PROCESS SYNCHRONIZATION

Ruben Acuña

Spring 2018

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.

```c
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;
    }
}
```

```c
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 P∊PS 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.

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);
```

16 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*.

18 SEMAPHORES (5.6)

SEMAPHORES

- Generalization over mutex to support more than one resource.
  - A more mutex is about excluding an action (running code in a second process) while a semaphore is about acquiring some unique resource from a set.
  - Semaphores end up working a little like malloc/free.
- Let some integer S be called a semaphore value, and define two operations:
  - Wait(): blocks until S is positive
  - Signal(): increases S by 1.
  - Both should be atomic operations.

```
wait(S) {
        while (S <= 0);
        S--;
}

signal(S) {
        S++;
}
```

USAGE

- Two samples:
  - Solving the critical-section problem.
  - Enforcing execution order.

```c
//critical section problem

//shared data
semaphore lock = 1;

do {
    wait(&lock);
    // critical section
    sign(&lock);
    // remainder section
} while (true);
```

What would happen if this was 0?

```c
//enforcing execution order

//shared data
semaphore lock = 0;

void proc1() {
    printf("line 1");
    signal(&lock);
}

void proc2() {
    wait(&lock);
    printf("line 2");
}
```

IMPLEMENTATION

- Say we want to implement a semaphore S where S ϵ I, and want to avoid busy waiting.
- The solution is relatively simple, use a list to store process and block them instead of looping.

```
typedef struct {
          int value;
          struct process *list;
} semaphore;

wait(semaphore* S) {
    S->value--;
    if (S->value < 0) {
        //add this process to S->list
        block();
    }
}


signal(semaphore* S) {
    S->value++;
    if (S->value <= 0) {
        //remove a process P from S->list
        wakeup(P);
    }
}
```

TERMINOLOGY

- Consider the case of processes P_1, and P_2. Both of these processes need access to resources controlled by semaphores S_1, and S_2. If the processes lock a (different) semaphore in parallel, then both will be unable to lock the second semaphore and go into *deadlock*.

- We also use the term *starvation* to refer to a process that gets "stuck" on a semaphore and cannot proceed.

- Consider the case of a some process P_1, that requires a resource used by P_2. Let P_1 be of a much higher priority than P_2. Then, P_1 may wait because P_2 isn't given a chance to execute and release the resource. This is called *Priority Inversion*. Inversion in the sense that the lower priority process should now be considered higher priority.
- One solution is *priority-inheritance*, which basically says if a resource is needed by a high priority process, then other users of that resource should inherit the high priority.

23

CLASSIC PROBLEMS OF SYNCHRONIZATION (5.7)

THE BOUNDED BUFFER PROBLEM

- Let's try rephrasing the producer-consumer problem (over a bounded buffer), using semaphores.
- Going to have four variables:

```
//shared data
int n;
//also a buffer data structure
semaphore buf_mutex = 1;
semaphore empty = n;
semaphore full = 0
```

```
//producer
do {
        // produce next_produced
        wait(empty);
        wait(buf_mutex);
        // add next_produced to the buffer
        signal(buf_mutex);
        signal(full);
} while (true);
```

```
//consumer
do{
        wait(full);
        wait(buf_mutex);
        // move from buffer to next_consumed
        signal(buf_mutex);
        signal(empty);
        // consume next_consumed
} while (true);
```

THE READER-WRITERS PROBLEM

- New problem: considering a single source of data that multiple threads may read or write into. Multiple reads are fine, but when a write is happening, the data should be constant.
- There are a couple of design approaches – here we'll prioritize the readers of the writers. (Means that writers may starve).
- Need three variables:

```
//shared data
semaphore rw_mutex = 1;
semaphore rc_mutex = 1;
int read_count = 0;
```

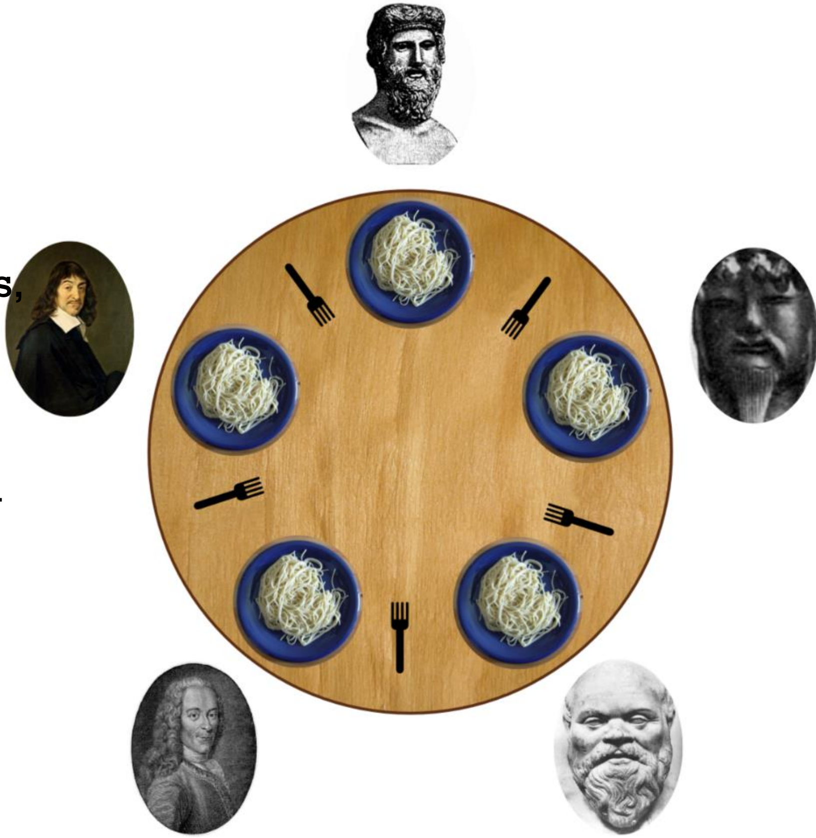
```
//writer process
do {
    wait(rw_mutex);
    /* writing is performed */
    signal(rw_mutex);
} while (true);

//reader process
do {
    wait(rc_mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(rc_mutex);
    /* reading is performed */
    wait(rc_mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(rc_mutex);
} while (true);
```

THE DINING PHILOSOPHERS

- Consider the following scenario: you have 5 philosophers sitting around a table, on the table are five plates of food, and five chopsticks.
- Occasionally, a philosopher gets hungry, in which case they must acquire two chopsticks, eat, and then puts those chopsticks down.
- Clearly there aren't enough chopsticks for everyone to eat at once…
- How well would the following solution fair in feeding philosophers?

```
//Philosopher i
do {
    wait(chopStick[i]);
    wait(chopStick[(i + 1) % 5]);
    // eat
    signal(chopStick[i]);
    signal(chopStick[(i + 1) % 5]);
    //  think
} while (1);
```



MONITORS (5.8)

USING SEMAPHORES

- Consider the three mini-programs to the right. For each case, picture what will happen if we run threads on this piece of code.
- What will happen with program A? B? C?
- Next, we'll look at monitors which is an OOP-style approach to blackboxing the mutual exclusion of threads that we are after.

```
//program A
signal(mutex);
//critical section
signal(mutex);
```

```
//program B
signal(mutex);
//critical section
wait(mutex);
```

```
//program C
wait(mutex);
critical section
wait(mutex);
```

MONITOR CONCEPT

- Assume that we have some class that is a so-called "monitor".
- Then, the methods in that class all have access to the instance variables, but only one process at a time is allowed to run within the monitor class.

```
monitor class Account {
    int balance;

    Account(int opening) {
        balance = opening;
    }

    void deposit(int amount) {
        balance = balance + amount;
    }

    void withdraw(int amount) {
        balance = balance - amount;
    }
};
```

CONDITIONAL VARIABLES

- In additional, we'll introduce a new idea to alter program flow: conditional variables.
- A conditional variable exists within a monitor, and is acted upon by two functions (wait, and signal) so that processes using that monitor continue in certain way.
- When a process calls wait, it will block until the conditional is ready.
- When a process calls signal, then (typically) the monitor will transfer control to one of the processes currently waiting on that condition.
  - See signal-and-wait vs signal-and-continue.

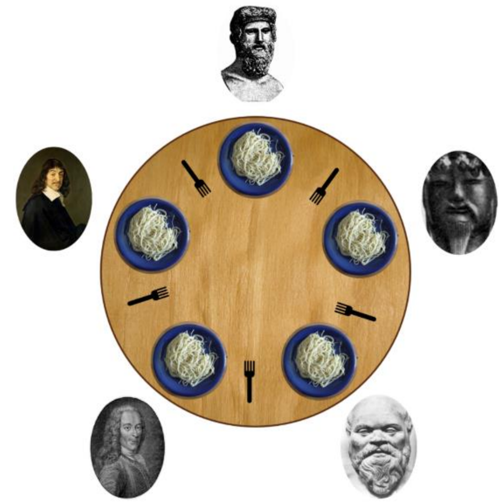
```
monitor class BoundedBuffer {
    //shared data
    int buffer[MAX];
    int fill, use;
    int fullEntries = 0;
    cond_t empty;
    cond_t full;

    void produce(int element) {
        if (fullEntries == MAX)
            wait(&empty);
        buffer[fill] = element;
        fill = (fill + 1) % MAX;
        fullEntries++;
        signal(&full);
    }

    int consume() {
        if (fullEntries == 0)
            wait(&full);
        int tmp = buffer[use];
        use = (use + 1) % MAX;
        fullEntries--;
        signal(&empty);
        return tmp;
     }
}
```


Code from http://pages.cs.wisc.edu/~remzi/

BACK TO DINING



```
monitor DiningPhilosophers {
    enum { THINKING; HUNGRY, EATING) state[5];
    cond_t self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
                state[i] = EATING;
                self[i].signal();
        }
    }

    void initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

IMPLEMENTING A MONITOR USING SEMAPHORES

- Each monitor includes a class wide mutex semaphore that processes must use to access functions.
- As an alternative to a mutex, processes actively in a monitor may "pause" by signaling mutex and instead waiting on next.
  - next_count is number of waiting processes.

```
F() {
  wait(mutex);

  //body of F

  //continue next process
  if  (next_count > 0)
    signal(next);
  else
    signal(mutex);
}
```

- Each conditional variable x has a semaphore (_sem) and int (_count).

```
//x.wait()
x_count++;
if(next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;


//x.signal()
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

SYNCHRONIZATION EXAMPLES (5.9)

Skipping 5.9.1 (Synchronization in Windows), and 5.9.3 (Synchronization in Solaris)

LINUX

- There are various mechanisms in the Linux kernel for providing low-level synchronization:
  - Atomic integer operations (arch/*/atomic.h):
    - `typedef atomic_t`
    - `int atomic_set(atomic_t *a, int value);`
    - `int atomic_add(int value, atomic_t* a)`
    - `int atomic_sub(int value, atomic_t* a)`
    - `int atomic_inc(&counter)`
    - `int atomic_read(atomic_t *a)`
  - Spinlocks (`spin_lock_irqsave, spin_lock_irqsave`)
  - Mutex support (kernel/locking/mutex.c):
    - `int mutex_init(mutex_t *mp, int type, void * arg);`
    - `int mutex_lock(mutex_t *mp);`
    - `int mutex_unlock(mutex_t *mp);`

PTHREADS

- Syntax very straight forward.

```c
#include <pthread.h>
pthread_mutex_t mutex;

// create the mutex lock
pthread_mutex_init(&mutex, NULL);

// acquire the mutex lock
pthread_mutex_lock(&mutex);

// critical section

// release the mutex lock
pthread_mutex_unlock(&mutex);
```

```c
#include <semaphore.h>
sem_t sem;

// Create the semaphore and set it to 1
sem_init(&sem, 0, 1);

// acquire the semaphore
sem_wait(&sem);

// critical section

// release the semaphore
sem_post(&sem);
```

37

ALTERNATIVE APPROACHES (5.10)

SOLVING THE CRITICAL SECTION

- Transactional Memory: the basic idea of transactions emerged from databases (finance) where need to ensure that all of set of operations happens at once (i.e., atomically).

- Thinking back to the discussion on automatic threading, OpenMP also supports marking something as a critical region.

- Most of our problems come from mutable state – if we do a way with state, then many problems vanish… so, just use *functional style programming*! All data will be immutable.

```
#pragma omp parallel
{
  //stuff in parallel

  #pragma omp critical
  {
    printf(“critical");
  }

}
```