Operačné systémy

Seminár 4-5. Procesy

Ing. Martin Vojtko, PhD.

Fakulta informatiky a informačných technológií
STU v Bratislave

2020/2021

Utilizácia CPU

Majme výpočtový systém v ktorom procesy bežne trávia 80% času čakaním na I/O.

- Akú utilizáciu CPU dosiahneme ak výpočtový systém dokáže udržať v pamäti 3 procesy?
- Koľko procesov minimálne musíme vykonávať aby sme dosiahli utzilizáciu aspoň 99%
- Aké maximálne percento čakania na I/O je akceptovateľné ak 3 procesy majú vyťažiť CPU na 99%

Utilizácia CPU

Majme výpočtový systém v ktorom procesy bežne trávia 80% času čakaním na I/O.

- Akú utilizáciu CPU dosiahneme ak výpočtový systém dokáže udržať v pamäti 3 procesy?
- Koľko procesov minimálne musíme vykonávať aby sme dosiahli utzilizáciu aspoň 99%
- Aké maximálne percento čakania na I/O je akceptovateľné ak 3 procesy majú vyťažiť CPU na 99%

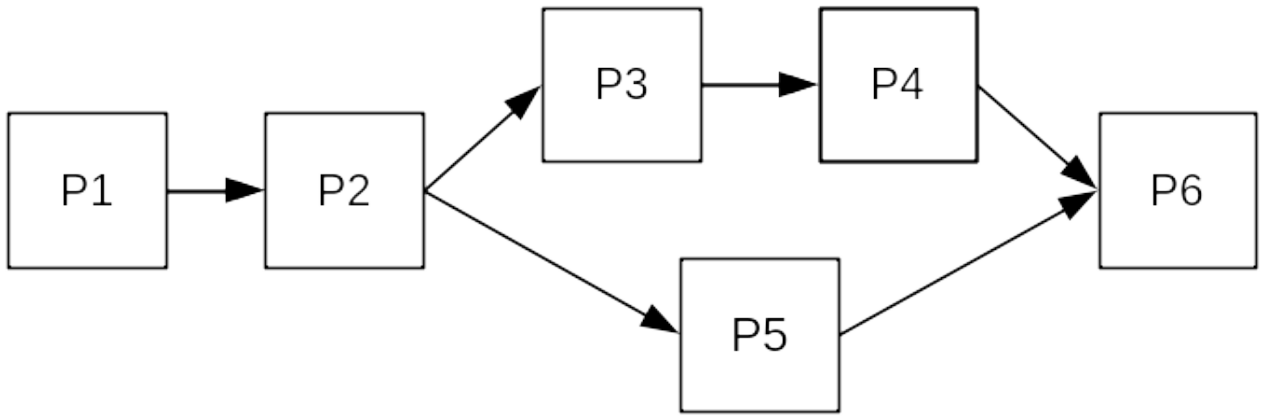
Utilizácia CPU

Majme výpočtový systém v ktorom procesy bežne trávia 80% času čakaním na I/O.

- Akú utilizáciu CPU dosiahneme ak výpočtový systém dokáže udržať v pamäti 3 procesy?
- Koľko procesov minimálne musíme vykonávať aby sme dosiahli utzilizáciu aspoň 99%
- Aké maximálne percento čakania na I/O je akceptovateľné ak 3 procesy majú vyťažiť CPU na 99%

Usporiadanie procesov

Navrhnite programy jednotlivých procesov tak aby bolo zachované stanovené poradie určené prechodovými hranami.



Mutual Exclusion - Peterson N procesov

```
#define MAX_PROC = 100;
int proc_level[MAX_PROC];  //povedzme ze kontrolujeme kolko procesov moze vzniknut
int proc_last_entered[MAX_PROC - 1];

void enter_section(int pid)
{
    for (int level = 0; level < MAX_PROC - 1; l++)
    {
        proc_level[pid] = level;
        proc_last_entered[level] = pid;
        while (proc_last_entered[level] == pid &&
               other_proces_has_higher_level(pid, level)) {;}
    }
}

bool other_proces_has_higher_level(int pid, int level)
{
    for (int oPid = 0; oPid < MAX_PROC; oPid++)
    {
        if (oPid == pid) continue;
        if (proc_level[oPid] >= level) return true;
    }
    return false;
}

void exit_section(int pid) { proc_level[pid] = -1; }
```

Mutual Exclusion - Bakery Algorithm

```
#define MAX_PROC = 100;
int ticketN = 0;         //zanedbajme ze ticket moze pretiect
int choosing[MAX_PROC]; //povedzme ze kontrolujeme kolko procesov maximalne moze vzniknut
int ticket[MAX_PROC];

void enter_section(int pid)
{
    choosing[pid] = true;
    ticket[pid] = ++ticketN; //môže sa stať, že viac procesov dostane rovnaké číslo
    choosing[pid] = false;

    for (int oPid = 0; oPid < MAX_PROC; oPid++)
    {
        while (choosing[oPid] == true)
        {;}

        while (ticket[oPid] != 0 && (ticket[oPid] < ticket[pid] ||
                                     (ticket[oPid] == ticket[pid]) && oPid < pid))
        {;}
    }
}

void exit_section(int pid)
{
    ticket[pid] = 0;
}
```

Producer-Consumer problem - s Race condition

Premenná count nie je chránená. Dochádza k strate wake-up signálov.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Producer-Consumer problem - Semafor

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE is the constant 1 */
                item = produce_item( );         /* generate something to put in buffer */
                down(&empty);                   /* decrement empty count */
                down(&mutex);                   /* enter critical region */
                insert_item(item);              /* put new item in buffer */
                up(&mutex);                     /* leave critical region */
                up(&full);                      /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item( );          /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

Producer-Consumer problem - POSIX threads

```
#include <stdio.h>
#include <pthread.h>

#define MAX 1000000000                  /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;            /* used for signaling */
int buffer = 0;                         /* buffer used between producer and consumer */

void *producer(void *ptr)               /* produce data */
{   int i;

    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i;                       /* put item in buffer */
        pthread_cond_signal(&condc);      /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr)               /* consume data */
{   int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
        while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0;                       /* take item out of buffer */
        pthread_cond_signal(&condp);      /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Producer-Consumer problem - Monitor(Java)

```
public class ProducerConsumer {
    static final int N = 100;       // constant giving the buffer size
    static producer p = new producer( );     // instantiate a new producer thread
    static consumer c = new consumer( );  // instantiate a new consumer thread
    static our_monitor mon = new our_monitor( );       // instantiate a new monitor

    public static void main(String args[ ]) {
       p.start( );       // start the producer thread
       c.start( );       // start the consumer thread
    }

    static class producer extends Thread {
       public void run( )  { // run method contains the thread code
          int item;
          while (true) {      // producer loop
             item = produce_item( );
             mon.insert(item);
          }
       }
       private int produce_item( ) { ... }      // actually produce
    }

    static class consumer extends Thread {
       public void run( )  { run method contains the thread code
          int item;
          while (true) {      // consumer loop
             item = mon.remove( );
             consume_item (item);
          }
       }
       private void consume_item(int item) { ... }// actually consume
    }

    static class our_monitor {   // this is a monitor
       private int buffer[ ] = new int[N];
       private int count = 0, lo = 0, hi = 0;   // counters and indices

       public synchronized void insert(int val) {
          if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
          buffer [hi] = val;  // insert an item into the buffer
          hi = (hi + 1) % N;        // slot to place next item in
          count = count + 1;       // one more item in the buffer now
          if (count == 1) notify( );         // if consumer was sleeping, wake it up
       }

       public synchronized int remove( ) {
          int val;
          if (count == 0) go_to_sleep( );      // if the buffer is empty, go to sleep
          val = buffer [lo];  // fetch an item from the buffer
          lo = (lo + 1) % N;        // slot to fetch next item from
          count = count – 1;       // one few items in the buffer
          if (count == N – 1) notify( );  // if producer was sleeping, wake it up
          return val;
       }
       private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

Producer-Consumer problem - Message Passing

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Dining Philosophers - Deadlock

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                       /* philosopher is thinking */
                take_fork(i);                   /* take left fork */
                take_fork((i+1) % N);           /* take right fork; % is modulo operator */
                eat( );                         /* yum-yum, spaghetti */
                put_fork(i);                    /* put left fork back on the table */
                put_fork((i+1) % N);            /* put right fork back on the table */
        }
}
```

Dining Philosophers - Semafor

```
#define N              5                    /* number of philosophers */
#define LEFT           (i+N-1)%N            /* number of i's left neighbor */
#define RIGHT          (i+1)%N              /* number of i's right neighbor */
#define THINKING       0                    /* philosopher is thinking */
#define HUNGRY         1                    /* philosopher is trying to get forks */
#define EATING         2                    /* philosopher is eating */

typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {                          /* repeat forever */
        think();                            /* philosopher is thinking */
        take_forks(i);                      /* acquire two forks or block */
        eat();                              /* yum-yum, spaghetti */
        put_forks(i);                       /* put both forks back on table */
    }
}

void take_forks(int i)                      /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                           /* enter critical region */
    state[i] = HUNGRY;                      /* record fact that philosopher i is hungry */
    test(i);                                /* try to acquire 2 forks */
    up(&mutex);                             /* exit critical region */
    down(&s[i]);                            /* block if forks were not acquired */
}

void put_forks(i)                           /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                           /* enter critical region */
    state[i] = THINKING;                    /* philosopher has finished eating */
    test(LEFT);                             /* see if left neighbor can now eat */
    test(RIGHT);                            /* see if right neighbor can now eat */
    up(&mutex);                             /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```