Communication and Synchronisation

Files

Signals (UNIX)

Events, exceptions (Windows)

Pipes

Message Queues (UNIX)

Mailslots (Windows)

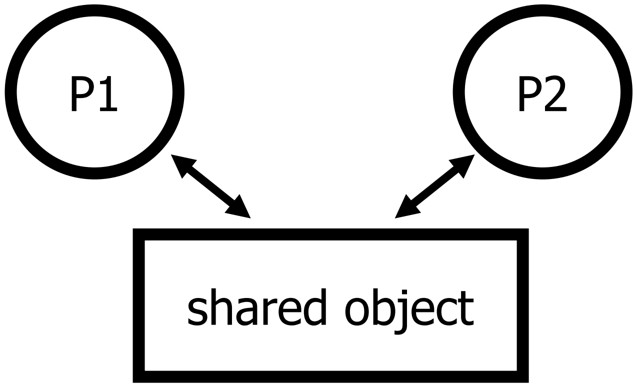
Sockets – in NDS course

Shared memory

Semaphores, Locks, Monitors

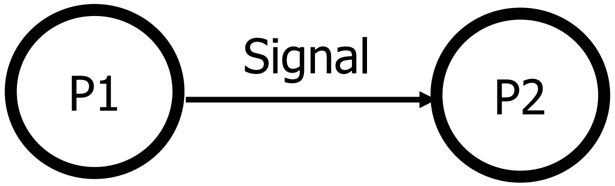
Types of Process Interaction

Sharing



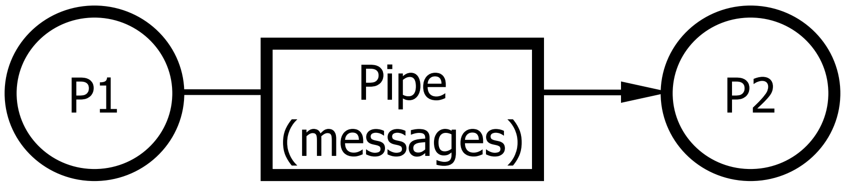
Require mutually exclusive access to prevent interference

Synchronisation



P1 informs P2 that some event 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 effective user ID of the receiving process must match that of the sending process or the user must have appropriate privileges (such as given by 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()`** system call

UNIX Signals – Examples

`SIGINT`	Interrupt from keyboard
`SIGABRT`	Abort signal from **`abort`**
`SIGFPE`	Floating point exception
`SIGKILL`	Kill signal
`SIGSEGV`	Invalid memory reference
`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/handled: **`SIGKILL`** and **`SIGSTOP`**

```
signal(SIGINT, my_handler);

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

Signal Handlers – Example

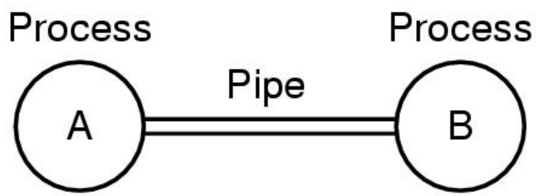
```
#include <signal.h>
#include <stdio.h>

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

int main(int argc, char *argv[])
{
  signal(SIGINT, my_handler);
  while (1) {}
}
```

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

UNIX Pipes

A **pipe** is a method of connecting the *standard output* of one process to the *standard input* of another

- Allows for **one-way** communication between processes



Widely-used on the command line and in shell scripts

- `ls | less`
- `cat file.txt | grep hello | wc -l`

Two types of pipes

- unnamed
- named

pipe()

```
int pipe(int fd[2])
```

Returns two file descriptors in `fd`:

- `fd[0]` – the read end of the pipe
- `fd[1]` – the write end of the pipe

The sender should close the read end

The receiver should close the write end

If the receiver reads from an empty pipe, it blocks until data is written at the other end

If the sender attempts to write to a full pipe, it blocks until data is read at the other end

Processes must have child-parent relationship

pipe() example

```
int main(int argc, char *argv[]) {
  int fd[2]; char buf;
  assert(argc == 2);
  if (pipe(fd) == -1) exit(1);

  if (fork() != 0) {
    close(fd[0]);  /* Parent = writer, so close read */
    write(fd[1], argv[1], strlen(argv[1]));
    close(fd[1]);
    waitpid(-1, NULL, 0);
  } else {
    close(fd[1]); /* Child = reader, so close write */
    while (read(fd[0], &buf, 1) > 0)
      printf("%c", buf);
    printf("\n");
    close(fd[0]);
  }
}
```

```
$ ./a.out abc
abc
```

UNIX Named Pipes (FIFOs)

Persistent pipes that outlive the process which created them

Stored on the file system

Any process can open it like a regular file

- *Why use named pipes instead of files?*

```
$ mkfifo /tmp/abc
$ echo ABC >/tmp/abc
```

```
$ cat /tmp/abc
ABC
```

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 sockets:

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

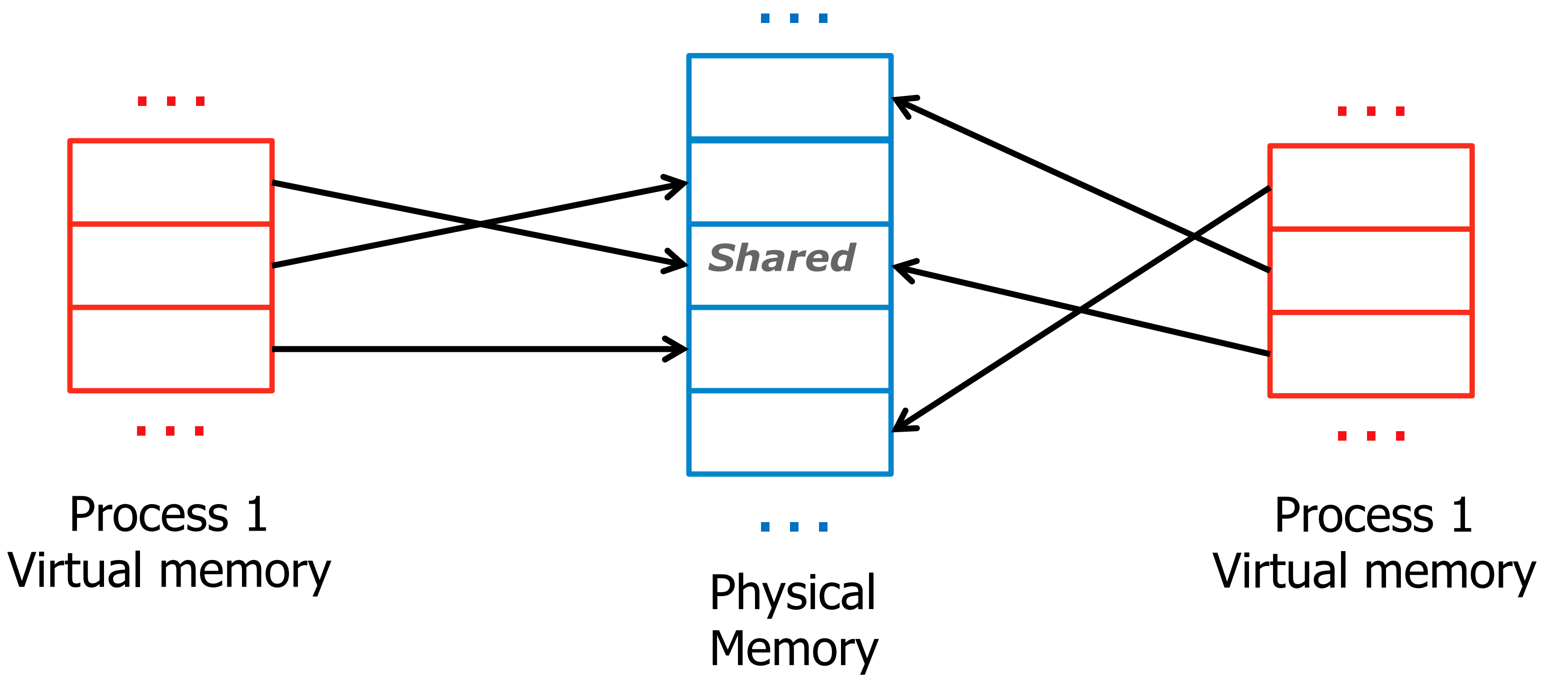
Covered in Networks and Distributed Systems course

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`	Attaches a shared memory segment to the address space of a process
`shmctl`	Changes the properties associate with a shared memory segment
`shmdt`	Detaches a shared memory segment from a process

Synchronisation

Process Synchronization

How do processes synchronize their operation to perform a task?

Key concepts:

- Critical sections
- Mutual exclusion
- Atomic operations
- Race conditions
- Synchronization mechanisms
  - Locks, semaphores, monitors, etc.
- Deadlock
- Starvation

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

Shared Data Example

Account #1234: £10,000

Extract £1000
from account 1234

Extract £1000
from account 1234

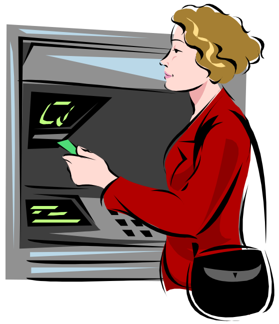
Shared Data Example

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

`Acc[1234]` `10,000`

`B = 10,000`
`Acc[1234] = 9000`

`B = 9,000`
`Acc[1234] = 8000`



`Extract(1234, 1000)`

`Extract(1234, 1000)`

Shared Data Example

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

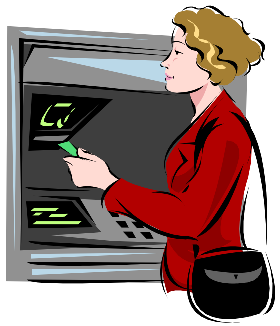
Critical section!
Need **mutual exclusion**

Acc[1234] 10,000

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 shared memory location with a value which depends on what it read

Mutual exclusion ensures that if a process is executing its critical section, no other process can be executing it

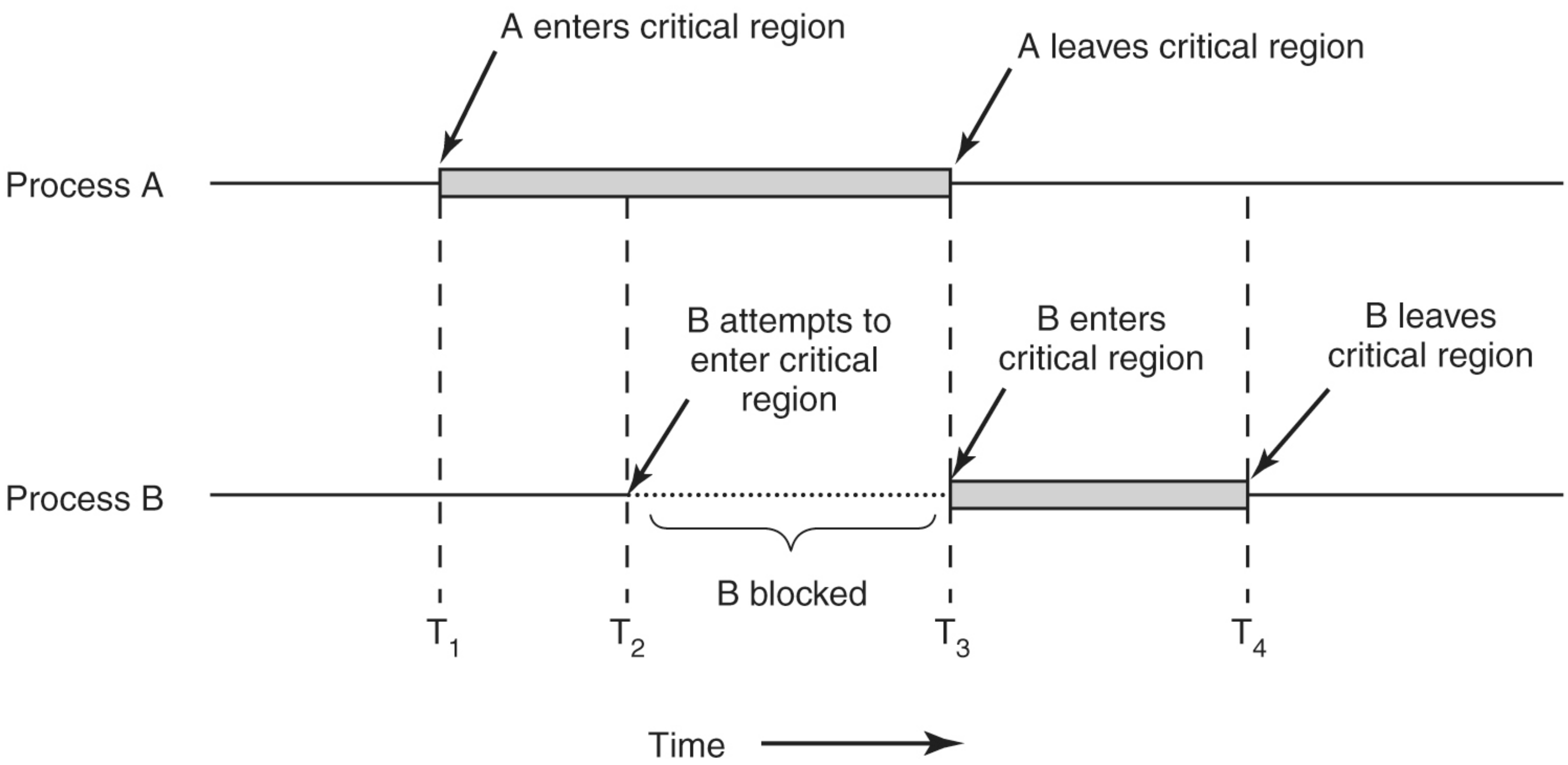
- Processes must request ***permission*** to enter critical sections

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 process is inside a critical section, any process requesting permission to enter must be allowed to do so immediately
- No process requiring access to its critical section 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[acc_no] = B - sum;
  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

P_0 turn $\boxed{0}$ **P_1**

```
while (true) {
   while (turn != 0)
     /* loop */ ;
   critical_section()
   turn = 1;
   noncritical_section0();
}
```

```
while (true) {
   while (turn != 1)
     /* loop */ ;
   critical_section()
   turn = 0;
   noncritical_section1();
}
```

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

Atomic Operations

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

```
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_no];
  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

TSL L	Read L and set condition code if L=0
BNZ	jumps if Z is not set.
MOV #n,L	sets 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 following scenario:

- H is waiting for I/O
- L acquires lock A and enters critical section
- 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[acc_no];
  Acc[acc_no] = B - sum;
  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[acc_no];
  Acc[acc_no] = B - sum;
  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: a measure of the 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 AddAccount(int acc_no, int 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);`

P3: `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 **`Extract`** example → final value of account 8,000 or 9,000

Thread Interleavings

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

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)

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 will continue if it has received a specific signal

Semaphores are *special variables*, accessible 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`**

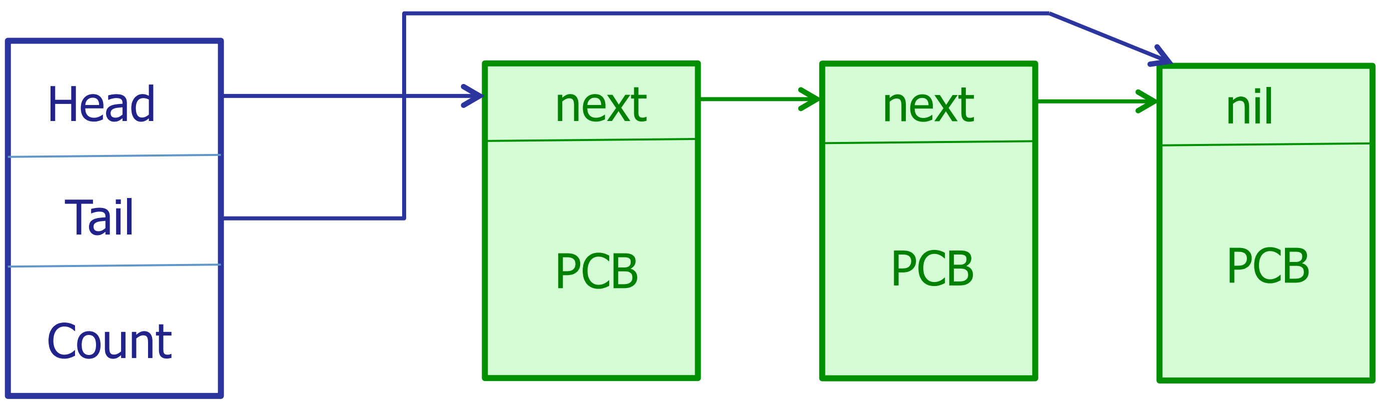
down() also called **P()** (*probeer te verlagen)*

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) ::= if counter(s) > 0
               counter(s) = counter(s) - 1
            else
               add P to queue(s)
               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

```
process A                              process B
  ...                                    ...
  down(s)                                down(s)
    critical section                       critical section
  up(s)                                  up(s)
end                                    end

main() {
  var s:Semaphore
  ...
  init(s, 1) /* initialise semaphore */
  ...
    start processes A and B in random order
  ...
}
```

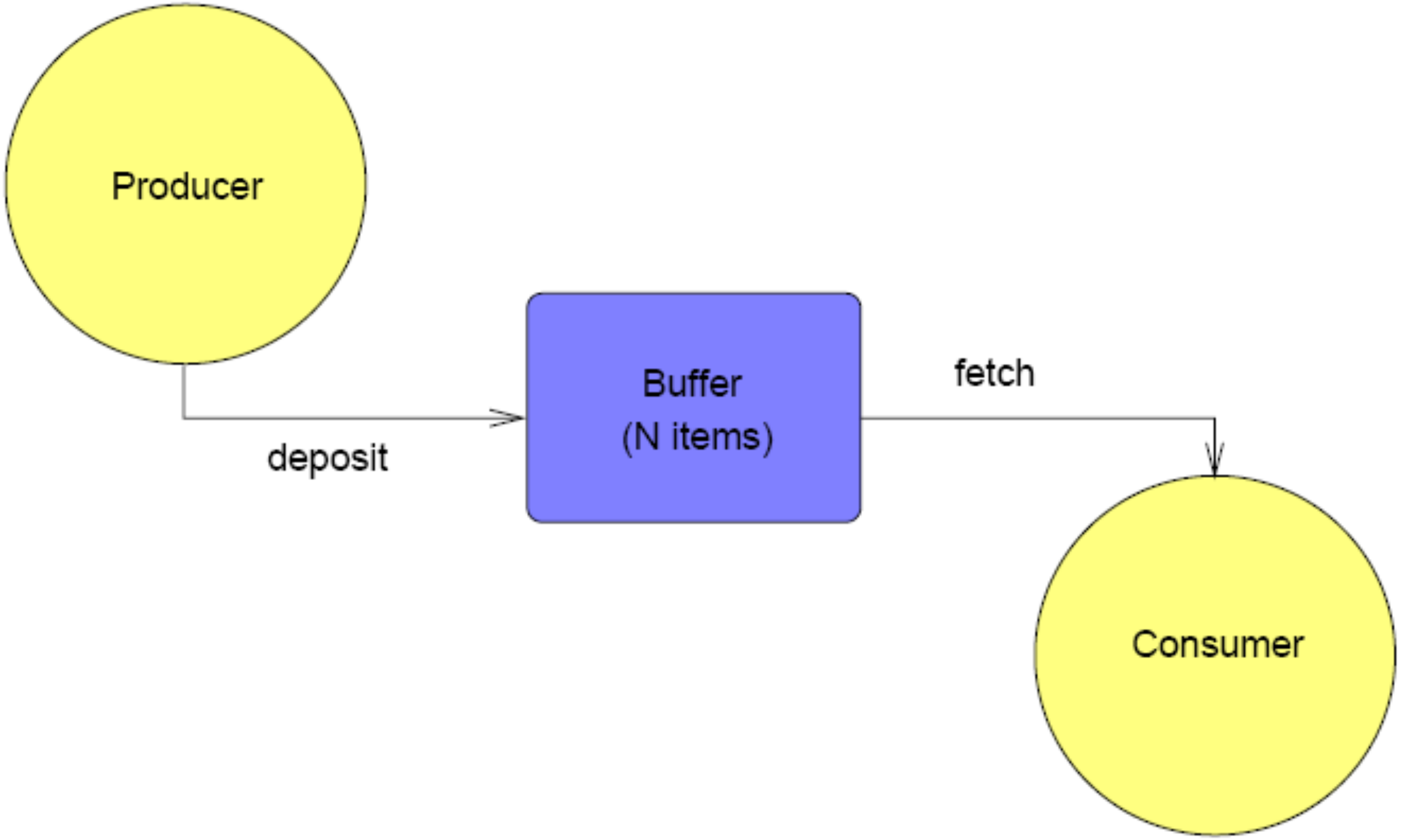
Note: for binary semaphore if s =1, up (s) leaves s = 1

General Semaphores

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

`counter(s) >= 0:` how many processes can execute down without being blocked

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 (items in buffer < N)
- Items can only be deposited in buffer 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
    down(mutex)
    down(item)
    fetch item
    up(space)
    up(mutex)
    consume item
  end loop
end Producer
```

What is wrong with this?

Producer/Consumer

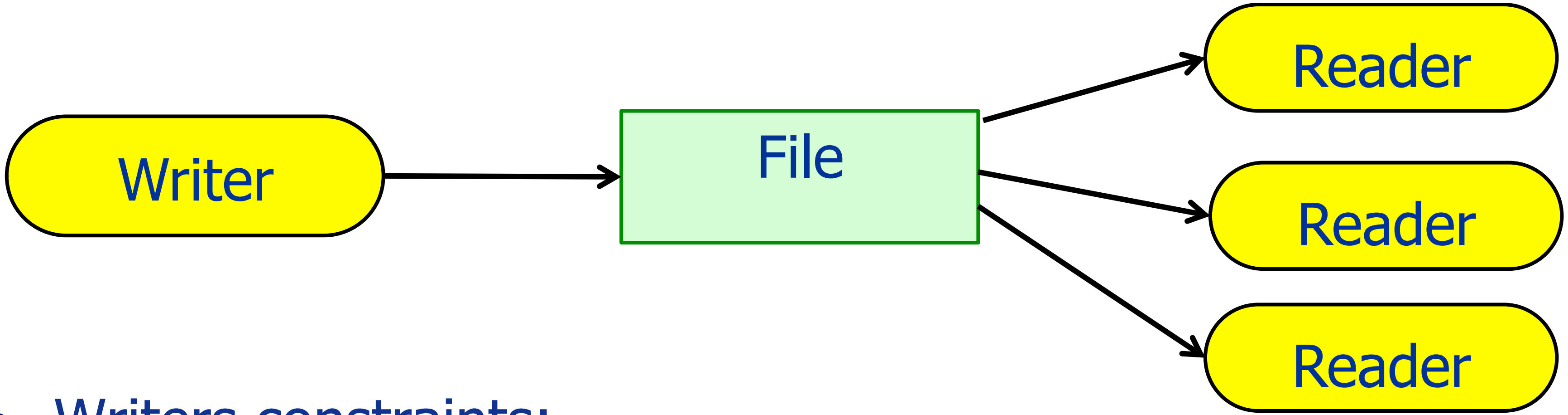
```
var item, space, mutex: semaphore
init (item, 0)      /* Semaphore to ensure buffer is not empty */
init (space, N)     /* Semaphore to ensure buffer is not full */
init (mutex, 1)     /* Semaphore to ensure mutual exclusion */
```

```
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)
    fetch item
    up(mutex)
    up(space)
    consume item
  end loop
end Producer
```

Works for multiple producers & consumers
What happens when space = 0 or items = 0?

Readers/Writers



- Writers constraints:
  - items can only be written if no other process is writing;
  - items can only be written if no other process 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.

47

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)
        down(wrt);
    up(mutex);
    read item
    down(mutex);
    read_cnt -= 1
    If (read_cnt == 0)
       up(wrt);
    up(mutex);
    consume item
  end loop
end reader
```

Is this fair?

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 procedures

- Can be called only from monitor procedures

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)**: releases monitor lock and waits for **c** to be signalled
- **signal(c)**: wakes up one process waiting 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 signal and waking up from a wait are not atomic

– More difficult to understand, need to take extra 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 procedure ***F*** will be replaced by

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

Mutual exclusion within a monitor is ensured

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_count++;
if (next_count > 0) up(next);
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--;
}
```

Producer/Consumer with Monitors

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

    entry procedure insert(item)
       if (count == N) wait(not_full);
       insert_item(item); count++;
       signal(not_empty);

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

Does this work?

Producer/Consumer with Lampson Monitors

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

    entry procedure insert(item)
       while (count == N) wait(not_full);
       insert_item(item); count++;
       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 a solution:

- 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 possible fair solution:

- Once a reader is waiting, readers will get in next.
- If a writer is waiting, one writer will get in next.

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 == 1 or WaitWriters > 0)
    {
      ++WaitReaders; Wait(CanRead); --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 methods
- no condition variables
  - wait() and notify()

Bohr and Heisen bugs

Bohrbugs:

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

Heisenbugs

- Non-deterministic, hard to reproduce bugs
  - ✦ Often caused by race conditions
- Suffer from the observer effect (Heisenberg 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: code in which process accesses shared resource

Mutual exclusion: only 1 process at a time within CS

Disabling interrupts: may not be effective

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