Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department
1400-1 semester

Zeinab Zali

Session 13: Synchronization

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

Illustration of the problem: Calculating summation of numbers with some threads

- We want to calculate the summation of some numbers with more than one thread to speed up the operation
- Consider a global variable sum
- We use data parallelism, divide numbers in to some sets, create a thread for each set to add numbers to sum
- Execute the code in next page and see the result

Illustration of the problem: Calculating summation of numbers with some threads

```
/******************************************************
Concurrency problem
******************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>


struct thread_input{
int b;
int e;
};

int sum = 0;

void *summation_thread(void *input)
{
   struct thread_input *arg;
   arg = (struct thread_input*)input;
   for (int i=arg->b; i<=arg->e; i++){
  sum += i;
}
   pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
{
  int n = 10000000;
  pthread_t threads[NUM_THREADS];
  int rc, t;
  struct thread_input beg_end[NUM_THREADS];
  int d = n / NUM_THREADS ;
  for(t=0;t<NUM_THREADS;t++){
        beg_end[t].b = t * d + 1;
         beg_end[t].e = beg_end[t].b + d - 1;
         rc = pthread_create(&threads[t], NULL, summation_thread, (void *)&beg_end[t]);
         if (rc){
                    printf("ERROR; return code from pthread_create() is %d\n", rc);
                    exit(-1);
          }
  }
  for (t=0;t<NUM_THREADS;t++){
          pthread_join(threads[t], NULL);
  }
  printf("sum= %d\n", sum);
  /* Last thing that main() should do */
  pthread_exit(NULL);
}
```

Illustration of the problem: updating a linked list

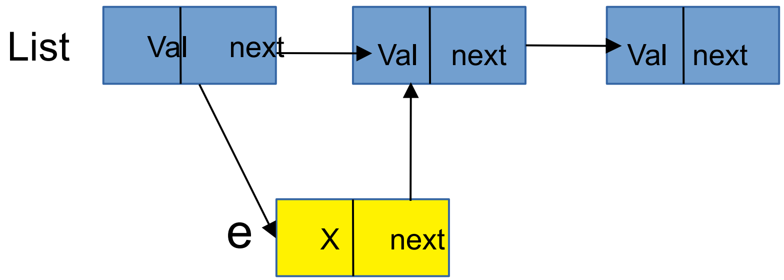
Create new list element e

Set e.value = X

Read list and list.next

Set e.next=list.next

Set list.next=e



What happens if two threads try to add an element concurrently to the same list??

Illustration of the problem: producer-consumer 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.
- **counter** is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

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

```
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
```

- `counter--` could be implemented as

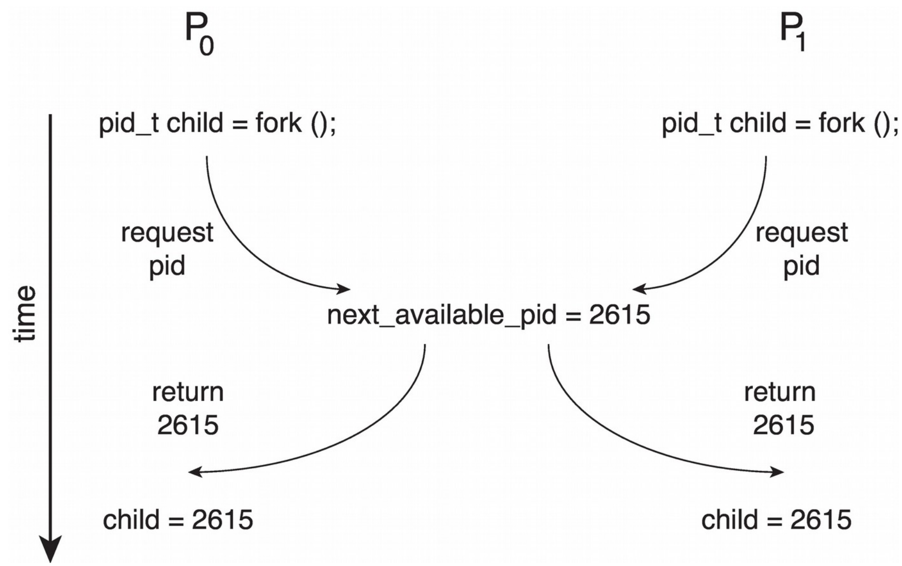
```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving 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

- 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}}
- 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) {

        entry section

            critical section

        exit section

            remainder section

}
```

Critical-Section Problem (Cont.)

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

Definitions

- **Critical resource (منبع بحرانی)** : a shared resource between more than one threads (processes) that can not be used or updated concurrently by them.
- **Critical section (ناحیه بحرانی):** a section of code for updating a critical resource. Ex: write to printer buffer, updating a table in database, writing to a file
- **Race condition (شرایط رقابتی):** a situation where several processes access and manipulate the same data (a critical resource) concurrently.
- **Synchronizion (همگام سازی):** Orderly execution of cooperating processes that share a critical resource
- **Deadlock (بن بست):** two or more processes are blocked with each other to enter the critical section and progress execution.
- **Starvation (گرسنگی):** a process wait indefinitely for entering the critical section

Critical-Section Handling in OS

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

- **Preemptive** **قبضه شدنی یا غیرانحصاری** – allows preemption of process when running in kernel mode
- **Non-preemptive** **قبضه نشدنی یا انحصاری** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
- preemptive kernels are difficult to design especially in SMP (Symmetric Multi-Processing), but they are more responsive and suitable for real-time

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
- 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;

    /* remainder section */

}
```

Correctness of the Software Solution

- Mutual exclusion is preserved

  P_i enters critical section only if:

  `turn = i`

  and **`turn`** cannot be both 0 and 1 at the same time
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

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

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
     critical section
   flag[i] = false;
     remainder section
} while (true);
```

```
do {
   flag[j] = true;
   turn = i;
   while (flag[i] && turn == i);
     critical section
   flag[j] = 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 `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied

     after critical section flag[i] set to be false
  3. Bounded-waiting requirement is met

     it is achieved through setting turn correctly
- Peterson’s solution is not guaranteed to work on modern computer architectures
  - Because of reordering read and write operations that have 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 that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Modern Architecture Example

- Two threads share the data:

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

- Thread 1 performs

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

- Thread 2 performs

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

- What is the expected output?

100

Modern Architecture Example (Cont.)

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

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

  for Thread 2 may be reordered

- If this occurs, the output may be 0!

Reordering problem for peterson

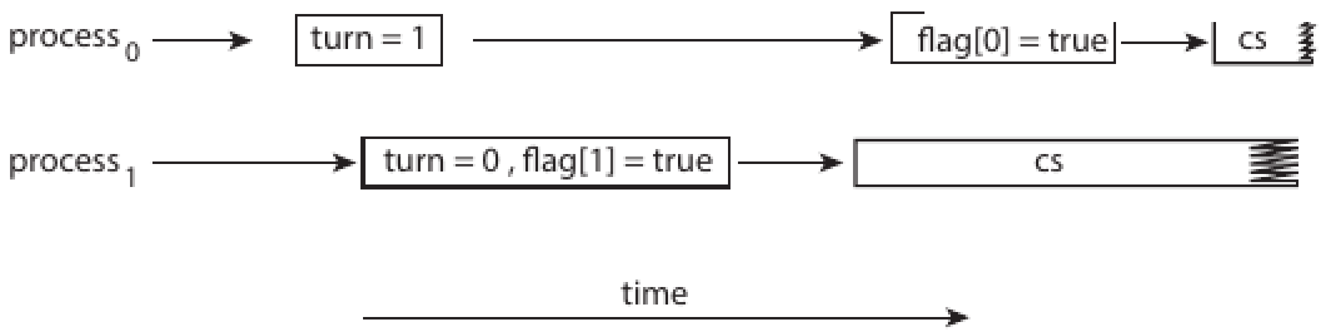
```
boolean flag = false;
int x = 0;
```

Thread 1

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

Thread 2

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



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**.