Systems 3

Coordination

Marcel Waldvogel

(Handout)

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

Chapter Goals

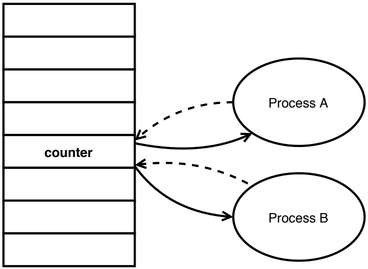
- What is a critical region?
- How can mutual exclusion be achieved?
- What happens if mutual exclusion would be required but is not enforced?
- What are the preconditions for deadlocks?
- How can deadlocks be avoided?

- How do Mutexes, Semaphores, Monitors, and Message Passing work?
- How do they achieve mutual exclusion?
- What are their relationships (similarities, differences)?

Race conditions

```c
int counter = 0;

int main()
{
    while(1) {
        counter = counter + 1;
    }
    return 0;
}
```



1. Why does this not work? (Multiple reasons)
2. Why `counter = counter + 1` instead of `counter++`?

Critical Sections: Avoiding Race Conditions

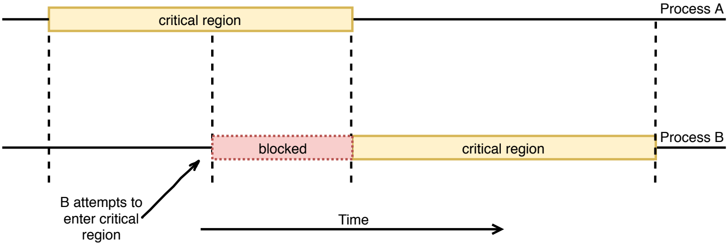
Basic assumptions necessary:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.[1]
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

⇒ Mutual exclusion

[1] One(!) or more. Are CPUs always the same speed?

Mutual exclusion



Strict Alternation

```
  int turn = 0;
```

Thread 0:

```
while(1)
{
   while (turn != 0) {}
   critical_region();
   turn = 1;
   noncritical_region();
}
```

Thread 1:

```
while(1)
{
   while (turn != 1) {}
   critical_region();
   turn = 0;
   noncritical_region();
}
```

Two processes, alternating order.

Peterson's Solution

```
int turn;             /* tie-breaker */
int interested[2];  /* all values initially 0 */

void enter_region(int process)  /* process is 0 or 1 */
{
    int other = 1 - process;    /* number of the other process */
    interested[process] = 1;    /* show that you are interested */
    turn = other;               /* set flag */
    while (turn == other && interested[other]) {}
}

void leave_region(int process)
{
    interested[process] = 0;    /* no longer in critical region */
}
```

1. At most one process in critical region?
2. Progress is being made (no dead-lock)?
3. Bounded waiting time?

Two processes, any order.

Test-Set-Lock Instruction

```
enter_region:
    tsl register,lock    | atomically copy lock to register and set lock to 1
    cmp register,#0      | was lock zero?
    jne enter_region     | if it was non zero, lock was set, so loop
    ret                  | return to caller; cr entered

leave_region:
    move lock,#0         | store a 0 in lock
    ret                  | return to caller
```

Any number of processes, any order.

Atomic Support Examples

```
bool __atomic_clear(void *ptr, int memorder);
bool __atomic_test_and_set(void *ptr, int memorder);

char lock;

/* Acquire */
while (__atomic_test_and_set(&lock, __ATOMIC_ACQUIRE)) {
   /* Wait for lock */
}

/* Critical section here */

/* Release */
__atomic_clear(&lock, __ATOMIC_RELEASE);
```

Producer-Consumer

Bank Teller

- Any number of customers
- Customers can come whenever they want
- Any number of tellers
- Tellers work in parallel

Holiday card writers

- Any number of card writers
- Put finished cards on the shared desk
- Any number of envelope packagers
- Pick up cards for packaging

```c
#define N 100   // number of slots in the buffer
int count = 0;  // number of items in the buffer
```

```c
void producer(void)
{
  int item;
  while (1) {
    item = produce_item();
    if (count == N) sleep();
    insert_item(item);
    count = count + 1;
    if (count == 1) wakeup(consumer);

  }
}
```

```c
void consumer(void)
{
  int item;
  while (1) {

    if (count == 0) sleep();
    item = remove_item();
    count = count - 1;
    if (count == N-1) wakeup(producer);
    consume_item(item);
  }
}
```

Semaphores

Semaphore

- Counter
- Special functions
  - `up()` aka `signal()`[2] aka `wakeup()`
  - `down()` aka `wait()` aka `sleep()`[3]
- Initialized to a value `N`[4]
- When `down()` would like to make the counter negative, the calling thread is blocked until another thread calls. `up()`

---

[2] Not to be confused with Unix/POSIX `signal`(2) handlers!

[3] Not to be confused with `sleep`(2)!

[4] How often one may call `down()` before an `up()` is needed.
For mutual exclusion, this is 1, for a buffer with `N` spaces, this is `N`

```
#define N 100

typedef int semaphore;

semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

```
void producer(void)
{
   int item;
   while (1) {
      item = produce_item();
      // Reserve empty space, any order
      down(&empty);
      // Enforce ordering now
      down(&mutex);
      insert_item(item);
      up(&mutex);
      // One more item is in
      up(&full);

   }
}
```

```
void consumer(void)
{
   int item;
   while (1) {

      // Reserve one item, any order
      down(&full);
      // Enforce ordering
      down(&mutex);
      item = remove_item();
      up(&mutex);
      // One more empty space
      up(&empty);
      consume_item(item);
   }
}
```

Monitors

Mutexes/Semaphores are hard to handle

- Hard to use (3 semaphores for the simple mechanism above!)
- Hard to maintain
- Hard to find (timing) bugs

Need a simpler construct: Monitors

- Simple concept
- Often slightly larger critical regions
- Easy to maintain
- Race conditions, deadlocks rare

→ Only one of the monitor functions active at a time[5]

[5] `synchronized` in Java

```
monitor ProducerConsumer
   condition full, empty;
   integer count;

   procedure insert(item: integer);
   begin
      if count = N then wait(full);
      insert_item(item);
      count := count + 1;
      if count = 1 then signal(empty)
   end;

   function remove: integer;
   begin
      if count = 0 then wait(empty);
      remove = remove_item;
      count := count - 1;
      if count = N - 1 then signal(full)
   end;

   count := 0;
end monitor;
```

```
procedure producer;
begin
   while true do
   begin
      item = produce_item;
      ProducerConsumer.insert(item)
   end
end;

procedure consumer;
begin
   while true do
   begin
      item = ProducerConsumer.remove;
      consume_item(item)
   end
end;
```

Message Passing

Message Passing

- Alternative to Mutex, Semaphore, Monitor
- Conceptually simple(r)[6]
- Does not require shared memory[7]

```
// could be implemented as

send(destination, &message);

receive(source, &message);
```

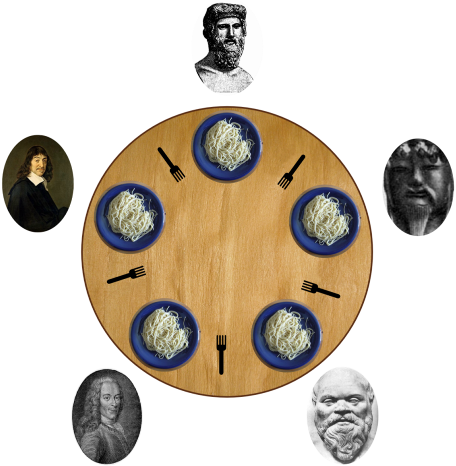
[6] Often implemented using the mechanisms described above

[7] I.e., also works over machine boundaries, over the network

```c
#define N 100
void producer(void)
{
   message m;
   while (TRUE) {
      m.item = produce_item();
      receive(consumer,  &m);
      build_message(&m, item);
      send(consumer, &m);
   }
}

void consumer(void)
{
   int item, i;
   message m;
   for (i = 0; i < N; i++) send(producer, &m);
   while (TRUE) {
      receive(producer, &m);
      send(producer, &m);
      consume_item(m.item);
   }
}
```

The Dining Philosophers Problem



Deadlock

"A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause."

Conditions for Deadlock

- Mutual exclusion
- Hold and wait
- No (lock) preemption
- Circular wait

Examples

- User level
  - Data structures: Hash tables, trees, linked lists, ...
  - SQL with two tables, A and B
  - Locking two files, A and B
- Kernel level
  - Locking two directories, A and B, when moving files between them
  - Writeback buffer, especially under memory pressure
- Linux
  - Interrupt disable
  - Big Kernel Lock[8]
  - Individual locks

[8] Introduced 1996 in Linux 2.0 for multi-processor support; gone as of Linux 3.0 (2011).

Evolution: Fighting programmer's mistakes

- Memory allocation: premature/missing release
  1. C: Ownership strategy
  2. C: Reference counting
  3. Not exception-safe (break, return, ...)
  4. HLL: Garbage collection
- Locks: Premature/missing release
  1. Ownership, counting semaphores
  2. Java: synchronized
  3. Python: with

A nonsolution to the dining philosophers problem

```
#define N 5   // number of philosophers

void philosopher(int i)
{
   while (TRUE) {
      think();
      take_fork(i);           // left
      take_fork((i+1) % N);   // right
      eat();
      put_fork(i);            // left
      put_fork((i+1) % N);    // right
   }
}
```

Why not?

```c
#define N 5
#define LEFT (i+N-1)%N // Left neighbor
#define RIGHT (i+1)%N  // Right neighbor

typedef int semaphore;

enum{THINKING, HUNGRY, EATING} state[N];
// Is initialized to THINKING (=0)

semaphore mutex = 1;
semaphore s[N]; // Initialized to all 0s

void philosopher(int i)
{
  while (1) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}
```

```c
void take_forks(int i)
{
  down(&mutex);
  state[i] = HUNGRY;
  test(i);
  up(&mutex);
  down(&s[i]);
}
void put_forks(int i)
{
  down(&mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  up(&mutex);
}
void test(int i)
{
  if (state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
    state[i] = EATING;
    up(&s[i]);
  }
}
```

Readers and Writers

Example

- Given a shared data structure (hash, linked list, tree, database, ...)
- Some threads want to modify the data structure (read-write access[9])
- Some threads only want to look up information (read-only access[10])

1. Mutual exclusion among readers is wasteful.
2. Only required among writers or between readers and writers.

Problem

How to allow concurrent readers? How to still lock out writers?

1. Only first reader in locks the database; last reader out unlocks.
2. Writers always lock/unlock.

---

[9] aka "writer"

[10] aka "reader"

A solution to the readers and writers problem

```c
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0; // Reader count

void reader(void)
{
   while (1) {
      down(&mutex); // First in?
      rc = rc + 1;
      if (rc == 1) down(&db);
      up(&mutex);
      read_database();
      down(&mutex); // Last out?
      rc = rc - 1;
      if (rc == 0) up(&db);
      up(&mutex);
      use_data_read();
   }
}
```

```c
void writer(void)
{
   while (1) {
      think_up_data();
      down(&db);
      write_database();
      up(&db);
   }
}
```

Expensive operations in blue and red; only the latter need mutual exclusion.