Introduction to Operating Systems

- Introduction
- Processes and Threads
  - *IPC (Interprocess Communication)*
  - *Process synchronization*
- Memory Management
- File Systems
- Input / Output
- Deadlocks

Semaphores

- Semaphore is a synchronization primitive, higher level than locks
- ***Dijkstra (1965)*** proposed a solution based on the fundamental principle of cooperation based on signals, so that a process can be forced to stop at a required place and later starts when instructed through a signal.
- These signals are called semaphores, to transmit a signal the process executes ***signal (s) (or up, V)*** and to receive a signal it executes ***wait (s) (or down, P)***.
  - *does not require busy waiting*
  - *higher level than locks*

Semaphores

- **Semaphore Properties:**
  - *It is initialized to a non-negative value*
  - *Wait operation decrements semaphore value, if it goes negative the corresponding process is blocked*
  - *Signal operation increments semaphore value, if the value is not positive, a process can be unblocked from the suspended list.*
  - *Signal and Wait are atomic (cannot be interrupted)*

Semaphores

- Semaphore, could be declared as a simple structure having a variable declaration, may be an integer for count and another variable/structure for a process queue.

```
wait (s)                                  signal (s)
     count = count - 1;                        count = count + 1;
     if count < 0                              if count ≤ 0
          put the process in queue                  get a process from the queue
          block the process                         unblock the process and put it in ready queue

        Critical Section
```

- The order of removal from the queue is not defined, FIFO etc., the only requirement is that no process should wait indefinitely in the queue.
- Example, set count = 1; which process will enter into critical section ?

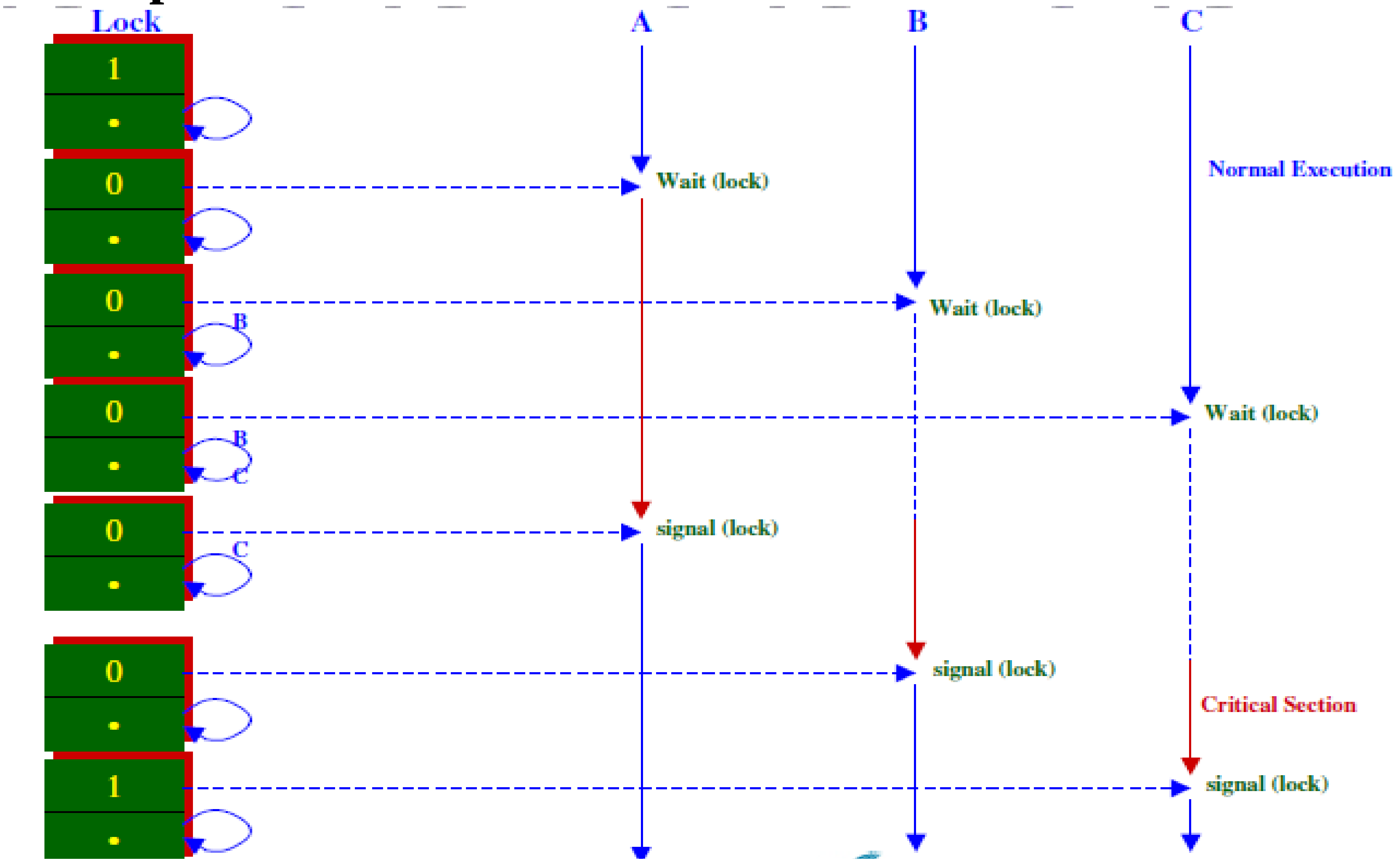
Semaphores

- **There are two types of semaphores**
  - *Binary semaphore (similar to mutex semaphore)*
    - *A key difference to mutex is that the process that locks the mutex must be the one to unlock it. In contrast, it is possible for one process to lock a binary semaphore and another to unlock it.*
    - *Guarantees mutually exclusive access to resource*
    - *Only one thread/process allowed entry at a time*
    - *count is initialized to 1*
    - *A blocking-lock (not a spinlock)*

    Example:

```
        Semaphore S=1;    //  initialized to 1
        wait (S);
        Critical Section
        signal (S);
```

Semaphores


Lock
A
B
C
1
0
0
0
0
0
1
B
B
C
C
Wait (lock)
Normal Execution
Wait (lock)
Wait (lock)
signal (lock)
signal (lock)
Critical Section
signal (lock)

Semaphores

- *Counting semaphore*
  - *Represents a resources with many units available*
  - *Allows threads/process to enter as long as more units are available*
  - *count is initialized to N*

    N = number of units available

Semaphores

- **Semaphore as a general synchronization tool**
  - *Execute B in P_j only after A executed in P_i*
  - *Use semaphore* flag *initialized to 0*

  Code:

```
        Pi                      Pj
        ⋮                       ⋮
        A                  wait (flag)
   signal (flag)                B
```

Semaphores

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
  - *Let* S *and* Q *be two semaphores initialized to 1*

```
P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S);
 ⋮                   ⋮
signal(S);          signal(Q);
signal(Q)           signal(S);
```

Classical IPC Problems

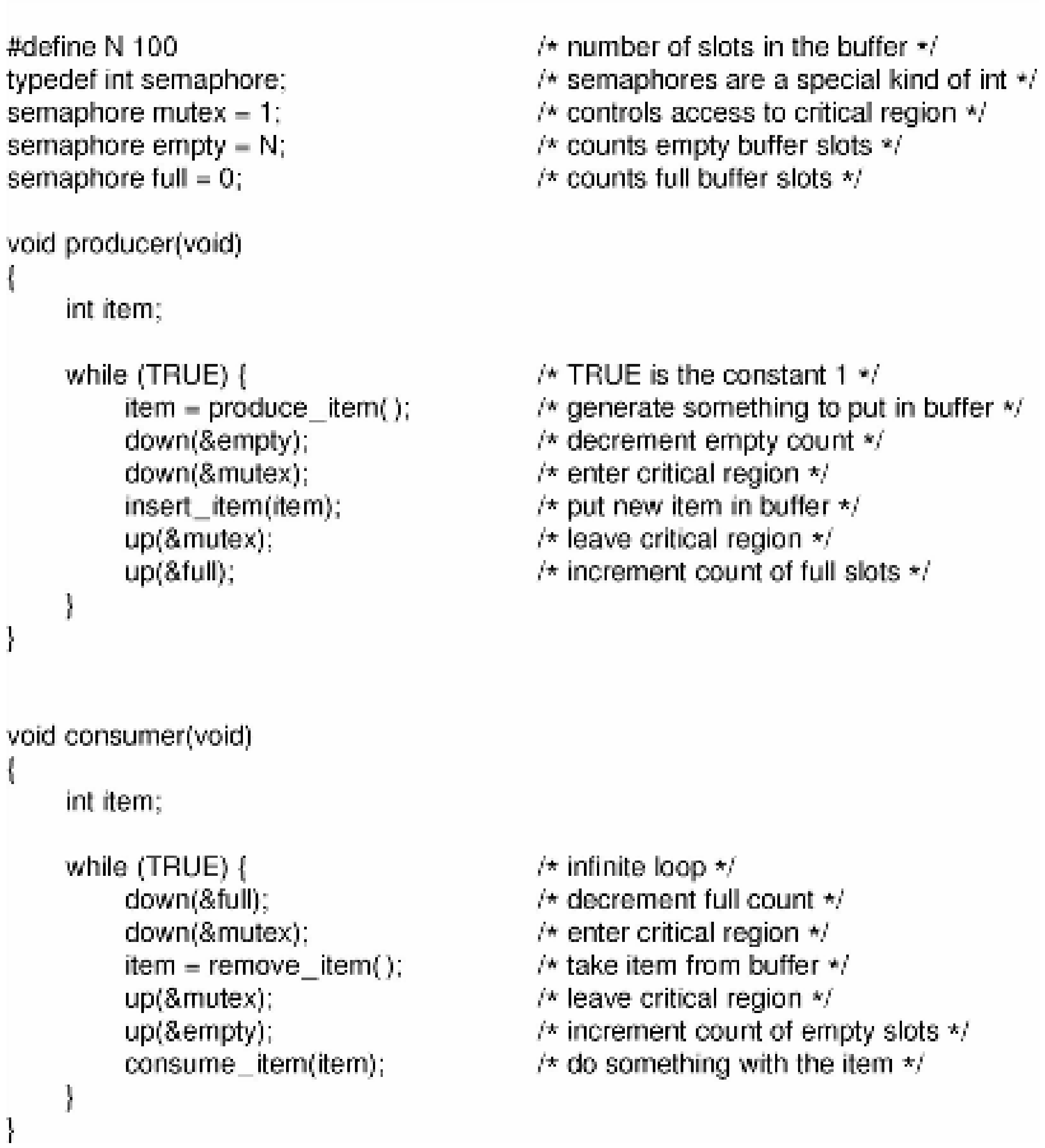
- Bounded Buffer Problem
  - *Producer/Consumer Problem*
- Readers/Writers Problem
- Dining Philosophers Problem
- Sleeping Barber Problem

Classical IPC Problems
(Producer/Consumer -Bounded Buffer Problem)

- Problem Definition
  - *There is a buffer in memory with finite size N entries.*
  - *One or more producers are generating data and placing these in a buffer*
  - *A single consumer is taking items out of the buffer one at time*
  - *Only one producer or consumer may access the buffer at any one time*
  - *Processes are concurrent*
    - *so, we must use synchronization constructs to control access to shared variables describing buffer state*

Classical IPC Problems

(Bounded Buffer Problem using Semaphore)



```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
      int item;

      while (TRUE) {                            /* TRUE is the constant 1 */
            item = produce_item( );             /* generate something to put in buffer */
            down(&empty);                       /* decrement empty count */
            down(&mutex);                       /* enter critical region */
            insert_item(item);                  /* put new item in buffer */
            up(&mutex);                         /* leave critical region */
            up(&full);                          /* increment count of full slots */
      }
}


void consumer(void)
{
      int item;

      while (TRUE) {                            /* infinite loop */
            down(&full);                        /* decrement full count */
            down(&mutex);                       /* enter critical region */
            item = remove_item( );              /* take item from buffer */
            up(&mutex);                         /* leave critical region */
            up(&empty);                         /* increment count of empty slots */
            consume_item(item);                 /* do something with the item */
      }
}
```

Classical IPC Problems
(Readers/Writers Problem)

- Problem Definition
  - *Object (file, record etc.) is shared among several processes*
  - *Any number of readers may simultaneously read the object*
    - *When there is already at least one reader reading, subsequent readers need not wait before entering*
  - *Only one writer at a time may write to the object*
  - *If a writer is writing to the object, no reader may read it*

- This problem models access to a database.
  - *For example, airline reservation system, with many competing processes wishing to read and write it.*

Classical IPC Problems
(Readers/Writers Problem using Semaphore)

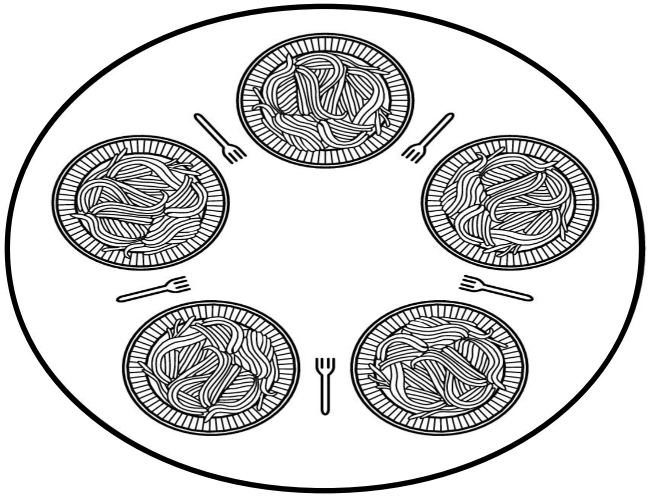
```
typedef int semaphore;                /* use your imagination */
semaphore mutex = 1;                  /* controls access to 'rc' */
semaphore db = 1;                     /* controls access to the database */
int rc = 0;                           /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                   /* repeat forever */
          down(&mutex);               /* get exclusive access to 'rc' */
          rc = rc + 1;                /* one reader more now */
          if (rc == 1) down(&db);     /* if this is the first reader ... */
          up(&mutex);                 /* release exclusive access to 'rc' */
          read_data_base( );          /* access the data */
          down(&mutex);               /* get exclusive access to 'rc' */
          rc = rc – 1;                /* one reader fewer now */
          if (rc == 0) up(&db);       /* if this is the last reader ... */
          up(&mutex);                 /* release exclusive access to 'rc' */
          use_data_read( );           /* noncritical region */
     }
}


void writer(void)
{
     while (TRUE) {                   /* repeat forever */
          think_up_data( );           /* noncritical region */
          down(&db);                  /* get exclusive access */
          write_data_base( );         /* update the data */
          up(&db);                    /* release exclusive access */
     }
}
```

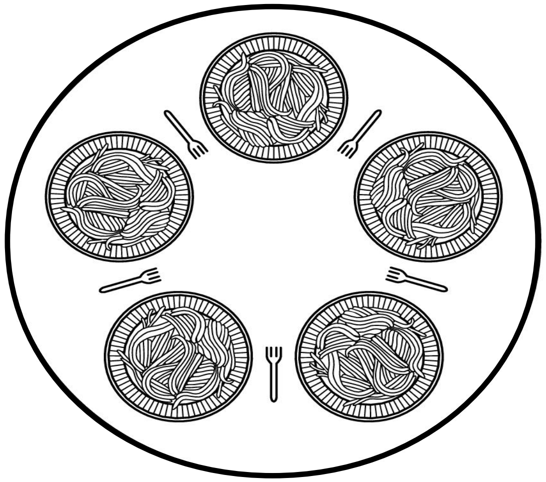
Classical IPC Problems
(Dining Philosophers Problem)

- Problem Definition
  - *There are 5 philosophers, 5 forks*
  - *Eating needs two forks*
  - *Pick one fork at a time*
  - *How to prevent deadlock (if every philosopher take their left forks simultaneously)*
  - *How to prevent starvation (All the philosopher starts the algorithm simultaneously, picking up their left fork, seeing that their right forks were not available, putting down their left forks, waiting and soon. Program continue to run but fail)*



Classical IPC Problems
(Dining Philosophers Problem)

- This problem models processes that are competing for exclusive access to a limited number of resources such as I/O devices.



```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                  /* philosopher is thinking */
          take_fork(i);              /* take left fork */
          take_fork((i+1) % N);      /* take right fork; % is modulo operator */
          eat( );                    /* yum-yum, spaghetti */
          put_fork(i);               /* put left fork back on the table */
          put_fork((i+1) % N);       /* put right fork back on the table */
     }
}
```

A nonsolution

Classical IPC Problems

(Dining Philosophers Problem using semaphore)

```
#define N            5            /* number of philosophers */
#define LEFT         (i+N−1)%N    /* number of i's left neighbor */
#define RIGHT        (i+1)%N      /* number of i's right neighbor */
#define THINKING     0            /* philosopher is thinking */
#define HUNGRY       1            /* philosopher is trying to get forks */
#define EATING       2            /* philosopher is eating */
typedef int semaphore;            /* semaphores are a special kind of int */
int state[N];                     /* array to keep track of everyone's state */
semaphore mutex = 1;              /* mutual exclusion for critical regions */
semaphore s[N];                   /* one semaphore per philosopher */

void philosopher(int i)           /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                /* repeat forever */
        think( );                 /* philosopher is thinking */
        take_forks(i);            /* acquire two forks or block */
        eat( );                   /* yum-yum, spaghetti */
        put_forks(i);             /* put both forks back on table */
    }
}
```

Classical IPC Problems

(Dining Philosophers Problem using semaphore)

```
void take_forks(int i)              /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = HUNGRY;            /* record fact that philosopher i is hungry */
      test(i);                      /* try to acquire 2 forks */
      up(&mutex);                   /* exit critical region */
      down(&s[i]);                  /* block if forks were not acquired */
}

void put_forks(i)                   /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = THINKING;          /* philosopher has finished eating */
      test(LEFT);                   /* see if left neighbor can now eat */
      test(RIGHT);                  /* see if right neighbor can now eat */
      up(&mutex);                   /* exit critical region */
}

void test(i)                        /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

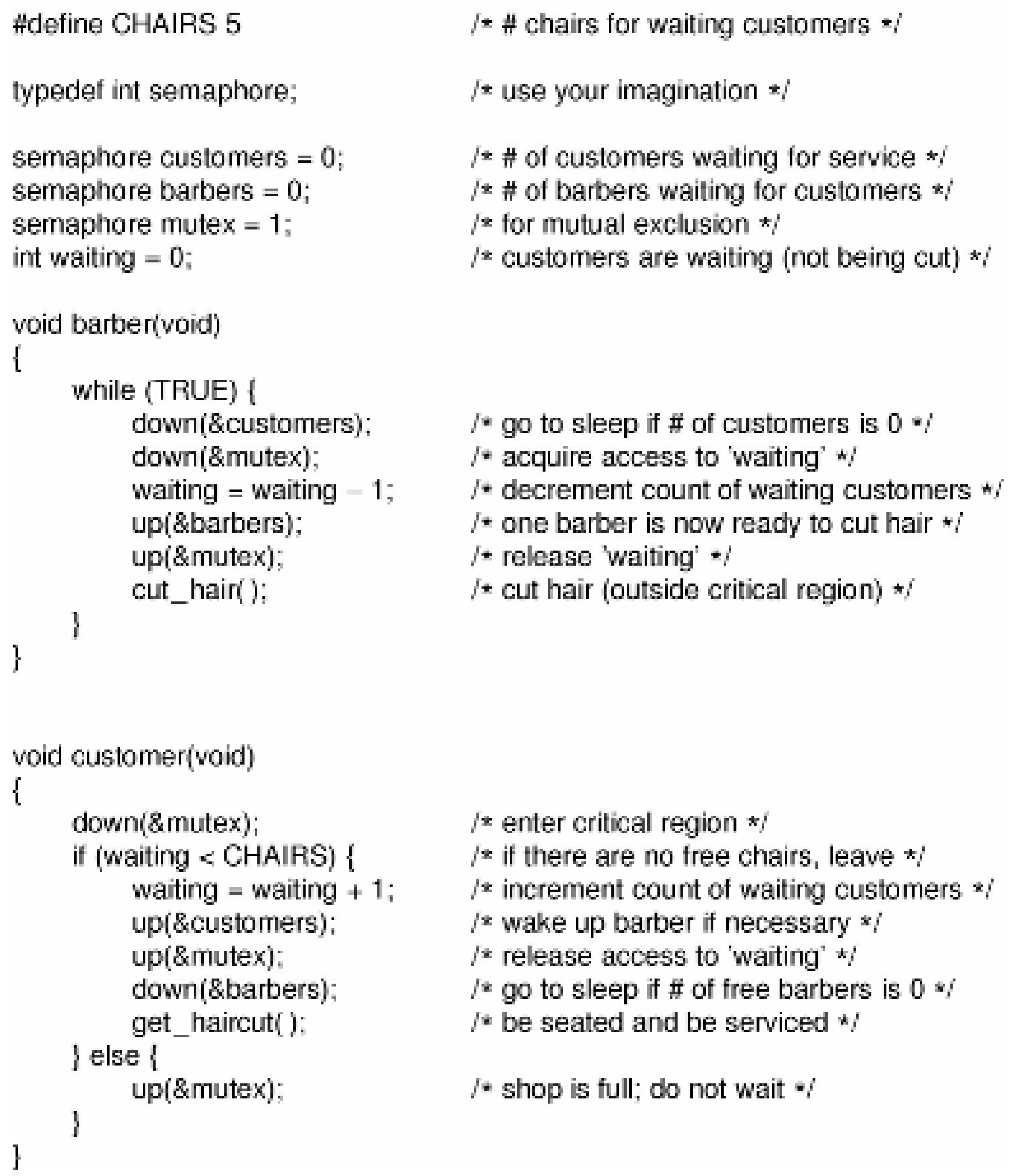
Classical IPC Problems

(The Sleeping Barber Problem)

- Problem Definition
  - *A barber shop with "N" waiting chairs and 1 barber chair*
  - *No customers → Barber sleeps*
  - *A customer enters and no free chairs → customer leaves*
  - *If the barber is busy, but chairs available → sit down*
  - *If the barber is asleep, wake him up*

- This problem models processes consists of queuing situations.
  - *For example, a multi-person helpdesk with computerized call waiting system for holding a limited number of incoming calls.*

Classical IPC Problems
(Solution to the Sleeping Barber Problem)



```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
     while (TRUE) {
          down(&customers);         /* go to sleep if # of customers is 0 */
          down(&mutex);             /* acquire access to 'waiting' */
          waiting = waiting − 1;    /* decrement count of waiting customers */
          up(&barbers);             /* one barber is now ready to cut hair */
          up(&mutex);               /* release 'waiting' */
          cut_hair( );              /* cut hair (outside critical region) */
     }
}


void customer(void)
{
     down(&mutex);                  /* enter critical region */
     if (waiting < CHAIRS) {        /* if there are no free chairs, leave */
          waiting = waiting + 1;    /* increment count of waiting customers */
          up(&customers);           /* wake up barber if necessary */
          up(&mutex);               /* release access to 'waiting' */
          down(&barbers);           /* go to sleep if # of free barbers is 0 */
          get_haircut( );           /* be seated and be serviced */
     } else {
          up(&mutex);               /* shop is full; do not wait */
     }
}
```

Problems with Semaphores

- They can be used to solve any of the traditional synchronization problems, but:
  - *Very hard to program* → *bad software engineering*
  - *There is no connection between the semaphore and the data being controlled by it*
  - *Used for both critical sections (mutual exclusion) and for coordination (scheduling)*

Problems with Semaphores

- No control over their use, no guarantee of proper usage:
- It is up to the programmer to correctly use the semaphores to access critical regions
  - *signal(mutex) <Critical Section> wait(mutex)* → *No mutual exclusion*
  - *wait(mutex) <Critical Section> wait(mutex)* → *Deadlock*
  - *<Critical Section> signal(mutex)* → *Omitted wait(mutex)*
  - *wait(mutex)<Critical Section>* → *Omitted signal(mutex): Deadlock*
- Thus, they are prone to bugs
  - *Another (better?) approach: use programming language support*
    - *Monitors with condition variables*

POSIX Semaphore

- POSIX

```
sem_t sem;
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
```

- Windows

```
HANDLE sem = CreateSemaphore(name, iniVal, maxVal)
WaitForSingleObject(sem, timeout)
ReleaseSemaphore(sem, val, &oldVal);
```

Monitors

- A programming language construct that supports controlled access to shared data
  - *Synchronization code added by compiler, enforced at runtime*
  - *Programmer is freed from using semaphores in the correct manner*
  - *Monitors are a language concept and C does not have them, but Pidgin, Java, C#, Ruby, Python, etc.*

- Monitor is a software module that encapsulates:
  - *shared data structures*
  - *procedures that operate on the shared data*
  - *synchronization between concurrent processes that invoke those procedures*

- Monitor protects the data from unstructured access
  - *Access data only through procedures*

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

Monitors

- Mutual exclusion
  - *only one process can be executing inside at any time*
    - *thus, synchronization implicitly associated with monitor*
  - *if a second process tries to enter a monitor procedure, it blocks until the first has left the monitor*
    - *more restrictive than semaphores!*
      - *Only a portion of the procedure may be the critical section!*
    - *but easier to use most of the time*

Monitors

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count − 1;
                if count = N − 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

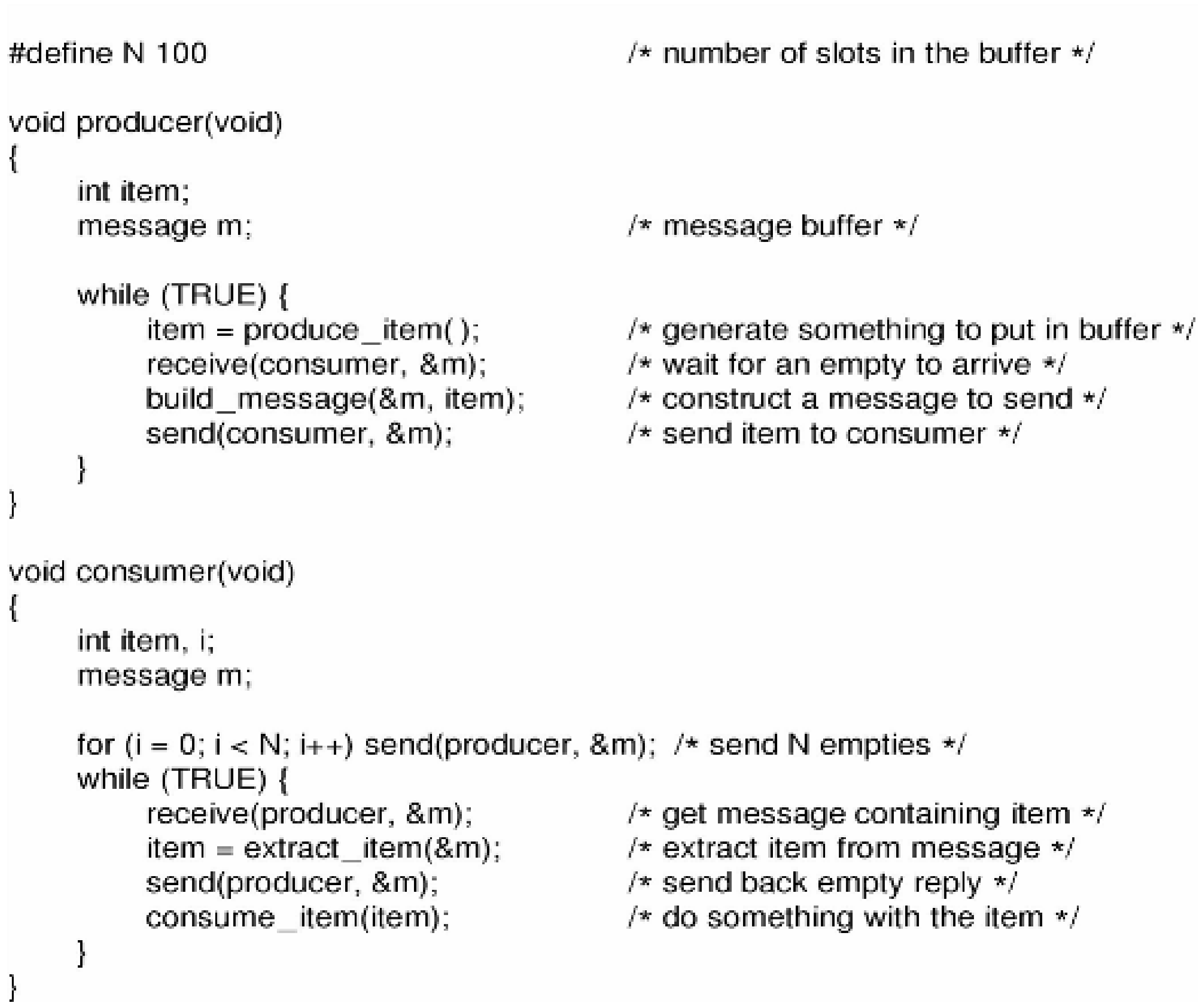
- Outline of producer-consumer problem with monitors
  - *only one monitor procedure active at one time*
  - *buffer has* N *slots*

Synchronization with Message Passing

- Locks, Semaphores and Monitors are all designed for solving mutual exclusion problem on one or more CPUs that ALL have access to a common memory
  - *Recall that threads share everything except local variables*
  - *Process share data using shared memory*

- In a distributed system consisting of several machines each with its own private memory, connected by a LAN, these primitives become inapplicable
  - *Locks, semaphores and monitors do not provide for information exchange between machines*
  - *How do we solve producer/consumer problem if producer is running in one machine, and the consumer is running in another?*
- Solution?
  - *Message Passing – Also applicable within a single machine*

Synchronization with Message Passing



```
#define N 100                                  /* number of slots in the buffer */

void producer(void)
{
      int item;
      message m;                               /* message buffer */

      while (TRUE) {
            item = produce_item( );            /* generate something to put in buffer */
            receive(consumer, &m);             /* wait for an empty to arrive */
            build_message(&m, item);           /* construct a message to send */
            send(consumer, &m);                /* send item to consumer */
      }
}

void consumer(void)
{
      int item, i;
      message m;

      for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
      while (TRUE) {
            receive(producer, &m);             /* get message containing item */
            item = extract_item(&m);           /* extract item from message */
            send(producer, &m);                /* send back empty reply */
            consume_item(item);                /* do something with the item */
      }
}
```

- The producer-consumer problem with N messages

Conclusion

- Threads/Processes cooperate
  - *Web Servers*
  - *File Systems*
  - *Database Management Systems*
  - *Almost every non-trivial application is multi-threaded*

- We need to control this cooperation so that the results are predictable
  - *Locks – Primitives using which everything else can be built*
  - *Semaphores & Mutexes – Dijkstra's synchronization tools*
  - *Monitors – PL constructs to make life easier*
  - *Message Passing – Used when data is not available in a common (shared) memory*
    - *Mostly used in distributed systems but applicable even in a single machine system*