CS3733: Operating Systems

Topics: Synchronization, Critical Sections and Semaphores (SGG Chapter 6)

Instructor: Dr. Tongping Liu

Outline
- Memory Model of Multithreaded Programs
- Synchronization for coordinated processes/threads
- The produce/consumer and bounded buffer problem
- Critical Sections and solutions' requirements
- Peterson's solution for two processes/threads
- Synchronization Hardware: atomic instructions
- Semaphores
  - Two operations (wait and signal) and standard usages
  - Bounded buffer problem with semaphores
    - Initial values and order of operations

Threads Memory Model

- Conceptual model:
  - Multiple threads run in the same context of a process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, and GP registers
  - All threads share the remaining process context
    - Code, data, heap, and shared library segments
    - Open files and installed handlers
- Operationally, this model is not strictly enforced:
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread

Example Program to Illustrate Sharing

```
char **ptr; /* global */
int main()
{
  int i;
  pthread_t tid;
  char *msgs[2] = {"Hello from foo", "Hello from bar"};
  ptr = msgs;
  for (i = 0; i < 2; i++)
    pthread_create(&tid, NULL, thread, (void *)i);
  pthread_exit(NULL);
}

/* thread routine */
void *thread(void *vargp)
{
  int myid = (int)vargp;
  static int cnt = 0;
  printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++cnt);
}
```

Mapping Variable Instances to Memory

- Global variables
  - Def: Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable
- Local variables
  - Def: Variable declared inside function without static attribute
  - Each thread stack contains one instance of each local variable
- Local static variables
  - Def: Variable declared inside function with the static attribute
  - Virtual memory contains exactly one instance of any local static variable.

Peer threads reference main thread's stack indirectly through global ptr variable
Mapping Variable Instances to Memory

```
Mapping Variable Instances to Memory

| Global var: 1 instance (ptr [data]) |
| Local var: 1 instance (i, n, msgs[r]) |
| Local var: 2 instances (myid.c0, myid.t1) |

char **ptr; /* global */

int main()
{
  int i;
  pthread_t tid;
  char *msgs[2] = {
    "Hello from foo",
    "Hello from bar"
  };
  ptr = msgs;
  for (i = 0; i < 2; i++)
    pthread_create(&tid, NULL, thread, (void *)i);
}

/* thread routine */
void *thread(void *vargp)
{
  int myid = (int)vargp;
  static int cnt = 0;
  printf("[%d]: %s (svar=%d)
", myid, ptr[myid], ++cnt);
}
```

What is possible output?

Different outputs

[0]: Hello from foo (svar=1)
[1]: Hello from bar (svar=2)
[1]: Hello from bar (svar=1)
[0]: Hello from foo (svar=1)
[0]: Hello from foo (svar=1)

Concurrent Access to Shared Data

- Two threads A and B have access to a shared variable “Balance”

- Thread A:
  Balance = Balance + 100
  A1. LOAD R1, BALANCE
  A2. ADD R1, 100
  A3. STORE BALANCE, R1

- Thread B:
  Balance = Balance - 200
  B1. LOAD R1, BALANCE
  B2. SUB R1, 200
  B3. STORE BALANCE, R1

What is the problem then?

- Observe: In a time-shared system, the exact instruction execution order cannot be predicted

  - Scenario 1:
    A1. LOAD R1, BALANCE
    A2. ADD R1, 100
    A3. STORE BALANCE, R1
    Context Switch!
    B1. LOAD R1, BALANCE
    B2. SUB R1, 200
    B3. STORE BALANCE, R1
    Context Switch!
  - Scenario 2:
    B1. LOAD R1, BALANCE
    B2. ADD R1, 100
    A3. STORE BALANCE, R1
    Context Switch!
    A1. LOAD R1, BALANCE
    A2. ADD R1, 100
    Context Switch!
    B3. STORE BALANCE, R1

Race Conditions

- Multiple processes/threads write/read shared data and the outcome depends on the particular order to access shared data are called race conditions
  A serious problem for concurrent system using shared variables!

  How do we solve the problem?!

  Need to make sure that some high-level code sections are executed atomically
  Atomic operation means that it completes in its entirety without worrying about interruption by any other potentially conflict-causing process

What is Synchronization?

- Cooperating processes/threads share data & have effects on each other → executions are NOT reproducible with non-deterministic exec. speed
- Concurrent executions
  - Single processor → achieved by time slicing
  - Parallel/distributed systems → truly simultaneous
- Synchronization → getting processes/threads to work together in a coordinated manner
Producer/Consumer Problem

- The producer/consumer problem
  - A producer process/thread *produces/generates* data
  - A consumer process/thread *consumes/uses* data
  - Buffer is normally used to exchange data between them

- Bounded buffer: shared by producer/consumer
  - The buffer has a *finite amount* of buffer space
  - Producer must wait if no buffer space is available
  - Consumer must wait if all buffers are empty and no data is available

Bounded Buffer: A Simple Implementation

- Shared circular buffer of size n
  ```
  item buffer[n];
  int in, out, counter;
  ```
  - in points to the next free buffer
  - out points to the first full buffer
  - counter contains the number of full buffers

- No data available when counter = 0
- No available space to store additional data when counter = n

Producer/Consumer Loops

- Producer Loop
  ```
  produce an item in nextp:
  while (counter == n) {
    buffer[in] = nextp;
    in = (in+1) % n;
    counter++;
  }
  ```
  For the shared variable counter, What may go wrong?

- Consumer Loop
  ```
  while (counter == 0) {
    nexto = buffer[out];
    out = (out+1) % n;
    counter--;
    consume the item in nexto
  }
  ```

++/-- Operations in Hardware

- Hardware
  - Data in memory cannot be operated directly
  - Data in memory should be loaded to cache and registers first
  - Operations can be applied to registers only
  - Data in registers can be saved to memory

```
R1 = counter
R1 = R1 + 1
counter = R1
```
```
R2 = counter
R2 = R2 - 1
counter = R2
```

Executions of these instructions can be interrupted *anywhere*.
What will be potential problem?

Mutual Exclusion

Critical Sections

- Critical section (CS)
  - A section of code that modify the same shared variables/data that must be executed mutually exclusively in time

- General structure for processes/threads with CS
  - `entry section`: The code which requests permission to enter the critical section.
  - `critical section`: as above
  - `exit section`: The code which removes the mutual exclusion.
  - `remainder section`: Everything else
### General Structure for Critical Sections

```c
do {
    ...  
    entry section
    critical section
    exit section  
    remainder section
} while (1);
```

In the **entry section**, the process requests "permission".

### Requirements for CS Solutions

- **Mutual Exclusion**
  - At most one process/thread in its CS at any time
- **Progress**
  - If all other processes/threads are in their remainder sections, a process/thread is allowed to enter its CS
  - Only those processes that are not in their remainder section can decide which process can enter its CS next, and this decision cannot be postponed indefinitely.
- **Bounded Waiting**
  - Once a process has made a request to enter its CS → other processes can only enter their CSs with a bounded number of times

### A Simple (Wrong) Solution

```c
int turn; // indicate whose turn to enter CS  
T0 and T1: alternate between CS and remainder
```

- Process 0:
  ```c
  while(TRUE) {  
    while (turn != 0) ;  
    critical section  
    turn = 1;  
    remainder section  
  }
  ```

- Process 1:
  ```c
  while(TRUE) {  
    while (turn != 1) ;  
    critical section  
    turn = 0;  
    remainder section  
  }
  ```

**What is the problem with this solution?**
- Mutual exclusion?
- Progress?
- Bounded Waiting?

### Peterson’s Solution

- **Three shared variables**: `turn` and `flag[2]`

#### How does this solution satisfy the CS requirements (mutual exclusion, progress, bounded waiting)?

- **Mutual Exclusion**
  - Q: when process/thread 0 is in its critical section, can process/thread 1 get into its critical section?
  - \( \text{flag}[0] = 1 \), and either \( \text{flag}[1]=0 \) or \( \text{turn} = 0 \)
  - Thus, for process/thread (1): \( \text{flag}[0]=1 \); if \( \text{flag}[1]=0 \), it is in reminder section; if \( \text{turn} = 0 \), it waits before its CS

- **Progress**: can P0 be prevented into CS, e.g. stuck in while loop?
- **Bounded Waiting**

#### How does this solution break the CS requirements (mutual exclusion, progress, bounded waiting)?

- **Toggle first two lines for one process/thread**

```c
Process 0 loop:  
------------------  
flag[1] = 1;  
while (flag[1] && turn==1);  
while (flag[1] && turn==0);  
flag[1] = 0;  
remainder section
```
Hardware Solution: Disable Interrupt

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Inefficient on multiprocessor systems

```c
do {
    DISABLE_INTERRUPT
    critical section
    ENABLE_INTERRUPT
    remainder statements
} while (1);
```

What are the problems with this solution?
1. Time consuming, decreasing efficiency
2. Clock problem
3. Machine dependent

Hardware Support for Synchronization

- Synchronization
  - Need to test and set a value atomically
  - IA32 hardware provides a special instruction: `xchg`
    - When the instruction accesses memory, the bus is locked during its execution and no other process/thread can access memory until it completes!!
  - Other variations: `xchgb`, `xchgw`, `xchg1`

- Other hardware instructions
  - `TestAndSet (a)`
  - `Swap (a,b)`

Hardware Instruction `TestAndSet`

- The `TestAndSet` instruction tests and modifies the content of a word atomically (non-interruptable)
  - Keep setting the lock to 1 and return old value.

```c
bool TestAndSet(bool *target){
    boolean m = *target;
    *target = true;
    return m;
}
```

What's the problem?
1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting

Another Hardware Instruction: Swap

- Swap contents of two memory words
  ```c
  void Swap (bool *a, bool *b){
      bool temp = *a;
      *a = *b;
      *b = temp;
  }
  ```

- `lock = FALSE;` 
  ```c
  While(true){
      bool key = TRUE;
      while(key == TRUE) {
          Swap(&key, &lock) ;
      }
      critical section;
      lock = FALSE; //release permission
  }
  ```

What's the problem?
1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting

Semaphores

- Synchronization without busy waiting
  - Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied

- Semaphore S – integer variable
  - Two indivisible (atomic) operations: `wait(s)` (also called `P(s)` or `down(s)` or `acquire()`);
    `signal(s)` (also called `V(s)` or `up(s)` or `release()`)
  - User-visible operations on a semaphore
  - Easy to generalize, and less complicated for application programmers

Semaphore Operations

- Semaphore is an integer.
  ```c
  wait(value) {
  }
  signal(value) {
  }
  ```

  ```c
  void signal(value){
      value++;
  }
  ```

- `wait(s):` //wait until s.value > 0; 
  - s.value--; /* Executed atomically */
  - The value of s could be negative → the number of waits

- A process execute the wait operation on a semaphore with value <=0 is blocked
  - Blocked → non-runnable state, yield CPU to others

- `signal(s):` s.value++; /* Executed atomically */
  - If =1, wake up only one blocked process; which one?!
Semaphores without Busy Waiting

- The idea:
  - Once need to wait, remove the process/thread from CPU
  - The process/thread goes into a special queue waiting for a semaphore (like an I/O waiting queue)
  - OS/runtime manages this queue (e.g., FIFO manner) and remove a process/thread when a signal occurs
- A semaphore consists of an integer (S.value) and a linked list of waiting processes/threads (S.list)
  - If the integer is 0 or negative, its magnitude is the number of processes/threads waiting for this semaphore.
- Start with an empty list, and normally initialized to 0

Implement Semaphores

typedef struct{
  int value;
  struct process *list;
} semaphore;

wait(semaphore * s){
  s->value--;
  if (s->value <0){
    enlist(s->list);
    block();
  }
}

signal(semaphore * s){
  s->value++;
  if (s->value <= 0)
    delist(P, s->list);
  wakeup(p);
}

Is this one without busy waiting?

Semaphore Usage

- Counting semaphore –
  - Can be used to control access to a given resources with finite number of instances
- Binary semaphore – integer value can range only between 0 and 1; Also known as mutex locks

Attacking CS Problem with Semaphores

- Shared data
  - semaphore mutex = 1; /* initially mutex = 1 */
- For any process/thread
  do {
    ...
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
  } while(1);

Revisit “Balance Update Problem”

- Shared data:
  int Balance;
  semaphore mutex; // initially mutex = 1

- Process A:
  ******
  wait (mutex);
  Balance = Balance – 100;
  signal (mutex);
  ******

- Process B:
  ******
  wait (mutex);
  Balance = Balance – 200;
  signal (mutex);
  ******
Semaphore for General Synchronization

- Execute code B in P_j after code A is executed in P_i.
- Use semaphore flag: what is the initial value?
- Code
  
  \[
  \begin{aligned}
  P_i: & \quad P_j: \\
  \vdots & \quad \vdots \\
  A & \quad \text{wait(flag)} \\
  \text{signal(flag)} & \quad B
  \end{aligned}
  \]

What about 2 threads wait for 1 thread? Or 1 thread waits for 2 threads?

Classical Synchronization Problems

- Producer-Consumer Problem
  - Shared bounded-buffer
  - Producer puts items to the buffer area, wait if buffer is full
  - Consumer consumes items from the buffer, wait if is empty

- Readers-Writers Problem
  - Multiple readers can access concurrently
  - Writers mutual exclusive with writes/readers

- Dining-Philosophers Problem
  - Multiple resources, get one at each time

Producer-Consumer Problem

With Bounded-Buffer

- Need to make sure that
  - The producer and the consumer do not access the buffer area and related variables at the same time
  - No item is available to the consumer if all the buffer slots are empty.
  - No slot in the buffer is available to the producer if all the buffer slots are full

What Semaphores are needed?

- semaphore mutex, full, empty;

  **What are the initial values?**

  - Initially:
    
    \[
    \begin{aligned}
    \text{full} = 0 & /* The number of full buffer slots */ \\
    \text{empty} = n & /* The number of empty buffer slots */ \\
    \text{mutex} = 1 & /* controlling mutual access to the buffer pool */ 
    \end{aligned}
    \]

Producer/Consumer Codes

- Producer Loop
  
  ```
  do { … produce an item in nextp … wait(empty); wait(mutex); … add nextp to buffer … signal(mutex); } while (1)
  ```

- Consumer Loop
  
  ```
  do { … wait(full); wait(mutex); … remove an item from buffer to nextc … signal(mutex); signal(empty); … consume the item in nextc } while (1)
  ```

What will happen if we change the order?
Summary

- Synchronization for coordinated processes/threads
- The produce/consumer and bounded buffer problem
- **Critical Sections** and solutions’ requirements
- **Peterson’s solution** for two processes/threads
- Synchronization Hardware: **atomic** instructions
- Semaphores
  - Its two operations **wait** and **signal** and standard usages
- Bounded buffer problem with semaphores
  - Initial values and order of operations are crucial