

ID1217 Concurrent Programming

Lecture 10



ROYAL INSTITUTE
OF TECHNOLOGY

Semaphores

Vladimir Vlassov
KTH/ICT/EECS



ROYAL INSTITUTE
OF TECHNOLOGY

Outline

- Origin, syntax and semantics of semaphores
- Binary, split binary, general semaphores
- Use of binary semaphores for synchronization
 - Critical sections with semaphores
 - Signaling semaphores
 - Barriers using semaphores
 - Split binary semaphores
 - The Consumer-Producer problem
- Use of general semaphores
 - The bounded buffer problem (the consumer-producer problem)
- Selective mutual exclusion using semaphores
 - The dining philosophers problem
 - The readers-writers problem
- Resource allocation and scheduling using semaphores
- Semaphores in the Pthreads API



ROYAL INSTITUTE
OF TECHNOLOGY

Semaphores

- ***Semaphores*** is a unified, low-level but efficient signaling mechanism for both mutual exclusion and condition synchronization.
- Origin of semaphores:
 - Invented in 1968 by Edsger Dijkstra, Dutch computer scientist
 - Inspired by a railroad semaphore – Up/Down signal flag
 - THE system – the first OS used semaphores
 - Semaphore operations
 - P stands for Dutch “Proberen” (to test) or “Passeren” (to pass)
 - V stands for Dutch “Verhogen” (to increment) or “Vrijgeven” (to release)

Syntax and Semantics of Semaphores

- A ***semaphore*** is a special kind of a shared integer variable (or an abstract data type) which can only be accessed using the following two atomic operations:

```
P(s): < await (s > 0) s = s - 1; > // Pass (Passeren)  
V(s): < s = s + 1; > // Release (Vrijgeven)
```

- The initial semaphore value should be **non-negative**: $s \geq 0$
- Declaration and initialization:

```
sem s = expr; // single semaphore  
sem s[1:n] = ([n] expr); // array of semaphores
```

- If not initialized, the semaphore value defaults to zero

Implementation

- **Blocking semantics:** To avoid busy-waiting, a semaphore should have an associated queue of processes (usually a FIFO).
- **P(s) operation:**

```
if (s > 0) s--;  
else { /* wait on the semaphore */  
    Suspend the process and place it to the tail of the semaphore's queue;  
}
```

– The blocked process can be released by V(s);

- **V(s) operation:**

```
if (queue is empty) s++;  
else { /* resume a waiting process */  
    Resume the process in the head of the semaphore queue, i.e. take the  
    process off the semaphore's queue and move it to the ready queue;  
}
```

Usage Types of Semaphores

- Depending on usage, semaphores can be distinguished as follows.
- **Binary semaphore** takes the value 0 or 1
 - Used as a mutex (lock) for mutual exclusion; initialized to 1
 - Used as a flag (signaling semaphore) for condition synchronization; initialized to 0
- **Split Binary semaphore** is a set of binary semaphores where at most one semaphore is 1 at a time:

$$0 \leq s_0 + s_1 + \dots + s_{n-1} \leq 1$$

- Can be used for both mutual exclusion and condition synchronization
- **General (counting) semaphore** takes any nonnegative integer value
 - Serves as a resource counter: counts the number of recourse units available
 - Used for condition synchronization.

Critical Sections Using a Binary Semaphore (Mutex)

- A critical section of code `< S; >` can be executed with mutual exclusion by enclosing it between P and V operations on a binary semaphore.
 - The mutex semaphore is initialized to 1 to indicate CS is free

```
sem mutex = 1;

process CS[i = 1 to n] {
    while (true) {
        P(mutex);
        critical section;
        V(mutex);
        noncritical section;
    }
}
```



ROYAL INSTITUTE
OF TECHNOLOGY

Signaling Semaphores

- *Signaling semaphore* is a binary semaphore used as a flag for signaling events or conditions.
 - Usually initialized to 0 to indicate absence of an event
 - A process signals an event by $V(s)$ – “sends” a signal
 - A process waits for event by $P(s)$ – “receives” a signal

Barriers Using Signaling Semaphores

- Use two binary semaphores per a pair-wise barrier to signal arrival at the barrier and to control departure
- N-process barrier can be built by combining pair-wise barriers

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
    ...
    V(arrive1);      /* signal arrival          */
    P(arrive2);      /* wait for other process */
    ...
}
process Worker2 {
    ...
    V(arrive2);      /* signal arrival          */
    P(arrive1);      /* wait for other process */
    ...
}
```

Mutual Exclusion Using a Binary Semaphore

- **Mutual exclusion** $\langle S; \rangle$
 - Use a mutex semaphore:

```
sem entry = 1; // CS entry semaphore
P( entry );
S;
V( entry );
```

Condition Synchronization Using Binary Semaphores

- **Condition synchronization** `<await (B) S;>`
 - Use two semaphores
 - a mutex semaphore for exclusive access
 - a signaling semaphore to wait for condition

```
sem entry = 1; // CS entry semaphore
sem cond = 0;  // signaling semaphore
P(entry);
while (!B) { V(entry); P(cond); P(entry); }
S;
V(entry);
```

- The “Passing the baton” technique can be used to optimize the code above.

Passing-the-Baton Technique

- Assume, a proc **W** awaits for a condition in its CS:

```
W: P(entry); while (!B) { V(entry); P(cond); P(entry); }
```

- To proceed, it waits for two semaphores **cond** and **entry**

- Assume, a proc **S** sets the condition, signals, and releases lock:

```
S: P(entry); B = true; V(cond); V(entry);
```

- It increments **cond** (to signal condition), and **entry** (to release CS)

- Passing-the-baton technique**

- a signaling proc (S) passes ownership of the mutex semaphore to the waiting proc (W) by using the signaling semaphore

```
W: P(entry); while (!B) { V(entry); P(cond); } S; V(entry);  
S: P(entry); B = true; V(cond);
```

- Note that the above code is unsafe: it deadlocks if S executes first. To avoid deadlock, S should check whether W is waiting. See next slide

Passing-the-Baton Illustrated

```
sem entry = 1, /* controls entry to CS */
    cond = 0; /* to wait for the condition */
int dp = 0; /* counts delayed processes */
process W[i = 0 to n - 1] {
    ...
    P(entry);
    if (!B) {
        dp++;
        V(entry);
        P(cond); /* delays on cond */
    }
    critical section; /* critical section */
    if (dp > 0) {
        dp--;
        V(cond); /* pass the entry "baton" */
    } else V(entry); /* release entry */
    ...
}
```

Passing entry sem

```
process S {
    ...
    P(entry);
    change B to true;
    if (dp > 0) {
        /* pass the "baton" */
        dp--;
        V(cond);
    } else V(entry);
    ...
}
```



ROYAL INSTITUTE
OF TECHNOLOGY

Split Binary Semaphores

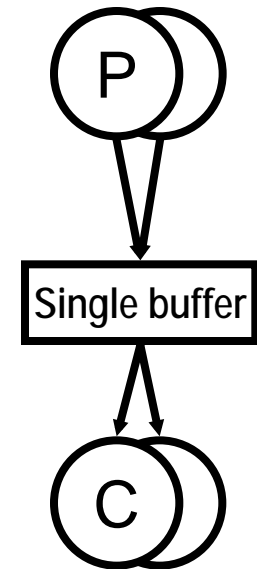
- *A Split Binary semaphore* is a set of binary semaphores where at most one semaphore is 1 at a time:

$$0 \leq s_0 + s_1 + \dots + s_{n-1} \leq 1$$

- combines mutual exclusion and signaling;
- used to indicate alternate states
 - For example: Two binary semaphores **full** and **empty** forms a split binary semaphore that indicates whether a variable (buffer) is full or empty

Example 1: Consumer/Producer Problem (revisited)

- Illustrates a typical usage of binary semaphores
- Producers (P) and Consumers (C) interact using a single shared buffer.
 - The buffer is assumed to be empty initially.
 - A Producer must be delayed until the buffer is empty
 - A Consumer must be delayed until the buffer is full.



Solution to the Consumer/Producer Problem

Using Semaphores

- Mutual exclusive access to the buffer
- Condition synchronization
 - Consumer should wait until the buffer is full;
 - Producer should wait until the buffer is empty.



Consumer/Producer Using Semaphores

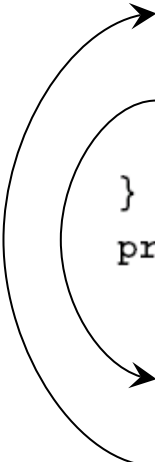
- Use of two binary semaphores allows to achieve both, mutual exclusion and condition synchronization
- Two binary semaphores, **full** and **empty**, form a split binary semaphore.
- Either of them or non of them is 1 at a time:

$$0 \leq \text{full} + \text{empty} \leq 1$$

- When **full** is 1 and **empty** is 0 – the buffer is full and unlocked for Consumer but locked for Producer
- When **empty** is 1 and **full** is 0 – the buffer is empty and unlocked for Producer but locked for Consumer
- When both sems are 0 – the buffer is locked and accessed by only one process (either Consumer or Producer)

Producers and Consumers Using Semaphores

```
typeT buf;      /* a buffer of some type T */
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
    while (true) {
        ...
        /* produce data, then deposit it in the buffer */
        P(empty);
        buf = data;
        V(full);
    }
}
process Consumer[j = 1 to N] {
    while (true) {
        /* fetch result, then consume it */
        P(full);
        result = buf;
        V(empty);
        ...
    }
}
```





ROYAL INSTITUTE
OF TECHNOLOGY

Property of a Split Binary Semaphore

- Suppose,
 - Processes are using a split binary semaphore and each process has its binary semaphore in the split semaphore to wait on;
 - Every critical section in every proc starts with a P(my semaphore) and ends with a V(other semaphore).
- Then all statements between any P and the next V execute with mutual exclusion.
 - By definition: $0 \leq s_0 + s_1 + \dots + s_{n-1} \leq 1$
 - Observation: wherever any process is between any P and the next V, all the semaphores are zero
 - This implies that no any other process can complete a P until the first process executes a V



ROYAL INSTITUTE
OF TECHNOLOGY

General Semaphores

- Can be associated with a shared resource and serve as a resource counter:
 - Initialized to some integer value – the total amount of resource units available;
 - Takes any nonnegative integer value – current amount of resource units available;
 - Used for condition synchronization, e.g. wait until a unit of resource is available and can be occupied.

Example 2: Bounded Buffer Problem

- *Producers* and *Consumers* communicate via a *bounded buffer* – a multi-slot communication buffer limited in size (no underflow, no overflow)
- Buffer of size **n** as an array of some type T :

TypeT buf[n];

```
int front = 0, rear = 0;
```

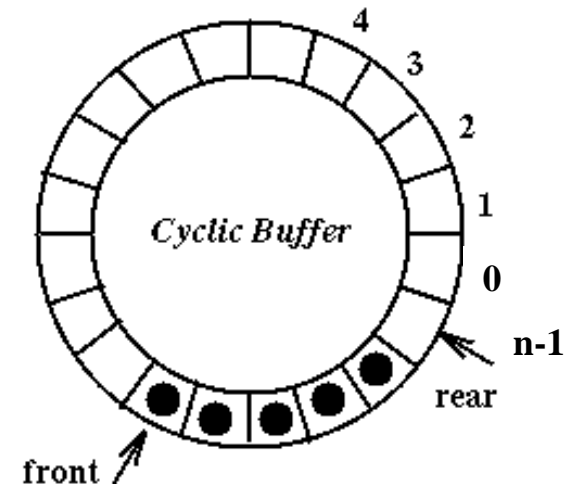
- Accessing buffer
 - **rear** points to an empty slot
 - **front** points to the head data item

```
deposit: buf[rear] = data;
```

```
rear = (rear + 1) % n;
```

```
fetch:    result = buf[front];
```

```
front = (front + 1) % n;
```



Bounded Buffer Using General Semaphores

- Synchronized access to the bounded buffer:
- Producer waits until there is an empty slot in the buffer
- Consumer must wait until there is a full slot in the buffer.
- Two semaphores:
 - sem empty = n;**
 - sem full = 0;**
 - **empty** counts empty slots,
 - **full** counts full slots.

```
typeT buf[n];          /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0; /* n-2 <= empty+full <= n */

process Producer {
    while (true) {
        ...
        produce message data and deposit it in the buffer;
        P(empty);
        buf[rear] = data; rear = (rear+1) % n;
        V(full);
    }
}

process Consumer {
    while (true) {
        fetch message result and consume it;
        P(full);
        result = buf[front]; front = (front+1) % n;
        V(empty);
        ...
    }
}
```

Example 3: Multiple Producers/Multiple Consumers and a Bounded Buffer

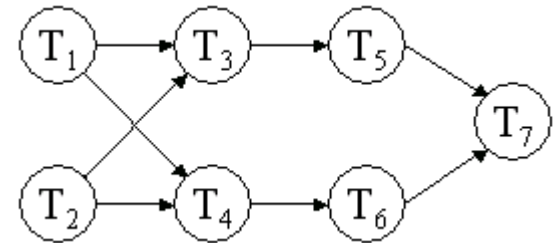
- Deposit becomes critical section for Producers:
`buf[rear] = data;`
`rear = (rear + 1) % n;`
- Fetch becomes critical section for Consumers:
`result = buf[front];`
`front = (front + 1) % n;`
- Deposit and fetch are not critical to each other because they access different locations: deposit uses **rear**, fetch – **front**
- To achieve mutual exclusion, use two binary semaphores
 - **mutexD** to protect **rear** in deposit called by Producers
 - **mutexF** to protect **front** in fetch called by Consumers

Producers/Consumers Using Semaphores

```
typeT buf[n];          /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0;    /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1; /* for mutual exclusion */
process Producer[i = 1 to M] {
    while (true) {
        ...
        produce message data and deposit it in the buffer;
        P(empty);
        P(mutexD);
        buf[rear] = data; rear = (rear+1) % n;
        V(mutexD);
        V(full);
    }
}
process Consumer[j = 1 to N] {
    while (true) {
        fetch message result and consume it;
        P(full);
        P(mutexF);
        result = buf[front]; front = (front+1) % n;
        V(mutexF);
        V(empty);
        ...
    }
}
```


Exercise: An Execution Order

- Consider the following precedence graph of processes:



- Each process has the following code outline:

```
process Ti (i = 1, ..., 7) {  
    wait for predecessors, if any;  
    execute the task;  
    signal successor, if any;  
}
```
- For example, in the above graph above, task T5 has to wait for task T3 and signals task T7.
- Develop an implementation of the wait and signal code for each of the seven tasks in the above graph. Use semaphores for synchronization. Try to use a minimal number of semaphores.

Solutions

- Using 5 semaphores, one per edge target:

```
sem S[3:7] = ([5] 0); // init
T1:: ... V(S[3]); V(S[4]);
T2:: ... V(S[3]); V(S[4]);
T3:: P(S[3]); P(S[3])... V(S[5]);
T4:: P(S[4]); P(S[4])... V(S[6]);
T5:: P(S[5]);... V(S[7]);
T6:: P(S[6]);... V(S[7]);
T7:: P(S[7]); P(S[7]);...
```

- As T7 executes after T3 (and T4), one semaphore, e.g. , S3, can be reused.
One of possible solutions using 4 semaphores:

```
sem S[3:6] = ([4] 0); // init
T1:: ... V(S[3]); V(S[4]);
T2:: ... V(S[3]); V(S[4]);
T3:: P(S[3]); P(S[3])... V(S[5]); V(S[5]);
T4:: P(S[4]); P(S[4])... V(S[6]);
T5:: P(S[5]);... V(S[3]);
T6:: P(S[6]);... V(S[3]);
T7:: P(S[5]); P(S[3]); P(S[3]);...
```

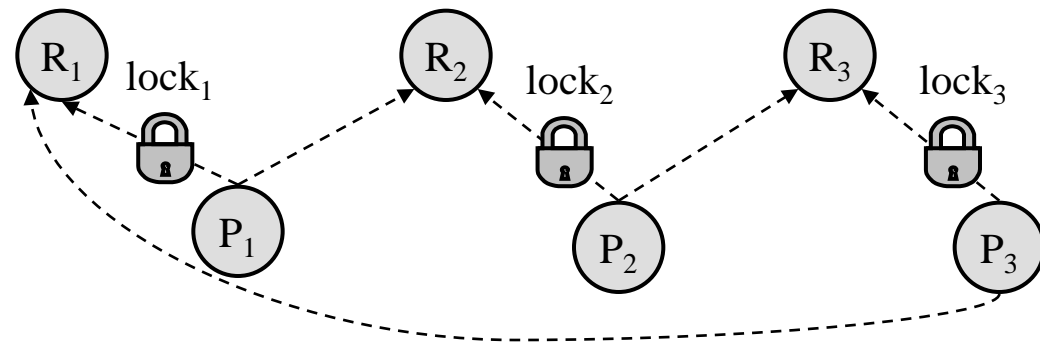
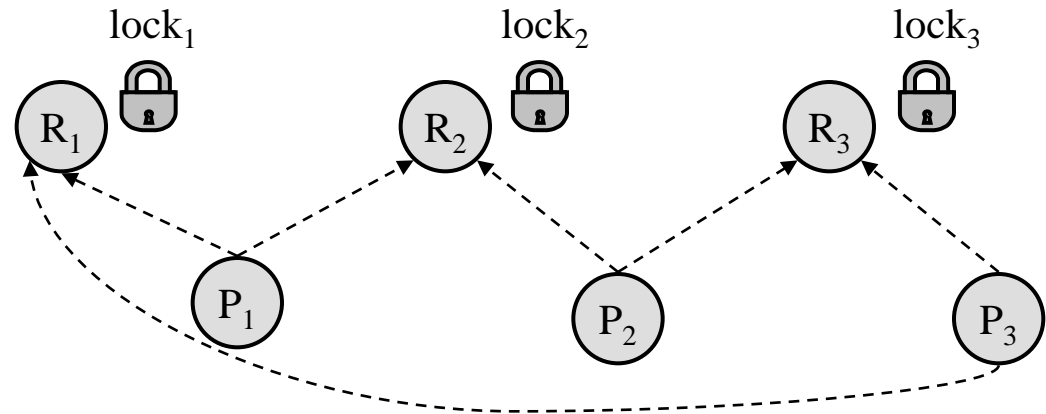


Selective Forms of Mutual Exclusion

- Two classical synchronization problems of *selective mutual exclusion*
- *Dining Philosophers problem*
 - Several (identical) processes competing for overlapping sets of shared resources
 - Each process needs several shared resources
- *Readers/Writers problem*
 - Different classes of processes (or different types of accesses) competing for the same shared resource , e.g. buffer, database
 - Concurrent access can be allowed for some processes, e.g. readers
 - Mutual exclusion for other processes, e.g. writers

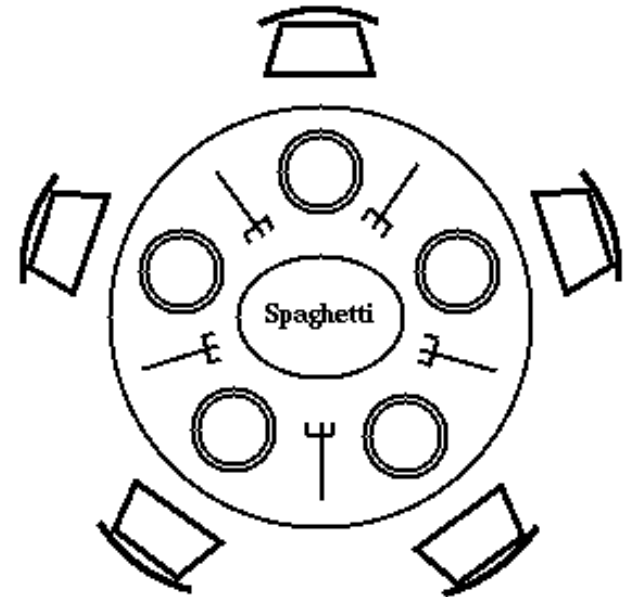
Several Processes Competing for Several Shared Resources

- Consider several processes (P) and several resources (R) each protected with a lock
 - A proc must acquire locks for all resource it needs
- Deadlock may occur when several proc compete for overlapping sets of resources
- For example,
 - Each $P[i]$ needs $R[i]$ and $R[(i+1) \bmod n]$
 - Deadlock**: each proc acquires lock[i] and waits for lock[(i+1) mod n] which is already taken by $P[(i+1) \bmod n]$



Dining Philosophers

- Several processes (Philosophers) competing for several shared resources (forks)
- Dining Philosophers:
 - Five (generally, n) philosophers sit around a table. Each philosopher spends his life alternately thinking and eating.
 - There are a plate of spaghetti (bowl of rice) and five (generally, n) forks (chopsticks): one fork between each pair of philosophers
 - To eat, each philosopher needs two forks, and they agreed that each will use only the forks to his immediate left and right



The Dining Philosophers Problem

- To write a program simulating the behavior of the philosophers.
 - The program must avoid deadlock: the situation in which each philosopher picks up exactly one fork and all the philosophers starve to death.
 - Assume, the lengths of the thinking and eating periods vary.
 - The program outline (model):

```
process Philosopher[i = 0 to 4] {  
    while (true) {  
        think;  
        acquire a pair of forks;  
        eat;  
        release the forks;  
    }  
}
```



ROYAL INSTITUTE
OF TECHNOLOGY

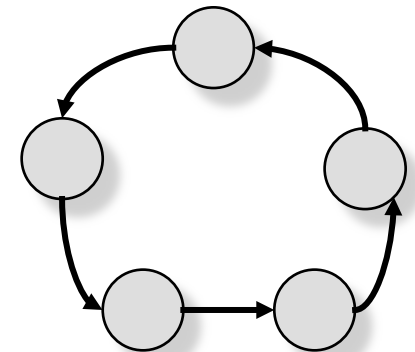
Protect Forks with Binary Semaphores

- Idea: to protect the forks with binary semaphores:
 - Each fork can be used by only one philosopher at a time;
 - Eating phase is a critical section w.r.t. to two forks: one to the left and another to the right;
 - To get two forks: **P(left); P(right)**

First attempt to solve the problem

- Symmetrical: Each gets right fork then tries to get left

```
sem fork[5] = ( [5] 1 );  
process Philosopher[i = 0 to 4] {  
    while (true) {  
        P(fork[i]); P(fork[(i+1)%5]);  
        eat;  
        V(fork[i]); V(fork[(i+1)%5]);  
        think;  
    }  
}
```



- Leads to deadlock because of possible circular waiting

Deadlock-Free Solutions

- How to avoid the deadlock? Break the circle!
 - Introduce asymmetry in acquiring semaphores (getting forks)
 - Odd-numbered phs get their right then left forks, even-numbered get their left then right forks:
 - Limit the number of eating philosophers to $n-1$ at a time:

```
sem fork[5] = ( [5] 1 );
process Philosopher [i = 0 to 4] {
    while (true) {
        if (i%2 == 0) {
            P(fork[(i+1)%5]); P(fork[i]);
        } else {
            P(fork[i]); P(fork[(i+1)%5]);
        }
        eat;
        V(fork[i]); V(fork[(i+1)%5]);
        think;
    }
}
```

```
sem fork[5] = ( [5] 1 );
sem limit = 4; /* res. counter */
process Philosopher[i = 0 to 4] {
    while (true) {
        P(limit);
        P(fork[i]); P(fork[(i+1)%5]);
        eat;
        V(limit);
        V(fork[i]); V(fork[(i+1)%5]);
        think;
    }
}
```



ROYAL INSTITUTE
OF TECHNOLOGY

The Readers/Writers Problem

- A specific case of the Consumer/Producer problem
- Models accessing to shared resource(s) by different classes of processes
 - Selective mutual exclusion.
 - Very common for many concurrent applications.
- Assume, two kinds of processes, Readers and Writers, share a database.
 - Readers execute read transactions that examine database records.
 - Writers execute update transactions that both examine and update the records.
- ***The Readers/Writers problem:*** To develop a synchronization mechanism that isolates any updates issued by a Writer process from transactions issued by other processes.
 - Concurrent reads are allowed.
 - Writes are executed with mutual exclusion with reads and other writes.

(a) An Over-Constrained Solution using One Semaphore

- All processes access the database with mutual exclusion
 - At most one process (either Reader or Writer) at a time
 - Does not allow simultaneous access,
- The solution precludes concurrent reads

```
sem rw = 1;

process Reader[i = 1 to M] {
    while (true) {
        ...
        P(rw);    # grab exclusive access lock
        read the database;
        V(rw);    # release the lock
    }
}

process Writer[j = 1 to N] {
    while (true) {
        ...
        P(rw);    # grab exclusive access lock
        write the database;
        V(rw);    # release the lock
    }
}
```

(b) A Reader's Preference Solution

- Let's relax the over-constrained solution just enough to allow concurrent reads.
- Idea:
 - A reader that starts reading first does **P(rw)**, i.e. it locks the database for its own and further reads if any.
 - A reader that finishes reading last does **V(rw)**, i.e. it unlocks the database.
 - This requires a counter of concurrent reads, **nr**
 - Each reader increments the counter when starts reading and decrements the counter when finishes reading
 - **nr** is a shared variable that has to be accessed atomically (with mutual exclusion) – use a binary semaphore, **mutexR**, to protect it

The Reader's Preference Solution to the Readers/Writers Problem

```
int nr = 0;          # number of active readers
sem rw = 1;          # lock for access to the database
sem mutexR = 1;      # lock for reader access to nr

process Reader[i = 1 to m] {
    while (true) {
        ...
        P(mutexR);
        nr = nr+1;
        if (nr == 1) P(rw); # if first, get lock
        V(mutexR);
        read the database;
        P(mutexR);
        nr = nr-1;
        if (nr == 0) V(rw); # if last, release lock
        V(mutexR);
    }
}

process Writer[j = 1 to n] {
    while (true) {
        ...
        P(rw);
        write the database;
        V(rw);
    }
}
```

Shortcomings of Solution (b)

- Guarantees appropriate exclusion, but it is not fair: it gives Readers preference in accessing the database
- How to make the solution fair? or give writers preference?
 - Very tricky to modify the code (b) to do this.
- A fair solution should use condition synchronization and the “passing the baton” technique, i.e. **passing the mutex baton (semaphore) to one process at a time (if any waiting).**

(c) Readers/Writers Using “Passing the baton”

- Let:
 - **nr** is the number of active readers (initially zero)
 - **nw** is the number of active writers (initially zero)
 - **dr** is the number of delayed readers (initially zero)
 - **dw** is the number of delayed writers (initially zero)
- Define a global invariant (RW):
 - BAD states (to be avoided): $(nr > 0 \wedge nw > 0) \vee (nw > 1)$
 - GOOD states:
$$RW: (nr == 0 \vee nw == 0) \wedge nw \leq 1$$
- To guarantee *RW*:
 - reader: `<await (nw == 0) nr++>; reads; <nr-->;`
 - writer: `<await (nr == 0 ∧ nw == 0) nw++>; writes; <nw-->;`

(c) A Readers/Writers Solution Using Condition Synchronization and “Passing the Baton” technique

- If no Writers is writing, a Reader passes the baton to all pending Readers if any otherwise to a pending Writer if any;
- When finishes writing, a Writer passes the baton to all pending Readers if any otherwise to a pending Writer if any.
- **Still gives preference to readers**

```

int nr = 0,    ## RW: (nr==0 or nw==0) and nw<=1
    nw = 0;
sem e = 1,    # controls entry to critical sections
    r = 0,    # used to delay readers
    w = 0;    # used to delay writers
              # at all times 0 <= (e+r+w) <= 1
int dr = 0,    # number of delayed readers
    dw = 0;    # number of delayed writers

process Reader[i = 1 to M] {
    while (true) {
        # <await (nw == 0) nr = nr+1;>
        P(e);
        if (nw > 0) { dr = dr+1; V(e); P(r); }
        nr = nr+1;
        if (dr > 0) { dr = dr-1; V(r); }
        else V(e);
        read the database;
        # <nr = nr-1;>
        P(e);
        nr = nr-1;
        if (nr == 0 and dw > 0) { dw = dw-1; V(w); }
        else V(e);
    }
}

process Writer[j = 1 to N] {
    while (true) {
        # <await (nr == 0 and nw == 0) nw = nw+1;>
        P(e);
        if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
        nw = nw+1;
        V(e);
        write the database;
        # <nw = nw-1;>
        P(e);
        nw = nw-1;
        if (dr > 0) { dr = dr-1; V(r); }
        elseif (dw > 0) { dw = dw-1; V(w); }
        else V(e);
    }
}

```


A Writer's Preference Solution

- Feature of solution (c): Always pass the mutual exclusion baton to one process at a time
 - We can control passing the baton
- **To give preference to Writers:**
 - New reader are delayed if a writer is waiting (modify the first **if** in Readers):

```
if (nw > 0 or dw > 0) { dr = dr+1; V(e); P(r); }
```
 - A delayed reader is awoken only if no writer is waiting (modify the last **if** in Writers)

```
if (dw > 0) { dw = dw-1; V(w); }  
else if (dr > 0) { dr = dr-1; V(r); }  
else V(e);
```



ROYAL INSTITUTE
OF TECHNOLOGY

To Ensure Fairness in Readers-Writers

- Assume, semaphores are fair
- For a fair solution to the Readers/Writers problem, we need:
 - Delay a new reader when a writer is waiting;
 - See the previous slide
 - Delay a new writer when a reader is waiting;
 - in a similar way as with readers
 - Awaken one waiting writer (if any) when a reader finishes;
 - The solution (c) on Slide 41 already meets this
 - Awaken all waiting readers (if any) when a writer is finishes; otherwise awaken one waiting writer (if any).
 - The solution on Slide 41 already meets this



ROYAL INSTITUTE
OF TECHNOLOGY

Resource Allocation Using Semaphores

- Assume, processes compete for units of a shared resource
- General outline of resource allocation
 - Request and release procedures (operations):

```
request(parameters):< await (request can be satisfied) take units; >  
release(parameters): < return units;>
```

- In general case: request parameters
 - Number of units;
 - Time to occupy the resource;
 - Other specific.



ROYAL INSTITUTE
OF TECHNOLOGY

Resource Allocation (cont'd)

- A general semaphore can be used to count resource units
- Units can be controlled (counted) by a general semaphore
 - $P(r)$ – request a resource unit, if not available wait
 - $V(r)$ – release a resource unit
- Passing the baton technique can be used to pass the resource lock (permission to use) from one process to another



ROYAL INSTITUTE
OF TECHNOLOGY

Resource Allocation Using Passing the Baton

```
sem e = 1;    // a CS entry semaphore
sem delay = 0;    // a signaling semaphore to delay a process
request( parameters ):
    P(e);
    if (request cannot be satisfied)
        DELAY; /* record the delayed process; V(e); P(delay); */
    take units;
    SIGNAL; /* awaken a delayed proc if any V(delay) otherwise V(e) */

release(parameters):
    P(e);
    return units;
    SIGNAL; /* awaken a delayed proc if any V(delay) otherwise V(e) */
```



ROYAL INSTITUTE
OF TECHNOLOGY

Shortest Job Next (SJN) Allocation***

- *SJN* is an allocation policy which allocates a resource to a requesting process which will use the resource for the shortest amount of time.

```
request(time, id) {  
    if (free) take resource;  
    else  
        delay by time; /* delayed processes are ordered by time */  
}  
  
release(id) {  
    if (some process delayed)  
        awaken first one ( with the min value of time) and give it the resource;  
    else  
        make resource free;  
}
```

SJN Allocation Using Semaphores

- Need to maintain a list of delayed proc ordered by **time** in requests
 - Here: **Pairs** is a set of pairs (**time**, **id**) ordered by **time**
- Each proc has its private signaling semaphore **b[id]** to wait for the resource

```
bool free = true;
sem e = 1, b[n] = ([n] 0); # for entry and delay
typedef Pairs = set of (int, int);
Pairs pairs = ∅;
## SJN: pairs is an ordered set  $\wedge$  free  $\Rightarrow$  (pairs == ∅)

request(time,id):
    P(e);
    if (!free) {
        insert (time,id) in pairs;
        V(e);          # release entry lock
        P(b[id]);      # wait to be awakened
    }
    free = false;
    V(e);              # optimized since free is false here

release():
    P(e);
    free = true;
    if (P ( pairs != ∅) {
        remove first pair (time,id) from pairs;
        V(b[id]);      # pass baton to process id
    }
    else V(e);
```



ROYAL INSTITUTE
OF TECHNOLOGY

Pros and Cons of Semaphores

- Semaphores: binary mutex (typically initialized to 1), signaling (typically initialized to 0), general (initialized to a positive integer)
- Sufficiently general to solve any mutual exclusion or condition synchronization problem
- Disadvantages:
 - Complexity: Can be difficult to understand as the same operations are used for both mutual exclusion and condition synchronization.
 - Error prone: It is easy to make mistakes, such as forgetting a V operation, especially when several semaphores are used. Such error typically causes a deadlock.
 - Loss of modularity since the correct use of a semaphore in one process often depends on its use in another process.
 - Semaphores are too low-level.
- A combination of locks with condition variables have the same synchronization power as semaphores

Semaphores in Pthreads

- The semaphore extension for pthreads is implemented with mutexs and condition variables

To program: **#include <semaphore.h>**

- Defines semaphore types and functions

To link: **-lpthread**

- Semaphore functions

- A *semaphore* is an object of the **sem_t** type that can take any nonnegative value and can be altered by P (decrement) and V (increment) operations
 - If the semaphore is zero, the P operation blocks the calling thread until the semaphore is positive.

Semaphores in Pthread API (cont'd)

- Declaration: **sem_t sem;**
- Functions
 - **set_wait(&sem)** – P(sem) – decrements the semaphore if it's positive, otherwise blocks the process until the semaphore is positive
 - **sem_post(sem)** – V(sem) – increments the semaphore
- For example:

```
sem_init(&sem, SHARED, 1); /* sem = 1; */  
...  
sem_wait(&sem);           /* P(sem) */  
Critical section;  
sem_post(&sem);           /* V(sem) */
```



ROYAL INSTITUTE
OF TECHNOLOGY

Example 4: Simple Producer/Consumer Using Semaphores

```
#ifndef _REENTRANT
#define _REENTRANT
#endif
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1
void *Producer(void *); /* the two threads */
void *Consumer(void *);
sem_t empty, full; /* the global semaphores */
int data; /* shared buffer */
int numIters;
/* main() -- read command line, create threads,
join them and print result */
int main(int argc, char *argv[]) {
    pthread_t pid, cid;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    numIters = atoi(argv[1]);
    sem_init(&empty, SHARED, 1); /* sem empty = 1 */
    sem_init(&full, SHARED, 0); /* sem full = 0 */
    printf("main started\n");
    pthread_create(&pid, &attr, Producer, NULL);
    pthread_create(&cid, &attr, Consumer, NULL);
    pthread_join(pid, NULL);
    pthread_join(cid, NULL);
    printf("main done\n");
}
```

URL of the course /labs/pc.sems.c

Example4: Simple Producer/Consumer

Using Semaphores (cont' d)

```
/* deposit 1, ..., numIters into the data buffer */
void *Producer(void *arg) {
    int produced;
    printf("Producer created\n");
    for (produced = 0; produced < numIters; produced++) {
        sem_wait(&empty);
        data = produced;
        sem_post(&full);
    }
}

/* fetch numIters items from the buffer and sum them */
void *Consumer(void *arg) {
    int total = 0, consumed;
    printf("Consumer created\n");
    for (consumed = 0; consumed < numIters; consumed++) {
        sem_wait(&full);
        total = total+data;
        sem_post(&empty);
    }
    printf("for %d iterations, the total is %d\n", numIters, total);
}
```

Exercise 1: The Cigarette Smokers Problem

(based on Exercise 4.27 in the MPD book)

- Suppose there are three Smoker processes and one Agent process. Each smoker continuously makes a cigarette and smokes it. Making a cigarette requires three ingredients: tobacco, paper, and a match. One smoker process has tobacco, the second paper, and the third matches. Each has an infinite supply of these ingredients (i.e. generates them). The agent places a random two ingredients on the table. The smoker who has the third ingredient picks up the other two, makes a cigarette, and then smokes it. The agent waits for the smoker to finish smoking. The cycle then repeats.
- Develop a solution to this problem using semaphores for synchronization, i.e. develop an outline of a parallel program that simulates smokers and the agent.

Possible Solution Using 4 Semaphores

- Denote 1 - Tobacco, 2 - Paper, 3 - Match.

```
sem missing[1:3] = ([3]0), finished = 0;
set ingredients = {};
process Agent {
    while(true) {
        int i = choose_randomly(from {1, 2, 3});
        ingredients = {1, 2, 3} - {i};
        V(missing[i]);
        P(finished);
    }
}
process Smoker [i = 1 to 3] { // my ingredient is i
    while (true) {
        P(missing[i]);
        make_cigarette(ingredients + {i}); smoke();
        V(finished);
    }
}
```



ROYAL INSTITUTE
OF TECHNOLOGY

Exercise 2: Sleep and Wakeup

- The UNIX kernel provides two atomic operations similar to the following:
 - **sleep()**: block the executing process
 - **wakeup()**: awaken all blocked processes

A call of **sleep** always blocks the caller. A call of **wakeup** awakens every process that has called **sleep** since the last time that **wakeup** was called.

- Develop an implementation of these primitives using semaphores for synchronization. Be sure to declare and initialize any variables and semaphores that you use.

Exercise 3: Broadcast and Listen

- Consider the following primitives:
 - **broadcast(message)**: Give a copy of a given message to every process that is *currently* listening, and wait for all of them to take it.
 - **listen(x)**: Wait for the *next* broadcast, then get a copy of the message in a local variable x.
- A producer process uses broadcast to give a message to all consumers who are *currently* listening. If there are none, then broadcast(m) has no effect. If consumers are listening, the producer waits for each of them to have taken a copy of m before proceeding. Thus *both* primitives broadcast and listen block until the message has been transferred.
- Develop an outline of broadcast and listen. Assume that messages are single integers. Use semaphores for synchronization. Assume that there are multiple producers and multiple consumers, but that there can only be one broadcast in progress at a time. Be sure to declare and initialize shared variables and semaphores.