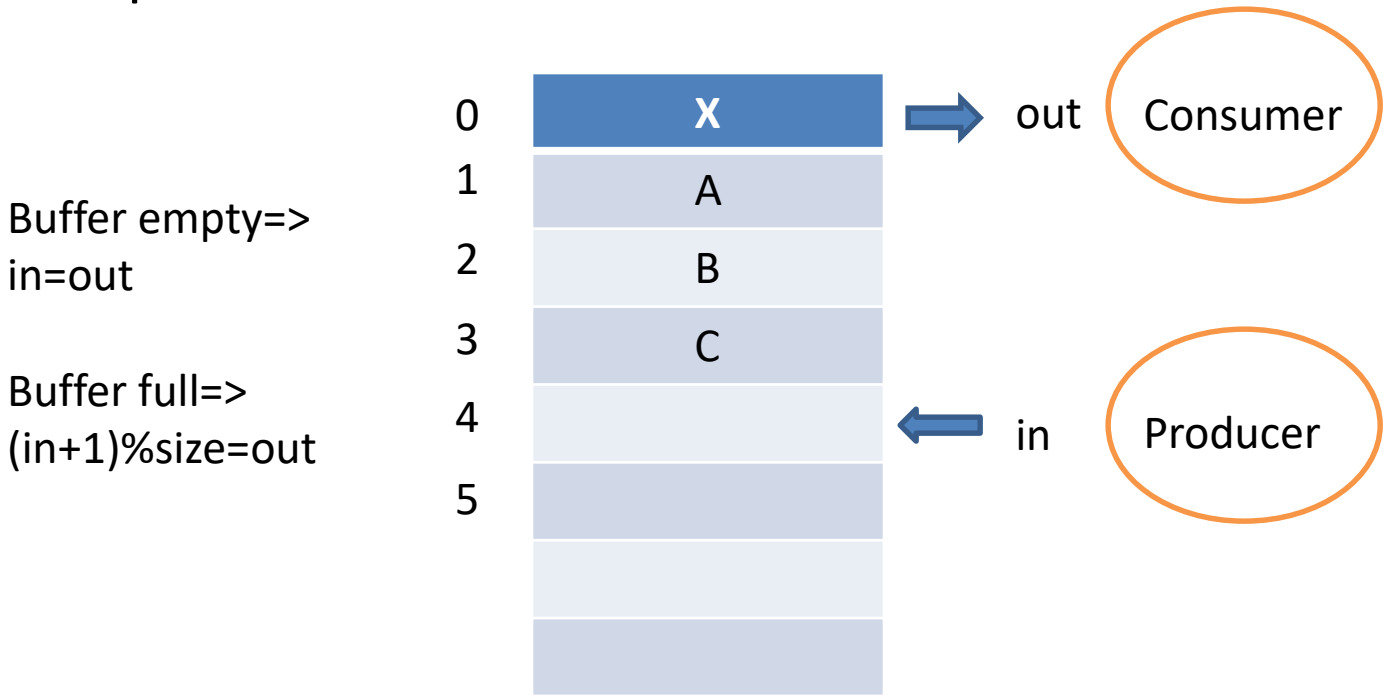
Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data

- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process



- *unbounded-buffer* places no practical limit on the size of the buffer
- *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

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

Bounded-Buffer – Producer

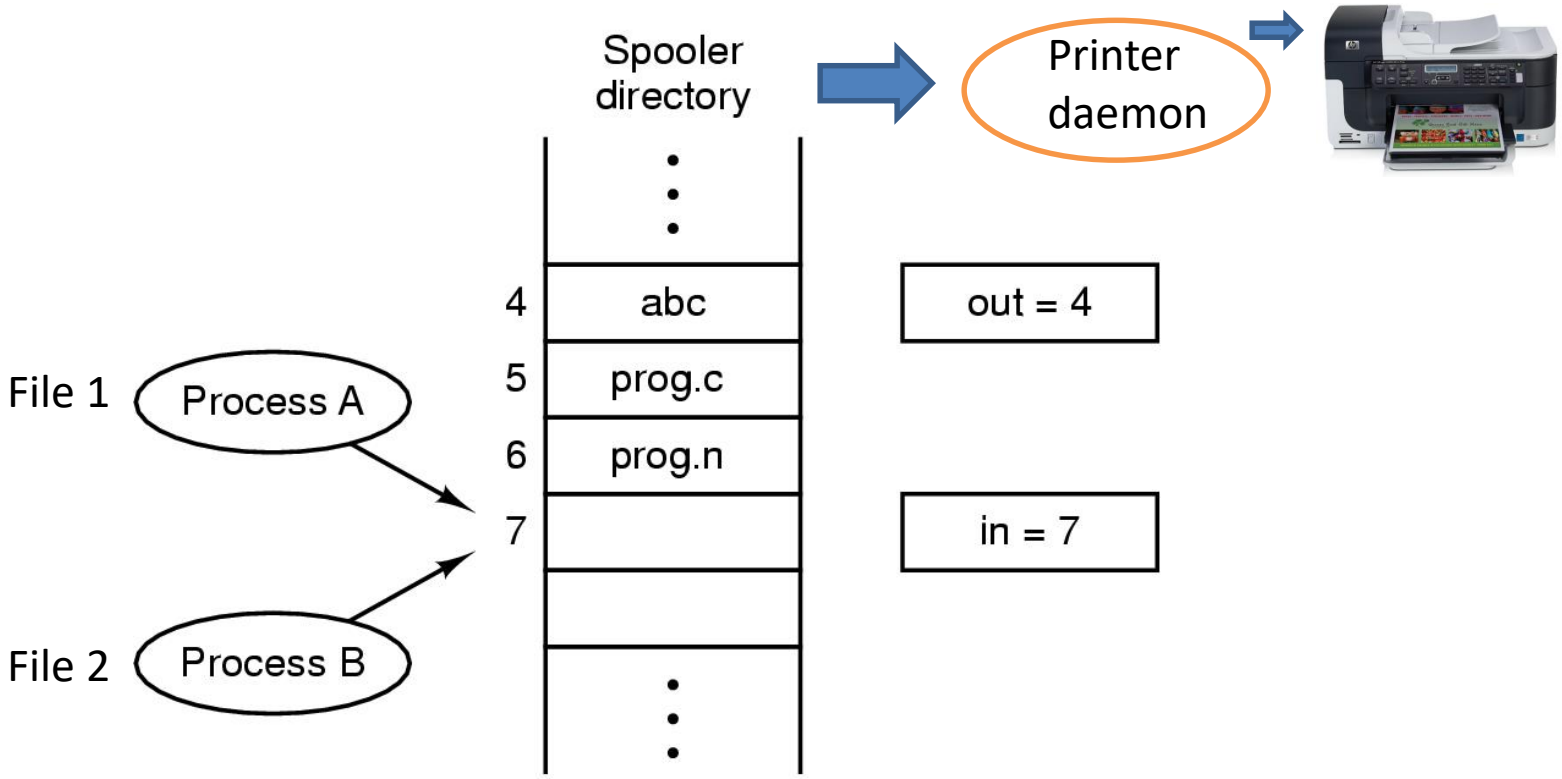
```
while (true) {
   /* Produce an item */
   while (((in + 1) % BUFFER SIZE)  == out)
      ;   /* do nothing -- no free buffers */
     buffer[in] = item;
     in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Consumer

```
while (true) {
    while (in == out)
        ; // do nothing -- nothing to consume

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
return item;
}
```

Interprocess Communication

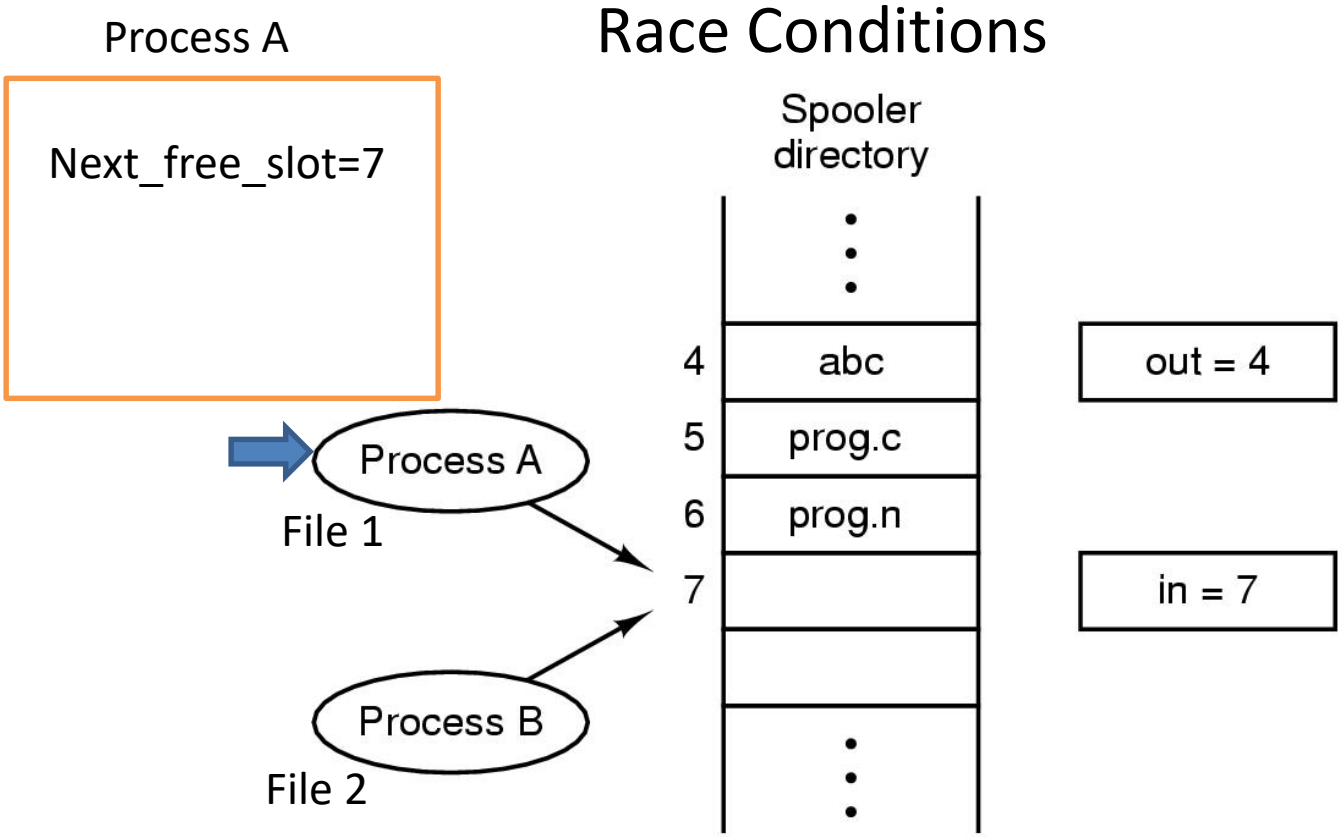
Race Conditions



Two processes want to access shared memory at same time

Interprocess Communication

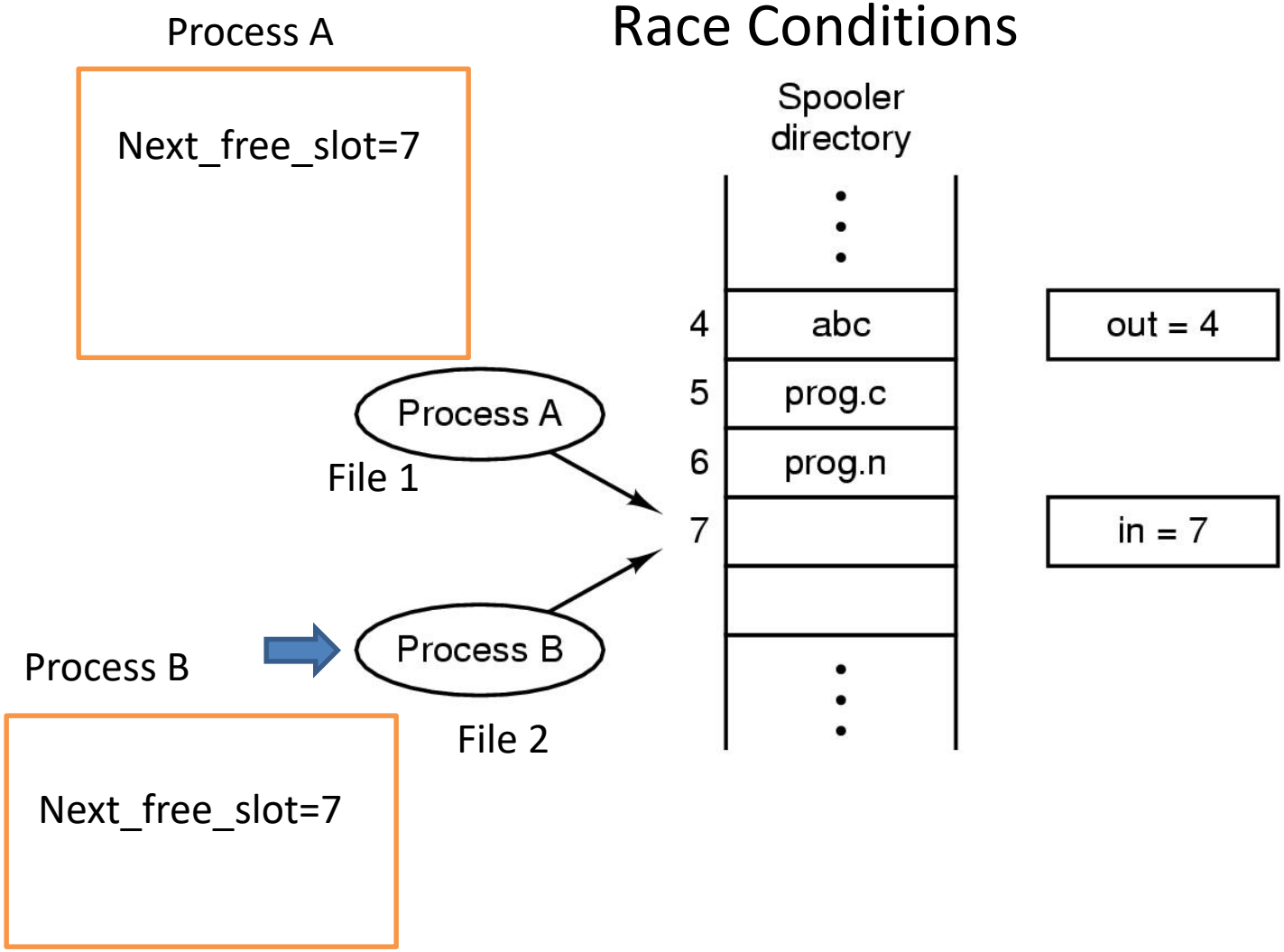
Race Conditions



Two processes want to access shared memory at same time

Interprocess Communication

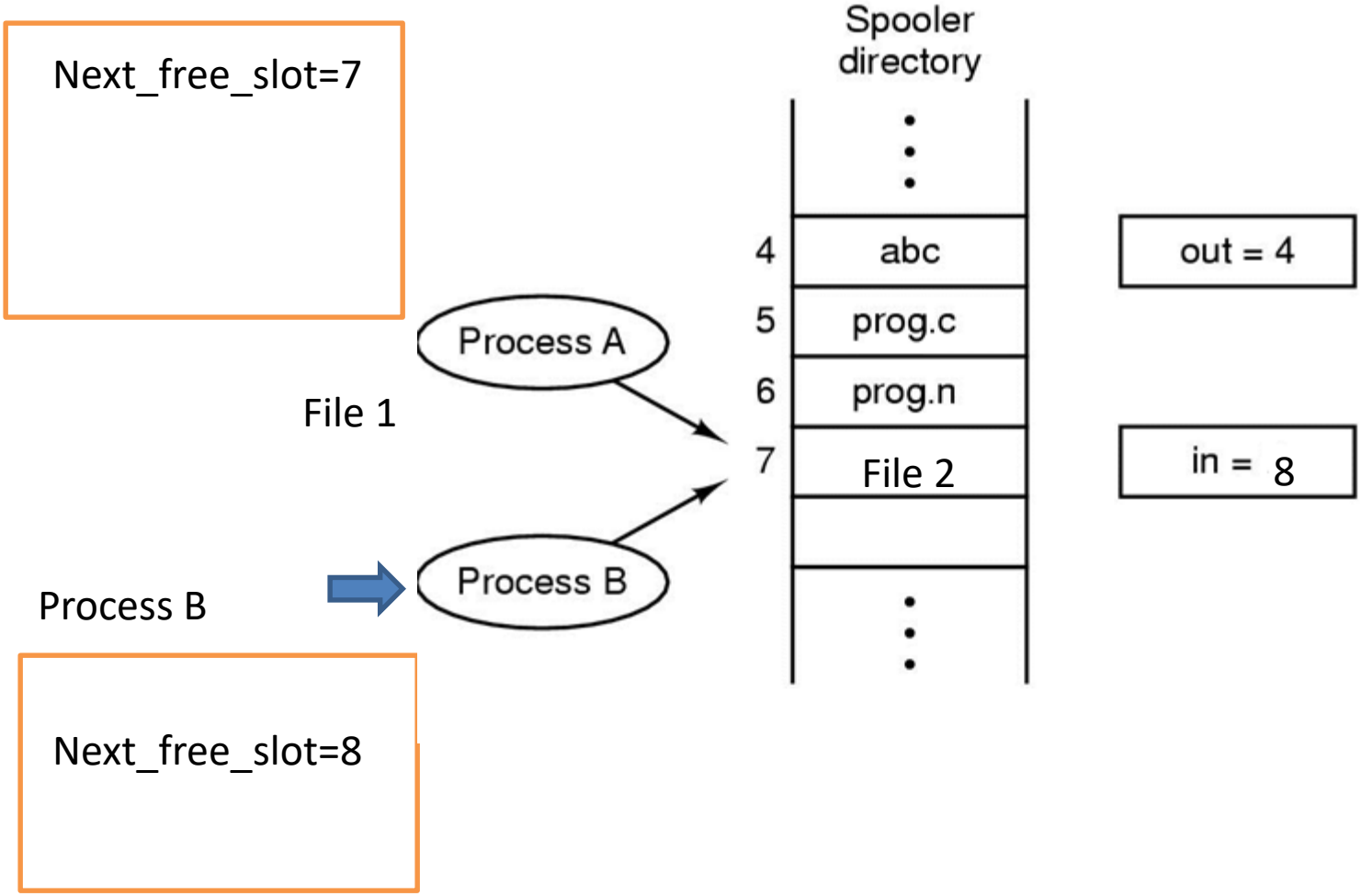
Race Conditions



Two processes want to access shared memory at same time

Interprocess Communication

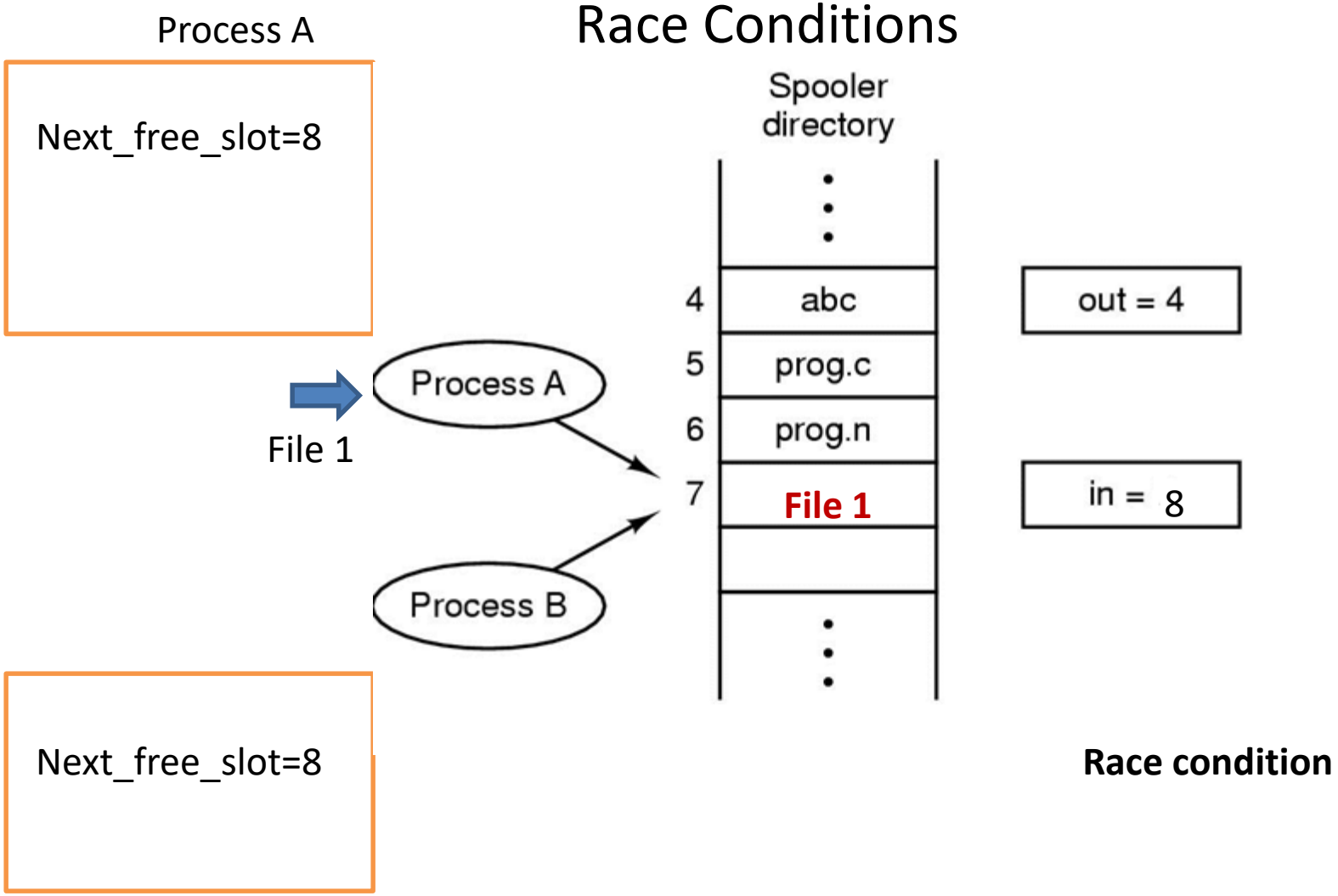
Race Conditions



Two processes want to access shared memory at same time

Interprocess Communication

Race Conditions



Two processes want to access shared memory at same time

Race condition

- Race condition
  - Two or more processes are reading or writing some shared data and the final result depends on who runs precisely when
  - In our former example, the possibilities are various
  - Hard to debug

Critical Section Problem

- Critical region
  - Part of the program where the shared memory is accessed
- Mutual exclusion
  - Prohibit more than one process from reading and writing the shared data at the same time

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section Problem

do {

entry section

critical section

exit section

remainder section

} while (TRUE);

General structure of a typical process P_i

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** –

- If no process is executing in its critical section
- and there exist some processes that wish to enter their critical section
- then only the processes outside remainder section (i.e. the processes competing for critical section, or exit section) can participate in deciding which process will enter CS next

3. **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
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

Critical Section Problem

do {

entry section

critical section

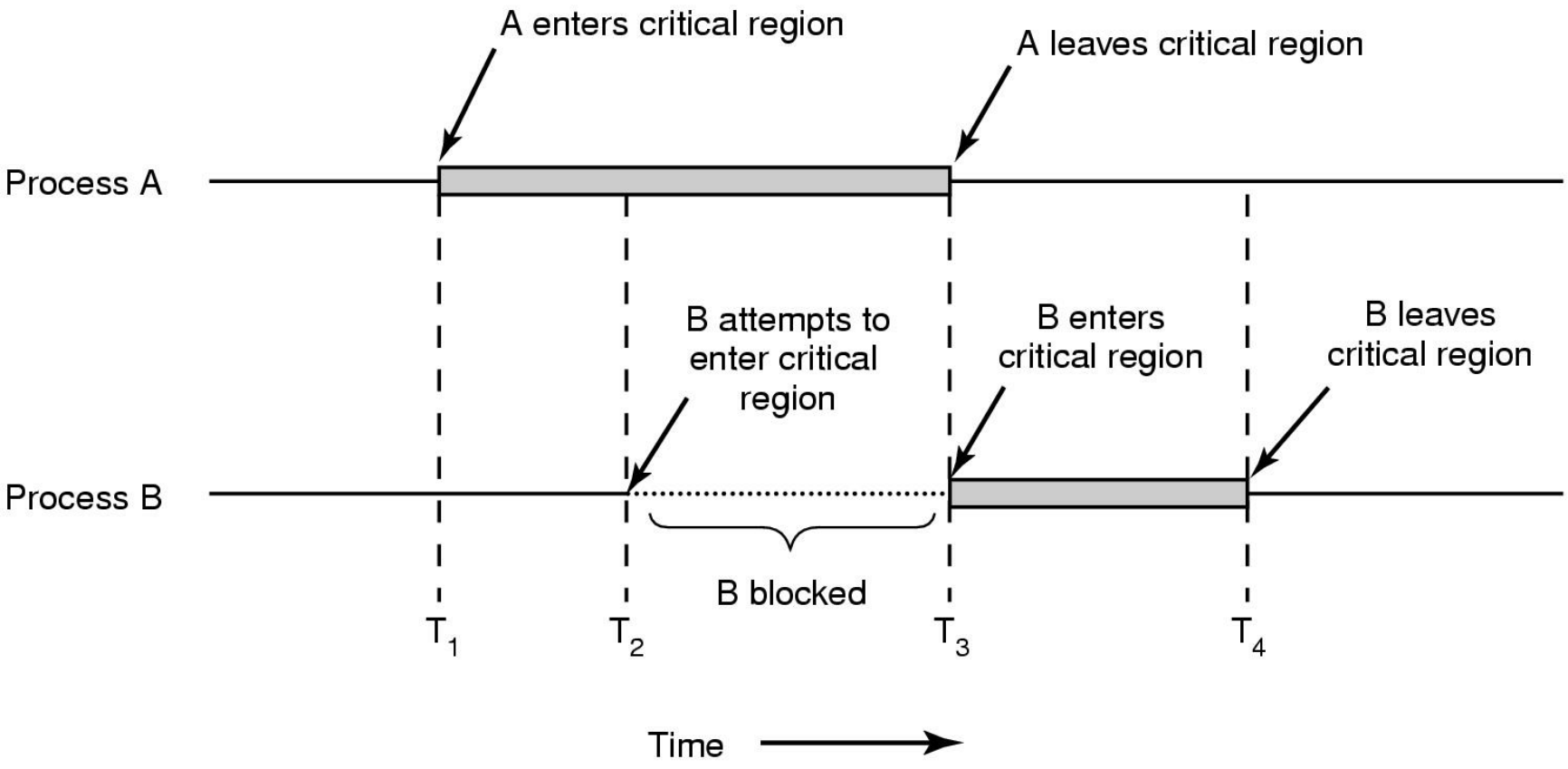
exit section

remainder section

} while (TRUE);

General structure of a typical process P_i

Critical Section Problem



Mutual exclusion using critical regions

Mutual Exclusion

- Disable interrupt
  - After entering critical region, disable all interrupts
  - Since clock is just an interrupt, no CPU preemption can occur
  - Disabling interrupt is useful for OS itself, but not for users…

Mutual Exclusion with busy waiting

- Lock variable
  - A software solution
  - A single, shared variable (lock)
  - before entering critical region, programs test the variable,
  - if 0, enter CS;
  - if 1, the critical region is occupied
  - **What is the problem?**

```
While(true)
{
        while(lock!=0);
        Lock=1
        CS()
        Lock=0
        Non-CS()
}
```

Concepts

- Busy waiting
  - Continuously testing a variable until some value appears
- Spin lock
  - A lock using busy waiting is call a spin lock
- CPU time wastage!

Mutual Exclusion with Busy Waiting : strict alternation

```
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)

Proposed solution to critical region problem

(a) Process 0. (b) Process 1.

1.Mutual exclusion is preserved? **Y**
2.Progress requirement is satisfied? **N**
3.Bounded-waiting requirement is met? **N**

Peterson's Solution

- Two process solution

- The two processes share two variables:
  - int **turn**;
  - Boolean **interested [2]**

- The variable **turn** indicates whose turn it is to enter the critical section

- The **interested** array is used to indicate if a process is interested to enter the critical section.
- **interested[i]** = true implies that process $\mathbf{P_i}$ is interested!

Mutual Exclusion with Busy Waiting (2) : a workable method

```
#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 */
}
```

Peterson's solution for achieving mutual exclusion

Algorithm for Process P_i

```
do {
        interested[i] = TRUE;
       turn = j ;
       while (interested[j] && turn == j);
                critical section
        interested[i] = FALSE;
                remainder section
} while (TRUE);
```

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

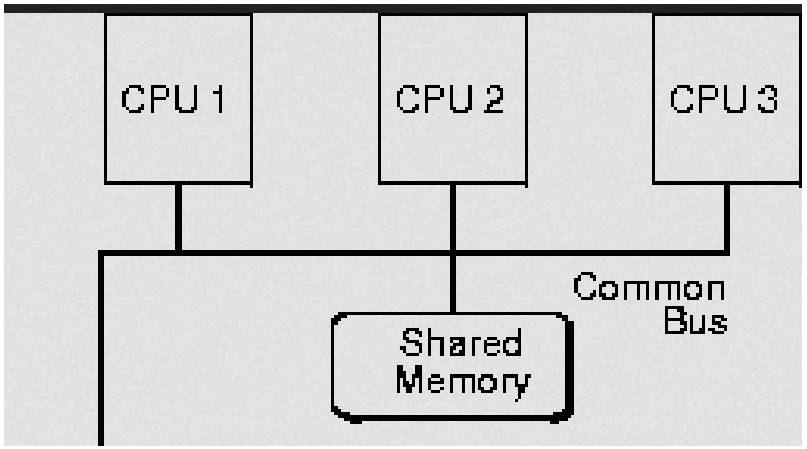
Does this alter the sequence?

Hardware Instruction Based Solutions
Multiprocessor system

- Some architectures provide special instructions that can be used for synchronization

- TSL: Test and modify the content of a word atomically

```
        TSL Reg, lock
{
            Reg= lock;
            lock = true;
}
```



Hardware Instruction Based Solutions

```
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 non 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
```

Does it satisfy all the conditions?

Entering and leaving a critical region using the TSL instruction

System call version

- Special system call that can be used for synchronization

- TestAndSet: Test and modify the content of a word atomically

```
boolean TestAndSet (boolean &target) {
    boolean v = target;
    target = true;
    return v;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

  Does it satisfy all the conditions?

- Process P_i

```
do {
   while (TestAndSet(lock)) ;
      critical section
   lock = false;
      remainder section
}
```

Swap Instruction

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Concepts

- Busy waiting
  - Continuously testing a variable until some value appears
- Spin lock
  - A lock using busy waiting is call a spin lock
- CPU time wastage!

- Drawback of Busy waiting
  - A lower priority process has entered critical region
  - A higher priority process comes and preempts the lower priority process, it wastes CPU in busy waiting, while the lower priority don’t come out
  - Priority inversion problem

Producer-consumer problem

- Two processes share a common, fixed-sized buffer
- Producer puts information into the buffer
- Consumer takes information from buffer
- A simple solution

Sleep and Wakeup

```
#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 */
     }
}
```

Producer-Consumer 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 */
     }
}
```

- What can be the problem?
- Signal missing
  - Shared variable: counter
  - When consumer read count with a 0 but didn't fall asleep in time
  - then the signal will be lost

Producer-consumer problem with fatal race condition

Tasks

- We must ensure proper process synchronization
  - Stop the producer when buffer full
  - Stop the consumer when buffer empty
- We must ensure mutual exclusion
  - Avoid race condition
- Avoid busy waiting

Semaphore

- Widely used synchronization tool
- Does not require busy-waiting
  - CPU is not held unnecessarily while the process is waiting
- A Semaphore *S* is
  - A data structure with an integer variable *S.value* and a queue *S.list* of processes (shared variable)
  - The data structure can only be accessed by two atomic operations, *wait(S)* and *signal(S)* (also called *down(S), P(S)* and *Up(s), V(S)*)

- Value of the semaphore S = value of the integer *S.value*

```
typedef struct {
        int value;
        struct process *list;
        } semaphore
```

Semaphore

Wait(S) S<= semaphore variable

- When a process P executes the wait(S) and finds
- S==0
  - Process must wait => block()
  - Places the process into a waiting queue associated with S
  - Switch from running to waiting state
- Otherwise decrement S

Signal(S)

When a process P executes the signal(S)

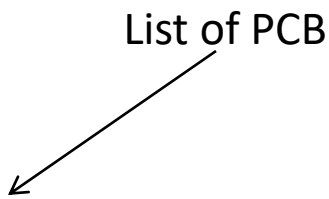
- Check, if some other process Q is waiting on the semaphore S
- Wakeup(Q)
- Wakeup(Q) changes the process from waiting to ready state

- Otherwise increment S

Semaphore (wait and signal)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```



Atomic/
Indivisible

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Note: which process is picked for unblocking may depend on policy.

Usage of Semaphore

- **Counting** semaphore – integer value can range over an unrestricted domain
  - Control access to a shared resource with finite elements
  - Wish to use => wait(S)
  - Releases resource=>signal(S)
  - Used for synchronization
- **Binary** semaphore – integer value can range only between 0 and 1
  - Also known as **mutex locks**
  - Used for mutual exclusion

Ordering Execution of Processes using Semaphores (Synchronization)

- Execute statement *B* in P_j only after statement *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
Stmt. *A*	*wait*(*flag*)
signal(*flag*)	Stmt. *B*

- Multiple such points of synchronization can be enforced using one or more semaphores

Semaphore: Mutual exclusion

- Shared data:
  semaphore mutex; / initially mutex = 1 */*
- Process P_i:

```
do {
  wait(mutex);

     critical section

   signal(mutex);

        remainder section

 } while (1);
```

Producer-consumer problem : Semaphore

- Solve producer-consumer problem
  - Full: counting the slots that are full; initial value 0
  - Empty: counting the slots that are empty, initial value N
  - Mutex: prevent access the buffer at the same time, initial value 1 (**binary semaphore**)
  - Synchronization & mutual exclusion

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 */
    }
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Let S and Q be two semaphores initialized to 1

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Starvation** – indefinite blocking
  - LIFO queue
  - A process may never be removed from the semaphore queue in which it is suspended

Tutorial problems

Problem 1

```
flag[i] = true;
turn = i;
while ((flag[j] == true) && (turn == j)) { }
/* CRITICAL SECTION */
flag[i] = false;
/* Remainder section */
```

Does this solution work?

Problem 2

```
flag[i] = true;
turn = i;
while ((flag[j] == true) || (turn == j)) { }
/* CRITICAL SECTION */
flag[i] = false;
/* Remainder section */
```

Does this solution work?

Algorithm for Process P_i

```
do {
        interested[i] = TRUE;
       turn = j ;
       while (interested[j] && turn == j);
                critical section
        interested[i] = FALSE;
                remainder section
} while (TRUE);
```

Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Does this alter the sequence?

CASwap Instruction

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Problem 3: Atomic increment using CAS

Implement atomic increment using CAS

```
increment(&sequence);
```

Atomic increment using CAS

```
void increment(atomic_int *v)
{
   int temp;

   do {
      temp = *v;
   }
   while (temp != compare_and_swap(v, temp, temp+1));
}
```

Problem 4: Mutex Lock

```
while (true) {
     acquire lock
          critical section
     release lock
          remainder section
}
```

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
}
```

```
release() {
   available = true;
}
```

```
available = 1
acquire()
{ while(CAS(available, 1, 0)==0);
}
```

Problem 5: Does TSL and CAS Satisfies all properties of critical section solution?

```
do {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section
}
```

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

System call version

- Special system call that can be used for synchronization

- TestAndSet: Test and modify the content of a word atomically

```
boolean TestAndSet (boolean &target) {
    boolean v = target;
    target = true;
    return v;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

  Does it satisfy all the conditions?

- Process P_i

```
do {
   while (TestAndSet(lock)) ;
      critical section
   lock = false;
      remainder section
}
```

Swap Instruction

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

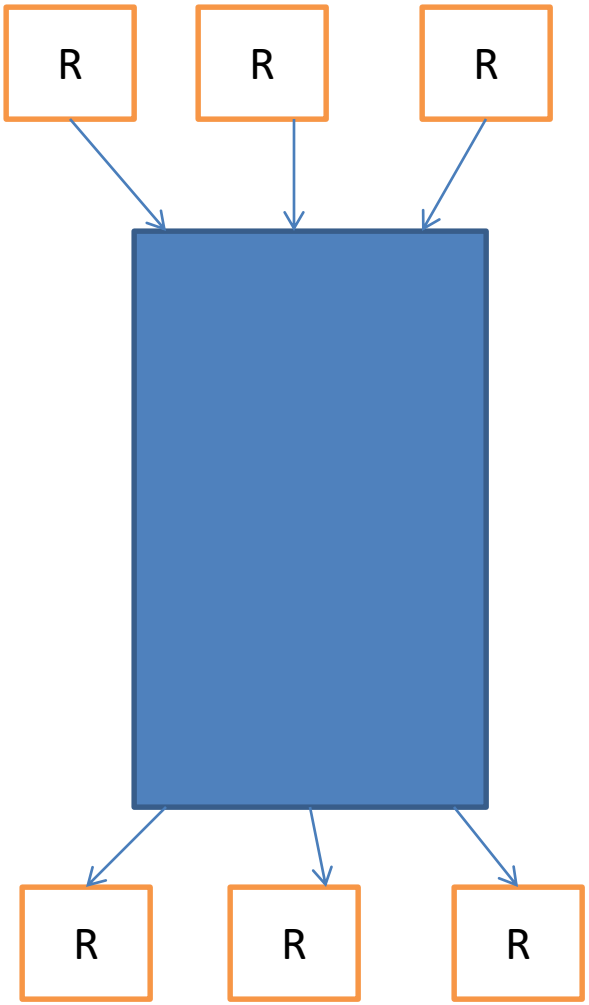
```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Readers-Writers Problem

- A database is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Database
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem

Writer



Writer

- Task of the writer
  - Just lock the dataset and write

Reader

- Task of the **first** reader
  - Lock the dataset
- Task of the **last** reader
  - Release the lock
  - Wakeup the any waiting writer

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex)

            // reading is performed

         wait (mutex) ;
         readcount  - - ;
         if (readcount  == 0)
                signal (wrt) ;
         signal (mutex) ;
    } while (TRUE);
```

Readers-Writers Problem (Cont.)

- Models database access
- Current solution
  - Reader gets priority over writer
- Home work
  - Writer gets priority