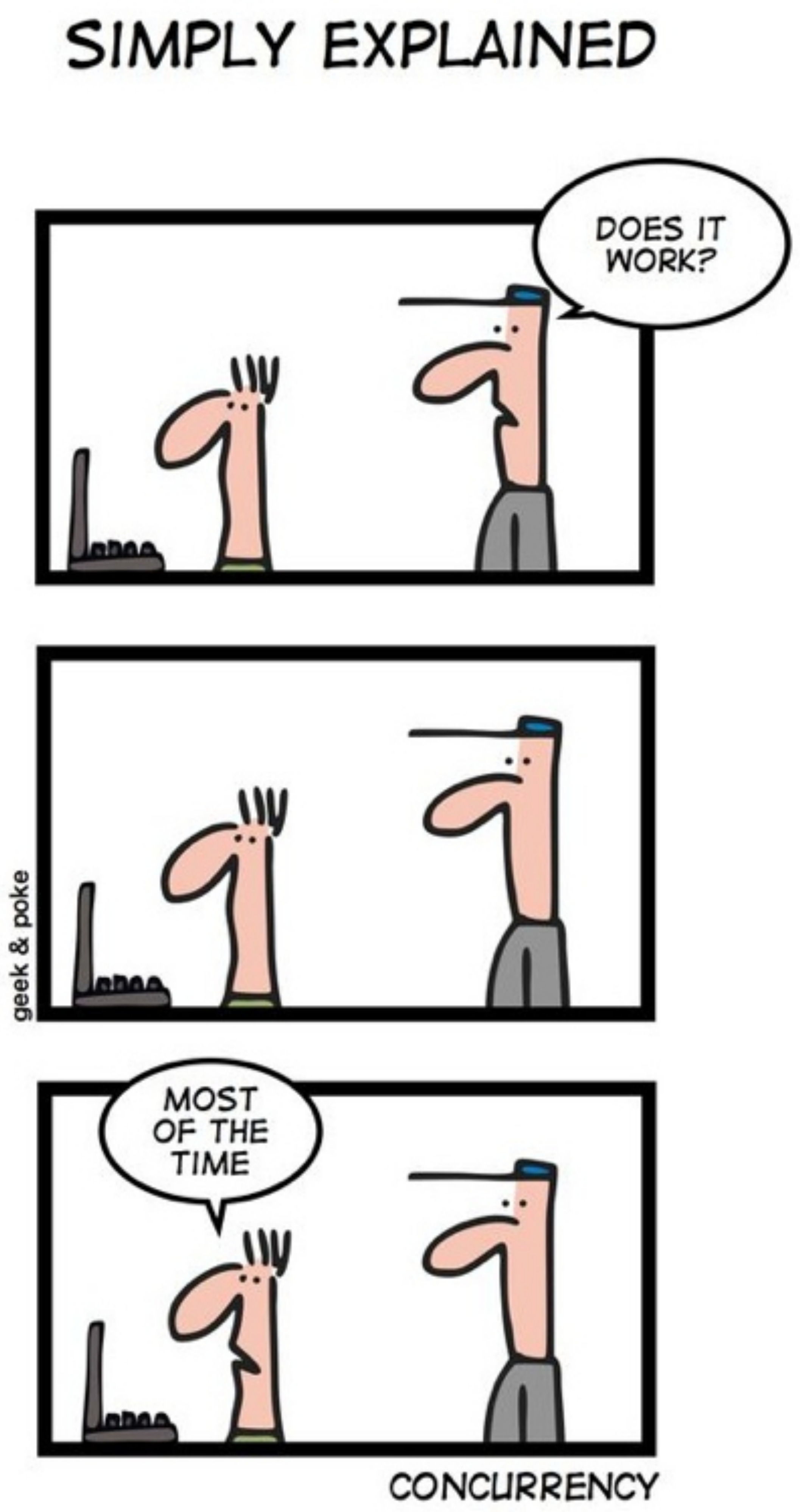
Concurrency (Part I):
Mutual Exclusion, Synchronization, Deadlock, and Starvation

Professor Travis Peters
CSCI 460 Operating Systems
Fall 2019



—http://www.datamation.com/news/tech-comics-quantum-physics-2.html

Some slides & figures adapted from Stallings instructor resources.

Some slides adapted from Adam Bates's F'18 CS423 course @ UIUC
https://courses.engr.illinois.edu/cs423/sp2018/schedule.html

Goals for Today

Learning Objectives

- Dive into core topics in concurrency
- Discuss common mechanisms for achieving mutual exclusion & synchronization

Announcements

- Homework 2 (Chapters 3-4) out later today
- *Coming Soon…*
  - 1st Programming Assignment (Concurrency)
  - Homework 3 (Chapters 5-6)
  - Exam will be held *in-class* next week

Concurrency—*What is it? & Why is it?*

Concurrency is all about managing shared resources whilst interleaving & overlapping execution.

Recall…
Thread = a single (separately schedulable) execution sequence

Servers
Multiple connections handled simultaneously

Parallel Programs
To achieve better performance

Programs w/ User Interfaces
To acheive user responsiveness while doing computation

Network & Disk Bound Programs
To hide network/disk latency

Concurrency—*Why is it challenging?*

Concurrency is all about managing shared resources whilst interleaving & overlapping execution.

Difficult for OS to manage resources in an **optimal** way…
Difficult to **debug** programing errors (non-deterministic and hard to reproduce)…

Programmer vs. Processor View

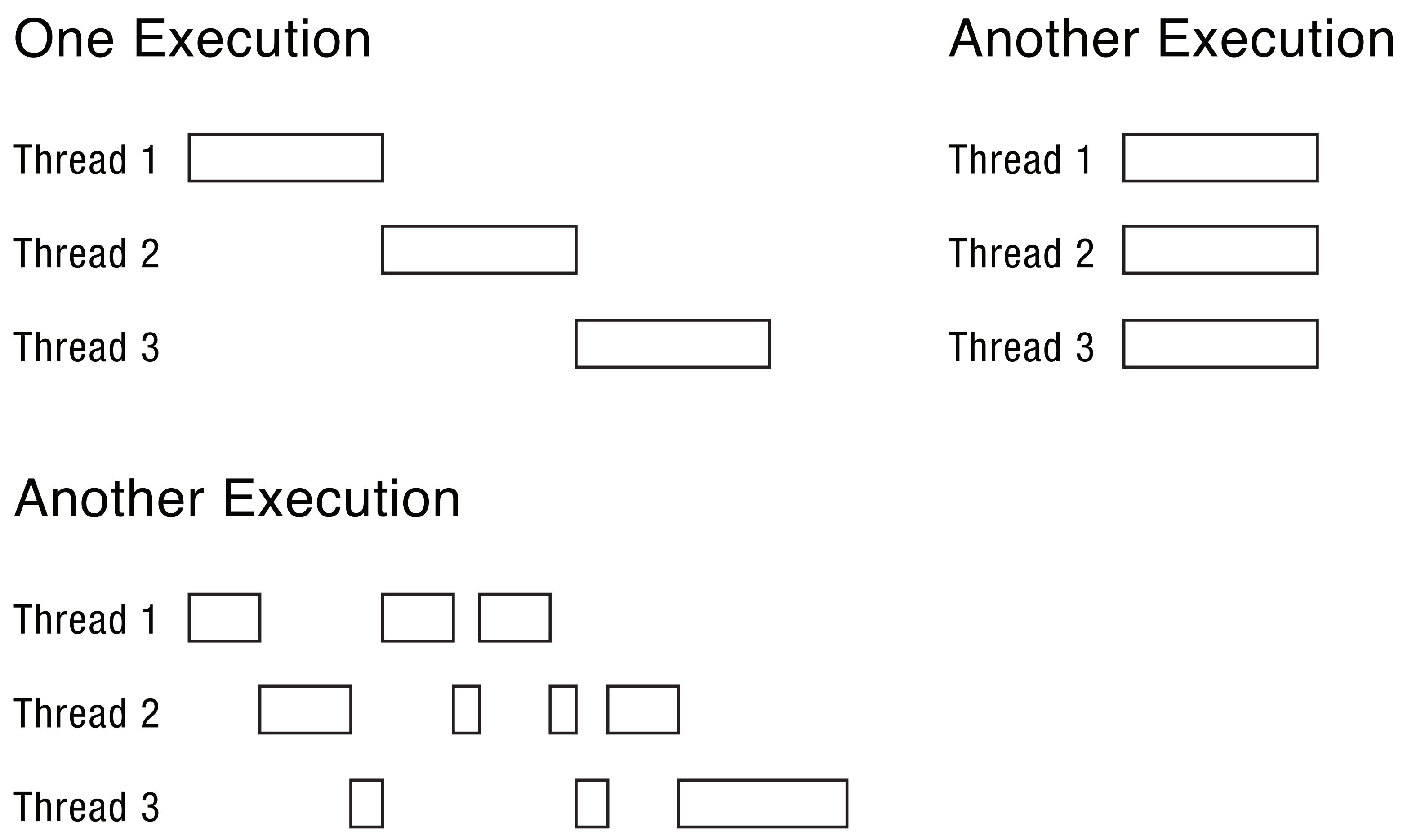
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3

.
.
.
`x = x + 1;`
`y = y + x;`
`z = x + 5y;`
.
.
.
	.
	.
	.
	`x = x + 1;`
	`y = y + x;`
	`z = x + 5y;`
	.
	.
	.
		.
		.
		.
		`x = x + 1;`
		....
		Thread is suspended.
		Other thread(s) run.
		Thread is resumed.
		....
		`y = y + x;`
		`z = x + 5y;`
			.
			.
			.
			`x = x + 1;`
			`y = y + x;`
			....
			Thread is suspended.
			Other thread(s) run.
			Thread is resumed.
			....
			`z = x + 5y;`

Program must anticipate all of these possible executions!

Possible Executions

<u>**Processor View**</u>



(Keep this in mind for later when we discuss scheduling)

Some Key Terminology Related to Concurrency

atomic operation—A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.

critical section—A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.

deadlock—A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

livelock—A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

mutual exclusion—The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

race condition—A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

starvation—A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Mutual Exclusion—*What is it? & What are the requirements?*

- Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:
  1. Mutual exclusion must be enforced: ***only one process at a time is allowed into its critical section***, among all processes that have critical sections ***for the same resource or shared object***
  2. A process that ***halts*** must do so without interfering with other processes
  3. It must not be possible for a process requiring access to a critical section to be ***delayed indefinitely***: no deadlock or starvation
  4. When ***no process is in a critical section***, any process that request entry to its critical section must be permitted to enter without delay
  5. ***No assumptions*** are made about ***relative process speeds*** or ***number of processes***
  6. A process remains inside its ***critical section for a finite time only***

Solutions?

Solutions for Mutual Exclusion

*To acheive correct & meaningful solutions to concurrency problems, **mutual exclusion** is a must!*

Software Support (today)

- *Assume elementary mutual exclusion at the memory access level; serialized by "memory arbiter"*
- Decker's Algorithm, Peterson's Algorithm

Hardware Support (today-ish)

- Interrupt Disabling
  - **Disadvantages:** inhibits processor's ability to interleave processes; doesn't work across processors.
- Special Instructions
  - **Compare&Swap:** compare values => if values are the same, swap!
  - **Exchange (XCHG):** exchanges the contents of a register w/ that of a memory location
  - **Advantages:** simple & easy to implement; can be used on multi-processor machines
  - **Disadvantages:** possibly expensive busy-waiting; starvation & deadlock are still possibe

Programming Language Mechanisms

- Semaphores, Mutex (Lock), Condition Variables, Monitors, …oh my!

Mutual Exclusion—*Workings towards a solution…*

Mutual Exclusion—*Workings towards a solution…*

/* PROCESS 0 /*	/* PROCESS 1 */
• • **while** (turn != 0) /* do nothing */ ; /* critical section*/; turn = 1; •	• • **while** (turn != 1) /* do nothing */; /* critical section*/; turn = 0; •

(a) First attempt

Attempt 1: Dependent Turn-taking (1 flag)

- If value of `turn` == process #, process can proceed
- What if one process takes a long turn?

Mutual Exclusion—*Workings towards a solution…*

```
/* PROCESS 0 /*                 /* PROCESS 1 */

•                               •
•                               •
while (turn != 0)               while (turn != 1)
   /* do nothing */ ;              /* do nothing */;
/* critical section*/;          /* critical section*/;
turn = 1;                       turn = 0;
•                               •
```

(a) First attempt

Attempt 1: Dependent Turn-taking (1 flag)

- If value of `turn` == process #, process can proceed
- What if one process takes a long turn?

```
/* PROCESS 0 */                 /* PROCESS 1 */

•                               •
•                               •
while (flag[1])                 while (flag[0])
   /* do nothing */;               /* do nothing */;
flag[0] = true;                 flag[1] = true;
/*critical section*/;           /* critical section*/;
flag[0] = false;                flag[1] = false;
•                               •
```

(b) Second attempt

**while other process is not in the critical section*

Attempt 2: Independent Turn-taking (2+ flags)

- Each process can proceed independently*
- What if one process fails in critical section?

Mutual Exclusion—*Workings towards a solution…*

```
/* PROCESS 0 /*                 /* PROCESS 1 */
•                               •
•                               •
while (turn != 0)               while (turn != 1)
   /* do nothing */ ;              /* do nothing */;
/* critical section*/;          /* critical section*/;
turn = 1;                       turn = 0;
•                               •
```

(a) First attempt

Attempt 1: Dependent Turn-taking (1 flag)

- If value of `turn` == process #, process can proceed
- What if one process takes a long turn?

```
/* PROCESS 0 */                 /* PROCESS 1 */
•                               •
•                               •
while (flag[1])                 while (flag[0])
   /* do nothing */;               /* do nothing */;
flag[0] = true;                 flag[1] = true;
/*critical section*/;           /* critical section*/;
flag[0] = false;                flag[1] = false;
•                               •
```

(b) Second attempt

**while other process is not in the critical section*

Attempt 2: Independent Turn-taking (2+ flags)

- Each process can proceed independently*
- What if one process fails in critical section?
- **=> Also, *possibly incorrect! Mutual exclusion broken! (TOCTOU)***

```
Problematic Sequence...
P0 executes the while statement and finds flag[1] set to false
P1 executes the while statement and finds flag[0] set to false
P0 sets flag[0] to true and enters its critical section
P1 sets flag[1] to true and enters its critical section
```

Mutual Exclusion—*Workings towards a solution…*

Attempt 1: Dependent Turn-taking (1 flag)

- If value of `turn` == process #, process can proceed
- What if one process takes a long turn?

Attempt 2: Independent Turn-taking (2+ flags)

- Each process can proceed independently*
- What if one process fails in critical section?
- **=> Also, *possibly incorrect! Mutual exclusion broken! (TOCTOU)***

Attempt 3: Set THEN check

- Mutual exclusion fixed…
- What if both processes set flags to `true`?

```
Problematic Sequence...
P0 executes the while statement and finds flag[1] set to false
P1 executes the while statement and finds flag[0] set to false
P0 sets flag[0] to true and enters its critical section
P1 sets flag[1] to true and enters its critical section
```

```
/* PROCESS 0 /*                  /* PROCESS 1 */
•                                •
•                                •
while (turn != 0)                while (turn != 1)
   /* do nothing */ ;               /* do nothing */;
/* critical section*/;           /* critical section*/;
turn = 1;                        turn = 0;
•                                •
```

(a) First attempt

```
/* PROCESS 0 */                  /* PROCESS 1 */
•                                •
•                                •
while (flag[1])                  while (flag[0])
   /* do nothing */;                /* do nothing */;
flag[0] = true;                  flag[1] = true;
/*critical section*/;            /* critical section*/;
flag[0] = false;                 flag[1] = false;
•                                •
```

(b) Second attempt

while other process is not in the critical section

```
/* PROCESS 0 */                  /* PROCESS 1 */
•                                •
•                                •
flag[0] = true;                  flag[1] = true;
while (flag[1])                  while (flag[0])
   /* do nothing */;                /* do nothing */;
/* critical section*/;           /* critical section*/;
flag[0] = false;                 flag[1] = false;
•                                •
```

(c) Third attempt

Mutual Exclusion—*Workings towards a solution…*

Attempt 1: Dependent Turn-taking (1 flag)

- If value of `turn` == process #, process can proceed
- What if one process takes a long turn?

Attempt 2: Independent Turn-taking (2+ flags)

- Each process can proceed independently*
- What if one process fails in critical section?
- **=> Also, *possibly incorrect! Mutual exclusion broken! (TOCTOU)***

Attempt 3: Set THEN check

- Mutual exclusion fixed…
- What if both processes set flags to `true`?
- ***=> Deadlock is now possible!***

```
Problematic Sequence...
P0 executes the while statement and finds flag[1] set to false
P1 executes the while statement and finds flag[0] set to false
P0 sets flag[0] to true and enters its critical section
P1 sets flag[1] to true and enters its critical section
```

/* PROCESS 0 /*	/* PROCESS 1 */
• • **while** (turn != 0) /* do nothing */ ; /* critical section*/; turn = 1; •	• • **while** (turn != 1) /* do nothing */; /* critical section*/; turn = 0; •

(a) First attempt

/* PROCESS 0 */	/* PROCESS 1 */

•
•
while (flag[1])
/* do nothing */;
flag[0] = true;
/*critical section*/;
flag[0] = false;
•
	•
	•
	while (flag[0])
	/* do nothing */;
	flag[1] = true;
	/* critical section*/;
	flag[1] = false;
	•

(b) Second attempt

**while other process is not in the critical section*

/* PROCESS 0 */	/* PROCESS 1 */

•
•
flag[0] = true;
while (flag[1])
/* do nothing */;
/* critical section*/;
flag[0] = false;
•
	•
	•
	flag[1] = true;
	while (flag[0])
	/* do nothing */;
	/* critical section*/;
	flag[1] = false;
	•

(c) Third attempt

Mutual Exclusion—*Workings towards a solution…*

Attempt 1: Dependent Turn-taking (1 flag)

- If value of `turn` == process #, process can proceed
- What if one process takes a long turn?

Attempt 2: Independent Turn-taking (2+ flags)

- Each process can proceed independently*
- What if one process fails in critical section?
- **=> Also, *possibly incorrect! Mutual exclusion broken! (TOCTOU)***

Attempt 3: Set THEN check

- Mutual exclusion fixed…
- What if both processes set flags to `true`?
- ***=> <u>Deadlock</u> is now possible!***

```
Problematic Sequence...
P0 executes the while statement and finds flag[1] set to false
P1 executes the while statement and finds flag[0] set to false
P0 sets flag[0] to true and enters its critical section
P1 sets flag[1] to true and enters its critical section
```

Attempt 4: Set THEN check (THEN back-off?)

- Processes will “back-off”
- What if both processes alternate in deferring to the other?

/* PROCESS 0 /*	/* PROCESS 1 */

•
•
`while (turn != 0)`
`   /* do nothing */ ;`
`/* critical section*/;`
`turn = 1;`
•
	•
	•
	`while (turn != 1)`
	`   /* do nothing */;`
	`/* critical section*/;`
	`turn = 0;`
	•

(a) First attempt

/* PROCESS 0 */	/* PROCESS 1 */

•
•
`while (flag[1])`
`   /* do nothing */;`
`flag[0] = true;`
`/*critical section*/;`
`flag[0] = false;`
•
	•
	•
	`while (flag[0])`
	`   /* do nothing */;`
	`flag[1] = true;`
	`/* critical section*/;`
	`flag[1] = false;`
	•

(b) Second attempt

*while other process is not in the critical section

/* PROCESS 0 */	/* PROCESS 1 */

•
•
`flag[0] = true;`
`while (flag[1])`
`   /* do nothing */;`
`/* critical section*/;`
`flag[0] = false;`
•
	•
	•
	`flag[1] = true;`
	`while (flag[0])`
	`   /* do nothing */;`
	`/* critical section*/;`
	`flag[1] = false;`
	•

(c) Third attempt

/* PROCESS 0 */	/* PROCESS 1 */

•
•
`flag[0] = true;`
`while (flag[1]) {`
`  flag[0] = false;`
`  /*delay */;`
`  flag[0] = true;`
`}`
`/*critical section*/;`
`flag[0] = false;`
•
	•
	•
	`flag[1] = true;`
	`while (flag[0]) {`
	`  flag[1] = false;`
	`  /*delay */;`
	`  flag[1] = true;`
	`}`
	`/* critical section*/;`
	`flag[1] = false;`
	•

(d) Fourth attempt

Mutual Exclusion—*Workings towards a solution…*

Attempt 1: Dependent Turn-taking (1 flag)

- If value of `turn` == process #, process can proceed
- What if one process takes a long turn?

```
/* PROCESS 0 /*                    /* PROCESS 1 */

•                                  •
•                                  •
while (turn != 0)                  while (turn != 1)
   /* do nothing */ ;                 /* do nothing */;
/* critical section*/;             /* critical section*/;
turn = 1;                          turn = 0;
•                                  •
```

(a) First attempt

Attempt 2: Independent Turn-taking (2+ flags)

- Each process can proceed independently*
- What if one process fails in critical section?
- **=> Also, *possibly incorrect! Mutual exclusion broken! (TOCTOU)***

```
/* PROCESS 0 */                    /* PROCESS 1 */

•                                  •
•                                  •
while (flag[1])                    while (flag[0])
   /* do nothing */;                  /* do nothing */;
flag[0] = true;                    flag[1] = true;
/*critical section*/;              /* critical section*/;
flag[0] = false;                   flag[1] = false;
•                                  •
```

(b) Second attempt

**while other process is not in the critical section*

Attempt 3: Set THEN check

- Mutual exclusion fixed…
- What if both processes set flags to `true`?
- **=> *Deadlock is now possible!***

```
Problematic Sequence...
P0 executes the while statement and finds flag[1] set to false
P1 executes the while statement and finds flag[0] set to false
P0 sets flag[0] to true and enters its critical section
P1 sets flag[1] to true and enters its critical section
```

```
/* PROCESS 0 */                    /* PROCESS 1 */

•                                  •
•                                  •
flag[0] = true;                    flag[1] = true;
while (flag[1])                    while (flag[0])
   /* do nothing */;                  /* do nothing */;
/* critical section*/;             /* critical section*/;
flag[0] = false;                   flag[1] = false;
•                                  •
```

(c) Third attempt

Attempt 4: Set THEN check (THEN back-off?)

- Processes will "back-off"
- What if both processes alternate in deferring to the other?
- **=> *Livelock is now possible!***

```
Problematic Sequence...
P0 sets flag[0] to true
P1 sets flag[1] to true
P0 checks flag[1]
P1 checks flag[0]
P0 sets flag[0] to false
P1 sets flag[1] to false
P0 sets flag[0] to true
Pl sets flag[1] to true
```

```
/* PROCESS 0 */                    /* PROCESS 1 */

•                                  •
•                                  •
flag[0] = true;                    flag[1] = true;
while (flag[1]) {                  while (flag[0]) {
  flag[0] = false;                   flag[1] = false;
  /*delay */;                        /*delay */;
  flag[0] = true;                    flag[1] = true;
}                                  }
/*critical section*/;              /* critical section*/;
flag[0] = false;                   flag[1] = false;
•                                  •
```

(d) Fourth attempt

Peterson’s Algorithm for Two Processes

➡ *Generalizable to 3+ processes*

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          turn = 1;
          while (flag [1] && turn == 1) /* do nothing */;
          /* critical section */;
          flag [0] = false;
          /* remainder */;
     }
}
void P1()
{
     while (true) {
          flag [1] = true;
          turn = 0;
          while (flag [0] && turn == 0) /* do nothing */;
          /* critical section */;
          flag [1] = false;
          /* remainder */
     }
}
void main()
{
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
}
```

➡ ***See Also: Dekker’s Algorithm***

Synchronization Roadmap

Concurrent Applications

Shared Objects

Bounded Buffers | Barrier

Synchronization Variables

Semaphores | Locks | Condition Variables

Atomic Instructions

Disable Interrupts | Test-and-Set

Hardware

Multiple Processors | Hardware Interrupts