Module 20: Multi-core Computing Multi-processor Scheduling

Lecture 40: Multi-core Computing Synchronization

The Lecture Contains:

Process Dispatching

- After assignment, deciding who is selected from among the pool of waiting processes
    - Process dispatching.
- Single processor multiprogramming strategies may be counter-productive here.
- Priorities and process history may not be sufficient.

Process Scheduling

- Single queue of processes or if multiple priority is used, multiple priority queues, all feeding into a common pool of processors.
- Multi-server queuing model: multiple-queue/single queue, multiple server system.
    - Inference: Specific scheduling policy does not have much effect as the number of processors increase.
- Conclusion: Use FCFS with priority levels.

Previous Next

Thread Scheduling

- An application can be implemented as a set of threads that cooperate and execute concurrently in the same address space.
- Load Sharing: pool of threads, pool of processors.
- Gang scheduling: Bunch of related threads scheduled together.
- Dedicated processor assignment: Each program gets as many processors as there are parallel threads.
- Dynamic scheduling: More like demand scheduling.

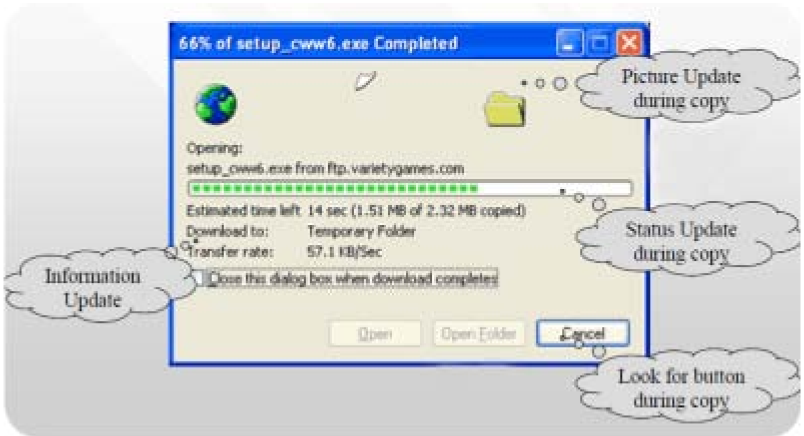
Multi-core Computing Synchronization

Problem

- Multiple concurrent processes or threads using shared memory to communicate.
    - Concurrent access to the same data may lead to inconsistencies.
- An innocent looking code may not work when concurrency is involved.
- Remember *Producer-Consumer* code.

Previous Next

Anatomy of Downloading Window



The Producer-Consumer Problem

```
Produce(item_t item) {
while (count==bufsz);
buffer[in]=item;
in=(in+1)%bufsz;
count=count+1;
}
```

```
Consume(item_t *item) {
while(count==0);
*item=buffer[out];
out=(out+1)%bufsz;
count=count-1;
}
```

Shared Variables

buffer, count

Previous Next

What is Wrong?

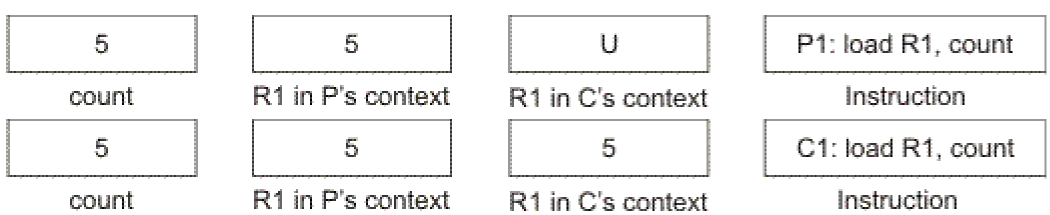
- Variable count is shared.
- Both process read and modify this variable
- The assembly code for these two statement may be something like following:

	//count=count+1			//count=count-1	
P1:	load	R1,count	C1:	load	R1,count
P2:	add	R1,1	C2:	sub	R1,1
P3:	store	count,R1	C3:	store	count,R1

- Two processes run concurrently (Single or multiple CPUs)

A Possible Scenario

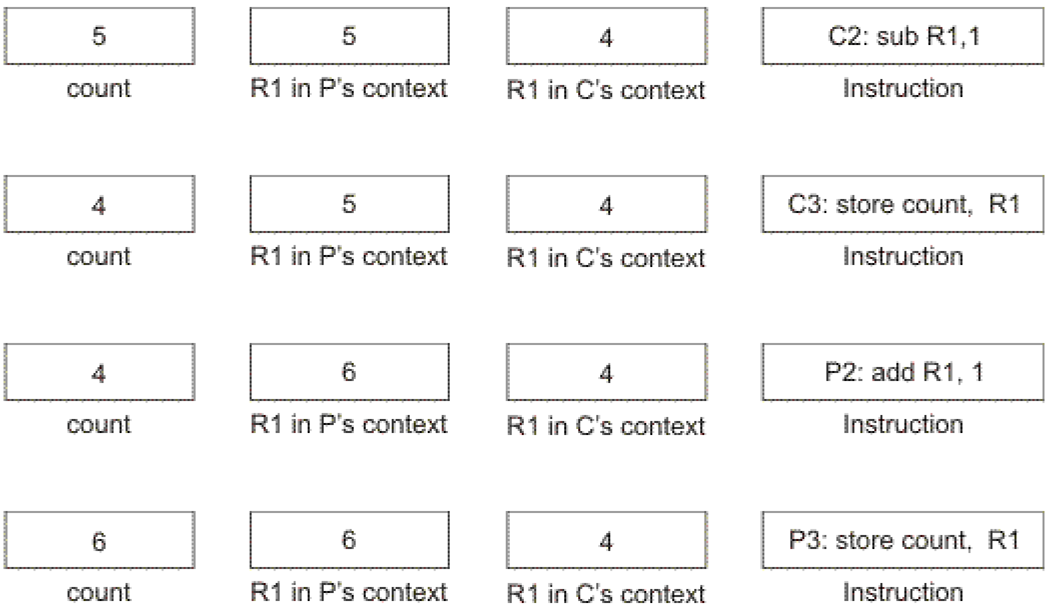
- The producer and consumer processes may be scheduled in any order and may be preempted.
- Consider the following sequence of statements
  - P1, *CSwitch*, C1, C2, C3, *CSwitch*, P2, P3.



Previous Next

A Possible Scenario

- The producer and consumer processes may be scheduled in any order and may be preempted.
- Consider the following sequence of statements
  - P1, *CSwitch*, C1, C2, C3, *CSwitch*, P2, P3.



Previous Next

Race Conditions

- Situation when several concurrent processes operate on the same variable and the result of the computation depends upon the order in which they execut.
- In the preceding example, the value of count could be 4, 5 or 6.
  - C1,P1,P2,P3,C2,C3 → final value 4
  - C1,C2,C3,P1,P2,P3 → final value 5
  - P1,C1,C2,C3,P2,P3 → final value 6

Critical Section Problem

- We model the problem using the notion of *Critical Sections*.
  - Critical sections are with respect to the shared data.
- There are *n* processes, each sharing some common resource.
  - Each wants to modify the common resource.
- For each process, define the region of code where it accesses a shared piece of data as a critical section.
- In a correct system of multiple cooperating processes,
  - Only one process must be inside its critical section.

Problem Abstraction

- Processes execute code similar to the following.

while (1) {
Non critical code
Entry code
Critical Section Code
Exit Code
Non Critical Code
}

Previous Next