Semaphores

- Chapter 3 used busy-waiting loops.
- Semaphores are easy to implement.
- Virtually all concurrent libraries include semaphores.
- Make critical sections easy.
- Can be used for signalling and scheduling.

```
sem mutex = 1;

process CS[i = 1 to n] {
  while (true) {
    P(mutex);
    critical section;
    V(mutex);
    noncritical section;
  }
}
```

Figure 4.1 Semaphore solution to the critical section problem.

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
  ...
  V(arrive1);    /* signal arrival         */
  P(arrive2);    /* wait for other process */
  ...
}
process Worker2 {
  ...
  V(arrive2);    /* signal arrival         */
  P(arrive1);    /* wait for other process */
  ...
}
```

Figure 4.2 Barrier synchronization using semaphores.

- A two-process barrier.
- Can use `count` to generalize (see Downey).
- Can use butterfly or dissemination-barrier.

```
typeT buf;     /* a buffer of some type T */
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
  while (true) {
    ...
    /* produce data, then deposit it in the buffer */
    P(empty);
    buf = data;
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    /* fetch result, then consume it */
    P(full);
    result = buf;
    V(empty);
    ...
  }
}
```

Figure 4.3 Producers and consumers using semaphores.

- Split binary semaphores.

```
typeT buf[n];      /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;   /* n-2 <= empty+full <= n */
process Producer {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    buf[rear] = data; rear = (rear+1) % n;
    V(full);
  }
}
process Consumer {
  while (true) {
    fetch message result and consume it;
    P(full);
    result = buf[front]; front = (front+1) % n;
    V(empty);
    ...
  }
}
```

Figure 4.4 Bounded buffer using semaphores.

- `empty` is initalized to `n`

```
typeT buf[n];      /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;     /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1;  /* for mutual exclusion   */
process Producer[i = 1 to M] {
  while (true) {
    ...
    produce message data and deposit it in the buffer;
    P(empty);
    P(mutexD);
    buf[rear] = data; rear = (rear+1) % n;
    V(mutexD);
    V(full);
  }
}
process Consumer[j = 1 to N] {
  while (true) {
    fetch message result and consume it;
    P(full);
    P(mutexF);
    result = buf[front]; front = (front+1) % n;
    V(mutexF);
    V(empty);
    ...
  }
}
```

Figure 4.5 Multiple producers and consumers using semaphores.

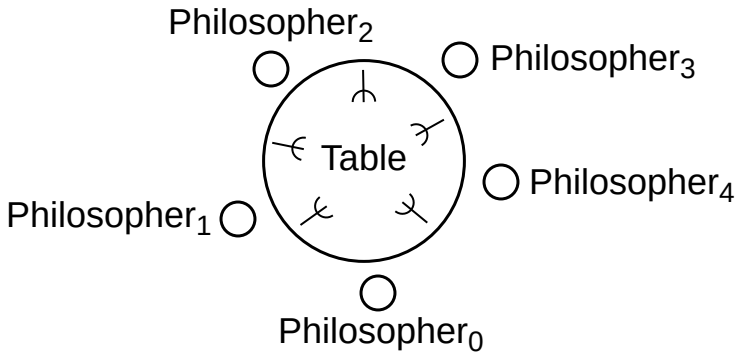

Figure 4.6 The dining philosophers.


```
process Philosopher[i = 0 to 4] {
  while (true) {
    think;
    acquire forks;
    eat;
    release forks;
  }
}
```

```
sem fork[5] = {1, 1, 1, 1, 1};
process Philosopher[i = 0 to 3] {
  while (true) {
    P(fork[i]); P(fork[i+1]); # get left fork then right
    eat;
    V(fork[i]); V(fork[i+1]);
    think;
  }
}
process Philosopher[4] {
  while (true) {
    P(fork[0]); P(fork[4]);   # get right fork then left
    eat;
    V(fork[0]); V(fork[4]);
    think;
  }
}
```

Figure 4.7 Dining philosophers solution using semaphores.

```
sem rw = 1;
process Reader[i = 1 to M] {
  while (true) {
    ...
    P(rw);    # grab exclusive access lock
    read the database;
    V(rw);    # release the lock
  }
}
process Writer[j = 1 to N] {
  while (true) {
    ...
    P(rw);    # grab exclusive access lock
    write the database;
    V(rw);    # release the lock
  }
}
```

Figure 4.8 An overconstrained readers/writers solution.

```
int nr = 0;       # number of active readers
sem rw = 1;       # lock for reader/writer exclusion
process Reader[i = 1 to M] {
  while (true) {
    ...
    ⟨ nr = nr+1;
      if (nr == 1) P(rw);  # if first, get lock
    ⟩
    read the database;
    ⟨ nr = nr-1;
      if (nr == 0) V(rw);  # if last, release lock
    ⟩
  }
}
process Writer[j = 1 to N] {
  while (true) {
    ...
    P(rw);
    write the database;
    V(rw);
  }
}
```

Figure 4.9 Outline of readers and writers solution.

```
int nr = 0;       # number of active readers
sem rw = 1;       # lock for access to the database
sem mutexR = 1;   # lock for reader access to nr
process Reader[i = 1 to m] {
  while (true) {
    ...
    P(mutexR);
      nr = nr+1;
      if (nr == 1) P(rw);  # if first, get lock
    V(mutexR);
    read the database;
    P(mutexR);
      nr = nr-1;
      if (nr == 0) V(rw);  # if last, release lock
    V(mutexR);
  }
}
process Writer[j = 1 to n] {
  while (true) {
    ...
    P(rw);
    write the database;
    V(rw);
  }
}
```

Figure 4.10 Readers and writers exclusion using semaphores.

```
int nr = 0, nw = 0;
## RW:  (nr == 0 ∨ nw == 0) ∧ nw <= 1
process Reader[i = 1 to m] {
  while (true) {
    ...
    ⟨await (nw == 0) nr = nr+1;⟩
    read the database;
    ⟨nr = nr-1;⟩
  }
}
process Writer[j = 1 to n] {
  while (true) {
    ...
    ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
    write the database;
    ⟨nw = nw-1;⟩
  }
}
```

Figure 4.11 A coarse-grained readers/writers solution.

```
if (nw == 0 and dr > 0) {
  dr = dr-1; V(r);  # awaken a reader, or
  }
elseif (nr == 0 and nw == 0 and dw > 0) {
  dw = dw-1; V(w);  # awaken a writer, or
  }
else
  V(e);             # release the entry lock
```

SIGNAL code for Figure 4.12.

- Signal *exactly one* of the semaphores.
- `r`, `w`, and `e` are a split binary semaphore.
- **Passing the baton** to one another.

```
int nr = 0,   ## RW: (nr == 0 or nw == 0) and nw <= 1
    nw = 0;
sem e = 1,    # controls entry to critical sections
    r = 0,    # used to delay readers
    w = 0;    # used to delay writers
              # at all times 0 <= (e+r+w) <= 1
int dr = 0,   # number of delayed readers
    dw = 0;   # number of delayed writers
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r); }
      nr = nr+1;
      SIGNAL;      # see text for details
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      SIGNAL;
  }
}
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      SIGNAL;
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      SIGNAL;
  }
}
```

Figure 4.12 Outline of readers and writers with passing the baton.

```
int nr = 0,   ## RW: (nr==0 or nw==0) and nw<=1
    nw = 0;
sem e = 1,    # controls entry to critical sections
    r = 0,    # used to delay readers
    w = 0;    # used to delay writers
              # at all times 0 <= (e+r+w) <= 1
int dr = 0,   # number of delayed readers
    dw = 0;   # number of delayed writers
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r); }
      nr = nr+1;
      if (dr > 0) { dr = dr-1; V(r); }
      else V(e);
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      if (nr == 0 and dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      V(e);
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      if (dr > 0) { dr = dr-1; V(r); }
      elseif (dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
```

```
bool free = true;
sem e = 1, b[n] = ([n] 0);  # for entry and delay
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
## SJN:  pairs is an ordered set ∧ free ⇒ (pairs == ∅)
request(time,id):
  P(e);
  if (!free) {
    insert (time,id) in pairs;
    V(e);           # release entry lock
    P(b[id]);       # wait to be awakened
  }
  free = false;
  V(e);    # optimized since free is false here
release():
  P(e);
  free = true;
  if (P != ∅) {
    remove first pair (time,id) from pairs;
    V(b[id]);  # pass baton to process id
  }
  else V(e);
```

Figure 4.14 Shortest-job-next allocation using semaphores.

```
#include <pthread.h>      /* standard lines */
#include <semaphore.h>
#define SHARED 1
#include <stdio.h>

void *Producer(void *);  /* the two threads */
void *Consumer(void *);

sem_t empty, full;       /* global semaphores */
int data;                /* shared buffer     */
int numIters;

/* main() -- read command line and create threads */
int main(int argc, char *argv[]) {
  pthread_t pid, cid;     /* thread and attributes */
  pthread_attr_t attr;    /* descriptors           */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  sem_init(&empty, SHARED, 1);  /* sem empty = 1 */
  sem_init(&full, SHARED, 0);   /* sem full = 0  */

  numIters = atoi(argv[1]);
  pthread_create(&pid, &attr, Producer, NULL);
  pthread_create(&cid, &attr, Consumer, NULL);
  pthread_join(pid, NULL);
  pthread_join(cid, NULL);
}

/* deposit 1, ..., numIters into the data buffer */
void *Producer(void *arg) {
  int produced;
  for (produced = 1; produced <= numIters; produced++) {
    sem_wait(&empty);
    data = produced;
    sem_post(&full);
  }
}

/* fetch numIters items from the buffer and sum them */
void *Consumer(void *arg) {
  int total = 0, consumed;
  for (consumed = 1; consumed <= numIters; consumed++) {
    sem_wait(&full);
    total = total + data;
    sem_post(&empty);
  }
  printf("the total is %d\n", total);
}
```

Figure 4.15 Simple producer/consumer using Pthreads.