Classical IPC Problems

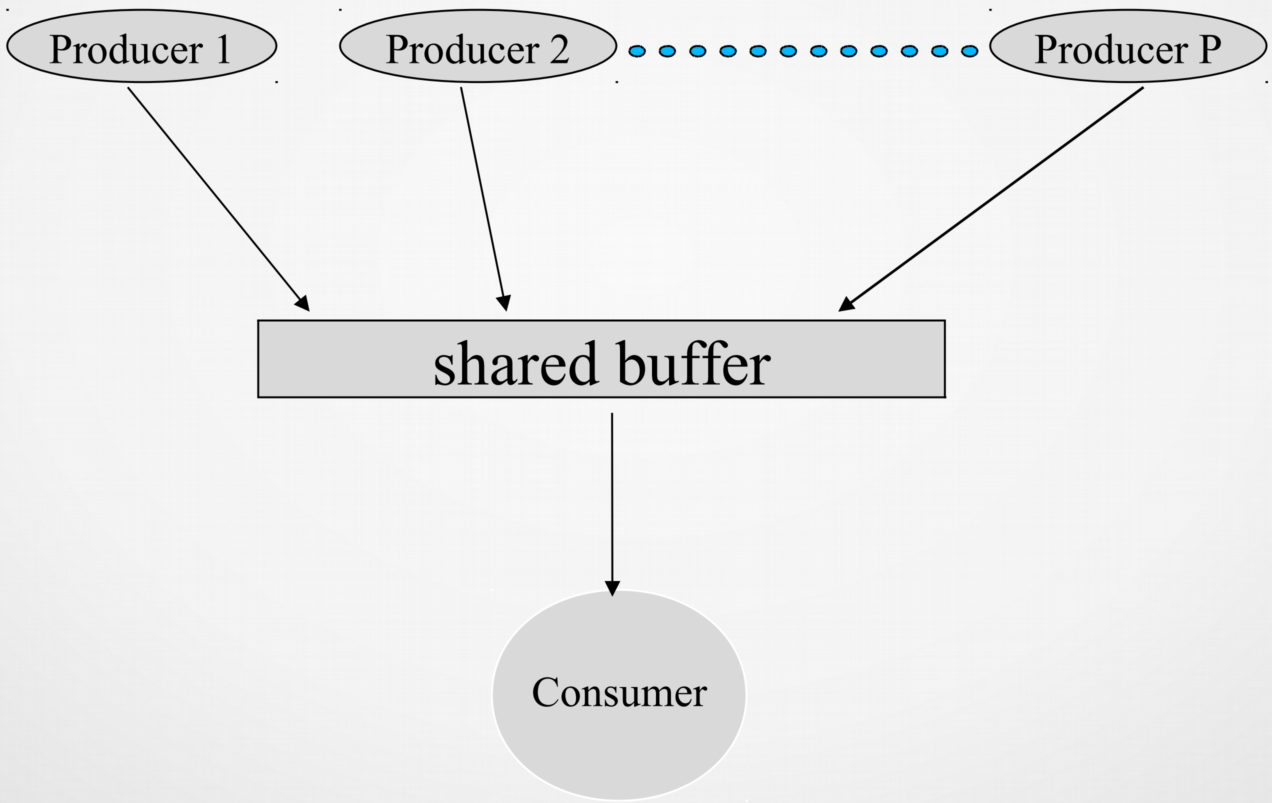
Computer Operating Systems
BLG 312E

2016-2017 Spring

Problems

- producer – consumer
- readers – writers
- dining philosophers
- sleeping barber

Producer - Consumer



Producer - Consumer

- access to shared buffer through mutual exclusion
- circular buffer
- if buffer empty → consumer waits (synchronization)

Producer – Consumer

- use counting semaphores
  - takes on ≥ 0 integers
  - used when resource capacity > 1
  - initial value = initial free resource capacity
  - P: one more unit of capacity in use
  - V: one unit of capacity freed

Producer – Consumer

- shared buffer implemented through a shared array of size N
  - array[N]
- binary semaphore:

  mutex $\leftarrow$ 1
- counting semaphores:

  full $\leftarrow$ 0 : number of full buffer locations

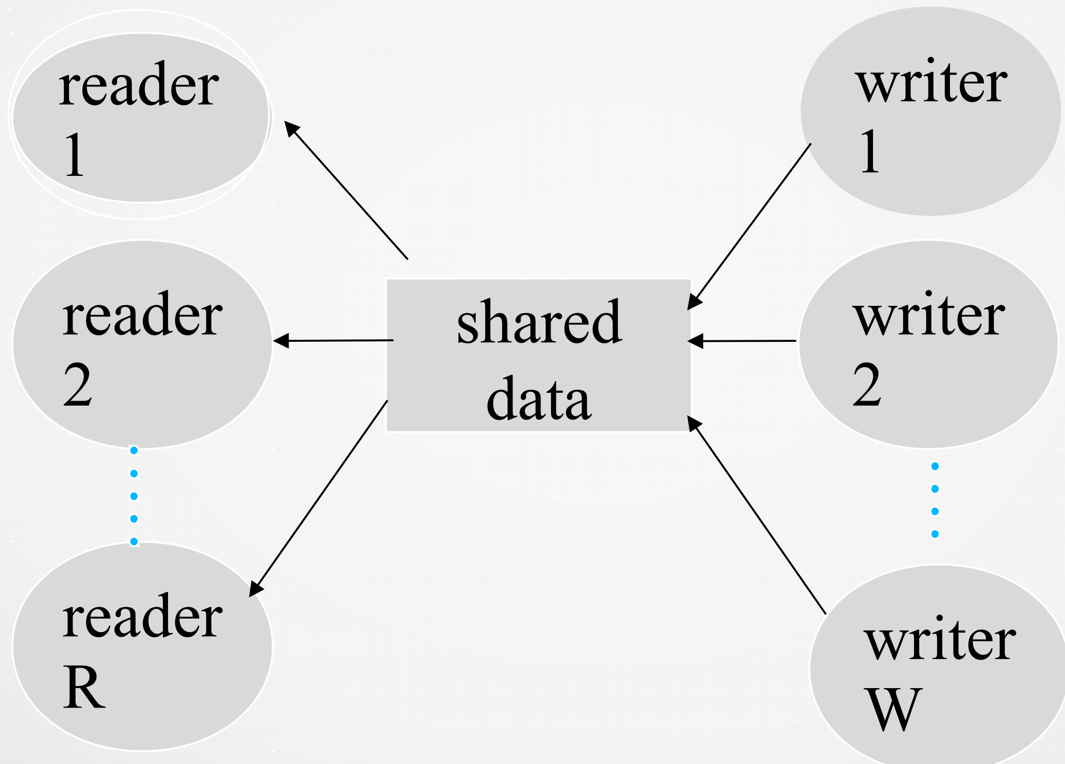
  empty $\leftarrow$ N : number of free **buffer locations**

```
constant N=100;
semaphore full=0, empty=N, mutex=1;
item array[N];
int in=0, out=0;
item data;
```

```
process producer(){
  while (true) {
    -- produce data –
    p(empty);
    P(mutex);
      array[in]=data;
      in=(in+1)%N;
    v(mutex);
    v(full);
  }
}
```

```
process consumer(){
  while (true) {
    p(full);
    p(mutex);
      data=array[out];
      out=(out+1)%N;
    v(mutex);
    v(empty);
    -- use data --
  }
}
```

Readers - Writers



Readers - Writers

- more than one reader may read shared data (no writers)
- when a writer uses shared data, all other writers and readers must be excluded

```
int reader=0;
semaphore read_mutex=1, data_mutex=1;
```

```
process reader() {
  while (true) {
    p(read_mutex);
      reader=reader+1;
      if (reader==1)
        p(data_mutex);
    v(read_mutex);
    -- read data --
    p(read_mutex);
      reader=reader-1;
      if (reader==0)
        v(data_mutex);
    v(read_mutex);
  }
}
```

```
process writer() {
  while (true) {
    -- execute --
    p(data_mutex);
    -- write data --
    v(data_mutex);
  }
}
```

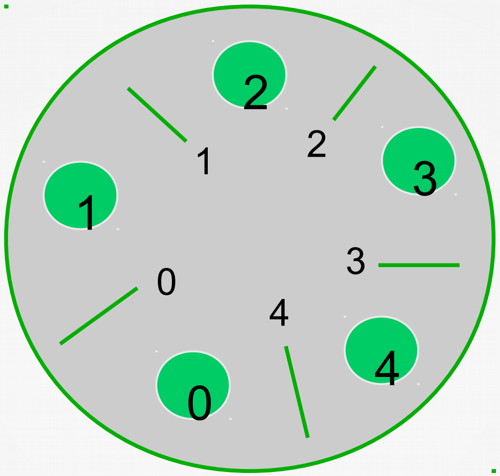
readers have priority over writers!

Possible indefinite postponement !

Readers - Writers

- must find a fair solution
- apply rules for access order:
  - if a writer is waiting for readers to be finished, do not allow any more readers
  - if a reader is waiting for a writer to finish, give reader priority

Dining Philosophers



Problem: share resources (forks) among philosophers without causing deadlock or starvation

Dining Philosophers

- philosophers
  - eat pasta
  - think
- philosophers need two forks to eat

Dining Philosophers

- fact: two philosophers sitting side by side cannot eat at the same time
  - e.g. for N=5, at most 2 philosophers can eat at the same time
- solution must provide maximum amount of parallelism

Dining Philosophers

```
philosopher(i) {
  while (true) {
    think();
    take_fork(i); //left fork
    take_fork((i+4)%5);//right fork
    --- eat -----
    leave_fork(i);
    leave_fork ((i+4)%5);
  }
}
```

what happens if all philosophers take their left forks?

Dining Philosophers

```
philosopher(i) {
  while (true) {
    think();
    take_fork(i); //left fork
    if (fork_free((i+4)%5)==FALSE)
       leave_fork(i);
    else {
      take_fork((i+4)%5);//right fork
      --- eat -----
      leave_fork(i);
      leave_fork ((i+4)%5);
    }
  }
}
```

is it possible that all philosophers starve?

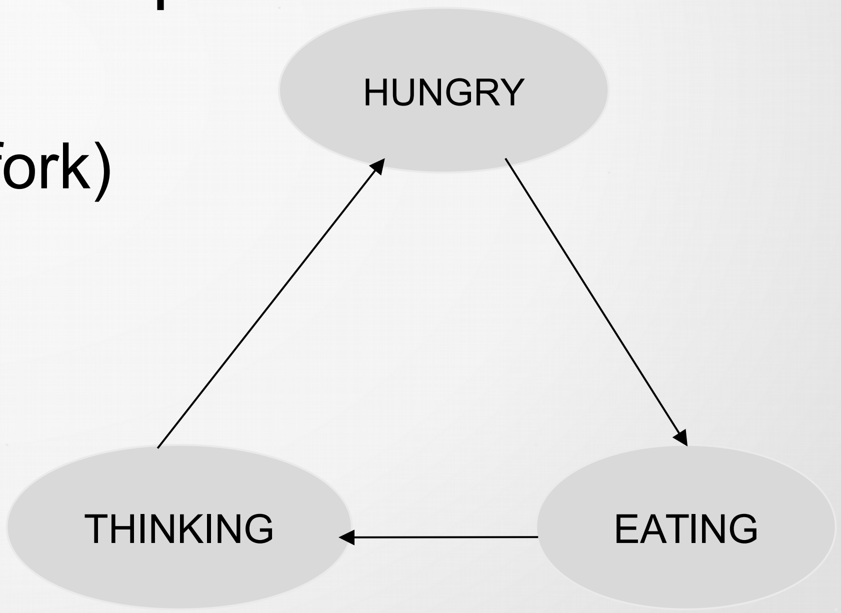
Dining Philosophers

```
philosopher(i) {
  while (true) {
    P(mutex); //binary semaphore
      think();
      take_fork(i); //left fork
      take_fork((i+4)%5);//right fork
      --- eat -----
      leave_fork(i);
      leave_fork ((i+4)%5);
    V(mutex);
  }
}
```

at most how many philosophers can eat together?

Dining Philosophers (Correct Solution)

- state[i] : state of *i*th philosopher
  - 0 : THINKING
  - 1 : HUNGRY (wait for fork)
  - 2 : EATING



Dining Philosophers (Correct Solution)

- a philosopher can be “EATING” only if both neighbors are <u>not</u> “EATING”
- use a binary semaphore per philosopher
  - blocks on semaphore if a fork is not available when requested

Variables:

- N=5 philosophers
- states:

  THINKING = 0

  HUNGRY = 1

  EATING = 2
- state[5]: array of size 5
- semaphores:

  mutex ← 1

  s[5] ← 0 array of size 5

```
process philosopher(i){
  while (true) {
    think();
    take_fork(i);
    --- eat ---
    leave_fork(i);
  }
}
```

```
take_fork(i) {
  P(mutex);
    state[i]=HUNGRY;//request to eat
    try[i]; //try to take forks
  V(mutex);
  P(s[i]); //blocks if can't take forks
}
```

```
leave_fork(i) {
  left=(i+1)%5;
  right=(i+4)%5;
  P(mutex);
    state[i]=THINK;
    try(left);
    try(right);
  V(mutex);
}
```

```
try(i) {
  left=(i+1)%5;
  right=(i+4)%5;
  if ((state[i]=HUNGRY) ∧
      (state[left]≠EATING) ∧
      (state[right]≠EATING))
  {
    state[i]=EATING;
    v(s[i]);
  }
}
```

Sleeping Barber

- in a barber shop
  - 1 barber
  - 1 customer seat
  - N waiting seats
- barber sleeps if there are no customers
- arriving customer wakes barber up
- if barber is busy when customer arrives
  - waits if waiting seats available
  - leaves if no waiting seats available

Sleeping Barber

- 3 semaphores needed for the solution
  - `customers` : number of customers waiting (excluding the one in the customer seat)
  - `barbers` : number of available barbers (0/1 in this problem)
  - `mutex` : for mutual exclusion

```
constant CHAIRS=5;
int waiting=0;
semaphore customers=0,barber=0,mutex=1;
```

```
process barber() {
  while(true) {
    P(customers); //sleep if no customers
    P(mutex);
      waiting--; //remove customer
      V(barber); //barber ready to cut hair
    V(mutex);
    -- cut hair -
  }
}
```

```
process customer() {
  P(mutex);
  if (waiting<CHAIRS){ //shop full?
    waiting=++; //admite customer
    V(customers); //wake-up barber (possibly)
    V(mutex);
    P(barber); //sleep if barber busy
    -- cut hair -
  }
  else
    V(mutex); //shop is full, so leave
}
```