澳 門 理 工 學 院
Instituto Politécnico de Macau
Macao Polytechnic Institute

School of Applied Sciences (B.Sc. in Computing)

Notes #5: Mutual Exclusion and Synchronization

COMP 213

(211/212)

Operating Systems

2019-2020 1st Semester

Review

- Multiprogramming
- Multithreading
- Multi-processor
- User-level and kernel-level threads
- Most OSs have `pthread` implementations

Eddie Law

Topics

- Concurrency
- Mutual exclusion
- Synchronization
- Chapters 5.1 – 5.6, Appendices A.1, A.3, + this notes

On User-level Threads

- Some calls for `pthread` (e.g., C's APIs)
  - `#include <pthread.h>`
  - `pthread_create(…)`
  - `pthread_join(…)`
  - `pthread_exit(…)`

  - `#include <sched.h>`
  - `sched_yield(void)`
- How does a dispatcher get control of CPU back?
  - Internal events: thread returns control voluntarily
  - External events: thread ***gets preempted***

Internal Events

- Blocking on I/O
  - Requesting I/O implicitly yields the CPU
- Waiting on a "signal" from other thread
  - A thread asks to wait, and thus yields the CPU
- Thread executes a yield,
  i.e., `sched_yield()` ← a thread volunteers to give up CPU

```
thread1() {
        while(TRUE) {
                repeatDoingSameThingForAwhile();
                sched_yield();
        }
}
```

Eddie Law

External Events

- In a multi-threaded system, if a thread does not release CPU, then the dispatcher can regain control through ***external events***
- Examples:
  - I/O Interrupts
  - Timer – a supervisor call
- External events should occur frequently enough to ensure dispatcher runs
  - That is, fine enough scaled time quantum

Eddie Law

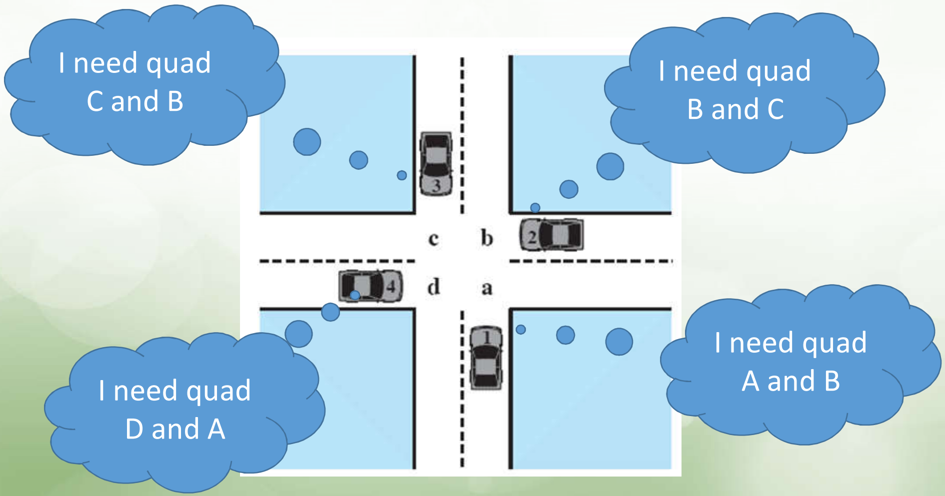
Key Terms Related to Concurrency

- **Deadlock**
  - Some guys are locked or held unmovable forwardly
- **Livelock**
  - Something like déjà vu, coming back to the same places again and again
- **Starvation**
  - Being ignored and overlooked indefinitely, there are always some favourites in front
- **Critical section**
  - A section is shared by many
- **Race condition**
  - Many participants in a race, but the “winner” could be the “loser”??
- **Mutual exclusion**
  - Like going to a single stall restroom …

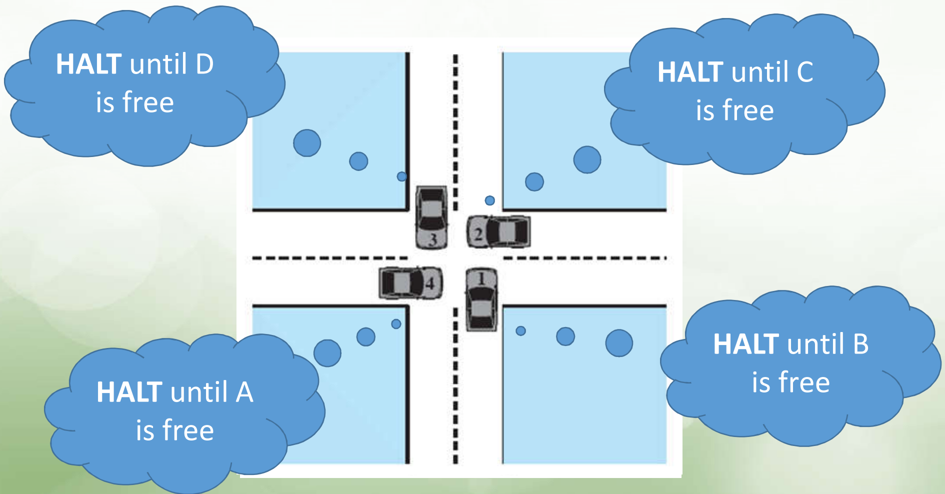
Deadlock

- Permanent blocking of a set of processes that compete for system resources
  - Example: Two processes P1 and P2 require **both** resources R1 and R2 to perform some operations. Suppose P1 obtains R1 and P2 obtains R2...

Potential Deadlock



Actual Deadlock



Starvation

- A process can never obtain access to resources it needs
  - Example: Processes P1, P2 and P3 require periodic access to resource R
  - However, the OS only assigns access to P1 and P2

Livelock

- Two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work
  - Example: when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time

Systems

In the following, "***thread***" and "***process***" might be used interchangeably. It should be self-explanatory if they should mean differently

- Uni-processing and ***multiprocessing*** systems
- Concurrency on threads (and processes)
  - ***Multiprogramming***
  - ***Multithreading***
- There are unrelated processes, and there are cooperative processes
  - Communication among processes
  - Sharing resources
  - Synchronization of multiple processes
  - Allocation of processor time

Eddie Law

What to Consider for Concurrency?

- What are the shared **global** resources?
- How to manage the resource allocation **optimally**?
- Is it difficult to locate programming errors? Is it time consuming to do debugging!?
  - Not necessarily any syntax bugs; let's have a look …

Eddie Law

A Simple Example

- Unrelated processes: echo characters to screen typed on keyboard
  - Suppose `get_char` reads a byte from keyboard
  - And `put_char` prints a character to screen

```
#include <stdio.h>
void echo() {
    c_in = get_char();
    c_out = c_in;
    put_char(c_out);
}
```

Any syntax error?

Eddie Law 15

A Simple Example (cont'd)

- (uniprocessor or multiprocessors) Two instances of the same program are running

Process P1	Process P2
.	.
S1: c_in = get_char()	.
.	S1: c_in = get_char()
.	S2: c_out = c_in;
S2: c_out = c_in;	.
S3: put_char(c_out)	.
.	S3: put_char(c_out)

time

- Global variables: interactions of processes!

Eddie Law 16

Another Example: Race Condition

- Another example on sharing global data

Process P1		Process P2
S1: $a=a+1$;	↓ time	S3: $b=2*b$;
S2: $b=b+1$;		S4: $a=2*a$;

FYI, in fact, an instruction of a high-level programming language could be translated into multiple machine code

- If inputs $a = b$, consistent identical results should be expected for each process
- The processes run simultaneously, could be,

$a = a + 1$;	$b = 2 \times b$;	$b = 2 \times b$;	$a = a + 1$;
$b = 2 \times b$;	$a = a + 1$;	$a = a + 1$;	$b = b + 1$;
$b = b + 1$;	$b = b + 1$;	$a = 2 \times a$;	$b = 2 \times b$;
$a = 2 \times a$;	$a = 2 \times a$;	$b = b + 1$;	$a = 2 \times a$;

Suppose a = b = 1 when started?

This execution order corrupts the stack. The data is not stored properly. This is called '**Lost update problem**'.

Eddie Law

Functions of an OS for Concurrency

- Since OS decides which process to run, which processes to stop; OS should:
- Allocate and deallocate resources for each active process
  - Processor time
  - Memory
  - Files
  - I/O devices
- Keep track of various processes
- Protect the data and physical resources of each process against interference by other processes
- Output of a process must be independent of the speed of execution of any other concurrent processes

Eddie Law

Interactions among Processes

- Processes sharing resources
- Awareness among them: 3 situations

Degree of Awareness	Relationship	Influence that one process has on the other	Potential control problems
Processes **unaware** of each other	Competition	1. Results of one process independent of others' actions 2. Timing of process may be affected	1. Mutual exclusion 2. Deadlock (renewable resource) 3. Starvation
Processes **indirectly aware** of each other (e.g., shared object, I/O buffer)	Cooperation by sharing	1. Results of one process may depend on information obtained from others 2. Timing of process may be affected	1. Mutual exclusion 2. Deadlock (renewable resource) 3. Starvation 4. Data coherence
Processes **directly aware** of each other (have communication primitives available to them)	Cooperation by communication	1. Results of one process may depend on information obtained from others 2. Timing of process may be affected	1. Deadlock (consumable resource) 2. Starvation

Eddie Law

Processes with Associations

- 1) Define problems, and 2) provide solutions
  - **Mutual exclusion**
  - **Semaphore**
    - **Binary**
    - **Counting or general**
  - **Monitors**
  - **Producer/consumer problem**
  - **Readers and writers problem**
  - **Dining philosophers** (some books used the name “dinning lawyers”)

Eddie Law

On Setting up Mutual Exclusion

- First of all, identify all ***critical sections***
- Processes may access **shared data** simultaneously — Examples were observed!
  - Who is the winner!? The ***slow*** one or the ***fast*** one… <---- There are "race conditions"
- Different orders of active processes give different results <---- Not desiarable
  - Shared memory is the "**critical section**"
- Then, what to do with critical section??
  - Only ***one process at a time*** is allowed to go in the critical section
- Without control, potential problematic consequences are, e.g.,
  - ***Deadlock***, ***livelock***, ***starvation***

Eddie Law

Critical Region: Some Terms

- **Mutual Exclusion**:
  - *When a process is in the critical session, then no other processes can execute within the critical section*
  - Like I lock it, no one else can come in

- **Progress**:
  - If no process is in critical section and several processes are trying to get in this critical section, then entry to the critical section cannot be postponed indefinitely
  - No process running outside the critical region can block other processes

Eddie Law

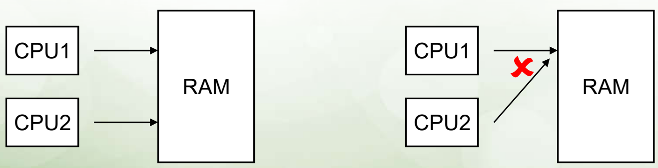
Critical Region: Some Terms (cont'd)

- **Bounded Wait**:
  - A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical section
  - No process should have to wait forever to enter its critical region (a.k.a. ***starvation***)
- **Speed and Number of CPUs**:
  - No assumptions should be made about the speeds or the number of CPUs
  - That is, they should not be factors

Eddie Law

Basic Assumption on Multiprocessors

- Only one access to a memory location can be made at a time



Read/Write different location at the same time is allowed.

Read/Write the same location at the same time is **NOT allowed**

Eddie Law 24

Mutual Exclusion

Mutual Exclusion

- Similar to a binary logic
  - Either ***you have it***, or ***you don't have it***
  - Either ***true***, or ***false***
  - Either ***1*** or ***0***
  - …
- How to fit mutual exclusion designs into the ***access controls***?

Eddie Law

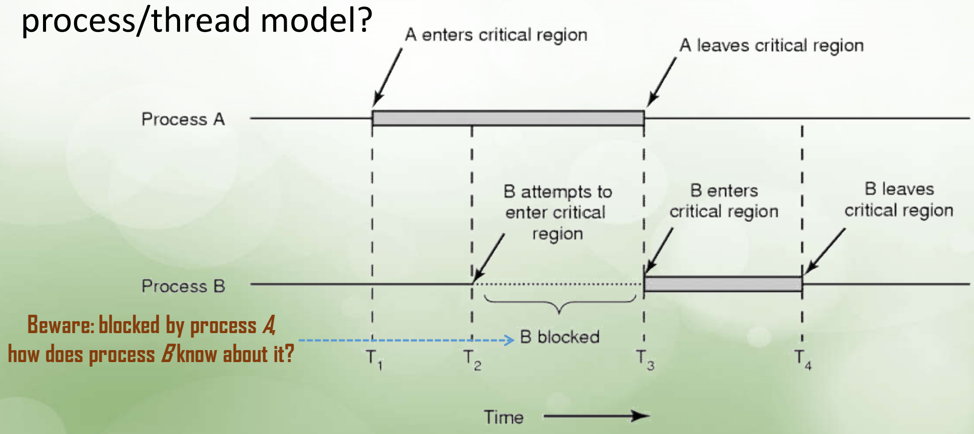
Requirements for Mutual Exclusion

- Based on the 4 conditions on critical sections
  - Only one process at a time is allowed in the critical section for a resource
  - If a process halts outside its critical section, it must not interfere with other processes
  - A process requiring the critical section must not be delayed indefinitely; no deadlock or starvation
  - A process must not be delayed access to a critical section when there is no other process using it
  - No assumptions are made about relative process speeds or number of processes
  - A process remains inside its critical section for a finite time only

Eddie Law 27

The Blocked and Ready States

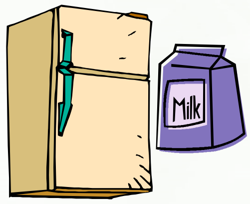
- Does the "critical section" relate to the "blocked" state in the process/thread model?



Mutual exclusion with critical regions

Eddie Law

Motivation: “Too much milk”



- Analogy between problems in OS and problems in real life
  - Help you understand real life problems better
- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk in fridge	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk in fridge

Too much milk!

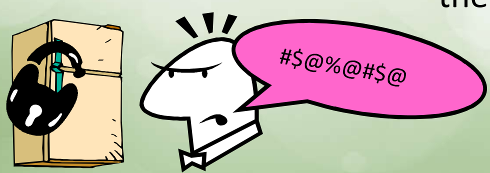
Eddie Law

The Sharing Milk Problem

- Sharing milk problem
- The ***critical section*** would be, like

```
if (noMilk) {
   buy milk;
}
```

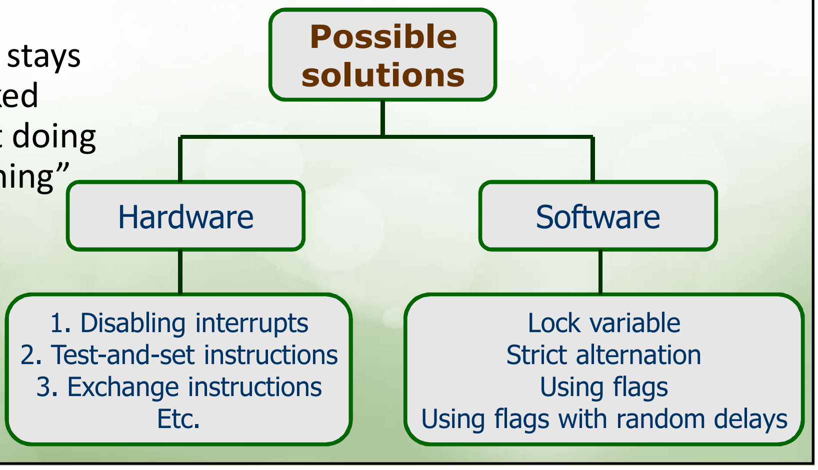
- No milk, lock the fridge and go buy → Other no idea if there is milk or not!



Eddie Law

Using Busy Waiting Methods

- OK, now how to lock the critical section?
- Busy waiting – process stays outside a properly locked "critical section," is not doing anything if in the "running" state



1 Hardware Solutions: Disabling Interrupts

- How do they work??
- ***Disabling interrupts*** guarantees mutual exclusion
  - Disable all interrupts just after entering a critical section and re-enable them just before leaving it
- Why it works??
  - E.g., disable the clock interrupts
  - OS only switches from one process to another upon clock or other interrupts, and with interrupts disabled, no switching can occur
  - No multiprocessing

External interrupts

Disabling Interrupts (cont'd)

- Potential problems:
  - Processor is limited in its ability to interleave programs
  - What if the process forgets to enable the interrupts?
  - Multiprocessor? Disabling interrupts only affects one CPU and not guarantees mutual exclusion
- Should only be used inside kernel

2 Hardware and Software

- Special machine instructions offered by chip vendors
  - Software solutions with hardware supports
  - ***Must be performed in one single instruction cycle*** <---- We call it "atomic"
  - Access to the memory location is blocked for other processes
- What are they?
  - ***Test-and-set***
  - ***Compare-and-swap***
  - ***Exchange***

Special Machine Instructions

- ***Test-and-Set*** instruction for an **atom**ic hardware operation
  - Conceptually, to think "test-and-set" as a routine with pseudo code

```
boolean testset (int i) {
    if (i == 0) {     // not set
        i = 1;        // we set it to 1
        return true;  // successful
    }
    else {             // is set already
        return false;  // failed
    }
}
```

Eddie Law 35

TSL Instruction

- The operation …

```
enter_region:
  tsl register,lock    | copy lock to register, then set lock to 1
  cmp register,#0      | was lock zero?
  jne enter_region     | if was non-zero (another using it), lock was set, loop
  ret                  | back to caller, critical region entered

leave_region:
  move lock,#0         | store a 0 in lock
  ret                  | return to caller
```

jne: Jump if not equal

Eddie Law

comp&swap() Machine Instruction

- Another **atomic** hardware instruction
- Conceptually, to think the "**compare-and-swap**" instruction in a software routine

```
boolean comp&swap(int register, int memory) {
      int temp;

      temp = memory;            // old data
      memory = register;        // put in new data
      if (temp != register) {   // old <> new; e.g., or (temp==0)
              register = temp;
              return(true);      // successful
      } else {                   // old == new
              return(false);     // fails
      }
}
```


Using **testset()** Instruction

Examine the while loop
==================
If *have_milk = 1*, then have milk
Not modified and returns false,
∴ testset(have_milk)=0, or
(!testset(have_milk))=1
∴ Loop the while(1) loop!

If *have_milk = 0*, then no milk
Then it is set and returns true,
∴ testset(have_milk)=1, and
(!testset(have_milk))=0
∴ Pass the while(0) loop!


```
/* testset mutual exclusion */
const int n; /* # of processes */
int have_milk;
void P(int i) {
    while (true) {
        while (!testset(have_milk)) {       enteringCS()
            /* do nothing */
        }
        critical_section(); /* buy milk */
        have_milk = 0; /* empty milk */  ← leavingCS()
        non_critical_section();
    }
}
void main()
{
    have_milk = 0; /* 0 is no milk */
    parbegin(P(1), P(2), ..., P(n));
}
```

Using **comp&swap()** Instruction

Examine the while loop
==================
If *have_milk = 1*, then have milk
Same value, no changes needed
∴ comp&swap(1,have_milk)=0, or
(!comp&swap(1,have_milk))=1
∴ Loop the while(1) loop!

If *have_milk = 0*, then no milk
Different numbers, swap,
and returns true,
∴ comp&swap(1,have_milk)=1, or
(!comp&swap(1,have_milk))=0
∴ Pass the while(0) loop!

```
/* comp&swap M.E. */
const int n; /* # of processes */
int have_milk;
void P(int i) {
    while (true) {
        while (!comp&swap(1,have_milk)){
            /* do nothing */
        }
        critical_section();
        comp&swap(0, have_milk);
        non_critical_section();
    }
}
void main()
{
    have_milk = 0; /* 0 is no milk */
    parbegin(P(1), P(2), …, P(n));
}
```


The **exchange()** Atomic Instruction

- Exchange instruction:
  - Push the logic control into processes
- All processes have ***keys***
- Only ***one token*** (***have_milk = 0***) ***for all processes to gain access***
  - Simplifies the exchange procedure
  - Returning token is nonzero if taken

- Conceptually in software

```
void exchange(int reg, int mem) {
    int temp;
    temp = mem;
    mem = reg;
    reg = temp;
}
```

Using **exchange()** Instruction

```
/* exchange */
const int n; /* # of processes */
int have_milk;
void P(int i) {
    int keyi;
    while (true) {
        keyi = 1;
        while (keyi != 0) {
            exchange (keyi, have_milk);
        } /* waiting if keyi is not zero */
        critical_section();
        exchange(keyi, have_milk);
        non_critical_section();
    }
}
```

```
void main()
{
    have_milk = 0; /* 0 is no milk */
    parbegin(P(1), P(2), ..., P(n));
}
```


Using Machine Instructions

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - Simple and therefore easy to verify
  - Can be used to support multiple critical sections

Using Machine Instructions (cont')

- Disadvantages
  - Busy waiting
    - ***Starvation*** is possible if multiple processes are waiting
    - Because of ***no explicit waiting queue control***
  - Wasting CPU time, the resource
  - Deadlock
    - A high priority waits for a low priority to leave the critical section
    - The low priority can never execute if the high priority is holding something the low priority one needs
    - The solution: **the priority inversion** (discussed later)

Eddie Law

Possible Software Solutions

- Evaluating solutions
  1. Lock variable
  2. Strict alternation
  3. Using flags
  4. Using flags with random delays
- Algorithms
  - Dekker's
  - Peterson's
  - Lamport's barkery

Eddie Law

A Common Problem

- Problem with most software solutions
  - The process dies inside the critical section
  - The state is not reset
  - Others cannot move in

1. Lock Variable

- A global variable, **lock**,

```
    // initialization
   int lock = 0;

    ...
    while (lock);   // busy wait
    lock = 1;
         Critical Section:
              the shared variable
    lock = 0;
```

- Does this code work?

Example

- P1 and P2, with **`lock`** = 0 when starts

Process P1

. one instruction cycle, *I*

*I*1:lock=0;

It is zero, let's go to CS

...

*I*3:while(lock);

.

Yes, I think I am safe with ME → *I*5:lock=1;

.

~~Running perfectly!!~~

time

Process P2

.

.

Yes, it is still zero, let's go to CS

*I*2:lock=0;

.

*I*4:while(lock);

.

*I*6:lock=1; ← *I thought I have exclusive access now*

A "lock" variable on CS failed, not ME!!

Eddie Law

My Turn and Your Turn

- Use a ***global variable*** to tell whose **`turn`** it is

```
int turn;
turn = my turn;
I go buy milk;
finish all milk;
turn = your turn;
```

EXAMINE IT!

Eddie Law

2. Strict Alternation, Me First

- Spin waiting

Other thread decides

```
thread me {      // many threads running
  while (true) {
    while (turn != my thread id);    // spin if true
    critical();
    turn = other thread id;
    non-critical();
  }
}
```

- “**turn**” assigned to a thread/process to access critical section
  - Using “**while {turn!=my_thread_id} {}** /* busy waiting*/”
  - The “**turn**” lock variable, uses busy waiting, is called a ***spin lock***

Eddie Law

Examining it!

- Remember the 4 criteria

- For “My Turn” method
  - Satisfies “**mutual exclusion**” – let other go in first
  - **No** “**progress**” – my **turn** may never come back, if other process does not enter critical section and reset it afterwards, but I want to enter now

Eddie Law 50

My "turn": Other Potential Problems

- Critical section is not fairly shared, a process can monopolize through **`critical()`**
- Still a centralized process (i.e., OS) to assign the value of "**`turn`**" (at least the initial one)
- If more than 2 processes/threads??
  - How to get to know a new roommate, and who you assign it to next??
  - Set "**`turn`**" to unknown thread, an unknown guy, then??
- If a process fails anywhere before changing "**`turn`**", another one may be permanently blocked

3a. A Process A Flag

- A ***personal*** "**`flag`**"
  - Controlled by a process to indicate that ***this process wants to enter critical section***

- More than a thread, be a nice guy
  - Check others' "**`flag`**"s and let other go first, then set my "**`flag`**" afterward

3a. A Process A Flag (cont'd)

- Starting it with
```
typedef enum {false, true} boolean;
boolean flag[2] = {false, false};
```
- Is it better than the "strict alternation"?

- Set my "`flag`" to "`true`" after "`while`" loop!


```
/* Process 0 */
…
while (flag[1]) {
    /* looping while(); */
}
flag[0] = true;
critical();
flag[0] = false;
…
```

```
/* Process 1 */
…
while (flag[0]) {
    /* looping while(); */
}
flag[1] = true;
critical();
flag[1] = false;
…
```

You're going?

If you don't,
I go

Interleaving instructions breaks M.E.

Big Problems??

- It does **not guarantee** mutual exclusion (worse than "strict alternation")
  - Similar to that in "`lock`" variable
  - Go through "**`while`**" statements in P_0 and P_1 in order
  - Go set "**`flag`**" to true in both P_0 and P_1
  - Both P_0 and P_1 enter `critical();`
- If one process dies outside `critical()`
  - Others are okay
- But if a process dies inside `critical()`
  - Others are blocked permanently, but M.E. still works

3b. Using Flags: Another Design

- Using flags, set my "`flag`" before the "`while`" loop

```
int flag[2] = {false, false};
thread me {
   while (true) {
      flag [my thread id] = true;
      while (flag [other thread id]) {
              /* looping while */
      }
      critical();
      flag [my thread id] = false;
      non-critical();
   }
}
```

Last design, set "flag" after the "while" loop

- Does it work?

Eddie Law 55

3b. A Flag A Process (cont'd)

- Set "`flag`" to "`true`" before "`while`" to avoid previous mistake!


```
/* Process 0 */                 /* Process 1 */
…                               …
flag[0] = true;                 flag[1] = true;          ← I go
while (flag[1]) {               while (flag[0]) {        ← Do you go?
    /* looping while(); */          /* if yes, I wait here */
}                               }
critical();                     critical();
flag[0] = false;                flag[1] = false;
…                               …
```

Deadlock can occur

Eddie Law 56

Problems??

- Again, can block indefinitely!
  - Common problem for "using `flags`"
  - If process dies in `critical()`, then other processes are blocked
- Again, M.E. is satisfied
  - If P_0 sets "`flag[0]`," other hasn't set "`flag[1]`", so OK
  - Go through "`flag`" in order, but both cannot pass the "`while`" loop
  - Yes to M.E., but creates ***deadlock*** as discussed
  - Problem if both "`flag`"s are set
- But, ***deadlock*** cannot be reset as both processes stay permanently in the "`while`" loops ← What can we do to solve it?

4. Add Random Delays (Based on 3b)

```
/* Process 0 */
…
flag[0] = true;
while (flag[1]) {
   flag[0] = false;
   delay();
   flag[0] = true;
}
critical();
flag[0] = false;
…
```

```
/* Process 1 */
…
flag[1] = true;
while (flag[0]) {
   flag[1] = false;
   delay();
   flag[1] = true;
}
critical();
flag[1] = false;
…
```

Solve the Deadlock?

- Introduce "**`flag`**" **reset** with random delay to solve the deadlock
- But problems:
  - With probability, the durations of the two **`delay()`** operations could be identical $\Rightarrow$ ***Livelock***!!
- Now how to solve it?
  - The "mutual courtesy" issue!?
  - Different seeds for **`delay()`**'s

Recap on Software Solution Attempts

1. Global "lock" variable for critical section → M.E. failed
2. Courtesy set "turn" variable to other thread → **M.E. ok**, but progress problem, if other does not set it to me
3. A thread a flag, check other's flag firstly before setting my own flag and entering CS → M.E. failed
4. Set my flag before checking other's flag → Deadlock!
5. Set my flag before checking other's flag; if other's flag is also positive, then reset my own flag, add a delay before resetting my flag → **M.E. works**, livelock if delays are identical!

→ No solutions so far

Algorithmic Approaches

- Learn through using "lock", "alternation", "individual flags", "random delay setting flags"
  - Each of them not works well by itself, two cases (designs 2 & 4) can do M.E.
  - Find ways to mix them!!
- Working algorithms
  - Dekker's algorithm
  - Peterson's algorithm
  - Lamport's bakery algorithm

Eddie Law

Dekker's Algorithm

- For two processes
- Q: Can the "**`delay()`**" in the last design be removed??
- Learned from "strict alternation"
  - Add a global "**`turn`**" variable, M.E. works
  - Use ID with courtesy
    - Let other enter first
  - But "no progress" problem

```
…
while (turn != my thread id);
critical();
turn = other thread id;
…
```

- **Dekker's algorithm**
  - Modify the case 4
  - Combining "**`turn`**" and "**`flag`**"
  - One "**`flag`**" per process
  - Plus a global variable "**`turn`**"
    - "**`turn`**" is used only if both "**`flag`**"s are set
    - Avoid "no progress" problem

```
…
flag[0] = true;
while (flag[1]) {
    flag[0] = false;
    delay();
    flag[0] = true;
}
…
```

Eddie Law

Dekker's Algorithm: Pseudo Code

Changes replaces the delay()

```
/* Process 0 */                          /* Process 1 */
flag[0] = true;                          flag[1] = true;
while (flag[1]) {                        while (flag[0]) {
    if (turn==1) {                           if (turn==0) {
        flag[0] = false;                         flag[1] = false;
        while (turn==1) {} // looping            while (turn==0) {} // looping
        flag[0] = true;                          flag[1] = true;
    }                                        }
}                                        }
critical();                              critical();
turn = 1;                                turn = 0;
flag[0] = false;                         flag[1] = false;
non-critical();                          non-critical();
```

Same old design 4's conditioning cases

"turn" inside the while() loop

What changes made to the case 4?

Global unique variable From design 2

Conditioning cases

Go inside while() only if both processes want to get in CS

Eddie Law 63

Dekker's Algorithm: 4 Conditions

- Proof on 4 conditions: M.E., progress, bounded wait, # CPUs
- Satisfy "**progress**" - selected process entering `critical()` is not delayed indefinitely
    1. If only one process enters `critical()`, `flag` by itself works, `turn` = 0 or 1 is not a factor
    2. If two processes want to enter `critical()`, then `turn` is used
    - Sufficient to consider all possible cases for P_0, conditioning cases are:
    - P_0 can enter only if `turn=0` (i.e., P_0's turn, set by P_1)
    - (`turn=0`, and `flag[0]=false`) are impossible, won't enter the "conditioning cases" part, the "while" loop
    - (`turn=0`, and `flag[0]=true`), then P_0 must enter

Eddie Law 64

Dekker's Algorithm: 4 Conditions (cont'd)

- "**Mutual Exclusion**" is enforced upon entering `critical()` if
  - {`flag[i]` and (`!flag[1-i]`)} is TRUE, where `i`=process 0 or 1
  - P_0 will loop within the `while()`, only if `turn`=1, this implies P_1 is or can go inside the `critical()`
  - `turn` is not a real factor here → cannot cause "no progress"!!

```
int main() {
    flag[0] = false;
    flag[1] = false;
    turn = 1;          ← Is it okay? Yes, P0 can go in, even if turn = 1
    parbegin(P0, P1);
}
```

Eddie Law 65

Dekker's Algorithm: 4 Conditions (cont'd)

- "**`turn`**" used but doesn't suffer the permanent block if a process dies outside **`critical()`**. Why? How?
  - "**`turn`**" is known for replacing the `delay()` only…
  - Not blocking as it is uniquely set, either 0 or 1…
  - "**`turn`**" is assigned through an OS or main
- Fairness issue in **`critical()`** section
  - ***Bounded-wait*** may have trouble

Eddie Law 66

Peterson Algorithm

- Issues with Dekker's algorithm
  - Difficult to follow as "**`turn`**" set by processes
  - Hierarchical single parameter conditioning events
  - Dekker's statement: "I like to go in CS, but you can go first"
- Can we improve on Dekker's?
  - Also using "**`flag`**" and "**`turn`**"
  - Peterson's statement: "***I like to go in CS, but you can go first only if you also say you like to go in***"

Peterson's Algorithm: Pseudo Code

```
/* Process 0 */
while (1) {
  …
  flag[0] = true;
  // turn = 1;
  while (flag[1]&&(turn==1)) {} //loop
  critical();
  flag[0] = false;
  turn=1 // here, or above
  non-critical();
}
```

Unique variable per process

```
/* Process 1 */
while (1) {
  …
  flag[1] = true;
  // turn = 0;
  while(flag[0]&&(turn==0)) {} //loop
  critical();
  flag[1] = false;
  turn=0 // kinder here??
  non-critical();
}
```

P_0 says yes!
P_0's turn too!

Courtesy part

Courtesy part

Peterson's Algorithm: Proof

- "turn" plays active role, is set distributedly
- If no ***deadlock*** → then there is "**progress**"
- Consider that P_0 is blocked at "while" loop!
  - Only one possible case, i.e., must be positive for both "flag[1]=true" and "turn=1"
  - Otherwise, if either "flag[1]=false" or "turn=0" or both, P_0 must enter critical() immediately
- Consider following scenarios
  a) If P_1 is not interested in critical(), then flag[1]=false;
  b) If P_1 is waiting at while(), this is impossible, as "turn=1" set by P_0 → this implies P_1 can and must enter critical()
  c) P_1 uses critical() repeatedly and exclusively ← impossible given flag[0]=true because it must set "turn=0" after each round!

Eddie Law 69

Peterson's Algorithm: Proof (cont'd)

- **M.E.** is also preserved
  - If P_1 is not in critical(), as long as "flag[0]=1", P_1 cannot enter because P_1 sets "turn=0"
  - If P_1 is in critical(), then "flag[1]=1" must be set and P_0 cannot enter (similar reason as above)
- OS (or main routine) can assign "turn" → make critical() fairly shared among processes
  - This method achieves ***bounded-wait***

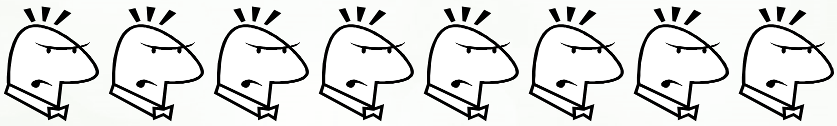
Eddie Law 70

Peterson's Algorithm: Observations

- "**`turn`**" makes **`critical()`** exclusive
  - A global unique variable
- The last one to set "**`turn`**" to other may not win
  - In fact, the last process assigned to "**`turn`**" can always win
- "**`turn`**" introduces delay only…
  - This is important!
  - Does not introduce permanent blocking if a process dies outside **`critical()`**
- Permanent blocking is still possible – the only case is
  - P_0 finishes "**`flag[0]=true;`**" and "**`turn=1;`**", OS runs "**`flag[1]=true;`**" instruction in P_1, then P_1 dies
  - P_0 is permanently blocked

Eddie Law

Lamport's Bakery Algorithm



- Too many customers
  - How to solve with Dekker's and Peterson's algo??
- Lamport: simple basic design
  - A process wants to enter critical section, takes a number and waits for its turn
- Real trick:
  - The process with the lowest number gets in first

Eddie Law

Bakery Algorithm (cont'd)

```
taking_ticket[i] = true;
my_ticket[i] = 0;
for (j=0; j<N; j++) {
    if (my_ticket[i] <= my_ticket[j])
        my_ticket[i] = my_ticket[j] + 1;
}
taking_ticket[i] = false;
for (j=0; j<N; j++) {
    while (taking_ticket[j]) {
        /* looping_while(); */
    }
    while ( {my_ticket[i] > my ticket[j]}
      OR {(my_ticket[i] == my_ticket[j])
      AND ( j < i )} ) {
        / * Looping_while(); */
    }
}
critical();
my_ticket[i] = ∞;
```

- *N* customers to buy bread
- Code shown for the user #*i*
- The first block is protected by "**taking_ticket[] = true**" for all processes
  - Problem: multiple processes may get the same ticket numbers
- This is solved with the second block
  - By using **process IDs**

Bakery Algorithm (cont'd)

- Q: What do you think if replacing the loop of choosing the next ticket with a global variable

```
My_ticket[i] = current_ticket;
current_ticket++;
```

- Not atomic… problem again…
  - Ticket number plays no role
  - Follows only process ID order
  - Recall the "lock variable" case