PROCESS SYNCHRONIZATION I

Ruben Acuña

Spring 2019

2 BACKGROUND (5.1)

THE PROBLEM OF CONCURRENCY

- The case of a concurrent system is not so interesting if there is no cooperation between processes/threads – everything will just work.
- Instead, let's look at the more interesting case posed in the producer-consumer problem.
- In essence, we have two programs, which could be at any point in their operation, that must perform no operations that interferes with the other.

```
//counter, buffer, in, out are all shared
while (true) {
        // produce next_produced
        while (counter == BUFFER_SIZE);

        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}

while (true) {
        while (counter == 0);

        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        // consume next_consumed
}
```

PROGRAM SLICES

- Consider first that we breakdown C operations that normally are compiled to multiple instructions into those instructions.

```
while (true) {
    /* produce next_produced */
    while (counter == BUFFER_SIZE);

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    asm {  //counter++;
        reg = counter;
        reg = reg + 1;
        counter = reg;
    }
}
```

```
while (true) {
    while (counter == 0);

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    asm { //counter--;
        reg = counter;
        reg = reg – 1;
        counter = reg;
    }
    /* consume next_consumed */
}
```

- The issue: any "slice" across these programs may interact.
- Thinking about the regions marked with asm, is there any way these two programs could get an inconsistent view of shared data?

RACE CONDITIONS

- The issue here is that both threads of execution may read counter into a register. Then one finishes updating counter, and later other finishes and ends up replacing that value.
- We call this a *race condition* – the order of execution (which is not guaranteed!) impacts the state of the program afterward.

6

THE CRITICAL-SECTION PROBLEM (5.2)

DEFINITION

- Consider some program P. The program is divided into two sections: *critical* and *remainder*.

```
//start - critical section
//code here
//end - critical section

//start remainder
//code here
//end remainder
```

- For a set of programs, PS, the critical-section problem is to execute each PϵPS concurrently such that at no time are critical sections executing concurrently.
- A simple solution is to assume that critical sections are executed in kernel mode.
  - We can consider a *non-preemptive kernel*, where no context switching is allowed for a process in kernel mode - then the critical section will execute in one shot. Likewise, a *preemptive kernel* would allow a context switch (which doesn't address our problem).

PROPERTIES OF A SOLUTION

- We will be looking at several solutions to this problem – how do we judge their effectiveness?

- We need to argue for each solution, that we have:
  - *Mutual Exclusion*: At no time should any two processes be executing code in their critical region.
  - *Progress*: If no processes are in a critical section and some other process is ready to enter the section, then it will be entered.
  - *Bounded Waiting Time*: Once a process needs to execute a critical section, the waiting time (in terms of other processes) is bounded by the number of concurrent processes.

9

PETERSON'S SOLUTION (5.3)

AN ALGORITHM

- Peterson's algorithm is method to solve the critical section problem. (Assuming that load/stores are atomic.)
- The algorithm can run identically for two processes P_i and P_j.
- The idea is to have a variable (turn) which selects which process should go next, and an array (flag) that indicates when a process is ready to enter it's critical section.

```
//shared memory
int turn = 0;
bool flag[2] = { false, false };


//for some process i
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    //critical section
    flag[i] = false;
    //remainder section
} while (true);
```

TRACE

```
//P0
do {                                          //P0L1
    flag[0] = true;                           //P0L2
    turn = 1;                                 //P0L3
    while (flag[1] && turn == 1);             //P0L4
    printf("P0: CS"); //critical              //P0L5
    flag[0] = false;                          //P0L6
} while (true);                               //P0L7
```

```
//P1
do {                                          //P1L1
    flag[1] = true;                           //P1L2
    turn = 0;                                 //P1L3
    while (flag[0] && turn == 0);             //P1L4
    printf("P1: CS"); //critical              //P1L5
    flag[1] = false;                          //P1L6
} while (true);                               //P1L7
```

Initial State:

- turn=0
- flag = {false, false}

Run P0L1-P0L3:

- turn=1
- flag = {true, false}

Run P1L1-P1L3:

- turn = 0
- flag = {true, true}

Run P1L4:

- Same; loop.

Run P0L4:

- Same; next instruction.

Run P1L4:

- Same; loop.

- Run P0L5:
  - Prints "P1: CS"

Run P1L4:

- Same; loop.

PROOF OF CORRECTNESS

- Mutual Exclusion: Assume P0 is in critical section. Then flag[1] == 0 or turn == 0. We must show that if either of these applies, then the premise holds.
  - If flag[1] == 0, then P1 cannot be in execution since flag[1] == 1 for duration of P1's critical section, the first condition holds.
  - If turn == 0, then P1 cannot be in critical section since a precondition is turn == 0.
- Progress: Assume P0 is waiting for critical section and P1 is either waiting or in it's remainder section. (We want to show that some P will enter it's critical section.)
  - If P1 is in remainder section, then flag[1]=0, and so P0 will exit the loop.
  - If P1 is waiting, then flag[1]=1 and turn=0, and so P0 will exit the loop.
- Bounded Waiting Time: Assume P0 is waiting to execute critical section since P1 is in it's critical section. Then flag[0]=1, flag[1]=1, and turn = 1. Once P1 completes, turn must be set to 0 (regardless of flag[1]) which blocks P2 and causes P1 to enter. Hence waiting time is always 1.

```
//P0
do {
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn == 1);
    //critical section
    flag[0] = false;
} while (true);


//P1
do {
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn == 0);
    //critical section
    flag[1] = false;
} while (true);
```

13

SYNCHRONIZATION HARDWARE (5.4)

TEST_AND_SET

- Two initial ideas:
  - *Locks* – a way to limit access to a resource.
  - *Atomic* – execute multiple instructions as a unit.

- We'll look at two atomic hardware calls.

- Problem: when doing an assignment into a variable, it's hard to tell the exact state that is being replaced. (Recall the issue of load/inc/store from the critical section problem.)
- Solution: provide a function that retrieves a variable's value, and sets it equal to true into a single atomic operation:
  - boolean test_and_set(boolean* target)

```
//mutual exclusion example

//shared data
boolean lock = false;

do {
    while (test_and_set(&lock));
    // critical section
    lock = false;
    // remainder section
} while (true);
```

COMPARE_AND_SWAP (CAS)

- The issue: say we have an assignment guarded by an if-statement check on that variable. Normally, there is a chance that the value of the variable will change between the condition and the assignment. This violates the "if" logic.
- Solution: combine comparison and setting a value on equality into a single atomic operation:
  - int compare_and_swap(int* value, int expected, int new_value)
    - (Note: book's implementation returns the initial stored value no matter what.)

```
//mutual exclusion example

//shared data
int lock = 0;

do {
    while (compare_and_swap(&lock, 0, 1)
                != 0);
    // critical section
    lock = 0;
    // remainder section
} while (true);
```

MUTEX LOCKS (5.5)

MUTUALLY EXCLUSIVE LOCKS

- The hardware level commands are okay, but a little decoupled from the problems we are typically trying to solve. One abstraction is based on implementing two functions:
  - Acquire(lock) – blocks until a lock (resource) is available and acquires it.
  - Release(lock) – releases a lock.
  - Both should be atomic operations!

- How might this be used to solve the critical section problem?

- As a side note: what should the process do while waiting to acquire a lock? Previously, we just used a while loop… this means we are using CPU cycles really for nothing: *busy waiting*.