Lecture 4: Inter-Process Communication (Jan 27, 2005)

February 17, 2005

1 Review

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

  A: exec* (6 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).

2 Inter Process Communication

- Why we need IPC:

  – pipes, parent-child communication,

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

  – proper sequencing of actions across processes

  – 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.

- 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. HINT: the commands "Balance++" and "Balance–" are non-atomic.

- SOLUTION STRATEGY:

  We must prevent the interleaving of certain code fragments. This is called the **mutual exclusion 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.

- PROPERTIES:

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

  2. [SPEED] The speeds and number of processes are arbitrary.

  3. [NONBLOCK] Processes outside a critical section may block other processes.

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

  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  and  turn=1

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:

1. If at least one of them does not "want", it is safe!

2. So the danger is when both "wants". In that case, the last process to set "turn" goes first!

VERIFY PROPERTIES 1-4.

REMARKS: This extends to any number of processes. See Operating Systems Review Jan 1990, pp. 18-22. See Tanenbaum p.106. Also, each process MUST know a unique value (e.g., its PID).

- 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.

```
loop forever
    while (TAS(boolVar)) {}
    CS
    boolVar<--false
    NCS
```

- Sleep and Wakeup

  THE ABOVE solutions requires busy-waiting.

  WHAT COULD GO WRONG: Process H and L with HIGH and LOW priorities. After L enters critical region, H is busy waiting for L. Now both are stuck.

  To solve this, two new system calls:

  – SLEEP() causes process to block.

  – WAKEUP(pid) causes process pid to unblock.

- Producer-Consumer Problem.

  Example: Producer=ttyin() keyboard interrupt service routine, Consumer=getchar().

Example: Producer=ttyout() serial interface transmit interrupt, Consumer=putchar().

[Source: http://www-rohan.sdsu.edu/faculty/gleonard/html/570/lecture15.html]

Buffer of size N.

Count variable, 0¡=COUNT¡=N

```
Producer:
    loop forever:
       item= produce();
       if (COUNT == N) SLEEP();
       insert(item)
       COUNT++
       if (COUNT == 1) WAKEUP(Consumer)

Consumer:
    loop forever:
       if (COUNT == 0) SLEEP();
       item=remove(item)
       COUNT--
       if (COUNT == N-1) WAKEUP(Producer)
       consume(item)
```

WHAT ARE RACE CONDITIONS HERE?

Consumer reads COUNT=0

- P and V and Semaphores:

  DICTIONARY: Semaphore – any system of signaling (e.g., with lights or flags).

  HERE, semaphore is just a special variable (representing a resource).

  Note: Tanenbaum does both busy waiting (like above) and blocking (process switching) solutions. We will only do busy waiting.

  The entry code is often called P and the exit code V (Tanenbaum only uses P and V for blocking, but we use it for busy waiting). So the critical section problem is to write P and V so that

```
loop forever
    P
    critical-section
    V
    non-critical-section
```

  satisfies our list of requirements

  1. Mutual exclusion

2. No speed assumptions

3. No blocking by processes in NCS

4'. Reasonable Progress (weaker form of above condition 4)

A binary semaphore abstracts the TAS solution we gave for the critical section problem.

* A binary semaphore S takes on two possible values "open" and "closed"

* Two operations are supported

* P(S) is

```
        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.