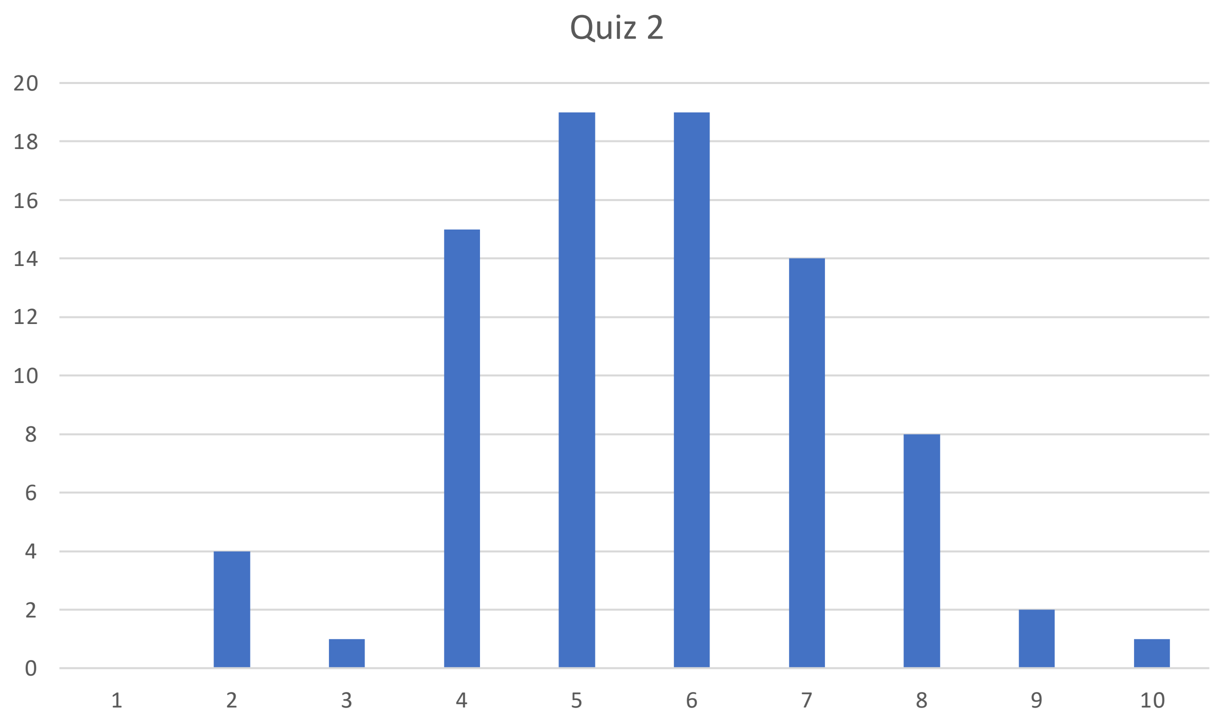
Agenda 6 October

Read Chapters 28 – 32 (Concurrency): Module 5
Quiz Review
Attendance

Synchronization Problems
Introduction to Locks
Test_and_set compare_and_swap


Quiz 2
20
18
16
14
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
9
10

Three processes are executed using a **SCTF** scheduler:

Process	Arrival time	Total CPU time
P1	0	5
P2	4	4
P3	2	2

At what time does P1 end?

What is the SCTF algorithm or rules? Arrival time preemption & evaluate remaining process times

		0	1	2	3	4	5	6	7	8	9	10
P1	0,5											
P2	4,4											
P3	2,2											

fork(): creates a new process, copying code and data to the child
As the processes continue to run, their memory spaces and variable are separate (independent)

Thread: all threads in the same process share all variables in the address space of the process.

exec() overlays the code and data of the process. Any code in the process following the exec() no longer exists.

When a fork() is given, 2 processes continue to run. If these processes fork() again, there are then 4 processes running.

fork(); statement a; statement b; fork(); statement c; statement d;

Real time in a system: time from initiation to completion: wait time + execution time

Attempt at a ‘Signaling’ Solution for Synchronizing

Two Processes Cooperative in the solution of a problem

One generates data for the other to use.

Since they run asynchronously, we need a solution that is independent of the order of execution of their statements.

A high level example Requiring a Coarse-grained Atomic Action

```
var
        boolean operation_aj_performed;
        boolean pi_blocked;
begin
  operation_aj_performed := FALSE;
  Pi_blocked := FALSE;
```

We want P_i to perform operation a_i ONLY AFTER P_j performs operation a_j

P_i

```
....
if operation_aj_performed == FALSE
then
   pi_blocked := TRUE;
   block(Pi);
   {perform operation ai }
.....
```

P_j

```
...
{perform operation aj}
if pi_blocked == TRUE
then
  pi_blocked := FALSE;
  activate (Pi);
else
  operation_aj_performed := TRUE
...
```

The 'Signaling' Solution Fails

- P_i may face indefinite blocking in some situation
- Use *indivisible* or *atomic operations* instead

Time	Actions of process P_i	Actions of process P_j
t_1	**if** *action_aj_performed* == *false*	
t_2		{*perform action* a_j}
t_3		**if** *pi_blocked* == *true*
t_4		*action_aj_performed* :=*true*
⋮		
t_{20}	*pi_blocked* :=*true*;	
t_{21}	*block* (P_i);	

Update to the ‘Signaling’ Solution

- Atomic ⇔ Indivisible
- Two atomic functions make the solution work

```
Procedure check_aj
begin
    if operation_aj_performed = FALSE
    then
        pi_blocked := TRUE;
        block(Pi);
end
```

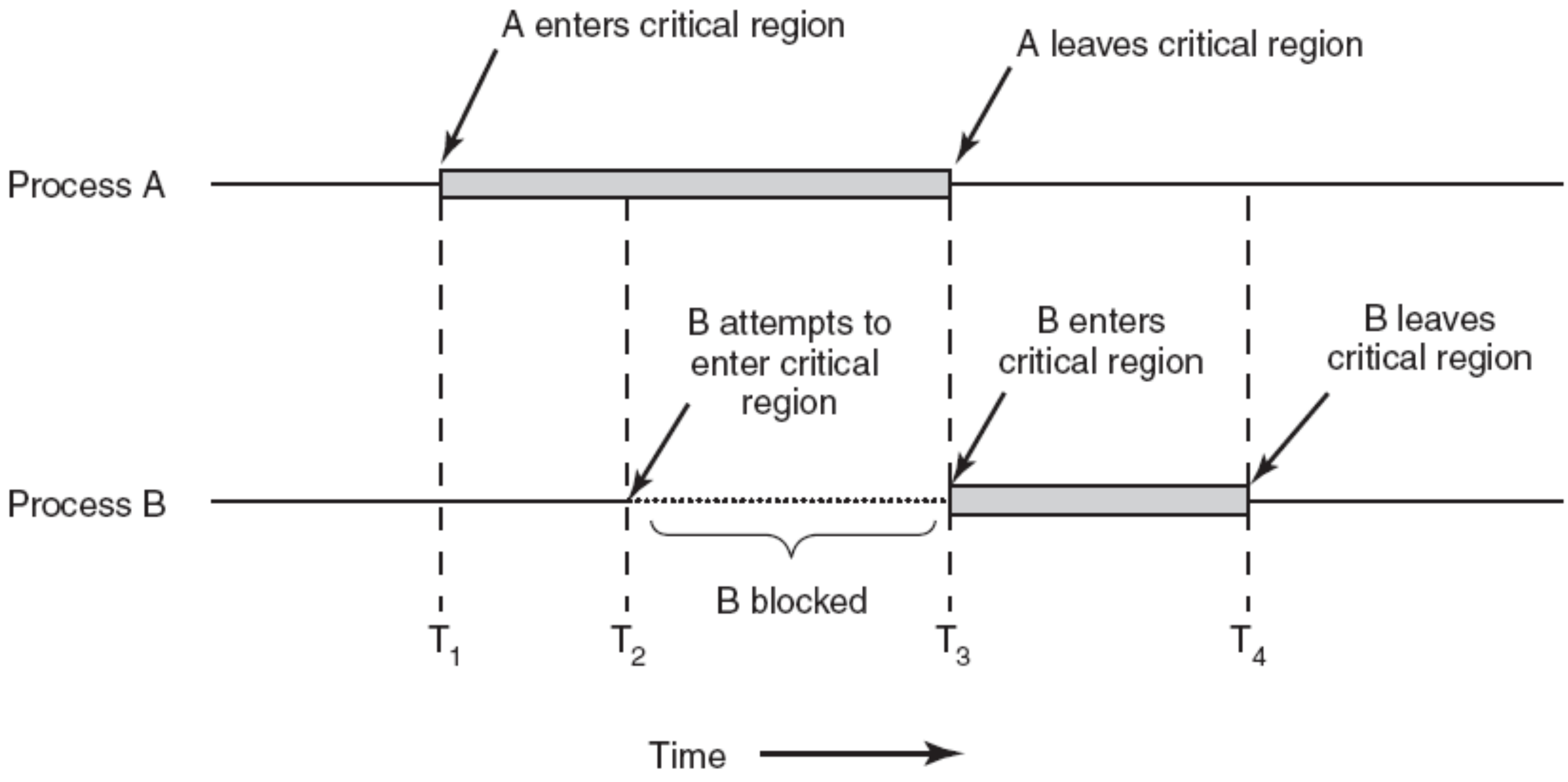
```
Procedure post_aj
begin
    if pi_blocked = TRUE
    then
        pi_blocked := FALSE;
        activate(Pj)
     else
         operation_aj_performed := true;
end;
```

Definition: Indivisible Operation /Atomic

An operation on a set of data items that cannot be executed concurrently, either with itself or with any other operation on a data item included in the set.

Critical Sections (Regions)

- Mutual Exclusion & Critical Regions



Synchronization Approaches

- Busy Waiting versus Blocking
- Hardware Support for Process Synchronization
- Algorithmic Approaches, Synchronization Primitives, and Concurrent Programming Constructs

Busy Waiting

- A form of Synchronization in which a process repeatedly checks a condition until it becomes TRUE

```
While (' state FALSE') {
   };
```

```
AA: cmp B, C
      bne AA
```

while (some process is in a critical section on X_{shared}
or is executing an indivisible operation on X_{shared})

{ do nothing}

Critical Section or
Indivisible Operation
using X_{shared}

- Can implement busy-waiting using machine instructions
- Inefficient in a multi-tasking system, but may be OK in multi-processor systems (1 process per processor).

Strict Alternation or Hand-off Synchronization

- A **special case proposed solution** to the critical section problem.
  (a) Process 0 executes when ‘turn value 0’. (b) Process 1 executes when ‘turn value 1’. In both cases, be sure to note the semicolons terminating the while statements

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;   //DONE – Enable Other
    noncritical_region( );
}
```

(0)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;   //DONE – Enable Other
    noncritical_region( );
}
```

(1)

Safe Access to a Shared Critical Sections

Thread 1	Thread 2
CSEntry	CSEntry
Critical Section	Critical Section
CSExit	CSExit
non-critical section	non-critical section

CSEntry represents the “protocol” required to safely enter the CS
CSExit represents the “protocol” to safely release the CS

Critical Section Control Using Simple Lock Variables

Boolean lock = 0; //unlocked, open

```
While (TRUE){
        acquire (lock);
            Critical_section();
        release(lock);
            Non-critical_section();
}
```

```
While (TRUE){
        acquire (lock);
            Critical_section();
        release(lock);
            Non-critical_section();
}
```

Can this implementation produce a safe solution?

Implementing LOCKS: w/ Load+Store Code

uses a single shared lock variable

Lock Variable

0	1
Open	Closed
Unlocked	Locked
Free	Held

Acquire (perform Lock) : spin on a lock variable until it is unset, then set it in order to acquire lock
Release (Perform unlock): unset lock variable

```
boolean lock = false; // shared variable

void acquire(Boolean *lock) {
  while (*lock) /* wait */ ;
  *lock = true; }
```

CS

```
void release(Boolean *lock) {
*lock = false; }
```

Can this be used for Critical Section Control?

A failed lock implementation

- Thread 1 spins, lock is released, ends spin
- Thread 1 interrupted just before setting flag
- Race condition has moved to the lock acquisition code!

Both threads grab the lock

Problem: Testing lock and setting lock are not atomic

Thread 1

```
acquire(*lock)
        while (*lock)
```

Interrupt: Switch to Thread 2

Thread 2

```
acquire(*lock)
        while (*lock);
        *lock = 1;
```

Interrupt: switch to Thread 1

```
        *lock = 1:   //set flag to 1
```

Hardware Support for Process Synchronization

- Need Indivisible instructions for the “wait implementation” (the CSEntry)
  - Avoid race conditions on memory locations (i.e., wait until safe)
- Used with a “lock variable” to implement indivisible operations and Critical Section (CS) control

If OPEN, move forward with lock CLOSED for others

Need to Know:
If CS access is open, then we take it and no one else sees it open

entry_test:

```
If lock = closed
      then  goto entry_test;
lock = closed;
```

Performed by an indivisible instruction

{ Critical section or indivisible operation}

lock = open;

- *CS control and entry_test* performed with an indivisible instruction
  - Test-and-set (TS) instruction
  - Swap instruction

Synchronization Hardware

Many systems provide hardware support for critical section code and control

- **Disable Interrupts**
  - Uniprocessors
    - Currently running code would execute without preemption
    - Generally too inefficient on uniprocessor systems (all other activity on hold)
  - **Does not work for multiprocessor systems**

    Operating systems using this method are not broadly scalable

- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
    - Either test memory word AND set value
    - Or swap contents of two memory words

    Within 1 atomic instruction

Two Operations in One Instruction

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

lock() == acquireLock()
unlock() == releaseLock

A lock is just a variable whose value (state) is either:

Available / free / unlocked

or

Acquired / held / unlocked

Call lock() in order to acquire a lock. If the lock is free, the function returns. If not free, the caller waits until the lock is free and then acquires it.

Call unlock() to release or free the lock so that it is available for acquisition.

Building A Lock

Goals of a lock implementation

- o Mutual exclusion (obviously!)
- o Fairness: all threads should eventually get the lock, and no thread should starve
- o Low overhead: acquiring, releasing, and waiting for lock should not consume too many resources

- Implementation of locks are needed for both user space programs (e.g., pthreads library) and kernel code
- Implementing locks needs support from hardware and OS

Solution: Hardware atomic instructions

- Very hard to ensure atomicity only in software
- Modern architectures provide hardware atomic instructions
- Example of an atomic instruction: test-and-set
  - Update a variable and return old value, all in one hardware instruction

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

This is the operation of the hardware instruction TestAndSet

The Test & Set ‘Function’

TS() TestAndSet; where lock: Global, cc: Local

TS(lock, cc):   <cc: = lock; lock: = TRUE>

“Atomic” instruction <…> indivisible operations

```
Global int lock = FALSE;
CSEntry:
        int cc = FALSE;
        TS(lock, cc));
        While (cc){
                TS(lock, cc);
        }
```

SPIN-LOCK

```
CSExit:
        Lock = FALSE;
```

The Test & Set Lock Instruction

- Spin-Lock Implementation with a TSL instruction

Note:
REGISTER is
“local”

```
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 nonzero, 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
```

SPIN-LOCK

TSL causes the CPU executing the instruction to lock the memory bus to prohibit other CPUs from accessing memory until complete. Ensures reads and writes are done atomically

This safe locking result would NOT be Accomplished by Disabling Interrupts in a multiprocessor system