MARcin COPiK <MARCIN.COPIK@INF.ETHZ.CH>

DPHPC: Locks, SPIN Tutorial

Recitation session
New TA

Name: Marcin Copik
Mail: marcin.copik@inf.ethz.ch

Research areas:
• performance modelling (Extra-P)
• large-scale code analysis
• serverless architectures (AWS Lambda, Azure Functions)
New TA

Name: Marcin Copik
Mail: marcin.copik@inf.ethz.ch

Research areas:
• performance modelling (Extra-P)
• large-scale code analysis
• serverless architectures (AWS Lambda, Azure Functions)
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

write(flag[0]=true)

T0

T1
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

T0
```
write(flag[0]=true)
```

T1
```
write(flag[1]=true)
```
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

**Diagram:**

- **T0:**
  - write(flag[0]=true)
  - read(flag[1]) == true

- **T1:**
  - write(flag[1]=true)
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

- T0: write(flag[0]=true)
- T1: write(flag[1]=true)
- T0: read(flag[1])==true
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

Diagram:
- T0 writes `flag[0] = true`
- T1 writes `flag[1] = true`
- T0 tries to read `flag[1]` but T1 has not written it yet
- T1 reads `flag[0] = true`
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

T0
write(flag[0]=true)
read(flag[1])==true
write(flag[1]=true)
read(flag[0])==true

T1
write(flag[1]=true)
read(flag[0])==true
2-threads: LockOne

```c
volatile int flag[2];

void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}

void unlock() {
    flag[tid] = false;
}
```

Deadlock – lock acquisition and entering CS are not atomic.
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```

write(victim=0)
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```

write(victim=0)
read(victim==0)
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```

- **T0**: write(victim=0), read(victim==0)
- **T1**: write(victim=1), read(victim==1)
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```
2-threads: LockTwo

```c
volatile int victim;

void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}

void unlock() {}
```

Q: What might be wrong with this lock?
2-threads: Peterson lock

```c
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1;  // I'm interested
    victim = tid;  // other goes first
    while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0;  // I'm not interested
}
```
Locking
Locking

- Safe lock release
  - RAII in C++
  - try/finally clauses in Java/C#

```java
public class BankingSystem {
    ...
    public boolean transferMoney(Account from, Account to, int amount) {
        ...
        first.getLock().lock();
        second.getLock().lock();
        try {
            ...
        } finally {
            first.getLock().unlock();
            second.getLock().unlock();
        }
    }
}
```
Locking

• Safe lock release
  • RAII in C++
  • try/finally clauses in Java/C#

• Two phase locking
  • Multiple resources are locked? Lock all of them before performing any computations.
Locking

- **Safe lock release**
  - RAII in C++
  - try/finally clauses in Java/C#

- **Two phase locking**
  - Multiple resources are locked? Lock all

- **Ordered locking**
  - When locking multiple resources, define a total order

```java
public class BankingSystem {
    ...
    public boolean transferMoney(Account from, Account to, int amount) {
        Account first, second;
        // Introduce lock ordering:
        if (to.getId() > from.getId()) {
            first = from; second = to;
        } else {
            first = to; second = from;
        }
        ...
    }
}
```
Locking

• **Safe lock release**
  • RAII in C++
  • try/finally clauses in Java/C#

• **Two phase locking**
  • Multiple resources are locked? Lock all of them before performing any computations.

• **Ordered locking**
  • When locking multiple resources, define a total order

• **Efficient synchronization with atomics**
  • Entering critical section quite often? Locks causing too much overhead?
Locks
Locks

```c
void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid;    // other goes first
    while (flag[j] && victim == tid) {};
}
```
Locks

void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid;      // other goes first
    while (flag[j] && victim == tid) {}
}

volatile int flag[n] = {0,0,...,0};
volatile int label[n] = {0,0,...,0};

void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
    while ((∃k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {}
}

public void unlock() {
    flag[tid] = 0;
}

Lamport’s Bakery
void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid;      // other goes first
    while (flag[j] && victim == tid) {};
}

void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ..., label[n-1]) + 1; // take ticket
    while (\( \exists k \neq tid \) (flag[k] && (label[k], k) <* (label[tid], tid))) {};
}

public void unlock() {
    flag[tid] = 0;
}

Lamport's Bakery

bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = true;
    return old;
}  // all atomic!

volatile int flag[n] = {0,0,...,0};
volatile int label[n] = {0,0,...,0};
Locks

```java
template

void lock() {
  int j = 1 - tid;
  flag[tid] = true; // I'm interested
  victim = tid;      // other goes first
  while (flag[j] && victim == tid) {};
}

void lock() {
  flag[tid] = 1; // request
  label[tid] = max(label[0], ..., label[n-1]) + 1; // take ticket
  while (\(\exists k \neq tid \) (flag[k] && (label[k],k) <* (label[tid],tid))) {};
}

public void unlock() {
  flag[tid] = 0;
}

TATAS

volatile int lock = 0;

void lock() {
  do {
    while (lock == 1);
  } while (TestAndSet(&lock) == 1);
}

void unlock() {
  lock = 0;
}
```

bool test_and_set (bool *flag) {
  bool old = *flag;
  *flag = true;
  return old;
} // all atomic!
Locks

```java
void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid; // other goes first
    while (flag[j] && victim == tid) {};
}

volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,….,0};
void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
    while (\(\exists k \neq tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}

public void unlock() {
    flag[tid] = 0;
}
```

TATAS

```java
volatile int lock = 0;
void lock() {
    volatile int lock = 0;
    while (TestAndSet(&lock) == 1) {
        wait(time);
        time *= 2; // double waiting time
    }
}
void unlock() {
    lock = 0;
}
```
Locks

volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;

void lock() {
    index[tid] = GetAndInc(tail) % n;
    while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
    array[index[tid]] = 0; // I release my lock
    array[(index[tid] + 1) % n] = 1; // next one
}
Locks

typedef struct qnode {
    struct qnode *prev;
    int succ_blocked;
} qnode;

volatile qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    qn->prev = FetchAndSet(lck, qn);
    while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
    qnode *pred = (*qn)->prev;
    (*qn)->succ_blocked = 0;
    *qn = pred;
}

CLH Lock
Locks

void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid;      // other goes first
    while (flag[j] && victim == tid) {
    }
}

volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,….,0};

void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
    while ( (∃ k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {
    }
}

public void unlock() {
    flag[tid] = 0;
}

Lamport's Bakery

bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = true;
    return old;
} // all atomic!

volatile int lock = 0;

void lock() {
    do {
        while (lock == 1);
    } while (TestAndSet(&lock) == 1);
}

void unlock() {
    lock = 0;
}

TATAS + expo backoff

volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;

void lock() {
    index[tid] = GetAndInc(tail) % n;
    while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
    array[index[tid]] = 0; // I release my lock
    array[(index[tid] + 1) % n] = 1; // next one
}

Array Queue Lock

typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    qn->prev = FetchAndSet(lck, qn);
    while (qn->prev->succ_blocked); // the lock
}

void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we’re the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}

MCS Lock

typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;

qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
    qn->next = NULL;
    qnode *pred = FetchAndSet(lck, qn);
    if(pred != NULL) {
        qn->locked = 1;
        pred->next = qn;
        while(qn->locked);
    }
}

void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we’re the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}
Locks

```c
void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid;      // other goes first
    while (flag[j] && victim == tid) {};
}

volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,…,0};
void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ..., label[n-1]) + 1; // take ticket
    while ((∃ k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}

public void unlock() {
    flag[tid] = 0;
}
```

Lamport's Bakery

```c
bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = true;
    return old;
} // all atomic!
```

TATAS

```c
volatile int lock = 0;
void lock() {
    while (TestAndSet(&lock) == 1) {
        wait(time);
        time *= 2; // double waiting time
    }
}
void unlock() {
    lock = 0;
}
```

TATAS + exp backoff

```c
volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;
void lock() {
    index[tid] = GetAndInc(tail) % n;
    while (!array[index[tid]]); // wait to receive lock
}
void unlock() {
    array[index[tid]] = 0; // I release my lock
    array[(index[tid] + 1) % n] = 1; // next one
}
```

Array Queue Lock

```c
typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;
qnode *lck = new qnode; // node owned by lock
void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    qn->prev = FetchAndSet(lck, qn);
    while (qn->prev->succ_blocked);
}
void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we're the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}
```

CLH Lock

```c
typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;
qnode *lck = NULL;
void lock(qnode *lck, qnode *qn) {
    qn->next = NULL;
    qnode *pred = FetchAndSet(lck, qn);
    if(pred != NULL) {
        qn->locked = 1;
        pred->next = qn;
        while(qn->locked);
    }
}
void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we're the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}
```

CLH Lock with exp backoff

```c
typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;
qnode *lck = NULL;
void lock(qnode *lck, qnode *qn) {
    qn->next = NULL;
    qnode *pred = FetchAndSet(lck, qn);
    if(pred != NULL) {
        qn->locked = 1;
        pred->next = qn;
        while(qn->locked);
    }
}
void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we're the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}
```
Locks

void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid;      // other goes first
    while (flag[j] && victim == tid) {};
}

volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,….,0};
void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
    while ( (∃ k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}

public void unlock() {
    flag[tid] = 0;
}

Lamport's Bakery

bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = true;
    return old;
} // all atomic!

volatile int lock = 0;
void lock() {
    do {
        while (lock == 1);
    } while (TestAndSet(&lock) == 1);
}

void unlock() {
    lock = 0;
}

TATAS

volatile int lock = 0;
void lock() {
    while (TestAndSet(&lock) == 1) {
        wait(time);
        time *= 2; // double waiting time
    }
}

void unlock() {
    lock = 0;
}

TATAS + exp backoff

volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;
void lock() {
    index[tid] = GetAndInc(tail) % n;
    while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
    array[index[tid]] = 0; // I release my lock
    array[(index[tid] + 1) % n] = 1; // next one
}

Array Queue Lock

typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;
qnode *lck = new qnode; // node owned by lock
void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    qn->prev = FetchAndSet(lck, qn);
    while(qn->prev->succ_blocked);
}

void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we’re the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}

CLH Lock

typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;
qnode *lck = NULL;
void lock(qnode *lck, qnode *qn) {
    qn->next = NULL;
    qnode *pred = FetchAndSet(lck, qn);
    if(pred != NULL) {
        qn->locked = 1;
        pred->next = qn;
        while(qn->locked);
    }
}

void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we’re the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}

MCS Lock

Complicated

Difficult to optimize
Locks

```c
typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;

goide *lck = NULL;

void lock(qnode *lck, qnode *qn) {
    qn->locked = 1;
    pred->next = qn;
    while(qn->locked);
}

void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we’re the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}
```

Complicated

Over-optimization can quickly lead to incorrect locks

Difficult to optimize
Locks

void lock() {
  int j = 1 - tid;
  flag[tid] = true; // I'm interested
  victim = tid;      // other goes first
  while (flag[j] && victim == tid) {};
}

volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,….,0};

void lock() {
  flag[tid] = 1; // request
  label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
  while (exists(k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}

public void unlock() {
  flag[tid] = 0;
}

Lamport's Bakery

// all atomic!

volatile int lock = 0;

void lock() {
  do {
    while (lock == 1);
  } while (TestAndSet(&lock) == 1);
}

void unlock() {
  lock = 0;
}

TATAS

volatile int lock = 0;

void lock() {
  index[tid] = GetAndInc(tail) % n;
  while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
  array[index[tid]] = 0; // I release my lock
  array[(index[tid] + 1) % n] = 1; // next one
}

Array Queue Lock

typedef struct qnode {
  struct qnode *next;
  int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
  qn->succ_blocked = 1;
  qn->prev = FetchAndSet(lck, qn);
  while (qn->prev->succ_blocked);
}

void unlock(qnode *lck, qnode *qn) {
  if (qn->next == NULL) { // if we're the last waiter
    if (CAS(lck, qn, NULL)) return;
    while (qn->next == NULL); // wait for pred arrival
  }
  qn->next->locked = 0; // free next waiter
  qn->next = NULL;
}

CLH Lock

typedef struct qnode {
  struct qnode *next;
  int succ_blocked;
} qnode;

qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
  qn->next = NULL;
  qnode *pred = FetchAndSet(lck, qn);
  if (pred != NULL) {
    qn->locked = 1;
    pred->next = qn;
    while (qn->locked);
  }
}

void unlock(qnode *lck, qnode *qn) {
  if (qn->next == NULL) { // if we're the last waiter
    if (CAS(lck, qn, NULL)) return;
    while (qn->next == NULL); // wait for pred arrival
  }
  qn->next->locked = 0; // free next waiter
  qn->next = NULL;
}

MCS Lock

typedef struct qnode {
  struct qnode *prev;
  int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
  qn->prev->succ_blocked = 1;
  qn->prev = FetchAndSet(lck, qn);
  while (qn->prev->succ_blocked);
}

void unlock(qnode *lck, qnode *qn) {
  qnode *pred = (*qn)->prev;
  (*qn)->succ_blocked = 0;
  *qn = pred;
}

How can make sure locks (or other distributed algorithms) are correct in practice?

Complicated

Over-optimization can quickly lead to incorrect locks

Difficult to optimize

Over-optimization can quickly lead to incorrect locks
Locks

void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid;      // other goes first
    while (flag[j] && victim == tid) {};
}

volatile int flag[n] = {0,0,…,0};
volatile int label[n] = {0,0,….,0};
void lock() {
    flag[tid] = 1; // request
    label[tid] = max(label[0], ...,label[n-1]) + 1; // take ticket
    while ((∃ k != tid)(flag[k] && (label[k],k) <* (label[tid],tid))) {};
}

public void unlock() {
    flag[tid] = 0;
}

Lamport's Bakery

volatile int lock = 0;
void lock() {
    do {
        while (lock == 1);
    } while (TestAndSet(&lock) == 1);
}

void unlock() {
    lock = 0;
}

TATAS

volatile int lock = 0;
void lock() {
    while (TestAndSet(&lock) == 1) {
        wait(time);
        time *= 2; // double waiting time
    }
}

void unlock() {
    lock = 0;
}

TATAS + exp backoff

volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;
void lock() {
    index[tid] = GetAndInc(tail) % n;
    while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
    array[index[tid]] = 0; // I release my lock
    array[(index[tid] + 1) % n] = 1; // next one
}

Array Queue Lock

typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    qn->prev = FetchAndSet(lck, qn);
    while (qn->prev->succ_blocked);
}

void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we're the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}

CLH Lock

typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
} qnode;

qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
    qn->next = NULL;
    qnode *pred = FetchAndSet(lck, qn);
    if(pred != NULL) {
        qn->locked = 1;
        pred->next = qn;
        while(qn->locked);
    }
}

void unlock(qnode *lck, qnode *qn) {
    if(qn->next == NULL) { // if we're the last waiter
        if(CAS(lck, qn, NULL)) return;
        while(qn->next == NULL); // wait for pred arrival
    }
    qn->next->locked = 0; // free next waiter
    qn->next = NULL;
}

MCS Lock

typedef struct qnode {
    struct qnode *prev;
    int succBlocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->succBlocked = 1;
    qn->prev = FetchAndSet(lck, qn);
    while (qn->prev->succBlocked);
}

void unlock(qnode *lck, qnode *qn) {
    qnode *pred = (*qn)->prev;
    (*qn)->succBlocked = 0;
    *qn = pred;
}

Complicated

Over-optimization can quickly lead to incorrect locks

Difficult to optimize

How can make sure locks (or other distributed algorithms) are correct in practice?
How to check correctness?

- Common design flaws in designing distributed systems:
  - Deadlock
  - Livelock, starvation
  - Underspecification
    - Unexpected messages
  - Overspecification
    - Dead code
  - Constraint violations
    - Buffer overruns
    - Array bound violations
How to check correctness?

• Common design flaws in designing distributed systems:
  ▪ Deadlock
  ▪ Livelock, starvation
  ▪ Underspecification
    *Unexpected messages*
  ▪ Overspecification
    *Dead code*
  ▪ Constraint violations
    *Buffer overruns*
    *Array bound violations*
Model checking

- Model checking verifies a program by using software to analyze its state space

- Alternative: mathematical deductive methods
  - Constructing a proof requires mathematical insights and tenacity
  - The complexity depends on the algorithms itself

- Deductive proofs are more elegant and powerful, but model checking can be more practical

```plaintext
integer n=0

process P
  integer regP=0
  p1: load n into regP
  p2: increment regP
  p3: store regP into n
  p4: end

process Q
  integer regQ=0
  p1: load n into regQ
  p2: increment regQ
  p3: store regQ into n
  p4: end
```

Model checking

- Model checking verifies a program by using software to analyze its state space

- Alternative: mathematical deductive methods
  - Constructing a proof requires mathematical insights and tenacity
  - The complexity depends on the algorithms itself

- Deductive proofs are more elegant and powerful, but model checking can be more practical

Model checking

- Model checking verifies a program by using software to analyze its state space

- Alternative: mathematical deductive methods
  - Constructing a proof requires mathematical insights and tenacity
  - The complexity depends on the algorithms itself

- Deductive proofs are more elegant and powerful

### How many states?

- State: IP + 2 register (regP, regQ) + global var n
- IP: 4 value (1…4) – one per process
- Registers + n: 3 values (0, 1, 2)
- \(4 \times 4 \times 3 \times 3 \times 3 = 432\)

Space Explosion Problem

- The system is described as a State Transition Graph
- We have a combinatorial explosion of systems states

- How to handle it?
Space Explosion Problem

• The system is described as a State Transition Graph
• We have a combinatorial explosion of systems states

• How to handle it?

• Increase the abstraction
Space Explosion Problem

• The system is described as a State Transition Graph
• We have a combinatorial explosion of systems states

• How to handle it?
• Increase the abstraction
• Compressed representations
Space Explosion Problem

• The system is described as a State Transition Graph
• We have a combinatorial explosion of systems states

• How to handle it?

• Increase the abstraction

• Compressed representations

• Not all the states are actually reachable
  ▪ Generate the states on the fly: only the ones that can be reached are generated
Space Explosion Problem

- The system is described as a State Transition Graph
- We have a combinatorial explosion of systems states

- How to handle it?

- Increase the abstraction

- Compressed representations

- Not all the states are actually reachable
  - Generate the states on the fly: only the ones that can be reached are generated

- Avoid to visit the same state multiple times
  - E.g., keep a hash table to index the visited states
Space Explosion Problem

10^{500,000} \text{ states} \quad 10^{70} \text{ atoms}

Source: *Probabilistic Model Checking* lectures, Dave Parker.
SPIN – Introduction

• **SPIN: Simple Promela Interpreter**
  ▪ Goal: analyze the logical consistency of concurrent systems
  ▪ Concurrent systems are described in the modelling language called **Promela**

• **Promela: Protocol/Process Meta Language**
  ▪ Allows dynamic creation of concurrent processes
  ▪ Communication via message passing can be
    - **Synchronous (aka rendezvous)**
    - **Asynchronous (aka buffered)**
  ▪ C-like language
  ▪ Enables you to model a finite-state system

• **Warning:** If that description is “too far off” from our code, we risk specifying the wrong state machine!
Promela Model

Promela model consists of:
- Type declarations
- Channel declarations
- Variable declarations
- Process declarations
- [init process]

A process type (proctype) consists of:
- a name
- a list of formal parameters
- local variable declarations
- body
Promela - Processes

- Identified by the **proctype** keyword
- Can be executed concurrently
- You can create multiple processes of the same type

- Each process has its own local state defined by:
  - Program counter
  - Local variables

- Communication between processes:
  - Shared variables
  - Channels

- Processes can be created using the *run* keyword
  - It returns the *pid* of the created process
  - Can be called at any point

```plaintext
proctype Sender(chan in; chan out) {
  bit sndB, rcvB;
  do
    :: out ! MSG, sndB ->
    in ? ACK, rcvB;
    if
      :: sndB == rcvB -> sndB = l-sndB
      :: else -> skip
    fi
  od
}
```

The body consist of a sequence of statements.
Promela: Hello World

```promela
active proctype Hello() {
    printf("Hello process, my pid is: %d\n", _pid);
}

init{
    int lastpid;
    printf("init process, my pid is: %d\n", _pid);
    lastpid = run Hello();
    printf("last pid was: %d\n", lastpid);
}
```
Promela: Hello World

```promela
active proctype Hello() {
    printf("Hello process, my pid is: %d\n", _pid);
}

init{
    int lastpid;
    printf("init process, my pid is: %d\n", _pid);
    lastpid = run Hello();
    printf("last pid was: %d\n", lastpid);
}
```

```
$ spin -n2 hello.pr
init process, my pid is: 1
    last pid was: 2
Hello process, my pid is: 0
Hello process, my pid is: 2
3 processes created
```

random seed

running SPIN in random simulation mode
Promela: Variables and Types

- **Types:** 5 basic types
  - bit, bool, byte, short, int
- **Arrays**
  - byte a[27];
- **Records (structs)**
  - typedef Record{
    short f1;
    byte f2;
  }
  Record rr;
  rr.f1 = ...
- **Global and local variables are initialized to 0 by default**

```c
int ii;
bit bb;
bb=1;
ii=2;
short s=-1;
typedef Foo {
  bit bb;
  int ii;
};
Foo f;
f.bb = 0;
f.ii = -2;
ii*s+27 == 23;
printf("value: %d", s*s);
```
Promela: Statements

• The body of a process consists of a sequence of statements.
  ▪ Executable: the statement can be executed immediately
  ▪ Blocked: it cannot be executed

• Assignments are always executable

• An expression is executable only if it evaluates to non-zero
  ▪ $2 < 3$ always executable
  ▪ $x < 27$ executable only if $x < 27$
  ▪ $3 + x$ executable only if $x \neq -3$

• The assert(<expr>) statement is always executable
  ▪ SPIN exits with an error if an assert evaluates to 0
  ▪ Used to check if properties hold
Semantic

• Pamela processes are executed **concurrently** and scheduled in a **non-deterministic** fashion

• Execution of statements of different processes is interleaved
  ▪ All statements are atomic

• Each process may have multiple actions ready to be executed
  ▪ Only one is non-deterministically chosen to be executed
Promela: Mutual Exclusion? (1)

```promela
bit flag;         /* signal entering/leaving the section */
byte mutex;       /* # procs in the critical section */

proctype P(int i) {
    flag != 1;
    flag = 1;
    mutex++;
    printf("MSC: P(%d) has entered section.\n", i);
    mutex--;
    flag = 0;
}

proctype monitor() {
    assert(mutex != 2);
}

init { run P(0); run P(1); run monitor(); }
```

Promela: Mutual Exclusion? (1)

```promela
bit flag;     /* signal entering/leaving the section */
byte mutex;   /* # procs in the critical section. */

proctype P(int i) {
    flag != 1;
    flag = 1;
    mutex++;
    printf("MSC: P(%d) has entered section.\n", i);
    mutex--;
    flag = 0;
}

proctype monitor() {
    assert(mutex != 2);
}

init { run P(0); run P(1); run monitor(); }
```

Both processes can pass the flag!=1 "at the same time"
Promela: Mutual Exclusion? (1)

bit flag;         /* signal entering/leaving the section */
byte mutex;       /* # procs in the critical section. */

proctype P(int i) {
    flag != 1;
    flag = 1;
    mutex++;
    printf("MSC: P(%d) has entered section.\n", i);
    mutex--;
    flag = 0;
}

proctype monitor() {
    assert(mutex != 2);
}

init { run P(0); run P(1); run monitor(); }

Both processes can pass the flag!=1 “at the same time”
Promela: Mutual Exclusion? (2)

```promela
bit x, y;  /* signal entering/leaving the section */
byte mutex;  /* # of procs in the critical section. */

active proctype A() {
    x = 1;
    y == 0;
    mutex++; mutex--;
    x = 0;
}

active proctype B() {
    y = 1;
    x == 0;
    mutex++; mutex--;
    y = 0;
}

active proctype monitor() {
    assert(mutex != 2);
}
```

bit x, y;          /* signal entering/leaving the section */
byte mutex;       /* # of procs in the critical section. */

active proctype A() {
    x = 1;
    y == 0;
    mutex++; mutex--;  
x = 0;
}

active proctype B() {
    y = 1;
    x == 0;
    mutex++; mutex--;  
y = 0;
}

active proctype monitor() {
    assert(mutex != 2);
}
Promela: Mutual Exclusion? (2)

bit x, y;         /* signal entering/leaving the section */
byte mutex;  /* # of procs in the critical section. */

active proctype A() {
    x = 1;
    y == 0;
    mutex++; mutex--;
    x = 0;
}

active proctype B() {
    y = 1;
    x == 0;
    mutex++; mutex--;
    y = 0;
}

active proctype monitor() {
    assert(mutex != 2);
}

Both processes can pass execute x=1 and y=1 “at the same time”...
PROMELA Semantics: if

if
:: choice_1 -> stat1.1; stat1.2; stat1.3; …
:: choice_2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choice_n -> statn.1; statn.2; statn.3; …
:: else -> skip
fi;

• If there is at least one choice (guard) executable, the if statement is executable and SPIN non-deterministically chooses one of the executable choices.
• The “else” choice is executable iff no other choices are
• If no choice is executable, the if-statement is blocked
With respect to the choices, a do-statement behaves in the same way as an if-statement.

However, instead of ending the statement at the end of the chosen list of statements, a do-statement repeats the choice selection.

The (always executable) break statement exits a do-loop statement and transfers control to the end of the loop.
PROMELA Semantics: do

- With respect to the choices, a do-statement behaves in the same way as an if-statement.
- However, instead of ending the statement at the end of the chosen list of statements, a do-statement repeats the choice selection.
- The (always executable) break statement exits a do-loop statement and transfers control to the end of the loop.
PROMELA Semantics: Communication

• Communication between processes is via typed channels

```
chan <name> = [<dim>] of {<t1>,<t2>,...,<tn>};
```

• A channel can be synchronous (dim=0) or asynchronous (dim>0)
  ▪ In the first case, synchronization is needed
  ▪ In the second, the channels act like a FIFO-buffer

Example:
```
mtype = {MSG, ACK};
chan toS = [2] of {mtype, bit};
```
PROMELA Semantics: Communication

- Communication between processes is via typed channels

```plaintext
chan <name> = [<dim>] of {<t_1>, <t_2>, ..., <t_n>};
```

- Sending:
  - ```
  ch ! <expr_1>, <expr_2>, ..., <expr_n>;
  ```
  The values of `<expr_i>` must match the types of the channel declaration
  A send statement is executable if the channel is not full

- Receiving
  - ```
  ch ? <var_1>, <var_2>, ..., <var_n>;
  ```
  If the channel is not empty, the message is fetched from the channel
  - ```
  ch ? <const_1>, <const_2>, ..., <const_n>;
  ```
  If the channel is not empty and the message at the front of the channel evaluates to the individual `<const_i>`, the statement is executable and the message is removed from the channel
• Communication between processes is via typed channels

```
chan <name> = [<dim>] of {<t1>,<t2>,...,<tn>};
```

• Sending:
  ▪ `ch ! <expr1>, <expr2>, ..., <exprn>;
    The values of `<expri>` must match the types of the channel declaration
    A send statement is executable if the channel is not full

• Receiving
  ▪ `ch ? <var1>, <var2>, ..., <varn>;
    If the channel is not empty, the message is fetched from the channel
  ▪ `ch ? <const1>, <const2>, ..., <constn>;
    If the channel is not empty and the message at the front of the channel evaluates to the individual `<consti>`, the statement is executable and the message is removed from the channel

Message Matching
PROMELA Semantics: Communication

• Communication between processes is via typed channels

\[
\text{chan } <\text{name}> = [<\text{dim}>]\text{ of } \{<t_1>,<t_2>,\ldots,<t_n>\};
\]

• Sending:
  ▪ \( \text{ch } ! <\text{expr}_1>, <\text{expr}_2>, \ldots, <\text{expr}_n>; \)
    - The values of \(<\text{expr}_i>\) must match the types of the channel declaration
    - A send statement is executable if the channel is not full

• Receiving
  ▪ \( \text{ch } ? <\text{var}_1>, <\text{var}_2>, \ldots, <\text{var}_n>; \)
    - If the channel is not empty, the message is fetched from the channel
  ▪ \( \text{ch } ? <\text{const}_1>, <\text{const}_2>, \ldots, <\text{const}_n>; \)
    - Message Matching
    - If the channel is not empty and the message at the front of the channel evaluates to the individual \(<\text{const}_i>\), the statement is executable and the message is removed from the channel

<vars> and <consts> can be mixed
PROMELA Semantics: Communication

- Communication between processes is via typed channels

```
chan <name> = [<dim>] of {<t1>,<t2>,...,<tn>};
```

- Sending:
  - `ch ! <expr1>, <expr2>, ..., <exprn>;`
    - The values of `<expri>` must match the types of the channel declaration
    - A send statement is executable if the channel is not full

- Receiving
  - `ch ? <var1>, <var2>, ..., <varn>;`
    - If the channel is not empty, the message is fetched from the channel
  - `ch ? <const1>, <const2>, ..., <constn>;`
    - If the channel is not empty and the message at the front of the channel evaluates to the individual `<consti>`, the statement is executable and the message is removed from the channel

Rendezvous communication (dim==0):
A send `ch!` is executable only if there is a corresponding receive `ch?` that can be executed simultaneously

Message Matching

`<vars>` and `<consts>` can be mixed
Using spin

- Random simulation mode: debugging/testing. Randomly resolves non-determinism
  `spin -n<SEED> model.pr #fix the seed to reproduce scenarios`

- Guided simulation mode (-i): non-determinism solved by the user

- Step by with printing all states:
  `spin -uN -p -l model`

- Verification mode: analyze all the reachable states
  `spin -a model.pr`
  `gcc -O2 -o pan pan.c`
  `./pan #generates trail file if things go wrong`
  `spin -t -p model.pr`
  
  *Generates a verifier in C code, so that compiler can optimize it, then exhaustively searches all possible states. It can still be slow/eat all your memory.*
Probabilistic Model Checking

Source: Probabilistic Model Checking lectures, Dave Parker.
Probabilistic Model Checking

Source: *Probabilistic Model Checking* lectures, Dave Parker.
Why we need probabilistic behaviour?

- Properties: **qualitative vs quantitative**
- Randomisation – useful to debug distributed algorithms/
- Uncertainty – a certain event might happen with small likelihood
Why we need probabilistic behaviour?

- Properties: qualitative vs quantitative
- Randomisation – useful to debug distributed algorithms/
- Uncertainty – a certain event might happen with small likelihood

“what’s the probability of delivering message in time $T$?”

“is it true that the probability of recovering from error within $T$ time steps is larger than 0.99?”

“what’s the average power consumption during execution?”
If you want to learn more…

• SPIN
  - spinroot.com

• PRISM – probabilistic model checker
  - https://www.prismmodelchecker.org/
  - Lectures on PMC: https://www.prismmodelchecker.org/lectures/pmc/
Assignment

Hippie problem:  
4 Hippies want to cross a bridge. The bridge is fragile, it can only be crossed by <= 2 people at a time with a torchlight. The hippies have one torchlight and want to reach the other side within one hour. Due to different degrees of intoxication they require different amounts of time to cross the bridge: 5, 10, 20 and 25 minutes. If a pair crosses the bridge, they can only move at the speed of the slower partner.