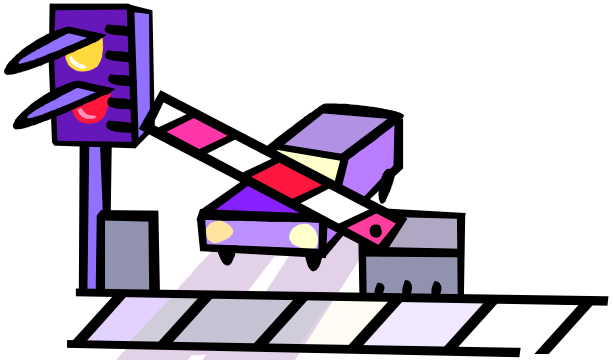
Synchronization



Synchronization

- Problem: coordinating simultaneous access to shared data

```
int cnt = 0;
```

Shared data

```
void * worker( void *ptr )
{
    int i;
    for (i = 0; i < ITERATIONS_PER_THREAD; i++)
        cnt++;
}
```

Critical section
(just one line in this simple example)

- Solution: mutually exclusive access to critical region
  - Only one thread/process accesses shared data at a time

Introducing: Critical Region (Critical Section)

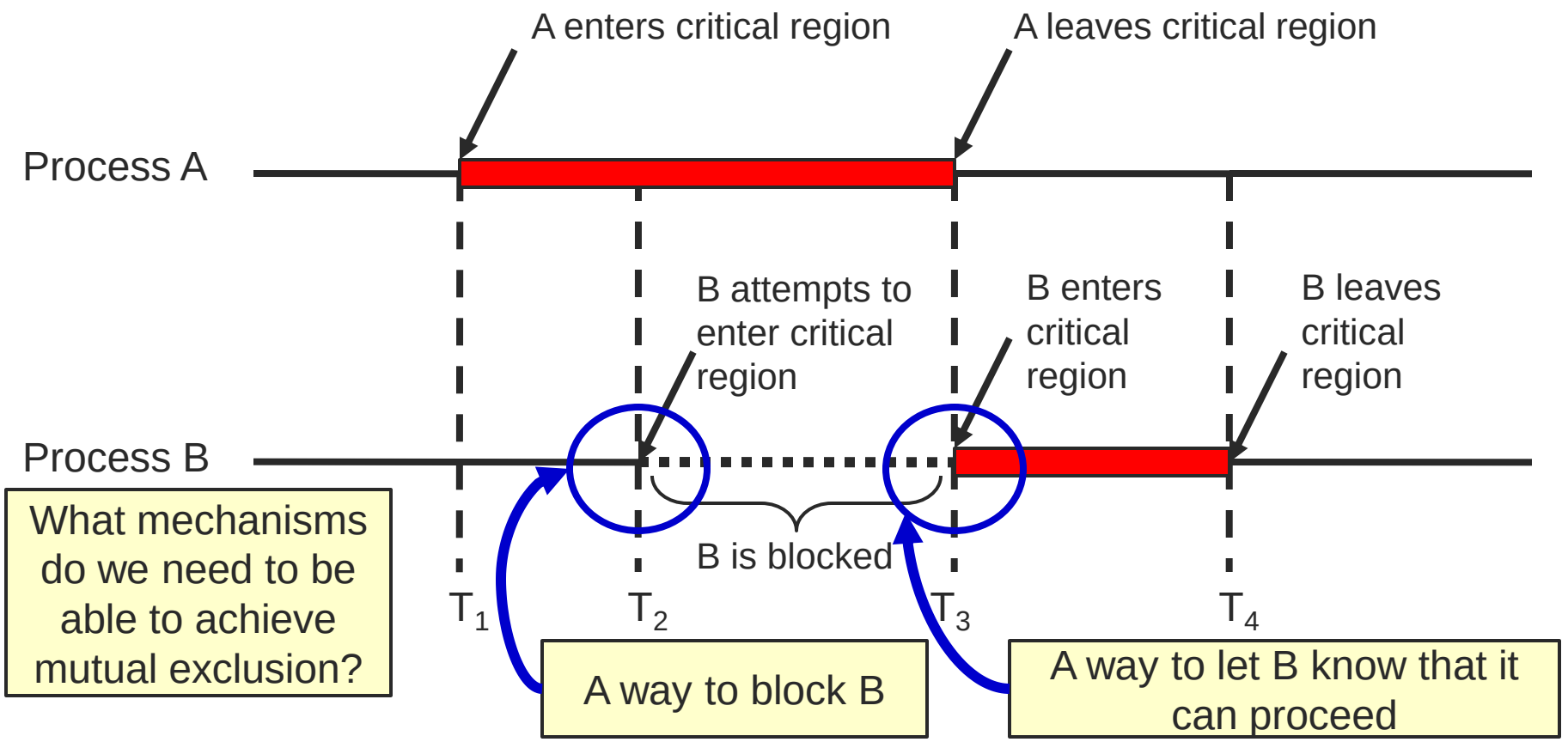
```
Process {
   while (true) {
      ENTER CRITICAL REGION
      Access shared variables;
      LEAVE CRITICAL REGION
      Do other work
   }
}
```

What goes here?

Critical Region Requirements

- Mutual Exclusion
  - Safety
- Progress
  - No deadlock
- Bounded Wait
  - No starvation

Critical Regions



Mutual exclusion using critical regions

Mutual Exclusion Solutions

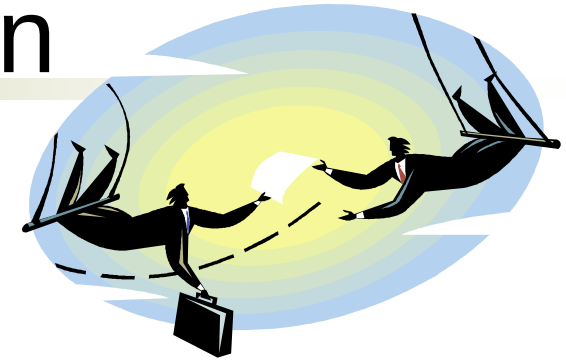
- Software-only candidate solutions (Two-Process Solutions)
  - Lock Variables
  - Turn Mutual Exclusion
  - Other Flag Mutual Exclusion
  - Two Flag Mutual Exclusion
  - Two Flag and Turn Mutual Exclusion
- Hardware solutions
  - Disabling Interrupts; Test-and-set; Swap (Exchange)
- Semaphores

Lock Variables

```
...
while (lock) {
   /* spin spin spin spin */
}
lock = 1;
/* EnterCriticalSection; */
access shared variable;
/* LeaveCriticalSection; */
lock = 0;
...
```

What's the problem?

Turn-based Mutual Exclusion with Strict Alternation

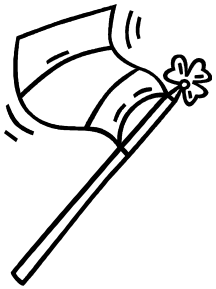


```
…
while (turn != my_process_id) {
   /* wait your turn */
}
access shared variables;
turn = other_process_id;

…
```

What's the problem?

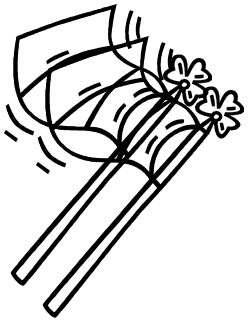
Other Flag Mutual Exclusion



```
int owner[2] = {false, false};
…
while (owner[other_process_id]) {
   /* wait your turn */
}
owner[my_process_id] = true;
access shared variables;
owner[my_process_id] = false;
…
```

What's the problem?

Two Flag Mutual Exclusion

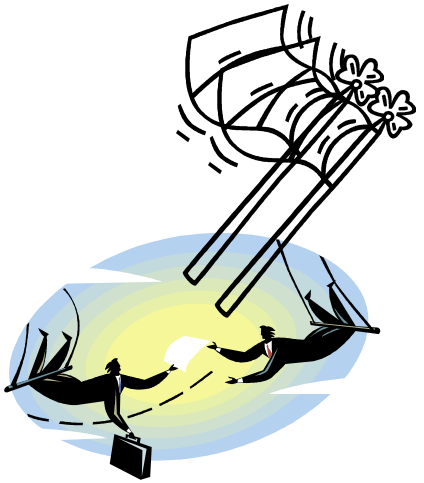


```
int owner[2] = {false, false};
…
owner[my_process_id] = true;
while (owner[other_process_id]) {
    /* wait your turn */
}
access shared variables;
owner[my_process_id] = false;
…
```

What's the problem?

Two Flag and Turn Mutual Exclusion

```
int owner[2]={false, false};
int turn;
…
owner[my_process_id] = true;
turn = other_process_id;
while (owner[other_process_id] and
        turn == other_process_id) {
    /* wait your turn */
}
access shared variables;
owner[my_process_id] = false;
…
```



Are we done?

- Peterson's algorithm works, but...
- Problem: software solutions can be slow
  - at just the moment we'd like to be fast: contention for shared resource
  - Solution: hardware support

Test and Set Instruction

```
boolean Test_And_Set(boolean* lock)
   atomic {
   boolean initial;
   initial = *lock;
   *lock = true;
   return initial;
}
```

atomic = *executed in a single shot without any interruption*

Using Test_And_Set for Mutual Exclusion

```
Pi {
    while(1) {
        while(Test_And_Set(lock)) {

        }

        /* Critical Section */
        lock =0;
        /* remainder */
    }
}
```

```
void main () {
    lock = 0;
    parbegin(P1,…,Pn);
}
```

What's the problem?

Understanding Test and Set

Original

```
boolean test_and_set(boolean* lock) atomic {
    boolean initial = *lock;
    *lock = true;
    return initial;
}
```

Functionally equivalent version

```
boolean test_and_set(boolean* lock) atomic {
    if (*lock == 1)
        return 1; // failure
    else {
        *lock = 1;
        return 0; // success
    }
}
```


Now are we done?

- Hardware solutions are fast, but...
- Problem: starvation
  - No guarantee about which process “wins” the test-and-set race
  - It’ll eventually happen, but a process could wait indefinitely
- Problem: deadlock
  - Proc. 1 enters critical section, gets interrupted by higher priority Proc. 2
  - P1 can’t make progress: waiting to run until P2 is done
  - P2 can’t make progress: busy-waiting until P1 exits critical section
- Problem: busy-waiting
  - Critical section might be arbitrarily long
  - Waiting processes all still spend CPU time!
- These problems occur for software solutions too
- Solution: Semaphores

Semaphores

- Fundamental Principle:
  - Two or more processes want to cooperate by means of simple signals
- Special Variable: **`semaphore s`**
  - A special kind of “int” variable
  - Can’t just modify or set or increment or decrement it

Semaphores for Mutual Exclusion

- Basic idea
  - Associate a unique semaphore **`mutex`** with each shared variable
    - Initially 1
  - Surround corresponding critical sections
    - **`semWait(mutex)`**
    - **`semSignal(mutex)`**

Semaphore Terminology

- Binary semaphore
  - Value is always 0 or 1
- Mutex
  - Binary semaphore used for mutual exclusion
    - Wait operation: “locking” the mutex
    - Signal operation: “unlocking” or “releasing” the mutex
- Counting semaphore
  - Count a set of available resources
  - Value starts at max

Semaphores

- Before entering critical section
  - **`semWait(s)`**
    - Receive signal via semaphore **`s`**
    - “down” on the semaphore
    - Also: **`P`** – proberen
- After finishing critical section
  - **`semSignal(s)`**
    - Transmit signal via semaphore **`s`**
    - “up” on the semaphore
    - Also: **`V`** – verhogen
- Implementation requirements
  - **`semSignal`** and **`semWait`** must be atomic

Semaphores vs. Test_and_Set

Semaphore

```
semaphore s = 1;
Pi {
    while(1)   {
        semWait(s);
        /* Critical Section */
        semSignal(s);
        /* remainder */
    }
}
```

Test_and_Set

```
lock = 0;
Pi {
    while(1) {
        while(Test_And_Set(lock));
        /* Critical Section */
        lock =0;
        /* remainder */
    }
}
```

- Avoid busy waiting by suspending
  - Block if **`s == False`**
  - Wakeup on signal (**`s = True`**)

Inside a Semaphore

- Requirement
  - No two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time!

- Critical section
  - **`wait()`** and **`signal()`** code
  - Now have busy waiting in critical section implementation
    - Implementation code is short
    - Little busy waiting if critical section rarely occupied
    - Bad for applications may spend lots of time in critical sections

Inside a Semaphore

- Add a waiting queue
- Multiple process waiting on **`s`**
  - Wakeup one of the blocked processes upon getting a signal

- Semaphore data structure

```
typedef struct {
    int count;
    queueType queue;
    /* queue for procs.
    waiting on s */
} SEMAPHORE;
```


Binary Semaphores

```
typedef struct bsemaphore {
    enum {0,1} value;
    queueType queue;
 } BSEMAPHORE;
```

```
void semWaitB(bsemaphore s) {
    if (s.value == 1)
        s.value = 0;
    else {
        place P in s.queue;
        block P;
    }
}
```

```
void semSignalB (bsemaphore s)
    {
    if (s.queue is empty())
        s.value = 1;
    else {
        remove P from s.queue;
        place P on ready list;
    }
}
```


General Semaphore

```
typedef struct {
    int count;
     queueType queue;
} SEMAPHORE;
```

```
void semWait(semaphore s) {
    s.count--;
        if (s.count < 0) {
        place P in s.queue;
        block P;
    }
}
```

```
void semSignal(semaphore s) {
    s.count++;
    if (s.count ≤ 0) {
        remove P from s.queue;
        place P on ready list;
    }
}
```


Making the operations atomic

- Isn't this exactly what semaphores were trying to solve? Are we stuck?!
- Solution: resort to test-and-set

```
typedef struct {
  boolean lock;
  int count;
  queueType queue;
} SEMAPHORE;
```

```
void semWait(semaphore s) {
  while (test_and_set(lock)) { }
  s.count--;
  if (s.count < 0) {
    place P in s.queue;
    block P;
  }
  lock = 0;
}
```


Making the operations atomic

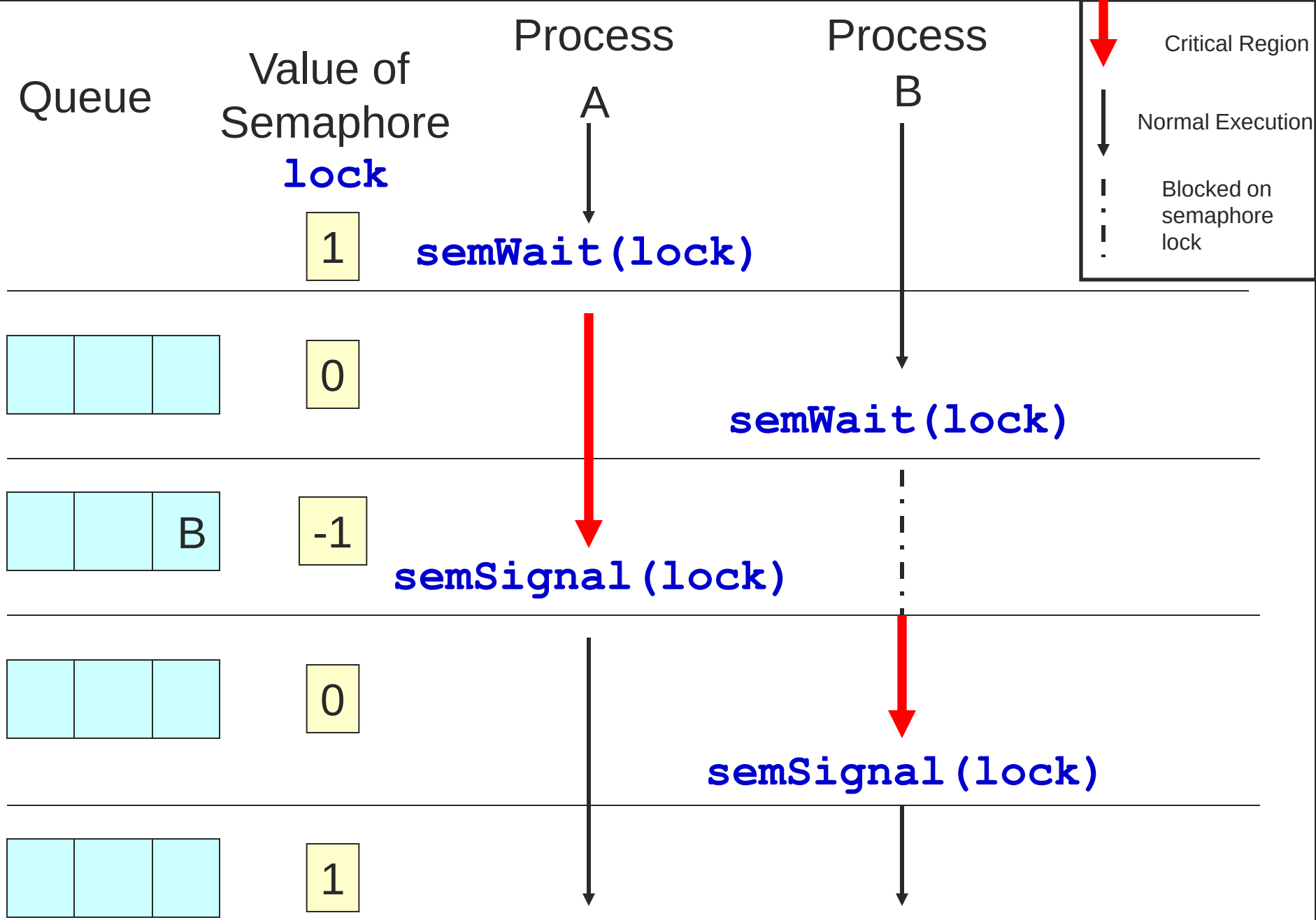
- Busy-waiting again!
- Then how are semaphores better than just using test_and_set?

```
void semWait(semaphore s) {
  while (test_and_set(lock)) { }
  s.count--;
  if (s.count < 0) {
    place P in s.queue;
    block P;
  }
  lock = 0;
}
```

- T&S: busy-wait during critical section
- Sem.: busy-wait just during semWait, semSignal: very short operations!

Mutual Exclusion Using Semaphores

```
semaphore s = 1;
Pi {
   while(1) {
      semWait(s);
      /* Critical Section */
      semSignal(s);
      /* remainder */
   }
}
```

Semaphore Example 1

```
semaphore s = 2;
Pi {
   while(1) {
      semWait(s);
      /* CS */
      semSignal(s);
      /* remainder */
   }
}
```

- What happens?
- When might this be desirable?

Semaphore Example 2

```
semaphore s = 0;
Pi {
    while(1) {
        semWait(s);
        /* CS */
        semSignal(s);
        /* remainder */
    }
}
```

- What happens?
- When might this be desirable?

Semaphore Example 3

```
semaphore s = 0;
P1 {
   /* do some stuff */
   semWait(s);
   /* do some more stuff */
}
```

```
semaphore s; /* shared */
P2 {
   /* do some stuff */
   semSignal(s);
   /* do some more stuff */
}
```

- What happens?

- When might this be desirable?

Semaphore Example 4

Process 1 executes:

```
while(1) {
    semWait(S);
    a;
    semSignal(Q);
}
```

Process 2 executes:

```
while(1) {
    semWait(Q);
    b;
    semSignal(S);
}
```

- Two processes
  - two semaphores: S and Q
  - Protect two critical variables ‘a’ and ‘b’.
- What happens in the pseudocode if Semaphores S and Q are initialized to 1 (or 0)?

Summary

- Synchronization is important for correct multi-threading programs
- Critical regions
- Solutions to protect critical regions
  - Software-only approaches
  - Other hardware solutions
  - Semaphores