  - Initially, counter is set to 0.
  - The producer produces a new buffer: increment the counter

Producer-Consumer Problem

- Providing a solution to the producer-consumer problem that fills all the buffers.
- Having an integer counter that keeps track of the number of full buffers.
  - Initially, counter is set to 0.
  - The producer produces a new buffer: increment the counter
  - The consumer consumes a buffer: decrement the counter

Producer

- Producer

```
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

- Consumer

```
while (true) {
  while (counter == 0); /* do nothing */

  next_consumed = buffer[out];

  out = (out + 1) % BUFFER_SIZE;

  counter--;
  /* consume the item in next consumed */
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

Race Condition

- counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Race Condition

- counter++ could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- counter-- could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with `count = 5` initially:
  S0: producer: `register1 = counter`: `register1 = 5`
  S1: producer: `register1 = register1 + 1`: `register1 = 6`
  S2: consumer: `register2 = counter`: `register2 = 5`
  S3: consumer: `register2 = register2 - 1`: `register2 = 4`
  S4: producer: `counter = register1`: `counter = 6`
  S5: consumer: `counter = register2`: `counter = 4`

The Critical-Section (CS) Problem

The Critical-Section Problem (1/2)

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

The Critical-Section Problem (1/2)

- Consider system of n processes $\{p_0, p_1, \cdots, p_{n-1}\}$.
- Each process has CS segment of code.

The Critical-Section Problem (1/2)

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

- Each process has CS segment of code.
  - Process may be changing common variables, updating table, writing file, etc.

The Critical-Section Problem (1/2)

- Consider system of n processes $\{p_0, p_1, \cdots, p_{n-1}\}$.
- Each process has CS segment of code.
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in CS, no other may be in its CS.

The Critical-Section Problem (2/2)

- CS problem is to design protocol to solve this.
- Each process must ask permission to enter CS in entry section, may follow CS with exit section, then remainder section.

```
do {
    entry section

        critical section

    exit section

        remainder section

} while (true);
```

General structure of process P_i.

CS Problem Solution Requirements (1/3)

- Mutual Exclusion: if process P_i is executing in its CS, then no other processes can be executing in their CSs.

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

CS Problem Solution Requirements (2/3)

- Progress: if no process is executing in its CS and there exist some processes that wish to enter their CS, then the selection of the processes that will enter the CS next cannot be postponed indefinitely.

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

CS Problem Solution Requirements (3/3)

- Bounded Waiting: a bound must exist on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted.

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

CS Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive.

CS Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive.

- Preemptive: allows preemption of process when running in kernel mode.

CS Handling in OS

- 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.

Peterson's Solution

Peterson's Solution

- Two-process solution.

Peterson's Solution

- Two-process solution.
- The two processes share two variables:
  - `int turn`
  - `boolean flag[2]`

Peterson's Solution

- Two-process solution.
- The two processes share two variables:
  - `int turn`
  - `boolean flag[2]`
- `turn`: indicates whose turn it is to enter the CS.

Peterson's Solution

- Two-process solution.
- The two processes share two variables:
  - `int turn`
  - `boolean flag[2]`
- `turn`: indicates whose turn it is to enter the CS.
- `flag`: indicates if a process is ready to enter the CS, i.e., `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);
```

CS Requirements

- Provable that the three CS requirement are met:

CS Requirements

- Provable 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`

CS Requirements

- Provable 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.

CS Requirements

- Provable 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.

Synchronization Hardware

Synchronization Hardware

- Many systems provide hardware support for implementing the CS code.

Synchronization Hardware

- Many systems provide hardware support for implementing the CS code.
- Uniprocessors: could disable interrupts: running code without preemption.

Synchronization Hardware

- Many systems provide hardware support for implementing the CS code.
- Uniprocessors: could disable interrupts: running code without preemption.
- Modern machines provide atomic hardware instructions:
  atomic = non-interruptible

Solution to CS Problem Using Locks

- Protecting CS via locks.

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

Atomic Instructions

- `test_and_set`

- `compare_and_swap`

test_and_set Instruction

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

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

Solution Using test_and_set

- Shared boolean variable lock, initialized to false.

```
do {
  while (test_and_set(&lock)); /* do nothing */

  /* critical section */

  lock = false;

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

compare_and_swap Instruction

- Executed atomically.
- Returns the original value of passed parameter value.
- Set the value the value of the passed parameter new_value but only if value == expected.

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

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

Solution Using compare_and_swap

- Shared integer `lock` initialized to 0.

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

  /* critical section */

  lock = 0;

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

compare_and_swap and compare_and_swap Problem

- Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

Mutex Locks

Mutex Locks (1/2)

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

Mutex Locks (2/2)

- Protect a CS by first `acquire()` a lock then `release()` the lock.
  - Boolean variable indicating if lock is available or not.

Mutex Locks (2/2)

- Protect a CS by first `acquire()` a lock then `release()` the lock.
  - Boolean variable indicating if lock is available or not.
- Calls to `acquire()` and `release()` must be atomic.
  - Usually implemented via hardware atomic instructions.

Mutex Locks (2/2)

- Protect a CS by first `acquire()` a lock then `release()` the lock.
  - Boolean variable indicating if lock is available or not.
- Calls to `acquire()` and `release()` must be atomic.
  - Usually implemented via hardware atomic instructions.
- But this solution requires busy waiting.
  - This lock therefore called a spinlock.

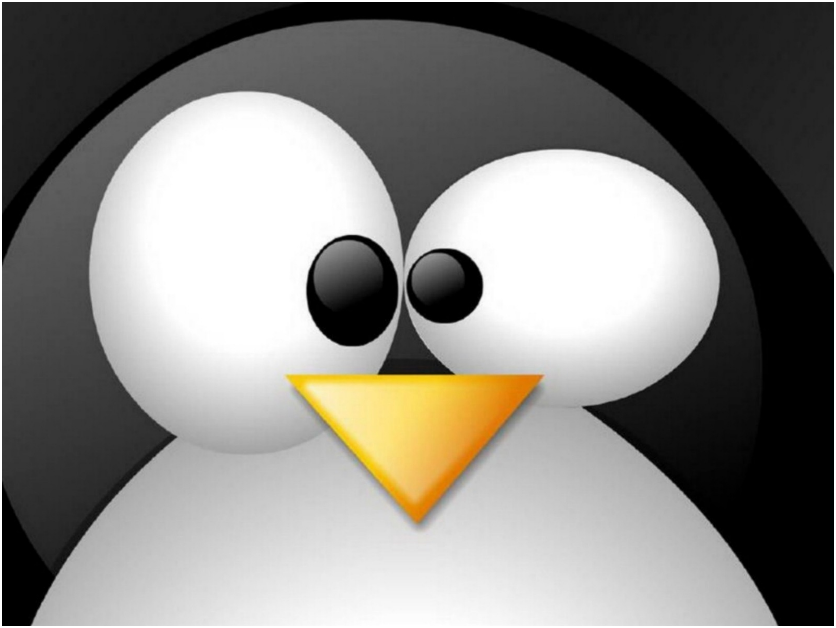
acquire() and release()

```
do {
  acquire lock
     critical section
  release lock
     remainder section
} while (true);
```

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

  available = false;
}

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



Initializing Mutexes

- Mutexes are represented by the `pthread_mutex_t` object.

```
/* define and initialize a mutex named 'mutex' */

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Locking Mutexes

- pthread_mutex_lock() locks (acquires) a pthreads mutex.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Unlocking Mutexes

- pthread_mutex_unlock() unlocks (releases) a pthreads mutex.

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Mutex Example

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int withdraw(struct account *account, int amount) {
  pthread_mutex_lock(&mutex);

  const int balance = account->balance;

  if (balance < amount) {
    pthread_mutex_unlock(&mutex);
    return -1;
  }

  account->balance = balance - amount;

  pthread_mutex_unlock(&mutex);
  disburse_money(amount);
  return 0;
}
```

Condition Variable

- Mutex is very resource consuming since the thread would be continuously busy in this activity.

Condition Variable

- Mutex is very resource consuming since the thread would be continuously busy in this activity.
- A condition variable is a way to achieve the same goal without polling.

Condition Variable

- Mutex is very resource consuming since the thread would be continuously busy in this activity.
- A condition variable is a way to achieve the same goal without polling.
- A condition variable allows one thread to inform other threads about changes in the state of a shared variable.

Waiting on Condition Variables

- pthread_cond_wait() must be called after pthread_mutex_lock() and before pthread_mutex_unlock().
- pthread_cond_wait() release the mutex lock while it is waiting, so that pthread_cond_signal(), which is also called in the mutex should get access and can give signal to waiting thread to awake.

```c
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Signaling on Condition Variables

- This routine is used to wakeup the thread which is waiting for any condition to occur.
- If in any case `pthread_cond_signal()` calls first before the execution of `pthread_cond_wait()` then it cannot awake the thread which is waiting.

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

Condition Variable Calling Format

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

s = pthread_mutex_lock(&mtx);

while (/* Check that shared variable is not in state we want */)
  pthread_cond_wait(&cond, &mtx);

/* Now shared variable is in desired state; do some work */

s = pthread_mutex_unlock(&mtx);
```

Condition Variable Example

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count() {
    pthread_mutex_lock(&count_lock);
    while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
    count = count - 1;
    pthread_mutex_unlock(&count_lock);
}

increment_count() {
    pthread_mutex_lock(&count_lock);
    if (count == 0)
        pthread_cond_signal(&count_nonzero);
    count = count + 1;
    pthread_mutex_unlock(&count_lock);
}
```

Summary

Summary

- Access to shared data

Summary

- Access to shared data
- Race condition

Summary

- Access to shared data
- Race condition
- The critical-section problem

Summary

- Access to shared data
- Race condition
- The critical-section problem
- Requirements: mutual-exclusion, progress, bounding waiting

Summary

- Access to shared data
- Race condition
- The critical-section problem
- Requirements: mutual-exclusion, progress, bounding waiting
- Peterson solution

Summary

- Access to shared data
- Race condition
- The critical-section problem
- Requirements: mutual-exclusion, progress, bounding waiting
- Peterson solution
- Synchronization hardware

Summary

- Access to shared data
- Race condition
- The critical-section problem
- Requirements: mutual-exclusion, progress, bounding waiting
- Peterson solution
- Synchronization hardware
- Mutex lock and condition variable

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.