Operating Systems (INFR09047)

2019/2020 Semester 2

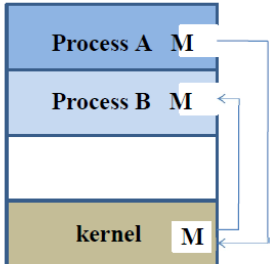
Synchronization

abarbala@inf.ed.ac.uk

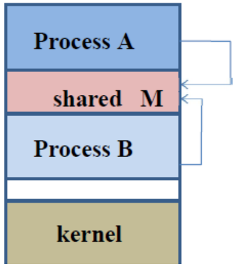
Chapter 6 (6.1 – 6.5)

Why Synchronization?

- **Cooperating tasks** access the same data
  - Two or more **threads** of the same process
    - Shared *data* and code
  - Two or more **processes**
    - *Data* on shared memory
    - ~~*Data* exchange via message passing~~



Message Passing



Shared Memory

- **Concurrent or parallel** access to shared data
  - **Concurrent** processes/threads *virtually* run *at the same time* on a core
    - Switch between them at any time – *scheduling*
  - **Parallel** process/threads run *at the same time* on different cores

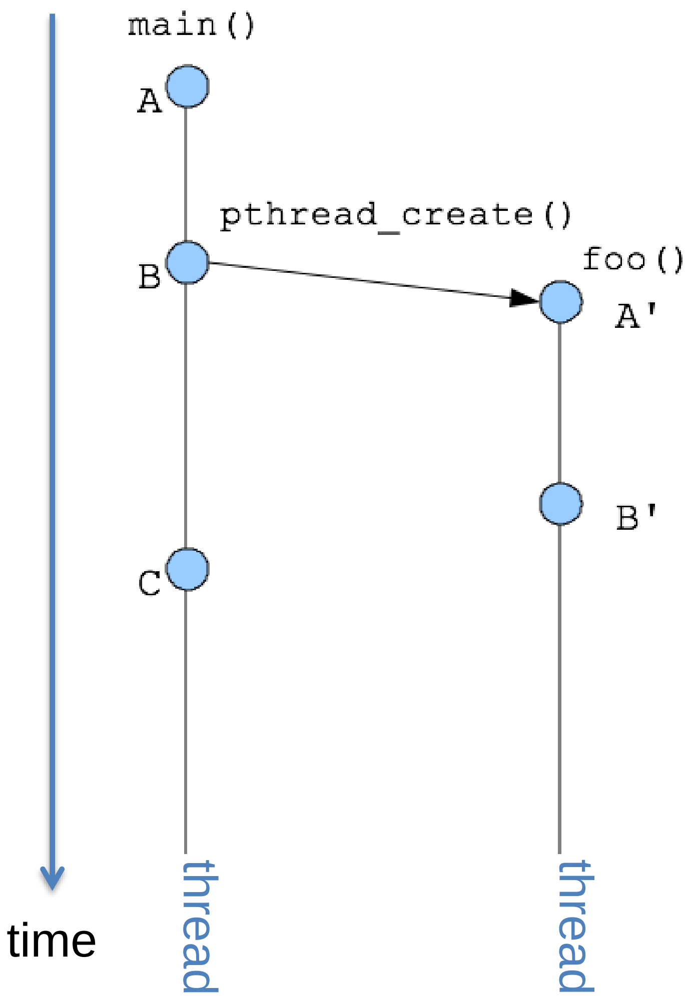
- May result in **data inconsistency** (integrity of data)
  - How to ensure **ordered** execution of cooperating tasks?
    - Synchronization

Ordered Execution: Temporal Relations

- Instructions executed by a single thread/process are **totally ordered**
  - A < B < C < …
  - A’ < B’ < …
  - X < Y means X event **happened before** Y event

- In **absence of synchronization**
  - Instructions executed by distinct threads/processes must be considered **unordered / simultaneous**
  - On **one** core, or **multiple** cores
  - Not X < X’, and not X’ < X
    - C == A’ or C == B’

- **Creation** relations always hold
  - But A < B < A’ < B’ < …



Example: Shared Bank Account #1

- A function to **withdraw money** from a bank account

```
int withdraw(account, amount) {
  int balance = get_balance(account);        // read
  balance -= amount;                         // modify
  put_balance(account, balance);             // write
  return balance
}
```

- You and your mother **share a bank account** with a balance of £1500.00
- What happens if you both go to **separate** ATM machines, and **simultaneously** withdraw £50.00 from the account?

Example: Shared Bank Account #2

- Bank's application is **multi-threaded**
- A separate **thread for each ATM** doing a withdrawal
  - Both threads run on the same bank server
- Each thread can **context switch** after each instruction

Thread YOU

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Thread MOTHER

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Example: Shared Bank Account #3

- **Interleaved** execution of the two threads

Execution sequence as seen by CPU

```
balance = get_balance(account);
balance -= amount;
```

MOTHER

context switch

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
return balance;
```

YOU

context switch

```
put_balance(account, balance);
return balance;
```

MOTHER

- What is the account balance after this sequence?
  - Who is happy, the bank or you and your mother?
  - How often is this sequence likely to occur?

Example: Shared Bank Account #4

- Which **interleavings** are OK?
- Which are not?

Thread YOU

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Thread MOTHER

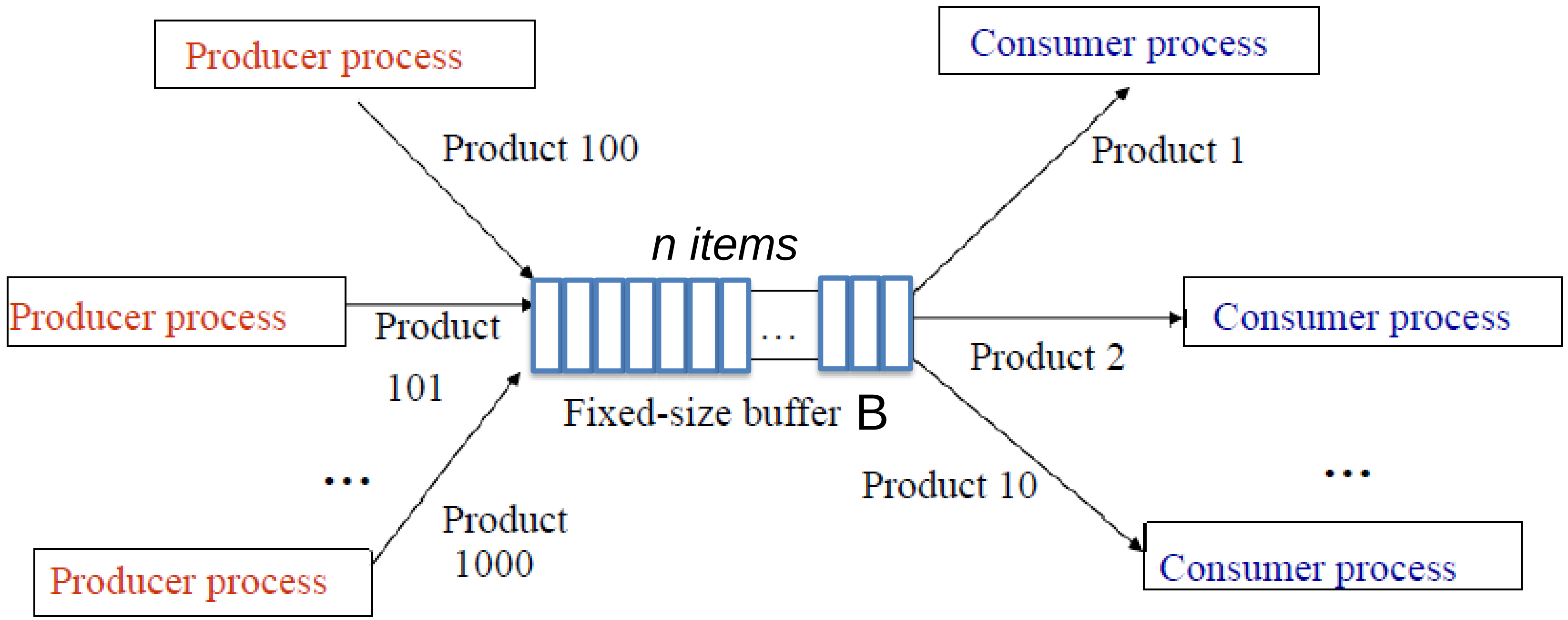
```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Example: Producer-consumer Problem #1

- Also known as the bounded-buffer problem
  - Fixed-size buffer ***B*** (with ***n*** elements)
  - ***p*** producer processes
  - ***c*** consumer processes
  - Producer and consumer processes share the buffer B
    - Producer process puts info into the buffer B
    - Consumer process takes info out of the buffer B

Barber shop analogy
n = number of waiting seats
p = all clients
c = barbers

Example: Producer-consumer Problem #2



p processes

c processes

Example: Producer-consumer Problem #3

- **Single-**producer, **single-**consumer problem (*p=1, c=1*)

```
Shared variable:
   const int n;
   int  count=0;
  Item buffer [n];
```

```
Producer
     while (1){
            …
            produce an  item A;
            …
            while(count==n);
            insert(item);
            count++;
}
```

```
Consumer
     while (1){
            while(count==0);
            item=remove_item();
            count--;
            …
            consume an item;
}
```

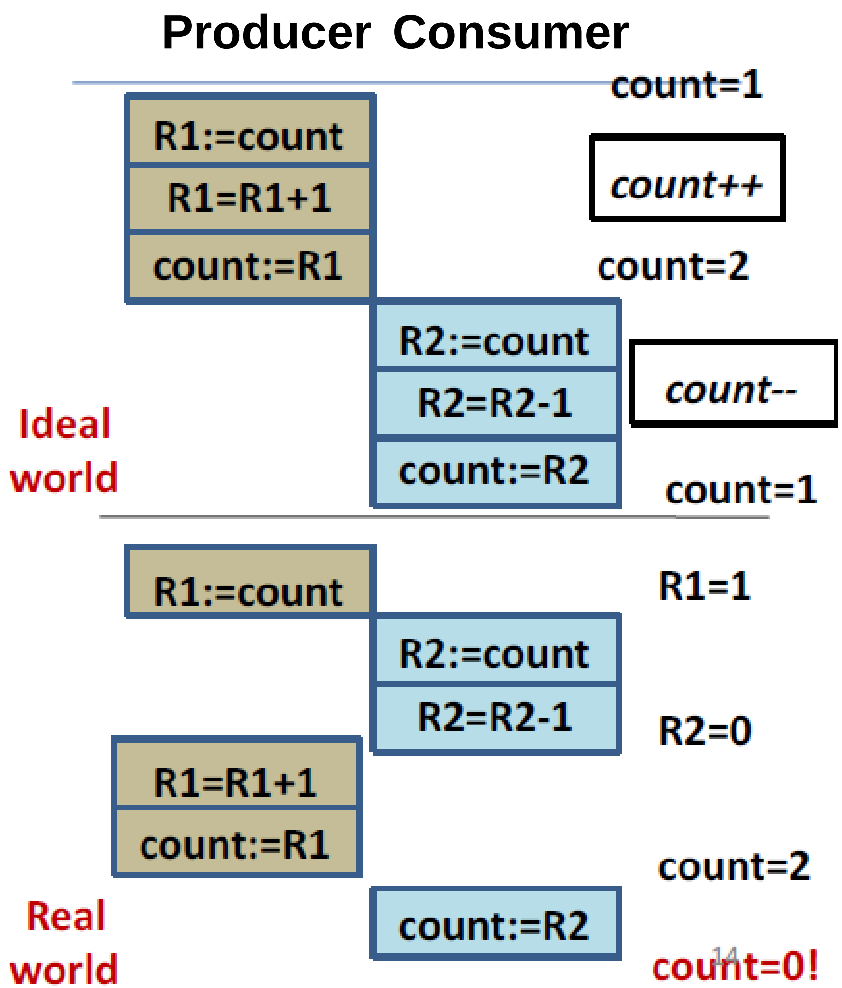
Example: Producer-consumer Problem #4

- `count++; count--;` instructions are **not guaranteed** to execute as a single machine instruction

- `count++;` a possible assembly implementation

```
R1 := count;
R1 = R1+1;
count := R1;
```

Note R1 is a machine register



Race Conditions

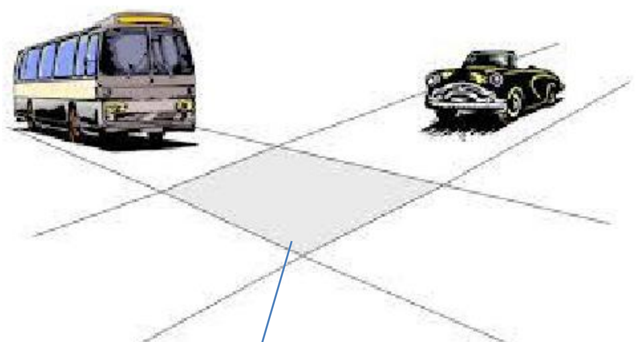
- Examples
  - Results of concurrent or parallel access is **non-deterministic**
  - **Depends on**
    - Timing
    - When context switches occurred
    - What process/thread ran at context switch

- **Race condition**
  - Two or more processes reading or writing shared data
  - The final result depends on who runs precisely when

- How **to avoid** race conditions?

Modeling Programs to Solve Race Conditions

- **Critical Region or Section**
  - Part of the program where the **shared data is accessed**
    - **Uncoordinated** read/write of the data in critical section may lead to races



Critical Region/Section

- **Common pattern** to identify
  - Read-modify-write of a shared data (variable)
    - Globals and heap-allocated variables
  - In code that can be executed by concurrent or parallel threads

Critical Region in Bank Example

Thread YOU

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

Thread MOTHER

```
int withdraw(account, amount) {
  int balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

From read to write the balance

Critical Region in Producer-consumer Example

Access to the count variable

```
Shared variable:
   const int n;
   int  count=0;
  Item buffer [n];
```

```
Producer
      while (1){
            ...
            produce an  item A;
            ...
            while(count==n);
            insert(item);
            count++;
}
```

```
Consumer
      while (1){
            while(count==0);
            item=remove_item();
            count--;
            ...
            consume an item;
}
```

Avoid Race Conditions

- Find some way to **prohibit more than one process** in its critical region(s) at the same time

- Solution: **Mutual exclusive** access to critical regions
  - Some way of making sure that
    - If one process is using a shared variable
    - The other processes will be excluded from doing the same thing
  - **Only one** process/thread in a critical section **at any time**

Mutual Exclusion

This is not an example of mutual exclusion

Requirements to Avoid Race Conditions

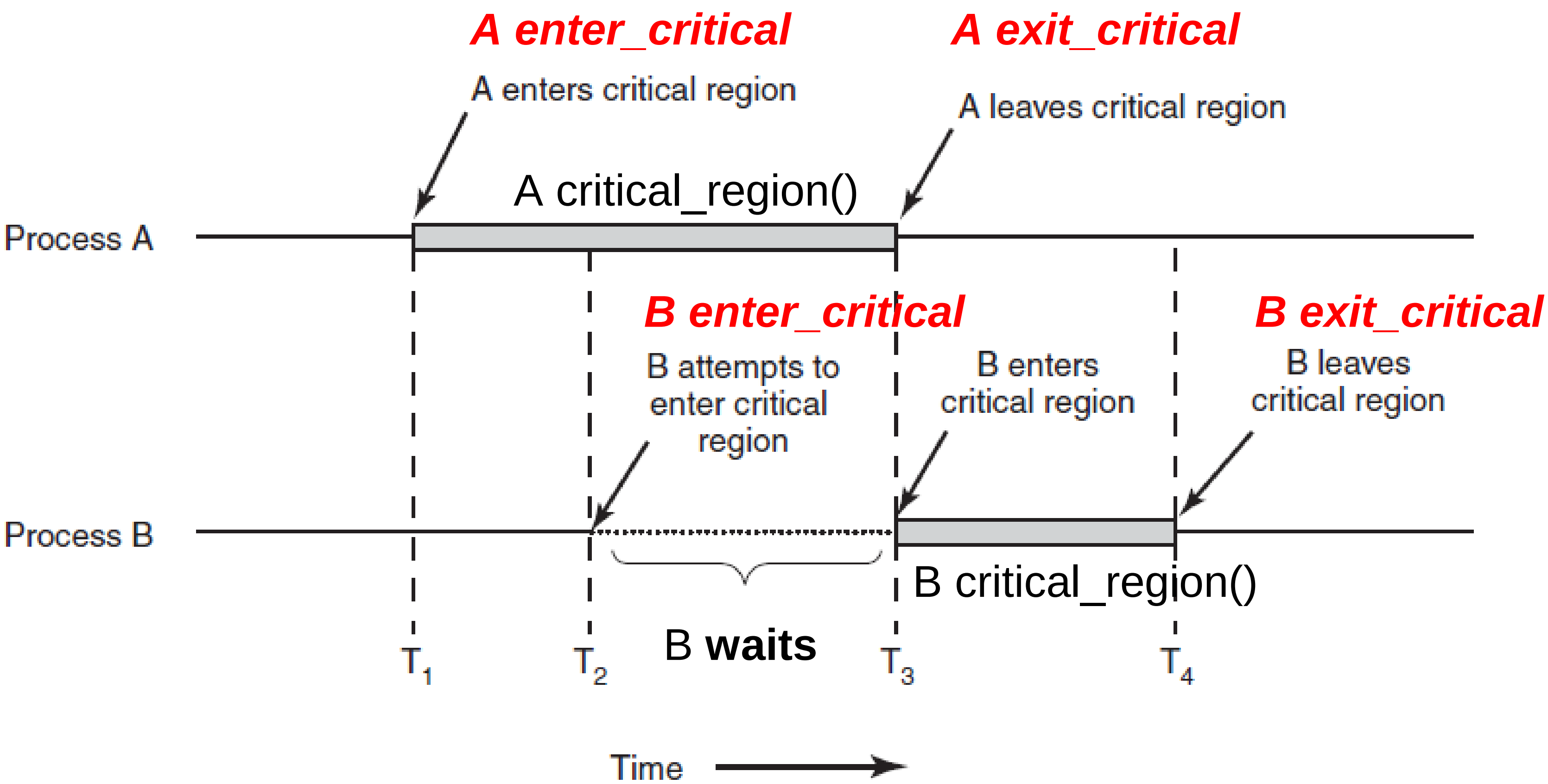
- Critical Regions are **not enough**
  - **Mutual exclusion**
    - At most one thread/process is in the critical section
  - **Progress**
    - No process running outside its critical region may block other processes
  - **Bounded waiting**
    - No process should have to wait forever to enter its critical region
  - **Performance**
    - Overhead of entering and exiting critical section is small wrt the work being done within it
    - No assumption can be made about processing speed

How the Code of a **Critical Region** Looks Like?

- Pseudocode example

```
repeat
    do other work            /* do other work */
    enter_critical           /* enter critical region, may wait to enter */
        critical_region()    /* access shared variables */
    exit_critical            /* leave critical region */
    remainder_region         /* do other work */
until condition
```

- How to implement *enter_critical*, and *exit_critical* **to guarantee mutual exclusion**?


A enter_critical
A enters critical region
A exit_critical
A leaves critical region
A critical_region()
Process A
B enter_critical
B attempts to enter critical region
B enters critical region
B exit_critical
B leaves critical region
Process B
B critical_region()
B waits
T1
T2
T3
T4
Time

Mechanisms

- Disable Interrupts***
  - Needs operating systems, high overhead (especially on multi-CPU)
- Locks/Spinning locks (Spinlocks)
  - Primitive, minimal semantics, used to build others

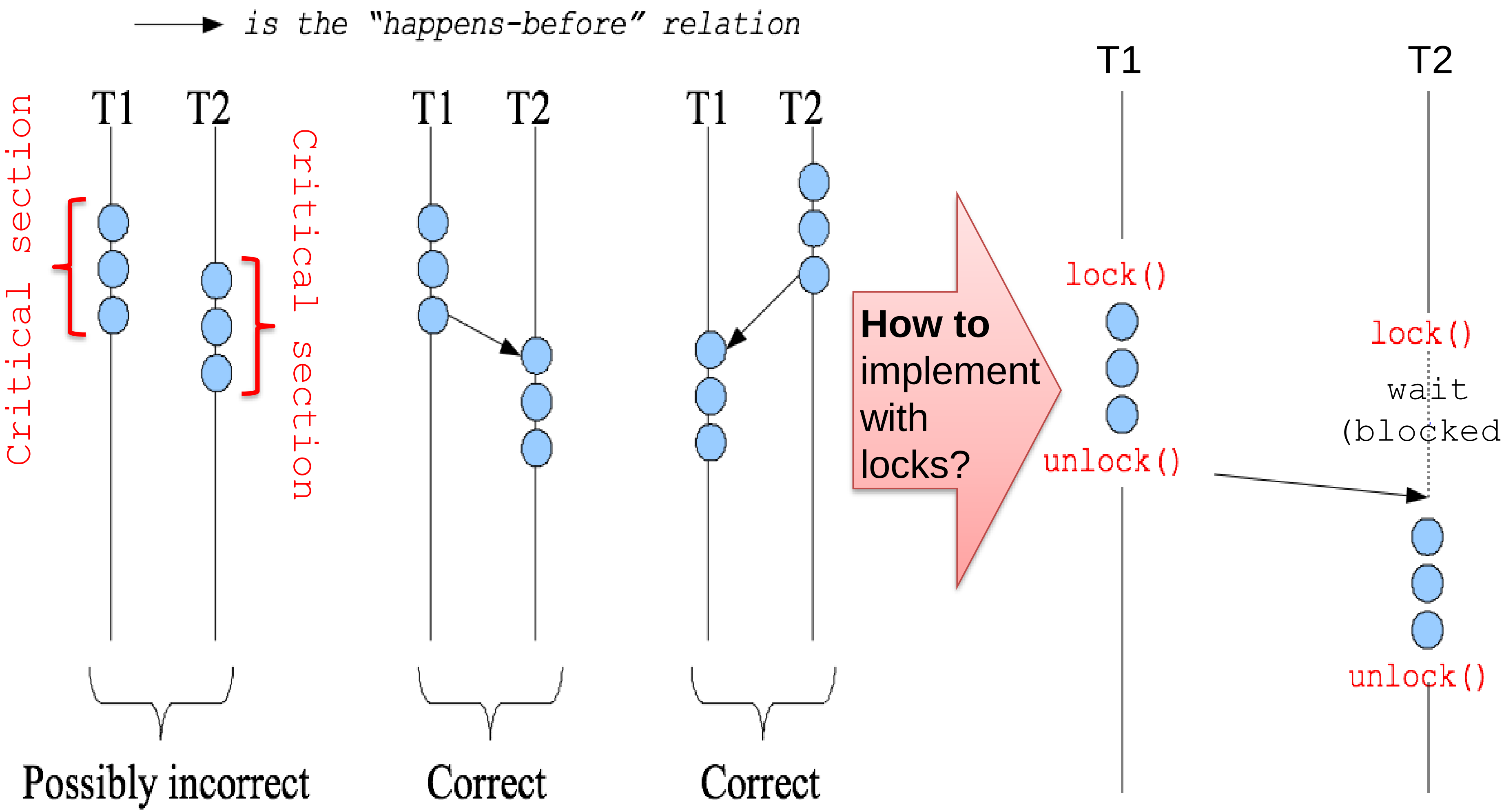
- **Semaphores** (and non-spinning locks)
  - Basic, easy to get the hang of, somewhat hard to program with
- **Monitors**
  - Higher level, require language support, easier to program with

Locks

- A lock is a an object with methods
  - **`acquire()`:** obtain the right to enter the critical section
    - **Prevents progress** of the task until the lock is acquired
  - **`release()`:** give up the right to be in the critical section
    - Immediate

- Note
  - Terminology varies
    - `acquire()/release()`
    - `lock()/unlock()`

Locks: Example



Acquire/Release

- Programmer **pairs up** calls to `acquire()` and `release()`
  - Per thread
    - Or single threaded process
  - From `acquire()` to `release()`
    - A thread **holds the lock**
  - `acquire()` does not return until the caller “owns” (holds) the lock
    - Thread waits (blocked) to enter its critical section
    - At most **one thread** can hold a lock at any time

- What happens if calls aren’t paired?
- What happens if two threads acquire different locks?
- What is the right granularity of locking?

Using Locks

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  return balance;
}
```

critical section

`acquire(lock)`
`balance = get_balance(account);`
`balance -= amount;`
	MOTHER

`acquire(lock)`	YOU
`put_balance(account, balance);` `release(lock);`	MOTHER

`balance = get_balance(account);`
`balance -= amount;`
`put_balance(account, balance);`
`release(lock);`
`return balance;`
	YOU

`return balance;`	MOTHER

time

How to Implement Locks?

```
while (TRUE) {
  acquire(lock);
  critical_region(); /* work */
  release(lock);
  noncritical_region();
}
```

- Spinning on a lock variable

```
struct lock_t {
  int held = 0;
}
void acquire(lock_t* lock) {
  while (lock->held) { };
  lock->held = 1;
}
void release(lock_t* lock) {
  lock->held = 0;
}
```

spinning

```
while (TRUE) {
  while (lock->held != 0); /* loop */
  lock->held = 1;
  critical_region(); /* work */
  lock->held = 0;
  noncritical_region();
}
```

T1

```
while (TRUE) {
  while (lock->held != 0); /* loop */
  lock->held = 1;
  critical_region(); /* work */
  lock->held = 0;
  noncritical_region();
}
```

T2

Spinning on a Lock Variable

- **Race condition** may happen
  1. Process T1 sees lock=0 moves to line 3
  2. Process T1 is descheduled and Process T2 is scheduled
  3. Process T2 sees lock=0, moves to line 3, sets lock =1, **enters critical section**
  4. Process T2 is descheduled and Process T1 is scheduled
  5. Process T1 sets lock=1, **enters critical section**

```
1 while (TRUE) {
2   while (lock->held != 0); /* loop */
3   lock->held = 1;
4   critical_region(); /* work */
5   lock->held = 0;
6   noncritical_region();
7 }
```

T1

```
1 while (TRUE) {
2   while (lock->held != 0); /* loop */
3   lock->held = 1;
4   critical_region(); /* work */
5   lock->held = 0;
6   noncritical_region();
7 }
```

T2

How to Fix That?

- **Problem**
  - Implementation of spinning on a lock variable **has critical sections**
  - The acquire/release must be **atomic**
    - Executes **“all or nothing”**
      - From the view of all CPUs/cores

- Can we solve with **Software**?
  - **Strict alternation**
    - “Progress” property missing
  - **Peterson** Solution
    - Doesn’t easily scale to multiple threads
  - Locking in the OS
    - It works only for user code, very expensive

- Need **hardware**
  - Atomic instructions (Test-and-set, compare-and-swap, …)

Hardware Test-and-Set

- CPU provides the following as a **single atomic instruction**

```
bool test_and_set(bool *flag) {
  bool old = *flag;
  *flag = True;
  return old;
}
```

- Different CPUs implement it differently
  - But in **one assembly instruction**
    - `TAS register, flag_address`

Implementing Locks with Test-and-Set

```
while (TRUE) {
  acquire(lock);
  critical_region(); /* work */
  release(lock);
  noncritical_region();
}
```

- Spinning on a lock variable (with TAS)

```
struct lock_t {
  int held = 0;
}
void acquire(lock_t* lock) {
   while(test_and_set(&lock->held)) { };
}
void release(lock_t* lock) {
  lock->held = 0;
}
```

```
while (TRUE) {
  acquire:
    TAS REGISTER, &(lock->held)
    CMP REGISTER,#0
    JNE acquire
  critical_region(); /* work */
  release:
    MOV &(lock->held),#0
  noncritical_region();
}
```

T1

```
while (TRUE) {
  acquire:
    TAS REGISTER, &(lock->held)
    CMP REGISTER,#0
    JNE acquire
  critical_region(); /* work */
  release:
    MOV &(lock->held),#0
  noncritical_region();
}
```

T2

Using Locks with Test-and-Set

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  return balance;
}
```

critical section

MOTHER

```
acquire(lock)
balance = get_balance(account);
balance -= amount;
```

YOU

```
acquire(lock)
```

MOTHER

```
put_balance(account, balance);
release(lock);
```

YOU

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
return balance;
```

MOTHER

```
return balance;
```

time

- How does a thread blocked on an “acquire” (that is, stuck in a test-and-set loop) **yield** the CPU?
  – **Voluntarily** calls yield( ) *(spin-then-block)*
  – **Involuntary** context switch (e.g., timer interrupt)

Spinning on Lock Variables …

- Spinlocks
  - **Wastes CPU resources**
  - When a thread is spinning on a lock
    - Thread holding the lock **cannot make any progress**
    - **Just burns** CPU cycles

- How to solve this?

- Spinlocks
  - **Primitives** to build higher-level synchronization constructs
  - Ensure acquiring only happens for a short time

Summary

- Threads/processes need access shared data
- Synchronization
  - Introduces temporal ordering
  - May eliminate races
  - Provided by
    - Disable interrupts***
    - Locks (this class)
    - Semaphores (next class)
    - Monitors (next class)

- Spinlocks are the lowest-level mechanism
  - Primitive in terms of semantics
    - Error-prone!
  - Implemented by spin-waiting
    - Crude!