

# **Notes for ECE4820J**

Tianzong Cheng

# Contents

1. Operating Systems Overview .....	3
1.1. Computers and Operating Systems .....	3
1.2. Hardware .....	3
1.3. Basic concepts .....	3
2. Processes and Threads .....	3
2.1. Processes .....	4
2.2. Threads .....	4
2.3. Implementation .....	5
3. Interprocess Communication .....	5
3.1. Exhibiting the problem .....	6
3.2. Solving the Problem .....	6
3.2.1. Peterson's Algorithm .....	6
3.2.2. Mutual Exclusion at Hardware Level .....	6
3.2.3. Semaphore .....	6
3.3. More Solutions .....	8
3.3.1. Monitors .....	8
4. Scheduling .....	8
4.1. Requirements .....	9
4.2. Common Scheduling Algorithms .....	9
4.3. Notes and Problems .....	10
4.3.1. Dining Philosophers Problem .....	10

# **1. Operating Systems Overview**

## **1.1. Computers and Operating Systems**

## **1.2. Hardware**

## **1.3. Basic concepts**

## 2. Processes and Threads

### 2.1. Processes

- **Process is the unit for resource management**
- Multiprogramming issue: rate of computation of a process is not uniform / reproducible
- Process hierarchies
  - UNIX
    - parent-child
    - “process group”
  - Windows
    - All processes are equal
    - A parent has a token to control its child
    - A token can be given to another process
- A simple model for processes:
  - A process is a data structure called process control block
  - The structure contains important information such as:
    - State
      - ready (input available)
      - running (picked by scheduler)
      - blocked (waiting for input)
    - Program counter
    - Stack pointer
    - Memory allocation
    - Open files
    - Scheduling information
  - All the processes are stored in an array called process table
- Upon an interrupt the running process must be paused:
  1. Push on the stack the user program counter, PSW (program status word), etc.
  2. Load information from interrupt vector
  3. Save registers (assembly)
  4. Setup new stack (assembly)
  5. Finish up the work for the interrupt
  6. Decides which process to run next
  7. Load the new process, i.e. memory map, registers, etc. (assembly)
- Difference between concurrency and parallelism
  - concurrency: multitasking on a single-core machine
  - parallelism: multicore processor

### 2.2. Threads

- **A thread is the basic unit of CPU utilisation**
- Each thread has its own
  - thread ID
  - program counter
  - **registers**
  - **stack**

- Threads within a process share
  - code
  - data
  - OS resources
- In C or C++, if one thread crashes, it can cause the whole thread to crash.
  - For example, if one thread crashes due to invalid memory access, it can corrupt the data in the shared memory, causing other threads to crash.

## 2.3. Implementation

- POSIX threads (pthread)
  - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)`
  - `void pthread_exit(void *retval)`
  - `int pthread_join(pthread_t thread, void **retval)`
  - Release CPU to let another thread run: `int pthread_yield(void)`
  - `int pthread_attr_init(pthread_attr_t *attr)`
  - `int pthread_attr_destroy(pthread_attr_t *attr)`
- Threading models
  - Threads in user space - N:1
    - Multiple user-level threads are mapped to a single kernel thread
    - Scheduling and management are handled at the user level
    - Pros
      - Fast because context switching in kernel level is reduced
    - Cons
      - If one thread is blocked, the entire process is blocked
      - Time slices are allocated to processes, not threads. So each thread gets a smaller time slice.
  - Threads in kernel space - 1:1
    - Each user-level thread is mapped to a separate kernel thread
    - Improves responsiveness and parallelism
    - Pros
      - Blocking one thread does not affect other threads
      - Processes with several threads get more CPU time
    - Cons
      - Bigger system cost
  - Hybrid threads - M:N
    - A threading library schedules user threads on available kernel threads
    - Pros
      - Combine the strengths of user-space threads and kernel-space threads
    - Cons
      - Complexity of implementation
  - Remarks
    - Hybrid threads are less common in modern OSes

## 3. Interprocess Communication

### 3.1. Exhibiting the problem

#### Race conditions

Problems often occur when one thread does a “check-then-act”:

```
if (x == 5) { // The "Check"
    // x is modified in another thread
    y = x * 2; // The "Act"
}
```

A typical solution is:

```
// Obtain lock for x
if (x == 5) {
    y = x * 2;
}
// Release lock for x
```

### 3.2. Solving the Problem

**Critical region:** Part of the program where shared memory is accessed:

- No two processes can be in a critical region at a same time
- No assumption on the speed or number of CPUs
- No process outside a critical region can block other processes
- No process waits forever to enter a critical region

The lock should be obtained before checking and modifying the resource.

#### 3.2.1. Peterson’s Algorithm

Peterson’s algorithm is symmetric for two processes.

```
#define TRUE 1
#define FALSE 0
int turn;
int interested[2];
void enter_region(int p) {
    int other;
    other = 1 - p;
    interested[p] = TRUE;
    turn = p;
    while (turn == p && interested[other] == TRUE)
    }
void leave_region(int p) {
    interested[p] = FALSE;
}
```

#### 3.2.2. Mutual Exclusion at Hardware Level

- Disabling interrupts
- Use atomic operations
  - Test and set lock TSL

#### 3.2.3. Semaphore

A semaphore is a positive integer and is only managed by two actions:

```

sem.down() {
    while(sem==0) sleep();
    sem--;
}

sem.up() {
    sem++;
}

```

An awoken sleeping process can complete its `down` .

Checking or changing the value and sleeping are done atomically:

- Single CPU: disable interrupts
- Multiple CPUs: use `TSL` to ensure only one CPU accesses the semaphore

A mutex is a semaphore taking values 0 (unlocked) or 1 (locked).

On the request of locking a mutex, if the mutex is already locked, put the calling thread to sleep.

The implementation of a mutex using `TSL` :

```

mutex-lock:
    TSL REGISTER, MUTEX
    CMP REGISTER, #0
    JZ ok
    CALL thread_yield
    JMP mutex-lock
ok:
    RET
mutex-unlock:
    MOVE MUTEX, #0
    RET

```

Using a mutex to solve the producer-consumer problem:

```

pthread_mutex_t m;
pthread_cond_t cc, cp;
int buf = 0;

void *prod() {
    for (int i = 1; i < MAX; i++) {
        pthread_mutex_lock(&m);
        while (buf != 0)
            pthread_cond_wait(&cp, &m);
        buf = 1;
        pthread_cond_signal(&cc);
        pthread_mutex_unlock(&m);
    }
    pthread_exit(0);
}

void *cons() {
    for (int i = 1; i < MAX; i++) {
        pthread_mutex_lock(&m);
        while (buf == 0)
            pthread_cond_wait(&cc, &m);
        buf = 0;
        pthread_cond_signal(&cp);
        pthread_mutex_unlock(&m);
    }
}

```

```
pthread_exit(0);  
}
```

### 3.3. More Solutions

#### 3.3.1. Monitors

Basic idea behind monitors:

- The mutual exclusion is not handled by the programmer
- Locking occurs automatically
- Only one process can be active within a monitor at a time
- A monitor can be seen as a “special type of class”
- Processes can be blocked and awoken **based on condition variables and wait and signal functions**

Monitors are useful when several processes must complete before the next phase.



## 4. Scheduling

### 4.1. Requirements

- When to decide what process to run next
  - A new process is created
  - A process exits or blocks
  - IO interrupt from a device that has completed its task
- Compute bound v.s. input-output bound
- Two main strategies
  - Preemptive
    - A process is run for at most  $n$  ms
    - If it is not completed by the end of the period then it is suspended
  - Non-preemptive
    - A process runs until it blocks or voluntarily releases the CPU
    - It is resumed after an interrupt unless another process with higher priority is in the queue
- Goals when scheduling
  - All systems
    - Fairness: fair share of the CPU for each process
    - Balance: all parts of the system are busy
    - Policy enforcement: follow the defined policy
  - Interactive systems
    - Response time: quickly process requests
    - Proportionality: meet user's expectations
  - Batch systems
    - Throughput: maximise the number of jobs per hour
    - Turnaround time: minimise the time between submission and termination of a job
    - CPU utilisation: keep the CPU as busy as possible
  - Real-time systems
    - Meet deadlines: avoid any data loss
    - Predictability: avoid quality degradation, e.g. for multimedia

### 4.2. Common Scheduling Algorithms

- Simplest algorithm but non-preemptive:
  - CPU is assigned in the order it is requested
  - Processes are not interrupted, they can run as long as they want
  - New jobs are put at the end of the queue
  - When a process blocks the next in line is run
  - Any blocked process becoming ready is pushed to the queue
- Shortest job first (SJF)
  - non-preemptive
  - Minimizes turnaround time
- Round-Robin scheduling
  - preemptive
  - A process runs until

- Getting blocked
  - Its quantum has elapsed
  - Being completed
- Priority scheduling
  - Processes are more or less important, e.g. printing
  - Creates priority classes
  - Use Round-Robin within a class
  - Run higher priority processes first
- Lottery scheduling
  - preemptive
  - Processes get lottery tickets
  - When a scheduling decision is made a random ticket is chosen
  - Prize for the winner is to access resources
  - High priority processes get more tickets
- Earliest deadline first
  - preemptive
  - Process needs to announce (i) its presence and (ii) its deadline
  - Scheduler orders processes with respect to their deadline
  - First process in the list (earliest deadline) is run

### 4.3. Notes and Problems

- Limitations of the previous algorithms
  - They all assume that processes are competing
  - Parent could know which of its children is most important

Threads in user space is not able to run in the order of A1 B1 A2 B2 A3 B3 ( A1 A2 A3 are threads of process A ). Note that the kernel is not aware of the status of the threads in this case.

#### 4.3.1. Dining Philosophers Problem

What is the purpose of the semaphore?