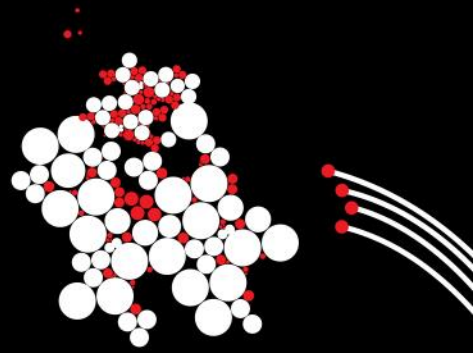
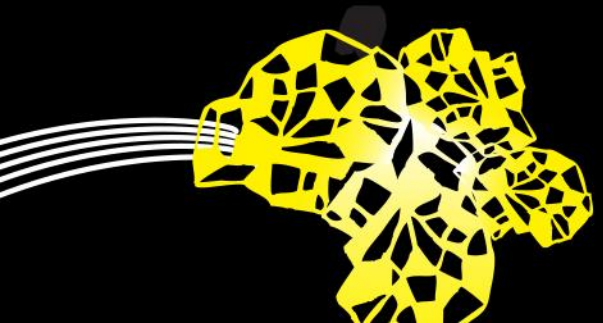
UNIVERSITY OF TWENTE.



OPERATING SYSTEMS

CONCURRENCY

ERIK TEWS <e.tews@utwente.nl>



CONCURRENT PROGRAMMING

THE POTENTIAL FOR PARALLELISM

- Set of sequential processes competing for shared resources
- The parallelism is abstract:
  - Omit unnecessary detail such as precise timing
  - Only assume an order on the execution of statements
  - No physically separate processor required (but supported)

RELATED CONCEPTS

- **Multiprogramming** - independent processes (on a single core system)
- **Multiprocessing** - using more than one core at the same time
- **Parallel programming** – using (physically) separate processors
- **Real-time programming** - absolute timing

OTHER WAYS TO MAKE YOUR CODE PARALLEL

- **SISD** – Single Instruction Single Data
  - This is probably what you are doing now
  - You can do bit level parallelism here
- **SIMD** – Single Instruction Multiple Data
  - Execute a single instruction on multiple data (ADD, MULT)
- **MIMD** – Multiple Instructions Multiple Data
  - Execute different instructions on multiple data
  - To some extend that what Intel tried with the Itanium (IA64) arch

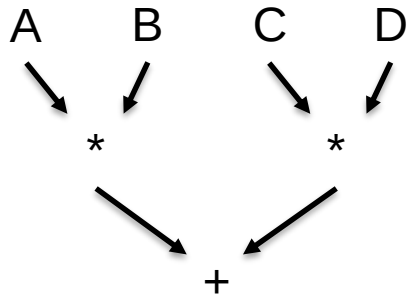
RUNNING PARTS OF YOUR CODE IN THE BACKGROUND

- **Async-I/O**
  - Work with multiple I/O requests at the same time
- **(GPU) offloading**
  - Offload certain tasks to other (dedicated) parts of the hardware
  - Might work with some networking cards as well

PRINCIPLE OF MODULAR DESIGN

CONCURRENCY CAN BE USED TO STRUCTURE COMPUTATION

- Divide and conquer
  - Example: Quicksort



- Pipeline

```
cat index.html|sort|uniq -c|sort -rn|pr -2
2012-07-23 16:08                                Page 1
     11                          1   <TBODY>
      9   <TR>                   1   </TBODY>
      9   </TR>                  1 <TABLE border=1>
```

BASICS : INTERLEAVING

process 1	process 2
INC(N)	DEC(N)

process 1		process 2	
load	R,N	load	R,N
add	R,#1	sub	R,#1
store	R,N	store	R,N

JAVA THREAD EXAMPLE WITH A RACE CONDITION

- Output?
- javac Count.java
- java Count

```
import java.util.concurrent.*;

class Count extends Thread {

    public int inc;
    public Count(int inc) {
        this.inc = inc;
    }

    static int ctr = 0;
```

```
    public void run() {
        int loc;
        for(int i = 0; i < 10; i++) {
            loc = ctr + inc;
            System.out.println(inc + "\t" + loc);
            ctr = loc;
        }
    }

    public static void main(String [] args) {
        Count p = new Count(1);
        Count q = new Count(-1);
        System.out.println("inc\tctr\n");
        p.start();
        q.start();
        try { p.join(); q.join(); }
        catch(InterruptedException e) { }
    }
}
```

PROBLEMS AND SOLUTIONS

Race conditions	→	Mutal exclusion
Deadlocks	→	Monitoring, Tools
Starvation	→	Priority scheduling
Errors are hard to reproduce	→	Tools, Logic, Abstractions
Real time requirements	→	Restrictions, Predictions

MUTUAL EXCLUSION

- Software: Semaphores, Message passing
- Hardware: Compare and Swap
- Tooling: Monitors

Edsger Dijkstra

SEMAPHORE DEFINITION

- A semaphore S is a variable with a natural number value on which two atomic operations are defined:

  - Wait(S) if S>0 then S:=S-1
    else suspend execution of the current process on S.
  - Signal(S) if processes are suspended on S then notify one of them
    else S:=S+1

SEMAPHORE INVARIANTS

- A semaphore satisfies the following invariants:

  a) $S \geq 0$

  b) $S = S_0 + \#Signals - \#Waits$

where

- S_0 is the initial value of S
- #Signals is the number of executed Signal(S) operations
- #Waits is the number of completed Wait(S) operations

COUNTING WITH SEMAPHORE

```
import java.util.concurrent.*;

class Count extends Thread {
    public int inc;
    public Count(int inc) {
        this.inc = inc;
    }
    static int ctr = 0;
    static Semaphore s =
        new Semaphore(1);
```

```
public void run() {
        int loc;
        for(int i = 0; i < 10; i++) {
            try { s.acquire(); }
                catch(InterruptedException e) {}
            loc = ctr + inc;
            System.out.println(inc + "\t" + loc);
            ctr = loc;
            s.release();
        }
    }

    public static void main(String [] args) {
        Count p = new Count(1);
        Count q = new Count(-1);
        System.out.println("inc\tctr\n");
        p.start(); q.start();
        try { p.join(); q.join(); }
        catch(InterruptedException e) { }
    }
}
```

Critical Section!

SEMAPHORE PROPERTIES

- Can we prove (informally) that:
  - Mutual exclusion (i.e. exclusive access to a resource)
  - No deadlock (i.e. no processes waiting for each other)
  - No starvation (i.e. eventually allowed to enter the critical section)

```
a0: initialise semaphore S to 1;

while(true) {
 a1:      Non_Critical_Section_1;
 b1:      Wait(S);
 c1:      Critical_Section_1;
 d1:      Signal(S);
}
```

```
while(true) {
 a2:      Non_Critical_Section_2;
 b2:      Wait(S);
 c2:      Critical_Section_2;
 d2:      Signal(S);
}
```

COUNT EXAMPLE IN C

- Output?
- gcc Count.c -lpthread
- ./a.out

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t Mutex;
int Ctr = 0;

int main(int argc, char * argv[]) {
    int targ[2] = {1,-1};
    pthread_t thread[2];
    sem_init(&Mutex, 0, 1);
    pthread_create(&thread[0], NULL,
        (void *) &tproc,(void *) &targ[0]);
    pthread_create(&thread[1], NULL,
        (void *) &tproc,(void *) &targ[1]);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    sem_destroy(&Mutex);
    return 0;
}
```

```c
void tproc(void *ptr) {
    int i, loc, inc = *((int *) ptr);
    for(i = 0; i < 10; i++) {
        sem_wait(&Mutex);
        loc = Ctr+inc ;
        printf("%d\t%d\n", inc, loc);
        Ctr = loc ;
        sem_post(&Mutex);
    }
    pthread_exit(0);
}
```

Critical Section!

HARDWARE: COMPARE AND SWAP INSTRUCTION

- How many function calls per second?
- gcc Spinlock.c -lpthread
- ./a.out

```c
int Lock = 1;

void *tproc(void *ptr) {
    sleep(1);
    Lock = 0;
    pthread_exit(0);
}
```

```c
int main(int argc, char * argv[]) {
    int i ;
    pthread_t thread;
    pthread_create(&thread, NULL, &tproc, NULL);
    for(i=0; !__sync_bool_compare_and_swap(&Lock, 0, 2); i++) {
        sched_yield();
    }
    pthread_join(thread, NULL);
    printf("%d\n",i);
    return 0;
}
```

EVALUATION OF THE HARDWARE SOLUTION

Advantages

- Simple
- Usable where memory is shared

Disadvantages

- Busy waiting costs CPU time
- Starvation is possible

SEMAPHORES IN ACTION

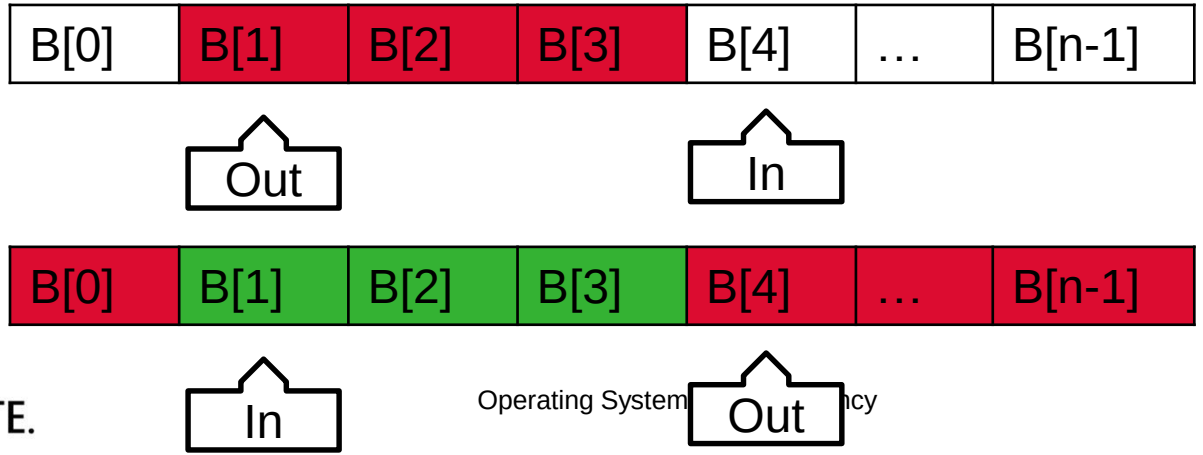
PRODUCER CONSUMER PROBLEM WITH A BOUNDED BUFFER

- ≥ 2 processes as follows:
  - ≥ 1 Producer
  - 1 Consumer

Cyclic buffer

In : next location where data can be written

Out : next location where data can be read



ONE PRODUCER AND ONE CONSUMER

- How to initialise semaphores?
- gcc ProdCons.c -lpthread

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <assert.h>
#define N 4
#define M 10
sem_t Elements, Spaces;
int B[N];
int In = 0, Out = 0;

void *tprod(void *ptr) {
    int i;
    for(i=0;i<M;i++) { /* Produce(i) */
        sem_wait(&Spaces);
        B[In] = i;
        printf("   B[%d] := %d\n", In, i);
        In = (In + 1) % N;
        sem_post(&Elements);
    }
    pthread_exit(0);
}
```

```c
void tcons(void *ptr) {
    int i,k;
    for(i=0;i<M;i++) {
        sem_wait(&Elements);
        k = B[Out];
        printf("%d := B[%d]\n", k, Out);
        Out = (Out + 1) % N;
        sem_post(&Spaces);
        assert( i==k ) ; /* Consume(k) */
    }
    pthread_exit(0);
}
```

MANY PRODUCERS AND ONE CONSUMER

- Output?
- Why the Mutex? Try without…
- Why no Mutex in the Consumer?
- gcc ProdManyCons.c -lpthread
- ./a.out

```
void *tprod(void *ptr) {
    int i,k,arg=*((int *) ptr);
    for(i=0;i<M;i++) {
        k = i+arg; /* Produce(k) */
        sem_wait(&Spaces);
        sem_wait(&Mutex);
        B[In] = k;
        printf("B[%d]=%d\n", In, k);
        In = (In + 1) % N;
        sem_post(&Mutex);
        sem_post(&Elements);
    }
    pthread_exit(0);
}
```

MONITORS

- “Semaphores are the goto’s of concurrent programming” (why?)
- Monitors are a structuring concept:
  - Mutual exclusion is automatic
  - Synchronisation is programmed
- A monitor in Java is a Class with:
  - private for all variables
  - synchronized for all methods
- Every monitor object has:
  - public final void notify[All]()
  - public final void wait()

MONITORS IN JAVA

- unzip ProdCons.zip
- cd ProdCons
- javac *.java
- java ProdCons

- Can this be done in C?

```
class Buffer{
    private int []B;
    private int Cnt = 0, In = 0, Out = 0;

    Buffer(int size) {
       B = new int[size];
    }

    public synchronized void Put(int i) {
        while(Cnt == B.length) {
            try{ wait(); }
            catch(InterruptedException e) { }
            finally{ }
        }
        B[In] = i;
        System.out.println("   B[" + In + "]=" + i);
        In = (In + 1) % B.length;
        Cnt++;
        notify();
    }
}
```

SEMAPHORES VS MESSAGE PASSING

Semaphores :

- Centralised
- Needs shared memory
- Synchronous
- No communication
- Programmed mutex and synchronisation

Message passing :

- Decentralised
- Can be used over a network
- Synchronous/asynchronous
- One way communication
- Automatic

SYNCHRONOUS VS ASYNCHRONOUS

Synchronous :

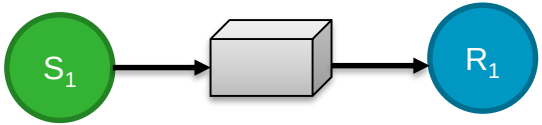
- Sender and receiver wait for each other
- Low level

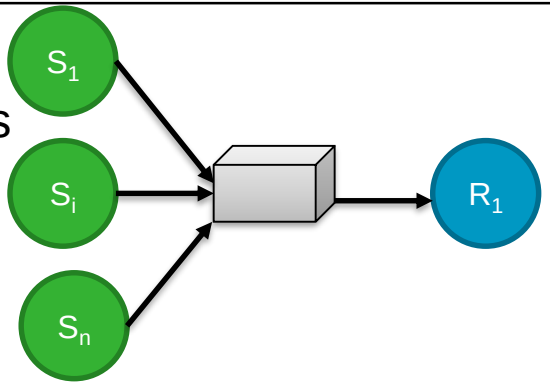
Asynchronous :

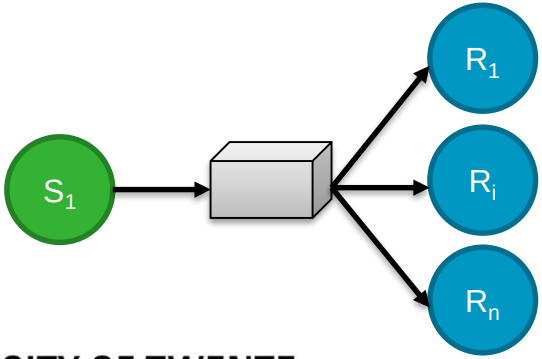
- Four combinations possible, but sender does not normally wait
- High level

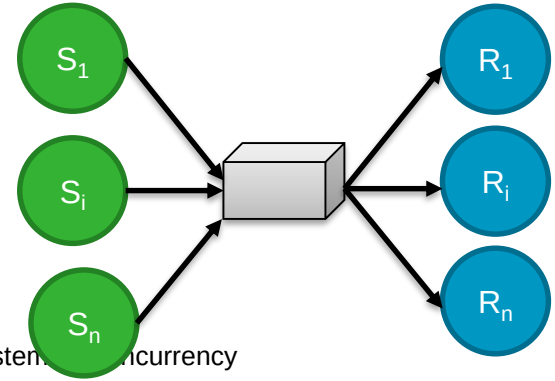
PROCESS IDENTIFICATION

- Dedicated channels, like dedicated phone lines









MESSAGE PASSING

- Output?
- No global variables!
- gcc MesPass.c
- –lrt –lpthread
- Where is the queue?

```
for(i=0;i<N*P;i++) {
    mq_receive(mqd,buf,M,0);
    k = atoi(buf);
    printf("recv %d\n",k);
    sum += k;
}
```

```
mqd_t my_open(int oflags) {
    mode_t mode = S_IRUSR | S_IWUSR |
        S_IRGRP | S_IWGRP | S_IROTH |
        S_IWOTH;
    char *path = queue_name;
    struct mq_attr attr = {0, N, M, 0};
    return mq_open(path,oflags,mode,&attr);
}

void *tprod(void *ptr) {
    int i,arg=*((int *) ptr);
    mqd_t mqd = my_open(O_WRONLY);
    char buf[M];
    for(i=0;i<N;i++) {
        sprintf(buf,"%d",arg+i);
        mq_send(mqd,buf,strlen(buf)+1,1);
        printf("%*csend %s\n", arg+4,' ',buf);
    }
    mq_close(mqd);
    pthread_exit(0);
}
```

NON BLOCKING SEND

- Output?
- gcc –DNONBLOCK MesPass.c –lrt -lpthread

```
void *tprod(void *ptr) {
    int i,arg=*((int *) ptr);

#ifdef NONBLOCK
    mqd_t mqd = my_open(O_WRONLY | O_NONBLOCK);
#else
    mqd_t mqd = my_open(O_WRONLY);
#endif
    …
}
```

SUMMARY

- Mutual exclusion requires:
  - Only one process at the time in the critical section
  - Any process may halt outside the critical section
  - A process wanting to enter the critical section will eventually do so
  - No assumptions about relative execution speeds
  - A process may remain in the critical section for a finite amount of time
- Many design choices
  - Programmed/automatic synchronization or communication
  - Centralized/decentralized
  - Synchronous/asynchronous
  - Named/anonymous channels