Synchronization II

To do ...

- Condition Variables
- Semaphores and monitors
- Some classical problems
- Next time: Deadlocks
Condition variables

- Many times a thread wants to check whether a condition is true before continuing execution
  - A parent waiting on a child, a consumer waiting on something to consume, …
  - But spinning on a shared variable is inefficient

- Condition variables
  - An explicit queue where threads can go when some state is not what they want (waiting on a change)
  - Until some other thread changes the state and informs them of it, signaling on the condition

```c
pthread_cond_t c;
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthreadcond_t *c);
```
Before we move on, did you notice?

```c
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
```

- Assumes mutex is locked before wait is called
  - Wait must release it and put the thread to sleep, atomically
  - When the thread wakes up, re-acquires the lock before returning
- All to prevent race condition when a thread is trying to put itself to sleep
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: done\n");
    return 0;
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return 0;
}

void thr_join() {
    pthread_mutex_lock(&m);
    while (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

void thr_exit() {
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}
void thr_join() {
    pthread_mutex_lock(&m);
    while (done == 0)
        pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m);
}

void thr_exit() {
    pthread_mutex_lock(&m);
    done = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

That while doesn’t seem strictly necessary, wouldn’t an if do ... wait a bit

Two cases to consider
- Parent creates the child and continue running
  - Gets the lock, check if done and put itself to sleep
  - Child runs, gets the lock, sets done and signals the parent
  - Parent returns from wait with lock held, unlocks it and is done
- If child runs first, sets done, signals (nobody is waiting) and returns; parent check child is done and returns
Non-working approaches

```c
void thr_exit() {
    pthread_mutex_lock(&m);
    /* done = 1; */
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}

void thr_join() {
    pthread_mutex_lock(&m);
    /* while (done == 0) */
    pthread_cond_wait(&c, &m);
    pthread_mutex_unlock(&m)
}

void thr_exit() {
    done = 1;
    pthread_cond_signal(&c);
}

void thr_join() {
    if (done == 0)
        pthread_cond_wait(&c);
}
```

**Do you need done?**

- If the child runs immediately, the signal will be lost
- Parent will call wait (there’s nothing to check) and go to sleep for ever

**Do you need that mutex?**

- What would happen if the parent is interrupted after checking ‘done’ but before going to sleep on wait?
- Child runs, signals nobody (parent is not there yet!) and ..
- When parent continues it goes to sleep, *for ever!*
Producer/consumer problem

- Producer-consumer problem, aka bounded buffer
  - Two or more processes & one shared, fixed-size buffer
  - Some put data times into a buffer, others takes them out
  - E.g., Web server with producers taken orders and consumer threads processing them

```c
int buff[MAX];
int fill = 0;
int use = 0;
int count = 0;

void put(int value) {
    buff[fill] = value;
    fill = (fill + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    count--;
    return tmp;
}
```
Producer/consumer problem

- “Simple solution”
  - If buffer empty, producer goes to sleep to be awaken when the consumer has removed one or more items
  - Similarly for the consumer
  - (a first try)

```c
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == MAX)
            pthread_cond_wait(&cond, &mutex);
        put(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == 0)
            pthread_cond_wait(&cond, &mutex);
        int tmp = get(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}
```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == MAX)
            pthread_cond_wait(&cond, &mutex);
        put(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == 0)
            pthread_cond_wait(&cond, &mutex);
        int tmp = get(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

With condition variables, always use while loops
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == MAX)
            pthread_cond_wait(&cond, &mutex);
        put(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 0)
            pthread_cond_wait(&cond, &mutex);
        int tmp = get(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

2 consumers/1 producer and lets MAX = 1

• Both consumers try to get the item, find buffer empty and go to sleep
• Producer puts item, wakes up a consumer (1) and goes to sleep
• Consumer comes along and gets the one item and signals …
• but who!? Both producer and Consumer 2 are sleeping
Finally a solution

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == MAX)
            pthread_cond_wait(&empty, &mutex);
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 0)
            pthread_cond_wait(&fill, &mutex);
        int tmp = get(i);
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);
    }
}
```

Simple solution – two condition variables

Producer waits on “empty” and signals “fill”
Consumers do the opposite – wait on “fill” and signal “empty”
Semaphores

- A synchronization primitive
- Higher level of abstraction than locks, also replacing condition variables
- Invented by Dijkstra in ‘68 as part of THE OS
- Atomically manipulated by two operations

```
sem_wait(sem_t *sem) / P / down(sem)
sem_post(sem_t *sem) / V / up(sem)
```

- The initial value determine its behavior, so it must be first initialized

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

Ignored this for now, but basically shared by all threads of a process (0) or by processes through shared memory (!=0)
Blocking in semaphores

- Each semaphore has an associated queue of processes/threads
  - `sem_wait / P`
    - Decrement the value of the semaphore by 1
    - If sem was “unavailable” (non-positive), wait on the queue
  - P – not really for *proberen* or *passeer* but for a made-up word *prolaag* – “try to reduce”

```c
int sem_wait(sem_t *s){
    s.value--;  
    wait in a queue of s until (s.value > 0);
}
```
Semaphores

...  
- `sem_post` / `V`
  - Increment the value of the semaphore by one
  - If thread(s) are waiting on the queue, unblock one
- `V` – *verhogen* – increase in Dutch

```c
int sem_post(sem_t *s) {
    s.value++;
    if there are 1+ threads waiting
    wake one thread up;
}
```

Atomic action
Binary semaphores - locks

```c
sem_t m;
sem_init(&m, 0, 1);

sem_wait(&m);
/* critical section */
sem_post(&m);
```

Why 1? Look at the definition of wait and post

```c
int sem_wait(sem_t *s) {
    s.value--;
    wait in a queue of s until (s.value > 0);
}

int sem_post(sem_t *s) {
    s.value++;
    if there are 1+ threads waiting
    wake one thread up;
}
```

So, if m = 1 the first thread will go in and decrement its value, the following thread will wait ... until the thread inside increments it within sem_post()
Semaphores as condition variables

- Waiting on a condition, as when parent waits for child to terminate

```c
sem_t s;

void *child(void *arg) {
    printf("child\n");
    sem_post(&s);
    return NULL;
}

int main(int argc, char*argv[]) {
    sem_init(&s, 0, 0);
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(c, NULL, child, NULL);
    sem_wait(&s);
    printf("parent: end\n");
    return 0;
}
```

So, if \( m = 0 \) and parent runs, will wait until the child runs and sets value to 1; If child runs first, the value will be 1 and the parent will go on without waiting
Semaphores – Producer/consumer v1

```c
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
    }
}

int main ...
...
sem_init(&empty, 0, MAX); /* MAX buffers are empty ... */
sem_init(&full, 0, 0); /* and 0 are full */
...
```

Yeap, those are CSs

```c
void put(int value) {
    buff[fill] = value;
    fill = (fill + 1) % MAX;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) & MAX;
    return tmp;
}
```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        int tmp = get();
        sem_post(&mutex);
        sem_post(&empty);
    }
}

int main ...
...
    sem_init(&empty, 0, MAX); /* MAX buffers are empty ... */
    sem_init(&full, 0, 0);    /* and 0 are full */
    sem_init(&mutex, 0, 1);   /* set to 1, it’s a lock */
...

Protect the critical section

Protect the critical section
Readers-writers problem

- The need for a more flexible type of lock, imagine a database or a simple linked list
  - Not problem with multiple readers allowed at once
  - Only one writer allowed at a time
    - If writers is in, nobody else is

```c
typedef struct _rwlock_t {
    sem_t lock;
    sem_t writelock;
    int readers;
} rwlock_t;

void rwlock_acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->writelock);
    rw->readers++;
    if (rw->readers == 1)
        sem_wait(&rw->writelock);
    sem_post(&rw->lock);
}

void rwlock_release_writelock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0)
        sem_post(&rw->writelock);
    sem_post(&rw->lock);
}
```

**Simple, only a single writer allowed**

**First reader blocks the writer from entering**

**Last reader lets the writer in**
Dining philosophers

- Another one by Dijkstra
- Philosophers eat/think
  - To eat, a philosopher needs 2 chopsticks
  - Picks one at a time
- *How to prevent deadlock and starvation*

```c
#define N 5

void philosopher(int i) {
  while (TRUE) {
    think();
    take_chopstick(i);
    take_chopstick((i+1)%N);
    eat();
    put_chopstick(i);
    put_chopstick((i+1)%N);
  }
}
```

*Why not just protect all this with a mutex?*

*Nonsolution*
void philosopher(int i) {
    while(TRUE) {
        think();
        take_chopstick(i);
        eat();
        put_chopstick(i);
    }
}

void put_chopstick(int i) {
    sem_wait(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}

void test(int i) {
    if ((state[i] == hungry &&
        state[LEFT] != eating &&
        state[RIGHT] != eating) {
        state[i] = EATING;
        sem_post(&s[i]);
    }
}

state[] — too keep track of philosopher’s state
(eating, thinking, hungry)
s[] — array of semaphores, one per philosopher
Semaphores and deadlocks

- Semaphores solves most synchronization problems
  - But no control or guarantee of proper usage

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);
        put(i);
        sem_post(&mutex);
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        int tmp = get();
        sem_post(&mutex);
        sem_post(&empty);
    }
}

Minor change?
```
Watch for deadlocks

- **Deadlock!**
  - Consumer holds the mutex and goes to sleep, to wait for the producer to put something
  - Producer can’t put anything because consumer holds the lock!

```c
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
    }
}
```
Issues with semaphores

- Solves most synchronization problems, but:
  - We have seen, no control over their use, no guarantee of proper usage (our deadlock example)
  - Semaphores are essentially shared global variables
    - Can be accessed from anywhere (bad software engineering)
  - No connection between the semaphore & the data controlled by it
  - Used for both critical sections & for coordination (scheduling)
Monitors

- Higher level synchronization primitive – Monitors
  - A programming language construct
    - Set of procedures, variables and data structures
    - Monitor’s internal data structures are private

- Monitors and mutual exclusion
  - Only one process active at a time

- To enforce sequences of events – Condition variables
  - Only accessed from within the monitor
  - Three operations – wait, signal & broadcast
Monitors

- **Wait**
  - Atomically releases the lock
  - Suspends execution of the calling thread, places it in the waiting queue
  - When the calling thread is re-enable, it requires the lock before returning from the wait

- A thread that waits “steps outside” the monitor (to the associated wait queue)

- A condition variable is memoryless, it has no internal state (the shared object defines its own); so, *wait* is not a counter – signal may get lost
Monitors

- **Signal**
  - Takes a waiting thread off the condition variable’s waiting queue and marks it as eligible to run

- **Broadcast**
  - Like signal but for all threads waiting

- **What happen after the signal?**
  - Hoare – process awakened run
  - Brinch Hansen – process signaling must exit
  - *Mesa* – process signaling continues to run

- **As a programmer – always check the condition after being woken! i.e., call within a while**

  ```
  while (predicateOnStateVar(...)) wait(&lock);
  ```
Monitor ProdCons {
    condition full, empty;
    int count;

    void insert(int item) {
        if (count == N) wait(full);
        insert_item(item);
        count++;
        if (count == 1) signal(full)
    }

    int remove(void) {
        if count == 0 wait(empty);
        return remove_item;
        count--;
        if (count == N - 1)
            signal(full);
    }
    count := 0;
}

void producer() {
    while TRUE {
        item = produce_item;
        ProdCons.insert(item);
    }
}

void consumer() {
    while TRUE {
        item = ProdCons.remove;
        consume_item(item);
    }
}
Monitors

- Monitors and mutual exclusion
  - Only one process active at a time – how?
  - Synchronization code is added by the compiler (or the programmer using locks)

- Clear similarities between the two – you can use one to implement the other
Monitors and semaphores

- A semaphore implemented as a monitor

Monitor class Semaphore {
    int s;
    Semaphore(int value) {
        s = value;
    }
    void wait() {
        while (s <= 0)
            wait();
        s--;
    }
    void post() {
        s++;
        signal();
    }
}

Using it as a binary semaphore

Semaphore s(1);

s.wait();
/* Critical section */
s.post();
Monitors

- Monitors - higher level synchronization primitive
  - A programming language construct
    - Collection of procedures, variables and data structures
    - Monitor’s internal data structures are private

- Monitors and mutual exclusion
  - Only one process active at a time
  - Synchronization code is added by the compiler (or the programmer using locks)

- To enforce sequences of events – Condition variables
  - Only accessed from within the monitor
  - Three operations – wait, signal & broadcast
Coming up

- **Deadlocks**
  How deadlock arise and what you can do about them