Chapter 7 - Process Synchronization

- Problem: Processes share data, but is it kept consistent?
- Example: Producer-Consumer
- Interleaving of machine code in concurrent systems
- Race condition

Producer:

```
while (1) {
  while (counter == BUFFER_SIZE);
  buffer[in] := produce_item();
  in = (in+1) % BUFFER_SIZE;
  counter = counter+1;
}
```

Consumer:

```
while (1) {
  while (counter == 0);
  consume_item(buffer[out]);
  out = (out+1) % BUFFER_SIZE;
  counter = counter-1;
}
```


Chapter 7 - Critical Sections

- System with n processes: $\{P_0, P_1, ..., P_{n-1}\}$
- Characteristic of system: Only **one** process may be executing inside its critical section
- Mutual exclusion
- Three requirements must be satisfied:
  1. Mutual exclusion
  2. Progress
  3. Bounded waiting

Process i

```
do {
  while (turn != i);
  /* Critical Section */
  turn = j
  /* Remainder Section */
} while (1);
```

Process j

```
do
  while (turn != j);
  /* Critical Section */
  turn = i
  /* Remainder Section */
} while (1);
```

Chapter 7 - Synchronization Hardware

- Can we solve the critical section problem by simply disabling interrupts?
- What are the implications for multi-processor systems?
- What are the implications for time-sharing systems?
- *Test-and-Set* and *Swap* instructions execute atomically.


```
do {
  waiting[i] = true;
  key = true;
  while (waiting[i] && key)
    key = test_and_set(lock);
  waiting[i] = false;
  /* Critical Section */
  j := (j+1) % n;
  while ((j != i) && !waiting[j])
    j = (j+1) % n;
  if (j == i) lock = false
  else waiting[j] = false;
  /* Remainder Section */
} while (1);
```


Chapter 7 - Semaphores

- Definition
- Atomic operations: *Signal* (V) and *Wait* (P)
- Single, shared semaphore can guarantee mutual exclusion between n processes
- Enforce specific execution sequences
- Implementation: Spinlocks, context switching and queues
- Deadlock and starvation

Chapter 7 - Example of Semaphores

Each process adheres to the following structure:

```
do {
  wait(mutex);
  /* Critical Section */
  signal(mutex);
  /* Remainder Section */
} while (1);
```

Suppose there are n processes: $\{P_0, P_1, ..., P_{n-1}\}$ and a semaphore, `mutex`. The following situation is possible:

1. P_0 executes `wait(mutex)`, setting `mutex = 0`
2. P_0 is interrupted. $P_1..P_{n-1}$ must wait since `mutex = 0`
3. P_0 enters its critical section while the other processes are still waiting
4. P_0 executes `signal(mutex)`, setting `mutex = 1`
5. Another process may now enter its critical section

Chapter 7 - Case Studies

- Readers-Writers
- Dining Philosophers


```
semaphore mutex, wrt;
int readcount;
```

Writer:

```
wait(wrt);
/* Write something */
signal(wrt);
```

Reader:

```
wait(mutex);
readcount = readcount+1;
if (readcount == 1) wait(wrt);
signal(mutex);
/* Read something */
wait(mutex);
readcount = readcount-1;
if (readcount == 0) signal(wrt);
signal(mutex);
```


Chapter 7 - Critical Regions

- Incorrect use of semaphores will result in program errors
- Critical regions are language constructs and is an attempt to reduce errors
- Shared variables are explicitly declared and may only be accessed inside a critical region, for example

```
  T shared v;
...
region v when B {
  S
}
```

- What impact does this have on compilers?

- Language construct that encapsulates procedures and data to manage shared resources (Dijkstra, 1971)

- Procedures and variables only accessible inside the monitor construct

- Synchronization is based on condition variables

- Signal and Wait operations are redefined for condition variables