Software solution: Peterson’s algorithm

\( p_0 \):

```c
... flag[0] = true;
turn = 1;
while (flag[1] && turn == 1) /* do nothing */
<critical section>
flag[0] = false;
...```

\( p_1 \):

```c
... flag[1] = true;
turn = 0;
while (flag[0] && turn == 0) /* do nothing */
<critical section>
flag[1] = false;
...```

Peterson’s algorithm:

- Implements mutual exclusion
- Not widely used:
  - Burns CPU cycles
  - Can be extended to work for \( n \) processes, but overhead increases
  - Cannot be extended to work for an unknown number of processes

Hardware Approaches to Mutual Exclusion

- How can the hardware help us to implement mutual exclusion?
  - Interrupt disabling:
    - Useful on uniprocessor systems only
    - Prevents preemption
    ```c
    ... <disable interrupts/signals>
    <critical section>
    <enable interrupts/signals>
    ...
    ```
    Example: OS/161
    ```c
    spl = splhigh();
    <critical section>
    splx(spl);
    ```
    - Useful within OS, not appropriate for user processes

Special Machine Instructions

- Software approaches exploit a property guaranteed by the hardware:
  - Each memory access is atomic
- Problems occurred as we sometimes would like a number of memory accesses to be atomic
- Could the hardware provide complex atomic operations that help us?
### Slide 5

**Test and set:**

```c
atomic bool testset (int i)
{
    if (0 == i)
    {
        i = 1;
        return true;
    }
    else
    return false;
}
```

**Exchange:**

```c
atomic void exchange (int register, int memory)
{
    int tmp;
    tmp = memory;
    memory = register;
    register = tmp;
}
```

### Slide 6

**Mutual exclusion with test-and-set:**

```c
int bolt = 0;
void proc (int i) {
    for (;;) {
        while (!testset (bolt))
        {
            /* do nothing */;
            <critical section>
            bolt = 0;
            <remainder>
        }
    }
}
void main () {
    bolt = 0;
    parbegin (proc (1), proc (2), ..., proc (N));
}
```

### Slide 7

**Advantages of special machine instructions:**

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and therefore easy to verify

**Disadvantages of special machine instructions:**

- Busy-waiting consumes processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting.
- Deadlock
  - If a low priority process has the critical region and a higher priority process requires access, the higher priority process will obtain the processor to wait for the critical region

### Slide 8

**Semaphores**

- Dijkstra (1965) introduced the concept of a semaphore in his study of cooperating sequential processes
- Semaphores are variables that are used to signal the status of shared resources to processes

**How does that work?**

- If a resource is not available, the corresponding semaphore blocks any process waiting for the resource
- Blocked processes are put into a process queue maintained by the semaphore (avoids busy waiting!)
- When a process releases a resource, it signals this by means of the semaphore
- Signalling resumes a blocked process if there is any
- Wait and signal operations cannot be interrupted
- Complex coordination can be specified by multiple semaphores
How are semaphores implemented?

- A semaphore is a variable $s$ consisting of
  - an integer value $count$ and
  - a process queue $queue$

- Initially, $count$ is set to a nonnegative value and $queue$ is empty

There are two operations that a process $current$ can apply:

- $wait(s)$: Decrement $count$; if $count$ becomes negative, put $current$ into $queue$
- $signal(s)$: Increment $count$; if $count$ is not positive, unblock a process from $queue$

```
typedef struct {
    int count;
    queue_t queue;
} semaphore;

void wait (semaphore s) {
    s.count--; if (s.count < 0) {
        <place current in s.queue>
        <block current>
    }
}

void signal (semaphore s) {
    s.count++; if (s.count <= 0) {
        <remove a process P from s.queue>
        <place P on ready list>
    }
}
```

There are various flavours of semaphores:

- Counting semaphores versus binary semaphores:
  - In a counting semaphore, $count$ can take arbitrary integer values
  - In a binary semaphore, $count$ can only be 0 or 1
  - Counting semaphores can be implemented in terms of binary semaphores (how?)

- Strong semaphores versus weak semaphores:
  - In a strong semaphore, $queue$ adheres to the FIFO policy
  - In a weak semaphore, any process may be taken from $queue$
  - Strong semaphores can be implemented in terms of weak semaphores (how?)

```
void main () {
    parbegin (proc (1), proc (2), ..., proc (n));
}
```

Mutual Exclusion

Implementation of mutual exclusion with semaphores:

```
semaphore s;
    s.count = 1;
    s.queue = empty_queue ();

void proc (int i) {
    for (;;) {
        wait (s);
        <critical section>
        signal (s);
        <remainder>
    }
}
```
Mutex:
- A semaphore that allows only one process in a critical section is often called a mutex.
- There exist various flavours, such as, read-write mutexes and read-write-update mutexes.
- Given exchange or test-and-set are available, easy to implement in user-level:
  1. test-and-set lock
  2. if successful, return
  3. if not, yield current thread, repeat.

Semaphore in OS/161:
- Defined in src/kern/thread/synch.c and src/kern/include/synch.h.
- Operations are called:
  - P (proberen: try), instead of wait
  - V (verhogen: increase), instead of signal
- Definition of data type semaphore:

```c
struct semaphore {
    char *name;
    volatile int count;
};
```

```c
struct semaphore* sem_create (const char *name, int initial_count);
void          P (struct semaphore *);
void          V (struct semaphore *);
void          sem_destroy(struct semaphore *);
```

- Where is the queue??

```
void P(struct semaphore *sem) {
    int spl;
    assert(sem != NULL);

    /* May not block in an interrupt handler.
     * For robustness, always check, even if we can actually
     * complete the P without blocking. */
    assert(in_interrupt==0);

    spl = splhigh();
    while (sem->count==0) {
        thread_sleep(sem); }
    assert(sem->count>0);
    sem->count--;
    splx(spl);
}
```

```
void V(struct semaphore *sem) {
    int spl;
    assert(sem != NULL);
    spl = splhigh();
    sem->count++;
    assert(sem->count>0);
    thread_wakeup(sem);
    splx(spl);
}
```
### Mutexes in OS/161

```c
struct lock {
    char * name;
    struct thread * volatile holder;
};

struct lock * lock_create (const char * name);
void lock_acquire (struct lock *);
void lock_release (struct lock *);
int lock_do_i_hold (struct lock *);
void lock_destroy (struct lock *);
```

### PRODUCER/CONSUMER PROBLEM

- One or more **producers** are generating data and placing these in a buffer
- A single **consumer** is taking items out of the buffer one at a time
- Only one producer or consumer may access the buffer at any one time

```c
int in, out;
elem_t b[];

producer:
for (;;) {
    <produce item v>
    b[in] = v;
    in++;
}

gconsumer:
for (;;) {
    while (in <= out)
        /* do nothing */
    w = b[out];
    out++;
    <consume item w>
}
```

### PRODUCER WITH CIRCULAR BUFFER

- semaphore n = init_sem (0); /* number of items in buffer */
- semaphore s = init_sem (1); /* access to critical section */

```c
void producer () {
    for (;;) {
        v = produce ();
        wait (s);
        append (v);
        signal (s); signal (n);
    }
}

void consumer () {
    for (;;) {
        wait (n); wait (s);
        v = take ();
        signal (s);
        consume (v);
    }
} 
```
Producer with Circular Buffer

```c
int in, out;
elem_t b[];

Producer:
for (;;) {
    <produce item v>
    while (((in + 1) % n == out)
        /* do nothing */;
    b[in] = v;
    in = (in + 1) % n;
}
```

Consumer:
```c
for (;;) {
    <consume item w>
    while (in == out)
        /* do nothing */;
    w = b[out];
    out = (out + 1) % n;
}
```

Monitors

- A monitor is a software module implementing mutual exclusion
- Monitors are easier to program than semaphores
- Natively supported by a number of programming languages: Concurrent Pascal, Modula-(23) & Java
- Chief characteristics:
  - Local data variables are accessible only by the monitor (not externally)
  - Process enters monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time
- Main problem: provides less control, coarse grain

Synchronisation in a monitor:

- `cwait (c)`: Suspend current on condition c (opens monitor to other processes)
- `csignal (c)`: Resume execution of a processes suspended on condition c (ignored if no such process)
Structure of a monitor:

Producer/consumer using a monitor:

```java
char buffer[N];
int nextin = 0, nextout = 0, count = 0;
condition_t not_full, not_empty;

void append (char c) {
    if (N == count)
        cwait (not_full);
    buffer[nextin] = c;
    nextin = (nextin + 1) % N;
    count++;
    csignal (not_empty);
}

void take (char c) {
    if (0 == count)
        cwait (not_empty);
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    csignal (not_full);
}
```

Monitors in Java

Resources or critical sections can be protected using the `synchronized` keyword:

```java
synchronized (<expression>) {
    <statements>
}
```
- `<expression>` must evaluate to an object or array
- thread only proceeds after obtaining the lock of the object
- `synchronized` can be applied to a method: entire method is a critical section