OPERATING SYSTEMS

by
Marwa Yusuf

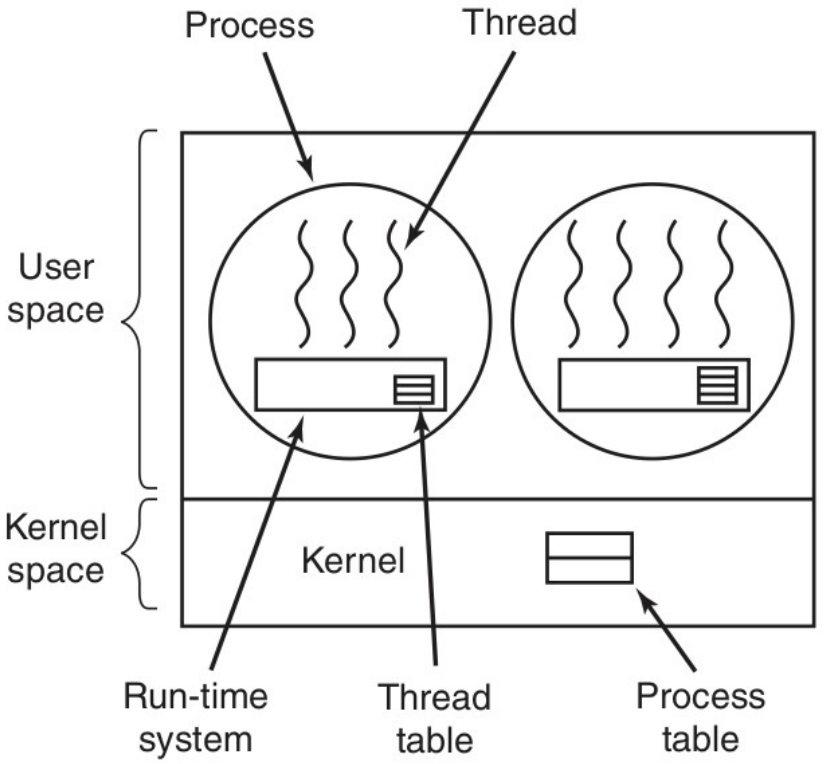
Lecture 5
Thursday 5-11-2020

Chapter 2 (2.2.4 to 2.3.2)

Processes and Threads

User Space Threads

- Thread library entirely in user space, kernel has no idea.
- Can be implemented for an OS with no thread support. (the only type in old days)
- Threads run on top of run-time system (a collection of procedures)
- Each process has a thread table managed by the run-time system. Table (PC, SP, REGs, state, ...etc)
- At thread switch, thread calls run-time that handles switching to another ready thread. This is much faster than calling the kernel (advantage).
  - May have 1 inst. To save/load all registers at one s



User Space Threads (cont.)

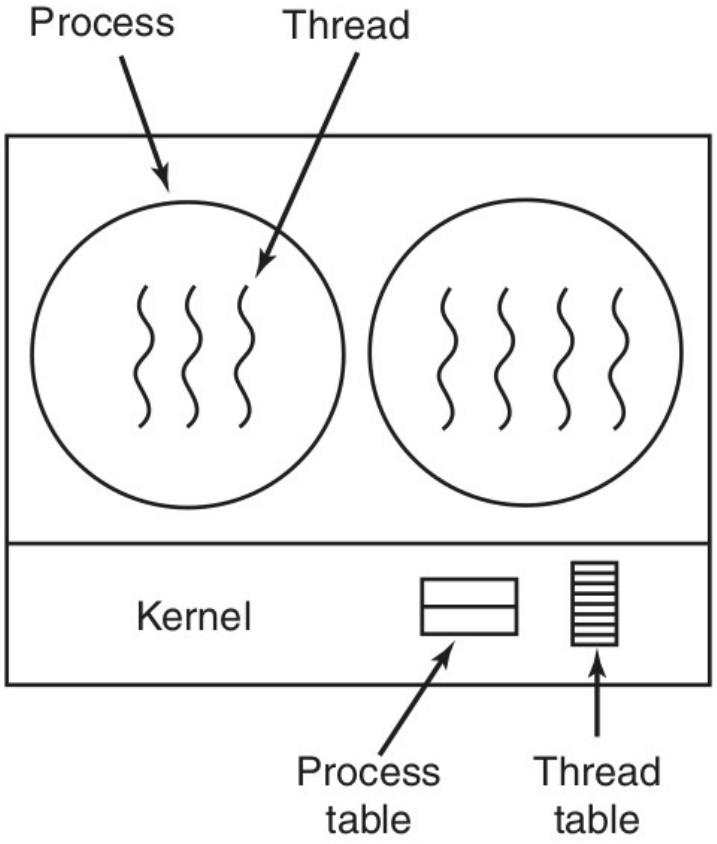
- Advantages:
  - Can be implemented for an OS with no thread support.
  - No trap, no context switch, no memory cache flush, … etc at thread switch, just a local procedure.
  - Customized scheduling algorithms per process (ex. Garbage collection thread)
  - Scale better (no kernel space needed)

User Space Threads (cont.)

- Problems:
  - Blocking system calls (If a thread blocks, the entire process will block)
    - Change the **read** to non-blocking, requires OS changes, undesirable, plus changing other programs.
    - **Select** call checks if the **read** will block (a **jacket** or a **wrapper**). Not very elegant way, changes the system call library.
  - Page faults (similar to the prev.)
  - The thread must yield voluntarily (no clock interrupts, no round-robin)
    - Run-time system require periodic clock interrupts (messy, overhead, possible interference with another thread requiring clock interrupt.)
  - The biggest problem: programmers use threads mostly for applications with a lot of (blocking) system calls. If already trapped to OS, the thread switch work is negligible. If the thread is CPU bound, what's the point?

Kernel Space Threads

- Thread creation and destruction are system calls, reflected on the thread table, managed by the kernel itself (no run-time).
- All blocking calls are system calls with greater cost.
- At thread block, the kernel chooses another thread either from the same process or from another one.
- To decrease costs, OS **recycle** threads (**not runnable** thread **reactivated**)
  - Recycling possible but not worth it in user level threa

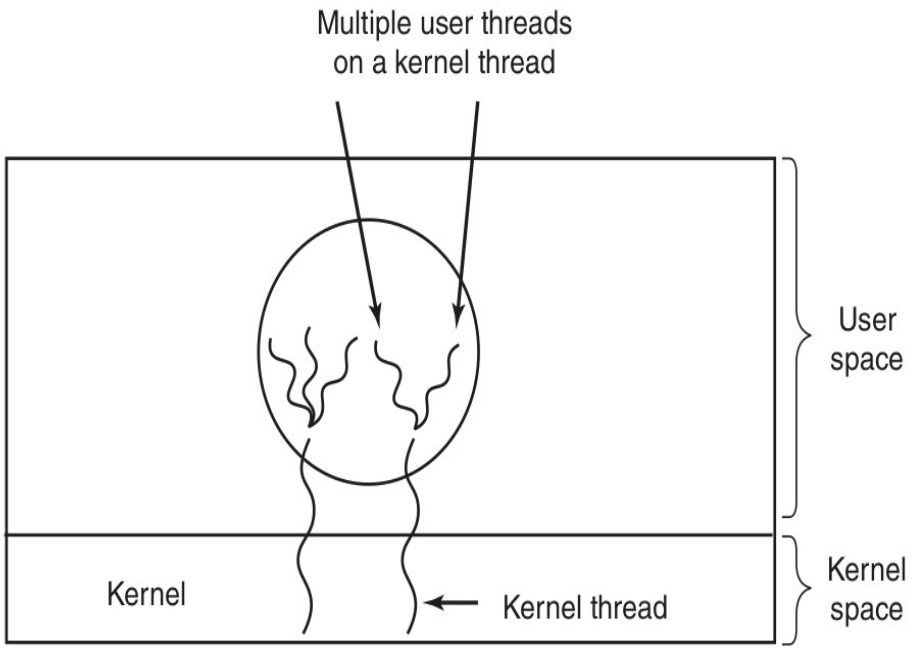


Kernel Space Threads (cont.)

- Problems:
  - What happens with **fork**?
    - If **exec** is coming, then 1 thread, otherwise duplicate.
  - Signals are delivered to processes. Which thread receives a signal?
    - The interested thread registers. But what if more than one thread register?

Hybrid

- Multiplex user level threads on top of kernel level ones.
- Program chooses how many threads on both levels.
- The most flexible.
- Kernel only knows and schedules ke threads. Multiple level threads may be multiplexed on top of them.
- User level threads are managed like pure user level threads.
- Each kernel thread has a set of user threads switching to use it.



Scheduler Activations

- Approach by Anderson et al. (1992) to improve the slowness of kernel level threads.
- Challenging bonus explained next online lecture by one of you. (for reading only).
  - Hint: read the related paper.

Pop-up Threads

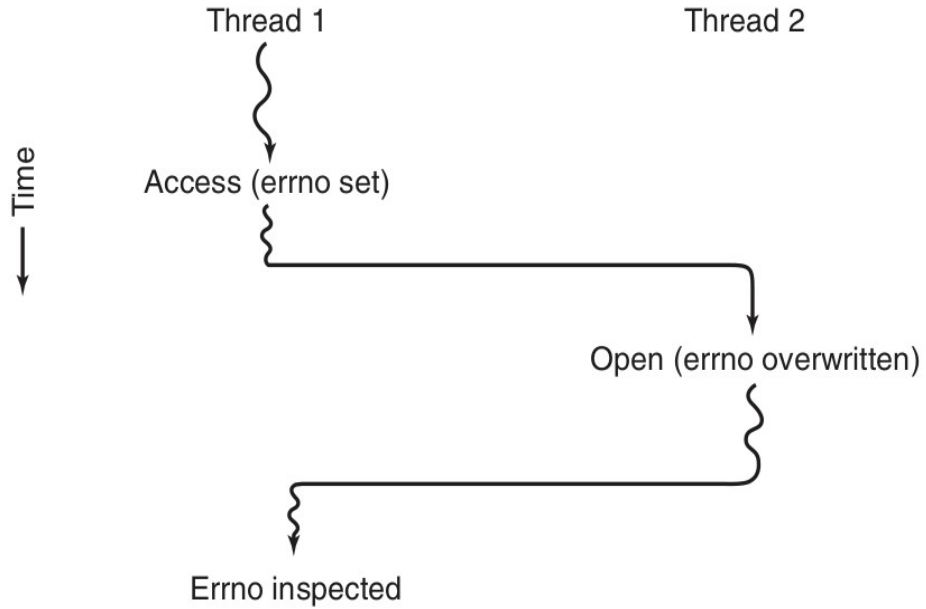
- Another bonus, similar to the previous (Are there any related papers?)

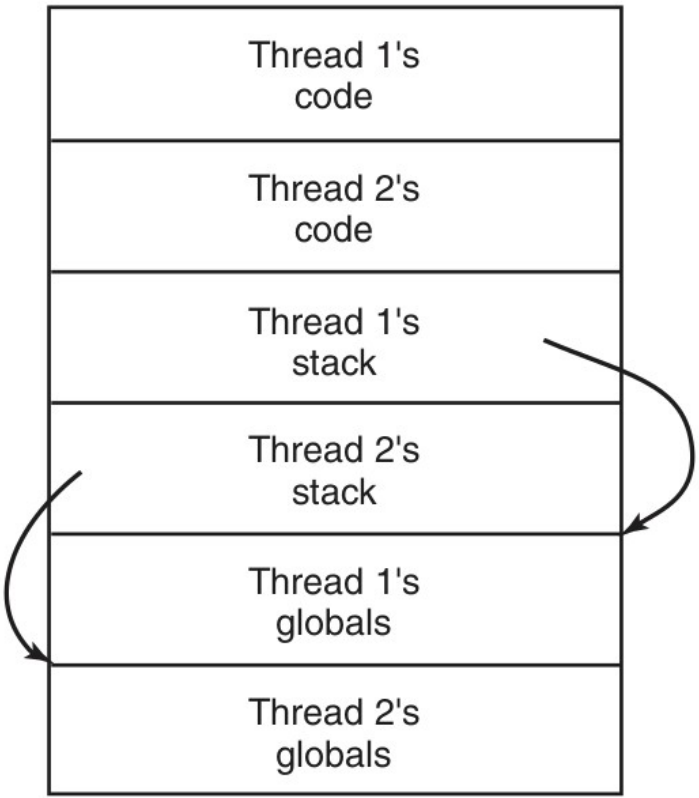
From Single-threaded to Multi-threaded

- Existing programs usually written to be single-threaded. Converting them is tricky.
- We will see some example problems.

Thread Globals

- Need for a thread global scope. Ex: *errno.*
  - Prohibit globals!! What about existing SW?
  - Private global area/thread → new scoping level + procedure local and program global.





Thread Globals (cont.)

- How to access those areas? Programming languages do not know how to express this new scope.
  - Allocate a chunk of memory /thread and pass to each procedure. Not elegant but works.
  - New library procedures:
    - `create_global("bufptr");` //in the heap or special area dedicated for the calling thread
    - `set global("bufptr", &buf);`
    - `bufptr = read global("bufptr");`

Non-reentrant Procedures

- Examples:
  - Sending messages over the network through a buffer.
  - *Malloc*: updating memory usage tables (free chunks linked list)
- Rewriting entire library → not trivial, may introduce errors.
- A jacket for each procedure (in use bit) blocking multi-access to the same procedure. → Limits parallelism.

Signals

- Some signal are thread specific, ex: alarm.
  - Kernel does not know about user level threads, how to deliver?
  - A process may have only one pending alarm, what about several threads calling alarm?
- Other signals like key strokes are not thread specific:
  - Who catch them?
  - What if 1 thread changes handlers?
  - What if a thread wants to catch a signal while another wants it to kill process?
- Signals are a source of chaos (single- or multi-threaded)

Stack Management

- Kernel provides more stack space in case of stack overflow.
- But what if the kernel does not know about the existence of threads?

Is it so dark?

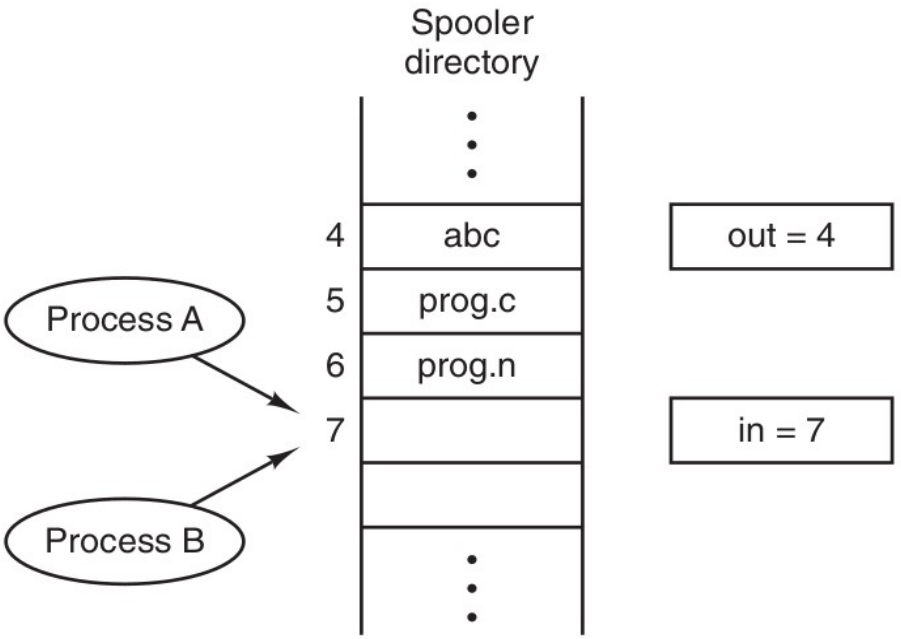
- No, these problems are solvable.
- However, solutions may require redesign of system calls and rewriting of library routines, ...etc, while keeping backward compatibility
- NOTHING is free of charge.

Interprocess Communication (IPC)

- Ex: shell pipeline (ls | grep Music)
- 3 issues:
  1) How to pass info to another process?
  2) How two processes do not collide with each other (ex: seat reservation)?
  3) Proper sequencing in case of dependencies.
- The last 2 issues apply equally to threads. (why not the first one?)

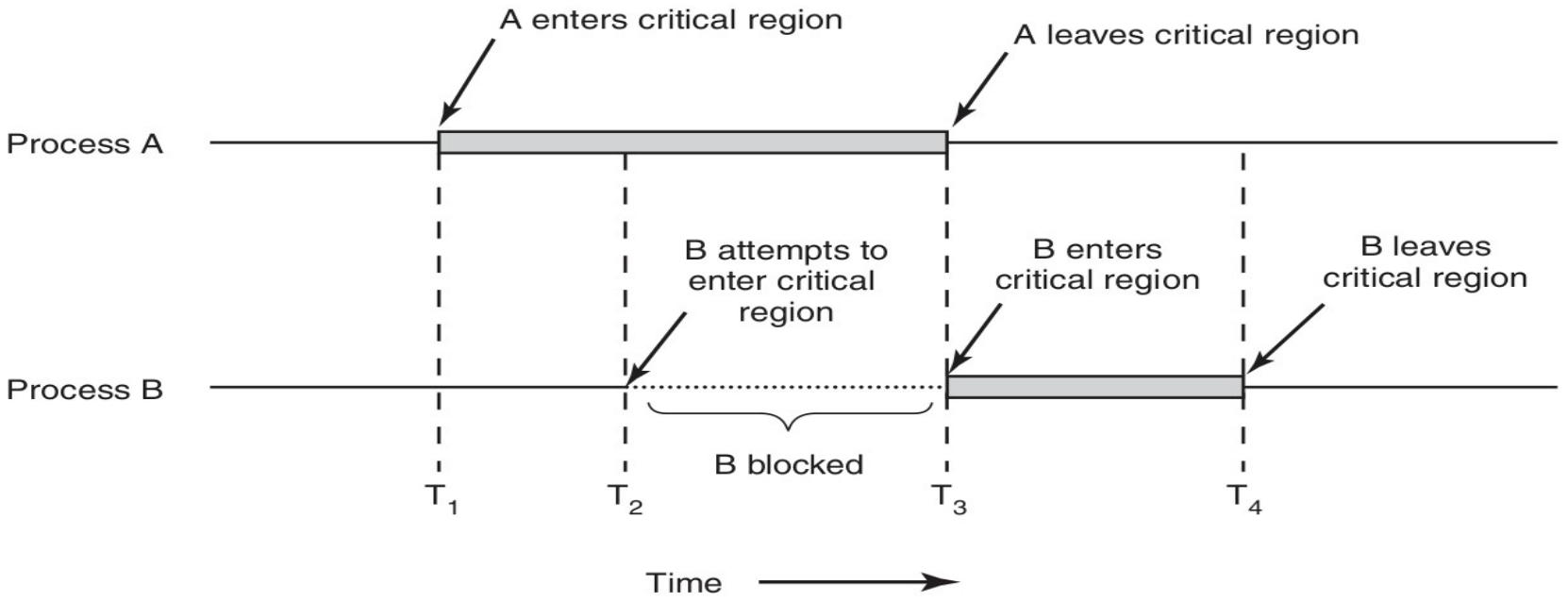
Race Condition

- In some OSs, processes may share some memory (in RAM, kernel data structure, file).
- Ex: Spooler directory.
- X = 7
- Write in 7 “text”
- Y = 7
- Write in 7 “data”
- In = 8



Critical Regions (Sections)

- The solution to race condition is to achieve **mutual exclusion** protecting **critical sections**.



Critical Regions (cont.)

- Just mutual execution is not enough by itself.
- Conditions for correct and efficient cooperation:
  1) Only one process at a time in the critical region.
  2) No assumptions about speed or CPU number.
  3) No process outside critical region can block another process.
  4) No process should wait forever.