Lecture 34: Synchronization and Communication	Problem From Last Time	
	 Simultaneous operations on data from two different programs can cause incorrect (even bizarre) behavior. 	
	• Example: In	
	Program #1Program #2balance = balance + depositbalance = balance + deposit	
	both programs can pick up the old value of deposit before either of them has incremented it. One deposit is lost.	
	• We define the desired outcomes as those that would happen if with- drawals happened sequentially, in <i>some</i> order.	
	 The nondeterminism as to which order we get is acceptable, but results that are inconsistent with both orderings are not. 	
	 These latter happen when operations overlap, so that the two pro- cesses see inconsistent views of the account. 	
	 We want the withdrawal operation to act as if it is atomic—as if, once started, the operation proceeds without interruption and with- out any overlapping effects from other operations. 	
Last modified: Thu Apr 24 17:15:50 2014 CS61A: Lecture #34 1	Last modified: Thu Apr 24 17:15:50 2014 CS61A: Lecture #34 2	
One Solution: Critical Sections	Aside: Context managers	
 Some programming languages (e.g., Java) have special syntax for this. In Python, we can arrange something like this: 	• The with statement may be used for anything that requires estab- lishing a (temporary) <i>local context</i> for doing some action.	
<pre>manager = CriticalSection() def with decomposition</pre>	• A common use: files:	
<pre>def withdraw(amount): with manager: if amount > selfbalance: if amount > selfbalance:</pre>	<pre>with open(input_name) as inp, open(output_name, "w") as out: out.write(inp.read()) # Copy from input to output</pre>	
<pre>raise ValueError("insufficient funds") else:</pre>	• inp and out are local names for two files created by open.	
<pre>selfbalance -= amount return self. balance</pre>	• File objects happen to have <u>enter</u> and <u>exit</u> methods.	
• The with construct essentially does this:	 Theexit method on files closes them. 	
<pre>managerenter() try:</pre>	 Thus, the program above is guaranteed to close all its files, no mat- ter what happens. 	
if amount > selfbalance:	• [End of Aside]	
<pre>finally: managerexit()</pre>		
 Idea is that our <i>CriticalSection</i> object should let just one process through at a time. How? 		

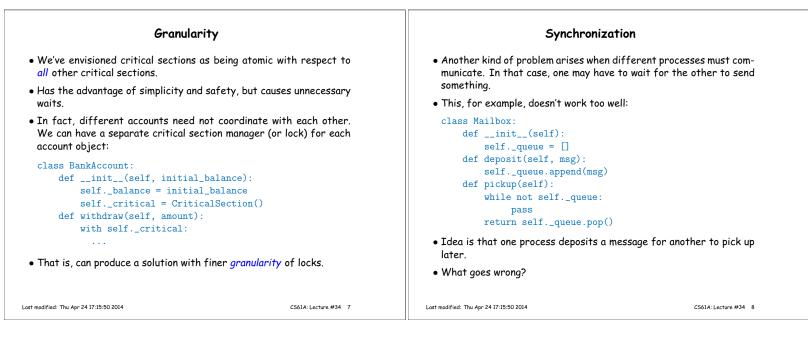
Last modified: Thu Apr 24 17:15:50 2014

CS61A: Lecture #34 3

Last modified: Thu Apr 24 17:15:50 2014

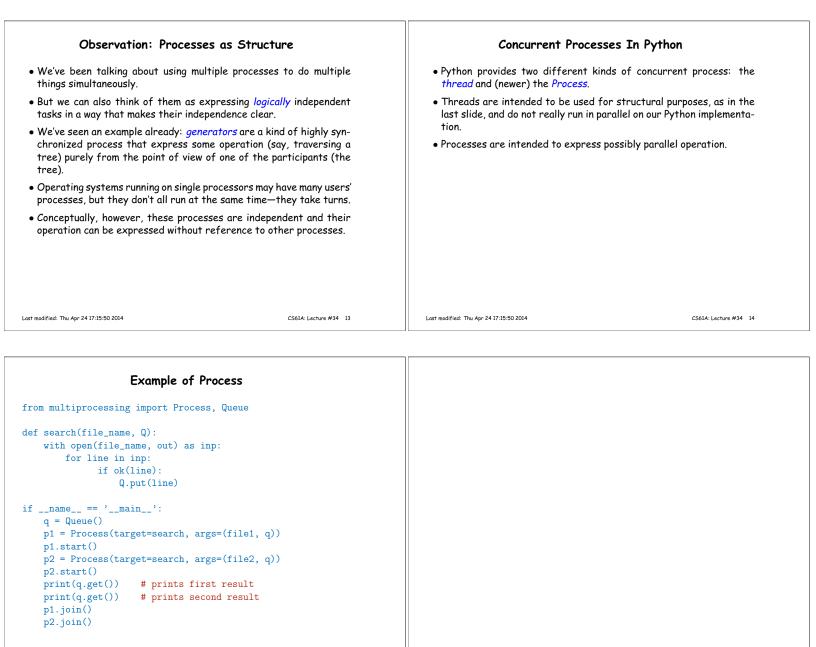
CS61A: Lecture #34 4

Locks	Implementing Critical Regions
 To implement our critical sections, we'll need some help from the operating system or underlying hardware. A common low-level construct is the <i>lock</i> or <i>mutex</i> (for "mutual exclusion"): an object that at any given time is "owned" by one process. If L is a lock, then L.acquire() attempts to own L on behalf of the calling process. If someone else owns it, the caller <i>waits</i> for it to be release. L.release() relinquishes ownership of L (if the calling process owns it). 	 Using locks, it's easy to create the desired context manager: from threading import Lock class CriticalSection: definit(self): selflock = Lock() defenter(self): selflock.acquire() defexit(self, exception_type, exception_val, traceback): selflock.release() CriticalSectionManager = CriticalSection() The extra arguments toexit provide information about the exception, if any, that caused the with body to be exited. (In fact, the bare Lock type itself already hasenter andexit procedures, so you don't really have to define an extra type).
Last modified: Thu Apr 24 17:15:50 2014 C\$61A: Lecture #34 5	Last modified: Thu Apr 24 17:15:50 2014 CS61A: Lecture #34 6



 class Mailbox: definit(self): selfqueue = [] def deposit(self, msg): selfqueue.append(msg) def pickup(self): while not selfqueue.pop() Inconsistency: Two processes picking up mail can find the queue occupied simultaneously, but only one will succeed in picking up mail, and the other will get exception. Busy-waiting: The loop that waits for a message uses up processor time. Deadlock: If one is running two logical processes on one processor, busy-waiting can lead to nobody making any progress. Starvation: Even without busy-waiting one process can be shut out from ever getting mail. Conditions act like locks with methods wait, notify (and others). wait releases the lock, waits for someone to call notify, and then reacquires the lock. 	Problems with the Naive Mailbox	Conditions	
	<pre>definit(self): selfqueue = [] def deposit(self, msg): selfqueue.append(msg) def pickup(self): while not selfqueue: pass return selfqueue.pop() • Inconsistency: Two processes picking up mail can find the queue occupied simultaneously, but only one will succeed in picking up mail, and the other will get exception. • Busy-waiting: The loop that waits for a message uses up processor time. • Deadlock: If one is running two logical processes on one processor, busy-waiting can lead to nobody making any progress. • Starvation: Even without busy-waiting one process can be shut out from ever getting mail.</pre>	<pre>from threading import Condition class Mailbox: definit(self): selfqueue = [] selfcondition = Condition() def deposit(self, msg): with selfcondition: selfqueue.append(msg) selfcondition.notify() def pickup(self): with selfcondition: while not selfqueue: selfcondition.wait() return selfqueue.pop() e Conditions act like locks with methods wait, notify (and others). wait releases the lock, waits for someone to call notify, and then reacquires the lock.</pre>	

Another Approach: Messages		Rendezvous	
 Turn the problem inside out: instead of client pro how to coordinate their operations on data, let the its actions. 		 Following ideas from C.A.R Hoare, of a rendezvous for this purpose: 	the Ada language used the notion
 From the Mailbox's perspective, things look like thi 	5:	task type Mailbox is entry deposit(Msg: String);	
<pre>selfqueue = [] while True: wait for a request, R, to deposit or pickup</pre>		entry pickup(Msg: out Strin end Mailbox;	1g);
if R is a deposit of msg: selfqueue.append(msg)		task body Mailbox is Queue:	
<pre>send back acknowledgement elif selfqueue and R is a pickup: msg = selfqueue.pop()</pre>		begin loop	
send back msgFrom a bank account's:		select accept deposit(Msg: or when not Queue.empt	<pre>String) do Queue.append(Msg); end; ty =></pre>
while True:		· · · · · · · · · · · · · · · · · · ·	out String) do Queue.pop(Msg); end;
wait for a request, R, to deposit or withdraw		end select;	
if R is a deposit of d:		end loop;	
<pre>self.balance += d elif R is a withdrawal of w:</pre>		end;	
self.balance -= W Last modified: Thu Apr 24 17:15:50 2014	CS61A: Lecture #34 11	Last modified: Thu Apr 24 17:15:50 2014	CS61A: Lecture #34 12



```
Last modified: Thu Apr 24 17:15:50 2014
```

CS61A: Lecture #34 15