Principles of Operating Systems
CS 446/646

2. Processes

a. Process Description & Control

b. Threads

c. **Concurrency**

- ✓ Types of process interaction
- ✓ Race conditions & critical regions
- ✓ Mutual exclusion by busy waiting
- ✓ Mutual exclusion & synchronization
  - mutexes
  - semaphores
  - monitors
  - message passing

d. **Deadlocks**

2.c Concurrency

Types of process interaction

- Concurrency refers to any form of <u>interaction</u> among processes or threads
  - ✓ concurrency is a fundamental part of O/S design
  - ✓ concurrency includes
    - communication among processes/threads
    - sharing of, and competition for system resources
    - cooperative processing of shared data
    - synchronization of process/thread activities
    - organized CPU scheduling
    - solving deadlock and starvation problems

2.c Concurrency

Types of process interaction

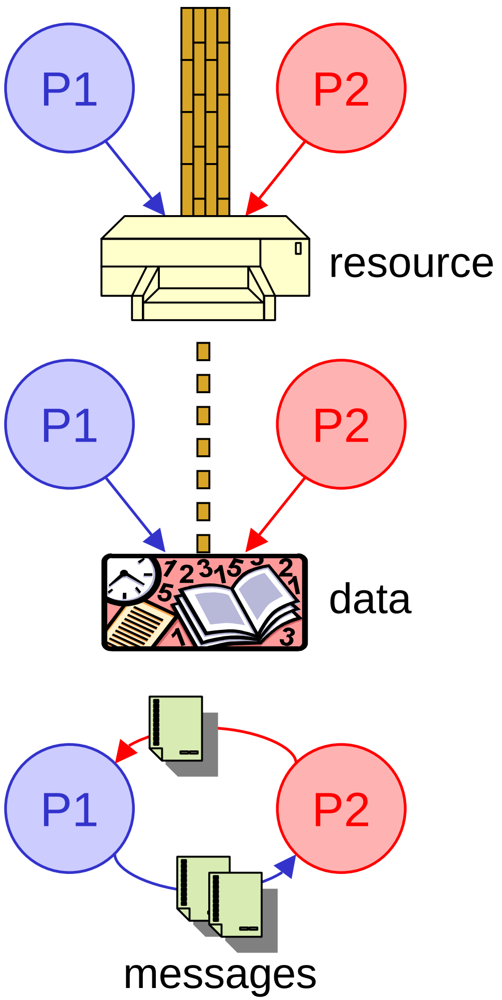
- Concurrency arises in the same way at different levels of execution streams
  - ✓ **multiprogramming** — interaction between multiple processes running on one CPU (pseudoparallelism)
  - ✓ **multithreading** — interaction between multiple threads running in one process
  - ✓ **multiprocessors** — interaction between multiple CPUs running multiple processes/threads (real parallelism)
  - ✓ **multicomputers** — interaction between multiple computers running a distributed processes/threads

  → *the principles of concurrency are basically the same in all of these categories (possible differences will be pointed out)*

2.c Concurrency

Types of process interaction

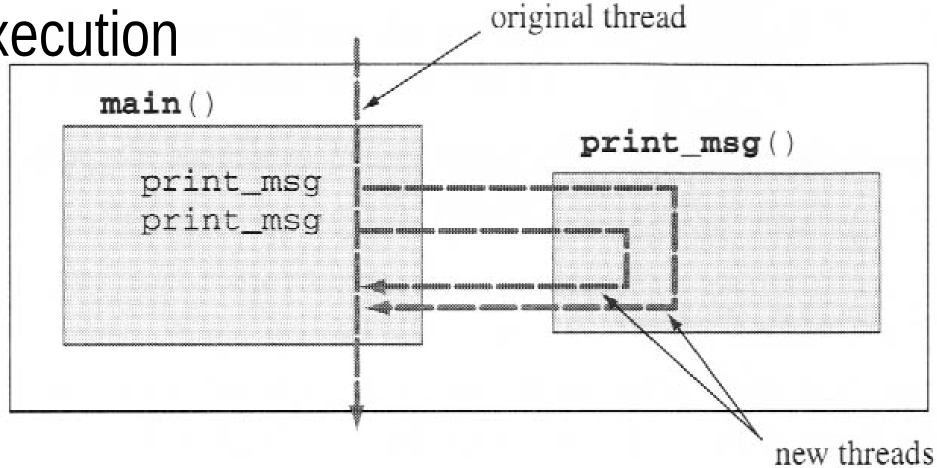
- Whether processes or threads: three basic interactions
  - ✓ **processes unaware of each other** — they must use shared resources independently, without interfering, and leave them intact for the others
  - ✓ **processes indirectly aware of each other** — they work on common data and build some result together via the data ("stigmergy" in biology)
  - ✓ **processes directly aware of each other** — they cooperate by communicating, e.g., exchanging messages



2.c Concurrency

Race conditions & critical regions

- **Inconsequential race condition in the shopping scenario**
  - ✓ there is a "race condition" if the outcome depends on the order of the execution



Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```

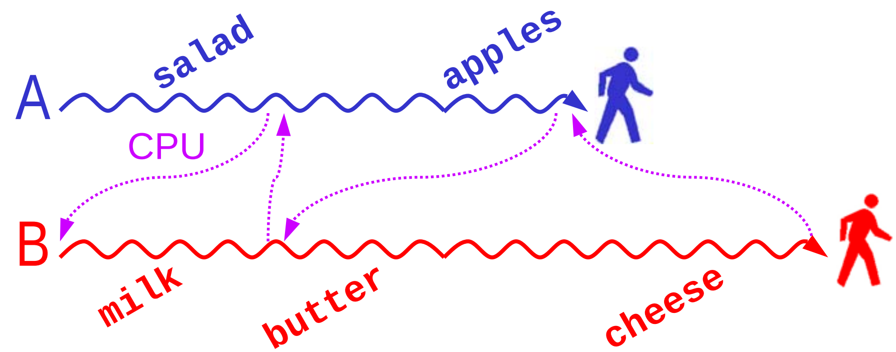
```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

Multithreaded shopping diagram and possible outputs

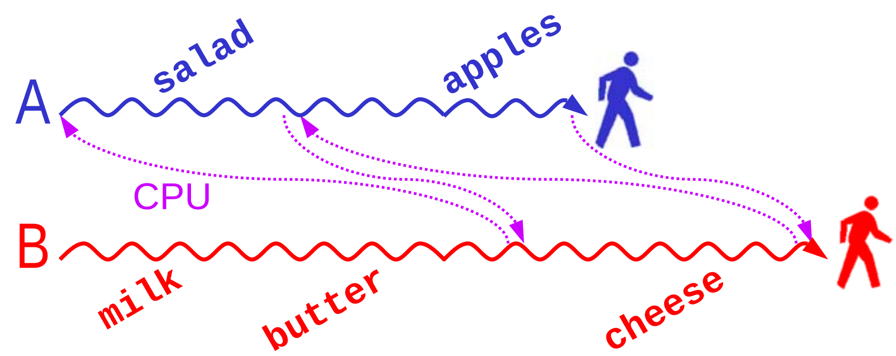
2.c Concurrency
Race conditions & critical regions

- Inconsequential race condition in the shopping scenario
  - ✓ the outcome depends on the CPU scheduling or "interleaving" of the threads (separately, each thread always does the same thing)

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```



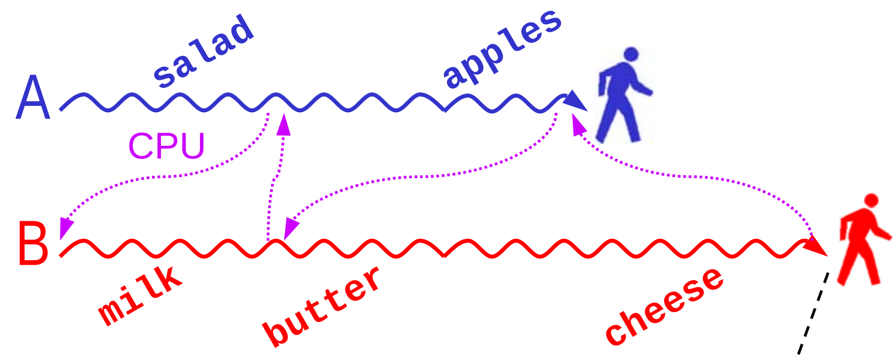
```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

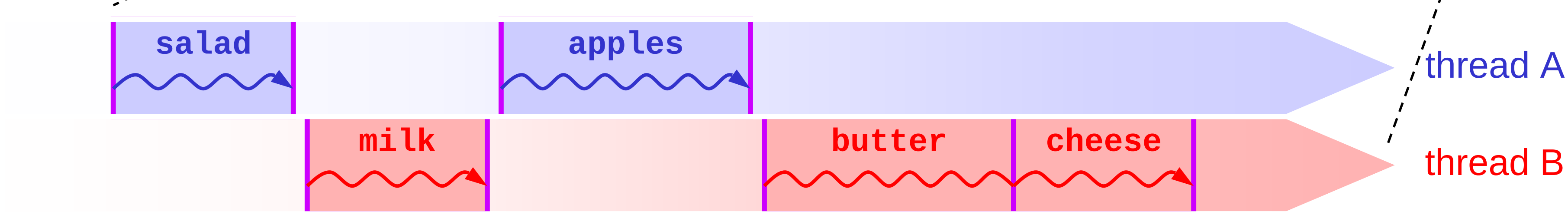


2.c Concurrency
Race conditions & critical regions

- Inconsequential race condition in the shopping scenario
  - ✓ the CPU switches from one process/thread to another, possibly on the basis of a preemptive clock mechanism

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```





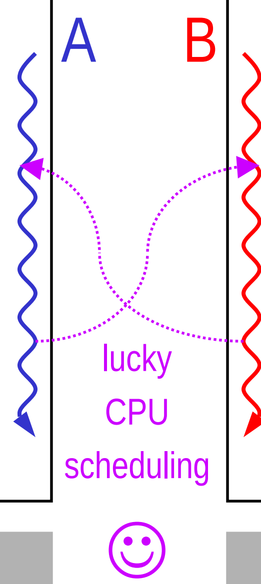
Thread view expanded in real execution time

2.c Concurrency

Race conditions & critical regions

- Consequential race conditions in I/O & variable sharing

```
char chin, chout;

void echo()
{
  do {
  1 chin = getchar();
  2 chout = chin;
  3 putchar(chout);
  }
  while (...);
}
```




```
char chin, chout;

void echo()
{
  do {
  4 chin = getchar();
  5 chout = chin;
  6 putchar(chout);
  }
  while (...);
}
```

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

```
> ./echo
Hello world!
Hello world!
```

Multithreaded echo (lucky)

2.c Concurrency

Race conditions & critical regions

- Consequential race conditions in I/O & variable sharing

```
char chin, chout;

void echo()
{
  do {
  1 chin = getchar();
  5 chout = chin;
  6 putchar(chout);
  }
  while (...);
}
```

A   B

unlucky
CPU
scheduling

☹

```
char chin, chout;

void echo()
{
  do {
  2 chin = getchar();
  3 chout = chin;
  4 putchar(chout);
  }
  while (...);
}
```

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

```
> ./echo
Hello world!
ee....
```

Multithreaded echo (unlucky)

2.c Concurrency

Race conditions & critical regions

- Consequential race conditions in I/O & variable sharing

changed to local variables →

A

```
void echo()
{
  char chin, chout;

  do {
1   chin = getchar();
5   chout = chin;
6   putchar(chout);
  }
  while (...);
}
```

B

```
void echo()
{
  char chin, chout;

  do {
2   chin = getchar();
3   chout = chin;
4   putchar(chout);
  }
  while (...);
}
```

unlucky CPU scheduling ☹

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

```
> ./echo
Hello world!
eH...
```

Multithreaded echo (unlucky)

2.c Concurrency

Race conditions & critical regions

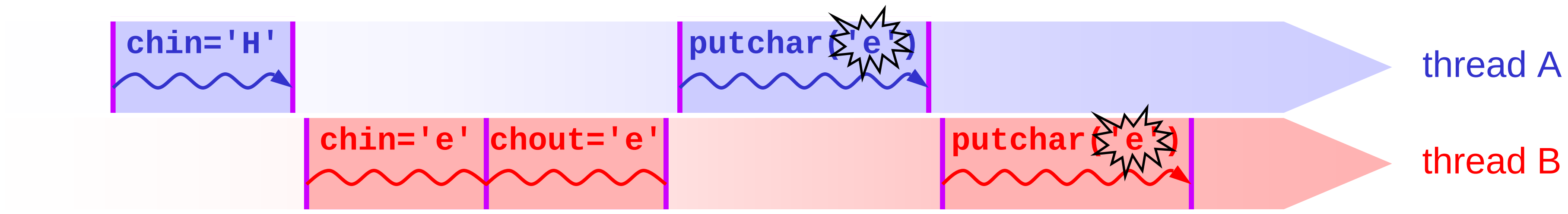
- **Consequential race conditions in I/O & variable sharing**
  - ✓ note that, in this case, replacing the global variables with local variables did not solve the problem
  - ✓ we actually had <u>two</u> race conditions here:
    - one race condition over assigning values to shared variables
    - another race condition over which thread is going to write to output first; this one persisted even after making the variables local to each thread

→ *generally, problematic race conditions may occur whenever resources and/or data are shared (by processes unaware of each other or processes indirectly aware of each other)*

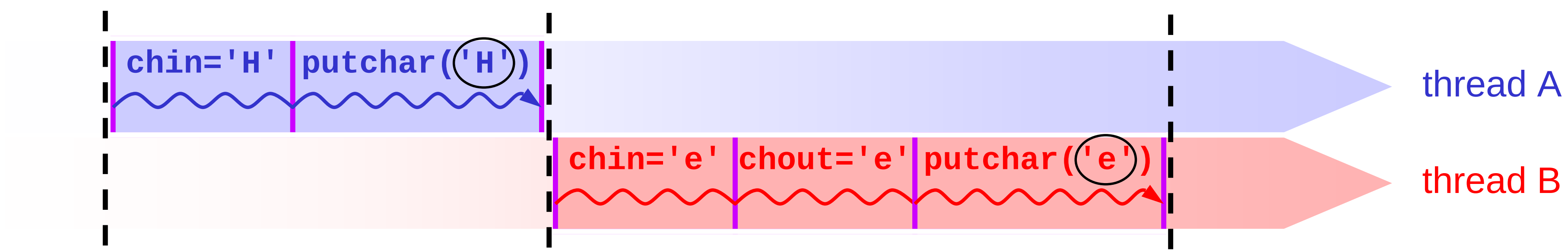
2.c Concurrency

Race conditions & critical regions

- How to avoid race conditions?
  - ✓ find a way to keep the instructions together
  - ✓ this means actually reverting from too much interleaving and going back to “indivisible” blocks of execution!



(a) too much interleaving may create race conditions

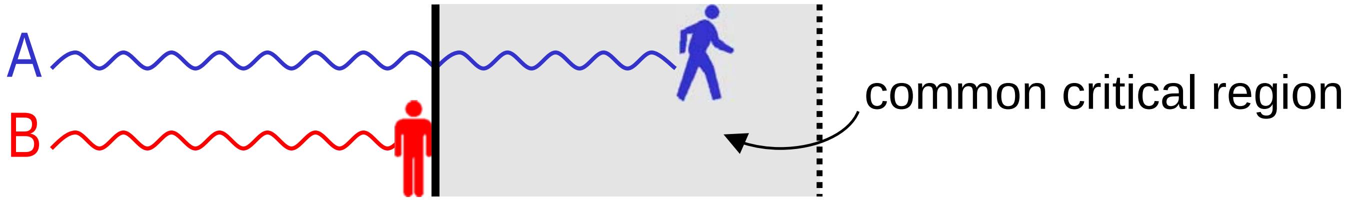


(b) keeping “indivisible” blocks of execution avoids race conditions

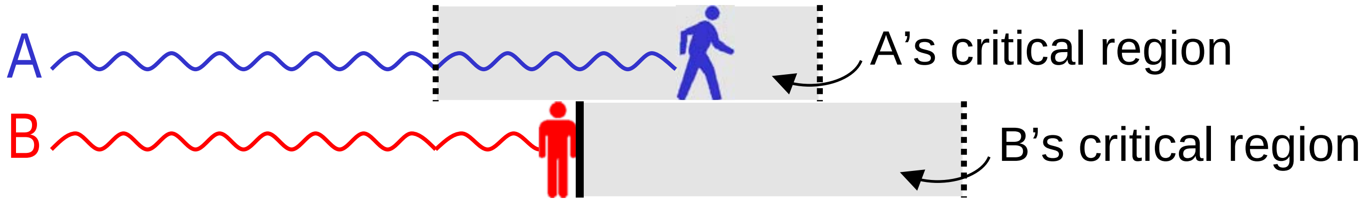
2.c Concurrency

Race conditions & critical regions

- The "indivisible" execution blocks are critical regions
  - ✓ a critical region is a section of code that may be executed by only one process or thread at a time



  - ✓ although it is not necessarily the same region of memory or section of program in both processes



→ *but physically different or not, what matters is that these regions cannot be interleaved or executed in parallel (pseudo or real)*

2.c Concurrency

Race conditions & critical regions

- We need mutual exclusion from critical regions
  - ✓ critical regions can be protected from concurrent access by padding them with entrance and exit mechanisms (we'll see how later): a thread must try to check in, then it must check out

A

```
void echo()
{
   char chin, chout;
   do {
      enter critical region?
      chin = getchar();
      chout = chin;
      putchar(chout);
      exit critical region
   }
   while (...);
}
```

B

```
void echo()
{
   char chin, chout;
   do {
      enter critical region?
      chin = getchar();
      chout = chin;
      putchar(chout);
      exit critical region
   }
   while (...);
}
```

2.c Concurrency

Race conditions & critical regions

Chart of mutual exclusion

1. **mutual exclusion inside** — only one process at a time may be allowed in a critical region

2. **no exclusion outside** — a process stalled in a *non*critical region may not exclude other processes from their critical regions

3. **no indefinite occupation** — a critical region may be only occupied for a finite amount of time

2.c Concurrency

Race conditions & critical regions

Chart of mutual exclusion (cont'd)

4. **no indefinite delay when excluded** — a process may be only excluded for a finite amount of time (no deadlock or starvation)

5. **no delay when not excluded** — a critical region free of access may be entered immediately by a process

6. **nondeterministic scheduling** — no assumption should be made about the relative speeds of processes

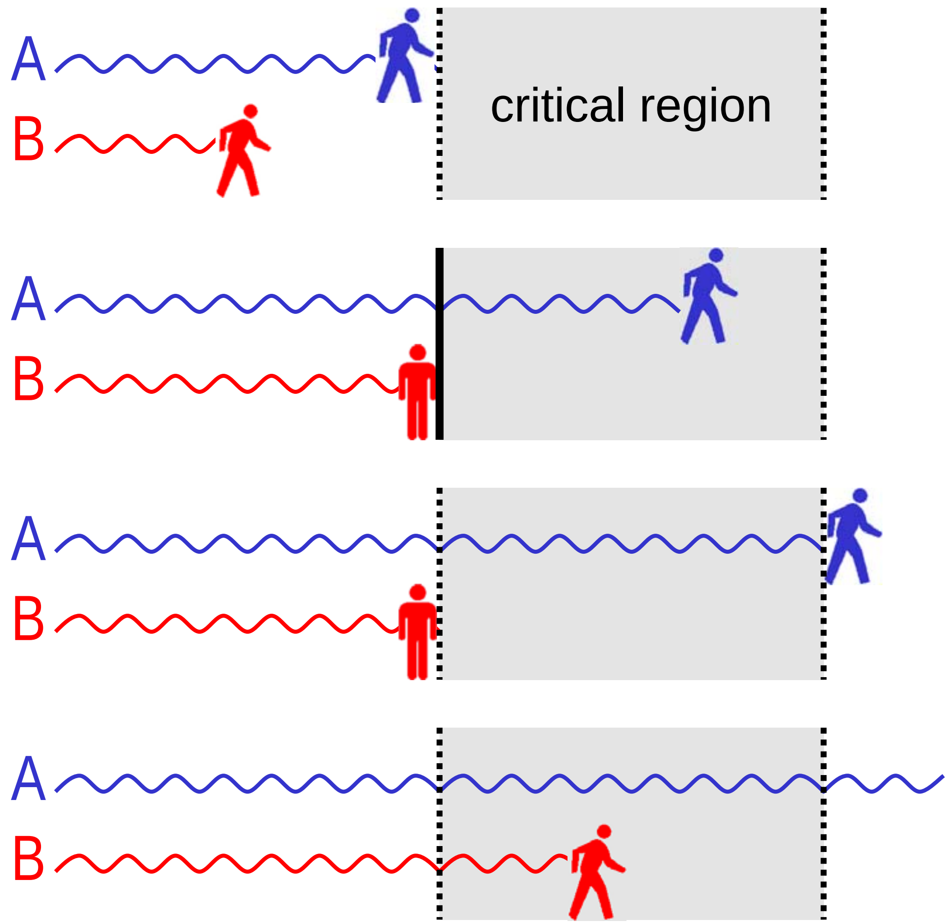
2.c Concurrency

Mutual exclusion by busy waiting

➢ Desired effect: mutual exclusion from the critical region

1. thread A reaches the gate to the critical region (CR) before B
2. thread A enters CR first, preventing B from entering (B is waiting or is blocked)
3. thread A exits CR; thread B can now enter
4. thread B enters CR

HOW is this achieved??

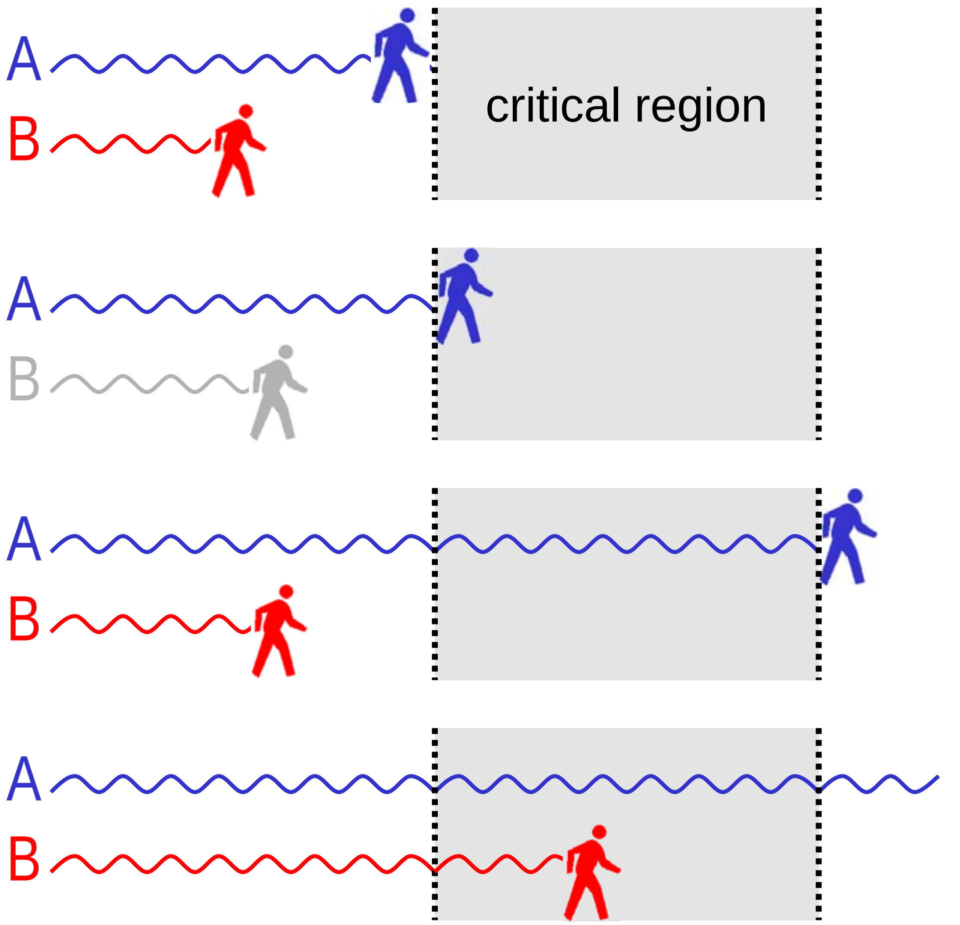


2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 0 — disabling hardware interrupts

1. thread A reaches the gate to the critical region (CR) before B
2. as soon as A enters CR, it disables all interrupts, thus B cannot be scheduled
3. as soon as A exits CR, it reenables interrupts; B can be scheduled again
4. thread B enters CR



2.c Concurrency

Mutual exclusion by busy waiting

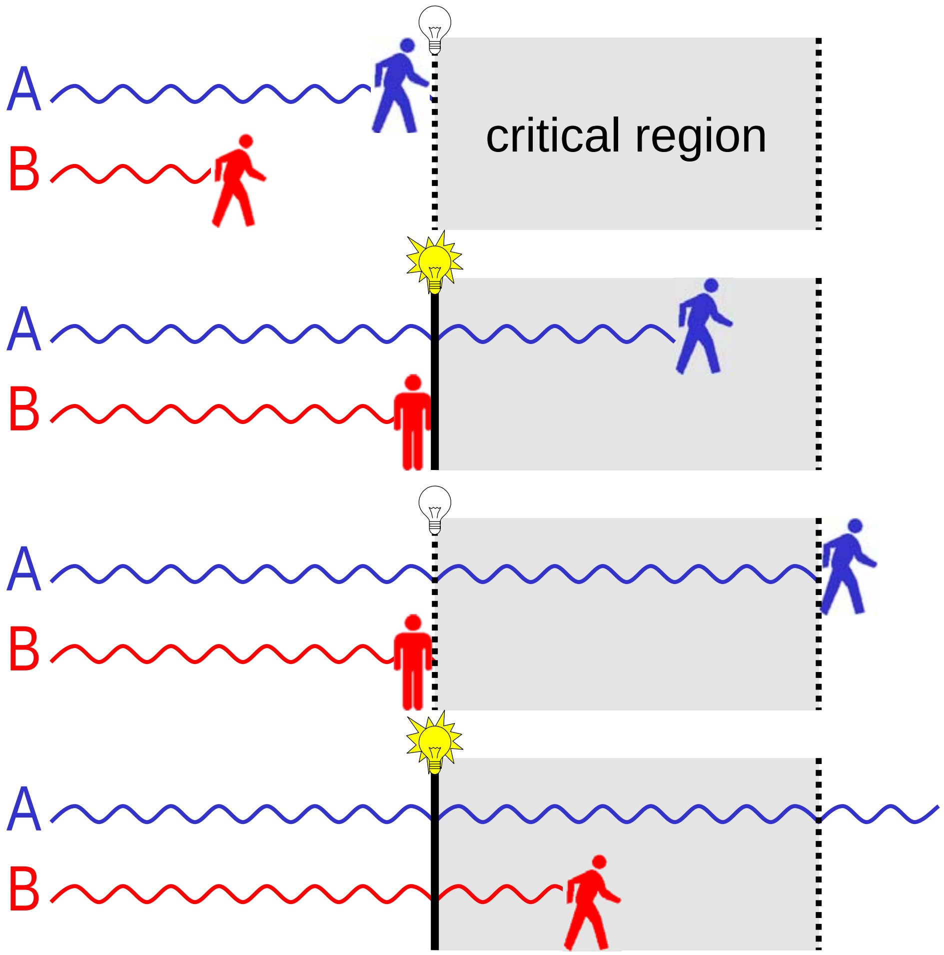
- Implementation 0 — ~~disabling hardware interrupts~~
  - ✓ it works, but it is foolish
  - ✓ what guarantees that the user process is going to ever exit the critical region?
  - ✓ meawhile, the CPU cannot interleave any other task, even unrelated to this race condition
  - ✓ the critical region becomes one *physically* indivisible block, not logically
  - ✓ also, this is not working in multi-processors

```
void echo()
{
   char chin, chout;
   do {
      disable hardware interrupts
      chin = getchar();
      chout = chin;
      putchar(chout);
      reenable hardware interrupts
   }
   while (...);
}
```

2.c Concurrency
Mutual exclusion by busy waiting

- Implementation 1 — simple lock variable

1. thread A reaches CR and finds a lock at 0, which means that A can enter
2. thread A sets the lock to 1 and enters CR, which prevents B from entering
3. thread A exits CR and resets lock to 0; thread B can now enter
4. thread B sets the lock to 1 and enters CR



2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 1 — simple lock variable
  - ✓ the "lock" is a shared variable
  - ✓ entering the critical region means testing and then setting the lock
  - ✓ exiting means resetting the lock

```
while (lock);
   /* do nothing: loop */
lock = TRUE;
```

```
lock = FALSE;
```

```
bool lock = FALSE;

void echo()
{
   char chin, chout;
   do {
      test lock, then set lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      reset lock
   }
   while (...);
}
```

2.c Concurrency

Mutual exclusion by busy waiting

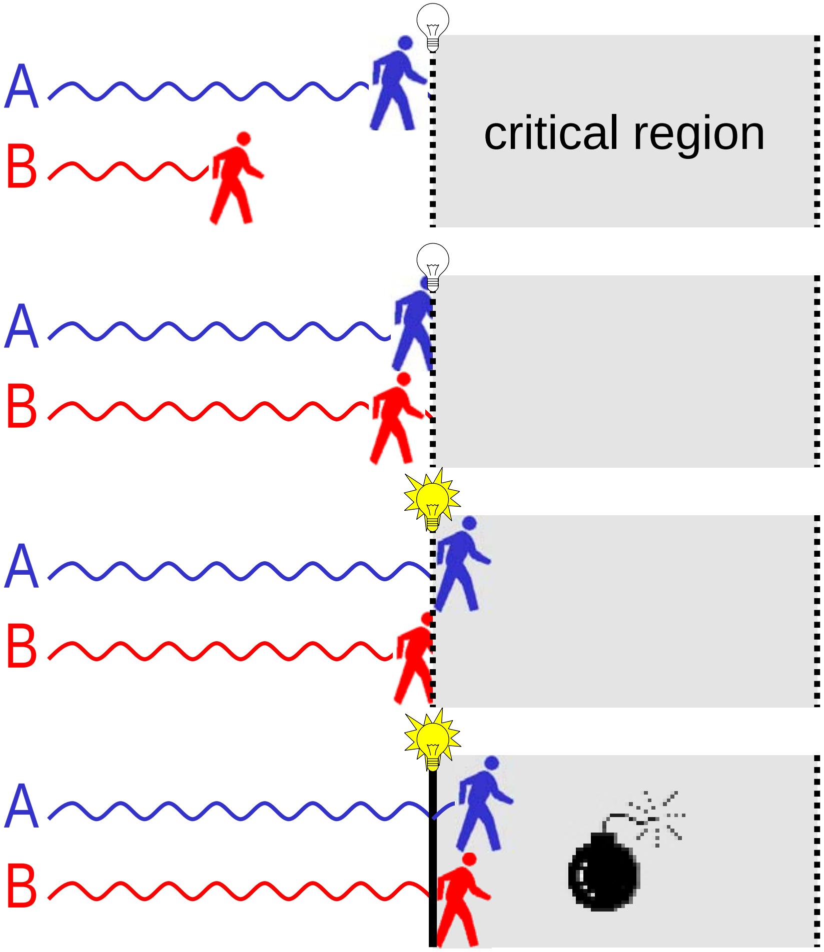
- Implementation 1 — ~~simple lock variable~~

1. thread A reaches CR and finds a lock at 0, which means that A can enter

1.1 but before A can set the lock to 1, B reaches CR and finds the lock is 0, too

1.2 A sets the lock to 1 and enters CR but cannot prevent the fact that . . .

1.3 . . . B is going to set the lock to 1 and enter CR, too



2.c Concurrency

Mutual exclusion by busy waiting

➢ Implementation 1 — ~~simple lock variable~~ 👎

- ✓ suffers from the very fatal flaw we want to avoid: a race condition
- ✓ the problem comes from the small gap between testing that the lock is off and setting the lock

```
while (lock);   lock = TRUE;
```

- ✓ it may happen that the other thread gets scheduled exactly inbetween these two actions (falls in the gap)
- ✓ so they both find the lock off and then they both set it and enter

```
bool lock = FALSE;

void echo()
{
   char chin, chout;
   do {
      test lock, then set lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      reset lock
   }
   while (...);
}
```

2.c Concurrency

Mutual exclusion by busy waiting

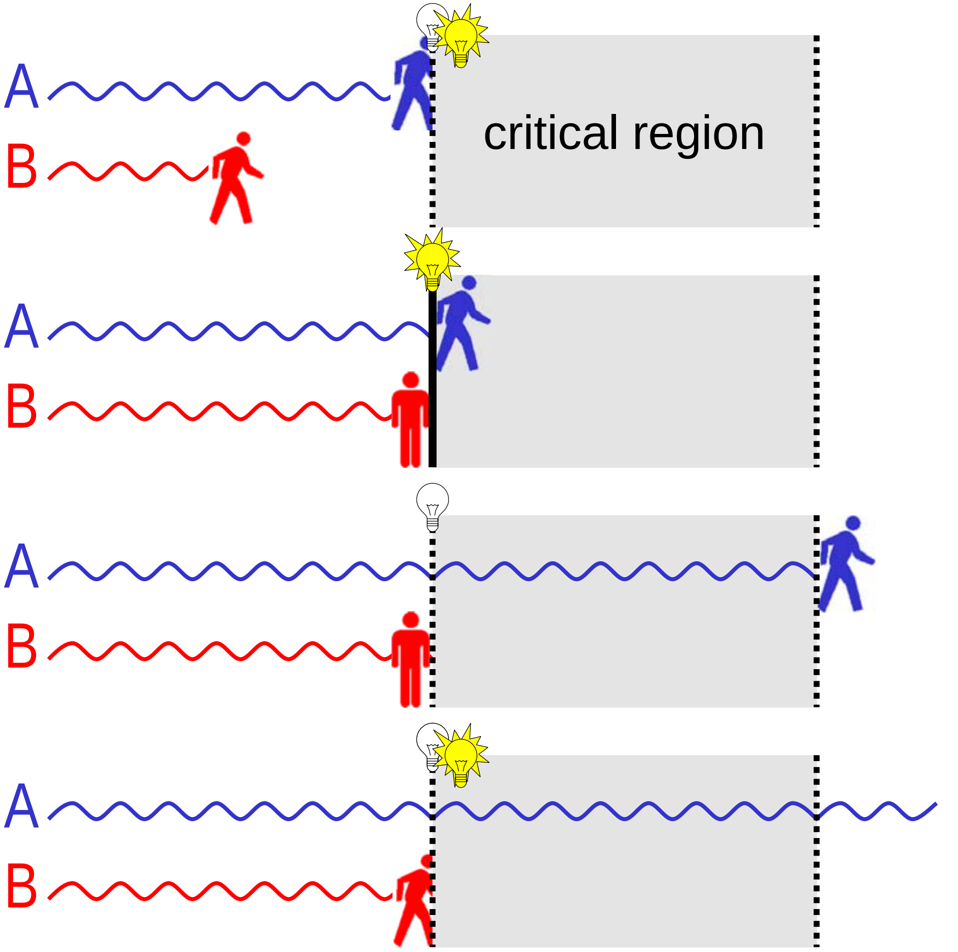
➢ Implementation 2 — "indivisible" lock variable

1. thread A reaches CR and finds the lock at 0 *and* sets it in one shot, then enters

1.1' even if B comes right behind A, it will find that the lock is already at 1

2. thread A exits CR, then resets lock to 0

3. thread B finds the lock at 0 and sets it to 1 in one shot, just before entering CR



2.c Concurrency

Mutual exclusion by busy waiting

- ➢ Implementation 2 — "indivisible" lock variable 👍
  - ✓ the indivisibility of the "test-lock-and-set-lock" operation can be implemented with the hardware instruction **TSL**

```
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
```

```
void echo()
{
    char chin, chout;
    do {
        test-and-set-lock
        chin = getchar();
        chout = chin;
        putchar(chout);
        set lock off
    }
    while (...);
}
```

Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

2.c Concurrency

Mutual exclusion by busy waiting

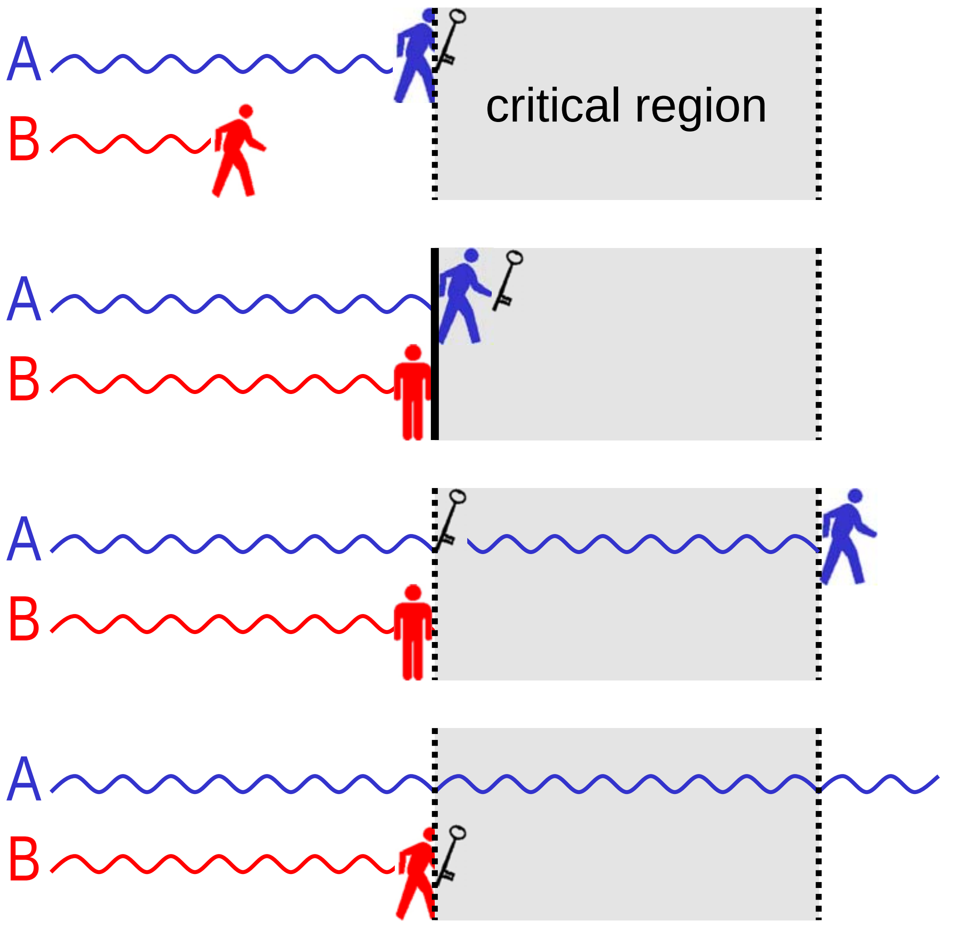
- Implementation 2 — "indivisible" lock ⇔ one key ☝

1. thread A reaches CR and <u>finds a key *and* takes it</u>

1.1' even if B comes right behind A, it will not find a key

2. thread A exits CR and puts the key back in place

3. thread B finds the key and takes it, just before entering CR



2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 2 — "indivisible" lock ⇔ one key ☝
  - ✓ "holding" a unique object, like a key, is an equivalent metaphor for "test-and-set"
  - ✓ this is similar to the "speaker's baton" in some assemblies: only one person can hold it at a time
  - ✓ holding is an indivisible action: you see it and grab it in one shot
  - ✓ after you are done, you release the object, so another process can hold on to it

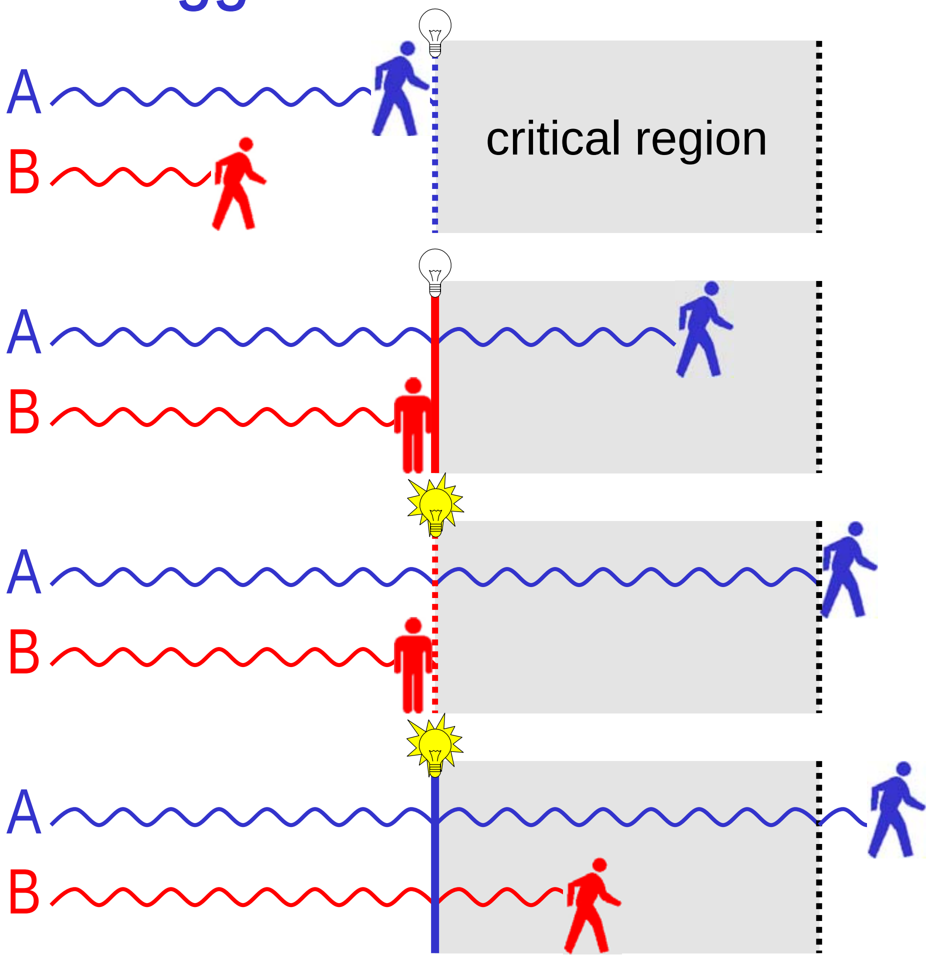
```
void echo()
{
   char chin, chout;
   do {
      take key and run
      chin = getchar();
      chout = chin;
      putchar(chout);
      return key
   }
   while (...);
}
```

2.c Concurrency

Mutual exclusion by busy waiting

➢ Implementation 3 — TSL-free toggle for two threads

1. thread A reaches CR, finds a lock at 0, and enters <u>without changing the lock</u>
2. however, the lock has an <u>opposite meaning for B</u>: "off" means do not enter
3. only when A exits CR does it change the lock to 1; thread B can now enter
4. thread B sets the lock to 1 and enters CR: it will reset it to 0 for A after exiting



2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 3 — TSL-free toggle for two threads
  - ✓ the "toggle lock" is a shared variable used for strict alternation
  - ✓ here, entering the critical region means <u>only testing</u> the toggle: it must be at 0 for A, and 1 for B
  - ✓ exiting means <u>switching</u> the toggle: A sets it to 1, and B to 0

```
bool toggle = FALSE;

void echo()
{
   char chin, chout;
   do {
        test toggle
        chin = getchar();
        chout = chin;
        putchar(chout);
        switch toggle
   }
   while (...);
}
```

A's code	B's code
`while (toggle); /* loop */`	`while (!toggle); /* loop */`
`toggle = TRUE;`	`toggle = FALSE;`

2.c Concurrency

Mutual exclusion by busy waiting

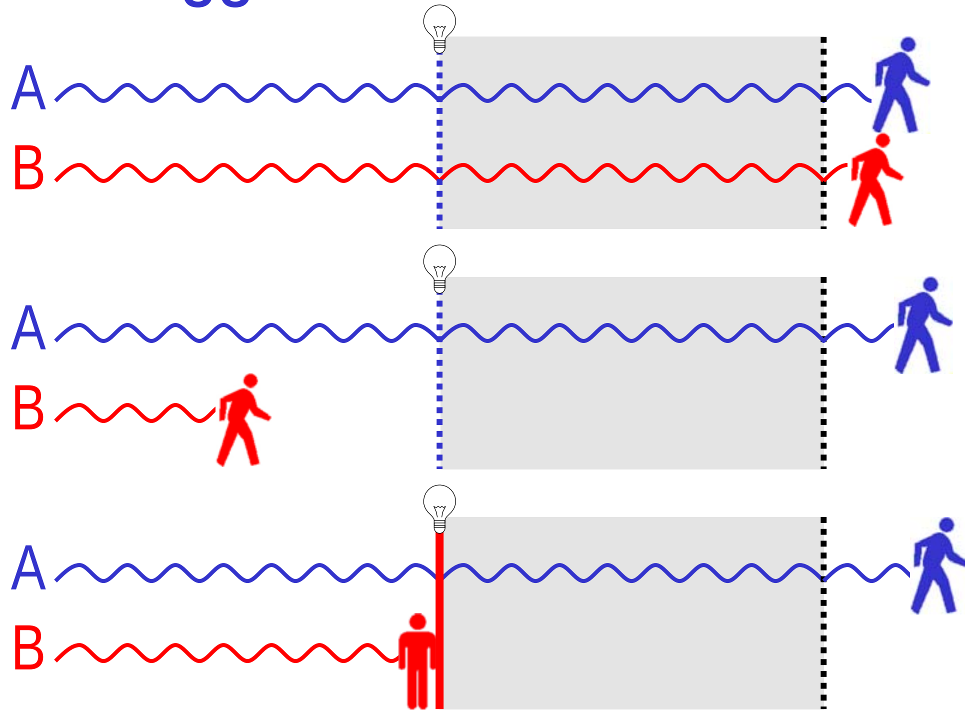
- Implementation 3 — ~~TSL-free toggle for two threads~~

5. thread B exits CR and switches the lock back to 0 to allow A to enter next

5.1 but scheduling happens to make B faster than A and come back to the gate first

5.2 as long as A is still busy, slow or interrupted in its <u>noncritical</u> region, B is barred access to its CR

→ *this violates item 2. of the chart of mutual exclusion*



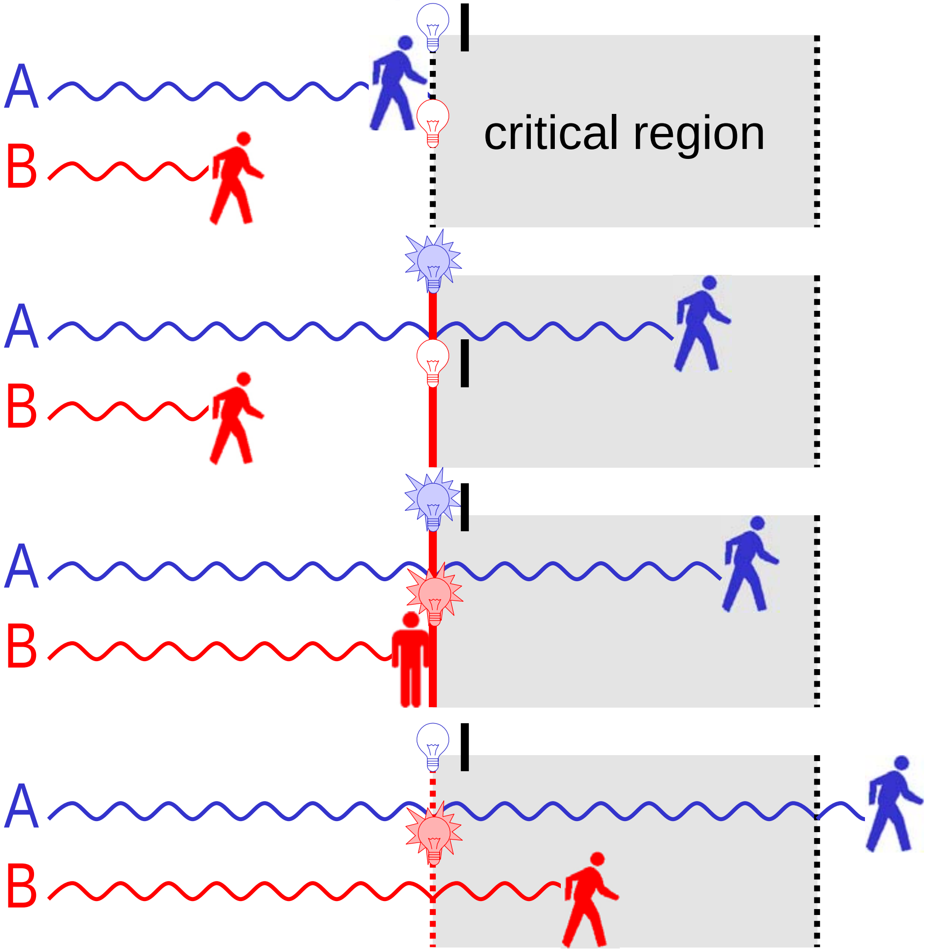
→ *this implementation avoids TSL by splitting test & set and putting them in enter & exit; nice try... but flawed!*

2.c Concurrency

Mutual exclusion by busy waiting

➢ Implementation 4 — Peterson's no-TSL, no-alternation

1. A and B each have their own lock; an extra toggle is also masking either lock
2. A arrives first, sets its lock, pushes the mask to the other lock and may enter
3. then, B also sets its lock & pushes the mask, but must wait until A's lock is reset
4. A exits the CR and resets its lock; B may now enter



2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 4 — Peterson's no-TSL, no-alternation
  - ✓ the mask & two locks are shared
  - ✓ entering means: setting one's lock, pushing the mask and tetsing the <u>other's</u> combination
  - ✓ exiting means resetting the lock

A's code

```
lock[A] = TRUE;
mask = B;
while (lock[B] &&
       mask == B);
   /* loop */

lock[A] = FALSE;
```

B's code

```
lock[B] = TRUE;
mask = A;
while (lock[A] &&
       mask == A);
   /* loop */

lock[B] = FALSE;
```

```
bool lock[2];
int mask;
int A = 0, B = 1;
void echo()
{
   char chin, chout;
   do {
      set lock, push mask, and test
      chin = getchar();
      chout = chin;
      putchar(chout);
      reset lock
   }
  while (...);
}
```

2.c Concurrency

Mutual exclusion by busy waiting

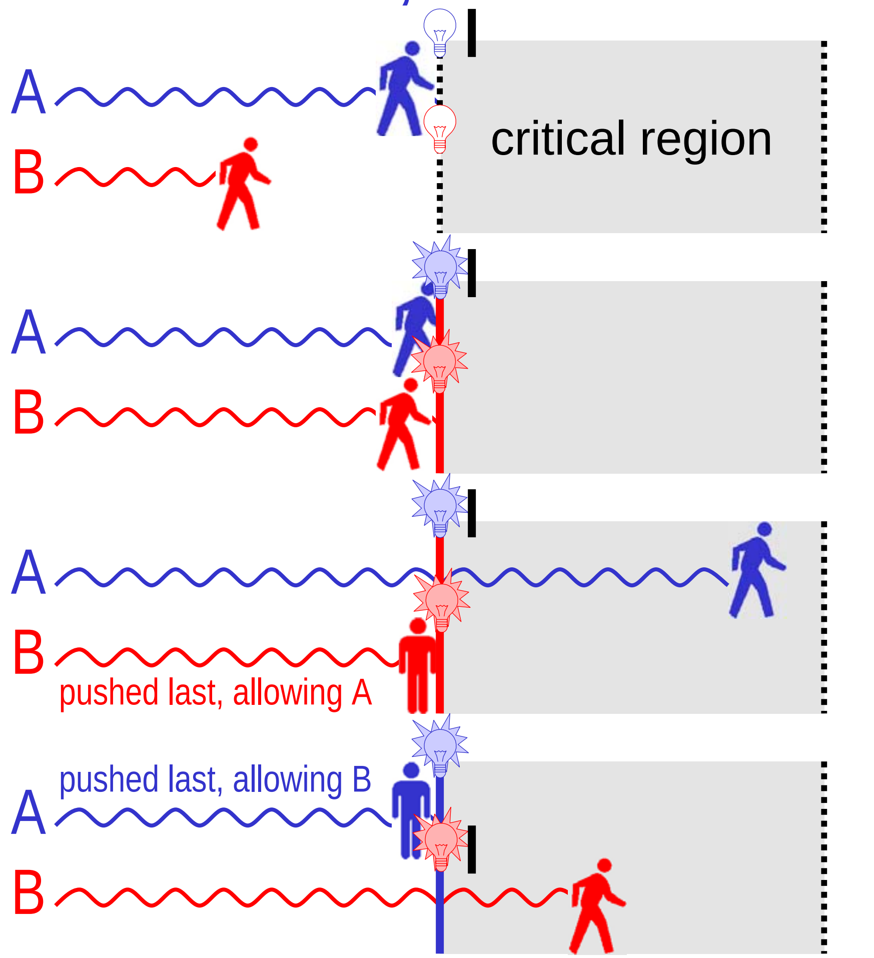
- Implementation 4 — Peterson's no-TSL, no-alternation

1. A and B each have their own lock; an extra toggle is also masking either lock

2.1 A is interrupted between setting the lock & pushing the mask; B sets its lock

2.2 now, both A and B race to push the mask: whoever does it <u>last</u> will allow the <u>other</u> one inside CR

→ *mutual exclusion holds!!* *(no bad race condition)*



2.c Concurrency

Mutual exclusion by busy waiting

- Summary of these implementations of mutual exclusion
  - ✓ Impl. 0 — disabling hardware interrupts
    - NO: race condition avoided, but can crash the system!
  - ✓ Impl. 1 — simple lock variable (unprotected)
    - NO: still suffers from race condition
  - ✓ Impl. 2 — indivisible lock variable (TSL)
    - YES: works, but requires hardware

    this will be the basis for "mutexes"
  - ✓ Impl. 3 — TSL-free toggle for two threads
    - NO: race condition avoided inside, but lockup outside
  - ✓ Impl. 4 — Peterson's no-TSL, no-alternation
    - YES: works in software, but processing overhead

2.c Concurrency
Mutual exclusion by busy waiting

- Problem: all implementations (1-4) rely on <u>busy waiting</u>
  - ✓ "busy waiting" means that the process/thread continuously executes a tight loop until some condition changes
  - ✓ busy waiting is bad:
    - **waste of CPU time** — the busy process is not doing anything useful, yet remains "Ready" instead of "Blocked"
    - **paradox of inversed priority** — by looping indefinitely, a higher-priority process B may starve a lower-priority process A, thus preventing A from exiting CR and . . . liberating B! (B is working against its own interest)

→ *we need for the waiting process to <u>block</u>, not keep idling*

2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Implementation 2' — indivisible <u>blocking</u> lock = mutex
  - ✓ a mutex is a safe lock variable with blocking, instead of tight looping
  - ✓ if **TSL** returns 1, then <u>voluntarily yield the CPU</u> to another thread

```
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 later
ok:  RET                  | return to caller; critical region entered
```

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

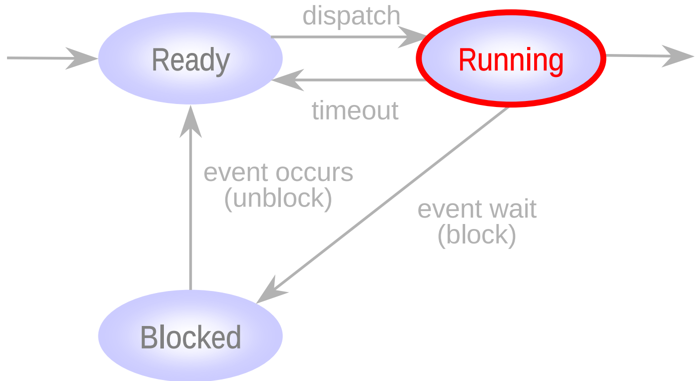
```
void echo()
{
   char chin, chout;
   do {
      test-and-set-lock  or BLOCK
      chin = getchar();
      chout = chin;
      putchar(chout);
      set lock off
   }
   while (...);
}
```

Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

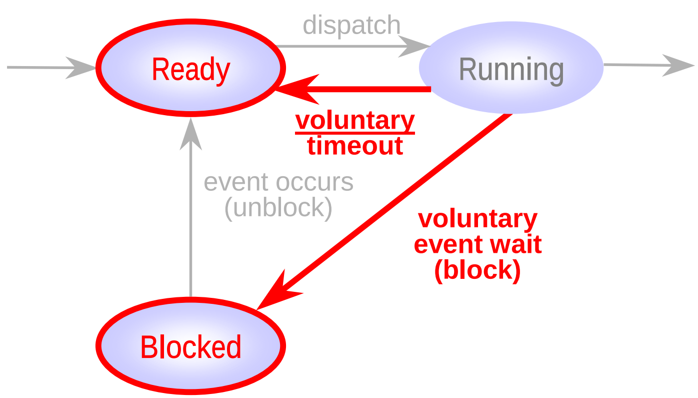
2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Difference between busy waiting and blocked
  - ✓ in busy waiting, the PC is always looping (increment & jump back)
  - ✓ it can be preemptively interrupted but will loop again tightly whenever rescheduled → *tight polling*



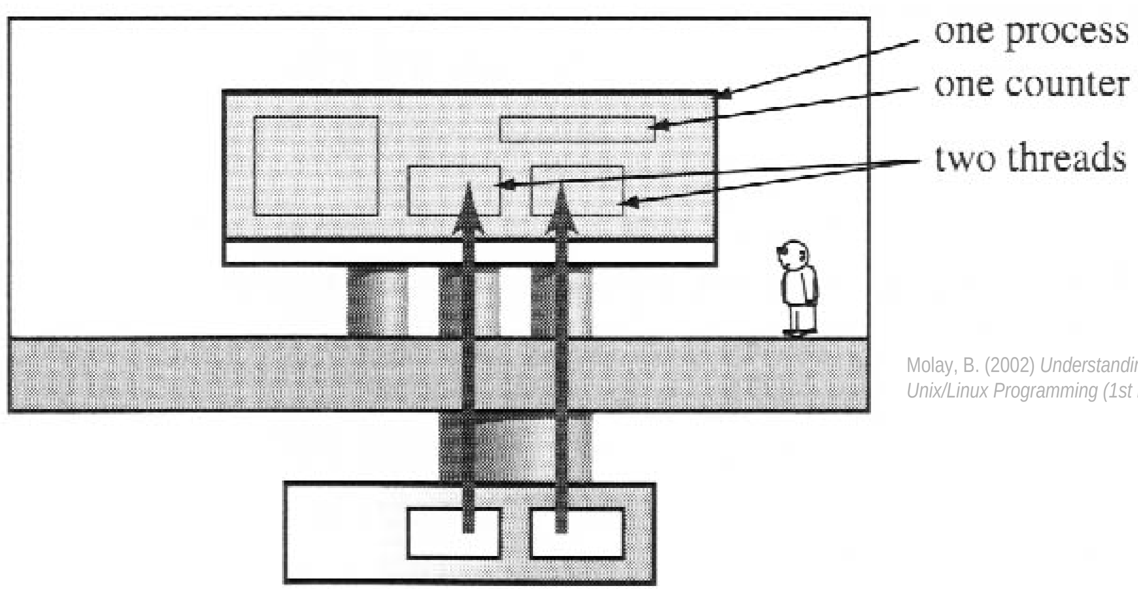
  - ✓ when blocked, the process's PC stalls after executing a "yield" call
  - ✓ either the process is only timed out, thus it is "Ready" to loop-and-yield again → *sparse polling*
  - ✓ or it is truly "Blocked" and put in event queue → *condition waiting*



2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Illustration of mutex use: shared word counter
  - we want to count the total number of words in 2 files
  - we use 1 global counter variable and 2 threads: each thread reads from a different file and increments the shared counter



Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

A common counter for two threads

2.c Concurrency

Mutual exclusion & synchronization — mutexes

```
int total_words;

void main(...)
{
    ...declare, initialize...
    pthread_create(&th1, NULL, count_words, (void *)filename1);
    pthread_create(&th2, NULL, count_words, (void *)filename2);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("total words = %d", total_words);
}

void *count_words(void *filename)
{
    ...open file...
    while (...get next char...) {
        if (...char is not alphanum & previous char is alphanum...) {
            total_words++;
        }
        ......
```

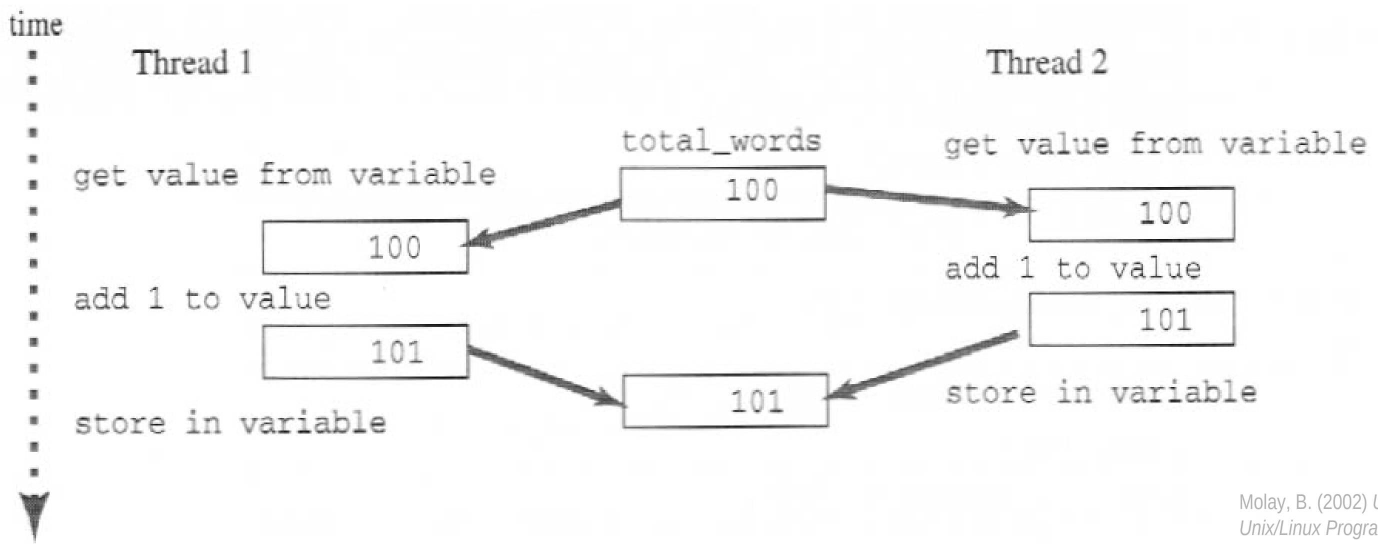
`total_words = total_words + 1;`
is not necessarily atomic! (depends on machine code and stage of execution)

Multithreaded shared counter with possible race condition

2.c Concurrency

Mutual exclusion & synchronization — mutexes

- A race condition can occur when incrementing counter
  - ✓ if not atomic, the increment block of thread 1, “get1-add1” may be interleaved with the increment block of thread 2, “get2-add2” to produce “get1-get2-add1-add2” or “get1-get2-add2-add1”
  - → *this results in missing one count*



Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

Two threads race to increment the counter

2.c Concurrency

Mutual exclusion & synchronization — mutexes

```
int total_words;
pthread_mutex_t counter_lock = PTHREAD_MUTEX_INITIALIZER;

void main(int ac, char *av[])
{
    ...declare, initialize...
    pthread_create(&th1, NULL, count_words, (void *)filename1);
    pthread_create(&th2, NULL, count_words, (void *)filename2);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("total words = %d", total_words);
}

void *count_words(void *filename)
{
    ...open file...
    while (...get next char...) {
        if (...char is not alphanum & previous char is alphanum...) {
            pthread_mutex_lock(&counter_lock);
            total_words++;
            pthread_mutex_unlock(&counter_lock);
        }
        ......
```

protect the critical region with mutual exclusion

Mulithreaded shared counter with mutex protection

2.c Concurrency

Mutual exclusion & synchronization — mutexes

- System calls for thread exclusion with mutexes
  - ✓ `err = pthread_mutex_lock(pthread_mutex_t *m)`

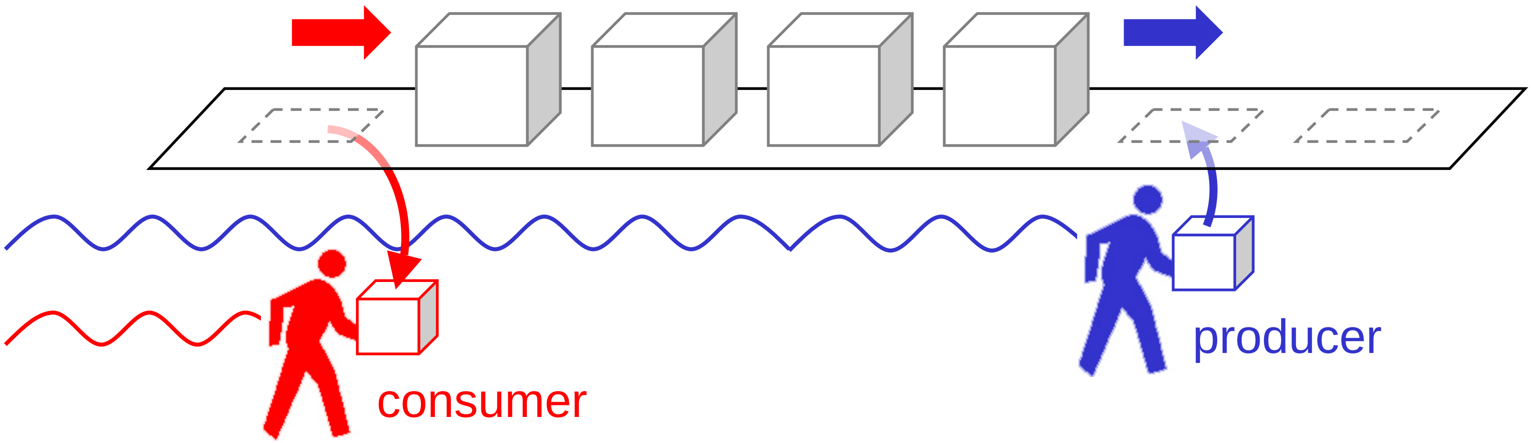
    locks the specified mutex
    - if the mutex is unlocked, it becomes locked and owned by the calling thread
    - if the mutex is already locked by another thread, the calling thread is blocked until the mutex is unlocked
  - ✓ `err = pthread_mutex_unlock(pthread_mutex_t *m)`

    releases the lock on the specified mutex
    - if there are threads blocked on the specified mutex, one of them will acquire the lock to the mutex

2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Real-world mutex use: the producer/consumer problem
  - ✓ producer — generates data items and places them in a buffer
  - ✓ consumer — takes the items out of the buffer to use them
  - ✓ example 1: a print program produces characters that are consumed by a printer
  - ✓ example 2: an assembler produces object modules that are consumed by a loader

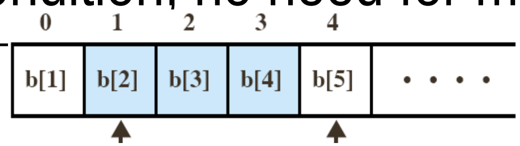


2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Unbounded buffer, 1 producer, 1 consumer
  - ✓ **in** modified only by producer and **out** only by consumer
  - ✓ no race condition; no need for mutexes, just a while loop

```
item[] b;
int in, out;
```




```
void producer()
{
  while (true) {
    item = produce();

    b[in] = item;
    in++;
  }
}
```

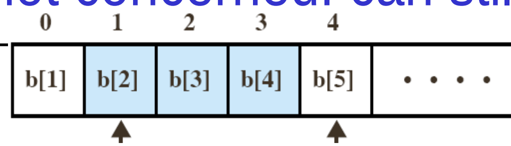
```
void consumer()
{
  while (true) {
    while (out == in);

    item = b[out];
    out++;

    consume(item);
  }
}
```

2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Unbounded buffer, 1 producer, *N* consumers
  - ✓ **`out`** shared by all consumers → mutex among consumers
  - ✓ producer not concerned: can still add items to buffer at any time

```
item[] b;
int in, out;
```




```
mutex out_mutex;
```

```
void producer()
{
  while (true) {
    item = produce();

    b[in] = item;
    in++;
  }
}
```

but this implementation is flawed: all consumers pass the "while" at once, end up waiting at lock, then enter even if buffer is empty . . .

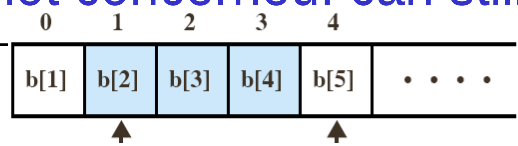
```
void consumer()
{
  while (true) {
    while (out == in);
    lock(out_mutex);
    item = b[out];
    out++;
    unlock(out_mutex);
    consume(item);
  }
}
```

2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Unbounded buffer, 1 producer, *N* consumers
  - ✓ **out** shared by all consumers → mutex among consumers
  - ✓ producer not concerned: can still add items to buffer at any time

```
item[] b;
int in, out;
```




```
mutex out_mutex;
```

```
void producer()
{
  while (true) {
    item = produce();

    b[in] = item;
    in++;
  }
}
```

```
void consumer()
{
  while (true) {
    lock(out_mutex);
    while (out == in);
    item = b[out];
    out++;
    unlock(out_mutex);
    consume(item);
  }
}
```

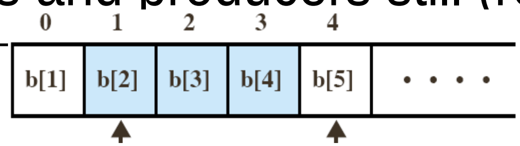
this implementation is correct: even if a consumer loops inside, it means the buffer is empty anyway, so the others may as well be blocked outside . . .

2.c Concurrency

Mutual exclusion & synchronization — mutexes

- Unbounded buffer, *N* producers, *N* consumers
  - ✓ **`in`** shared by all producers → other mutex among producers
  - ✓ consumers and producers still (relatively) independent

```
item[] b;
int in, out;
```




```
mutex out_mutex;
mutex in_mutex;
```

```
void producer()
{
  while (true) {
    item = produce();
    lock(in_mutex);
    b[in] = item;
    in++;
    unlock(in_mutex);
  }
}
```

```
void consumer()
{
  while (true) {
    lock(out_mutex);
    while (out == in);
    item = b[out];
    out++;
    unlock(out_mutex);
    consume(item);
  }
}
```

still correct, but in all cases the consumers are busy waiting . . .

2.c Concurrency

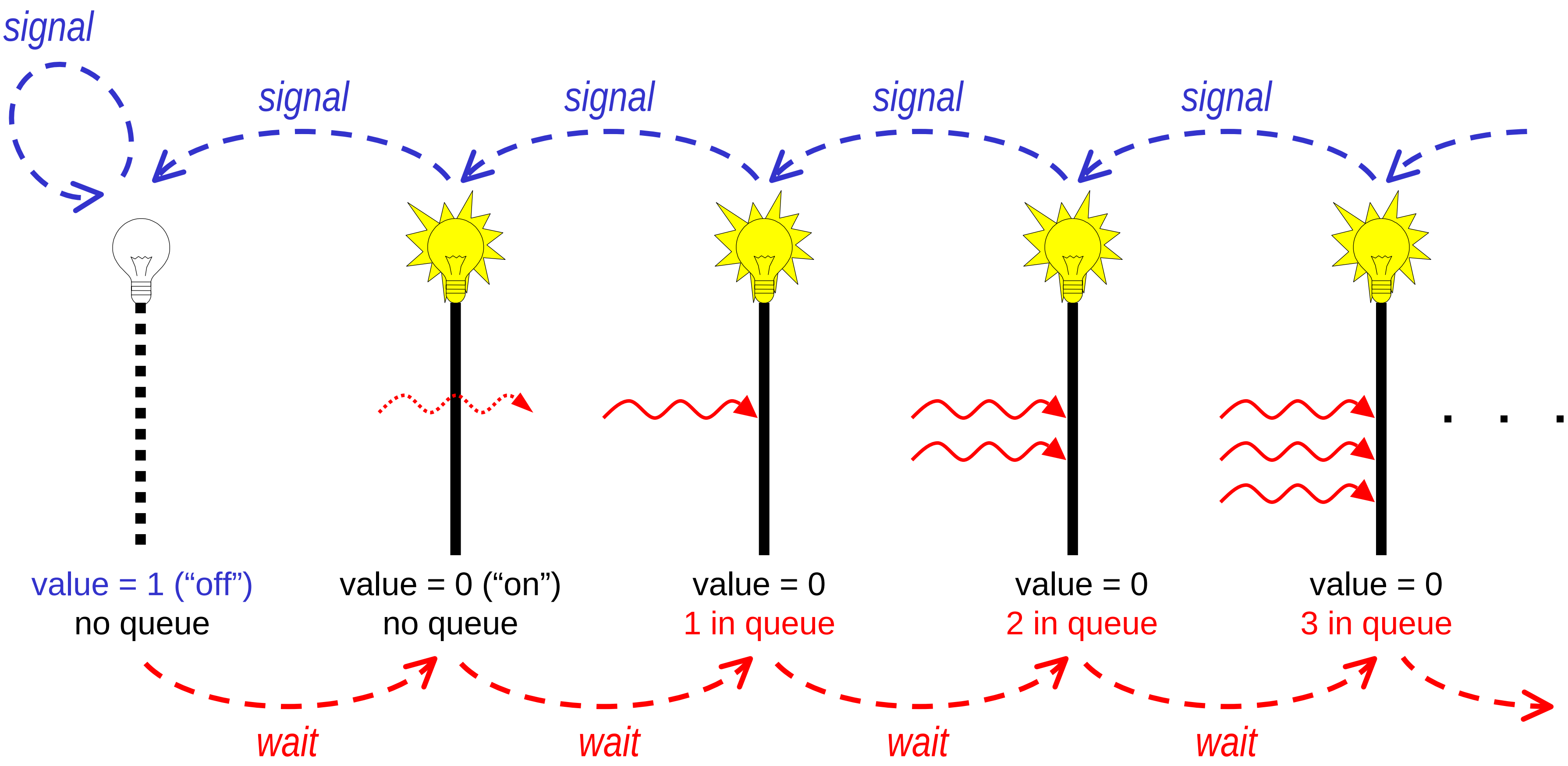
Mutual exclusion & synchronization — semaphores

- **Synchronization**
  - ✓ processes can also **cooperate** by means of simple signals, without defining a “critical region”
  - ✓ like mutexes: instead of looping, a process can block in some place until it receives a specific **signal** from the other process
- **Binary semaphore ⇔ mutex**
  - ✓ a binary semaphore is a variable that has a value 0 or 1
  - ✓ a **wait** operation attempts to <u>decrement</u> the semaphore
    - $1 \rightarrow 0$ and goes through; $0 \rightarrow$ blocks
  - ✓ a **signal** operation attempts to <u>increment</u> the semaphore
    - $1 \rightarrow 1$, no change; $0 \rightarrow$ unblocks or becomes 1

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Binary semaphore ⇔ mutex

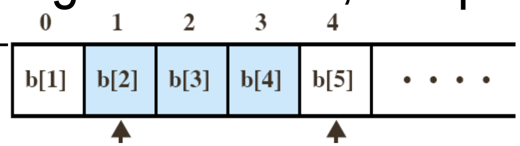


2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Unbounded buffer, 1 producer, 1 consumer <u>with sync</u>
  - ✓ if buffer is empty, the consumer waits on a semaphore
  - ✓ if buffer just got one item, the producer signals to the consumer

```
item[] b;
int in, out;
```



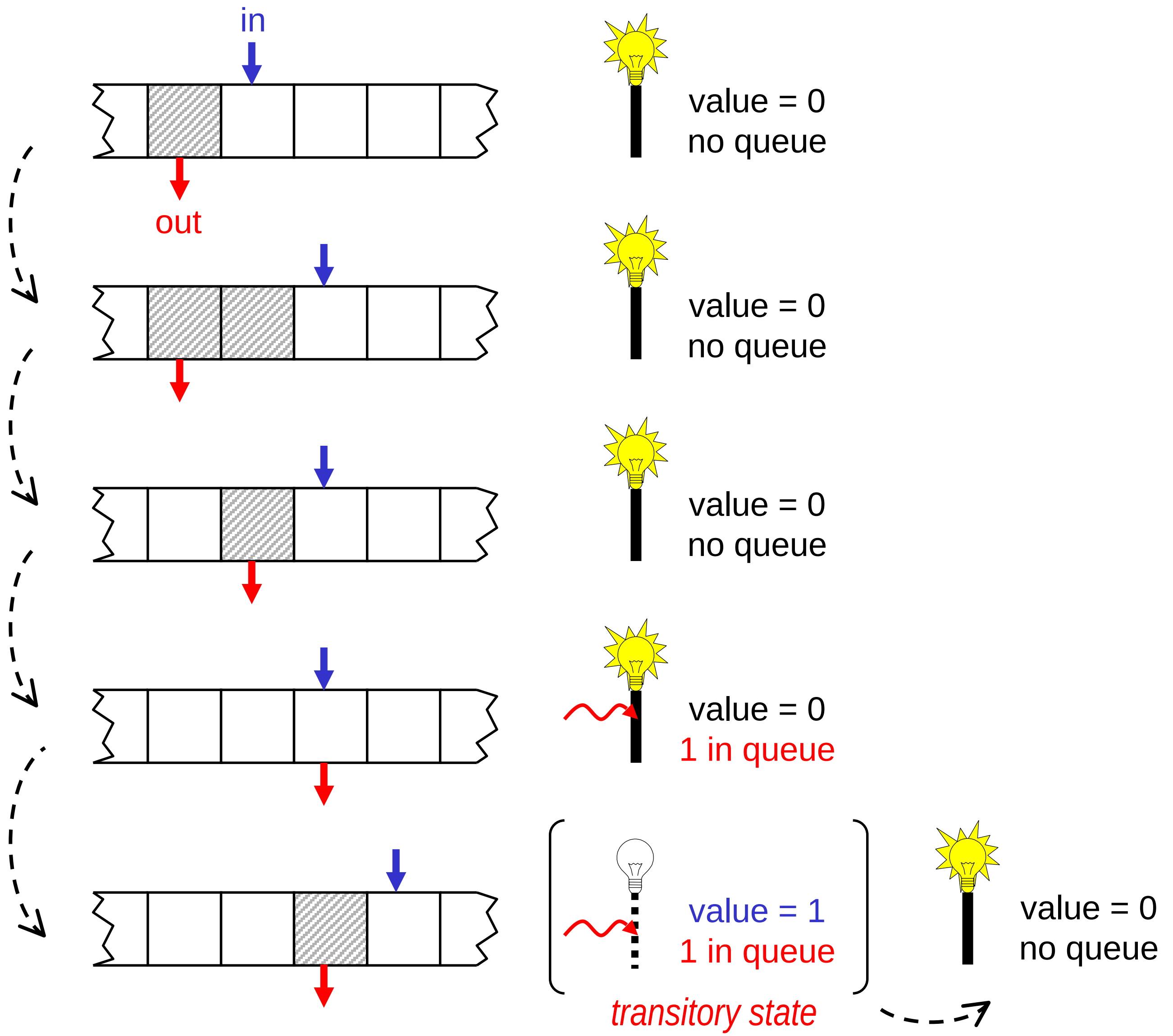

```
bin_semaphore Bsem = 0;
```

```
void producer()
{
  while (true) {
    item = produce();

    b[in] = item;
    in++;
    if (out == in-1)
      signal(Bsem);
  }
}
```

unfortunately, this can lead to an inconsistent semaphore state . . .

```
void consumer()
{
  while (true) {
    while (out == in);
    if (out == in)
      wait(Bsem);
    item = b[out];
    out++;

    consume(item);
  }
}
```

2.c Concurrency
Mutual exclusion & synchronization — semaphores



1.
```
x = produce();
b[in++] = x;
```

2.
```
x = b[out++];
consume(x);
```

3.
```
x = b[out++];
consume(x);
wait(sem);
```
empty →

4.
```
x = produce();
b[in++] = x;
signal(sem);
```
one item →

2.c Concurrency

Mutual exclusion & synchronization — semaphores

3.'
```
x = b[out++];
consume(x);
```

4.'
```
x = produce();
b[in++] = x;
signal(sem);
```

5.'

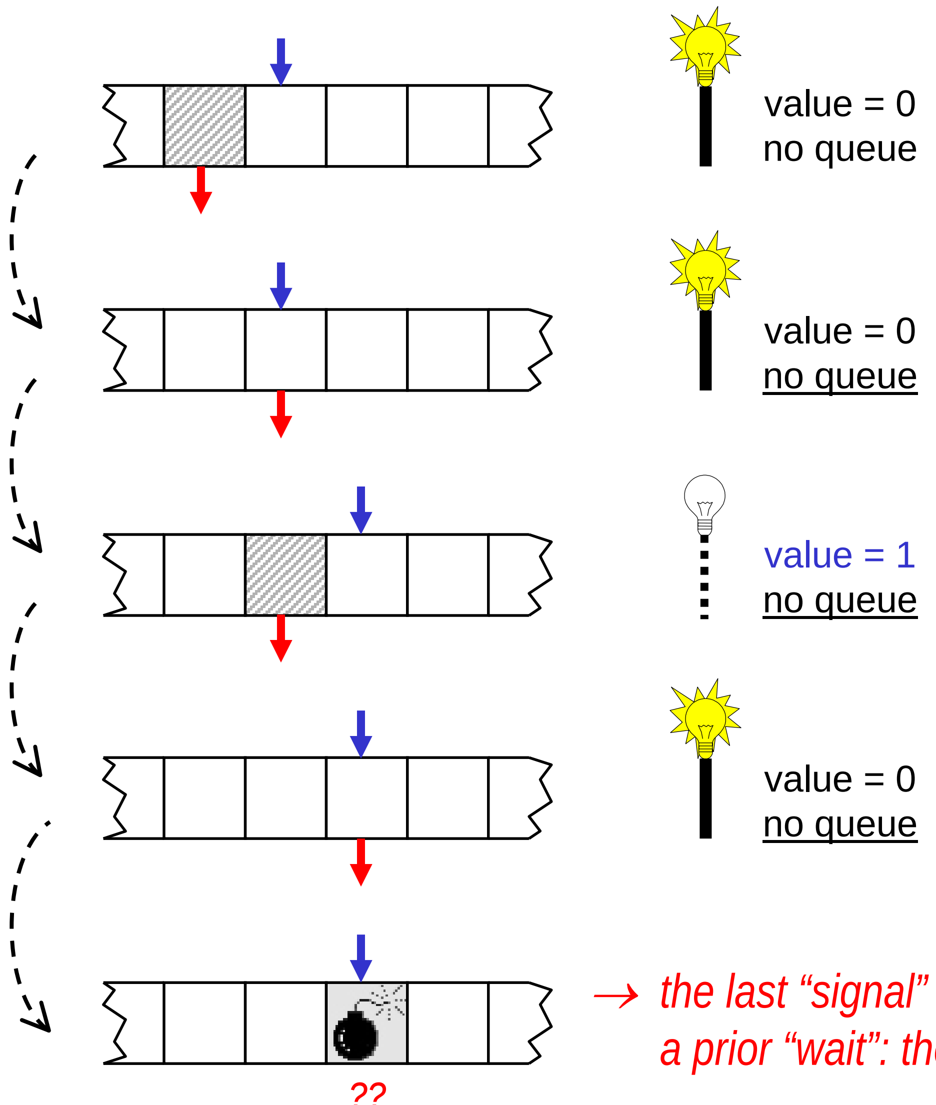
NOT empty → ~~`wait(sem);`~~

```
x = b[out++];
consume(x);
wait(sem);
```

now empty → `wait(sem);` *→*

. . . but goes through

6.'
```
x = b[out++];
consume(x);
```



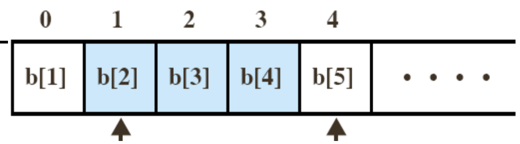
→ the last "signal" was not matched by a prior "wait": they missed each other

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Unbounded buffer, 1 producer, 1 consumer <u>with sync</u>
  - ✓ we need to create critical areas to keep “consuming” and “checking the semaphore” together

```
item[] b;
int in, out;
```




```
bin_semaphore Bsem = 0;
```

```
void producer()
{
  while (true) {
    item = produce();
    lock(buf_mutex);
    b[in] = item;
    in++;
    if (out == in-1)
      signal(Bsem);
    unlock(buf_mutex);
  }
}
```

```
void consumer()
{
  while (true) {
    lock(buf_mutex);
    if (out == in)
      wait(Bsem);
    item = b[out];
    out++;
    unlock(buf_mutex);
    consume(item);
```

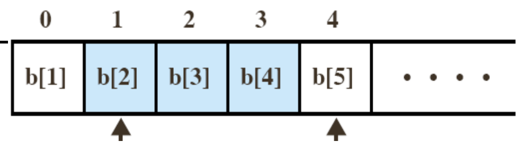
but there is a deadlock: here the consumer is blocking the producer, not other consumers . . .

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- **Unbounded buffer, 1 producer, 1 consumer with sync**
  - ✓ the consumer needs to remember the current state of `in` & `out`, so it can exit the CR before checking the semaphore

```
item[] b;
int in, out;
```




```
bin_semaphore Bsem = 0;
```

```
void producer()
{
  while (true) {
    item = produce();
    lock(buf_mutex);
    b[in] = item;
    in++;
    if (out == in-1)
      signal(Bsem);
    unlock(buf_mutex);
  }
}
```

finally correct!

```
void consumer()
{
  while (true) {
    if (out == in0)
      wait(Bsem);
    lock(buf_mutex);
    item = b[out];
    out++; in0 = in;
    unlock(buf_mutex);
    consume(item);
  }
}
```

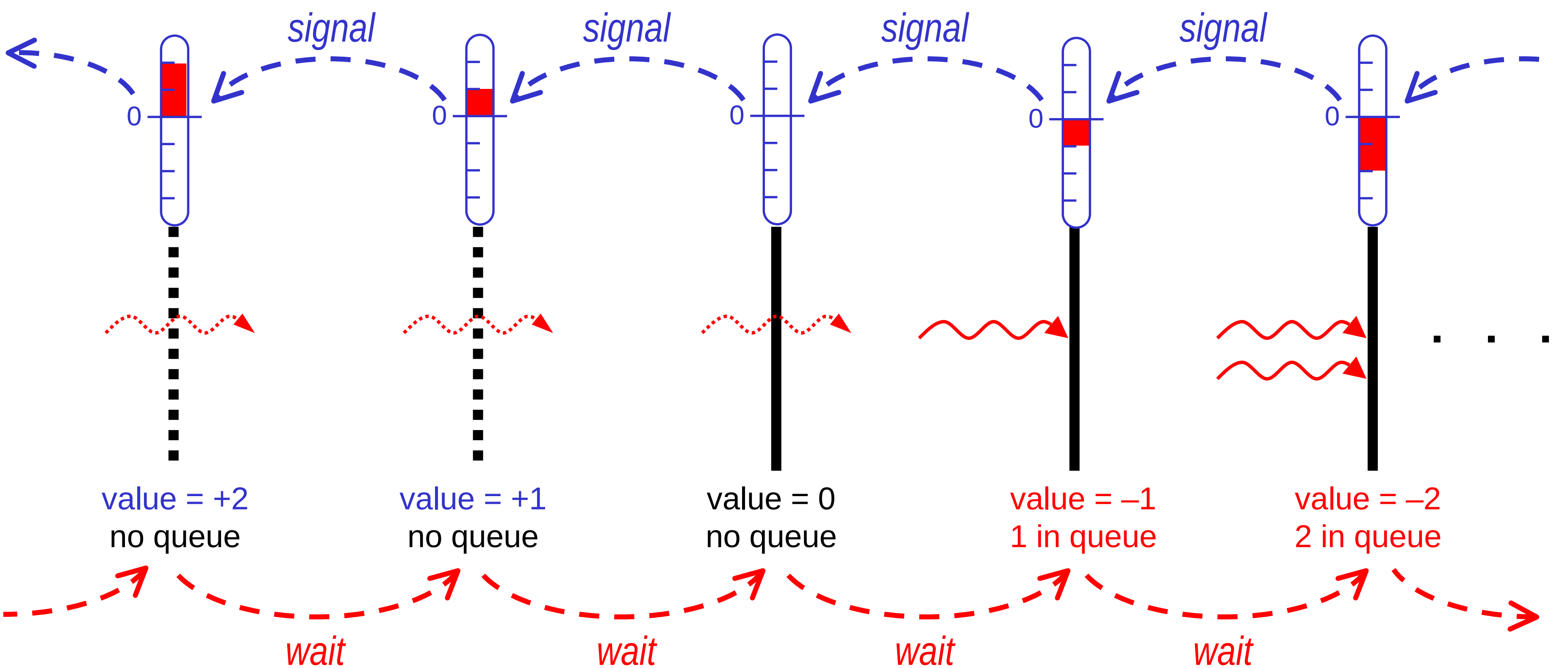
2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Semaphores are used for signaling between processes
  - ✓ semaphores can be used for **mutual exclusion**
  - ✓ binary semaphores are the same as mutexes
  - ✓ integer semaphores can be used to allow more than one process inside a critical region; generally:
    - the positive value of an integer semaphore corresponds to a maximum number of processes allowed concurrently inside a critical region
    - the negative value of an integer semaphore corresponds to the number of processes currently waiting in the queue
  - ✓ binary and integer semaphores can also be used for **synchronization**

Mutual exclusion & synchronization — semaphores

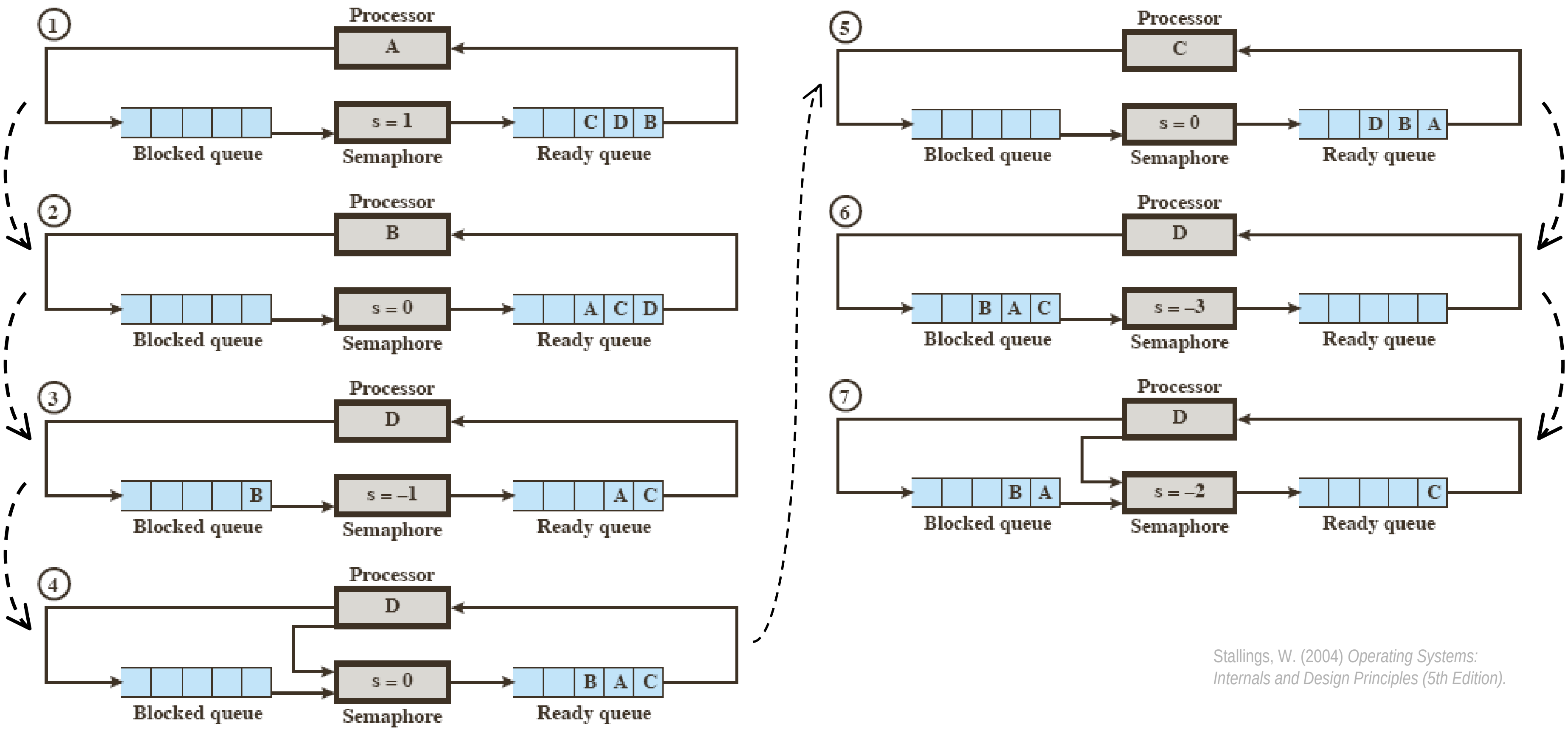
- Integer semaphore ⇔ "thermometer"



2.c Concurrency

Mutual exclusion & synchronization — semaphores

- All semaphores maintain a queue of waiting processes



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

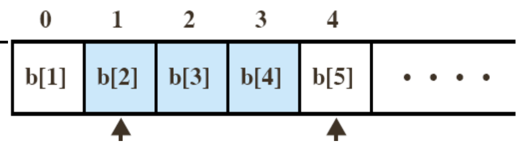
Example of semaphore mechanism

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- **Producer/consumer with an integer semaphore**
  - ✓ no need for a condition: the semaphore itself keeps track of the size of the buffer

```
item[] b;
int in, out;
```



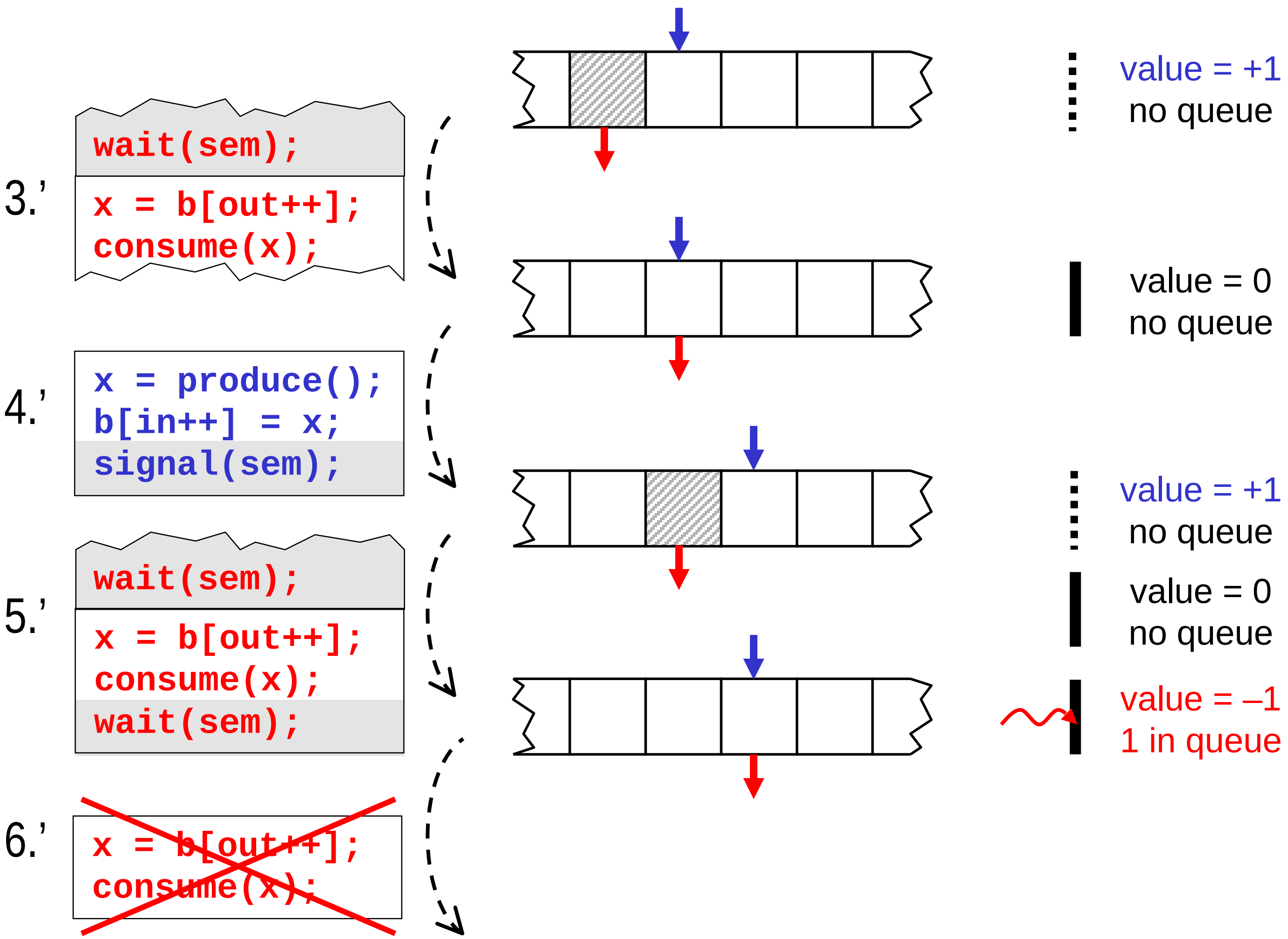

```
semaphore sem = 0;
```

```
void producer()
{
  while (true) {
    item = produce();
    lock(buf_mutex);
    b[in] = item;
    in++;
    ~~if (out == in-1)~~
      signal(sem);
    unlock(buf_mutex);
  }
}
```

correct!

```
void consumer()
{
  while (true) {
    ~~if (out == in0)~~
      wait(sem);
    lock(buf_mutex);
    item = b[out];
    out++;
    unlock(buf_mutex);
    consume(item);
  }
}
```

2.c Concurrency
Mutual exclusion & synchronization — semaphores



the consumer is blocked, as it should be; the producer may proceed . . .

2.c Concurrency

Mutual exclusion & synchronization — semaphores

➢ How semaphores may be implemented

```
semWait(s)
{
   while (!testset(s.flag))
      /* do nothing */;
   s.count--;
   if (s.count < 0)
   {
      place this process in s.queue;
      block this process (must also set s.flag to 0)
   }

      s.flag = 0;
}

semSignal(s)
{
   while (!testset(s.flag))
        /* do nothing */;
   s.count++;
   if (s.count <= 0)
   {
      remove a process P from s.queue;
      place process P on ready list
   }
   s.flag = 0;
}
```

(a) Testset Instruction

```
semWait(s)
{
      inhibit interrupts;
      s.count--;
      if (s.count < 0)
      {
            place this process in s.queue;
            block this process and allow interrupts
      }
      else
         allow interrupts;
}

semSignal(s)
{
      inhibit interrupts;
      s.count++;
      if (s.count <= 0)
      {
            remove a process P from s.queue;
            place process P on ready list
      }
      allow interrupts;
}
```

(b) Interrupts

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Two possible implementations of semaphores

2.c Concurrency

Mutual exclusion & synchronization — semaphores

- Bounded buffer, 1 producer, 1 consumer with sync

2.c Concurrency

Mutual exclusion & synchronization — monitors

- A monitor is a language-level encapsulation construct

2.c Concurrency

Mutual exclusion & synchronization — monitors

- Producer/consumer problem with monitors

2.c Concurrency

Mutual exclusion & synchronization — message passing

- Message passing: senders, receivers and mailboxes

2.c Concurrency

Mutual exclusion & synchronization — message passing

- Producer/consumer problem with message passing

Principles of Operating Systems

CS 446/646

2. Processes

a. Process Description & Control

b. Threads

c. **Concurrency**

- ✓ Types of process interaction
- ✓ Race conditions & critical regions
- ✓ Mutual exclusion by busy waiting
- ✓ Mutual exclusion & synchronization
  - mutexes
  - semaphores
  - monitors
  - message passing

d. **Deadlocks**

Principles of Operating Systems
CS 446/646

2. Processes

a. Process Description & Control

b. Threads

c. Concurrency

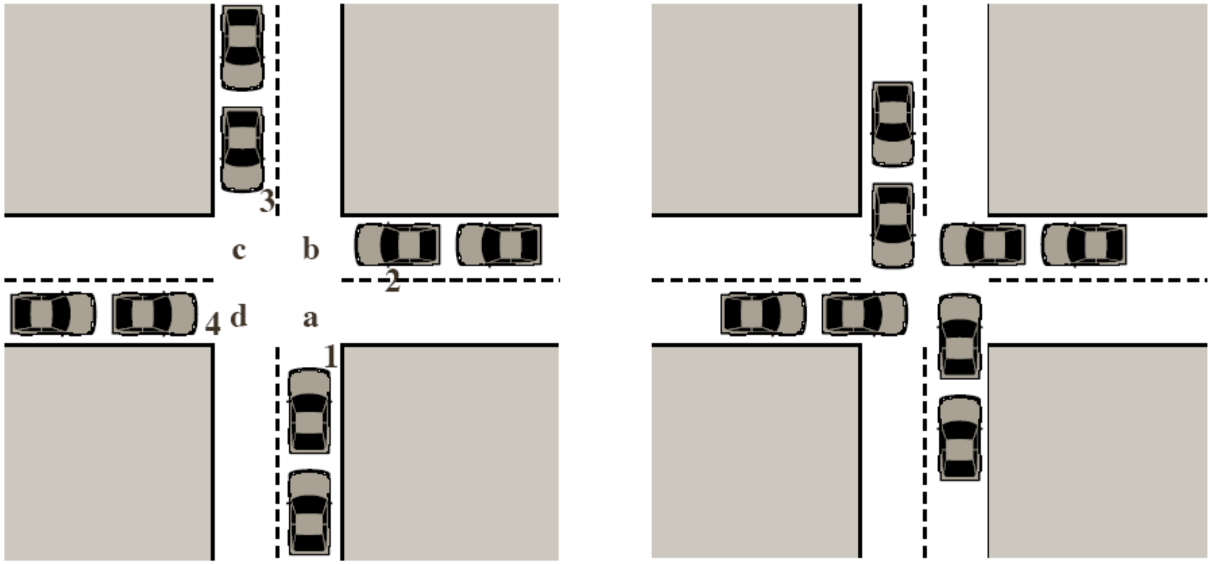
d. **Deadlocks**

- ✓ Deadlock principles: diagrams and graphs
- ✓ Deadlock prevention: changing the rules
- ✓ Deadlock avoidance: optimizing the allocation
- ✓ Deadlock detection: recovering after the facts

2.d Deadlocks

Deadlock principles: diagrams and graphs

- A deadlock is a <u>permanent blocking</u> of a set of threads
  - ✓ a deadlock can happen while threads/processes are competing for system resources or communicating with each other
  - ✓ there is no universal efficient solution against deadlocks



(a) Deadlock possible

(b) Deadlock

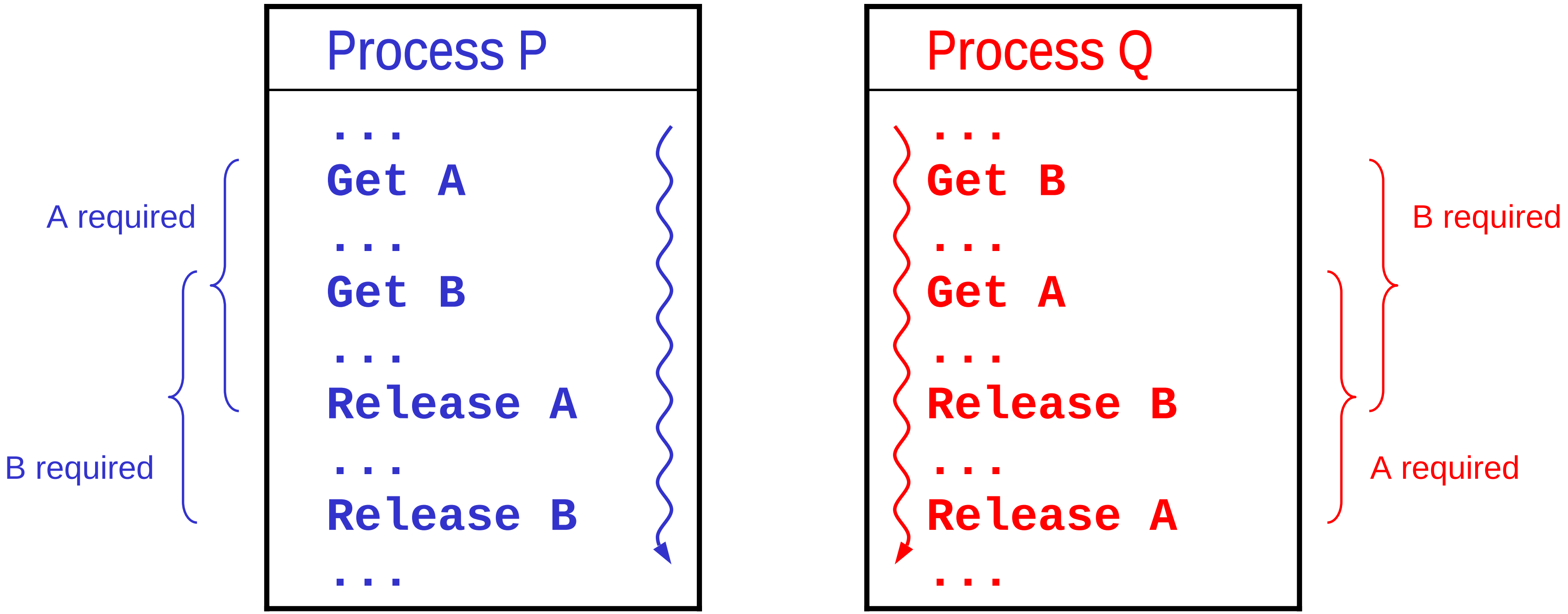
Tanenbaum, A. S. (2001) *Modern Operating Systems (2nd Edition).*

Illustration of a deadlock

2.d Deadlocks

Deadlock principles: diagrams and graphs

- Illustration of a deadlock
  - ✓ two processes, P and Q, compete for two resources, A and B
  - ✓ each process needs exclusive use of each resource

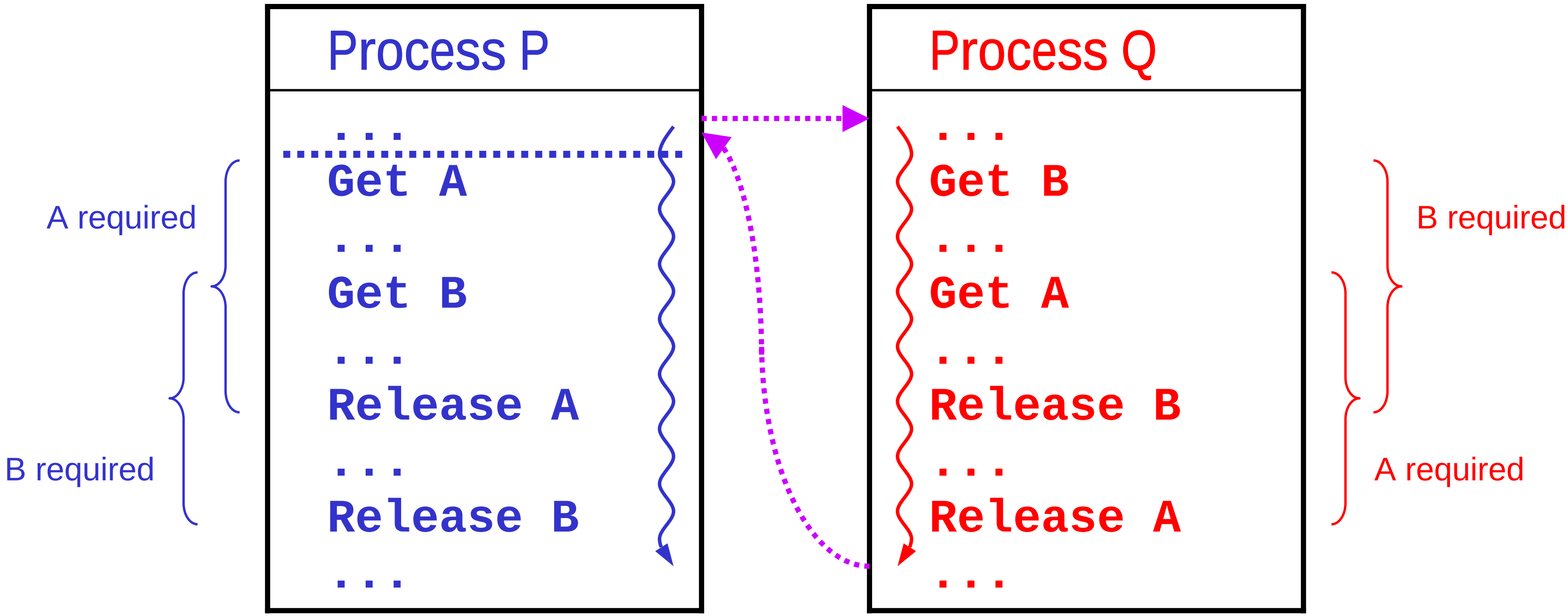


Competing processes

2.d Deadlocks

Deadlock principles: diagrams and graphs

- Illustration of a deadlock — scheduling path 1 ☺
  - ✓ Q executes everything before P can ever get A
  - ✓ when P is ready, resources A and B are free and P can proceed

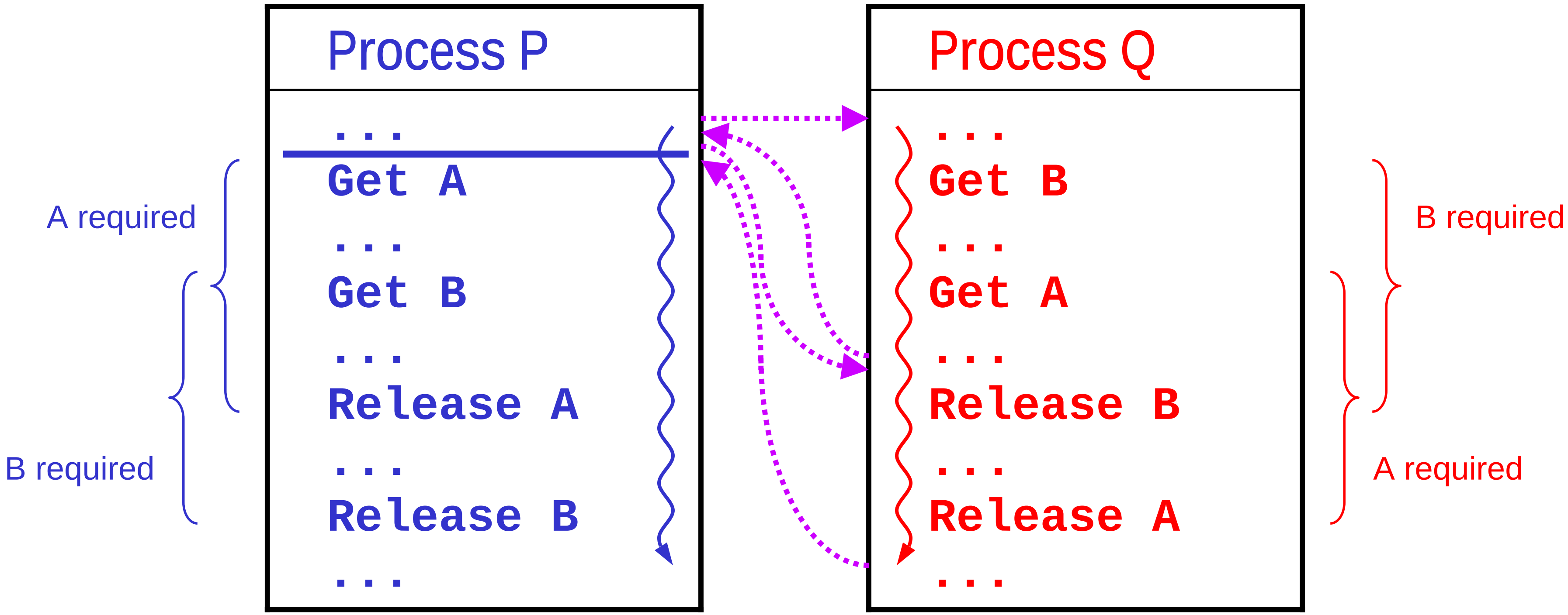


Happy scheduling 1

2.d Deadlocks

Deadlock principles: diagrams and graphs

- Illustration of a deadlock — scheduling path 2 ☺
  - ✓ Q gets B and A, then P is scheduled; P wants A but is blocked by A's mutex; so Q resumes and releases B and A; P can now go

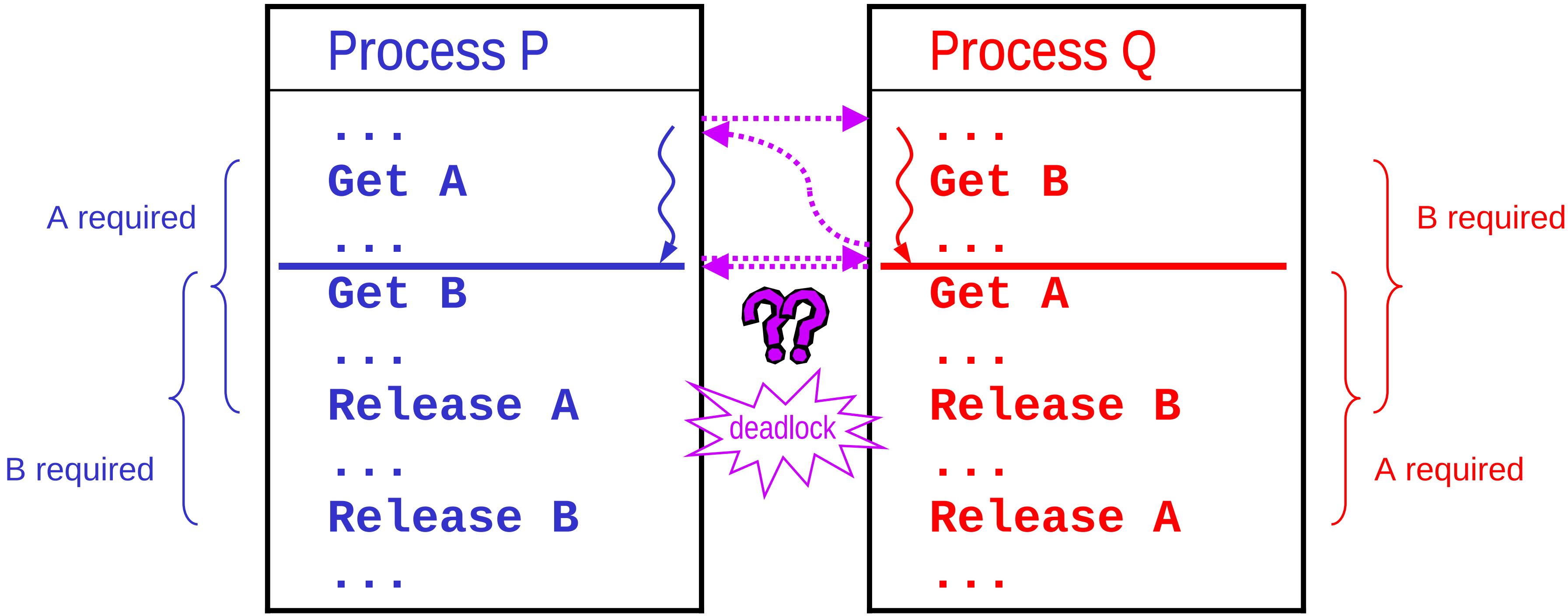


Happy scheduling 2

2.d Deadlocks

Deadlock principles: diagrams and graphs

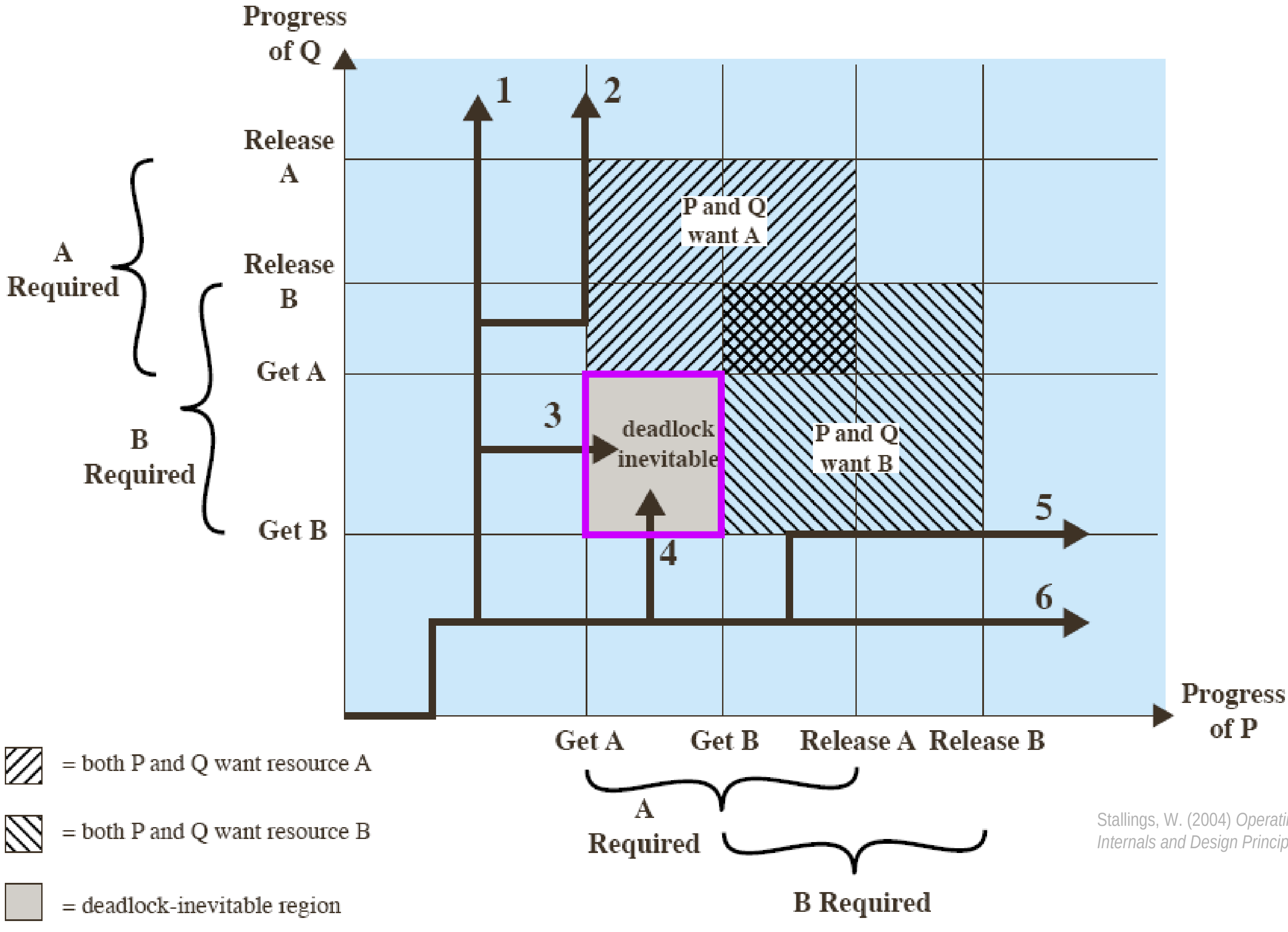
- Illustration of a deadlock — scheduling path 3 ☹
  - ✓ Q gets only B, then P is scheduled and gets A; now both P and Q are blocked, each waiting for the other to release a resource



Bad scheduling → deadlock

2.d Deadlocks

Deadlock principles: diagrams and graphs



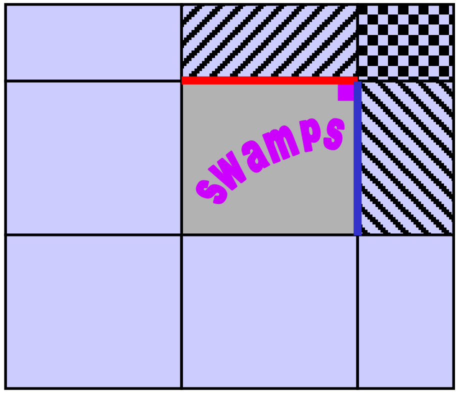
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Joint progress diagram

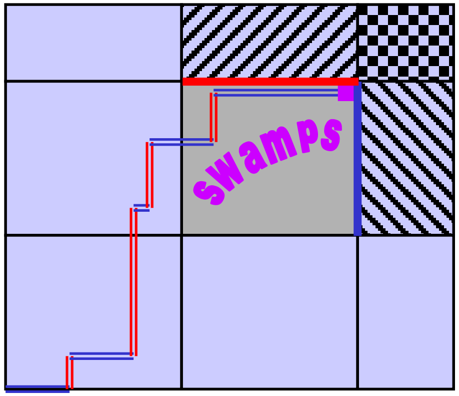
2.d Deadlocks

Deadlock principles: diagrams and graphs

- Deadlocks depend on the program and the scheduling
  - ✓ program design
    - the order of the statements in the code creates the "landscape" of the joint progress diagram
    - this landscape may contain gray "swamp" areas leading to deadlock



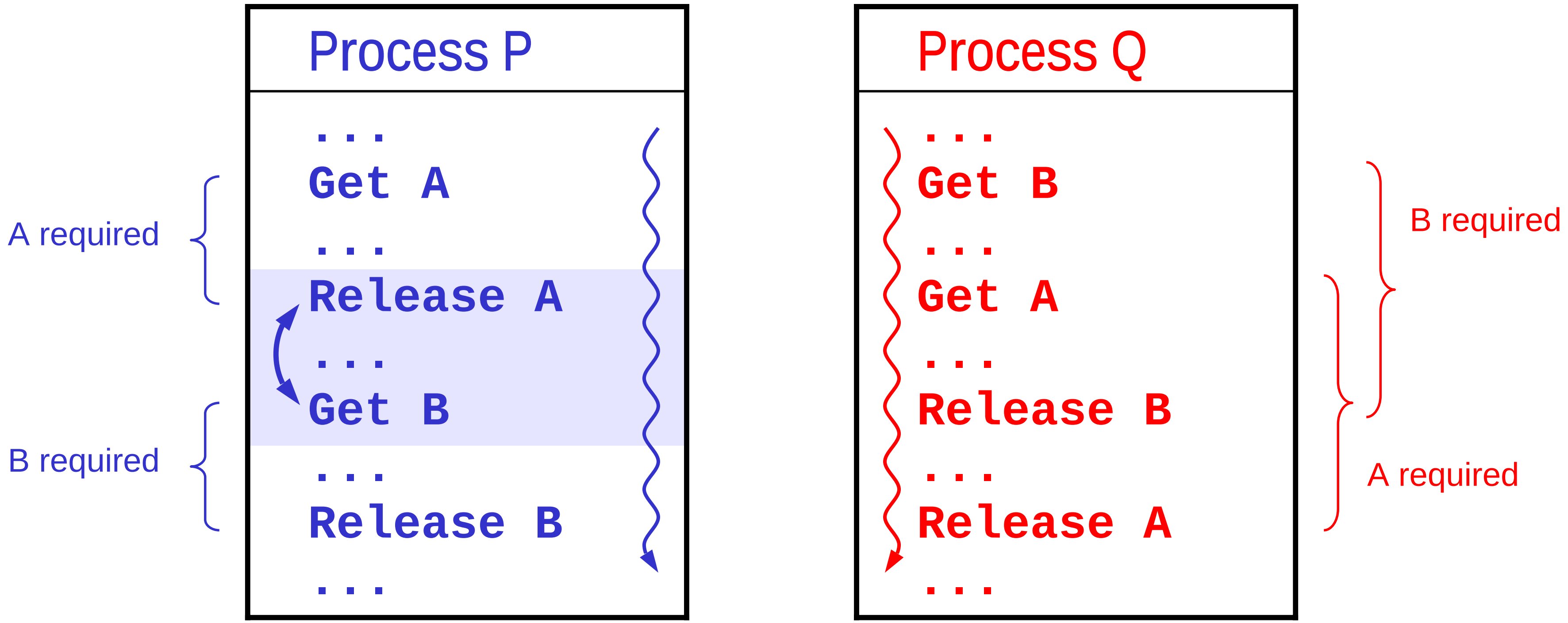
  - ✓ scheduling condition
    - the interleaved dynamics of multiple executions traces a "path" in this landscape
    - this path may sink in the swamps



2.d Deadlocks

Deadlock principles: diagrams and graphs

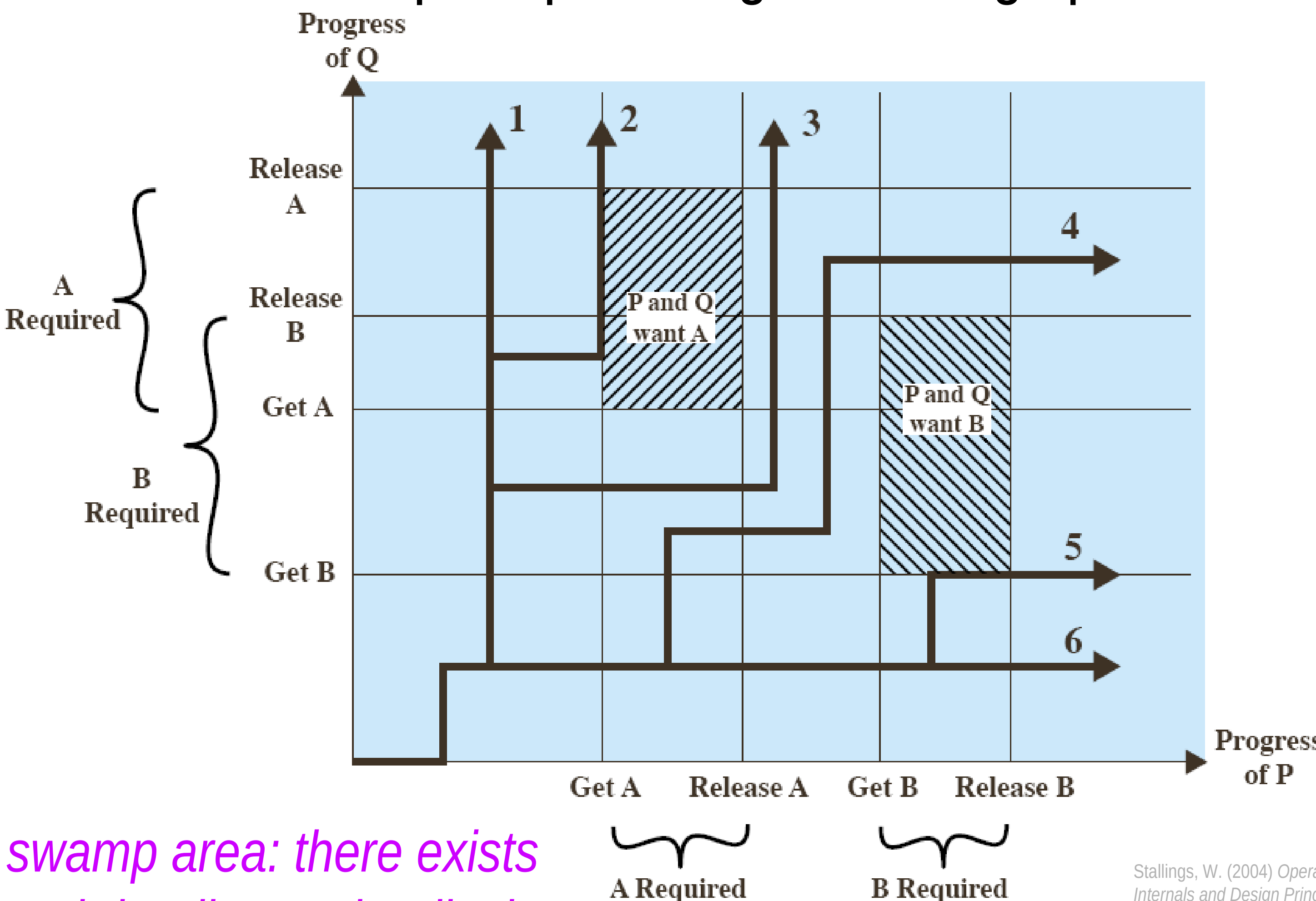
- Changing the program changes the landscape
  - ✓ here, P releases A before getting B
  - ✓ deadlocks between P and Q are not possible anymore



Competing processes

2.d Deadlocks

Deadlock principles: diagrams and graphs



→ *no swamp area: there exists no path leading to deadlock*

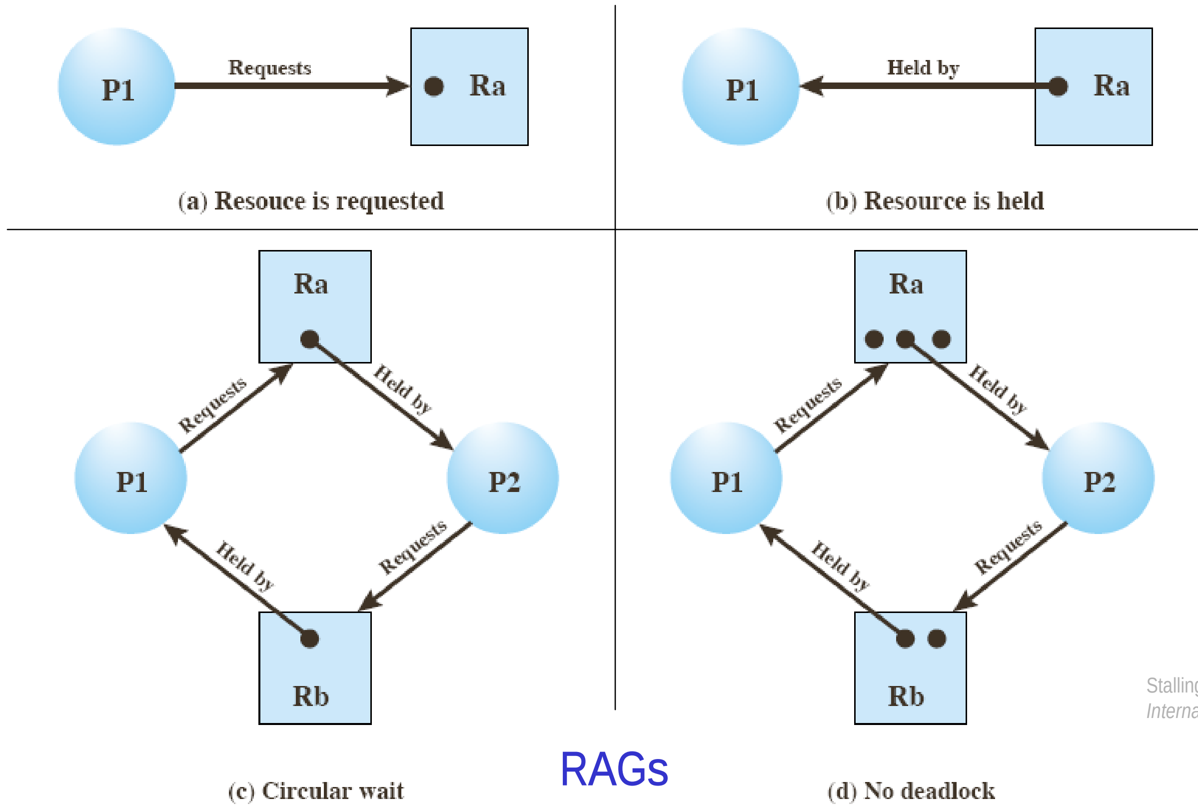
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Joint progress diagram

2.d Deadlocks

Deadlock principles: diagrams and graphs

- Snapshot of concurrency: Resource Allocation Graph
  - ✓ a resource allocation graph is a directed graph that depicts a state of the system of resources and processes



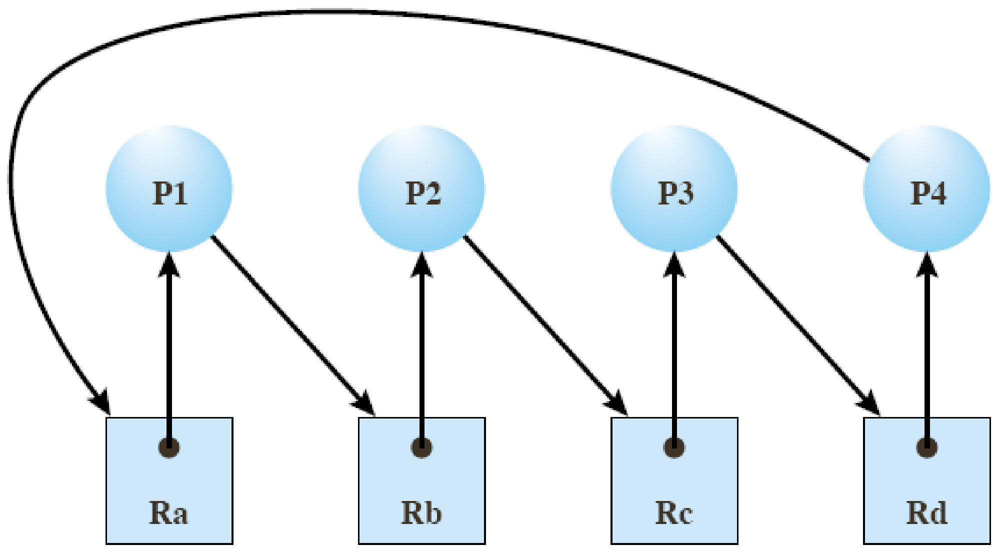
RAGs

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

2.d Deadlocks

Deadlock principles: diagrams and graphs

- Resource allocation graphs & deadlocks
  - ✓ there is deadlock when a closed chain of processes exists
  - ✓ each process holds at least one resource needed by the next process



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

A deadlock's RAG

2.d Deadlocks

Deadlock principles: diagrams and graphs

➢ Design conditions for deadlock (create the swamps)

1. **mutual exclusion** — the design contains protected critical regions; only one process at a time may use these
2. **hold & wait** — the design is such that, <u>while</u> inside a critical region, a process may have to wait for <u>another</u> critical region
3. **no resource preemption** — there must not be any hardware or O/S mechanism forcibly removing a process from its CR

\+ Scheduling condition for deadlock (go to the swamps)

4. **circular wait** — two or more hold-&-wait's are happening in a circle: each process holds a resource needed by the next

= Deadlock!

2.d Deadlocks

Deadlock principles: diagrams and graphs

- Three strategies for dealing with deadlocks
  - ✓ **deadlock prevention** — changing the rules
    - one or several of the deadlock conditions 1., 2., 3. or 4. are removed *a priori* (design decision)
  - ✓ **deadlock avoidance** — optimizing the allocation
    - deadlock conditions 1., 2., 3. are maintained but resource allocation follows extra cautionary rules (runtime decision)
  - ✓ **deadlock detection** — recovering after the facts
    - no precautions are taken to avoid deadlocks, but the system cleans them periodically (“deadlock collector”)

2.d Deadlocks

Deadlock prevention: changing the rules

- Remove one of the design or scheduling conditions?
  - ✓ remove “mutual exclusion”?
    - → *not possible: must always be supported by the O/S*
  - ✓ remove “hold & wait”?
    - ▪ require that a process gets all its resources at one time
    - → *inefficient and impractical: defeats interleaving, creates long waits, cannot predict all resource needs*
  - ✓ remove “no preemption” = allow preemption?
    - ▪ require that a process releases and requests again → *ok*
  - ✓ remove “circular wait”?
    - ▪ ex: impose an ordering of resources → *inefficient, again*

2.d Deadlocks

Deadlock avoidance: optimizing the allocation

- Allow all conditions, but allocate wisely
  - ✓ given a resource allocation request, a decision is made dynamically whether granting this request can potentially lead to a deadlock or not
    - do not start a process if its demands might lead to deadlock
    - do not grant an incremental resource request to a running process if this allocation might lead to deadlock
  - ✓ avoidance strategies requires knowledge of future process request (calculating "chess moves" ahead)

2.d Deadlocks

Deadlock avoidance: optimizing the allocation

- Resource allocation denial: the "banker's algorithm"
  - ✓ at any time, the state of the system is the current allocation of multiple resources to multiple processes
    - a safe state is where there is at least one sequence that does not result in deadlock
    - an unsafe state is a state where there is no such sequence
  - ✓ analogy = banker refusing to grant a loan if funds are too low to grant more loans + uncertainty about how long a customer will repay

2.d Deadlocks

Deadlock avoidance: optimizing the allocation

- **Resource allocation denial: the “banker's algorithm”**
  - ✓ can a process run to completion with the available resources?

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

compare what is still needed with what is left

(a) **Initial state**

(a)

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
6	2	3

Available vector V

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

(b) **P2 runs to completion**

(b)

Determination of a safe state

2.d Deadlocks

Deadlock avoidance: optimizing the allocation

- Resource allocation denial: the “banker's algorithm”
  - ✓ idea: refuse to allocate if it may result in deadlock

(c)

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

(c) P1 runs to completion

(d)

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

all could run to completion:
→ thus, (a) was a safe state

(d) P3 runs to completion

Determination of a safe state (cont'd)

2.d Deadlocks

Deadlock avoidance: optimizing the allocation

- ➢ Resource allocation denial: the "banker's algorithm"
  - ✓ idea: refuse to allocate if it may result in deadlock

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

(a) Initial state

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

(a) safe ← (a')

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

(b) P1 requests one unit each of R1 and R3

(b') unsafe

potential for deadlock (we don't know how long Ri will be kept) → thus, (b') is an unsafe state: don't allow (b') to happen

Determination of an unsafe state

2.d Deadlocks

Deadlock detection: recovering after the facts

Principles of Operating Systems

CS 446/646

2. Processes

a. Process Description & Control

b. Threads

c. Concurrency

d. **Deadlocks**

- ✓ Deadlock principles: diagrams and graphs
- ✓ Deadlock prevention: changing the rules
- ✓ Deadlock avoidance: optimizing the allocation
- ✓ Deadlock detection: recovering after the facts

Principles of Operating Systems

CS 446/646

2. Processes

a. Process Description & Control

b. Threads

c. Concurrency

d. Deadlocks

Principles of Operating Systems
CS 446/646

0. Course Presentation
1. Introduction to Operating Systems
2. Processes
3. Memory Management
4. CPU Scheduling
5. Input/Output
6. File System
7. Case Studies