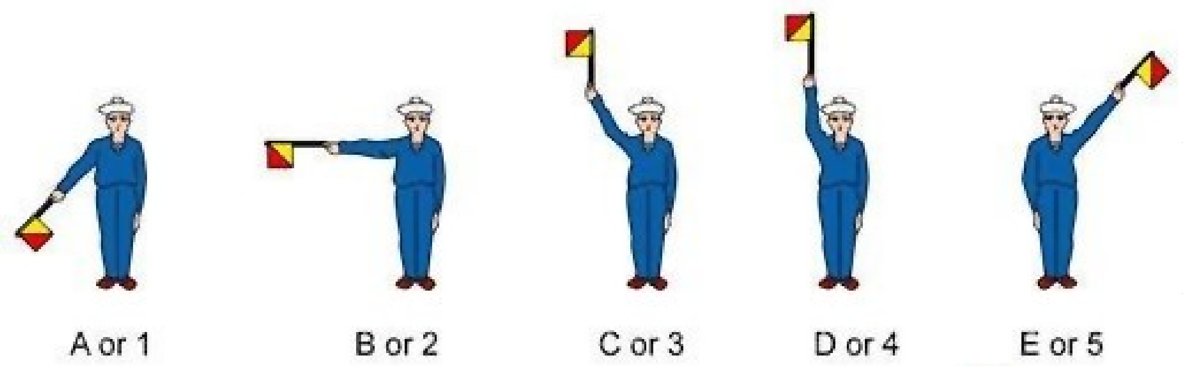
Lecture 3

Semaphores

Semaphores

- Summary: Last time
  - Shared update problem/critical section
  - Semaphores and locks



- Today
  - All you need to know about semaphores

Semaphore Specification

- An abstract datatype containing a nonnegative integer accessed by two atomic operations P and V

```
class Semaphore {

   private int sv;

   Semaphore(int init): <sv = init>
   P(s): <await (sv>0) sv = sv –1>
   V(s): <sv = sv + 1>
}
```

Critical Section – Semaphores

- JR has built in semaphores

```
sem mutex = 1;
process CS ((int i=0;i<2;i++)) {
   while (true) {
      //Non-critical section
      P(mutex);
      //Critical Section
      V(mutex);
   }
}
```

Critical Section – Semaphores

- Java has a library support
  - `java.util.concurrent`

```
Semaphore mutex = new Semaphore(1, true);

public void run() {
   while (true) {
      //Non-critical section
      mutex.acquire();
      //Critical Section
      mutex.release();
   }
}
```

Binary Semaphores and Locks

- A semaphore which only ever takes on the values 0 and 1 is called a *binary semaphore*
- When a binary semaphore s is used for simple mutex:

```
P(mutex);
//Critical Section
V(mutex);
```

  - it is also referred to as a lock.
    - P(s) – "acquiring the lock"
    - V(s) – "releasing the lock"

Java Built-In Locks

- A lock is created for every object in Java
- To use this lock we employ the keyword `synchronized`

```
class MutexCounter {
   private int counter = 0;

   public synchronized void increment() {
            counter++;
   }
}
```

The Dining Philosophers

- A bunch (N) of philosophers who spend their time thinking and eating
- Constraints:
  - Only N forks (university funding cuts!)
  - Can't eat with only one fork
  - May only take forks from left and right

TDP – General Program

- Write a program which simulates the behaviour

```
process Philosopher ((int i=0;i<N;i++)) {
   while (true) {
      Think
      Acquire forks
      Eat
      Release forks
   }
}
```

Purpose

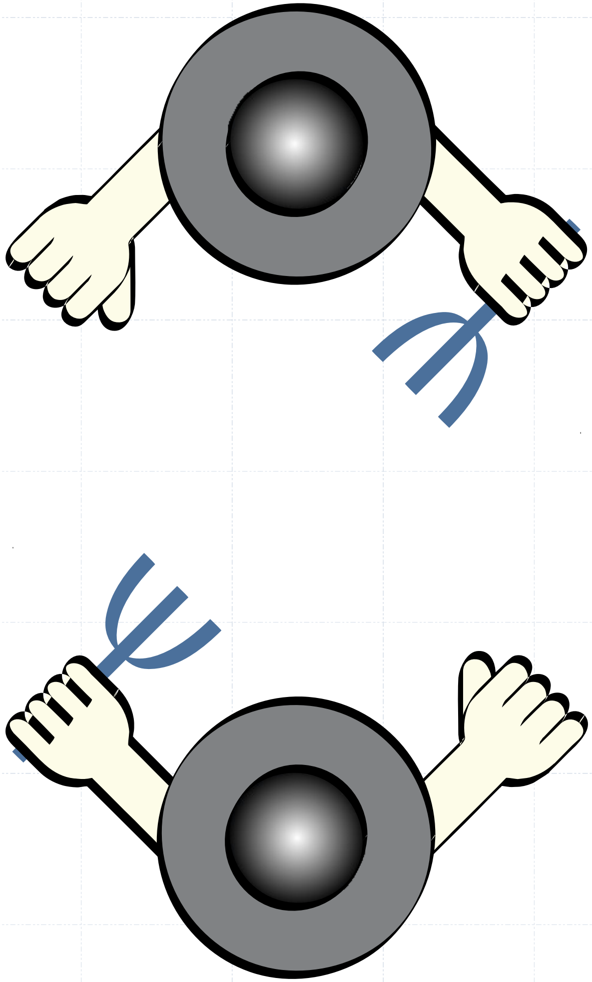
- Classical problem illustrating
  - Mutex: only one philosopher at a time may have a fork
  - Conditional synchronisation: may eat only when has two forks
  - Livelock
  - Deadlock
  - Indefinite postponement (starvation)

First Attempt

```
process Philosopher ((int i=0;i<N;i++)) {
   int left = i,
       right = (i+1)%N;
   while (true) {
      //Think
      P(forks[left])
      P(forks[right])
      //Eat
      V(forks[left])
      V(forks[right])
   }
}
```

Deadlock Problem

- If all manage to pick up their left-hand fork at about the same time then a deadlock occurs
- In general deadlock occurs because there is a circular waiting



Solution 1: Break the Symmetry

- Prevention by not allowing the circular waiting to arise.
- By making one of the philosophers different we break the cycle

```
process Philosopher ((int i=0;i<N;i++)) {
   int left, right;
   if (i==0) {   left = 1; right = 0;   }
   else {   left = i; right = (i+1)%N;   }
   …
```

Solution 2: General Semaphore

- If at most N-1 philosophers are eating then at least one will always have two forks.
- Use a general semaphore to represent N-1 available chairs

```
sem chairs = N - 1
…
    P(chairs)
    P(forks[left])
    P(forks[right])
    //Eat
    V(forks[left])
    V(forks[right])
    V(chairs)
…
```

Analysis

- Starvation with a fair scheduler?
  - Depends on the implementation of semaphores!
    - Blocked queue (FIFO) guarantees fairness
    - Other “queue” policy might not
- JR: FIFO
- Java
  - Default constructor: *no*
  - Semaphore(int permits, boolean fair)

A Simple Deadlock Prevention Method

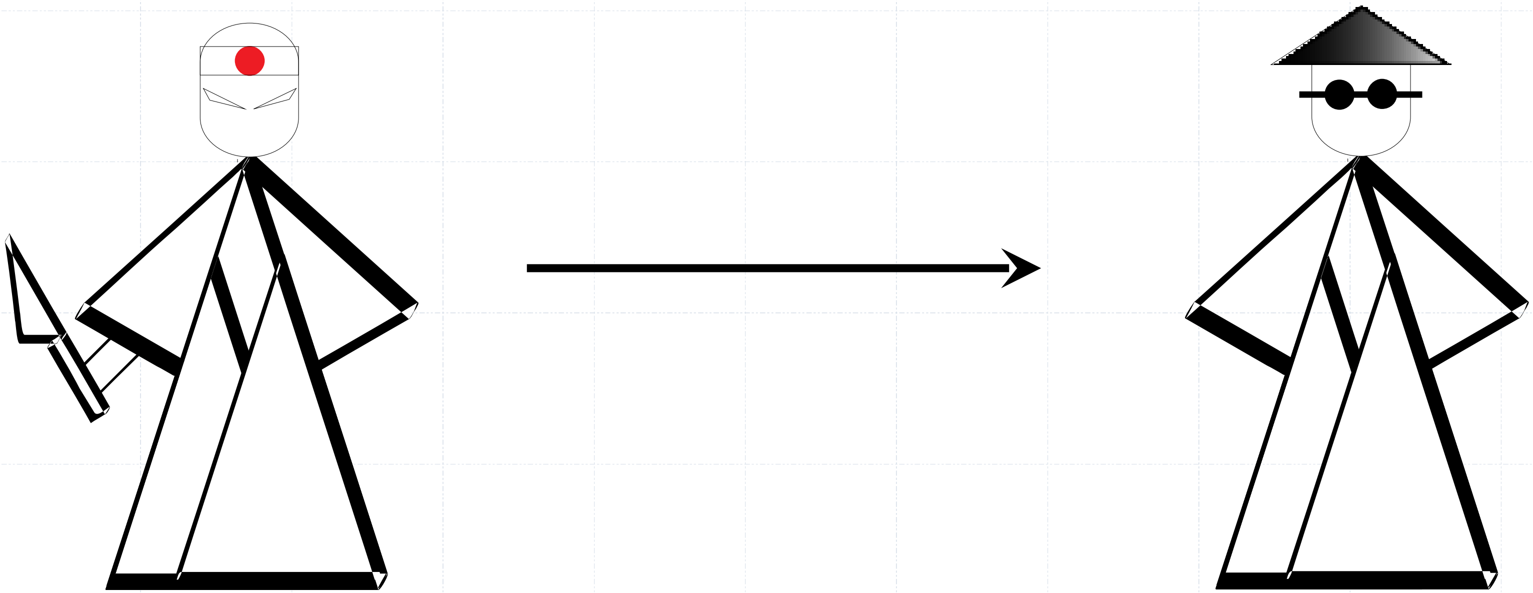
- One simple method for guaranteeing deadlock freedom when using locks for resource protection:
  - Assuming that locks are used correctly to provide mutually exclusive access to individual resources a, b, c, …
  - Fix a (linear) order for resources. Only allowed to possess multiple resources if they are acquired in that order
    - solution 1 of the dining philosophers

Preventing Deadlock

- Another simple for guaranteeing deadlock freedom
  - Identify the “circular waiting chains”
  - Don’t allow enough processes to “fill” the chain
    - solution 2 of the dining philosophers

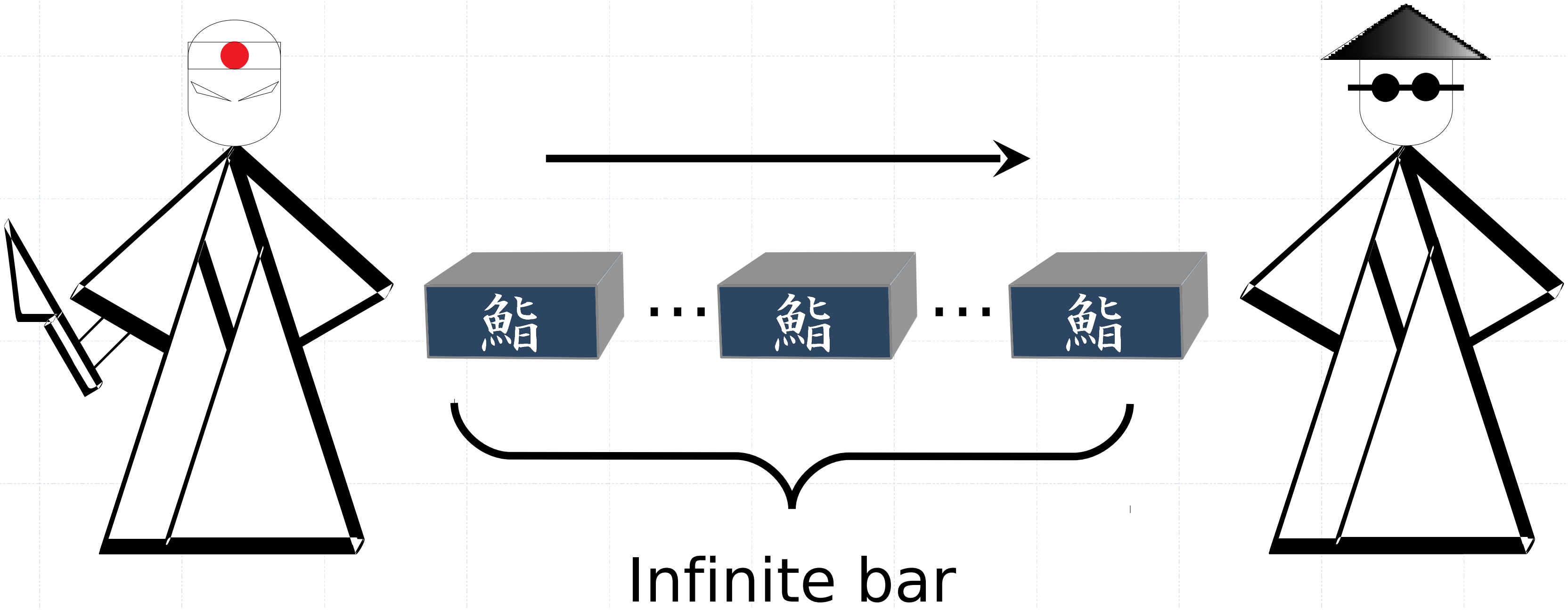
Producer Consumer

- Producer-consumer relationships between processes are a very common pattern
- Problem: how do we allow for different speeds of production vs consumption?



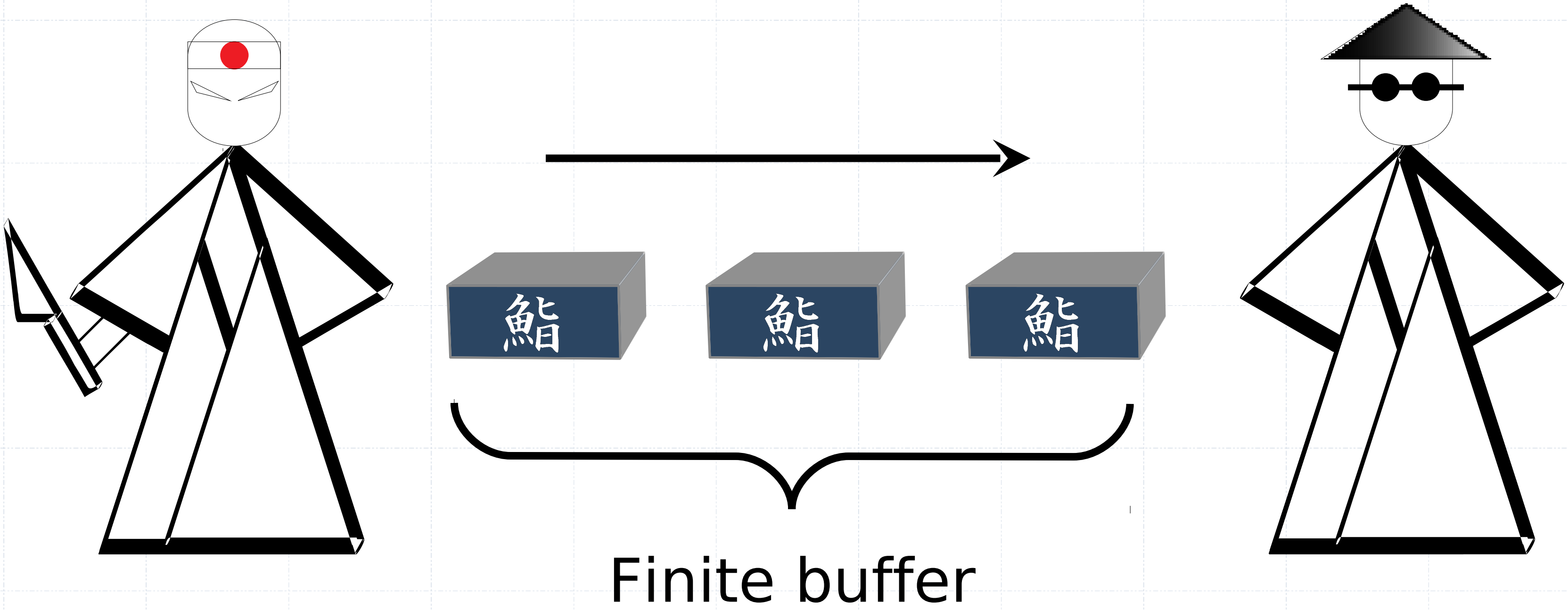
Unbounded Buffers

- Producer can work freely
- Consumer must wait for producer



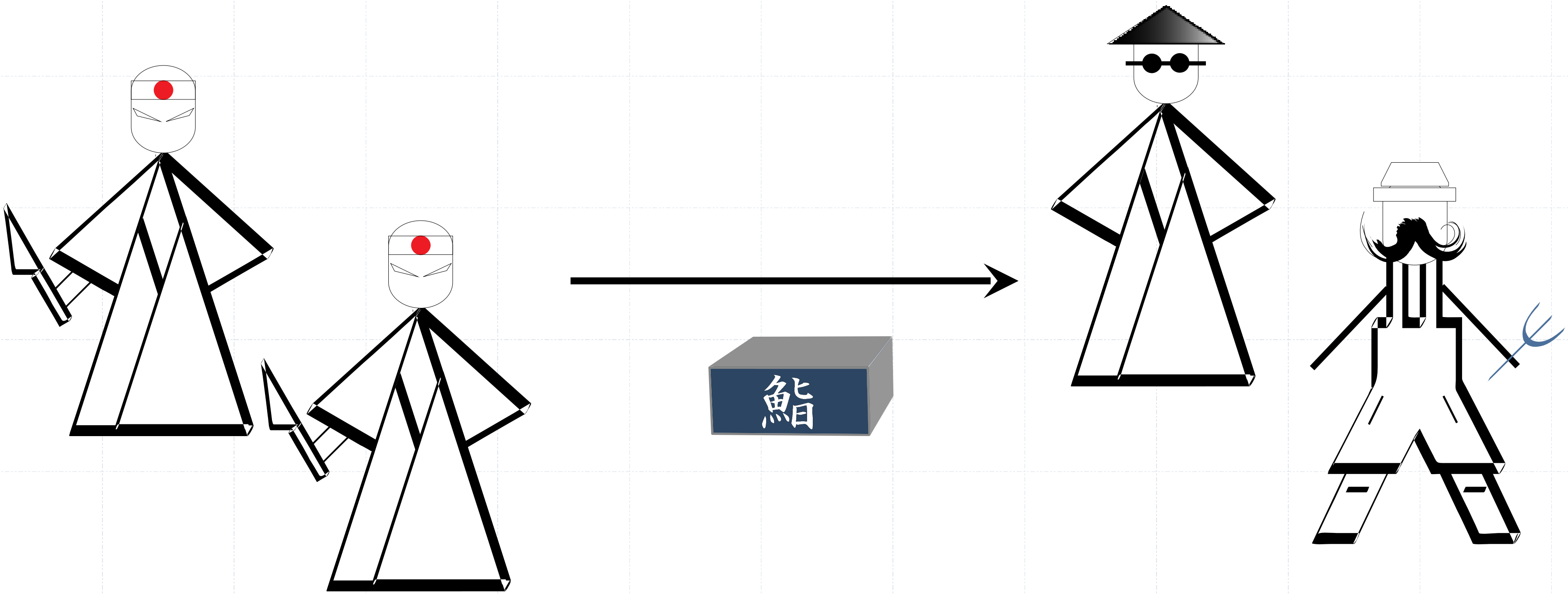
Bounded Buffers

- Producer must wait if buffer is full
- Consumer must wait if buffer is empty



Buffers Using Semaphores

- One-slot buffer
  - Multiple producer processes
  - Multiple consumer processes
  - Semaphores



Split Binary Semaphores

- Two reasons to block:
  - Producers wait for an (the) empty slot to be available
  - Consumers wait for a filled slot to be available
- Initialisation
  - One empty slot
  - Zero full slots
- Invariant
  - `empty+full<=1`

Shared

```
sem empty = 1;
sem full  = 0;

Data buf;
```

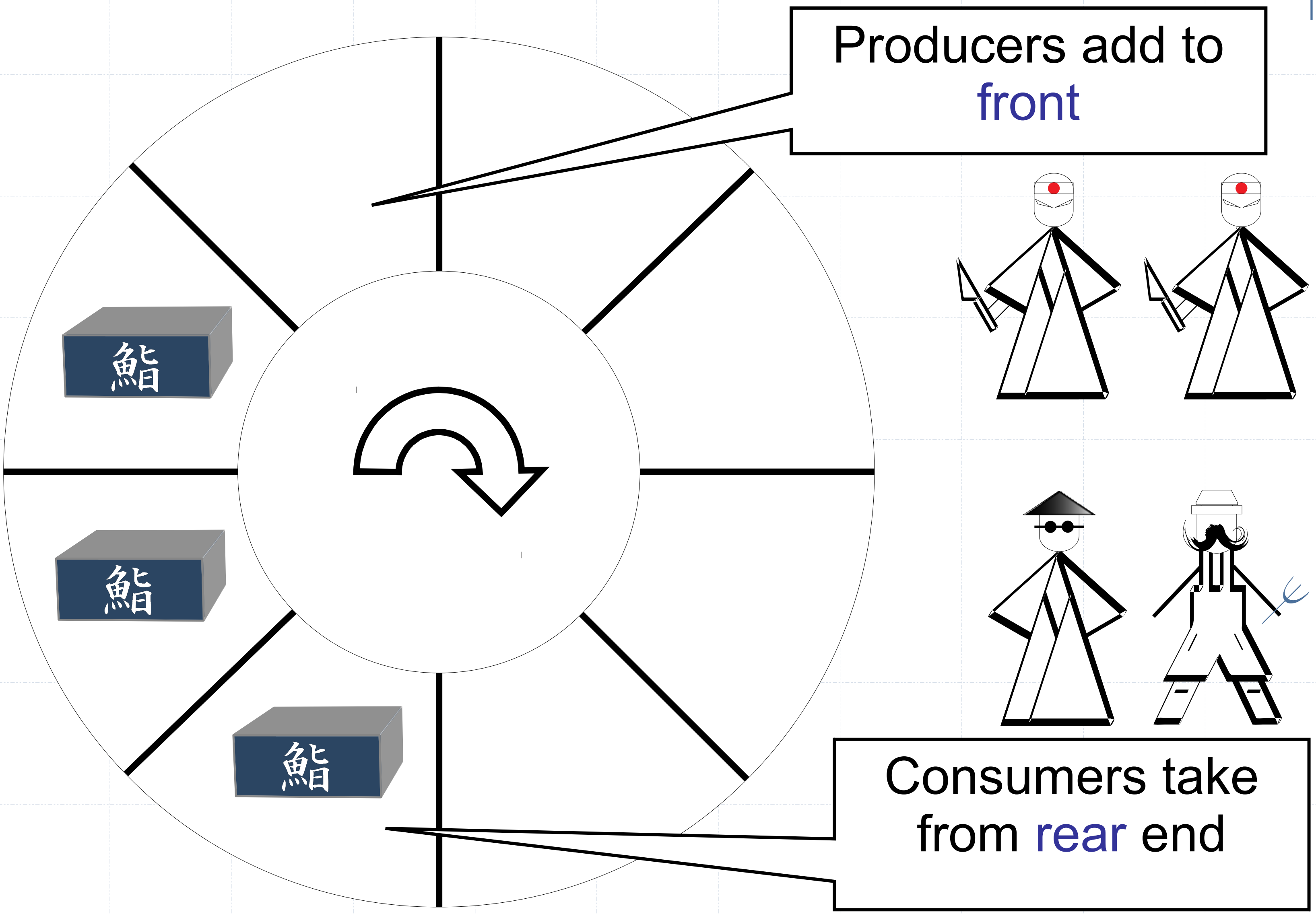
One-Slot Buffer

```
sem empty = 1, full  = 0;
Data buf = null;
```

```
process P(1...M) {
   while(true){
      //Produce data
      P(empty);
      buf = data;
      V(full);
   }
}
```

```
process C(1...N) {
   Data myData
   while(true){
      P(full);
      myData = buf;
      V(empty);
      //Consume myData
   }
}
```

General N-slot Buffer



General Semaphore

- Counting resources
- Initialisation
  - S empty slots
  - Zero full slots
- Invariant
  - `empty+full<=S`

Shared

```
sem empty = S;
sem full  = 0;

Data buf[S];

int front = 0,
    rear  = 0;
```

General N-slot Buffer

- Single producer

```
process Producer {
   while(true){
      //Produce data
      P(empty);
      buf[front] = data;
      front = (front+1)%S;
      V(full);
   }
}
```

- Multiple Producers?

General N-slot Buffer

- Multiple producers

```
sem mutexP = 1;
process Producer((int i=1;i<M;i++)) {
   while(true){
      //Produce data
      P(empty);
      P(mutexP);
      buf[front] = data;
      front = (front+1)%S;
      V(mutexP);
      V(full);
}}
```

General N-slot Buffer

- Multiple consumers

```
sem mutexC = 1;
process Consumer((int i=1;i<N;i++)) {
   Data myData;
   while(true){
      P(full);
      P(mutexC);
      myData = buf[rear];
      rear = (rear+1)%S;
      V(mutexC);
      V(empty);
      //Consume myData
}}
```

Preparing for Blocking

- A call to `P(s)` may involve blocking for a long time
  - A process might need to take precautions before waiting (e.g. set controlled device in safe state)
  - Problem: Precautions unnecessarily taken also when no waiting occurs

```
//now need to acquire s
take_precautions();
P(s)
```

One More Semaphore Operation

- `java.util.concurrent.Semaphore` has more operations. In particular
  - `boolean tryAcquire()`
    - A non-blocking operation acquiring the semaphore (and returns true) if it's possible at time of invocation. Otherwise, returns false (without acquiring the semaphore).
    - Ignores fairness setting!
  - `boolean tryAcquire(0, TimeUnit.SECONDS)`
    - Try to acquire the semaphore while possibly waiting 0 seconds
    - Fair equivalent

Stopping a Process – Java

- Semaphore

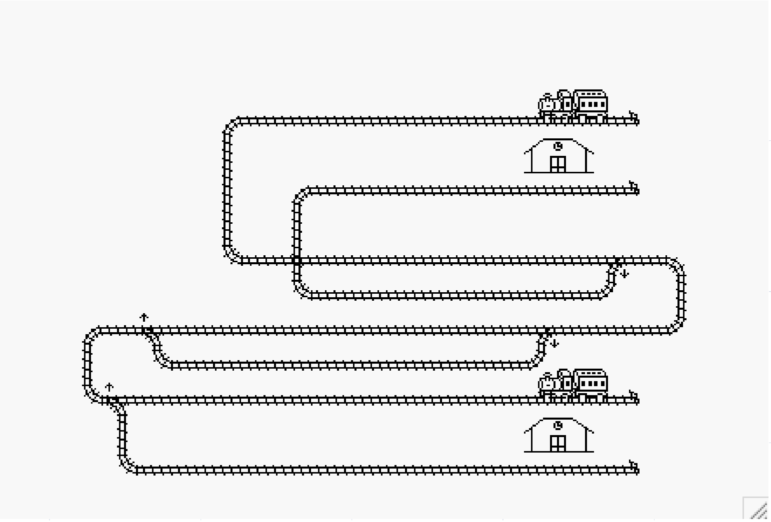
```
Semaphore terminate = new Semaphore(0);

public void run() {
   while (!terminate.tryAcquire())
      //Do some work here
}

public void shutdown() {
   terminate.release();
}
```

JR – tryAcquire

- JR does not have a direct equivalent
- But JR semaphores are *not* really semaphores
  - They are special channels
- Alternative constructs in JR exist

Assignment 1: Trainspotting

- Write a train controller
  - Independent movement of trains
  - No crash
    - synchronised using semaphores



Assignment 1

- Interface package `TSim`
  - Available on all Linux machines
  - Downloadable
- Special command 2
  - Available on all student machines
- The track is fixed
  - But you need to provide sensors
  - Find the critical sections
- Language Requirement
  - Must use Java (tryAcquire)
- Test, test, test, or prove correctness

Readers/Writers Problem

- Another classic synchronisation problem
- Two kinds of processes share access to a "database"
  - Readers examine the contents
  - Multiple readers allowed concurrently
  - Writers examine and modify
  - A writer must have mutex
- Invariant
  - $(nr == 0 \lor nw == 0) \land nw <= 1$

R/W – Coarse-grained Solution

```
process R((int i=0;i++;i<M)) {
   while(true){
      <await (nw==0) nr++;>
      //read database
      <nr--;>
}}
```

```
process W((int i=0;i++;i<N)) {
   while(true){
      <await (nr==0 && nw==0) nw++;>
      //write database
      <nw--;>
}}
```

R/W – Passing The Baton

- Split binary semaphore
  - `r` – for await in readers
  - `w` – for await in writers
  - `e` – for controlling entry into the "protocol"
  - `r+w+e == 1`
  - Initially `e == 1`
- Counters for waiting processes
  - `dr` – await in readers
  - `dw` – await in writers

R/W – Passing The Baton

```
process R((int i=0;i++;i<M)) {
   while(true){
      //<await (nw==0) nr++;>
      P(e);
      if (nw>0) { dr++; V(e); P(r); }
      nr++;
      Signal();
      //read database
      //<nr--;>
      P(e);
      nr--;
      Signal();
}}
```

R/W – Passing The Baton

```
process W((int i=0;i++;i<N)) {
   while(true){
      //<await (nr==0 && nw==0) nw++;>
      P(e);
      if (nr>0 || nw>0) { dw++; V(e); P(w); }
      nw++;
      Signal();
      //write database
      <nw--;>
      P(e);
      nw--;
      Signal();
}}
```

R/W – Passing The Baton

```
public void Signal() {
   if (nw==0 && dr>0) {
      dr--;
      V(r);
   } else if (nr==0 && nw==0 && dw>0) {
      dw--;
      V(w);
   } else
      V(e);
```

R/W – Correctness

- Split binary semaphore
  - Every execution path starts with P and ends with V
  - mutual exclusion in-between
- Await guards are guaranteed
  - Either true when checked with if-statement,
  - Or waiting on a semaphore that is signaled only when the condition becomes true
- Invariant
  - Initially true
  - True just before every V

Passing The Baton

- General technique
  - Implements any await statement
- Flexible scheduling policies
  - Readers preference as shown,
  - But the baton can be passed in different ways
    - New readers are delayed if a writer is waiting
    - A delayed reader is awakened only if no writer is currently waiting
    - Or use additional parameters to fine-tune scheduling

Summary – Semaphores

- Good news
  - Simple, efficient, expressive
    - Passing the Baton – any await statement
- Bad news
  - Low level, unstructured
    - omit a V: deadlock
    - omit a P: failure of mutex
  - Synchronisation code not linked to the data
    - Synchronisation code can be accessed anywere,
    - but good programming style helps!