InterProcess Communication & Synchronization

I233 Operating Systems

InterProcess Communication & Synchronization

- 2.3 INTERPROCESS COMMUNICATION
  - 2.3.1 Race Conditions
  - 2.3.2 Critical Regions
  - 2.3.3 Mutual Exclusion with Busy Waiting
  - 2.3.4 Sleep and Wakeup
  - 2.3.5 Semaphores
  - 2.3.6 Mutexes
  - 2.3.7 Monitors
  - 2.3.8 Message Passing
  - 2.3.9 Barriers
  - 2.3.10 Avoiding Locks: Read-Copy-Update
- 2.5 CLASSICAL IPC PROBLEMS
  - 2.5.1 The Dining Philosophers Problem
  - 2.5.2 The Readers and Writers Problem

Why we discuss

InterProcess “Communication”

and

“Synchronizing”

at the same time?

Three issues to be solved

- How one process can pass information to another.
- How to correctly control reading and writing to shared memory by two or more processes.
- A way to execute processes in the proper sequencing when dependencies are present.

These are all the same issues.

Race Conditions

- What is “Race Conditions” ?
  - Where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when.

Race Conditions

- What is “Race Conditions” ?
  - Where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when.

```
spool_file ( file )
{
        next_free_slot = read_int ( in );
        register_file ( next_free_slot, name );
        next_free_slot ++;
        write_int ( in, next_free_slot );
}
```

Procedures to be executed by Process A & B

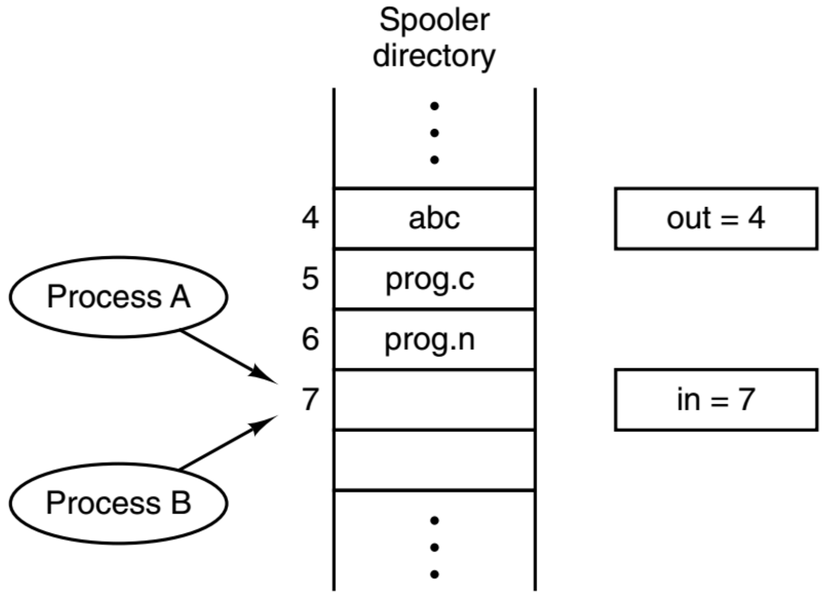
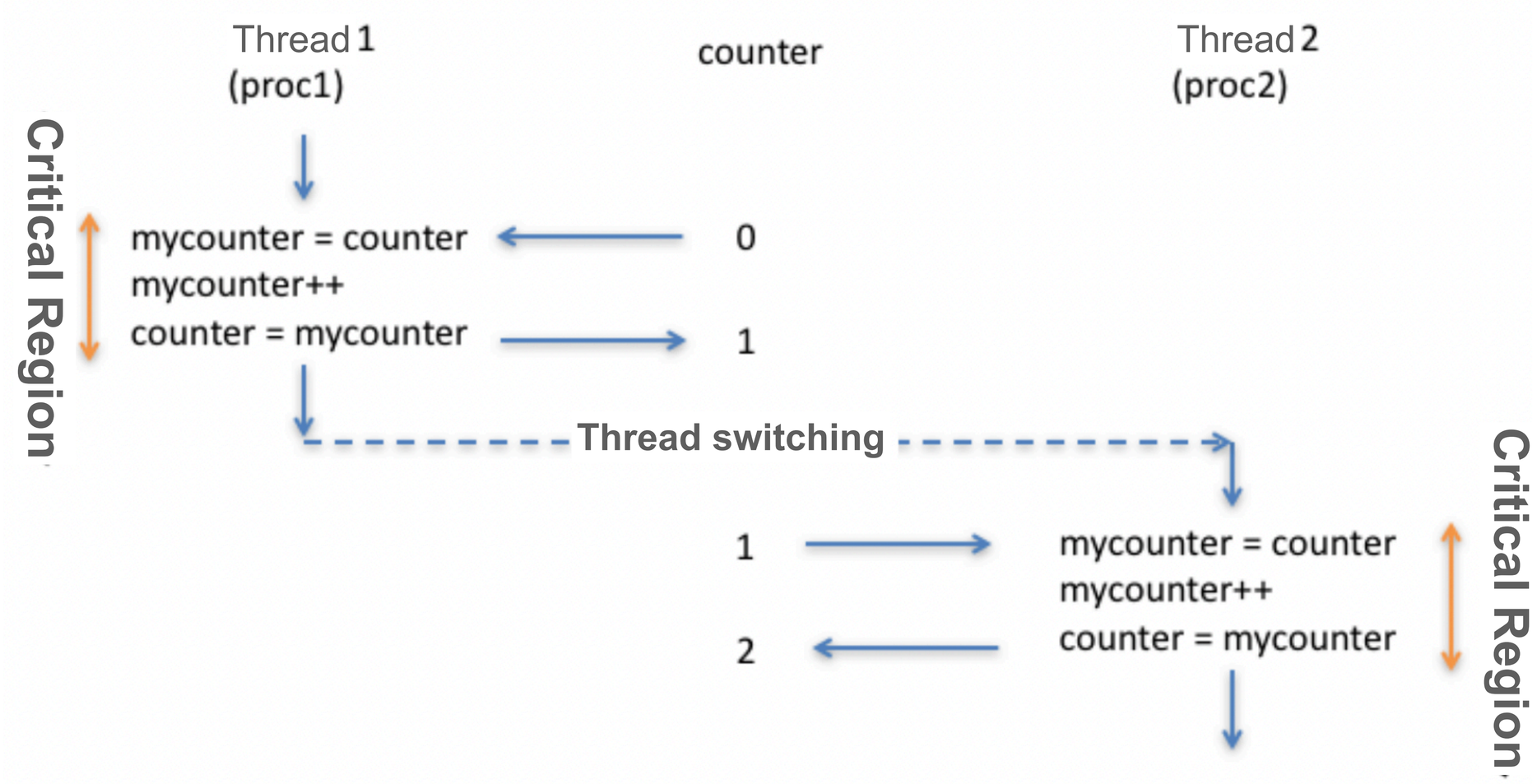


Figure 2-21. Two processes want to access shared memory at the same time.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.120.

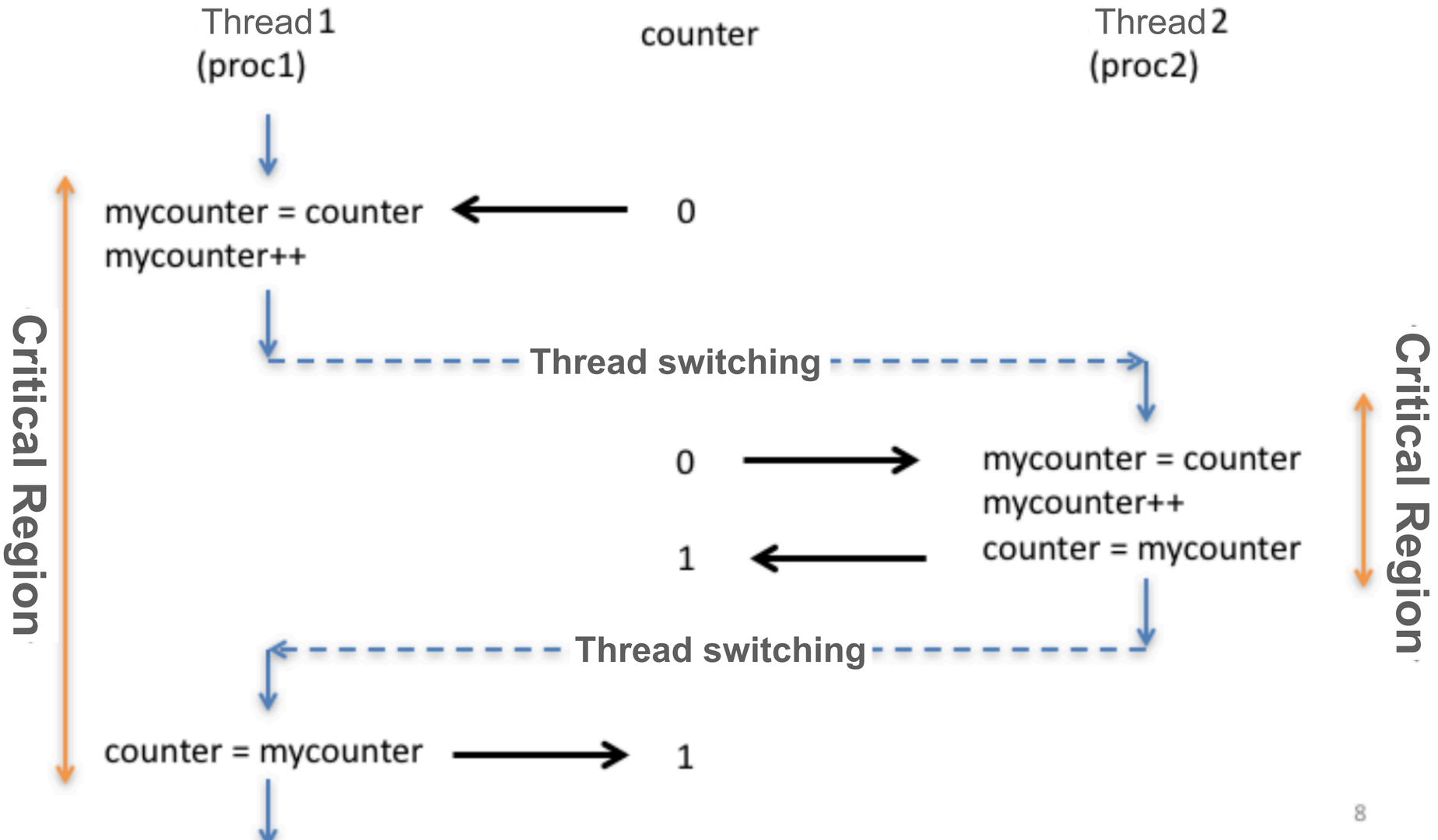
Example on thread: Lost update (1)

- Run without lost update:



Example on thread: Lost update (2)

- Run with lost update:



Critical Region

- In order to avoid race conditions, it is necessary to prevent other processes from entering the critical region (mutual exclusion).

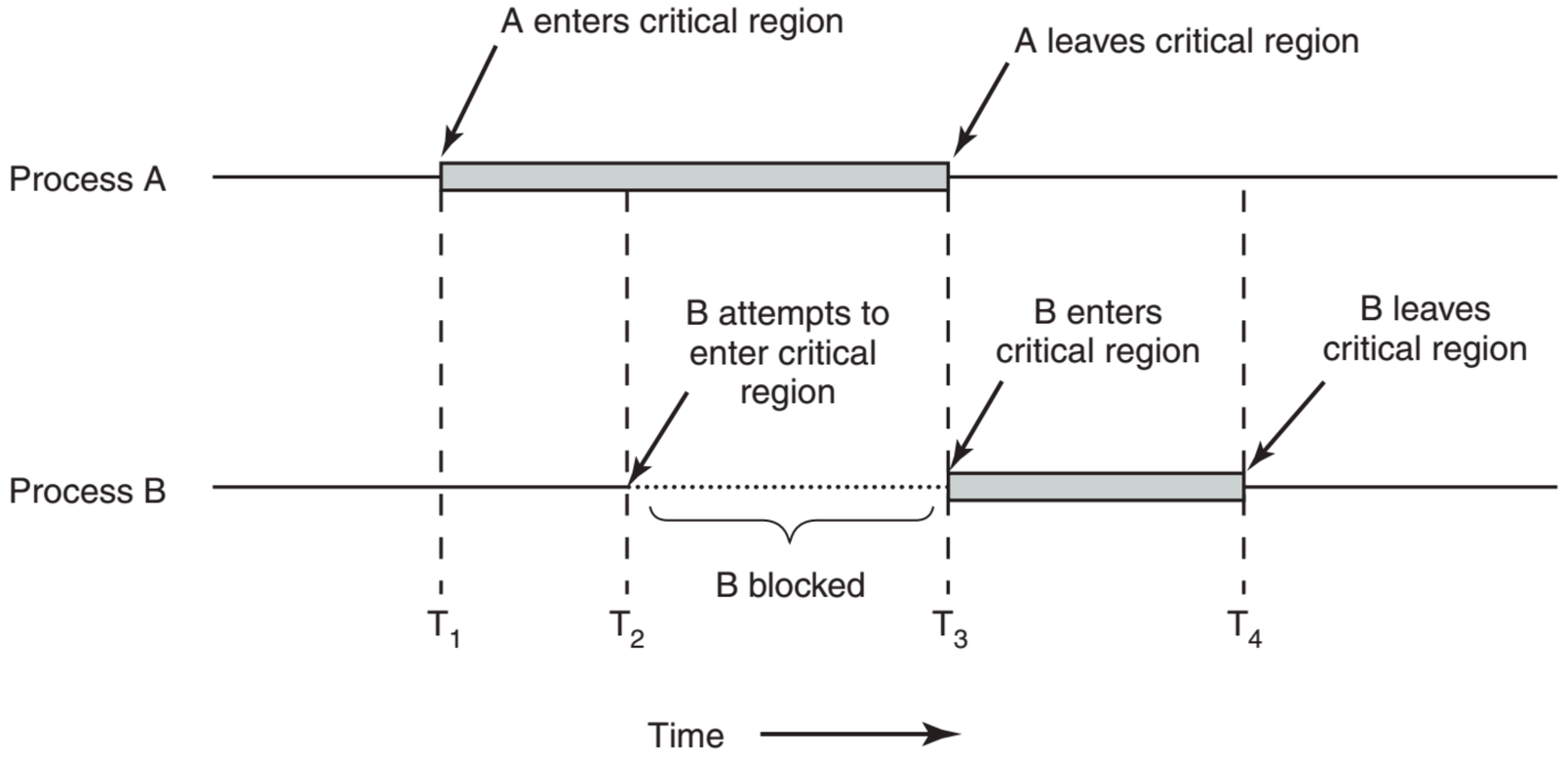


Figure 2-22. Mutual exclusion using critical regions.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.122.

Some undesirable solutions: Disabling Interrupts

- Disable (inhibit) interrupt
  - A method of disallowing (disabling) interrupts at the timing when a process enters a critical region so that switching of processes (or threads) does not occur.
  - Unfavorable points:
    - The user must be granted the authority to control the interrupt.
    - In a multiprocessor system, interrupts are disabled only on the processor that has executed the interrupt disable instruction.
  - It is often the case that the kernel disables interrupts for a very short time within itself.

Some undesirable solutions: Lock variables

- Attempt to mutually exclude each other with a lock variable that indicates whether or not to enter the critical region.
  - Unfavorable points:
    - “`while ( lock == 1 ); lock = 1;`” and “`lock = 0;`” will form a separate critical region for the lock variable “lock”.
    - Lock checking will be performed with busy wait.

```
while ( lock == 1 );     // wait until it unlocks.
lock = 1;                // lock it

critical_region();       // execute critical region.

lock = 0;                // unlock it

non_critical_region();
```

Some undesirable solutions: Strict Alternations

- Complete alternating (or round-robin) execution
  - Use a control variable that indicates which process can enter the critical region next.
  - The process will busy wait (continuously check the value of the control variable) until the value of the control variable matches its own number.
  - Unfavorable points:
    - Alternation (or round robin) must be required.
    - Problems with differences in process execution time.

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.124.

Peterson's Solution

Software solution to the mutual exclusion problem

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Source: Andrew S. Tanenbaum. "Modern Operating Systems", 4th ed., global ed., pp.125.

Summary of software based solutions

- Mutual exclusion by software is not easy.
  - Disabling Interrupts (never the preferred solution)
  - Always accompanied by a busy wait if interrupts are not disabled.
  - Lock variables (not the correct solution)
  - Strict Alternations (some flaws)
  - Peterson’s Solution (non-trivial code & busy wait)

RMW Instructions

- In order to give a general solution to the mutual exclusion problem, we need a hardware mechanism that performs the inspection and writing (locking) of shared variables in an "atomic" manner.
- Machine instructions with the RMW (read-modify-write) attribute can be used for this purpose.
- Example of RMW instruction:
  - `TSL rx, lock` : Reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock. (sample on textbook)
- Other instructions:
  - e.g. `XCHG rx, lock` : exchanges the contents of two locations atomically, for example, a register and a memory word.
- Some instructions will work correctly even in a multiprocessor environment. In X86, this is possible by qualifying the RMW instructions with the LOCK prefix.

Lock (Mutual exclusion) by TSL Instruction (w/ busy wait)

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was not zero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.127.

The need for sleep and wakeup

- Issue on the busy wait
  - Wasting CPU time
  - Priority inversion problem
    - If a process has an execution priority, once the process with the higher priority starts busy wait, the process with the lower priority cannot be unlocked, and the busy wait will not be terminated.
- Introduce “sleep” and “wakeup”
  - When a process is unable to enter a critical region, it immediately abandons its own execution (gives up its execution rights to another process). = “sleep”
  - A process that exits a critical region sends a signal to the sleeping process that it is ready to enter. = “wakeup”

However... lost wakeup problem

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.129.

The essence of the lost wakeup problem

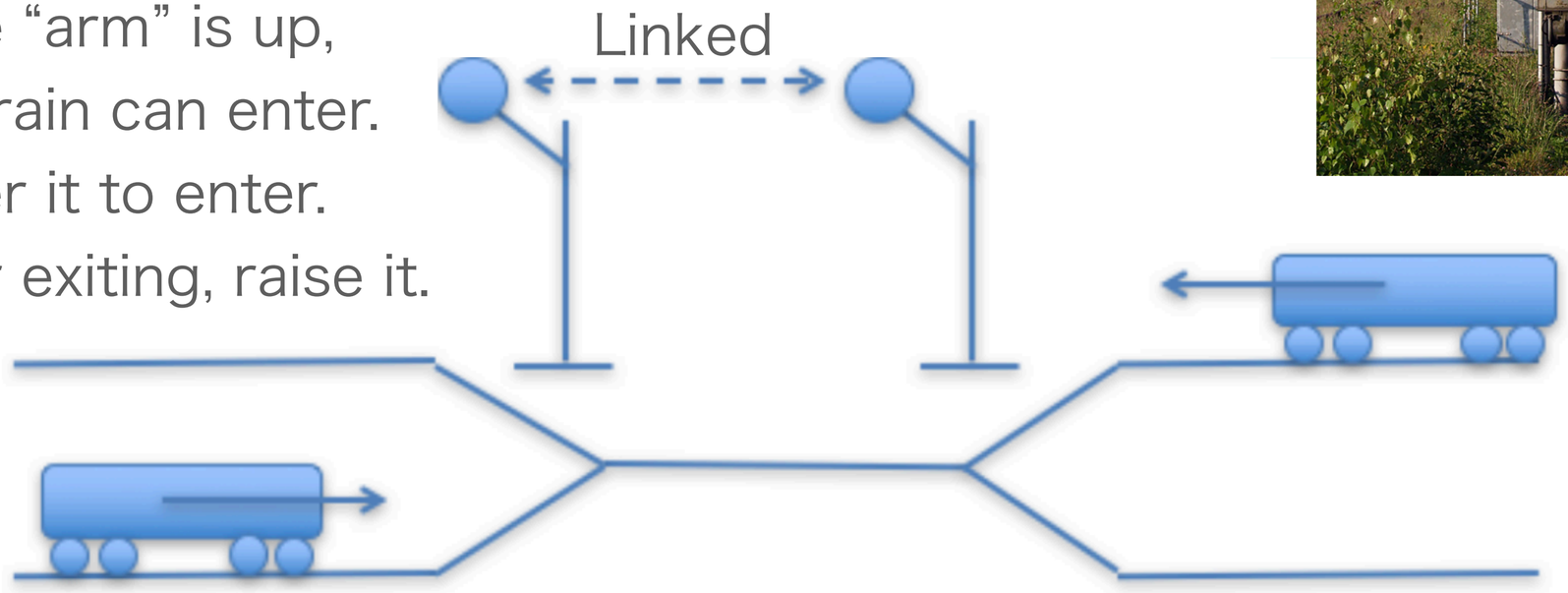
- The loss of wakeup signal sent to a process that is not yet in sleep.
- Add a new variable to hold wakeups for processes that are not sleeping, so that processes that are going to sleep can check this variable and if a wakeup has already been received, simply reset this variable to continue execution without sleeping.

Various synchronizing mechanisms

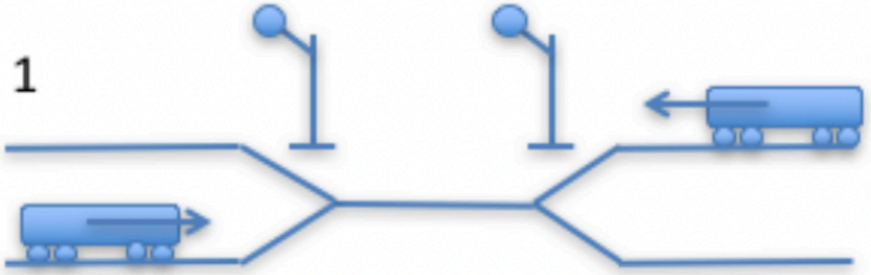
Semaphore

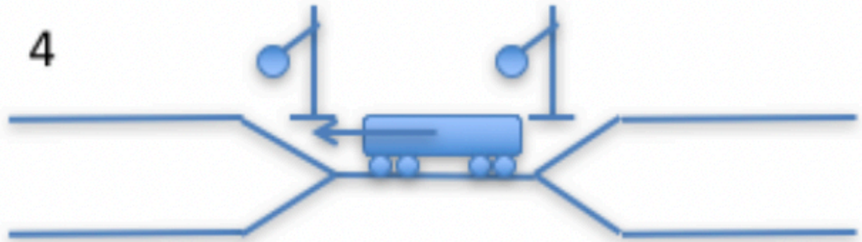
- Analogy of an "Semaphore signal" controlling entry to a single-track section.

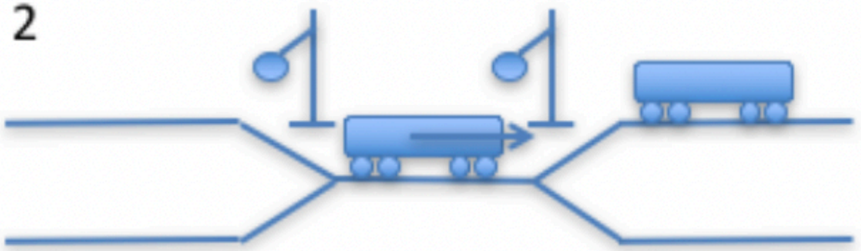
If the “arm” is up,
the train can enter.
Lower it to enter.
After exiting, raise it.

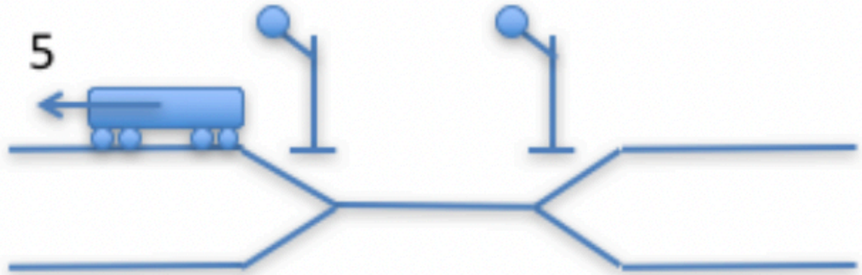


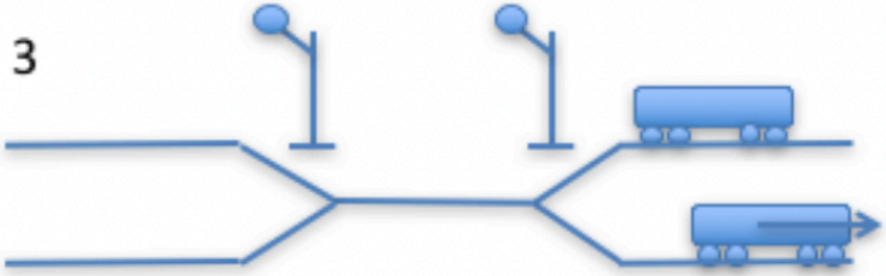
Semaphore (2)











Semaphore (3)

- Semaphore operations
  - down(sem)
    1. If sem is already down, wait until it goes up, then go to 1 again.
    2. Lower the sem
  - up(sem)
    1. Raise the sem
- Classic: the down operation was called the “P” operation, and the up operation was called the “V” operation. (“P”rolagen: decrease semaphore, “V”erhogen: increase semaphore)
- Instead of the state of the arm, there is also a "multi-valued semaphore" that has a value representing the number of available resources and waits if it is zero. (v.s. binary semaphore)

Solving the Producer-Consumer Problem Using Semaphores

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                          /* decrement empty count */
          down(&mutex);                          /* enter critical region */
          insert_item(item);                     /* put new item in buffer */
          up(&mutex);                            /* leave critical region */
          up(&full);                             /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                           /* decrement full count */
          down(&mutex);                          /* enter critical region */
          item = remove_item( );                 /* take item from buffer */
          up(&mutex);                            /* leave critical region */
          up(&empty);                            /* increment count of empty slots */
          consume_item(item);                    /* do something with the item */
     }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.131.

Mutex — shared variable managing mutual exclusion

- A simplified version of the semaphore for managing mutual exclusion

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Figure 2-29. Implementation of *mutex_lock* and *mutex_unlock*.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.133.

Monitor — A higher-level synchronization primitive

- By using semaphores, the cooperative process can be written correctly.
- However, it is not always easy.

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .           .           .
        end;
end monitor;
```

Figure 2-33. A monitor.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.140.

The producer-consumer problem with monitors

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.141.

The producer-consumer problem with Java

```
public class ProducerConsumer {
    static final int N = 100;        // constant giving the buffer size
    static producer p = new producer( );     // instantiate a new producer thread
    static consumer c = new consumer( ); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor( );       // instantiate a new monitor

    public static void main(String args[ ]) {
        p.start( );       // start the producer thread
        c.start( );       // start the consumer thread
    }

    static class producer extends Thread {
        public void run( )  { // run method contains the thread code
            int item;
            while (true) {      // producer loop
                item = produce_item( );
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... }       // actually produce
    }

    static class consumer extends Thread {
        public void run( )  { run method contains the thread code
            int item;
            while (true) {      // consumer loop
                item = mon.remove( );
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... }// actually consume
    }

    static class our_monitor {   // this is a monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0;   // counters and indices

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
            buffer [hi] = val;  // insert an item into the buffer
            hi = (hi + 1) % N;        // slot to place next item in
            count = count + 1;       // one more item in the buffer now
            if (count == 1) notify( );        // if consumer was sleeping, wake it up
        }

        public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
            val = buffer [lo];  // fetch an item from the buffer
            lo = (lo + 1) % N;        // slot to fetch next item from
            count = count – 1;       // one few items in the buffer
            if (count == N – 1) notify( );  // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

Figure 2-35. A solution to the producer-consumer problem in Java.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.142.

Message Passing

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}


void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Figure 2-36. The producer-consumer problem with N messages.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.146.

Barriers

- Synchronization mechanism required when multiple processes are performing phase-aware computations.

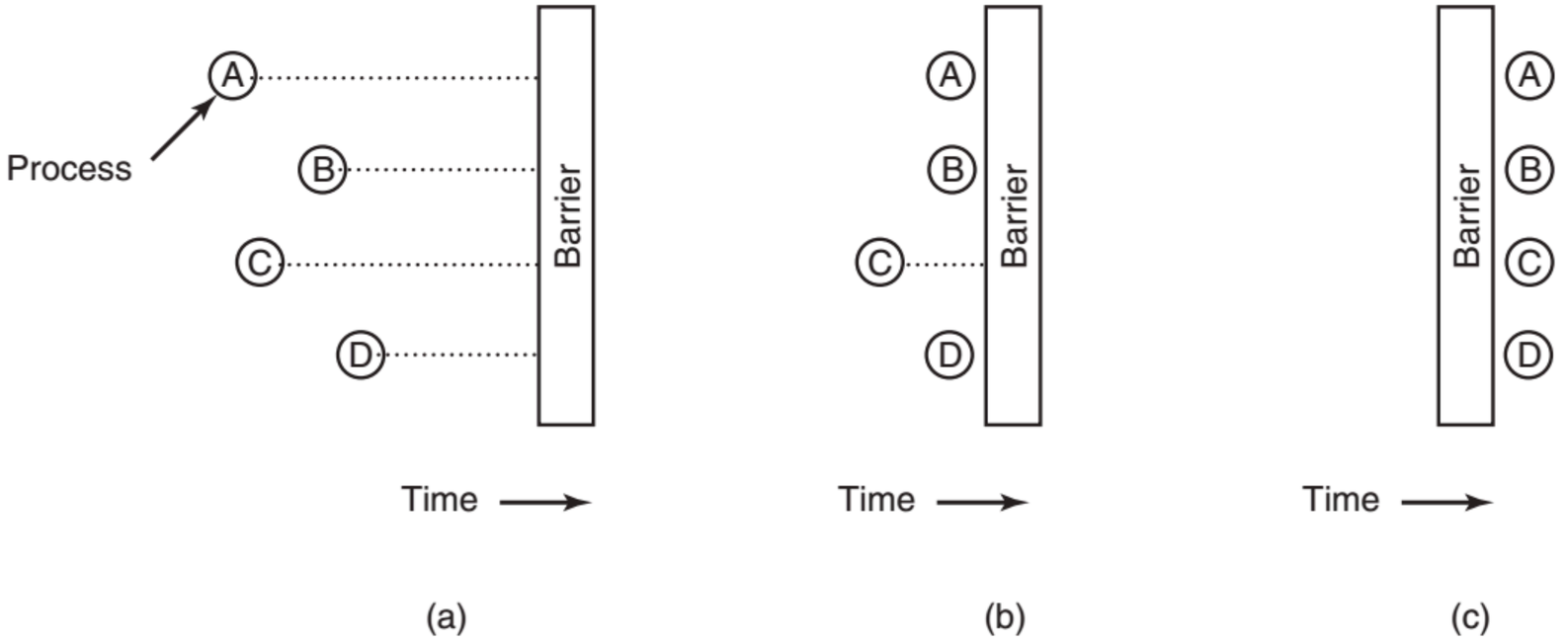


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.147.

Capabilities of synchronizing primitives (1)

<Proposition> Do the various synchronization primitives have differences high/low in their descriptive ability (ability to solve synchronization problems)?

- For methods a and b, if question p can be described by a but not by b, then the descriptive ability of a is higher than that of b. (denote this as "$p \mid a > b$")

- In order to show that it is $^{\forall}p \in P \mid a > b$, we need try to describe all problems $p \in P$ (P is the set of all synchronization problems) ?

Capabilities of synchronizing primitives (2)

- If b can be described by a, then $^{\forall}p \in P \mid a \geq b$.
  - All problems that can be described in terms of b can also be described in terms of a by using b that describes b in terms of a.
  - Some problems that cannot be written in b can be written in a.
  - Some problems that cannot be described by b cannot be described by a either.
- If a can be described by b, then $^{\forall}p \in P \mid a \leq b$.
- If the above two equations are true simultaneously, then $^{\forall}p \in P \mid a = b$.
  - Only equal capability for all problems, not whether or not they can be described.

Capabilities of synchronizing primitives (3)

- How to proof that the capabilities of multiple synchronization primitives are all equal?
  - Describing the Mutex with the Message Passing.
  - Describing the Message Passing with the Mutex.
  - Describing the Mutex with the Semaphore.
  - Describing the Semaphore with the Mutex.

The Dining Philosophers Problem

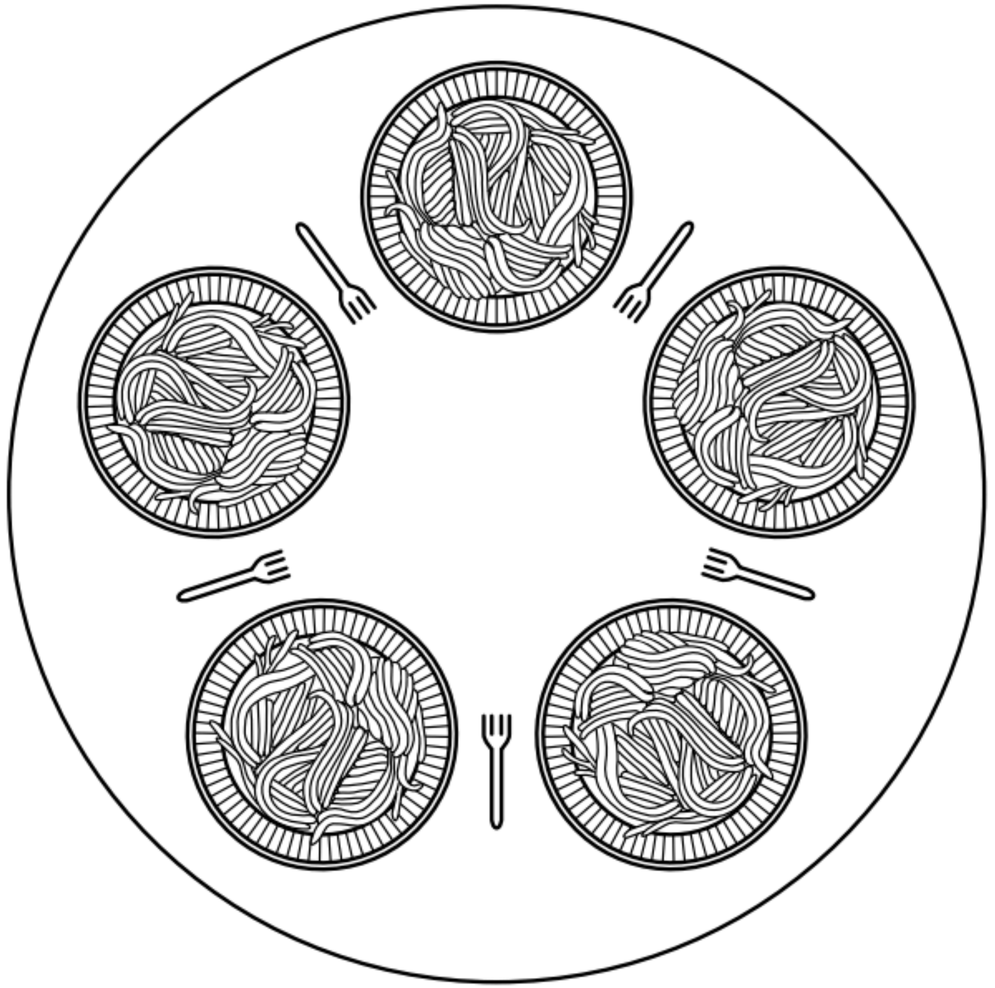


Figure 2-45. Lunch time in the Philosophy Department.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.168.

Deadlock or starvation

- Another problem on synchronization: Starvation

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                       /* philosopher is thinking */
                take_fork(i);                   /* take left fork */
                take_fork((i+1) % N);           /* take right fork; % is modulo operator */
                eat( );                         /* yum-yum, spaghetti */
                put_fork(i);                    /* put left fork back on the table */
                put_fork((i+1) % N);            /* put right fork back on the table */
        }
}
```

Figure 2-46. A nonsolution to the dining philosophers problem.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.169.

A solution to The Dining Philosopher Problem

```
#define N           5                   /* number of philosophers */
#define LEFT        (i+N–1)%N           /* number of i's left neighbor */
#define RIGHT       (i+1)%N             /* number of i's right neighbor */
#define THINKING    0                   /* philosopher is thinking */
#define HUNGRY      1                   /* philosopher is trying to get forks */
#define EATING      2                   /* philosopher is eating */

typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */


void philosopher(int i)                 /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                      /* repeat forever */
        think( );                       /* philosopher is thinking */
        take_forks(i);                  /* acquire two forks or block */
        eat( );                         /* yum-yum, spaghetti */
        put_forks(i);                   /* put both forks back on table */
    }
}

void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
    test(i);                            /* try to acquire 2 forks */
    up(&mutex);                         /* exit critical region */
    down(&s[i]);                        /* block if forks were not acquired */
}
```

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
    test(i);                            /* try to acquire 2 forks */
    up(&mutex);                         /* exit critical region */
    down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = THINKING;                /* philosopher has finished eating */
    test(LEFT);                         /* see if left neighbor can now eat */
    test(RIGHT);                        /* see if right neighbor can now eat */
    up(&mutex);                         /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Figure 2-47. A solution to the dining philosophers problem.

Source: Andrew S. Tanenbaum. "Modern Operating Systems", 4th ed., global ed., pp.170.

The Readers and Writers Problem

- Many competing processes wishing to read and write a database.
- It is acceptable to have multiple processes reading the database at the same time.
- But if one process is updating (writing) the database, no other processes may have access to the database, not even readers.
- We have to avoid starving the write process due to the read processes arriving one after another.

A solution to The Readers and Writers Problem

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

Figure 2-48. A solution to the readers and writers problem.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.171.

日本語資料

プロセス間通信・同期

- 2.3 プロセス間通信
  - 2.3.1 競合状態
  - 2.3.2 クリティカルリージョン
  - 2.3.3 ビジーウェイトを用いた相互排除
  - 2.3.4 スリープとウェイクアップ
  - 2.3.5 セマフォ
  - 2.3.6 Mutex
  - 2.3.7 モニタ
  - 2.3.8 メッセージ通信
  - 2.3.9 バリア
  - 2.3.10 ロックの回避: Read-Copy-Update
- 2.5 古典的なプロセス間通信問題
  - 2.5.1 哲学者の食事問題
  - 2.5.2 リーダ・ライタ問題

なぜ

プロセス間『通信』

と

『同期』

を同時に扱うのか?

解決すべき３つの問題

- あるプロセスから別のプロセスへ情報を渡す方法。
- 複数のプロセスによる共有メモリへの読み書きを正しく制御する方法。
- プロセス間に依存関係がある場合に、正しい順序でそれらのプロセスを実行する方法。

これらはすべて同じ問題である

競合状態

- 競合状態 (Race Conditions) とは?

  複数のプロセスが、同一の共有データへの読み書きを行う状況に置いて、最終状態が最後にどのプロセスがこのデータへの書き込みを行ったかに依存する状態。

競合を起こすプロセス

- 競合状態 (Race Conditions) とは
  - 複数のプロセスが、同一の共有データへの読み書きを行う状況において、最終状態が最後にどのプロセスがこのデータへの書き込みを行ったかに依存する状態。

```
spool_file ( file )
{
        next_free_slot = read_int ( in );
        register_file ( next_free_slot, name );
        next_free_slot ++;
        write_int ( in, next_free_slot );
}
```

プロセス A & B が実行する手続き

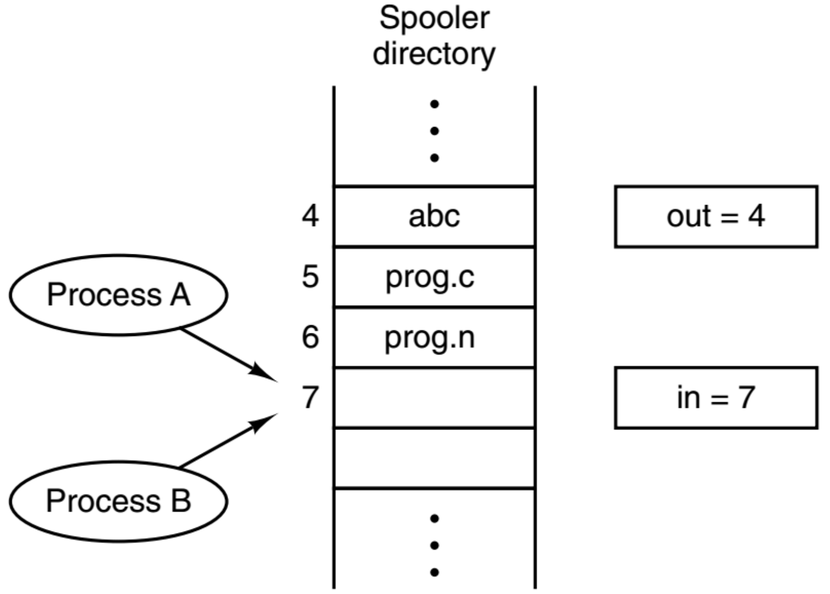
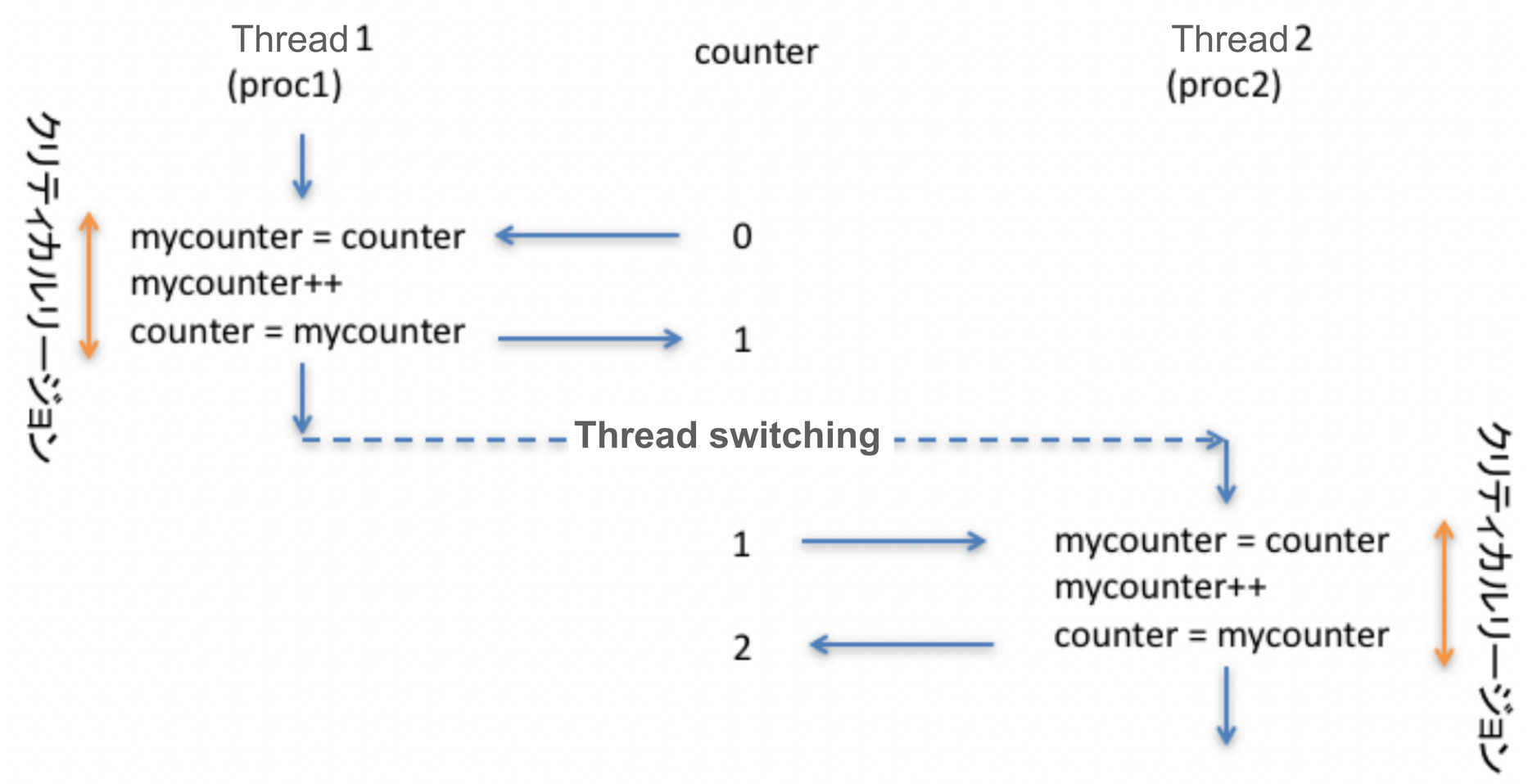


Figure 2-21. Two processes want to access shared memory at the same time.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.120.

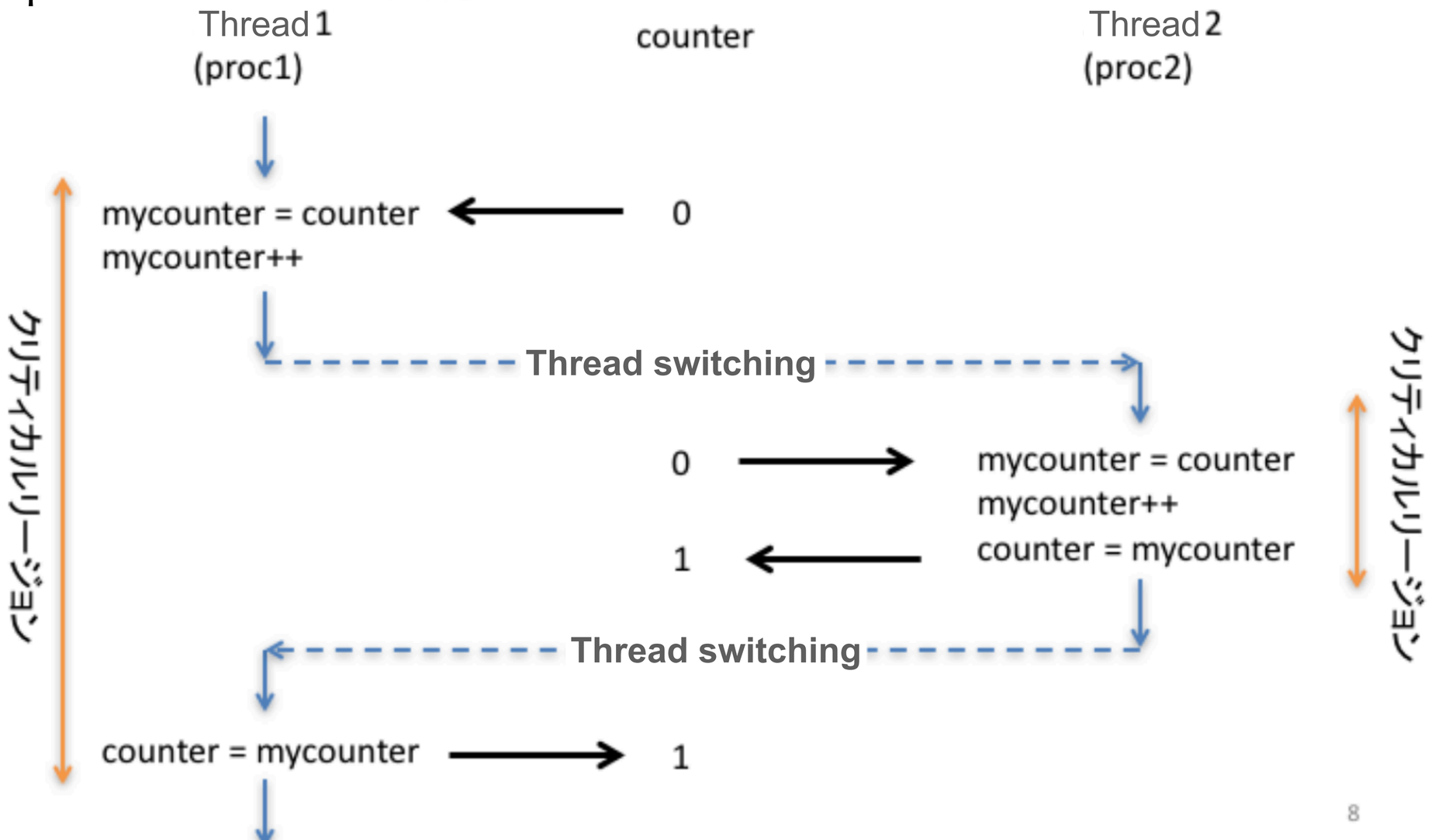
スレッドによる例題: Lost update (1)

- Lost update が起きない実行



スレッドによる例題: Lost update (2)

- Lost update が起きる実行



クリティカルリージョン

- 競合状態を避けるためには、クリティカルリージョンへの他プロセスの新入を防止すること(相互排除)が必要である。

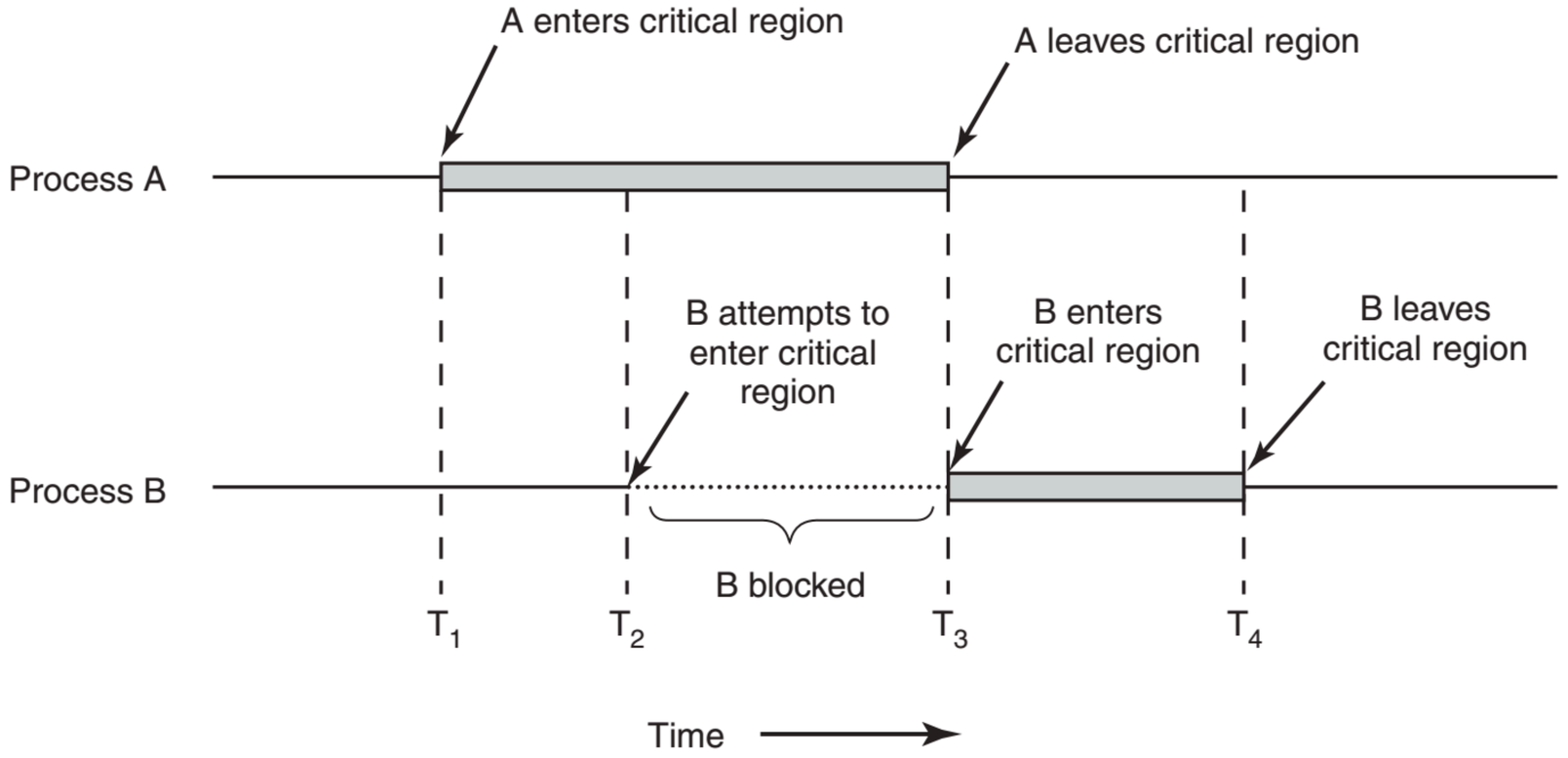


Figure 2-22. Mutual exclusion using critical regions.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.122.

いくつかの好ましくない解法 — 割り込みの不許可(禁止)

プロセスがクリティカルリージョンに進入するタイミングで割り込みを不許可(禁止)とし、プロセス(あるいはスレッド)の切り替えが起きないようにする方法。

- 不利な点
    - ユーザに割り込みの制御を行う権限を与えなければならない。
    - マルチプロセッサシステムでは、割り込み禁止命令を実行したプロセッサでのみ割り込みが禁止される。
    - カーネルが自身の内部で、ごく短時間だけ割り込み禁止にすることはしばしば行われる。

いくつかの好ましくない解法 — ロック変数の利用

クリティカルリージョンへの侵入の可否を表すロック変数による相互排除を試みる。

- 不利な点
  - “`while ( lock == 1 ); lock = 1;`” と “`lock = 0;`” は、ロック変数 lock に関する別のクリティカルリージョンを形成する。
  - ロック検査が busy wait で行われる。

```
while ( lock == 1 );     // wait until it unlocks.
lock = 1;                // lock it

critical_region();       // execute critical region.

lock = 0;                // unlock it

non_critical_region();
```

いくつかの好ましくない解法 — 完全な交互(あるいはラウンドロビン)実行

次にどのプロセスがクリティカルリージョンに進入できるかを示す制御変数を用いる。

プロセスは、制御変数の値が自身の番号と一致するまで busy wait (連続して制御変数の値を検査) する。

- 不利な点
  - 交互(あるいはラウンドロビン)が要求されること。
  - プロセスの実行時間に差がある場合の問題。

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Figure 2-23. A proposed solution to the critical-region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.124.

ソフトウェアによる相互排除 (Petersonの方法)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Source: Andrew S. Tanenbaum. "Modern Operating Systems", 4th ed., global ed., pp.125.

ソフトウェアによる解法のまとめ

- ソフトウェアによる相互排除は容易ではない。
  - 割り込み禁止 (決して好ましいものではない)
  - 割り込みを禁止しない場合は、必ず busy wait を伴う。
  - ロック変数 (正しい解法ではない)
  - 交互実行 (いくつかの欠点がある)
  - Peterson の方法 (non-trivial なコード & busy wait)

RMW 命令

- 相互排除問題に一般的な解を与えるためには、共有変数の検査と書き込み(ロック)を「不可分(atomic)」に実行するハードウェア機構が必要である。
- RMW (read-modify-write) 属性を持つ機械命令は、この用途に用いることができる。
- RMW 命令の例:
  - `TSL rx, lock`：メモリ番地 lock の内容をレジスタ rx に読み出し、続いて 1 を lock に書き込む(教科書で用いている命令)。
- 他にも使える命令がある:
  - 例： `XCHG rx, lock`：メモリ番地 lock の内容とレジスタ rx の内容を入れ替える。状態レジスタは rx の結果を反映して更新される。
- 命令によっては、マルチプロセッサ環境下でも正しく動作するものがある。X86 では、LOCK プリフィックスで RMW 命令を修飾することで、これが可能になる。

TSL 命令によるロック (busy wait 版)

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was not zero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.127.

sleep と wakeup の必要性

- Busy wait の問題
  - CPU の浪費
  - 優先度逆転問題
    - プロセスに実行優先度がある場合、優先度が高いプロセスが busy wait を始めてしまうと、より優先度が低いプロセスによるロック解除が行えなくなり、busy wait が終了しなくなる。
- “sleep” と “wakeup” の導入
  - クリティカルリージョンへの進入が出来ない場合、そのプロセスは、即座に自身の実行を放棄する (他のプロセスに実行権を譲る) = “sleep”。
  - クリティカルリージョンを脱出するプロセスは、sleep しているプロセスに進入可能になったことを知らせる信号を送る = “wakeup”。

ところが... lost wakeup 問題

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Figure 2-27. The producer-consumer problem with a fatal race condition.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.129.

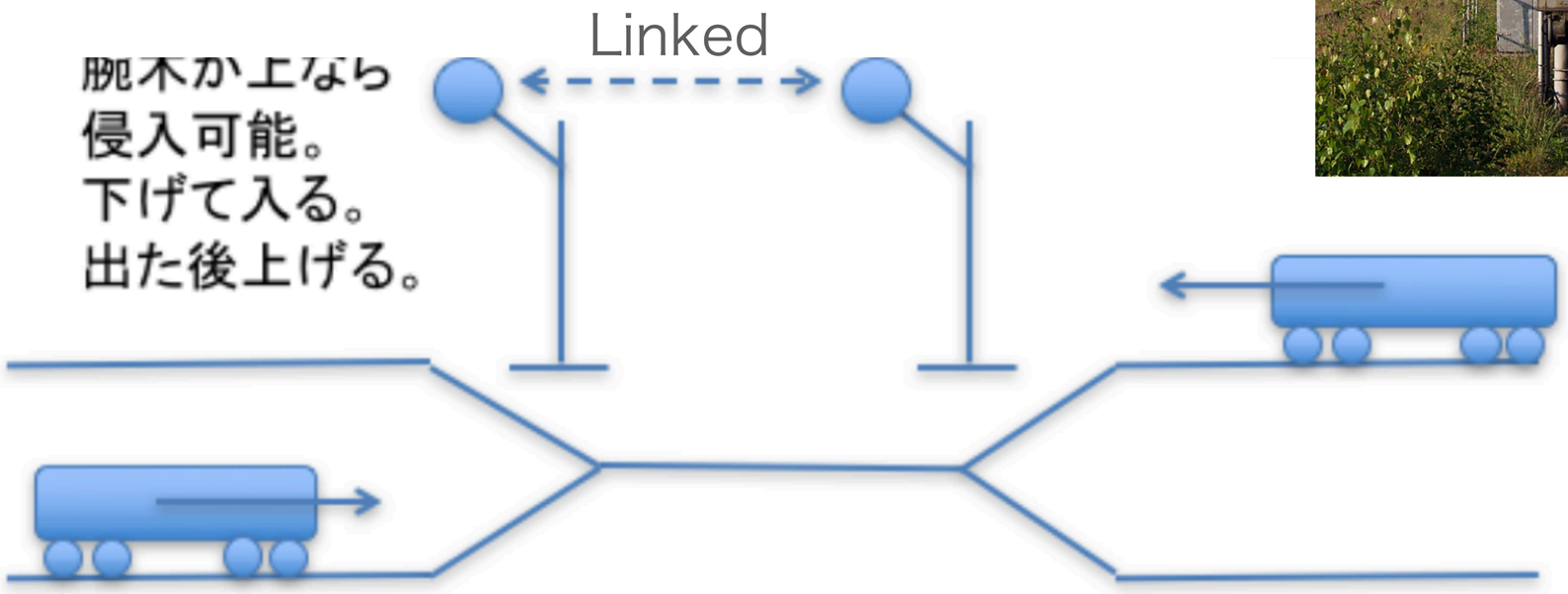
lost wakeup 問題の本質

- まだ sleep していないプロセスに送られた wakeup が失われてしまうこと。
- sleep していないプロセスに対する wakeup を保持しておくための変数を新たに増設し、sleep しようとするプロセスは、この変数を検査し、すでに wakeup が送られていればこの変数をリセットするだけで sleep せずに実行を続けるようにする。

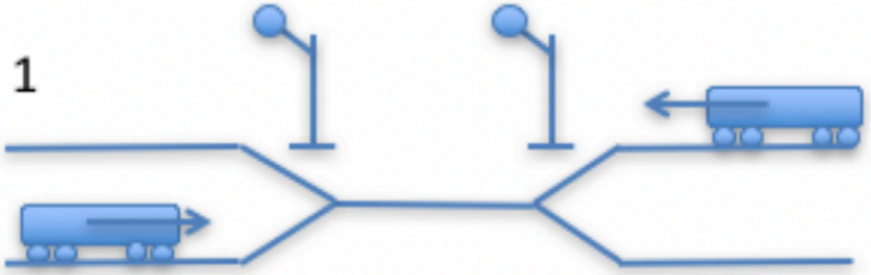
さまざまな同期機構

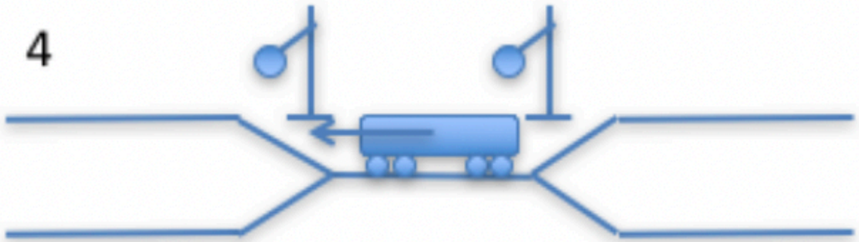
セマフォ

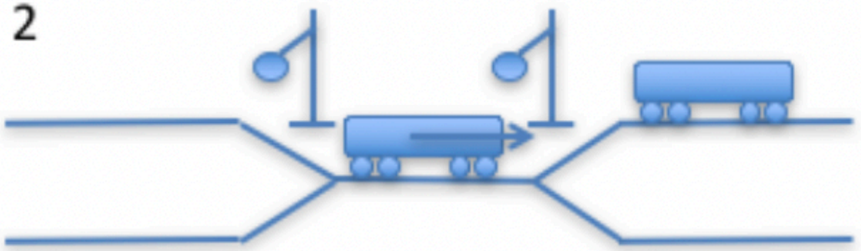
- 単線区間への進入を制御している
  「腕木式信号機」のアナロジー

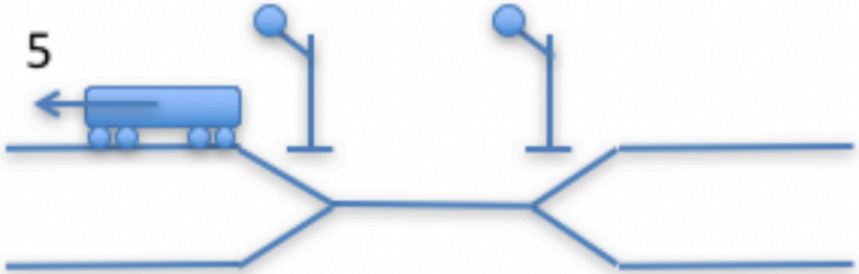


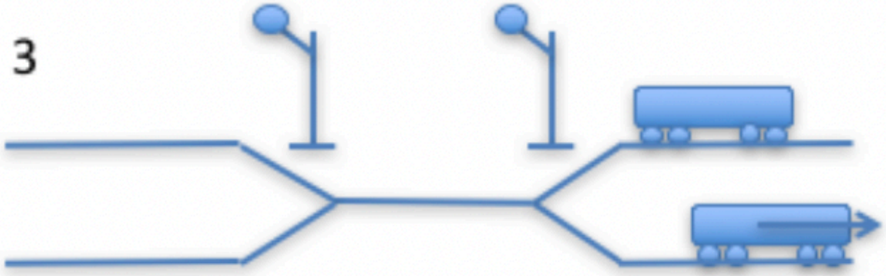
セマフォ (2)











セマフォ (3)

- セマフォの操作
  - down(sem)
    1. sem がすでに下がっていた場合には、上がるまで待機し、再度 1 へ。
    2. セマフォ sem を下げる
  - up(sem)
    1. セマフォ sem を上げる。
- 古典: down 操作は “P” 操作、up 操作は “V” 操作と呼ばれていた。
  (“P”rolagen = 小さくする、“V”erhogen = 大きくする)
- 腕木の状態の代わりに利用可能な資源の個数を表す値を持たせ、ゼロの場合は待機させるという「多値セマフォ」もある (v.s. 二値セマフォ = binary semaphore)

セマフォによる「生産者-消費者 問題」の解法

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

Figure 2-28. The producer-consumer problem using semaphores.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.131.

相互排除変数 — Mutex

- 相互排除専用のセマフォ

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Figure 2-29. Implementation of *mutex_lock* and *mutex_unlock*.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.133.

モニタ — 同期メカニズムの高級言語的隠蔽

- セマフォを用いれば、協調プロセスを正しく書くことが出来る。
- しかしながら、それは必ずしも容易ではない。

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .           .           .
        end;
end monitor;
```

Figure 2-33. A monitor.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.140.

モニタによる「生産者-消費者 問題」(モニタの記述)

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.141.

モニタによる「生産者-消費者 問題」(Java)

```
public class ProducerConsumer {
    static final int N = 100;        // constant giving the buffer size
    static producer p = new producer( );     // instantiate a new producer thread
    static consumer c = new consumer( ); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor( );       // instantiate a new monitor

    public static void main(String args[ ]) {
        p.start( );       // start the producer thread
        c.start( );       // start the consumer thread
    }

    static class producer extends Thread {
        public void run( )  { // run method contains the thread code
            int item;
            while (true) {      // producer loop
                item = produce_item( );
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... }       // actually produce
    }

    static class consumer extends Thread {
        public void run( )  { run method contains the thread code
            int item;
            while (true) {      // consumer loop
                item = mon.remove( );
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... }// actually consume
    }

    static class our_monitor {   // this is a monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0;   // counters and indices

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
            buffer [hi] = val;  // insert an item into the buffer
            hi = (hi + 1) % N;        // slot to place next item in
            count = count + 1;       // one more item in the buffer now
            if (count == 1) notify( );        // if consumer was sleeping, wake it up
        }

        public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
            val = buffer [lo];  // fetch an item from the buffer
            lo = (lo + 1) % N;        // slot to fetch next item from
            count = count – 1;       // one few items in the buffer
            if (count == N – 1) notify( );  // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

Figure 2-35. A solution to the producer-consumer problem in Java.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.142.

メッセージ通信

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}


void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Figure 2-36. The producer-consumer problem with N messages.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.146.

バリア同期

- 複数のプロセスによる、フェーズを意識した計算を行っている場合に必要な同期機構。

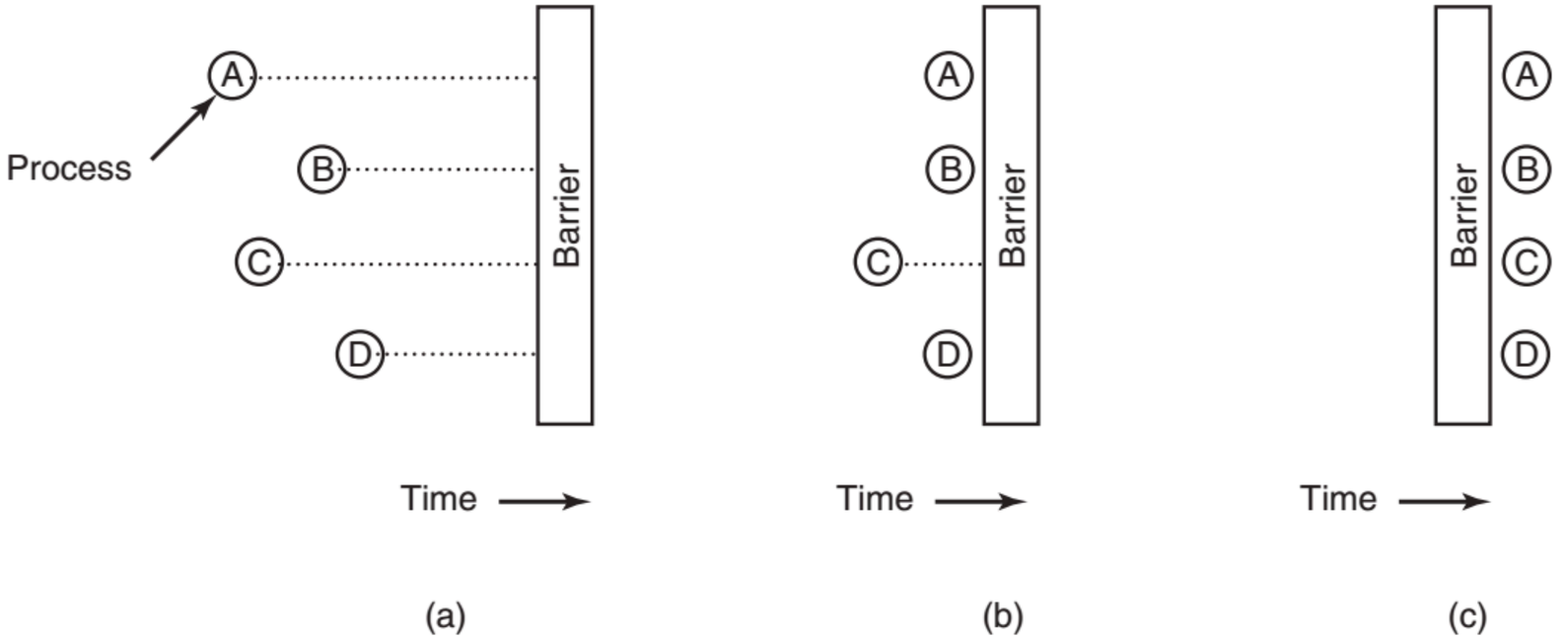


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.147.

同期プリミティブの能力

【問題】さまざまな同期プリミティブは、その記述能力(同期の問題を解く能力)に高低・優劣があるのだろうか?

- 記述方法 a と b があった場合、問題 p に対して a では記述できるが、b では記述できない場合、a の記述能力は b よりも高い (これを “$p|a > b$” と書こう)
- すべての問題 $p \in P$ (P は全ての同期問題の集合) を記述してみなければ、$^{\forall}p \in P|a > b$ が言えない?

同期プリミティブの能力 (2)

- a で b が記述できる場合、$^{\forall}p \in P | a \geq b$ である。
  - b で記述できる問題は、a で b を記述した b を用いれば、全て a でも記述できる。
  - b で記述できない問題で、a で記述できるものがある。
  - b で記述できない問題で、a でも記述できないものがある。
- b で a が記述できる場合、$^{\forall}p \in P | a \leq b$ である。
- 上記の 2 つが同時に成り立てば、$^{\forall}p \in P | a = b$ である。
  - 全ての問題に対して能力が等しいだけで、記述できるかどうかは別。

同期プリミティブの能力 (3)

- 複数の同期プリミティブの能力がすべて等しいことを証明するには?
  - メッセージ通信による mutex の記述?
  - mutex によるメッセージ通信の記述?
  - セマフォによる mutex の記述?
  - mutex によるセマフォの記述?
  - …

哲学者の食事問題 (The Dining Philosophers Problem)

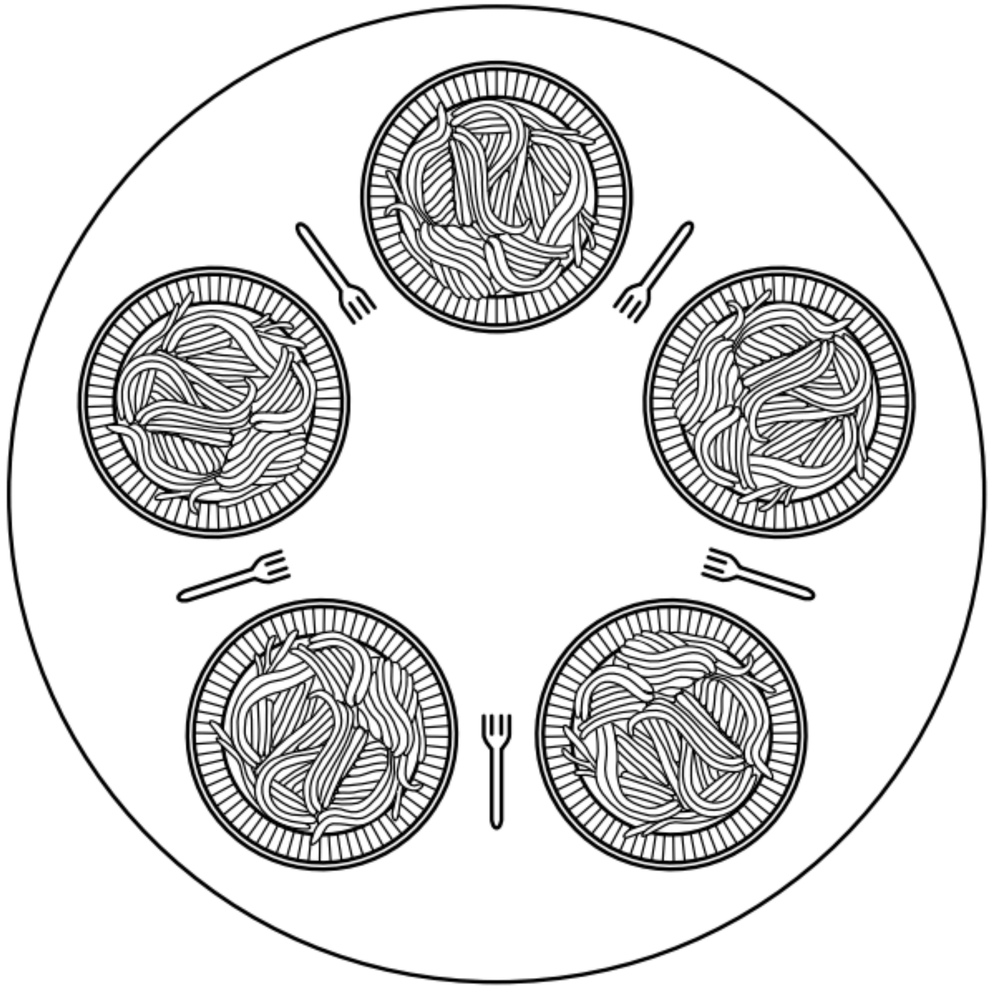


Figure 2-45. Lunch time in the Philosophy Department.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.168.

凍り付くあるいは餓死する哲学者

- 同期に関するもう一つの問題: 餓死状態 (starvation)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

Figure 2-46. A nonsolution to the dining philosophers problem.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.169.

正しい解法の一つ

```
#define N                   5                       /* number of philosophers */
#define LEFT                (i+N–1)%N               /* number of i's left neighbor */
#define RIGHT               (i+1)%N                 /* number of i's right neighbor */
#define THINKING            0                       /* philosopher is thinking */
#define HUNGRY              1                       /* philosopher is trying to get forks */
#define EATING              2                       /* philosopher is eating */

typedef int semaphore;                              /* semaphores are a special kind of int */
int state[N];                                       /* array to keep track of everyone's state */
semaphore mutex = 1;                                /* mutual exclusion for critical regions */
semaphore s[N];                                     /* one semaphore per philosopher */


void philosopher(int i)                             /* i: philosopher number, from 0 to N–1 */
{
      while (TRUE) {                                /* repeat forever */
            think( );                               /* philosopher is thinking */
            take_forks(i);                          /* acquire two forks or block */
            eat( );                                 /* yum-yum, spaghetti */
            put_forks(i);                           /* put both forks back on table */
      }
}

void take_forks(int i)                              /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                                 /* enter critical region */
      state[i] = HUNGRY;                            /* record fact that philosopher i is hungry */
      test(i);                                      /* try to acquire 2 forks */
      up(&mutex);                                   /* exit critical region */
      down(&s[i]);                                  /* block if forks were not acquired */
}
```

```
void take_forks(int i)                              /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                                 /* enter critical region */
      state[i] = HUNGRY;                            /* record fact that philosopher i is hungry */
      test(i);                                      /* try to acquire 2 forks */
      up(&mutex);                                   /* exit critical region */
      down(&s[i]);                                  /* block if forks were not acquired */
}

void put_forks(i)                                   /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                                 /* enter critical region */
      state[i] = THINKING;                          /* philosopher has finished eating */
      test(LEFT);                                   /* see if left neighbor can now eat */
      test(RIGHT);                                  /* see if right neighbor can now eat */
      up(&mutex);                                   /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Figure 2-47. A solution to the dining philosophers problem.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.170.

Readers - Writers 問題

- データベースの読み書きを行っている複数のプロセス。
- 複数のプロセスが同時に読み出しアクセスを行うことは問題がない。
- 書き込みプロセスがアクセス中には、他の全てのプロセスはデータベースをアクセスしてはならない。
- 次々と到着するプロセスのせいで、書き込みプロセスが餓死状態になってはならない。

Readers - Writers 問題の解法の一つ

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

Figure 2-48. A solution to the readers and writers problem.

Source: Andrew S. Tanenbaum. “Modern Operating Systems”, 4th ed., global ed., pp.171.