Operating system concepts

Kernel basics

OS kernel

- accessible through interrupts
- consists of data and code
- protected from user threads by memory protection and processor modes
- main responsibilities:
  - thread management (scheduling, synchronization and communication)
  - resource management (memory, UI, CPU time)

Kernel data structures

- for thread management:
  - thread descriptors:
    - thread id,
    - priority, scheduling parameters (policy, timings),
    - memory locations (stack, private thread data, …)
    - context, …
  - thread states – thread lists:
    - active thread – currently running (more on multiprocessors)
    - ready threads (usually sorted by priorities)
    - blocked threads: delayed, synchronization related, UI related
    - passive threads – threads that finished its programs or were terminated (e.g. due to an error)

Kernel data structures

- for process management:
  - process descriptors:
    - memory locations - code, data, stack(s), virtual memory data
    - resource descriptors - id’s of used system resources
      - UI devices
      - synchronization and communication mechanisms
      - file descriptors
      - …
    - owner information (user, id of parent process, …)
    - priority, scheduling parameters
    - thread list
    - …

- other resources – memory, UI, file systems, network…
  - memory locations, buffers, lists for blocked threads…

Kernel functions

- called through interrupt mechanism
- processing is performed with disabled interrupts (at least parts of it)
- typical processing scenario:
  - interrupt signal (or instruction)
  - accepting interrupt, processor behavior:
    - disable interrupts,
    - change processor operation mode,
    - save minimal context on stack,
    - jump to *interrupt processing routine*
  - *interrupt processing routine*:
    - save full context
    - determine and call required **kernel function**
    - restore context, restore thread (interrupted or other)

Kernel functions example – binary semaphore

- a simple synchronization primitive
- per semaphore data (for `Sem[id]`):
  - `value` – current value: zero or one
  - `queue` – queue for blocked threads

```
k-function BSemWait(id)
{
  if (Sem[id].value == 1) {
     Sem[id].value = 0;
  }
  else {
     Enqueue(ActiveThread, Sem[id].queue);
     ActiveThread = GetFirst(ReadyQueue);
  }
}
```

Kernel functions example – binary semaphore

```
k-function BSemSignal(id)
{
  if (Sem[id].queue is not empty) {
     Enqueue(ActiveThread, ReadyQueue);
     first = GetFirst(Sem[id].queue);
     Enqueue(first, ReadyQueue);
     ActiveThread = GetFirst(ReadyQueue);
  }
  else {
     Sem[id].value = 1;
  }
}
```

- Only basic functionality is presented! More on this later…

Kernel functions

- Most of kernel functions may use the same principles as shown on previous example
  - synchronization functions
  - time management
  - UI, …

Multiprocessor kernel support ?

- kernel data must reside in shared memory space
- critical section can’t be secured by disabling interrupts (calling through interrupt is not enough)

- Test and Set (or similar) instruction is used in ***spinlock***

```
spinlock: TAS lock_id, reg;
          if reg == 1 then goto spinlock;
```

- TAS uses two consecutive bus cycles to:
  - read given memory location into register in first cycle
  - store value 1 in same location in second cycle

- “busy waiting” is unavoidable in multiprocessor systems

Multiprocessor kernel extension example

```
k-function BSemSignal(id) {
  klock: TAS kernel_lock, reg;
         if reg == 1 then goto klock;
  if (Sem[id].queue is not empty) {
     Enqueue(ActiveThread[P], ReadyQueue);
     first = GetFirst(Sem[id].queue);
     Enqueue(first, ReadyQueue);
     ActiveThread[P] = GetFirst(ReadyQueue);
  }
  else {
     Sem[id].value = 1;
  }
  kernel_lock = 0;
}
```

Kernel practices

- ready threads are placed into multilevel queues, one level for each priority – higher priority threads are scheduled first
- in (today) multiprocessors, ready threads are allocated per processor (not in single ready queue/structure)
  - performance related decision – maximize cache usage
    - “hot-cache” objective – returning thread may find some of its data still in processor cache
  - balancing issue – if ready queues over multiple processors are not balanced, scheduling would not be fair!
- kernel overhead
  - switching tasks (saving/restoring context)
  - processor operation mode switch (not insignificant!)

Thread management

Synchronization

Need for synchronization?

- Many tasks – few resources
  - only limited number of tasks may use available resources at the same time
  - in most cases, “limited number” equals one!
    - only a single task may use a resource at a time, ALL other tasks must wait (be blocked)!

- Single task with multiple threads
  - threads share common objects →using a shared object is a critical operation, must be performed sequentially
  - threads cooperate on single operation – might require synchronization (e.g. when dividing work between them)
  - “pipe-line” synchronization
    - results from first task are input for next
  - …

Available synchronization through OS

- Most effective synchronization is through OS interface
  - others require spinlocks!

- Critical section (CS), mutual exclusion synchronization
  - Disable/enable interrupts! (on single processor systems)
  - Binary semaphore
  - Mutex (CS object)

- Counter type synchronization (number of resources $\geq$ 1)
  - Semaphores (general)

- Complex synchronizations
  - Semaphores (more than one!)
  - Monitors (mutex + conditional variables)

Disabling/enabling interrupts

- Disabling and enabling interrupt is privileged operation
  - requires that program runs on high privilege level
- Must be used VERY carefully:
  - **blocking in critical section protected by disabled interrupts stops everything (system deadlock)!**
- Very simple, very effective when used appropriately
  - appropriate use: only for very short critical sections
- Mostly used only in:
  - **kernel**
  - **embedded systems** (and RT systems)

Disabling/enabling interrupts – example

```
.
.  (non-critical section)
.
disable_interrupt();

CRITICAL_SECTION;
```

(only one thread may be here)

```
enable_interrupt();
.
.  (non-critical section)
.
```

Binary semaphore – basic operations

- **`BSemWait(s_id)`**
  - *synonyms*: acquire, lock
  - *operation*: lock semaphore object identified with **`s_id`**
    - locks only this object!
    - *programmers view*: locking a semaphore gives access to a single resource (semaphore ⇔ resource)
    - not a global lock(like with disabling interrupts!)
    - if the semaphore is already locked (owned by other thread):
      - calling thread is blocked – put in queue associated with semaphore
- **`BSemSignal(s_id)`**
  - *synonyms*: release, unlock, post
  - *operation*: release semaphore object
    - if semaphore queue is not empty (threads are waiting): assign semaphore to first thread in queue – release thread form queue (move it to ready thread queue)
    - otherwise (empty queue): mark semaphore as free (*signaled*)

Binary semaphore – CS example

```
.
.  (non-critical section)
.
BSemWait(s1);

CRITICAL_SECTION;
```
(only one thread may be here)

```
BSemSignal(s1);
.
.  (non-critical section)
.
```

Binary semaphore – forcing alternation

- Except for crit. sect. binary semaphore can be used for synchronization where two (or more) threads must alternate through their crit. sect.

```
Thread I:

while(1) {
  BSemWait(s1);

  thread_I_turn();

  BSemSignal(s2);
}
```

```
Thread J:

while(1) {
  BSemWait(s2);

  thread_J_turn();

  BSemSignal(s1);
}
```

- Initially only one semaphore (**s1** or **s2**) must be set (in *signaled* state)

Semaphore (general)

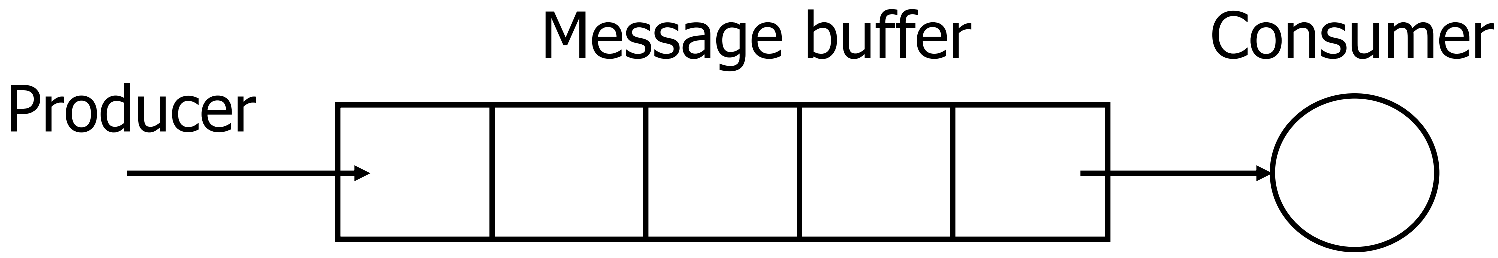
- Semaphore is used for counting available resources
  - e.g. numbers of messages in queue, list elements, …
- Semaphore *value*:
  - if *value* = 0, then semaphore is in *non-signaled state*
    - will block all threads that require resource it protect (threads will be put in queue)
  - if *value* > 0, then semaphore is in *signaled state*
    - at least one thread will pass over semaphore without blocking

- E.g. a *consumer* thread processes messages from buffer which is protected with counting semaphore **`sb`**:

```
…
SemWait(sb);            //blocks thread if buffer is empty
(get next message from buffer)
…
```

Semaphore example: producer/consumer

- Producer/consumer problem demonstrate usage of semaphores when producer and consumer communicate through buffer with size N (in messages).
- Producer *produce* messages and puts them into queue
- Consumer reads messages form buffer and *consumes* them
- Producer must be blocked if message buffer is full!
- Consumer must be blocked if message buffer is empty!



Semaphore example: producer/consumer

```
Producer:

while(1) {
  P = produce();
  SemWait(s_empty);
  PutIntoBuffer(P);
  SemSignal(s_full);
}
```

```
Consumer:

while(1) {
  SemWait(s_full);
  R = GetFromBuffer();
  SemSignal(s_empty);
  consume(R);
}
```

- Initial value of semaphores: `s_empty=N; s_full=0;`

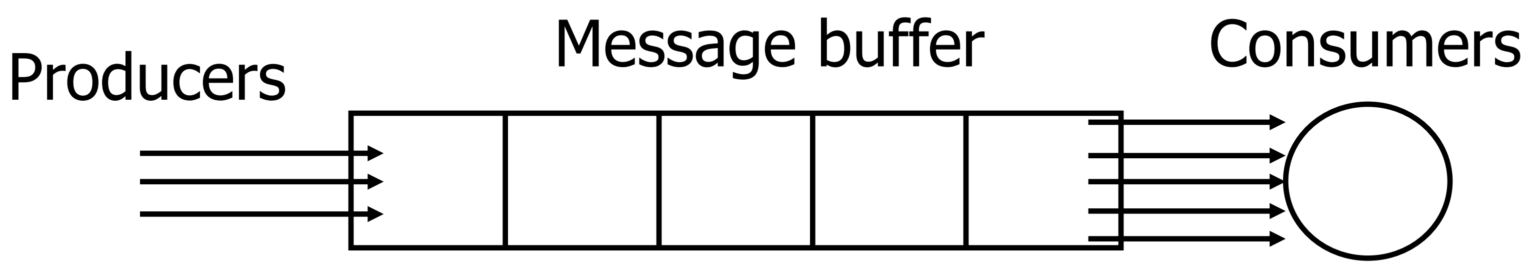
Semaphore problems

- Semaphores are the most used mechanisms for simple synchronizations:
  - supported by all OS-es (some even with more interfaces!)
  - simple semantic and usage

- If the problem is not simple, **more than one** semaphore is required
  - if more than one resource is needed – more semaphores must be acquired simultaneously
  - semantic for such synchronization is not obvious – coding is very difficult
  - more semaphores – greater the chance for deadlock!

Producers and consumers

- If same example from before were extended with more producers and consumers
  - producers must not simultaneously put message in buffer
    - buffer manipulation require additional variables
    - some messages may be overwritten
    - additional semaphore is required (will function as binary)
  - similar problems with consumers
    - additional semaphore is required



Producers/consumers – wrong solution

- The same binary semaphore **s_buffer** is used for buffer protection both for producers and consumers, initialized to 1

```
Producers:

while(1) {
   P = produce();

   SemWait(s_buffer);

   SemWait(s_empty);
   putintobuffer(P);
   SemSignal(s_full);

   SemSignal(s_buffer);
}
```

```
Consumers:

while(1) {
   SemWait(s_full);

   SemWait(s_buffer);
   R = getfrombuffer();
   SemSignal(s_buffer);

   SemSignal(s_empty);

   consume(R);
}
```

- When buffer becomes full, next producer will block on **s_empty**, while holding lock on **s_buffer**: **deadlock**!

Deadlock – typical scenario

- Two (or more) threads, two (or more) resources

```
Thread I:

   ...
   SemWait(s1);
   ...
   SemWait(s2);
   ...
   ...
   SemSignal(s1);
   ...
   SemSignal(s2);
   ...
```

```
Thread J:

   ...
   ...
   SemWait(s2);
   ...
   SemWait(s1);
   ...
   SemSignal(s1);
   ...
   SemSignal(s2);
   ...
```

DEADLOCK!

Deadlock – possible prevention

- Some operating systems have interfaces that can perform multiple operations on multiple semaphores as an atomic operation – if any one operation cannot be performed, none are performed

- example (UNIX*):
  - **`semop (id, array_of_op, number_of_op);`**

- with this interface all resources can be obtained at once or none will be reserved and the thread is blocked

- use of other synchronization mechanisms
  - **monitors** (or equivalent)

Monitors

- operate on sensitive data (shared data/resources) in a controlled environment – in “monitor functions”
- monitor functions are critical sections where:
  - only one thread can be running (active or in ready state)
  - thread can perform critical operations
  - thread can *check for resource availability* – in user space, using adequate data structures
    - if resources are available – take them and continue,
    - if resources are not available – block thread and “virtually” leave monitor function
  - thread can release resources
    - if threads are waiting for them, release the first thread (or all)
      - released threads must acquire lock on monitor before continuing (otherwise more than one function may be active in monitor!)

Monitors

- monitor may be supported implicitly by programming language (i.e. keyword *synchronized* in Java)

- the interface must include:
  - a mechanism for protected *monitor entrance*
  - a mechanism for *leaving the monitor* (and releasing the thread waiting on entrance)
  - a mechanism for *blocking the thread inside monitor* and temporarily releasing the monitor
  - a mechanism for *releasing blocked thread inside monitor*

- in most environments monitors are implemented with:
  - *mutexes* (from: *mutual exclusion object*) and
  - *conditional variables*

Mutex

- Mutex is very similar to binary semaphore
- But binary semaphore
  - is rarely offered through OS interface
  - is only a concept, realized through other sync. funct.
    - general semaphore (and careful initialization and usage)
    - mutex

- Mutex interface:
  - **`MutexLock(m_id)`** (synonyms: *acquire, enter*)
  - **`MutexUnlock(m_id)`** (synonyms: *release, leave*)

- Difference with binary semaphore:
  - designed **only for critical section** synchronization
  - extra functionality when used with *conditional variables*: ***monitors***

Conditional variables

- Sometimes there is a need for a mechanism which will *just* put a thread to queue
  - thread may find that conditions for its continuing execution are inappropriate (e.g. through checking state variables) and therefore ask to be blocked, put in particular queue
  - only when conditions are improved, at some point in future, thread should be unblocked (by the thread that changed conditions)
  - since checking and changing conditions through shared objects is critical operation, it should usually be done in critical section
    - but, blocking thread in critical section is one step from deadlock!
    - blocking must be accompanied with temporary release of critical section object (if its acquired)
- The described mechanism is called a *conditional variable*

Conditional variables

- Conditional variables *may* be used without (companion) critical section objects, but its potential is fully valuable when used with *mutexes*.

- Conditional variables interface:
  - **`CondWait(cond_id, m_id)`** (synonym: *wait*)
    - put thread in queue *and release mutex object*
  - **`CondSignal(cond_id)`** (synonym: *signal*)
    - release first thread form queue
  - **`CondSignalAll(Cond_id)`** (synonym: *broadcast*)
    - release all threads form queue

Typical monitor usage scenario – acquire

```
m-function get_resources()
{
  MutexLock(m_id)

  while (not all resources are available)  //not "if"
      CondWait(cond_id, m_id);

  mark resources as used - give them to thread;
  (or just use resources here, inside monitor)

  MutexUnlock(m_id);
}
```

Complex condition

Typical monitor usage scenario - release

```
m-function release_resources()
{
  MutexLock(m_id)

  mark resources as free;

  if (threads are waiting for resources)
      CondSignal(cond_id);
```

(signaling can be done even without checking if some threads are waiting, or even with `CondSignalAll(cond_id)`, if “Acquire” function is made with “`while`” instead of “`if`”)

```
  MutexUnlock(m_id);
}
```

Monitor example - messages

- Monitors may be used for simple and complex synchronization problems
- Because of “clear” synchronization objectives (given through explicit state variables checking), **monitors are preferred synchronization primitives**

- Example “problem and environment”
  - In an example system, several threads wait on messages
  - All messages come through the same channel and should be forwarded to the appropriate thread
  - Forwarding is not performed explicitly – threads are activated to check if the message belongs to either of them
  - A single message is intended only for one thread (other threads don’t have interest in it)

Monitor example - messages

- Threads:
  - Delivery thread
    - waits on device – source of the message
    - when message arrives, wakes processing threads
  - Processing threads
    - each thread waits for particular message type
    - upon examining the message header thread will take it or leave it (for the next thread)
- Threads are *cyclic*; they repeat their operations until end is signaled with **`job_not_finished`** function (or variable)

Monitor example - messages

- Data structure:
  - monitor:
    - mutex **`m1`** (monitor function guard)

    - conditional variables **`c1`** and **`c2`**
      - **`c1`** – queue for threads that are waiting on message to be delivered (by delivery thread)

      - **`c2`** – queue for delivery thread which is waiting for signal that the message has been taken

  - **`msg_assigned`** – shared variable (global) that shows if last arrived message is taken by some thread or not yet
    - if “false” threads will inspect last message contents

Monitor example – handling messages

```
Delivery_thread ()
{
  msg_assigned = true; // shared variable!

  while (job_not_finished()) {
      wait_for_message;

      MutexLock(m1);
      msg_assigned = false;
      CondSignalAll(c1);
      // wait till some thread gets message
      while (msg_assigned == false) // will work even without
            CondWait(c2, m1);
      MutexUnlock(m1);
  }
}
```

Monitor example – handling messages

Complex condition

```
Thread_I()
{
  MutexLock(m1);
  while (job_not_finished()) {
     if (msg_assigned == false &&
        (received message belongs to thread I) ) {
           take_message();
           msg_assigned = true;
           CondSignal(c2);
           MutexUnlock(m1);
           process_message();
           MutexLock(m1);
     } else
           CondWait(c1, m1);
  }
  MutexUnlock(mfm);
}
```

Same problem with semaphores?

```
Delivery_thread ()
{
  msg_assigned = true; // shared variable!

  while (job_not_finished()) {
      wait_for_message;

      msg_assigned = false;
      for i = 1 to number_of_threads
            SemSignal(s1);
      // or SemSignal(s1, number_of_threads); if supported

      SemWait(s2);
  }
}
```

Same problem with semaphores?

```
Thread_I()
{
  while (job_not_finished()) {
      if (msg_assigned == false &&
         (received message belongs to thread I) ) {
            take_message();
            msg_assigned = true;
            SemSignal(s2);
            process_message();
      } else
            SemWait(s1);
  }
}
```

- Problems: many; Solutions: many; **Good solutions**?
  - try: assign separate semaphore to each thread (instead of `s1`)
  - “thinking like in monitors” might sometimes work even with semaphores

Other examples with monitors

- Dining philosophers
- Several problems:
http://www.cs.berkeley.edu/~kubitron/courses/cs162-F06/hand-outs/synch-problems.html
- and solutions:
http://www.cs.berkeley.edu/~kubitron/courses/cs162-F06/hand-outs/synch-solutions.html
- More on synchronization:
http://www.zemris.fer.hr/~leonardo/unofficial/radovi/Sinkronizacija_MIPRO07.pdf (in Croatian)