Operating Systems

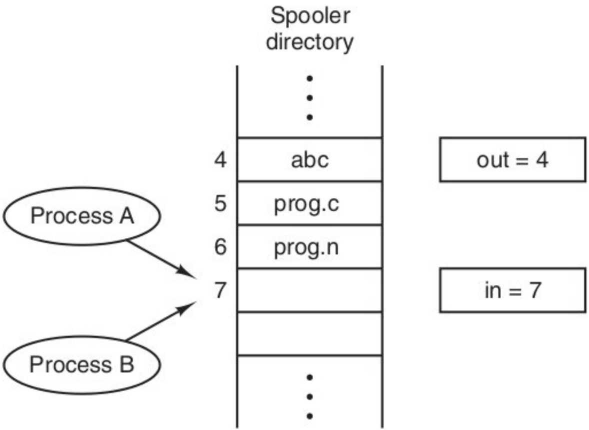
Synchronization

Carlos Alberto Llano R.

24 de septiembre de 2015

Contents

Background - Producer and Consumer example 1



Background - Producer and Consumer example 2

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

Background - Producer and Consumer example

- cont = cont + 1

```
mov  [cont],AX
add  AX,1
mov  AX,[cont]
```

- cont = cont – 1

```
mov  [cont],BX
add  BX,-1
mov  BX,[cont]
```

Background - Producer and Consumer example

Time	Process	Instruction	Register
T0	Producer	AX = cont	AX = 9
T1	Producer	AX = AX + 1	AX = 10
T2	Consumer	BX = cont	BX = 9
T3	Consumer	BX = BX - 1	BX = 8
T4	Producer	cont = AX	cont = 10
T5	Consumer	cont = BX	cont = 8

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

Race condition

When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Race condition

Examples in C: bad_thread.c

Examples in C++: race_condition.cc

Critical Section or Critical Regions

The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time.

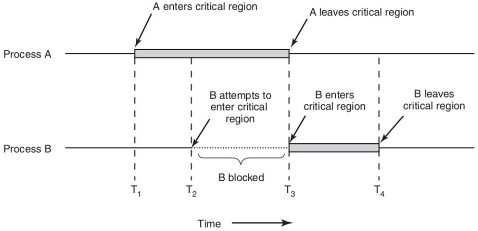
We need a mutual exclusion, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

That part of the program where the shared memory is accessed is called the critical region or critical section.

Critical Section or Critical Regions

We need four conditions to hold to have a good solution:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block any process.
- No process should have to wait forever to enter its critical region.



Critical Section Problem, in other words...

A solution to the critical section problem must satisfy the following three requirements:

- Mutual exclusion
- Progress
- Bounded waiting

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the n processes.

Critical Section Solution - Struture

do {

$\boxed{\textit{entry section}}$

critical section

$\boxed{\textit{exit section}}$

remainder section

} while (true);

Proposals for achieving mutual exclusion

- Disabling Interrupts
- Lock Variables
- Strict Alternation
- Peterson's Solution
- The TSL Instruction
- Sleep and Wakeup
- Mutex Lock
- Semaphores

Disabling Interrupts

With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process.

Lock Variables

When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.

When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.

Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory.

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Continuously testing a variable until some value appears is called busy waiting. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a spin lock.

Peterson's Solution

```
do {

    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The TSL Instruction

To use the TSL instruction, we will use a shared variable, lock, to coordinate access to shared memory. When lock is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets lock back to 0 using an ordinary move instruction.

```
enter_region:
    TSL REGISTER,LOCK        | copy lock to register and set lock to 1
    CMP REGISTER,#0          | was lock zero?
    JNE enter_region         | if it was not zero, lock was set, so loop
    RET                      | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0             | store a 0 in lock
    RET                      | return to caller
```

Sleep and Wakeup

Both Peterson's solution and the solutions using TSL or XCHG are correct, but both have the defect of requiring busy waiting (waste CPU time).

```
#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 */
     }
}
```

Mutex Locks

We use the mutex lock to protect critical regions and thus prevent race conditions.

That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire()function acquires the lock, and the release() function releases the lock.

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

```
release() {
   available = true;
}
```

```
do {
      acquire lock
         critical section
      release lock
         remainder section
} while (true);
```

Mutex Locks in C/C++

C Examples: mutex_lock1.c and mutex_lock2.c

C++ Examples: thread_mutex.cc

Mutex Locks in python

```
import threading

L = threading.Lock()

L.acquire()
\# The critical section ...
L.release()
```

Condition Variables

Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue.

Two primary routines are used by programs wishing to interact in this way:

- int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
- int pthread_cond_signal(pthread_cond_t *cond);

Condition Variables

1. The mutex lock associated with the condition variable must be locked before the pthread_cond_wait() function is called.
2. Once this lock is acquired, the thread can check the condition.
3. If the condition is false, the thread then invokes pthread_cond_wait(), passing the mutex lock and the condition variable as parameters.
4. Calling pthread_cond_wait() releases the mutex lock, thereby allowing another thread to access the shared data and possibly update its value so that the condition clause evaluates to true.
5. A thread that modifies the shared data can invoke the pthread_cond_signal() function, thereby signaling one thread waiting on the condition variable.

Condition Variables

A typical usage looks like this:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (ready == 0)
        pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock);
```

Condition Variables

The code to wake a thread, which would run in some other thread, looks like this:

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

Condition Variable in C

You have to check these to be sure:

Example: condition_variables.c

Condition Variable in Java

The Java programming language provides two basic synchronization idioms: synchronized methods and synchronized statements.

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Condition Variable in Java

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Condition Variable in Java

You have to check these to be sure too:

Example: WaitNotifyTest.java

Condition Variable in C++

to check:

Example: condition_variable.cc

Producer – Consumer Problem

The producer – consumer problem is a common implementation pattern for cooperating processes or threads. Put simply, a producer produces information that is later consumed by a consumer.

Traditionally, the problem is defined as follows:

- To allow the producer and consumer to run concurrently, the producer and consumer must share a common buffer.
- So that the consumer does not try to consume an item that has not yet been produced, the two processes must be synchronized.
- If the common data buffer is bounded, the consumer process must wait if the buffer is empty, and the producer process must wait if the buffer is full

Producer – Consumer Problem - Solution in C

producer_consumer_mutex.c

Producer – Consumer Problem - Solution in Java

Example: ProducerConsumerSolution.java

Semaphores - E. W. Dijkstra (1965)

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

```
wait(S) {
    while (S <= 0)
      ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

Semaphores - E. W. Dijkstra (1965)

```
wait(S) {                    signal(S) {
    while (S <= 0)               S++;
       ; // busy wait        }
    S--;
}
```

The wait operation on a semaphore checks to see if the value is greater than 0.

If so, it decrements the value and just continues.

If the value is 0, the process is put to sleep without completing the down for the moment.

Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible atomic action.

Semaphores - E. W. Dijkstra (1965)

```
wait(S) {                        signal(S) {
    while (S <= 0)                   S++;
       ; // busy wait             }
    S--;
}
```

The signal operation increments the value of the semaphore addressed.

If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its wait.

Python Semaphores

```
import threading

S = threading.Semaphore(5)

S.acquire()
\#
\# The critical section ...
\#
S.release()
```

Poxis Semaphores

Operations:

- semaphore.h
- int sem_init(sem_t sem, int pshared, unsigned int value)
- int sem_wait(sem_t sem)
- int sem_post(sem_t sem)
- int sem_getvalue(sem_t sem, int valp)

Two threads using a semaphore

Declare the semaphore global (outside of any funcion):

```
sem_t mutex;
```

Initialize the semaphore in the main function:

```
sem_init(&mutex, 0, 1);
```

Thread 1	Thread 2	data
sem_wait (&mutex);	---	0
---	sem_wait (&mutex);	0
a = data;	/* blocked */	0
a = a+1;	/* blocked */	0
data = a;	/* blocked */	1
sem_post (&mutex);	/* blocked */	1
/* blocked */	b = data;	1
/* blocked */	b = b + 1;	1
/* blocked */	data = b;	2
/* blocked */	sem_post (&mutex);	2
[data is fine. The data race is gone.]		

Two threads using a semaphore with states

Value	Thread 0	State	Thread 1	State
1		Running		Ready
1	call `sem_wait()`	Running		Ready
0	`sem_wait()` returns	Running		Ready
0	`(crit sect: begin)`	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	call `sem_wait()`	Running
-1		Ready	`decrement sem`	Running
-1		Ready	`(sem<0)→sleep`	Sleeping
-1		Running	*Switch→T0*	Sleeping
-1	`(crit sect: end)`	Running		Sleeping
-1	call `sem_post()`	Running		Sleeping
0	`increment sem`	Running		Sleeping
0	`wake(T1)`	Running		Ready
0	`sem_post()` returns	Running		Ready
0	*Interrupt; Switch→T1*	Ready		Running
0		Ready	`sem_wait()` returns	Running
0		Ready	`(crit sect)`	Running
0		Ready	call `sem_post()`	Running
1		Ready	`sem_post()` returns	Running

Semaphore examples

Java Examples: SemaphoreTest.java

C Examples: good_semaphores.c

Producer – Consumer Problem - Solution

producer_consumer_semaphores.c

The Readers – Writers Problem

Suppose that a database is to be shared among several concurrent processes.

Some of these processes may want only to read the database, whereas others may want to update the database.

If two readers access the shared data simultaneously, no adverse effects will result.

If a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

The Readers – Writers Problem

The conditions that must be satisfied are as follows:

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file.
- If a writer is writing to the file, no reader may read it.

The Readers – Writers Solution (Readers have priority)

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

The Readers – Writers Problem

```
typedef struct _rwlock_t {
        sem_t lock; // binary semaphore (basic lock)
        sem_t writelock; // used to allow ONE writer or MANY readers
        int readers; // count of readers reading in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
        rw->readers = 0;
        sem_init(&rw->lock, 0, 1);
        sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
        sem_wait(&rw->lock);
        rw->readers++;
        if (rw->readers == 1)
                sem_wait(&rw->writelock); // first reader acquires writelock
        sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
        sem_wait(&rw->lock);
        rw->readers--;
        if (rw->readers == 0)
                sem_post(&rw->writelock); // last reader releases writelock
        sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) { sem_wait(&rw->writelock); }
void rwlock_release_writelock(rwlock_t *rw) { sem_post(&rw->writelock); }
```

The Sleepy Barber Problem

Conditions:

- 1 waiting room with N chairs.
- 1 barber room with 1 barber chair.
- If there is not customers, the barber goes to sleep.
- If a customer enters the shop:
  - If all chairs are occupied, the customer leaves.
  - If the barber is busy and chairs are available, the customer sits in one of the free chairs.
  - If the barber is asleep, the customer wakes up the barber.

The Sleepy Barber Problem - Solution

Semaphores:

- **customers** which counts waiting customers (excluding the customer in the barber chair)
- **barbers** the number f barbers (0 or 1)

Variables:

- **waiting** which also counts the waiting customers.

The Sleepy Barber Problem - Solution

```
#define CHAIRS 5              /* # chairs for waiting customers */

semaphore customers = 0;      /* # of customers waiting for service */
semaphore barbers = 0;        /* # of barbers waiting for customers */
semaphore mutex = 1;          /* for mutual exclusion */
int waiting = 0;              /* customer are waiting (not being cut) */

void barber(void) {
    while (TRUE) {
        wait(&customers);     /* go to sleep if # of customers is 0 */
        wait(&mutex);         /* acquire access to waiting */
        waiting = waiting - 1; /* decrement count of waiting customers */
        signal(&barbers);     /* one barber is now ready to cut hair */
        signal(&mutex);       /* release waiting */
        cut_hair();           /* cut hair (outside critical region */
    }
}

void customer(void) {
    wait(&mutex);             /* enter critical region */
    if (waiting < CHAIRS) {   /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        signal(&customers);   /* wake up barber if necessary */
        signal(&mutex);       /* release access to waiting */
        wait(&barbers);       /* go to sleep if # of free barbers is 0 */
        get_haircut();        /* be seated and be served */
    }
    else {
        signal(&mutex);       /* shop is full; do not wait */
    }
}
```

Important things to remember when you use any thread library

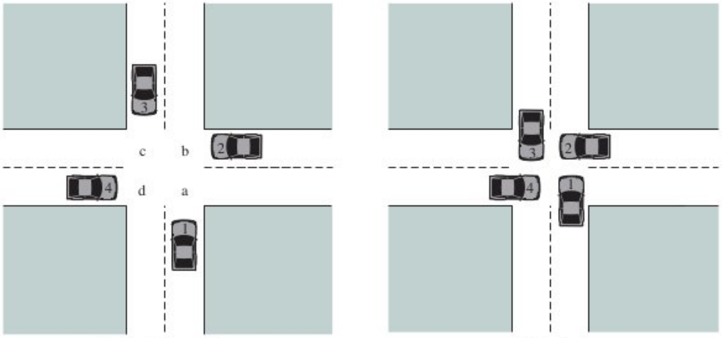
- Keep it simple
- Minimize thread interactions
- Initialize locks and condition variables: (sometimes works and sometimes fails in very strange ways)
- Check your return codes
- Be careful with how you pass arguments to, and return values from, threads. (variable allocated on the stack)
- Each thread has its own stack
- Always use condition variables to signal between threads
- Use the manual pages or manuals

Deadlocks

Deadlock can be defined as the permanent blocking of a set of process that either compete for system resources or communicate with each other.

A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.

Deadlocks



(a) Deadlock possible (b) Deadlock

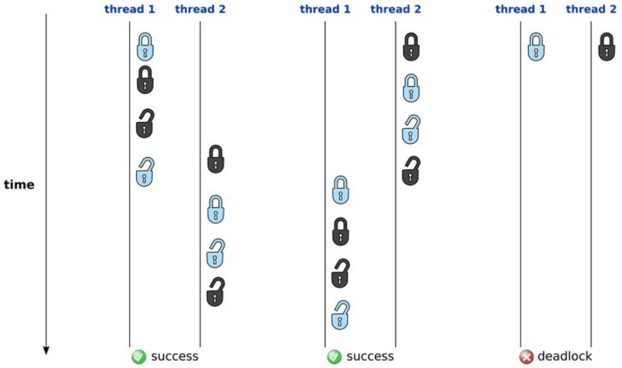
Deadlocks

Deadlocks

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

Deadlocks example

deadlock1.c



How to avoid deadlock

Though locks are the most frequent cause of deadlock, it does not just occur with locks. We can create deadlock with two threads and no locks just by having each thread call join() on the std::thread object for the other. In this case, neither thread can make progress because it's waiting for the other to finish.

For example deadlock2.cc

How to avoid deadlock

The guidelines for avoiding deadlock all boil down to one idea: don't wait for another thread if there's a chance it's waiting for you.

Monitors

Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.

This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

Monitors

Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes:

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

In this case, a deadlock will occur.

Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

Monitors

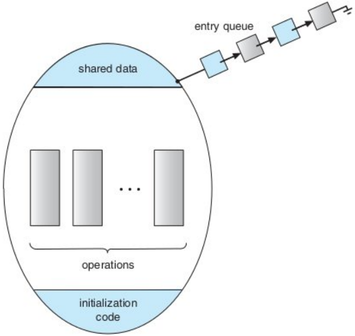
Monitors are based on abstract data types: modules that encapsulate storage, private procedures for manipulating the storage, and a public interface, including procedures and type declarations that can be used to manipulate the information in the storage.

A monitor is an abstract data type for which only one process/thread may be executing any of its member procedures at any given time.

That mean:
The methods of a monitor are executed in mutual exclusion

Schematic View of Monitors

Can think of a monitor as one big lock for a set of operations/methods, in other words, a language implementation of mutexes.



Monitors

Like a class, the abstract data type exports a public interface with member functions and possibly some data.

```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
     . . .
  }

  function P2 ( . . . ) {
     . . .
  }

        .
        .
        .
  function Pn ( . . . ) {
     . . .
  }

  initialization_code ( . . . ) {
     . . .
  }
}
```

Monitors

Other software manipulates the data type instance using the public member functions instead of directly manipulating the abstract data type's internal structures.

A monitor extends this approach by forcing a process to wait if another process is currectly executing one of the monitor's member functions.

Monitors

Abstract data types were invented to encapsulate data manipulation.

This prevented code in one module from directly manipulating the data in another module.

Member function execution is treated like a critical section!

Conceptual view of monitor

```
monitor anADT {
        private:
                semaphore mutex = 1;
                <ADT data structures>
                ...
        public:
                proc_i(...){
                        P(mutex);
                        <processing for proc_i>
                        V(mutex);
                        ...
                }
}
```

How to implement monitors?

Compiler automatically inserts lock and unlock operations upon entry and exit of monitor procedures.

```
class account {
    int balance;
    public synchronized void deposit() {
        ++balance;
    }
    public synchronized void withdraw() {
        --balance;
    }
};
```

```
lock(this.m);
++balance;
unlock(this.m);
```

```
lock(this.m);
--balance;
unlock(this.m);
```

Need wait and wakeup as in semaphores

Monitors and condition variables

wait(): suspends the calling thread and releases the monitor lock. When it resumes, reacquire the lock. Called when condition is not true

signal(): resumes one thread waiting in wait() if any. Called when condition becomes true and wants to wake up one waiting thread.

broadcast(): resumes all threads waiting in wait(). Called when condition becomes true and wants to wake up all waiting threads

Producer-Consumer with monitors

```
monitor ProducerConsumer {
   int nfull = 0;
   cond has_empty, has_full;

   producer() {
      if (nfull == N)
          wait (has_empty);
      ... // fill a slot
      ++ nfull;
      signal (has_full);
   }

   consumer() {
      if (nfull == 0)
           wait (has_full);
      ... // empty a slot
      -- nfull;
      signal (has_empty);
   }
};
```

Languages to support the monitor concept

- Pascal
- Modula-2
- Modula-3
- Ada
- Java, e.g. using the synchronized word on objects or methods

Monitors in Python

```
import threading

global available
C = threading.Condition()

C.acquire()
while not available:
    C.wait()
available = False
C.release()

# The critical section. Note that no locks are held.

C.acquire()
available = True
C.notify()
C.release()

# alternately, we could notify all waiting threads
#    C.notifyAll()
```

Monitors in C/C++

C/C++ don't provide monitors; but we can implement monitors using pthread mutex and condition variable.

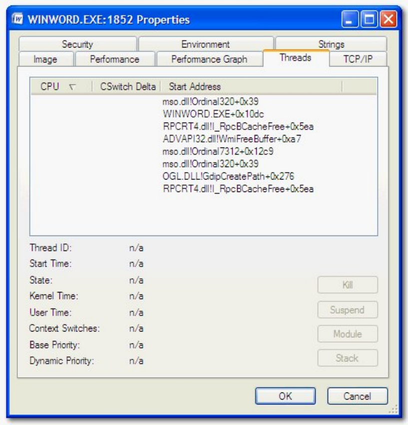
```
class ProducerConsumer {
        int nfull = 0;
        pthread_mutex_t m;
        pthread_cond_t has_empty, has_full;
public:
        producer() {
                pthread_mutex_lock(&m);
                while (nfull == N)
                        ptherad_cond_wait (&has_empty, &m);
                        ... // fill slot
                ++ nfull;
                pthread_cond_signal (has_full);
                pthread_mutex_unlock(&m);
        }
        ...
};
```

Monitors in Java

In Java, mutual exclusion of methods has to be explicitly specified with the keyword synchronized.

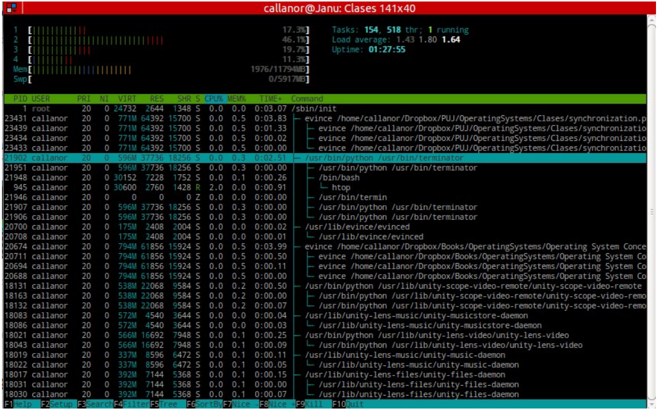
```
public class PCbuffer {
    private Object buffer[];    /* Shared memory    */
    private int N, count, in, out;  /* Buffer capacity, nb of elements, pointers */
    public PCbuffer(int argSize){
        N = argSize;
        buffer = new Object[N];
        count = 0; in = 0; out = 0;
    }
    public synchronized void append(Object data) {
        buffer[in] = data;
        in = (in + 1) % N; count++;
    }
    public synchronized Object take() {
        Object data;
        data = buffer[out];
        out = (out + 1) % N;
        count--; return data;
    }
}
```

How view threads?



How view threads?

htop (F5)



How view threads?

- top -H -p <pid>
- ps -e -T | grep <application name or pid>
- /usr/bin/pstree $PID

Future in C++

If a thread needs to wait for a specific one-off event, it somehow obtains a future representing this event. The thread can then periodically wait on the future for short periods of time to see if the event has occurred (check the departures board) while performing some other task (eating in the overpriced café) in between checks.