Outline for Today

• Objective: To define process and thread. Introduce the critical section problem. Reason about correctness of concurrent programs.
• Administrative details:
  – Groups are listed on the web site: groups.html
  – Thursday meetings: what about time slot after this?
  – In the Makefile.common, in the top level of the code you have installed, look for these two lines:
    \texttt{CC = g++}
    \texttt{LD = g++}
  Change them to:
    \texttt{CC = /usr/pkg/gcc-2.7.2/sun4m_55/bin/g++}
    \texttt{LD = /usr/pkg/gcc-2.7.2/sun4m_55/bin/g++}

The Basics of Processes

• Processes are the OS-provided abstraction of multiple tasks (including user programs) executing concurrently.
• One instance of a program (which is only a passive set of bits) in execution (implying an execution context – register state, memory resources, etc.)
• OS schedules processes to share CPU.

Textbook Readings

This week
• Chapter 6 - Process Mgt
• Chapter 8 - Basic Synchronization
• Chapter 9 - High-level Synchronization
Next week
• Chapter 7 - Scheduling
• Chapter 10 - Deadlock

Why Use Processes?

• To capture naturally concurrent activities within the structure of the programmed system.
• To gain speedup by overlapping activities or exploiting parallel hardware.
  – From DMA to multiprocessors
**Context Switching**

- When a process is running, its program counter, register values, stack pointer, etc. are contained in the hardware registers of the CPU. The process has direct control of the CPU hardware for now.
- When a process is not the one currently running, its current register values are saved in a process descriptor data structure (PCB - process control block).
- Context switching involves moving state between CPU and various processes’ PCBs by the OS.

**Process Abstraction**

- Unit of scheduling
- One (or more*) sequential threads of control
- program counter, register values, call stack
- Unit of resource allocation
- address space (code and data), open files
- sometimes called tasks or jobs
- Operations on Processes: fork (clone-style creation), wait (parent on child), exit (self-termination), signal, kill.

Process-related System Calls.

---

**Process State Transitions**

- Ready
- Create
- Process
- Wakeup (due to interrupt)
- Blocked
- suspended
- while another process scheduled
- Running
- scheduled
- sleep (due to outstanding request of syscall)
- Done

---

**Interleaved Schedules**

- Initial concept / multiprocessor implementation
- Uni-processor implementation
- Interpreted process
- Context switch
Separation of Policy and Mechanism

- “Why and What” vs. “How”
- Objectives and strategies vs. data structures, hardware and software implementation issues.
- Process abstraction vs. Process machinery

Can you think of examples?

Process Mechanisms

- PCB data structure in kernel memory represents a process (allocated on process creation, deallocated on termination).
- PCBs reside on various state queues (including a different queue for each “cause” of waiting) reflecting the process’s state.
- As a process executes, the OS moves its PCB from queue to queue (e.g. from the “waiting on I/O” queue to the “ready to run” queue).

PCBs & Queues

Threads and Processes

- Decouple the resource allocation aspect from the control aspect
- Thread abstraction - defines a single sequential instruction stream (PC, stack, register values)
- Process - the resource context serving as a “container” for one or more threads (shared address space)
- Kernel threads - unit of scheduling (kernel-supported thread operations -> still slow)
### Threads and Processes

- **Thread**
- **Address Space**

### User-Level Threads
- To avoid the performance penalty of kernel-supported threads, implement at user level and manage by a run-time system:
  - Contained “within” a single kernel entity (process)
  - Invisible to OS (OS schedules their container, not being aware of the threads themselves or their states). Poor scheduling decisions possible.
- User-level thread operations can be 100x faster than kernel thread operations, but need better integration / cooperation with OS.

### An Example

- Editing thread: Responding to your typing in your doc
- Autosave thread: periodically writes your doc file to disk

### Nachos

- User Programs
  - Syscalls
    - OS
    - Kernel
  - Machine instructions
    - MIPS HW
  - MIPS instr
  - SPARC HW
A Nachos Thread

```cpp
// Nachos Thread
Thread* t = new Thread(name);
t->Fork(MyFunc, arg);
currentThread->Sleep();
currentThread->Yield();
```

Thread Operations

- **new thread**: initiates a thread control block
- **Thread::Fork**: runs a specified procedure in a newly created thread (allocates stack and makes ready to run)
- **Thread::Finish**: cleans up its state
- **Thread::Yield**: gives up CPU to another thread ready to run and invokes scheduler to choose new running thread
- **Thread::Sleep**: blocks thread (not on ready queue)

Nachos Context Switches: Voluntary vs. Involuntary

On a uniprocessor, the set of possible execution schedules depends on *when context switches can occur*.

- **Voluntary**: one thread explicitly yields the CPU to another.
  - A Nachos thread can suspend itself with `Thread::Yield`.
  - It may also block to wait for some event with `Thread::Sleep`.
- **Involuntary**: the system scheduler suspends an active thread, and switches control to a different thread.
  - Thread scheduler tries to share CPU fairly by timeslicing.
  - Suspend/resume from a timer interrupt handler (e.g., `nachos -rs`)
  - This can happen "any time", so concurrency races can occur.
Blocking or Sleeping

- An executing thread may request some resource or action that causes it to block or sleep awaiting some event.
  - passage of a specific amount of time (a pause request)
  - completion of I/O to a slow device (e.g., keyboard or disk)
  - release of some needed resource (e.g., memory)
- In Nachos, threads block by calling `Thread::Sleep`.
- A sleeping thread cannot run until the event occurs.
- The blocked thread is awakened when the event occurs.
  - E.g., `Wakeup` or `NachosScheduler::ReadyToRun(Thread* t)`
- In an OS, processes may sleep while executing in the kernel to handle a system call or fault.

Nachos Thread State Transitions

The core of Nachos is the `Scheduler` class:
- one global shared scheduler object
- pool of ready threads (the ready list)

```c
new = scheduler->FindNextToRun(); /* get next ready thread */
scheduler->Run(t); /* run it */
```

Run calls `SWITCH(currentThread, new)` to suspend current thread and pass control to new thread.
/* Save context of the calling thread (old), restore registers of the next thread to run (new), and return in context of new. */

switch/MIPS (old, new) {
old->stackTop = SP;
save RA in old->MachineState[PC];
save callee registers in old->MachineState
restore callee registers from new->MachineState
RA = new->MachineState[PC];
SP = new->stackTop;
return (to RA)
}

A Nachos Context Switch

Caller-saved registers (if needed) are already saved on the thread's stack.
Caller-saved registers restored automatically on return.
Return to last procedure that called switch in new.

The Trouble with Threads...

while (i<10) { x = x + 1; i++;}

What is the value of x when both threads leave this while loop?

Nondeterminism

- What unit of work can be performed without interruption? **Indivisible** or **atomic** operations.

- **Interleavings** - possible execution sequences of operations drawn from all threads.

- **Race condition** - final results depend on ordering and may not be "correct".

Reasoning about Interleavings

- On a uniprocessor, the possible execution sequences depend on when context switches can occur:
  - Voluntary context switch - the process or thread explicitly yields the CPU (blocking on a system call it makes, invoking a yield operation).
  - Interrupts or exceptions occurring - an asynchronous handler activated that disrupts the execution flow.
  - Preemptive scheduling - a timer interrupt may cause an involuntary context switch at any point in the code.

- On multiprocessors, the ordering of operations on shared memory locations is the important factor.
Critical Sections

• If a sequence of non-atomic operations must be executed as if it were atomic in order to be correct, then we need to provide a way to constrain the possible interleavings in this critical section of our code.
  – Critical sections are code sequences that contribute to “bad” race conditions.
  – Synchronization needed around such critical sections.
• Mutual Exclusion - goal is to ensure that critical sections execute atomically w.r.t. related critical sections in other threads or processes.
  – How?

Implementation Options for Mutual Exclusion

• Disable Interrupts
• Busywaiting solutions - spinlocks
  – execute a tight loop if critical section is busy
  – benefits from specialized atomic (read-mod-write) instructions
• Blocking synchronization
  – sleep (enqueued on wait queue) while C.S. is busy
Synchronization primitives (abstractions, such as locks) which are provided by a system may be implemented with some combination of these techniques.

The Critical Section Problem

Each process follows this template:
while (1)
{
  ...other stuff... //processes in here shouldn’t stop others
  enter_region();
  critical section
  exit_region();
}
The problem is to define enter_region and exit_region to ensure mutual exclusion with some degree of fairness.
Proposed Algorithm for 2 Process Mutual Exclusion

```c
Boolean flag[2];
proc (int i) {
    while (TRUE){
        compute;
        flag[i] = TRUE;
        while(flag[(i+1) mod 2]) ;
        critical section;
        flag[i] = FALSE;
    }
}
```

Is it correct?

Flag[0] = Flag[1] = FALSE;
fork (proc, 1, 0);
fork (proc, 1, 1);
Is it correct?

Assume they go lockstep. Both set their own flag to TRUE. Both busywait forever on the other's flag -> deadlock.

Is it correct?

Proposed Algorithm for 2 Process Mutual Exclusion

- enter_region:
  needin [me] = true;
  turn = you;
  while (needin [you] && turn == you) {no_op};
- exit_region:
  needin [me] = false;

Is it correct?

Interleaving of Execution of 2 Threads (blue and green)

```c
enter_region:
    needin [me] = true;
    turn = you;
    while (needin [you] && turn == you) {no_op};
Critical Section
exit_region:
    needin [me] = false;
```

```c
enter_region:
    needin [me] = true;
    turn = you;
    while (needin [you] && turn == you) {no_op};
Critical Section
exit_region:
    needin [me] = false;
```

needin [blue] = true;
needin [green] = true;
turn = green;
turn = blue;
while (needin [green] && turn == green)
    Critical Section
while (needin [blue] && turn == blue){no_op};
while (needin [blue] && turn == blue){no_op};
needin [blue] = false;
while (needin [blue] && turn == blue)
    Critical Section
needin [green] = false;
Greedy Version (turn = me)

```java
needin [blue] = true;
needin [green] = true;
turn = blue;
while (needin [green] && turn == green)
    Critical Section
    turn = green;
while (needin [blue] && turn == blue)
    Critical Section
    Oooops!
```

Peterson’s Algorithm for 2 Process Mutual Exclusion

```java
• enter_region:
    needin [me] = true;
turn = you;
while (needin [you] && turn == you) {no_op};
• exit_region:
    needin [me] = false;
```

What about more than 2 processes?

Synchronization

• We illustrated the dangers of race conditions when multiple threads execute instructions that interfere with each other when interleaved.
• Goal in solving the critical section problem is to build synchronization so that the sequence of instructions that can cause a race condition are executed AS IF they were indivisible (just appearances)
• “Other stuff” can be interleaved with critical section code as well as the enter_region and exit_region protocols, but it is deemed OK.

Outline for Today

• Objective:
  – To clarify threads and processes.
  – To continue talking about the critical section problem and get more practice thinking about possible interleavings.
  – Start talking about synchronization primitives.
  – Introduce other “classic” concurrency problems
• Administrative details:
  – Look on the web for office hours at the beginning of next week for assignment 1.
  – New TA and UTA
Can we extend 2-process algorithm to work with \( n \) processes?

\[ \text{needin[me]} = \text{true}; \]
\[ \text{turn} = \text{you}; \]
\[ \text{needin[me]} = \text{true}; \]
\[ \text{turn} = \text{you}; \]
\[ \text{needin[me]} = \text{true}; \]
\[ \text{turn} = \text{you}; \]
\[ \text{needin[me]} = \text{true}; \]
\[ \text{turn} = \text{you}; \]

Idea: Tournament
Details: Bookkeeping (left to the reader)

Lamport’s Bakery Algorithm

- **enter_region:**
  - \( \text{choosing[me]} = \text{true}; \)
  - \( \text{number[me]} = \max(\text{number[0:n-1]} + 1; \)
  - \( \text{choosing[me]} = \text{false}; \)
  - \( \text{for (j=0; n-1; j++)} \)
    - \( \text{while (choosing[j] != 0)} \) (skip)
    - \( \text{while((number[j] != 0) and ((number[j] < number[me]) or ((number[j] == number[me]) and (j < me))))} \) (skip)
- **exit_region:**
  - \( \text{number[me]} = 0; \)

Interleaving / Execution Sequence with Bakery Algorithm

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choosing= False</td>
<td>Choosing= False</td>
<td>Choosing= False</td>
<td>Choosing= False</td>
</tr>
<tr>
<td>Number [0]= 0</td>
<td>Number [1]= 0</td>
<td>Number [2]= 0</td>
<td>Number [3]= 0</td>
</tr>
</tbody>
</table>
for (j=0; n-1; j++) {
    while (choosing[j] != 0) {
        skip
    }
    while((number[j] != 0) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) {
        skip
    }
}
for (j=0; n-1; j++) {
  { while (choosing[j] != 0) {skip}
    while((number[j] != 0 ) and ((number[j] < number[me])
      or ((number[j] == number[me]) and (j < me)))) {skip}
  }
}

Hardware Assistance

• Most modern architectures provide some support for building synchronization: atomic
  read-modify-write instructions.
• Example: test-and-set (loc, reg)
  [ sets bit to 1 in the new value of loc; returns old value of loc in reg ]
• Other examples: compare-and-swap, fetch-and-op

[ ] notation means atomic
Busywaiting with Test-and-Set

- Declare a shared memory location to represent a busyflag on the critical section we are trying to protect.

- `enter_region (or acquiring the “lock”):`
  
  ```
  waitloop: tsl busyflag, R0 // R0 = busyflag; busyflag = 1
  bnz R0, waitloop // was it already set?
  ```

- `exit region (or releasing the “lock”):`
  
  ```
  busyflag = 0
  ```

- The Alpha and MIPS 4000 processor architectures have no atomic read-modify-write instructions, i.e., no test-and-set-lock instruction (TS). Atomic update is supported by pairs of load_locked (LDL) and store-conditional (STC) instructions.

- The semantics of the Alpha architecture’s LDL and STC instructions are as follows. Executing an LDL Rx, y instruction loads the memory at the specified address (y) into the specified general register (Rx), and holds y in a special per-processor lock register. STC Rx, y stores the contents of the specified general register (Rx) to memory at the specified address (y), but only if y matches the address in the CPU’s lock register. If STC succeeds, it places a one in Rx; if it fails, it places a zero in Rx. Several kinds of events can cause the machine to clear the CPU lock register, including traps and interrupts. Moreover, if any CPU in a multiprocessor system successfully completes a STC to address y, then every other processor’s lock register is atomically cleared if it contains the value y.

- Show how to use LDL and STC to implement safe busy waiting

Pros and Cons of Busywaiting

- Key characteristic - the “waiting” process is actively executing instructions in the CPU and using memory cycles.

- Appropriate when:
  
  - High likelihood of finding the critical section unoccupied (don’t take context switch just to find that out) or estimated wait time is very short

- Disadvantages:
  
  - Wastes resources (CPU, memory, bus bandwidth)
Blocking Synchronization

- OS implementation involving changing the state of the “waiting” process from running to blocked.
- Need some synchronization abstraction known to OS - provided by system calls.
  - mutex locks with operations acquire and release
  - semaphores with operations P and V (down, up)
  - condition variables with wait and signal

Pros and Cons of Blocking

- Waiting processes/threads don’t consume resources
- Appropriate: when the cost of a system call is justified by expected waiting time
  - High likelihood of contention for lock
  - Long critical sections
- Disadvantage: OS involvement -> overhead

Template for Implementing Blocking Synchronization

- Associated with the lock is a memory location (busy) and a queue for waiting threads/processes.
- Acquire syscall:
  ```
  while (busy) {enqueue caller on lock’s queue}
  /* upon waking to nonbusy lock */
  ```
- Release syscall:
  ```
  busy = false;
  /* wakup */ move any waiting threads to Ready queue
  ```

Semaphores

- Well-known synchronization abstraction
- Defined as a non-negative integer with two atomic operations
  ```
  P(s) - [wait until s > 0; s--]
  V(s) - [s++]
  ```
- The atomicity and the waiting can be implemented by either busy waiting or blocking solutions.
Semaphore Usage

- Binary semaphores can provide mutual exclusion (solution of critical section problem)
- Counting semaphores can represent a resource with multiple instances (e.g., solving producer/consumer problem)
- Signaling events (persistent events that stay relevant even if nobody listening right now)

The Critical Section Problem

```c
while (1) {
    other stuff...
    P(mutex)
    critical section
    V(mutex)
}
```

Semaphore: mutex initially 1

Monitor Abstraction

- Encapsulates shared data and operations with mutual exclusive use of the object (an associated lock).
- Associated Condition Variables with operations of Wait and Signal.

Condition Variables

- We build the monitor abstraction out of a lock (for the mutual exclusion) and a set of associated condition variables.
- Wait on condition: releases lock held by caller, caller goes to sleep on condition’s queue. When awakened, it must reacquire lock.
- Signal condition: wakes up one waiting thread.
- Broadcast: wakes up all threads waiting on this condition.
Monitor Abstraction

**Entry Queue**

**Initialization**

**Conditions**

**EnQ**

```plaintext
{ acquire (lock);
  if (head == null)
    { head = item;
      signal (lock, notEmpty);
    } else tail->next = item;
  tail = item;
  release(lock);
}
```

**deQ**

```plaintext
{ acquire (lock);
  if (head == null)
    wait (lock, notEmpty);
  item = head;
  if (tail == head) tail = null;
  head = item->next;
  release(lock);
}
```
Monitor Abstraction

EnQ {acquire (lock);
   if (head == null)
      (head = item;
       signal (lock, notEmpty);
      )
   else tail->next = item;
   tail = item;
   release (lock);}
deQ {acquire (lock);
   if (head == null)
      wait (lock, notEmpty);
   item = head;
   if (tail == head) tail = null;
   head = item->next;
   release (lock);}

Nachos-style Synchronization

synch.h, cc

- Semaphores
  Semaphore::P
  Semaphore::V
- Locks and condition variables
  Lock::Acquire
  Lock::Release
  Condition::Wait (conditionLock)
  Condition::Signal (conditionLock)
  Condition::Broadcast (conditionLock)

Outline for Today

- Objective:
  - Continue talking about synchronization primitives.
  - Introduce other “classic” concurrency problems
- Administrative details:
  - Program 1 demo details
Tweedledum and Tweedledee

- Separate threads executing their respective procedures. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation.
- The `Sleep()` and `Wakeup()` routines operate as follows:
  - `Sleep` blocks the calling thread,
  - `Wakeup` unblocks a specific thread if that thread is blocked, otherwise its behavior is unpredictable.

Classic Problems

There are a number of “classic” problems that represent a class of synchronization situations
- Critical Section problem
- Producer/Consumer problem
- Reader/Writer problem
- 5 Dining Philosophers

The Critical Section Problem

```c
while (1)
{
  ... other stuff...
  critical section
  ...
}
```

Producer / Consumer

```c
Producer:
while(whatever)
{
  locally generate item
  fill empty buffer with item
}
Consumer:
while(whatever)
{
  get item from full buffer
  use item
}
```
Producer / Consumer

Producer:
while(whatever)
{
   locally generate item
   P(emptybuf);
   fill empty buffer with item
   V(fullbuf);
}

Consumer:
while(whatever)
{
   get item from full buffer
   use item
   V(emptybuf);
   P(fullbuf);
}

Semaphores: emptybuf initially N; fullbuf initially 0;

Template for Philosopher

while (food available)
{
   /*pick up forks*/
   eat;
   /*put down forks*/
   think awhile;
}

5 Dining Philosophers

Simplest Example of Deadlock

Thread 0       Interleaving       Thread 1
P(R1)  P(R1)  P(R2)  P(R2)       P(R1)  P(R2)  P(R1)  P(R2)
V(R1)  P(R1)  V(R1)  P(R1)  V(R1)  P(R2)  V(R2)  P(R2)
V(R2)  P(R2)  V(R2)  P(R2)  V(R2)  P(R1)  V(R1)

R1 and R2 initially 1 (binary semaphore)
**Conditions for Deadlock**

- Mutually exclusive use of resources
  - Binary semaphores R1 and R2
- Circular waiting
  - Thread 0 waits for Thread 1 to V(R2) and Thread 1 waits for Thread 0 to V(R1)
- Hold and wait
  - Holding either R1 or R2 while waiting on other
- No pre-emption
  - Neither R1 nor R2 are removed from their respective holding Threads.

**Dealing with Deadlock**

It can be **prevented** by breaking one of the prerequisite conditions:

- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem)
- Circular waiting
  - Example: Define an order on resources and acquire them in order
- Hold and wait
- No pre-emption

**Philosophy 101**

(or why 5DP is interesting)

- How to eat with your Fellows without causing Deadlock.
  - Circular arguments (the circular wait condition)
  - Not giving up on firmly held things (no preemption)
  - Infinite patience with Half-baked schemes (hold some & wait for more)
- Why Starvation exists and what we can do about it.

```c
while (food available)
{
    if (me == 0) { P(fork[left(me)]); P(fork[right(me)]); }
    else { P(fork[right(me)]); P(fork[left(me)]); }
    eat:
    V(fork[left(me)]); V(fork[right(me)]);
    think awhile;
}
```
Hold and Wait Condition

while (food available)
    { P(mutex);
      while (forks [me] != 2)
          { blocking[me] = true; V(mutex); P(sleepy[me]); P(mutex);}
      forks [leftneighbor(me)] --;  forks [rightneighbor(me)]--;
      V(mutex);
      eat;
      P(mutex); forks [leftneighbor(me)] ++;  forks [rightneighbor(me)]++; }
      if (blocking[leftneighbor(me)]) V(sleepy[leftneighbor(me)]);
      if (blocking[rightneighbor(me)]) V(sleepy[rightneighbor(me)]);
      V(mutex);
      think awhile;
    }

Starvation

The difference between deadlock and starvation is subtle:
- Once a set of processes are deadlocked, there is no future execution sequence that can get them out of it.
- In starvation, there exists some execution sequence that is favorable to the starving process although there is no guarantee it will ever occur.
- Rollback and Retry solutions are prone to starvation.
- Continuous arrival of higher priority processes is another common starvation situation.

5DP - Monitor Style

Boolean eating [5];
Lock forkMutex;
Condition forksAvail;

void PickupForks (int i) {
    forkMutex.Acquire();
    while (eating[(i+1)%5] || eating[(i+1)%5])
        forksAvail.Wait(& forkMutex);
    eating[i] = true;
    forkMutex.Release();
}

void PutdownForks (int i) {
    forkMutex.Acquire();
    eating[i] = false;
    forksAvail.Broadcast(&forkMutex);
    forksAvail.Wait(&forkMutex);
    eating[i] = true;
    forkMutex.Release();
}

What about this?

while (food available) {
    forkMutex.Acquire();
    while (forks [me] != 2) { blocking[me] = true; forkMutex.Release(); P(sleep); forkMutex.Acquire();}
    forks [leftneighbor(me)] --;  forks [rightneighbor(me)]--;
    forkMutex.Release();
    if (blocking[leftneighbor(me)]) ++;  forks [rightneighbor(me)]++; }
    if (blocking[leftneighbor(me)] || blocking[rightneighbor(me)])
        wakeup ++; forkMutex.Release();
    think awhile;
}
Readers/Writers Problem

Synchronizing access to a file or data record in a database such that any number of threads requesting read-only access are allowed but only one thread requesting write access is allowed, excluding all readers.

Template for Readers/Writers

```
Reader()
{while (true)
   {read
    /*request r access*/
    /request r access*/
    /*release r access*/
    write
   }
}

Writer()
{while (true)
   {write
    /*request w access*/
    /*release w access*/
    read
    close(fd);
   }
}
```

```
fd = open(foo, 0);
/*request r access*/
/*release r access*/
fd = open(foo, 1);
/*request w access*/
/*release w access*/
close(fd);
```

```
startRead();
/*request r access*/
/*release r access*/
startWrite();
/*request w access*/
/*release w access*/
endRead();
endWrite();
```
```c
Boolean busy = false;
int numReaders = 0;
Lock filesMutex;
Condition OKtoWrite, OKtoRead;

void startRead () {
    filesMutex.Acquire();
    while (busy)
        OKtoRead.Wait(&filesMutex);
    numReaders ++;
    filesMutex.Release();
}

void endRead () {
    filesMutex.Acquire();
    numReaders --;
    if (numReaders == 0)
        OKtoWrite.Signal(&filesMutex);
    filesMutex.Release();
}

void startWrite () {
    filesMutex.Acquire();
    while (busy || numReaders != 0)
        OKtoWrite.Wait(&filesMutex);
    busy = true;
    filesMutex.Release();
}

void endWrite () {
    filesMutex.Acquire();
    busy = false;
    OKtoRead.Broadcast(&filesMutex);
    OKtoWrite.Signal(&filesMutex);
    filesMutex.Release();
}
```