3) Double Buffering

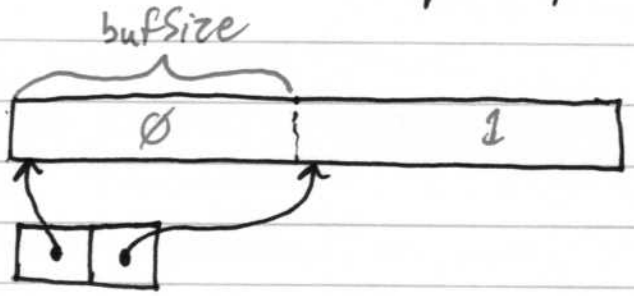
- 2 message buffers allow parallel production and consumption
- used in graphics to avoid "tearing"
- we need a pair of semaphores (empty, full) per buffer

e.g.

producer	consumer	buffer 0 empty	buffer 0 full	buffer 1 empty	buffer 1 full	sum empty	sum full
		1	0	1	0	2	0
wait b0 empty	wait b0 full	0	0	1	0	1	0
produce	block						
signal b0 full		0	1	1	0	1	1
wait b1 empty	unblock	0	0	0	0	0	0
produce	consume						
signal b1 full	signal b0 empty	1	0	0	1	1	1
wait b0 empty	wait b1 full	0	0	0	0	0	0
produce	consume						
signal b0 full	signal b1 empty	0	1	1	0	1	1
⋮	⋮						

- we can combine the "empty" semaphores and initialize to 2
- we can combine the "full" semaphores and initialize to 0
  - as long as the producer(s) and consumer(s) use the buffers in the same order

- we can statically allocate the buffers and pass pointers to them to the tasks



① statically allocated storage

② array of buffer pointers

e.g. two 512B buffers

```
const uint32_t bufSize = 512;
① uint8_t  bufData[bufSize * 2];
② uint8_t *bufPtr[2] = {bufData, bufData + bufSize};
                        &bufData[bufSize]
```

```
sem_t empty, full;
init(&empty, 2);
init(&full,  0);
```

producer

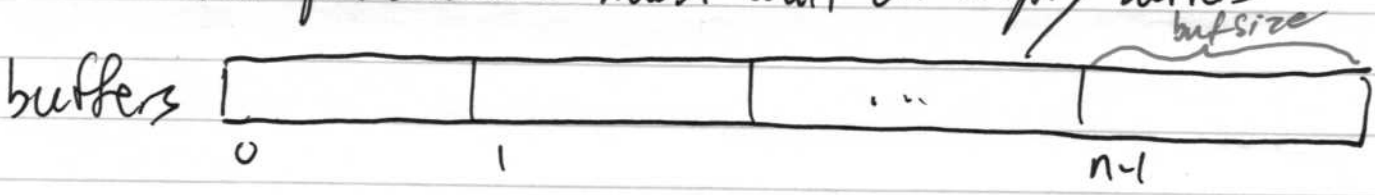
```
uint32_t index = 0;

while(true) {
    wait(&empty);
    ... fill buffer at bufPtr[index]
    signal(&full);
    index = (index + 1) % 2;
}
```

consumer

see assignment 4

4) Bounded-Buffer Problem

- generalizes double-buffering to n buffers
- "bounded" refers to the fact that the buffer is not infinite
  i.e. producers must wait on empty buffers



- solution: use an array-based queue of buffers (or buffer pointers)
  - two semaphores: empty = n
    full = 0
  - two indices: tail = 0 for producers
    head = 0 for consumers
  - two mutexes: tailMutex protects tail index updates by multiple producers
    headMutex protects head index updates by multiple consumers

- see assignment 4

5) CMSIS-RTX Queues

- two types:

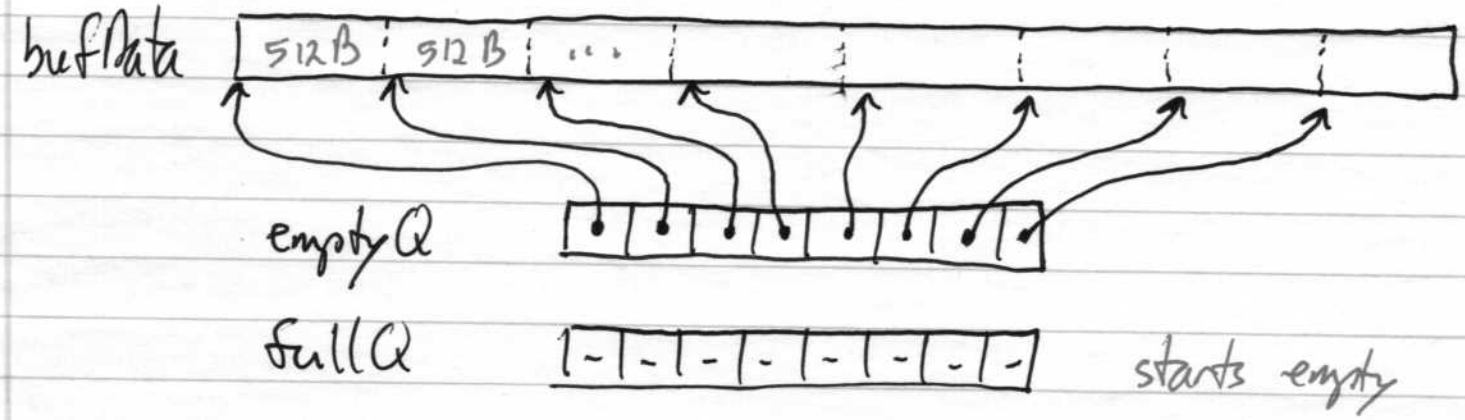
① osMessageQ - passes uint32_t or void * between threads
- user manages storage

② osMailQ - passes pointers to storage managed by the queue

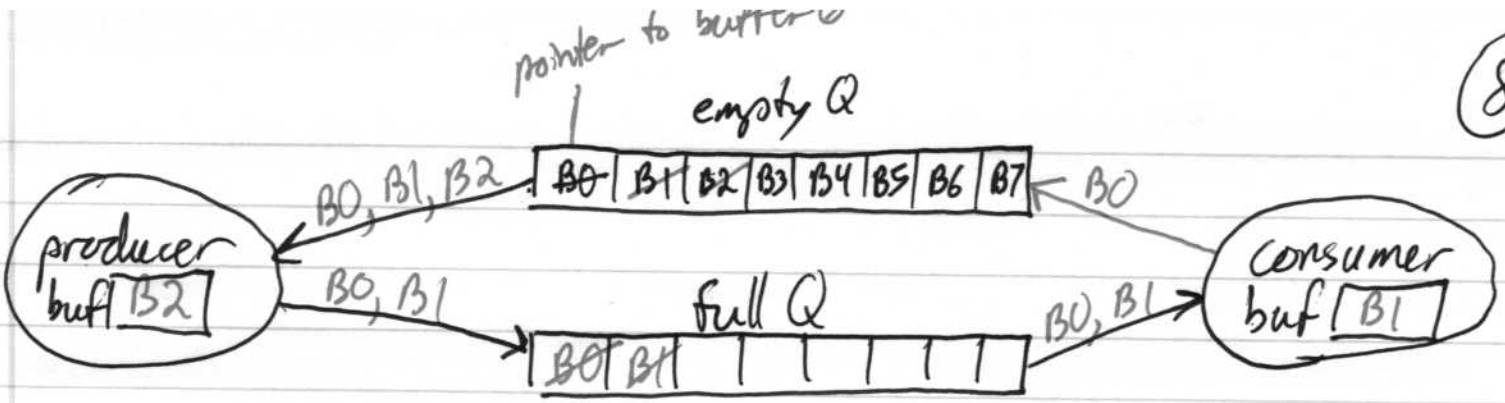
e.g. bounded-buffer solution using osMessageQ
- 8 512B buffers, 2 producers, 2 consumers

```
const uint32_t  bufSize = 512;
const uint32_t  n = 8;
uint8_t  bufData[n * bufSize];
```
} data segment

```
osMessageQDef(emptyQ, n, uint8_t *);   // queue name, queue size, element type
osMessageQId (emptyQId);
osMessageQDef(fullQ, n, uint8_t *);
osMessageQId (fullQId);
```



```
int main(void) {

    osKernelInitialize();

    emptyQId = osMessageCreate(osMessageQ(emptyQ),
                               NULL);   // always NULL
    fullQId = osMessageCreate(osMessageQ(fullQ), NULL);

    osKernel start();

    for (uint32_t i=0; i<n; i++)
        osMessagePut(emptyQId,
            (uint32_t)(bufData + i*bufSize), osWaitForever);
                       // &bufData[i*bufSize]   args

    osThreadCreate(osThread(producer), NULL);
    osThreadCreate(osThread(consumer), NULL);
}
```




```
void producer (void const *args) {
  while (true) {
    osEvent event = osMessageGet(emptyQId, osWaitForever);

    uint8_t *buf =
      event.value.p;

    ... fill buffer pointed to
        by buf

    osMessagePut(fullQId, (uint32_t)buf, osWaitForever);
  }
}
```

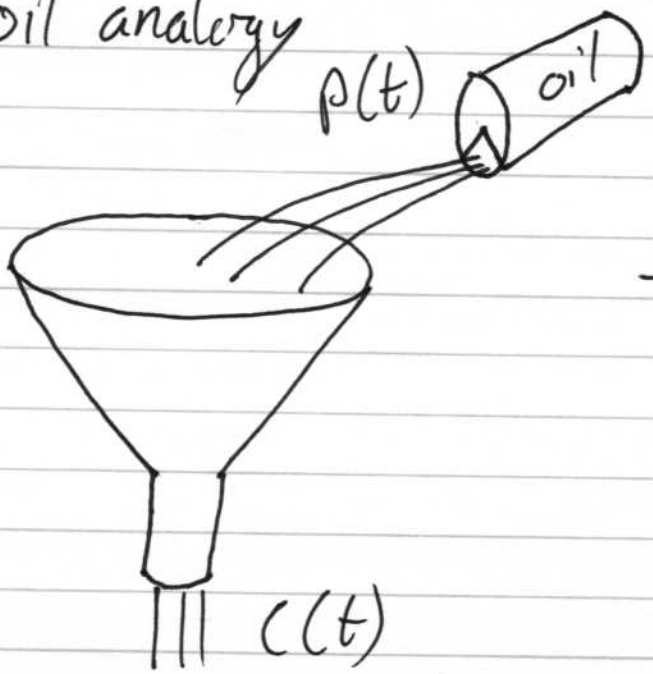
```
typedef struct {
  osStatus status;
  union {
    uint32_t v;
    void *p;
    int32_t signals;
  } value;
  ...
} osEvent;
```

- consumer is the mirror image of the producer

6) Buffer Sizing

- buffers allow production rates $P(t)$ to temporarily exceed consumption rates $C(t)$ for limited periods of time

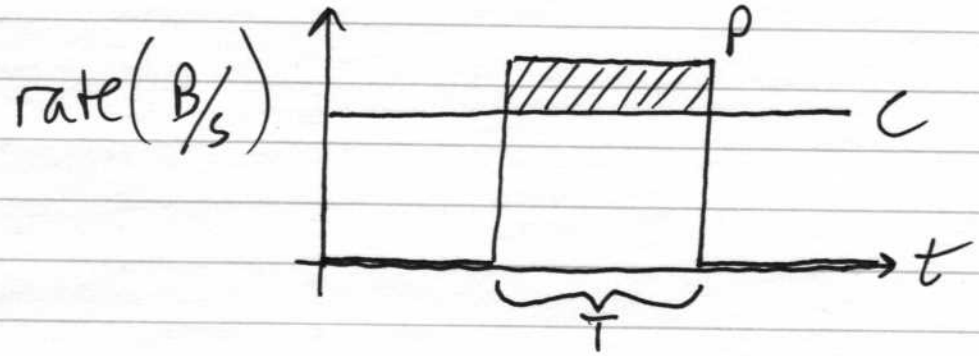
- engine oil analogy [LaPlante, 1997]



- the funnel buffers the oil so none is lost (if it is big enough)

6.1) Constant $P(t)_1^{burst}$ and Constant $C(t)$

- if $P > C$ for limited time T, how big should the buffer be?



- data produced during burst = PT
- data consumed during burst = CT
- buffer size $B = PT - CT = (P-C)T$

e.g. I/O device produces data at 9600 B/s for a 1 s burst every 20 s
- a consumer task consumes this data at 800 B/s

$$B = (9600\ B/s - 800\ B/s) \times 1\ s = 8800\ B$$

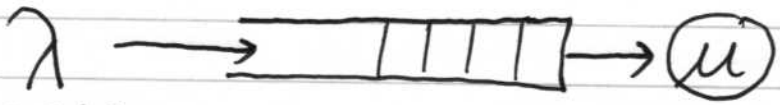
$\therefore$ buffer size should be ≥ 8800 bytes

- this assumes that there is time to drain the buffer between bursts

$$t = 8800\ B / 800\ B/s = 11\ s \quad (< 19\ s)$$ ✓

7) Queuing Theory [15.2]

- models the real world as client/server systems
  e.g. clients lining up for service at a bank
  requests arriving at a web server
- clients arrive and wait in a queue to be served
  (producer = client, consumer = server)



λ: average client arrival rate

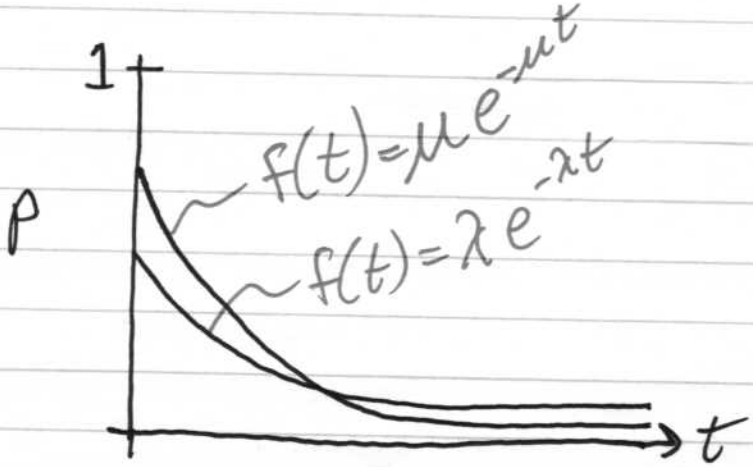
μ: average service rate

$$\rho = \text{load factor} = \frac{\lambda}{\mu}$$

- the arrival and service rates can be:
  1. deterministic 'D'
     - clients arrive with regular interval $T = \frac{1}{\lambda}$
     - server time = $\frac{1}{\mu}$

(2) Markov 'M'

- a Markov process is a memoryless stochastic process
  - stochastic process: a sequence of random variables
  - memoryless: time between each arrival/service is independent of the previous time



probability density function
- probability of the next event occurring at time t

- Kendall notation for queuing models:

arrival model / service model / n

- arrival model: D or M
- service model: D or M
- n: # of servers

7.1) M/M/1 Queues

- equations:

average clients in system $L = \frac{\rho}{1-\rho}$

queued + the client being served

average time in system $W = \frac{1/\mu}{1-\rho}$

time waiting + time being served

Little's Rule $L = \lambda W \left(= \lambda \frac{1/\mu}{1-\rho} = \frac{\lambda/\mu}{1-\rho} = \frac{\rho}{1-\rho} \right)$