CSE 3221.3 Operating System Fundamentals

No.5

Process Synchronization(1)

Prof. Hui Jiang Dept of Computer Science and Engineering York University

Background: cooperating processes with shared memory

- Many processes or threads are cooperating: – One way is to use shared memory.
- But concurrent access to shared data may result in data inconsistency.

Process Synchronization

- How data inconsistency happens?
- Example: producer-consumer problem using a bounded-buffer
- Pure software solution:
- 2-process: Peterson's algorithm
- N-process: Bakery algorithm
- Synchronization hardware
- Semaphores
- Three classic synchronization problems:
- The bounded-buffer problem.
- The reader-writer problem.
- The dining-philosopher problem.

Producer-Consumer Problem: using shared memory

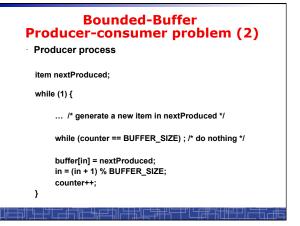
Producer-Consumer problem:

- Two parties: producer & consumer processes
- A producer process produces information that is consumed by a consumer process.
- Shared memory:
 - Bounded buffer: a fixed buffer size (producer blocks when the buffer is full)
- Example:
 - Printer program → printer driver
 - Compiler → assembler

Bounded-Buffer Producer-consumer problem (1) Shared data #define BUFFER_SIZE 10 twood of struct /

typedef struct {

```
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```



[•] To share data among processes (threads), we need some mechanisms to ensure the orderly execution of cooperating processes (threads) to maintain data consistency.

Bounded-Buffer Producer-consumer problem(3)

Consumer process

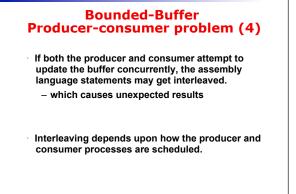
item nextConsumed;

while (1) {

}

while (counter == 0) ; /* do nothing */
nextConsumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;

... /* consume the item in nextConsumed */



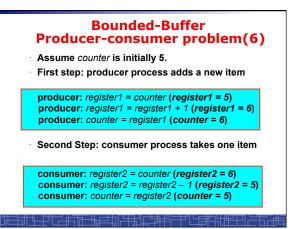
Bounded-Buffer Producer-consumer problem (5)

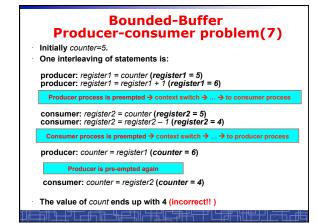
The statement "counter++" may be implemented in machine language as:

register1 = counter register1 = register1 + 1 counter = register1

The statement "counter--" may be implemented as:

register2 = counter register2 = register2 - 1 counter = register2







 Race condition: The situation where several processes (or threads) access and manipulate shared data concurrently. The final value of the shared data depends upon the particular order in which the access takes place.

General to all shared data in multiprogramming systems.

• Race condition happens in:

- Multiple processes with shared memory

- Multi-threaded program
- Preemptive OS kernel

Race Conditions in OS

Non-preemptive kernels

- No race condition occurs in kernel.
- Preemptive kernels
 - Race condition could occur in kernel.
 - Protection techniques are needed for all shared data.
 Examples:
 - Moving several PCB's from one waiting queue to ready queue; moving several PCB's to the same waiting queue.
 - · Kernel counters; kernel flags, ...

Bounded-Buffer Producer-consumer problem (8)

The statements

counter++; (in producer)

counter --; (in consumer)

must be performed atomically.

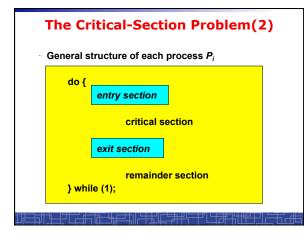
• Atomic operation means an operation that completes in its entirety without interruption.

Process Synchronization

- Process synchronization: To prevent race conditions, concurrent processes must be synchronized to ensure an orderly execution sequence for all processes.
 - <u>To ensure only one process can manipulate the shared data at a time.</u>
 (the key idea of process synchronization)

The Critical-Section Problem(1)

- n processes all competing to use some shared data.
- Each process has code segments, called *critical* section, in which the shared data is accessed.
- Ensure that when one process is executing in its critical section, no other processes are allowed to execute in their critical sections.
 - The execution of critical sections by processes is mutually exclusive in time.



Solution to Critical-Section Problem

- 1. Mutual Exclusion
 - If a process is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress

 If no process in its critical section and some processes wish to enter their critical sections, then only these processes wishing to enter the critical section can participate in the decision on which will enter the critical section next, and the decision selection of the processes cannot be postponed indefinitely.

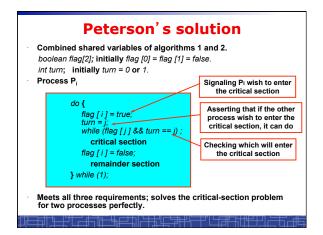
3. Bounded Waiting

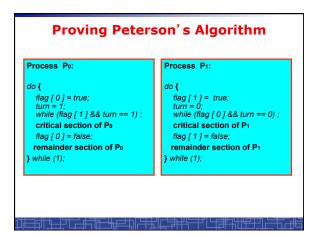
 After a process has made a request to enter its critical section, there much be a bound on the number of times that other processes are allowed to enter their critical sections before that request is granted.

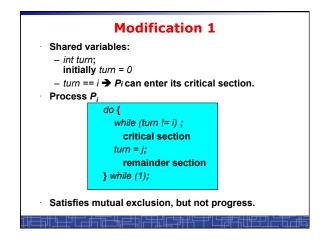
Prepared by Prof. Hui Jiang (CSE 3221)

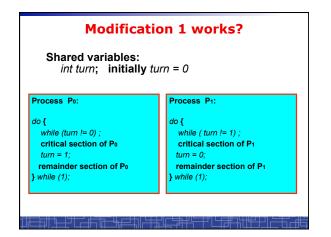
Software Solution to the critical section problem

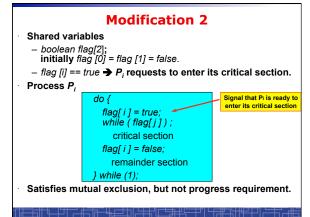
- Assume each process is executing at a non-zero speed. No assumption on their relative speed.
- No assumption on special hardware instructions except each instruction is executed atomically.
- No assumption on the number of CPU's in the system.
- Starting from the case with only two processes:
 Process P₀ and P₁
 - When presenting Pi, use Pj to indicate another (j=1-i)

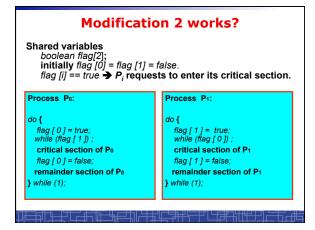


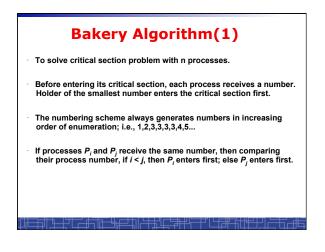












Bakery Algorithm(2)

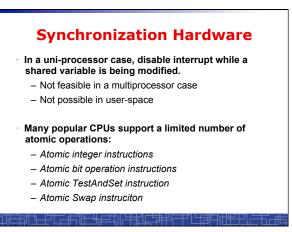
Notation: (ticket #, process id #)

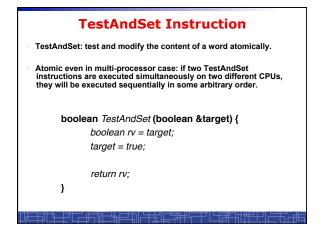
- (a,b) < (c,d) if a < c or if a = c and b < d
- max (a₀, a₁, ..., a_{n-1}) is a number, k, such that k >= a_i for i = 0, ..., n - 1

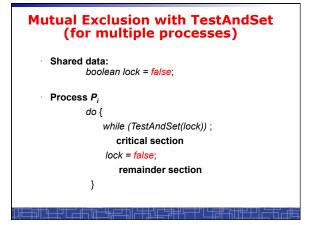
Shared data boolean choosing[n]; (Initially false) int number[n]; (Initially 0)

Bakery Algorithm(3) The Structure of process Pi do { choosing[i] = true; number[i] = max(number[0], number[1], ..., number [n - 1])+1; choosing[i] = false; for (j = 0; j < n; j++) { while (choosing[j]); while ((number[j]!= 0) && (number[j];j) < (number[i]:i,i); } critical section number[i] = 0; remainder section } while (1); </pre>

| Bakery Algorithm(3) The Structure of process Pi | |
|-------------------------------------------------|-----------------------------------------------------------------|
| | |
| numb | er[i] = max(number[0], number[1],, number [n – 1])+1; |
| for (j : | = 0; j < n; j++) { |
| | while ((number[j] != 0) && (number[j],j) < (number[i],i)) |
| } | cal section |
| | er[i] = 0; |
| | ainder section |
| } while (1 |); |







Swap(lock,key);

