Notes for ECE4820J

Tianzong Cheng

Contents

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 resuorce management
- Multiprogramming issue: rate of computatino of a process is not uniform / reproducible
- Process hierachies
  - 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)

2.2. Threads

- A thread is the basic unit of CPU utilisation
- Each thread has its own
  - thred ID

    - program counter
    - **registers**
    - **stack**
- Threads within a process share
    - code
    - data
    - OS resources

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 kenel thread
        - Scheduling and management are handled at the user level
        - If one thread is blocked, the entire process is blocked
    - Threads in kernel space - 1:1
        - Each user-level thread is mappend to a separate kernel thread
        - Improves responsiveness and parallelism
    - Hybrid threads - M:N
        - A threading library schedules user threads on available kernel threads

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 awaken 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 awaken **based on condition variables and wait and signal functions**

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