CSE 344 System Programming

Synchronizing with semaphores: classic problems

- Producer/consumer
- Dining philosophers
- Cigarette smokers
- Synchronization barrier
- Readers/writers
- Sleeping barber

Synchronization

Example: a stack consisting of a shared array T and an index S

```
push(v)          v ← pop
++S              v=T[S]
T[S]=v              --S
                 return v
```

Process 1 pushes | **Process 2 pops**

```
            ++S
context
 switch
                                   v=T[S]      // wrong
                                   --S
                                   return v
context
 switch
            T[S]=v      // wrong
```

Synchronization

There is **no way** of stopping process scheduling or predicting it reliably.

A **critical section** is a block of code using a shared resource, such as the stack in our example.

The golden rule is that at any given moment, there must be at most one process (or thread) inside a critical section.

This way even if the kernel schemes against us, no harm can come to our resources.

Synchronization

Assuming that m is a semaphore initialized to 1.

```
Process p1          Process 2
wait(m)             wait(m)
push(v)             v = pop()      // critical section
post(m)             post(m)
```

A semaphore acquiring only the values 0 and 1 is called a binary semaphore, or a mutex (**mut**ual **ex**clusion).

Generally semaphores are used to model “a number of available resources” (that’s why they are called **counting semaphores**).

Synchronization

The **producer-consumer** model is by far the most widely encountered synchronization model. A producer process produces data, and a consumer process consumes the said data.

1) Unbounded buffer case: the consumer process must execute only if there is something to consume in the buffer; otherwise it must wait.

The `full=0` semaphore represents the number of products in the buffer

Producer	Consumer
`while(true)`	`while(true)`
`    add(buffer, data)`	`    wait(full)`
`    post(full)`	`    take(buffer)`

That's fine; but what happens if both enter the buffer at the same time?

Synchronization

What happens if both processes enter the buffer at the same time?

Then the buffer becomes corrupted like our stack! We need to protect the access to the critical section through a mutex m = 1!

```
Producer                         Consumer

while(true)                      while(true)
    wait(m)                          wait(full)
    add(buffer, data)                wait(m)
    post(m)                          take(buffer)
    post(full)                       post(m)
```

We solved the underflow problem, but what about the overflow problem?

Synchronization

2) Bounded buffer case

Now we also have an upper limit to our buffer. The producer must not produce if the buffer is full! Empty spaces are now a resource too!

Semaphore `full`: number of products in the buffer

Semaphore `empty`: number of empty spaces in the buffer

Semaphore `m`: concurrent access lock

Initially

```
full=0      // no products
empty=N     // size of the buffer
m=1         // unlocked
```

Synchronization

```
Producer
while(true)
    wait(empty)
    wait(m)
    add(buffer, data)
    post(m)
    post(full)
```

```
Consumer
while(true)
    wait(full)
    wait(m)
    take(buffer)
    post(m)
    post(empty)
```

At any given moment `empty+full <= N`

Synchronization

The order of waits is crucial! Let's see what happens if we exchange them.

```
Producer                         Consumer
while(true)                      while(true)
    wait(m)                          wait(full)
    wait(empty)                      wait(m)
    add(buffer, data)                take(buffer)
    post(m)                          post(m)
    post(full)                       post(empty)
```

Imagine the producer getting the lock and then encountering a full buffer..the system will be blocked indefinitely!

Synchronization

What happens when the consumer needs more than 1 resource?

Process P1 needs 3 resources and process P2 needs 2 resources.

Initially we have s=2 resources.

	P1	P2
	`wait(s) // s=1`	
context switch		`wait(s) // s=0`
		`wait(s) // blocked`
context switch	`wait(s) //blocked`	
	`wait(s)`	

It's a pity, process 2 could have been served with 2 resources; now they'll have to wait until some other process calls `post`.

Synchronization

Calling wait k times is not the same as an atomic wait decreasing the semaphore by k.

This functionality is provided readily by System V semaphores; (POSIX semaphores can do it too, albeit indirectly :)

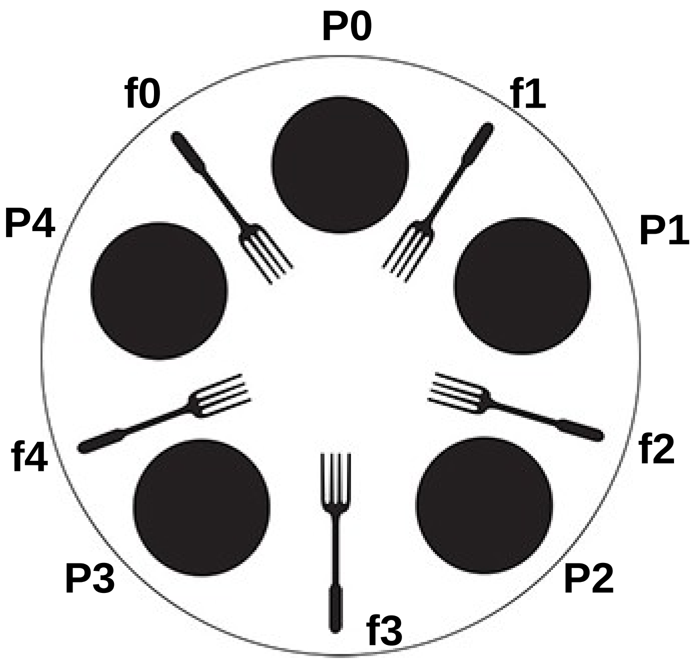
P1	P2
`wait(s,3)`	`wait(s,2)`
`// work`	`// work`
`post(s,3)`	`post(s,2)`

Synchronization

The **dining philosophers** is a classic synchronization problem introduced by Dijkstra.

Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers.

In order to eat, a philosopher needs to pick up the two forks that lie at the philosopher's left and right sides

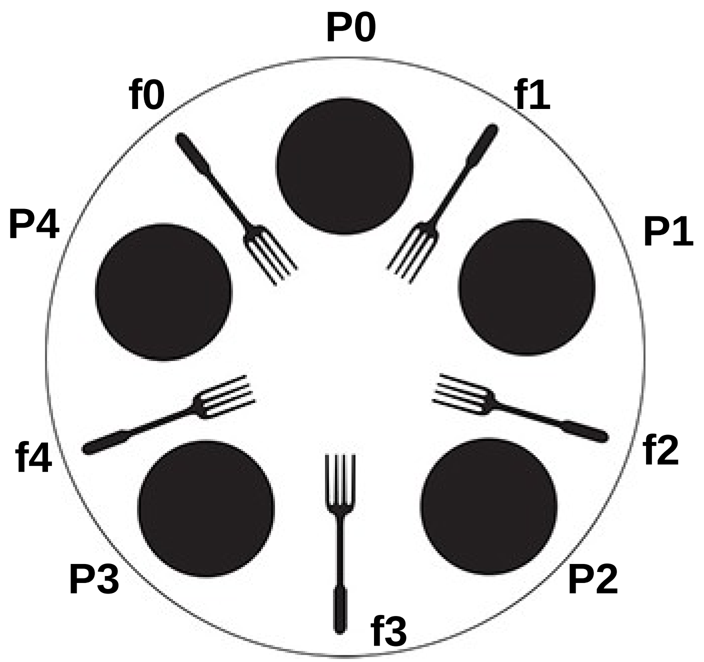


Synchronization

Each philosopher alternates between thinking (non-critical section) and eating (critical section).

Since the forks are shared, there is a synchronization problem between philosophers (processes or threads).

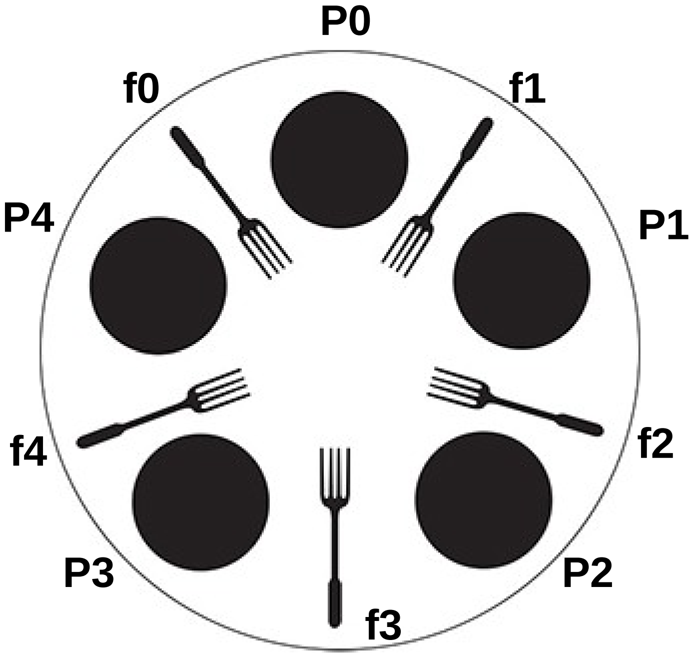
The forks are our shared resources, so we'll have a semaphore representing each of them.



Synchronization

A first attempt at solving the problem:

```
P2
think()
wait(f2)    // get fork
wait(f3)    // get fork
eat()       // crit. section
post(f2)    // put down fork
post(f3)    // put down fork
```



Synchronization

This scheduling can happen if the kernel dislikes you:

P0	P1	P2	P3	P4	
`wait(f0)`					
	`wait(f1)`				**context switches**
		`wait(f2)`			
			`wait(f3)`		
				`wait(f4)`	

All philosophers end up starving even though 2 of them could have been served.

Main challenge: a philosopher must either get both forks, if available, or otherwise none!

Synchronization

Easy to solve with IPC/System V semaphores

```
int forks = semget(666, 5, IPC_CREAT|ICP_EXCL|0600);
...
int getFork(int i){
   // for the i-th philosopher
   struct sembuf ops[2];
   ops[0].sem_num = i;          // semaphore to process
   ops[1].sem_num = (i+1) % 5;// semaphore to process
   ops[0].sem_op = ops[1].sem_op = -1;
   ops[0].sem_flg = ops[1].sem_flg = 0;
   return sem_op(forks, ops, 2);
}
```

Synchronization

Another classic problem are the **cigarette smokers** (1971).

Assume a cigarette requires three ingredients to make and smoke: tobacco, paper, and matches.

There are three smokers around a table, each of whom has an infinite supply of one of the three ingredients — one smoker has an infinite supply of tobacco, another has paper, and the third has matches.

Synchronization

There is also a non-smoking agent who enables the smokers to make their cigarettes by arbitrarily (non-deterministically) selecting two of the supplies to place on the table.

The smoker who has the third supply should remove the two items from the table, using them (along with their own supply) to make a cigarette, which they smoke for a while.

Once the smoker has finished his cigarette, the agent places two new random items on the table. This process continues forever.

The ingredients are resources so we’ll have one semaphore for each.

Synchronization

A first attempt

```
[has matches]
while(true)
   wait(tobacco)
   wait(paper)
   get_ingred.()   // crit. sect.
   smoke()
   post(done)
```

```
[agent]
while(true)
   post(tobacco)
   post(paper)
   wait(done)
```

Looks good…?

Synchronization

Looks good? No..

```
[has matches]
wait(tobacco)
wait(paper)
get_ingred.()
smoke()
post(done)
```

```
[has tobacco]
wait(paper)
wait(matches)
get_ingred.()
smoke()
post(done)
```

What if the agent brings **tobacco and paper**, but one smoker gets the tobacco and the other the paper? None will be able to smoke, the system will be deadlocked (good for the smokers, but for the system)!

Synchronization

Similarly to the dining philosophers, each smoker must either get both ingredients, if available, or otherwise none; in order to avoid effectively the deadlocks. e.g.:

```
[has matches]
while(true){
   wait(tobacco, paper)
   get_ingred.()
   smoke()
   post(done)
}
```

Synchronization

The **synchronization barrier** is another often encountered problem. We have N processes (or threads), and any of them reaching this point must stop and cannot proceed unless all other threads/processes have reached this barrier.

e.g. worker processes and a boss process: the boss process does not pay their salary, unless all workers have completed a required task.

```
boss                      worker x N

prepare()

                          work()
---------------------------------------------  Synchronization barrier
pay_salaries()
```

Synchronization

We have a single semaphore T initialized at N.

Every process/thread reaching the rendezvous point will call `wait` on it, and then wait for it to become zero. If T becomes zero, that means everyone has reached the barrier.

```
// worker
wait(T)
zero(T) // wait for T to become zero
```

`zero` is a call specific to System V/IPC semaphores.

Synchronization

```
int barr_init(int semid, int num, int N)
{return semctl(semid, SETVAL, N);}  // initialization


int barr_wait(int semid, int num){
   struct sembuf w,z;             // two distinct operations
   w.sem_num = z.sem_num = num;
   w.sem_flg = z.sem_flg = 0;
   w.sem_op = -1;                    // wait
   z.sem_op = 0;                     // zero
   return sem_op(semid,&w,1) !=-1 &&               // WAIT
               sem_op(semid,&z,1) !=-1? 0: -1;  // ZERO
}
```

Synchronization

The **readers-writers** is another classic synchronization problem (1971).

We have a shared resource that two types of processes (or threads) access:

- The readers: that do not modify the resource
- The writers: that modify the resource

Readers can access the data in any order and number they like. However at most one writer is allowed to write at any given moment. And of course no reader should be reading while a writer is writing.

Synchronization

<u>READER</u>

```
wait(mutex)          // avoid race
++readers
if(readers = = 1)    // 1st reader
   wait(rsc)         // no writers allowed
post(mutex)
read() // read the data
wait(mutex)
- - readers
if(readers = = 0)       // last reader
   post(rsc)            // let the writer enter
post(mutex)
```

<u>WRITER</u>

```
wait(rsc)
write()
post(rsc)
```

Initially: mutex=1, rsc=1

Synchronization

`readers`: the number of active readers

`rsc`: makes sure we have only one writer

`mutex`: makes sure the shared variable readers is modified safely

While a writer is writing, the first reader will be blocked at `wait(rsc)` and the subsequent ones at `wait(mutex)`

In the database world this is known as a “lock”.

Readers ask the Database Managament System (DBMS) for a “**shared lock**” and writers ask for a “**exclusive lock**”.

Synchronization

However, imagine the following scenario:

```
Reader1 is reading
Writer1 is blocked at wait(rsc)
Reader2 is reading
Reader1 exits (Writer1 is still blocked)
Reader3 and Reader4 are reading
Reader2 exits (Writer1 is still blocked)
Reader4 exits (Writer1 is still blocked)
Reader5 is reading…
```

i.e. if the readers are too many, a writer might have to wait indefinitely.

Synchronization

Solution: prioritize writers!

i.e. no writer, once added to the queue, shall be kept waiting longer than absolutely necessary. This is also called **writers-preference**.

This is accomplished by forcing every reader to lock and release a "*readtry*" semaphore individually. The writers on the other hand don't need to lock it individually.

Only the first writer will lock the "*readtry*" and then all subsequent writers can simply use the resource as it gets freed by the previous writer. The very last writer must release the "*readtry*" semaphore, thus opening the gate for readers to try reading.

Synchronization

READER

```
  wait(readTry)     // a reader is trying to enter
  wait(rmutex)      // avoid race condition with other readers
  readcount++;      //report yourself as a reader
  if (readcount == 1)     // if you are the first reader
    wait(rsc)             // lock  the resource and prevent writers
  post(rmutex);           // allow other readers
  post(readTry)           // you are done trying to access the resource
  read()
  wait(rmutex)            // avoids race condition with readers
  readcount--;            //indicate you're leaving
  if (readcount == 0)     // if you are last reader leaving
    post(rsc)             // you must release the locked resource
  post(rmutex)            //release exit section for other readers


readTry=1, rmutex=1, rsc=1, readcount=0
```

Synchronization

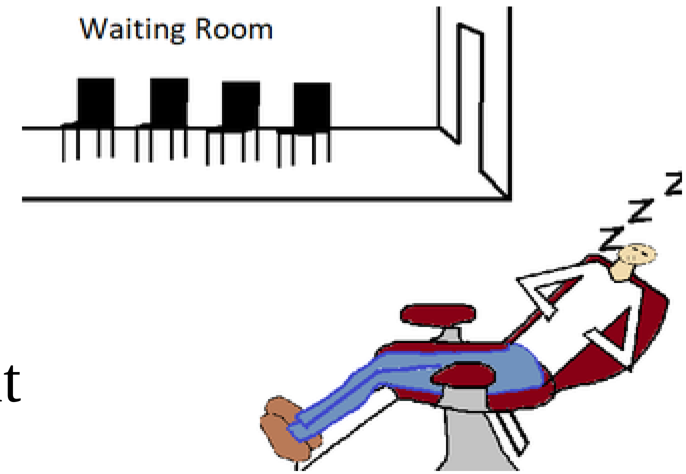
```
WRITER
  wait(wmutex)                // avoids race conditions
  writecount++;               // report yourself as a writer entering
  if (writecount == 1)        // if you're the first writer
    wait(readTry)             // no new readers!
  post(wmutex)                // release entry section
  wait(rsc)                   // prevents other writers
  write()
  post(rsc)                    // release resource
  wait(wmutex)                 // reserve exit section
  Writecount--;                // indicate you're leaving
  if (writecount == 0)         // checks if you're the last writer
    post(readTry)  // if you're the last writer, unlock the readers
  post(wmutex)

wmutex=1
```

Synchronization

Another famous problem is the **sleeping barber**; also attributed to Dijkstra (1965).

The barber shop has one barber and two rooms. The waiting room with N chairs, and the cutting room with a single chair.

Barber: When he finishes cutting a customer's hair, he dismisses the customer and goes to the waiting room to see if there are others waiting. If there are, he brings one of them back to the chair and cuts his hair. If there are none, he returns to the chair and sleeps in it



Synchronization

Customer: each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, the customer wakes him up and sits in the cutting room chair. If the barber is cutting hair, the customer stays in the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free chair, the customer leaves.

All actions (cutting hair, etc) can take an unknown amount of time. This can cause a lot of issues.

Synchronization

Issues: for instance a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While they're on their way, the barber finishes their current haircut and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to their chair and sleeps. The barber is now waiting for a customer, but the customer is waiting for the barber.

Or, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

Synchronization

// mutex; whether the barber is ready to cut or not

```
Semaphore barberReady = 0
```

// **mutex** to control access to the number of waiting room seats

```
Semaphore accessWRSeats = 1
```

// the number of customers currently waiting at the waiting room

```
Semaphore custReady = 0
```

// total number of seats in the waiting room

```
int numberOfFreeWRSeats = N
```

Synchronization

<u>Barber</u>

```
while(true)
    wait(custReady)  // acquire a customer - if none, sleep
    wait(accessWRSeats)          // there is a customer
    numberOfFreeWRSeats += 1  // increase # of free seats
    post(barberReady)            // ready to cut.
    post(accessWRSeats)     // no need for chair lock any more
    cutting_hair()
```

Synchronization

Customer

```
wait(accessWRSeats)              // access waiting room chairs
if(numberOfFreeWRSeats > 0)   // If there are any free seats:
   numberOfFreeWRSeats -= 1   // sit down in a chair
   post(custReady)          // notify the barber that there is a customer
   post(accessWRSeats)     // release the chairs' lock
   wait(barberReady)       // wait until the barber is ready
   have_haircut()
else                       // there are no free seats
   post(accessWRSeats)     // release the lock
   leave_without_a_haircut()
```