# INF-2201
05 – Threads and critical sections

John Markus Bjørndalen
2024

Status now

- Kernel is getting more defined

  - Functionality accessible through system calls

  - Protected from user programs/code

- Multiprogramming introduced (term not mentioned in previous lecture)

  - Essentially means running multiple programs or tasks

  - Introduced processes last week

    - A process is the execution of a program

    - Each process has individual data, registers, stack and a PCB

    - May share code with other processes (ex: may run multiple instances of the same program)

- Some of the boot process (ex: no Power On Self Test (POST) yet, …)

User program

RAM

Kernel

BIOS

ROM
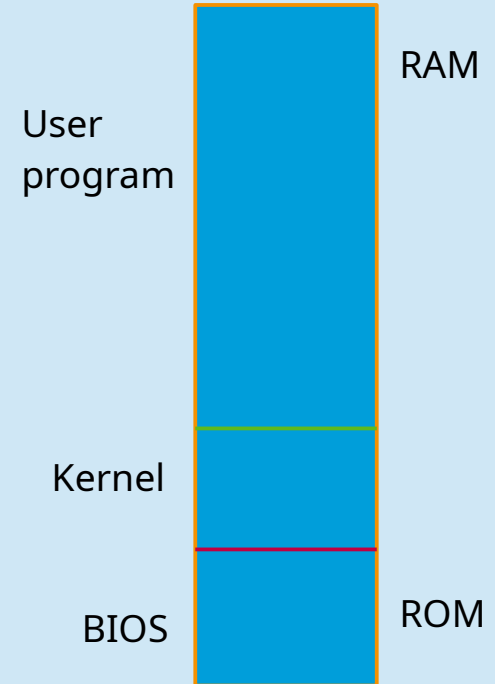
Problem: starting and stopping new processes

How do we create new processes from user level (from a process)?

- "Parent" calls fork()

  - Makes a copy of the current process.

  - New process (child) gets a new PCB. Starts executing at the same instruction as the parent

  - Identifying which copy is the parent or child: Return value of fork() is

    - Parent: PID (Process ID) of child

    - Child: 0

This assumes Unix/POSIX API (other operating systems have similar mechanisms)

```c
/* Source code from "main wait" on Ubuntu 22.04 */

#include <sys/wait.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    pid_t cpid, w;
    int wstatus;

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {            /* Code executed by child */
        printf("Child PID is %jd\n", (intmax_t) getpid());
        if (argc == 1)
            pause();                    /* Wait for signals */
        _exit(atoi(argv[1]));

    } else {                    /* Code executed by parent */
        do {
            w = waitpid(cpid, &wstatus, WUNTRACED | WCONTINUED);
            if (w == -1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }

            if (WIFEXITED(wstatus)) {
                printf("exited, status=%d\n", WEXITSTATUS(wstatus));
            } else if (WIFSIGNALED(wstatus)) {
```

Problem: starting and stopping new processes

Waiting for a child process to terminate?

- Using one of the wait-calls (here: waitpid)

```c
/* Source code from "main wait" on Ubuntu 22.04 */

#include <sys/wait.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    pid_t cpid, w;
    int wstatus;

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {            /* Code executed by child */
        printf("Child PID is %jd\n", (intmax_t) getpid());
        if (argc == 1)
            pause();                    /* Wait for signals */
        _exit(atoi(argv[1]));

    } else {                    /* Code executed by parent */
        do {
            w = waitpid(cpid, &wstatus, WUNTRACED | WCONTINUED);
            if (w == -1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }

            if (WIFEXITED(wstatus)) {
                printf("exited, status=%d\n", WEXITSTATUS(wstatus));
            } else if (WIFSIGNALED(wstatus)) {
```

Problem: child should run a different program

Sometimes need to run a different program in the child than in the parent

- Solution: call exec() in child after returning from fork()

- Replaces the program code (and data) in the current program with new code and data

  - Typically starts running from main()

  - Can be called at any point in time (not just after fork)

  -

Python exec example: replace Python process with /bin/ls

>>> import os
>>> os.execv("/bin/ls", ["/bin/ls", "/usr/local"])
bin etc games include lib man sbin share src
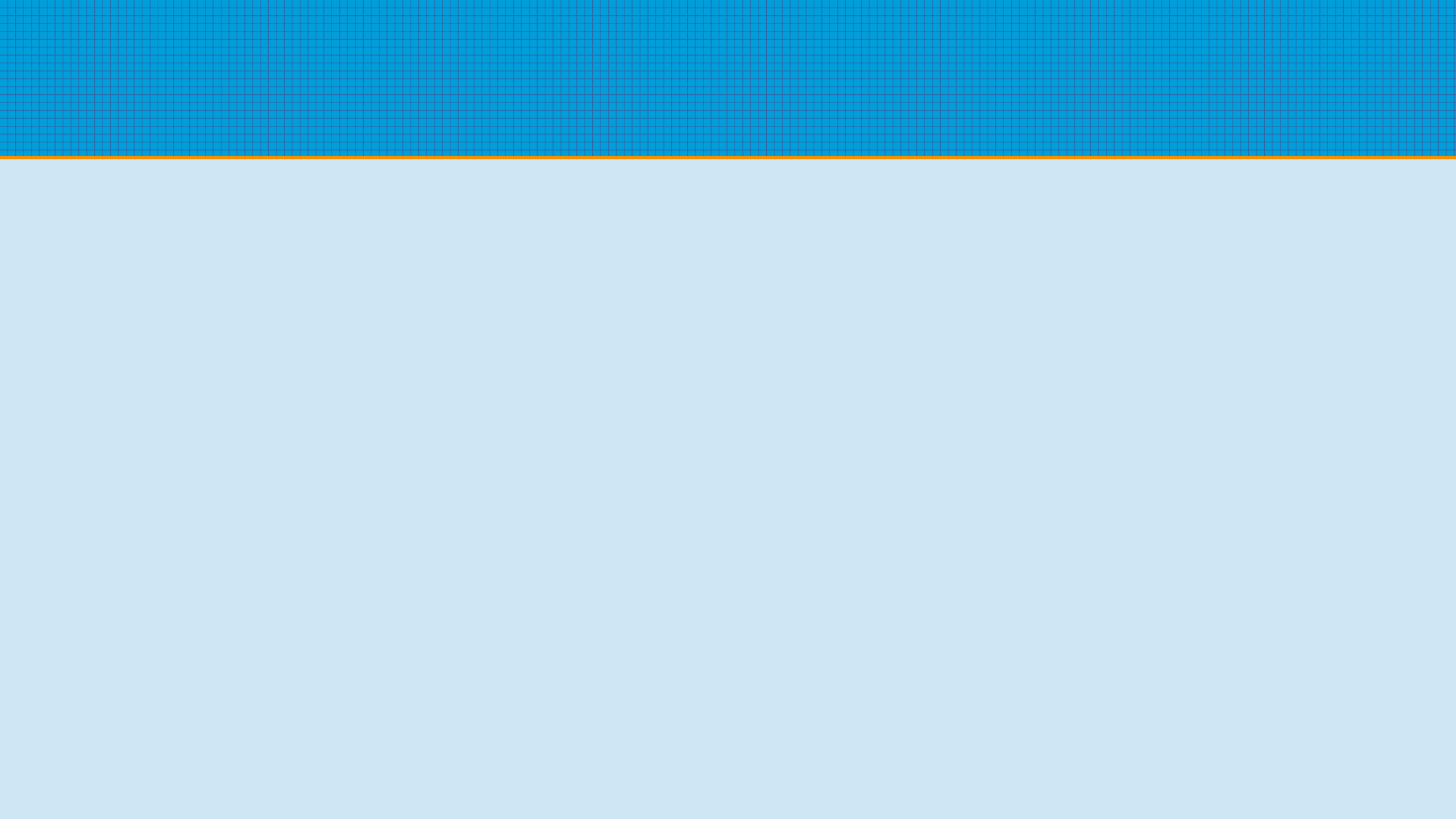
Problem: processes may hog the CPU or block

- No progress done on other processes

- Need some method for "taking away" CPU

- Solution: preemptive multitasking

  - Interrupt process while it is computing without waiting for it to yield

  - Yield can still be a system call for voluntary yielding

  - Need some way of interrupting process

Problem: processes may hog the CPU or block

Preemptive multitasking

- Use the interrupt mechanism that we already saw from system calls

  - Triggered by hardware (I/O devices that have data for the operating system) or software (traps/faults or software interrupts)

  - Additionally, let a periodic clock trigger interrupts

- When an interrupt occurs:

  - The current execution is stopped

  - State is saved and the CPU switches to kernel mode

  - The CPU looks up the interrupt number/id in a table and runs a service routine (similar to the early system calls)

  - The service routine can then decide whether it wants to take CPU away from the current process and let somebody else run (similar to earlier yield + schedule)
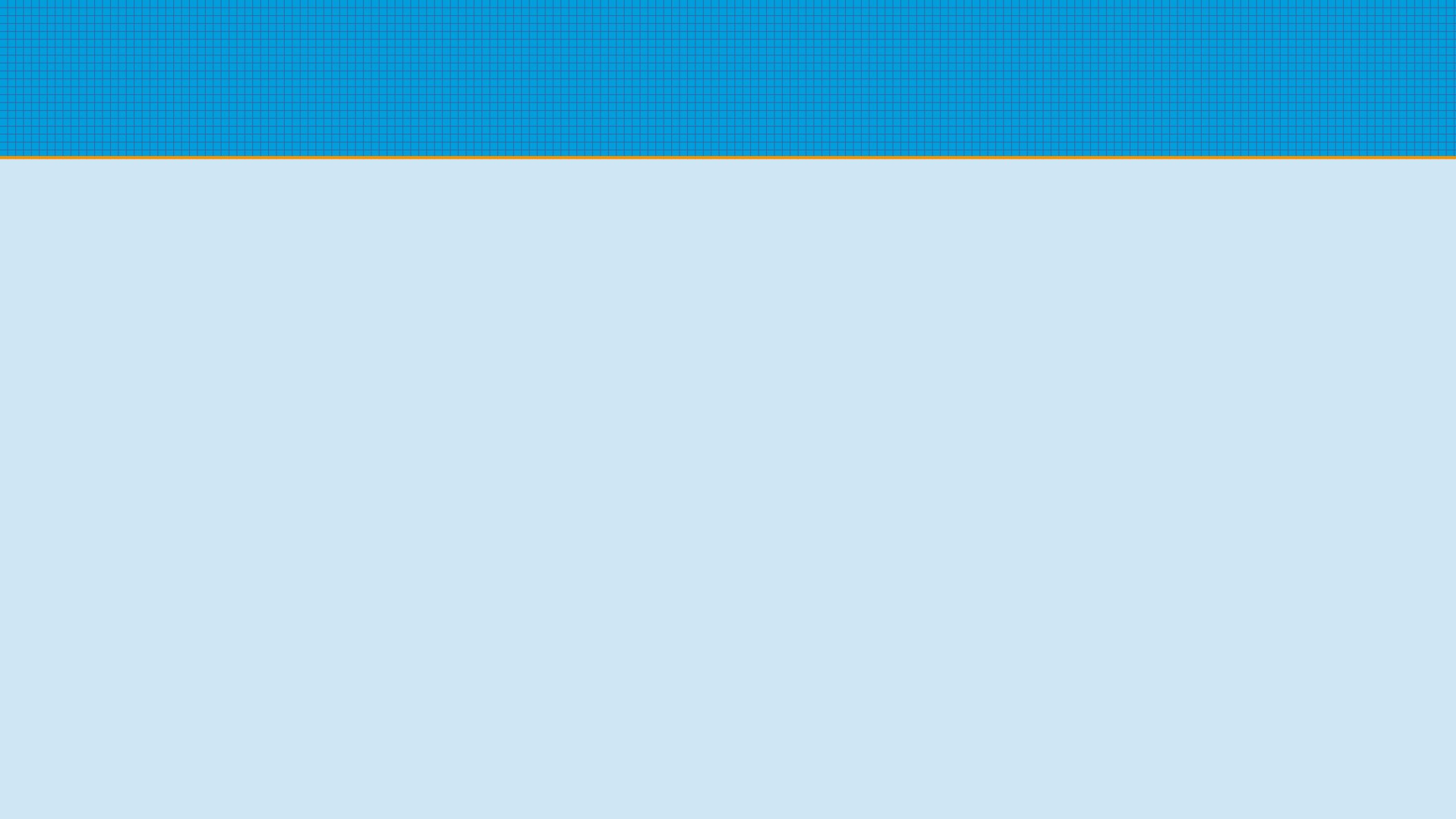
P1   P2

Kernel
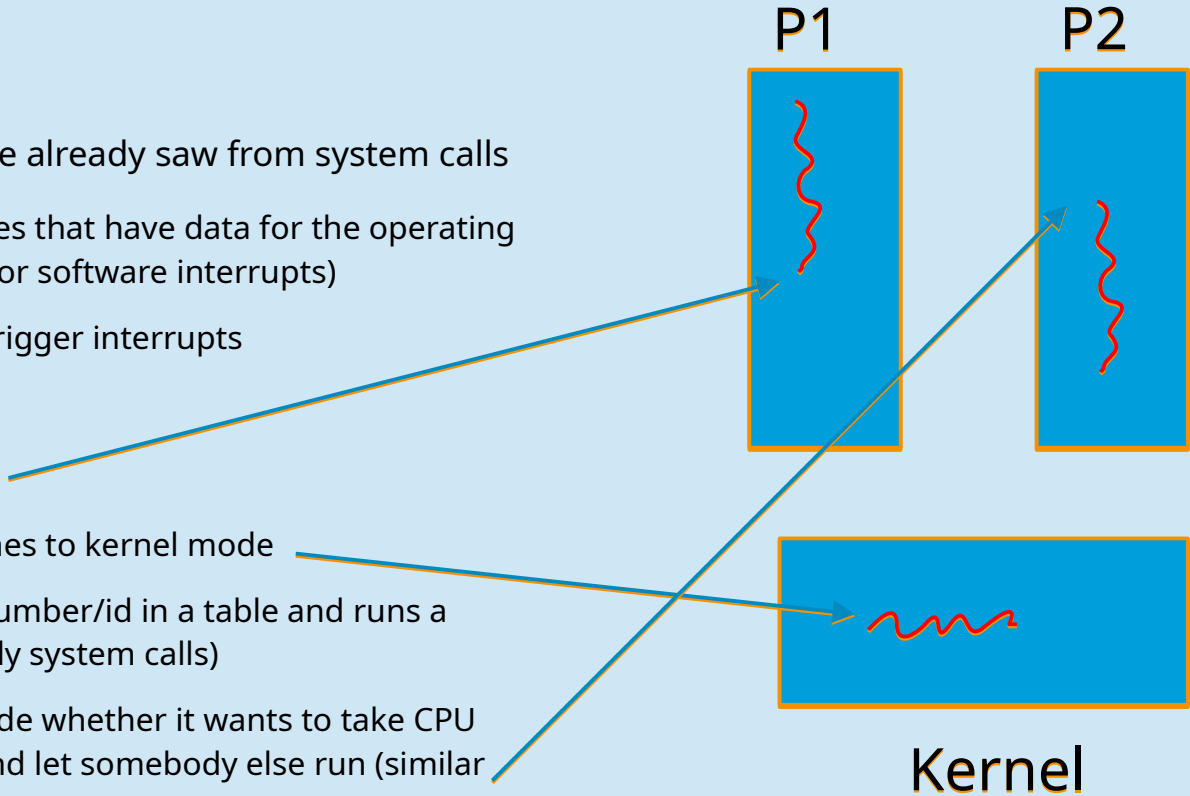
Here be dragons

- But we're pretending to not see them (yet)

- At least not until we have created some more problems

"A dragon debuggin a multithreaded program"
Made with Image Creator from Designer. Powered by DALL·E 3.
(I was tired, but Skype saw through the spelling mistake)

Problem: multiple activities in a single program

- Some system calls are blocking

  - "wait for user input"

  - "wait for file to be updated by another program"

- Each of these block would block the entire process

  - Ex: if program is waiting for file updates, it will not make any progress on user input

- Could support multiple activities by forking

  - One process to handle each activity

- Several challenges:

  - Memory overhead (complete copy of program)

  - Communication overhead (processes need to coordinate, keep copies of shared state consistent, ...)

  - Context switch overhead (switch between processes)

  - Fork/join overhead

Solution: threads

Like processes, threads are used to keep track of execution. They differ in some ways:

- Parent and child thread share memory, open files etc.

- Use a different API (typical on Unix: pthreads), but similar concepts to process creation

- A thread is typically viewed as executing "inside" a process

  - One view: a process is a container that always has at least one default thread to track the execution. Can then add more threads.

  - Threads have individual Thread Control Blocks (TCB), but may share PCBs.

  - Less overhead for context switching

    - Some of it related to topics we haven't covered yet

  - Potentially less overhead for keeping shared state consistent (it's shared memory)

    - Some issues introduced though...

Threads

Example: phtreads API

- Create a thread: pthread_create
  - Specify function to call and parameters to function
  - Returns thread ID
- Wait for a thread: pthread_join
  - Specify thread ID

- What will this program print?

```c
#include <pthread.h>
#include <stdio.h>

const int N = 10;

typedef struct {
    int th_no;
} thread_parm;

void* thread_proc(void *vargs)
{
    thread_parm *parm = (thread_parm*) vargs;
    printf("This is thread with param %d\n", parm->th_no);
    return NULL;
}

int main(int argc, char* argv[])
{
    pthread_t threads[N];
    thread_parm parms[N];

    // Create N threads.
    for (int i = 0; i < N; i++) {
        parms[i].th_no = i;
        pthread_create(&threads[i], NULL, thread_proc, &parms[i]);
    }
    printf("All spawned, now waiting for them to complete\n");
    for (int i = 0; i < N; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

Threads

Example: phtreads API

```
>cc -Wall thread1.c
>./a.out
This is thread with param 0
This is thread with param 3
This is thread with param 4
This is thread with param 1
This is thread with param 2
This is thread with param 5
This is thread with param 6
This is thread with param 7
This is thread with param 8
All spawned, now waiting for them to complete
This is thread with param 9
```

The system is not required to run the threads in the order you created them!

```c
#include <pthread.h>
#include <stdio.h>

const int N = 10;

typedef struct {
    int th_no;
} thread_parm;

void* thread_proc(void *vargs)
{
    thread_parm *parm = (thread_parm*) vargs;
    printf("This is thread with param %d\n", parm->th_no);
    return NULL;
}

int main(int argc, char* argv[])
{
    pthread_t threads[N];
    thread_parm parms[N];

    // Create N threads.
    for (int i = 0; i < N; i++) {
        parms[i].th_no = i;
        pthread_create(&threads[i], NULL, thread_proc, &parms[i]);
    }
    printf("All spawned, now waiting for them to complete\n");
    for (int i = 0; i < N; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

Side note

- Linux has more flexibility when creating threads or processes when using the low-level **clone** system call

    - https://man7.org/linux/man-pages/man2/clone.2.html

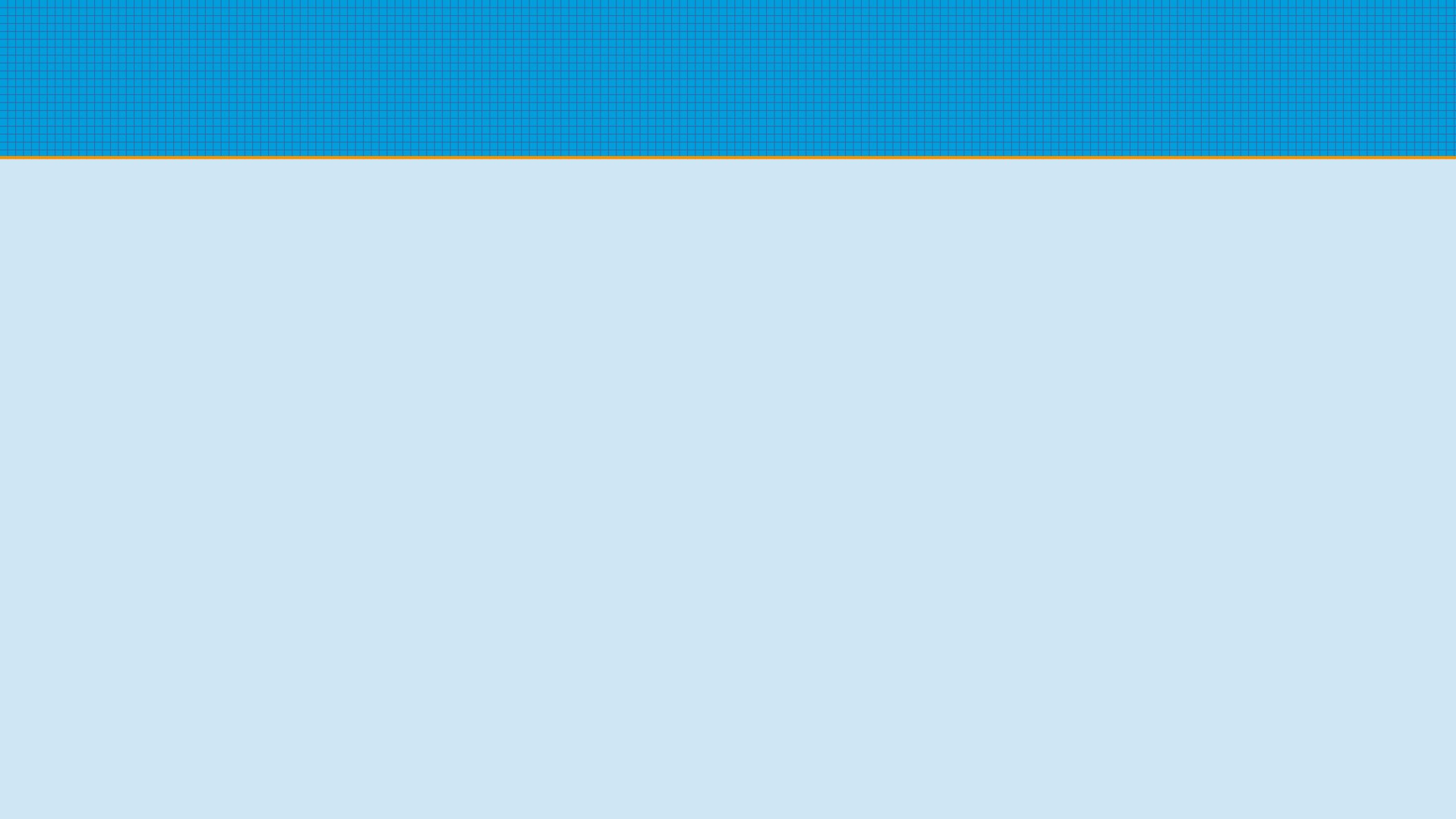- Lets caller control which resources should be shared or not shared between the parent and child.
Whether it is a traditional process or thread depends on the set of flags specified when using clone.

-

Some options for implementing threads

- Let the kernel manage and schedule threads (kernel threads)
  - Implement similarly to how we implement processes, the main difference is in what threads share compared to processes
- Implement entirely at user level (user level threads / user threads)
  - Similar to the early process implementation we defined (cooperative multitasking)
  - User threads yield to let other threads to run
  - No preemption (usually)
  - Very low overhead (kernel not involved in switching and scheduling)
  - Blocking system calls still cause issues: one blocked user level thread will block all of them

Some options for implementing threads

- Hybrid user and kernel threads: combine the best of both
    - Runtime at user level implements user level threads that can yield
    - Multiple kernel threads participate in scheduling and executing the user threads
        - Combines low overhead switching (done at user level) with some robustness with blocking system calls
        - Can also benefit from multi-core, hyperthreading, multi-cpu
    - Example language support: The go programming language (goroutines)
- One issue: complicated to get right (both performance and correctness)

Warning: threads are "broken" in the OS code

TODO: this one may no longer be true in/after 2025

- They work, but they are confusing

- Kernel threads do not normally mean "execute in kernel" – it means "managed by the kernel"

- In the OS code

  - Threads only execute inside the kernel

  - Threads have no connection to any processes
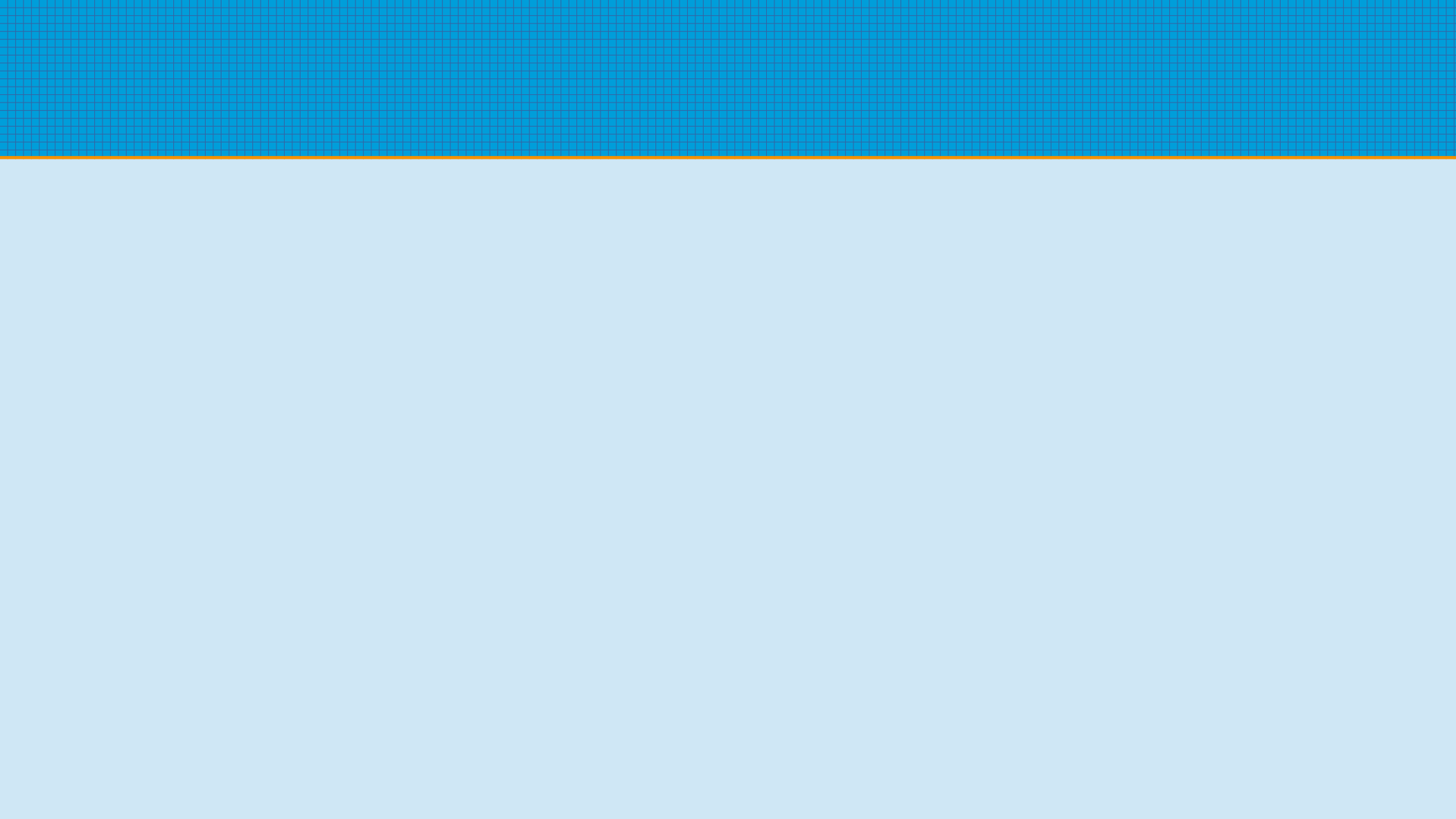
Where's the problem?

"A dragon hiding under the table. A monkey is debugging a program"
Made with Image Creator from Designer. Powered by DALL·E 3.

Not quite what I intended.

A few assumptions

Atomic reads and writes of variables

Where's the problem?

The "Too Much Milk" Problem

Person A

Look in fridge: no milk

Leave for store

Arrive at store

Leave store

Arrive home, insert milk

Note: I have not found the original description of this problem.
It's frequently found in lecture notes with vague references that
do not appear to be correct.

Where's the problem?

The "Too Much Milk" Problem

Person A

Look in fridge: no milk

Leave for store

Arrive at store

Leave store

Arrive home, insert milk

Person B

Look in fridge: no milk

Leave for store

Arrive at store

Leave store

Arrive home, too much milk!

Note: I have not found the original description of this problem. It's frequently found in lecture notes with vague references that do not appear to be correct.

Too much milk problem

Pseudocode

```
// Person A
if (!milk) {
    goto_store();       // Person B
    ...                 if (!milk) {
    insert_milk();          goto_store();
}                           ...
                            insert_milk();
                        }
insert_milk:
milk += 1
```

This is a **race condition**.
The correctness depends on who arrives when to each part.
If both end up inserting, we get two bottles instead of the one we wanted.

NB: milk += 1 is a race condition in itself unless it's implemented as an atomic operation (like atomic_add).
This about what the operation does: 1) read, 2) modify, 3) write.
Anything can get into there.
We will assume atomic_add for the rest of the discussion.

Too much milk problem

Solution 1: leave a note

```
// Person A                          // Person B
If (!milk) {                         If (!milk) {
    if (!note) {                         if (!note) {
        note = 1;                            note = 1;
        goto_store();                        goto_store();
        ...                                  ...
        insert_milk();                       insert_milk();
        note = 0;                            note = 0;
    }                                    }
}                                    }
```

Sufficient for non-preemptive / cooperative?

How about preemptive multithreading?
    Person A and B are threads in the same program.

Too much milk problem

Solution 1: leave a note

```
// Person A                              // Person B
If (!milk) {                             If (!milk) {
    if (!note) {                             if (!note) {
        note = 1;                                note = 1;
        goto_store();                            goto_store();
        ...                                      ...
        insert_milk();                           insert_milk();
        note = 0;                                note = 0;
    }                                        }
}                                        }
```

Simple lock (note)

Sufficient for non-preemptive / cooperative?
- OK. Why?

How about preemptive multithreading?
- No... key observation: look at observation → action
  - !note → note = 1 ... can be preempted between the two
  - Both can observe milk == 0, then note == 0

Too much milk problem

Solution 1: leave a note

```
// Person A
If (!milk) {
    if (!note) {
        note = 1;
        goto_store();
        ...
        insert_milk();
        note = 0;
    }
}
```

```
// Person B
If (!milk) {
    if (!note) {
        note = 1;
        goto_store();
        ...
        insert_milk();
        note = 0;
    }
}
```

Simple lock (note)

Need **mutual exclusion** mechanism to provide a **critical section** (code that only one process/thread is in at the same time). Four conditions from the book:

1) No two processes may be simultaneously inside their critical regions.
2) No assumptions may be made about speeds or the number of CPUs.
3) No process running outside its critical region may block any process.
4) No process should have to wait forever to enter its critical region.

Too much milk problem

Solution 1b: turn off interrupts and leave a note

```
// Person A
turn_off_interrupts()
If (!milk) {
    if (!note) {
        note = 1;
        goto_store();
        ...
        insert_milk();
        note = 0;
    }
}
turn_on_interrupts()
```

```
// Person B
turn_off_interrupts()
If (!milk) {
    if (!note) {
        note = 1;
        goto_store();
        ...
        insert_milk();
        note = 0;
    }
}
turn_on_interrupts()
```

Simple lock (note)

Turned the preemptive problem back to a cooperative problem.

It has some issues

- What if one of them blocks or spins with the interrupts turned off? (back to resource hog)
- Will it work for multicore CPUs? (you need more)
- User level code should not be allowed to turn off interrupts, but the kernel may use this trick inside interrupt handlers since it needs to make sure new interrupts don't arrive while servicing the first!

Too much milk problem

Solution 2: both leave a note

```
// Person A
note_a = 1
if (!note_b) {
    if (!milk) {
        goto_store();
        ...
        milk = 1;
    }
}
note_a = 0
```

```
// Person B
note_b = 1
if (!note_a) {
    if (!milk) {
        goto_store();
        ...
        milk = 1;
    }
}
note_b = 0
```

Safe?

Too much milk problem

Solution 2: both leave a note

```
// Person A
note_a = 1
if (!note_b) {
    if (!milk) {
        goto_store();
        ...
        milk = 1;
    }
}
note_a = 0
```

```
// Person B
note_b = 1
if (!note_a) {
    if (!milk) {
        goto_store();
        ...
        milk = 1;
    }
}
note_b = 0
```

Safe?

What if both leave a note at the same time and then check for the other's note?

=> starvation (nobody buys milk)

Too much milk problem

Solution 3: both leave a note

```
// Person A
note_a = 1
while (note_b) {
    do_nothing();
}
if (!milk) {
    goto_store();
    ...
    milk = 1;
}
note_a = 0
```

```
// Person B
note_b = 1
while (note_a) {
    do_nothing();
}
if (!milk) {
    goto_store();
    ...
    milk = 1;
}
note_b = 0
```

Busy wait / spinning

Safe?

Too much milk problem

Solution 3: both leave a note

```
// Person A
note_a = 1
while (note_b) {
    do_nothing();
}
if (!milk) {
    goto_store();
    ...
    milk = 1;
}
note_a = 0
```

```
// Person B
note_b = 1
while (note_a) {
    do_nothing();
}
if (!milk) {
    goto_store();
    ...
    milk = 1;
}
note_b = 0
```

Busy wait / spinning

Safe?

Not quite: look at what happens if both write their notes at the same time!

Both will wait for the other indefinitely: deadlock!
We will cover deadlocks more later.

Too much milk problem

Solution 4: make it asymmetric

```
// Person A
note_a = 1
while (note_b) {
    do_nothing();
}
if (!milk) {
    goto_store();
    ...
    milk = 1;
}
note_a = 0
```

```
// Person B
note_b = 1
if (!note_a) {
    if (!milk) {
        goto_store();
        ...
        Milk = 1;
    }
}
note_b = 0
```

Note: no busy wait here!

Safe?

Too much milk problem

Solution 4: make it asymmetric

```
// Person A
note_a = 1
while (note_b) {
    do_nothing();
}
if (!milk) {
    goto_store();
    ...
    milk = 1;
}
note_a = 0
```

```
// Person B
note_b = 1
if (!note_a) {
    if (!milk) {
        goto_store();
        ...
        Milk = 1;
    }
}
note_b = 0
```

Safe?

If they both go in at the same time, A will wait until B leaves.
B will notice that A has the note set, so B will leave without checking for milk.
A ends up buying the milk.

Fair solution?
Easy to reason about? What about the other cases (all possible cases of a and b executing individual steps)?

Use atomic operations to create critical regions

- Define enter_region and leave_region operations

- XCHG is an atomic operation

- Note: it's using a busy loop for entry, so it will waste cycles

  - It can be very quick if the lock is free though

- Can lead to **priority inversion** if p1 of high priority spins waiting for p2 of low priority to leave the region

- What happens if you disable interupts and then try enter_region, going into an infinite loop?

```
enter_region:
      MOVE REGISTER,#1              I put a 1 in the register
      XCHG REGISTER,LOCK           I swap the contents of the register and lock variable
      CMP REGISTER,#0              I was lock zero?
      JNE enter_region             I if it was non zero, lock was set, so loop
      RET                          I return to caller; critical region entered


leave_region:
      MOVE LOCK,#0                 I store a 0 in lock
      RET                          I return to caller
```

Figure 2-26. Entering and leaving a critical section using the XCHG instruction.
Modern Operating Systems, Tanenbaum & Bos

More about this later

- This is where operating systems (and concurrent programming) get complicated very quickly

  - It can also get very interesting

- Things that can help

  - Theory (and well-known solutions)

  - Proofs

  - Model checkers

    - ex: see the Harmony programming language:
      https://harmony.cs.cornell.edu/

TODO

- Paterson's Solution (also using spin locks)

- Starving Philosophers ("wot, no chickens?"):
  https://www.cs.kent.ac.uk/projects/ofa/java-threads/0.html

- More about concurrent programming (and why this is relevant for concurrent programming)