Operating Systems
Processes and Threads

Tomasz Kruk, Ph.D.
Faculty of Electronics and Information Technology
Warsaw University of Technology

https://www.linkedin.com/in/tomasz-jordan-kruk

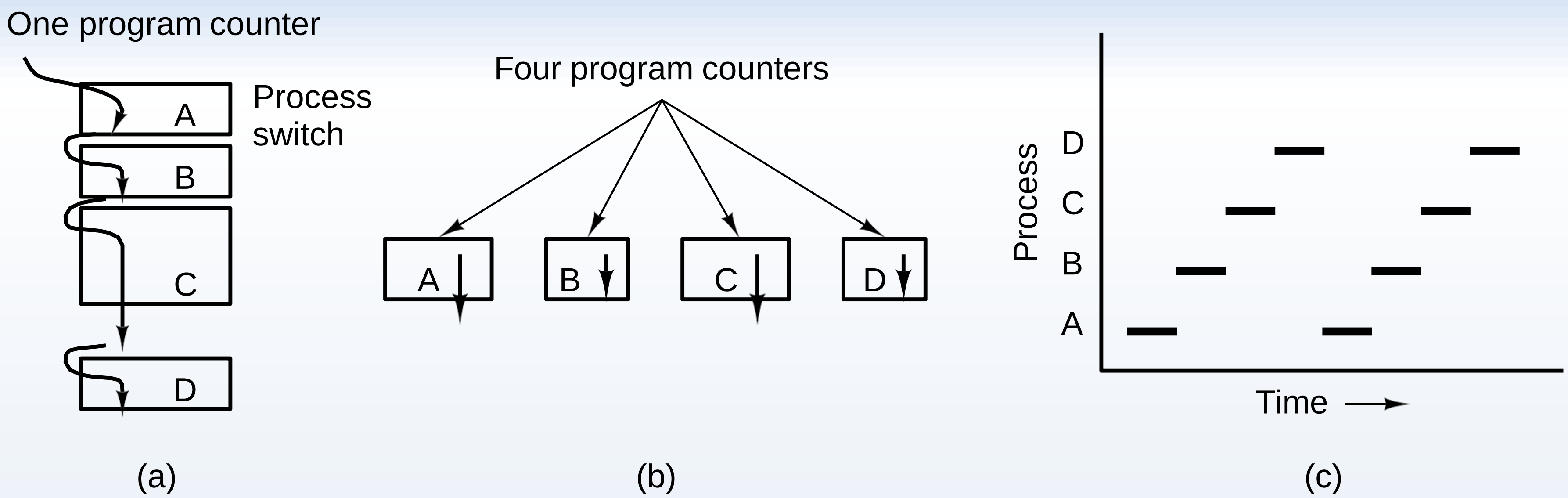
Process in the Operating System

Process - an abstraction of a running program with its computing environment. Process is a basic dynamic object in the operating system.

Requirements to be met by the operating system with reference to processes:

- √ interleaving the execution of multiple processes to maximize processor utilization while providing reasonable response time,
- √ allocating resources to processes in conformance with a specified policy, while at the same time avoiding deadlock,
- √ supporting interprocess communication,
- √ supporting user creation of processes.

Multiprogramming



- process must not be programmed with built-in assumptions about timing,
- the difference between a process and a program,
- processes **sequential**, **concurrent**, **parallel** and **distributed**,

Process Creation and Termination

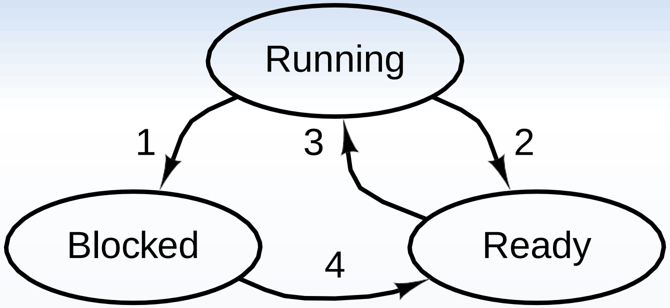
Four principal events that cause processes to be **created**:

- √ system initialization,
- √ execution of a process creation system call by a running process,
- √ a user request to create a new process,
- √ initiation of a batch job.

Process may **terminate** due to one of the following conditions:

- √ normal exit (voluntary),
- √ error exit (voluntary),
- √ fatal error (involuntary),
- √ killed by another process (involuntary).

Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

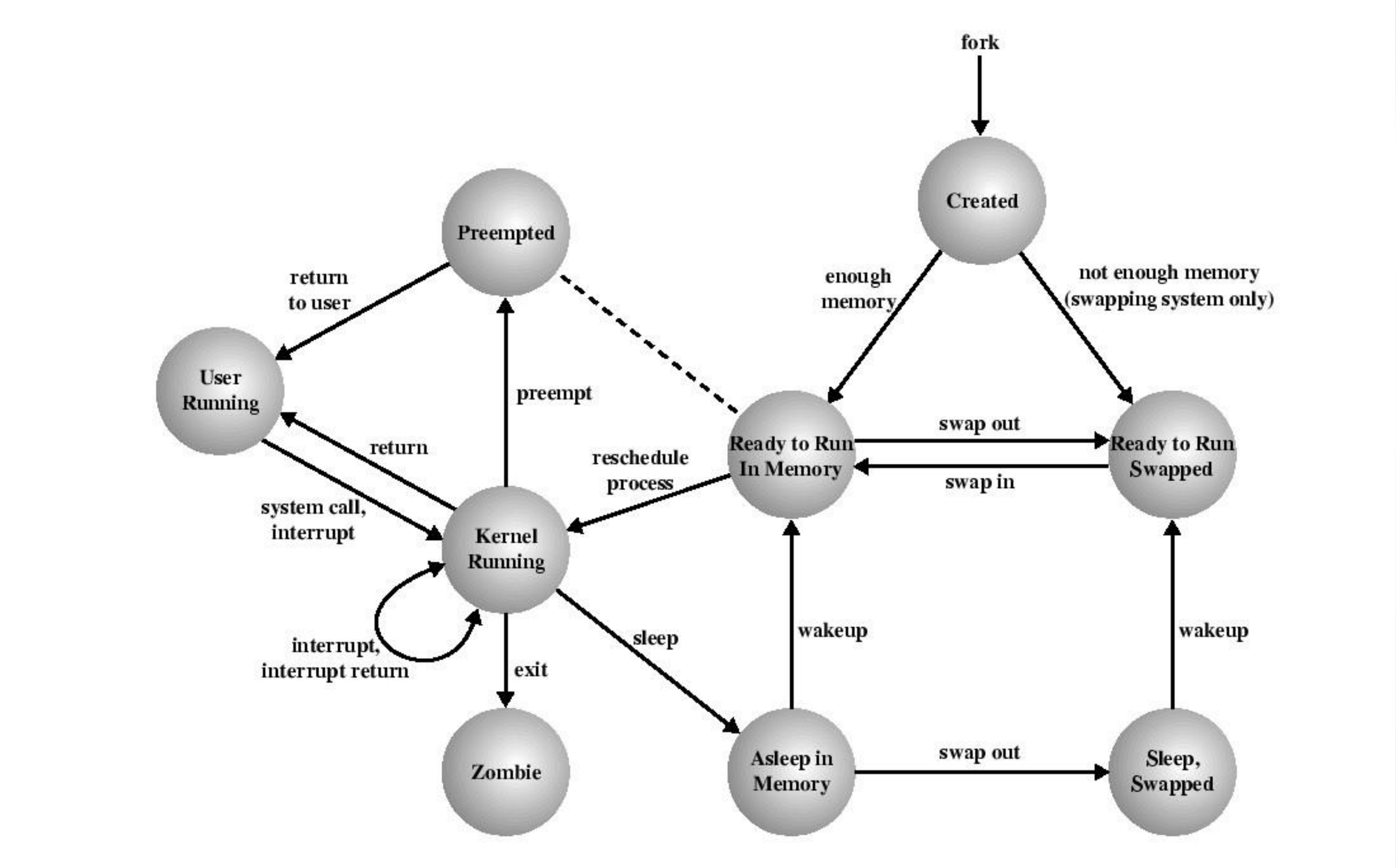
The basic states of a process:

- √ **running** - actually using the CPU at that instant,
- √ **ready** - runnable, temporarily stopped to let another process run,
- √ **blocked** - unable to run until some external event happens.

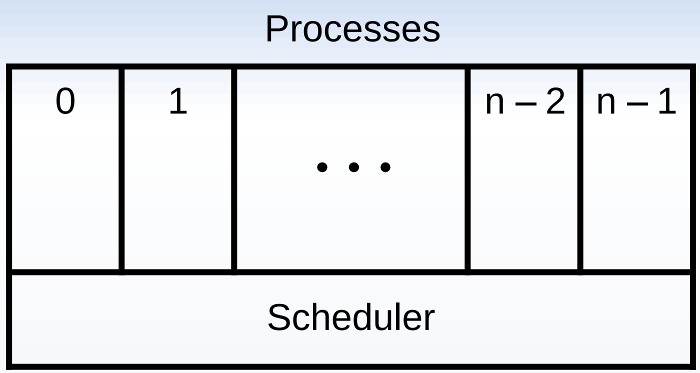
Process States in the Unix System

- **user running**,
- **kernel running**,
- **ready to run, in memory**,
- **asleep in memory**,
- **ready to run, swapped**,
- **sleeping, swapped**,
- **preempted**,
- **created**,
- **zombie**.

Unix Process State Transition Diagram



Scheduler



The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer there are sequential processes.
For processor allocation for particular procesess a piece of an OS kernel called **scheduler** is responsible.

Process implementation:

- the OS maintains a table (an array of structures) called the **process table**, with one entry per process. Sometimes those entries are called **process descriptors** or **process control blocks, PCB**.

Some Fields of a Typical Process Table Entry

Process management	Memory management
Registers	Pointer to text segment
Program counter	Pointer to data segment
Program Status Word	Pointer to stack segment
Stack pointer	
Process state	
Priority	
Scheduling parameters	**File management**
Process ID	*root* directory
Parent proces	Working directory
Process group	File descriptors
Signals	User ID
Time when process started	Group ID
CPU time used	
Time of next alarm	

Interrupt Revisited

- √ when an I/O device has finished the work given to it, it causes an interrupt by asserting a signal on a bus line it has been assigned,
- √ the signal detected by the **interrupt controller** chip on the motherboard,
- √ if no interrupts pending, the interrupt controller processes the interrupt immediately - it puts a number on the address lines specifying which device wants attention and asserts a signal that interrupts the CPU,
- √ the CPU stops current work and uses the number on the address lines as an index into a table called the **interrupt vector** to fetch a new program counter,
- √ the counter points to the start of the corresponding interrupt service procedure,
- √ shortly after starting running, the interrupt service procedure acknowledges the interrupt by writing a certain value to one of the interrupt controller's I/O ports - the controller is now free to issue another interrupt,
- √ the hardware always saves certain information before starting the service procedure, at least the program counter but in some architectures all the visible registers and a large number of internal ones.

Activities of the OS When an Interrupt Occurs

1. Save any registers (including the PSW) that have not already been saved by the interrupt handler.
2. Set up a context for the interrupt service procedure. Doing this may involve setting up the TLB, MMU and a page table.
3. Set up a stack for the interrupt service procedure.
4. Acknowledge the interrupt controller. If there is no centralised interrupt controller, reenable interrupts.
5. Copy the registers from where they were saved (possibly some stack) to the process table.
6. Run the interrupt service procedure. It will extract information from the interrupting device controller’s registers.
7. Choose which process to run next.
8. Set up the MMU context for the process to run next. Some TLB setup may also be needed.
9. Load the new process’ registers, including the PSW.
10. Start running the new process.

Threads of Execution

When there is a need for concurrent threads of execution organized as a group of processes, having separated protected address spaces means:

- √ from the point of view of protection: an advantage, but here we protect our processes against our processes,
- √ from the point of view of communication: a drawback,
- √ from the point of view of the level of simplicity in sharing resources: a drawback,
- √ from the point of view of performance: a drawback, at least if processes not parallel,

Thus, maybe we should consider putting together cooperating threads of execution into one shared address space, and this would meant:

- √ from the point of view of protection: a drawback, but we are the author of the cooperating threads codes and we should know what we do,
- √ from the point of view of communication: an advantage,
- √ from the point of view of the level of simplicity in sharing resources: an advantage.

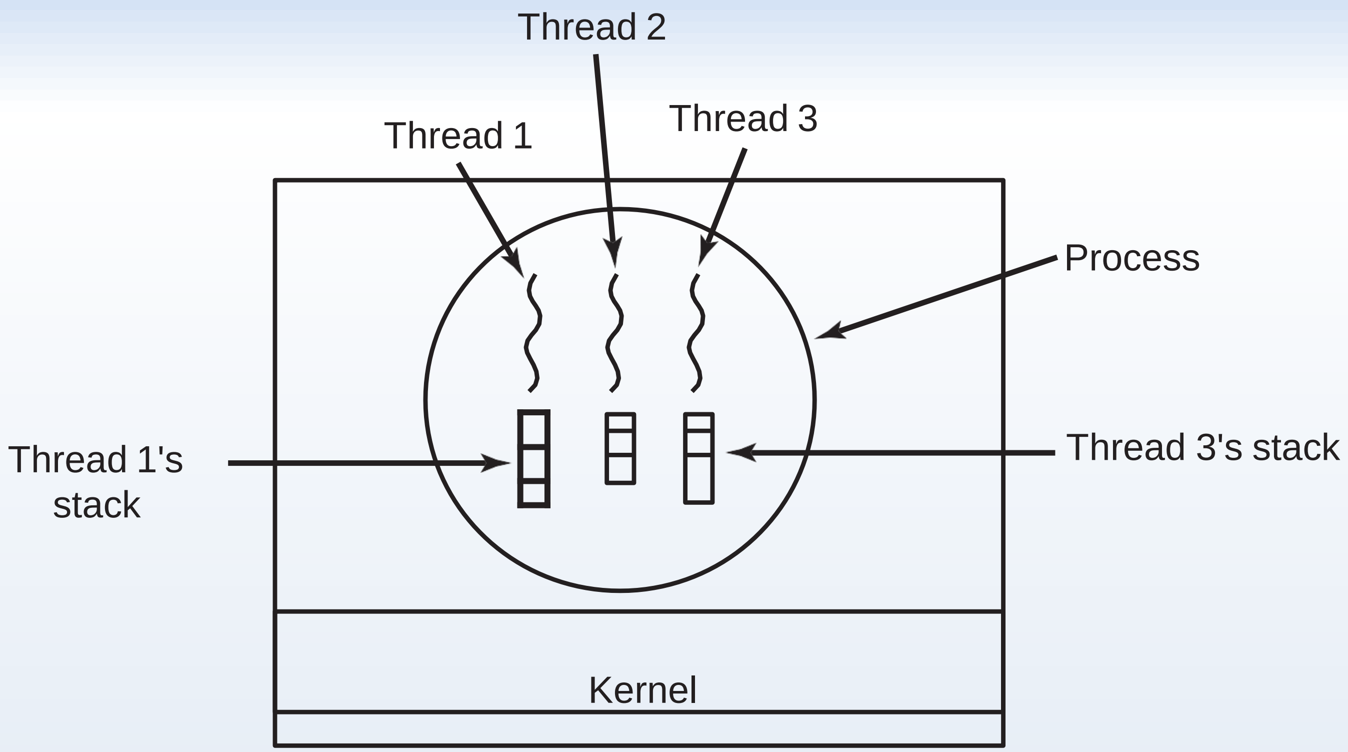
Processes and Threads Attributes

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	

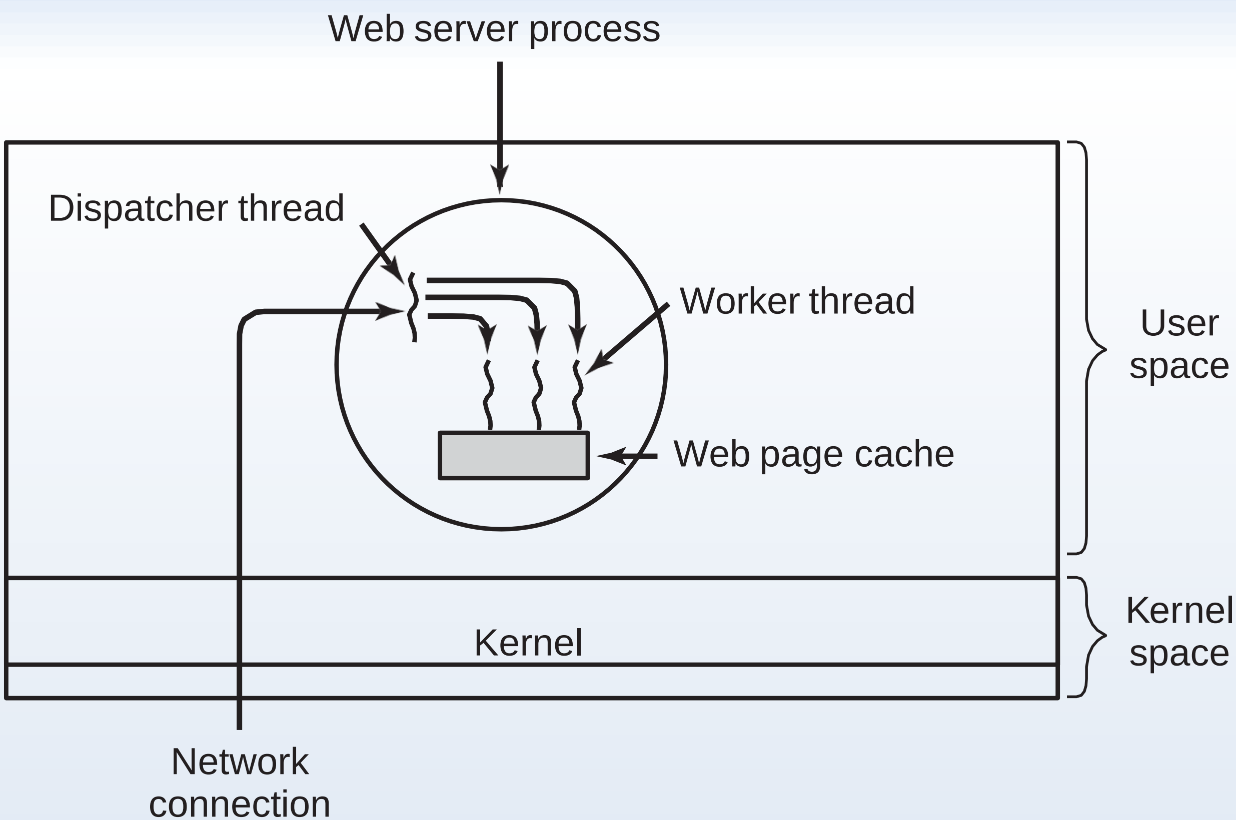
Threads of the same process may exchange information with usage of global variables of the process.

- √ what with threads aspects when the subprocess is created?
- √ what is the correct layer for servicing signals?

Threads Stack



Multithreaded Server



An Outline of the Multithreaded Server Algorithm

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

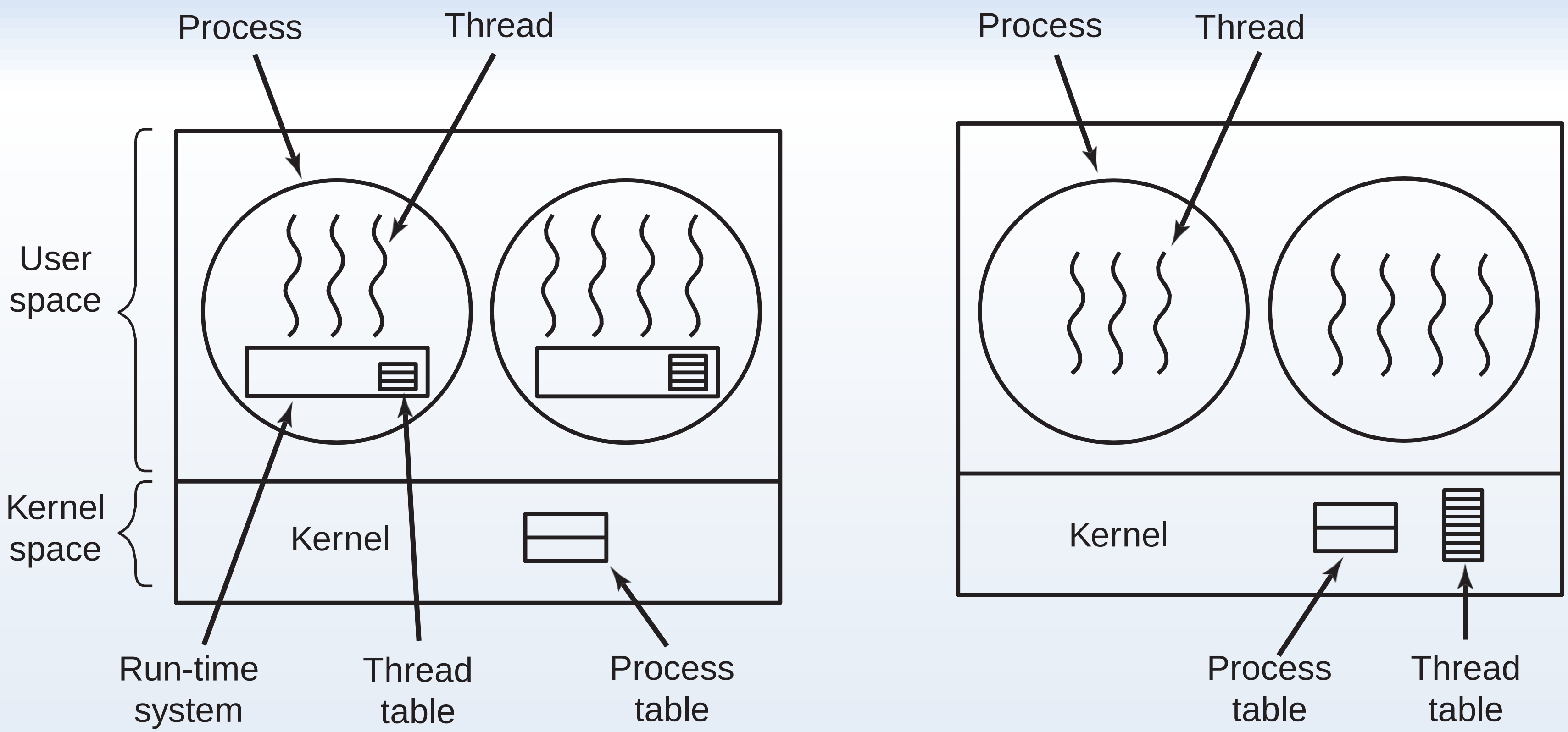
(b)

Methods of Server Construction

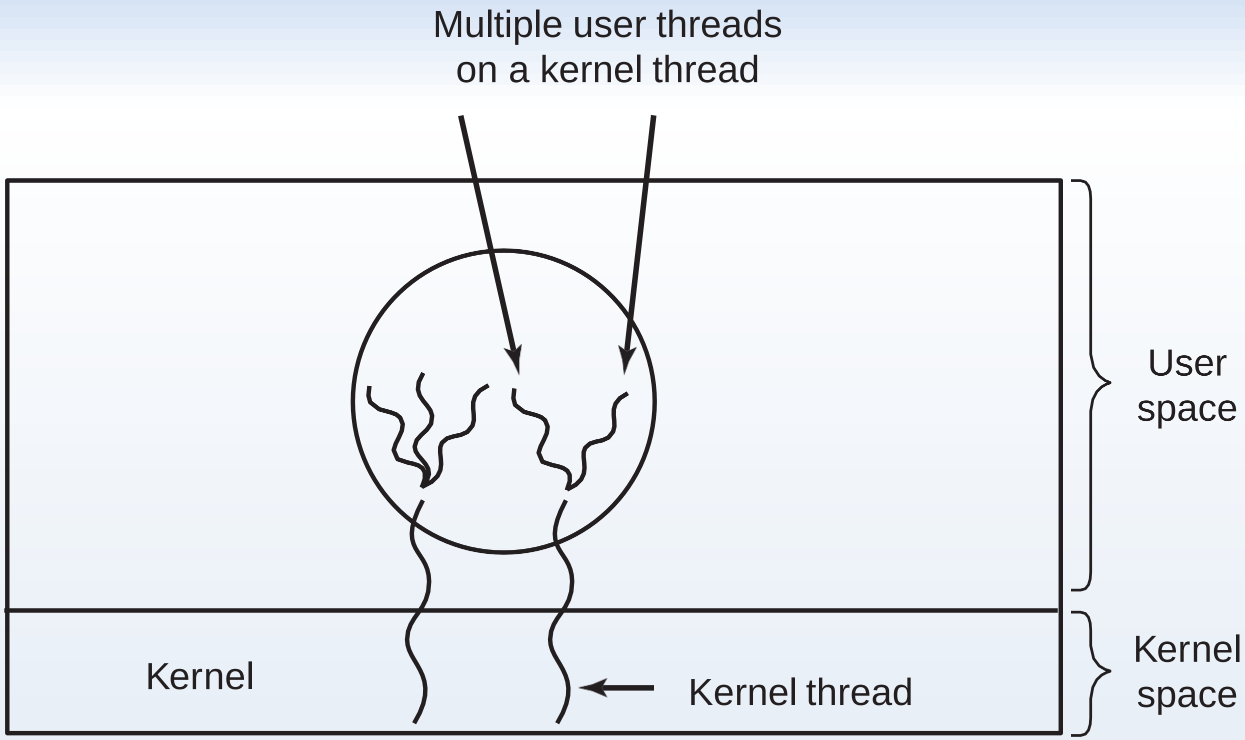
Three ways to construct a server

- √ **threads** - parallelism, blocking system calls,
- √ **single-threaded process** - no parallelism, blocking system calls,
- √ **finite-state machine** - parallelism, nonblocking system calls, interrupts.

Kernel-level threads and User-level Threads



Threads – Hybrid Solutions

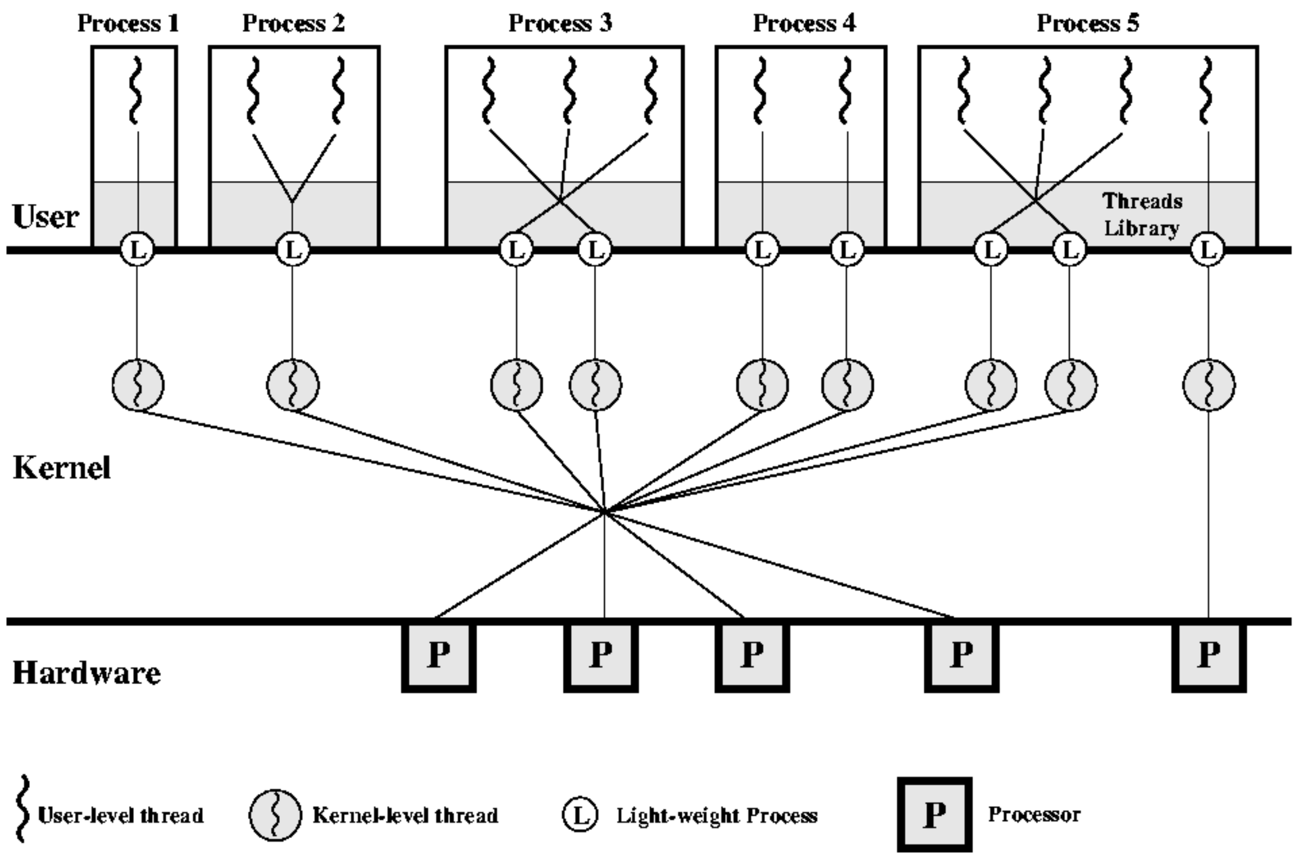


Multithreaded Architecture under Solaris OS

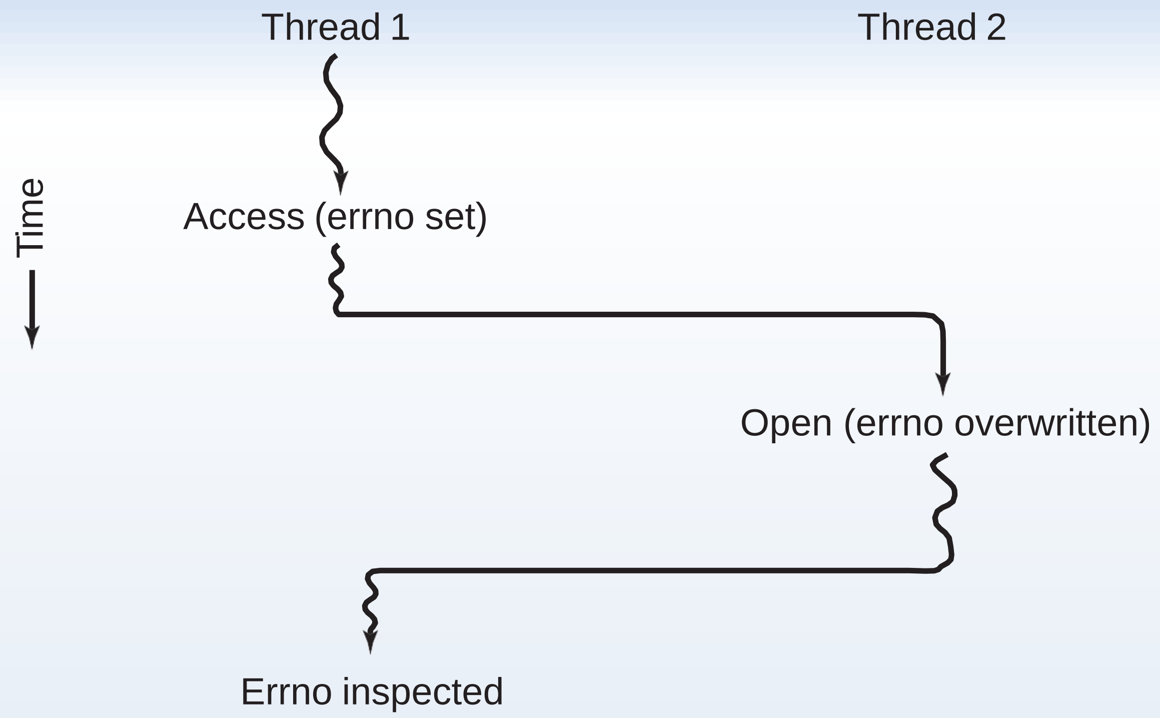
Solaris makes use of four separate thread-related concepts:

- √ **Process** - the normal Unix process,
- √ **User-level threads** - ULTs, implemented through a threads library in the address space of a process,
  - ⋆ invisible to the operating system,
  - ⋆ interface for application parallelism.
- √ **Lightweight processes** - LWPs, a mapping between ULTs and kernel threads,
  - ⋆ each LWP supports one or more ULTs and maps to one kernel thread,
  - ⋆ LWPs are scheduled by the kernel independently,
  - ⋆ LWPs may execute in parallel on multiprocessors.
- √ **Kernel threads** fundamental entities that can be scheduled and dispatched to run on one of the system processors.

Threads under Solaris – an Example

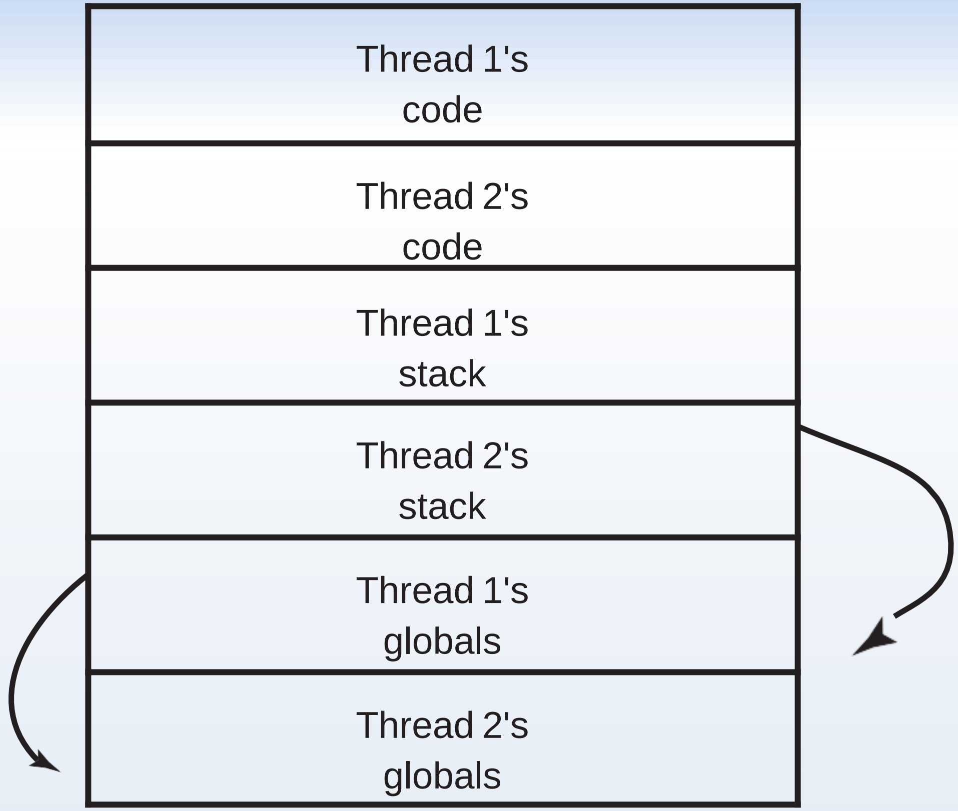


Migration to the Multithreaded Code

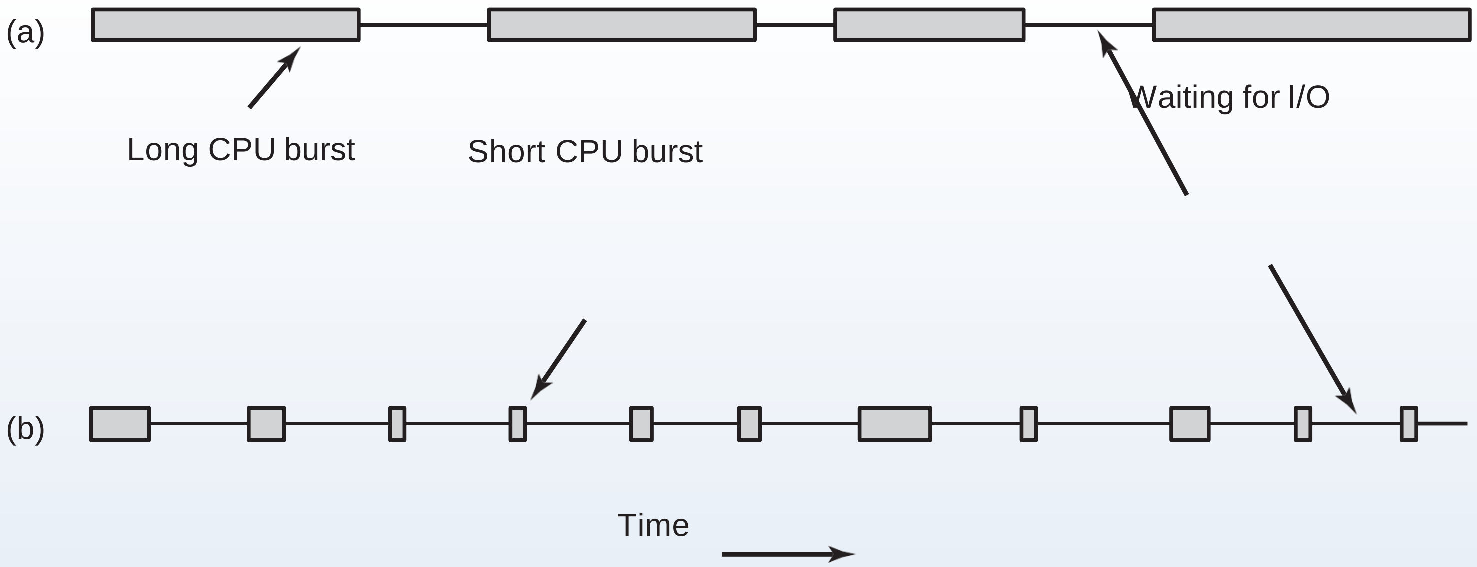



```
extern int *__errno();
#define errno (*(__errno()))
```

Private Global Variables



Processor-bound and I/O-bound Processes



Processes Scheduling

There are two basic scheduling techniques:

- √ **nonpreemptive scheduling**,
- √ **preemptive scheduling**.

There are different requirements for different environments: batch systems, interactive systems, real-time systems.

Features of the Good Scheduling Algorithm

All systems

- √ **fairness** - giving each process a fair share of the CPU, **policy**
- √ **enforcement** - seeing that stated policy is carried out,
- √ **balance** - keeping all parts of the system busy.

Batch systems

- √ **throughput** - maximize jobs per hour,
- √ **turnaround time** - minimize time between submisission and terminantion,
- √ **CPU utilization** - keep the CPU busy all the time.

Interactive systems

- √ **response time** - respond to requests quickly,
- √ **proportionality** - meet users' expectations.

Real-time systems

- √ **meeting deadlines** - avoid losing data,
- √ **predictability** - avoid quality degradation in mulimedia systems.

Scheduling in Batch Systems

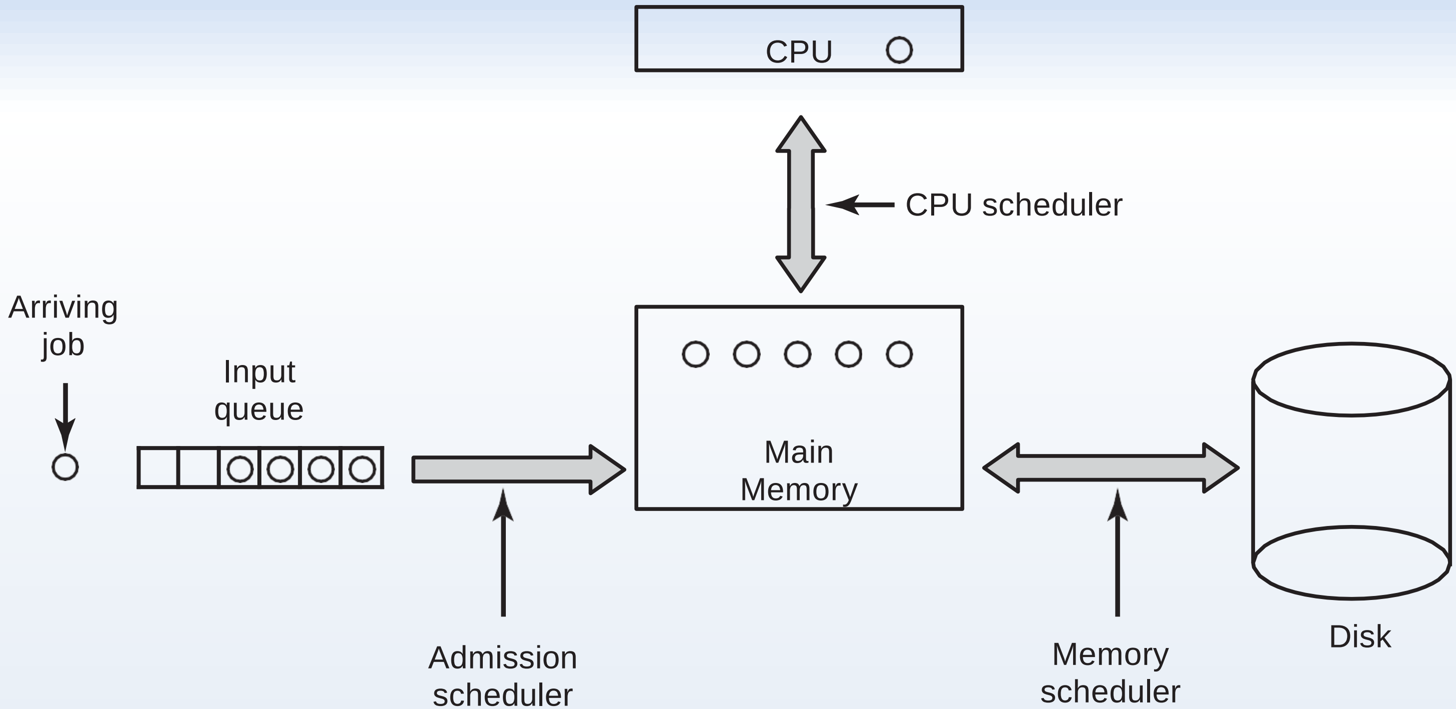
(a) (b)

Shortest Job First Scheduling

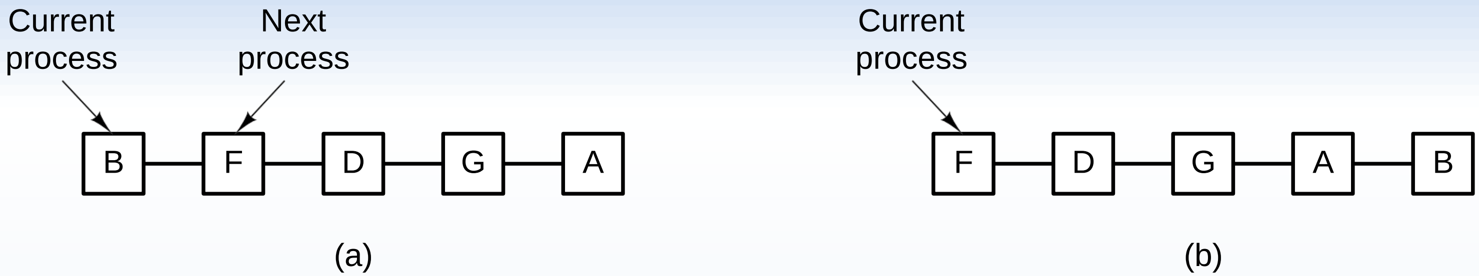
Scheduling in batch systems

- FCFS, **First-Come First-Served**,
- SJF, **Shortest Job First**,
- SRTN, **Shortest Remaining Time Next**,
- **Three-Level Scheduling**.

Three-Level Scheduling



Scheduling in Interactive Systems

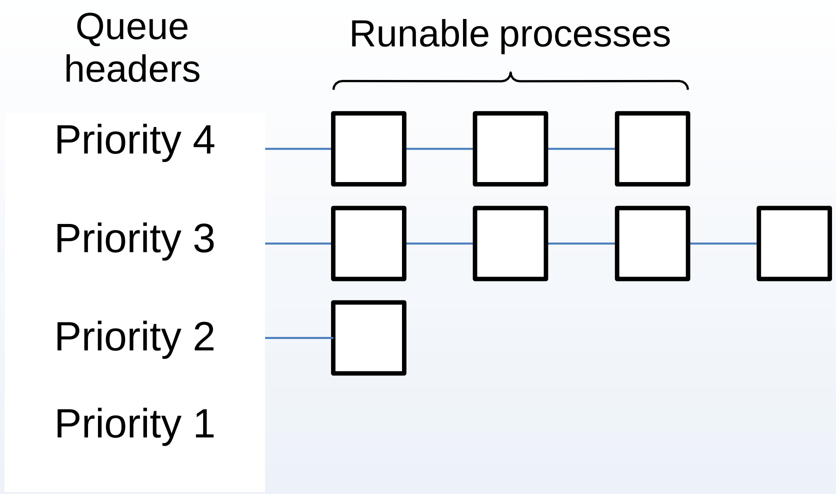


Round-Robin scheduling

Scheduling in interactive systems

- Round-Robin algorithm,
- priority scheduling,
- shortest process next (estimation).

Scheduling with Classes of Priorities



Scheduling in Real-time Systems

- systems with soft and hard requirements,
- periodic and aperiodic events,

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

A real-time system that meets this criteria is said to be **schedulable**.

- C_i time of one service of periodic event,
- P_i period of periodic event occurence.