2.c Concurrency

Mutual exclusion by busy waiting

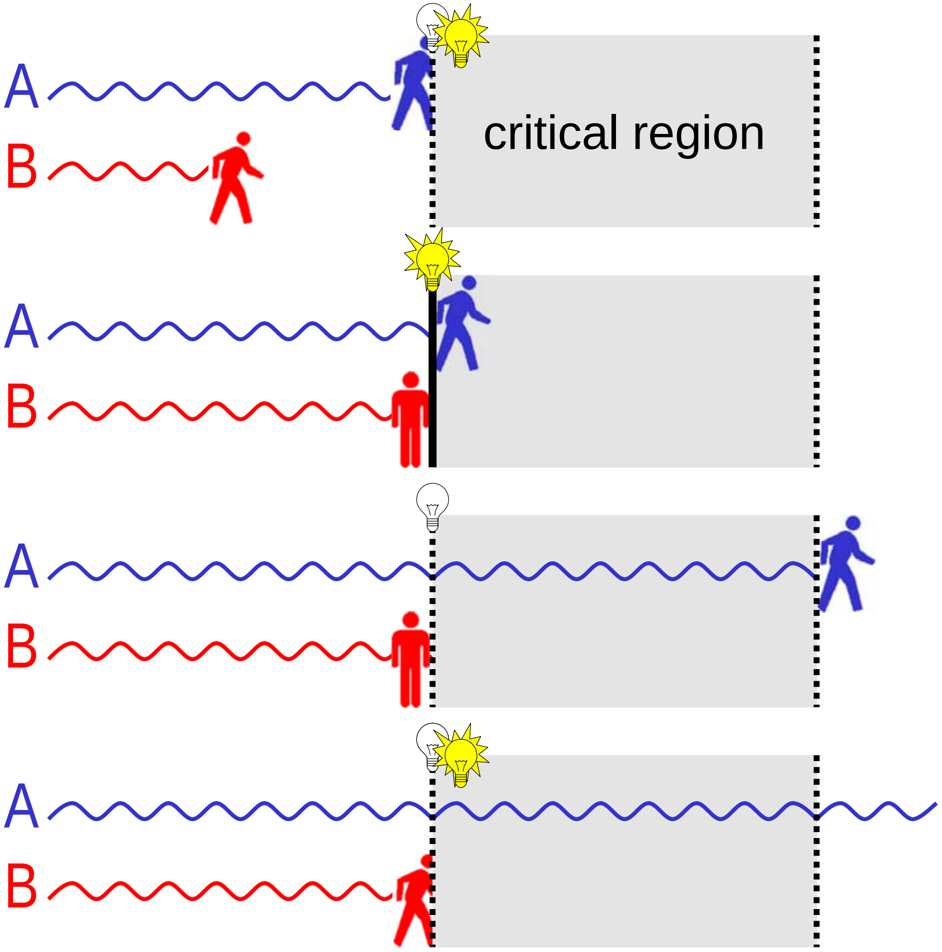
- Implementation 2 — "indivisible" lock variable

1. thread A reaches CR and <u>finds the lock at 0 *and* sets it in one shot</u>, 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

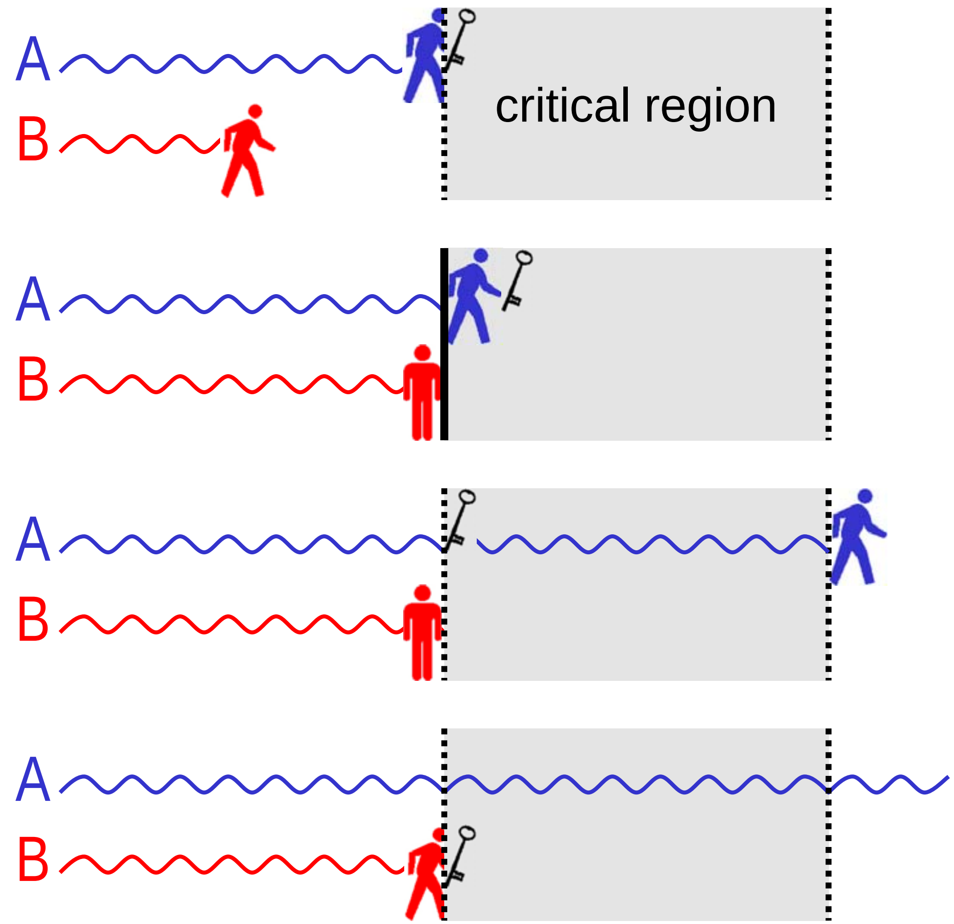
- Implementation 2 — “indivisible” lock ⇔ one key 👍

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

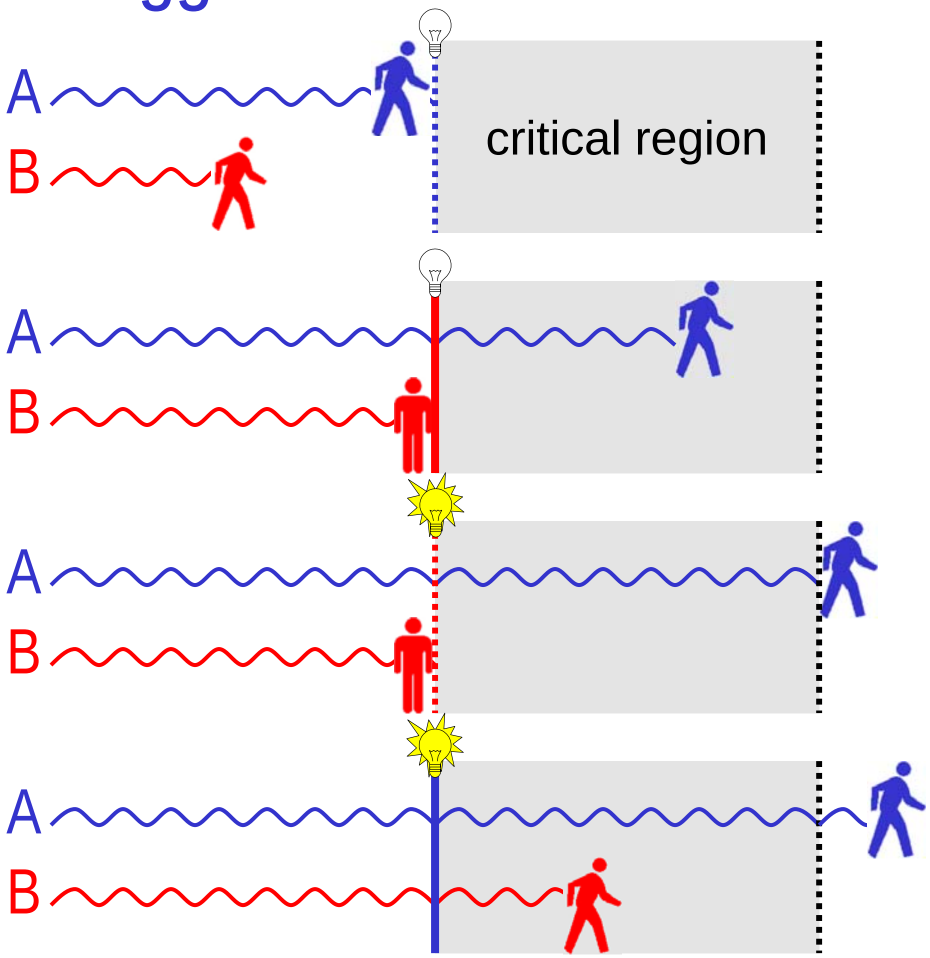
```
void echo()
{
   char chin, chout;
   do {
      take key and run
      chin = getchar();
      chout = chin;
      putchar(chout);
      return key
   }
   while (...);
}
```

2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 3 — no-TSL 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 — no-TSL 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

```
while (toggle);
   /* loop */

toggle = TRUE;
```

B's code

```
while (!toggle);
   /* loop */

toggle = FALSE;
```

2.c Concurrency

Mutual exclusion by busy waiting

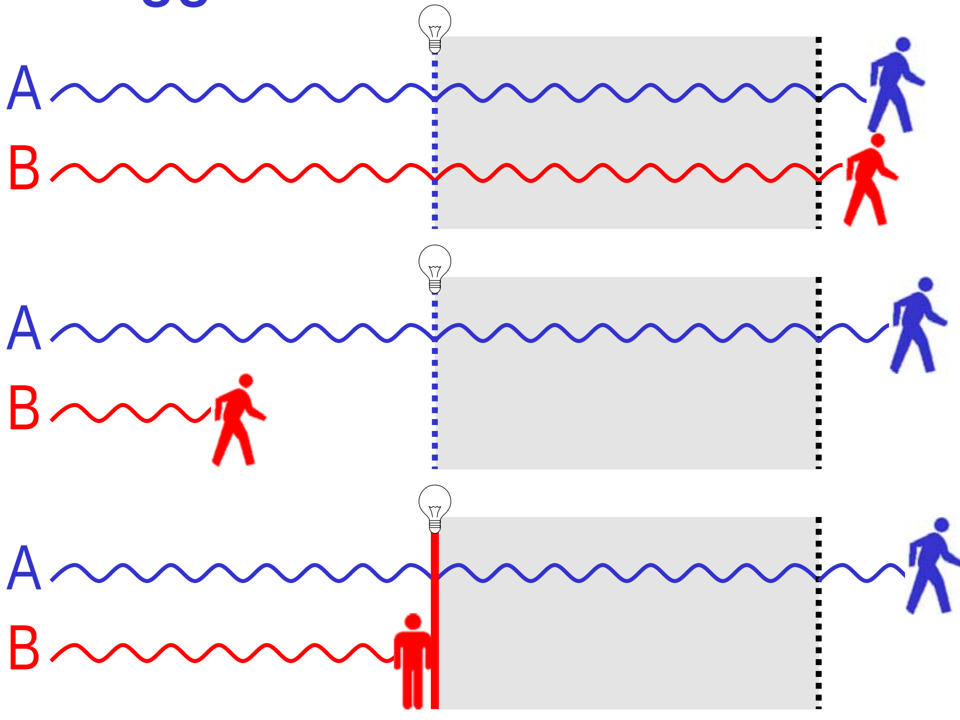
- Implementation 3 — ~~no-TSL toggle for two threads~~

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 or interrupted in its noncritical region, B is barred access to its CR

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

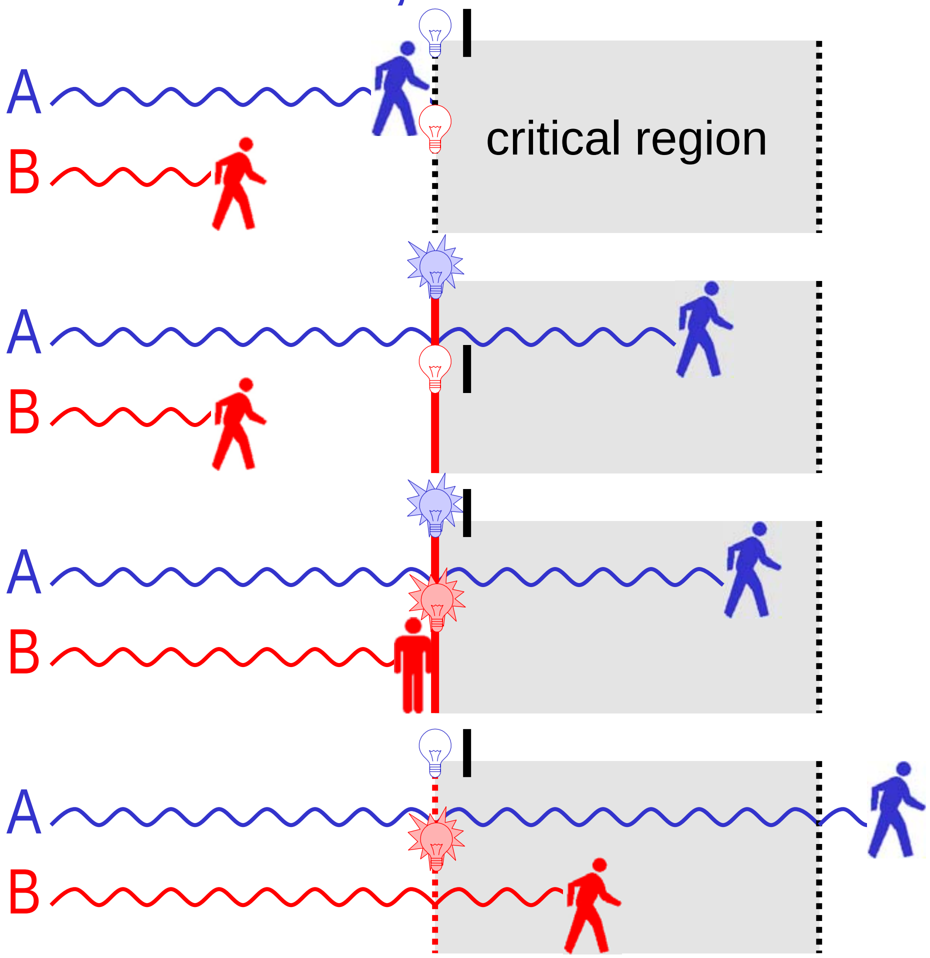


→ *this implementation avoids TSL by splitting test & set and putting them in enter & exit; nice try... but flawed!*

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

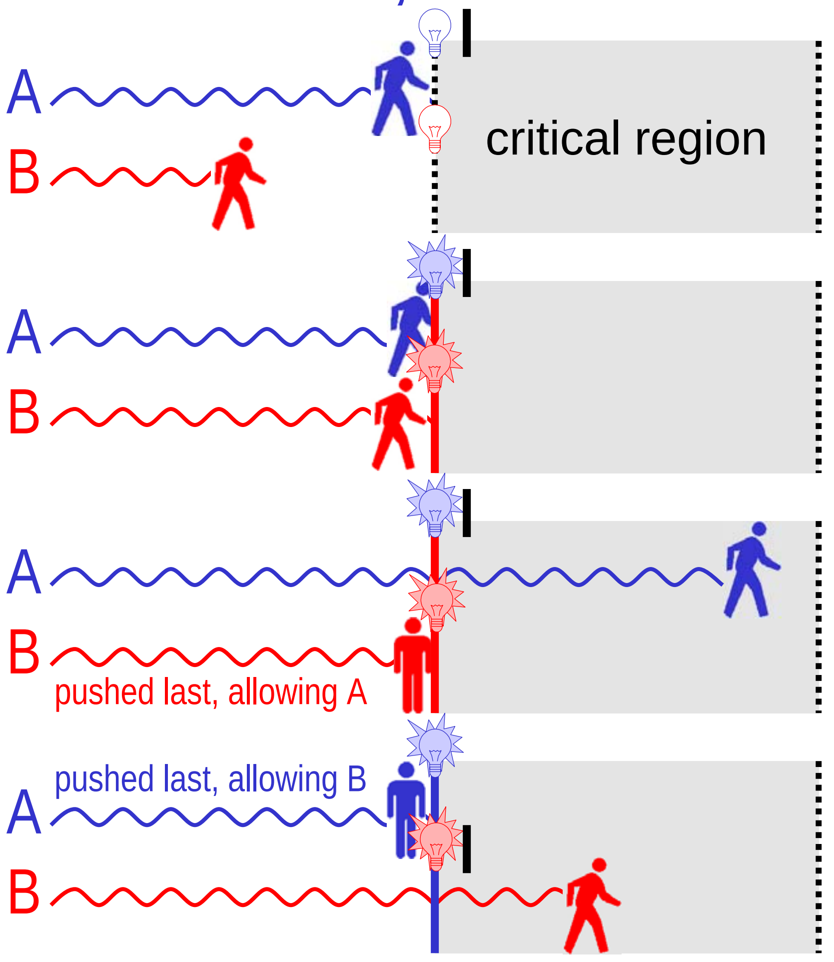
- Implementation 4 — Peterson's no-TSL, no-alternation

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 last will allow the other 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 — no-TSL 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
```

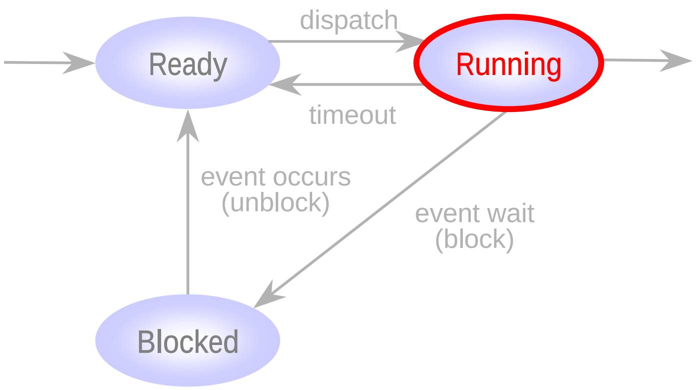
```
void echo()
{
   char chin, chout;
   do {
      test-and-set-lock  or BLOCK
      chin = getchar();
      chout = chin;
      putchar(chout);
      set lock off
   }
   while (...);
}
```

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

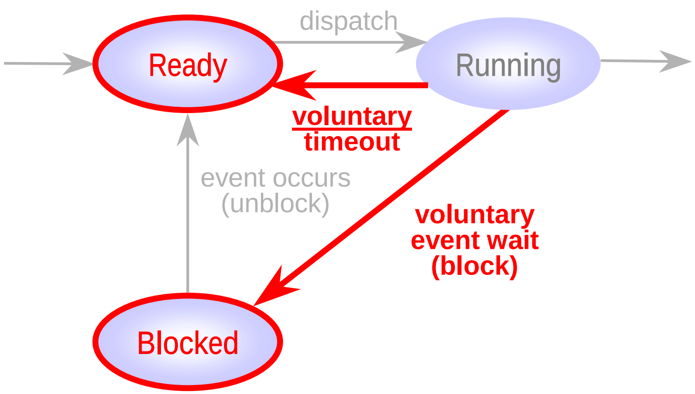
2.c Concurrency

Mutual exclusion & synchronization — mutexes

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



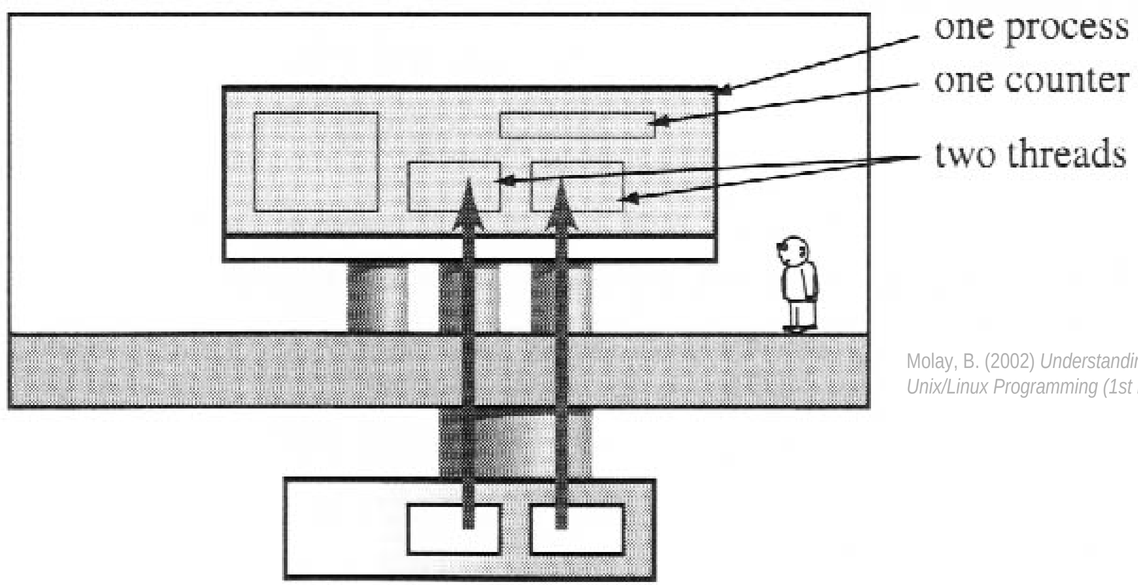
  - ✓ when <u>blocked</u>, the process's PC stalls after executing a "yield" call
  - ✓ either the process is only timed out, thus it is "Ready" to loop-and-yield again → *sparse polling*
  - ✓ or it is truly "Blocked" and put in event queue → *condition waiting*



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++;
        }
        ......
```

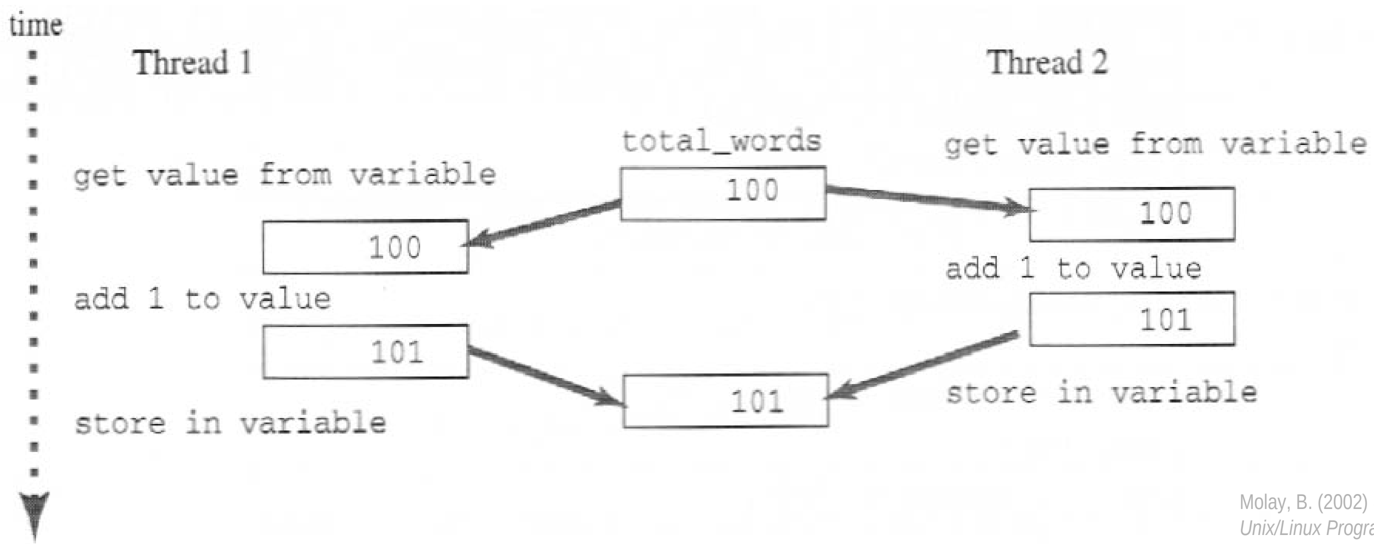
`total_words = total_words + 1;` *is not necessarily atomic! (depends on machine code and stage of execution)*

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)`

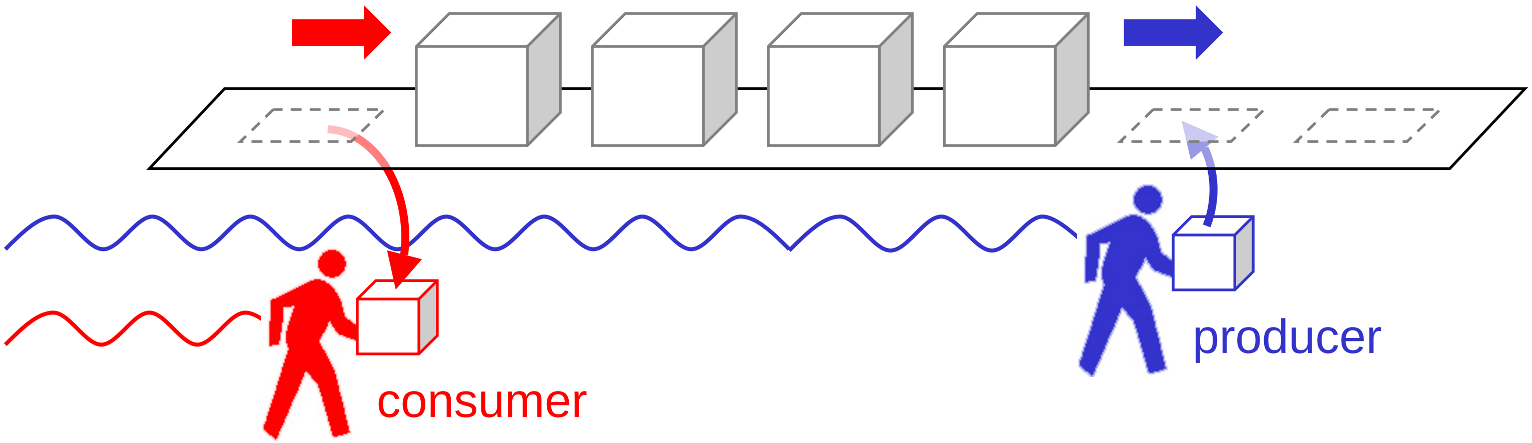
    locks the specified mutex
    - if the mutex is unlocked, it becomes locked and owned by the calling thread
    - if the mutex is already locked by another thread, the calling thread is blocked until the mutex is unlocked
  - ✓ `err = pthread_mutex_unlock(pthread_mutex_t *m)`

    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

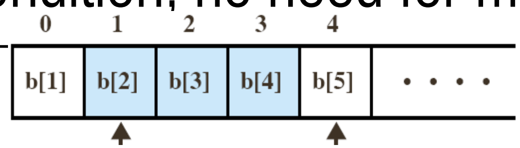


2.c Concurrency

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);
  }
}
```