Communication and Synchronisation

Files

Signals (U

Events, exceptions (Windo

Pipes

Message Queues (UNIX)

Mailslots (Windows)

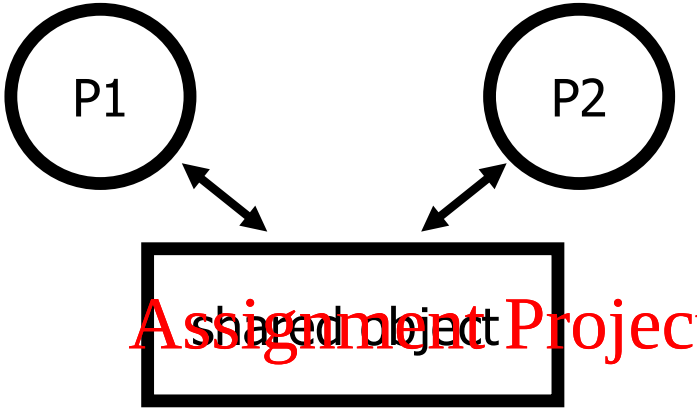
Sockets – in NDS course

Shared memory

Semaphores, Locks, Monitors

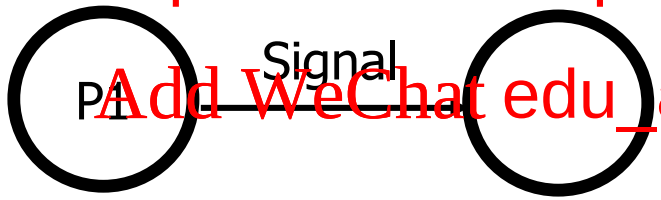
Types of Process Interaction

Sharing



Require mutually exclusive access to prevent interference

Synchronisation



informs P2 that some
nt has happened.
P2 waits for it

Communication

P1 sends P2 data.
P1 blocks when buffer is full, P2 blocks when buffer is empty.

Mutual Exclusion, **Synchronisation** and **Communication** are closely related.

UNIX Signals

Inter-Process Communication (IPC) mechanism

Signal delivery similar to delivery of hardware interrupts

Used to notify processes when an event occurs

A process can send a signal to another process if it has permission

- *"the real or e* *ceiving process must match that of the sending p* *e user must have appropriate privileges (such a set-user-ID program or the user is the super-user)."* (man page)
- The kernel can send signals to any process

When Are Signals Generated?

When an exception occurs

- e.g., division by zero => **`SIGFPE`**,
  segment violation => **`SIGSEGV`**

When the kernel wants to notify the process of an event

- e.g., if process writes to a closed pipe => **`SIGPIPE`**

When certain key combinations are typed in a terminal

- e.g., Ctrl-C => **`SIGINT`**

Explicitly using the **`kill()`** sys

UNIX Signals – Examples

SIGINT	Interrupt from keyboard
SIGABRT	
SIGFPE	
SIGKILL	Kill signal
SIGSEGV	Invalid memory
SIGPIPE	Broken pipe: write to pipe with no readers
SIGALRM	Timer signal from **`alarm`**
SIGTERM	Termination signal

UNIX Signals

The default action for most signals is to terminate the process

But the receiving process may choose to

- Ignore it
- Handle it by installing a signal handler
- Two signals cannot be ignored or handled: **`SIGKILL`** and **`SIGSTOP`**


```
signal(SIGINT, my_handler);

void my_handler(int sig) {
    printf("Received SIGINT. Ignoring...")
}
```

Signal Handlers – Example

```
#include <signal.h>
#include <stdio.h>

void my_handler(int sig) {
  fprintf(stderr, "SIGINT caught!”);
}

int main(int argc, c
{
  signal(SIGINT, my_handler);
  while (1) {}
}
```

```
$ ./a.out
[ctrl-C]
SIGINT caught
```

Sockets

Allow bidirectional communication

Can be used to exchange information both locally and across a network

- Unlike pipes which are identified by machine specific file descriptors

Two types of sock

- TCP (stream sockets)
- UDP (datagram sockets)

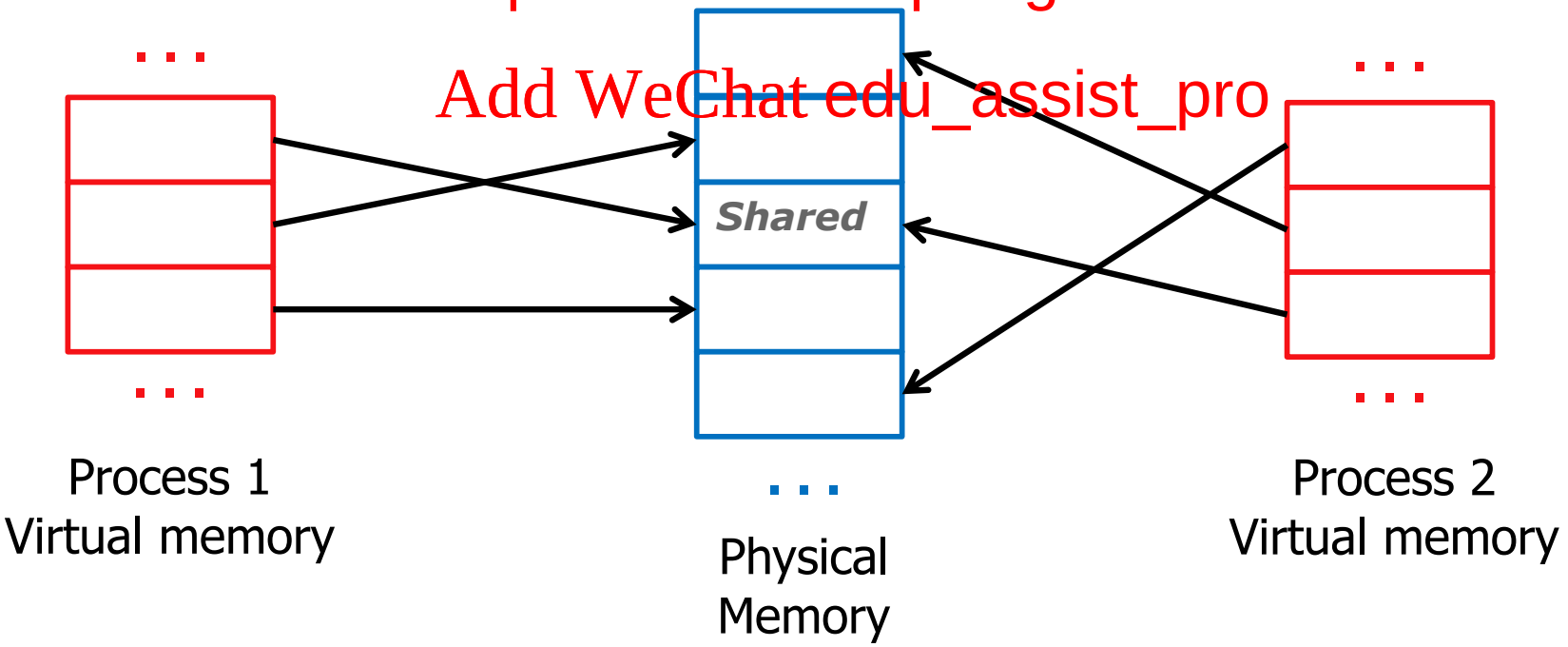
Covered in Networks and Distributed Systems course

Shared Memory

Processes can set up shared memory areas

- Implicitly or explicitly mapped to files on disk

After shared memory is established, no need for kernel involvement



Shared Memory – System V API

shmget	Allocates a shared memory segment
shmat	segment to the address space o
shmctl	Changes the pr iate with a shared memory segment
shmdt	Detaches a shared memory segment from a process

Process Synchronization

How do processes synchronize their operation to perform a task?

Key concepts:

- Critical sec
- Mutual exc
- Atomic operations
- Race conditions
- Synchronization mechanisms
  - ✦ Locks, semaphores, monitors, etc.
- Deadlock
- Starvation

Concepts relevant to both **processes** and **threads**

Shared Data Example

```
Extract £1000
from account 1234
```

```
Extract £1000
from account 1234
```

Shared Data Example

```
void  Extract(int acc_no, int sum)
{
   int B = Acc[acc_no];
   Acc[acc_no] = B - sum;
}
```


Acc[1234]

B = 10,000
Acc[1234] = 9000

Extract(1234, 1000)

B = 9,000
Acc[1234] = 8000

Extract(1234, 1000)

Shared Data Example

```
void  Extract(int acc_no, int sum)
{
   int B = Acc[acc_no];
   Acc[acc_no] = B - sum;
}
```

Critical section!
need mutual exclusion

Acc[1234]

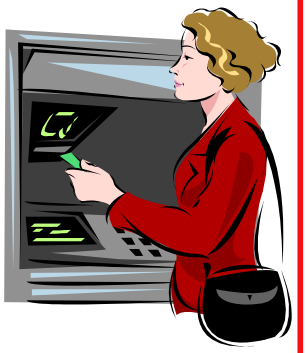
B = 10,000

Acc[1234] = 9000

Extract(1234, 1000)

B = 10,000

Acc[1234] = 9000



Extract(1234, 1000)

Critical Sections and Mutual Exclusion

Critical section/region: section of code in which processes access a shared resource – executed by only one process at a time.
A code section is critical if it:

1. Reads a memory location which is shared with another process
2. Updates a sh a value which depends on what it re

Mutual exclusion ensures tha ss is executing its critical section, no other process can be executing it

- Processes must request ***permission*** to enter critical sections

A ***synchronisation mechanism*** is required at the entry and exit of the critical section

Requirements for Mutual Exclusion

- No two processes may be simultaneously inside a critical section
- No process running outside the critical section may prevent other processes from entering the critical section
  - When no proc n, any process requesting permission to n so immediately
- No process requiring access to it tion can be delayed forever
- No assumptions are made about relative the speed of processes

Critical Sections and Mutual Exclusion

Disabling Interrupts

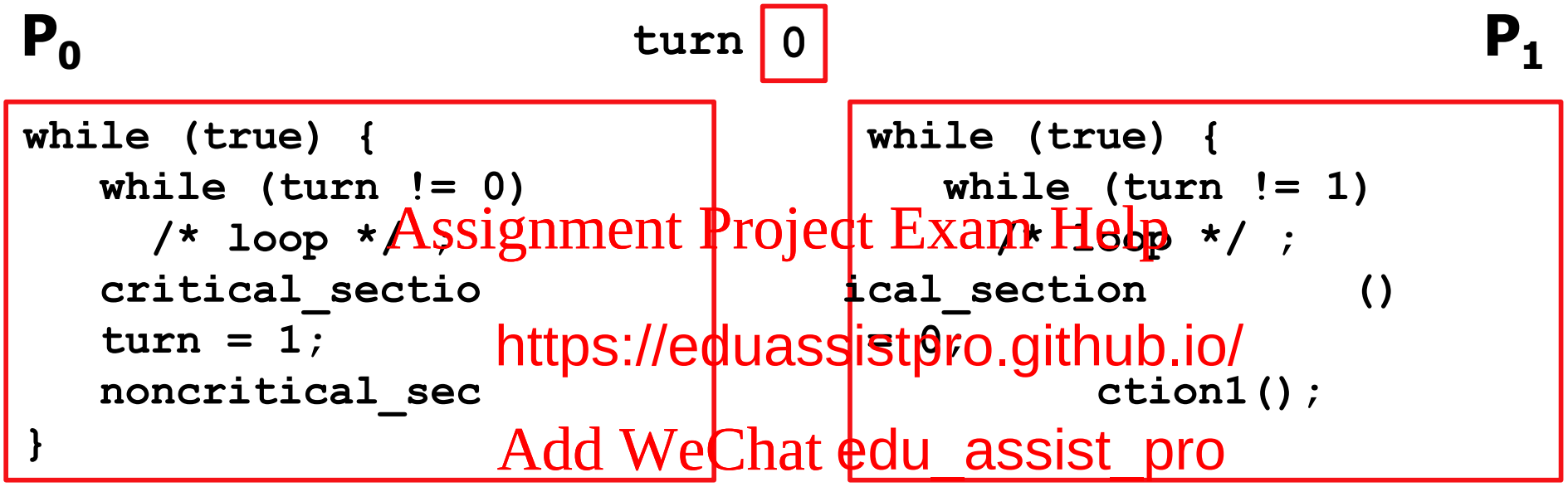
```
void Extract(int acc_no, int sum)
{
   CLI();
   int B = Acc[acc_no];
   Acc[
   STI();
}
```


Works only on single-processor systems, but not with user level threads.

Misbehaving/buggy processes may never release CPU

- Mechanism usually only used by kernel code

Software Solution – Strict Alternation



What happens if P_0 takes a long time in its non-critical section?

- Remember: No process running outside its critical section may prevent other processes from entering the critical section

Can we have P_1 execute its loop twice in a row (w/o P_0 executing in-between)?

Busy Waiting

Strict alternation solution requires continuously testing the value of a variable

Called **busy waiting**

- Wastes CPU ti
- Should only b expected to be short

Atomic Operations

```
void Extract(int acc_no,
             int sum)
{
  int B = Acc[acc_no];
  Acc[acc_no] =
}
```

```
void Extract(int acc_no,
             int sum)
{
    Acc[acc_no] -= sum;
```


Does this work?

- Not atomic!

Atomic operation: a sequence of one or more statements that is/appears to be indivisible

Lock Variables

L= 0 lock open/free
L=1 locked

```
void Extract(int acc_no, int sum)
{
  lock(L);
  int B = Acc[acc
  Acc[acc_no] = B - sum;
  unlock(L);
}
```

```
void lock(int L)
{
    while (L != 0)
      /* wait */ ;
    L = 1;
```



```
void unlock(int L)
{
    L = 0;
}
```

- Does this work?

TSL (Test and Set Lock) Instruction

Atomic instruction provided by most CPUs

`TSL(LOCK)`

- Atomically sets memory location **`LOCK`** to **`1`** and returns old value

Pseudocode

```
void lock(int L)
{
    while (TSL(L) != 0)
      /* wait */ ;
}
```


Assembler

```
                ad L and set
                ndition code if L=0
BN              s if Z is not set.
M               L to constant n
```

```
LOCK:   TSL L
        BNZ LOCK

UNLOCK: MOV #0, L
```

Spin Locks

Locks using busy waiting are called spin locks

Waste CPU

- Should only be used when the wait is expected to be short

May run into *priority inversion problem*

Priority Inversion Problem and Spin Locks

Two processes:

- H with high priority
- L with low priority
- H should always be scheduled if runnable

Assume the follow

- H is waiting for I/O
- L acquires lock A and enters tion
- I/O arrives and H is scheduled
- H tries to acquire lock A that L is holding

What happens?

Lock Granularity

```
void Extract(int acc_no, int sum)
{
  lock(L);
  int B = Acc[ac
  Acc[acc_no] =
  unlock(L);
}
```


T1: `Extract(1, 40);`

T2: `Extract(2, 40);`

What happens if there are concurrent accesses to *different* accounts?

Lock Granularity

```
void Extract(int acc_no, int sum)
{
  lock(L[acc_no]);
  int B = Acc[ac
  Acc[acc_no] =
  unlock(L[acc_no]);
}
```

T1: `Extract(1, 40);`

T2: `Extract(2, 40);`

Lock granularity: the amount of data a lock is protecting

Is finer granularity always better?

Lock Overhead and Lock Contention

Lock overhead: a measure of the cost associated with using locks

- Memory space
- Initialization
- ***Time required to acquire and release locks***

Lock contention: number of processes waiting for a lock

- More contention, less parallelism

- **Coarser granularity:**
  - Lower overhead
  - More contention
  - Lower complexity
- **Finer granularity:**
  - Higher lock overhead
  - Less contention
  - Higher complexity

Minimizing Lock Contention/Maximizing Concurrency

Choose finer lock granularity

- But understand tradeoffs

Release a lock as soon as it is not needed

- Make critical sections small!


```
void AddA                                    nt   balance)
{
  lock(L_Acc);
  CreateAccount(acc_no);
  lock(L[acc_no]);
  Acc[acc_no] = balance;
  unlock(L[acc_no]);
  unlock(L_Acc);
}
```

?

Read/Write Locks

```
void ViewHistory(int acc_no)
{
    print_transactions(acc_no);
}
```

```
P1: ViewHistory(1234);
P2: ViewHistory(1234);
    ViewHistory(1234);
```


Any locks needed?

Race Condition

Occurs when multiple threads or processes read and write **shared data** and the final result depends on the relative timing of their execution

- i.e. on the exact process or thread **interleaving**

E.g., the **`Extrac`** e of account 8,000 or 9,000

Thread Interleavings

```
= 2;
```

```
int a, b; // shared
void  P1()                    void P2()
{                             {
   a = 1;                        b = 2;
   b =
}
```


a = 1	a = 1	a = 1	b = 2	b = 2	b = 2
b = 1	b = 2	b = 2	a = 2	a = 1	a = 1
b = 2	b = 1	a = 2	a = 1	a = 2	b = 1
a = 2	a = 2	b = 1	b = 1	b = 1	a = 2
(2, 2)	(2, 1)	(2, 1)	(1, 1)	(2, 1)	(2, 1)

Thread Interleaving

Menti.com Q1-3 76 84 50

Consider the following three threads:

T1: {a=1; b=2;} T2: {b =1;} T3: {a=2;}

1. How many different thread interleavings are there?

2. If all thread interleur, what is the probability to have a=1 and b=1 afs complete execution?

3. What about a=2 and b=2?

Semaphores

Blocking synchronization mechanism invented by Dijkstra in 1965

Idea: Processes will cooperate by means of *signals*

- A process will block, waiting for a specific signal
- A process [illegible] eived a specific signal

Semaphores are *special va* [illegible] essible via the following *atomic* operations:

- `down(s):` receive a signal via semaphore **s**
- `up(s):` transmit a signal via semaphore **s**
- `init(s, i):` initialise semaphore `s` with value `i`

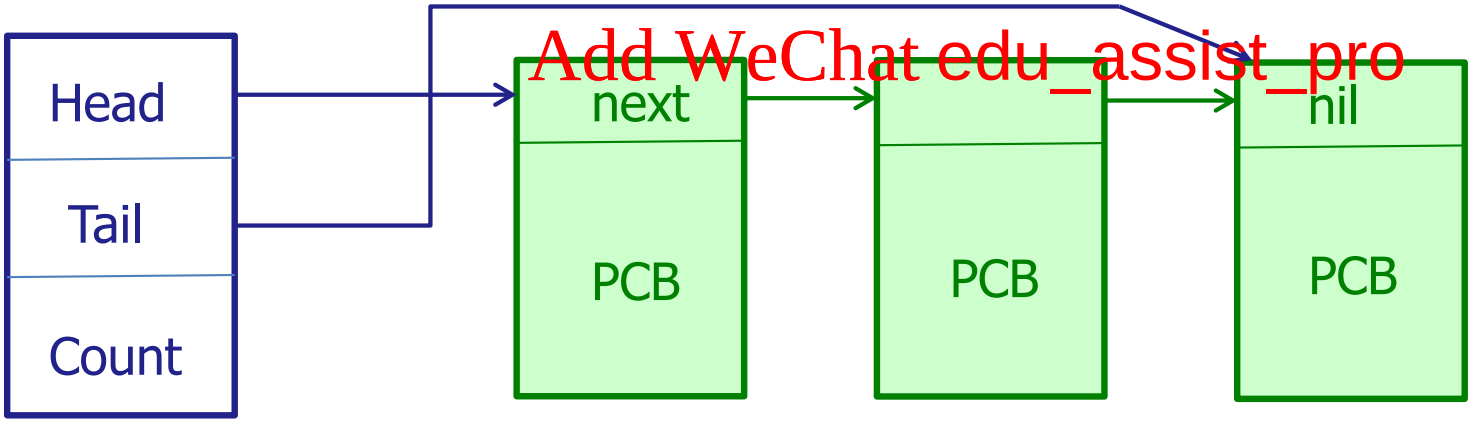
down() also called **P()** (*probeer te verlagen)*

up() also called **V()** (*verhogen*)

Semaphores

Semaphores have two private components:

- A counter (non-negative integer)
- A queue of processes currently waiting for that semaphore
  Queue is typically first in first out (FIFO)



Semaphore Data Structure

Queue of processes waiting on Semaphore

Semaphore Operations

```
init(s, i) ::= counter(s) = i
               queue(s) = {}
```



```
down(s) ::= i
er(s)                                  - 1
            else
                 add P to q
                 suspend current process P
```



```
up(s) ::= if queue(s) not empty
             resume one process in queue(s)
          else
             counter(s) = counter(s) + 1
```

Semaphores for Mutual Exclusion

Binary semaphore: counter is initialized to 1

Similar to a lock/mutex

Note: for binary semaphore if s =1, up (s) leaves s = 1

General Semaphores

The initial value of a semaphore counter indicates how many processes can access shared data at the same time

```
counter(s) >= 0:
```

Initial value defines how many processes can execute down without being block

Menti.com Q4, 5 threads & semaphore 76 84 50

Producer / Consumer

There can be multiple producers and consumers

Producer / Consumer

Buffer constraints:

- Buffer can hold between **0** and **N** items

Producer constraints:

- Items can only be deposited in buffer if there is space (ite

- Items can only be deposi r if mutual exclusion is ensured

Consumer constraints:

- Items can only be fetched from buffer if it is not empty (items in buffer > 0)
- Items can only be fetched from buffer if mutual exclusion is ensured

Producer/Consumer?

```
var item, space, mutex: semaphore
init (item, 0)      /* Semaphore to ensure buffer is not empty */
init (space, N)     /* Semaphore to ensure buffer is not full */
init (mutex, 1)     /* Semaphore to ensure mutual exclusion */
```

```
process Producer
  loop
    produce item
    down(mutex)
    down(space)
    deposit item
    up(item)
    up(mutex)
  end loop
end Producer
```

```
process Consumer
  loop
        tex     )
      em)
    up(mutex)
    consume item
  end loop
end Producer
```


What is wrong with this?

Producer/Consumer

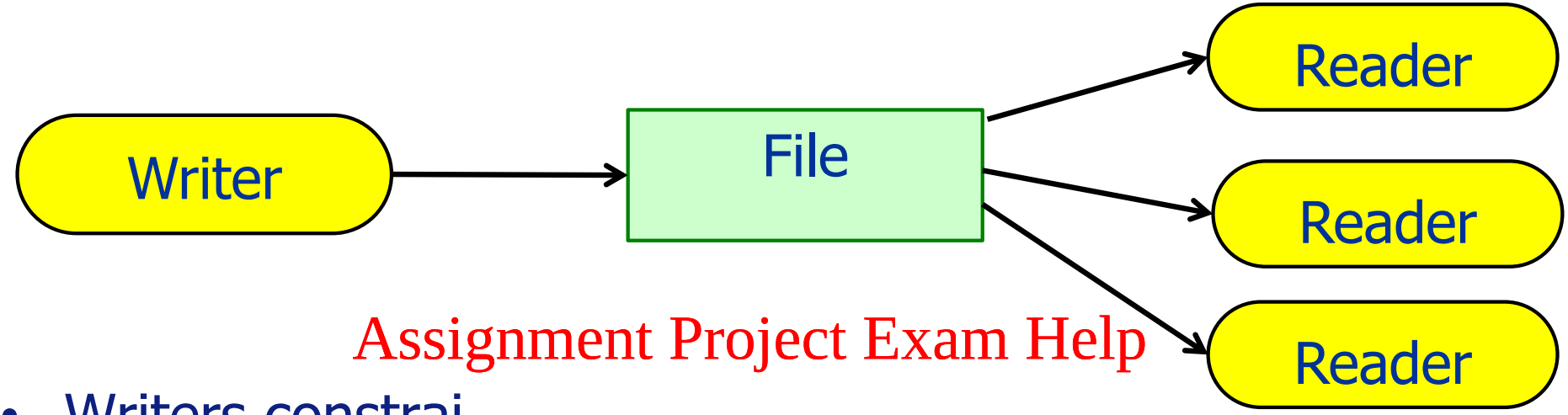
```
var item, space, mutex: semaphore
init (item, 0)      /* Semaphore to ensure buffer is not empty */
init (space, N)     /* Semaphore to ensure buffer is not full */
init (mutex, 1)     /* Semaphore to ensure mutual exclusion */
```

```
process Producer
  loop
    produce item
    down(space)
    down(mutex)
    deposit item
    up(mutex)
    up(item)
  end loop
end Producer
```

```
process Consumer
  loop
    down(item)
    down(mutex)
    remove item
    up(mutex)
    up(space)
    consume item
  end loop
end Producer
```


Works for multiple producers & consumers
What happens when space = 0 or items = 0?
Animation: https://www.youtube.com/watch?v=NuvAjMk9bZ8

Readers/Writers



- Writers constrai
  - items can o roces is writing;
  - items can only be written if cess is reading.
-
- Readers constraints:
  - items can only be read if no other process is writing;
  - items can be read if there are other processes reading.
-
- File can hold an arbitrary number of items.

Readers/Writers With Semaphores

```
semaphore mutex, wrt;
int read_cnt = 0;
init(mutex, 1);
init(wrt, 1);

process writer()
  loop
    produce item
    down(wrt);
    write item
    up(wrt);
  end loop
end writer
```

```
process reader()
  loop
    if(read_cnt == 0)
      //1st reader
      down(wrt);
    down(mutex)
_cnt          += 1;
    utex   );
        em
          mutex);
    read_cnt -= 1
    up(mutex);
    If (read_cnt == 0)
       up(wrt);
    consume item
  end loop
end reader
```


Does this work?

Readers/Writers With Semaphores

```
semaphore mutex, wrt;
int read_cnt = 0;
init(mutex, 1);
init(wrt, 1);

process writer()
  loop
    produce item
    down(wrt);
    write item
    up(wrt);
  end loop
end writer
```

```
process reader()
  loop
    down(mutex)
    read_cnt += 1;
    if(read_cnt == 1)
    //1st reader
        down(wrt);
    up(mutex);
    em
        mutex);
    read_cnt -= 1
    If (read_cnt == 0)
       up(wrt);
    up(mutex);
    consume item
  end loop
end reader
```


Is this fair?

Semaphore Question

The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0, S2 = 0.

```
Process P0                Process P1          Process P2
 while true                 down(S1);          down(S2);
   { down(S0);                                 up(S0);
     print '0';
     up(S1);
     up(S2) }
```


How many times will P0 print '0'?

Menti.com Q6 76 84 50

General Semaphore Using Binary Semaphores

Describe a suitable data structure for a general semaphore and give a pseudocode outline for the following operations in terms of the operations down(s) and up(s) on a binary semaphore s.

init (value, gs)	ral semaphore
gen_down (var gs)	the down on a general semaphore gs
gen_up (var gs)	the up operation on a general semaphore gs

Monitors

Higher-level synchronization primitive

Introduced by Hansen (1973) and Hoare (1974)

Refined by Lampson (1980)

Monitors

Ensure mutual exclusion for shared resource (data)

Entry procedures

- Can be called from outside the monitor

Internal proced

- Can be cal ocedures

An (implicit) monitor lock

One or more condition variables

Processes can only call entry procedures

- cannot directly access internal data

Only one process can be in the monitor at one time

Condition Variables

Associated with high-level conditions

- "some space has become available in the buffer"
- "some data has arrived in the buffer"

Operations:

- **wait(c)**: r waits for **c** to be signalled
- **signal(c)**: wakes up one iting for **c**
- **broadcast(c)**: wakes up all processes waiting for **c**

Signals do not accumulate i.e c is not a counter.

- If a condition variable is signalled with no one waiting for it, the signal is lost

What happens on signal?

[Hoare] A process waiting for signal is immediately scheduled

+ Easy to reason about

– Inefficient: the process that signals is switched out, even if it has not finished yet with the monitor

– Places extra constraints on the scheduler

[Lampson] Sending si a wait are not atomic

– More difficult to understand, need a care when waking up from a wait()

+ More efficient, no constraints on the scheduler

+ More tolerant of errors: if the condition being notified is wrong, it is simply discarded when rechecked (see next slides)

*Usually **[Lampson]** is used*

Hoare Monitor Implementation Using Semaphores

Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

Each access procedure F will be replaced by

```
wait(mut
    …
body of access proce
    …
if (next_count > 0)
   up(next)
else
   up(mutex);
```


Mutual exclusion within a monitor is ensured

Note: this code is generated by a compiler and is not seen by the programmer who writes the access procedures.

Monitor Implementation – Condition Variables

For each condition variable **c**, we have:

```
semaphore c_sem; // (initially = 0)
int c_count = 0;
```

The operation wait (c) can be implemented as:

```
c_coun
if (ne                    t);
else up(mutex);
down(c_sem);
c_count--;
```


The operation signal (c) can be implemented as:

```
if (c_count > 0) {
   next_count++; up(c_sem);
   down(next); next_count--;
}
```

Menti.com Q7 monitor signal 76 84 50

Producer/Consumer with Monitors

```
monitor ProducerConsumer
    condition not_full, not_empty;
    integer count = 0;

    entry procedure insert(item)
        if (count     1                    );
        insert_it
        signal(not_empty);

    entry procedure remove(item)
        if (count == 0) wait(not_empty);
        remove_item(item); count--;
        signal(not_full);
end monitor
```


oes this work?

Producer/Consumer with Lampson Monitors

```
monitor ProducerConsumer
    condition not_full, not_empty;
    integer count = 0;

    entry procedure insert(item)
       while (co                full        );
       insert_it
       signal(not_empty);

    entry procedure remove(item)
       while (count == 0) wait(not_empty);
       remove_item(item); count--;
       signal(not_full);
end monitor
```

Readers/Writers Revisited

Correctness Constraints:

- Readers can access file when no writers
- Writers can access file when no readers or writers
- Only one thread manipulates state variables at a time

Basic structure of

- Reader()

  Wait until no writers
  Access file
  Check out – wake up a waiting writer
- Writer()
  Wait until no active readers or writers
  Access file
  Check out – wake up waiting readers or writer

Readers/Writers: Fairness?

Problem statement clarification

- Suppose that a writer is active and a mixture of readers and writers now shows up. Who should get in next?
- If a writer is waiting and an endless of stream of readers keeps showing up. Is it fair for them to become active?

Alternation is a po

- Once a reader i in next.
- If a writer is waiting, one writer ext

State variables needed (Protected by a lock called "lock"):

- int NReaders: Number of active readers; initially = 0
- int WaitReaders: Number of waiting readers; initially = 0
- int NWriters: Number of active writers; initially = 0
- int WaitWriters: Number of waiting writers; initially = 0
- Condition CanRead = NIL, CanWrite = NIL

Readers/Writers with Monitors

```
monitor ReadersNWriters
  integer WaitWriters, WaitReaders,
          NReaders, NWriters;
  condition CanRead, CanWrite;

  entry procedure StartRead()
    if(NWriters                    > 0)
    {
      ++WaitReaders; Wait(Ca          WaitReaders;
    }
    ++Nreaders;
    Signal(CanRead);
  end StartRead

  entry procedure EndRead()
    If(--Nreaders == 0) Signal(CanWrite);
  end EndRead
```

Reader/Writer contd

```
entry procedure StartWrite()
   if(NWriters == 1 or NReaders > 0)
   {
     ++WaitWriters; wait(CanWrite); --WaitWriters;
   }
   NWriters = 1;
end StartWrite;

entry procedure EndWrite()
   NWriters = 0;
   if(WaitReaders > 0) Signal(CanRead);
     else Signal(CanWrite);
end EndWrite;

end monitor
```

Monitors

Monitors are a language construct

Not supported by C

Java

- synchronized
- no condition
  - ✦ wait() and notify()

Synchronization within Monitors

Synchronization within monitors uses condition variables and two special operations, **wait** and **signal**. A more general form of synchronization would be to have a single primitive, **waituntil**, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for example,

waituntil ( )

The signal primitive would be needed. This scheme is clearly more general than that of Hoare, but it is not used. Why not?
(Hint: think about the implementation.)

Bohr and Heisen bugs

Bohrbugs:

- Deterministic, reproducible bugs
- Behave similar to Bohr's atom model where electrons deterministically orbit the nucleus

Heisenbugs

- Non-deter ce bugs
  - Often caused by race c

- Suffer from the observer enberg Uncertainty Principle): attempts to observe them (i.e., printfs) make them disappear!

Which bug would you rather have?

- During development/testing: _____
- During deployment: _____

Communication & Synchronization Summary

Signals: really interaction with kernel, to wake a waiting process or indicate a problem.

Pipes: simple read, write type communication

Shared memory: requires synchronisation to prevent corruption

Critical section: cod ses shared resource

Mutual exclusion: o ithin CS

Disabling interrupts: may not be ef

Locks: low level, busy wait, very difficult to program correctly

Semaphores: blocks waiting program, but difficult to program

Monitors: easier to program, but signal semantics can be tricky