Previous class…

- Process state transition
  - Ready, blocked, running
- Call stack
  - Calling convention
- Context switch
  - Process switch
- Interrupt vs. exception vs. signal
- Mode switch
  - Interrupt/exception handling
- `fork()` and Shell
The three basic process states and the four transitions between them

Three states: Running, Blocked, Ready
(1) Running -> Blocked: I/O; wait for a lock
(2) Blocked -> Ready: I/O is ready; the lock is acquired
(3) Ready -> Running: a ready process is picked to run
(4) Running -> Ready: time slice used up
What is a Call Stack?

A stack data structure that stores the information for the active function calls; it consists of stack frames, each of which corresponds to an active function call.
What is Execution Context? Give two examples that are part of the context

The contents of the CPU registers at any point of time. 

- **Program counter**: a register that points to the instruction to be executed
- **Call stack pointer**: a register that points to the top of the call stack, e.g., esp in x86 CPU
What is Context Switch? When does it occur?

Also called process switch. Suspending one process and resuming another.

The current process has used up its time slice, blocks (due to I/O or requesting lock), or exits
What is Mode Switch? When does it occur?

Program execution switches between different CPU modes
User -> Kernel: system calls, interrupts, exceptions
Kernel -> User: finish handling of system calls / interrupts / exceptions
Difference between Interrupts, Exceptions, and Signals. Give examples

**Interrupts:** due to hardware; e.g., timer, I/O devices

**Exceptions:** due to software; e.g., breakpoints inserted by debugger, divide-by-zero, null pointer exceptions

**Signals:** due to exception handling or signaling API calls; e.g., SIGSEGV (due to illegal memory access)
How does a computer respond to a keystroke?

**Hardware part:** an interrupt is generated by the keyboard, and CPU automatically sets the program counter to the corresponding interrupt handler (recall IDT[n])

**Software (Kernel):** the interrupt handler executes the device driver for the keyboard, which reads the character from the keyboard buffer to the kernel-space buffer

**Software (User and Kernel):** when the system call `read(STDIN)` is invoked (by a user program), the kernel copies the character from the kernel-space buffer to the user-space buffer
Previous class...

How does the system respond to divide-by zero (e.g., a/0) in user code?

1. When a divide-by-zero operation is performed, an exception (also called s/w interrupt or trap) is triggered
2. To handle the exception, execution switches to the kernel mode
3. The kernel handles the exception by generating a SIGFPE signal for the faulty process
4. After exception handling, execution switches back to the user mode
5. The signal handler is executed (the default handler is implemented and registered by libc; it terminates the process)
<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Term</td>
<td>Hangup detected on controlling terminal or death of controlling process</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Term</td>
<td>Interrupt from keyboard</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit from keyboard</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort signal from <code>abort(3)</code></td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Floating point exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Term</td>
<td>Kill signal</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Core</td>
<td>Invalid memory reference</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>13</td>
<td>Term</td>
<td>Broken pipe: write to pipe with no readers</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>Term</td>
<td>Timer signal from <code>alarm(2)</code></td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Term</td>
<td>Termination signal</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>30,10,16</td>
<td>Term</td>
<td>User-defined signal 1</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>31,12,17</td>
<td>Term</td>
<td>User-defined signal 2</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>20,17,18</td>
<td>Ign</td>
<td>Child stopped or terminated</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>19,18,25</td>
<td>Cont</td>
<td>Continue if stopped</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>17,19,23</td>
<td>Stop</td>
<td>Stop process</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>18,20,24</td>
<td>Stop</td>
<td>Stop typed at terminal</td>
</tr>
<tr>
<td>SIGTTIN</td>
<td>21,20,26</td>
<td>Stop</td>
<td>Terminal input for background process</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>22,22,27</td>
<td>Stop</td>
<td>Terminal output for background process</td>
</tr>
</tbody>
</table>

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.
Implement a simplest Shell using fork(), exec(), wait()

fork(): create a child process.
exec(): replace the execution of the current program with the designated one
wait(): suspend the calling process until any of its child process exits
Previous class…

What is a Zombie Process? Why is it harmful? How to prevent it?

When a process exits, it becomes a zombie unless it is waited for by its parent or its parent has explicitly expressed no interest in the exit state.

A zombie process occupies precious kernel resources, such as PCB.
Outline

• IPC for passing data
  – Pipe, FIFO, Message Queue, Shared Memory

• Concepts
  – Race condition, critical section, and mutual exclusion

• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions

• Synchronization without busy waiting
Pipe is an IPC technique for passing information from one process to another.

- Byte stream; no notion of “messages”
- Example:
  - Search lines that contain some word: `cat * | grep “os”`
  - # of files in a directory: `ls | wc -l`
Understand a pipe as a circular ring buffer

Conceptual view of a circular queue

Not yet processed data in FIFO order

Read position

Write position

Capacity: 64kB
pipe() - details

- int pipe(int fd[2])
  - creates a new pipe and returns two file descriptors
  - fd[0]: read end; fd[1]: write end
- If read() from a write-closed pipe: end-of-file
- If write() to a read-closed pipe: SIGPIPE -> write failure
- If write() too large a message: the message will be split into pieces with each = PIPE_BUF (4KB) by the system
- By default, blocking I/O
  - If a process attempts to read from an empty pipe, then read() will block until data is available
  - If a process attempts to write to a full pipe, then write() blocks until min(buf, PIPE_BUF = 4KB) is available
  - fcntl() can change it to non-blocking behavior
pipe(), fork(), and close() are good friends

- pipe(): create a pipe and return fd[0] and fd[1]
- fork(): child inherits fd[0] and fd[1]
- close(): close unused file descriptors
Example: passing an integer between two processes

Child:
int n = something();
write(fd[1], &n, sizeof(n));

Parent:
int n;
read(fd[0], &n, sizeof(n));

Q1: How does a writer indicate the end of data sending?
A: Either send the size in the beginning; or send an indicator char (e.g., EOF=-1) in the end

Q2: Can a writer close the pipe immediately after sending data?
A: No. read() will return 0 immediately without receiving the data in the buffer

Q3: Then how can the writer know when to close the pipe?
A: Semaphore (to be covered soon)
IPC – FIFO (also called Named Pipe)

• Unlike a pipe, a FIFO is not limited to be used between the parent and child processes
**Example of FIFO**

```c
#define FIFO_FILE "MYFIFO"

FILE *fp;
char readbuf[80];

/* Create the FIFO if it does not exist */
mkfifo(myfifo, 0666);

fp = fopen(FIFO_FILE, "r");
fgets(readbuf, 80, fp);
printf("Received string: %s\n", readbuf);
fclose(fp);
```

```c
#define FIFO_FILE "MYFIFO"

fp = fopen(FIFO_FILE, "w")
fputs("Hello!", fp);
fclose(fp);
return(0);
```
IPC – **Message queues**

Conceptually, a queue in kernel

<table>
<thead>
<tr>
<th>POSIX APIs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qid = mq_open(QUEUE_NAME, O_CREAT</td>
<td>O_RDONLY, 0644, &amp;attr);</td>
</tr>
<tr>
<td>qid = mq_open(QUEUE_NAME, O_WRONLY);</td>
<td>Open a queue</td>
</tr>
<tr>
<td>mq_send(mq, buffer, MAX_SIZE, 0)</td>
<td>Send a message; can specify the priority of the message</td>
</tr>
<tr>
<td>mq_receive(mq, buffer, MAX_SIZE, NULL)</td>
<td>Receive a message; can specify the priority also</td>
</tr>
</tbody>
</table>
IPC – Shared Memory

• A region of physical memory that is mapped to two (or more) processes, such that when a process updates the share memory the other process can see the modification immediately, and vice versa
Shared Memory

POSIX Shared Memory

Process P1

shared memory (mapped)

Process P2

shared memory (mapped)
Shared Memory

• The previous three mechanisms (Pipes, FIFOs, and Message Queues) all require mode switch when sending/receiving information (why? recall system calls), while shared memory does NOT
How to create a Shared Memory?

• shmget: System V API
• shm_open: POSIX API
• mmap + fork -> the easiest way to create a shared memory between processes
  – mmap(MAP_ANONYMOUS | MAP_SHARED)
## Compare different IPCs

<table>
<thead>
<tr>
<th>IPC method</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipes</td>
<td>Can only be used among parent and child</td>
</tr>
<tr>
<td>FIFO (named pipes)</td>
<td>Pipe is named using a string, so doesn’t have the limitation above</td>
</tr>
<tr>
<td>Message Queues</td>
<td>Supports message boundary and message types / priorities</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Data passing doesn’t go through kernel, so it is usually the most efficient one</td>
</tr>
</tbody>
</table>
Outline

• IPC for passing data
  – Pipe, FIFO, Message Queue, Shared Memory

• Concepts
  – Race condition, critical section, and mutual exclusion

• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions

• Synchronization without busy waiting
Race condition bug

- A race condition exists if the final program result depends on the execution sequence.
- Let's consider a counter, $c$, that is shared by two processes.

```
++c;
(1) movl c, %eax // copy to eax
(2) addl $1, %eax // increment
(3) movl %eax, c // copy back to c
```

- Assume $c = 0$ initially, and the two processes both execute “++c”.
- Consider the execution sequence: after process 0 executes (1) (2), it is scheduled out; and then process 1 executes (1) – (3)?
- You get $c=1$ here, while in other execution sequences you may get 2.
- A concurrent program as simple as this has a race condition bug.
Intuitive attempts to fix the bug

• Use atomic_add instructions
  – atomic_add supports you to do atomic increment
  – It works in that extremely simple example
  – But it is not a general solution; consider another example

```plaintext
// critical section: withdraw $100 from account
(1) if(account >= 100)
(2) account -= 100;
// Is it possible to withdraw $200, given account = 100?
```

• Disabling interrupts to prevent the process from being schedule out
  – Does not work well on multi-core machines
Intuitive attempts don’t work; we need a formal treatment by introducing new concepts
Critical section and mutual exclusion

- A **Critical section** is a program region that has to access shared data exclusively; otherwise, race condition may occur.
- **Mutual exclusion** is to make sure no two processes are simultaneously inside critical sections that access the same shared data.
  - Each synchronization primitive that enforce mutex provides the two APIs: `enter_cs()` / `leave_cs()`.
  - A critical section is surrounded by the two API calls.
Why is Mutual Exclusion important?

// Why is it impossible to withdraw 200 given account = 100?
enter_cs();
// critical section
if(account >= 100)
    account -= 100;
exit_cs();
Mutual Exclusion (Mutex)

• A good solution to mutual exclusion should satisfy
  – Mutual exclusion
  – No assumptions made about processor speed or number
  – No outside blocker
  – No infinite wait
A failed attempt: strict alternation

- What if process 0 quickly finishes one iteration and wants to execute another, while process 1 has not entered the critical section?
- It violates Condition 3 “no outside blocker”
Outline

• IPC for passing data
  – Pipe, FIFO, Message Queue, Shared Memory
• Concepts
  – Race condition, critical section, and mutual exclusion
• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions
• Synchronization without busy waiting
Dekker’s algorithm – First solution

The two flags indicate the intention of the two processes to enter critical sections, respectively. If both intend to enter, “turn” decides who wins.

```
//flag[] is boolean array;
flag[0] = false
flag[1] = false
turn   = 0    // or 1
```

```
P0:
   flag[0] = true;
   while (flag[1] == true) {
      if (turn != 0) {
         flag[0] = false;
         while (turn != 0) {
            // busy wait
         }
      }
      flag[0] = true;    // re-indicate intention
   }
   // critical section
   ...
   turn   = 1;
   flag[0] = false;
   // remainder section

P1:
   flag[1] = true;
   while (flag[0] == true) {
      if (turn != 1) {
         flag[1] = false;
         while (turn != 1) {
            // busy wait
         }
      }
      flag[1] = true;
   }
   // critical section
   ...
   turn   = 0;
   flag[1] = false;
   // remainder section
```
Lamport’s bakery algorithm – N processes

“A New Solution of Dijkstra's Concurrent Programming Problem”, Lamport 1974

```c
lock (int pid) {
    // enter doorway
    choosing[pid] = 1;
    number[pid] = 1+max(number[0],...,number[N-1]);
    choosing[pid] = 0;

    // enter bakery
    while(j<N) {
        if (j != pid) {
            while(choosing[j]);
            while(number[j] &&
                ((number[j] < number[pid]) ||
                ((number[j]==number[pid]) && j<pid))
            ); // spin
        }
        j++;
    }
}

unlock (int pid) { number[pid] = 0; }
```

The current process is choosing its number

Process j’s choosing its number

The process that has the smallest number wins

Number 0 is reserved to indicate the exit of critical section
Hardware-assisted solutions

test_and_set(int* p){
    int t = *p;
    *p = 1;
    return t;
}

//compare-and-swap
CAS(p, old, nvalue){
    if(*p != old)
        return false;
    *p = nvalue;
    return false;
}

Load-link/store-conditional,
or LL/SC // CAS equivalent in RISC architectures

XCHG(int *x, int *p){
    int t = *x;
    *x = *p;
    *p = t;
}

enter_region() {
    while(test_and_set(&lock) == 1); 
}

leave_region() { lock = 0;}

enter_region() {
    a = 1;
    do { XCHG(&a, &lock);
        } while (a == 1);
}

leave_region() { lock = 0;}
Outline

• IPC for passing data
  – Pipe, FIFO, Message Queue, Shared Memory
• Concepts
  – Race condition, critical section, and mutual exclusion
• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions
• Synchronization without busy waiting
  – Kernel-assisted solutions: semaphore
  – Language-assisted solutions: monitor
Semaphore – avoids busy-waiting

- Previous solutions perform busy waiting
- Semaphore
  - A counter to indicate the number of resources
  - A queue for processes waiting for the semaphore
- Semaphore needs operating system support

Semaphore – Two main APIs

// Atomic
down(S) { // also called P operation
    if(S.num > 0)
        S.num--;
    else
        put the current process in queue;
        the current process blocks;
}

// Atomic
up(S) { // also called V operation
    if(any process is in S’s wait queue)
        pop a process from the queue;
        resume this process;
    else
        S.num++;
}
Enforcing Mutex (mutual exclusion) using a binary semaphore

• If a semaphore’s counter value is restricted to 0 and 1, this semaphore is a binary semaphore
• A binary semaphore can be used to enforce mutual exclusion
  – The semaphore’s counter is initialize to 1
  – After a process X calls P(), all other process that call P() sleep in the wait queue of the semaphore
  – After X calls V(), one of the processes in the wait queue is waken up
When does Semaphore perform poor?

• Suspending and resuming processes, i.e., context switching, is expensive

• Assume a critical section contains $N$ instructions and a context switch takes $M$ instructions.

• If $N < M$, it actually saves CPU cycles to use busy waiting, compared to using Semaphore

• That is why some systems use a hybrid method
  – The process first spins for a while, and then
  – Blocks
## Semaphore and pthread_mutex in Linux

<table>
<thead>
<tr>
<th>function</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sem_open()</code></td>
<td>Create or connect to a named semaphore (and increment the reference count), which can be conveniently shared inter-process</td>
</tr>
<tr>
<td><code>sem_unlink()</code></td>
<td>Remove the named semaphore once its reference count = 0</td>
</tr>
<tr>
<td><code>sem_close()</code></td>
<td>Decrement the reference count (<em>exit</em>() does this automatically)</td>
</tr>
<tr>
<td><code>sem_init()</code></td>
<td>Create an unnamed semaphore (usually a global variable)</td>
</tr>
<tr>
<td><code>sem_destroy()</code></td>
<td>Destroy an unnamed semaphore</td>
</tr>
<tr>
<td><code>sem_wait()</code></td>
<td>P operation</td>
</tr>
<tr>
<td><code>sem_post()</code></td>
<td>V operation</td>
</tr>
<tr>
<td><code>pthread_mutex_init()</code></td>
<td>PTHREAD_PROCESS_SHARED: process-shared mutex</td>
</tr>
<tr>
<td></td>
<td>Other APIs: <code>_destroy()</code>, <code>_lock()</code>, <code>_trylock()</code>, <code>_unlock()</code></td>
</tr>
</tbody>
</table>
Monitor

• A programming language construct that supports controlled access to shared data; it encapsulates
  – Shared data
  – Procedures that operate on the shared data
  – Synchronization between concurrent processes that invoke those procedures (only one thread can execute any of the procedures at a time)

• A Monitor has a mutex lock and a queue
  – Processes has to acquire the lock before invoking a procedure in the monitor
  – If the lock has been acquired by a process, other requesting processes are put in the queue
Two Classes of Synchronization Problems

- Uniform resource usage with simple scheduling constraints
  - No other variables needed to express relationships
  - Use one semaphore for every constraint
  - Examples: thread join and producer/consumer

- Complex patterns of resource usage
  - Cannot capture relationships with only semaphores
  - Need extra state variables to record information
  - Use semaphores such that
    - One is for mutual exclusion around state variables
    - One for each class of waiting

Always try to cast problems into first, easier type.

Monitors

- A programming language construct that supports controlled access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why does this help?

- Monitor is a software module that encapsulates:
  - Shared data structures
  - Procedures that operate on the shared data
  - Synchronization between concurrent processes that invoke those procedures

Monitor protects the data from unstructured access
- Guarantees only access data through procedures, hence in legitimate ways

(1) Shared data can only be accessed through procedures
(2) Only one thread can enter the monitor to invoke procedures at a time
(1) + (2) => It is guaranteed that all threads access the shared data in a mutual exclusive way
Monitor is easy to use and fool-proof, but less flexible than semaphores

Graph credit: Michael Swift @ wisc univ
Monitor in Java

• “Synchronized” non-static methods are to implement a Monitor object
  – A thread acquires the lock of an object to execute a synchronized method
  – If another thread tries to execute any of the synchronized methods, the thread blocks

• “synchronized” static method is to implement a Monitor class

• “synchronized block” is more flexible
Problem

• Consider that, inside a monitor procedure, the execution may can’t continue unless some condition is met (and the condition satisfaction relies on another thread)
• But recall that the thread executing the procedure is occupying the lock of the monitor object; so it has to relinquish the lock to allow other threads to come in
• Seems that the thread has to exit the procedure immaturely (and re-executes it later)?
• Any better solution?
Condition Variable (CV)

- Condition variables provide such a solution
  - So that the thread can be suspended at the condition evaluation location, and resumes from where it is suspended when the condition changes
- Each condition variable correspond to a wait queue, and there are three operations
  - Wait(c): perform the following two actions atomically
    - The calling process is put in the wait queue
    - Release the monitor lock (so somebody else can get in)
  - Signal(c)
    - Wake up one process in the wait queue, and the process goes to wait in the queue of the lock
    - If the queue is empty, the signal is lost
  - Broadcast(c)
    - Wake up all processes in the wait queue
## Condition Variable

<table>
<thead>
<tr>
<th>Operations</th>
<th>Posix</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait(c)</td>
<td><code>pthread_cond_wait (cond, mutex)</code></td>
<td><code>wait()</code></td>
</tr>
<tr>
<td>Signal(c)</td>
<td><code>pthread_cond_signal(cond)</code></td>
<td><code>notify()</code></td>
</tr>
<tr>
<td>Broadcast(c)</td>
<td><code>pthread_cond_broadcast(cond)</code></td>
<td><code>notifyAll()</code></td>
</tr>
</tbody>
</table>
How to use a Condition Variable

1. Always hold the lock while performing CV operations
2. Always put the wait operation in a loop
   - In order to check whether the condition is true
   - Note the the \texttt{signal(c)} only hints the conditions changes; it does not imply that the condition is definite true. Moreover, after signaling, some other processes may affect the condition

```c
Strange\_withdraw(x)\{ 
  ... 
  pthread\_mutex\_lock(&m);
  while(account < x) // "if" will not work
    pthread\_cond\_wait(&c, &m);
  account -= x;
  pthread\_mutex\_unlock(&m);
\}
```

```c
Deposit(y){
  pthread\_mutex\_lock(&m);
  account += y;
  pthread\_cond\_signal(&c);
  pthread\_mutex\_unlock(&m);
}\}
```
“Always hold the lock while performing CV operations”, why?

• The correct use of “Wait(c)” requires that the caller holds the lock, since the semantics of this call implies the release of the lock
• If the caller of “signal(c)” does not hold the lock, there may be race conditions
  – Consider w.1 → s.1 → s.2 → w.2 => signal lost

Strange_withdraw(x) {
  ...
  pthread_mutex_lock(&m);
  while(account < x) // w.1
    pthread_cond_wait(&c, &m); // w.2
  account -= x;
  pthread_mutex_unlock(&m);
}

Deposit(y){
  // pthread_mutex_lock(&m);
  account += y; // s.1
  pthread_cond_signal(&c); // s.2
  // pthread_mutex_unlock(&m);
}
Relations between Condition Variable & Monitor

• A Monitor may contain zero or more CVs
  – Very often, procedures in Monitor rely on CVs to implement complex synchronization
  – Recall that a CV has to be used with a lock; a CV implicitly uses the lock of a Monitor, so you do not have to explicitly use a lock for employing a CV in a Monitor

• The use of CVs is not limited to Monitors
  – E.g., Pthread library provides CVs but not Monitors
Condition variable VS Semaphore

- A Semaphore has a counter and a wait queue, while a Condition Variable only has a wait queue
  - Plus, you need to initialize the counter when using a Semaphore, and the initialization value means the number of resources. A Condition Variable has no notion of “the number of resources”
- Condition Variables allow broadcast() operation, while Semaphores do not
Issues with locks (mutex and semaphore)

- You may introduce **deadlock**
- **Nobody can make progress** sometimes (when the lock owner is scheduled out)
- **Priority inversion**: a high priority thread is waiting for the lock occupied by a low priority thread, which is then preempted by a medium priority thread
- **Solution**: non-blocking programming / data structures (but they are very complex and, very often, inefficient)
Sync. Primitives

Busy-waiting

Software solutions (Dekker’, Bakery, etc.)

Hardware-assisted solutions (based on atomic read-modify-write instructions)

Semaphore: it contains an internal counter indicating the number of resources.

Binary Semaphore is a special semaphore, whose counter value can only be 0 or 1; it can be used as a mutex

Monitor: a high-level synchronization construct; shared data + procedures + lock

Condition variable: guarded by a lock to wait on some event

Blocking
Take away…

• IPC
  – Pipes, FIFO, message queues, and shared memory
• Concepts
  – Race condition, critical section, mutual exclusion
• Pure software based locks
  – Dekker’s, Peterson’s, Bakery algorithms
• Atomic read-modify-write (RMW) instructions
  – Test-and-set, xchg, compare-and-swap
  – Mutex based on (RMW)
• Semaphores and binary semaphores
• Monitors and Conditional variables
• Lock-free concurrent computing