Operating Systems and Concurrency

Concurrency 4 COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham United Kingdom

2023

©University of Nottingham 1/134

Module Feedback

Pre-lecture Request

Please log in to Moodle and complete the module feedback form near the top of the page.

©University of Nottingham 2/134

Recap Last Lecture

- Concurrency synchronisation using **semaphores**.
- Producer / consumer problems and difficulties of concurrent programming.

©University of Nottingham 3/134

Goals For Today

Classic Synchronisation Problems

- Continuing with the bounded buffer problem
- The dining philosophers problem

©University of Nottingham 4/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); 0 => -1
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 5/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
{
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
    sem_post(&sync);
    if(items == 0)
        sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync); 1 => 0
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 6/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
{
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
    sem_post(&sync);
    if(items == 0)
        sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++; 0 => 1
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 7/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
   {
      sem_wait(&sync);
      items--;
      printf("%d\n", items);
      sem_post(&sync);
      if(items == 0)
           sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 8/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
  sem_wait(&delay_consumer);
  while(1)
  {
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
    sem_post(&sync);
    if(items == 0)
    sem_wait(&delay_consumer);
  }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 9/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
                                          void * producer(void * p)
 sem wait(&delay consumer); (wakeup)
 while(1)
                                           while (1)
  sem wait(&sync);
                                            sem wait(&sync);
                                            items++;
  items--:
  printf("%d\n", items);
                                            printf("%d\n", items);
  sem post(&sync);
                                            if(items == 1)
  if(items == 0)
                                             sem post(&delay consumer); -1 => 0
   sem wait (&delay consumer);
                                            sem post(&sync);
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 10/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
  sem_wait(&delay_consumer);
  while(1)
  {
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
    sem_post(&sync);
    if(items == 0)
    sem_wait(&delay_consumer);
  }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync); 0 => 1
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 11/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 12/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
{
    sem_wait(&sync);
    items--; 1 => 0
    printf("%d\n", items);
    sem_post(&sync);
    if(items == 0)
        sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 13/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
{
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
    sem_post(&sync);
    if(items == 0)
        sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 14/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(items == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 15/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync); 1 => 0
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 16/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++; 0 => 1
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 17/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
   {
      sem_wait(&sync);
      items--;
      printf("%d\n", items);
      sem_post(&sync);
      if(items == 0)
      sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 18/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
   {
      sem_wait(&sync);
      items--;
      printf("%d\n", items);
      sem_post(&sync);
      if(items == 0)
           sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 19/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
   {
      sem_wait(&sync);
      items--;
      printf("%d\n", items);
      sem_post(&sync);
      if(items == 0)
        sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer); 0 => 1
    sem_post(&sync);
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 20/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
   {
      sem_wait(&sync);
      items--;
      printf("%d\n", items);
      sem_post(&sync);
      if(items == 0)
      sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync); 0 => 1
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 21/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
{
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
    sem_post(&sync);
    if(items == 0)
        sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 22/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
{
    sem_wait(&sync); 1 => 0
    items--;
    printf("%d\n", items);
    sem_post(&sync);
   if(items == 0)
     sem_wait(&delay_consumer);
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 23/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--; 1 => 0
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 24/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 25/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
{
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
   sem_post(&sync); 0 => 1
   if(items == 0)
    sem_wait(&delay_consumer);
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 26/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 27/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
  sem_wait(&delay_consumer);
  while(1)
  {
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
    sem_post(&sync);
    if(items == 0)
    sem_wait(&delay_consumer); 1 => 0
  }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 28/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
{
    sem_wait(&sync); 1 => 0
    items--;
    printf("%d\n", items);
    sem_post(&sync);
   if(items == 0)
    sem_wait(&delay_consumer);
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 29/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
   {
      sem_wait(&sync);
      items--; 0 => -1
      printf("%d\n", items);
      sem_post(&sync);
      if(items == 0)
      sem_wait(&delay_consumer);
   }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 30/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        sem_post(&sync);
        if(items == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 31/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
   sem_wait(&delay_consumer);
   while(1)
{
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
   sem_post(&sync); 0 => 1
   if(items == 0)
    sem_wait(&delay_consumer);
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 32/134

One Consumer, One Producer, Unbounded Buffer: Non-Existing Items

```
void * consumer(void * p)
{
  sem_wait(&delay_consumer);
  while(1)
  {
    sem_wait(&sync);
    items--;
    printf("%d\n", items);
    sem_post(&sync);
    if(items == 0)
    sem_wait(&delay_consumer);
  }
}
```

```
void * producer(void * p)
{

while(1)
{
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
}
```

Figure: Single producer/consumer and an unbounded buffer: Race condition (non-existing element => items = -1)

©University of Nottingham 33/134

One Consumer, One Producer, Unbounded Buffer

- Although we synchronise access to items, this did not prevent a race condition involving this variable!
- We did not preserve the relationship between items and the semaphore delay_consumer.

©University of Nottingham 34/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); 0 => -1
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
        sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 35/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
        sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync); 1 => 0
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 36/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
        sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++; 0 => 1
    printf("&d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 37/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
        sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 38/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
        sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if (items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 39/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); (wakeup)
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
        sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer); -1 => 0
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 40/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
        sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{

while(1)
{
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync); 0 => 1
}
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 41/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync); 1 => 0
        items--;
        printf("%d\n", items);
        if(items == 0)
        sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 42/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--; 1 => 0
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 43/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
            sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 44/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        printf("%d\n", items);
        if(items == 0)
        sem_wait(&delay_consumer);
        sem_post(&sync);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("&d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 45/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
                                              void * producer(void * p)
 sem wait (&delay consumer);
while(1)
                                               while(1)
                                                sem wait(&sync);
  sem wait(&sync);
  items--:
                                                items++;
  printf("%d\n", items);
                                                printf("%d\n", items);
  if(items == 0)
                                                if(items == 1)
   sem wait(&delay consumer); 0=>-1 (sleep)
                                                 sem post(&delay consumer);
  sem post(&sync);
                                                sem post(&sync);
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 46/134

One Consumer, One Producer, Unbounded Buffer: Deadlocks

```
void * consumer(void * p)
                                                void * producer(void * p)
 sem wait (&delay consumer);
 while(1)
                                                 while(1)
  sem wait(&svnc);
                                                  sem wait(&sync); 0 \Rightarrow -1 (sleep)
  items--:
                                                  items++:
  printf("%d\n", items);
                                                  printf("%d\n", items);
  if(items == 0)
                                                  if(items == 1)
                                                   sem post(&delay consumer);
   sem wait(&delay consumer);
  sem post(&sync);
                                                  sem post(&sync);
```

Figure: Single producer/consumer and an unbounded buffer: deadlocks

©University of Nottingham 47/134

One Consumer, One Producer, Unbounded Buffer: solution

- Use a temporary variable:
 - Copies the value of items inside the critical section
 - Decrements the delay_consumer semaphore to make it consistent

©University of Nottingham 48/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); 0 => -1
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
            sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 49/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync); 1 => 0
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 50/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++; 0 => 1
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 51/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 52/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 53/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer); (wakeup)
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer); -1 => 0
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 54/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync); 0 => 1
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 55/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync); 1 => 0
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 56/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--; 1 => 0
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 57/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 58/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 59/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync); 0 => 1
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 60/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        temp = items;
        printf("&d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync); 1 => 0
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 61/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++; 0 => 1
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 62/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 63/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        temp = items;
        printf("&d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if (items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 64/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer); 0 => 1
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 65/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync);
        items--;
        temp = items;
        printf("&d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync); 0 => 1
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 66/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 67/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer); 1 => 0
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 68/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
{
        sem_wait(&sync); 1 => 0
        items--;
        temp = items;
        printf("&d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 69/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--; 1 => 0
        temp = items;
        printf("&d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 70/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 71/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 72/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("&d\n", items);
        sem_post(&sync); 0 => 1
        if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
        sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 73/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
{
    sem_wait(&delay_consumer);
    while(1)
    {
        sem_wait(&sync);
        items--;
        temp = items;
        printf("%d\n", items);
        sem_post(&sync);
    if(temp == 0)
        sem_wait(&delay_consumer);
    }
}
```

```
void * producer(void * p)
{
  while(1)
  {
    sem_wait(&sync);
    items++;
    printf("%d\n", items);
    if(items == 1)
    sem_post(&delay_consumer);
    sem_post(&sync);
  }
}
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 74/134

One Consumer, One Producer, Unbounded Buffer: solution

```
void * consumer(void * p)
                                              void * producer(void * p)
 sem wait(&delay consumer);
 while(1)
                                               while(1)
  sem wait (&sync);
                                                sem wait(&sync);
  items--:
                                                items++:
  temp = items;
                                                printf("%d\n", items);
  printf("%d\n", items);
                                                if(items == 1)
  sem post(&sync);
                                                  sem post(&delay consumer);
  if(temp == 0)
                                                sem post(&sync);
   sem_wait(&delay_consumer); 0 => -1
```

Figure: Single producer/consumer and an unbounded buffer: correct solution

©University of Nottingham 75/134

Multiple Producers, Multiple Consumers, Bounded Buffer

- The previous code (one consumer, one producer) is made to work by storing the value of items
- A different variant of the problem has n consumers, m producers, and a fixed buffer size N. The solution is based on 3 semaphores:
 - sync: used to enforce mutual exclusion for the buffer
 - empty: keeps track of the number of empty buffers, initialised to N
 - full: keeps track of the number of full buffers, initialised to 0
- The empty and full are counting semaphores and represent resources

©University of Nottingham 76/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
    }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 77/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty); 3 => 2
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 78/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync); 1 => 0
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 79/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 80/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 81/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync); 0 => 1
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 82/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full); 0 => 1
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 83/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty); 2 => 1
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 84/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync); 1 => 0
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 85/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 86/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 87/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync); 0 => 1
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 88/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full); 1 => 2
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 89/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty); 1 => 0
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 90/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync); 1 => 0
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 91/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 92/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 93/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync); 0 => 1
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 94/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full); 2 => 3
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 95/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty); 0 => -1 (sleep)
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 96/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full); 3 => 2
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 97/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
    }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync); 1 => 0
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 98/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 99/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 100/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync); 0 => 1
    sem_post(&empty);
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 101/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty); (wakeup)
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync);
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty); -1 => 0
}
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 102/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync); 1 => 0
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
   while(1)
   {
       sem_wait(&full); 2 => 1
       sem_wait(&sync);
       items--;
       printf("Consumer: %d\n", items);
       sem_post(&sync);
       sem_post(&empty);
   }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 103/134

Multiple Producers & Consumers

```
void * producer(void * a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    items++;
    printf("Producer: %d\n", items);
    sem_post(&sync);
    sem_post(&full);
  }
}
```

```
void * consumer(void * a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync); 0 => -1 (sleep)
    items--;
    printf("Consumer: %d\n", items);
    sem_post(&sync);
    sem_post(&empty);
  }
}
```

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 104/134

Multiple Producers & Consumers

Figure: Multiple Producers and Consumers with Semaphores (N = 3)

©University of Nottingham 105/134

The Dining Philosophers Problem

Description

- The problem is defined as:
 - Five philosophers are sitting on a round table
 - Each one has one has a plate of spaghetti
 - The spaghetti is too slippery, and each philosopher needs 2 forks to be able to eat
 - When hungry (in between thinking), the philosopher tries to acquire the forks on their left and right
- Note that this reflects the general problem of sharing a limited set of resources (forks) between a number of processes (philosophers)

©University of Nottingham 106/134

The Dining Philosophers Problem

Description

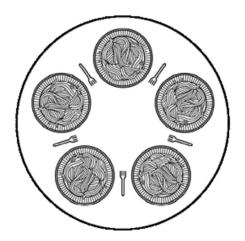


Figure: Tanenbaum, 4th edition

©University of Nottingham 107/134

The Dining Philosophers Problem

Solution 1

- Forks are represented by semaphores (initialised to 1)
 - 1 if the fork is available: the philosopher can continue
 - 0 if the **fork is not available**: the philosopher goes to **sleep** if trying to acquire it
- First approach: Every philosopher picks up one fork and waits for the second one to become available (without putting the first one down)

©University of Nottingham 108/134

Solution 1: Naive will Deadlock

```
#define N 5
sem t forks[N];
void * philosopher(void * id) {
  int i = *((int *) id);
  int left = (i + N - 1) % N;
  int right = i % N:
  while(1) {
    printf("%d is thinking\n", i);
    printf("%d is hungry\n", i);
    sem wait (&forks[left]);
    sem_wait(&forks[right]);
    printf("%d is eating\n", i);
    sem post(&forks[left]);
    sem_post(&forks[right]);
```

©University of Nottingham 109/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<pre>wait(&f[4) 1 => 0 wait(&f[0)</pre>	wait(&f[0) wait(&f[1)	wait(&f[1) wait(&f[2)	<pre>wait(&f[2) wait(&f[3)</pre>	wait(&f[3) wait(&f[4)
// eating	// eating	// eating	// eating	// eating
post(&f[4) post(&f[0)	post(&f[0) post(&f[1)	post(&f[1) post(&f[2)	post(&f[2) post(&f[3)	post(&f[3) post(&f[4)

©University of Nottingham 110/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4) wait(&f[0)	<pre>wait(&f[0) 1 => 0 wait(&f[1)</pre>	<pre>wait(&f[1) wait(&f[2)</pre>	wait(&f[2) wait(&f[3)	wait(&f[3) wait(&f[4)
// eating	// eating	// eating	// eating	// eating
post(&f[4) post(&f[0)	post(&f[0) post(&f[1)	post(&f[1) post(&f[2)	post(&f[2) post(&f[3)	post(&f[3) post(&f[4)

©University of Nottingham 111/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4) wait(&f[0)	<pre>wait(&f[0) wait(&f[1)</pre>	<pre>wait(&f[1) 1 => 0 wait(&f[2)</pre>	wait(&f[2) wait(&f[3)	wait(&f[3) wait(&f[4)
// eating	// eating	// eating	// eating	// eating
post(&f[4) post(&f[0)	post(&f[0) post(&f[1)	post(&f[1) post(&f[2)	post(&f[2) post(&f[3)	post(&f[3) post(&f[4)

©University of Nottingham 112/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4) wait(&f[0)	wait(&f[0) wait(&f[1)	<pre>wait(&f[1) wait(&f[2)</pre>	<pre>wait(&f[2) 1 => 0 wait(&f[3)</pre>	wait(&f[3) wait(&f[4)
// eating	// eating	// eating	// eating	// eating
post(&f[4) post(&f[0)	post(&f[0) post(&f[1)	post(&f[1) post(&f[2)	post(&f[2) post(&f[3)	post(&f[3) post(&f[4)

©University of Nottingham 113/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4)	wait(&f[0)	wait(&f[1)	wait(&f[2)	<pre>wait(&f[3) 1 => 0 wait(&f[4)</pre>
wait(&f[0)	wait(&f[1)	wait(&f[2)	wait(&f[3)	
// eating	// eating	// eating	// eating	 // eating
post(&f[4)	post(&f[0)	post(&f[1)	post(&f[2)	post(&f[3)
post(&f[0)	post(&f[1)	post(&f[2)	post(&f[3)	post(&f[4)

©University of Nottingham 114/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4) wait(&f[0) 0 => -1	<pre>wait(&f[0) wait(&f[1)</pre>	wait(&f[1) wait(&f[2)	wait(&f[2) wait(&f[3)	<pre>wait(&f[3) wait(&f[4)</pre>
// eating	// eating	// eating	// eating	// eating
post(&f[4) post(&f[0)	post(&f[0) post(&f[1)	post(&f[1) post(&f[2)	post(&f[2) post(&f[3)	post(&f[3) post(&f[4)

©University of Nottingham 115/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4) wait(&f[0)	wait(&f[0) wait(&f[1) 0 => -1	<pre>wait(&f[1) wait(&f[2)</pre>	wait(&f[2) wait(&f[3)	<pre>wait(&f[3) wait(&f[4)</pre>
// eating	// eating	// eating	// eating	// eating
post(&f[4) post(&f[0)	post(&f[0) post(&f[1)	post(&f[1) post(&f[2)	post(&f[2) post(&f[3)	post(&f[3) post(&f[4)

©University of Nottingham 116/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4) wait(&f[0)	<pre>wait(&f[0) wait(&f[1)</pre>	wait(&f[1) wait(&f[2) 0 => -1	<pre>wait(&f[2) wait(&f[3)</pre>	wait(&f[3) wait(&f[4)
// eating	// eating	// eating	// eating	// eating
post(&f[4) post(&f[0)	post(&f[0) post(&f[1)	post(&f[1) post(&f[2)	post(&f[2]) post(&f[3])	post(&f[3) post(&f[4)

©University of Nottingham 117/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[4) wait(&f[0)	<pre>wait(&f[0) wait(&f[1)</pre>	<pre>wait(&f[1) wait(&f[2)</pre>	wait(&f[2) wait(&f[3) 0 => -1	<pre>wait(&f[3) wait(&f[4)</pre>
// eating	// eating	// eating	// eating	// eating
post(&f[4) post(&f[0)	post(&f[0) post(&f[1)	post(&f[1) post(&f[2)	post(&f[2) post(&f[3)	post(&f[3) post(&f[4)

©University of Nottingham 118/134

Solution 1: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<pre>wait(&f[4) wait(&f[0)</pre>	wait(&f[0) wait(&f[1)	wait(&f[1) wait(&f[2)	wait(&f[2) wait(&f[3)	wait(&f[3) wait(&f[4) 0 => -1
// eating	// eating	// eating	// eating	// eating
post(&f[4) post(&f[0)	post(&f[0) post(&f[1)	post(&f[1) post(&f[2)	post(&f[2]) post(&f[3])	post(&f[3) post(&f[4)

©University of Nottingham 119/134

Solution 1: Deadlock

- The naive solution can deadlock
- Deadlocks can be prevented by:
 - Putting the forks down and waiting a random time potential for other bugs.
 - Putting one additional fork on the table answering an easier problem!
 - One global mutex set by a philosopher when they want to eat only one can eat at a time.

©University of Nottingham 120/134

Solution 1: Deadlock

- The naive solution can deadlock
- Deadlocks can be prevented by:
 - Putting the forks down and waiting a random time potential for other bugs.
 - Putting one additional fork on the table answering an easier problem!
 - One global mutex set by a philosopher when they want to eat only one can eat at a time.
 - Solution does not result in maximum parallelism as only one eats at a time.

©University of Nottingham 121/134

Solutions 2: Global Mutex/Semaphore

```
sem t eating;
void * philosopher(void * id)
 int i = (int) id;
 int left = (i + N - 1) % N;
 int right = i % N:
 while(1)
   printf("%d is thinking\n", i);
   printf("%d is hungry\n", i);
   sem_wait(&eating); /**** mutex/semaphore ****/
   sem wait (&forks[left]);
   sem wait (&forks[right]);
   printf("%d is eating\n", i);
   sem_post(&forks[left]);
   sem post(&forks[right]);
```

©University of Nottingham 122/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<pre>wait(&eating) 1=>0 wait(&forks[4]) wait(&forks[0])</pre>	wait(&eating) wait(&forks[0]) wait(&forks[1])	<pre>wait(&eating) wait(&forks[1]) wait(&forks[2])</pre>	<pre>wait(&eating) wait(&forks[2]) wait(&forks[3])</pre>	wait(&eating) wait(&forks[3]) wait(&forks[4])
// eating	// eating	// eating	// eating	// eating
post(&forks[4]) post(&forks[0]) post(&eating)	post(&forks[0]) post(&forks[1]) post(&eating)	post(&forks[1]) post(&forks[2]) post(&eating)	<pre>post(&forks[2]) post(&forks[3]) post(&eating)</pre>	post(&forks[3]) post(&forks[4]) post(&eating)

©University of Nottingham 123/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<pre>wait(&eating) wait(&forks[4]) 1: wait(&forks[0])</pre>	<pre>wait(&eating) =>0 wait(&forks[0]) wait(&forks[1])</pre>	wait(&eating) wait(&forks[1]) wait(&forks[2])	<pre>wait(&eating) wait(&forks[2]) wait(&forks[3])</pre>	<pre>wait(&eating) wait(&forks[3]) wait(&forks[4])</pre>
// eating	// eating	// eating	// eating	// eating
post(&forks[4]) post(&forks[0]) post(&eating)	post(&forks[0]) post(&forks[1]) post(&eating)	post(&forks[1]) post(&forks[2]) post(&eating)	post(&forks[2]) post(&forks[3]) post(&eating)	post(&forks[3]) post(&forks[4]) post(&eating)

©University of Nottingham 124/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<pre>wait(&eating) wait(&forks[4]) wait(&forks[0]) 1=</pre>	wait(&eating) wait(&forks[0]) =>0 wait(&forks[1])	<pre>wait(&eating) wait(&forks[1]) wait(&forks[2])</pre>	<pre>wait(&eating) wait(&forks[2]) wait(&forks[3])</pre>	<pre>wait(&eating) wait(&forks[3]) wait(&forks[4])</pre>
// eating	// eating	// eating	// eating	// eating
post(&forks[4]) post(&forks[0]) post(&eating)	post(&forks[0]) post(&forks[1]) post(&eating)	post(&forks[1]) post(&forks[2]) post(&eating)	post(&forks[2]) post(&forks[3]) post(&eating)	post(&forks[3]) post(&forks[4]) post(&eating)

©University of Nottingham 125/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating) wait(&forks[4]) wait(&forks[0])	<pre>wait(&eating) 0=>-1 wait(&forks[0]) wait(&forks[1])</pre>	wait(&eating) wait(&forks[1]) wait(&forks[2])	<pre>wait(&eating) wait(&forks[2]) wait(&forks[3])</pre>	<pre>wait(&eating) wait(&forks[3]) wait(&forks[4])</pre>
// eating	// eating	// eating	// eating	// eating
post(&forks[4]) post(&forks[0]) post(&eating)	post(&forks[0]) post(&forks[1]) post(&eating)	post(&forks[1]) post(&forks[2]) post(&eating)	post(&forks[2]) post(&forks[3]) post(&eating)	<pre>post(&forks[3]) post(&forks[4]) post(&eating)</pre>

©University of Nottingham 126/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<pre>wait(&eating) wait(&forks[4]) wait(&forks[0])</pre>	<pre>wait(&eating) wait(&forks[0]) wait(&forks[1])</pre>	<pre>wait(&eating) -1=>- wait(&forks[1]) wait(&forks[2])</pre>	-2 wait(&eating) wait(&forks[2]) wait(&forks[3])	wait(&eating) wait(&forks[3]) wait(&forks[4])
// eating	// eating	// eating	// eating	// eating
post(&forks[4]) post(&forks[0]) post(&eating)	post(&forks[0]) post(&forks[1]) post(&eating)	post(&forks[1]) post(&forks[2]) post(&eating)	post(&forks[2]) post(&forks[3]) post(&eating)	post(&forks[3]) post(&forks[4]) post(&eating)

©University of Nottingham 127/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<pre>wait(&eating) wait(&forks[4]) wait(&forks[0])</pre>	<pre>wait(&eating) wait(&forks[0]) wait(&forks[1])</pre>	<pre>wait(&eating) wait(&forks[1]) wait(&forks[2])</pre>	<pre>wait(&eating) -2=>- wait(&forks[2]) wait(&forks[3])</pre>	<pre>-3 wait(&eating) wait(&forks[3]) wait(&forks[4])</pre>
// eating	// eating	// eating	// eating	// eating
post(&forks[4]) post(&forks[0]) post(&eating)	<pre>post(&forks[0]) post(&forks[1]) post(&eating)</pre>	<pre>post(&forks[1]) post(&forks[2]) post(&eating)</pre>	<pre>post(&forks[2]) post(&forks[3]) post(&eating)</pre>	<pre>post(&forks[3]) post(&forks[4]) post(&eating)</pre>

©University of Nottingham 128/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<pre>wait(&eating) wait(&forks[4]) wait(&forks[0])</pre>	<pre>wait(&eating) wait(&forks[0]) wait(&forks[1])</pre>	<pre>wait(&eating) wait(&forks[1]) wait(&forks[2])</pre>	<pre>wait(&eating) wait(&forks[2]) wait(&forks[3])</pre>	<pre>wait(&eating) -3=>-4 wait(&forks[3]) wait(&forks[4])</pre>
// eating				
post(&forks[4]) post(&forks[0]) post(&eating)	post(&forks[0]) post(&forks[1]) post(&eating)	post(&forks[1]) post(&forks[2]) post(&eating)	post(&forks[2]) post(&forks[3]) post(&eating)	<pre>post(&forks[3]) post(&forks[4]) post(&eating)</pre>

©University of Nottingham 129/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating) wait(&forks[4]) wait(&forks[0])	<pre>wait(&eating) wait(&forks[0]) wait(&forks[1])</pre>	<pre>wait(&eating) wait(&forks[1]) wait(&forks[2])</pre>	<pre>wait(&eating) wait(&forks[2]) wait(&forks[3])</pre>	<pre>wait(&eating) wait(&forks[3]) wait(&forks[4])</pre>
// eating	// eating	// eating	// eating	// eating
<pre>post(&forks[4]) (post(&forks[0]) post(&eating)</pre>	=>1 post(&forks[0]) post(&forks[1]) post(&eating)	post(&forks[1]) post(&forks[2]) post(&eating)	post(&forks[2]) post(&forks[3]) post(&eating)	<pre>post(&forks[3]) post(&forks[4]) post(&eating)</pre>

©University of Nottingham 130/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
<pre>wait(&eating) wait(&forks[4]) wait(&forks[0])</pre>	wait(&eating) wait(&forks[0]) wait(&forks[1])	<pre>wait(&eating) wait(&forks[1]) wait(&forks[2])</pre>	<pre>wait(&eating) wait(&forks[2]) wait(&forks[3])</pre>	<pre>wait(&eating) wait(&forks[3]) wait(&forks[4])</pre>
// eating	// eating	// eating	// eating	// eating
<pre>post(&forks[4]) post(&forks[0]) 0=> post(&eating)</pre>	post(&forks[0]) 1 post(&forks[1]) post(&eating)	post(&forks[1]) post(&forks[2]) post(&eating)	<pre>post(&forks[2]) post(&forks[3]) post(&eating)</pre>	post(&forks[3]) post(&forks[4]) post(&eating)

©University of Nottingham 131/134

Solutions 2: Illustration

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&eating) wait(&forks[4]) wait(&forks[0])	<pre>wait(&eating) (wake wait(&forks[0]) wait(&forks[1])</pre>	<pre>wait(&eating) wait(&forks[1]) wait(&forks[2])</pre>	<pre>wait(&eating) wait(&forks[2]) wait(&forks[3])</pre>	<pre>wait(&eating) wait(&forks[3]) wait(&forks[4])</pre>
// eating	// eating	// eating	// eating	// eating
<pre>post(&forks[4]) post(&forks[0]) post(&eating) -4=>-</pre>	post(&forks[0]) post(&forks[1]) post(&eating)	<pre>post(&forks[1]) post(&forks[2]) post(&eating)</pre>	post(&forks[2]) post(&forks[3]) post(&eating)	post(&forks[3]) post(&forks[4]) post(&eating)

©University of Nottingham 132/134

Solutions 2: Illustration

Question in a Previous Year:

"Can I initialise the value of the eating semaphore to 2 to create more parallelism"

©University of Nottingham 133/134

Test your understanding

- Can you find a better solution for the single producer consumer unbounded buffer problem?
- With $2 \times N$ philosophers, what is the maximum that can eat at once?
- What about $2 \times N + 1$ philosophers in a circle, what is the maximum number that can eat at once?

©University of Nottingham 134/134