Synchronization issues

Definition
Implementing critical sections

Outline

Synchronization issues

Programming with synchronization constraints

Objects for Ensuring Mutual Exclusion
Example: using mutex and condition variables
### Synchronization issues

**Definition**

**Implementing critical sections**

### Programming with synchronization constraints

#### Why is Synchronization Necessary?

- Know for a process / thread at which execution point is another process / thread?
- Ensure shared data consistency

→ Where is my money?!
Critical Section Problem: Definition

Critical code sections must satisfy the following requirements:

1. **Mutual exclusion (or safety condition)**: At most one process at a time is allowed to execute code inside a critical section of code.
2. **Machine independence**: No assumptions should be made about speeds or the number of CPUs.
3. **Progress**: Process running outside a critical section may not block other processes.
4. **Bounded waiting (or liveness)**: Process should be guaranteed to enter a critical section within a finite time.

Critical-sections are:

- Used when resources are shared between different processes.
- Supported by many programming mechanisms.

---

**Mutual Exclusion Using Critical Sections**

![Diagram of mutual exclusion using critical sections.]

- **Process A**
  - Executing
  - Enters critical section
  - Executing in critical section
  - Leaves critical section
  - Executing

- **Process B**
  - Executing
  - Attempts to enter critical section
  - Enters critical section
  - Leaves critical section
  - Executing in critical section
  - Blocked
  - Executing
Synchronization issues
Programming with synchronization constraints
Definition
Implementing critical sections

The Deadlock Issue

Problematic
- Use of shared resources: request, use, release
- Deadlock = situation in which a process waits for a resource that will never be available

Handling deadlocks: Use a protocol to ensure that the system will never enter a deadlock state
- **Deadlock prevention**: Restraining how requests can be made
- **Deadlock avoidance**: More information from user on the use of resources

The Deadlock Issue (Cont.)

Handling deadlocks: Allow the system to enter a deadlock state and then recover
- **Process termination**
- **Resource preemption**

Ignore the problem (i.e. assume deadlocks never occur in the system)
- **Most OS**, including UNIX
Disabling Interrupts

- Unwise to empower user processes to turn off interrupts!

Lock variables

- Procedure
  - Reads the value of a shared variable
  - If 0, sets it to 1 and enters the critical section
  - If 1, waits until the variable equals to 0
- Possible scheduling is a major flaw!
  - Can you guess why?

Strict alternation

- Busy waiting (waste of CPU)
- Violates the progress requirement of critical-sections
  - Can you guess why?

Process 0

```java
while (true) {
    while (turn != 0);
    /* begin critical section */
    turn = 1;
    /* end critical section */
}
```

Process 1

```java
while (true) {
    while (turn != 1);
    /* begin critical section */
    turn = 0;
    /* end critical section */
}
```
Synchronization issues
Programming with synchronization constraints

Software Approaches (Cont.)

Dekker’s and Peterson’s solution
- 1965, 1981
- Alternation + lock variables

Process 0
```c
while (true) {
    flag[0] = true;
    turn = 0;
    while (flag[1] && (turn == 0)) {
        /* Critical section */
        ...
        flag[0] = false;
        /* End critical section */
    }
}
```

Process 1
```c
while (true) {
    flag[1] = true;
    turn = 1;
    while (flag[0] && (turn == 1)) {
        /* Critical section */
        ...
        flag[1] = false;
        /* End critical section */
    }
}
```

Hardware Approaches

The Test and Set Lock (TSL) Instruction
- Special assembly instruction which is **atomic**
- TSL Rx, LOCK
  - Reads the content of the memory at address lock and stores it in register Rx

Assembly code to enter / leave critical sections
```
```
```c
Enter_critical_section:
    TSL register, LOCK
        | copy lock to register and set lock to 1
    CMP register, #0
        | was lock equal to 0?
    JNE Enter_critical_section
        | if != 0 -> lock was set -> loop
    RET
        | return to caller -> ok to enter critical section

Leave_critical_section:
    MOVE LOCK, #0
        | Store a 0 in lock
    RET
        | return to caller
```

Ludovic Apvrille
OS - VII. Synchronization - Fall 2009
11 of 25
And so...

**Limits of Peterson's and TSL solutions**

- Busy waiting
- **Priority inversion problem**: If a lower priority process is in critical section and a higher priority process busy waits to enter this critical section, the lower priority process never gains CPU → higher priority processes can never enter critical section

**Solution: Sleep and wakeup system calls**

- \textit{Sleep()}: Caller is blocked on a given address until another process wakes it up
- \textit{Wakeup()}: Caller wakes up all processes waiting on a given address

---

**Outline**

Synchronization issues

Programming with synchronization constraints

Objects for Ensuring Mutual Exclusion

Example: using mutex and condition variables
Semaphores

**Definition**

- A semaphore is a **counter** shared by multiple processes.
- Processes can **increment** or **decrement** this counter in an atomic way.
- Mainly used to protect access to shared resources.
- But semaphores are quite complex to use.

**Semaphores: Main functions**

**Creating Semaphores**

```c
int semget(key_t key, int nsems, int flag);
```

**Modifying the value of a semaphore**

```c
struct sembuf {
   ushort sem_num; /*member # in the set of semaphores */
   short sem_op;  /* operation: negative (-1), 0, +1, etc.
short sem_flg; /* IPC_NOWAIT, SEM_UNDO */
}

int semop(int semid, struct sembuf semoarray[], size_t nops);
```
Using Semaphores to Share Access to a Resource

**Initialization**
- A semaphore is associated to each resource
- Value of the semaphore is initialized with `semctl()` to the maximum number of processes which can access to this resource at the same time

**Using the Semaphore: `semop()`**
- Before accessing a given shared resource, a process tries to decrease the value of the semaphore
  - If the value is 0, the process is suspended until the value of the semaphore becomes positive
- To release a resource, a process increments the value of the semaphore

**Mutexes**

**Definition**
- **Mutual Exclusion**
- A mutex has two states: locked, unlocked
- Only one thread / process at a time can lock a mutex
- When a mutex is locked, other processes / threads block when they try to lock the same mutex:
  - Locking stops when the mutex is unlocked
  - One of the waiting process / thread succeeds in locking the mutex
Mutexes: Main functions

**Initialize a mutex**

```c
pthread_mutex_t mymutex;
```

**Lock the mutex**

- Waits for the lock to be released if mutex is already locked
  ```c
  pthread_mutex_lock(&mymutex);
  ```

- Returns immediately if mutex is locked
  ```c
  pthread_mutex_trylock(&mymutex);
  ```

**Unlock the mutex**

```c
pthread_mutex_unlock(&mymutex);
```

Condition Variables

- Used to signal a condition has changed

**To wait on a condition**

- Put lock on mutex
- Wait on that condition → Automatic release of the lock

**To signal a change on a condition**

- Put lock on mutex
- Signal that condition
Use of Condition Variables

Producer / Consumer Example

```c
#include <stdlib.h>
#include <pthread.h>

#define N_THREADS_PROD 3
#define N_THREADSCONS 4

void *produce(void *); void *produceData(int id);
void *consume(void *); void *consumeData(int id);

int data = 0; int maxData = 5;
pthread_mutex_t myMutex;
pthread_cond_t full, empty;

int main(void) {
    int i;
    pthread_t tid_p[N_THREADS_PROD];
    pthread_t tid_c[N_THREADSCONS];
    for (i = 0; i < N_THREADS_PROD; i++) {
        pthread_create(&tid_p[i], NULL, produce, (void *)i);
    }
    for (i = 0; i < N_THREADSCONS; i++) {
        pthread_create(&tid_c[i], NULL, consume, (void *)i);
    }

    for (i = 0; i < N_THREADS_PROD; i++) {
        pthread_join(tid_p[i], NULL);
    }
    for (i = 0; i < N_THREADSCONS; i++) {
        pthread_join(tid_c[i], NULL);
    }
    return (0);
}
```
Synchronization issues
Programming with synchronization constraints
Objects for Ensuring Mutual Exclusion
Example: using mutex and condition variables

Producer / Consumer Example (Cont.)

```c
#include <stdio.h>
#include <string.h>
#include <pthread.h>

int maxData = 100;

void *produce(void *arg) {
    int myId = (int)arg;
    while(1) {
        produceData(myId);
        sleep(random() % 5);
    }
}

void *consume(void *arg) {
    int myId = (int)arg;
    printf("I am the consumer %d\n", myId);
    while(1) {
        consumeData(myId);
        sleep(random() % 5);
    }
}

int consumeData(int id) {
    pthread_mutex_lock(&myMutex);
    while (data == 0) {
        printf("%d is waiting for more data; data = %d\n", id, data);
        pthread_cond_wait(&empty, &myMutex);
    }
    data--;
    printf("%d is consuming data; data = %d\n", id, data);
    pthread_mutex_unlock(&myMutex);
    pthread_mutex_unlock(&myMutex);
    return 0;
}

int produceData(int id) {
    pthread_mutex_lock(&myMutex);
    while (data == maxData) {
        printf("%d is waiting for less data; data = %d\n", id, data);
        pthread_cond_wait(&full, &myMutex);
    }
    data++;
    printf("%d is producing data; data = %d\n", id, data);
    pthread_mutex_unlock(&myMutex);
    return 0;
}
```

Why do we use `while` and not `if`?
Synchronization issues
Programming with synchronization constraints
Objects for Ensuring Mutual Exclusion
Example: using mutex and condition variables

Producer / Consumer Example: Execution

$ gcc -lpthread prod cons.c
$ prod

#3 is waiting for more data; data = 0
#1 is waiting for more data; data = 0
#2 is producing data; data = 1
#2 is producing data; data = 2
#0 is consuming data; data = 1
#2 is consuming data; data = 0
#2 is waiting for more data; data = 0
#3 is waiting for more data; data = 0
#0 is producing data; data = 1
#1 is consuming data; data = 0
#2 is producing data; data = 1
#2 is consuming data; data = 0
#1 is producing data; data = 1
#3 is consuming data; data = 0
#0 is waiting for more data; data = 0
#0 is producing data; data = 1
#0 is consuming data; data = 0
#0 is waiting for more data; data = 0
#3 is waiting for more data; data = 0
#1 is waiting for more data; data = 0
#2 is producing data; data = 1
#2 is producing data; data = 2
#2 is consuming data; data = 1
#0 is consuming data; data = 0
#1 is producing data; data = 1