Today's plan

- Minix scheduling
- Interprocess communication
- Race conditions
- Locks
- Pipes
- Message passing, including in Minix
- Monitors

Minix Scheduling

- sixteen priority levels: (textbook, p. 182)
  - clock and system tasks have the highest priority (0)
  - drivers have the next highest priorities: tty (1), disk, log, mem (2)
  - system servers get the next highest priorities: rs and pm (3), fs (4)
  - user processes get the lowest priorities
- strict priority scheduling, as long as a process doesn't use up its quantum
  - higher priority processes get a bigger quantum
  - processes that use up their quantum twice in a row (without scheduling any other process in the interval) have their priority reduced
- scheduler is called on every interrupt or system call (system calls are software interrupts), and may reschedule the same process
- careful design must handle re-entrant behavior: interrupt handler and scheduler may themselves be interrupted
- interrupt handler postpones interrupt when scheduler is already active

Inter-Process Communication

- processes may depend on each other
- example: a consumer may wait for the producer to produce, or a producer may wait for the consumer to consume and free up space in the shared buffer
- IPC mechanisms may support one or more of the following:
  - synchronization: needed so process A can tell when process B has reached a certain point. Example: barrier synchronization stops all processes in a group until the last one has arrived at the barrier
  - data exchange: process A receives data from process B. May involve synchronization so process A waits for process B to complete preparing the data. Example: Unix pipes
  - mutual exclusion (a special case of synchronization): at most one process may be accessing a certain set of variables. If a process expects to be the only one accessing a given set of variables, that process is in its **critical region**

Race Conditions

- incrementing is not an atomic operation
- if:
  1. one process (A) begins to increment a memory location by reading a value into a register
  2. then A gets suspended, then
  3. another process (B) increments the memory location, then
  4. the first process writes the (now) incorrect value back to memory
- the final result should be *n+2*, but is only *n+1*
- other non-atomic operations include inserting a value into a data structure (e.g. adding a file to a print spool or adding a value to a queue) or writing contents to a file

Avoiding Race Conditions

- ***careful coding!***
- prevent execution of any other process while process A is in its critical region:
  - by disabling interrupts, or
  - by requiring all code to acquire a **mutual exclusion lock** before entering a critical region, and release it at the end of the critical region, or
  - by requiring all code to acquire a **binary semaphore**
- code that cannot acquire a lock must spin (loop)

Avoiding Race Conditions, part 2

- code that cannot acquire a lock or *down* a semaphore will be put to sleep (suspended), and woken up once the lock/semaphore is released
  - suspending uses less CPU than spinning, but takes longer to awaken a task
- setting locks or semaphores generally requires atomic operations:
  - interrupts can be disabled while the lock or semaphore is accessed, or
  - hardware atomic operations such as Test-And-Set instructions can perform the operation while guaranteeing atomicity, even in the case of multiple processors
  - more complex software-only solutions are available as well, including Peterson's algorithm
- semaphores are more general than locks, allowing up to *n* acquire operations. This can allow semaphores to be used, for example, to keep track of the number of items (or free spaces) in a buffer.

Reader and Writer Locks

- some data structures have *reader* processes and *writer* processes
- any data structure could have any number of readers if there were no writers
  - a read lock can be acquired any number of times, as long as there are no writers
- the corresponding write lock can only be acquired if there are no other readers or writers
- as an alternative:
  - writers increment a counter when they enter and exit their critical section
  - writers only enter if the counter was even
  - readers record the value of the counter at the beginning of the critical section, and only enter if the counter is even
  - readers repeat their critical section if the counter has changed by the end of the critical section
  - very efficient for writers, can be very slow (livelock) for readers

Read-Copy-Update (RCU)

- Reminder: some data structures have reader processes and writer processes
- any data structure could have any number of readers if there were no writers
- instead of having reader/writer locks,
- readers could simply notify the system that they are reading, and when they are done reading
- writers could
  - create and modify a copy of part of the data structure
  - wait for all the readers to finish reading
  - destructively update the data structure to point to the new copy
- this is called RCU, and is very efficient for readers

Pipes

- ideal for consumer-producer problem
- on Unix/Posix: pipes of bytes. In theory pipes of arbitrary data structures could also be useful
- on Unix/Posix can use arbitrary file operations, including read and write, select, and close.
- consumer reads from pipe, up to a maximum specified by the read call, may block if no data is available
- producer writes to pipe, may block if no buffer space is available

Implementation of Pipes

- in a system with very little memory, could synchronize producer and consumer to copy data from producer's buffer directly to consumer's buffer
- in practice a system buffer gives better performance than no buffer, so most systems allow the producer to write up to a certain amount of data (typically 4096 bytes), which consumer can read all at once
- a single thread that is both producer and consumer of the same pipe will block if the process tries to write more than the buffer holds

Message Passing

- can extend pipes to pass structured data rather than just bytes
- can also extend pipes to data among different processors or across a network
- this is called **message passing**
- message passing systems must answer questions of:
  - identifying receiver, e.g. a process or a mailbox (a mailbox could be accessed via a file descriptor, e.g. a socket)
  - dealing with lost messages or overflowing buffers: acknowledgements, sequence numbers
  - efficiency, avoiding copying unless absolutely necessary -- important on message-passing supercomputers

Synchronization for Message Passing

- message passing may not provide enough synchronization, so external forms of synchronization, e.g. barrier synchronization, can be used
- another form of synchronization is to have the sender **rendezvouz** (first process to reach the rendezvouz point waits for the other) with the receiver, so data can be copied directly from the sender's to the receiver's buffer
- much in common with networking, but sometimes on a single system

Minix Message Passing

- sender has a message in a buffer that it wants to give to a specific receiver
- receiver has a message buffer that it wants to fill from a given or from any sender
- first one to send or receive blocks
- blocked sender is queued on receiver's process table entry
- all "regular" (Posix) system calls are implemented by sending a message to the File System or the Process Manager server (or the INET server)
- all system calls send, then receive (sendrec), otherwise a caller could block a server forever
- message passing also used among tasks and the kernel

Monitors

- a monitor is a module (similar to a C++/Java class) with some private variables that only the monitor routines can access
- at most one monitor routine can be active at any given time
- when a second monitor routine is called, it suspends (before executing) until the first has completed
- a monitor routine never needs to be re-entrant, that is, it is guaranteed to complete before the next call to the same routine (or any other routine in the same monitor)
- Java `synchronized` classes are similar to monitors
- a monitor routine that cannot continue can wait, and another is allowed to run (and eventually signal the one that called wait)
- writing re-entrant code is hard, and errors are hard to debug, so monitors help write correct concurrent programs

Common Practice

- monitors require language support
  - e.g. Java synchronized classes
- without language support, programmers must
  - identify related global variables that are changed as individual critical regions
  - create a lock associated with each group of variables
  - acquire the lock each time before reading or modifying the variables
  - release the lock each time after reading or modifying the variables

```
lock.acquire ()
critical region
lock.release ()
```

- these operations are error-prone
- failure to release a lock may deadlock the entire system, since all processes may eventually be waiting for the lock to be released
- it is easy to forget to release a lock -- just return from within the critical region