OPERATING SYSTEMS

Cooperating Processes

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

中山大學 SUN YAT-SEN UNIVERSITY

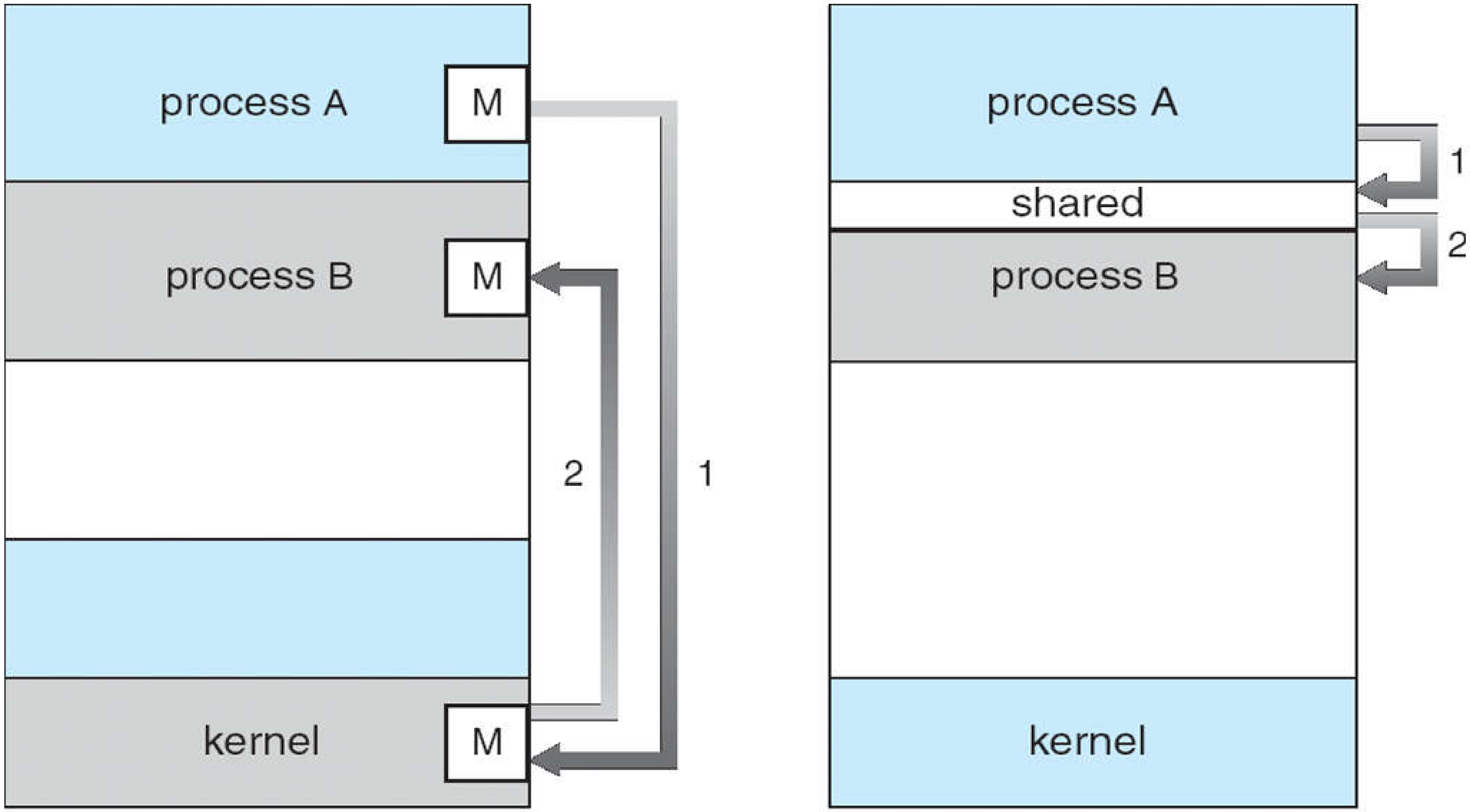
Contents

Cooperating Processes

- Processes within a system may be independent or cooperating.
  - Independent process cannot affect or be affected by the execution of another process.
  - Cooperating process can affect or be affected by other processes, including sharing data.
- Reasons for cooperating processes:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Cooperating Processes

- Cooperation Models



(a) (b)

Data Consistency with Concurrent Execution

- Concurrent processes or threads often need to share data (maintained either in shared memory or files) and other resources.
- If there is no controlled access strategy, concurrent access to shared data may result in data inconsistency.
- The action performed by concurrent processes will then depend on the *order* in which their execution is interleaved.
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Data Consistency with Concurrent Execution

- Example 1.
  - Shared data

```
int a = 1, b = 2, c = 3;
```

  - Thread T_1: / Thread T_2:

```
Thread T1:              Thread T2:
                        int d = 4;
                        a = a – d
c = a + b;
                        b = b + d;
                        c = a + b;
```

  - It seemed that T_1 and T_2 should have the same results c = 3.
  - But if T_1 and T_2 are concurrently executing:
    - Suppose T_1 computes a + b after T_2 has done a = a - d, but before T_2 does b = b + d.
    - At this point, a = -3 and T_1 will not obtain the correct result for c = 3, but c = -1.

Data Consistency with Concurrent Execution

- Example 2.
  - Process P_1 and P_2 are running this same procedure and have access to the same variable inchar. Processes can be interrupted anywhere.
  - Shared data

```
Static char inchar;
```

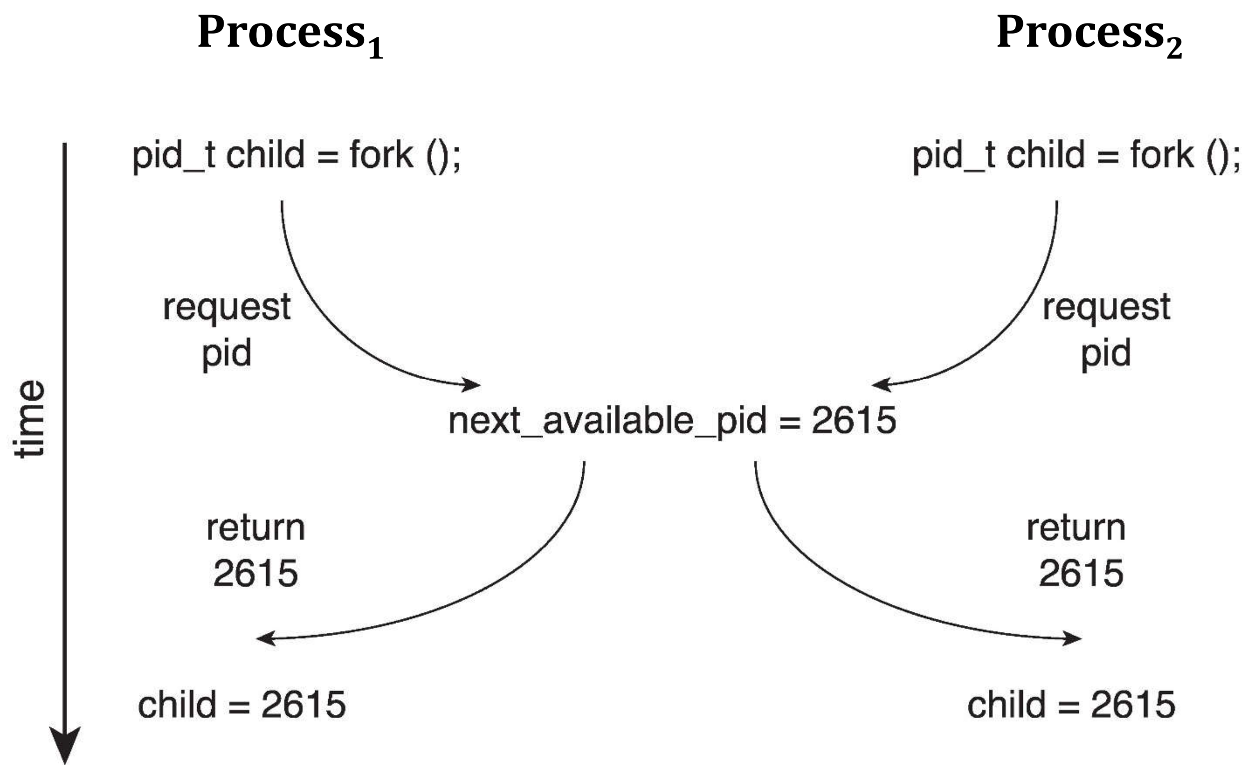
  - Process P_1, P_2:

```
void echo() {
    cin >> inchar;
    cout << inchar;
}
```

  - If P_1 is first interrupted after user input and P_2 executes entirely. Then the character echoed by P_1 will be the one read by P_2. We lost the data consistency.

Race Condition

- Example: Race condition when assigning a pid.



Race Condition

- *Race Condition* is the situation where several processes access and manipulate shared data concurrently. The value of the shared data depends upon the sequence of processes applying to the data.
  - To prevent race conditions, concurrent processes must coordinate or, in other words, be *synchronized*.
- Examples: Race condition when updating a variable.
  - Shared data:

```
double balance;
```

Process$_1$:

```
… …
balance += amount;
… …
```

Code for Process$_1$:

```
… …
Load  R1, balance
Load  R2, amount
Add   R1, R2
Store R1, balance
… …
```

Process$_2$:

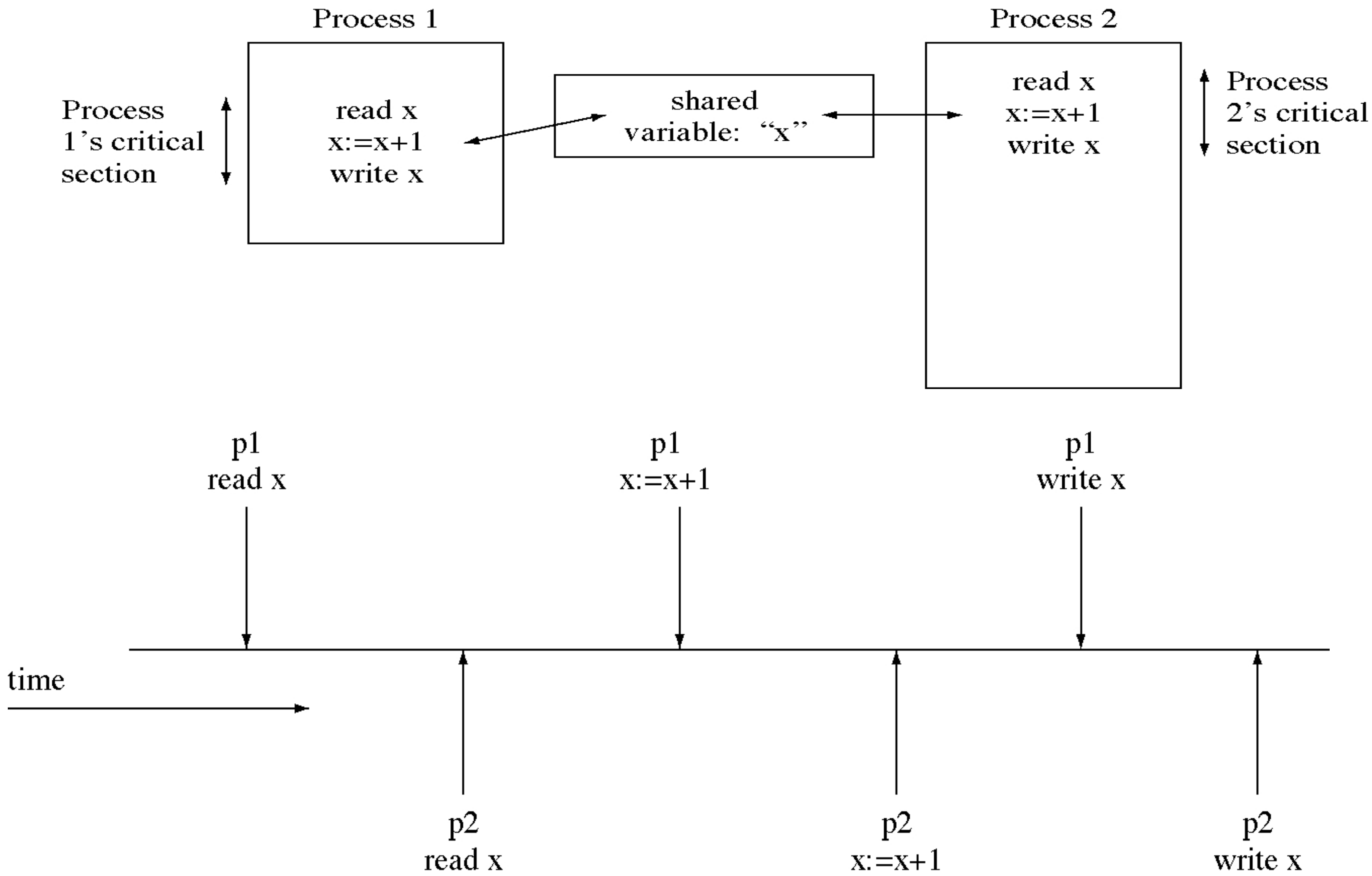
```
… …
balance += amount;
… …
```

Code for Process$_2$:

```
… …
Load  R1, balance
Load  R2, amount
Add   R1, R2
Store R1, balance
… …
```

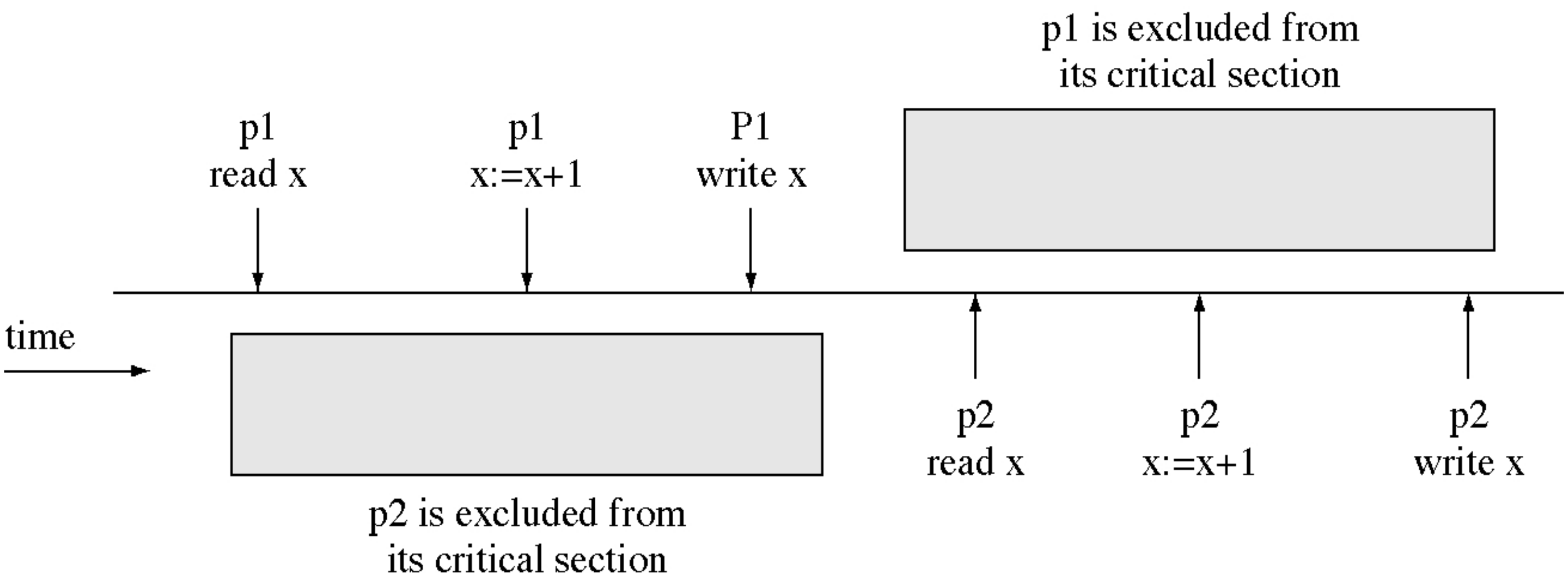
Race Condition

- Examples: Race condition when updating a variable. (cont.)



Race Condition

- Examples: Race condition when updating a variable. (cont.)
  - Set *Critical Sections* (临界区) to prevent a race condition.



  - Multiprogramming allows logical parallelism, uses devices efficiently, but we lose correctness when there is a race condition.
  - So we forbid logical parallelism inside *critical section*, losing some parallelism but regaining correctness.

Race Condition

- Review: Producer-Consumer Problem with Shared-memory (Lecture08)
  - Shared data

```
#define BUFFER_SIZE 10

typedef struct {
    … …     /* item structure */
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

  - The shared buffer is implemented as a *circular array* with two logical pointers: in and out.
    - The variable in points to the next free position in the buffer; out points to the first full position in the buffer.
    - The buffer is empty when in == out;
    - The buffer is full when ((in + 1) % BUFFER_SIZE) == out.
    - This scheme allows at most BUFFER SIZE - 1 items in the buffer at the same time.

Race Condition

- Review: Producer-Consumer Problem with Shared-memory
  - Producer:

```
item next_produced;
while (true) {
        … …   /* produce an item saved in next_produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ;     /* buffer full, do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

  - Consumer:

```
item next_consumed;
while (true) {
    while (in == out)
        ;     /* buffer empty, do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
              /* consume the item in next_consumed */
}
```

Race Condition

- A Shared Counter Solution to P/C Problem
  - Suppose that we wanted to provide a solution to the producer – consumer problem that fills all the buffer (not only BUFFER_SIZE-1 items available). We can do so by having an integer *count* that keeps track of the number of items in the buffer.
  - Initially, the *count* is set to 0. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item.
  - Shared data

```
#define BUFFER_SIZE 10

typedef struct {
    … …    /* item structure */
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int count = 0;
```

Race Condition

- A Shared Counter Solution to P/C Problem
  - Producer Process

```
item nextProduced;
while (TRUE) {
        … …     /* produce an item and put in nextProduced */
    while (count == BUFFER_SIZE)
        ;       /* do nothing – no free slots */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

  - Consumer Process

```
item nextConsumed;
while (TRUE) {
    while (count == 0)
        ;      /* do nothing – nothing to consume */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
           /* consume the item in nextConsumed */
}
```

Race Condition

- Atomic Operations
  - An *atomic/Indivisible operation* (原语) means an operation that completes in its entirety without interruption.
  - In the Producer Process and Consumer Process, the statements

    ```
    count++;
    ```

    and

    ```
    count--;
    ```

    must be performed *atomically*.
  - The statement "count++" could be implemented in machine language as:

    ```
    register1 = count
    register1 = register1 + 1
    count = register1
    ```

  - The statement "count--" could be implemented as:

    ```
    register2 = count
    register2 = register2 – 1
    count = register2
    ```

Race Condition

- Atomic Operations
  - If both the Producer and Consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
    - The interleaving depends upon how the Producer and Consumer processes are scheduled.
  - Consider this execution interleaving with "count = 5" initially:

    Producer: `register1 = count` (*register1 = 5*)
    Producer: `register1 = register1 + 1` (*register1 = 6*)
    Consumer: `register2 = count` (*register2 = 5*)
    Consumer: `register2 = register2 – 1` (*register2 = 4*)
    Producer: `count = register1` (*count = 6*)
    Consumer: `count = register2` (*count = 4*)
  - The value of count may be either 4 or 6, whereas the correct result should be 5.
  - This is again the *Race Condition.*

Race Condition

- Cooperation by Sharing
  - Cooperating processes use and update shared data such as shared variables, memory, files, and databases.
  - Writing must be mutually exclusive to prevent a race condition leading to inconsistent data views.
    - *Critical Sections* are used to provide this data integrity.
    - A process requiring the critical section must not be delayed indefinitely; no deadlock or starvation.
- Cooperation by Message-passing
  - Communication by messages provides a way to synchronize, or coordinate, the various activities.
    - Possible to have *Deadlock*
      - each process waiting for a message from the other process
    - Possible to have *Starvation*
      - two processes sending a message to each other while another process waits for a message

The Critical-Section Problem

- Suppose that
  - n processes are competing to use some shared data.
  - No assumptions may be made about speeds or the number of CPUs.
  - Each process has a code segment, called *critical section* (CS), in which the shared data is accessed.
- When a process executes code that manipulates shared data or resource, we say that the process is *in it's critical section* for that shared data or resource.
- The Critical-Section Problem
  - *The critical section problem* is to design a protocol that the competing processes can use to synchronize their activity so as to cooperatively share data.
  - The execution of critical sections must be *Mutually Exclusive* (互斥).
    - To ensure that when one process is executing in its critical section, no other processes are allowed to execute in their critical sections (even with multiple processors).
      - That is, no two processes are executing in their critical sections at the same time.

The Critical-Section Problem

- The Critical-Section Problem Dynamics
  - Entry Section
    - Each process must first request permission to enter its critical section. The section of code implementing this request is called the *entry section* (ES).
  - Leave/Exit Section
    - The critical section might be followed by a *leave/exit section* (LS).
  - Remainder Section
    - The remaining code is the *remainder section* (RS).
  - General structure of process P_i :

```
while (TRUE) {
    entry section
    critical section
    leave section
    remainder section
};
```

The Critical-Section Problem

- There are three essential criteria that must stand for a correct solution to the critical-section problem:
  - Mutual Exclusion 互斥
  - Progress 推进
  - Bounded Waiting 受限等待
- Mutual Exclusion
  - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
  - Implications:
    - critical sections better be focused and short
    - better not get into an infinite loop in critical sections
    - If a process somehow halts/waits in its critical section, it must not interfere with other processes.

The Critical-Section Problem

- Progress
  - If *no process* is executing in its critical section and there exist some processes that wish to enter their critical sections, then the selection of the process that will enter the critical section next cannot be postponed indefinitely:
    - If only one process wants to enter, it should be able to.
    - If two or more want to enter, one of them should succeed.
- 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 (i.e., during the waiting period of the requesting process).
    - assume that each process executes at a nonzero speed
    - no assumption concerning relative speed of all the processes
  - 在某个进程等待进入其临界区期间，其他并发进程进入临界区的次数必须受到限制。
  - 注意到这并不意味着该进程是有限等待的，比如在死锁发生的情况下。

The Critical-Section Problem

- Preemptive and Non-preemptive Kernels
  - Many kernel-mode processes may be concurrently running in the operating system. Kernel code implementing an operating system is subject to several possible race conditions.
    - Consider the kernel data structures that are prone to possible race conditions include structures for maintaining open file lists, for maintaining memory allocation, for maintaining process lists, and for interrupt handling.
  - *Non-preemptive Kernels* and *Preemptive Kernels* are two general approaches used to handle critical sections in operating systems.
  - Non-preemptive Kernels
    - A non-preemptive kernel (非抢占式内核) does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.
    - Non-preemptive kernels are essentially free from race conditions *on kernel data structures*, as only one process is active at a time.

The Critical-Section Problem

- Preemptive and Non-preemptive Kernels
  - Preemptive Kernels (抢占式内核)
    - A preemptive kernel allows a process to be preempted while it is running in kernel mode.
    - Preemptive kernels must ensure that shared kernel data are free from race conditions. It is especially difficult for *SMP* architectures where two kernel-mode processes may run simultaneously on different processors.
    - Preemptive kernels are more responsive. There is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing (放弃) the processor to waiting processes.
    - Preemptive kernels are more suitable for real-time programming. It will allow a real-time process to preempt a process currently running in the kernel.

The Critical-Section Problem

- Types of Solutions to Critical-Section Problem
  - Software-based solutions
    - algorithms who's correctness does not rely on any other assumptions
  - Hardware-based solutions
    - synchronization hardware
      - rely on some special machine instructions
  - Operating System solutions
    - functions and data structures to the programmer through system/library calls
  - Programming Language solutions
    - linguistic constructs provided as part of a language

Software Solutions to Critical-Section Problem

- We consider first the case of only two processes.
  - Algorithm 1, 2 and 3 can not satisfy the three essential criteria.
  - Algorithm 4 is correct.
    - It is *Peterson*'s algorithm (*G. L. Peterson*, 1981)
- Then we generalize to N processes.
  - *Peterson*'s algorithm for N processes
  - *Lamport*'s Bakery algorithm (*Leslie Lamport*, 1974)
  - *Eisenberg-McGuire*'s algorithm (*Murray A. Eisenberg* & *Michael R. McGuire*, 1972)
- Initial notation in the case of only two processes
  - They are numbered as P_0 and P_1.
  - When presenting process P_i`(Larry,I,i)`, use P_j`(Jim,J,j)` to denote the other process.
    - In the case of only two processes, `j = i - 1`.

Software Solutions to Critical-Section Problem

- Initial attempts
  - General structure of process P_i (The other is P_j)

```
do {
    entry section
    critical section
    leave section
    remainder section
} while (TRUE);
```

  - Processes may share some common data to synchronize their actions.
    - These shared data are initialized and can not be accessed from any remainder sections.

Software Solutions to Critical-Section Problem

- Algorithm.1 – Larry/Jim version.
  - Shared variable:

```
string turn = “Larry”;
  /* initially turn=“Larry” or “Jim” (no matter) */
```

非并发式

Process Larry:	Process Jim:

`do {`
`    while (turn != “Larry”)`
`        sleep(1); /* busy waiting */`
`    Larry’s critical section`
`    turn = “Jim”;`
`      /* Jim can enter its CS */`
`    Larry’s remainder section`
`} while (TRUE);`
	`do {`
	`    while (turn != “Jim”)`
	`        sleep(1); /* busy waiting */`
	`    Jim’s critical section`
	`    turn = “Larry”;`
	`      /* Larry can enter its CS */`
	`    Jim’s remainder section`
	`} while (TRUE);`

  - Mutual exclusion: *Larry* can enter his CS only if turn is “Larry”.

不符合

  - Not progress: *Larry* can not enter his CS for the second time if *Jim* keep working in the remainder section.
  - Bounded-waiting: *Larry* set turn to “Jim” as he leaves his CS and the sleeping *Jim* can wake up

Software Solutions to Critical-Section Problem

- Algorithm.1 – P_i/P_j version.
  - Shared variable:

```
int turn = 0;
  /* initially turn = 0. turn = i means process i can enter
     its critical section */
```

  - Process P_i:

```
do {
    while (turn != i)
        sleep(1);  /* busy waiting */
    critical section of process i
    turn = j;
    remainder section of process i
} while (TRUE);
```

Software Solutions to Critical-Section Problem

- Algorithm.2 – Larry/Jim version.
  - Shared variables:

```
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
  /* Larry ready to enter his critical section*/
```

Process Larry:	Process Jim:

`do {`
`    while (flag_jim)`
`        sleep(1); /* busy waiting */`
`    flag_larry = TRUE;`
`    Larry's critical section`
`    flag_larry = FALSE;`
`      /* Jim can enter its CS */`
`    Larry's remainder section`
`} while (TRUE);`
	`do {`
	`    while (flag_larry)`
	`        sleep(1); /* busy waiting */`
	`    flag_jim = TRUE;`
	`    Jim's critical section`
	`    flag_jim = FALSE;`
	`      /* Larry can enter its CS */`
	`    Jim's remainder section`
	`} while (TRUE);`

  - Not mutual exclusion: `flag_larry` is FALSE at the start of the second iteration of *Larry*. If *Jim* start his iteration at this point, both processes will move into critical sections at the same time.

Software Solutions to Critical-Section Problem

- Algorithm.2 – P_i/P_j version
  - Shared variables:

```
boolean flag[2];
  /* initially flag[0] = flag[1] = FALSE */
flag [i] = TRUE;
  /* Pi ready to enter its critical section */
```

  - Process P_i:

```
do {
    while (flag[j]);
        sleep(1); /* busy waiting */
    flag[i] = TRUE;
    critical section of process i
    flag[i] = FALSE;
    remainder section of process i
} while (TRUE);
```

Software Solutions to Critical-Section Problem

- Algorithm.3 – Larry/Jim version
  - Shared variables:

```
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
  /* Larry ready to enter its critical section */
```

Process Larry:	Process Jim:

`do {`
`    flag_larry = TRUE;`
`    while (flag_jim)`
`        sleep(1); /* busy waiting */`
`    Larry's critical section`
`    flag_larry = FALSE;`
`    Larry's remainder section`
`} while (TRUE);`
	`do {`
	`    flag_jim = TRUE;`
	`    while (flag_larry)`
	`        sleep(1); /* busy waiting */`
	`    Jim's critical section`
	`    flag_jim = FALSE;`
	`    Jim's remainder section`
	`} while (TRUE);`

  - Not progress: Starting at the same time, *Larry* and *Jim* both will stick in waiting loops.

Software Solutions to Critical-Section Problem

- Algorithm.3 – P_i/P_j version
  - Shared variables:

```
boolean flag[2];
  /* initially flag[0] = flag[1] = FALSE */
flag[i] = TRUE;
  /* Pi ready to enter its critical section */
```

  - Process P_i:

```
do {
    flag[i] = TRUE;
    while (flag[j])
       sleep(1); /* busy waiting */
    critical section of process i
    flag [i] = FALSE;
    remainder section of process i
} while (TRUE);
```

Software Solutions to Critical-Section Problem

- Algorithm.4 – Larry/Jim version (*Peterson's solution*)
  - Combined shared variables of algorithms 1 and 2/3:

去看书

```
string turn = “Larry”;
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
```

Process Larry:	Process Jim:

<pre>do {
    flag_larry = TRUE;
    turn = “Jim”;
    while (flag_jim && turn==“Jim”);
        sleep(1); /* busy waiting */
    Larry’s critical section
    flag_larry = FALSE;
    Larry’s remainder section
} while (TRUE);</pre>
	<pre>do {
	    flag_jim = TRUE;
	    turn = “Larry”;
	    while (flag_larry && turn==“Larry”);
	        sleep(1); /* busy waiting */
	    Jim’s critical section
	    flag_jim = FALSE;
	    Jim’s remainder section
	} while (TRUE);</pre>

  - Algorithm.4 meets all essential criteria: mutual exclusion, progress, and bounded waiting; solves the critical section problem for two processes.

Software Solutions to Critical-Section Problem

- Algorithm.4 – P_i/P_j version (*Peterson's solution*)
  - Combined shared variables of algorithms 1 and 2/3:

```
int turn;
turn = i;
boolean flag[2];
flag[i] = TRUE;
```

  - Process P_i:

```
do {
    flag[i] = TRUE;   不可交换顺序
    turn = j;
    while (flag[j] && turn == j);
        sleep(1); /* busy waiting */
    critical section of process i
    flag[i] = FALSE;
    remainder section of process i
} while (TRUE);
```

Software Solutions to Critical-Section Problem

- *Peterson*'s solution
  - Restricted to two processes that alternate execution between their critical sections and remainder sections, algorithm.4 is known as *Peterson's solution* (*Gary. L. Peterson*, 1981).
  - *Peterson*'s algorithm provides a good algorithmic description of solving the critical-section problem. To prove that *Peterson*'s algorithm satisfied the three essential criteria of mutual exclusion, progress, and bounded waiting, refer to the Text Book of *A. Silberschatz*'s Operating System Concepts, 10th Edition, Chapter 6.3.
    - provided that changes to the variables turn, flag[0] and flag[1] propagate immediately and atomically.

      `__sync_lock_test_and_set(&turn, i)`
    - The do-while works even with preemption.

Software Solutions to Critical-Section Problem

- *Peterson*'s solution
  - Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that *Peterson*'s solution will work correctly on such architectures.
    - To improve system performance, processors and/or compilers may reorder read and write operations that have no dependencies.
  - For a single threaded application, this reordering is immaterial (非实质性的) as far as program correctness is concerned, as the final values are consistent with what is expected.
  - But for a multithreaded application with shared data, the reordering of instructions may render inconsistent or unexpected results.
  - Some memory barrier mechanism may be used to prevent this confusion.
    - Like __syn_synchronize() in Linux, or

```
#define barrier() __asm__ __volatile__("": : :"memory")
```

Software Solutions to Critical-Section Problem

- Algorithm.5 – Larry/Jim version
  - Like Algorithm.4, but with the first 2 instructions of the entry section swapped.
  - Question: is it still a correct solution? Is it mutual exclusive?
  - Shared variables:

```
string turn = “Larry”;
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
```

Process Larry:	Process Jimy:

`do {`
`turn = “Jim”;`
`flag_larry = TRUE;`
`while (flag_jim && turn==“Jim”);`
`sleep(1); /* busy waiting */`
Larry’s critical section
`flag_larry = FALSE;`
Larry’s remainder section
`} while (TRUE);`
	`do {`
	`turn = “Larry”;`
	`flag_jim = TRUE;`
	`while (flag_larry && turn==“Larry”);`
	`sleep(1); /* busy waiting */`
	Jim’s critical section
	`flag_jim = FALSE;`
	Jim’s remainder section
	`} while (TRUE);`

Software Solutions to Critical-Section Problem

- Algorithm.5 – Larry/Jim version

```
string turn = "Larry";
boolean flag_larry = TRUE;
Boolean flag_jim = FALSE;
```

Process Larry:	Process Jimy:

`do {`
`turn = "Jim";`
`flag_larry = TRUE;`
`while (flag_jim && turn=="Jim");`
`sleep(1); /* busy waiting */`
Larry's critical section
`flag_larry = FALSE;`
Larry's remainder section
`} while (TRUE);`
	`do {`
	`turn = "Larry";`
	`flag_jim = TRUE;`
	`while (flag_larry && turn=="Larry");`
	`sleep(1); /* busy waiting */`
	Jim's critical section
	`flag_jim = FALSE;`
	Jim's remainder section
	`} while (TRUE);`

Larry	Jim
	`turn = "Larry";`

`turn = "Jim";`
`flag_larry = TRUE;`
`(initially flag_jim == FALSE)`
Larry's critical section	

	`flag_jim = TRUE;`
	`(turn == "Jim")`
	Jim's critical section

Time Line

Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - *Peterson*'s algorithm can be generalized for more than two process. The following algorithm generalizes *Peterson*'s algorithm for N processes.
  - It uses N different levels
    - Each level represents another 'waiting room' before the critical section.
    - Each level will allow at least one process to advance, while keeping one process in waiting.

Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - Shared data:

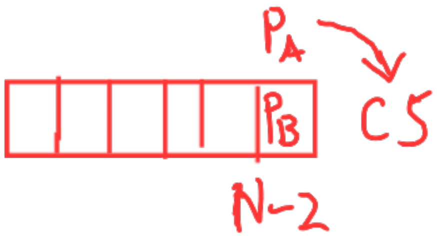
```
int level[N];
  /* current level number of processes 0..N-1 */
int waiting[N-1]; /* the waiting process of each level.
                     levels numbered 0..N-2 */
memset(level,(-1),sizeof(level));
memset(waiting,(-1),sizeof(waiting));
```

  - Process P_i:

```
for (lev = 0; lev < N-1; ++lev) {
    level[i] = lev;
    waiting[lev] = i;
    while (waiting[lev] == i && (there exists k ≠ i, such
that level[k] >= lev))
        sleep(1); /* busy waiting */
}
critical section of process i
level[i] = -1;
  /* allow other process of level n-2 to exit the while loop
     and enter his critical section */
remainder section of process i
```

level = N-1 才可进 CS

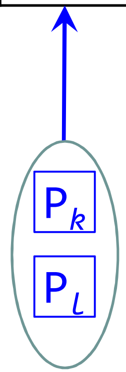
被踢出 waiting 或没有进程 level 比i高 才升级



Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - The process P_i reaching level N-1 (`level(i)==N-1`) will exit the **for** loop and enter his critical section.
  - Any process P_i would upgrade its level lev to lev+1 (i.e., exit the **while** loop) either:
    - Some other process P_j upgrades its level to the level of P_i (followed by `level(j)==lev` and `waiting[lev]==j`) or:
    - The level of any other process is less than lev.

Level No.	0	...	q	q+1	q+2	...	N-2	N-1
Waiting Room	P_m	. . .	P_j	P_i	P_t	. . .		

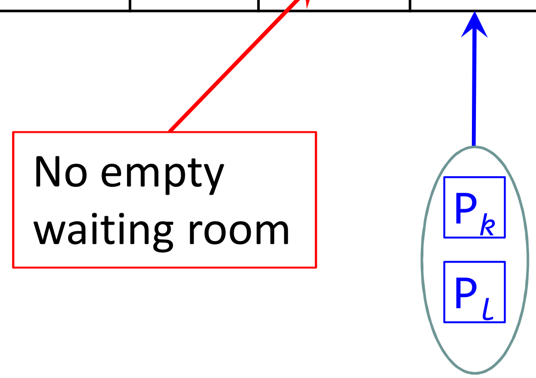


A process in the waiting room can not exits the while loop and upgrade even being scheduled, unless it is the only process with the current greatest level. Any process not in the waiting room will exit the while loop and upgrade if being scheduled.

Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - The process P_i reaching level N-1 (`level(i)==N-1`) will exit the **for** loop and enter his critical section.
  - Any process P_i would upgrade its level lev to lev+1 (i.e., exit the **while** loop) either:
    - Some other process P_j upgrades its level to the level of P_i (followed by `level(j)==lev` and `waiting[lev]==j`) or:
    - The level of any other process is less than lev.

Level No.	0	…	q	q+1	q+2	…	N-2	N-1
Waiting Room	P_m	. . .	P_j	P_i	P_t	. . .		

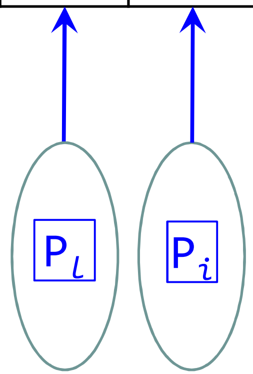


And thus any waiting room which level is less than the current greatest level must be occupied.

Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - The process P_i reaching level N-1 (`level(i)==N-1`) will exit the **for** loop and enter his critical section.
  - Any process P_i would upgrade its level lev to lev+1 (i.e., exit the **while** loop) either:
    - Some other process P_j upgrades its level to the level of P_i (followed by `level(j)==lev` and `waiting[lev]==j`) or:
    - The level of any other process is less than lev.

Level No.	0	...	q	q+1	q+2	...	N-2	N-1
Waiting Room	P_m	. . .	P_j	P_k	P_t	. . .		

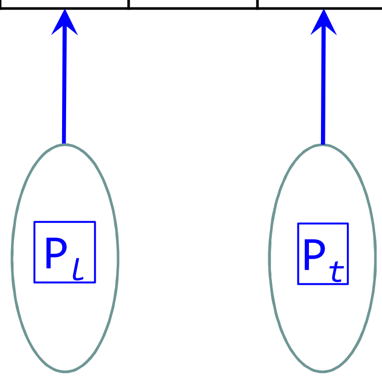


When P_k of level q is scheduled, it exits the while loop, upgrades its level to q+1 and occupies the waiting room of level q+1. P_i is moved out of the waiting room.

Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - The process P_i reaching level N-1 (`level(i)==N-1`) will exit the **for** loop and enter his critical section.
  - Any process P_i would upgrade its level lev to lev+1 (i.e., exit the **while** loop) either:
    - Some other process P_j upgrades its level to the level of P_i (followed by `level(j)==lev` and `waiting[lev]==j`) or:
    - The level of any other process is less than lev.

Level No.	0	…	q	q+1	q+2	…	N-2	N-1
Waiting Room	P_m	. . .	P_j	P_k	P_i	. . .		

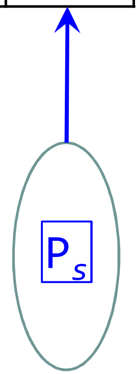


When P_i of level q+1 is scheduled, it exits the while loop, upgrades its level to q+2 and occupies the waiting room of level q+2. P_t is moved out of the waiting room.

Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - The process P_i reaching level N-1 (`level(i)==N-1`) will exit the **for** loop and enter his critical section.
  - Any process P_i would upgrade its level lev to lev+1 (i.e., exit the **while** loop) either:
    - Some other process P_j upgrades its level to the level of P_i (followed by `level(j)==lev` and `waiting[lev]==j`) or:
    - The level of any other process is less than lev.

Level No.	0	…	q	q+1	q+2	…	N-2	N-1
Waiting Room	P_m	. . .	P_j	P_k	P_i	. . .	P_t	



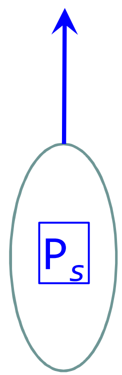
An extreme case: rooms of level 0 to N-2 are all occupied with P_t and P_s in the same level N-2. P_t can not exit the while loop even being scheduled because it has the same level with P_s.

Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - The process P_i reaching level N-1 (`level(i)==N-1`) will exit the **for** loop and enter his critical section.
  - Any process P_i would upgrade its level lev to lev+1 (i.e., exit the **while** loop) either:
    - Some other process P_j upgrades its level to the level of P_i (followed by `level(j)==lev` and `waiting[lev]==j`) or:
    - The level of any other process is less than lev.

Level No.	0	…	q	q+1	q+2	…	N-2	N-1
Waiting Room	P_m	. . .	P_j	P_k	P_i	. . .	P_t	

When P_s is scheduled, it exits the while loop because it is not in the waiting room, and immediately ends the for loop, entering its critical section.



Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - The process P_i reaching level N-1 (`level(i)==N-1`) will exit the **for** loop and enter his critical section.
  - Any process P_i would upgrade its level lev to lev+1 (i.e., exit the **while** loop) either:
    - Some other process P_j upgrades its level to the level of P_i (followed by `level(j)==lev` and `waiting[lev]==j`) or:
    - The level of any other process is less than lev.

Level No.	0	...	q	q+1	q+2	...	N-2	N-1
Waiting Room	P_m	. . .	P_j	P_k	P_i	. . .	P_t	

Processes entering critical sections would be in the order of their waiting room numbers N-2, N-3, …, 2, 1, and 0 finally.

Software Solutions to Critical-Section Problem

- *Peterson*'s algorithm for N processes.
  - It is not hard to prove that *Peterson*'s algorithm satisfied the three essential criteria of mutual exclusion, progress, and bounded waiting.
  - Exercise
    - Write a program with N competitive threads, using *Peterson*'s algorithm to solve the critical-section problem.

Software Solutions to Critical-Section Problem

alg.15-1-peterson-counter.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <signal.h>
#define MAX_N 1024
```

```
static int counter = 0;  /* number of process(s) in the critical section */
int level[MAX_N];  /* level of processes 0 .. MAX_N-1 */
int waiting[MAX_N-1];  /* waiting process of each level number 0 .. MAX_N-2 */
int max_num = 20; /* default max thread number */

static void *ftn(void *arg)
{
    int *numptr = (int *)arg;
    int thread_num = *numptr;
    int lev, k, j;

    printf("thread-%3d, ptid = %lu working\n", thread_num, pthread_self( ));
    for (lev = 0; lev < max_num-1; ++lev) { /* at least max_num-1 waiting rooms */
        level[thread_num] = lev;
        waiting[lev] = thread_num;
        while (waiting[lev] == thread_num) { /* busy waiting */
            for (k = 0; k < max_num; k++) {
                if(level[k] >= lev && k != thread_num)
                    break;
                if(waiting[lev] != thread_num)  /* check again */
                    break;
            }
            if(k == max_num) { /* lev greater than any other processes */
                break;
            }
        }
    }
```

Software Solutions to Critical-Section Problem

alg.15-1-peterson-counter.c (2)

```
        /* critical section of process thread_num */
    printf("thread-%3d, ptid = %lu entering the critical section\n", thread_num,
pthread_self( ));
    counter++;                  检查是否有多线程进入CS
    if (counter > 1) {
        printf("ERROR! more than one processes in their critical sections\n");
        kill(getpid(), SIGKILL);
    }
    counter--;
        /* end of critical section */

    level[thread_num] = -1;
        /* allow other process of level max_num-2 to exit the while loop
           and enter his critical section */
    pthread_exit(0);
}

int main(int argc, char *argv[])
{
    printf("Usage: ./a.out total_thread_num\n");
    if(argc > 1) {
        max_num = atoi(argv[1]);
    }
    if (max_num < 0 || max_num > MAX_N) {
        printf("invalid max_num\n");
        exit(1);
    }
```

Software Solutions to Critical-Section Problem

- alg.15-1-peterson-counter.c (3)

```
    memset(level, (-1), sizeof(level));
    memset(waiting, (-1), sizeof(waiting));

    int i, ret;
    int thread_num[max_num];
    pthread_t ptid[max_num];

    for (i = 0; i < max_num; i++) {
        thread_num[i] = i;
    }
    printf("total thread number = %d\n", max_num);
    printf("main(): pid = %d, ptid = %lu.\n", getpid( ), pthread_self( ));

    for (i = 0; i < max_num; i++) {
        ret = pthread_create(&ptid[i], NULL, &ftn, (void *)&thread_num[i]);
        if(ret != 0)
            fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
    }

    for (i = 0; i < max_num; i++) {
        ret = pthread_join(ptid[i], NULL);
        if(ret != 0)
           perror("pthread_join()");
    }

    return 0;
}
```

防止冲突

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.15-1-peterson-counter.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out 10
Usage: ./a.out total_thread_num
total thread number = 10
main(): pid = 113819, ptid = 140496837703488.
thread-  0, ptid = 140496829191936 working
thread-  0, ptid = 140496829191936 entering the critical section
thread-  4, ptid = 140496795621120 working
thread-  3, ptid = 140496804013824 working
thread-  5, ptid = 140496787228416 working
thread-  2, ptid = 140496812406528 working
thread-  8, ptid = 140496627832576 working
thread-  9, ptid = 140496762050304 working
thread-  1, ptid = 140496820799232 working
thread-  6, ptid = 140496778835712 working
thread-  4, ptid = 140496795621120 entering the critical section
thread-  3, ptid = 140496804013824 entering the critical section
thread-  9, ptid = 140496762050304 entering the critical section
thread-  1, ptid = 140496820799232 entering the critical section
thread-  2, ptid = 140496812406528 entering the critical section
thread-  8, ptid = 140496627832576 entering the critical section
thread-  5, ptid = 140496787228416 entering the critical section
thread-  6, ptid = 140496778835712 entering the critical section
thread-  7, ptid = 140496770443008 working
thread-  7, ptid = 140496770443008 entering the critical section
isscgy@ubuntu:/mnt/os-2020$
```

Software Solutions to Critical-Section Problem

- *Lamport*'s Bakery Algorithm
  - By *Leslie Lamport*, 1974
    - Inventor of LaTex, Paxos algorithm, 2013 Turing Award winner.
  - Critical Section for n processes:
    - Before entering its critical section, each process receives a number (or a ticket, like in a bakery). The process holding *the smallest number* enters the critical section.
    - The numbering scheme here always generates numbers in increasing order of enumeration without upper bounds. For example:

```
1, 2, 3, 3, 3, 3, 4, 5, ...
```

    - Suppose that processes P_i and P_j (PID assumed unique) receive the same number.
      - If $i < j$, then P_i has priority over P_j in entering the critical section.

Software Solutions to Critical-Section Problem

- *Lamport*'s Bakery Algorithm
  - Choosing a number from $\max(a_0, \ldots, a_{n-1}) + 1$:
    - Function $\max(a_0, \ldots, a_{n-1})$ returns a number k, such that

      $k \geq a_i$ for $0 \leq i \leq n - 1$. 大顶堆
  - Notation for lexicographical order (ticket #, PID #)

    (a, b) < (c, d) if a < c or if a == c and b < d.
  - Shared data:

```
boolean choosing[n];
int number[n];
```

  - Data structures are initialized to FALSE and 0, respectively.
  - choosing[i] == TRUE means process P_i is getting its number.
  - number[i] == 0 means process P_i is removed out of the waiting list.
  - If process P_i is failing in execution, number[i] is compelled to 0.

Software Solutions to Critical-Section Problem

- *Lamport*'s Bakery Algorithm
  - Process P_i:

```
do {
    choosing[i] = TRUE;
    number[i] = max(number[0], …, number[n – 1]) + 1;
    choosing[i] = FALSE;
    for (j = 0; j < n; j++) { /* be sure that each of the n
   process has got a number */
        while (choosing[j]);
          /* wait until process j receives its number */
        while ((number[j] != 0) &&
             ((number[j], j) < (number[i], i)));
          /* wait until all processes with smaller numbers or
             with the same number, but with higher priority,
             leave their critical sections */
    }
    critical section of process i
    number[i] = 0;
    remainder section of process i
} while (TRUE);
```

Software Solutions to Critical-Section Problem

- *Lamport*'s Bakery Algorithm – another description

```
Lock(int i) {
    choosing[i] = TRUE;
    number[i] = max(number[0], …, 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)));
}

Unlock(int i) {
    number[i] = 0;
}
```

```
Process (int i) {
    while (TRUE) {
        Lock(i);
        critical section of process i
        Unlock(i);
        remainder section of process i
    }
}
```

Software Solutions to Critical-Section Problem

- *Lamport*'s Bakery Algorithm
  - Bakery algorithm meets all the three essential criteria
    - Mutual Exclusion
    - Progress
    - Bounded Waiting
  - It solves the critical section problem for more processes with shared-memory, need no more supports such as atomic instruction set-and-test or semaphores.
    - choosing[i] and number[i] are modified only by P_i.
  - Bakery algorithm is deadlock-free without starvation.
    - There must be a process with least number in the waiting list, getting the permission to enter its critical section.
    - Any FIFO and deadlock-free algorithm must be starvation-free.

Software Solutions to Critical-Section Problem

- *Eisenberg-McGuire*'s Algorithm
  - The *Eisenberg & McGuire algorithm* (*Murray A. Eisenberg* and *Michael R. McGuire*, 1972) is a correct solution solving the critical sections problem for the N-process case. It is also a general version of the dining philosophers problem.
- Shared data:

```
enum pstates {IDLE, WAITING, ACTIVE};
pstates flags[n];
int turn;
```

- Initially the variable turn is set arbitrarily to a number between 0 and n−1, representing the process chosen to enter its critical section.
- The flags variable for each process is initialized to IDLE and is set to WAITING whenever it intends to enter the critical section.
- Values of flags[i]
  - WAITING: process P_i is waiting for the resource.
  - ACTIVE: P_i is *tentatively* claiming the resource (not allocated yet).
  - IDLE: for other cases.

Software Solutions to Critical-Section Problem

- *Eisenberg-McGuire*'s Algorithm
  - Initialization:

```
int index; /* index is local, not shared! */
  ...
turn = 0;
  ...
for (index = 0; index < n; index++) {
    flags[index] = IDLE;
}
```

Software Solutions to Critical-Section Problem

- *Eisenberg-McGuire*'s Algorithm
  - Entry Protocol (for process P_i):

```
repeat {
      /* announce that process i need the resource */
    flags[i] = WAITING;

      /* scan processes from the one with the turn up to i */
      /* repeat if necessary until the scan finds all processes
         in IDLE */
    index = turn;
    while (index != i) { /* exit if all flags from turn to i
clockwise are IDLE */
        if (flag[index] != IDLE)
            index = turn;
        else
            index = (index + 1) mod n;
    }


      /* now tentatively claim the resource */
    flags[i] = ACTIVE;
```

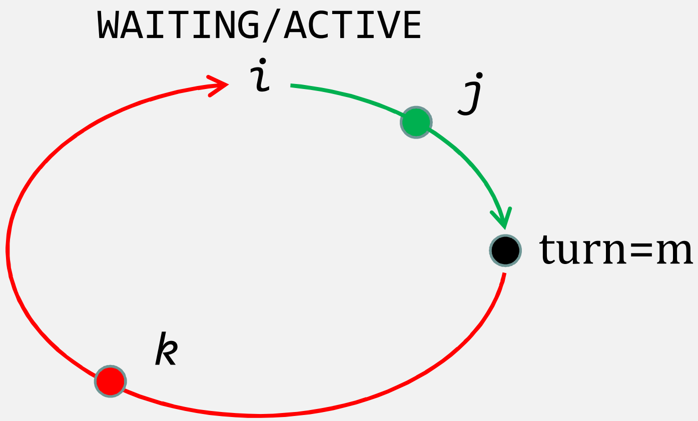
顺时针从turn开始到i

The Critical-Section Problem

Software Solutions to Critical-S

- *Eisenberg-McGuire*'s Algorithm
  - Entry Protocol (for process P_i

```
repeat {
      /* announce that process
    flags[i] = WAITING;

      /* scan processes from t
      /* repeat if necessary u
         in IDLE */
    index = turn;
    while (index != i) { /* exit if all flags from turn to i
clockwise are IDLE */
        if (flag[index] != IDLE)
            index = turn;
        else
            index = (index + 1) mod n;
    } /* by his protocol, other flags from i up to the turn
(however it moving forward clockwise) cannot change from WAITING
to ACTIVE until flag[i] gets IDLE from the EXIT protocol */
      /* now tentatively claim the resource */
    flags[i] = ACTIVE;
```



flags[j] can not changed to ACTIVE because the scan clockwise from m up to j will pass i where flag[i] $\neq$ IDLE. It can not be guaranteed for flag[k].

Software Solutions to Critical-Section Problem

- *Eisenberg-McGuire*'s Algorithm
  - Entry Protocol (for process P_i):

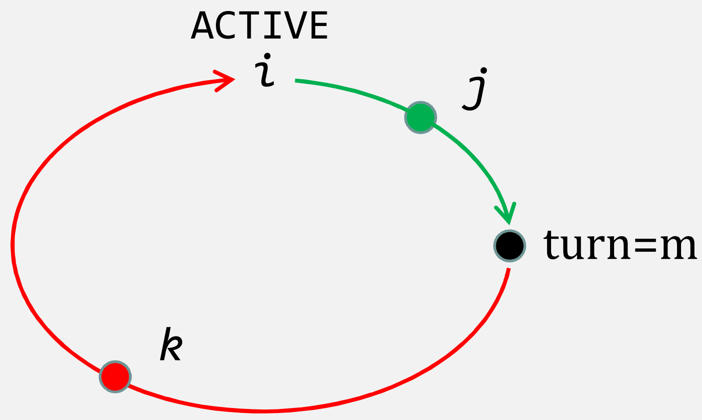
```
    /* find the first active process besides Pi, if any */
  index = 0; /* scan from 0 to n-1 */
  while ((index < n) &&
                  ((index == i) || (flags[index] != ACTIVE))) {
      index = index + 1;
  }




    /* if there were no other active processes than Pi, AND if
Pi have the turn or else whoever has it is IDEL, then exits the
repeat loop and proceed. Otherwise, repeat the whole sequence. */
} until ((index >= n) &&
                   ((turn == i) || (flags[turn] == IDLE)));

  /* claim the turn and proceed */
  /* the resource is allocated to Pi */
turn = i;
```

The Critical-Section Problem

Software Solutions to Critical-S

- *Eisenberg-McGuire*'s Algorithm
  - Entry Protocol (for process P_i

```
    /* find the first active
  index = 0; /* scan from 0 t
  while ((index < n) &&
                    ((index ==
      index = index + 1;
  }
```



Suppose *k* is the smallest number where flags[*k*] is ACTIVE. When process *m* returning , the *turn* moves forward and gets *k*.

```
/* currently flags from the turn up to i may have changed to
ACTIVE. In this case when process m returning the resource, the
turn moves forward and gets the first non-IDLE process previous
to i. Repeat the whole sequence with flag[i]=WATING again */
} until ((index >= n) &&
                    ((turn == i) || (flags[turn] == IDLE)));

  /* claim the turn and proceed */
  /* the resource is allocated to Pi */
turn = i;
```

Software Solutions to Critical-Section Problem

- *Eisenberg-McGuire*'s Algorithm
  - Exit Protocol (for Process P_i):

```
  /* turn == i */
  /* find a process which is not IDLE */
  /* if there are no others, we will find Pi */
index = (turn + 1) mod n;
while (flags[index] == IDLE) {
    index = (index + 1) mod n;
}

  /* give the turn to someone that needs it, or keep it */
turn = index;

  /* we're finished now */
flag[i] = IDLE;
```

- *Eisenberg-McGuire* Algorithm still has the busy-waiting problem.

Software Solutions to Critical-Section Problem

- What about process failures?
  - If all three criteria
    - Mutual Exclusion
    - Progress
    - Bounded Waiting

    are satisfied, then a valid solution will provide robustness against failure of a process *in its remainder section*.
    - since failure in remainder section is just like having an infinitely long remainder section.
  - However, no valid solution can provide robustness against a process *failing in its critical section*.
    - A process P_i that fails in its critical section does not signal its failure to other processes
      - for them P_i is still in its critical section.

Software Solutions to Critical-Section Problem

- Drawbacks of software solutions to critical section problem
  - Software solutions are very delicate.
  - Processes that are requesting to enter their critical sections are *busy waiting*.
    - It consumes processor time needlessly.
    - If critical sections are long, it would be more efficient to block processes that are waiting.