Operating Systems

Synchronization
Based on Ch. 5 of
OS Concepts by SGG

Need to Synchronize

- We already saw that if we write on a shared data structure w/o synchronization bad things can happen.
- The parts of the code that have this problem are called critical sections
  - A CS involves
    - A shared resource (usually shared memory)
    - Two or more processes/threads
- This is a central problem in OS design and parallel processing of any kind.

Producer-Consumer Problem

- It appears in many situations (pipes most notably)
- We play with the producer-consumer because we know about it (seen it already)
- We will look at an alternative implementation

Atomic operations

- An operation is atomic when it appears to happen instantaneously, ie nothing can happen between its start and finish.
  - Load a single register from the memory is an example of an atomic operation (there are some caveats)
  - Loading two registers from two places in memory is not because between loading the first and loading the second, one of the places in the memory may change inconsistently
- We have to make a few reasonable assumptions

Producer-Consumer

```
while (true) {
  //produce something
  while (cnt==BUF_SZ)
    ;
  buffer[in] = product;
  in = (in+1)%BUF_SZ;
  cnt++;
}
```

```
while (true) {
  while (cnt==0)
    ;
  food = buffer[out];
  out = (out+1)%BUF_SZ;
  cnt--;
  //Consume food;
}
```

Problem...

- We use the whole buffer now (before we could not use one cell).
- But when both processes execute cnt++ and cnt --, bad things can happen
- Incrementing a variable in the memory is not an atomic operation because it involves reading the original value into a register, incrementing the register and saving the register contents.
- Many things can happen between any pair of these operations

Recipe for Chaos

```
cnt++ is:
  R1 <- cnt
  R1 <- R1+1
  cnt <- R1
```

```
cnt-- is:
  R1 <- cnt
  R1 <- R1-1
  cnt <- R1
```

What if it happens Like this:

```
R1 <- cnt
R1 <- cnt
R1 <- R1+1
R1 <- R1-1
cnt <- R1
cnt <- R1
```

Or it happens Like this:

```
R1 <- cnt
R1 <- cnt
R1 <- R1-1
R1 <- R1+1
cnt <- R1
cnt <- R1
```

The Critical Section Problem

- We now know how to create chaos
- To avoid chaos we have to understand critical sections
- Some terminology
  - Entry section is the part of the code where we prepare for the entry to CS
  - Exit section where we clean up before we go
  - Remainder section, everything else

Critical Section Problem

- To avoid chaos and other unpleasantness we have to guarantee
  - Mutual exclusion: obvious
  - Progress: only processes that do not execute in the remainder section can decide and do so in finite time
  - Bounded waiting: There exists a bound on the number of times other processes are allowed to enter the critical section after a process made a request to enter and the request is granted.

Preemption

- Kernels can be of two kinds:
  - Preemptive
  - Non-preemptive
- In preemptive ones the CPU can be taken away from the running process at any time
  - We deal mainly with this case
- In non-preemptive kernels that CPU can be lost only when a process in the kernel releases the CPU
  - Used only in simpler or very specialized systems
  - Or inside older kernels

Peterson's Solution

- A simple solution for two processes only
  - A generalization of it is the Bakery Algorithm.
- Assumes that only LOAD and STORE of an integer is atomic.
- It uses busy-waiting aka spinlock.

Peterson's Solution

```
Process i
```

```
while (true){
  flag[i]=true;
  turn = j;
  while (flag[j]&&turn==j)
    ;
  /*Critical section*/
  flag[i]=false;
}
```

```
Process j
```

```
while (true){
  flag[j]=true;
  turn = i;
  while (flag[i]&&turn==i)
    ;
  /*Critical section*/
  flag[j]=false;
}
```

Correctness of Peterson's solution

- Satisfies Mutual Exclusion
- Satisfies Progress
  - If both processes were stuck then
- Satisfies Bounded-Waiting

Problems with Peterson's solution

- Works only for two processes
  - Can be generalized, though
- Does not get any help from the hardware
  - Only needs atomic `LOAD` and `STORE`.
- Uses spinlock
- Modern computers re-order read and write instructions (both compile-time and run-time) as long as (local) dependencies are not violated

Re-ordering Instructions

```
boolean flag=false;
Int     x=0;
```

```
while (!flag)
   ;
print x;
```

```
x=100;
flag=true;
```

Hardware to the rescue

- Hardware designers can create other atomic operations
  - `test_and_set`
  - `compare_and_swap`
  - Load-linked and Store-conditional.
- Can be done in various ways
  - Disable the interrupts on uniprocessors
    - There are not many of them anymore; Even raspberry pi has 4 cores!
  - Lock the bus
  - Listen to the bus and if a write to the same memory address is heard, abort and restart.
    - Modern CPUs have `load-linked` and `store-conditional`

Memory Barriers

- There are two memory models:
  - Weakly ordered
    - When writes are not immediately visible to other processors
  - Strongly ordered
    - When writes are immediately visible
      - What on earth does this even mean?
- OSes should work with any model
- Most architectures provide instructions that force memory writes to propagate
  - They are called memory barriers

Re-ordering Instructions

```
boolean flag=false;
Int     x=0;
```

```
x=100;
mem_barrier();
flag=true;
```

```
while (!flag)
  mem_barrier();
print x;
```

Test and Set

```
boolean testandset(boolean *tgt)
{
  boolean prev = *tgt;

  *tgt = true;
  return prev;
}
```

```
while (true){
  while (testandset(&lock))
    ;
  /*Critical section*/
  lock = false;
  /* Remainder Section */
}
```

```
/*Initialize*/
lock = false;
```

Test and Set

- Seems to work...
- Kind of...
  - Assures *mutual exclusion*
  - Can assure *bounded waiting* with a bit of extra coding
  - Can assure *progress* only if used in a proper manner

Compare and Swap

```
boolean compareandswap(boolean *v, boolean exp, boolean new)
{
  boolean prev = *v;
  if (*v == exp)
    *v = new;
  return prev;
}
```

```
while (true){
  while (compareandswap(&lock, 0, 1)
    ;
  /* Critical Section */
  lock = 0;
  /* Remainder Section */
}
```

Compare and Swap

- Same deal as with test and set
- The difference is minor

How to satisfy Bounded Waiting

- The trick is simple:
  - Instead of leaving it the the scheduler to decide who runs next and acquires the lock after we release it
  - We decide who runs first by passing them the lock directly (in a virtual sense)
- We have a couple more variables now...

```
boolean lock = 0;
boolean waiting[N]={0, 0, ...};
```

Solution with bounded waiting

```
while (1){
  local key, j;

  waiting[i]=1;
  key=1;
  while (waiting[i]&&key)
    key = test_and_set(&lock);
  waiting[i] = 0;
  /* Critical Section */
  j = (i+1)%N;
  while (j!=i&&~waiting[j])
    j=(j+1)%N;
  if (i==j)
    lock=0;
  else
    waiting[j]=0;
}
```

How it is implemented

- Intel has cmpxchg which is compare-and-swap
  - If executed with bus lock
  - Lock guarantees that nothing will happen during the execution of of compare-and-swap
- RISC-V has LL/SC
  - Load-linked, store-conditional
  - Aka load-reserve
  - Here is a version of CaS.

```
cas:
  lr.w t0, (a0)
  bne  t0, a1, fail
  sc.w a0, a2, (a0)
  jr   ra
fail:
  li   a0, 1
  jr   ra
(RISC-V manual)
```

Mutex Locks

- The techniques so far were for the system programmer.
- An application programmer needs easier to use tools.

```
acquire(){
  while (!available)
    ;
  available = false;
}
```

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

Mutex Locks

- To use them we acquire the mutex before the CS and release it afterwards

```
while (true){
  acquire();
  /*Critical section*/
  release();
  /*Remainder Section*/
}
```

Some Terminology

- Lock Contention
  - Low contention locks: few threads attempt to acquire them at the same time
  - High Contention Locks. The opposite
- Short Duration Spinlock
  - When the expected duration is less than 2 context switches
  - Makes sense in multicore CPUs.

Semaphores

- The real celebrity of synch mechanisms.
- It is an integer (at least in theory)
- Accessed only through two operations
  - `signal()`
  - `wait()`
- Most OSes have an implementation of semaphores

Semaphores

- There are two kinds
  - Counting semaphores
    - The most flexible/powerful
  - Binary semaphores
    - Mutexes, really.
- There is nothing that the one can do and the other cannot.

Semaphores

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

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

How to use the Semaphores

```
/* Initialize */
Scond = 0;
/* This use is uncommon, but
a variant is common */
```

```
/* P1 */
/* Critical Section */
signal(Scond);
```

```
/* P2 */
wait(Scond);
/* Critical Section */
```

```
/* Initialize */
Smutex = 1;
```

```
while (1)
{
   wait(Smutex);
   /* Critical Section */
   signal(Smutex);
   /* remainder */
}
```

How to Abuse Semaphores

```
/* Process 1 */
while (true){
  wait(mutex1);
  wait(mutex2);
    /* CS */
  signal(mutex1);
  signal(mutex2);
}
```

```
/* Process 2 */
while (true){
  wait(mutex2);
  wait(mutex1);
    /* CS */
  signal(mutex1);
  signal(mutex2);
}
```

Semaphore Implementation

- Semaphores are simple and powerful but still use spinlocks (busy waiting)
- Can be implemented so that they avoid long spinlocks.
- The spinlock cannot be completely avoided
  - It is needed for mutual exclusion within the semaphore.
  - The CS within the semaphore lasts very little so it is unlikely that any process will find this CS occupied.

Semaphore Implementation

```
typedef struct{
  int val;
  struct proc *list
} sem;
```

```
signal(sem *S){
  S->val++;
  if (S->val<=0)
  {
    dequeue(&proc,&(S->list));
    wakeup(proc);
  }
}
```

```
wait(sem *S){
  S->val--;
  if (S->val<0)
  {
    enqueue(me,&(S->list));
    block();
  }
}
```

Semaphore Implementation

- What could go wrong?
  - Assume we have mutex etc...
  - Before blocking, the process has to release the mutex (o/w nobody will be able to unlock it)
  - Between releasing the mutex and blocking, another process may execute signal and the wakeup call finds nobody sleeping. Meanwhile the process that goes to sleep will be sleeping ever-after.

Semaphore Implementation

- We have to protect the code with mutexes
- We have to release the mutex and block in one go (atomic operation)
- It is hard (even counterproductive) to eliminate spinlocks completely since they are used during the short period where we check the variables and rearrange the queues.

Priority inversion

- Consider three processes H, M, and L that have high, medium and low priority respectively.
- Process L holds a resource R (usually to update something in the kernel)
- Process M becomes ready and, since it has higher priority than L, takes the CPU
- Then process H comes and tries to get resource R and blocks. Meanwhile process M runs for as long as it desires.

Priority Inversion

- Solution is priority inheritance.
  - The priority of a process is the max of its inherent priority and the priority of the resources it holds.
  - The priority of a resource is the max of the priorities of the processes that wait for it and zero o/w
- In the situation above, process L would run temporarily with the same priority as H because it holds a resource that H waits for.

Monitors

- Semaphores are great, but more suitable for low level programming
  - Simple, powerful and efficient
  - Easy to write great code or poor code
  - Concurrent programs are hard to debug, with semaphores even harder
  - Hard to write modern object oriented code with raw semaphores
- Hence monitors

What can go wrong with Semaphores

- For example:
  - This ----------->

```
while (true){
   /* become hungry */
   signal(chair)
   /* get to eat */
   wait(chair);
   /* think */
}
```

What can go wrong with Semaphores

- We may have wait and signal in the wrong order, or two waits or two signals in a row.
- The compiler cannot detect this
  - In many cases a signal has to come before a wait.
  - Very often a wait and the corresponding signal are in different functions, even different files.
  - Most commonly semaphores (or other similar primitive objects) are used to provide mutual exclusion or as a place for a process to block until a condition is satisfied.

Enough Already
What is a Monitor?

- Different languages that use monitors have different approaches and interpretations
  - Java has provisions for `synchronized` methods in classes, along with condition variables, semaphores and mutexes.
  - Concurrent Pascal has practically vanilla monitors.
  - Pthreads has a bare bones system of mutexes and condition variables.

Enough Already
What is a Monitor?

- Looks like this:

Monitor

```
monitor Buffer{
   /* shared variables */
   int saved;
   condition emptyq, fullq;
   ...
   /* methods */
   function put(int item){
   ...
   }
   /* initialization code */
   init_code(){
   ...
   }
}
```

How it works

- Only one process can execute an operation inside the monitor
- If a process has to wait for a resource (ie a condition to become true) it joins a queue and blocks. The queue is (usually) called condition variable.
- If a process causes a change in the state of a condition, should signal this condition.

What is more Powerful?

- Is there anything that one can do with monitors that is not possible with semaphores?
- We find the answer in one of two ways
  - Solve all the problems using semaphores
    - (There are too many problems...)
  - Implement monitors using semaphores.
- We, of course, choose the second option.

Implement Monitors Using Semaphores

Signal Operation

```
if (xcnt>0){
   next_cnt++;
   signal(x_sem);
   wait(next);
   next_cnt--;
}
```

Externally callable methods

```
wait(mutex);
/* Body of method */
if (next_cnt>0)
   signal(next);
else
   signal(mutex);
```

Variables

```
next: semaphore for processes
      to wait
next_cnt: number of processes
          signaled but waiting
mutex: mutex semaphore
x_sem: semaphore for the x
   condition variable
xcnt: number of processes
      waiting
```

Wait Operation

```
xcnt++;
if (next_cnt>0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
xcnt--;
```

Bounded Buffer Problem

- The variety of Producer-Consumer problems that have fixed buffer size (like we saw in the beginning of the chapter)
- Needs three semaphores
  - One for mutex
  - One to block the consumer if the buffer is empty
  - One to block the producer if the buffer is full

Bounded Buffer Problem

```
#define N 512;

Semaphore mutex = 1;
Semaphore empty_spot = N;
Semaphore full_spot = 0;
```

```
while (true){
   /* Produce something */
   wait(empty_spot);
   wait(mutex);
     /* place in buffer */
   signal(mutex);
   signal(full_spot);
}
```

```
while (true){
   wait(full_spot);
   wait(mutex);
     /* get from buffer */
   signal(mutex);
   signal(empty_spot);
   /* Consume item */
}
```

Readers-Writers

- Assume that there are several processes that need to read some data
- There are also processes that update the data
- We can let many readers to read at the same time
- But writers have to be alone to avoid inconsistent states.

Readers Writers

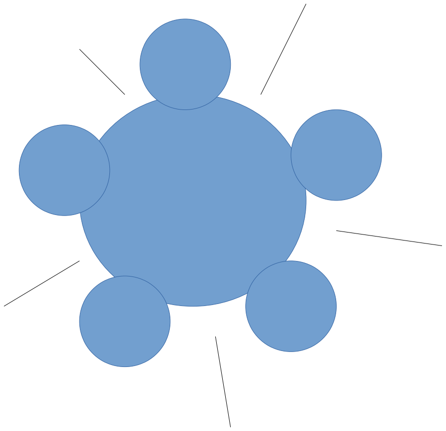
```
while (true){
  wait(wmutex);
    /* Do updates */
  signal(wmutex);
}
```

```
while (true){
  wait(rmutex);
  readcnt++;
  if (readcnt==1)
    wait(wmutex);
  signal(rmutex);
    /* Read... */
  wait(rmutex);
  readcnt--;
  if (readcnt==0)
    signal(wmutex);
  signal(rmutex);
}
```

The Dining Philosophers Problem

- The iconic synchronization problem
- There are N philosophers (make N=5) around a table.
- There are also N chopsticks between every two neighboring philosophers
- Philosophers are in one of three states: thinking, eating, hungry.
- A philosopher needs two chopsticks to eat.

Dining Philosophers Problem



How to Avoid Deadlock

- There are many different ways to do it:
  - Let N-1 philosophers to the table.
  - Let even numbered philosophers pick up the left chopstick first and odd numbered the right
  - Allow philosophers to pick up the chopsticks only when both are available.
  - Allow philosophers pick up the lower indexed chopstick first

Naive Solution w/ semaphores

```
            Philosopher i

while (true){
  /* become hungry */
  wait(c[i]);
  wait(c[(i+1)%5]);
  /* get to eat */
  signal(c[(i+1)%5]);
  signal(c[i]);
  /* think */
}
```

Table Restrictions

```
            Philosopher i

while (true){
  /* become hungry */
  wait(chair)
  wait(c[i]);
  wait(c[(i+1)%5]);
  /* get to eat */
  signal(c[(i+1)%5]);
  signal(c[i]);
  signal(chair);
  /* think */
}
```

```
c[i] = 1;
chair = 4;
```

Odd-Even

```
           Philosopher i

while (true){
  /* become hungry */
  if (even(i)){
    wait(c[i]);
    wait(c[(i+1)%5]);
    } else {
    wait(c[(i+1)%5]);
    wait(c[i]);
  }
  /* get to eat */
  signal(c[(i+1)%5]);
  signal(c[i]);
  /* think */
}
```

Both or Nothing

Philosopher i

```
while (true){ /*become hungry*/
  wait(mutex);
  st[i]=HUNGRY;
  if (st[(i-1)%5]!=EATING &&
      st[(i+1)%5]!=EATING){
    st[i] = EATING;
    signal(mutex);
  } else {
    signal(mutex);
    wait(P[i]);
  }
  /* get to eat */
```

Philosopher i (cont.)

```
  /* get to eat */
  wait(mutex);
  st[i] = THINKING; /* think */
  if (st[(i-2)%5] != EATING &&
      st[(i-1)%5] == HUNGRY){
    st[(i-1)%5] = EATING;
    signal(P[(i-1)%5]);
  }
  /* same for the right
     Philosopher */
  signal(mutex);
}
```

Dining Philosophers with Monitors

```
enum {THINKING, HUNGRY, EATING} st[5];
condition P[i];
```

```
init_code(){
  for (int i=0; i<5; i++)
    st[i]=THINKING;
}
```

```
monitor DinPhil{
  void pickup(int i){
    st[i] = HUNGRY;
    test(i);
    if (st[i]!=EATING)
      P[i].wait();
  }
  void putdown(int i){
    st[i]=THINKING;
    test((i+4)%5);
    test((i+1)%5);
  }
  void test(int i){
    If ((st[(i+4)%5]!=EATING)&&
        (st[i]==HUNGRY)&&
        (st[(i+1)%5]!=EATING)){
      st[i]=EATING;
      P[i].signal();
  }
}
```

Synchronization in Windows

- For things that are of short duration inside the kernel
  - Uses interrupt masks on uniprocessors
  - Uses spinlocks on multiprocessors
  - Avoids preemption while on spinlock
- Outside the kernel uses dispatcher objects
  - Mutexes
  - Semaphores
  - Events (similar to condition variables)
  - Timers (wake a process that blocked more than a specified time)

Synchronization in Windows

- Dispatcher objects can be
  - In signaled state
  - In non-signaled state.
- Processes blocked on a non-signaled state are placed on a queue. And the state of the process is waiting.
- When a signal is executed one (or more) processes will be woken up.

Synchronization on Linux

- For very simple kernel operations Linux provides atomic increment, addition, etc for integers.
  - These are implemented with the help of hardware (bus locking or monitoring)
- For more complex kernel operations Linux has mutexes, semaphores, reader-writer locks etc.
- In single processor systems interrupt disabling is used
- In multi processor systems spinlocks are used
  - System does not preempt processes that hold locks

Pthreads

- The main API for multithreading in Linux.
- Defined by POSIX not by the Linux kernel
- Provides mutex locks, condition variables and read-write locks
- A wait() on a condition variable requires a mutex as a second argument to release atomically before blocking.
- Many implementations of Pthreads offer semaphores as well.

Pthreads vs Monitors

- Both use condition variables.
- Both have a kind of mutex mechanism
- Little else

Mutex

- There are different kinds of mutex in Pthreads
  - Fast and risky (PTHREAD_MUTEX_NORMAL)
  - Slow and safe (PTHREAD_MUTEX_ERRORCHECK)
  - There are others too

Condition Variables

- Condition variables are similar to the condition variables in monitors
- When one locks a condition variable, one has to provide the mutex.
- When one unlocks a condition variable (signal or broadcast) no mutex needs to be provided.
- If a process wakes up from a condition variable, it has to compete with other processes to re-acquire the mutex.