Operating Systems 412

Producer-Consumer preface

```c
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

```c
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);
        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

Problem is we are not enforcing mutual exclusion over the put() and get(). Need to add mutual exclusion back in!

```c
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);
        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, fixed

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty); // Line P0 (NEW LINE)
        sem_wait(&mutex);  // 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

```c
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

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.

```c
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);
}

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);
}

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**

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

- Issues:
  - correctness?
  - performance?

```c
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)
}
```

- 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

- **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

\[
\begin{align*}
\text{arrival rate} & \quad 1/4 \\
\text{load} & \quad 3/4 \\
\text{uniform} & \\
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 & \quad 8 & \quad 9 & \quad 10 & \quad 11 & \quad 12 & \quad 13 & \quad 14 & \quad 15 \quad t
\end{align*}
\]

- 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 \( \lambda \) 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

<table>
<thead>
<tr>
<th>customer</th>
<th>( A_i )</th>
<th>( S_i )</th>
<th>( D_i )</th>
<th>( R_i )</th>
<th>( W_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>3.0</td>
<td>4.0</td>
<td>3.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>2.0</td>
<td>6.0</td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
<td>1.0</td>
<td>7.0</td>
<td>4.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Repeats every 10 seconds

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

**SJF** non-preemptive

<table>
<thead>
<tr>
<th>customer</th>
<th>( A_i )</th>
<th>( S_i )</th>
<th>( D_i )</th>
<th>( R_i )</th>
<th>( W_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>3.0</td>
<td>4.0</td>
<td>3.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>2.0</td>
<td>7.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
<td>1.0</td>
<td>5.0</td>
<td>2.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Repeats every 10 seconds

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

<table>
<thead>
<tr>
<th>customer</th>
<th>$A_i$</th>
<th>$S_i$</th>
<th>$D_i$</th>
<th>$R_i$</th>
<th>$W_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>3.0</td>
<td>5.0</td>
<td>4.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>2.0</td>
<td>7.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
<td>1.0</td>
<td>3.5</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>customer</th>
<th>$A_i$</th>
<th>$S_i$</th>
<th>$D_i$</th>
<th>$R_i$</th>
<th>$W_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>3.0</td>
<td>7.0</td>
<td>6.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>2.0</td>
<td>6.0</td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
<td>1.0</td>
<td>5.0</td>
<td>2.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

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: $N = \lambda \times R = \frac{3}{10} \times \frac{12.5}{3} = \frac{12.5}{10}$