



T-106.420  
Concurrent Programming  
Semaphores



# Spin lock ie. busy-wait solutions to CS

```
...                               # {count == 0 }
while FA(count) skip; # tmp = count; count+=1; ret tmp;
CS;                               # {count == 1 }
count =1;
...
```

- complex without HW-support, like FA-machine instruction
- HW-mechanisms are not fair, so no liveness guarantees can be given
- inefficient - especially under heavy contention
  - in multitasking (processor sharing) CPU cycles are burned totally in vain
  - in multiprocessing memory contention may slow down all processors
- => processor should taken away from the process in a busy-wait and given to another one that can progress
- synchronization mechanism has to be combined with CPU scheduling
- an OS-kernel issue.



# Semaphores: syntax and semantics

- a *semaphore* is a synchronization primitive provided and implemented in the OS-kernel.
- semaphore is a shared, nonnegative integer variable operated exclusively with two operations,  $P$  and  $V$ .
- with  $P(s)$  a process decrements  $s$  ( $> 0$ ) by one. If  $s$  is 0, the process is put to wait “in  $P(s)$ ” until  $s$  is positive so that  $s$  can be decremented and the process may proceed.
- with a  $V(s)$  a process increments  $s$  by one. If  $s$  was 0, and there are one or more processes waiting “in  $P(s)$ ”, one of them may complete  $P(s)$  and proceed.



# Semaphores: syntax and semantics (2)

- Declaration

```
sem s;
```

```
sem lock = 1; /* with initialization*/
```

```
sem forks[5] = ([5] 1); /* array with init */
```

- Semantics can be defined in terms of :

```
P(s) : ⟨await (s>0) s = s-1;⟩ //## SEM: s>=0
```

```
V(s) : ⟨s = s+1;⟩
```

- Two ways to use a semaphore in a designing synchronization algorithms:

- a semaphore is assumed to have values  $\geq 0$  (*general semaphore*)

- a semaphore is confined to be either 0 or 1 (*binary semaphore*)



# semaphores

## basic problems and techniques

- semaphores can be used in many ways:
  - mutual exclusion, i.e. critical sections
  - simple forms of conditional synchronization
  - critical sections
  - barriers
  - producers/consumers
  - bounded buffers
  - ...



# semaphore solution to critical section

CS using  $\langle$ await $\rangle$  statements:

```
bool lock=false;      # {lock <=> "cs is occupied"}
process p {
  while (true) {
     $\langle$ await (!lock) lock=true;  $\rangle$  /* entry */
    critical section;           /* {lock == true}
    lock=false;                 /* exit */
    noncritical section;
  }
}
```

CS using semaphores:

```
sem mutex = 1;        # {mutex ==0 <=> "cs is occupied"}
process p {
  while (true) {
    P(mutex)           /* entry */
    critical section;  /* {mutex ==0}*/
    V(mutex);         /* exit */
    noncritical section;
  }
}
```



## Global invariants for semaphores

- initialisation: sem  $s = S_0$ ;
- $s$  is always nonnegative:  
 $\{s \geq 0\}$
- $\#P(s), \#V(s)$  = number of *finalized* P- and V-operations on  $s$
- semaphore invariant which is always true for any  $s$ :  
 $\{s = S_0 - \#P(s) + \#V(s)\}$
- Exercise: Prove that critical section solution provides mutual exclusion using the semaphore invariant.



## semaphores for synchronizing producers and consumers

- The producer-consumer problem involves delivering something between the two groups of participants through a shared buffer.
- The *producers* will perform the operation *deposit* and the *consumers* the operation *fetch*.
- *Multiple* producers and consumers
- *Buffer Invariant* : How to synchronize access to the shared buffer so that:
  - no messages overwritten
  - messages received only once



## A producer and a consumer for a 1-element buffer

```
typeT buf;          /* a buffer of some type T*/
sem empty=1, full=0;

process Producer {
  while (true) {
    ...           /* produce data, and put it to buffer */
    P(empty);
    buf = data;
    V(full);
  }
}

process Consumer {
  while (true) {
    P(full);
    result = buf;
    V(empty);
    ...           /* fetch result, then consume it */
  }
}
```



## Split binary semaphores

- *split binary semaphore* :
  - a way to use more than one semaphore in one design so that at any time at most one of them is 1 and others are 0, i.e. the sum of their values is always confined to be either 0 or 1.
  - lets call the individual semaphores as *components* of the split semaphore
- design rules:
  - initially one of the component is assigned to 1, and others are 0.
  - for *every* history, (= trace) of the system the P and V operations on the split semaphore *alternate*, i.e. each P-operation on one of the components is followed by a V on one of the components and vice versa.
    - This is can normally be checked by static (compile time) analysis of the code, i.e. for each execution path P and V alternate.
- generalised mutual exclusion property:
  - all statements between any P and the next V are executed in *mutual exclusion*, i.e they form (generalised) critical sections w.r.t. each other.
- alternative signaling at the exit of the generalised critical section:
  - At the exit of each critical section several alternative V-operations may be possible.



## Proof for a 1-element buffer

```
C(empty): (p = c and (all i: i ≤ p : data[i] = result[i]))
C(full): (p = c+1 and buff = data[p] and ((all i: i < p: data[i] = result[i]))
```

```
type T buf;          /* a buffer of some type T */
sem empty=1, full=0;

process Producer {
  while (true) {
    ...              /* produce data, and put it to buffer */
    P(empty);        /* {C(empty)}
    buf = data;      /* p=p+1 {C(full)}
    V(full);
  }}

process Consumer {
  while (true) {
    P(full);         /* {C(full)}
    result = buf;    /* c=c+1 {C(empty)}
    V(empty);
    ...             /* fetch result, then consume it */
  }}
}}
```



# We have seen this before: producer/consumer synchronization

- **one element buffer: buf**
- ```
int buf, p=0, c=0;
process Producer {
    int a[n];
    while (p<n) {
        while not (p==c); #buffer empty
        buf = a[p];
        p=p+1;
    }
}
process Consumer {
    int b[n];
    while (c<n) {
        while not (p>c); #buffer full
        b[c]=buf;
        c=c+1;
    }
}
```

Note: Because the synchronisation conditions  $(p==c)$  and  $(p>c)$  are atomic (at-most-once), we can evaluate them in a while loop. (In a uniprocessor this kind of busy wait would not make too much sense.)

*Consistency requirement I :*

$(c \leq p \leq c+1)$  and  
 $(p = c \Rightarrow b[c]=a[p])$  and  
 $(p > c \Rightarrow buff = a[p])$ .

*and also:*

$(p = c) \Rightarrow (\text{for all } i: i < p: (b[i] = a[i]))$   
 $(p = c = n) \Rightarrow (\text{all } i: i < n: (b[i] = a[i]))$



## n-element bounded buffer

- producer and consumer execution is often bursty: with a too small buffer producer and consumer can't proceed freely, but they have to synchronize to each other tightly, by each data unit, causing a lot of context switches
- in such cases a large enough buffer will increase performance significantly
- let us assume just one producer and one consumer, and an n-element array *buf* that is used in a circular fashion
- producer deposits a message with data as follows  
`[rear]=data; rear=(rear+1)%n`
- consumer fetches a message into its local var *result* as  
`result=buf[front]; front=(front+1)%n;`
- when "half full" deposit and fetch can be executed simultaneously,
- only when the buffer is empty/full the consumer/producer has to wait.



## n-element bounded buffer with one producer and one consumer

```
typeT buf[n];          /* an array of some type T*/
  int front=0; rear=0;
  sem empty=n, full=0; /* n-2 <= empty+full <=n */

process Producer {
  while (true) {
    ...
    /*produce data, then deposit it in the buffer*/
    P(empty);
    buf[rear]=data; rear=(rear+1)%n
    V (full);
  }
}

process Consumer {
  while (true) {
    /*fetch result then consume it */
    P(full);
    result=buf[front]; front=(front+1)%n;
    V(empty);
    ...
  }
}
```



## n-element buffer with several concurrent producers and consumers

```
typeT buf[n];          /* an array of some type T */
int front=0; rear=0;
sem empty=n, full=0;  /* n-2 <= empty+full <=n */
sem mutexD=1, mutexF=1; /* mutual exclusion separately
                           for producers and consumers */

process Producer [i=1 to M] {
  while (true) {
    ...
    /* produce data, then deposit it in the buffer */
    P(empty);
    P(mutexD);
    buf[rear]=data; rear=(rear+1)%n
    V(mutexD);
    V(full);
  }
}

process Consumer [j=1 to N] {
  while (true) {
    /* fetch result then consume it */
    P(full);
    P(mutexF);
    result=buf[front]; front=(front+1)%n;
    V(mutexF);
    V(empty);
    ...
  }
}
```



## dining philosophers problem

- problem description:

```
process Philosopher [i=0 to 4] {
    while (true) {
        acquire forks;
        eat;
        release forks;
        think;
    }
}
```

- solution: map each fork with one semaphore to guarantee exclusive access:

```
sem fork[5]=(1,1,1,1,1);
process Philosopher [i=0 to 4] {
    while (true) {
        P(fork[i]); P(fork[i+1]); /*get left, then right fork*/
        eat;
        V(fork[i]); V(fork[i]);
        think;
    }
}
```

- problems: - potential deadlock (why ?)  
- not very optimal w.r.t. fork usage (why?)
- in general: - a set of processes competing for an exclusive access for a set of resources  
- each resource is allocated independently with non-preemption.



## semaphores solving the dining philosophers problem

- solution: break the circular wait by introducing asymmetry

```
sem fork[5]=(1,1,1,1,1);
process Philosopher [i=0 to 3] {
  while (true) {
    P(fork[i]); P(fork[i+1]); /* get left fork, then right */
    eat;
    V(fork[i]); V(fork[i]);
    think;
  }
}
process Philosopher [4] {
  while (true) {
    P(fork[0]); P(fork[4]); /* get right fork, then left */
    eat;
    V(fork[0]); V(fork[4]);
    think;
  }
}
```

- not too nice because of the "hardcoded" and static asymmetry. The symmetry should be broken with more dynamics and using paramateric data.
- exercise: this is still a very non-optimal (why?), try to make an optimal solution.



# readers and writers

## selective mutual exclusion

- problem:
  - reader and writer processes share a database
  - a writer process must have exclusive access
  - if no writer is accessing the DB, any number of readers may concurrently execute
- solution
  - approach as exclusion problem
  - approach as condition synchronization problem



## readers and writers as exclusion problem (1)

- A simple minded approach: each reader and writer has exclusive access:

```
sem rw=1;
process Reader [i=0 to M] {
  while (true) {
    ...
    P(rw);      /* grab exclusive access lock */
    read the DB;
    V(rw);      /* release the lock          */
  }
}
process Writer [j=1 to N] {
  while (true) {
    ...
    P(rw);      /* grab exclusive access lock */
    write the DB;
    V(rw);      /* release the lock          */
  }
}
```

- This is a *overconstrained* solution for read access, let's try to relax the constraints



## readers and writers: readers as one group

- Handle readers as now group: the first reader grabs lock and it's released only after last reader finishes:

```
int nr=0;                /* number of active readers    */
sem rw=1;
process Reader [i=0 to M] {
    while (true) {
        ...
        <nr=nr+1;
        if (nr==1)P(rw)          /* if first, get lock    */
        >
        read the DB;
        <nr=nr-1;
        if (nr==0)V(rw)          /* if last, release lock */
        >
    }
}
process Writer [j=1 to N] {
    while (true) {
        ...
        P(rw);                   /* grab exclusive access lock */
        write the DB;
        V(rw);                   /* release the lock        */
    }
}
```



## readers and writers: readers as one group

```
int nr=0;                /* number of active readers      */
sem rw=1;
sem mutexR=1;         /* cs btw readers to protect reader access to nr */
process Reader [i=0 to M] {
  while (true) {
    ...
    P(mutexR) ;
    nr=nr+1;
    if (nr==1)P(rw)      /* if first, get lock      */
    V(mutexR) ;
    read the DB;
    P(mutexR) ;
    nr=nr-1;
    if (nr==0)V(rw)     /* if last, release lock */
    V(mutexR) ; }}
process Writer [j=1 to N] {
  while (true) {
    ...
    P(rw);              /* grab exclusive access lock */
    read the DB;
    V(rw);              /* release the lock          */
  }}
}}
```

- Reader's preference: if a reader has access then another arriving reader has preference over a writer. Because this unbounded overtaking liveness of the writers can not be guaranteed.



## barrier synchronization

- Many iterative tasks can be speeded up by breaking the body of the loop to several concurrent parts
- At end of the loop the parts have to be synchronized for the next round in order to interchange the results in a consistent form
- a *barrier* is a synchronization point which all asynchronous processes must reach before any process is allowed to proceed
- In a synchronous parallel system (eg. SIMD) this is provided automatically
- Note: Remember the story about the missionary giving a farewell talk to the unfaithful village.



# barrier synchronization

- iterative algorithms often do the same computation on an array of values; terminate when answer is computed or results converge;

- the pattern is

```
while (true) {  
    co [i=1 to n]  
        code to implement task i;  
    oc  
}
```

- problem: This is inefficient, because n processes created and destroyed for each iteration will cause a huge overhead
- A solution with a static set of processes synchronized could be more efficient

```
process Worker [i=1 to n] {  
    while (true) {  
        code to implement task i;  
        barrier; /* wait for all n tasks to complete */  
    }  
}
```



# types of barrier synchronization

- shared counter
- flags and coordinators
- symmetric barriers



# barrier synchronization with shared counter $\langle \rangle$

```
int count =0;
process Worker[i=1 to n] {
  while (true) {
    code to implement task i;
     $\langle$ count=count+1; $\rangle$ 
     $\langle$ await (count==n) ; $\rangle$ 
  }
}
```



# barrier synchronization with shared counter; FA

```
int count =0;
process Worker[i=1 to n] {
    while (true) {
        code to implement task i;
        FA(count,1);
        while (count!=n) skip;
    }
}
```



# barrier synchronization with shared counter (2)

- problems:
  - *count* has to be reset to 0 after all processes have passed the barrier
  - *count* has to be reset before **any** process again tries to increment it (why ?)
  - shared counter
    - processes may have different speeds => in worst case  $n-1$  processes wait for the last => memory contention, need of cache update



# barrier synchronization with flags and coordinators

- modify implementation of *count* by introducing *arrive[1:n]*

```
count=count+1; replaced by  
arrive[i]=1
```

```
⟨await (count==n);⟩ replaced by  
⟨await ((arrive[1]+ ... + arrive[n])==n);⟩
```

## **global invariant**

```
count == (arrive[1]+ ... + arrive[n])
```

- problems:
  - reintroduces memory contention
  - *arrive[i]* has to be continually updated by every Worker



# barrier synchronization with flags and coordinators (2)

- solution
  - additional set of shared values *continue[1:n]* and *Coordinator* process
  - **Worker** waits for a *single value* to become true

```
arrive[i]= 1;  
⟨await (continue[i]==1);⟩
```
  - **Coordinator process**

```
for [i=1 to n] ⟨await (arrive[i]==1);⟩  
for [i=1 to n] continue[i]=1;
```
- *arrive* and *continue* are so-called *flag variables*



# barrier synchronization with flags and coordinators (3)

- flag synchronization principles
  1. the process that waits for a synchronization flag to be set is the one that should clear the flag
  2. a flag should not be set until it is known that it is clear
- for example
  - Worker [i]

```
arrive[i]= 1;  
<await (continue[i]==1);>  
continue[i]=0;
```
  - Coordinator process

```
for [i=1 to n]{ <await (arrive[i]==1);>  
arrive[i]=0; }  
for [i=1 to n] continue[i]=1;
```

clear arrive before  
setting continue



# barrier synchronization with flags and coordinators (4)

```
int arrive[1:n]=([n] 0), continue[1:n]=([n] 0);
process Worker[i=1 to n] {
  while (true){
    code to implement task i;
    arrive[i]=1;
    ⟨await (continue[i]==1);⟩
    continue[i]=0;
  }
}
process Coordinator {
  while (true){
    for [i=1:n]
      ⟨await (arrive [i]==1);⟩
      arrive[i]=0;
    }
    for [i=1:n] continue[i]=1);
  }
}
```



# barrier synchronization with flags and coordinators (5)

- problems
  - requires an extra process, Coordinator
  - execution time of coordinator is proportional to the number of Workers
- solution
  - make Worker also a Coordinator, e.g. organize them into a *tree*, i.e. *arrive* signals up the tree and *continue* signals down



# symmetric barriers

- symmetric  $n$ -process barrier is constructed from pairs of simple, two-process barriers
- *butterfly barrier*: in phase  $s \geq 1$  each process synchronizes with a process at a distance  $2^{s-1}$  from it, giving  $\lceil \log_2 n \rceil$  phases
- *dissemination barrier*: each process raises first the flag of its right-hand neighbour, waits for and then lowers the flag of its left-hand neighbour; in phase  $s$  the neighbours are at a distance of  $2^{s-1}$  from each other (circularly)



# Elegance

Aim at simple and scalable designs by applying:

- *symmetry*: all processes use the same code
- *distribution*: minimize the shared state space by dividing it between pairs of processes.



# semaphores implementing **barriers** ( used for signaling events)

- 2-process barrier
  - neither process can get past the barrier until both have arrived
  - the barrier must be reusable
  - uses one semaphore for each synchronization flag
- a semaphore is used for **signalling** an event, i.e. as a "synchronization flag" according to the following rules:
  - (usually) initialized to 0
  - a process "sets the flag" i.e. **signals** an event by **V(s)**
  - other processes **wait** for the event and "reset the flag" by **P(s)**



# barriers

```
sem arrive1=0, arrive2=0;
process Worker1 {
    ...
    V(arrive1);          /* signal own arrival to barrier */
    P(arrive2);          /* wait for other process   */
    ...
}
process Worker2 {
    ...
    V(arrive2);          /* signal own arrival to barrier */
    P(arrive1);          /* wait for other process   */
    ...
}
```

- Semaphores used as *signalling flags*:
  - to indicate when processes reach critical execution points, i.e. like flag variables, according to Flag Synchronization Principles
    - Exercise: Formulate the barrier property and prove it using the semaphore invariants