Semaphores and their implementation

Reminder: The Semaphore concept

A semaphore is a shared integer variable. Its value is positive or 0 and it can only be accessed through the two operations `wait(s)` and `signal(s)`, where `s` is an identifier representing the semaphore.

- `wait(s)` decrements `s` if `s > 0` ; if not, the process executing the operation `wait(s)` is suspended.

- `signal(s)` increments `s`. The execution of `signal(s)` can have as result (possibly delayed) that a process waiting on the semaphore `s` resumes its execution. Executing a `wait(s)` or a `signal(s)` operation is done without any possible interaction (atomically).

Mutual exclusion with semaphores

Semaphores make a very simple implementation of mutual exclusion possible.

```
semaphore s = 1;
```

Process 1 :

```
while (True)
{   nc1: /* non critical
            section */ ;
         wait(s);
  crit1: /* critical section */ ;
         signal(s);
}
```

Process 2 :

```
while (True)
{   nc2: /* non critical
            section */
         wait(s);
  crit2: /* critical section */
         signal(s)
}
```

Implementing Semaphores

- Semaphores are implemented in the system kernel.
  - The semaphore values are kept in a table stored in kernel memory. A semaphore is identified by a number corresponding to a position in this table.
  - There are system calls for creating or freeing semaphores, as well as for executing the `wait` and `signal` operations. These operations are executed in supervisor mode and hence atomically (interrupts are disabled in supervisor mode).

- In ULg03, to execute for instance a `wait` operation, the arguments of the system call, i.e. the semaphore number and a code `WAIT`, are placed on the stack. Assuming that the semaphore number is contained in `r0`, this can be done as follows.

```
PUSH(r0)          | 2nd argument
CMOVE(WAIT,r1)    | 1st argument
PUSH(r1)
SVC()             | system call
```

The SVC handler

Before getting into the details of the handlers for semaphore operations, we will look at the general handler needed for the SVC instruction. The handler is reached through the following stub.

```
h_stub:  ST(r0, User, r31)     | save
         ST(r1, User+4, r31)   | the registers
              . . .
         ST(r30, User+30*4, r31)
         CMOVE(KStack, SP)     | Load the system SP
         BR(Svc_handler,LP)    | Call the handler
         LD(r31, User, r0)     | restore
         LD(r31, User+4, r1)
         LD(r31, User+30*4, r30)
         JMP(XP)               | return to application
```

Note that the address saved in XP is that of the instruction following SVC.

The `SVC` handler (continued)

- The program `Svc_handler` identifies the nature of the call and switches to the corresponding routine.

- One difficulty is that the arguments of `SVC` are on the stack of the process that has executed the system call, i.e. in its *virtual space*.

- These addresses must thus be translated before being used.

The SVC handler: data structures

The data structures used by the kernel are the following.

```
struct Mstate { int R0; ..., R30;} User;
       /* The saved state of the current process */

struct PTentry {short valid, resid ; int PhysPage;};
      /* An entry in the page table */

struct PTentry PageMap[262144];
      /* The page table of the current process, 2^18 entries */

struct PD {struct Mstate state ; int status, semwait ;
           struct PTentry PageMap[262144];} Proctbl[N];
      /* The process table with page table, status, and the address
         of a blocking semaphore if any */

int sem[K]; /* The table of semaphore values */

int Cur;  /* The index of the current process */
```

The SVC handler: translating virtual addresses

To have access to the stack of the process that has executed the instruction SVC, the system needs to translate into a physical address the address contained in r29 (SP). Indeed, even if the page table cache is up to date, address translation is not done by the hardware since we are in supervisor mode. This address translation can be done as follows.

```
int Vmap(int Vaddress)

 { int  VpageNo, PageNo, Offset;
   VpageNo = Vaddress>>14; Offset = Vaddress & 16383;
   if (PageMap[VpageNo].valid == 0 || PageMap[VpageNo].resid == 0)
           Pagerror(VpageNo);
   PageNo =  PageMap[VpageNo].PhysPage ;
   return((PageNo<<14) | Offset);
 }
```

The SVC handler: general structure

The SVC handler extracts the code identifying the call and switches to the corresponding function.

```
Svc_handler()
{ int code;
  code = *Vmap(User.R29-4)
  switch (code)
       {  ....

          case WAIT   : Wait_h(*Vmap(User.R29-8)); break;
          case SIGNAL : Signal_h(*Vmap(User.R29-8)); break;
          ....
       }
}
```

The addresses of the arguments of the system call are not computed exactly as in the case of a procedure call since, in the case of a system call, nothing is added to the stack of the calling process after the call.

The handlers `Wait_h` and `Signal_h`: first version

```
Wait_h(int semno)
 { if (sem[semno] <= 0) {
      User.R30 = User.R30 - 4; /*  SVC will be executed again */
      Proctbl[Cur].status = WAIT; Proctbl[Cur].semwait = semno;
      scheduler();
      }  else
          sem[semno] = sem[semno] - 1;
 }
```

When the value of the semaphore is not positive, the process is suspended; it will reexecute the `SVC` when it is reactivated.

```
Signal_h(int semno)
 { for (i = 0; i<N; i++) {
      if (Proctbl[i].status == WAIT && Proctbl[i].semwait == semno)
                  Proctbl[i].status = RUN;
   }
   sem[semno] = sem[semno] + 1;
 }
```

The processes that are waiting on the semaphore are reactivated and the semaphore is incremented.

All the waiting processes will reexecute the system call `wait`; the one that will find a nonzero value for the semaphore can be anyone of these.

For mutual exclusion, this implementation will not guarantee that each process can access its critical section.

The handlers `Wait_h` and `Signal_h`: second version

```
Wait_h(int semno)
 { if (sem[semno] <= 0) {
      Proctbl[Cur].status = WAIT; Proctbl[Cur].semwait = semno;
      scheduler();
      }  else
          sem[semno] = sem[semno] - 1;
 }
```

The system call is not reexecuted when a process leaves its wait state. Indeed, the handler for `signal` will take care of reactivating a waiting process if there is one, without the semaphore being incremented and decremented.

```
Signal_h(int semno)
 { int i,wokeup = 0;
   for (i = 0; i<N; i++) {
      if (Proctbl[i].status == WAIT && Proctbl[i].semwait == semno) {
          Proctbl[i].status = RUN;
          wokeup = 1; break;
      }
   }
   if (wokeup == 0) sem[semno] = sem[semno] + 1;
 }
```

This implementation guarantees that, if only two processes use the semaphore, none will wait indefinitely.

A fair implementation for more than two processes will use a wait queue per semaphore for handling the waiting processes.

Implementing semaphores on a multiprocessor

- On a multiprocessor machine, execution in supervisor mode does not guarantee mutual exclusion since it can occur simultaneously on more than one processor.

- Another mechanism for implementing mutual exclusion is thus needed.

- Atomic memory reads and writes are not sufficient for a practical solution.

- One thus introduces a special instruction that can atomically read AND modify memory.

The instruction "Test and Clear"

This instruction copies a memory word to a register and sets it to 0.

TCLR(Ra,literal,Rc) :
$$\begin{aligned} PC &\leftarrow PC + 4 \\ EA &\leftarrow Reg[Ra] + SEXT(literal) \\ Reg[Rc] &\leftarrow Mem[EA] \\ Mem[EA] &\leftarrow 0 \end{aligned}$$

It's opcode is chosen to be `0x04`

```
.macro TCLR(Ra,Lit,Rc)          ENC_LIT(0b000100,Ra,Rc,Lit)
```

The microcode of TCLR

TCLR(Ra, Literal, Rc) (supervisor mode)

Opcode = 000100 IRQ = * PC31 = 1

Phase	Fl.	$\overline{\text{Latch flags}}$	UC/D	ALU F,$\overline{C_{in}}$,M	LD SEL	DR SEL	PC+	SVR	
00000	*	1	0	000000	0011	0001	0	0	SMAR <- Ra
00001	*	1	0	000000	0001	0100	0	0	A <- SRAM
00010	*	1	0	000000	0010	0010	0	0	B <- Lit
00011	*	1	0	000000	0011	0000	0	0	SMAR <- Rc
00100	*	1	0	100110	0100	0011	0	0	DMAR <- A+B
00101	*	1	0	000000	0101	0101	0	0	SRAM <- DRAM
00110	*	1	0	001101	0110	0011	0	0	DRAM <- 0
00111	*	1	0	000000	0100	0110	1	0	DMAR <- PC; PC+
01000	*	1	0	000000	0000	0101	0	0	INSTREG <- DRAM

Mutual exclusion with TCLR

It is quite simple to implement mutual exclusion with TCLR.

```
wait: TCLR(r31,lock,r0)
      BEQ(r0,wait)

      ... section critique ...

      CMOVE(1,r0)
      ST(r0,lock,r31)
```

This implementation uses an active wait and does not guarantee fairness among the waiting processes.

It is not used to implement mutual exclusion in general, but it is perfectly adequate for implementing semaphores on a multiprocessor machine.

The "buffer" or "producer-consumer" problem

A *Producer* process sends a stream of information to a *Consumer* process. This stream of information goes through a buffer modeled as a table in shared memory accessed with the two functions (`append` and `take`).

```
/* shared memory */
int in = 0, out = 0;
int buf[N];
```

```
append(int v)
 { buf[in]= v;
   in = (in+1) % N;
 }
```

```
int take()
 { int v;
   v = buf[out];
   out = (out+1) % N;
   return v;
 }
```

As such these operations allow writing in a full buffer or reading from an empty one. *they need to be synchronized*.

The buffer problem: Synchronizing the consumer

To prevent the consumer from taking an element from an empty buffer, a semaphore that counts the number of elements in the buffer is used.

```
/* shared memory */
int in, out = 0;
int buf[N];
semaphore n = 0;
```

```
append(int v)
 { buf[in]= v;
   in = (in+1) % N;
   signal(n);
 }
```

```
int take()
 { int v;
   wait(n);
   v = buf[out];
   out = (out+1) % N;
   return v;
 }
```

It is still necessary to limit the producer when the buffer is full.

The buffer problem: Synchronizing the consumer and the producer

A second semaphore, initialized to the number of empty slots in the buffer, is used to synchronize the producer.

```
/* shared memory */
int in, out = 0;
int buf[N];
semaphore n = 0, e = N;
```

```
append(int v)
 { wait(e);
   buf[in]= v;
   in = (in+1) % N;
   signal(n);
 }
```

```
int take()
 { int v;
   wait(n);
   v = buf[out];
   out = (out+1) % N;
   signal(e);
   return v;
 }
```

The buffer problem: Mutual exclusion of the operations

If there are several producers and consumers, it is necessary to ensure that the buffer manipulation operations are performed in mutual exclusion. This can be done with an additional semaphore.

```
/* shared memory */
int in, out = 0; int buf[N];
semaphore n = 0, e = N, s = 1;
```

```
append(int v)
 { wait(e);
   wait(s);
   buf[in] = v;
   in = (in+1) % N;
   signal(s);
   signal(n);
 }
```

```
int take()
 { int v;
   wait(n);
   wait(s);
   v = buf[out];
   out = (out+1) % N;
   signal(s);
   signal(e);
   return v;
 }
```

Binary semaphores

- A binary semaphore is a semaphore whose value can only be 0 or 1.
- Thus type of semaphore makes it possible to synchronize processes, but not to count.
- A general semaphore may nevertheless be simulated by a binary semaphore and a counter.
- Some problems can nevertheless be caused by the split between the semaphore and the counter.

The buffer with binary semaphores

```
/* shared memory */
int in = 0, out = 0, count = 0;
int buf[N];
semaphore n = 0, e = 0, s = 1;
```

```
append(int v)
 { wait(s);
   if (count == N) {
      signal(s);wait(e);wait(s);}
   buf[in] = v; in = (in+1)%N;
   count = count+1;
   if (count == 1) signal(n);
   signal(s);
 }
```

```
int take()
 { int v;
   wait(s);
   if (count == 0) {
    signal(s);wait(n);wait(s);}
   v = buf[out];out = (out+1)%N;
   count = count-1;
   if (count == N-1) signal(e);
   signal(s); return v;
 }
```

This solution is *not correct* since it is possible to execute `append`; `take`; `take`. The problem is that the presence of an element is signaled twice: by setting `count` to 1 and by the operation `signal(n)`.

The buffer with binary semaphores: Solution for one producer and one consumer

```
/* shared memory */
int in = 0, out = 0, count = 0;
int buf[N]; int ewait = 0 ,nwait = 0;
semaphore n = 0, e = 0, s = 1;
```

```
append(int v)
 { wait(s);
   if (count == N) {
     ewait = 1; signal(s);
     wait(e);wait(s);ewait = 0;}
   buf[in] = v; in = (in+1)%N;
   count = count+1;
   if (nwait > 0) signal(n);
   signal(s);
 }
```

```
int take()
 { int v;
   wait(s);
   if (count == 0) {
     nwait = 1; signal(s);
     wait(n);wait(s);nwait = 0;}
   v = buf[out];out = (out+1)%N;
   count = count-1;
   if (ewait > 0) signal(e);
   signal(s); return v;
 }
```

- To avoid signaling twice, `signal(n)` or `signal(e)` are only executed if a process is waiting.

- One can imagine extending this solution to several producers and consumers by counting the number of waiting processes.

- However, one must for example avoid a situation in which a second consumer overtakes a consumer that has just been allowed to proceed (has been released) and consumes the element intended for this consumer.

- This can be done by transferring mutual exclusion to the process that has been released.

The buffer with binary semaphores: Solution for several producers and consumers

```
/* shared memory */
int in = 0, out = 0, count = 0;
int buf[N]; int ewait = 0 ,nwait = 0;
semaphore n = 0, e = 0, s = 1;
```

```
append(int v)
 { wait(s);
   if (count == N) {
     ewait++;signal(s);wait(e);
     ewait--;}
   buf[in] = v;in = (in+1)%N;
   count = count+1;
   if (nwait > 0) signal(n);
             else  signal(s);
 }
```

```
int take()
 { int v;
   wait(s);
   if (count == 0) {
     nwait++; signal(s);wait(n);
     nwait--;}
   v = buf[out];out = (out+1)%N;
   count = count-1;
   if (ewait > 0) signal(e);
                  else signal(s);
   return v;
 }
```

Conclusions on semaphores

- Semaphores are an excellent mechanism for handling processes that have to be suspended (put in a wait state).

- When suspending processes is combined with manipulating a counter, semaphores are perfectly well adapted.

- Attempting to use binary semaphores shows that difficulties can occur when waiting implemented with semaphores is combined with manipulating another data structure.

- It would be useful to have a systematic approach for handling this type of problem.