

The critical section problem

- A critical section is a code segment of a concurrent process in which a shared resource is accessed
- Concurrent access to a shared variable is potentially dangerous
 - Example: if $a=0$, what is the result of the command $a=a+1$ executed simultaneously by processes A and B?
 - A common solution is the mutual exclusion i.e. serialization of accesses

Mutual exclusion

- Mutual exclusion is a concern in both uni- and multi-processor machines, but it is trickier on the latter
- A mechanism for mutual exclusion must have the following properties:
 - only one process in a critical section at a time
 - no delay in entering a critical section if no other process is executing in the critical section
 - Whane processes compete to enter a c.s., a selection must be completed in a finite amount of time
 - The selection must be fair

Early mechanisms for mutual exclusion

- Busy waiting on a status variable (spin lock)
 - disadvantage: waste of CPU cycles and memory access bandwidth
- On uniprocessors: Disabling interrupts
 - disabling interrupts is risky
- On multiprocessors: Test-and-set instructions
 - test-and-set: operation to read and modify a register in a single atomic operation

Example of busy waiting on a lock (1/2)

- One could think of using a variable as a flag to be checked upon entering a critical section ...
- ... but the lock itself is a critical section!

```
Shared integer lock = 0;
Process i
.
.
.
while lock = 1;
lock = 1;
execute CS;
lock = 0;
.
.
.
```

Process A	Process B
...	...
while lock = 1;	while lock = 1;
lock = 1;	lock = 1;
...	...

Possible race condition

Example of busy waiting on a lock (2/2)

- The correct implementation uses a test-and-set instruction to avoid race conditions

Semantic of test-and-set instruction

```
int test-and-set (int a) {
    int rv = a;
    a = 1;
    return rv;
}
```

Correct lock implementation

```
Process A
Shared integer lock = 0;
.
.
while test-and-set(lock) = 1
;
.
.
```

Locks: pros and cons

- Pros:
 - simple and fast
 - ubiquitous: every processor has a test-and-set or equivalent operation
- Cons:
 - busy waiting is wasteful of resources (CPU cycles, memory bandwidth)

Mutual exclusion: software solution

- We saw that a special instruction (test-and-set) is needed to avoid race conditions
- However purely software solutions to the mutual exclusion problem do exist
 - the trick is to write a lock that works even in presence of race conditions
 - solution at right works for two processes, can be generalized to N
- Same drawback as the simple lock
 - solutions based on spin lock waste a lot of CPU cycles

```

Process i
.
.
flag[i] = true;
turn = j;
while (flag[j] && turn==j);
critical section
flag[i] = false;
.
.
    
```

Semaphores - definition

- Proposed by Dijkstra, it was the first high level constructs used to synchronize concurrent processes.
- A semaphore S is an integer variable on which two atomic operations are defined, P(S) and V(S), and with an associated queue.
- P and V semantic:

```

P(S) : if S ≥ 1 then S := S - 1
      else <block and enqueue the process>;
    
```

```

V(S) : if <some process is blocked on the queue> then
      <unblock a process>
      else S := S + 1;
    
```

Semaphores - properties

- The P operation may block a process, but V does not
- Two type of semaphores
 - binary: initial value is 1
 - resource counting: any initial value
- P and V are atomic operations

```

P(S) : if S ≥ 1 then S := S - 1
      else <block and enqueue the process>;
    
```

```

V(S) : if <some process is blocked on the queue> then
      <unblock a process>
      else S := S + 1;
    
```

Example of use

Shared var mutex: semaphore = 1;

Process i

```

begin
.
.
P(mutex);
execute CS;
V(mutex);
.
.
End;
    
```

Sample synchronization problems

- Semaphore can be used in other synchronization problems besides Mutual Exclusion
- The Producer-Consumer problem
 - a finite buffer pool is used to exchange messages between producer and consumer processes
- The Readers-Writers Problem
 - reader and writer processes accessing the same file
- The Dining Philosophers Problem
 - five philosophers competing for a pair of forks



Producer-Consumer Problem

- The shared resource is a common buffer pool used to exchange messages between processes
 - two set of processes: producers and consumers
 - Processes act asynchronously, i.e. they extract/deposit messages at any instant
- The synchronization constraints that need to be enforced are:
 - No consumer process can extract messages when the pool is empty
 - No producer process can deposit a message when the pool is full

Producer-Consumer: solution #1

```
Process producer          Process consumer
.                          .
.                          .
while count = N          while count = 0
.                          .
.                          .
P(mutex)                P(mutex)
count = count + 1         count = count - 1
write(head_ptr)           read(tail_ptr)
head_ptr = (head_ptr + 1) mod N
V(mutex)                V(mutex)
.                          .
.                          .
```

- Semaphore mutex ensures mutual exclusion in accessing the pool, however solution shown is not correct because variable *count* is not protected (for example two producers could enter when *count* = N-1)