Preview

Inter-Process Communication

- Race Condition
- Critical Section
- Mutual Exclusion in a Critical Section
  - With Busy Waiting
    - Disabling Interrupts –non-preemptive kernel
    - Lock Variables
    - Strict Alternation
    - Peterson's Solution
    - Hardware Solution
      - Test and Set Lock –
      - Memory Barriers
      - Atomic Variable
    - Priority Inversion problems with busy waiting

Interprocess Communication

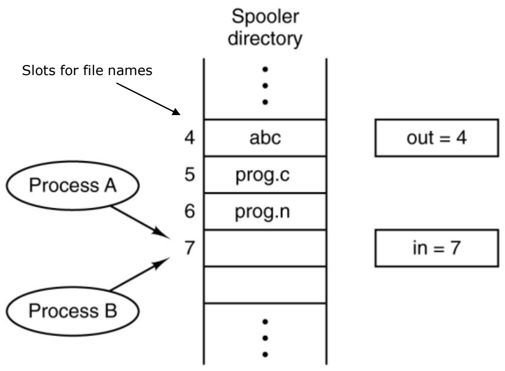
- Three issues in interprocess communication

1. How one process can pass information to another (communication between processes)
2. How to make sure two or more processes do not get into the critical section (mutual exclusion)
3. Proper sequencing (Synchronization) when dependencies are present (ex. A create outputs, B consume the outputs)

Interprocess Communication
(Race Condition)

- Race Condition
  - A situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race condition.
- Critical section (critical region)
  - The part of program where the shared memory is accessed.
- Mutual Exclusion in a critical section can avoid races condition:
  - If we could arrange matters such that no two processes were ever in their critical regions at the same time, we can avoid races condition.

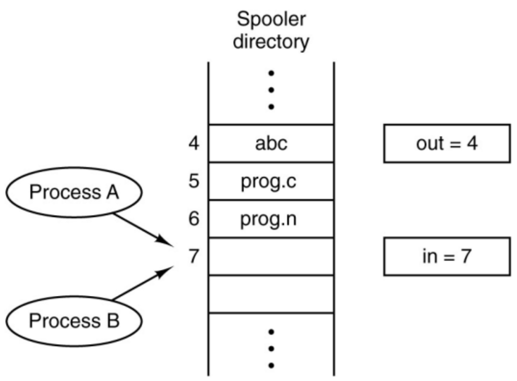
Interprocess Communication
(Race Condition)



- When a process want to print a file, it enter a file name in a special spooler directory
- Printer daemon periodically check spooler directory any file need to be printed.

Interprocess Communication
(Race Condition)

- Process A tried to send a job to spooler, Process A read *in = 7*, process A time out and go to ready state before updating *in = in + 1*.
- Process B tried to send a job to spooler. Process B read in = 7, load its job name in slot 7, update *i = i + 1 = 8* and then go to block state for waiting for job.
- Process A is rescheduled by scheduler. Process A already read *in = 7*, Process A load its job name in slot 7, update *i = i + 1 = 9* and then go to blocked state waiting for this job finish.



Interprocess Communication
(Race Condition)

- How to avoid race condition?

  Mutual exclusion – some way of making sure that <u>if one process is using a shared variable or file, the other processes will be excluded from doing the same thing</u>.
- The choice of the algorithm for achieving mutual exclusion is a major design issue in any operating system.
- **A solution** for the race condition <u>**should have** following four conditions</u>

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

Interprocess Communication
(Race Condition)

- Two approaches for mutual exclusion solutions.
  - Busy wait – A process will wait until resource become available or CPU time term expired.
  - Sleep and Wakeup- a process check a resource, if not available go to sleep. When the resource become available, the process will be waked up by system or the process release the resource.

Mutual Exclusion with Busy Waiting

- Each process has time term. When a process check the possibility to get into critical section
- Mutual Exclusion with Busy Waiting
  - Disabling Interrupts –non-preemptive kernel
  - Lock Variables
  - Strict Alternation
  - Peterson's Solution
  - Test and Set Lock – help from hardware

Mutual Exclusion with Busy Waiting (Disabling Interrupt – Nonprimitive Kernel)

Disabling Interrupt

- Once a process get into the critical section, interrupts set to disable.
- Other process cannot get CPU time until the process finish its job in the critical section.
- Since each user process has power to control interrupt, it might cause the end of system.
- We can build a simple program which can disable entire system since user has control system interrupt. (vulnerable system)

Mutual Exclusion with Busy Waiting (Disabling Interrupt – Nonprimitive Kernel)

Ex) End of the system with Disabling interrupt

1. A process get into the critical section.
2. It make disable all the interrupts – which means all other process are sleeping until the job is done in the critical section.
3. The process has blocked outside critical section just before make enable all the interrupts and never return again, cause end of the system.

Mutual Exclusion with Busy Waiting (Using Lock Variable)

- There are variable called "Lock"
  - A process can enter in its critical section when Lock = 0.
  - Lock =0 means no process is currently running in the critical section, set Lock =1 and enter in the critical section.
  - Once a process finish its job in critical section, set Lock = 0 and let other process in the critical section
  - Lock = 1 means there is a process running in the critical section, a process do busy waiting until Lock become 0.

Mutual Exclusion with Busy Waiting (Using Lock Variable)

```
int lock = 0;
repeat
    while lock ≠ 0do
          ; (no-operation) // Busy waiting
     lock = 1;
        Critical Section
      lock = 0;
        Remainder Section
until false
```

Mutual Exclusion with Busy Waiting
(Using Lock Variable)

```
int lock = 0;
repeat
    while lock ≠ 0 do
          ; (no-operation)
     lock = 1;
        Critical Section
      lock = 0;
        Remainder Section
until false
```

Scenario)

1. Initially lock = 0.
2. A process P_1 tries get into critical section. The process P_1 check lock value = 0.
3. Process P_1 CPU time is over and go to ready state, before updating lock = 1.
4. Process P_2 tries get into critical section. P_2 check lock value lock = 0
5. P_2 set lock = 1 and go to critical section.
6. P_2 CPU time is over and P_1 is rescheduled.
7. P_1 already read lock = 0, P_1 set lock = 1 and go to Critical section. Now P_1 and P_2 are in the critical section at the same time

Violating condition #1: mutual exclusion

Mutual Exclusion with Busy Waiting
(Strict Alternation)

- Variable turn can be i or j.
- if turn = i, process P_i can go to the critical section.
- Once P_i finish its job in critical section, P_i set turn = j, let process P_j enter critical section

```
turn is i or j
repeat
    while turn ≠ i do
          ; (no-operation)
        Critical Section
     turn = j;
        Remainder Section
    until false
```

Mutual Exclusion with Busy Waiting
(Strict Alternation)

```
turn is i or j
repeat
    while turn ≠ i do
          ; (no-operation)

        Critical Section

     turn = j;

         Remainder Section

    until false
```

Let assume initially turn = 0

1. P_0 is in CS while P_1 is in remaining section.
2. P_0 done C.S. and set turn = 1, P_1 is still in remaining section.
3. P_0 done remaining section and want to go to C.S. but turn= 1.
4. P_1 has fatal error in remainder section and trapped out by OS.
5. P_0 is waiting forever to enter the C.S.

Violating #2 condition
No process running outside its critical region may block other processes

Mutual Exclusion with Busy Waiting
(Peterson's Solution)

- Peterson's solution provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P_0 and P_1.
- For convenience, when presenting P_i, we use P_j to denote the other process; that is, j equals 1 − i.

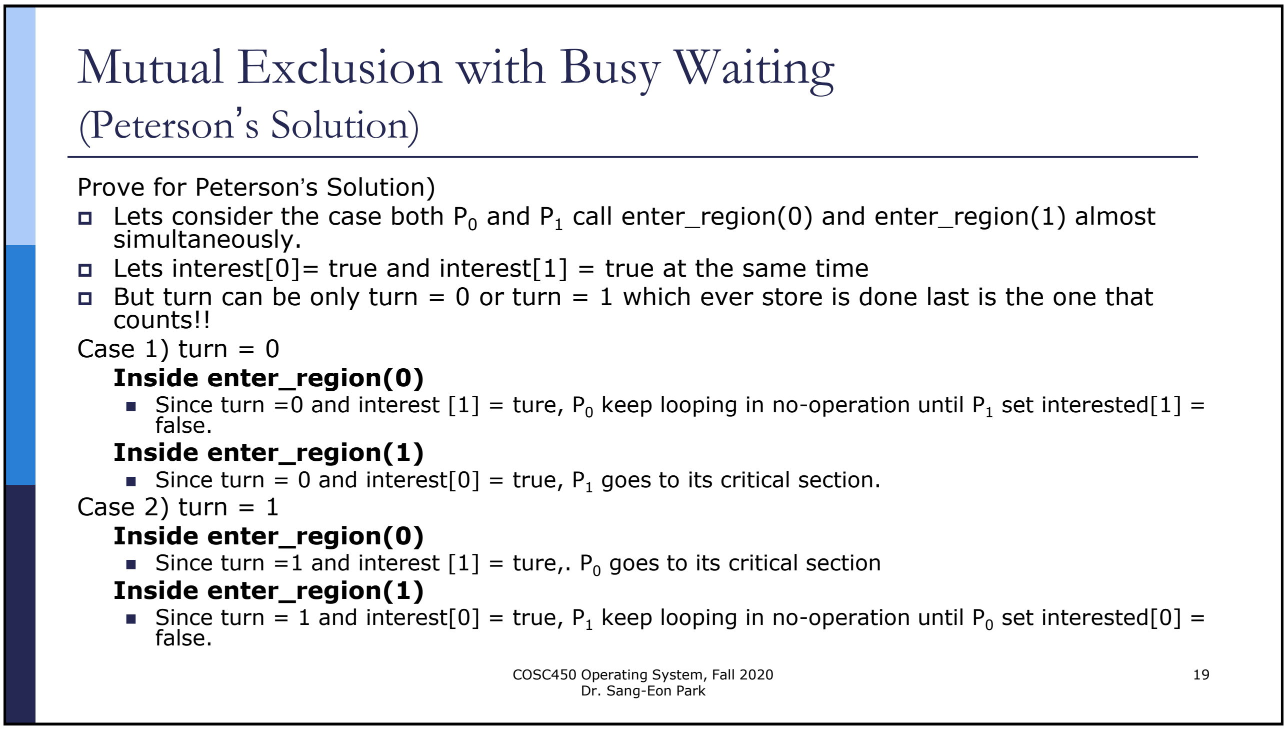
Mutual Exclusion with Busy Waiting (Peterson's Solution)

```
#define false 0
#define ture 1
#define n 2
int  turn
int interested[n]

void enter_region(int process);
{
    int other;
    other = 1 – process
    interested[process] = true
    turn = process;
    while (turn ==process && interest[other]==ture)
        ; /*no operation –busy waiting*/
}
void leave_region(int process)
{
    interest[process] = false;
}
```

```
void main()
{
  repeat
          enter_region (int i)

           Critical Section

          leave_resion (int i)

          Remainder Section

    until false
}
```

Mutual Exclusion with Busy Waiting (Peterson's Solution)

1. Initially, neither process is in the critical section
2. A process P_0 call enter_region (0)
   a) Set interested[0] = true;
   b) Set turn = 0
3. go to critical section
4. the process P_1 call enter_region(1) to get into its critical section
   a) set interested[1] = true;
   b) set turn = 1;
5. since interested[0] = true, it is keep looping for interest [0] = false
6. finally process P_0 finish its critical section and call leave_region(0)
   1. set interested[0] = false
7. now P_1 find out interest[0] = false, P_1 goes to its critical section



Mutual Exclusion with Busy Waiting
(Peterson's Solution)

Prove for Peterson's Solution)

- Lets consider the case both P_0 and P_1 call enter_region(0) and enter_region(1) almost simultaneously.
- Lets interest[0]= true and interest[1] = true at the same time
- But turn can be only turn = 0 or turn = 1 which ever store is done last is the one that counts!!

Case 1) turn = 0

Inside enter_region(0)

- Since turn =0 and interest [1] = ture, P_0 keep looping in no-operation until P_1 set interested[1] = false.

Inside enter_region(1)

- Since turn = 0 and interest[0] = true, P_1 goes to its critical section.

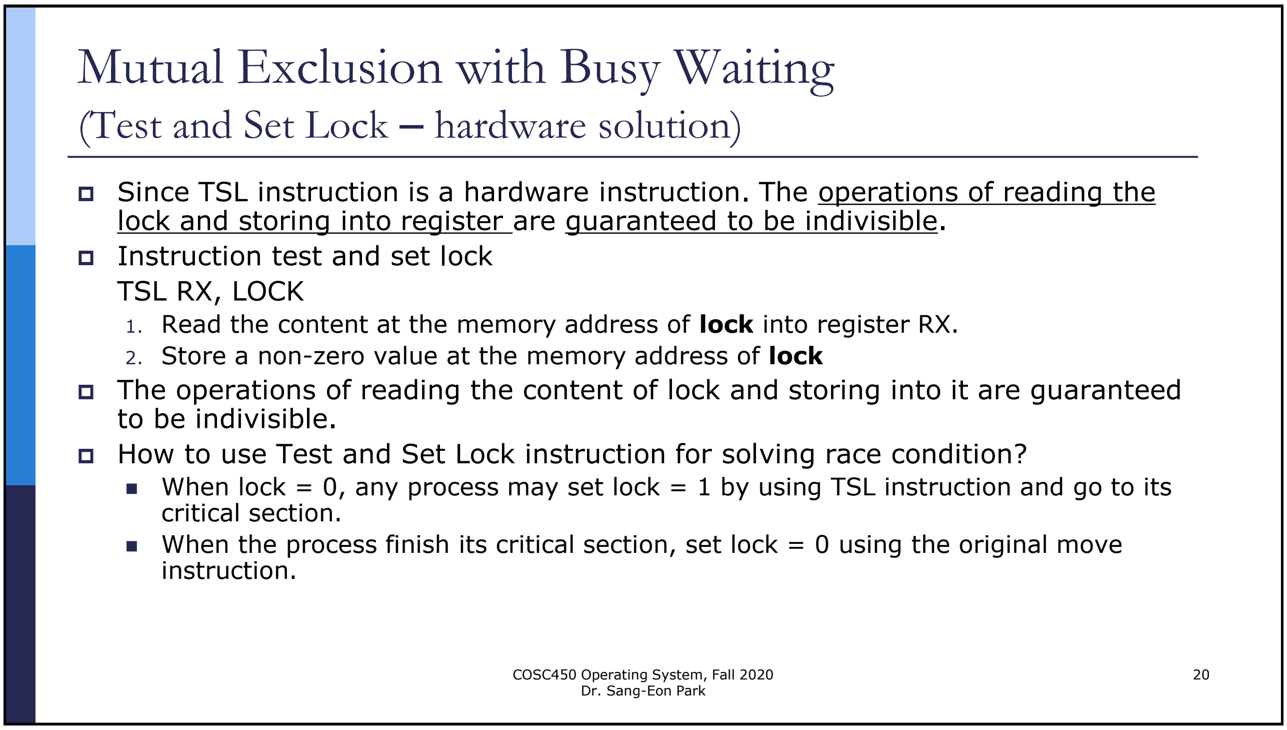
Case 2) turn = 1

Inside enter_region(0)

- Since turn =1 and interest [1] = ture,. P_0 goes to its critical section

Inside enter_region(1)

- Since turn = 1 and interest[0] = true, P_1 keep looping in no-operation until P_0 set interested[0] = false.



Mutual Exclusion with Busy Waiting
(Test and Set Lock – hardware solution)

- Since TSL instruction is a hardware instruction. The operations of reading the lock and storing into register are guaranteed to be indivisible.
- Instruction test and set lock

  TSL RX, LOCK

  1. Read the content at the memory address of **lock** into register RX.
  2. Store a non-zero value at the memory address of **lock**
- The operations of reading the content of lock and storing into it are guaranteed to be indivisible.
- How to use Test and Set Lock instruction for solving race condition?
  - When lock = 0, any process may set lock = 1 by using TSL instruction and go to its critical section.
  - When the process finish its critical section, set lock = 0 using the original move instruction.

Mutual Exclusion with Busy Waiting
(Test and Set Lock – hardware solution)

```
Enter_region
    TSL Register, Lock
    CMP Register, #0
    JNE Enter_region
    Set Lock, #1
    RET

Leave_region
    MOVE Lock, #0
    RET
```

```
Repeat
    Enter_region

        Critical Section

    Leave_region

        Remainder Section

until false
```

Mutual Exclusion with Busy Waiting
(Memory Barriers– hardware solution)

- Two general memory models
  - Strongly ordered Memory –a memory modification on one processor is immediately visible to all other processors
  - Weekly ordered Memory – a memory modification on one processor may not be immediately visible to other processors.
- With Strongly ordered memory, computer architectures provide instructions that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors.
- Such instructions are known as memory barriers or memory fences.

Mutual Exclusion with Busy Waiting
(Memory Barriers– hardware solution)

- A memory barrier is a type of barrier instruction that causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction.
- This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.

Mutual Exclusion with Busy Waiting
(Memory Barriers– hardware solution)

- Ex)
- Lets assume two threads are running on different part of program by sharing two variables. (bool flag = false; int x=0;)

Thread 1

```
while (!flag)
print x;
```

Thread 2

```
x = 100
flag = true;
```

- Thread 1 might print 0 or 100 depends on the order of execution.
- By using memory barrier instruction Thread1 always print 100.

Mutual Exclusion with Busy Waiting (Memory Barriers– hardware solution)

Thread 1

```
while (!flag)
   memory_varrier();
    print x;
```

Thread 2

```
x = 100;
Memory_barrier();
flag = true;
```

- Now it is guarantee that the value of flag is loaded before the value of x
- Also it is guarantee that assignment to x occurs before the assignment to flag.
- So Thread 1 always print 100

Mutual Exclusion with Busy Waiting (Atomic Variables– hardware solution)

- We can avoid mutual exclusion by using atomic operations.
- When a thread performs an atomic operation, the other threads see it as happening instantaneously. The advantage of atomic operations is that they are relatively quick compared to locks, and do not suffer from deadlock and convoying. The disadvantage is that they only do a limited set of operations, and often these are not enough to synthesize more complicated operations efficiently. But nonetheless you should not pass up an opportunity to use an atomic operation in place of mutual exclusion.

Mutual Exclusion with Busy Waiting
(Priority Inversion Problem)

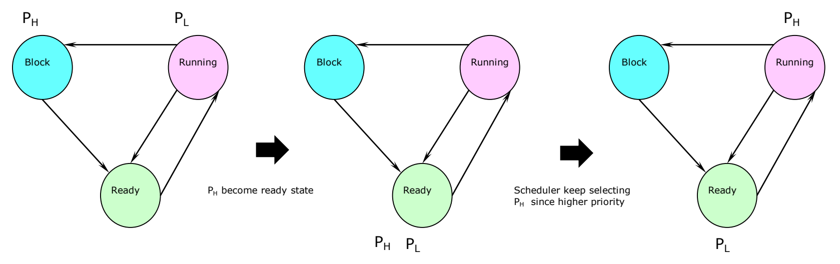
- Peterson's solution, test and set lock method – busy waiting – (wasting CPU time)

Priority Inversion problem with busy waiting method
- A computer with two processes P_H with high priorities, and P_L with low priorities.
- The scheduling rules are such that P_H get CPU time whenever it is in ready state.

1. At a time T_0: P_L is in critical section, and P_H is in block state.
2. At a time T_1: P_H change sate from block to ready state and try to enter the critical section. P_L still in critical section.
3. Based on scheduling rule, short-term scheduler select P_H, P_H hold CPU and try to enter into critical section.
4. Since P_L is in critical section, P_H run busy waiting outside critical section forever since P_L does not have a chance to get CPU time to finish its critical section.

Mutual Exclusion with Busy Waiting
(Priority Inversion Problem)

The scheduling rules : CPU scheduler will always select higher priority process.



T_0 :
P_L: running state in critical section
P_H: block state

T_1 : P_H become ready state
P_L: ready state in critical section
P_H: ready state
CPU scheduler will select higher priority process

$T_{i>1}$
P_L: ready state in critical section
P_H: busy waiting in running state