• Last class:
  – CPU Scheduling
• Today:
  – Start synchronization
Synchronization

• Processes (threads) share resources.
  – How do processes share resources?
  – How do threads share resources?

• It is important to coordinate their activities on these resources to ensure proper usage.
Resources

• There are different kinds of resources that are shared between processes:
  – Physical (terminal, disk, network, …)
  – Logical (files, sockets, memory, …)

• For the purposes of this discussion, let us focus on “memory” to be the shared resource
  – i.e. processes can all read and write into memory (variables) that are shared.
Problems due to sharing

• Consider a shared printer queue, $\text{spool\_queue}[N]$
• 2 processes want to enqueue an element each to this queue.
• $\text{tail}$ points to the current end of the queue
• Each process needs to do
  $\text{tail} = \text{tail} + 1$;
  $\text{spool\_queue}[\text{tail}] = \text{“element”}$;
What we are trying to do …

Process 1

\[ \text{tail} = \text{tail} + 1; \]
\[ \text{Spool\_queue}[\text{tail}] = X \]

Process 2

\[ \text{tail} = \text{tail} + 1; \]
\[ \text{Spool\_queue}[\text{tail}] = Y \]
What is the problem?

• tail = tail + 1 is NOT 1 machine instruction

• It can translate as follows:
  Load tail, R1
  Add R1, 1, R2
  Store R2, tail

• These 3 machine instructions may NOT be executed atomically.
Interleaving

- If each process is executing this set of 3 instructions, context switching can happen at any time.
- Let us say we get the following resultant sequence of instructions being executed:

  P1: Load tail, R1
  P1: Add R1, 1, R2
  P2: Load tail, R1
  P2: Add R1, 1, R2
  P1: Store R2, tail
  P2: Store R2, tail
Leading to …

Process 1

\[
tail = tail + 1;
Spool_queue[tail] = X
\]

Process 2

\[
tail = tail + 1;
Spool_queue[tail] = Y
\]
Race Conditions

• Situations like this that can lead to erroneous execution are called **race conditions**
  – The outcome of the execution depends on the particular interleaving of instructions

• Debugging race conditions can be fun!
  – since errors can be non-repeatable.
Avoiding Race Conditions

• If we had a way of making those (3) instructions atomic
  – i.e. while one process is executing those instructions, another process cannot execute the same instructions
  – then we could have avoided the race condition.

• These 3 instructions are said to constitute a critical section.
Requirements for Solution

1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the $N$ processes
How do we implement Critical Sections/Mutual Exclusion?

• Disable Interrupts
  – Effectively stops scheduling other processes.

• Busy-wait/spinlock Solutions
  – Pure software solutions
  – Integrated hardware-software solutions

• Blocking Solutions
Disabling Interrupts

• Advantages: Simple to implement

• Disadvantages:
  – Do not want to give such power to user processes
  – Does not work on a multiprocessor
  – Disables multiprogramming even if another process is NOT interested in critical section
How do we implement Critical Sections/Mutual Exclusion?

• Disable Interrupts
  – Effectively stops scheduling other processes.

• Busy-wait/spinlock Solutions
  – Pure software solutions
  – Integrated hardware-software solutions

• Blocking Solutions
S/W solns. with busy-waiting

• Overall philosophy: Keep checking some state (variables) until they indicate other process(es) are not in critical section.

• However, this is a non-trivial problem.
We have a race condition again since there is a gap between detection locked is FALSE, and setting locked to TRUE.
1. Strict Alternation

```
turn = 0;

P0 {
    while (turn != 0);
    critical section
    turn = 1;
}

P1 {
    while (turn != 1);
    critical section
    turn = 0;
}
```

It works!

Problems:
- requires processes to alternate getting into CS
- does NOT meet Progress requirement.
Fixing the “progress” requirement

bool flag[2]; // initialized to FALSE

P0 {
    flag[0] = TRUE;
    while (flag[1] == TRUE)
    {
        /* critical section */
        flag[0] = FALSE;
    }
}

P1 {
    flag[1] = TRUE;
    while (flag[0] == TRUE)
    {
        /* critical section */
        flag[1] = FALSE;
    }
}

Problem:
Both can set their flags to true and wait indefinitely for the other
Peterson’s Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process $P_i$ is ready!
2. Peterson’s Algorithm

int turn;
int interested[N];    /* all set to FALSE initially */

enter_CS(int myid) {    /* param. is 0 or 1 based on P0 or P1 */
    int other;
    otherid = 1 – myid;    /* id of the other process */
    interested[myid] = TRUE;
    turn = otherid;
    while (turn == otherid && interested[otherid] == TRUE)  ;
    /* proceed if turn == myid or interested[otherid] == FALSE */
}

leave_CS(int myid) {
    interested[myid] = FALSE;
}
Intuitively …

• This works because a process can enter CS, either because
  – Other process is not even interested in critical section
  – Or even if the other process is interested, it did the “turn = otherid” first.
Prove that

• It is correct (achieves mutex)
  – If both are interested, then 1 condition is false for one and true for the other.
  – This has to be the “turn == otherid” which cannot be false for both processes.
  – Otherwise, only one is interested and gets in
Prove that

• There is **progress**
  
  – If a process is waiting in the loop, the other person has to be interested.
  
  – One of the two will definitely get in during such scenarios.
Prove that

• There is **bounded waiting**
  – When there is only one process interested, it gets through
  – When there are two processes interested, the first one which did the “turn = otherid” statement goes through.
  – When the current process is done with CS, the next time it requests the CS, it will get it only after any other process waiting at the loop.
• We have looked at only 2 process solutions.

• How do we extend for multiple processes?
Multi-process solution

• Analogy to serving different customers in some serial fashion.
  – Make them pick a number/ticket on arrival.
  – Service them in increasing tickets
  – Need to use some tie-breaker in case the same ticket number is picked (e.g. larger process id wins).
3. Bakery Algorithm

Notation: \((a,b) < (c,d)\) if \(a < c\) or \(a = c\) and \(b < d\)

Every process has a unique id (integer) \(P_i\)

```c
bool choosing[0..n-1];
int number[0..n-1];

enter_CS(myid) {
    choosing[myid] = TRUE;
    number[myid] = max(number[0], number[1], ..., number[n-1]) + 1;
    choosing[myid] = FALSE;
    for (j=0 to n-1) {
        while (choosing[j]);
        while (number[j] != 0) &&
            ((number[j],Pj)<(number[myid],myid));
    }
}

leave_CS(myid) {
    number[myid] = 0;
}
```
Exercise

• Show that it meets
  – Mutex
  – Progress
  – Bounded waiting requirements
Where are we?

• Disable Interrupts
  – Effectively stops scheduling other processes.
• Busy-wait/spinlock Solutions
  – Pure software solutions
  – Integrated hardware-software solutions
• Blocking Solutions
Complications arose because we had atomicity only at the granularity of a machine instruction, and what a machine instruction could do was limited.

Can we provide specialized instructions in hardware to provide additional functionality (with an instruction still being atomic)?
Specialized Instructions

• `Bool Test&Set(bool)`

• `Swap (bool, bool)`

• Note that these are machine/assembly instructions, and are thus atomic.
Test&Set

Atomic bool Test&Set(bool x) {
    temp = x;
    x = TRUE;
    return (temp);
}

• Note that “=x” and “x=“ would have required at least 1 machine instruction each without this specialized instruction.
Using Test&Set()

Bool lock;

Enter_CS() {
    while (Test&Set(lock));
}

Exit_CS() {
    lock = FALSE;
}

NOTE: This solution does not guarantee bounded Waiting.

EXERCISE: Enhance this Solution for bounded waiting
Swap()

Atomic Swap(bool a, bool b) {
    temp = a;
    a = b;
    b = temp;
}

• Again, all this is done atomically!
Using swap()

Bool lock;

Enter_cs() {
    key = TRUE;         /* local var */
    while (key == TRUE) swap(key,lock);
}

Exit_cs() {
    lock = FALSE;
}
Where are we?

• Disable Interrupts
  – Effectively stops scheduling other processes.

• Busy-wait/spinlock Solutions
  – Pure software solutions
  – Integrated hardware-software solutions

• Blocking Solutions
Spinning vs. Blocking

- In the previous solns., we busy-waited for some condition to change.
- This change should be effected by some other process.
- We are “presuming” that this other process will eventually get the CPU (some kind of pre-emptive scheduler).
- This can be inefficient because:
  - You are wasting the rest of your time quantum in busy-waiting
  - Sometimes, your programs may not work! (if the OS scheduler is not pre-emptive).
• In blocking solutions, you relinquish the CPU at the time you cannot proceed, i.e. you are put in the blocked queue.

• It is the job of the process changing the condition to wake you up (i.e. move you from blocked back to ready queue).

• This way you do not unnecessarily occupy CPU cycles.
Example Blocking Implementation

Enter_CS(L) {
    Disable Interrupts
    Check if anyone is using L
    If not {
        Set L to being used
    }
    else {
        Move this PCB to Blocked queue for L
        Select another process to run
        from Ready queue
        Context switch to that process
    }
    Enable Interrupts
}

Exit_CS(L) {
    Disable Interrupts
    Check if blocked queue for L is empty
    if so {
        Set L to free
    }
    else {
        Move PCB from head of Blocked queue of L to Ready queue
    }
    Enable Interrupts
}

NOTE: These are OS system calls!
Until now …

• Exclusion synchronization/constraint
  – Typical construct mutual exclusion lock
    • Mutex_lock(m)
    • Mutex_unlock(m)

  – Do a man on pthread_mutex_lock() on your Solaris/Linux machine for further syntactic/semantic information.
Summary

• Synchronization
  – Exclusive access to critical sections
    • Mutual exclusion, progress, bounded waiting

• Approaches
  – Disable interrupts
  – Software only
  – Hardware-enabled
  – Spinning vs Blocking
• Next time: Synchronization