CRITICAL REGION PROBLEM: MUTUAL EXCLUSION

Module Number 2. Section 10
COP4600 – Operating Systems
Richard Newman

CLASSIC SYNCHONIZATION PROBLEMS

Critical Section Problem: Mutual exclusion – only one process can be in Critical Region at a time.

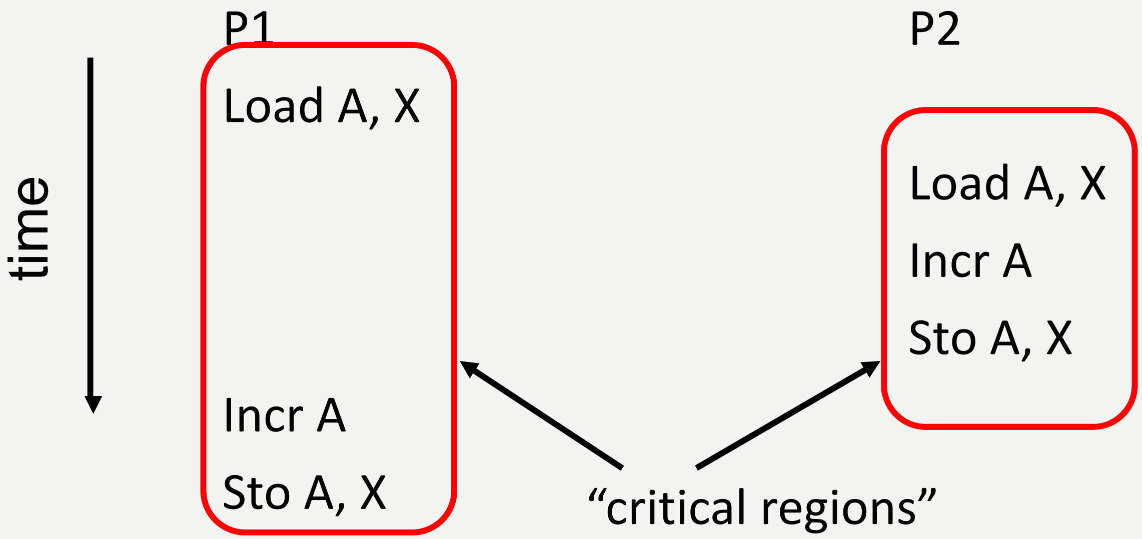
Producer-Consumer Problem: Producers produce items and write them into buffers; Consumers remove items from buffers and consume them. A buffer can only be accessed by one process at a time.

Bounded Buffer Problem: P-C with finite # buffers.

Dining Philosophers: Philosophers arranged in a circle share a fork between each pair of neighbors. Both forks are needed for a philosopher to eat, and only one philosopher can use a fork at a time.

Readers-Writers: Any number of readers can read from a DB simultaneously, but writers must access it alone

RACE CONDITION EXAMPLE



If initial value of X was 0, final value of X is 1, not 2.

Want to prevent other process from accessing X at same time!

CRITICAL REGIONS- MUTUAL EXCLUSION

Requirements for Mutual Exclusion Solution:

Concurrency: (**C1**) No assumptions may be made about speeds or the number of CPUs.

Safety: (**S1**) No two processes may be simultaneously inside their critical regions.

Liveness: (**L1**) No process running outside its critical region may block other processes;

(**L2**) No process should have to wait forever to enter its critical region.

SYNCHRONIZATION CODE

Generic Structure of a Process using a CR:

```
Process (int  i) {
      while (TRUE) {
      PreOther(i)         /* arbitrarily long time outside CR */
      Entry(i)            /* make sure it's OK to enter */
         Critical_Region(i)      /* exclusive access */
      Exit(i)             /* code to let others know I'm done */
      PostOther(i)        /* arbitrarily long time outside CR */
      }
}
```

SYNCHONIZATION MECHANISMS

Basic flavors of synchronization mechanisms:

- Busy waiting (spin locks) – S/W or H/W based
- Semaphores (scheduling based) – OS support
- Monitors – language support
- Message-based – blocking receive, OS support

CRITICAL REGIONS (2)

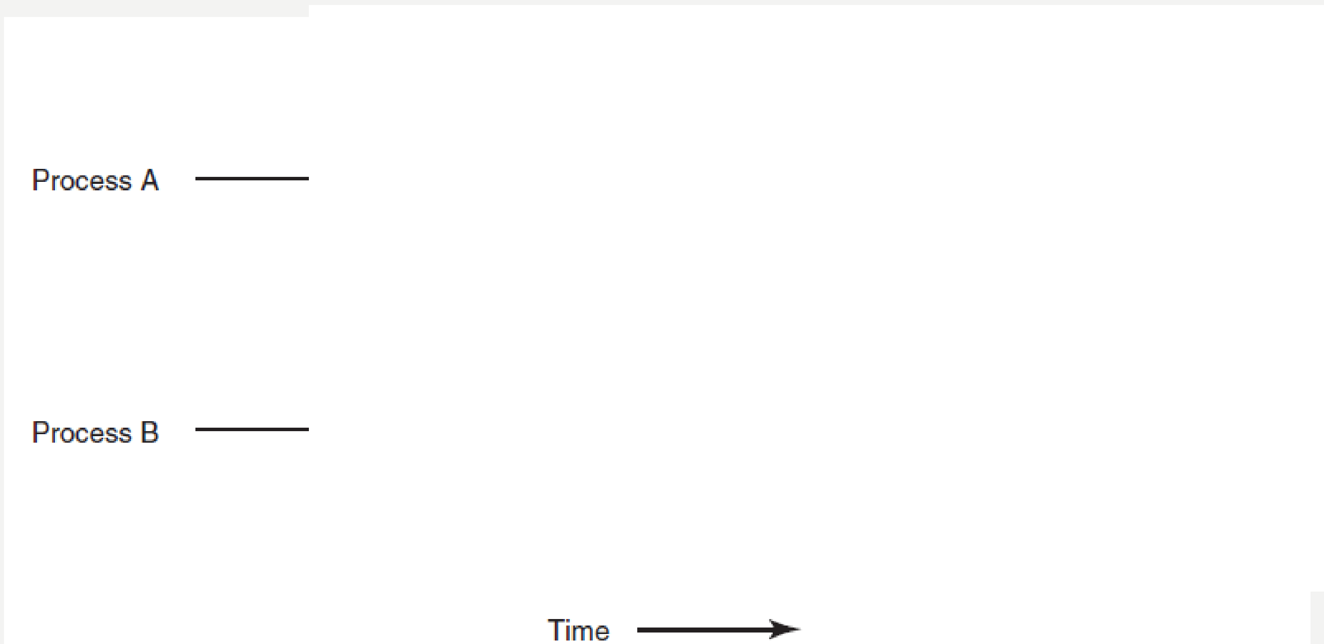


Figure 2-22. Mutual exclusion using critical regions.

MUTUAL EXCLUSION WITH BUSY WAITING: STRICT ALTERNATION

Where is entry code?

Tight spin on turn

```
while (TRUE) {
    while (turn != 0)          /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)          /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Where is exit code?

What is so bad about this solution?

Figure 2-23. A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the *while* statements.

MUTUAL EXCLUSION WITH BUSY WAITING: FLAG INTEREST

```
(void) enter_region(int process) {
   other = 1 – process;
   interested [process] =  TRUE;      /* signal desire to enter CR */
   while (interested[other]) ;        /* loop */
}
(void) leave_region(int process) {
   other = 1 – process;
   interested [process] =  FALSE;     /* signal no longer in CR */
}
```

What is so bad about *this* solution?

A non-solution

MUTUAL EXCLUSION WITH BUSY WAITING: PETERSON'S SOLUTION

```
#define FALSE  0
#define TRUE   1
#define N      2                          /* number of processes */

int turn;                                 /* whose turn is it? */
int interested[N];                        /* all values initially 0 (FALSE) */

void enter_region(int process);           /* process is 0 or 1 */
{
      int other;                          /* number of the other process */

      other = 1 – process;                /* the opposite of process */
      interested[process] = TRUE;         /* show that you are interested */
      turn = other;                       /* be polite */
      while (turn == other && interested[other]) ; /* loop */
}

void leave_region(int process)            /* process: who is leaving */
{
      interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Combines two partial solutions into one complete solution

MUTUAL EXCLUSION WITH BUSY WAITING: THE TSL INSTRUCTION

```
enter_region:
      TSL REGISTER,LOCK            | copy lock to register and set lock to 1
      CMP REGISTER,#0              | was lock zero?
      JNE enter_region             | if it was nonzero, lock was set, so loop
      RET                          | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0                 | store a 0 in lock
      RET                          | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

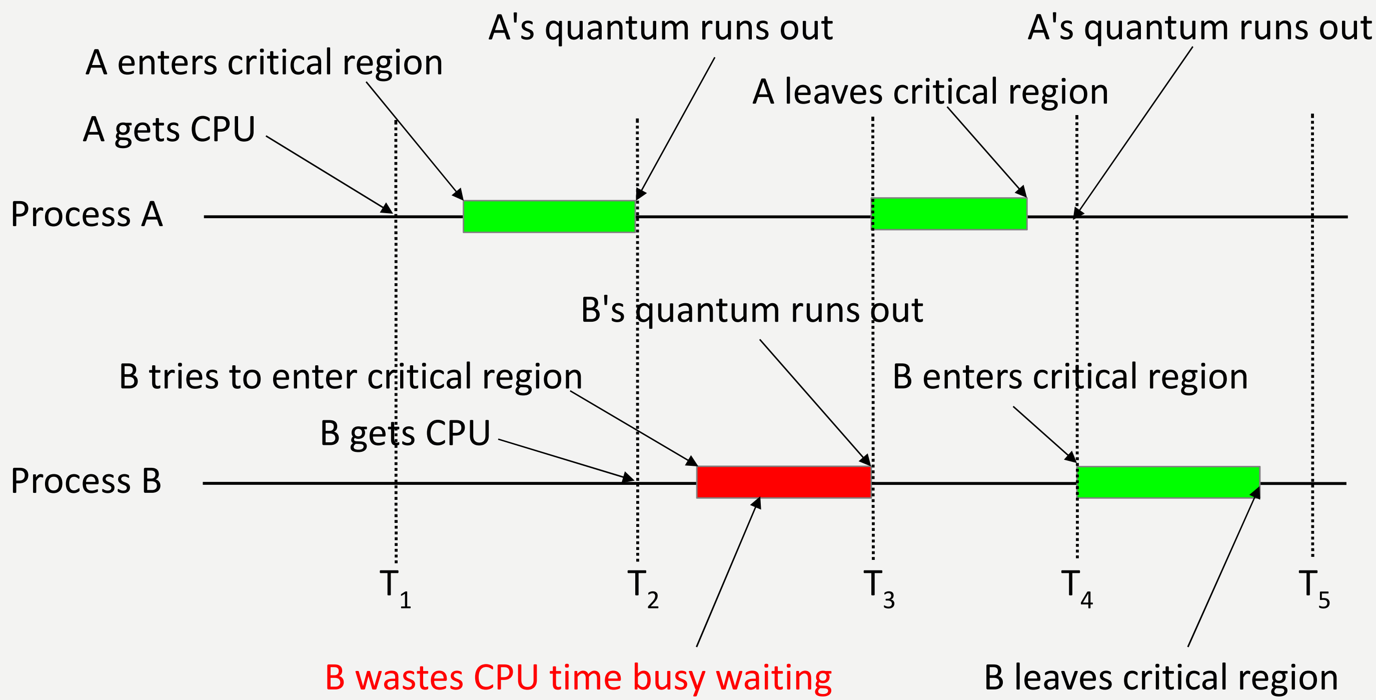
MUTUAL EXCLUSION WITH BUSY WAITING: THE SWAP INSTRUCTION

```
enter_region:
      MOVE REGISTER,#1              | put a 1 in the register
      XCHG REGISTER,LOCK            | swap the contents of the register and lock variable
      CMP REGISTER,#0               | was lock zero?
      JNE enter_region              | if it was non zero, lock was set, so loop
      RET                           | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0                  | store a 0 in lock
      RET                           | return to caller
```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction

MUTUAL EXCLUSION WITH BUSY WAITING



With busy waiting, B may be using the CPU while “blocked” and so A can't leave critical region (until A gets CPU)

Now imagine what would happen if B were higher priority than A!

SEMAPHORES

Semaphores are a *special type* of shared memory:

1 – A semaphore S can only be declared and initialized, or accessed by Up(S) or Down(S);

NO direct access

2 – When Down(S) is called, if S > 0, S-- and continue; otherwise block the caller and put on S's queue Q

3 – When Up(S) is called, if Q empty, S++ and continue; otherwise remove a process P from Q and unblock P

4 – Up() never blocks

5 – A process blocked on Down() does not use the CPU

SEMAPHORES FOR MUTUAL EXCLUSION

```
Semaphore S=1;

Process (int  i) {
      while (TRUE) {
      Other(i);        /* arbitrarily long time outside CR */
      Down(S);         /* code to make sure it's OK to enter */
         Critical_Region(i);         /* exclusive access */
      Up(S);               /* code to let others know I'm done */
      }
 }
```

MUTEXES

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Pthreads package mutex_lock and mutex_unlock are similar to semaphores – blocked thread does not consume cycles!

Figure 2-29. Implementation of *mutex_lock* and *mutex_unlock*.

PTHREADS MUTEX FOR MUTUAL EXCLUSION

```
pthread_mutex_t lock;

P(void *ptr) {
    while (TRUE) {
    Other();
    pthread_mutex_lock(&lock);
        Critical_Region();
     pthread_mutex_unlock(&lock);
    }
}
```

```
int main(int argc, char** argv) {
  int i;
  pthread_t proc[N];
  pthread_mutex_init(&lock, 0);
  for (i=0; i<N; ++i)
    pthread_create(&proc[i],0,P,0);
  for (i=0; i<N; ++i)
    pthread_join(proc[i],0);
  pthread_mutex_destroy(&lock);
}
```

MONITOR MUTUAL EXCLUSION

```
monitor mutex
    …
    procedure CR(int i)
    begin
        …
    end
    …
end monitor;
```

Shared variables in monitor are private

```
Process (int  i) {
    while (TRUE) {
    Other(i);
    mutex.CR(i);
    }
}
```

Note that entry and exit procedures normally surrounding the critical region code are *automatically provided* by the monitor

CRITICAL REGION SOLUTION WITH MESSAGE PASSING

```
Process (int  i) {
    message m;
    while (TRUE) {
        Other(i);
        Send("Req", Server);
        Receive(&m, Server);
          Critical_Region(i);
        Send(m, Server);
    }
}
```

```
Server () {
    message m;
    address src;
    while (TRUE) {
        Receive(&m, &src);
        Send("OK", src);
        Receive(&m, src);
    }
}
```

Receive blocks

Receive must be from token holder

SIGNIFICANCE OF SOLVING THE CR PROBLEM

Race conditions have manifested themselves since the beginning of computing

They arise naturally in hardware and in software

Solutions to the Critical Region problem are needed to solve many other, more complex problems, such as the other three classical synchronization problems

There are some distributed algorithms that enable concurrency without exclusive access …

SUMMARY

- Need for mutual exclusion – race conditions
- Requirements for critical region solution
- Shared Memory - Spin locks
    - Software only (Peterson)
    - Hardware – TSL, swap
- Semaphores
- Pthread mutexes
- Monitor Solution – just make critical region an entry procedure in monitor!
- Message-passing