Sep 18, 2006
Lecture 4: Process Synchronization

September 5, 2007

1 ADMIN

After Chapter 4 (Processes), please continue with reading Chapter 7 on Process Synchronization as a natural follow up. We will return to Chapters 5 and 6 later.

2 Review

- Q: What is the other command that seems to go hand-in-hand with fork()?

  A: exec* (7 variants of this command)

- Q: Suppose you have a Makefile that is ready made. How can you determine what actions you can take in this Makefile?

  A: Look at the "targets" in the Makefile. These are the list of names that begin a colon (":") terminated line. E.g. a line in the file

  action1 hw1:

  says that "action1" and "hw1" are names for the same target. You execute these actions by typing

  > make action1

  or

  > make hw1

- Q: Give two reasons why keyboard editors are more efficient and more productive than WYSIWYG editors, and one reason why people like WYSIWYG.

  A: Here are two reasons for superiority of keyboards:

  (1) Keyboard editors are more powerful because you can compose sophisticated and repetitive commands using the keyboard. With WYSIWYG, this ability is limited.

  E.g., You cannot tell WYSIWYG to "search the next 100 lines of file, and replace each occurrence of the word "GOOD" to "BAD".

  But in (say) vi/vim/gvim, you type this sequence:

  :.,+100s/GOOD/BAD/g

(2) WYSIWYG editors requires hand-eye coordination. With keyboard editors, you can can even look elsewhere if you know how to touch type!

Here is a reason why people seem to like WYSIWYG:

The learning curve for WYSIWYG is low – you can pick it up intuitively. The learning curve for keyboard editors is steep. But the payoff is more than worth the effort.

By the way, I think gvim is superior to emacs because you generally can do things with much fewer key strokes (because many basic facts about text files are built into the editor).

3 Inter Process Communication (IPC)

- One of the goals of an OS is to run a set of processes in such a way that each THINKS it has exclusive use of the CPU and resources!

  In other words, we seek to ISOLATE processes from each other. So, when we discuss IPC, we are in essense, backtracking a little.

- Why we need IPC:

  – parent-child communication (in UNIX, these are accomplished through "pipes")

  – non-interference of each other (coordination of shared resources)

  – proper sequencing of actions across processes (perhaps the processes are working together to accomplish a common task)

  – shared resource (memory or variables, files, devices)

  – E.g., print spooler

  To print a file, a process enters the name in the spooler directory. A printer daemon periodically checks this directory, prints the file, and removes the entry from spooler directory.

- Q: What is a "race condition"?

  A: When the outcome of two interacting processes depends on the speed or order in which the processes execute commands, but the outcome ought not to depend on such orders of execution.

- Q: Give an example.

  A: Let P and Q be processes that are both accessing a variable "Balance", initialized to 100.

  o P: Balance++

  o Q: Balance–

  o Intuitively, after both processes are done, Balance=100.

  o But we might get 101 or 99. HOW CAN THIS HAPPEN? HINT: the commands "Balance++" and "Balance–" are non-atomic.

  o ANSWER: Balance++ is really (X=Balance; X++; Balance=X) or alternatively (READ(X,Balance); INC(X); WRITE(Balance,X)) and X is a local variable. Similarly for Balance–.

  Now imagine that the two sequences Balance++, Balance– of atomic actions are interleaved...

- SOLUTION STRATEGY:

  We must prevent the interleaving of certain code fragments. This is called the **mutual exclusion problem** (MUTEX problem). The code fragments which must be "atomic" are called **critical sections**.

  We can associate each critical section with a variable (called a resource or semaphore). When a process is inside the critical section, we say it is accessing that resource/semaphore. Following Dijkstra, we write P(S), V(S) for accessing and releasing the semaphore S.

- PROPERTIES:

  1. [MUTEX] No two processes may be simultaneously access a resource.

  2. [SPEED] The speeds and number of processes are arbitrary but, equality important, the speed of each process is non-zero.

  3. [NONBLOCK] Processes outside a critical section may not block other processes.

  4. [FAIRNESS] Each process wanting to access a resouce will eventually be able to do so.

  Remark that Fairness is stronger than asserting "no deadlock". A scenario is where a fast process keeps getting to its critical section (hence no deadlock), but a slower process is repeatedly denied access (lockout). Stronger and weaker forms of FAIRNESS (4.) have been proposed. A weaker form is this: it is fair as long as each process makes progress.

- NO INTERRUPT SOLUTION: Turn off interrupts.

  We can allow the user programs to turn off interrupts, but this is clearly dangerous.

  We can restrict this ability only to system processes. But this will not help user processes achieve mutual exclusion (e.g., the Balance update problem).

  Also, this would not help in multi-CPU situations.

- SOME WRONG SOLUTIONS:

```
Initially P1wants = P2wants = false

P1:                                     P2:

Loop forever                            Loop forever
    P1wants <-- true          ENTRY         P2wants <-- true
                              NO MAN ZONE
    while (P2wants) {}                      while (P1wants) {}

    critical-section                        critical-section

    P1wants <-- false         EXIT          P2wants <-- false

    non-critical-section                    non-critical-section
```

  WHAT IS wrong? Can get stuck if both P1 and P2 enters the NO MAN ZONE at the same time. Neither can go forward!

  The trouble was that setting "want" before the loop. Try again:

```
Initially turn=1

P1:                                P2:

Loop forever                       Loop forever

    while (turn = 2) {}                 while (turn = 1) {}

    critical-section                    critical-section

    turn <-- 2                          turn <-- 1

    non-critical-section           non-critical-section
```

This one forces alternation – no process can enter its CS twice in a row! This is unfair for a fast process.

Specifically, it fails condition 3: a process in its NCS can stop another process from entering CS.

Many earlier "solutions" were found and several were published. The first correct solution was from Dekker (1964). It is clever, but complicated. Subsequent solutions with better fairness properties were found (e.g., no task has to wait for another task to enter the CS twice).

- Peterson's Solution (1981):

  When first published, it's simplicity was a surprise.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Initially P1wants=P2wants=false (Important!)
          and  turn=1 (Unimportant)

P1:                                  P2:

Loop forever                         Loop forever

    P1wants <-- true                     P2wants <-- true

    turn <-- 2                           turn <-- 1

    while (P2wants and turn=2) {}        while (P1wants and turn=1) {}

    critical-section                     critical-section

    P1wants <-- false                    P2wants <-- false

    non-critical-section                 non-critical-section
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

  JUSTIFICATION:

  Why is MUTEX achieved? Consider P1 (the argument is symmetrical with P2).

  0. Process P1 enters its C.S. iff (P2wants=false OR turn=1).

  1. If P2wants=false, it is surely safe. (Because P2wants=true throughout the moment P2 decides to enter the C.S.)

  2. If turn=1, it is also safe because it meant the last process to set turn was P2, and once P2 has set turn to 1, P2 will be waiting for P1 to reset it to 2 or to P1want=false. Since turn=1 and P1want=true, it it meant that P2 will be waiting.

  VERIFY THE REMAINING 3 PROPERTIES WE NEED.

  REMARK:
  1. This extends to any number of processes. See Operating Systems Review Jan 1990, pp. 18-22. See Tanenbaum p.106.
  2. This solution requires each process to know a unique value (e.g., its PID).
  3. Note that Semaphores (e.g., P1want, turn) are GLOBAL variables in a very strong sense, in the system-wide sense! Such variables are controlled by the OS. Normal "global variables" in processes are local to each process (they are only outside all "scopes" of that process).

- HARDWARE ASSISTED SOLUTION:

  (test and set lock)

  TSL (Reg, LockVar)

It copies LockVar into register Reg, and stores "1" (or any non-zero value) into LockVar. All this is ATOMIC.

ALTERNATIVE:

TAS (boolVar)

ATOMICALLY sets boolVar to TRUE and returns the OLD value of boolVar.

Now implementing a critical section for any number of processes is trivial.

```
%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%
while (S=closed) {}
S<--closed              //This is outside the body of the while
%%%%%%%%%%%%%%%%%%%%%%
```

where finding `S=open` and setting `S<--closed` is atomic

* That is, wait until the gate is open, then run through and atomically close the gate

* Said another way, it is not possible for two processes doing P(S) simultaneously to both see S=open (unless a V(S) is also simultaneous with both of them).

* V(S) is simply `S<--open`

The above code is not real, i.e., it is not an implementation of P. It is, instead, a definition of the effect P is to have.

To repeat: for any number of processes, the critical section problem can be solved by

```
%%%%%%%%%%%%%%%%%%%%%%
loop forever
    P(S)
    CS
    V(S)
    NCS
%%%%%%%%%%%%%%%%%%%%%%
```

The only specific solution we have seen for an arbitrary number of processes is the one just above with P(S) implemented via test and set.

Remark:
a. Peterson's solution requires shared variables across processes (one alternative is to use shared files)
b. Peterson's solution requires each process to know its processor number. The TAS soluton does not. Moreover the definition of P and V does not permit use of the processor number.

- Dekker's Solution. The first correct solution to MUTEX was from Dekker, which we present here to indicate its subtlety and to contrast with the simplicity of Peterson's solutions.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     Initially:
               P1wants=P2wants=false (Important!)
               turn=1                (Unimportant)
     ------------------------------------------------------------------
     P1:                               | P2:
                                       |
     Loop forever                      | Loop forever
                                       |
P:       P1wants <-- true              |     P2wants <-- true
                                       |
A:       while (P2wants) {             |     while (P1wants) {
           if (turn=2) {               |        if (turn=1) {
               P1wants <-- false       |            P2wants <-- false
B:             while (turn=2) {}       |            while (turn=1) {}
               P1wants <-- true        |            P2wants <-- true
           }                           |        }
         }                             |     }
                                       |
C:       critical-section              |     critical-section
                                       |
V:       P1wants <-- false             |     P2wants <-- false
```

```
         turn <-- 2                    |     turn <-- 1
                                       |
D:       non-critical-section          |     non-critical-section
                                       |
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Let us notice some salient features of this code. There are six interesting labeled "points": Points C and D correspond to the CS (critical section) and non-CS parts of the code. Points P and V correspond to the usual protocols we execute before and after a CS. Finally, there is a doubly nested while-loop, where the "outer-loop" is marked by point A and the "inner-loop" by point B.

Let us now show that Dekker's solution satisfies the 3 requirements of the text:

1. Mutual Exclusion: We discuss process P1, but the situation is entirely symmetrical with P2. It is important to exploit this symmetry.

1. First observe that "P1want" is only set by process P1. But "turn" is set by both processes.
2. When P1 exits from its outer-loop, but before it finishes its critical section (CS), we have the condition [P2wants=false]. However, "turn" can be 1 or 2.
3. Now [P2wants=false] can mean P2 is either (I) inside the if-statement of the outer-loop. or (II) between points V and D of its code. In other words, P2 is NOT in its critical section.
4. Can P2 move into its own CS as P1 proceeds to its CS? There are two possibilities.
   CASE (I): In this case, P2 in inside the if-statement of the outer-loop. Hence turn=1. But until P1 changed "turn" to 2, we will have turn=1. Until then, P2 will be stuck at point B. Thus P1 can safely proceed to its CS.
   CASE (II): In this case, P2 has to pass through its non-CS before reaching its CS again. But when P2 reaches point P, it will encounter P1wants=true. If turn=1, it will be stuck at point B (as in CASE (I)). If turn=2, it will be stuck in the outer-loop. It can only get out of these loops after P1 has safely gone through its CS.
5. We conclude that when P1 is in its CS, it is safe.

2. Progress: a process that is not trying to enter the CS should not hold up other processes.

1. To be precise, we say that process P1 is not trying to enter the CS when it is in its non-CS (point D in the code).
2. We must show that P2 will be able to get through both while-loops in finite time as long as P1 remains in its non-CS.
3. Notice that just before P1 enters its non-CS, it has (P1wants=false and turn=2). Now "turn" might be later changed to 1 by P2, but P1wants=false will not change as long as P1 is in non-CS.
4. We must prove that P2 will be able to enter its CS if it chooses to. Can P2 get stuck in its outer-loop? This is clearly impossible since P1wants=false.
5. But can P2 get stuck in its inner-loop? (Remember that because we make no assumptions about relative speeds of the 2 processes, P2 might be inside its outer-loop when P1 is in its non-CS).
6. But if P2 is inside its inner-loop, it means that turn=1. That means that "turn" has changed by P2 from 2 to 1 while P1 was in its non-CS. This meant that P2 has gone from point D to point B. But that is impossible as P1wants=false throughout this time. P1wants=false means P2 will never ever its outer-loop.

3. Bounded Waiting: we show a very strong version of this property – that P2 cannot get to its CS twice in a row while P1 is waiting to enter its CS

1. Suppose P1 waited for P2 twice-in-a-row in its outer-loop. THIS IS TRICKY TO STATE PRECISELY SINCE WE DO NOT ASSUME ANYTHING ABOUT RELATIVE SPEEDS OF PROCESSES. SO WHAT DO WE MEAN? It means there are two time instances, $t_1 < t_2$, such that the outer-loop of P1 found P2wants=true. But P2 has gone through its CS twice in the period $[t_1, t_2]$.

2. To show a contradiction, note that P1wants=true during $[t_1, t_2]$. Hence, when P2 goes from its CS back to its CS again, it would have gotten inside its outer-loop. Moreover, turn=1 (set by P2 itself). This meant that P1 would have been stuck inside its inner-loop. This contradicts our assumption that P2 proceeded to its CS the second time.
3. Now let us consider the other possibility: Suppose P1 waited for P2 twice-in-a-row in its inner-loop. WHAT DO WE MEAN? It means there are two time instances, $t_1 < t_2$, such that the inner-loop of P1 found turn=2. But P2 has gone through its CS twice in the period $[t_1, t_2]$.
4. After P2 goes through its CS the first time, it would set turn=1 at some time $t_3 \in (t_1, t_2)$. That means "turn" must be reset to 2 again in the period $[t_3, t_2]$. This is impossible since only P1 can reset "turn" but P1 was assumed to be inside the inner-loop.