Operating Systems 412

Pete Keleher

Producer-Consumer preface

```
int buffer[MAX];
int fill = 0;
int use  = 0;

void put(int value) {
    buffer[fill] = value;     // Line F1
    fill = (fill + 1) % MAX; // Line F2
}

int get() {
    int tmp = buffer[use];    // Line G1
    use = (use + 1) % MAX;    // Line G2
    return tmp;
}
```

Figure 31.9: **The Put And Get Routines**

Prod-Cons semaphores

Assume MAX = 1,
initially empty,
1 consumer, 1 producer

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);         // Line P1
        put(i);                   // Line P2
        sem_post(&full);          // Line P3
    }
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);          // Line C1
        tmp = get();              // Line C2
        sem_post(&empty);         // Line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    // ...
}
```

Figure 31.10: **Adding The Full And Empty Conditions**

Prod-Cons semaphores, flawed

Assume MAX = 10,
initially empty,
1 consumer, 1 producer

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);         // Line P1
        put(i);                   // Line P2
        sem_post(&full);          // Line P3
    }
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);          // Line C1
        tmp = get();              // Line C2
        sem_post(&empty);         // Line C3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0);    // 0 are full
    // ...
}
```

Problem is we are not enforcing mutual exclusion over the put() and get().

Need to add mutual exclusion back in!

Figure 31.10: **Adding The Full And Empty Conditions**

Prod-Cons semaphores, fixed

Deadlock!

empty buffer
consumer runs, blocks
producer runs, blocks

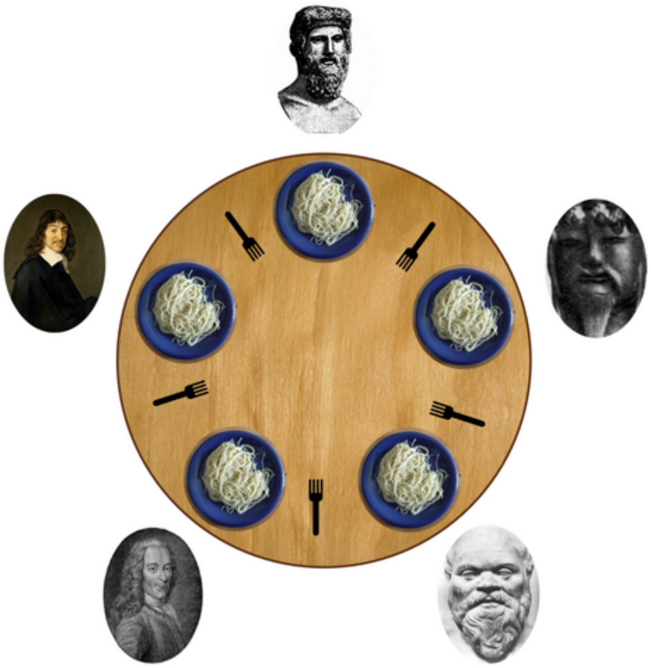
```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line P0 (NEW LINE)
        sem_wait(&empty);        // Line P1
        put(i);                  // Line P2
        sem_post(&full);         // Line P3
        sem_post(&mutex);        // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line C0 (NEW LINE)
        sem_wait(&full);         // Line C1
        int tmp = get();         // Line C2
        sem_post(&empty);        // Line C3
        sem_post(&mutex);        // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

Prod-Cons semaphores fixed, again

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);        // Line P1
        sem_wait(&mutex);        // Line P1.5 (MUTEX HERE)
        put(i);                  // Line P2
        sem_post(&mutex);        // Line P2.5 (AND HERE)
        sem_post(&full);         // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);         // Line C1
        sem_wait(&mutex);        // Line C1.5 (MUTEX HERE)
        int tmp = get();         // Line C2
        sem_post(&mutex);        // Line C2.5 (AND HERE)
        sem_post(&empty);        // Line C3
        printf("%d\n", tmp);
    }
}
```

Figure 31.12: **Adding Mutual Exclusion (Correctly)**

Dining Philosophers! semaphores



```
while (1) {
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

What could go wrong?

from Wikipedia

Dining Philosophers! semaphores

```
void get_forks(int p) {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
}

void put_forks(int p) {
    sem_post(&forks[left(p)]);
    sem_post(&forks[right(p)]);
}
```

- What could go wrong?
  - deadlock
- The problem:
  - symmetry
- Fix:
  - introduce some asymmetry

Reader-writer Locks

Either:

- one or more readers, or
- a single writer

may be in the critical section at one time.

Reader-writer Locks semaphores

Issues?
How to fix?

```
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // allow ONE writer/MANY readers
  int   readers;   // #readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}
```

readers

```
void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}
```

writer

```
void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Figure 31.13: **A Simple Reader-Writer Lock**

Mutual Exclusion just loads + stores

- How to enforce mutual exclusion?
  - use a simple flag on memory
  - essentially a spinlock w/ just no atomic instr
- Issues:
  - correctness?
  - performance?

$t_1$1
context switch

$t_2$1
$t_2$3
context switch

$t_1$3
both now in mutex

t_1
```
typedef struct { int flag; } lock_t;

void init(lock_t *mutex) {
     mutex->flag = 0;
}

void lock(lock_t *mutex) {
     while (mutex->flag == 1)     (1)
           ;                      (2)
     mutex->flag = 1;             (3)
}

void unlock lock_t *mutex) {
     mutex->flag = 0;             (5)
}
```

t_2
```
typedef struct { int flag; } lock_t;

void init(lock_t *mutex) {
     mutex->flag = 0;
}

void lock(lock_t *mutex) {
     while (mutex->flag == 1)     (1)
           ;                      (2)
     mutex->flag = 1;             (3)
}

void unlock lock_t *mutex) {
     mutex->flag = 0;             (5)
}
```

Mutual Exclusion peterson's algorithm

```
int flag[2] = {false, false};

int turn;
```

```
P0:  flag[0] = true;
     turn = 1;
     while (flag[1] && turn == 1)
     {
          // busy wait
     }

     // critical section
        ...
     // end of critical section

     flag[0] = false;
```

```
P1:  flag[1] = true;
     turn = 0;
     while (flag[0] && turn == 0)
     {
          // busy wait
     }

     // critical section
        ...
     // end of critical section

     flag[1] = false;
```

- Preemption?
- Mutual exclusion?
- Progress?
- Note that ordinary loads and stores aren't atomic anymore….

Mutual Exclusion atomic instructions

- What can we do w/ *atomic instructions ??*
  - spinlock!
  - work on multi-core machine?

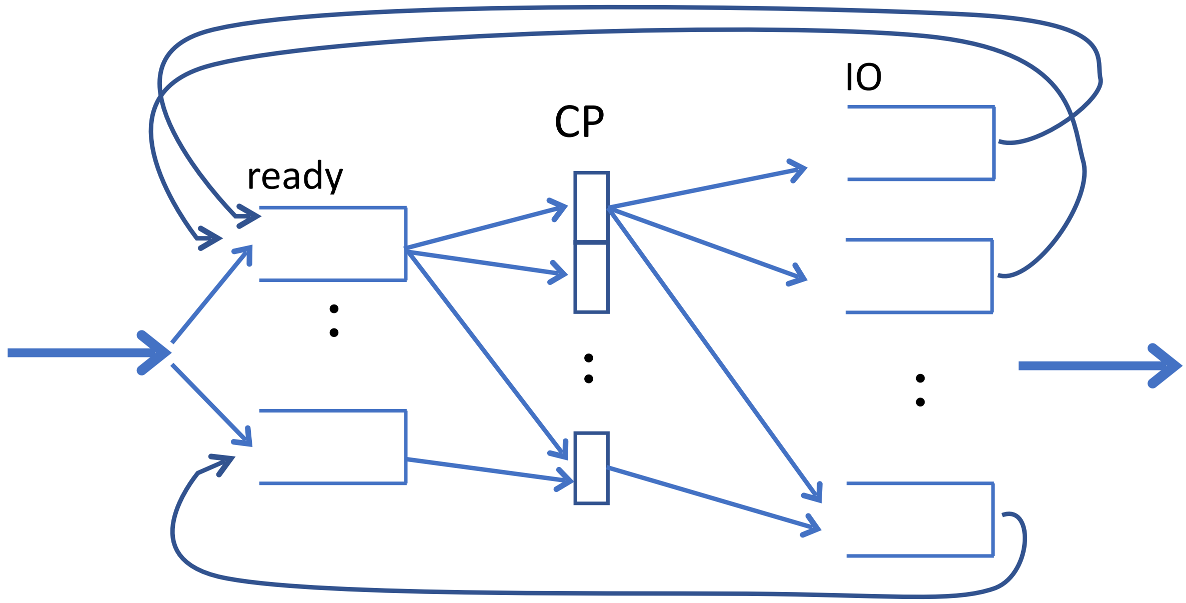
Yes! In fact it's an important use case…

```
Spin_Lock_INTERNAL:
     mov     ecx, [esp+4]

.still_locked_early:
     mov eax, [ecx]
     test eax, eax
     jnz .still_locked_early

.seems_unlocked:
     mov     eax, 1
     xchg    eax, [ecx]
     test    eax, eax
     jnz     Spin_Lock_INTERNAL
     inc     dword [lockops]
     ret

_____________________________-

Spin_Unlock_INTERNAL:
     mov     ecx, [esp+4]
     mov     eax, 0
     xchg    eax, [ecx]
     ret
```

Finishing Up mutual exclusion

- disabling interrupts
  - doesn't help w/ multi-core
- using only loads and stores
  - very cumbersome, inflexible
- using atomic instructions
  - works through memory, which is shared across cores/cpus
- locks / condition variables / semaphores
  - uses atomic instructions and *blocking*
  - more efficient
  - probably more correct than your code

Queuing Theory without probabilities



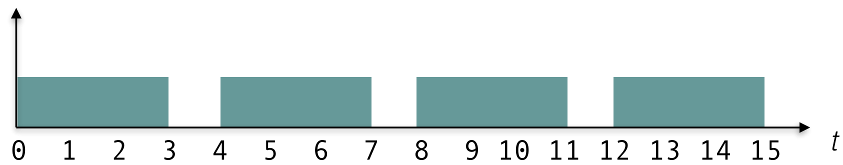
Queuing Theory without probabilities

- Queueing system
  - servers + waiting rooms
  - customers arrive, wait, get served, depart or go to next server
  - queueing disciplines
    - non-preemptive: fifo, priority, …
    - preemptive: round-robin, multi-level feedback, ...
- Operating systems are examples of queueing systems
  - servers: hw/sw resources (cpu, disk, req handler, …)
  - customers: PCBs, TCBs, ...
- Given: arrival rates, service times, queueing disciplines, ...
- Obtain: queue sizes, response times, fairness, bottlenecks, ...

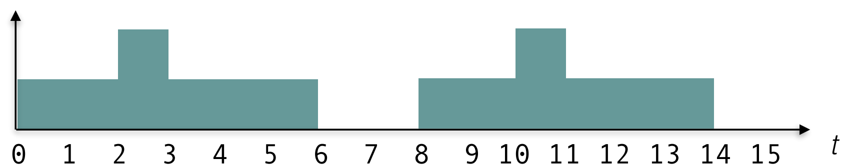
Queuing Theory without probabilities

- Consider cars traveling on a road with a turn
  - each car takes 3 seconds to go through the turn
  - at most one car can be in the turn at any time
- N(t): # cars in the turn and waiting to enter the turn

arrival rate 1/4
load 3/4
uniform



arrival rate 1/4
load 3/4
uniform



- Load < 1: *stable* w/ waits depending on burstiness
- Load > 1: *unstable*, ever-increasing waits

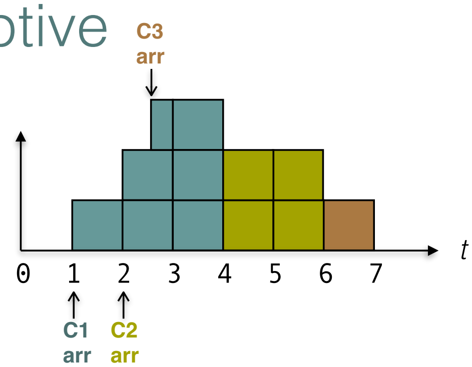
Queuing Theory without probabilities

- Assume unending stream of customers:
  - arrival rate λ or X: *# arrivals per second*
  - average service time S: *work needed per customer*
  - average response time R: *departure time D - arrival time A*
  - average wait time W: *response time - service time*
  - throughput X: *# departures per sec averaged over all time*
  - average customers in system N: *waiting or busy*
  - utilization U: *fraction of time server is busy*
- Typical goal
  - Given: arrival rate, avg service time, queueing discipline
  - Obtain: average response time, average queue size
- Little's Law (for any steady-state system):
  - $N = \lambda \times R$

FCFS non-preemptive

customer	A_i	S_i	D_i	R_i	W_i
1	1.0	3.0	4.0	3.0	0.0
2	2.0	2.0	6.0	4.0	2.0
3	2.5	1.0	7.0	4.5	3.5

repeats every 10 seconds

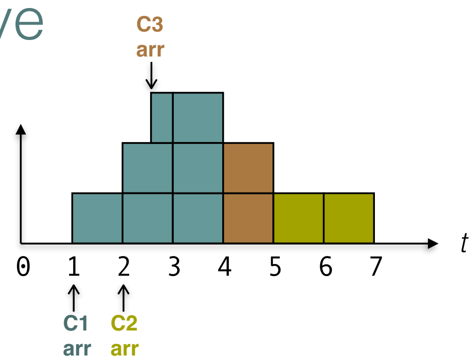


- System becomes empty at time 7 —> *stable*
  - Average response time: $R = \dfrac{3.0 + 4.0 + 4.0}{3} = \dfrac{11.5}{3}$ sec
  - Average wait time: $W = \dfrac{0.0 + 2.0 + 3.5}{3} = \dfrac{5.5}{3}$ sec
  - Arrival rate = throughput: $\lambda = \dfrac{3}{10}$ arrivals / sec
  - Utilization: $U = \dfrac{6}{10}$
  - Average number customers: $N = \lambda \times R = \dfrac{3}{10} \times \dfrac{11.5}{3} = \dfrac{11.5}{10}$

SJF non-preemptive

customer	A_i	S_i	D_i	R_i	W_i
1	1.0	3.0	4.0	3.0	0.0
2	2.0	2.0	7.0	5.0	3.0
3	2.5	1.0	5.0	2.5	1.5

repeats every 10 seconds

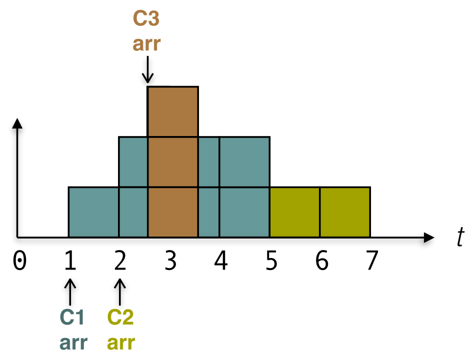


- System becomes empty at time 7 —> *stable*
  - Average response time: $R = \dfrac{3.0 + 5.0 + 2.5}{3} = \dfrac{10.5}{3}$ sec
  - Average wait time: $W = \dfrac{0.0 + 3.0 + 1.5}{3} = \dfrac{4.5}{3}$ sec
  - Arrival rate = throughput: $\lambda = \dfrac{3}{10}$ arrivals/sec
  - Utilization: $U = \dfrac{6}{10}$
  - Average number customers: $N = \lambda \times R = \dfrac{3}{10} \times \dfrac{10.5}{3} = \dfrac{10.5}{10}$

SJS preemptive

customer	A_i	S_i	D_i	R_i	W_i
1	1.0	3.0	5.0	4.0	1.0
2	2.0	2.0	7.0	5.0	3.0
3	2.5	1.0	3.5	1.0	0.0

repeats every 10 seconds

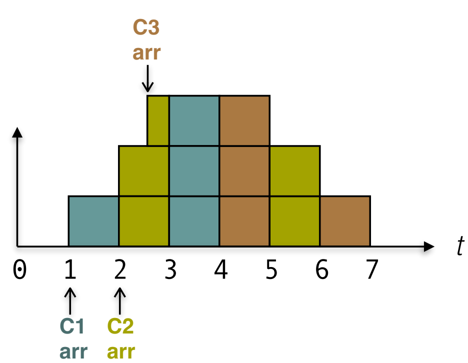


- System becomes empty at time 7 —> *stable*
  - Average response time: $R = \frac{4.0+5.0+1.0}{3} = \frac{10.0}{3}$ sec
  - Average wait time: $W = \frac{1.0+3.0+0.0}{3} = \frac{4.0}{3}$ sec
  - Arrival rate = throughput: $\lambda = \frac{3}{10}$ arrivals / sec
  - Utilization: $U = \frac{6}{10}$
  - Average number customers: $N = \lambda \times R = \frac{3}{10} \times \frac{10.0}{3} = \frac{10}{10}$

RR preemptive

customer	A_i	S_i	D_i	R_i	W_i
1	1.0	3.0	7.0	6.0	3.0
2	2.0	2.0	6.0	4.0	2.0
3	2.5	1.0	5.0	2.5	1.5

repeats every 10 seconds



- System becomes empty at time 7 —> *stable*
  - Average response time: $R = \frac{6.0+4.0+2.5}{3} = \frac{12.5}{3}$ sec
  - Average wait time: $W = \frac{3.0+2.0+1.5}{3} = \frac{6.5}{3}$ sec
  - Arrival rate = throughput: $\lambda = \frac{3}{10}$ arrivals / sec
  - Utilization: $U = \frac{6}{10}$
  - Average number customers: $N = \lambda \times R = \frac{3}{10} \times \frac{12.5}{3} = \frac{12.5}{10}$