Text - code
Data - global
Heap - dynamic
Stack - local

Zombie - parent didn't yet call wait
Orphan - parent never called wait

ms=1000us

IPC: Pipe, FIFO, Shared Memory, Socket, Message Queue
Thread: -addressSpace +pcRegistersStack -osResources

Peterson's Solution

Thread 1	Thread 2

```
do {
   flag[0] = TRUE;
   turn = 1;
   while (flag[1]  &&  turn==1)
   {};
   // critical section

   flag[0] = FALSE;

   // remainder section
} while (TRUE)
```

```
do {
   flag[1] = TRUE;
   turn = 0;
   while (flag[0]  &&  turn==0)
   {};
   // critical section

   flag[1] = FALSE;

   // remainder section
} while (TRUE)
```

Disabling interrupts in a single-core CPU facilitates locks

```
boolean TestAndSet (boolean *target)
{
      boolean rv = *target;
      *target = TRUE;
      return rv:
}
```

```
int mutex;
init_lock (&mutex);

do {

   lock (&mutex);
      critical section
   unlock (&mutex);

      remainder section
} while(TRUE);
```

```
void init_lock (int *mutex)
{
   *mutex = 0;
}

void lock (int *mutex)
{
   while(TestAndSet(mutex))
      ;
}

void unlock (int *mutex)
{
   *mutex = 0;
}
```

```
int CAS(int *value, int oldval, int newval)
{
      int temp = *value;
      if (*value == oldval)
        *value = newval;
      return temp;
 }
```

```
void lock (int *mutex) {
   while(CAS(mutex, 0, 1) != 0);
}
```

More reading: [mutex.c in .](#)

```
void mutex_init (mutex_t *lock)
{
   lock->value = 0;
   list_init(&lock->wait_list);
   spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
   spin_lock(&lock->wait_lock);
   while(TestAndSet(&lock->value)) {
      current->state = WAITING;
      list_add(&lock->wait_list, current);
      spin_unlock(&lock->wait_lock);
      schedule();
      spin_lock(&lock->wait_lock);
   }
  spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
   spin_lock(&lock->wait_lock);
   lock->value = 0;
   if (!list_empty(&lock->wait_list))
      wake_up_process(&lock->wait_list)
   spin_unlock(&lock->wait_lock);
}
```

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

Someone is waiting for the lock

Wake-up a waiting thread

P - lock. V - signal unlock

```
P(semaphore *S) {
      S->lock->Acquire();
      S–>value—;
      if(S–>value < 0) {
            addQ(&S->list, P);
            S->lock->Release();
            schedule( );
            S->lock->Acquire();
      }
      S->lock->Release();
}
```

```
V(semaphore *S) {
      S->lock->Acquire();
      S–>value++ ;
      if(S->value <= 0) {
            P = delQ(&S->list);
            wakeup(P);
      }
      S->lock->Release();
}
```

Monitor version

```
Mutex lock;
Condition full, empty;

produce (item)
{
    lock.acquire();
    while (queue.isFull())
        empty.wait(&lock);
    queue.enqueue(item);
    full.signal();
    lock.release();
}

consume()
{
    lock.acquire();
    while (queue.isEmpty())
        full.wait(&lock);
    item = queue.dequeue(item);
    empty.signal();
    lock.release();
    return item;
}
```

Semaphore version

```
Semaphore mutex = 1, full = 0,
empty = N;

produce (item)
{
    P(&empty);
    P(&mutex);
    queue.enqueue(item);
    V(&mutex);
    V(&full);
}

consume()
{
    P(&full);
    P(&mutex);
    item = queue.dequeue();
    V(&mutex);
    V(&empty);
    return item;
}
```

Reader-Writer Problem

```
semaphore mutex = 1, wrt = 1;
int readcount = 0;
```

Writer

```
do {
    P(wrt);
    write object resource
    V(wrt);
} while (TRUE);
```

Reader

```
do {
    P(mutex);
    readcount++;
    if (readcount == 1)
        P(wrt);
    V(mutex)
    read from object resource
    P(mutex);
    readcount--;
    if (readcount == 0)
        V(wrt);
    V(mutex)
} while (TRUE);
```

Deadlock conditions:

Mutual exclusion - one process per resource

No preemption - resource release must be voluntary
Hold and wait - holding some resources and waiting for more
Circular wait - circular dependency

Starvation can end, but deadlock can't

Banker's Algorithm

- General idea
  - Assume that each process's maximum resource demand is known in advance
    - Max[i] : process i's maximum resource demand vector
  - **Pretend** each request is granted, then run the deadlock detection algorithm
  - If a deadlock is detected, the do not grant the request to keep the system in a **safe** state

Banker's Algorithm

1. Initialize **Avail** and **Finish** vectors

   Avail = FreeResources;

   For i = 1,2, …, n, Finish[i] = false

2. Find an index i such that
   Finish[i] == false AND
   Max[i] – Alloc[i] ≤ Avail

   If no such i exists, go to step 4

3. Avail = Avail + Alloc[i] , Finish[i] = true
   Go to step 2

4. If Finish[i] == false, for some i, 1 ≤ i ≤ n,

   (a) then the system is in deadlock state

   (b) if Finish[i] == false, then P_i is deadlocked

- FreeResources: resource vecto
  [R1, R2] = [0,0]
- Alloc[i]*:* process i's allocated resource vector:
  Alloc[1] = [0,1], Alloc[2] = [1, 0]
- Request[i]: process i's requesting vector:
  Request[1] = [1,0]
  Request[2] = [0,0]
- **Max[i]:** process i's maximum resource demand vector

Response time - time to complete a task

Wait time - total waiting
Scheduling latency - time till first run
FCFS - FIFO (min overhead)
SJF - min burst len first
SRTF - remaining time - new shorter interrupts longer (smallest waiting and interactive)
RR - round robin - each job executed for a quantum time, then rotate (interactive)
CFS - completely fair scheduler - pick min weightCpu%*executedTime