MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT2100 Semester 2 2017

Lecture 4 (Part B): Concurrency (Part 1)

(Reading: Stallings, Chapter 5)

Jojo Wong

GROUP OF EIGHT AUSTRALIA

Lecture 4 (Part 2): Learning Outcomes

- Upon the completion of this lecture, you should be able to:
  - Discuss the basic concepts related to concurrency
  - Understand the concept of race condition
  - Describe the mutual exclusion requirements

Why concurrency is important in supporting multiprocessing?

Multiple Processes

- OS design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed processing

mgt of multiple processes within a uniprocessor system

mgt of multiple processes within a multi-processor system

mgt of multiple processes on a multiple distributed computer system

Concurrency: Three Different Contexts

Multiple Applications

invented to allow processing time to be shared among active applications

Structured Applications

extension of modular design and structured programming

Operating System Structure

OS themselves implemented as a set of processes or threads

Concurrency: Terminology

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Concurrency: Principles

- **Uniprocessor**: the relative speed of execution of processes cannot be predicted
  - depends on activities of other processes
  - the way OS handles interrupts
  - scheduling policies of the OS
- **Multiprocessor**: *interleaving* and *overlapping*
  - can be viewed as examples of concurrent processing
  - both present the same problems

Concurrency: Difficulties

- Sharing of global resources

race condition

- Difficult for OS to manage the allocation of resources optimally

deadlock

- Difficult to locate programming errors — results are not deterministic and reproducible

What are the concurrency problems?

Concurrency: Problems

- Concurrent access to shared data may result in data inconsistency — **Race Condition**
- A problem exists in multiprogramming on uni- and multi-processors
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating process

Race Condition

- Occurs when multiple processes or threads read and write data items
- Final result depends on the order of execution
  - "**Loser**" of the race is the process that updates last and will determine the final value of the variable
- To prevent race conditions, concurrent processes must be synchronised

Concurrency Problem: Uniprocessor Multiprogramming

- Consider the following procedure:

```
void echo()
{
    char-in = getchar();
    char-out = char-in;
    putchar(char-out);
}
```

Multiprogramming with uniprocessor

- Read a character from the keyboard and store in `char-in`.
- Transfer to `char-out` before being sent for display.
- Consider two different applications — P1 and P2 — make a call to this procedure.

Concurrency Problem: Uniprocessor Multiprogramming

- Consider the following procedure:

```
void echo()
{
    char-in = getchar();
    char-out = char-in;
    putchar(char-out);
}
```

Multiprogramming with uniprocessor

- P1 invokes **`echo()`** and is interrupted immediately after **`getchar`** returns its value and stores it in **`char-in`** (e.g. **`x`**).
- P2 is activated and invokes **`echo()`** and read a char (e.g. **`y`**) and runs to completion of the procedure.
- When P1 resumes the value of **`x`** has been overwritten in **`char-in`** by process P2 and therefore its value **`x`** is lost.

Concurrency Problem: Uniprocessor Multiprogramming

- Process P1:

```
char-in = getchar();



char-out = char-in;
putchar(char-out);
```

- Process P2:

```

char-in = getchar();
char-out = char-in;
putchar(char-out);
```

TIME

Concurrency Problem: Uniprocessor Multiprogramming

- Consider the following procedure:

```
void echo()
{
    char-in = getchar();
    char-out = char-in;
    putchar(char-out);
}
```

Multiprogramming with uniprocessor

- Assume only one process at a time to invoke and be in the **`echo`** procedure.
- P1 invokes **`echo()`** and is interrupted immediately after **`getchar`** returns its value and stores it in **`char-in`** (**`x`**).
- P2 is activated and invokes **`echo()`**.
- But since P1 is still inside the procedure, and currently suspended — **P2 is blocked** from entering the procedure.

Concurrency Problem (Summary):

- P1 invokes the **`echo`** procedure and is interrupted immediately after **`getchar`** returns its value and stores it in **`char-in`** (e.g. **`x`**).
- P2 is activated and invokes **`echo`** procedure and since the **`echo`** procedure is used by process P1, P2 is blocked from further execution.
- At some later time, P1 is resumed and completes the execution of **`echo`** and the proper input character will be displayed.
- When P1 exits **`echo`**, this removes the block on P2.
- When P2 is later resumed, the **`echo`** procedure is successfully invoked.

How about concurrency problems with multiprocessors?

Concurrency Problems: Multiprocessor Multiprogramming

- Same problem arises even when the processes — P1 and P2 — runs on different processors accessing unprotected shared variables.
- The solution outlined in the previous slides can work here.
- Protecting and controlling access to shared resources are critical.

Question: Race Condition

- Assume P1 and P2 share two variables **`a`** and **`b`** with initial values of **`a`** = 1 and **`b`** = 2
- P1 executes the statement: **`a = a + b`**
- P2 executes the statement: **`b = a + b`**
- What values are **`a`** and **`b`** if P1 executes before P2?
- What values are **`a`** and **`b`** if P2 executes before P1?

What are the responsibilities of OS?

Operating System: Concerns

- be able to keep track of various processes
- allocate and de-allocate resources for each active process
- protect the data and physical resources of each process against interference by other processes
- ensure that the processes and outputs are independent of the processing speed

How do processes interact with each other?

Types of Process Interaction

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

What are the control problems with concurrent processes?

Control Problems

- Concurrent processes come into conflict when they are competing for the same system resource — I/O devices, memory, processor time, etc.

In the case of competing processes three control problems must be faced:

- **mutual exclusion**
- **deadlock**
- **starvation**

Mutual Exclusion

- Suppose `n` processes all competing to use some shared data.
- Each process has a code segment — **critical section** — where the shared data is accessed or manipulated.
- Ensure that when one process is executing in its critical section, no other process is allowed in its critical section.

Mutual Exclusion: Example

To enforce mutual exclusion, two functions are provided — entercritical/ exitcritical with the resource (Ra) as the argument

```
/* PROCESS 1 */

void P1
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 */

void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

. . .

```
/* PROCESS n */

void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

Multiple Shared Data Resources

- The same problem exists even when processes access more than one shared resource.
- Processes must cooperate to ensure the shared data are properly managed.
- Control mechanisms are needed to ensure the integrity of the shared data.

Multiple Shared Data Resources: Example

- **P1**

```
a = a + 1 ;
b = b + 1 ;
```

- **P2**

```
b = 2 * b ;
a = 2 * a ;
```

- Assuming that `a = b` at the beginning, and consider the following concurrent execution sequence:

```
a = a + 1 ;   /* {P1}  */
b = 2 * b ;   /* {P2}  */
b = b + 1 ;   /* {P1}  */
a = 2 * a ;   /* {P2}  */
```

- At the end of this execution, the condition `a = b` ***no longer holds!***

Mutual Exclusion: Requirements

- Mutual Exclusion must be enforced.
- A process that halts must do so without interfering with other processes — **no deadlock or starvation**.
- A process must not be denied access to a critical section when there is no other process is using the shared resources (being manipulated by the critical section code).
- No assumptions are made about relative process speeds or the number of processors.
- A process remains inside its critical section for a finite time only.

Additional Control Problems

- **Deadlock**: two or more processes are waiting indefinitely for the other processes to release the system resources.

- **Starvation**: indefinite blocking of a process.

Summary of Lecture 4 (Part 2)

- Concurrency is the fundamental concern in supporting multiprogramming, multiprocessing, and distributed processing.
- Mutual exclusion is the condition where there is a set of concurrent processes — only one of which is able to access a given resource or perform a given function at at any time.

Next week: Concurrency mechanisms, deadlocks and starvations.