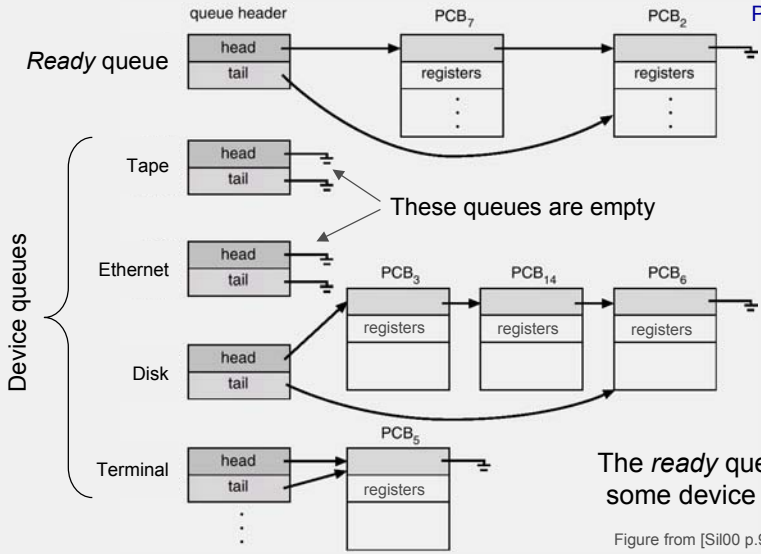
Scheduling

On a uniprocessor system there is only one process running, all others have to wait until they are scheduled. They are waiting in some **scheduling queue**:

- Job Queue
  Holds the future processes of the system.
- Ready Queue (also called CPU queue)
  Holds all processes that reside in memory and are ready to execute.
- Device Queue (also called I/O queue)
  Each device has a queue holding the processes waiting for I/O completion.
- IPC Queue
  Holds the processes that wait for some IPC (inter process communication) event to occur.

Scheduling



The *ready* queue and some device queues

Figure from [Sil00 p.92], modified

Scheduling

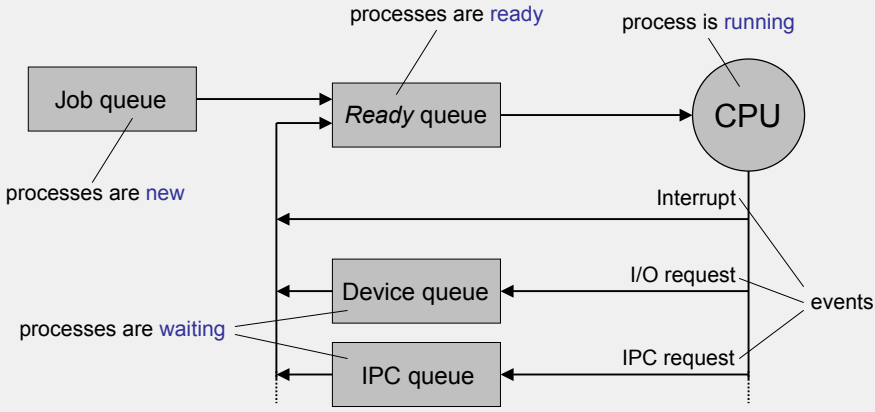
From the job queue a new process is initially put into the ready queue. It waits until it is *dispatched* (selected for execution). Once the process is allocated the CPU, one of these events may occur.

- **Interrupt**
  The *time slice* may be expired or some higher priority process is ready. Hardware error signals (exceptions) also may cause a process to be interrupted.
- **I/O request**
  The process requests I/O. The process is shifted to a device queue. After the I/O device has ready, the process is put into the *ready* queue to continue.
- **IPC request**
  The process wants to comunicate with another process through some *blocking* IPC feature. Like I/O, but here the „I/O-device“ is another process.

A note on the terminology: Strictly spoken, a process (in the sense of an active entity) only exists when it is allocated the CPU. In all other cases it is a ‚dead body‘.

Scheduling

Processes



Queueing diagram of process scheduling

Scheduling

The OS selects processes from queues and puts them into other queues. This selection task is done by schedulers.

- **Long-term Scheduler**

  Originates from batch systems. Selects jobs (programs) from the pool and loads them into memory.

  Invoked rather infrequently (seconds ... minutes). Can be slow.
  Has influence on the *degree of multiprogramming* (number of processes in memory).
  Some modern OS do not have a long-term scheduler any more.

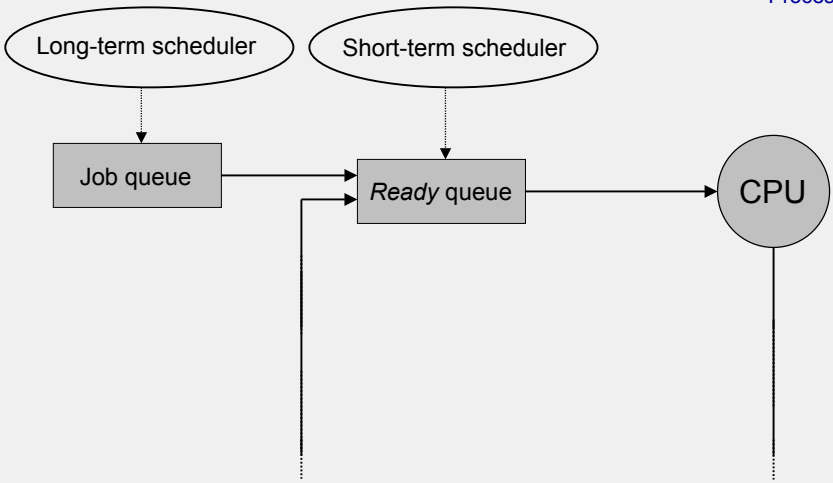
- **Short-term Scheduler**

  Selects one process from among the processes that are ready to execute, and allocates the CPU to it. Initiates the *context switches*.

  Invoked very frequently (in the range of milliseconds). Must be fast, that is, must not consume much CPU time compared to the processes.

Scheduling

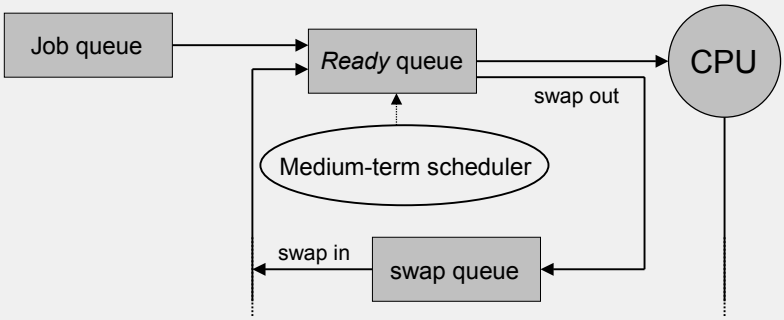
Processes



Schedulers and their queues

Scheduling

Sometimes it may be advantageous to remove processes temporarily from memory in order to reduce the degree of multi-programming. At some later time the process is reintroduced into memory and can be continued. This scheme is called *swapping*, performed by a *medium-term* scheduler.



Process Concept

- Program in execution
  Several processes may be carried out in parallel.
- Resource grouping
  Each process is related to a certain task and groups together the required resources (Address space, PCB).

Traditional multi-processing systems:

- Each process is executed sequentially
  No parallelism inside a process.
- Blocked operations → Blocked process
  Any blocking operation (e.g. I/O, IPC) blocks the process. The process must wait until the operation finishes.

In traditional systems each process has a single *thread* of control.

Process Management

- Processes
- Threads
- Interprocess Communication (IPC)
- CPU Scheduling
- Deadlocks

Threads

A **thread** is

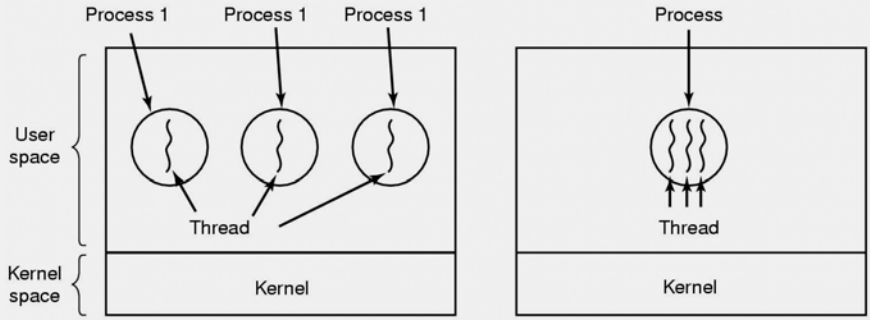
- a piece of yarn,
- a screw spire,
- a line of thoughts.

Here: **a sequence of instructions**
that may execute in parallel with others

A thread is a line of execution within the scope of a process. A single threaded process has a single line of execution (sequential execution of program code), the process and the thread are the same. In particular, a thread is

- a basic unit of CPU utilization.

Threads



Three *single threaded* processes in parallel

A process with three parallel threads.

Figure from [Ta01 p.82]

Threads

As an example, consider of a word processing application.

- Reading from keyboard
- Formatting and displaying pages
- Periodically saving to disk
- *... and lots of other tasks*

A single threaded process would quite quickly result in an unhappy user since (s)he always has to wait until the current operation is finished.

- Multiple processes?
  Each process would have its own isolated address space.
- Multiple threads!
  The threads operate in the same address space and thus have access to the data.

Threads

Three-threaded word processing application

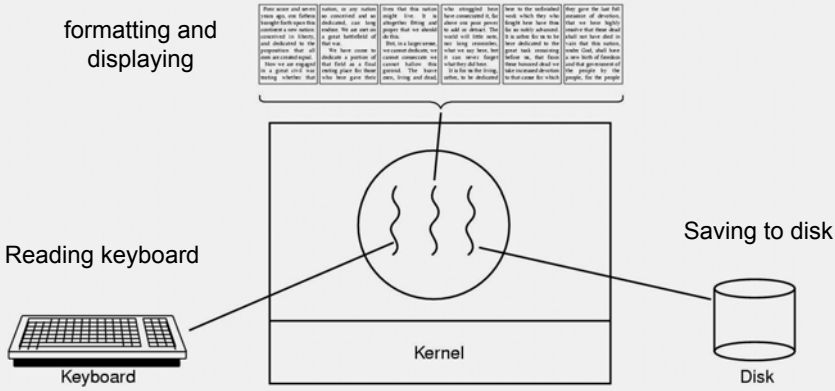


Figure from [Ta01 p.86]

Threads

- Multiple executions in same environment
  All threads have exactly the same address space (the process address space).
- Each thread has own registers, stack and state

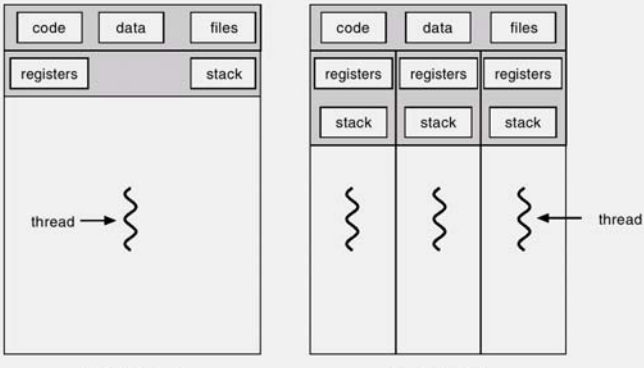


Figure from [Sil00 p.116]

Threads

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Items shared by all threads in a process

Items private to each thread

Table from [Ta01 p.83]

User Level Threads

- Take place in user space
  The operating system does not know about the applications' internal multi-threading.
- Can be used on OS not supporting threads
  It only needs some thread library (like *pthreads*) linked to the application.
- Each process has its own thread table
  The table is maintained by the routines of the thread library.
- Customized thread scheduling
  The processes use their own thread scheduling algorithm. However, no timer controlled scheduling possible since there are no clock interrupts inside a process.
- Blocking system calls do block the process
  All threads are stopped because the process is temporarily removed from the CPU.

User Level Threads

Thread management is performed by the application.

Examples
- POSIX *Pthreads*
- Mach *C-threads*
- Solaris *threads*

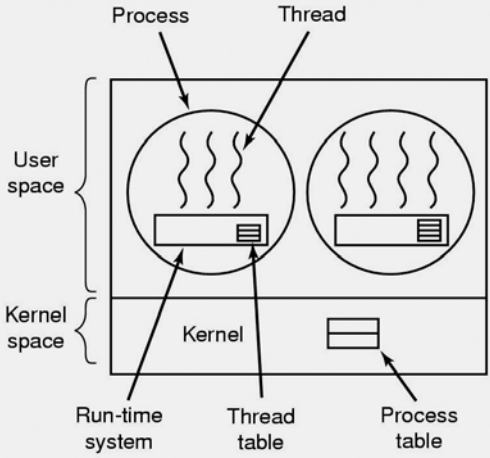


Figure from [Ta01 p.91]

Kernel Threads

- Take place in kernel
  The operating system manages the threads of each process
- Available only on multi-threaded OS‘s
  The operating system must support multi-threaded application programs.
- No thread administration inside process
  since this is done by the kernel. Thread creation and management however is generally somewhat slower than with user level threads [Sil00 p.118].
- No customized scheduling
  The user process cannot use its own customized scheduling algorithm.
- No problem with blocking system calls
  A blocking system call causes a thread to pause. The OS activates another thread, either from the same process or from another process.

Kernel Threads

Thread management is performed by the operating system.

Examples
- Windows 95/98/NT/2000
- Solaris
- Tru64 UNIX
- BeOS
- Linux

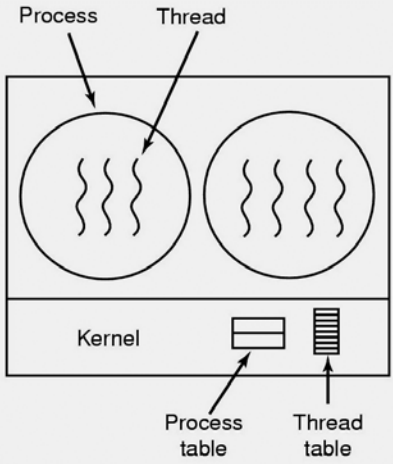
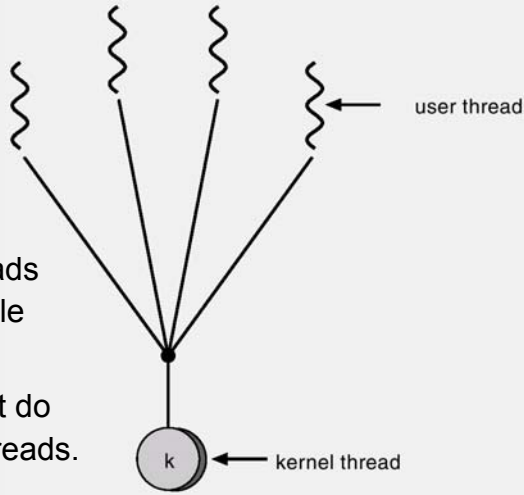


Figure from [Ta01 p.91]

Multithreading Models

Many-to-One Model

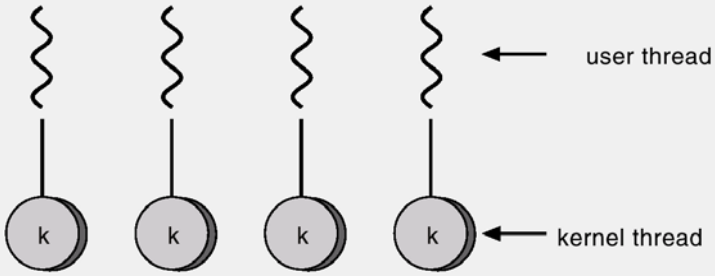


- Many user level threads are mapped to a single kernel thread.
- Used on systems that do not support kernel threads.

Figure from [Sil00 p.118]

Multithreading Models

One-to-One Model

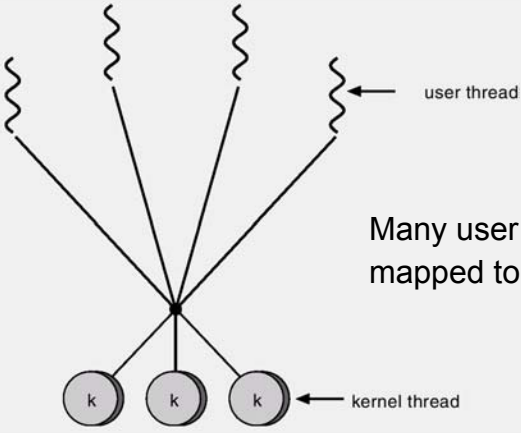


Each user level thread is mapped to one kernel thread.

Figure from [Sil00 p.119]

Multithreading Models

Many-to-Many Model



Many user level threads are mapped to many kernel threads.

Figure from [Sil00 p.119]

Multithreading

Solaris 2 multi-threading example

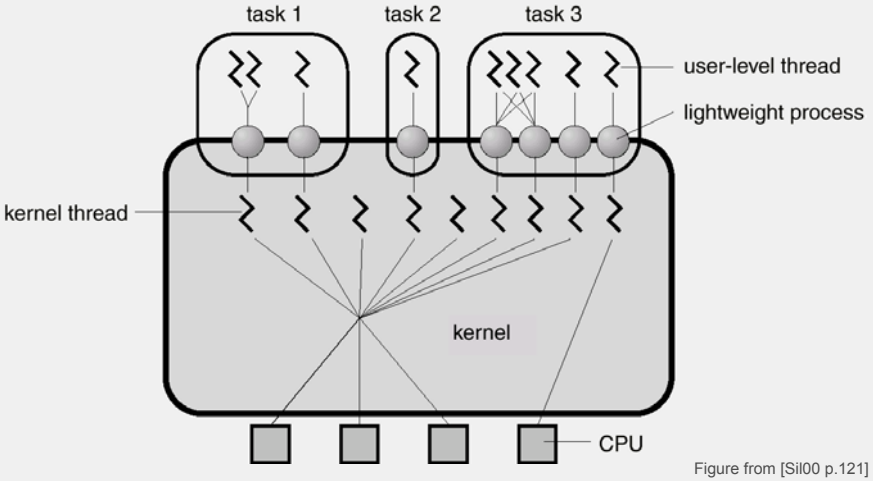


Figure from [Sil00 p.121]

Threads

Windows 2000:

- Implements *one-to-one* mapping
- Each thread contains
  - a thread id
  - register set
  - separate user and kernel stacks
  - private data storage area

Linux:

- One-to-one model (*pthreads*), many-to-many (*NGPT*)
- Thread creation is done through **`clone()`** system call.

  `clone()` allows a child to share the address space of the parent. This system call is unique to Linux, source code not portable to other UNIX systems.

Threads

Java:
- Provides support at language level.
- Thread scheduling in JVM

```
class Worker extends Thread {
  public void run() {
   System.out.println("I am a worker thread");
  }
}

public class MainThread {
  public static void main(String args[]) {
   Worker worker1 = new Worker();
   worker1.start();
   System.out.println("I am the main thread");
  }
}
```

thread creation and automatic call of run() method

Example: Creation of a thread by inheriting from `Thread` class

Process Management

- Processes
- Threads
- Interprocess Communication (IPC)
- CPU Scheduling
- Deadlocks

IPC

Purpose of **I**nter **P**rocess **C**ommunication

- **Managing critical activities**
  Making sure that two (or more) processes do not get into each others' way when engaging critical activities.
- **Sequencing**
  Making sure that proper sequencing is assured in case of dependencies among processes.

Process Synchronization
Thread Synchronization

- **Passing information**
  Processes are independent of each other and have private address spaces.
  How can a process pass information (or data) to another process?

Data exchange

Less important for threads since they operate in the same environment

Race Conditions

Print spooling example

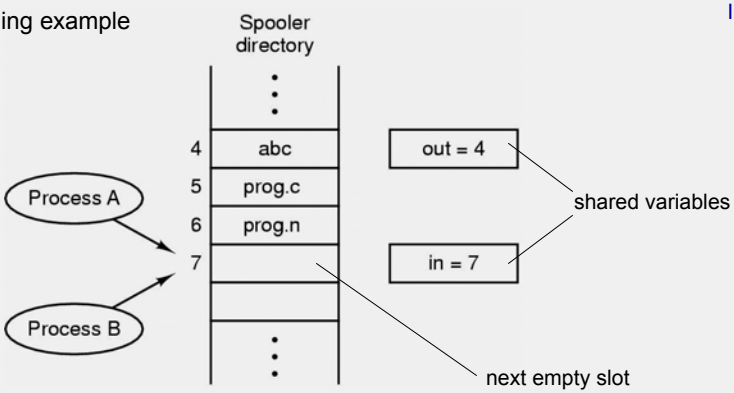


Figure from [Ta01 p.101]

Situations, where two or more processes access some shared resource, and the final result depends on who runs precisely when, are called *race conditions*.

Race Conditions

- Processes A and B want to print a file
- Both have to enter the file name into a spooler directory
- *out* points to the next file to be printed. This variable is accessed only by the printer daemon. The daemon currently is busy with slot 4.
- *in* points to the next empty slot. Each process entering a file name in the empty slot must increment *in*.

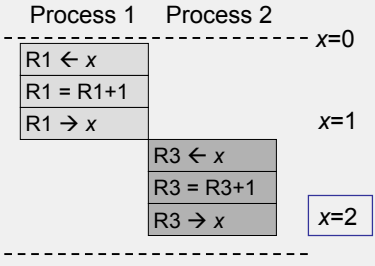
Now consider this situation:

- Process A reads *in* (value = 7) into some local variable. Before it can continue, the CPU is switched over to B.
- Process B reads *in* (value = 7) and stores its value locally. Then the file name is entered into slot 7 and the local variable is incremented by 1. Finally the local variable is copied to *in* (value = 8).
- Process A is running again. According to the local variable, the file name is entered into slot 7 – erasing the file name put by B. Finally *in* is incremented.
- User B is waiting in the printer room for years ...

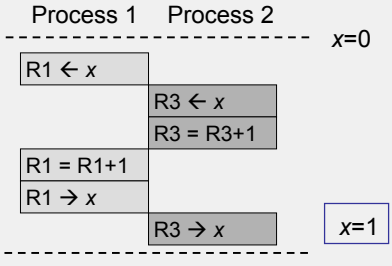
Race Conditions

Another example at machine instruction level

Shared variable *x* (initially 0)



Scenario 1



Scenario 2

Critical Regions

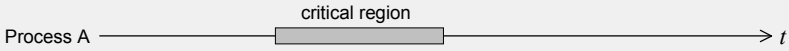
How to avoid race conditions?

Find some way to prohibit more than one process from manipulating the shared data at the same time.

→ „Mutual exclusion“

Part of the time a process is doing some internal computations and other things that do not lead to race conditions.

Sometimes a process however needs to access shared resources or does other critical things that may lead to race conditions. These parts of a program are called **critical regions** (or critical sections).



Critical Regions

Four conditions to provide correctly working *mutual exclusion*:

1. No two processes simultaneously in critical region
   which would otherwise controvert the concept of mutuality.
2. No assumptions about process speeds
   No predictions on process timings or priorities. Must work with all processes.
3. No process outside its critical regions must block other processes, simply because there is no reason to hinder others entering their critical region.
4. No process must wait forever to enter a critical region.
   For reasons of fairness and to avoid deadlocks.

Critical Regions

Mutual exclusion using critical regions

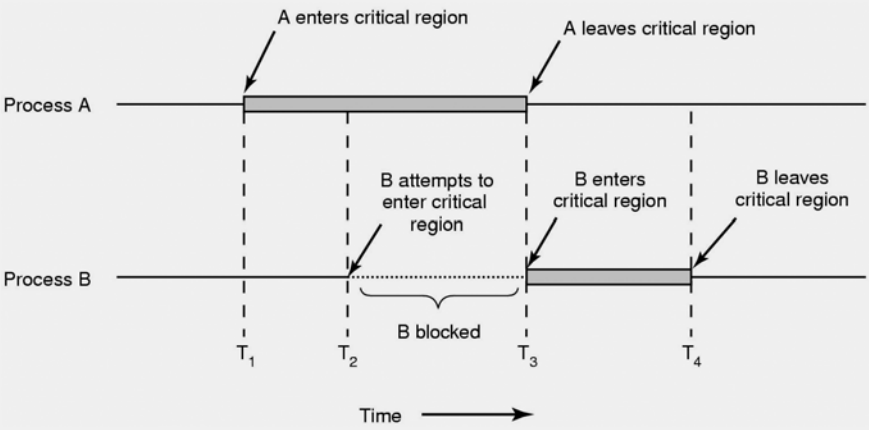


Figure from [Ta01 p.103]

Mutual Exclusion

Proposals for achieving mutual exclusion

- Disabling interrupts
  The process disables all interrupts and thus cannot be taken away from the CPU.
  - Not appropriate. Unwise to give user process full control over computer.
- Lock variables
  A process reads a shared lock variable. If the lock it is not set, the process sets the variable (locking) and uses the resource.
  - In the period between evaluating and setting the variable the process may be interrupted. Same problem as with printer spooling example.

Mutual Exclusion

Proposals for achieving mutual exclusion (continued)

- Strict Alternation

  The shared variable **turn** keeps track of whose turn it is. Both processes alternate in accessing their critical regions.

```
while (1) {
  while (turn != 0);
  critical_region();
  turn = 1;
  noncritical_region();
}
```

Process 0

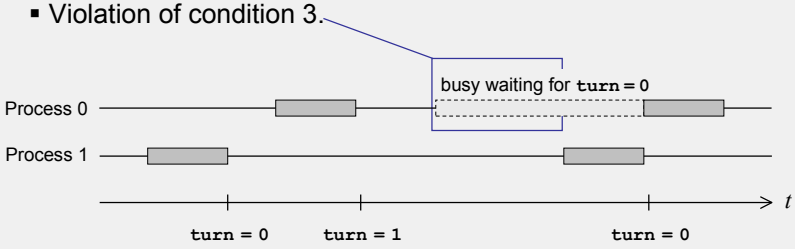
```
while (1) {
  while (turn != 1);
  critical_region();
  turn = 0;
  noncritical_region();
}
```

Process 1

Mutual Exclusion

Proposals for achieving mutual exclusion (continued)

- Strict Alternation (continued)
  - Busy waiting wastes CPU time.
  - No good idea when one process is much slower than the other.
  - Violation of condition 3.



Mutual Exclusion

Proposals for achieving mutual exclusion (continued)

- Peterson Algorithm

```
int turn;
bool interested[2];
```

shared variables

Two processes, number is either 0 or 1

```
void enter_region(int process) {
  int other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process && interested[other] == TRUE);
}

void leave_region(int process) {
  interested[process] = FALSE;
}
```


Mutual Exclusion

Peterson Algorithm (continued)

Assume process 0 and 1 both simultaneously entering **`critical_region()`**

Process 0:
```
other = 1
interested[0] = true
turn = 0
```

Process 1:
```
other = 0
interested[1] = true
turn = 1
```

Both are manipulating **`turn`** at the same time. Whichever store is last is the one that counts. Assume process 1 was slightly later, thus **`turn`** = 1.

```
while (turn == 0 && interested[1] == TRUE);
                    while (turn == 1 && interested[0] == TRUE);
```

Process 0 passes its while statement, whereas process 1 keeps busy waiting therein. Later, when process 0 calls **`leave_region()`**, process 1 is released from the loop.

- Good working algorithm, but uses busy waiting

Mutual Exclusion

Proposals for achieving mutual exclusion (continued)

- Test and Set Lock (TSL)

  Atomic operation at machine level. Cannot be interrupted. TSL reads the content of the memory word *lock* into register R and then stores a nonzero value at the memory address *lock*. The memory bus is locked, no other process(or) can access *lock*.

```
enter_region:   TSL R, lock
                CMP R, #0
                JNZ enter_region
                RET

leave_region:   MOV lock, #0
                RET
```

indivisible operation

- CPU must support TSL
- Busy waiting

Pseudo assembler listing providing the functions enter_region() and leave_region().