University of Nottingham
UK | CHINA | MALAYSIA

Operating Systems and Concurrency

Lecture 11: Concurrency

University of Nottingham, Ningbo China, 2024

Recap
Last Lecture

```
typedef struct {
    int value;
    struct process * list;
} semaphore;
```

Figure: Conceptual definition of a semaphore

```
wait(semaphore * S) {
  S->value--;
  if(S->value < 0) {
    add process to S->list
    block(); // system call
  }
}
```

```
signal(semaphore * S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P); // system call
  }
}
```

Figure: Conceptual implementation of a acquire() and post()

- Concurrency using semaphores
- Practical examples of how to use (code) semaphores
- Solve producer consumer problem using semaphore

Today’s class

- The Multiple Consumer, Multiple Producer, Bounded Buffer

- The dining philosophers problem
  - Scenarios
  - Solutions

The Producer/Consumer Problem

Multiple Consumer, Multiple Producer, 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
- empty and full are counting semaphores and sync is binary semaphore.

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

Action	empty=3	Syn=1	Item=0	Full=0
	2	1	0	0
Enter_CS	2	0	0	0

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A		

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0

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

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

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

The Producer/Consumer Problem-
Multiple Producers & Consumers

Buffer

A		

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0
	2	0	1	0

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A		

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0
	2	0	1	0
Exit_CS	2	1	1	0

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A		

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0
	2	0	1	0
Exit_CS	2	1	1	0
	2	1	1	1

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer | A | | |

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0
	2	0	1	0
Exit_CS	2	1	1	0
	2	1	1	1
	1	1	1	1

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A		

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0
	2	0	1	0
Exit_CS	2	1	1	0
	2	1	1	1
	1	1	1	1
Enter_CS	1	0	1	1

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0
	2	0	1	0
Exit_CS	2	1	1	0
	2	1	1	1
	1	1	1	1
Enter_CS	1	0	1	1
	1	0	2	1

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0
	2	0	1	0
Exit_CS	2	1	1	0
	2	1	1	1
	1	1	1	1
Enter_CS	1	0	1	1
	1	0	2	1
	1	0	2	1

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	

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

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

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0
	2	0	1	0
Exit_CS	2	1	1	0
	2	1	1	1
	1	1	1	1
Enter_CS	1	0	1	1
	1	0	2	1
	1	0	2	1
Exit_CS	1	1	2	1

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	

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

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

Action	empty=3	Syn=1	Item=0	Full=0
	2	0	0	0
Enter_CS	2	0	0	0
	2	0	1	0
	2	0	1	0
Exit_CS	2	1	1	0
	2	1	1	1
	1	1	1	1
Enter_CS	1	0	1	1
	1	0	2	1
	1	0	2	1
Exit_CS	1	1	2	1
	1	1	2	2

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2

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

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

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

The Producer/Consumer Problem-
Multiple Producers & Consumers

Buffer

A	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	2

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	2
	0	0	3	2

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	2
	0	0	3	2
Exit_CS	0	1	3	2

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

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

The Producer/Consumer Problem-
Multiple Producers & Consumers

Buffer

A	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	2
	0	0	3	2
Exit_CS	0	1	3	2
	0	1	3	3

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer | A | B | C |

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

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

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	2
	0	0	3	2
Exit_CS	0	1	3	2
	0	1	3	3
Block_C	-1	1	3	3

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	2
	0	0	3	2
Exit_CS	0	1	3	2
	0	1	3	3
Block_C	-1	1	3	3
	-1	1	3	2

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	2
	0	0	3	2
Exit_CS	0	1	3	2
	0	1	3	3
Block_C	-1	1	3	3
	-1	1	3	2
Enter_CS	-1	0	3	2

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

	B	C

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

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

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

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	2
	0	0	3	2
Exit_CS	0	1	3	2
	0	1	3	3
Block_C	-1	1	3	3
	-1	1	3	2
Enter_CS	-1	0	3	2
	-1	0	2	2

The Producer/Consumer Problem-
Multiple Producers & Consumers

Buffer

	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	1	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	2
	0	0	3	2
Exit_CS	0	1	3	2
	0	1	3	3
Block_C	-1	1	3	3
	-1	1	3	2
Enter_CS	-1	0	3	2
	-1	0	2	2
	-1	0	2	2

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

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	-1	0	2	2
Exit_CS	-1	1	2	2

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

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

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

The Producer/Consumer Problem-
Multiple Producers & Consumers

Buffer

	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	-1	0	2	2
Exit_CS	-1	1	2	2
Wakeup_C	0	1	2	2

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

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

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

The Producer/Consumer Problem-
Multiple Producers & Consumers

Buffer

	B	C

Action	empty=3	Syn=1	Item=0	Full=0
	-1	0	2	2
Exit_CS	-1	1	2	2
Wakeup_C	0	1	2	2
Enter_CS	0	0	2	2
	0	0	2	1

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

Buffer

A	B	C

Consumer tried to join the CS and blocked

Action	empty=3	Syn=1	Item=0	Full=0
	-1	0	2	2
Exit_CS	-1	1	2	2
Wakeup_C	0	1	2	2
	0	1	2	2
Enter_CS	0	0	2	2
	0	0	3	1
Tried&blocked to Enter_CS	0	-1	3	1

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

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

The Producer/Consumer Problem- Multiple Producers & Consumers

```
void * producer(void *a)
{
  while(1)
  {
    sem_wait(&empty);
    sem_wait(&sync);
    item++;
     . . .
     . . .
     . . .
  }
}
```

```
void * consumer(void *a)
{
  while(1)
  {
    sem_wait(&full);
    sem_wait(&sync); 0 => -1 (sleep)
     . . .
     . . .
     . . .
     . . .
  }
}
```

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

The Dining Philosophers
Problem- Description

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 5.13). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

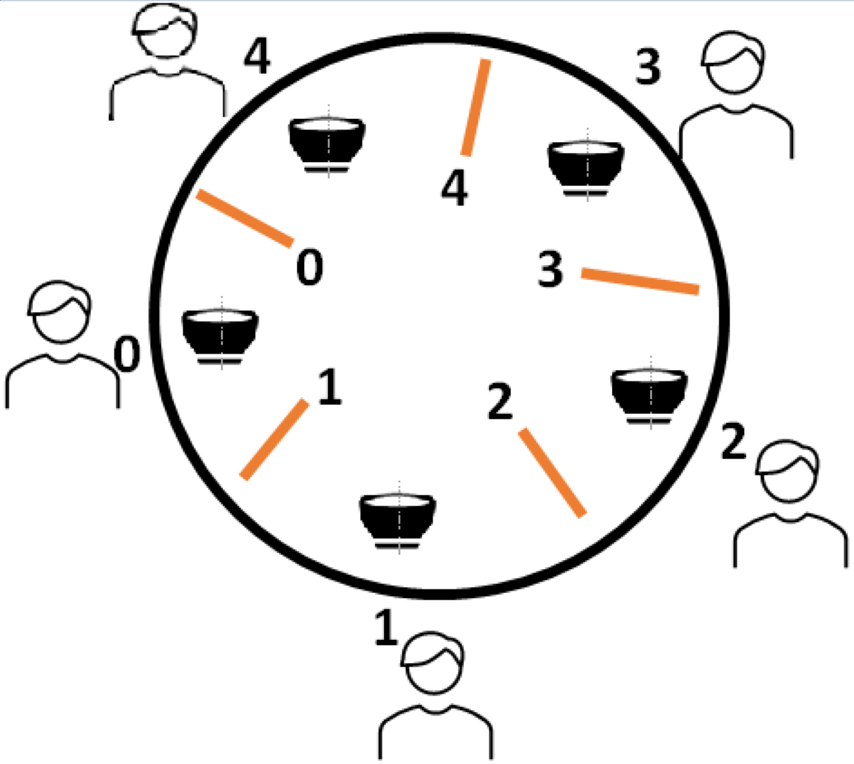

Fig: Silberschatz, 9th edition, page 22

- Note that this reflects the general problem of sharing a limited set of resources (forks) between a number of processes (philosophers)

The Dining Philosophers Problem- Solution 1 (Naïve will Deadlock)

- **First approach:** .
  - Represent each fork with the semaphore and `initialized to 1`.
    - `Semaphore fork[5]`
    - 0 is fork is not available
    - 1 is fork is available
  - A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
  - He/She releases her chopsticks by executing the signal() operation on the appropriate semaphores
- But, this could lead to **deadlock!!!!!**

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

The Dining Philosophers Problem- Solution 1 (Naïve will Deadlock)

```
Philosopher 1      Philosopher 2    Philosopher 3    Philosopher 4    Philosopher 5

wait(&f[5]) 1=>0   wait(&f[1])      wait(&f[2])      wait(&f[3])      wait(&f[4])
wait(&f[1])        wait(&f[2])      wait(&f[3])      wait(&f[4])      wait(&f[5])
...                ...              ...              ...              ...
//eating           //eating         //eating         //eating         //eating
...                ...              ...              ...              ...
post(&f[5])        post(&f[1])      post(&f[2])      post(&f[3])      post(&f[4])
post(&f[1])        post(&f[2])      post(&f[3])      post(&f[4])      post(&f[5])
```

The Dining Philosophers Problem- Solution 1 (illustration)

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[5])	**wait(&f[1]) 1=>0**	wait(&f[2])	wait(&f[3])	wait(&f[4])
wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4])	wait(&f[5])
…	…	…	…	…
//eating	//eating	//eating	//eating	//eating
...	...	...	...	...
post(&f[5])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])
post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])	post(&f[5])

The Dining Philosophers Problem- Solution 1 (illustration)

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[5])	wait(&f[1])	**wait(&f[2]) 1=>0**	wait(&f[3])	wait(&f[4])
wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4])	wait(&f[5])
…	…	…	…	…
//eating	//eating	//eating	//eating	//eating
...	...	...	...	...
post(&f[5])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])
post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])	post(&f[5])

The Dining Philosophers Problem- Solution 1 (illustration)

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[5])	wait(&f[1])	wait(&f[2])	**wait(&f[3]) 1=>0**	wait(&f[4])
wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4])	wait(&f[5])
...	...	...	...	...
//eating	//eating	//eating	//eating	//eating
...	...	...	...	...
post(&f[5])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])
post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])	post(&f[5])

The Dining Philosophers Problem- Solution 1 (illustration)

Philosopher 1	Philosopher 2	Philosopher 3	Philosopher 4	Philosopher 5
wait(&f[5])	wait(&f[1])	wait(&f[2])	wait(&f[3])	**wait(&f[4]) 1=>0**
wait(&f[1])	wait(&f[2])	wait(&f[3])	wait(&f[4])	wait(&f[5])
…	…	…	…	…
//eating	//eating	//eating	//eating	//eating
...	...	...	...	...
post(&f[5])	post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])
post(&f[1])	post(&f[2])	post(&f[3])	post(&f[4])	post(&f[5])

The Dining Philosophers Problem- Solution 1 (illustration)

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

The Dining Philosophers Problem- Solution 1 (illustration)

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

The Dining Philosophers Problem- Solution 1 (illustration)

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

The Dining Philosophers Problem- Solution 1 (illustration)

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

The Dining Philosophers Problem- Solution 1 (illustration)

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

The Dining Philosophers Problem- Solution 1 (Deadlock)

- The naive solution can deadlock

- Deadlocks can be prevented by:
  - **Putting the forks** down and waiting a random time (Ethernet networks)
  - Putting **one additional fork** on the table
  - **One global mutex/lock set** by a philosopher when (s)he wants to eat (only one can eat at a time)
    - Solution **does not result in maximum parallelism** (only one eats at a time)

The Dining Philosophers Problem- Solution 2 (Global Mutex/Semaphore)

```
sem_t eating;
void * philosopher(void * id) {
  int i = (int) id;

  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]);
    sem_post(&eating); /**** mutex/semaphore ****/
  }
}
```

The Dining Philosophers Problem- Solution 2 (Illustration)

```
Philosopher 1

wait(&eating)
wait(&f[5])
wait(&f[1])
...
//eating
...
post(&f[5])
post(&f[1])
post(&eating)
```

```
Philosopher 2

wait(&eating)
wait(&f[1])
wait(&f[2])
...
//eating
...
post(&f[1])
post(&f[2])
post(&eating)
```

```
Philosopher 3

wait(&eating)
wait(&f[2])
wait(&f[3])
...
//eating
...
post(&f[2])
post(&f[3])
post(&eating)
```

```
Philosopher 4

wait(&eating)
wait(&f[3])
wait(&f[4])
...
//eating
...
post(&f[3])
post(&f[4])
post(&eating)
```

```
Philosopher 5

wait(&eating)
wait(&f[4])
wait(&f[5])
...
//eating
...
post(&f[4])
post(&f[5])
post(&eating)
```

The Dining Philosophers Problem- Solution 2 (problem)

- The design still has limitations,
  - Primarily reduced concurrency and potential starvation, which make it less efficient for the Dining Philosophers problem.

- In practice, this solution would work but could be improved by allowing non-neighboring philosophers to eat simultaneously and by implementing mechanisms to reduce starvation.

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

- A **more sophisticated solution** is necessary to allow **maximum parallelism.**
- The solution uses:
  - state[N]: An array to represent each philosopher's state (THINKING, HUNGRY, or EATING).
    - int state[N] = {THINKING, THINKING, THINKING, THINKING, THINKING};
  - phil[N]: An array of binary semaphores (one per philosopher), each initialized to 0.
    - When a philosopher tries to take forks to eat and finds that a neighbor is eating, the philosopher is put to sleep.
    - The philosopher is woken up by a neighbor when that neighbor finishes eating and puts down the forks.
    - Philosopher i can set the variable state[i]=eating only if his two neighbors are not eating:
      - E.g. N=5, (state [(i+4) % 5] !=eating (right) and (state [(i+1)%5!=eating (left));
  - sync: A single semaphore (or mutex) used to enforce mutual exclusion in the critical section, ensuring that only one philosopher at a time can check or modify states.

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}

void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

	P0	P1	P2	P3	P4
State[N]	T	T	T	T	T
Sem_t phil[N]	0	0	0	0	0
sync	1				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

	P0	P1	P2	P3	P4
State[N]	T	T	H	T	T
Sem_t phil[N]	0	0	0	0	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

	P0	P1	P2	P3	P4
State[N]	T	T	H	T	T
Sem_t phil[N]	0	0	0	0	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
   state[i] = EATING;
   sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

	P0	P1	P2	P3	P4
State[N]	T	T	E	T	T
Sem_t phil[N]	0	0	0	0	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

	P0	P1	P2	P3	P4
State[N]	T	T	E	T	T
Sem_t phil[N]	0	0	1	0	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```c
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```c
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```c
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```c
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```c
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

	P0	P1	P2	P3	P4
State[N]	T	T	E	T	T
Sem_t phil[N]	0	0	1	0	0
sync	1				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

S[i] is 1, so sem_wait will not block. The value if s[i] decrements by 1.

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

	P0	P1	P2	P3	P4
State[N]	T	T	E	T	T
Sem_t phil[N]	0	0	0	0	0
sync	1				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

Let see another scenario where P3 move from the thinking state to take_fork() state(), while P2 is eating.

	P0	P1	P2	P3	P4
State[N]	T	T	E	H	T
Sem_t phil[N]	0	0	0	0	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

	P0	P1	P2	P3	P4
State[N]	T	T	E	H	T
Sem_t phil[N]	0	0	0	0	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```c
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}

void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```c
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```c
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```c
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

Blocked

	P0	P1	P2	P3	P4
State[N]	T	T	E	H	T
Sem_t phil[N]	0	0	0	-1	0
sync	1				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```c
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}

void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```c
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```c
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```c
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

Let say P2 decide to put down the fork!

Blocked

	P0	P1	P2	P3	P4
State[N]	T	T	T	H	T
Sem_t phil[N]	0	0	0	-1	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

Let say P2 decide to put down the fork!

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
   state[i] = EATING;
   sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

Blocked

	P0	P1	P2	P3	P4
State[N]	T	T	T	H	T
Sem_t phil[N]	0	0	0	-1	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```c
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```c
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```c
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

Let say P2 decide to put down the fork!

```c
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```c
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

Blocked

	P0	P1	P2	P3	P4
State[N]	T	T	T	H	T
Sem_t phil[N]	0	0	0	-1	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

Let say P2 decide to put down the fork!

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
   state[i] = EATING;
   sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

Blocked

	P0	P1	P2	P3	P4
State[N]	T	T	T	E	T
Sem_t phil[N]	0	0	0	-1	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}

void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

Let say P2 decide to put down the fork!

Wakeup P3

	P0	P1	P2	P3	P4
State[N]	T	T	T	E	T
Sem_t phil[N]	0	0	0	0	0
sync	0				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

Let say P2 decide to put down the fork!

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

Wakeup P3

	P0	P1	P2	P3	P4
State[N]	T	T	T	E	T
Sem_t phil[N]	0	0	0	0	0
sync	1				

The Dining Philosophers Problem- Solution 3 (Maximum Parallelism)

```
void * philosopher(void * id) {
  int i = *((int *) id);
  while(1)
  {
   printf("%d is thinking\n", i);
   take_forks(i);
   printf("%d is eating\n", i);
   put_forks(i);
  }
}
```

```
void take_forks(int i) {
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}
```

```
void put_forks(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}
```

Let say P2 decide to put down the fork!

```
void test(int i) {
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
   && state[left] != EATING
   && state[right] != EATING)
  {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3
int state[N] = {THINKING, THINKING,
THINKING, THINKING, THINKING};
sem_t phil[N]; // sends philosopher to
sleep
sem_t sync;
```

Wakeup P3

	P0	P1	P2	P3	P4
State[N]	T	T	T	E	T
Sem_t phil[N]	0	0	0	0	0
sync	1				

Recap
Take-Home Message

- Modern Operating Systems (Tanenbaum): **Chapter 2(2.3.5, 2.5.1)**
- Operating System Concepts (Silberschatz):**Chapter 6(6.6-7)**
- Operating Systems: Internals and Design Principles (Starlings): **Chapter 5(5.3, 5.6)**