

Outline for Today

- Objectives:
 - To define the **process** and **thread abstractions**.
 - To briefly introduce **mechanisms** for implementing processes (threads).
 - To introduce the **critical section problem**.
 - To learn how to reason about the correctness of concurrent programs.
- Administrative details:
 - Groups are listed on the web site, linked off main page under Communications.

1

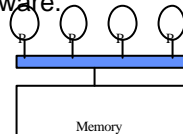
The Basics of Processes

- Processes are the *OS-provided abstraction* of multiple tasks (including user programs) executing concurrently.
- One instance of a program (which is only a passive set of bits) *executing* (implying an execution context – register state, memory resources, etc.)
- OS schedules processes to share CPU.

2

Why Use Processes?

- To capture naturally concurrent activities within the structure of the programmed system.
- To gain speedup by overlapping activities or exploiting parallel hardware.
 - From DMA to multiprocessors



3

Separation of Policy and Mechanism

- “Why and What” vs. “How”
- Objectives and strategies vs. data structures, hardware and software implementation issues.
- **Process abstraction vs. Process machinery**

Can you think of examples?

4

Process Abstraction

- Unit of scheduling
- One (or more*) sequential threads of control
 - program counter, register values, call stack
- Unit of resource allocation
 - address space (code and data), open files
 - sometimes called *tasks* or *jobs*
- Operations on processes: fork (clone-style creation), wait (parent on child), exit (self-termination), signal, kill.

[Process-related System Calls in Unix.](#)

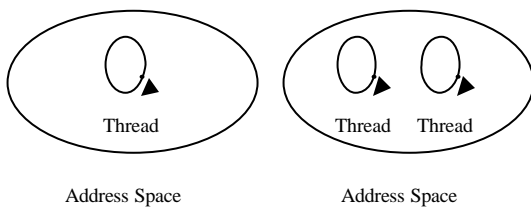
5

Threads and Processes

- Decouple the resource allocation aspect from the control aspect
- Thread abstraction - defines a single sequential instruction stream (PC, stack, register values)
- Process - the resource context serving as a “container” for one or more threads (shared address space)
- Kernel threads - unit of scheduling (kernel-supported thread operations → still slow)

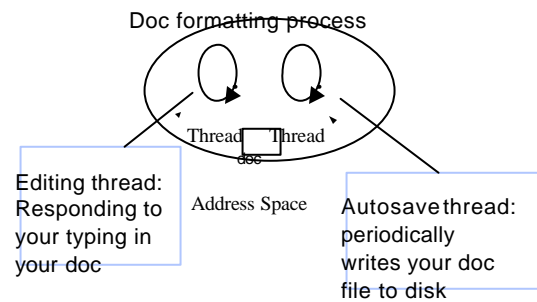
6

Threads and Processes



7

An Example



8

User-Level Threads

- To avoid the performance penalty of kernel-supported threads, implement at user level and manage by a run-time system
 - Contained “within” a single kernel entity (process)
 - Invisible to OS (OS schedules their container, not being aware of the threads themselves or their states). Poor scheduling decisions possible.
- User-level thread operations can be 100x faster than kernel thread operations, but need better integration / cooperation with OS.

9

Process Mechanisms

- PCB data structure in kernel memory represents a process (allocated on process creation, deallocated on termination).
- PCBs reside on various **state queues** (including a different queue for each “cause” of waiting) reflecting the process’s state.
- As a process executes, the OS moves its PCB from queue to queue (e.g. from the “waiting on I/O” queue to the “ready to run” queue).

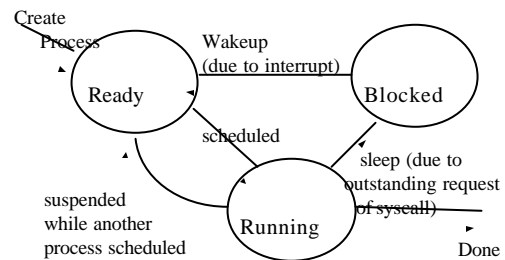
10

Context Switching

- When a process is running, its program counter, register values, stack pointer, etc. are contained in the hardware registers of the CPU. The process has direct control of the CPU hardware for now.
- When a process is not the one currently running, its current register values are saved in a process descriptor data structure (**PCB - process control block**)
- Context switching involves moving state between CPU and various processes’ PCBs by the OS.

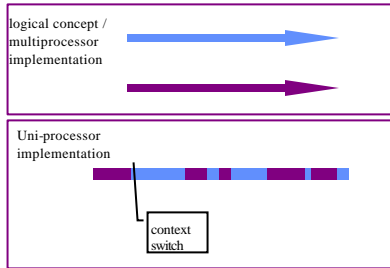
11

Process State Transitions



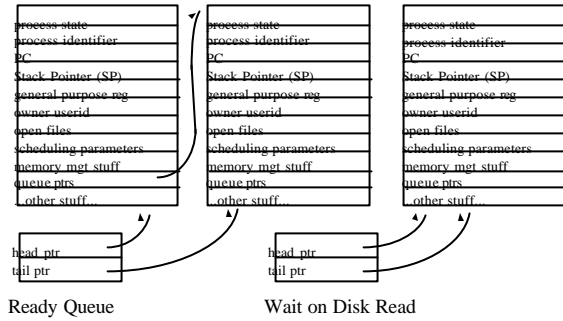
12

Interleaved Schedules



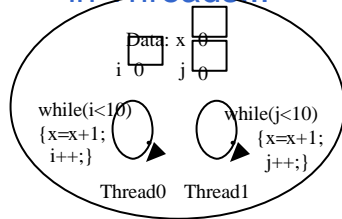
13

PCBs & Queues



14

The Trouble with Concurrency in Threads...



What is the value of x when both threads leave this while loop?

15

Nondeterminism

- What unit of work can be performed without interruption? **Indivisible** or **atomic** operations.
- **Interleavings** - possible execution sequences of operations drawn from all threads.
- **Race condition** - final results depend on ordering and may not be "correct".

```
while (i<10) {x=x+1; i++;}
load value of x into reg
yield ()
add 1 to reg
yield ()
store reg value at x
yield ()
```

16

Reasoning about Interleavings

- On a uniprocessor, the possible execution sequences depend on when context switches can occur
 - Voluntary context switch - the process or thread explicitly yields the CPU (blocking on a system call it makes, invoking a Yield operation).
 - Interrupts or exceptions occurring - an asynchronous handler activated that disrupts the execution flow.
 - Preemptive scheduling - a timer interrupt may cause an involuntary context switch at any point in the code.
- On multiprocessors, the ordering of operations on shared memory locations is the important factor.

17

Critical Sections

- If a sequence of non-atomic operations must be executed *as if* it were atomic in order to be correct, then we need to provide a way to constrain the possible interleavings in this **critical section** of our code.
 - Critical sections are code sequences that contribute to “bad” race conditions.
 - Synchronization needed around such critical sections.
- **Mutual Exclusion** - goal is to ensure that critical sections execute atomically w.r.t. related critical sections in other threads or processes.
 - How?

18

The Critical Section Problem

Each process follows this template:

```
while (1)
{ ...other stuff... //processes in here shouldn't stop others
  enter_region( );
  critical_section
  exit_region( );
}
```

The problem is to define enter_region and exit_region to ensure mutual exclusion with some degree of fairness.

19

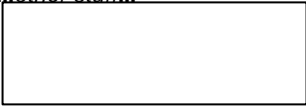

Implementation Options for Mutual Exclusion

- Disable Interrupts
 - Busywaiting solutions - spinlocks
 - execute a tight loop if critical section is busy
 - benefits from specialized atomic (read-mod-write) instructions
 - Blocking synchronization
 - sleep (enqueued on wait queue) while C.S. is busy
- Synchronization primitives (abstractions, such as locks) which are provided by a system may be implemented with some combination of these techniques.

20

The Critical Section Problem

```

while (1)
{
  ...other stuff...
  
  critical section
  exit_region( );
  
}

```

21

Proposed Algorithm for 2 Process Mutual Exclusion

```

Boolean flag[2];
proc (int i) {
  while (TRUE){
    compute;
    flag[i] = TRUE ;
    while(flag[(i+1) mod 2]) ;
    critical section;
    flag[i] = FALSE;
  }
}

```

```

flag[0] = flag[1]= FALSE;
fork (proc, 1, 0);
fork (proc, 1,1);
Is it correct?

```

Assume they go lockstep.
Both set their own flag to TRUE. Both busywait forever on the other's flag -> deadlock.

22

Proposed Algorithm for 2 Process Mutual Exclusion

- enter_region:


```

needin [me] = true;
turn = you;
while (needin [you] && turn == you) {no_op};

```
- exit_region:


```

needin [me] = false;

```

Is it correct?

23

Interleaving of Execution of 2 Threads (blue and green)

<pre> enter_region: needin [me] = true; turn = you; while (needin [you] && turn == you) {no_op}; <i>Critical Section</i> exit_region: needin [me] = false; </pre>	<pre> enter_region: needin [me] = true; turn = you; while (needin [you] && turn == you) {no_op}; <i>Critical Section</i> exit_region: needin [me] = false; </pre>
---	---

24

```

needin [blue] = true;
needin [green] = true;
turn = green;
turn = blue;
while (needin [green] && turn == green)
Critical Section
while (needin [blue] && turn == blue){no_op};
while (needin [blue] && turn == blue){no_op};
needin [blue] = false;
while (needin [blue] && turn == blue)
Critical Section
needin [green] = false;

```

25

Greedy Version (turn = me)

```

needin [blue] = true;
needin [green] = true;
turn = blue;
while (needin [green] && turn == green)
Critical Section
turn = green;
while (needin [blue] && turn == blue)
Critical Section
Ooops!

```

26

Synchronization

- We illustrated the dangers of race conditions when multiple threads execute instructions that interfere with each other when interleaved.
- Goal in solving the critical section problem is to build synchronization so that the sequence of instructions that can cause a race condition are executed **AS IF** they were indivisible (just appearances)
- "Other stuff" can be interleaved with critical section code as well as the enter_region and exit_region protocols, but it is deemed OK.

27

Peterson's Algorithm for 2 Process Mutual Exclusion

- enter_region:


```

needin [me] = true;
turn = you;
while (needin [you] && turn == you) {no_op};

```
- exit_region:


```

needin [me] = false;

```

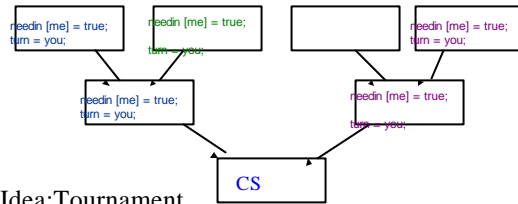
What about more than 2 processes?

28

Can we extend 2-process algorithm to work with n processes?

29

Can we extend 2-process algorithm to work with n processes?



Idea: Tournament

Details: Bookkeeping (left to the reader)

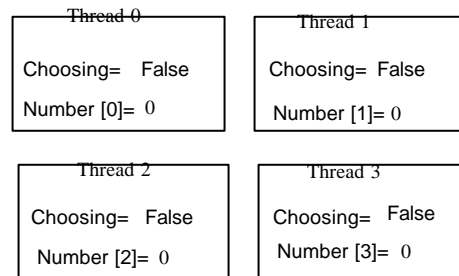
30

Lamport's Bakery Algorithm

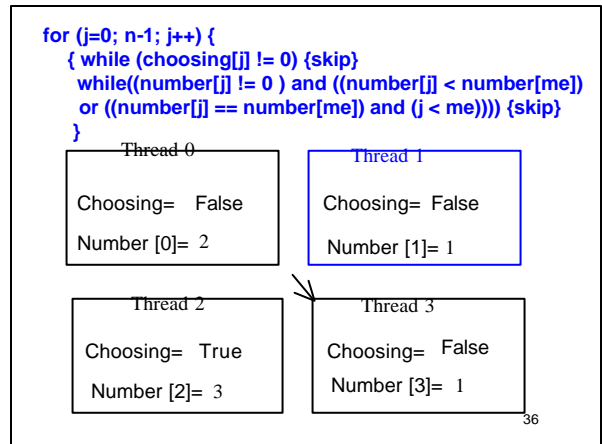
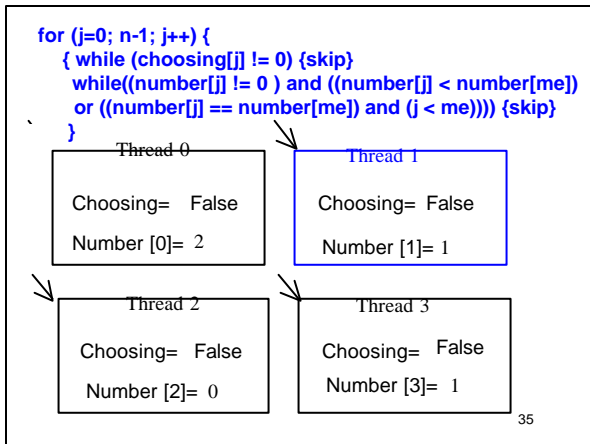
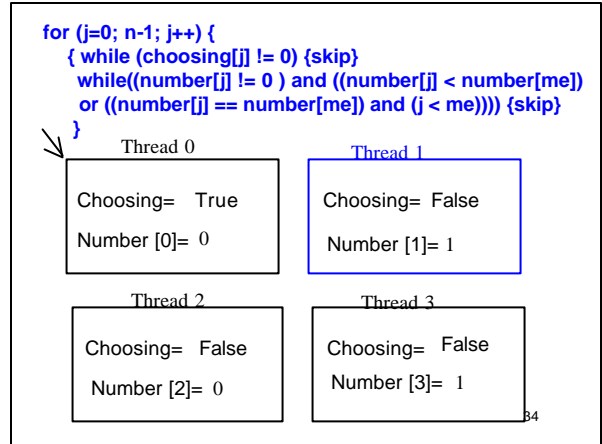
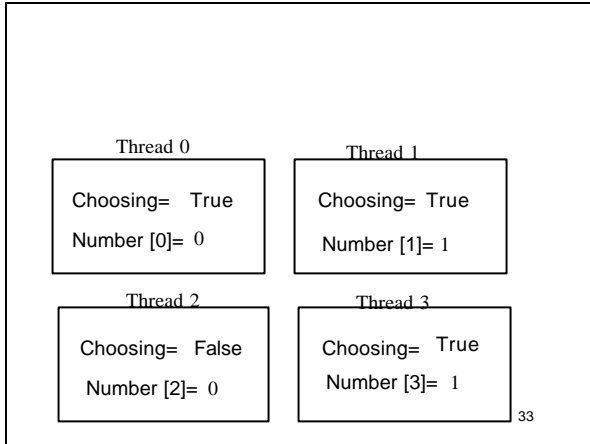
- **enter_region:**
`choosing[me] = true;`
`number[me] = max(number[0:n-1]) + 1;`
`choosing[me] = false;`
 for (j=0; j < n; j++) {
 { while (choosing[j] != 0) {skip}
 while((number[j] != 0) and ((number[j] < number[me])
 or ((number[j] == number[me]) and (j < me)))) {skip}
 }
 }
- **exit_region:**
`number[me] = 0;`

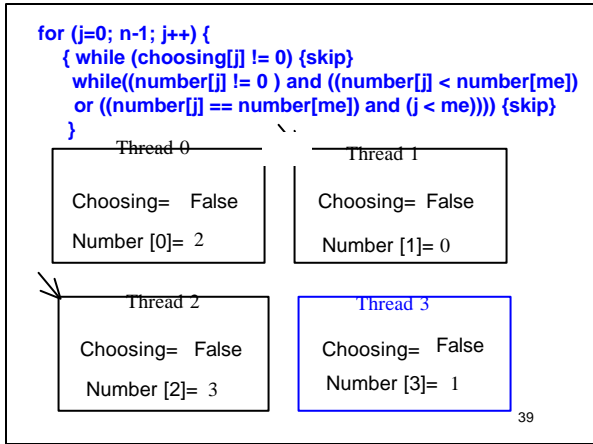
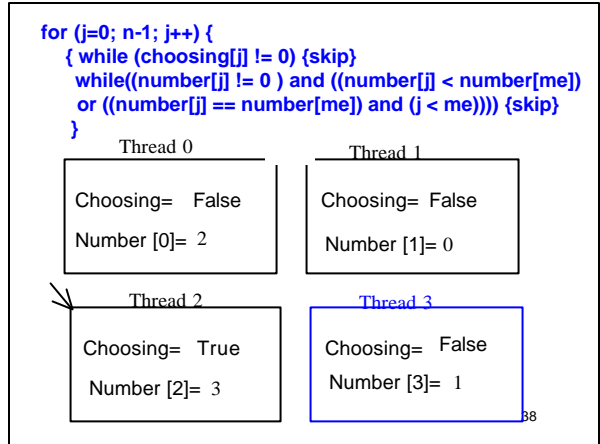
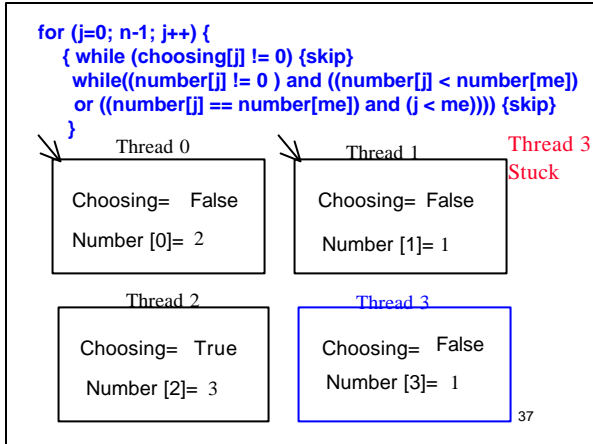
31

Interleaving / Execution Sequence with Bakery Algorithm



32





Hardware Assistance

- Most modern architectures provide some support for building synchronization: atomic **read-modify-write** instructions.
- Example: **test-and-set (loc, reg)**
 [sets bit to 1 in the new value of loc;
 returns old value of loc in reg]
- Other examples: *compare-and-swap, fetch-and-op*

[] notation means atomic

40

Busywaiting with Test-and-Set

- Declare a shared memory location to represent a *busyflag* on the critical section we are trying to protect.
- enter_region (or *acquiring* the “lock”):

```
waitloop: tsl busyflag, R0 // R0 = busyflag; busyflag = 1
          bnz R0, waitloop // was it already set?
```
- exit region (or *releasing* the “lock”):

```
busyflag = 0
```

41

Pros and Cons of Busywaiting

- Key characteristic - the “waiting” process is actively executing instructions in the CPU and using memory cycles.
- Appropriate when:
 - High likelihood of finding the critical section unoccupied (don't take context switch just to find that out) or estimated wait time is very short
- Disadvantages:
 - Wastes resources (CPU, memory, bus bandwidth)

42

Blocking Synchronization

- OS implementation involving changing the state of the “waiting” process from running to blocked.
- Need some synchronization abstraction known to OS - provided by system calls.
 - mutex locks with operations acquire and release
 - semaphores with operations P and V (down, up)
 - condition variables with wait and signal

43

Template for Implementing Blocking Synchronization

- Associated with the lock is a memory location (busy) and a queue for waiting threads/processes.
- Acquire syscall:

```
while (busy) {enqueue caller on lock's queue}
/* upon waking to nonbusy lock*/ busy = true;
```
- Release syscall:

```
busy = false;
/* wakeup */ move any waiting threads to Ready queue
```

44

Pros and Cons of Blocking

- Waiting processes/threads don't consume resources
- Appropriate: when the cost of a system call is justified by expected waiting time
 - High likelihood of contention for lock
 - Long critical sections
- Disadvantage: OS involvement
 - > overhead