Operating Systems

Lecture 10: Synchronization

IIIS
Department of Computer Science & Technology
Tsinghua University

Outline

- Background
- Basic Concepts
- Critical Section
- Approach 1: Disabling Hardware Interrupt
- Approach 2: Software-based Solution
- Approach 3: Higher-level Abstractions

Multi-programming

- So far in this course
  - Multi-programming: an important feature of modern OS
  - Parallelism is good (why?)
    - Hint: multiple concurrent entities: CPU(s), I/O, …, users, …
  - Process/thread: OS abstractions to support multi-programming
  - CPU schedule: mechanism to realize multi-programming
  - Scheduling algorithms – different policies

- This and next week
  - Collaborative multi-programming and the concurrency problem

Correctness with concurrent threads

- Independent threads:
  - No state shared with other threads
  - Deterministic 6 Input state determines results
  - Reproducible 6 Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter
- Cooperating threads:
  - Shared state between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

Why allow cooperating threads?

- People cooperate, so computers/devices must cooperate
- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - Chop large problem up into simpler pieces
    - To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
  - Makes system easier to extend

Example: Creating a New Process ID

- A program calls fork() to create a new process
  - OS needs to assign a new and unique process ID
  - So somewhere in the kernel, this system call will do
    - new_pid = next_pid++ ;
  - Translating into machine instructions
    - LOAD next_pid Reg1
    - STORE Reg1 new_pid
    - INC Reg1
    - STORE Reg1 next_pid
- Assume two processes execute concurrently
  - If next_pid is 100, then one process should get 100, the other should get 101, and next_pid should increase to 102

Work correctly under all possible interleaving?

- Process 1
- Process 2

LOAD next_pid Reg1
STORE Reg1 new_pid

LOAD next_pid Reg1
STORE Reg1 new_pid
INC Reg1
STORE Reg1 next_pid

INC Reg1
STORE Reg1 next_pid

- Gets 100 as the new PID
  - next_pid becomes 101
- Gets 100 as the new PID
  - next_pid becomes 101

Concurrency: Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down behavior first
  - Impulse is to start coding first, then when it doesn’t work, pull hair out
  - Instead, think first, then code

Outline

- Background
- Basic Concepts
- Critical Section
- Approach 1: Disabling Hardware Interrupt
- Approach 2: Software-based Solution
- Approach 3: Higher-level Abstractions

Concept: Race Condition

- A flaw in the system where the outcome depends on the sequence/timing of concurrent executions or events
  - Like the previous example
  - Non-deterministic
  - Non-reproducible
- How do you avoid such race condition in an OS design?

Concept: Atomic Operation

- An atomic operation is one that executes to completion without any interruption or failure
  - Either it executes to completion, or
  - it did not execute at all, and
  - no one else should see a partially-executed state
- Operations are often not atomic
  - Many that we thought to be are not so by the computer
  - Not even a simple statement like “x++” !
    - Translated into a sequence of 3 instructions
  - Sometimes not even so for a machine instruction
    - Remember pipeline, super-scalar, out-of-order, page fault?

Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

Thread A	Thread B
i = 0;	i = 0;
while (i < 10)	while (i > -10)
i = i + 1;	i = i – 1;
printf(“A wins!”);	printf(“B wins!”);

  - Assume that memory loads and stores are atomic, but incrementing and decrementing are not atomic
- Who wins?
- Is it guaranteed that someone wins?
- What it both threads have their own CPU running at same speed?

Concepts

- Critical section
  - A section of code within a process that requires access to shared resources and which may not be executed while another process is in a corresponding section of code.
- Mutual exclusion
  - The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
- Deadlock
  - A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
- Starvation
  - A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Motivation Example: “Too much milk”

- Great thing about OS’s – analogy between problems and problems in real life
  - Ø Help you understand real life problems better
  - Ø But, computers are much stupider than people
- Example: People need to coordinate:

bep o	Hovf , d i	Hovf , d L
3sr r	n, , wed gve OoPTl k , Fp eaw	
3sr 7	noyho F, v f k, vo	
3s1r	i weho ykfk, vo	n, , wed gve OoPTl k , Fp eaw
3s17	Ll B p eaw	noyho F, v f k, vo
3s2r	i weho m, p outl k p eaw y: yB	i weho ykfk, vo
3s27		Ll B p eaw
3s3r		i weho m, p outl k p eaw y: yB

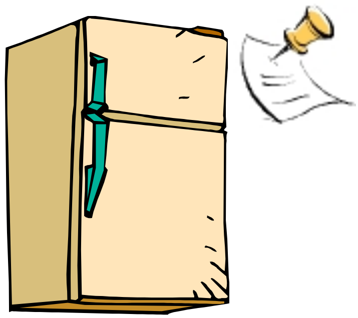
Too Much Milk: Correctness Properties

- What are the correctness properties for the "Too much milk" problem?
  - Never more than one person buys
  - Someone buys if needed
- Synchronization: finding a solution to this problem
  - Assume only LOAD and STORE are atomic
  - Important: all synchronization involves "waiting"
- For example, putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: what if someone only wants juice?
  - Of Course – how do we make this "lock" thing?

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if there is a note (i.e., wait until note is gone)
- Example program:

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove Note;
   }
}
```



- Does it work?

Solution #1

- Result
  - Still too much milk although only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
- Let's try a quick fix: just place note first

```
leave Note;
if (noMilk) {
   if (noNote) {
      buy milk;
   }
}
remove note;
```

- What happens here?
  - No one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

Thread A	Thread B
leave note A;	leave note B;
if (noNote B) {	if (noNoteA) {
if (noMilk) {	if (noMilk) {
buy Milk;	buy Milk;
}	}
}	}
remove note A;	remove note B;

- Does this work?

Solution #2

- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely that this would happen, but will at worse possible time
  - Probably something like this in UNIX
- This kind of lockup is called "starvation!"

Too Much Milk Solution #3

- A more complicated two-note solution:

Thread A

```
leave note A;
while (note B) {
    do nothing;
}
if (noMilk) {
    buy milk;
}
remove note A;
```

if no note B, safe for A to buy, otherwise wait for B to quit first

Thread B

```
leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

if no note A, safe for B to buy, otherwise A is either buying or waiting for B to quit

- Does this work now?
  - Yes. Either safe to buy, or other will buy so ok to quit
- But are you happy with the solution?

Solution #3 discussion

- It works, but it's really unsatisfactory
- Really complex – even for this simple an example
  - Hard to convince yourself that this really works
- A's code is different from B's
  - Code would have to be slightly different for each thread
  - What if lots of threads?
- While A is waiting, it is consuming CPU time
  - This is so called "busy-waiting"
- Is there a better way?

Goal is to Protect a Critical Piece of Code

- Solution #3 protects a single "critical-section" piece of code for each thread:

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

- A better way than solution #3
  - Have hardware provide better (higher-level) primitives than atomic LOAD and STORE
  - Build higher-level programming abstractions on this new hardware support

Too Much Milk: Solution #4

- Suppose we have some implementation of a lock
  - Lock.Acquire() – wait until lock is free, then grab
  - Lock.Release() – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then our milk problem is easy:

```
milklock.Acquire();
if (nomilk) {
  buy milk;
}
milklock.Release();
```

Outline

- Background
- Basic Concepts
- Critical Section
- Approach 1: Disabling Hardware Interrupt
- Approach 2: Software-based Solution
- Approach 3: Higher-level Abstractions

Where are we going with Synchronization?

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Programs	Shared Programs
OS Abstractions	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

Restricted Access to Critical Data/Resources

- Only through a piece of code segment
  - "Lock" before enter
  - Wait if already "locked"
  - Here comes the "critical section"
  - "Unlock" when done
- Important properties for this code segment
  - Can only be executed by one process/thread at a time
  - Must not be aborted (i.e., must eventually finish)
- Cooperative concurrent programs
  - Access data only through this code segment

Example Code Segment for Accessing next_pid

- Somewhere in the kernel, when serving fork() system call

```
…
ENTER_CRITICAL_SECTION
…
new_pid = next_pid++ ;
…
EXIT_CRITICAL_SECTION
…
```

Critical section

Code for entering and leaving the critical section

Critical Section

- An important concept in concurrent programming
  - A segment of important code involved in reading and writing a shared data area
  - Must be executed only by one process/thread at a time
- Key assumptions:
  - Finite Progress Axiom: Processes execute at a finite, but otherwise unknown, speed.
  - Processes cannot halt (by failing, or just terminating) inside critical section
- Used profusely in OS to protect data structures
  - Examples: queues, shared variables, lists, …

Critical Section Property

- Mutual exclusion: At most *k* threads are concurrently in the critical section (very often k is 1)
- Progress: A thread that wants to enter the critical section, will eventually succeed
- Bounded waiting: If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i’ s request is granted
- No busy waiting (optional): If a process is waiting for entering its critical section, it is suspended until it is permitted to enter.

Mechanisms for Implementing Critical Section

- Code for entering and leaving critical section
  - ENTER_CRITICAL_SECTION
  - EXIT_CRITICAL_SECTION
- Basic mechanisms
  - Disabling interrupt
  - Software solution (e.g., Peterson's algorithm)
  - Higher-level abstractions
- Comparing different mechanisms
  - Performance: concurrency level

Outline

- Background
- Basic Concepts
- Critical Section
- Approach 1: Disabling Hardware Interrupt
- Approach 2: Software-based Solution
- Approach 3: Higher-level Abstractions

Approach 1: Disabling Hardware Interrupt

- No interrupt, no context switch, hence no concurrency
  - Hardware delays the interrupt processing until interrupts are enabled again
  - Most modern computer architecture provides instructions to do this
- Entering critical section
  - Disable interrupts
- Exiting critical section
  - Enable interrupts

Disadvantage

- Once interrupts are disable, the thread can’t be stopped
  - Whole system put to a stop for you
  - Can starve other threads
- What if the critical section is arbitrarily long?
  - Can’t bound the amount of time needed to respond to interrupt (may have hardware implications)
- Must be used carefully!

Example: Disabling Interrupts in Linux

- Usually some small number of interrupt levels, statically assigned (e.g., reset = 0, timer = 1, network = 3, disk = 4, software = 7)
  - When you “disable interrupts” you disable them for your level and higher.
  - When you reenable interrupts, you need to do so at the previous level.

```
unsigned long flags;
local_irq_save( flags );  // Disable & save
CRITICAL SECTION GOES HERE;
local_irq_restore( flags ); // Reenable
```

Question

- What is wrong with this code?

```
unsigned long flags;
local_irq_save( flags );  // Disable & save
  …
  if(somethingBad) {
    return ERROR_BAD_THING;
  }
  …
local_irq_restore( flags ); // Reenable
return 0;
```

Using Interrupt Correctly

- Make sure to re-enable interrupts along every possible execution path.

```
unsigned long flags;
local_irq_save( flags );  // Disable & save
  …
  if(somethingBad) {
     local_irq_restore( flags );
     return ERROR_BAD_THING;
  }
  …
local_irq_restore( flags ); // Reenable
return 0;
```

S , : yM, l k ed / C Hc? C H p yxmedo-

Outline

- Background
- Basic Concepts
- Critical Section
- Approach 1: Disabling Hardware Interrupt
- Approach 2: Software-based Solution
- Approach 3: Higher-level Abstractions

OS

Approach 2: Peterson's Algorithm Initial Attempts to Solve Problem

Only 2 threads, *T*0 and *T*1

General structure of thread *Ti* (other thread *Tj*)

```
do {
    enter section
        critical section
    exit section
        reminder section
} while (1);
```

Threads may share some common variables to synchronize their actions.

Approach 2: Peterson's Algorithm
First Attempt: Algorithm 1

- Shared variables - initialization
  - int turn = 0;
- turn == i // indicates whose turn it is to enter the critical section
- Thread *Ti*

```
do {
    while (turn != i) ;
    critical section
    turn = j;
    reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress some time
  - (Ti do other thing, Tj want to continue to run, but have to wait Ti do critical section)

Approach 2: Peterson's Algorithm
Second Attempt: Algorithm 2

- Shared variables - initialization
  - int flag[2]; flag[0] = flag[1] = 0;
- flag[i] == 1 //indicate if process is ready to enter the critical section
- Thread Ti

```
do {
 while (flag[j] == 1) ;
 flag[i] = 1;
 critical section
 flag[i] = 0;
 remainder section
} while(1);
```

d, p l k yao(xa f e d

Approach 2: Peterson's Algorithm
Third Attempt: Algorithm 3

- Shared variables - initialization
  - int flag[2]; flag[0] = flag[1] = 0;
- flag[i] == 1 CC Ti want to enter its critical section
- Thread Ti

```
do {
        flag[i] = 1;
        while (flag[j] == 1) ;
        critical section
        flag[i] = 0;
        remainder section
} while(1);
```

/ykefFeof p l kl yao(xal f e, duMl kmyf . oy.)a, xwP

Approach 2: Peterson's Algorithm

- Classic software-based solution to achieve mutual exclusion between 2 processes Pi and Pj. (year 1981)
- Use two shared data items

int turn; //indicates whose turn it is to enter the critical section

boolean flag[]; //indicate if process is ready to enter the critical section

- Code for ENTER_CRITICAL_SECTION

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j) ;
```

- Code for EXIT_CRITICAL_SECTION

```
flag[i] = FALSE;
```

Approach 2: Peterson's Algorithm

Algorithm for Process P_i

```
do {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

            CRITICAL SECTION

        flag[i] = FALSE;

            REMAINDER SECTION

   } while (TRUE);
```

Approach 3: Dekkers's Algorithm

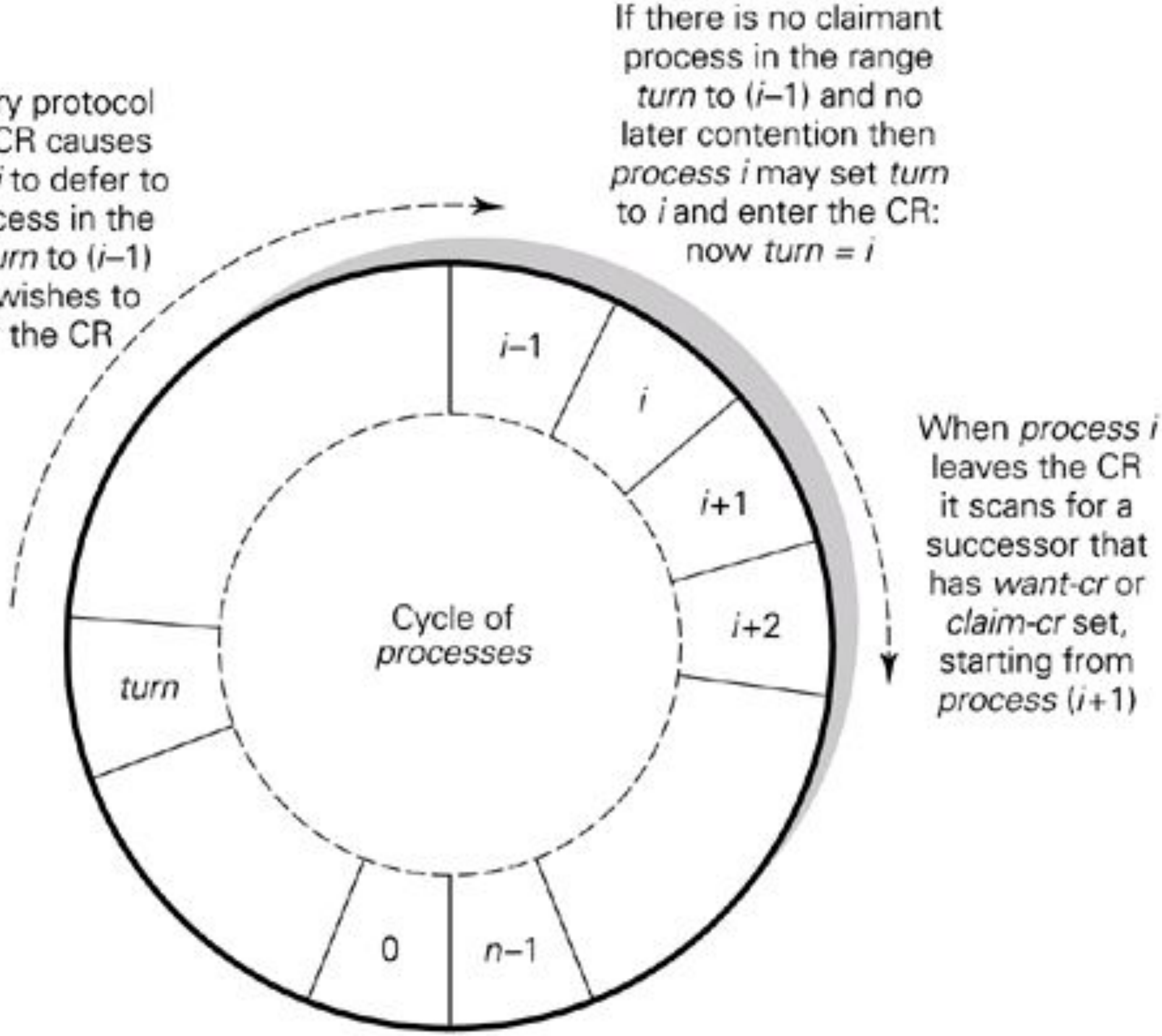
Algorithm for Process P_i

flag[0] := false flag[1] := false turn := 0 // or 1

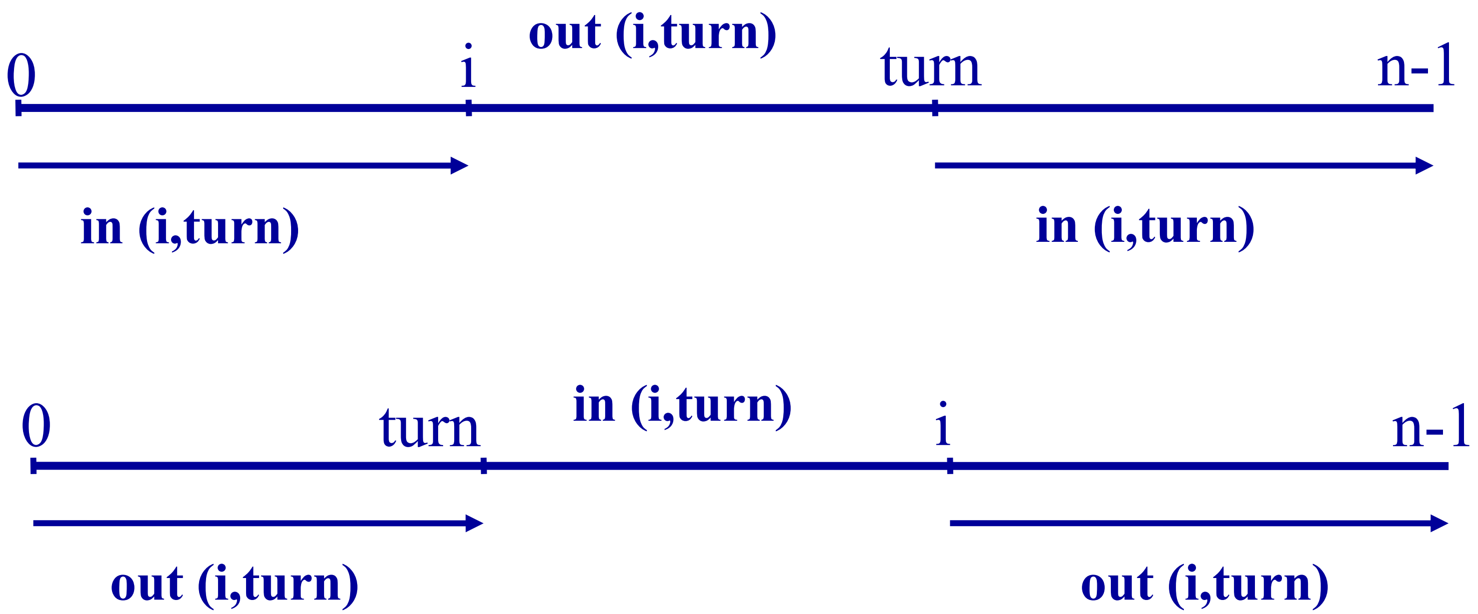
```
do {
        flag[i] = TRUE;
        while flag[j] == true {
            if turn ≠ i {
              flag[i] := false
              while turn ≠ i { }
              flag[i] := TRUE
            }
         }
         CRITICAL SECTION
        turn := j
        flag[i] = FALSE;
         EMAINDER SECTION
    } while (TRUE);
```

OS

An N-Process Solution: Eisenberg and McGuire's Algorithm



An N-Process Solution: Eisenberg and McGuire's Algorithm



ed~ekl vd>s kl vd; d)1， r; e)1 （kl vd<e） +kl vd; e)
1 ~kl vd{ e>
, l k~ekl vd>s e 1ukl vd （kl vd<e） +e 1; d)1，
r; kl vd ~kl vd{ e>

INITIALIZATION

```
f myvo. odl p  fkykof DEWhNuG i EbEV} ui ? bE[ N] FayOf *d )1!+
f myvo. edk kl vd+
edk ed. o( +      c=d, k f myvo. &=c
PPP
kl vd |  r +
PPP
F, v ~ed. o( | r +ed. o( { d+ed. o( l l >D
         FayOf *ed. o( ! |  EWhN+
]
```

ENTRY PROTOCOL (for Process i)

vot oyk D

FayOf *e! | G i EbEV} + c=ydd, l dxo kmyk : o doo. kmo vof , l vxo =c

c=f xyd t v, xof f of Fv, p kmo , do : ekmkmo kl vd l t k, , l vf oahof P=c

c=vot oyk eF doxof f yvB l dkeakmo f xyd Fed. f yaat v, xof f of e ao =c

ed. o(| kl vd+

while ~ed. o(&| e>D cckl vd; d)1， r ; e)1 （kvl d<e） +kl vd; e)1 ~kl vd{ e>等

eF ~FayO*ed. o(! &| EWhN>ed. o(| kl vd+

oaf o ed. o(| ed. o(l 1 p , . d+

] cc除Hkl vd外， H（, l k~eukl vd>） 会被挡住

FayOf *e! | i ? bE[N+c=d, : kodkykehoaB xayep kmo vof , l vxo =c

c=Fed. kmo ***first active process*** Mof e of , l vf oahof ueFydB =c

ed. o(| r +

while ~ ~ed. o({ d>q q ~ed. o(| | e>'' ~FayOf *ed. o(! &| i ? bE[N>>>D

ed. o(| ed. o(l 1+

]

c=eF kmovo : ovo d, , kmovyxkeho t v, xof f of ui VW eF: o myho kmo kl vd

, v oaf o : m, ohov myf ekef e aoukmod t v, xoo. P T kmov: ef ouvot oyk

kmo : m, ao f o..l odxoP=c cc如果Hkl vd不是EWhN, 多个H~ed~eukl vd>>会被挡住

] l dkea~ed. o(<| d>q q ~kl vd | | e>'' ~FayOf *kl vd! | | EWhN>>>+

kl vd | e+ c=xayep kmo kl vd yd. t v, xoo. =c

EXIT PROTOCOL (for Process i)

c=Fed. y t v, xoff : mexmef d, k EWhN =c

c=~eFkmovo yvo d, , kmovfu: o : eaaFed. , l vfoahof >=c

ed. o(| kl vdl 1 p , . d+

: meao ~FayOf *ed. o(! | | EWhN>D

ed. o(| ed. o(l 1 p , . d+

]

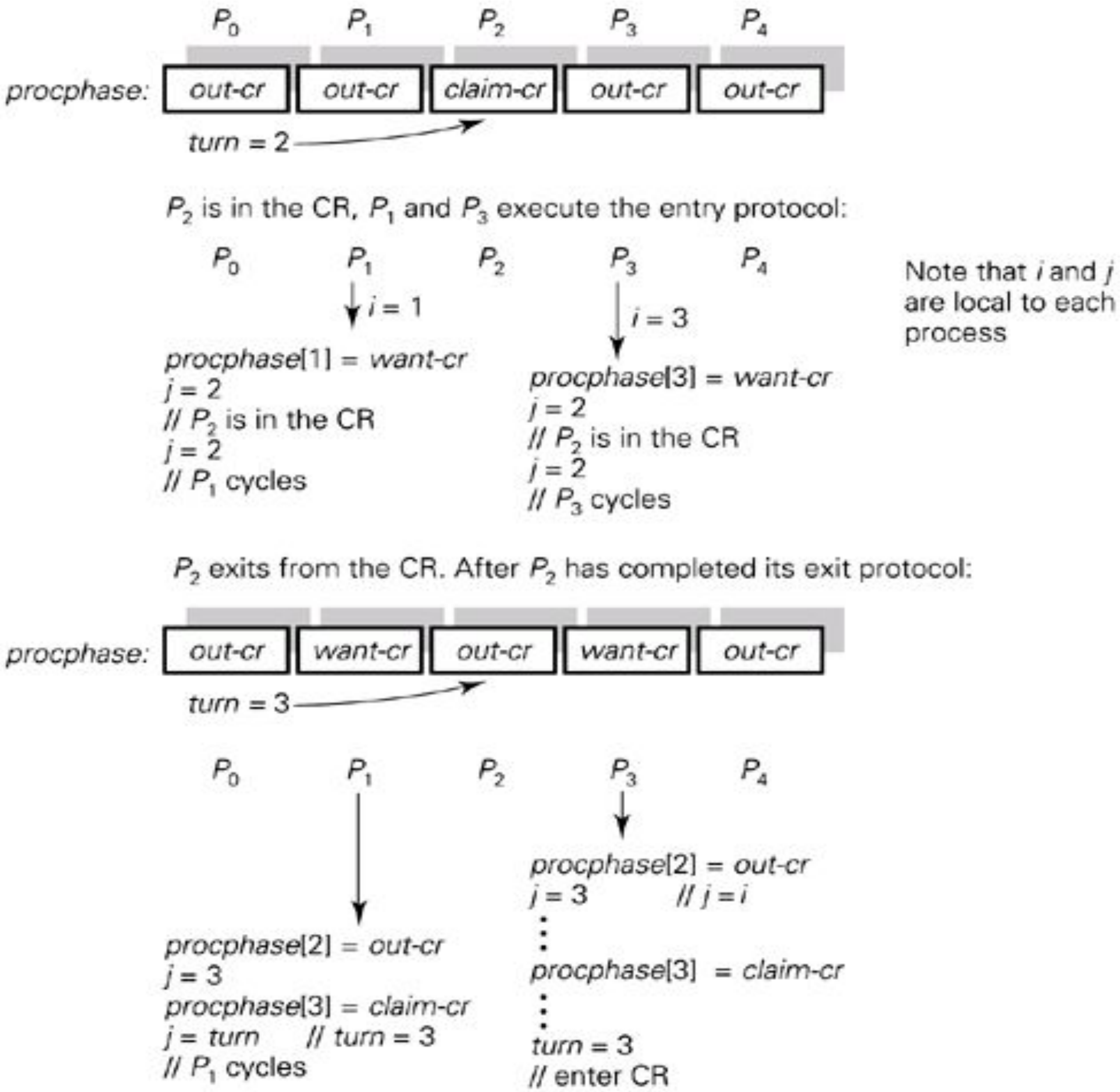
c=Oeho kmo kl vd k, f , p o, do kmyk doo. f eku, v woot ek =c

kl vd | ed. o(+

c=: o vo Fedefmo. d, : =c

FayO*e! | EWhN+

An N-Process Solution: Eisenberg and McGuire's Algorithm



N-Processes: Bakery Algorithm

Critical section for n processes

- Before entering its critical section, a process receives a number.
- The holder with the smallest number will enter the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

N-Processes: Bakery Algorithm

- Notation < lexicographical order

  (ticket #, process id #)

  - $(a,b) < (c,d)$ if $a < c$ or if $a == c$ and $b < d$
  - max $(a_0, \quad , a_{n-1})$ is a number, k, such that $k >= a_i$ for $i = 0, 1, 2, \ldots, n - 1$

- Shared data

  boolean choosing[n];

  int number[n]; //ticket

- Data structures are initialized to **false** and **0** respectively.

N-Processes: Bakery Algorithm

```
do {
    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;

    for (j = 0; j < n; j++) {
      while (choosing[j]) ;
      while ((number[j] != 0) && (  (number[j],j) < (number[i],i)  ) ) ;
    }
        critical section
    number[i] = 0;
        remainder section
} while (1);
```

Why "Choosing" ??

N-Processes: Bakery Algorithm

- If *Pi* is in its critical section and *Pj (j!=i)* has already chosen its *number[j] != 0*, then

  (number[i],i) < (number[j], j)

  or

  (number[i],i) is the smallest of *{(number[0], 0), (number[1], 1), (number[2], 2), … }*

- Mutual exclusion. Only the process with the smallest *(number[i],i)* can enter its critical section.
- Progesss requirement and bounded waiting. The processes enter their critical section on a first-come, first-served business.

Comments on Software-Based Solution

- Dekker's Algorithm (1965): This is the first correct solution proposed for the two-thread (two-process) case.
- Bakery Algorithm (Lamport 1979):A Solution to the Critical Section problem for n threads

- Complicated
  - Need shared data items between any two processes
- Need busy-waiting
  - Waste CPU time
- Really no pure software solution without some hardware guarantee!
  - Peterson's algorithm requires atomic LOAD and STORE instructions

Outline

- Background
- Basic Concepts
- Critical Section
- Approach 1: Disabling Hardware Interrupt
- Approach 2: Software-based Solution
- Approach 3: Higher-level Abstractions

Approach 3: Higher-level Abstractions

- Hardware provides some primitives
  - Like disabling interrupt, atomic instructions, etc.
  - Most modern architecture do
- OS provides higher-level programming abstractions to simplify concurrent programming
  - Examples: Locks, Semaphores
  - Constructed from hardware primitives

High-level Abstraction: Locks

- Lock as an abstract data type
  - One binary state (locked/unlocked), two methods
  - Lock::Acquire() – wait until lock is free, then grab it
  - Lock::Release() – release the lock, waking up a waiter if any
- Programming critical section with locks
  - Previous example becomes easy:

```
lock_next_pid->Acquire();
new_pid = next_pid++ ;
lock_next_pid->Release();
```

Lock Implementation with Disabling Interrupts

- A simple solution:

```
Lock::Acquire() {
  disable interrupts;
}
```

```
Lock::Release() {
  enable interrupts;
}
```

- A better solution:

```
class Lock { int value = FREE; }
```

```
Lock::Acquire() {
  disable interrupts;
  while (value != FREE) {
    enable interrupts;
    disable interrupts;
  }
  value = BUSY;
  enable interrupts;
}
```

```
Lock::Release() {
  disable interrupts;
  value = FREE;
  enable interrupts;
}
```

Hardware Primitives – Atomic Instructions

- Most modern architectures provide special atomic instructions
  - By special memory access circuitry
  - For both uni-processor and multi-processor
- Test-and-Set
  - Read a value from memory
  - Test if the value is 1 (and return true or false)
  - Set the memory value to 1
- Compare-and-Swap(Exchange)
  - Read a value from memory
  - Compare if the value equals a given constant
  - If true, write a given new value to the memory location

OS

```
boolean TestAndSet (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }

void Exchange (boolean *a, boolean *b)
    {
        boolean temp = *a;
        *a = *b;
        *b = temp:
    }
```

Implementing Locks with Test-and-Set

```
class Lock {
    int value = 0;
}
```

```
Lock::Acquire() {
   while (test-and-set(value))
      ; //spin
}
```

- If lock is free, then test-and-set reads 0 and sets value to 1 ➔ lock is set to busy and Acquire completes
- If lock is busy, the test-and-set reads 1 and sets value to 1 ➔ no change in lock's status and Acquire loops (spins)

```
Lock::Release() {
    value = 0;
}
```

Spinlock

```
Lock::Acquire() {
  while (test-and-set(value))
    ; //spin
}
```

- A lock that uses busy-waiting
  - Like the above implemented with test-and-set
  - Threads consume CPU cycles while waiting
- Can you do better? And when?

Implementing Locks without Busy Waiting

G ekmMl f B : yekedO

```
Lock::Acquire() {
   while (test-and-set(value))
      ; // spin
}

Lock::Release() {
    value = 0;
}
```

G ekm, l k Ml f B : yekedO

```
class Lock {
   int value = 0;
   WaitQueue q;
}

Lock::Acquire() {
   while (test-and-set(value)) {
      add this TCB to wait queue q;
      schedule();
   }
}

Lock::Release() {
   value = 0;
   remove one thread t from q;
   wakeup(t);
}
```

S , : k, l f o o(xmydOo k, ep t aop odk n, xw-

Implementing Locks using exchange

- Shared data (initialized to **0**):
  - int lock = 0;

Thread *Ti*

```
    int key;
    do {
            key = 1;
            while (key == 1) exchange(lock,key);
                critical section
            lock = 0;
                remainder section
    }
```

Codes: Spin Lock

bmvoy. 1

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;

spin_lock(&mr_lock);
/* critical section */
spin_unlock(&mr_lock);
```

Mutual Exclusion Machine Instructions

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

Mutual Exclusion Machine Instructions

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Implementing Locks: Summary

- Locks are higher-level programming abstraction
  - Mutual exclusion can be implemented using locks
  - Generally require some level of hardware support
- Two common implementation approaches
  - Disable interrupts (uni-processor only)
  - Atomic instructions (uni- and multi-processor arch.)
- Implementation alternative:
  - Busy-waiting
  - Minimal Busy-waiting
- Is this sufficient?
  - What if you want to synchronize on a condition?

Bounded Buffer Producer-Consumer Problem

- N buffers, one producer adds to the buffer, one consumer subtracts from the buffer
  - Must wait if buffer is empty or full
  - Locking is insufficient

```
class BoundedBuffer {
    …
    Lock lock;
    int count = 0;
}
```

```
BoundedBuffer::Deposit(c) {
    lock->Acquire();
    while (count == n) { … };
    Add c to the buffer;
    count++;
    lock->Release();
}
```

```
BoundedBuffer::Remove(c){
    lock->Acquire();
    while (count == 0) { … };
    Remove c from buffer;
    count--;
    lock->Release();
}
```

Reference

- materials from Dr. Zhang Yong Guang in MSRA
- William Stallings, Operating Systems-Internals and Design Principles(5th Edition), Prentice Hall, 2005
- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating system concepts (7th Edition), John Wiley & Sons, 2004
- An N-Process Solution: Eisenberg and McGuire's Algorithm, http://www.cs.wvu.edu/~jdm/classes/cs550/notes/tech/mutex/Eisenberg.html