Konzepte von Betriebssystem-Komponenten:
Konkurrenz und Koordinierung in Manycore-Systemen

Monitor-Konzepte

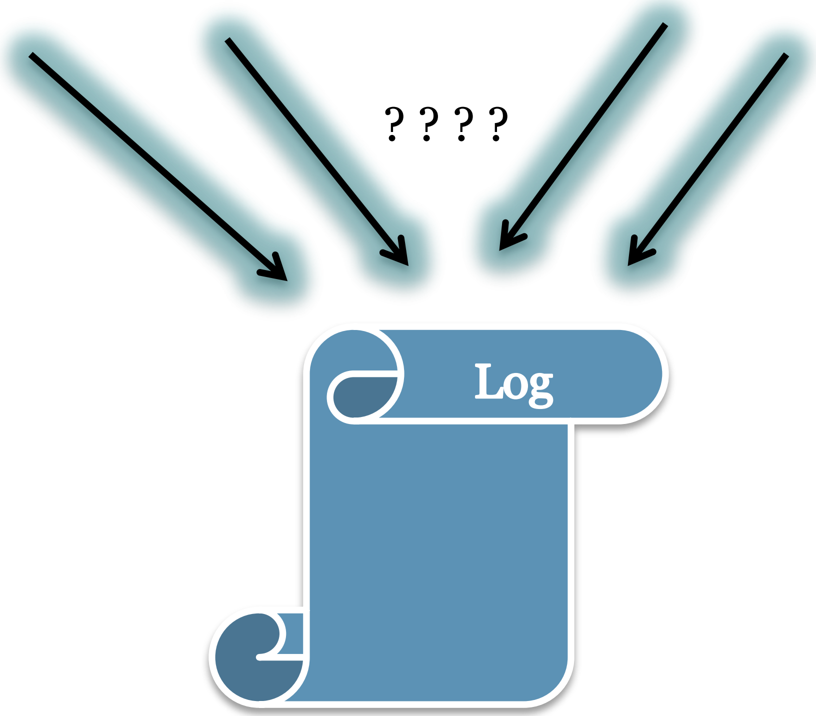
nach Brinch Hansen

18.07.2013

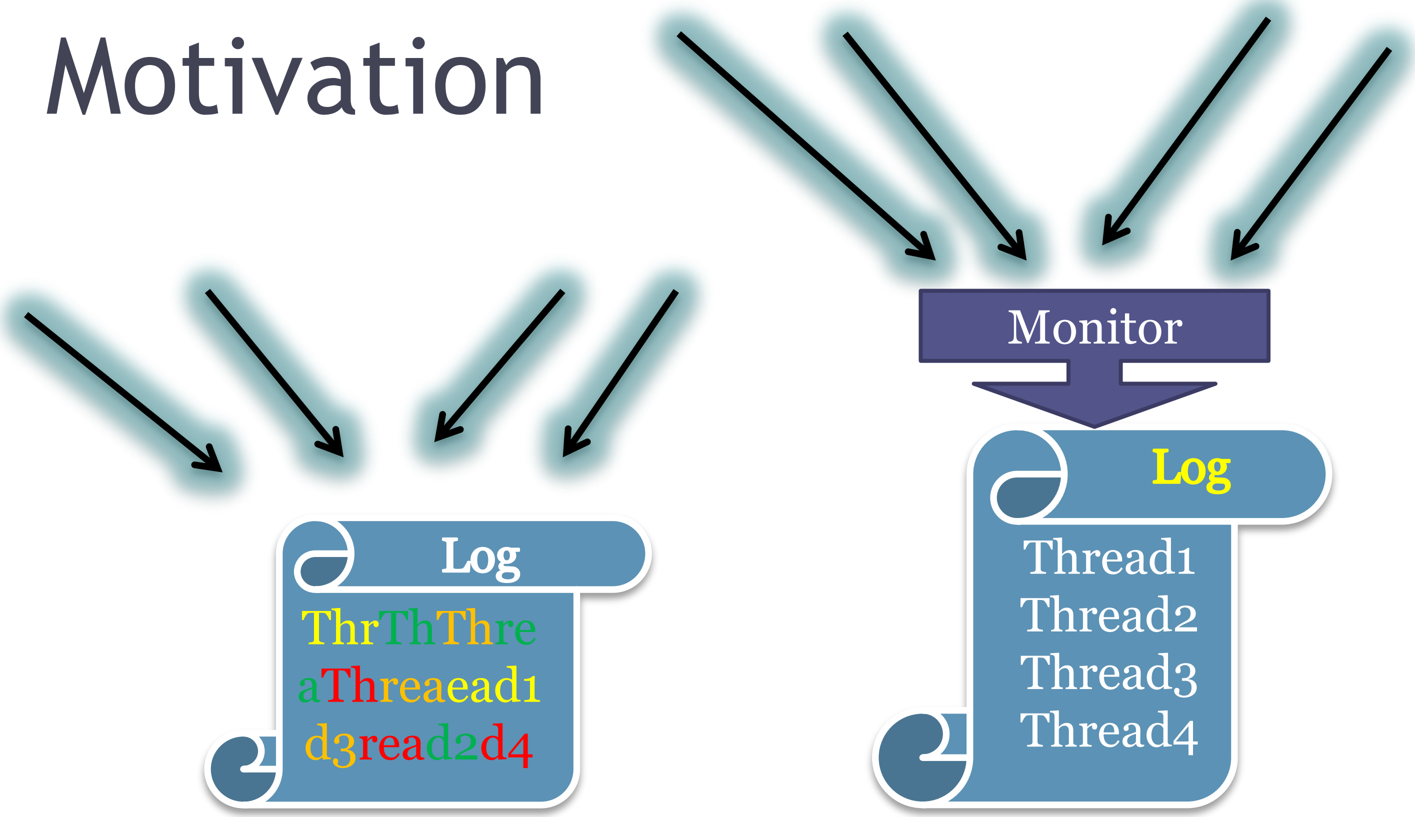
Krzysztof Mazur

Motivation

- Writing to a log file



Motivation



- writing of a log lines are not mixed with each other
- stream changes only takes place between writing of log lines

Content

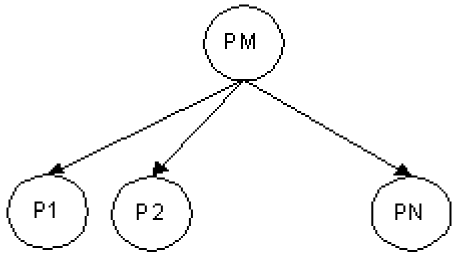
1. Introduction
2. Processes
3. Monitors
4. System design
5. Summary

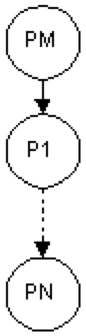
Introduction

- Aim of operating system – process management
- Resource allocation algorithms
  - Users → Programs → Processes → OS → Resources
- Concurrent programming tools
  - Processes
  - Monitors

Process component

- Each must have parent process
- Access rights
- Private data
- Sequential program
- Operating system gives resources
  - **Input** process
  - **Job** process
  - **Output** process
- Concurent processes – shared data





Processes

- Problem: data transmitted through the buffers
- Producer

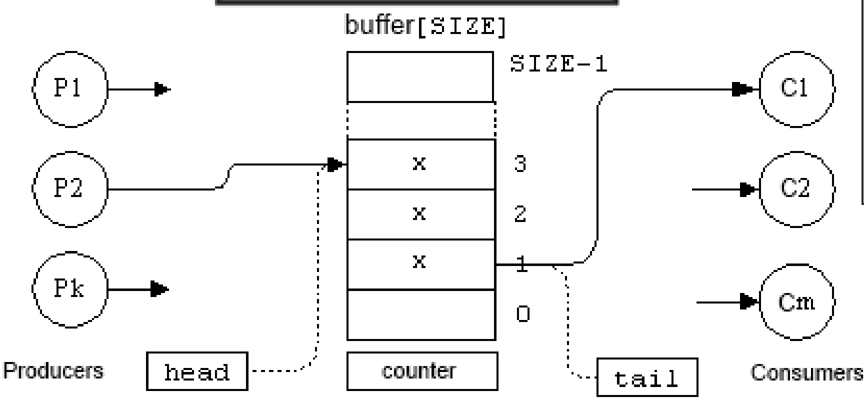
↓

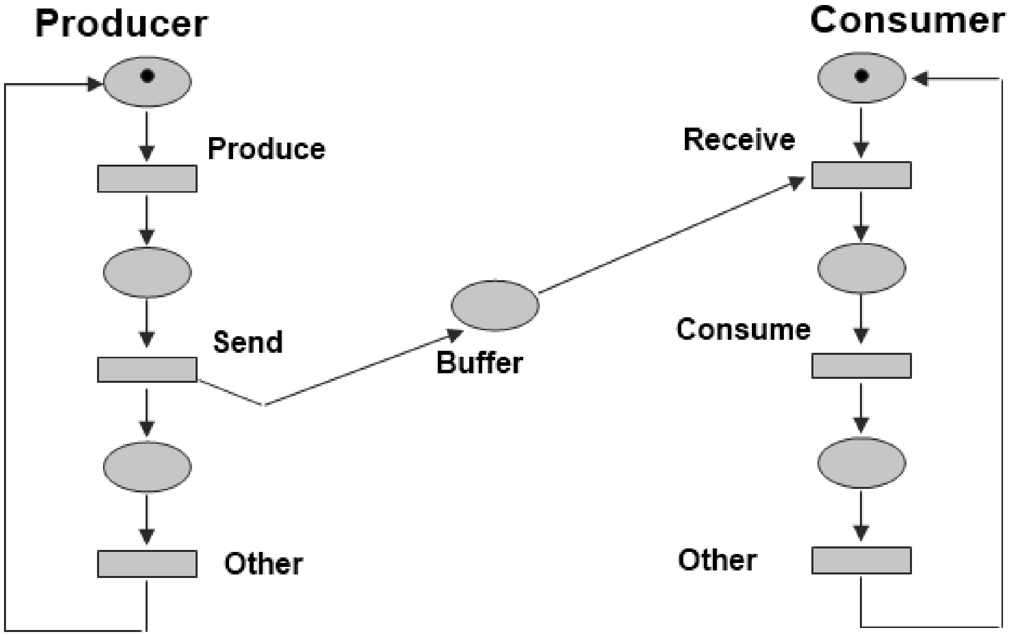
Buffer

↑

- Consumer

Buffer with limited size

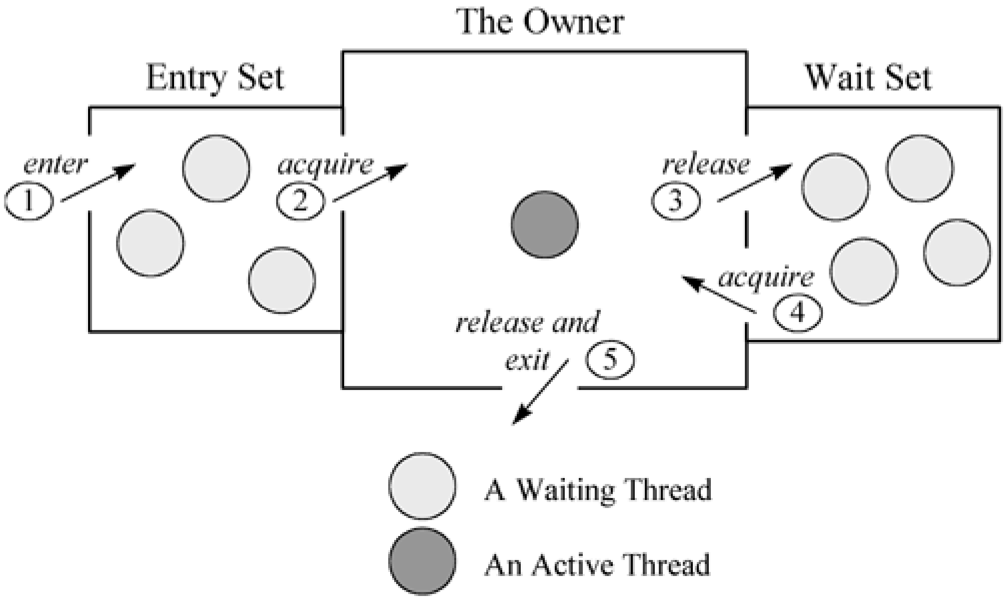




Unlimited buffer

Monitor

- Overall view



Monitors

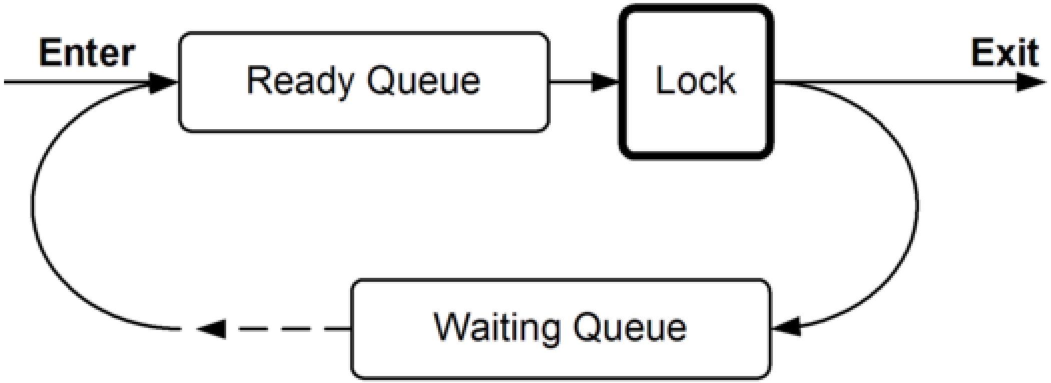
- Decide which process sends/receives data to/from buffer
- Content:
  - Access rights (Order control)
  - Shared data
  - Synchronizing operations
  - Initial operation
- Structure of data

Monitors

- Initialize once
- Variables:
  - Permanent variables – shared variables
  - Temporary variables – local variables
- Procedures:
  - Must be declared before it can be called
  - Definitions cannot be nested
  - Definitions cannot call themselves

Monitors

- Procedures
  - must be executed single
  - has exclusive access
- Hierarchization of calling the procedures
- Machine has to be able to schedule monitor calls



Monitor – producer/consumer threads

```
/* producer thread */
void *producer(void *arg)
{
  int value;

  do
  {
    value = rand() % 100;       /* produced value */
    printf("produced: %d\n", value);

    prod_cons_mon.put(value);  /* put value to the buffer */

    Sleep(rand() % 5);
  }
  while(value);

  pthread_exit(NULL);          /* end of thread */
  return NULL;
}
```

Monitor – producer/consumer threads

```
/* consumer thread */
void *consumer(void *arg)
{
  int value;

  do
  {
    /* get first value from the buffer */
    value = prod_cons_mon.get();

    printf("consumed: %d\n", value);

    Sleep(rand() % 5);
  }
  while(value);

  pthread_exit(NULL);          /* end of thread */
  return NULL;
}
```

Monitor - method put from monitor class

```
/* method puts new value to the buffer */
void monitor::put(const int value)
{
  pthread_mutex_lock(&mutex);        /* block a mutex */

  /* buffer is full - waiting for free place */
  if (nr_of_prod == SIZE)
    pthread_cond_wait(&not_full, &mutex);

  /* insert new value to the buffer */
  buffer[in] = value;
  nr_of_prod++;
  in = (in + 1) % SIZE;

  /* signal that new value appeared in the buffer */
  pthread_cond_signal(&not_empty);

  pthread_mutex_unlock(&mutex);      /* free a mutex */
}
```

Monitor - method get from monitor class

```
/* method gets first value from the buffer */
int monitor::get()
{
  pthread_mutex_lock(&mutex);         /* block a mutex */

  /* buffer is empty - waiting for products in buffer */
  if (nr_of_prod == 0)
    pthread_cond_wait(&not_empty, &mutex);

  /* get first value from the buffer */
  int value = buffer[out];
  nr_of_prod--;
  out = (out + 1) % SIZE;

  /* signal that there is a free space in the buffer */
  pthread_cond_signal(&not_full);

  pthread_mutex_unlock(&mutex);       /* release a mutex */
  return value;
}
```

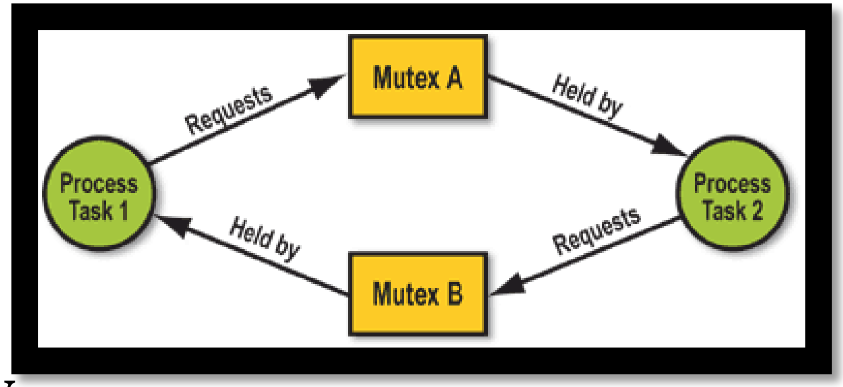
System design

- Processes and monitors (active and passive component)
  - Define data structures and operations
  - Abstract data types
- Explicit Queues – Delay and Continue
  - Wait: pthread_cond_wait(condition,mutex)

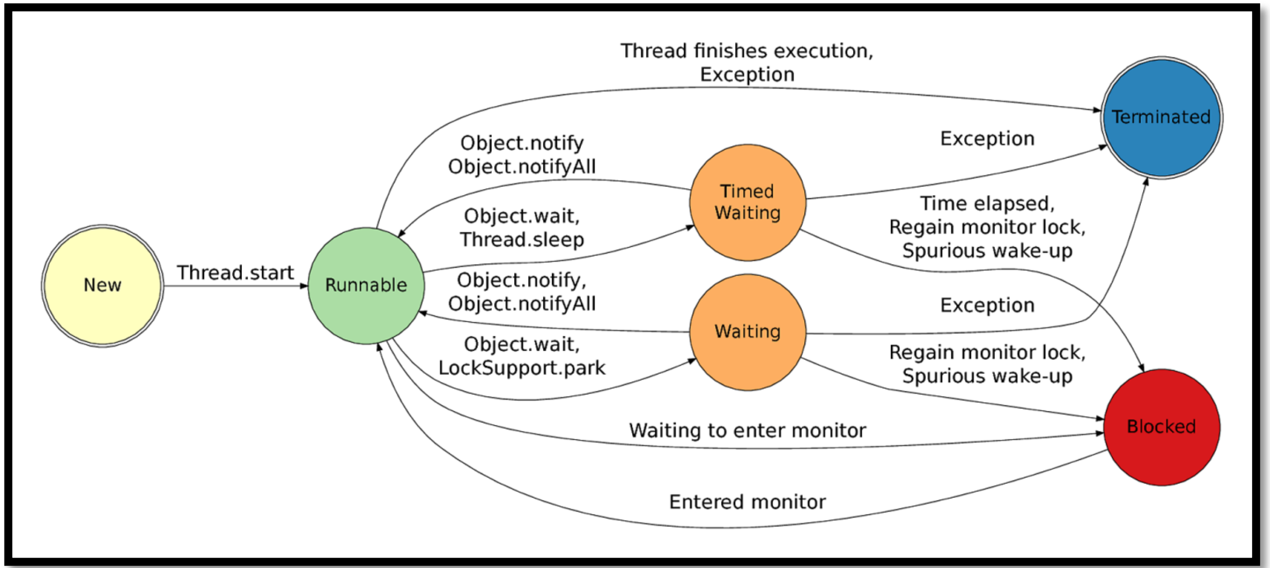
```
if (nr_of_prod == SIZE)
           pthread_cond_wait(&not_full, &mutex);
```

  - Notify/NotifyAll: pthread_cond_signal or pthread_cond_broadcast

Summary



- New dimension to programming languages: **modular concurrency**
- Using monitors communication is secure, but writing program there is possibility of deadlock
- Implementation of monitors is different in different languages, for example in Java:


New
Thread.start
Runnable
Thread finishes execution, Exception
Terminated
Object.notify Object.notifyAll
Object.wait, Thread.sleep
Timed Waiting
Exception
Time elapsed, Regain monitor lock, Spurious wake-up
Object.notify, Object.notifyAll
Object.wait, LockSupport.park
Waiting
Exception
Regain monitor lock, Spurious wake-up
Waiting to enter monitor
Blocked
Entered monitor

Monitor-Konzepte

nach Brinch Hansen