Documentation Requirements

For the Kernel Assignment:
- Description of all major components of the system, e.g. memory management, task management, context switching. Context switching should be described in detail.
- Description of kernel data structures and algorithms, e.g., task descriptors, scheduler, etc.
- Description of syscall implementation, including parameter passing.

Doc. Reqs., Cont’d

- Explain why your implementation meets real-time requirements, by giving the complexity of each kernel operation.
- Description of test cases, including that they cover what should be tested.
- User’s manual
- Tour of source code.

Hardware Interrupts

![Hardware Interrupt Diagram]
**Acronyms**

- **USART** = Universal Synchronous Asynchronous Receiver/Transmitter
- **ICU** = Interrupt Control Unit
- **RTC** = Real Time Clock
- **PIT** = Programmable Interval Timer

It bothers me that RTC and PIT are different, because of the chances for drift.

**How A Device Speaks to CPU**

1. External event occurs.
2. Device asserting interrupt asserts its interrupt line.
3. Interrupts are priority ranked by the ICU, which interrupts the CPU.
4. CPU reads IRQ (interrupt request) level from ICU data bus.
5. CPU begins interrupt processing.

**Interrupt Numbers**

- 0–31: Processor Internal (GPF, division by zero, etc.)
- 31–39: First ICU (IRQ0–IRQ7)
- 40–47: Second ICU (IRQ8–IRQ15)
- 48–255: Software Interrupts; for int n, be sure that n ≥ 48!

**To Make Interrupts Happen**

- Enable Interrupts by setting IF (Interrupt Enable Flag), which is stored in EFLAGS register.
- Instructions are:
  - STI — set IF (enable)
  - CLI — clear IF (disable)
Happening, Cont’d

- Interrupts are:
  - enabled in non-kernel tasks,
  - disabled in the kernel, and
  - enabled at boot up.
- Unmask interrupts of interest in ICUs.
- Configure each device to generate interrupts.

CPU’s response to an Interrupt

1. Push EFLAGS, including current IF.
2. Clear IF and TF (trap flag, to enable single-stepping; in single-step mode, each instruction is an interrupt).
3. Push CS
4. Push EIP
5. Load CS, EIP from IDT.

Interrupt Service Routine

1. Record interrupt number.
2. Switch into kernel.
3. Send non-specific EOI to ICUs, otherwise they won’t generate any more interrupts:
   - outb(IO_ICU1,0x20)
   - outb(IO_ICU2,0x20)

Event Abstraction

An event abstraction is the representation of an external event at the task level.

- More than one event can be associated with a physical device, e.g., as for serial input and output.
- int AwaitEvent(int EventNumber) — block and wait for an instance of the specified event to occur.
- Event may occur before int AwaitEvent is issued; therefore buffer at least one instance of each kind of event.
Event Abstraction, Cont’d

- Associate an event number with each hardware event.
- Can have also software events.
- int SignalEvent(int EventNumber) — signals an instance of the specified event, unblocking a task that is awaiting that event number.

A Possible Application of Events

Block a task until the fulfillment of a condition, but allow more than one task to fulfill the condition and then unblock the waiting task.

Server Implementation

Receive()

AwaitEvent()

Server Implementation, Cont’d

- On Receive(), server is RECEIVE_BLOCKED
- On AwaitEvent(), server is EVENT_BLOCKED

Cannot service clients while EVENT_BLOCKED.

Cannot respond to events while RECEIVE_BLOCKED.

∴, one type of event starves the other.

How can we prevent this starvation?
Noti®er

Noti®er()

while(1){
    AwaitEvent(eventNumber); /* transform event to */
    Send(Server,eventNumber,NULL); /* a message */
}

Then server needs to call only Receive(), and not
AwaitEvent().

Noti®er and clients are then serviced in the order in
which they send.

Server

Server()

Initialize(); CreateNoti®er(); RegisterAs(...);
while(1){
    (tid,msg) ← Receive();
    if (tid==Notifier){
        Reply(Notifier,NULL);
        serviceDevice();
    } else {
        serviceRequest();
    }
}
Clock Server

Delay(int t):

- Blocks caller for at least \( t \) ticks.
- A tick is 1/20 of a second.
- Implemented by sending a message to clock server.
- Clock server replies after at least \( t \) ticks.

```
Delay(int t){
    int clock = WhoIs("clockServer");
    Send(clock,(char *)&t,sizeof(t),NULL,0);
}
```

A Problem

What if ALL tasks, other than the clock server and notifier, call Delay()?

What happens between now and the next clock tick?
What Happens if All Delay?

- Kernel has no tasks to run.
- Kernel cannot wait for a hardware event to wake up a notifier, because interrupts are disabled!
∴ There needs to be a running task.

Always Running Task

Create an idle task that never blocks, and runs at the lowest priority!

IdleTask()
  while(1);

Clock Server

ClockServer()
  time = 0;
  InitializePIT();
  notifier = CreateClockNotifier();
  while(1){
    Loop Body
  }

Loop Body

(pid,request) ← Receive();
if(pid == NOTIFIER){
  time++;
  Reply(pid,NULL);
  while(nextWaitingTime() <= time){
    pid = dequeueWaitingTask();
    Reply(pid,NULL);
  }
} else /* assuming that only request is Delay */
  enqueueWaitingTask(pid,time + timeRequest);
**Loop Body, Cont’d**

This body assumes that there is only one kind of request, i.e., Delay.

If there are others, the `else` part will have to have a `case` to separate out which request it is.

---

**Clock Notifier**

```c
ClockNotifier()
{
    while(1)
    {
        AwaitEvent(PIT_EVENT);
        Send(MyParentPid(),NULL,NULL);
    }
}
```

---

**Programmable Interval Timer**

The programmable interval timer (PIT), the Intel 8253:

- Interrupt number 32
- Counter 0
- Mode 2

For interrupt number and counter, see Diagram on Page 4.

---

**Mode ?**

- Mode 1 = HW Interrupt or Exception in Virtual 8086 Mode
- Mode 2 = Maskable HW Interrupt in Virtual 8086 Mode
- Mode 3 = SW Interrupt in Virtual 8086 Mode
More Clock Primitives

int getTime() — returns the current tick count

DelayUntil(int t) — delay until a specified time t; the executing process is blocked to be awakened when tick count \( \geq t \).

These are optional in your kernel.

Delay vs DelayUntil

while (1){
    Delay(x);
    doSomething();
}

should have the same effect as

t = getTime();
while (1){
    t+=x; DelayUntil(t);
    doSomething();
}

Delay vs DelayUntil, Cont’d

but they don’t.

What’s the REAL Difference?

One Real Difference

The doSomething takes time.

\[ \therefore \text{the period in the first case is } x + \text{time (doSomething).} \]

and the period in the second case is \( x \).
Another Real Difference

Amount of delay $\geq x$, say $x+\varepsilon$.

These $\varepsilon$s accumulate under successive Delays, but ...

These $\varepsilon$s do not accumulate under successive DelayUntils.

$\therefore$, DelayUntil enforces stricter periodicity.

Scheduling Options

- time-slicing vs. run-to-completion
  - fair
  - efficient

When to Reschedule

Rescheduling when a task calls the kernel!

Pass() must reschedule!

Should interrupt currently executing task periodically, e.g., every $k$ ticks, to force rescheduling for round-robin purposes?

Preemption required when a task of a priority higher than that of the running task becomes READY due to an external event!

Serial Chip

Serial Chip, PC16550D, Universal Asynchronous Receiver/Transmitter (UART) (See Documentation from byterunner)

Registers:

- Transmit Holding Register — for reading from the serial port
- Receiver Buffer Register — for writing to the serial port
Interrupt Enable Register — for enabling and disabling interrupts

Interrupt Types:
- Received Data Available
- Transmit Holding Register Empty
- Receiver Line Status — for error conditions
- Modem Status — not needed

Interrupt Identification Register — to determine what kind of interrupt fired

Line Control Register — to initialize the chip with parity, stop bits, etc.

Line Status Register — diagnostics, e.g., ready, error conditions, etc.

ClockServer
- required
  - Delay(int t)
- optional
  - int GetTime()
  - DelayUntil(int t)

SerialServer
- required
  - byte=GetPort(port)
  - Put(byte, port)
- optional
  - write(port, buffer, length) — atomically
  - read(port, buffer, length)
  - readLine(port, buffer, length)

See Complete I/O Port List.
Serial Server

Like the producer–consumer problem, but with multiple producers and multiple consumers.

What If?

Too Many Readers

Too many readers or writers or both could starve the notifier, …

and the notifier could miss interrupts.

How can we ensure that the notifier does not miss interrupts and answers them on time?
Guard

Guard()
{
    serialServer = MyParentPid();
    while(1)
    {
        (tid,msg) ← Receive();
        replyMsg ← Send(serialServer,msg);
        Reply(tid,replyMsg);
    }
}

Should there be a delay guard for the clock server?