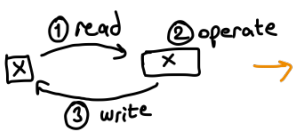
synchronization

shared resources = global variables, heap → dynamically-allocated data, (stack is local, not shared)

access controll mechanisms = low level: locks

↳ high level: mutexes, semaphores, monitors, condition variables

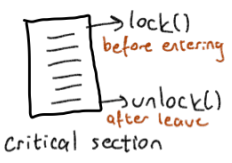


→ if one thread/process reads before other one writes → race condition (can happen only in writing mode)

- the result is non-deterministic

critical section = portion of the code that accesses the shared variables

locks



mutual exclusion = only one thread can execute code in their critical section one at a time

↳ all others are forced to wait, when one leaves other can enter

- shared (global) lock variable = 0 or 1

```
int withdraw(account, amount) {
  lock(lockvar);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  unlock(lockvar);
  return balance;
}
```
(balance = get_balance … put_balance(account, balance) : critical section)

spinlocking = check lock varible if it is 1 → loop to wait (while(lock_var))

atomic = means the code cannot be interrupted during execution. (lock() and unlock() must be atomic)

```
struct lock_type {
  int held = 0;
} lockvar;
void lock(lockvar) {
   while (lockvar->held);
   lockvar->held = 1;
}
void unlock(lockvar) {
  lockvar->held = 0;
}
```
wasteful! ↑ The caller busy waits for the lock to be released

→ if there is a context switch in here, both try to get in → race condition again

atomic lock/unlock =

↳ hardware solution: → disable interrupts = whole process cannot receive necessary interrupts

- even noncritical section threads are locked
- only implemented at kernel level and inefficient on a multiprocessor

↳ atomic update instruction = cmpxchg src, dest — compare and exchange

- large performance penalty

↳ software solution:

```
int flag[2] = {0,0};
int turn = 0;

void plock(int id) {
   int other = !id;
   flag[id] = 1;
   turn = other;
   while (flag[other] && turn == other) {
   };
}

void punlock(int id) {
   flag[id] = 0;
}
```

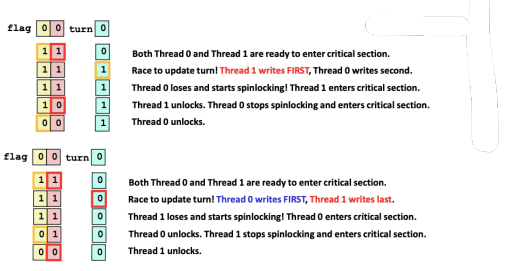
Peterson's Algorithm

turn : indicates whose turn is it to enter critical section.
If turn==i thread Ti is allowed to get in.
flag[2]: indicates if thread Ti is ready to enter critical section.
If flag[i] is set, then Ti is ready to enter critical section.

```
int flag[2] = {0,0};
int turn = 0;
```

Thread 0	Thread 1

void plock(0) {
  int other = 1;
  flag[0] = 1;
  turn = 1;
  while (flag[1] && turn == 1) {
  };
}
 	
void punlock(0) {
  flag[0] = 0;
}
	void plock(1) {
	  int other = 0;
	  flag[1] = 1;
	  turn = 0;
	  while (flag[0] && turn == 0) {
	  };
	}
	
	void punlock(1) {
	  flag[1] = 0;
	}



- works completely fine, but slow and need to know how many threads (→ no priority to any of threads)

Spinlocks problem ⇒ threads waiting for lock waste CPU, there is still context swich to them, but they do not execute their code just looping.

mutexes blocking locks

- inserting spinlocks threads into a wait queue
- do not context switch them unless critical section is unlocked
- wake one thread from wait queue

* frees up CPU for other threads

limitations ⇒ limiting writings but not limiting readings cannot be implemented

↳ exchanging data between two processes

Semaphores to allow one thread to write or any number of threads to read at a time

semaphore = shared counter that initialized at the beginning (init(sem, n))

↳ cannot be accessed directly

wait() - down() - p() = wait for semaphore value to become >0, then decrement it

signal() - up() - v() = increment semaphore value by 1 → always executed no waits

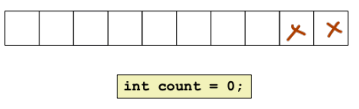
semaphore value 1 → thread 1 wait() (↳ starts to execution) → semaphore value 0 → thread 2 wait() (↳ put in wait queue) → semaphore value -1 → thread 3 signal() (↳ completes the execution) → thread 2 (↳ starts to execution) → semaphore value 0

```
semaphore my_semaphore;
init(my_semaphore, 1);
int withdraw(account, amount) {
  wait(my_semaphore);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  signal(my_semaphore);
  return balance;
}
```
(balance = get_balance … put_balance(account, balance) : critical section)

```
struct semaphore {
  int val;
  threadlist L;  // List of threads waiting for semaphore
}

init(semaphore s, int initval):
  s.val= initval;

wait(semaphore s): //Wait until > 0 then decrement
    if (s.val <= 0) {
        add this thread to S.L;
        block(this thread);
    }
    s.val = s.val -1;
    return;

signal(semaphore s): //Increment and wake up next thread
  s.val = s.val + 1;
  if (s.L is nonempty) {
   remove a thread T from S.L;
   wakeup(T);
  }
```
(init(), wait() and signal() must be atomic actions!)

the producer/consumer problem =


producer




consumer

int count = 0;

```
Producer Thread
Producer() {
int item;
 while (TRUE) {
     item = produce();
     if (count == N)
         sleep();
     insertitem(item);
     count = count + 1;
     if (count == 1)
         wakeup(consumer);
 }
}
```
→ if context switch ↳ error

```
Consumer Thread
Consumer() {
int item;
     while (TRUE) {
     if (count == 0)
         sleep();
     item = removeitem();
     count = count - 1;
     if (count == N-1)
         wakeup(producer);
     eat(item);
 }
}
```

- pushes item, waits if full
- pulls item, waits if empty

⇒ using binary semaphore

if empty is 0, block itself

```
semaphore mutex; init(mutex,1);
semaphore full; init(full,0);   → # of items
// N: size of the buffer
semaphore empty; init(empty,N); → # of empty spaces
```

```
Producer Thread
Producer() {
 int item;
 while (TRUE) {
    item = produce();
    wait(empty);
    wait(mutex);
    insertitem(item);
    signal(mutex);
    signal(full);
 }
}
```
(must be >0, decrease empty; must be 1, decrease to 0; increase to 1; increase full)

```
Consumer Thread
Consumer() {
 int item;
 while (TRUE) {
    wait(full);
    wait(mutex);
    item = removeitem();
    signal(mutex);
    signal(empty);
    eat(item);
 }
}
```
(must be>0, decrease full; decrease to 0; increase to 1; increase empty)

* mutex must be 1 and empty/full must be available to execute code

readers/writers

```
semaphore mutex; init(mutex,1);
semaphore write; init(write,1);
int readcount = 0;
```

```
Writer Thread
Writer(){
    wait(write);
    do_write();
    signal(write);
}
```
only one writer will be available at a time

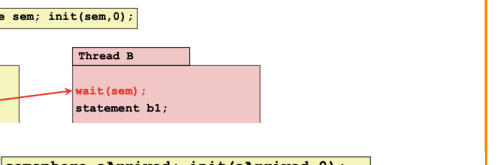
```
Reader Thread
Reader() {
    wait(mutex);
    readcount++;
    if (readcount == 1) {
        wait(write);
    }
    signal(mutex);
    do_read();
    wait(mutex);
    readcount--;
    if (readcount == 0) {
        signal(write);
    }
    signal(mutex);
}
```
(first one must lock the writers; lock mutex to prevent race condition among readers; last one must unlock the writers; if one at the end → only one reader at a time)

synchronization patterns

signalling = to guarantee that a1 is completed before b1 begins

rendezvous = same lines must be completed before next line

a1 must before b2, b1 must before a2



two As cannot arrive because first one waits B

barrier = generalized rendezvous, no thread executes critical point until after all the threads have executed rendezvous

```
n = thenumberofthreads;
count = 0;
semaphore mutex; init(mutex,1);
semaphore barrier; init(barrier,0);
```

```
All threads
rendezvous();

wait(mutex);
count = count + 1;
signal(mutex);
if (count == n) signal(barrier);
else{
    wait(barrier);
    signal(barrier);
}
criticalpoint();
```
→ can be used only once