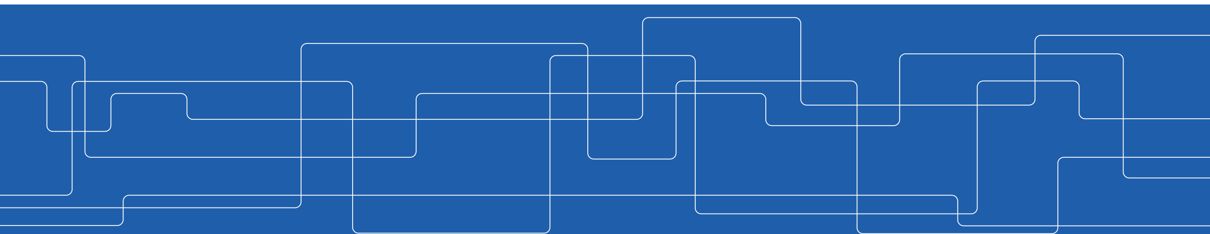
Processes Synchronization - Part I

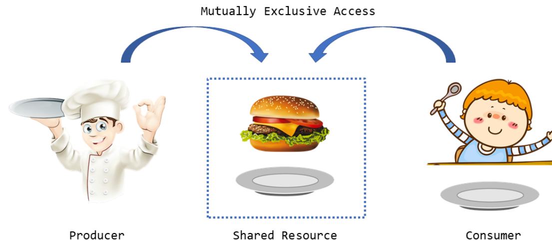
Amir H. Payberah
payberah@kth.se
2022



Processes Synchronization

Background

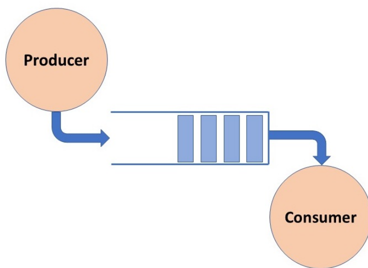
- Processes can execute concurrently.
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.



https://tinyurl.com/2yjcpx75

Producer-Consumer Problem

- The producer-consumer problem.
- Having an integer counter that keeps track of the number of full buffers.
  - Initially, counter is set to 0.
  - The producer produces a new buffer: increment the counter
  - The consumer consumes a buffer: decrement the counter



Producer

- Producer

```
while (true) {
  /* produce an item in next produced */

  while (counter == BUFFER_SIZE); /* do nothing */

  buffer[in] = next_produced;

  in = (in + 1) % BUFFER_SIZE;

  counter++;
}
```

Consumer

- Consumer

```
while (true) {
  while (counter == 0); /* do nothing */

  next_consumed = buffer[out];

  out = (out + 1) % BUFFER_SIZE;

  counter--;
  /* consume the item in next consumed */
}
```

Race Condition

- counter++ could be implemented as
  ```
  register1 = counter
  register1 = register1 + 1
  counter = register1
  ```
- counter-- could be implemented as
  ```
  register2 = counter
  register2 = register2 - 1
  counter = register2
  ```
- Consider this execution interleaving with `count = 5` initially:
  S0: producer: `register1 = counter`: `register1 = 5`
  S1: producer: `register1 = register1 + 1`: `register1 = 6`
  S2: consumer: `register2 = counter`: `register2 = 5`
  S3: consumer: `register2 = register2 - 1`: `register2 = 4`
  S4: producer: `counter = register1`: `counter = 6`
  S5: consumer: `counter = register2`: `counter = 4`

What's The Output?

```c
int counter = 0;

void* thread_func(void *arg) {
    counter++;
    printf("Job %d started.\n", counter);
    sleep(2);
    printf("Job %d finished.\n", counter);

    return NULL;
}

int main(void) {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, &thread_func, NULL);
    pthread_create(&t2, NULL, &thread_func, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

What's The Output?

```
Job 1 started.
Job 2 started.
Job 2 finished.
Job 2 finished.
```

The Critical-Section (CS) Problem

The Critical-Section Problem (1/2)

- Consider system of n processes $\{p_0, p_1, \cdots, p_{n-1}\}$.
- Each process has CS segment of code.
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in CS, no other may be in its CS.

The Critical-Section Problem (2/2)

- Each process must ask permission to enter CS in entry section, may follow CS with exit section, then remainder section.
- General structure of process P_i is below:

```
do {
    [entry section]
        critical section
    [exit section]
        remainder section
} while (true);
```

CS Problem Solution Requirements (1/3)

- Mutual Exclusion: if process P_i is executing in its CS, then no other processes can be executing in their CSs.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

CS Problem Solution Requirements (2/3)

- Progress: if no process is executing in its CS and there exist some processes that wish to enter their CS, then the selection of the processes that will enter the CS next cannot be postponed indefinitely.

```
do {
        entry section

            critical section

        exit section

            remainder section

} while (true);
```

CS Problem Solution Requirements (3/3)

- Bounded Waiting: a bound must exist on the number of times that other processes are allowed to enter their CSs after a process has made a request to enter its CS and before that request is granted.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

CS Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive.
- Preemptive: allows preemption of process when running in kernel mode.
- Non-preemptive: runs until exits kernel mode, blocks, or voluntarily yields CPU.
  - Essentially free of race conditions in kernel mode.

CS Solutions

- Peterson's solution
- Mutex lock
- Semaphore
- Monitor

Peterson's Solution

Peterson's Solution

- Two-process solution.
- The two processes share two variables:
  - `int turn`
  - `boolean flag[2]`
- `turn`: indicates whose turn it is to enter the CS.
- `flag`: indicates if a process is ready to enter the CS, i.e., `flag[i] = true` implies that process P_i is ready.

Algorithm for Process P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);
            critical section
    flag[i] = false;
            remainder section
 } while (true);
```

CS Requirements

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved:
     P_i enters CS only if: either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied.
  3. Bounded-waiting requirement is met.

Mutex Locks

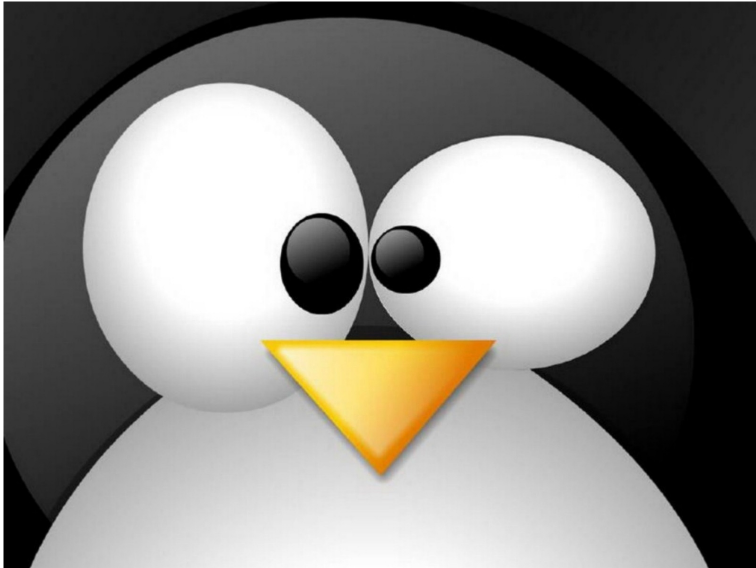
Mutex Locks

- Protect a CS by first `acquire()` a lock then `release()` the lock.
  - Boolean variable indicating if lock is available or not.
- Calls to `acquire()` and `release()` must be atomic.
  - Usually implemented via hardware atomic instructions.
- But this solution requires busy waiting.
  - This lock therefore called a spinlock.

acquire() and release()

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

```
acquire() {
  while (!available); /* busy wait */

  available = false;
}

release() {
  available = true;
}
```



pthread Mutexes

- Mutexes are represented by the `pthread_mutex_t` object.
- `pthread_mutex_lock()` locks (acquires) a pthreads mutex.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- `pthread_mutex_unlock()` unlocks (releases) a pthreads mutex.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

What's The Output?

```
int counter = 0;
pthread_mutex_t lock;

void* thread_func(void *arg) {
    pthread_mutex_lock(&lock);
    counter++;
    printf("Job %d started.\n", counter);
    sleep(2);
    printf("Job %d finished.\n", counter);
    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void) {
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL);

    pthread_create(&t1, NULL, &thread_func, NULL);
    pthread_create(&t2, NULL, &thread_func, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock);

    return 0;
}
```

What's The Output?

```
Job 1 started.
Job 1 finished.
Job 2 started.
Job 2 finished.
```

Semaphores

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore `S`: integer variable.
- Accessed via two atomic operations: `wait()` and `signal()`

wait() and signal()

```
wait(S) {
  while (S <= 0); // busy wait

  S--;
}
```

```
signal(S) {
  S++;
}
```

Counting and Binary Semaphore

- **Counting semaphore**: integer value can range over an unrestricted domain.
- **Binary semaphore**: integer value can range only between 0 and 1.
  - Same as a mutex lock.

Semaphore Usage (1/2)

- Initialize the semaphore to the number of available resources.
- Call `wait()` before using a resource.
- Call `signal()` after releasing a resource.
- If `S = 0`: all resources are used, and processes that wish to use a resource will block until the count becomes greater than 0.

Semaphore Usage (2/2)

- Consider P_1 and P_2 that require C1 to happen before C2.
- Create a semaphore S initialized to 0.

```
// Process P1
C1;
signal(S);

// Process P2
wait(S);
C2;
```

- The implementation still suffers from busy waiting.

Semaphore Implementation with no Busy Waiting (1/2)

- With each semaphore there is an associated waiting queue.
- Each entry in a waiting queue has two data items:
  - Value (of type integer).
  - Pointer to next record in the list.

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

Semaphore Implementation with no Busy Waiting (2/2)

- `block`: place the process invoking the operation on the appropriate waiting queue.
- `wakeup`: remove one of processes in the waiting queue and place it in the ready queue.

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    // add this process to S->list;
    block();
  }
}
```

```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    // remove a process P from S->list;
    wakeup(P);
  }
}
```

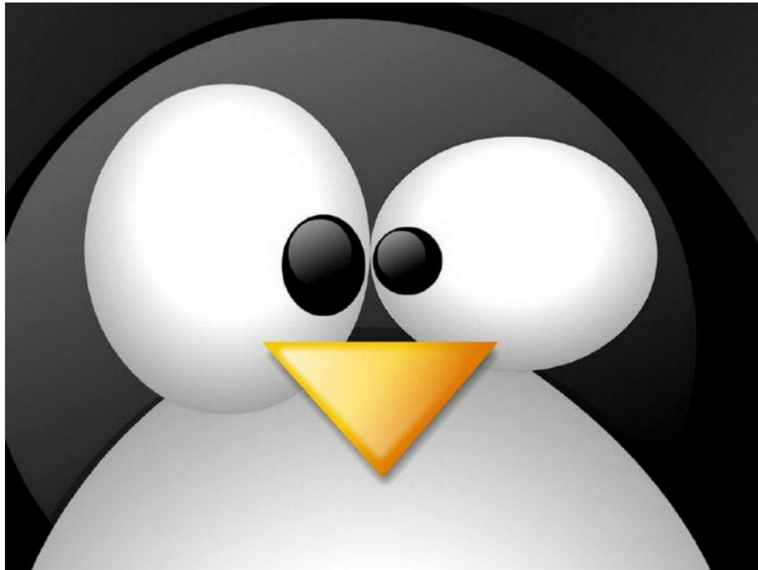
Deadlock

- Deadlock: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1.

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation

- Starvation: indefinite blocking.
- A process may never be removed from the semaphore queue in which it is suspended.
- If we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.



POSIX Semaphore

- sem_open() creates a new semaphore or opens an existing one.

```
sem_t *sem_open(const char * name , int oflag , ...);
```

- sem_wait() decrements the value of the semaphore.

```
int sem_wait(sem_t *sem);
```

- sem_post() increments the value of the semaphore.

```
int sem_post(sem_t *sem);
```

Parent-Child Example

```
void parent() {
  sem_t *sem_id = sem_open(sem_name, O_CREAT, 0600, 0);

  // The parent waits for its child to print
  sem_wait(sem_id);
  printf("Parent: Child Printed!\n");
  sem_close(sem_id);
  sem_unlink(sem_name);
}
```

```
void child() {
  sem_t *sem_id = sem_open(sem_name, O_CREAT, 0600, 0);

  printf("Child: Hello parent!\n");
  sem_post(sem_id);
}
```

Monitors

Problems with Semaphores

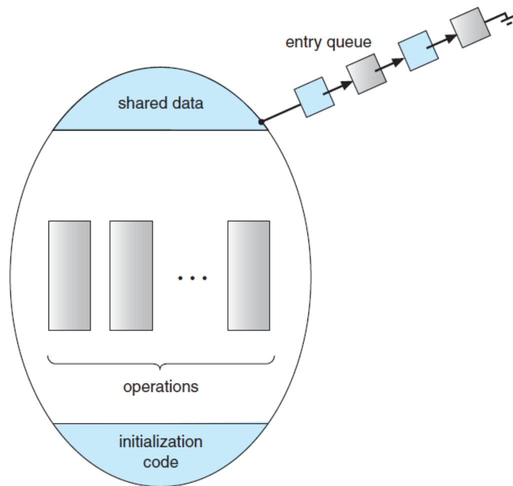
- Incorrect use of semaphore operations:
  - `signal(mutex)` ... `wait(mutex)`
  - `wait(mutex)` ... `wait(mutex)`
  - Omitting of `wait(mutex)` or `signal(mutex)` (or both)
- Deadlock and starvation are possible.

Monitors

- A high-level abstraction for process synchronization.
- Abstract data type, internal variables only accessible by code within the procedure.
- Only one process may be active within the monitor at a time.

```
monitor monitor_name {

  /* shared variable declarations */
  function P1(... ) { ... }

  function P2(...) { ... }

  function Pn(...) { ... }

  initialization code(...) { ... }
}
```

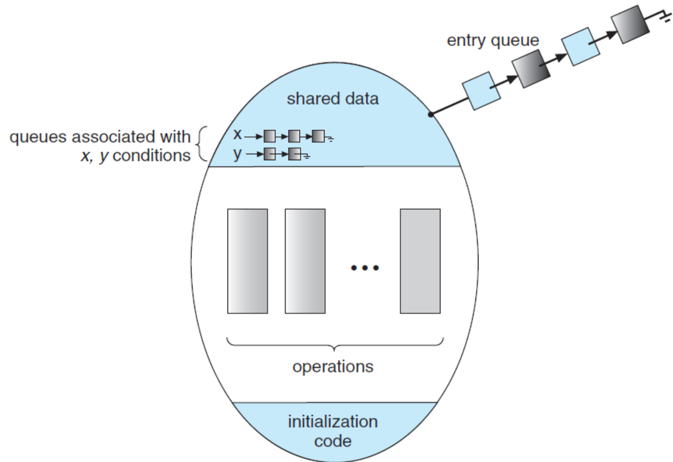
A Monitor



Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()`: a process that invokes the operation is suspended until `x.signal()`.
  - `x.signal()`: resumes one of processes (if any) that invoked `x.wait()`.
  - If no `x.wait()` on the variable, then it has no effect on the variable.

A Monitor with Condition Variables



Condition Variables Choices

- If process `P` invokes `x.signal()`, and process `Q` is suspended in `x.wait()`, what should happen next?
  - Both `Q` and `P` cannot execute in parallel. If `Q` is resumed, then `P` must wait.
- Options include:
  - Signal and wait: `P` waits until `Q` either leaves the monitor or it waits for another condition.
  - Signal and continue: `Q` waits until `P` either leaves the monitor or it waits for another condition.

Resuming Processes within a Monitor

- If several processes queued on condition `x`, and `x.signal()` executed, which should be resumed?
- FCFS (First-Come, First-Served) frequently not adequate.
- Conditional-wait construct of the form `x.wait(c)`:
  - Where `c` is priority number.
  - Process with lowest number (highest priority) is scheduled next.

Single Resource Allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource.

```
R.acquire(t);
  ...
  access the resource;
  ...
R.release();
```

- Where R is an instance of type ResourceAllocator monitor.

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator {
  boolean busy;
  condition x;

  void acquire(int time) {
    if (busy)
      x.wait(time);
    busy = true;
  }

  void release() {
    busy = false;
    x.signal();
  }

  initialization code() {
    busy = false;
  }
}
```

Readers and Writers Problem

Readers and Writers Problem (1/3)

- A shared data set among a number of concurrent processes:
  - Readers: only read the data set; they do not perform any updates.
  - Writers: can both read and write.

- Problem: allow multiple readers to read at the same time, only one single writer can access the shared data at the same time.

- Shared Data
  - Semaphore `rw_mutex` initialized to 1.
  - Semaphore `mutex` initialized to 1.
  - Integer `read_count` initialized to 0.

Readers and Writers Problem (2/3)

- The writer process.

```
do {
  wait(rw_mutex);
  ...
  /* writing is performed */
  ...
  signal(rw_mutex);
} while (true);
```

Readers and Writers Problem (3/3)

- The reader process.

```
do {
  wait(mutex);
  read_count++;
  if (read_count == 1)
    wait(rw_mutex);
  signal(mutex);
  ...
  /* reading is performed */
  ...
  wait(mutex);
  read_count--;
  if (read_count == 0)
    signal(rw_mutex);
  signal(mutex);
} while (true);
```

Summary

Summary

- Access to shared data
- The critical-section problem
- Requirements: mutual-exclusion, progress, bounding waiting
- CS solutions:
  - Peterson solution, mutex lock, semaphore, and monitor
- Reader/writer problem

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.