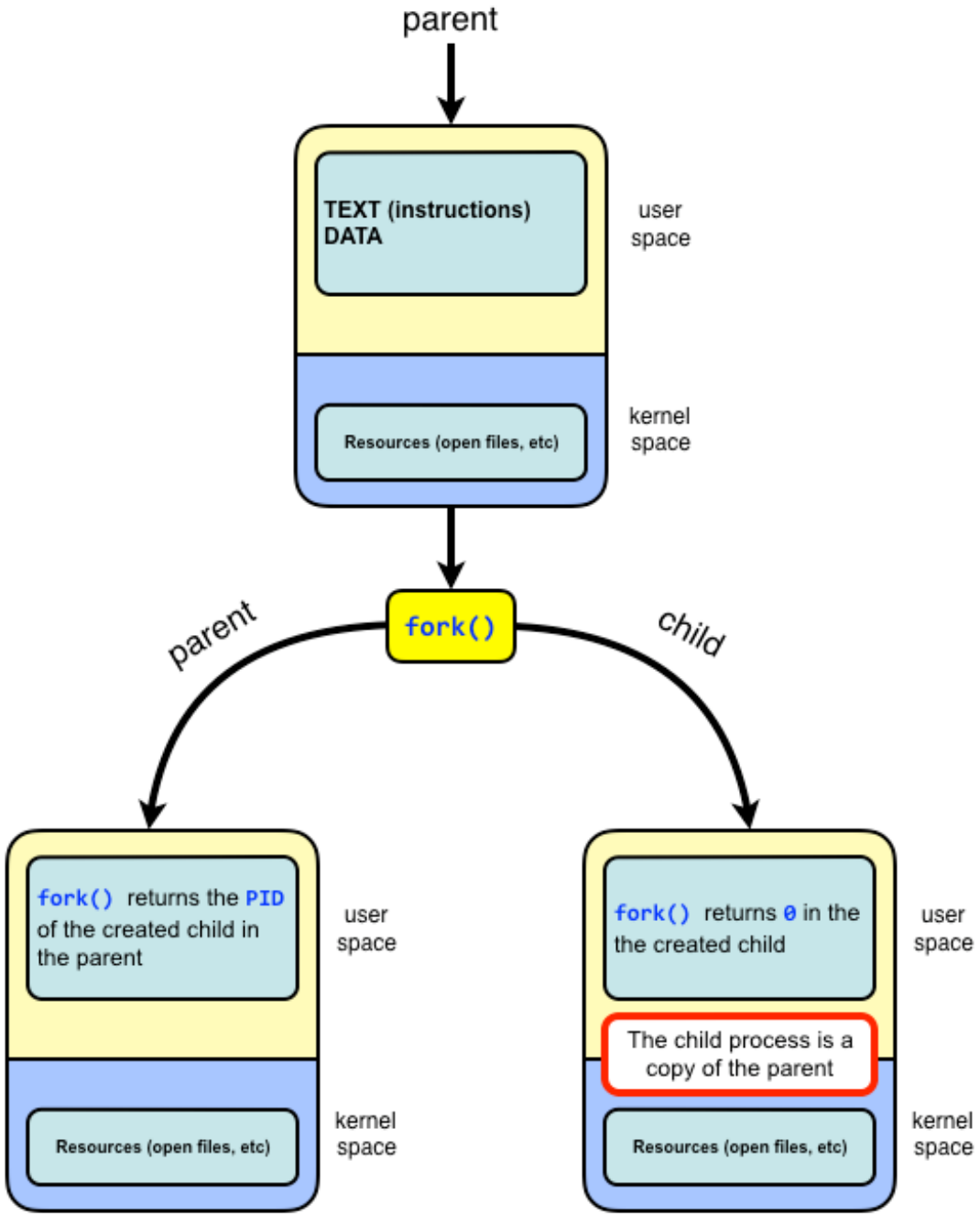
CSC 256: Final Review

Zonghua Gu
Department of Computer Science,
Hofstra University

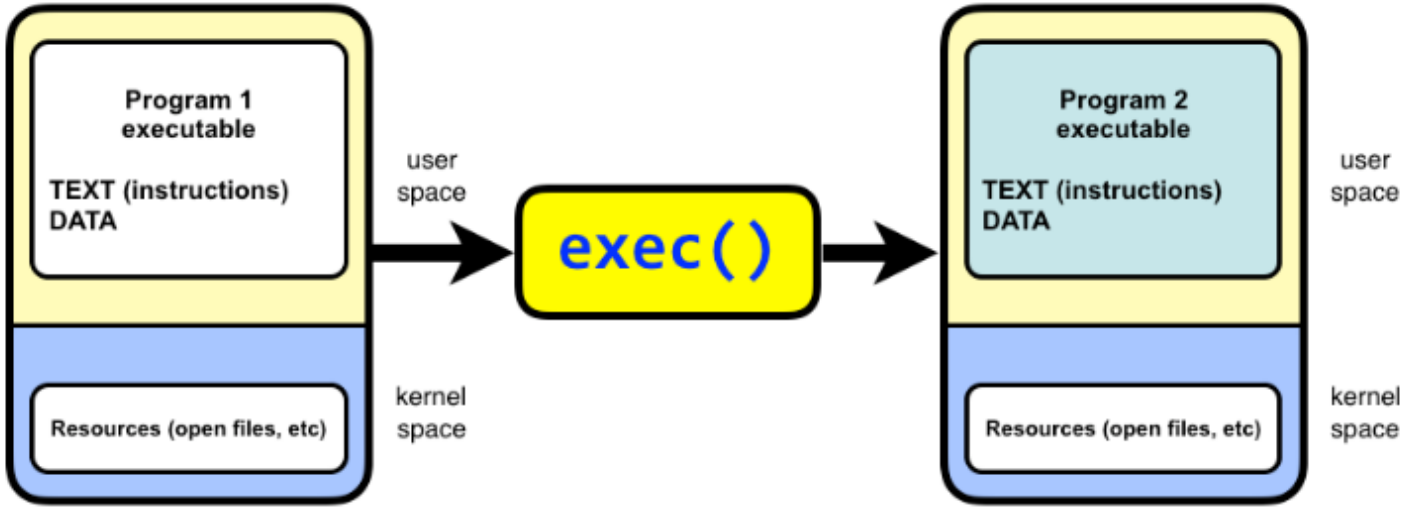
fork()

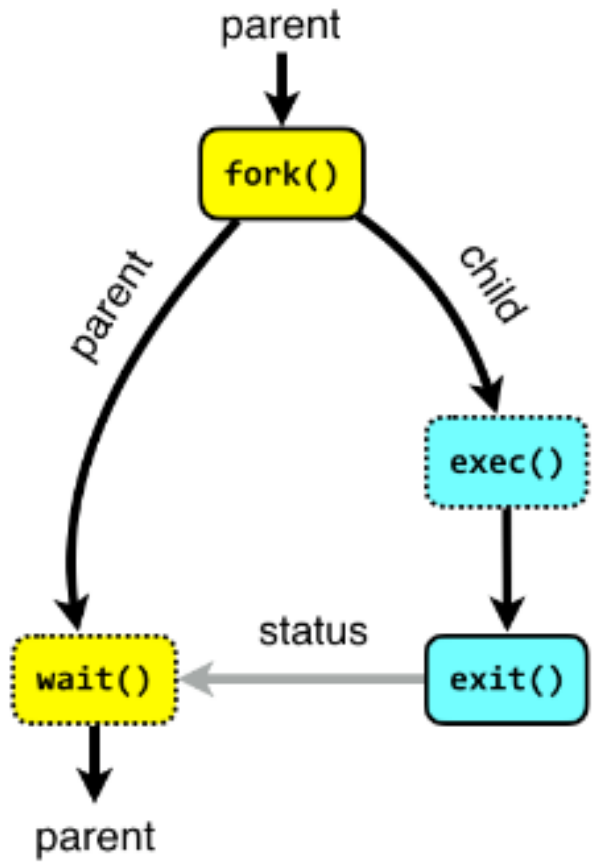
- A function without any arguments
  - **ret = fork()**
- Both **parent process** and **child process** continue to execute **the instruction following the fork()**
- The return value indicates which process it is (**parent** or **child**)
  - **ret > 0 (**pid of child process**)**: code running in the parent process,
  - **ret == 0**: code running in the newly-created child process
  - **ret == -1**: an error or failure occurred when creating new process
- Fun analogy: imaging you are a process after fork, but you don't know if you are the child or parent process, as if you are running inside of a Matrix. But you can identify which process you are running, by looking up to the sky and see the ret value from fork()
- Child process is a duplicate of its parent process and has same
  - **instructions, data, stack**
- Child and parents have different
  - **PIDs, memory spaces**



exec()

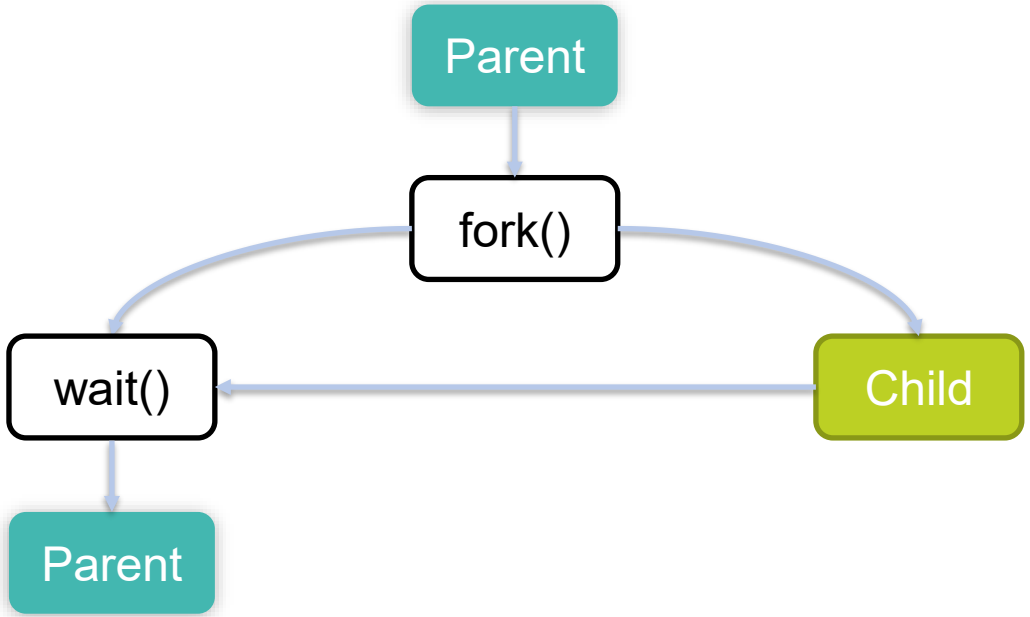
- **exec(cmd, argv)** replaces the current process image with a new process image specified by the path to an executable file.
  - It does not return. It starts to execute the new program.
- There is a family of **exec(),** e.g., **execl(), execvp()**
  - **execl()** takes a variable number of arguments that represent the program name and its arguments.
    - » int execl(const char *path, const char *arg, ..., NULL);
  - **execvp()** takes an array of arguments instead of a variable-length argument list
    - » int execvp(const char *file, char *const argv[]);





wait()

- Let the parent process wait for the completion of the child process
  - **pid = wait()**
- **wait()** suspends the execution of the calling process until one of its child processes terminates.
  - When a child process terminates, **wait()** retrieves its termination status and allows the system to clean up the resources associated with that child. If the parent does not call **wait()** to collect the child's exit status, the child becomes a zombie process, which means its PCB persists in the process table, even though it is no longer running.
    - » While zombie processes do not consume processor or memory resources, they occupy entries in the process table. The process table is of finite size, and if too many zombie processes accumulate, it can prevent new processes from being created.
  - If there are multiple child processes, **wait()** does not allow the parent to specify which child process to wait for. **waitpid(pid)** is an advanced version of wait. It allows the parent process to specify which child process (or group of processes) it wants to wait for.

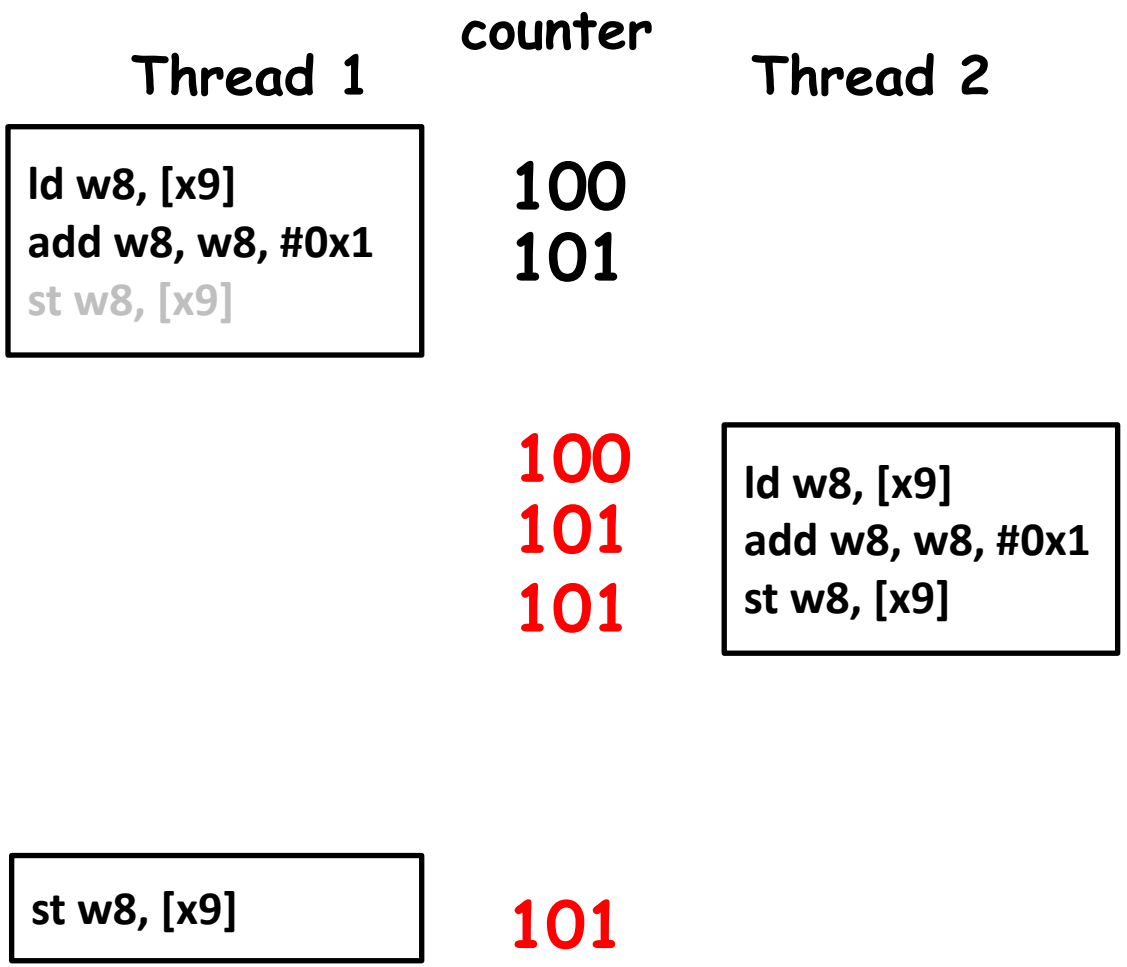


L2 Summary

- Processes
  - In OS, process is a running program and has an address space
  - We use process API to create and manage processes
  - fork() to duplicate a process, exec() to replace the command
- Threads:
  - Multiple threads per process / address space
  - Kernel threads are much more efficient than processes, but they're still not cheap
  - User-level threads are very efficient

Race Condition

- Incrementing **counter** has 3 instructions in assembly code:
- **ld w8, [x9]**: Read the value of counter at memory address x9 into register w8
- **add w8, w8, #0x1**: increment the value of register w8 by 1
- **st w8, [x9]**: write the new value of counter in register w8 to memory address x9
- When both threads read the same value of counter before writing to it, counter is incremented only by 1 instead of by 2!
- Note: threads in the same process share the same memory space, but have separate registers. So in both threads, [x9] refers to the same memory address at x9, but w8 refers to different registers in each thread.



counter++;

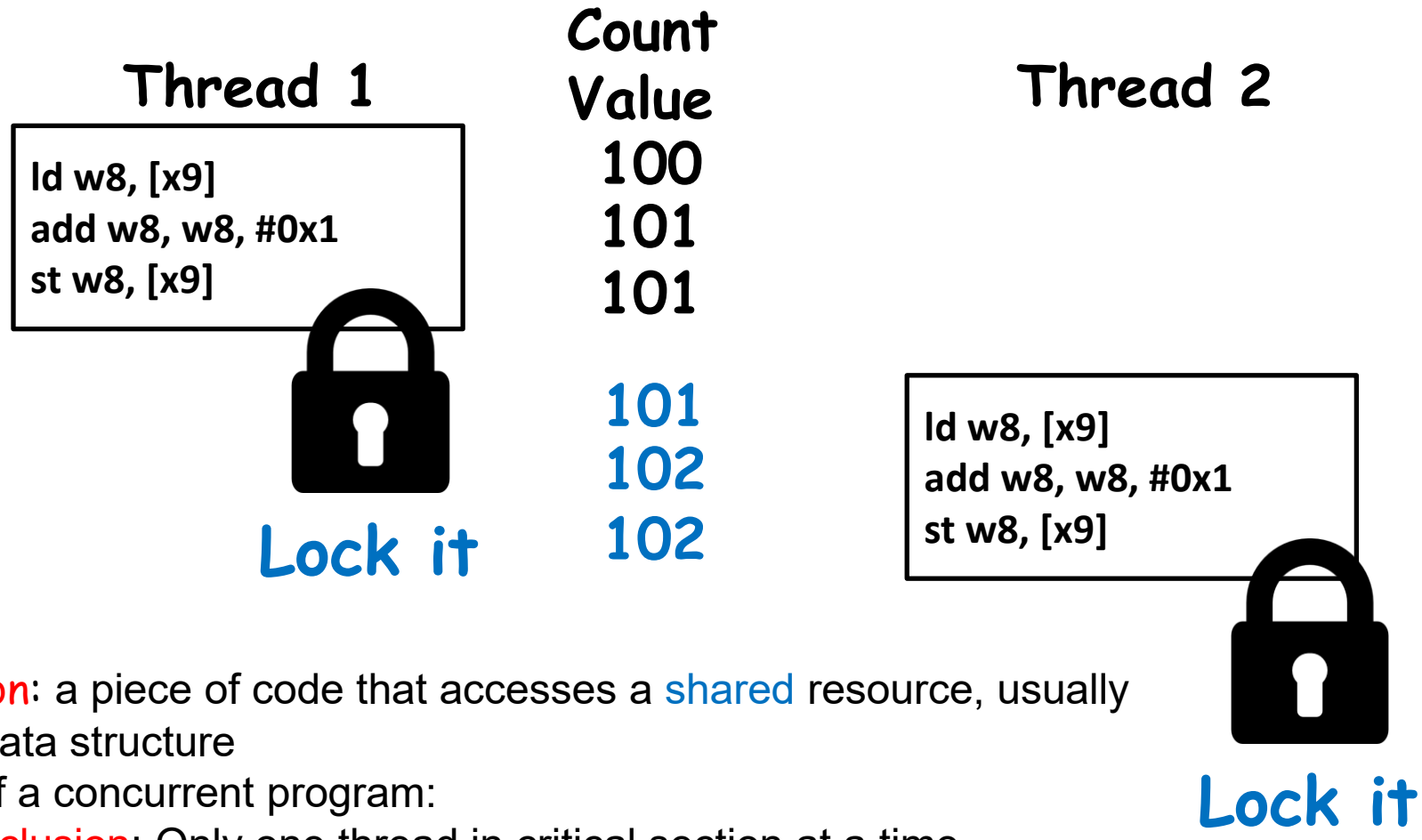
```
ld w8, [x9]
add w8, w8, #0x1
st w8, [x9]
```

Thread 1

```
ld w8, [x9]
add w8, w8, #0x1
st w8, [x9]
```

Thread 2

Lock to Protect a Critical Section



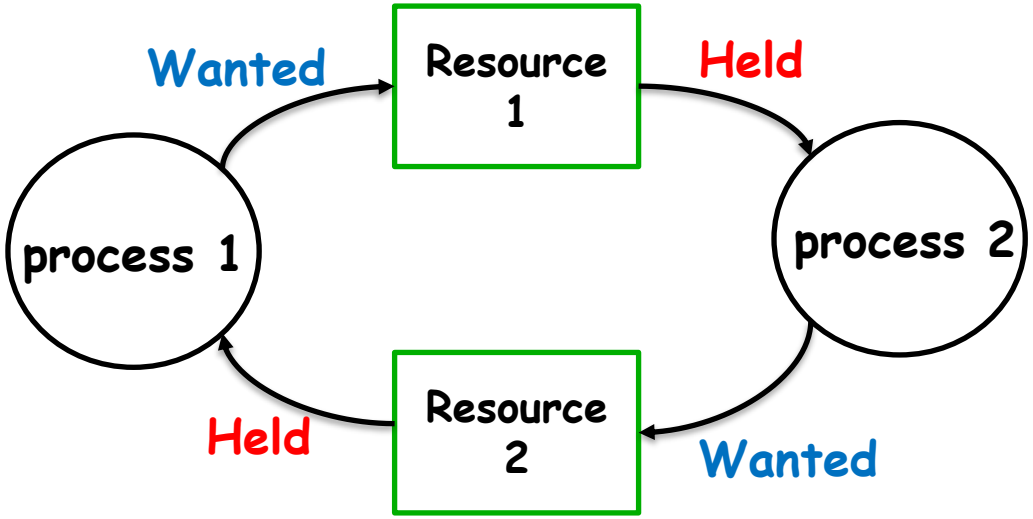
- Critical section: a piece of code that accesses a shared resource, usually a variable or data structure
- Correctness of a concurrent program:
  - Mutual exclusion: Only one thread in critical section at a time
  - Progress (deadlock-free): If several simultaneous requests, must allow one to proceed
  - Bounded waiting (starvation-free): Must eventually allow each waiting thread to enter

Semaphores

- Semaphores were proposed by a Dutch computer scientist Dijkstra in late 60s
- Definition: a semaphore has a non-negative integer value and supports the following operations:
  - sem_t sem or semaphore sem: Declare a semaphore
  - sem_init(&sem, 0, N): Initialize the semaphore to any non-negative value
  - sem_wait(&sem): also called down() or P(), an atomic operation that decrements it by 1 if non-zero. If the semaphore is equal to 0, go to sleep waiting to be signaled by another thread
  - sem_post(&sem): also called signal(), up() or V(), an atomic operation that increments it by 1, and wakes up a waiting/sleeping thread, if any
- Semaphores are also called sleeping locks, since the waiting thread goes to sleep instead of spin-waiting
  - If the waiting time is long, then sleeping is more efficient since the thread gives up the CPU to other threads, but incurs system call (kernel) overhead to go to sleep and wake up; if waiting time is short, then spinlock may be more efficient since it does not involve the kernel.
  - Spinlock may cause starvation, e.g., if the waiting thread has higher priority than the signaler thread under fixed priority scheduling (but not under round-robin scheduling).

Deadlock

- Definition: A set of processes are said to be in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set
- Conditions for Deadlock
- Mutual exclusion
  - Only one process at a time can use a given resource
- Hold-and-wait
  - processes hold resources allocated to them while waiting for additional resources
- No preemption
  - Resources cannot be forcibly removed from processes that are holding them; can be released only voluntarily by each holder
- Circular wait
  - There exists a circle of processes such that each holds one or more resources that are being requested by next process in the circle



Not a perfect analogy, just a fun image!

Banker's algorithm: preliminaries

- Compute *Need* = *Max* – *Allocation*
- To determine if a process *i* can run to completion, compare two vectors:
  - $(Need)_i$: row *i* in the Need Matrix of unmet resource needs
  - A: available resources vector *A*
  - $(Need)_i$ <= *A* if $Need_{ij}$ <= A_j for all resource types *j*

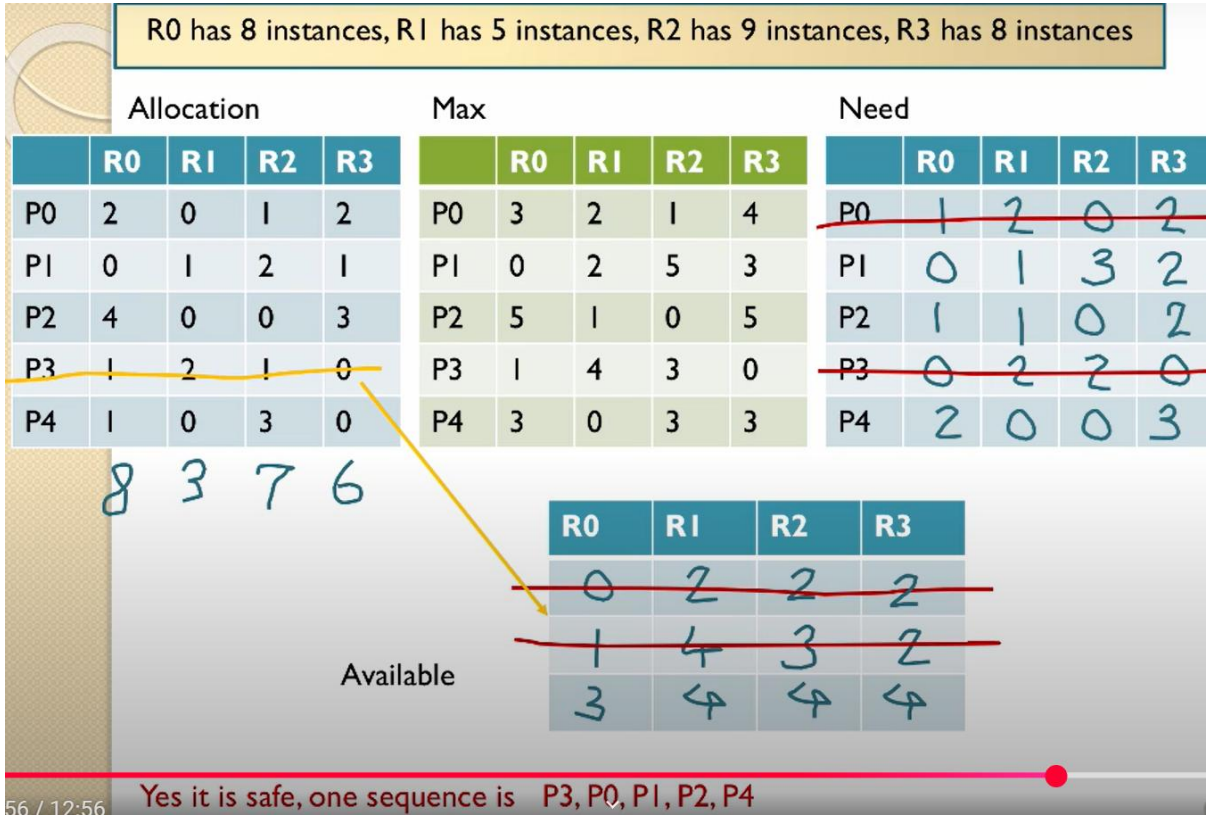
Banker's algorithm

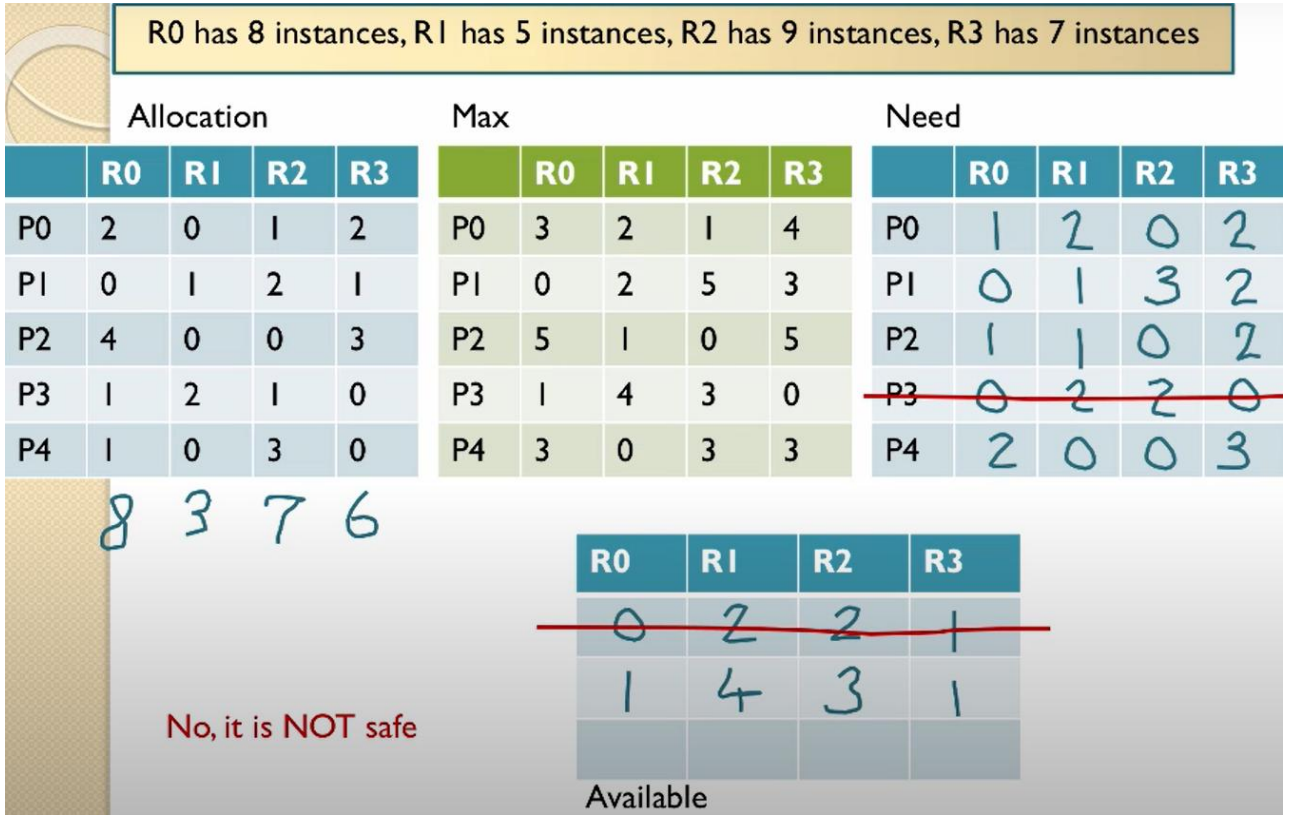
Algorithm CheckSafety() for checking to see if a state is safe:

1. Compute *Need* = *Max* – *Allocation*
2. Look for a process *i* that can run to completion by finding an unmarked row *i* with $(Need)_i \leq A$. If no such row exists, system will eventually deadlock since no process can run to completion
3. Assume process *i* requests all resources it needs and finishes. Mark process *i* as completed, free all its resources and add the *i-th* row of *Allocation* to the *Available* vector
4. Repeat steps 1 and 2 until either all processes are marked as completed (initial state is safe), or no process is left whose resource needs can be met (there is a deadlock, so initial state is unsafe).

Video tutorial of Banker's algorithm I

- Deadlock avoidance https://www.youtube.com/watch?v=AvPjOyeJbBM
- Total resources: [8, 5, 9, 8]



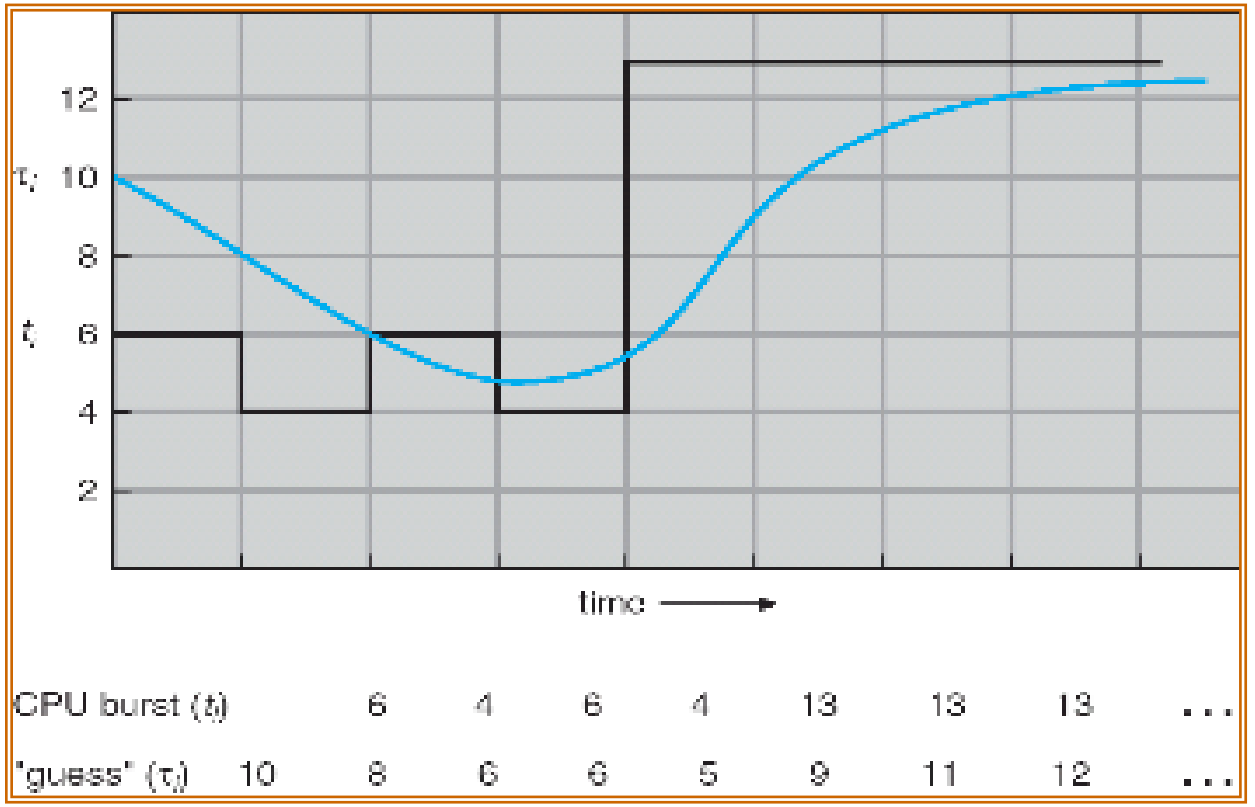


Predicting Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - Works because programs have predictable behavior
    - » If program was I/O bound in recent past, it is likely to be I/O bound in future
- We can use exponential moving averaging $t_n = \alpha x_n + (1 - \alpha) t_{n-1}$, where:
  - x_n is the new input data point
  - t_{n-1} is the previous exponential moving average
  - α is the smoothing factor ($0<\alpha<1$)
    - α large: fast update of τ_n based on new input. $\alpha = 1 \rightarrow t_n = x_n$ is equal to the new input data point at each step
    - α small: slow update of τ_n based on new input. $\alpha = 0 \rightarrow t_n = t_0$ stays constant and not affected by new input data point
    - Appropriate choice of α lets t_n track the input data points while smoothing out sensor noise

Predicting Length of the Next CPU Burst: α=0.5

- Compute $t_n = \alpha x_n + (1-\alpha)t_{n-1}$ with initial guess τ_0 = 10, assuming α=0.5
- $t_1 = \alpha x_1 + (1-\alpha)t_0 = 0.5*6 + 0.5*10 = 8$
- $t_2 = \alpha x_2 + (1-\alpha)t_1 = 0.5*4 + 0.5*8 = 6$
- $t_3 = \alpha x_3 + (1-\alpha)t_2 = 0.5*6 + 0.5*6 = 6$
- $t_4 = \alpha x_4 + (1-\alpha)t_3 = 0.5*4 + 0.5*6 = 5$
- $t_5 = \alpha x_5 + (1-\alpha)t_4 = 0.5*13 + 0.5*5 = 9$
- $t_6 = \alpha x_6 + (1-\alpha)t_5 = 0.5*13 + 0.5*9 = 11$
- $t_7 = \alpha x_7 + (1-\alpha)t_6 = 0.5*13 + 0.5*11 = 12$



Time Series 101: Exponential Moving Average, A Visual Guide
https://www.youtube.com/watch?v=joHKNtPYtLo

Predicting the Length of the Next CPU Burst: α=0.1 or 0.9

- Compute $t_n = \alpha x_n + (1-\alpha)t_{n-1}$ with initial guess τ_0 = 10, assuming α=0.1.
- $t_1 = \alpha x_1 + (1-\alpha)t_0 = 0.1*6 + 0.9*10 = 9.6$
- $t_2 = \alpha x_2 + (1-\alpha)t_1 = 0.1*4 + 0.9*9.6 = 9.0$
- $t_3 = \alpha x_3 + (1-\alpha)t_2 = 0.1*6 + 0.9*9.0 = 8.7$
- $t_4 = \alpha x_4 + (1-\alpha)t_3 = 0.1*4 + 0.9*8.7 = 8.3$
- $t_5 = \alpha x_5 + (1-\alpha)t_4 = 0.1*13 + 0.9*8.3 = 8.7$
- $t_6 = \alpha x_6 + (1-\alpha)t_5 = 0.1*13 + 0.9*8.7 = 9.2$
- $t_7 = \alpha x_7 + (1-\alpha)t_6 = 0.1*13 + 0.9*9.2 = 9.5$

- Compute $t_n = \alpha x_n + (1-\alpha)t_{n-1}$ with initial guess τ_0 = 10, assuming α=0.9.
- $t_1 = \alpha x_1 + (1-\alpha)t_0 = 0.9*6 + 0.1*10 = 6.4$
- $t_2 = \alpha x_2 + (1-\alpha)t_1 = 0.9*4 + 0.1*6.4 = 4.2$
- $t_3 = \alpha x_3 + (1-\alpha)t_2 = 0.9*6 + 0.1*4.2 = 5.8$
- $t_4 = \alpha x_4 + (1-\alpha)t_3 = 0.9*4 + 0.1*5.8 = 4.2$
- $t_5 = \alpha x_5 + (1-\alpha)t_4 = 0.9*13 + 0.1*4.2 = 12.1$
- $t_6 = \alpha x_6 + (1-\alpha)t_5 = 0.9*13 + 0.1*12.1 = 13.0$
- $t_7 = \alpha x_7 + (1-\alpha)t_6 = 0.9*13 + 0.1*13.0 = 13.0$

With low $\alpha = 0.1$, the EMA changes gradually and reacts slowly to new data, staying closer to the starting value.
With high $\alpha = 0.9$, the EMA responds quickly and closely tracks the latest data points.

Lecture 5 Scheduling Conclusion

- FCFS Scheduling:
  - Run jobs in the order of arrival
  - Cons: Short jobs can get stuck behind long ones
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least execution time/least remaining execution time
  - Pros: Optimal (in terms of average response time)
  - Cons: Hard to predict execution time, Unfair
- Priority-Based Scheduling
  - Each job is assigned a fixed priority
- Multi-Level Queue Scheduling
  - Multiple queues of different priorities and scheduling algorithms
- Multi-Level Feedback Queue Scheduling:
  - Automatic promotion/demotion of jobs between queues to approximate SJF/SRTF

Summary of Schedulability Analysis Algorithms

IMPORTANT

	Fixed-Priority Scheduling		Dynamic Priority Scheduling	
Optimal Scheduling Algorithm	**Rate Monotonic (RM)** Scheduling for implicit deadline taskset (D=T)	**Deadline Monotonic (DM)** Scheduling for constrained deadline taskset (D≤T)	**Earliest Deadline First (EDF)** Scheduling for implicit deadline taskset (D=T)	**Earliest Deadline First (EDF)** Scheduling for constrained deadline taskset (D≤T)
Schedulability Analysis Algorithm	Utilization Bound (UB) test $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$ (sufficient but not necessary condition) or Response Time Analysis (RTA) (necessary and sufficient) $R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq D_i$	RTA Response Time Analysis (RTA) (necessary and sufficient) $R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq D_i$	Utilization Bound (UB) test $U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$ (necessary and sufficient)	Density Bound test $\Delta = \sum_i \frac{C_i}{\min(D_i, T_i)} \leq 1$ (sufficient but not necessary condition) or Demand Bound Function (not covered)

PCP Blocking Time

A given task i is blocked (or delayed) by at most one critical section of any lower priority task locking a semaphore with priority ceiling greater than or equal to the priority of task i. We can explain that mathematically using the notation:

$$B_i = \max_{\{k,s \mid k \in lp(i) \wedge s \in used\_by(k) \wedge ceil(s) \geq pri(i)\}} cs_{k,s}$$

- Consider all lower-priority tasks ($k \in lp(i)$), and the semaphores they can lock (s)
- Select from those semaphores (s) with ceiling higher than or equal to $pri(i) = P_i$
- Take max length of all tasks (k)'s critical sections that lock semaphores (s)
- (The blocking time is valid even for a task that does not require any semaphores/critical sections, as it may experience push-through blocking.)