Critical Section in a Process: Environment

- *n* processes all competing to use some shared variables (data).
- Each process has multiple code segments, each called a ***critical section***, in which the shared data is accessed/modified.
- Problem – Make sure that the three critical section problem properties are always satisfied.

Program Structure (with a Single Critical Section)
(simplification for the time being)

repeat

$\boxed{\text{entry section}}$

critical section (CS)

$\boxed{\text{exit section}}$

remainder section (RS)

until *False*

General structure of process P_i (*only* in this part of slides)

Two-Process (P_0 and P_1) Synchronization: Algorithm 1

```
int  turn;              /* turn is a shared variable;  initially,  turn = 0
                           turn == i ⇒ P_i can enter its critical section */

P_i : repeat

        while (turn != i) do no-op;        /* turn=0 ⇒ P_0 can enter CS
                                              turn=1 ⇒ P_1 can enter CS */
        critical section

        turn := j;                         /*   j=1 – I,    0 ≤ i, j ≤ 1   */

        remainder section

      until false;
```

Correct? Satisfaction of (1) mutual exclusion?
(2) Progress?
(3) Bounded Waiting?

Two-Process (P_0 and P_1) Synchronization: Algorithm 1

```
int  turn:=0;    /* j=1 – i,  0 ≤ i, j ≤ 1

Pi : repeat
        while (turn != i) do no-op;
        CS
        turn := j;
        RS
     until false;
```

```
int  turn:=0;

Pj : repeat
        while (turn != j) do no-op;
        CS
        turn := i;
        RS
     until false;
```

Two-Process Synchronization: Algorithm 2

boolean flag[2]; /* shared variables; initially flag[0] = flag[1] = false
/* flag[i] == true $\Rightarrow$ P_i can enter its critical section

P_i : repeat

while (flag[j]) **do** *no-op***;**
flag[i] := *True*;

critical section

flag[i] := *False*;

remainder section

until *False;*

Correct? Satisfaction of (1) Mutual Exclusion? (2) Progress? (3) Bounded waiting?

Two-Process Synchronization: Algorithm 2

```
boolean flag[2]:=False;
   /* flag[i] == true ⇒
   /*           P_i can enter its CS.

P_i : repeat
       while ( flag [ j ] ) do no-op;
       flag[i] := True;
       CS
       flag[i] := False;
       RS
     until False;
```

```
boolean flag[2]:=False;
   /* flag[j] == true ⇒
   /*           P_j can enter its CS.

P_j : repeat
       while ( flag [ i ] ) do no-op;
       flag[j] := True;
       CS
       flag[j] := False;
       RS
     until False;
```

Two-Process Synchronization: Algorithm 3

boolean flag[2]; /* shared variables, initially flag[0] = flag[1] = false
/* flag[i] == true $\Rightarrow$ P_i ready to enter its critical section.

P_i : repeat

flag[i] := *True*;
while (flag[j]) **do** *no-op***;**

critical section

flag[i] := *False*;

remainder section

until *False;*

Correct? Satisfaction of (1) Mutual Exclusion? (2) Progress? (3) Bounded waiting?

Two-Process Synchronization: Algorithm 3

boolean flag[2]*:=False;*
/* flag[i] == true $\Rightarrow$
/* P_i ready to enter its CS.

P_i : repeat
flag[i] := *True*;
while (flag[j]) **do** *no-op***;**
CS
flag[i] := *False*;
RS
until *False;*

boolean flag[2]*:=False;*
/* flag[j] == true $\Rightarrow$
/* P_j ready to enter its CS.

P_j : repeat
flag[j] := *True*;
while (flag[i]) **do** *no-op***;**
CS
flag[j] := *False*;
RS
until *False;*

Two-Process Synchronization: Algorithm 4

(Peterson's Algorithm)

boolean flag[2]; /* shared variables, initially flag[0] = flag[1] = false
/* flag[i] == true $\Rightarrow$ P_i ready to enter its critical section.

int turn;

P_i: repeat

flag[i] := *True*;
turn := j;
while (flag[j] && turn == j) **do** *no-op***;**

critical section

flag[i] := *False*;

reminder section

until *False;*

Informal Proofs:
Mutual exclusion
Progress
Bounded waiting

Very fragile solution!

Meets all three requirements; solves the critical-section problem.

Two-Process Synchronization: Algorithm 4

```
boolean flag[2]:=False;
/* flag[i] == true ⇒
   Pi ready to enter its CS.
int  turn;


Pi: repeat
      flag[i] := True;
      turn := j;
      while (flag[j] && turn == j)
              do no-op;
      CS
      flag[i] := False;
      RS
   until False;
```

```
boolean flag[2]:=False;
/* flag[j] == true ⇒
   Pj ready to enter its CS.
int  turn;


Pj: repeat
      flag[j] := True;
      turn := i;
      while (flag[i] && turn == i)
              do no-op;
      CS
      flag[j] := False;
      RS
   until False;
```

Peterson’s algorithm

Idea: Three-step solution:

1. Declare intention to enter into the CS.
2. Let others enter into their CSs.
3. Enter the CS.

This is an “example reasoning” for many critical section algorithms.

Where is the fragility of algorithm 4?

Another Algorithm:
Dekker's (much more complicated)

```
Shared variables:     int turn;            // 0 or 1
                      boolean flag[2];     // initially  False
Pi:   repeat
        flag[i] = True;                              // I would like to enter.
        while (flag[j]) {                            // Does she?
                if (turn == j) {                     // yes, and it is her turn.
                        flag[i] = False;             // So I will yield
                        while (turn == j);           // waiting for her to leave.
                        flag[i] = True;              // Finally! She is done.
                }
        }

        critical section

        turn = j;                                    // I had my turn.
        flag[i] = False                              // Done

        remainder section

      until  False;
```

Proof: Mutual exclusion? Progress? Bounded waiting?

Review: So Far, Brute-Force Process Synchronization Algorithms--Review (no synchronization constructs)

- So far we have seen:
  Very simple concurrent program structure, and two-process synchronization algorithms.
  - Algorithms 1 to 4. Correct one: fourth algorithm.
- Next: the same simple concurrent program structure, and n-process synchronization algorithm.

N-process Synchronization: Bakery Algorithm (L. Lamport; 1974)

Program structure: As in the two process structure, except that there are now n processes, P_i, $0 \le i \le (n-1)$.

Solution Logic: Implement a bakery-ticket mechanism (sort of).

Before entering its critical section, a process gets a (ticket) number. But, multiple processes *can* have the same ticket number!

Holder of the smallest number enters the critical section.

If two processes P_i and P_j receive the same number and $i < j$, then P_i enters into its CS first; else P_j enters its CS first.

N-process Synchronization: Bakery Algorithm (L. Lamport; 1974)

```
Pi:  repeat

        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 ]) do no-op;
              while (number [ j ] != 0 && (number [ j ], j) < (number [ i ], i))
                                    do no-op ;
        }

        critical section

        number [ i ] := 0;

        remainder section

     until False;
```

N-process Synchronization: Bakery Algorithm

Shared data:

boolean choosing[n];

int number[n];

Initially, all choosing values are *False;*

All number values are 0.

Notation: lexicographical order (ticket #, process id #)

(a, b) < (c, d) **if** a < c or (*a* = *c* **and** b < d *)*

Bakery Algorithm: Proof of Mutual Exclusion

- **Observation**: If
  (a) P_i is already in critical session, and
  (b) any P_k, k!=i, has chosen its number (i.e., number[k]!=0),
  then (number[i], i) < (number[k], k)

  Proof:

  - Case 1: P_k *hasn't* started to choose number[k] when P_i executes while(choosing[k]). Obviously, number[i] < number[k].
  - Case 2: P_k *has* chosen number[k]!=0 before P_i executes "while(choosing[k])…".
    P_i wins, implying (number[i], i) < (number[k], k)
  - Case 3: P_k *is* choosing number[k] when P_i executes while(choosing[k]).
    P_i will wait. We will reduce this to Case 2.
- Because of this observation, if P_i is already in its CS, no other process can enter into its CS.

Why Do We Need choosing[ ]?

P_i: repeat

~~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]) **do** no-op;~~
    while (number[j] != 0 **&&** (number[j], j) < (number[i], i))
        do no-op ;
}

critical section

number[i] := 0;

remainder section

until *False;*

Why Do We Need choosing[ ]?

- Two processes, initially number[0]=number[1]=0.

P_0:	**P_1:**
evaluate max () function;	
… leave CPU…	
	evaluate max () function;
	number[1]:=1;
	enter critical section
	…leave CPU **while in CS**…
..start executing…	
number[0] := 1;	
while condition is false.	
enter critical section → **violation of ME.**	

Review: Brute-Force Process Synchronization Algorithms--Review (no synchronization constructs)

- So far, we have seen
  - Very simple program structure and two-process synchronization
    - Algorithms 1 to 4. Correct one: fourth algorithm.
  - Same simple program structure and n-process synchronization
    - Bakery Algorithm

Hardware Solutions to Synchronization

- Up until now, we have seen brute-force ways of providing a solution to the critical section (CS) problem—only when the program structure is very limited.
- Next, we will see hardware-based solutions to the CS problem.

  “Single CPU Cycle” (atomic) function as part of the hardware instruction set:

> **function** *Test-and-Set* (**var** target: bool): bool;
>   **begin**
>   *Test-and-Set* := *target*;
>   *target* := *True*;
>   **end;**

Mutual Exclusion with Test-and-Set

```
Shared data:
        bool lock := False;

Pi:  repeat
        begin
           while (Test-and-Set (lock)) do no-op;

              critical section

           lock := False;

              remainder section
        end
     until False;
```

Satisfaction of: Mutual exclusion? Bounded waiting? Progress?

Another Hardware Sol'n: Swap

Also a "single CPU cycle" (atomic) procedure as part of the hardware instruction set:

```
Procedure Swap (var a, b: bool)
    var temp: bool;
    begin
        temp := a;
        a := b;
        b := temp;
    end;
```

Mutual Exclusion with Swap

```
Shared data:
        bool lock:= False;

Pi : repeat
        key := True;                    /* Pj has its own key variable.
        repeat
          Swap(lock, key);
        until key = False;

            critical section

        lock := False;

            remainder section

    until False;
```

Satisfaction of: mutual exclusion? Bounded waiting? Progress?

Next: Three separate process synchronization constructs

- **Semaphores**
- **Critical region or conditional critical region statements—CCRs**
  --Only briefly. No CCR-related exam/assignment questions.
- **Monitors**
  - Btw, every Java object is/has a monitor: more on this later.

- **NOTE: NO MIXING OF THESE CONSTRUCTS in your algorithms!**
  - **Your concurrent process algorithm can only use ONE of the three constructs.**
    - **If your algorithm uses semaphores, it cannot use CCRs or monitors.**
    - **If your algorithm uses CCRs, it cannot use semaphores or monitors.**

- **NOTE: ALWAYS USE THE SYNTAX SPECIFIED IN SLIDES!!**

Binary Semaphores

- **Two *atomic operations* on a variable (semaphore) S:**
- **S below is a binary semaphore** (takes on 0 or 1 as value):

wait (*S*): **while** *S* $\leq$ 0 **do** *no-op*; Dijkstra: P(S)
S:=S – 1;

signal (*S*): *S:=S + 1;* Dijkstra: V(S)

- Semaphore S can **only** be **accessed via wait() and signal()**
  --two indivisible (i.e., atomic) operations.

- Initialization of S: either 0 or 1. Normally, 1.

- For a nice intro to semaphores, see the semaphores chapter in a online book.

Critical Section of *n* Processes

Shared data:

binary semaphore mutex; /* *mutex* is set to 1 initially.

P_i: repeat

$\boxed{\textit{wait (mutex);}}$

critical section

$\boxed{\textit{signal (mutex);}}$

remainder section

Until *False;*

Mutual exclusion? Progress? Bounded waiting?

Problems with Binary Semaphores

Binary semaphore S:

wait (S): **while** $S \leq 0$ **do** *no-op*;
$S:=S-1;$

signal (S): $S:=S+1;$

1. Busy waiting (i.e., no-op) above (in *wait(S)*) is wasteful.
2. Semaphore *S* above is binary.

   Nonbinary (counting) semaphore operations form a **higher-level synchronization mechanism** that does not require busy waiting.

Eliminating Busy Waiting and Bounded Waiting

```
type nonbinary semaphore = record
        value: integer;          /* value is usually
        L: list of process;      /* set to 1 initially
     end;
```

```
wait(S):
    S.value := S.value – 1;
    if (S.value < 0)
    then  begin
            add this process to S.L;
            block;
          end;
```

```
signal(S):
S.value := S.value + 1;
if (S.value ≤ 0)
then begin
        remove a process P from S.L;
        wakeup(P);
        end;
```

Two simple operations:

- *block* suspends the process that invokes it.
- *wakeup(P)* resumes the execution of a blocked process *P.*

Producer-Consumer (Bounded Buffer) Problem

Initialization:

binary semaphore *mutex:=1;*

nonbinary semaphore *full:=0; empty:=n;*

Producer:
repeat
...
Produce an item in nextp;
wait (empty); /* When empty ≤ 0,
wait (mutex); /* block producer.
add nextp to buffer;
signal (mutex);
signal (full);
until *False;*

Consumer:
repeat
wait(full); /*When full ≤ 0; block consumer.
wait (mutex);
remove an item from buffer to nextc;
signal(mutex);
signal(empty);
...
consume the item in nextc;
until *False;*

Proving Correctness:

Invariant I (single-producer and single-consumer):

$(n - 2) \leq empty + full \leq n$ and $0 \leq mutex \leq 1$

Safety: Prove that $I \rightarrow \neg P_{bad}$

Producer-Consumer (Bounded Buffer) Problem

Initialization:

binary semaphore *mutex:=1;*

nonbinary semaphore *full:=0;* *empty:=n;*

Producer:

repeat

...

Produce an item in nextp;

wait (empty); /* When empty ≤ 0,

wait (mutex); /* block producer.

add nextp to buffer;

signal (mutex);

signal (full);

until *False;*

Consumer:

repeat

wait(full); /*When full ≤ 0; block consumer.

wait (mutex);

remove an item from buffer to nextc;

signal(mutex);

signal(empty);

....

consume the item in nextc;

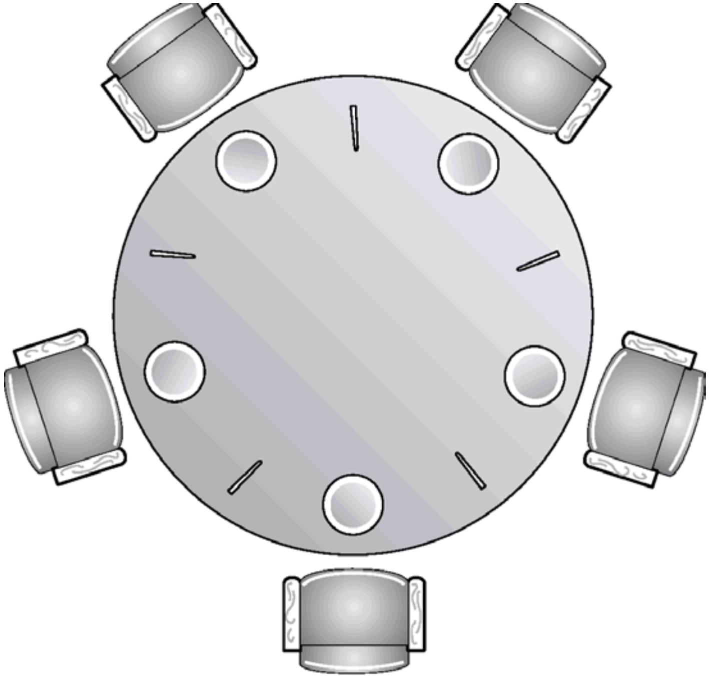
until *False;*

Note the symmetrical use of empty and full semaphores in entry and exit sections of producer and consumer.

Dining Philosophers Problem (1965)

- 5 philosophers; 5 chopsticks; one bowl of rice.

  nonbinary semaphore *chopstick [0..4]:=1;*
  /* All chopsticks are initialized to 1.



Dining Philosophers Problem

- A philosopher eats or thinks!
  - Each philosopher needs two chopsticks to eat, and chopsticks are picked up one at a time.
  - After successfully picking up two chopsticks, a philosopher eats for a while, and, then, puts down the chopsticks, and thinks.
- Problem: Decide how philosophers pick the chopsticks so that
  - Philosophers (processes) do not starve!
  - There is no deadlock.

Dining Philosophers Problem

Nonbinary semaphore *chopstick[0..4]:=1;*

Philosopher ***i*****:**
repeat

```
    wait (chopstick[i]);
    wait (chopstick[(i+1) mod 5]);
        …
        EAT
        …
    signal (chopstick[i]);
    signal (chopstick[(i+1) mod 5]);
        …
        THINK
        …
until False;
```

Problem: Deadlock

Dining Philosophers Problem

Invariant property characterization:

up[i] = count of the times chopstick i has been picked up.

down [i] = count of the times chopstick i has been put down.

A chopstick cannot be put down more times than it has been picked up.

Invariant property:

For all i: $[\ 1 \leq i \leq 5 \ \rightarrow\ (down[i] \leq up[i] \leq (down[i] + 1)\ )\ ]$

Note that a chopstick can be picked up by at most one philosopher at a time.

Dining Philosophers Problem—No Deadlocks

nonbinary semaphore *chopstick[0:4]:=1;*

Philosopher ***i*****:**

repeat

wait (*chopstick*[(i+1) **mod** 5]);
wait (*chopstick*[i]);

…

eat

…

signal (*chopstick*[i]);
signal (*chopstick*[(i+1) **mod** 5]);

…

think

…

until *False;*

Switched these two lines for philosopher 5, and NO deadlocks!

To Avoid Deadlocks-One Approach

- A number of processes (P_i) want to access an arbitrary number of resources (R_i), needing exclusive access. Processes must abide by the following rules:
  - Circular wait elimination via ordered resource requests: Processes request resources in increasing order of resource ids. A philosopher obtains a low-numbered fork first.
  - Eliminating no preemption: If a process has obtained a resource R_i and needs resource R_j, it must release R_i before making the request if $i > j$.

Readers-Writers Problem (1971)

Assumption: No reader waits unless a writer is writing! **Writers may starve!**

Initialization:

nonbinary semaphore *mutex:=1; wrt:=1;*

int *readcount:=0;*

Reader:

```
wait (mutex);
        readcount := readcount + 1;
        if readcount = 1 then wait (wrt);
signal (mutex);
```

In CS: Reading!

```
wait (mutex);
    readcount := readcount - 1;
    if readcount = 0 then signal (wrt);
signal (mutex);
```

Writer:

```
wait (wrt);
```

In CS: Writing!

```
signal (wrt);
```

Soln is NOT symmetric for readers and writers.
Not good.

Readers-Writers Problem

Invariant: Use two auxiliary variables that characterize the state of concurrent execution.

writing [j] = 1 if writer j is in CS, 0 otherwise.

reading = 1 if ***any*** reader is in its CS, 0 otherwise.

Initially, reading=writing[j]=0 for all j, $1 \le j \le n$

Invariant I: reading + writing [1] + … + writing [n] ≤ 1

Safety: Prove that $I \rightarrow \neg P_{bad}$

Readers-Writers Problem-Nobody Starves.

Reading/Writing Enforcement: FCFS

Initialization:

nonbinary semaphore *mutex:=1; wrt:=1;* ***rw:=1;***

int *readcount*: *=0;*

Reader:

wait(rw);
wait(mutex);
readcount := readcount + 1;
if *readcount* = 1 **then** *wait (wrt);*
signal (mutex);
signal(rw);

…

In CS: Reading!

…
wait (mutex);
readcount := readcount - 1;
if *readcount* = 0 **then** *signal (wrt);*
signal (mutex);

Writer:

wait(rw);
wait (wrt);

In CS: Writing!

signal (wrt);
signal(rw);

Yet Another Readers-Writers Problem: Readers Starve!

Specifications:

- If a reader finds that there are writers waiting, the reader must wait until all waiting writers are finished writing.
- Even if writers arrive after the reader does, writers write before the reader reads, i.e., the reader can read only if the writers queue is empty.
- Readers can starve.

Give a semaphore-based solution to the Readers-Writers problem where "readers starve" in the sense that a stream of writers can arrive and "write" one after another even when a reader has been waiting to read.

Global variables:

semaphore mutex: initially set to 1 rmutex, wtr, rdr : all initially set to 0
int nreaders = 0, nwriters = 0; boolean Busy = *False*; RBlocked = *False*;

Reader:

```
while True
 { wait (r-mutex);
   wait (mutex);
   if (nwriters > 0)
    then {RBlocked=True;
             signal(mutex);wait(rdr)}
    else {nreaders++;signal(mutex)
    };
   signal(r-mutex);
   READ;
   wait(mutex);
    nreaders--;
    if (nreaders = 0 and
       nwriters > 0)
    then {Busy = True; signal(wrt)};
    signal(mutex);
    DO-SOMETHING;
  }
```

Writer:

```
while True
 { wait(mutex);
   nwriters++;
   if ( Busy or nreaders > 0)
   then {signal(mutex); wait(wrt)}
     else { Busy = True;
              signal(mutex)};
   WRITE;
   wait(mutex);
   nwriters--;
   Busy = False;
   if (nwriters > 0)
   then {Busy = True; signal(wrt)}
   else if (RBlocked) then
       {RBlocked = False;
         nreaders++; signal(rdr)};
   signal(mutex);
   DO-SOMETHING;
 }
```

Other Problems: Sleeping Barber Problem

- A barbershop consists of a waiting room with *N* chairs. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

- Objectives
  - No starvation: a customer will have his hair cut once he sits down.
  - No deadlocks: both barber and customers could be waiting.

Cigarette-Smokers Problem

Consider a system with three smoker processes and one agent process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent on completion. The agent then puts out another two of the three ingredients, and the cycle repeats. Write a program to synchronize the agent and the smokers.

Exercises

- Three processes, P_0, P_1, and P_2. Write code with semaphores to synchronize them such that P_2 executes only after both P_0 and P_1 execute.

- Two processes, P_0 and P_1. Write code with semaphores to synchronize them such that P_0 executes at least twice as often as P_1 executes.