Process Synchronization

- Concurrent programs are executed by multiple cooperating processes that share some data
- Concurrent access to shared data may result in data inconsistency
- OS must provide mechanisms to synchronize and coordinate cooperating processes

Producer X Consumer

```
Producer:

shared int counter;
shared char buf[N];

int main()
{
  const int n = N;
  int in = 0;

  while (1) {
    while (counter == n);
    buf[in] = produce();
    in = ++in % n;
    counter++;
  }
}
```

```
Consumer:

shared int counter;
shared char buf[N];

int main()
{
  const int n = N;
  int out = 0;

  while (1) {
    while (counter == 0);
    consume (buf[out]);
    out = ++out % n;
    counter--;
  }
}
```

Race Conditions

```
Producer:                          Consumer:
  counter++;                       counter--
  load  R1,[counter]               load  R2,[counter]
  inc   R1                         dec   R2
  store R1,[counter]               store R2,[counter]


                                         R1 R2 [counter]
0) P: load  R1,[counter]                 5  -  5
1) P: inc   R1                           6  -  5
2) C: load  R2,[counter]                 6  5  5
3) C: dec   R2                           6  4  5
4) C: store R2,[counter]                 6  4  4
5) P: store R1,[counter]                 6  4  6
```

Critical Sections

- Sections of concurrent programs in which shared data is manipulated
- Conditions for proper execution
  - Mutual Exclusion: only a single process executes a critical section on a time
  - Execution progress: a process that is not executing a critical section cannot prevent others from doing it
  - Bounded waiting: a process cannot be deprived from execution a critical section indefinitely

Operating Systems

Synchronization Algorithm I

Process 0

```
shared int turn;

int main()
{
  while (1) {
    while(turn != 0);

    /* critical */

    turn = 1;

    /* remainder */
  }
}
```

Process 1

```
shared int turn;

int main()
{
  while (1) {
    while(turn != 1);

    /* critical */

    turn = 0;

    /* remainder */
  }
}
```

- Misses the progress condition

Operating Systems

Synchronization Algorithm II

Process 0

```
shared int flag[2];

int main()
{
  while (1) {
    flag[0] = 1;
    while(flag[1]);

    /* critical */

    flag[0] = 0;

    /* remainder */
  }
}
```

Process 1

```
shared int flag[2];

int main()
{
  while (1) {
    flag[1] = 1;
    while(flag[0]);

    /* critical */

    flag[1] = 0;

    /* remainder */
  }
}
```

- Misses the bounded waiting condition

LISHA

Operating Systems

Synchronization Algorithm III (Peterson)

Process 0

```
shared int turn;
shared int flag[2];

int main()
{
  while (1) {
    flag[0] = 1;
    turn = 1;
    while(flag[1] &&
                turn);
    /* critical */

    flag[0] = 0;
    /* remainder */
  }
}
```

Process 1

```
shared int turn;
shared int flag[2];

int main()
{
  while (1) {
    flag[1] = 1;
    turn = 0;
    while(flag[0] &&
               !turn);
    /* critical */

    flag[1] = 0;
    /* remainder */
  }
}
```

Operating Systems

Synchronization Hardware

- Test and Set Lock (TSL) instruction

```
int tsl(int * ptr)
{
  int tmp = *ptr;
  *ptr = 1;
  return tmp;
}
```

- Usage

```
shared int lock = 0;
int main()
{
   while (1) {
       while(tsl(lock));
       /* critical */
       lock = 0;
   }
 }
```

Semaphores

- Integer variable accessible through atomic operations P and V

```
p(s): while(s <= 0);          v(s): s++;
      s--;
```

- Usage

```
shared int mutex;
int main()
{
    while(1) {
        p(mutex);
        /* critical */
        v(mutex);
        /* remainder */
    }
}
```

Operating Systems

Semaphore Implementation

```
class Semaphore
{
public:
  Semaphore(int i) : s(i) {}
  void p();
  void v();
private:
  int s;
  list<Process> l;
};
extern Process * running;
```

```
void Semaphore::v()
{
  if(++s <= 0)
    l.pop()->wakeup();
}
```

```
void Semaphore::p()
{
  if (--s < 0) {
    l.push(running);
    running->sleep();
  }
}
```