Concurrency

Zhitao Gong

October 20, 2015

Outline

Topic

Concurrency Level

From lowest to highest

Instruction level executing two or more *machine instructions* simultaneously, more hardware related

Statement level executing two or more *high-level language statements* simultaneously

Unit level executing two or more *subprogram units* simultaneously

Program level executing two or more *programs* simultaneously

Concurrency Example – Dining Philosophers

- Philosopher either thinks or eats.
- Philosopher eats only when he has both forks.
- Philosopher can pick up a fork only when it is available.

The problem is to design a discipline of behavior, i.e., a concurrent algorithm, such that no philosopher will starve.

Hardware Support – Multiprocessor

Late 1950s	one general-purpose processor and one or more special-purpose processors for input and output operations
Early 1960s	multiple complete processors, used for program level concurrency
Mid 1960s	multiple partial processors, used for instruction level concurrency

Architecture Category

The *I*-nstruction and *D*-ata streams are *S*-ingle or *M*-ultiple.

SIMD Single-Instruction Multiple-Data, e.g., vector processors. All processors except the controller execute the same instructions. Support instruction level concurrency.

MIMD Multiple-Instruction Multiple-Data. Support unit level concurrency. May have *distributed* or *shared* memory.

MISD Multiple-Instruction Single-Data, e.g., pipeline architecture (instruction pipeline, graphics pipeline).

Design for Concurrency or Not

Why Not

Processor is faster

- Ever increasing clock rate
- Bultin Concurrency

Why Concurrency

- Processor speed increase limit
- Multiprocessors are favored
- Multiple processors on a single chip

Movivation for Concurrency

- Faster.
- Concurrency provides a different method of conceptualizing program solutions to problems, e.g., the dining philosopher problem, producer-consumer problem, sleeping barber problem, multiplayer online gaming, web server, browser, etc.

Concurrency Category

Physical concurrency Program units from the same program *literally* execute simultaneously assuming multiple processors.

Logical concurrency The actual execution of programs is taking place in *interleaved* fashion on a single processor.

Thread

Thread short for *thread of control*, is the sequence of program points reached as control flows through the program.

Programs that have coroutines but no concurrent subprograms, though they are sometimes called *quasi-concurrent*, have a single *thread of control*.

Concurrency vs Parallelism

Concurrency is when two tasks can start, run, and complete in *overlapping time periods*. It doesn't necessarily mean they'll ever both be running at the same instant. It is a property of program or system.

Parallelism is when tasks literally run at the same time, e.g., on a multicore processor. It is a run-time behavior.

Topic

Task

Task is a unit of a program that can be in concurrent execution with other units of the same program.
Different from subprograms:

- May be implicitly started.
- Caller need not wait for tasks to complete.
- Control may not return to caller.

Task Category

Heavyweight task executes in its own address space.

Lightweight task all run in the same address space.

Lightweight task is

- easier to implement and
- more efficient since less effort required to maintain

Task Communication

Task may need to communicate to *synchronize execution* and/or *share data* through

1. shared nonlocal variables,
2. message passing, or
3. parameters.

Synchronization

Synchronization (sync) is a mechanism that controls the order in which tasks execute.

Cooperation Sync A pauses and waits for B to complete. E.g., producer-consumer problem.

Competition Sync A and B visit the shared data location. E.g., race condition.

Competition Sync Case

We have two tasks

Task A `TOTAL = TOTAL + 1`

Task B `TOTAL = TOTAL * 2`

Suppose three steps are needed:

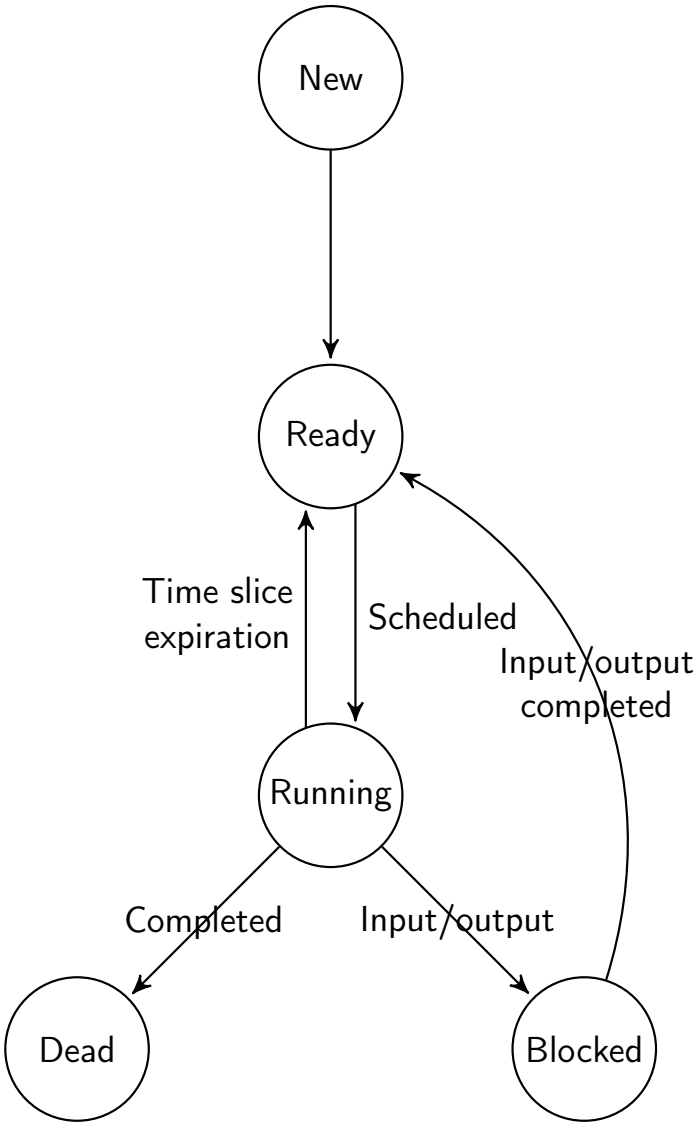
- Fetch operand
- Computer output
- Write back result

Mutually Exclusive Access

For shared resource, only one task is allowed to possess.

- Semaphore
- Monitor
- Message Passing

Task State



New Just created

Ready Awaiting execution on CPU

Running Being literally executed

Blocked Interrupted by events

Dead Finished or killed

Design Issues

- Cooperation and competition synchronization
- Task handling, creation, termination, scheduling, etc.

Topic

Guard

Guard is a linguistic device that allows the guarded code to be executed only when a specified condition is true.

Guard mechanism ensures

1. limited access to a data structure, by placing guards around the code that accesses the structure, and
2. all attempted executions of the guarded code eventually take place.

Semaphore

Dijkstra devised *semaphore* in 1965.
Semaphore is an implementation of *guard*.

Semaphore is a data structure that consists of an integer and a queue that stores task descriptors.

Task descriptor is a data structure that stores all of the relevant information about the execution state of a task.

Semaphore Operation

Essentially, two ATOMIC operations

- P *P*-robeer te verlagen, to try and decrease. When a process wants to use the resource, it performs a *P* operation: if this succeeds, it decrements the resouce count and the process continues; if all the resource is currently in use, the process has to wait.
- V *V*-erhogen, to increase. When a process is finished with the resource, it performs a V operation: if there were processes waiting on the resource, one of these is woken up; if there were no waiting processes, the semaphore is incremented indicating that there is now more of the resource free.

Producer-Consumer Naïve

Works but dangerous.

Producer()

```
1  while TRUE
2        item = produceItem()
3        if counter == BUFSIZE
4              sleep()
5        put item to buffer
6        counter = counter + 1
7        if counter == 1
8              wakeup(Consumer)
```

Consumer()

```
1  while TRUE
2        if counter == 0
3              sleep()
4        get and remove item from buffer
5        counter = counter − 1
6        if counter == BUFSIZE − 1
7              wakeup(Producer)
8        consumeItem(item)
```

Producer-Consumer with Semaphore

In case of cooperation sync with only one producer and consumer.

Producer()

```
1  while TRUE
2        item = produceItem()
3        P(emptyCounter)
4        put item to buffer
5        V(fullCounter)
```

Consumer()

```
1  while TRUE
2        P(fullCounter)
3        get and remove item from buffer
4        V(emptyCounter)
5        consumeItem(item)
```

Producer s -Consumer s with Semaphore

In case of competition sync with multiple producers and consumers.

Producer()

```
1  while TRUE
2        item = produceItem()
3        P(emptyCounter)
4        P(mutex)
5        put item to buffer
6        V(mutex)
7        V(fullCounter)
```

Consumer()

```
1  while TRUE
2        P(fullCounter)
3        P(mutex)
4        get and remove item from buffer
5        V(mutex)
6        V(emptyCounter)
7        consumeItem(item)
```

Comments on Semaphore

1. The semaphore is an elegant synchronization tool.
2. It is useful only for an ideal programmer who makes no mistakes.
3. Ideal programmers are rare.

Topic

Monitor

Monitor is a program unit that all synchronization operations on shared data are gathered into.

Monitor provides *competition synchronization* by transferring responsibility for synchronization to the run-time system from the programmer.

Buffer Monitor

Add(*item*)

```
1  while counter == BUFSIZE
2        wait(full)
3  put item to buffer
4  counter = counter + 1
5  if counter == 1
6        notify(empty)
```

Remove(*item*)

```
1  while counter == 0
2        wait(empty)
3  get and remove item from buffer
4  counter = counter − 1
5  if counter == BUFSIZE − 1
6        notify(full)
```

Producer-Consumer with Monitor

```
Producer()
1   while TRUE
2         item = produceItem()
3         buffer.add(item)
```

```
Consumer()
1   while TRUE
2         item = buffer.remove()
3         consumeItem(item)
```

Comments on Monitor

- Equally expressive as *semaphore*.
- Cooperation sync is still a problem.
- May be used to implement semaphore, vise versa.